Computational Approaches to Natural Product Drug Discovery

By

Imraan Hussein Alas

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Pharmaceutical Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2024

Date of final oral examination: 04/23/2024

The dissertation is approved by the following members of the Final Oral Committee:
Tim S. Bugni, Professor, Pharmaceutical Sciences
Anthony J. Gitter, Professor, Biostatistics and Medical Informatics and Computer Science
Lingjun Li, Professor, Pharmaceutical Sciences and Chemistry
Jason M. Peters, Assistant Professor, Pharmaceutical Sciences

Dedication

To my grandparents.

Abstract

In the last several decades, antibacterial resistance has continued to rise. Concurrently, the rate of novel antibacterial compounds discovered and approved has dropped. To combat these trends, the ability to leverage computational methods for the improvement of natural product (NP) drug discovery is of paramount importance (Chapter 1). Historically, our lab has prioritized bacterial strains from under-explored environments compared to traditional NP drug discovery research centers, due to the increased rates of re-discovery for antibacterial NPs derived from over-exploited environments.

In order to rigorously rationalize this prioritization scheme, I analyzed marine bacterial strains belonging to the *Micromonosporaceae* family for biosynthetic gene clusters (BGCs) related to secondary metabolism, serving as the genomic landscape for potential antibacterial NPs produced (Chapter 2). Through our untargeted collection of 38 marine *Micromonosporaceae*, I identified a novel genus and several under-studied species, broadly determined prospective NPs that were unique within our dataset, and compared our BGCs against public databases to assess novelty in relation to published literature.

While our methodologies to compare BGCs within our dataset and against public databases were robust, the readily available tools to visualize relationships were lacking. Therefore, I developed the BGC Prioritization Dashboard, which incorporates existing software analyses to generate visualizations that depict BGCs in relation to each other based on their similarity to known BGCs in public databases (Chapter 3). This has allowed for the identification of novel BGCs potentially produced by a given bacterial strain.

Though the previously described work focuses on computational approaches utilizing genomic data, the importance of characterizing NP compounds actively produced in a laboratory setting is equally if not more important. In parallel, we've developed a novel methodology to connect intracellular biological functions to NP structural information based on analytical chemistry techniques. By leveraging liquid chromatography tandem mass spectrometry (LC-MS/MS) to generate molecular fingerprints usable as inputs for machine learning models, we developed a proof-of-concept multiclass classification model to disambiguate NP compounds that modulate tubulin assembly based on simulated MS2 spectra (Chapter 4).

Overall, these findings describe the benefits of incorporating computational approaches into NP drug discovery for the efficient prioritization of novel and relevant compounds.

Acknowledgments

Simply put, I would not be here without my parents. Thanks Mom and Dad, for giving me space when I needed it, and helping me refocus when I needed it. For always offering a helping hand, regardless of what was happening, because they could.

I'd also like to thank Tim. He took a chance on me, as I was about to leave the PhD program, and offered to let me complete my Masters through his lab. And, when I found a burgeoning interest in computational genomics, he gave me full rein to explore and really see what I could do. He also taught me the importance of context, for any problem. Whenever I was struggling to see the larger picture and understand what insights would be useful for a chemistry-focused lab like his, he would always be there to teach how to see things from his point of view.

I would also love to thank Chuck Lauhon and Ken Niemeyer. I never truly knew the amount of work Chuck put in to try and help me finish my time here, and I will never feel anything other than gratitude for his unwavering dedication to all students under his care. Ken, as well, putting in invisible hours to make sure that the School of Pharmacy felt less like a school and more like a home away from home.

To the members of my thesis committee, thank you for your understanding during all this time, being patient with my setbacks but also thinking of me, as the occasional papers passed on were sincerely appreciated.

Thank you to the entirety of the Bugni Lab, including Bailey, Chris (Thomas and Roberts!), Deepa, Doug, Fan, Nathan, Shaurya, and Shukria, for showing me so many wonderful things (both research and outside of research). All of you made me feel welcomed, and if I helped any of you half as much as you've helped me, I would be here another 2 years.

Thanks to all the friends I've carried with me through high school and college. Thank you to CC, Reece, Paige, and Victoria for keeping me sane at late hours, and always be willing to let me share my hobbies with you. Thank you to David, Nick, and Zach for the fun moments, with a special thanks to David and Liam for being the best roommates one could have in the city of Madison.

A special thank you to my friends in Madison, who understood my erratic schedules and my exhausted days, and showed me kindness and new adventures all the time.

And finally, to my sister and brother, who were always there whenever I needed them with pictures of adorable cats (Lollipop and Slushy) and holding the fort down at home.

To everyone listed and not listed, from the bottom of my heart. Thank you.

Table of Contents

DEDICATION ..... **i**

ABSTRACT ..... **ii**

ACKNOWLEDGMENTS ..... **iv**

TABLE OF CONTENTS ..... **vi**

TABLE OF FIGURES ..... **ix**

CHAPTER 1: INTRODUCTION **1**

1.1 REFERENCES **8**

CHAPTER 2: MARINE *MICROMONOSPORACEAE* SECONDARY METABOLITE DIVERSITY ANALYSIS **13**

2.1 INTRODUCTION **13**

2.2 RESULTS & DISCUSSION **18**

2.2.1 *Marine bacterial strain genome characteristics* **18**

2.2.2 *Phylogenetic classification of marine bacterial strains* **19**

2.2.3 *Diversity and distribution of BGCs in marine Micromonosporaceae* **21**

2.2.4 *BiG-SLiCE queries reveal significant likelihood for previously unknown biosynthetic potentials* **25**

2.2.5 *BGC distribution of Micromonospora_E is statistically different from Micromonospora* **28**

2.3 MATERIALS & METHODS **30**

2.3.1 *Strain isolation and extraction* **30**

2.3.2 *Strain sequencing, assembly, and validation* **30**

2.3.3 *Annotation of biosynthetic gene clusters* **30**

2.3.4 *Phylogenomics, average identity estimation between genomic pairs, and taxonomic classification* **31**

2.3.5 *Analysis of intra-BGC diversity using marine Micromonosporaceae* **31**

2.3.6 *Analysis of BGC diversity against terrestrial bacteria* **32**

2.3.7 *Statistical analysis of BGC similarity distribution across genera* **33**

2.4 DATA SUMMARY 34

2.5 REFERENCES 37

CHAPTER 3: DEVELOPMENT OF BIOSYNTHETIC GENE CLUSTER PRIORITIZATION DASHBOARD USING SECONDARY METABOLITES THROUGH VISUALIZATION 43

3.1 INTRODUCTION 43

3.2 RESULTS & DISCUSSION 47

3.2.1 *Truly novel BGCs are identifiable using metric MDS and pairwise distance modeling* 47

3.2.2 *Prioritization of BGCs through BGCPD* 52

3.3 MATERIALS & METHODS 55

3.3.1 *Strain isolation and extraction* 55

3.3.2 *Strain sequencing, assembly, and validation* 55

3.3.3 *Annotation of biosynthetic gene clusters* 55

3.3.4 *Analysis of BGC diversity against terrestrial bacteria* 56

3.3.5 *Initial methodology for identification of novel BGCs* 56

3.3.6 *Robust refinements to identifying novel BGCs* 57

3.3.7 *Development of visualization dashboard for BGC prioritization* 57

3.4 DATA SUMMARY 59

3.5 REFERENCES 61

CHAPTER 4: PREDICTING TUBULIN-BINDING COMPOUNDS USING MS/MS STRUCTURALLY REPRESENTATIVE FINGERPRINTS WITH MACHINE LEARNING 64

4.1 INTRODUCTION 64

4.2 RESULTS & DISCUSSION 67

4.2.1 *Yeast Chemical Genomics (YCG) as a repository to link bioprocesses with compounds* 67

4.2.2 *Dataset augmentation scheme to account for low Positive instances* 67

4.2.3 *Hyperparameter optimization of binary classification models through F1 score and Average Precision (AP)* 68

4.2.4 *Application of best-performing binary classifiers reveals poor generalizability* **69**

4.2.5 *Comparisons between simulated and experimental datasets* **71**

4.2.6 *Multiclass classification models show enhanced capabilities* **73**

4.3 MATERIALS & METHODS **79**

4.3.1 *Binary labeling scheme of dataset using CG-Target scores* **79**

4.3.2 *Augmentation of Positive instances using structurally similar compounds* **79**

4.3.3 *Fingerprint generation through SIRIUS* **80**

4.3.4 *Train-test-validation split for binary classification* **80**

4.3.5 *Hyperparameter optimization of binary classification models* **80**

4.3.6 *Identifying the best performing binary classification model* **81**

4.3.7 *Compiling and generating MS2FP for experimental GNPS spectra* **81**

4.3.8 *Comparing simulated and experimental spectra datasets* **81**

4.3.9 *Multiclass dataset labeling using representative tubulin active structural families* **82**

4.3.10 *Train-test split for multiclass classification* **83**

4.3.11 *Hyperparameter optimization of multiclass models* **83**

4.3.12 *Evaluation of multiclass models on experimental GNPS MS/SM spectra* **83**

4.3.13 *Determination of False Positive Rate (FPR) for multiclass models* **84**

4.3.14 *Evaluation of multiclass models on experimental GNPS MS/SM spectra* **84**

4.4 REFERENCES **85**

CHAPTER 5: CONCLUSIONS AND FUTURE WORK **87**

5.1 CONCLUDING REMARKS **87**

5.2 FUTURE DIRECTIONS **88**

5.3 REFERENCES **90**

Appendix A **92**

Appendix B **117**

Table of Figures

Figure 2.1 Phylogenetic and BGC information **20**
Figure 2.2 Sequence Similarity Network **24**
Figure 2.3 Scatterplot of BGCs queried against BiG-SLiCE **26**
Figure 2.4 Distribution of BGC distance to GCF by genera **29**
Figure 3.1 BiG-SLiCE Prioritization Dashboard Methodology **45**
Figure 3.2 MDS scatterplot of *Micromonosporaceae* BGCs **48**
Figure 3.3 Annotated MDS scatterplot of *Micromonosporaceae* PKS/NRPS BGCs **49**
Figure 3.4 Improved MDS scatterplot of *Micromonosporaceae* BGCs **50**
Figure 3.5 Annotated and improved MDS scatterplot of *Micromonosporaceae* PKS/NRPS BGCs **51**
Figure 3.6 Visualization of BGC Prioritization Dashboard **52**
Figure 4.1 Machine Learning methodology overview **65**
Figure 4.2 t-SNE visualization of simulated and experimental fingerprints **72**
Figure 4.3 PCA and KMeans Clustering explorations of simulated and experimental fingerprints **73**
Figure 4.4 SVM multiclass classification model accuracy by spectra quality **78**

Chapter 1

Introduction

Antibiotic resistance poses an existential threat to human health[1,2]. The CDC 2019 Antibiotic Resistance Threats Report described that there were over 2.8 million antibiotic-resistant infections per year, and an associated 35,900 deaths from antibiotic resistance[1]. At the time, the CDC stated, "for further progress, the nation must continue to innovate and scale up effective strategies to prevent infections, stop spread, and save lives"[1]. Despite optimism regarding a decline in deaths from antimicrobial resistance from 2012 to 2017 in both hospitals and overall, the CDC 2022 Special Report on COVID-19 paints a much bleaker picture regarding healthcare-associated pathogens, even while highlighting a lack of data due to pandemic impacts[2]. In order to mitigate the effects of antibiotic resistance, the discovery of new antimicrobials is required.

In 1910, the first formally recognized antibiotic (salvarsan) was utilized against syphilis, specifically against the bacteria *Treponema pallidum*[3–6]. Following that, the compound termed penicillin was isolated in 1928 by Alexander Fleming[3,5,7]. Termed natural products (NPs), these compounds were considered part of an incredibly expansive space of diverse chemical entities, originating from plants, animals, bacteria, fungi, and so on[3,8–10]. However, a clear distinction was made, specifically regarding the intended function of these fascinating compounds. NPs that were specifically related to "primary metabolism," such as metabolites involved with the growth, development, and reproduction of the organism, were sectioned off and referred to as primary metabolites[11]. Moving forward, NPs now refer exclusively to compounds related to "secondary metabolism," or offshoots of primary metabolism that are not directly associated with growth, development, or reproduction of the organism[11–14].

NPs have historically been investigated for a wide range of uses across the pharmaceutical, herbicidal, and insecticidal industries due to their chemical diversity and complexity resulting in valuable chemical properties[8,14]. Taken wholesale or partially through NP-influenced design, NPs have resulted in the development of several medicines on the WHO's Model List of Essential Medicines, including gentamicin, azithromycin, and more[9,15,16]. Approximately 40% of the 1453 new chemical entities approved by 2013 are NPs or NP-inspired, and that percentage increases to 50% between 1983 and 2013[17]. Overall, as Atanasov states in 2021, the "NP pool is enriched with 'bioactive' compounds covering a wider area of chemical space compared to typical synthetic small-molecule libraries," further rationalizing prioritization for antimicrobial resistance[8].

However, uncovering NP drug candidates is not a simplistic affair. NP drug discovery often requires: a) finding an bioactive compound, b) culturing sufficient quantities of the bioactive compound, and c) characterization of the bioactive compound[18]. Each section of this process has unique issues, with low yield resulting in the missed detection of bioactive compounds, potentially requiring several months to produce large enough quantities to evaluate in models, and the bioactive compound being previously characterized and explored[18]. Pharmaceutical companies previously shifted to prioritizing high-throughput screens of synthetic compound libraries due to easily modifiable structures and shorter timelines necessary to discard compounds with issues[18]. In more recent years, there has been a prioritization of "diversity-oriented synthesis" and "privileged structures", which attempt to leverage structurally-diverse collections and scaffolds of existing pharmaceutical drugs to find compounds with enhanced bioactivity, often directly inspired by NPs[18,19]. Despite the push against NPs, the process of finding antibacterial compounds starts and ends with NPs.

NPs also exist within a specific context for an organism, often conferring some fitness advantage not directly related to growth or reproduction[14]. In response to environmental pressures, such as competing bacteria or predation, upregulation of biosynthetic gene clusters (BGCs) that encode for protective or offensive NP compounds occurs[14]. According to Medema et al[20], a BGC can be defined as a "physically clustered group of two or more genes in a particular genome that together encode a biosynthetic pathway for the production of a specialized metabolite (including its chemical variants)"[20–22]. Given that the entirety of a bacterial genome is assessed through next-generation sequencing, the surveilled pool of BGCs in each genome then defines the entire chemical space potentially produced by that bacterium in laboratory conditions.

Classes of BGCs such as type I polyketide synthases (T1PKS) and non-ribosomal peptide synthetases (NRPS) tend to act in a linear manner for assembly of the NPs, allowing for reasonable inference of structural information based on bioinformatic analysis[14,23–26]. However, non-linear systems such as terpenes and saccharides are much less amenable to gleaning structural information from genomic information, and exceptions to the linearity of T1PKS and NRPS continue to be found[14]. Automated identification of these BGCs through tools such as antiSMASH rely on manually curated rules, which incorporate rule-breaking BGCs into additional rules for future detection[27,28]. The identification of BGCs by leveraging sequencing information continues to improve in an iterative fashion, adding additional technologies such as deep learning strategies to improve detection of BGCs[29]. In the eventual future, snapshots of the BGCs identified in bacterial genomes will be comprehensive enough to accurately describe the entire chemical space of the NPs encoded.

Paired with this genomic viewpoint of the potential chemical space is the metabolomic lens, which describes the metabolites produced by a given organism for a specific time and

condition[30]. An organism, grown in laboratory conditions based on general optimization of growth media, produces specialized metabolites. These metabolites are extracted through the usage of organic solvents, separated with liquid chromatography (LC), and analyzed with tandem mass spectrometry (MS/MS). In the combined LC-MS/MS process, the metabolites are blasted with high energy gaseous ions to fragment them, then mass-to-charge ratios and retention times of the fragments are quantified.

A bacterium can generate enormous quantities of specialized metabolites, and the efficient prioritization of metabolites for conquering antimicrobial resistance is a must. Structural characterization of specific metabolites, which have typically already shown activity against a relevant panel of multi-drug resistant pathogens, often requires time-intensive labor and techniques such as nuclear magnetic resonance (NMR)[31]. This was necessary because interrogating biological function of NPs against antibacterial resistance pathogens often requires an understanding of the structure of the compound for determination of the pharmacophore. However, it is now possible to circumvent the need for full characterization of structures by instead utilizing approximate structural features through molecular fingerprints.

Molecular fingerprints refer to the encoded structural characteristics of a molecule to a vector for similarity comparison purposes[32]. Developments in software have allowed for the generation of molecular fingerprints based on MSMS datasets[33]. These MSMS datasets contain fragments of metabolites, which are leveraged to capture key structural features that serve as trademarks for a given molecule, generating the molecular fingerprint[33]. By connecting these metabolites to a fully characterized biological system such as *Saccharomyces cerevisiae*, one can interrogate the biological function of a NP *in silico* through purely MSMS information[34–37].

Yeast chemical genomics (YCG), which refers to the process of correlating biological processes and small molecules based on compound screening against a comprehensive knockout library, serves as the methodology to connect metabolites to impacted bioprocesses[35,38]. The database MOSAIC, which was designed to discover mode-of-action for compounds using chemical-genomic approaches, represents the most extensive chemical-genomic dataset to date[35]. As such, one can leverage molecular fingerprints with the extensively characterized YCG dataset of compounds and biological processes to predict the bioprocesses impacted by novel NPs.

The chosen methodology to coordinate labeled YCG information on biological processes and molecular fingerprints is machine learning. Machine learning describes algorithms designed to infer information based on pattern recognition[39–41]. As computational resources have increased and datasets have grown in complexity, machine learning has found success in image classification/generation, peptide identification, and more[40,42]. However, none have applied machine learning to connect NP molecular fingerprints to impacted bioprocesses, leaving a clear gap in literature.

In summation, most compounds produced by a bacterial strain likely have minimal to no negative effect on known pathogens, making them irrelevant towards resolving antibiotic resistance. At the same time, as more antimicrobial compounds were discovered, so too was the landscape of prospective novel compounds narrowed down, forming the basis of the "rediscovery" problem[3,8,12,13]. Rediscovery, in the natural products (NPs) field, refers to the issue of discovering supposedly novel antimicrobial compounds that were in fact already explored and studied[3,8,12,13]. To resolve rediscovery, natural products researchers have traditionally relied on exploiting novel environments, enhancing throughput of high-throughput screens, investigating under-studied bacterial families, and more[18,43,44]. **In this thesis, I describe the development of**

computational approaches to resolve three barriers in natural product drug discovery: 1) rationalizing the marine environment as a source of under-exploited bacteria 2) presenting a methodology to prioritize BGCs based on novelty to public databases 3) facilitating the discovery of NPs that impact key biological processes.

In the first chapter of this work, I explore an untargeted collection of marine bacteria belonging to the family *Micromonosporaceae* based on the BGCs identified. Our goal in this section is to perform a comparative genomic analysis of the marine *Micromonosporaceae*, allowing us to present the merits of exploiting understudied environmental sources for NP drug discovery. In doing so, I reveal the biosynthetic diversity of marine *Micromonosporaceae* compared to public literature, identifying both novel genus and species, and highlighting potentially new chemistry enclosed within.

In the second chapter of this work, I refine the general methodologies that allowed us to perform comparative genomics analysis of the family *Micromonosporaceae* in Chapter 1, culminating in the development of the BGC Prioritization Dashboard. The BGC Prioritization Dashboard allows us to explore our BGCs in relation to each other based on their novelty to public databases. This strategy results in visualizations that display similar BGCs closely together, while still allowing the viewer to contextualize their dataset based on published literature, indicating BGCs that are novel both within their dataset and within public databases.

In the third chapter of this work, we present a machine learning pipeline leveraging molecular fingerprints derived from MSMS datasets to interrogate biological function of NPs for the purpose of effective prioritization. Using modulators of tubulin assembly as a proof-of-concept biological process, we showcase machine learning models that predict whether a NP is a

modulator of tubulin assembly based purely on structural information. In doing so, we enhance NP drug discovery by reducing the need for time-intensive structural characterization techniques on compounds unrelated to key biological processes of interest.

With these chapters, I outline computational methods to improve NP drug discovery from genomics and metabolomics fields, representing the possible landscape of what is produced by a bacterium (genomics) and the physical reality of laboratory conditions (metabolomics).

1.1 REFERENCES

(1) Centers for Disease Control and Prevention (U.S.). *Antibiotic Resistance Threats in the United States, 2019*; Centers for Disease Control and Prevention (U.S.), 2019. https://doi.org/10.15620/cdc:82532.

(2) *COVID-19: U.S. Impact on Antimicrobial Resistance, Special Report 2022*; National Center for Emerging and Zoonotic Infectious Diseases, 2022. https://doi.org/10.15620/cdc:117915.

(3) Hutchings, M. I.; Truman, A. W.; Wilkinson, B. Antibiotics: Past, Present and Future. *Current Opinion in Microbiology* **2019**, *51*, 72–80. https://doi.org/10.1016/j.mib.2019.10.008.

(4) Vernon, G. Syphilis and Salvarsan. *Br J Gen Pract* **2019**, *69* (682), 246. https://doi.org/10.3399/bjgp19X702533.

(5) Christensen, S. B. Drugs That Changed Society: History and Current Status of the Early Antibiotics: Salvarsan, Sulfonamides, and β-Lactams. *Molecules* **2021**, *26* (19), 6057. https://doi.org/10.3390/molecules26196057.

(6) Yadav, S.; Shah, D.; Dalai, P.; Agrawal-Rajput, R. The Tale of Antibiotics beyond Antimicrobials: Expanding Horizons. *Cytokine* **2023**, *169*, 156285. https://doi.org/10.1016/j.cyto.2023.156285.

(7) Gaynes, R. The Discovery of Penicillin—New Insights After More Than 75 Years of Clinical Use - Volume 23, Number 5—May 2017 - Emerging Infectious Diseases Journal - CDC. https://doi.org/10.3201/eid2305.161556.

(8) Atanasov, A. G.; Zotchev, S. B.; Dirsch, V. M.; Supuran, C. T. Natural Products in Drug Discovery: Advances and Opportunities. *Nat Rev Drug Discov* **2021**, *20* (3), 200–216. https://doi.org/10.1038/s41573-020-00114-z.

(9) Newman, D. J.; Cragg, G. M. Natural Products as Sources of New Drugs over the Nearly Four Decades from 01/1981 to 09/2019. *J. Nat. Prod.* **2020**, *83* (3), 770–803. https://doi.org/10.1021/acs.jnatprod.9b01285.

(10) Porras, G.; Chassagne, F.; Lyles, J. T.; Marquez, L.; Dettweiler, M.; Salam, A. M.; Samarakoon, T.; Shabih, S.; Farrokhi, D. R.; Quave, C. L. Ethnobotany and the Role of Plant Natural Products in Antibiotic Drug Discovery. *Chem Rev* **2021**, *121* (6), 3495–3560. https://doi.org/10.1021/acs.chemrev.0c00922.

(11) Seyedsayamdost, M. R. Toward a Global Picture of Bacterial Secondary Metabolism. *J Ind Microbiol Biotechnol* **2019**, *46* (3–4), 301–311. https://doi.org/10.1007/s10295-019-02136-y.

(12) Scherlach, K.; Hertweck, C. Mining and Unearthing Hidden Biosynthetic Potential. *Nat Commun* **2021**, *12* (1), 3864. https://doi.org/10.1038/s41467-021-24133-5.

(13) Zhang, M. M.; Qiao, Y.; Ang, E. L.; Zhao, H. Using Natural Products for Drug Discovery: The Impact of the Genomics Era. *Expert Opin Drug Discov* **2017**, *12* (5), 475–487. https://doi.org/10.1080/17460441.2017.1303478.

(14) Jensen, P. R. Natural Products and the Gene Cluster Revolution. *Trends Microbiol* **2016**, *24* (12), 968–977. https://doi.org/10.1016/j.tim.2016.07.006.

(15) Purgato, M.; Barbui, C. What Is the WHO Essential Medicines List? *Epidemiol Psychiatr Sci* **2012**, *21* (4), 343–345. https://doi.org/10.1017/S204579601200039X.

(16) *WHO Model List of Essential Medicines - 23rd list, 2023*. https://www.who.int/publications-detail-redirect/WHO-MHP-HPS-EML-2023.02 (accessed 2024-02-18).

(17) Katz, L.; Baltz, R. H. Natural Product Discovery: Past, Present, and Future. *Journal of Industrial Microbiology and Biotechnology* **2016**, *43* (2–3), 155–176. https://doi.org/10.1007/s10295-015-1723-5.

(18) Li, J. W.-H.; Vederas, J. C. Drug Discovery and Natural Products: End of an Era or an Endless Frontier? *Science* **2009**, *325* (5937), 161–165. https://doi.org/10.1126/science.1168243.

(19) Newman, D. J. Natural Products as Leads to Potential Drugs: An Old Process or the New Hope for Drug Discovery? *J. Med. Chem.* **2008**, *51* (9), 2589–2599. https://doi.org/10.1021/jm0704090.

(20) Medema, M. H.; Kottmann, R.; Yilmaz, P.; Cummings, M.; Biggins, J. B.; Blin, K.; de Bruijn, I.; Chooi, Y. H.; Claesen, J.; Coates, R. C.; Cruz-Morales, P.; Duddela, S.; Düsterhus, S.; Edwards, D. J.; Fewer, D. P.; Garg, N.; Geiger, C.; Gomez-Escribano, J. P.; Greule, A.; Hadjithomas, M.; Haines, A. S.; Helfrich, E. J. N.; Hillwig, M. L.; Ishida, K.; Jones, A. C.; Jones, C. S.; Jungmann, K.; Kegler, C.; Kim, H. U.; Kötter, P.; Krug, D.; Masschelein, J.; Melnik, A. V.; Mantovani, S. M.; Monroe, E. A.; Moore, M.; Moss, N.; Nützmann, H.-W.; Pan, G.; Pati, A.; Petras, D.; Reen, F. J.; Rosconi, F.; Rui, Z.; Tian, Z.; Tobias, N. J.; Tsunematsu, Y.; Wiemann, P.; Wyckoff, E.; Yan, X.; Yim, G.; Yu, F.; Xie, Y.; Aigle, B.; Apel, A. K.; Balibar, C. J.; Balskus, E. P.; Barona-Gómez, F.; Bechthold, A.; Bode, H. B.; Borriss, R.; Brady, S. F.; Brakhage, A. A.; Caffrey, P.; Cheng, Y.-Q.; Clardy, J.; Cox, R. J.; De Mot, R.; Donadio, S.; Donia, M. S.; van der Donk, W. A.; Dorrestein, P. C.; Doyle, S.; Driessen, A. J. M.; Ehling-Schulz, M.; Entian, K.-D.; Fischbach, M. A.; Gerwick, L.; Gerwick, W. H.; Gross, H.; Gust, B.; Hertweck, C.; Höfte, M.; Jensen, S. E.; Ju, J.; Katz, L.; Kaysser, L.; Klassen, J. L.; Keller, N. P.; Kormanec, J.; Kuipers, O. P.; Kuzuyama, T.; Kyrpides, N. C.; Kwon, H.-J.; Lautru, S.; Lavigne, R.; Lee, C. Y.; Linquan, B.; Liu, X.; Liu, W.; Luzhetskyy, A.; Mahmud, T.; Mast, Y.; Méndez, C.; Metsä-Ketelä, M.; Micklefield, J.; Mitchell, D. A.; Moore, B. S.; Moreira, L. M.; Müller, R.; Neilan, B. A.; Nett, M.; Nielsen, J.; O'Gara, F.; Oikawa, H.; Osbourn, A.; Osburne, M. S.; Ostash, B.; Payne, S. M.; Pernodet, J.-L.; Petricek, M.; Piel, J.; Ploux, O.; Raaijmakers, J. M.; Salas, J. A.; Schmitt, E. K.; Scott, B.; Seipke, R. F.; Shen, B.; Sherman, D. H.; Sivonen, K.; Smanski, M. J.; Sosio, M.; Stegmann, E.; Süssmuth, R. D.; Tahlan, K.; Thomas, C. M.; Tang, Y.; Truman, A. W.; Viaud, M.; Walton, J. D.; Walsh, C. T.; Weber, T.; van Wezel, G. P.; Wilkinson, B.; Willey, J. M.; Wohlleben, W.; Wright, G. D.; Ziemert, N.; Zhang, C.; Zotchev, S. B.; Breitling, R.; Takano, E.; Glöckner, F. O. Minimum Information about a Biosynthetic Gene Cluster. *Nat Chem Biol* **2015**, *11* (9), 625–631. https://doi.org/10.1038/nchembio.1890.

(21) Kautsar, S. A.; Blin, K.; Shaw, S.; Navarro-Muñoz, J. C.; Terlouw, B. R.; van der Hooft, J. J. J.; van Santen, J. A.; Tracanna, V.; Suarez Duran, H. G.; Pascal Andreu, V.; Selem-Mojica, N.; Alanjary, M.; Robinson, S. L.; Lund, G.; Epstein, S. C.; Sisto, A. C.; Charkoudian, L. K.;

Collemare, J.; Linington, R. G.; Weber, T.; Medema, M. H. MIBiG 2.0: A Repository for Biosynthetic Gene Clusters of Known Function. *Nucleic Acids Research* **2020**, *48* (D1), D454–D458. https://doi.org/10.1093/nar/gkz882.

(22) Terlouw, B. R.; Blin, K.; Navarro-Muñoz, J. C.; Avalon, N. E.; Chevrette, M. G.; Egbert, S.; Lee, S.; Meijer, D.; Recchia, M. J. J.; Reitz, Z. L.; van Santen, J. A.; Selem-Mojica, N.; Tørring, T.; Zaroubi, L.; Alanjary, M.; Aleti, G.; Aguilar, C.; Al-Salihi, S. A. A.; Augustijn, H. E.; Avelar-Rivas, J. A.; Avitia-Domínguez, L. A.; Barona-Gómez, F.; Bernaldo-Agüero, J.; Bielinski, V. A.; Biermann, F.; Booth, T. J.; Carrion Bravo, V. J.; Castelo-Branco, R.; Chagas, F. O.; Cruz-Morales, P.; Du, C.; Duncan, K. R.; Gavriilidou, A.; Gayrard, D.; Gutiérrez-García, K.; Haslinger, K.; Helfrich, E. J. N.; van der Hooft, J. J. J.; Jati, A. P.; Kalkreuter, E.; Kalyvas, N.; Kang, K. B.; Kautsar, S.; Kim, W.; Kunjapur, A. M.; Li, Y.-X.; Lin, G.-M.; Loureiro, C.; Louwen, J. J. R.; Louwen, N. L. L.; Lund, G.; Parra, J.; Philmus, B.; Pourmohsenin, B.; Pronk, L. J. U.; Rego, A.; Rex, D. A. B.; Robinson, S.; Rosas-Becerra, L. R.; Roxborough, E. T.; Schorn, M. A.; Scobie, D. J.; Singh, K. S.; Sokolova, N.; Tang, X.; Udwary, D.; Vigneshwari, A.; Vind, K.; Vromans, S. P. J. M.; Waschulin, V.; Williams, S. E.; Winter, J. M.; Witte, T. E.; Xie, H.; Yang, D.; Yu, J.; Zdouc, M.; Zhong, Z.; Collemare, J.; Linington, R. G.; Weber, T.; Medema, M. H. MIBiG 3.0: A Community-Driven Effort to Annotate Experimentally Validated Biosynthetic Gene Clusters. *Nucleic Acids Research* **2023**, *51* (D1), D603–D610. https://doi.org/10.1093/nar/gkac1049.

(23) Wang, B.; Guo, F.; Huang, C.; Zhao, H. Unraveling the Iterative Type I Polyketide Synthases Hidden in Streptomyces. *Proc Natl Acad Sci U S A* **2020**, *117* (15), 8449–8454. https://doi.org/10.1073/pnas.1917664117.

(24) Kornfuehrer, T.; Eustáquio, A. S. Diversification of Polyketide Structures via Synthase Engineering. *Medchemcomm* **2019**, *10* (8), 1256–1272. https://doi.org/10.1039/c9md00141g.

(25) Fischbach, M. A.; Walsh, C. T. Assembly-Line Enzymology for Polyketide and Nonribosomal Peptide Antibiotics: Logic, Machinery, and Mechanisms. *Chem. Rev.* **2006**, *106* (8), 3468–3496. https://doi.org/10.1021/cr0503097.

(26) Sieber, S. A.; Marahiel, M. A. Molecular Mechanisms Underlying Nonribosomal Peptide Synthesis: Approaches to New Antibiotics. *Chem. Rev.* **2005**, *105* (2), 715–738. https://doi.org/10.1021/cr0301191.

(27) Medema, M. H.; Blin, K.; Cimermancic, P.; de Jager, V.; Zakrzewski, P.; Fischbach, M. A.; Weber, T.; Takano, E.; Breitling, R. antiSMASH: Rapid Identification, Annotation and Analysis of Secondary Metabolite Biosynthesis Gene Clusters in Bacterial and Fungal Genome Sequences. *Nucleic Acids Res* **2011**, *39* (Web Server issue), W339–W346. https://doi.org/10.1093/nar/gkr466.

(28) Blin, K.; Shaw, S.; Augustijn, H. E.; Reitz, Z. L.; Biermann, F.; Alanjary, M.; Fetter, A.; Terlouw, B. R.; Metcalf, W. W.; Helfrich, E. J. N.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 7.0: New and Improved Predictions for Detection, Regulation, Chemical Structures and Visualisation. *Nucleic Acids Research* **2023**, *51* (W1), W46–W50. https://doi.org/10.1093/nar/gkad344.

(29) Hannigan, G. D.; Prihoda, D.; Palicka, A.; Soukup, J.; Klempir, O.; Rampula, L.; Durcak, J.; Wurst, M.; Kotowski, J.; Chang, D.; Wang, R.; Piizzi, G.; Temesi, G.; Hazuda, D. J.; Woelk,

C. H.; Bitton, D. A. A Deep Learning Genome-Mining Strategy for Biosynthetic Gene Cluster Prediction. *Nucleic Acids Research* **2019**, *47* (18), e110. https://doi.org/10.1093/nar/gkz654.

(30) Alseekh, S.; Fernie, A. R. Metabolomics 20 Years on: What Have We Learned and What Hurdles Remain? *The Plant Journal* **2018**, *94* (6), 933–942. https://doi.org/10.1111/tpj.13950.

(31) McAlpine, J. B.; Chen, S.-N.; Kutateladze, A.; MacMillan, J. B.; Appendino, G.; Barison, A.; Beniddir, M. A.; Biavatti, M. W.; Bluml, S.; Boufridi, A.; Butler, M. S.; Capon, R. J.; Choi, Y. H.; Coppage, D.; Crews, P.; Crimmins, M. T.; Csete, M.; Dewapriya, P.; Egan, J. M.; Garson, M. J.; Genta-Jouve, G.; Gerwick, W. H.; Gross, H.; Harper, M. K.; Hermanto, P.; Hook, J. M.; Hunter, L.; Jeannerat, D.; Ji, N.-Y.; Johnson, T. A.; Kingston, D. G. I.; Koshino, H.; Lee, H.-W.; Lewin, G.; Li, J.; Linington, R. G.; Liu, M.; McPhail, K. L.; Molinski, T. F.; Moore, B. S.; Nam, J.-W.; Neupane, R. P.; Niemitz, M.; Nuzillard, J.-M.; Oberlies, N. H.; Ocampos, F. M. M.; Pan, G.; Quinn, R. J.; Reddy, D. S.; Renault, J.-H.; Rivera-Chávez, J.; Robien, W.; Saunders, C. M.; Schmidt, T. J.; Seger, C.; Shen, B.; Steinbeck, C.; Stuppner, H.; Sturm, S.; Taglialatela-Scafati, O.; Tantillo, D. J.; Verpoorte, R.; Wang, B.-G.; Williams, C. M.; Williams, P. G.; Wist, J.; Yue, J.-M.; Zhang, C.; Xu, Z.; Simmler, C.; Lankin, D. C.; Bisson, J.; Pauli, G. F. The Value of Universally Available Raw NMR Data for Transparency, Reproducibility, and Integrity in Natural Product Research. *Nat. Prod. Rep.* **2019**, *36* (1), 35–107. https://doi.org/10.1039/C7NP00064B.

(32) Capecchi, A.; Probst, D.; Reymond, J.-L. One Molecular Fingerprint to Rule Them All: Drugs, Biomolecules, and the Metabolome. *Journal of Cheminformatics* **2020**, *12* (1), 43. https://doi.org/10.1186/s13321-020-00445-4.

(33) Dührkop, K.; Fleischauer, M.; Ludwig, M.; Aksenov, A. A.; Melnik, A. V.; Meusel, M.; Dorrestein, P. C.; Rousu, J.; Böcker, S. SIRIUS 4: A Rapid Tool for Turning Tandem Mass Spectra into Metabolite Structure Information. *Nat Methods* **2019**, *16* (4), 299–302. https://doi.org/10.1038/s41592-019-0344-8.

(34) Piotrowski, J. S.; Li, S. C.; Deshpande, R.; Simpkins, S. W.; Nelson, J.; Yashiroda, Y.; Barber, J. M.; Safizadeh, H.; Wilson, E.; Okada, H.; Gebre, A. A.; Kubo, K.; Torres, N. P.; LeBlanc, M. A.; Andrusiak, K.; Okamoto, R.; Yoshimura, M.; DeRango-Adem, E.; van Leeuwen, J.; Shirahige, K.; Baryshnikova, A.; Brown, G. W.; Hirano, H.; Costanzo, M.; Andrews, B.; Ohya, Y.; Osada, H.; Yoshida, M.; Myers, C. L.; Boone, C. Functional Annotation of Chemical Libraries across Diverse Biological Processes. *Nat Chem Biol* **2017**, *13* (9), 982–993. https://doi.org/10.1038/nchembio.2436.

(35) Nelson, J.; Simpkins, S. W.; Safizadeh, H.; Li, S. C.; Piotrowski, J. S.; Hirano, H.; Yashiroda, Y.; Osada, H.; Yoshida, M.; Boone, C.; Myers, C. L. MOSAIC: A Chemical-Genetic Interaction Data Repository and Web Resource for Exploring Chemical Modes of Action. *Bioinformatics* **2018**, *34* (7), 1251–1252. https://doi.org/10.1093/bioinformatics/btx732.

(36) Costanzo, M.; VanderSluis, B.; Koch, E. N.; Baryshnikova, A.; Pons, C.; Tan, G.; Wang, W.; Usaj, M.; Hanchard, J.; Lee, S. D.; Pelechano, V.; Styles, E. B.; Billmann, M.; van Leeuwen, J.; van Dyk, N.; Lin, Z.-Y.; Kuzmin, E.; Nelson, J.; Piotrowski, J. S.; Srikumar, T.; Bahr, S.; Chen, Y.; Deshpande, R.; Kurat, C. F.; Li, S. C.; Li, Z.; Usaj, M. M.; Okada, H.; Pascoe, N.; San Luis, B.-J.; Sharifpoor, S.; Shuteriqi, E.; Simpkins, S. W.; Snider, J.; Suresh, H. G.; Tan, Y.; Zhu, H.; Malod-Dognin, N.; Janjic, V.; Przulj, N.; Troyanskaya, O. G.; Stagljar, I.; Xia, T.; Ohya, Y.;

Gingras, A.-C.; Raught, B.; Boutros, M.; Steinmetz, L. M.; Moore, C. L.; Rosebrock, A. P.; Caudy, A. A.; Myers, C. L.; Andrews, B.; Boone, C. A Global Genetic Interaction Network Maps a Wiring Diagram of Cellular Function. *Science* **2016**, *353* (6306), aaf1420. https://doi.org/10.1126/science.aaf1420.

(37) Usaj, M.; Tan, Y.; Wang, W.; VanderSluis, B.; Zou, A.; Myers, C. L.; Costanzo, M.; Andrews, B.; Boone, C. TheCellMap.Org: A Web-Accessible Database for Visualizing and Mining the Global Yeast Genetic Interaction Network. *G3 Genes|Genomes|Genetics* **2017**, *7* (5), 1539–1549. https://doi.org/10.1534/g3.117.040220.

(38) Enserink, J. M. Chemical Genetics: Budding Yeast as a Platform for Drug Discovery and Mapping of Genetic Pathways. *Molecules* **2012**, *17* (8), 9258–9273. https://doi.org/10.3390/molecules17089258.

(39) Greener, J. G.; Kandathil, S. M.; Moffat, L.; Jones, D. T. A Guide to Machine Learning for Biologists. *Nat Rev Mol Cell Biol* **2022**, *23* (1), 40–55. https://doi.org/10.1038/s41580-021-00407-0.

(40) Prihoda, D.; Maritz, J. M.; Klempir, O.; Dzamba, D.; Woelk, C. H.; Hazuda, D. J.; Bitton, D. A.; Hannigan, G. D. The Application Potential of Machine Learning and Genomics for Understanding Natural Product Diversity, Chemistry, and Therapeutic Translatability. *Nat. Prod. Rep.* **2021**, *38* (6), 1100–1108. https://doi.org/10.1039/D0NP00055H.

(41) Deo, R. C. Machine Learning in Medicine. *Circulation* **2015**, *132* (20), 1920–1930. https://doi.org/10.1161/CIRCULATIONAHA.115.001593.

(42) Gupta, R.; Srivastava, D.; Sahu, M.; Tiwari, S.; Ambasta, R. K.; Kumar, P. Artificial Intelligence to Deep Learning: Machine Intelligence Approach for Drug Discovery. *Mol Divers* **2021**, *25* (3), 1315–1360. https://doi.org/10.1007/s11030-021-10217-3.

(43) Ellis, G. A.; Thomas, C. S.; Chanana, S.; Adnani, N.; Szachowicz, E.; Braun, D. R.; Harper, M. K.; Wyche, T. P.; Bugni, T. S. Brackish Habitat Dictates Cultivable Actinobacterial Diversity from Marine Sponges. *PLOS ONE* **2017**, *12* (7), e0176968. https://doi.org/10.1371/journal.pone.0176968.

(44) Blunt, J. W.; Copp, B. R.; Hu, W.-P.; Munro, M. H. G.; Northcote, P. T.; Prinsep, M. R. Marine Natural Products. *Nat. Prod. Rep.* **2009**, *26* (2), 170–244. https://doi.org/10.1039/B805113P.

Chapter 2

Marine *Micromonosporaceae* Secondary Metabolite Diversity Analysis

Portions of this chapter have been published in Microbial Genomics as:

Alas, I. *et al. Micromonosporaceae* biosynthetic gene cluster diversity highlights the need for broad-spectrum investigations. *Microbial Genomics* **10**, 001167 (2024).

2.1 INTRODUCTION

Drug resistant infectious diseases have been recognized for decades as a growing threat to humanity[1,2]. More recent crises such as SARS, mpox and, most dramatically, the COVID-19 pandemic have exacerbated the issue significantly due to the increased use of antimicrobials and the predictable acceleration of healthcare-associated drug resistant infections in U.S. hospitals[3,4]. In tandem with these realizations, it has also long been recognized that bacterially-derived secondary metabolites (natural products, NPs) constitute an idealized repository of new drug leads with the potential to display novel mechanisms of action, and thus, the ability to circumvent current drug resistance mechanisms in pathogens[2,5,6]. However, decades of mining terrestrial sources for useful NPs have reduced the likelihood of identifying truly new and novel NPs, despite truly remarkable technical advances that have streamlined drug discovery processes[5,7]. In contrast, recent efforts have revealed marine-derived microbes to be extremely attractive and, now, tractable sources; this is especially true for Actinobacterial populations[7–13]. For instance, the marine *Actinomycete* genus *Salinispora* was prioritized for novel NP explorations and subsequently enabled the discovery of lomaiviticins, salinosporamides, and other antibacterial compounds[13].

Despite their inability to grow in deionized water compared to seawater, *Salinispora* have been disproportionately studied within the marine-derived NP community; the breadth of *Salinispora* studies to date has no doubt been a result of their well noted and chronologically early recognition as sources of NP originality[13–16]. Perceived limitations of *Salinispora* as a genus, and technological advances of the last ten years, have inspired recent campaigns to aggressively evaluate other marine-derived bacteria as sources for new NP scaffolds[12].

For example, phylogenetic analyses of brackish water sponges unveiled proportionally more *Micromonospora* species (spp.) compared to *Streptomyces* spp., contrasting earlier analyses of soil environment isolates[9,17]. Moreover, comparisons of brackish water habitats and tropical reef type habitats have revealed an abundance of four specific genera: *Streptomyces* spp., *Micromonospora* spp., *Solwaraspora* spp., and *Verrucosispora* spp[9]. With distributions of bacterial taxa differing across environments, notable metabolic diversity within *Micromonospora* spp., and the historical emphasis on *Streptomyces* spp. exploration, it is now clear that investigations of marine *Micromonosporaceae* are likely to unveil NPs that differ from those of terrestrial strains, including both general terrestrial bacterial strains and terrestrial *Micromonosporaceae Micromonospora* spp.[9,17,18]. (This study showed that members of the family *Micromonosporaceae* were more abundant among cultivated bacteria from tropical ecosystems versus brackish ones. In addition, metabolomics initiatives have shown their metabolites to differ substantially from those of brackish Streptomyces.)

Notably, the potential of *Micromonospora* to generate antimicrobials was first recognized in 1947 with the discovery of the polyketide micromonosporin and subsequently underscored with the discovery of gentamicin (Gentocin, Garamycin) from M. purpurea in 1963; importantly, gentamicin is listed by the World Health Organization (WHO) as an essential and critically

important medication[19–21]. In the time intervening gentamicin's discovery and now, over 740 antibiotics have been discovered from *Micromonospora* strains although remarkably few have received clinical approval. Not surprisingly, the metabolic capacities of *Micromonospora* family members and their impact on drug discovery initiatives have been the subject of several outstanding reviews in recent years[7,18,22]. Gentamicin, netilimicin, plazomicin (Zemdri), isepamicin, neomycin (neo-Fradin, neo-Tab), and sisomicin (bactoCeaze, Ensamycin) represent, by far, the most clinically successful *Micromonospora*-derived agents and all remain in service today to differing extents based on geographic location and specific antimicrobial applications[18,22]. In addition to aminoglycosides, *Micromonospora* are known to produce antimicrobials bearing macrolide, ansamycin, everninomicin and actinomycin scaffolds[7,18,22,23]. In addition to serving as sources of antimicrobials with activity against "susceptible pathogens" *Micromonospora* have also yielded metabolites with potent activity against multiple-drug-resistant microbes. For instance, turbinmicin, reported in 2020 from *Micromonospora* WMMC-415, displays broad-spectrum activity against multiple-drug-resistant (MDR) fungal pathogens, notably *Candida auris* and *Aspergillus fumigatus*[24,25] and is currently in clinical development. *Micromonospora* have also been shown to produce the enediyne-based DNA-damaging agents calicheamicin, yangpumicins and dynemicin; all hail from *Micromonospora* and have been applied to the creation and use of human therapeutics to differing extents[7,18,22,26–28]. Thus, it is now abundantly clear that *Micromonospora* and associated family members of the *Micromonosporaceae* represent outstanding resources for NP discovery, highlighting environments that produce these bacteria as primary investigative targets.

Concurrent with shifting views on the importance of marine versus terrestrial NP repositories has been the continuous advancement of technologies aimed at acquiring and exploiting genetic

information, specifically, the development of high throughput DNA sequencing methods. Compared to traditional 16S rRNA sequencing and phylogenetic strain classification, whole genome sequencing as used by the Genome Taxonomy Database has allowed for enhanced disambiguation of genomes/organisms[29,30]. Average nucleotide identity (ANI), accessible via whole genome sequencing, now enables novel bacteria to be better understood relative to publicly available literature; one result of this advance has been the ability to delineate classically understood genera into more representative genera[30]. Additional genomic information acquired via whole genome sequencing has shed significant light on orphan Biosynthetic Gene Clusters (BGCs) housed within bacteria and fungi[31]. BGCs represent groupings of genes involved with the production of secondary metabolites, thus linking genomics to the production and intracellular use of bioactive NPs[32]. Relatedly, the refinement of antiSMASH (antibiotics & Secondary Metabolite Analysis Shell) has enabled the expanded exploitation of bacterial genomes in the search for antimicrobial NPs[33]. In conjunction, the construction of databases from public literature sources such as MIBiG (Minimum Information about a Biosynthetic Gene cluster) has allowed for improved use of already described BGCs in the search for new NPs[32].

Importantly, tools like antiSMASH and MIBiG can be leveraged by downstream software such as BiG-SCAPE (Biosynthetic Gene Similarity Clustering and Prospecting Engine) and BiG-SLiCE (Biosynthetic Gene clusters – Super Linear Clustering Engine) to create sequence similarity networks (SSNs) across strain collections[32–35]. These genome-guided computational tools enable one to identify uncharacterized BGCs and their related products across large collections of strains and employ both publicly available literature and privately held datasets[36]. Such an approach minimizes the likelihood of NP rediscovery, a well-known hurdle to effective drug discovery efforts, and maximizes currently known correlations of genomic information to small molecule

structure and function. These newfound capabilities are ideally suited to performing critical analyses regarding new NP potentials of any collection of microbial NP producers one might wish to interrogate.

To date, a broad analysis of marine *Micromonosporaceae* genomes, BGCs related to the production of NPs, and their relation to public databases has been lacking. Hence, I provide here a comparative genomic analysis of marine *Micromonosporaceae* with the intention of identifying truly novel BGCs. Our analyses revealed the biosynthetic diversity of marine *Micromonosporaceae*, especially in comparison to publicly curated databases. Interestingly, we identified 4 new species belonging to *Micromonospora_E*, which is a new genus within the GTDB-tk2 database. These species represent a 4-fold increase in known *Micromonosporaceae Micromonospora_E* compared to the 311,480 bacterial strains stored in GTDB-tk2 from publicly available data. This study showcases: a) the utility of prioritizing BGCs based on novelty, relative to large existing databases; and b) the potential of renewed investigations into marine *Micromonosporaceae* as an outstanding repository for NP scaffolds.

2.2 RESULTS & DISCUSSION

Novel NPs from primarily terrestrial bacteria have served as the first and last lines of defense against antibiotic resistant pathogens for over 90 years[2]. However, trend analyses regarding decades of NP research indicate that terrestrial NP repositories are reaching critical exhaustion levels; in particular, the likelihood of discovering bioactive NPs with clinically employable novel mechanisms of action is becoming diminishingly small. Accordingly, drug discovery initiatives of the last 5-10 years are now aggressively pursuing alternative sources of microbially derived NPs with clinical potential. Inspired by our own interest in marine-derived microbes and the increasingly dire rise of antibiotic resistance, we evaluated the family *Micromonosporaceae*,

specifically marine-associated *Micromonosporaceae*, as a promising source of NPs with bioactivity against clinically relevant pathogens. However, while the dataset used for this study contains a significant number of *Micromonosporaceae* sequenced by our research group, it was not explicitly designed to be a diverse set of bacteria based on phylogenetic grounds and does not represent the entire marine-associated family with regards to metabolic potential. The approach taken, one that critically evaluates BGC content of assorted pools of microbes, enabled us to determine if marine-associated bacteria are sufficiently distinct from previously characterized terrestrial bacteria to warrant in depth marine-focused metabolomics. In this work, I incorporated genomics-based analytics to uncover potential bioactive secondary metabolites unrelated to known molecules based on comparative analyses of microbial BGC content.

2.2.1 Marine bacterial strain genome characteristics

The genome sequences of the 38 selected *Micromonosporaceae* strains, one selected *Streptomycetaceae* strain, associated QUAST (v5.0.2) annotations, and their respective BUSCO information are presented in Data S1. The *Micromonosporaceae* strains G+C content varied from approximately 70.41–73.94%. Genome sizes ranged from 6311079 base pairs (bp) to 8920919 bp, completeness scores ranged from 98.3% to 100%, and the number of genes ranged from 5688 to 8739.

2.2.2 Phylogenetic classification of marine bacterial strains

The phylogenetic classifications of the 38 selected *Micromonosporaceae* strains and one *Streptomycetaceae* strain were performed using GTDB-tk v2 (Data S6). AAI and ANI analyses were performed to identify shared genera and shared species, with a minimum threshold of 67.66% AAI being necessary to belong to the same genera and an upper bound of 95% ANI to be

considered a novel species strain (Data S6). WMMC500 (*Streptomycetaceae Streptomyces*) was found to be a novel species of Streptomyces. WMMD1047 (*Micromonosporaceae*) was uncovered as a novel *Micromonosporaceae* sp. through GTDB-tk v2 taxonomic classification. Utilizing GTDB-tk v2 in conjunction with ANI analysis revealed 18 bacterial strains as belonging to known species of *Micromonosporaceae Micromonospora*. Rauf Salamzade identified 11 bacterial strains as 8 novel species of *Micromonosporaceae Micromonospora*. Further investigation discovered 6 bacterial strains belonging to the *Micromonospora_E* genus, with 4 novel species of *Micromonosporaceae Micromonospora_E* identified. The designation *Micromonospora_E* comes from GTDB-tk2 splitting the current genus of *Micromonospora* into *Micromonospora*, *Micromonospora_E*, *Micromonospora_G*, and *Micromonospora_H* based on average nucleotide identity[37]. GTDB-tk2 also revealed the presence of 2 additional novel species: WMMD1127 (*Micromonosporaceae Asanoa*, unknown sp.) and WMMD1102 (*Micromonosporaceae Plantactinospora*, unknown sp.).

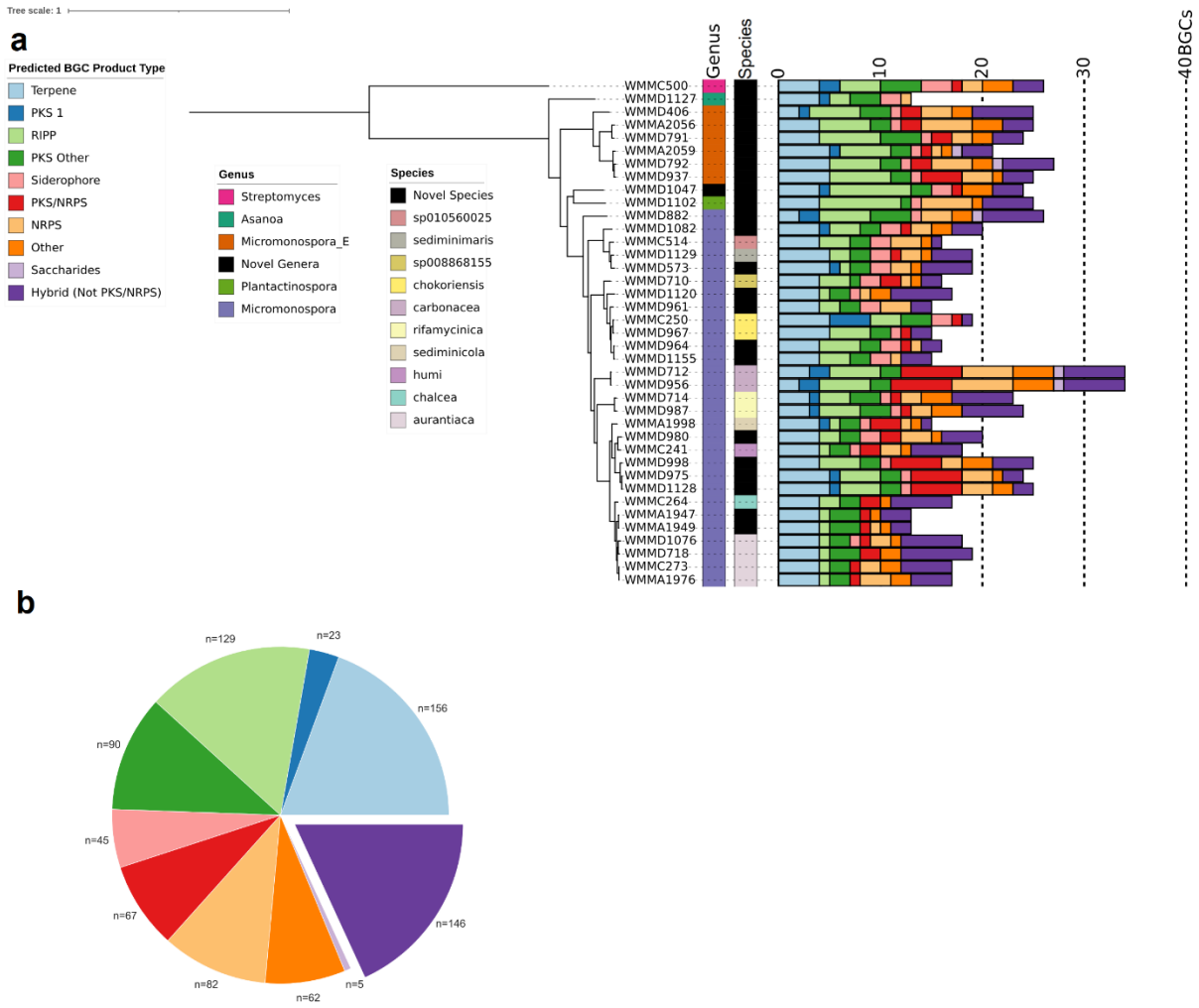


Figure 2.1. Overall representation of phylogeny and BGCs of *Micromonosporaceae* type strains included in this work. **(a)** Phylogenetic tree of 38 *Micromonosporaceae* strains and 1 *Streptomycetaceae* strain (WMMC500). The bar plot represents the antiSMASH v5.1.1 identified BGC regions, and the colors represent the predicted BGC product type. **(b)** Pie chart describing the 779 BGCs identified in the 38 *Micromonosporaceae* strains according to antiSMASH v5.1.1.'s predicted product type.

The phylogenetic analysis of 38 selected *Micromonosporaceae* strains and one *Streptomycetaceae* strain broadly conformed to expectation. WMMC500 (*Streptomycetaceae Streptomyces*, unknown sp.) correctly self-identified as a unique clade in our phylogenetic tree and was predicted as a new species of *Streptomyces* that is distinct from the genomes in GTDB-tk2. This result was expected,

as *Streptomycetaceae* should clearly diverge from *Micromonosporaceae* through taxonomic classification due to inter-genera differences. However, the discovery of 4 novel species of *Micromonospora_E* across 6 bacterial strains emphasizes the diversity of our marine collection, despite being so small. Currently, GTDB release 207 indicates that there are 331 *Micromonosporaceae Micromonospora* genomes, with two corresponding *Micromonosporaceae Micromonospora_E* genomes. Purely from a taxonomic perspective, we have uncovered an additional 6 *Micromonospora_E* strains that belong to a poorly represented genus in publicly available literature. Additionally, we have also discovered a total of 15 novel species of *Micromonosporaceae*. Even though this dataset was primarily cultivated for *Micromonosporaceae* strains, the taxonomic diversity acquired from collections carried out in only one marine environment hints at a dramatic and unexplored diversity in microbial NPs, clearly warranting future study.

2.2.3 Diversity and distribution of BGCs in marine *Micromonosporaceae*

The prediction, annotation, and characterization of BGCs related to secondary metabolism were generated using antiSMASH v5.1.1. From the 38 *Micromonosporaceae* strains, 779 BGCs were annotated with a distribution of 13-34 BGCs per genome (Fig. 2.1). Genomes WMMA1949 (n=13), WMMA1947 (n=13), and WMMD1127 (n=13) all had the lowest number of BGCs identified, while WMMD712 (n=34) and WMMD956 (n=34) carried the highest number of BGCs identified. Manual annotation was done using antiSMASH-predicted BGC product types using a modified BiG-SCAPE categorization scheme to decipher non-PKS/NRPS hybrid clusters, and this annotation revealed a significant number of hybrid products. Across the *Micromonosporaceae* strains, the most heavily represented type of BGC was predicted to be involved in the production of terpenes (n=156), followed by non-PKS/NRPS hybrids (non-polyketide synthase and non-

ribosomal peptide synthetase hybrids) (n=146) and RiPPs (ribosomally synthesized and post-translationally modified peptides) (n=129).

Terpenes, as the most heavily represented BGC type in our marine *Micromonosporaceae*, represent a structurally and functionally diverse family of natural products. Generally speaking, terpenes share a backbone chain assembly pathway that employs C5 units such as isopentenyl diphosphate (IPP) and dimethylallyl diphosphate (DMAPP) as critical building blocks[7]. Terpenes displaying moderate antibacterial activity against Methicillin-Resistant Staphylococcus Aureus (MRSA) and bacteriostatic properties are well established[7]. Terpenoids also have shown antimicrobial, antifungal, and anticancer cytotoxic activity[38–40]. Uncharacterized terpenes identified in our BiG-SCAPE analysis represent a largely underexploited NP grouping for possible applications against antimicrobial resistance mechanisms.

Non-PKS/NRPS hybrids were found to comprise the second largest group of BGCs in our *Micromonosporaceae* collection. I defined non-PKS/NRPS hybrids as antiSMASH annotated regions obeying two specific rules: 1) The region contains a hybrid of two or more products; and 2) If the hybrid contains only two products, those products must not only be a Type 1 PKS and non-ribosomal peptide synthetase (NRPS). As an example, a non-PKS/NRPS hybrid could have annotations for a Type 2 PKS and RiPP or be composed of a Type 1 PKS, NRPS, and Terpene. This super-category was constructed to identify potential chemical hybrids that are traditionally underexplored for interesting chemistry. For instance, in WMMC264, an interleaving candidate cluster containing an NRPS, oligosaccharide, and terpene overlapping was observed. NPs generally display much greater structural diversity than strictly synthetic agents and. accordingly, allow for much more extensive probing and utilization of chemical space; this logic is especially spurred on by the abundance of stereochemical features found in vast numbers of NPs (6). These

realizations make clear that further characterization of novel chemical spaces, as illuminated by our NP discovery efforts here, might well broaden synthetic drug approaches for target diversity[6].

Ribosomally synthesized and post-translationally modified peptides (RiPPs) make up the third-largest proportion of BGCs in our collection of *Micromonosporaceae* strains. RiPPs are primarily notable for being structurally diverse and displaying a wide array of biological activities[41], including potent antibiotic[42], antifungal[43,44], and anticancer[45,46] activities, and expressing unique mechanisms of action compared to clinically used drugs. RiPPs are also amenable to biosynthetic engineering to mitigate systemic issues such as poor solubility and limited bioavailability[42]. Of our 779 *Micromonosporaceae* BGCs analyzed with antiSMASH v5.1.1, 170 were found to encode a RiPP of some kind. Notably, bacteriocin (n=67), lanthipeptide (n=58), linear azol(in)e containing peptides (n=37), thiopeptide (n=36), and Tfua-related (n=20) species comprised over 97% of the RiPPs identified through antiSMASH (Data S13). Further characterization of these BGCs may reveal particularly malleable RiPPs with good antibiotic activities for engineering against resistant pathogens[42].

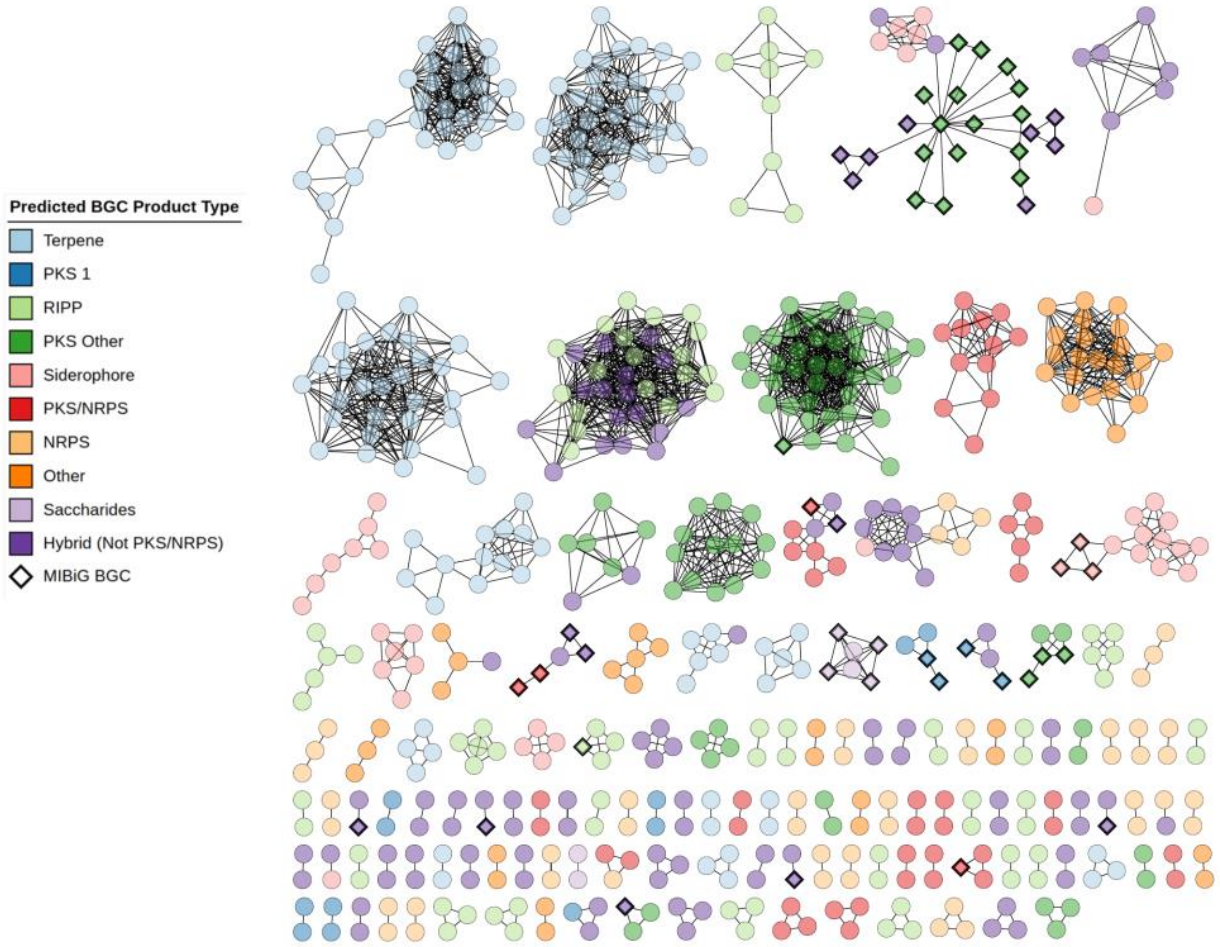


Figure 2.2. Sequence similarity network (SSN) produced by BiG-SCAPE when analyzing 38 *Micromonosporaceae* strains, visualized and annotated with Cytoscape. Nodes represent individual BGCs. BGC types are colored according to the color legend. *Micromonosporaceae* BGCs are represented as circles and MIBiG BGCs are represented as diamonds. Singletons (196 BGCs) have been removed from the visualization.

Analysis of the predicted BGCs, following removal of *Streptomyces* from the dataset, using the BiG-SCAPE workflow revealed 328 GCFs (Gene Cluster Families) and singletons (Fig. 2.2). My analysis identified the 51 MIBiG reference BGCs as similar to our *Micromonosporaceae* BGCs. Specifically, BGCs known to encode the biosynthesis of Gentamicin, Sisomicin, Rosamicin, and more were linked to BGCs in our collection (Data S13). Out of our 779 *Micromonosporaceae* BGCs, 196 BGCs (~25.1%) clustered separately into singletons, representing a degree of

uniqueness worth studying given their likelihood as beacons of unique chemical chemistry/NPs. The remaining 583 *Micromonosporaceae* BGCs clustered into 132 GCFs seen in Figure 2.2, with 83 of the *Micromonosporaceae* BGCs clustering into 16 GCFs containing BGCs from MIBiG, indicating potentially known chemical compound space. The remaining 500 *Micromonosporaceae* BGCs clustered into 116 unique GCFs, traditionally representing unique chemical classes and indicating additional novel chemical entities potentially able to occupy unique chemical spaces. Overall, if all BGCs were sufficiently expressed in lab conditions, I would expect to have observed 312 unique chemical classes from our marine-derived *Micromonosporaceae* collection.

2.2.4 BiG-SliCE queries reveal significant likelihood for previously unknown biosynthetic potentials

The previous analysis only explored the potential chemical diversity within our *Micromonosporaceae* BGCs. To investigate the diversity relative to published terrestrial bacteria sources, BiG-SLiCE was used. BiG-SLiCE contains over ~1.2 million BGCs from publicly available sources, including MIBiG. The majority of the described BGCs are from terrestrial organisms, due in part to exhaustive mining of local ecological niches[1]. For our work, I utilized the BiG-SLiCE model constructed using a default clustering threshold value of 900, resulting in 29,955 GCFs against ~1.2 million BGCs, and queried our 779 marine *Micromonosporaceae* BGCs to determine individual BGC similarity to BiG-SLiCE's GCFs (Fig. 2.3).

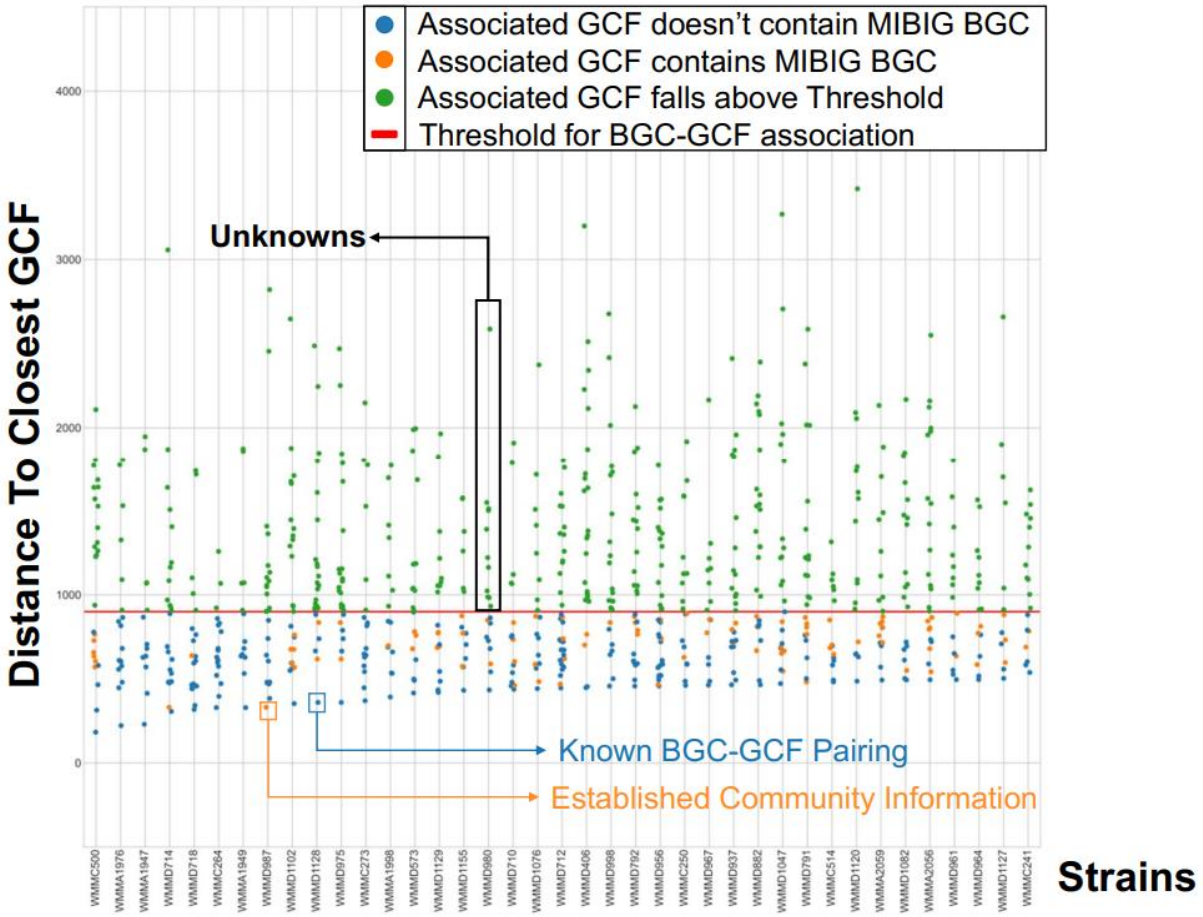


Figure 2.3. Scatter plot of *Micromonosporaceae* and *Streptomycetaceae* BGCs queried against BiG-SLiCE (Threshold = 900). Nodes represent individual BGCs. The horizontal red line indicates the threshold for successful clustering of a BGC into a GCF from BiG-SLiCE. The dots are colored as follows:

1) orange, if the GCF most similar to that BGC contained a BGC from MIBiG.

2) blue, if the GCF most similar to that BGC did not contain a BGC from MIBiG.

3) green, if the BGC's distance to the closest GCF fell above the clustering threshold of 900.

Our analysis revealed 413 BGCs that failed to cluster into the 29,955 GCF model in BiG-SLiCE (clustering threshold T=900); these BGCs thus represent distinctly novel BGCs relative to published literature, an important finding that is perhaps not that surprising given the historical bias towards terrestrial microbes reported in the literature. The remaining 366 BGCs that clustered

were further categorized into two groups, those associated with a cluster defined by association with MIBiG and those that did not. From this analysis, I uncovered 94 MIBiG-associated BGCs. Furthermore, 272 BGCs were identified as being distinct from MIBiG-associated GCFs within BiG-SLiCE. The 94 MIBiG-associated BGCs likely known natural products based on their similarity to community annotated BGCs. The remaining 272 BGCs that were not identified as MIBiG-associated likely result in natural products that are closely associated to knowns. In summation, the 413 BGCs that failed to cluster into the BiG-SLiCE model (T=900) and likely result in unknown chemical classes, with little to no information regarding potential products beyond antiSMASH annotations. Accordingly, these BGCs showcase the vast biosynthetic potential of the marine *Micromonosporaceae*.

To contextualize our findings, I combined our BiG-SCAPE analysis and our BiG-SLiCE results to identify strains that contained novel BGCs in relation to both our own dataset and to published literature. Notably, investigation of WMMD406 (*Micromonospora_E* unknown sp.) in BiG-SCAPE revealed 17 BGC singletons out of the 25 total BGCs identified through antiSMASH. Comparison with BiG-SLiCE queries revealed 21 BGCs that failed to cluster (T=900) into the 29,955 GCFs that represent primarily terrestrial public literature. Of the 17 BGC singletons identified in BiG-SCAPE, 16 were shared across the 21 novel BGCs determined through BiG-SLiCE, indicating potentially unique chemical classes across our dataset and against published literature. Systematic incorporation of BiG-SCAPE analysis and BiG-SLiCE results revealed that the 413 novel BGCs identified through BiG-SLiCE made up 148 of the 196 singletons and 108 of the 132 GCFs visualized in BiG-SCAPE, representing prospective chemical classes (Data S13). Further exploration of marine *Micromonosporaceae* strains will continue to yield new chemical diversity.

2.2.5 BGC distribution of *Micromonospora_E* is statistically different from *Micromonospora*

Micromonospora_E was found to represent an underexploited genusfor BGCs with potentially novel chemistry. In fact, analysis of the distribution of BGC distances to the closest GCF in BiG-SLiCE uncovered a statistically significant difference between the *Micromonospora* and *Micromonospora_E* BGCs with respect to our data collection. Specifically, the mean BGC distance to closest GCF in BiG-SLiCE for *Micromonospora_E* was ~1174, compared to *Micromonospora*'*s* ~1006. Similarly, the median BGC distance to closest GCF in BiG-SLiCE showed the same trend, with *Micromonospora_E* reporting a median of ~1004 and *Micromonospora* of ~908. The Mann-Whitney U test with a two-sided hypothesis yielded a p-value of 6.84e-04, indicating that the BGC distance distributions for *Micromonospora* and *Micromonospora_E* are different. In addition, analysis of the number of BGCs reporting a distance to the closest GCF in BiG-SLiCE above the threshold value (T=900), averaged over the number of strains per genera, revealed for *Micromonospora* a total of ~10.03 BGCs/strain, compared to *Micromonospora_E* with ~14.83 BGCs/strain. Although the BiG-SLiCE pre-processed dataset was developed in 2021, access to future datasets incorporating more publicly available genomes and metagenomes will decrease the overall distances reported. However, this metric makes clear that, if I were to query a marine-derived *Micromonospora* strain's BGCs against BiG-SLiCE's currently available pre-processed dataset, I would expect to see approximately 10 BGCs that would report a distance value of 900 or greater, representing dissimilarity to BiG-SLiCE's primarily terrestrial dataset, indicating potentially novel chemistry.

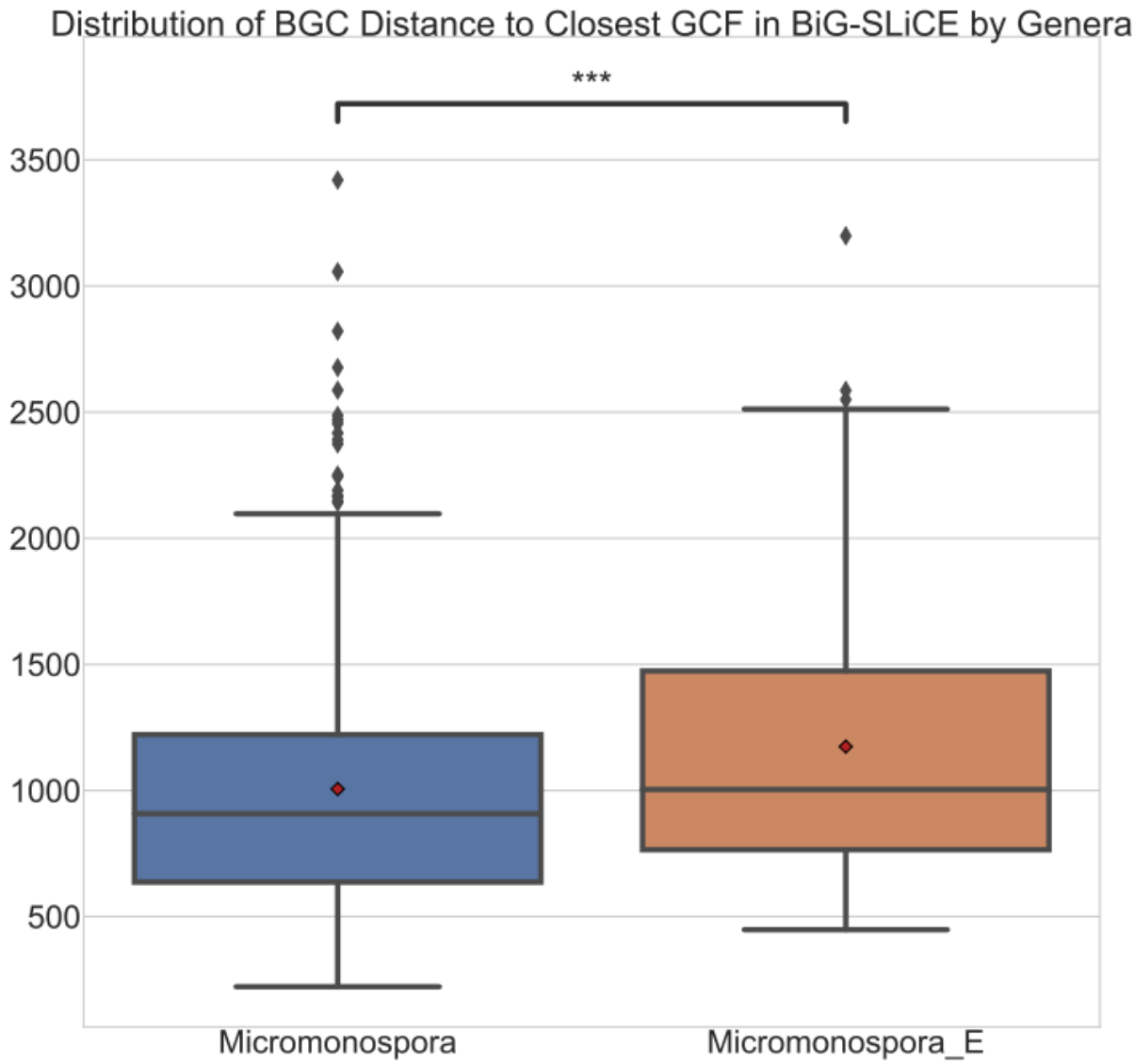


Figure 2.4. Box and whisker plots comparing the distribution of distances between individual BGCs and the associated closest GCFs in BiG-SLiCE (Threshold = 900). The distributions have been separated by genera, with 6 *Micromonospora_E* strains and 29 *Micromonospora.* The red dots represent the means of the distributions. Mann-Whitney U-test (double-sided) was used to confirm differences in the distributions.

2.3 MATERIALS & METHODS

2.3.1 Strain isolation & extraction

Bacterial strains were isolated using previously published methodologies[47]. 16S rRNA sequences were extracted using standard methods[47]. DNA sequences were extracted using standard protocols for our laboratory[48].

2.3.2 Strain sequencing, assembly, & validation

Genome sequencing used PacBio Sequel platforms using two Sequel single-molecule real-time (SMRT) cells (University of Wisconsin-Madison [UW-Madison], Biotechnology Center). PacBio data were corrected, trimmed, and assembled with Canu v1.8[49] using the parameter "genomeSize=8m" (UW-Madison, Center for High Throughput Computing). BUSCO v5.4.3[50] and QUAST v5.0.2[51] was used to assess each genome assembly based on completeness and quality respectively, and the results are listed in Data S4 and S6.

2.3.3 Annotation of biosynthetic gene clusters

To identify BGCs related to secondary metabolism, genome sequences in the Fasta format were annotated by installations of antiSMASH v5.1.1[52] and antiSMASH v6.1.0[53]. For antiSMASH v5.1.1, a minimal run omitting additional features was carried out using the Antibiotic Resistant Target Seeker Version 2 (ARTS)[54,55]. For antiSMASH v6.1.0, BGC prediction was performed with detection strictness relaxed, and the extra features KnownClusterBlast, ActiveSiteFinder, SubClusterBlast, RREFinder were active. The full parameters for the antiSMASH runs are listed in Data S8. The detailed information for the 39 strains is listed in DataS4, S5, and S6.

Pie plot of overall BGC counts per predicted product type was constructed using seaborn (v0.11.2)[56] in Python (v3.7.12)[57].

2.3.4 Phylogenomics, average identity estimation between genomic pairs, & taxonomic classification

We identified 922 single-copy core orthologs between the genomes of 39 isolates using OrthoFinder v2.5.4[58]. Multiple sequence alignment of protein sequences was performed for each ortholog using MUSCLE v5[59] and filtered for sites which featured gaps in more than 10% of samples using trimAI v1.4[60]. Filtered protein alignments were concatenated to produce an alignment of length 292,788 aa with coordinates for individual alignments of the 922 orthologs used to generate input for IQ-Tree v2.2.0.3[61] phylogeny construction using an edge-proportional partition model with ModelFinder Plus[62] and visualized using iTOL v6.6[63] in Figure 2.1a. CompareM v0.1.2 (https://github.com/dparks1134/CompareM) was used to assess average amino acid identity (AAI) between the 39 genomes. Genome Taxonomy Database (GTDB-tk v2)[30] was used to classify taxonomies for the 39 genomes with GTDB release R207[37], and the taxonomic identification results are listed in Data S6. FastANI[64] was used to group species across the 39 genomes using average nucleotide identity (ANI), and the species grouping results are listed in Data S6. This process was performed by Rauf Salamzade.

2.3.5 Analysis of intra-BGC diversity using marine *Micromonosporaceae*

Utilizing the *Micromonosporaceae* antiSMASH v5.1.1 GenBank files, pairwise distance across predicted BGCs and sequence similarity networks (SSNs) was constructed using BiG-SCAPE v1.1.0[34]. This run of BiG-SCAPE was performed a default cutoff of 0.3 with additional features such as the following: including BGCs from the MIBiG database v2.1[65], constructing an all-vs-

all distance network mixing all BGC product classes, including BGCs that do not have a distance lower than the cutoff distance specified, and including BGCs with hybrid predicted products into each subclass network. The full parameters for the BiG-SCAPE analysis are listed in Data S8. The network output of BiG-SCAPE was imported into Cytoscape v.3.9.1[66] (Figure 2.2). The BGC product annotations generated by antiSMASH v5.1.1 were categorized based on BiG-SCAPE cluster classes and were added to the Cytoscape visualization. The MIBiG BGC product types were identified using minimal runs of antiSMASH v5.1.1 through ARTS[54,55], manually annotated using BiG-SCAPE cluster classes categories, and added as annotations to the Cytoscape network.

2.3.6 Analysis of BGC diversity against terrestrial bacteria

Micromonosporaceae antiSMASH v5.1.1 outputs were queried against the pre-processed dataset of 29,955 Gene Cluster Family (GCF) models in BiG-SLiCE v1.1.1[35] using a clustering threshold value of 900. The 29,955 GCF models were previously computed by Kautsar et. al.[35] using a clustering threshold distance of 900 to group 1,225,071 BGCs into GCFs. For each BGC-GCF pairing, a membership score (distance) was generated, resulting in 779 BGCs having an associated 29,955 distances for each possible GCF. These distances corresponding to BGC-GCF instances were ranked from lowest to highest to generate the top-X hits, where X represents the X-best GCF hits for each BGC, with a lower distance indicating higher confidence in similarity. The full parameters for the BiG-SLiCE runs are listed in Data S8. The file-based SQL database was accessed in Python using the SQLite3 library. The best-performing GCF for each *Micromonosporaceae* BGC was investigated for the presence of MIBiG BGCs and the distance values associated with the BGCs and their top-1 GCF pair was plotted in Figure 2.3.

2.3.7 Statistical analysis of BGC similarity distribution across genera

The 570 *Micromonospora* BGCs and 147 *Micromonospora_E* BGCs were queried against BiG-SLiCE for the best GCF membership score. A comparison of the distributions of BGCs across genera was constructed using a box plot (Fig. 2.4, Table S1). The Mann-Whitney U test was applied to evaluate the statistically significance differences across the genera-specific distributions (Table S1). All analyses utilized a standard significance level of $p < 0.05$.

2.4 DATA SUMMARY

Large datasets (mostly as tables) and special files can be found in Zenodo (https://zenodo.org/) and Figshare (https://figshare.com/) under the following links: https://zenodo.org/records/8208940 and https://doi.org/10.6084/m9.figshare.24492253.v1. In particular, the following collections of data for this paper are included:

- Data S1: A folder with all the fasta files, representing the 42 strains (41 *Micromonosporaceae*, 1 *Streptomycetaceae*).
- Data S2: A folder with all the .gbk files for the biosynthetic gene cluster (BGC) regions predicted by antiSMASH v5.1.1. These files were used as inputs for BiG-SCAPE and BiG-SLiCE.
- Data S3: A folder with all the .gbk files for the BGC regions predicted by antiSMASH v6.1.0.
- Data S4: A folder containing all the Quast outputs for the 42 strains.
- Data S5: A folder containing all the BUSCO outputs for the 42 strains. Example scripts are provided for scraping relevant information from the individual BUSCO outputs.
- Data S6: A folder containing GTDB (Genome Taxonomy Database) classification results, and species-level grouping results using FastANI (95% cutoff).
- Data S7: A folder containing an Interactive Tree of Life (iTOL)-compatible bar chart annotation using antiSMASH v5.1.1 BGC region information.

- Data S8: A folder containing a Word document that describes the parameters used with Ubuntu WSL (Windows Subsystem for Linux) on the command line for programs antiSMASH v6.1.0, BiG-SCAPE v1.1.2 and BiG-SLiCE v1.1.1. Also included are parameters for metric MDS in python. An example script is also provided for batch queries of BGCs against BiG-SLiCE v1.1.1's pre-processed dataset of ~1.2 million BGCs.
- Data S9: A folder containing the BiG-SCAPE visualization of the 38 *Micromonosporaceae* (post-QC filtering, excluding WMMA1363, WMMB482, WMMB486 and WMMC500) in Cytoscape.
- Data S10: A folder containing:
    - The pre-processed dataset of 1.2 million BGCs from BiG-SLiCE.
    - All report folders generated by BiG-SLiCE for the 779 *Micromonosporaceae* BGCs queried against the 1.2 million BGCs.
    - The results data.db and associated folders for the pre-processed dataset of 1.2 million BGCs.
- Data S11: A folder containing the scripts necessary to regenerate the figures and perform independent analyses, and the relevant data used for the analyses.
- Data S12: A folder containing the NCBI blast query used to compare WMMD1947 region 12 against WMMD1120 region 14 using antiSMASH v6.1.0.

- Data S13: A folder containing the files pertaining to RiPP subclasses, BiG-SLiCE BGCs annotated with BiG-SCAPE information, and MIBiG BGCs identified as being related to BGCs in our dataset.

Table S1 and Figures S1-181 are located in the Supporting Information seen here: https://zenodo.org/records/10952346.

2.5 REFERENCES

(1) Aminov, R. I. A Brief History of the Antibiotic Era: Lessons Learned and Challenges for the Future. *Front Microbiol* **2010**, *1*, 134. https://doi.org/10.3389/fmicb.2010.00134.

(2) Rossiter, S. E.; Fletcher, M. H.; Wuest, W. M. Natural Products as Platforms To Overcome Antibiotic Resistance. *Chem. Rev.* **2017**, *117* (19), 12415–12474. https://doi.org/10.1021/acs.chemrev.7b00283.

(3) Segala, F. V.; Bavaro, D. F.; Di Gennaro, F.; Salvati, F.; Marotta, C.; Saracino, A.; Murri, R.; Fantoni, M. Impact of SARS-CoV-2 Epidemic on Antimicrobial Resistance: A Literature Review. *Viruses* **2021**, *13* (11), 2110. https://doi.org/10.3390/v13112110.

(4) *COVID-19: U.S. Impact on Antimicrobial Resistance, Special Report 2022*; National Center for Emerging and Zoonotic Infectious Diseases, 2022. https://doi.org/10.15620/cdc:117915.

(5) Patridge, E.; Gareiss, P.; Kinch, M. S.; Hoyer, D. An Analysis of FDA-Approved Drugs: Natural Products and Their Derivatives. *Drug Discovery Today* **2016**, *21* (2), 204–207. https://doi.org/10.1016/j.drudis.2015.01.009.

(6) Stratton, C. F.; Newman, D. J.; Tan, D. S. Cheminformatic Comparison of Approved Drugs from Natural Product versus Synthetic Origins. *Bioorganic & Medicinal Chemistry Letters* **2015**, *25* (21), 4802–4807. https://doi.org/10.1016/j.bmcl.2015.07.014.

(7) Qi, S.; Gui, M.; Li, H.; Yu, C.; Li, H.; Zeng, Z.; Sun, P. Secondary Metabolites from Marine *Micromonospora*: Chemistry and Bioactivities. *Chemistry & Biodiversity* **2020**, *17* (4), e2000024. https://doi.org/10.1002/cbdv.202000024.

(8) Manivasagan, P.; Venkatesan, J.; Sivakumar, K.; Kim, S.-K. Pharmaceutically Active Secondary Metabolites of Marine Actinobacteria. *Microbiological Research* **2014**, *169* (4), 262–278. https://doi.org/10.1016/j.micres.2013.07.014.

(9) Ellis, G. A.; Thomas, C. S.; Chanana, S.; Adnani, N.; Szachowicz, E.; Braun, D. R.; Harper, M. K.; Wyche, T. P.; Bugni, T. S. Brackish Habitat Dictates Cultivable Actinobacterial Diversity from Marine Sponges. *PLOS ONE* **2017**, *12* (7), e0176968. https://doi.org/10.1371/journal.pone.0176968.

(10) Jensen, P. R.; Gontang, E.; Mafnas, C.; Mincer, T. J.; Fenical, W. Culturable Marine Actinomycete Diversity from Tropical Pacific Ocean Sediments. *Environmental Microbiology* **2005**, *7* (7), 1039–1048. https://doi.org/10.1111/j.1462-2920.2005.00785.x.

(11) Selim, M. S. M.; Abdelhamid, S. A.; Mohamed, S. S. Secondary Metabolites and Biodiversity of Actinomycetes. *J Genet Eng Biotechnol* **2021**, *19*, 72. https://doi.org/10.1186/s43141-021-00156-9.

(12) Subramani, R.; Sipkema, D. Marine Rare Actinomycetes: A Promising Source of Structurally Diverse and Unique Novel Natural Products. *Mar Drugs* **2019**, *17* (5), 249. https://doi.org/10.3390/md17050249.

(13) Jensen, P. R.; Moore, B. S.; Fenical, W. The Marine Actinomycete Genus Salinispora: A Model Organism for Secondary Metabolite Discovery. *Nat Prod Rep* **2015**, *32* (5), 738–751. https://doi.org/10.1039/c4np00167b.

(14) Kim, H.; Kim, S.; Kim, M.; Lee, C.; Yang, I.; Nam, S.-J. Bioactive Natural Products from the Genus Salinospora: A Review. *Arch. Pharm. Res.* **2020**, *43* (12), 1230–1258. https://doi.org/10.1007/s12272-020-01288-1.

(15) Ziemert, N.; Lechner, A.; Wietz, M.; Millán-Aguiñaga, N.; Chavarria, K. L.; Jensen, P. R. Diversity and Evolution of Secondary Metabolism in the Marine Actinomycete Genus Salinispora. *Proc Natl Acad Sci U S A* **2014**, *111* (12), E1130–E1139. https://doi.org/10.1073/pnas.1324161111.

(16) Letzel, A.-C.; Li, J.; Amos, G. C. A.; Millán-Aguiñaga, N.; Ginigini, J.; Abdelmohsen, U. R.; Gaudêncio, S. P.; Ziemert, N.; Moore, B. S.; Jensen, P. R. Genomic Insights into Specialized Metabolism in the Marine Actinomycete Salinispora. *Environmental Microbiology* **2017**, *19* (9), 3660–3673. https://doi.org/10.1111/1462-2920.13867.

(17) Janssen, P. H. Identifying the Dominant Soil Bacterial Taxa in Libraries of 16S rRNA and 16S rRNA Genes. *Appl Environ Microbiol* **2006**, *72* (3), 1719–1728. https://doi.org/10.1128/AEM.72.3.1719-1728.2006.

(18) Yan, S.; Zeng, M.; Wang, H.; Zhang, H. *Micromonospora*: A Prolific Source of Bioactive Secondary Metabolites with Therapeutic Potential. *J. Med. Chem.* **2022**, *65* (13), 8735–8771. https://doi.org/10.1021/acs.jmedchem.2c00626.

(19) Weinstein, M. J.; Luedemann, G. M.; Oden, E. M.; Wagman, G. H.; Rosselet, J. P.; Marquez, J. A.; Coniglio, C. T.; Charney, W.; Herzog, H. L.; Black, J. Gentamicin,1 a New Antibiotic Complex from *Micromonospora*. *J. Med. Chem.* **1963**, *6* (4), 463–464. https://doi.org/10.1021/jm00340a034.

(20) Organization, W. H. World Health Organization Model List of Essential Medicines: 21st List 2019. **2019**.

(21) Waksman, S. A.; Geiger, W. B.; Bugie, E. Micromonosporin, an Antibiotic Substance from a Little-Known Group of Microorganisms. *J Bacteriol* **1947**, *53* (3), 355–357. https://doi.org/10.1128/jb.53.3.355-357.1947.

(22) Hifnawy, M. S.; Fouda, M. M.; Sayed, A. M.; Mohammed, R.; Hassan, H. M.; AbouZid, S. F.; Rateb, M. E.; Keller, A.; Adamek, M.; Ziemert, N.; Abdelmohsen, U. R. The Genus *Micromonospora* as a Model Microorganism for Bioactive Natural Product Discovery. *RSC Adv.* **2020**, *10* (35), 20939–20959. https://doi.org/10.1039/D0RA04025H.

(23) Dulin, C. C.; Sharma, P.; Frigo, L.; Voehler, M. W.; Iverson, T. M.; Bachmann, B. O. EvdS6 Is a Bifunctional Decarboxylase from the Everninomicin Gene Cluster. *Journal of Biological Chemistry* **2023**, *299* (7), 104893. https://doi.org/10.1016/j.jbc.2023.104893.

(24) Zhang, F.; Zhao, M.; Braun, D. R.; Ericksen, S. S.; Piotrowski, J. S.; Nelson, J.; Peng, J.; Ananiev, G. E.; Chanana, S.; Barns, K.; Fossen, J.; Sanchez, H.; Chevrette, M. G.; Guzei, I. A.; Zhao, C.; Guo, L.; Tang, W.; Currie, C. R.; Rajski, S. R.; Audhya, A.; Andes, D. R.; Bugni, T. S.

A Marine Microbiome Antifungal Targets Urgent-Threat Drug-Resistant Fungi. *Science* **2020**, *370* (6519), 974–978. https://doi.org/10.1126/science.abd6919.

(25) Zhao, M.; Zhang, F.; Zarnowski, R.; Barns, K.; Jones, R.; Fossen, J.; Sanchez, H.; Rajski, S. R.; Audhya, A.; Bugni, T. S.; Andes, D. R. Turbinmicin Inhibits Candida Biofilm Growth by Disrupting Fungal Vesicle–Mediated Trafficking. *J Clin Invest 131* (5), e145123. https://doi.org/10.1172/JCI145123.

(26) Wang, Z.; Wen, Z.; Liu, L.; Zhu, X.; Shen, B.; Yan, X.; Duan, Y.; Huang, Y. Yangpumicins F and G, Enediyne Congeners from *Micromonospora Yangpuensis* DSM 45577. *J. Nat. Prod.* **2019**, *82* (9), 2483–2488. https://doi.org/10.1021/acs.jnatprod.9b00229.

(27) Yan, X.; Chen, J.-J.; Adhikari, A.; Yang, D.; Crnovcic, I.; Wang, N.; Chang, C.-Y.; Rader, C.; Shen, B. Genome Mining of *Micromonospora Yangpuensis* DSM 45577 as a Producer of an Anthraquinone-Fused Enediyne. *Org. Lett.* **2017**, *19* (22), 6192–6195. https://doi.org/10.1021/acs.orglett.7b03120.

(28) Braesel, J.; Crnkovic, C. M.; Kunstman, K. J.; Green, S. J.; Maienschein-Cline, M.; Orjala, J.; Murphy, B. T.; Eustáquio, A. S. Complete Genome of *Micromonospora* Sp. Strain B006 Reveals Biosynthetic Potential of a Lake Michigan Actinomycete. *J. Nat. Prod.* **2018**, *81* (9), 2057–2068. https://doi.org/10.1021/acs.jnatprod.8b00394.

(29) Parks, D. H.; Chuvochina, M.; Rinke, C.; Mussig, A. J.; Chaumeil, P.-A.; Hugenholtz, P. GTDB: An Ongoing Census of Bacterial and Archaeal Diversity through a Phylogenetically Consistent, Rank Normalized and Complete Genome-Based Taxonomy. *Nucleic Acids Research* **2022**, *50* (D1), D785–D794. https://doi.org/10.1093/nar/gkab776.

(30) Chaumeil, P.-A.; Mussig, A. J.; Hugenholtz, P.; Parks, D. H. GTDB-Tk v2: Memory Friendly Classification with the Genome Taxonomy Database. *Bioinformatics* **2022**, btac672. https://doi.org/10.1093/bioinformatics/btac672.

(31) Hoskisson, P. A.; Seipke, R. F. Cryptic or Silent? The Known Unknowns, Unknown Knowns, and Unknown Unknowns of Secondary Metabolism. *mBio* **2020**, *11* (5), e02642-20. https://doi.org/10.1128/mBio.02642-20.

(32) Terlouw, B. R.; Blin, K.; Navarro-Muñoz, J. C.; Avalon, N. E.; Chevrette, M. G.; Egbert, S.; Lee, S.; Meijer, D.; Recchia, M. J. J.; Reitz, Z. L.; van Santen, J. A.; Selem-Mojica, N.; Tørring, T.; Zaroubi, L.; Alanjary, M.; Aleti, G.; Aguilar, C.; Al-Salihi, S. A. A.; Augustijn, H. E.; Avelar-Rivas, J. A.; Avitia-Domínguez, L. A.; Barona-Gómez, F.; Bernaldo-Agüero, J.; Bielinski, V. A.; Biermann, F.; Booth, T. J.; Carrion Bravo, V. J.; Castelo-Branco, R.; Chagas, F. O.; Cruz-Morales, P.; Du, C.; Duncan, K. R.; Gavriilidou, A.; Gayrard, D.; Gutiérrez-García, K.; Haslinger, K.; Helfrich, E. J. N.; van der Hooft, J. J. J.; Jati, A. P.; Kalkreuter, E.; Kalyvas, N.; Kang, K. B.; Kautsar, S.; Kim, W.; Kunjapur, A. M.; Li, Y.-X.; Lin, G.-M.; Loureiro, C.; Louwen, J. J. R.; Louwen, N. L. L.; Lund, G.; Parra, J.; Philmus, B.; Pourmohsenin, B.; Pronk, L. J. U.; Rego, A.; Rex, D. A. B.; Robinson, S.; Rosas-Becerra, L. R.; Roxborough, E. T.; Schorn, M. A.; Scobie, D. J.; Singh, K. S.; Sokolova, N.; Tang, X.; Udwary, D.; Vigneshwari, A.; Vind, K.; Vromans, S. P. J. M.; Waschulin, V.; Williams, S. E.; Winter, J. M.; Witte, T. E.; Xie, H.; Yang, D.; Yu, J.; Zdouc, M.; Zhong, Z.; Collemare, J.; Linington, R. G.; Weber, T.; Medema, M. H. MIBiG 3.0: A Community-Driven Effort to Annotate Experimentally Validated

Biosynthetic Gene Clusters. *Nucleic Acids Research* **2023**, *51* (D1), D603–D610. https://doi.org/10.1093/nar/gkac1049.

(33) Blin, K.; Shaw, S.; Augustijn, H. E.; Reitz, Z. L.; Biermann, F.; Alanjary, M.; Fetter, A.; Terlouw, B. R.; Metcalf, W. W.; Helfrich, E. J. N.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 7.0: New and Improved Predictions for Detection, Regulation, Chemical Structures and Visualisation. *Nucleic Acids Research* **2023**, *51* (W1), W46–W50. https://doi.org/10.1093/nar/gkad344.

(34) Navarro-Muñoz, J. C.; Selem-Mojica, N.; Mullowney, M. W.; Kautsar, S. A.; Tryon, J. H.; Parkinson, E. I.; De Los Santos, E. L. C.; Yeong, M.; Cruz-Morales, P.; Abubucker, S.; Roeters, A.; Lokhorst, W.; Fernandez-Guerra, A.; Cappelini, L. T. D.; Goering, A. W.; Thomson, R. J.; Metcalf, W. W.; Kelleher, N. L.; Barona-Gomez, F.; Medema, M. H. A Computational Framework to Explore Large-Scale Biosynthetic Diversity. *Nat Chem Biol* **2020**, *16* (1), 60–68. https://doi.org/10.1038/s41589-019-0400-9.

(35) Kautsar, S. A.; van der Hooft, J. J. J.; de Ridder, D.; Medema, M. H. BiG-SLiCE: A Highly Scalable Tool Maps the Diversity of 1.2 Million Biosynthetic Gene Clusters. *GigaScience* **2021**, *10* (1), giaa154. https://doi.org/10.1093/gigascience/giaa154.

(36) Saati-Santamaría, Z.; Selem-Mojica, N.; Peral-Aranega, E.; Rivas, R.; García-Fraile, P. Unveiling the Genomic Potential of Pseudomonas Type Strains for Discovering New Natural Products. *Microb Genom* **2022**, *8* (2), 000758. https://doi.org/10.1099/mgen.0.000758.

(37) Parks, D. H.; Chuvochina, M.; Waite, D. W.; Rinke, C.; Skarshewski, A.; Chaumeil, P.-A.; Hugenholtz, P. A Standardized Bacterial Taxonomy Based on Genome Phylogeny Substantially Revises the Tree of Life. *Nat Biotechnol* **2018**, *36* (10), 996–1004. https://doi.org/10.1038/nbt.4229.

(38) Guimarães, A. C.; Meireles, L. M.; Lemos, M. F.; Guimarães, M. C. C.; Endringer, D. C.; Fronza, M.; Scherer, R. Antibacterial Activity of Terpenes and Terpenoids Present in Essential Oils. *Molecules* **2019**, *24* (13), 2471. https://doi.org/10.3390/molecules24132471.

(39) Tomko, A. M.; Whynot, E. G.; Ellis, L. D.; Dupré, D. J. Anti-Cancer Potential of Cannabinoids, Terpenes, and Flavonoids Present in Cannabis. *Cancers (Basel)* **2020**, *12* (7), 1985. https://doi.org/10.3390/cancers12071985.

(40) Rao, A.; Zhang, Y.; Muend, S.; Rao, R. Mechanism of Antifungal Activity of Terpenoid Phenols Resembles Calcium Stress and Inhibition of the TOR Pathway. *Antimicrob Agents Chemother* **2010**, *54* (12), 5062–5069. https://doi.org/10.1128/AAC.01050-10.

(41) Scherlach, K.; Hertweck, C. Mining and Unearthing Hidden Biosynthetic Potential. *Nat Commun* **2021**, *12* (1), 3864. https://doi.org/10.1038/s41467-021-24133-5.

(42) Hudson, G. A.; Mitchell, D. A. RiPP Antibiotics: Biosynthesis and Engineering Potential. *Current Opinion in Microbiology* **2018**, *45*, 61–69. https://doi.org/10.1016/j.mib.2018.02.010.

(43) Hetrick, K. J.; van der Donk, W. A. Ribosomally Synthesized and Post-Translationally Modified Peptide Natural Product Discovery in the Genomic Era. *Curr Opin Chem Biol* **2017**, *38*, 36–44. https://doi.org/10.1016/j.cbpa.2017.02.005.

(44) Mohr, K. I.; Volz, C.; Jansen, R.; Wray, V.; Hoffmann, J.; Bernecker, S.; Wink, J.; Gerth, K.; Stadler, M.; Müller, R. Pinensins: The First Antifungal Lantibiotics. *Angewandte Chemie International Edition* **2015**, *54* (38), 11254–11258. https://doi.org/10.1002/anie.201500927.

(45) Russell, A. H.; Truman, A. W. Genome Mining Strategies for Ribosomally Synthesised and Post-Translationally Modified Peptides. *Comput Struct Biotechnol J* **2020**, *18*, 1838–1851. https://doi.org/10.1016/j.csbj.2020.06.032.

(46) Frattaruolo, L.; Lacret, R.; Cappello, A. R.; Truman, A. W. A Genomics-Based Approach Identifies a Thioviridamide-Like Compound with Selective Anticancer Activity. *ACS Chem. Biol.* **2017**, *12* (11), 2815–2822. https://doi.org/10.1021/acschembio.7b00677.

(47) Wyche, T. P.; Hou, Y.; Braun, D.; Cohen, H. C.; Xiong, M. P.; Bugni, T. S. First Natural Analogs of the Cytotoxic Thiodepsipeptide Thiocoraline A from a Marine Verrucosispora Sp. *J. Org. Chem.* **2011**, *76* (16), 6542–6547. https://doi.org/10.1021/jo200661n.

(48) Adnani, N.; Chevrette, M. G.; Adibhatla, S. N.; Zhang, F.; Yu, Q.; Braun, D. R.; Nelson, J.; Simpkins, S. W.; McDonald, B. R.; Myers, C. L.; Piotrowski, J. S.; Thompson, C. J.; Currie, C. R.; Li, L.; Rajski, S. R.; Bugni, T. S. Coculture of Marine Invertebrate-Associated Bacteria and Interdisciplinary Technologies Enable Biosynthesis and Discovery of a New Antibiotic, Keyicin. *ACS Chem. Biol.* **2017**, *12* (12), 3093–3102. https://doi.org/10.1021/acschembio.7b00688.

(49) Koren, S.; Walenz, B. P.; Berlin, K.; Miller, J. R.; Bergman, N. H.; Phillippy, A. M. Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation. *Genome Res.* **2017**, *27* (5), 722–736. https://doi.org/10.1101/gr.215087.116.

(50) *BUSCO Update: Novel and Streamlined Workflows along with Broader and Deeper Phylogenetic Coverage for Scoring of Eukaryotic, Prokaryotic, and Viral Genomes | Molecular Biology and Evolution | Oxford Academic*. https://academic.oup.com/mbe/article/38/10/4647/6329644 (accessed 2022-10-24).

(51) Mikheenko, A.; Prjibelski, A.; Saveliev, V.; Antipov, D.; Gurevich, A. Versatile Genome Assembly Evaluation with QUAST-LG. *Bioinformatics* **2018**, *34* (13), i142–i150. https://doi.org/10.1093/bioinformatics/bty266.

(52) Blin, K.; Shaw, S.; Steinke, K.; Villebro, R.; Ziemert, N.; Lee, S. Y.; Medema, M. H.; Weber, T. antiSMASH 5.0: Updates to the Secondary Metabolite Genome Mining Pipeline. *Nucleic Acids Research* **2019**, *47* (W1), W81–W87. https://doi.org/10.1093/nar/gkz310.

(53) Blin, K.; Shaw, S.; Kloosterman, A. M.; Charlop-Powers, Z.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 6.0: Improving Cluster Detection and Comparison Capabilities. *Nucleic Acids Research* **2021**, *49* (W1), W29–W35. https://doi.org/10.1093/nar/gkab335.

(54) Alanjary, M.; Kronmiller, B.; Adamek, M.; Blin, K.; Weber, T.; Huson, D.; Philmus, B.; Ziemert, N. The Antibiotic Resistant Target Seeker (ARTS), an Exploration Engine for Antibiotic Cluster Prioritization and Novel Drug Target Discovery. *Nucleic Acids Research* **2017**, *45* (W1), W42–W48. https://doi.org/10.1093/nar/gkx360.

(55) Mungan, M. D.; Alanjary, M.; Blin, K.; Weber, T.; Medema, M. H.; Ziemert, N. ARTS 2.0: Feature Updates and Expansion of the Antibiotic Resistant Target Seeker for Comparative

Genome Mining. *Nucleic Acids Research* **2020**, *48* (W1), W546–W552. https://doi.org/10.1093/nar/gkaa374.

(56) Waskom, M. L. Seaborn: Statistical Data Visualization. *Journal of Open Source Software* **2021**, *6* (60), 3021. https://doi.org/10.21105/joss.03021.

(57) Van Rossum, G.; Drake, F. L. *Python 3 Reference Manual*; CreateSpace: Scotts Valley, CA, 2009.

(58) Emms, D. M.; Kelly, S. OrthoFinder: Phylogenetic Orthology Inference for Comparative Genomics. *Genome Biology* **2019**, *20* (1), 238. https://doi.org/10.1186/s13059-019-1832-y.

(59) Edgar, R. C. MUSCLE v5 Enables Improved Estimates of Phylogenetic Tree Confidence by Ensemble Bootstrapping. bioRxiv June 21, 2021, p 2021.06.20.449169. https://doi.org/10.1101/2021.06.20.449169.

(60) Capella-Gutiérrez, S.; Silla-Martínez, J. M.; Gabaldón, T. trimAl: A Tool for Automated Alignment Trimming in Large-Scale Phylogenetic Analyses. *Bioinformatics* **2009**, *25* (15), 1972–1973. https://doi.org/10.1093/bioinformatics/btp348.

(61) Minh, B. Q.; Schmidt, H. A.; Chernomor, O.; Schrempf, D.; Woodhams, M. D.; von Haeseler, A.; Lanfear, R. IQ-TREE 2: New Models and Efficient Methods for Phylogenetic Inference in the Genomic Era. *Molecular Biology and Evolution* **2020**, *37* (5), 1530–1534. https://doi.org/10.1093/molbev/msaa015.

(62) Kalyaanamoorthy, S.; Minh, B. Q.; Wong, T. K. F.; von Haeseler, A.; Jermiin, L. S. ModelFinder: Fast Model Selection for Accurate Phylogenetic Estimates. *Nat Methods* **2017**, *14* (6), 587–589. https://doi.org/10.1038/nmeth.4285.

(63) Letunic, I.; Bork, P. Interactive Tree Of Life (iTOL) v5: An Online Tool for Phylogenetic Tree Display and Annotation. *Nucleic Acids Research* **2021**, *49* (W1), W293–W296. https://doi.org/10.1093/nar/gkab301.

(64) Jain, C.; Rodriguez-R, L. M.; Phillippy, A. M.; Konstantinidis, K. T.; Aluru, S. High Throughput ANI Analysis of 90K Prokaryotic Genomes Reveals Clear Species Boundaries. *Nat Commun* **2018**, *9* (1), 5114. https://doi.org/10.1038/s41467-018-07641-9.

(65) Kautsar, S. A.; Blin, K.; Shaw, S.; Navarro-Muñoz, J. C.; Terlouw, B. R.; van der Hooft, J. J. J.; van Santen, J. A.; Tracanna, V.; Suarez Duran, H. G.; Pascal Andreu, V.; Selem-Mojica, N.; Alanjary, M.; Robinson, S. L.; Lund, G.; Epstein, S. C.; Sisto, A. C.; Charkoudian, L. K.; Collemare, J.; Linington, R. G.; Weber, T.; Medema, M. H. MIBiG 2.0: A Repository for Biosynthetic Gene Clusters of Known Function. *Nucleic Acids Research* **2020**, *48* (D1), D454–D458. https://doi.org/10.1093/nar/gkz882.

(66) Shannon, P.; Markiel, A.; Ozier, O.; Baliga, N. S.; Wang, J. T.; Ramage, D.; Amin, N.; Schwikowski, B.; Ideker, T. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Res* **2003**, *13* (11), 2498–2504. https://doi.org/10.1101/gr.1239303.

Chapter 3

Development of Biosynthetic Gene Cluster Prioritization Dashboard Using Predicted Secondary Metabolites Through Visualization

Portions of this chapter have been published in *Microbial Genomics* as:
Alas, I. *et al.* Micromonosporaceae biosynthetic gene cluster diversity highlights the need for broad-spectrum investigations. *Microbial Genomics* **10**, 001167 (2024).

3.1 INTRODUCTION

For an overarching motivation regarding the prioritization of biosynthetic gene clusters (BGCs), see Section 2.1. Briefly, healthcare-associated drug resistant infections have skyrocketed in recent years, and a limited number of antimicrobial drug leads have developed following the "Golden Age"[1–4]. Due to most bacterial strains characterized for potential antibacterial compounds being isolated from soil, the rediscovery rate of finding compounds already explored is very high[5,6]. Other environments, like brackish water habitats and tropical reef habitats showcase different bacterial genera compared to soil environments, serving as the rationale for exploring marine environments for unexplored compounds[7]. The bacterial family *Micromonosporaceae*, especially, is abundant in marine environments and has historically produced famous antimicrobials such as gentamicin, and are relatively understudied compared to the ever-popular bacterial genus *Streptomyces*[8–11].

Computational tools such as BiG-SCAPE and BiG-SLiCE allow researchers to investigate their BGC collections for similar BGCs in both local collections and public databases[12,13]. They are invaluable for capturing which BGCs' likely produce compounds already studied, and in doing so, allow researchers to observe novel BGCs showcasing novel chemistry. By leveraging

antiSMASH information from bacterial genomes, they break down whole sets of BGCs into digestible chunks of chemical classes[14].

We sought to combine information from BiG-SLiCE and BiG-SCAPE to get a snapshot of the BGCs truly unrelated to all other BGCs both within our dataset and in public literature. However, neither program lends itself to doing so in a systematic fashion, and that is what the BGC Prioritization Dashboard attempts to approximate. By leveraging BiG-SLiCE information about public database BGCs, we can mesh the ability of BiG-SCAPE to compare BGCs within our dataset with the novelty-identifying power of BiG-SLiCE, thus clearly pointing researchers towards BGCs of interest based on prospective new chemistry for prioritization based on visual information (Figure 3.1).

The BiG-SLiCE Prioritization Dashboard leverages antiSMASH data fed into BiG-SLiCE to quantify similarity of individual BGCs to GCFs in BiG-FAM's pre-processed database of approximately 1.2 million BGCs[15]. I leverage this similarity information and compare each BGC against each other, to capture how identical each BGC is to each other, effectively inferring similarity between BGCs from information acquired in BiG-SLiCE.

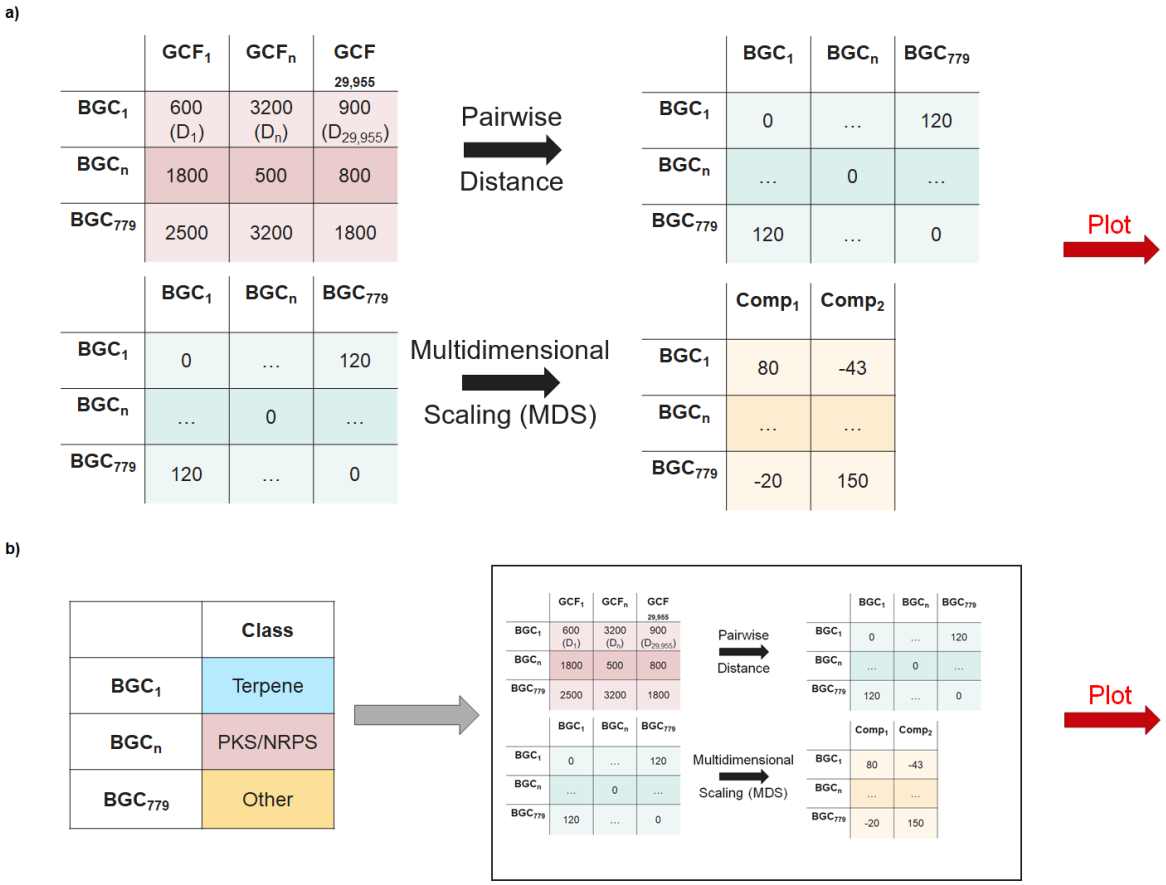


Figure 3.1. Overview of BiG-SLiCE Prioritization Dashboard methodology. **(a)** Initial methodology for inferring BGC similarity through BiG-SLiCE data. Each BGC is queried against BiG-SLiCE's pre-processed database of 29,955 GCFs. The distance to each GCF per BGC is stored in a matrix. Each row of the matrix, representing individual BGCs, was compared against each other using pairwise distance metrics. The resultant 779x779 matrix underwent metric multidimensional scaling (MDS), resulting in a 779x2 dataset. **(b)** Prior to comparing each BGC against each other using pairwise distance metrics, the dataset of 779 BGCs was split into sub-datasets based on BiG-SCAPE classes. Subsequent analysis is identical to **(a)**.

By reducing the data down for the purposes of visualization, I allow researchers to explore a visually digestible interface to determine which BGCs would be worth exploring based on true novelty. In addition to the tools BiG-SCAPE and BiG-SLiCE, I present this prioritization display as an overlay for researchers to explore and justify prioritization of BGCs. The display also allows for researchers to curate the visualizations to understand the underlying patterns in the

data, emphasizing the display as a visualization tool rather than a strict view of the BGCs within a given dataset. This dashboard showcases the practical ability to help researchers understand which BGCs to prioritize in their datasets in a systematic fashion.

3.2 RESULTS & DISCUSSION

For descriptions of the marine *Micromonosporaceae* dataset as described in Section 3.2, see Sections 2.2.1-2.2.3..

3.2.1 Truly novel BGCs are identifiable using metric MDS and pairwise distance modeling

Our previous analyses highlighted the merits of combining BiG-SCAPE and BiG-SLiCE to identify strains and BGCs that are distinct within our dataset and distinct relative to published literature sources (Figure 3.1). To do this in a systematic fashion, I constructed a pairwise distance matrix (Chebyshev) of our 779 BGCs using relational information regarding individual BGC distances to all 29,955 GCFs in BiG-SLiCE (T=900). Using metric MDS, I visualized similarity of our marine *Micromonosporaceae* BGCs against each other, and split them into subplots to show BGC similarity across individual BGC product types (Figure 3.2). From these subplots, I observed siderophores, terpenes, and RiPPs as primarily co-localizing to similar spaces based on similar GCF distance values in BiG-SLiCE. Notably, the BiG-SLiCE authors have highlighted that BGCs primarily encoding for terpenes and RiPPs tend to disproportionately over-cluster due to the presence of fewer extracted features; our observations here are consistent with these earlier postulates[13]. It also is worth noting that, in the context of Figure 3.2, the Hybrid (Not PKS/NRPS) category served as the positive control since it is not limited to a singular product class. Variation in this specific case was defined as the largest spread in absolute distance between BGCs belonging to the same category. Manual investigation of Hybrid BGCs revealed that, in specific overlapping

cases, an individual component of the Hybrid BGC was shared with the overlapping BGCs in other categories; the instances of overlap were confirmed via BLAST alignments. Specifically, WMMA1947 region 12, encoding putative siderophore and lanthipeptide units, showed 80.74% identity in a BLAST nucleotide comparison with the siderophore subunit's core biosynthetic gene relative to a siderophore moiety predicted to be produced by WMMD1120 region 14, also a siderophore core biosynthetic gene, seen in Data S12.

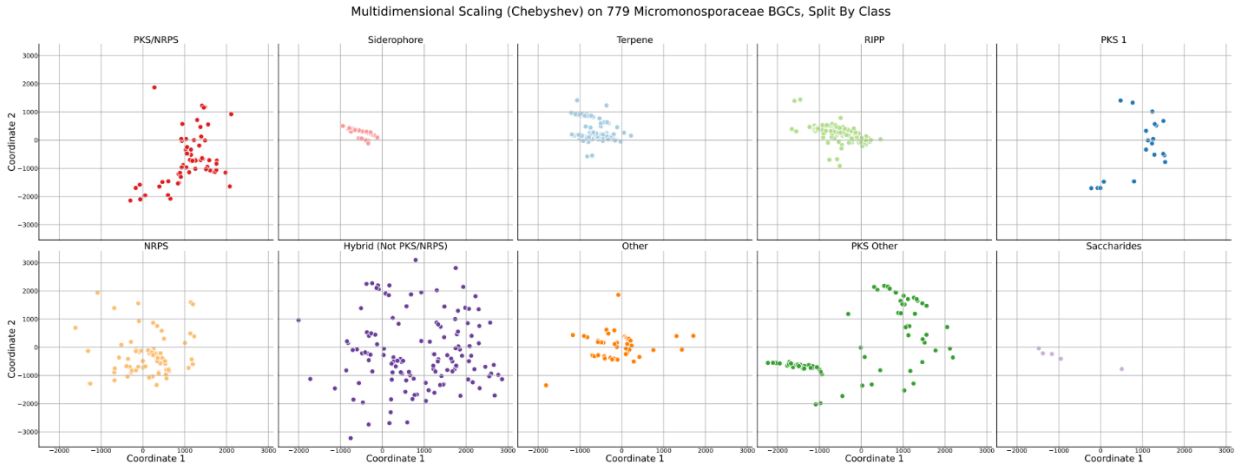


Figure 3.2. Scatter plots of *Micromonosporaceae* BGCs analyzed via metric multidimensional scaling using Chebyshev pairwise distance. Each dot represents an individual BGC. Distance between BGCs is associated with the GCFs in BiG-SLiCE (Threshold=900) they were most similar to. Visualization was separated post-analysis according to predicted BGC product type. The color code represents the predicted BGC product type. Constructed using methodology seen in Figure 3.1a.

To determine how BGCs were localized following metric MDS, I scaled the marker size and color of each plot in Figure 3.2 based on the average distance of the BGC to the top-3 GCFs in BiG-SLiCE. An example visualization of PKS/NRPS can be seen in Figure 3.3, which as a class was selected due to ease of rationalizing BGC similarity based on BiG-SCAPE clustering. Further investigation into specific BGCs that were close in distance revealed that our metric MDS methodology primarily captured variation across BGC product types but that it was insufficient to

resolve similarity across BGCs belonging to the same predicted BGC product type (Figure 3.1a). Although there is utility in comparing across BGC product types, especially when comparing BGCs that fall within the Hybrid (Not PKS/NRPS) category to other BGC predicted product categories that may share singular components to the Hybrid section, further refinements were clearly warranted.

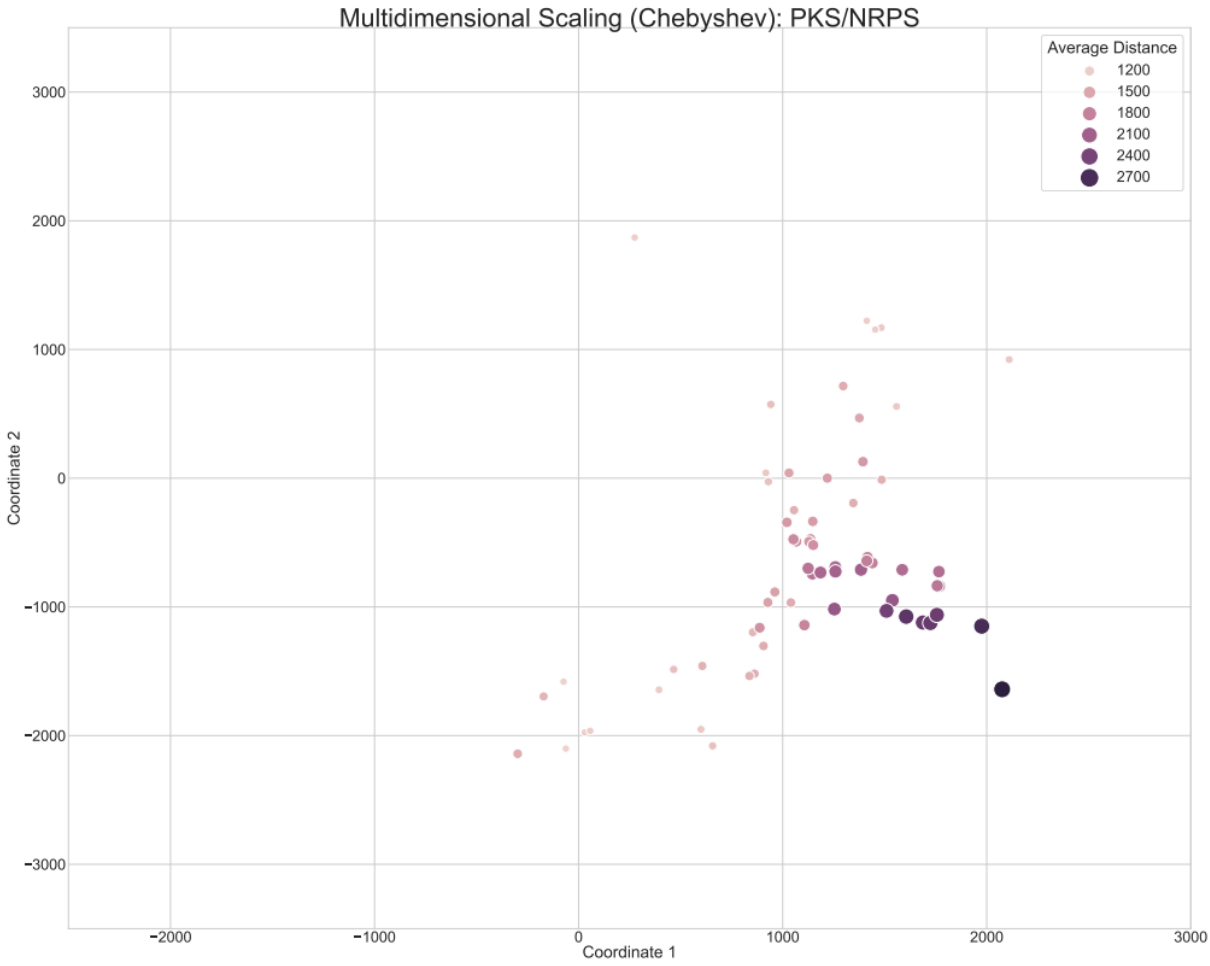


Figure 3.3. Annotated version of the scatter plot depicted in **Figure 3.2** of PKS/NRPS *Micromonosporaceae* BGCs. The color and size of the dots were scaled based on the average distance of the BGCs, across the entire dataset, to the closest 3 GCFs in BiG-SLiCE (Threshold=900).

As such, I separated our BGCxGCF dataset into sub-datasets based on the 10 predicted product types used within this paper, then constructed pairwise distance matrices using the distance metrics described earlier, and finally performed metric MDS to enhance the power of our visualizations to resolve similar BGCs within our *Micromonosporaceae* (Figures 3.1b, 3.4). This enhanced resolution allows for users to prioritize novel BGCs based on the visualization that aligned with BiG-SLiCE and BiG-SCAPE information, without needing to run BiG-SCAPE.

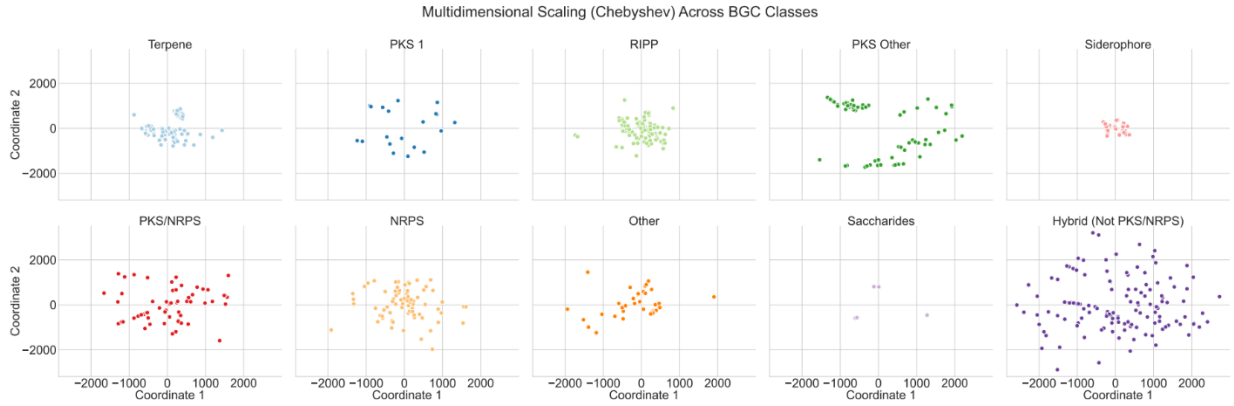


Figure 3.4. Improved scatter plots of *Micromonosporaceae* BGCs, separated by predicted BGC product type prior to analysis via metric multidimensional scaling with Chebvyshev pairwise distance. Distances between BGCs of a specific product type are associated with similar GCFs in BiG-SLiCE (Threshold=900). Predicted BGC product types are color coded. Constructed using methodology seen in Figure 3.1b.

Utilizing this methodology, manual comparison with BiG-SCAPE's SSNs and examination of antiSMASH outputs revealed that the distance between BGCs in the metric MDS visualizations corresponded strongly to the presence of shared BGC elements. For instance, WMMD975 region 20, reported in BiG-SCAPE as part of a triplet with WMMD1128 region 24 and D998 region 3, showed a shared PKS/NRPS unit. Investigation of the PKS/NRPS metric MDS revealed that WMMD975 region 20 and WMMD1128 overlapped significantly, with WMMD998 region 3 distanced further away. AntiSMASH analysis (v5.1.1) revealed that WMMD975 region 20 and

WMMD1128 shared the same core biosynthetic genes for the PKS and NRPS subunits, while WMMD998 contained an additional NRPS core biosynthetic gene, potentially resulting in further modifications to the prospective compound. This methodology allowed us to visually ascertain similarities across BGCs of the same predicted product type, while retaining antiSMASH-level information for the differentiation of BGC pockets.

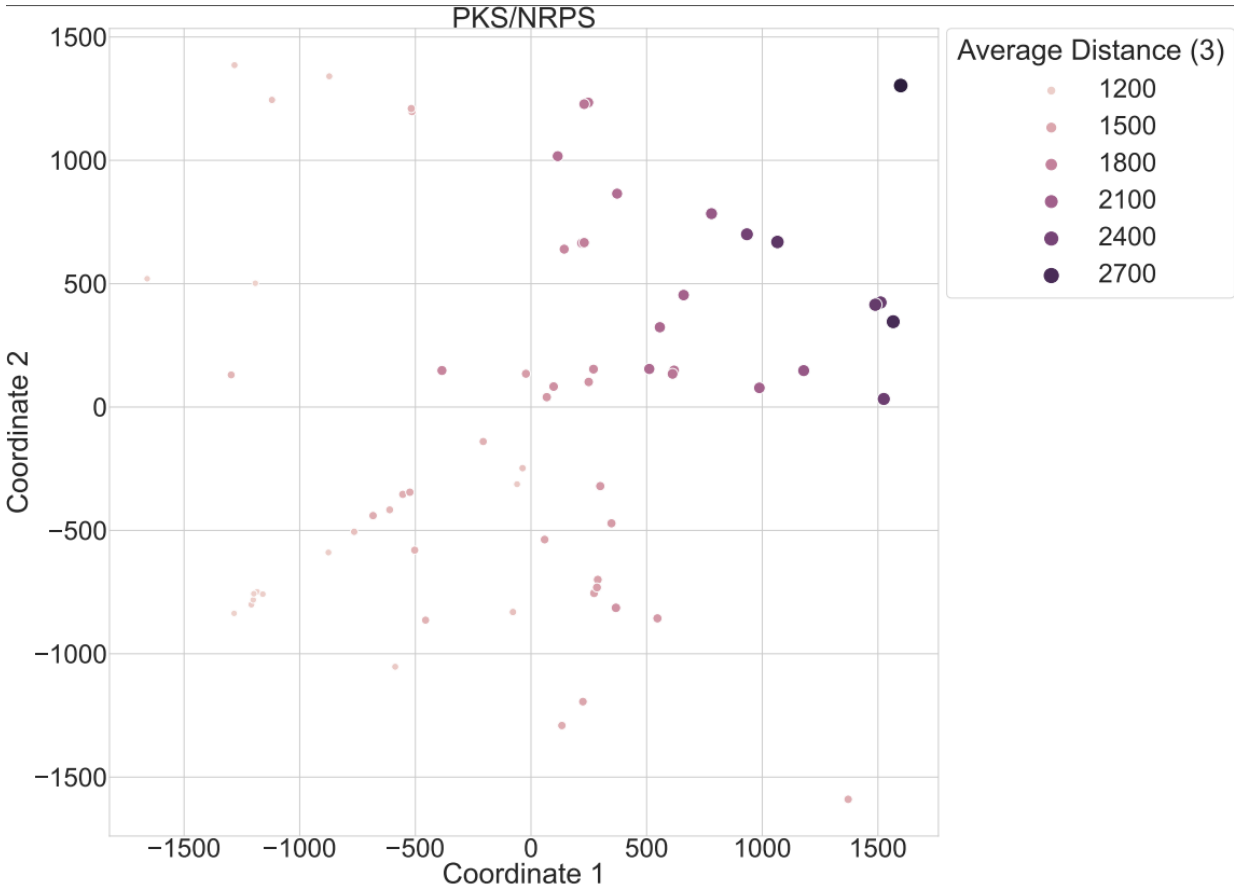


Figure 3.5. Annotated version of the improved scatter plot of PKS/NRPS *Micromonosporaceae* BGCs from **Figure 3.4.** The color and size of the dots were scaled based on the average distance, across the predicted BGC product type PKS/NRPS, to the top-3 GCFs in BiG-SLiCE (Threshold=900).

In particular, scaling the marker size and color based on the average distance of individual BGCs to their top-3 GCFs in BiG-SLiCE provided a more visually informative way to identify BGCs

deviating from terrestrially derived bacteria (Figure 3.5). Analysis of the PKS/NRPS category following Chebyshev-based MDS analysis revealed pockets of BGCs able to serve as representative markers of unique groups. For instance, the information regarding WMMD975, WMMD1128, and WMMD998's PKS/NRPS subunits were consistent; the average distance to the closest 3 GCFs in BIG-SLiCE for all BGCs exceeded 2400, indicating the likelihood of novel chemistry. Incorporation of the average distance in our methodology allowed for a visually informative way to further prioritize BGCs that are likely uncharacterized and novel compared to publicly available terrestrial data from BIG-SLiCE's ~1.2 million BGCs (29,955 GCFs, T=900).

3.2.2 Prioritization of BGCs through BGCPD

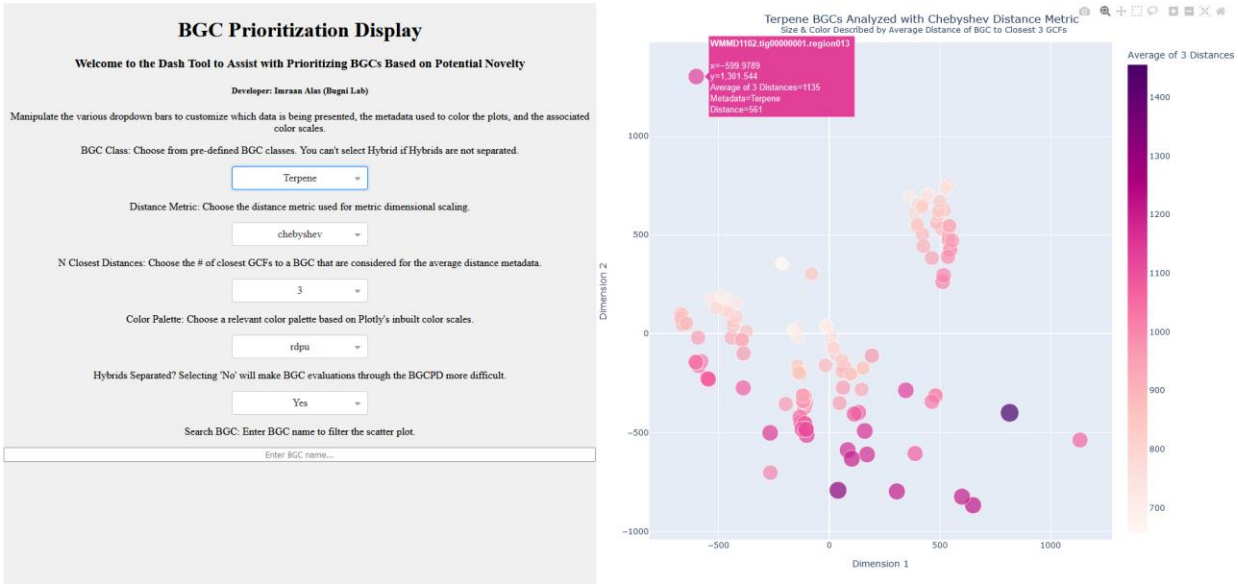


Figure 3.6. Visualization of the BGC Prioritization Dashboard. Analysis was run using only Terpene BGCs with the Chebyshev distance metric. The size and color of the dots were derived from N Closest Distances. Hybrids (of any Terpene/X combination, where X is any other BGC class), were separated for the analysis, and are not present here. Each dot has hover text that contains information on BGC classes, distance to closest GCF in BiG-SLiCE, and average distance to closest Y GCFs, where Y is set by N Closest Distances. BGC names are searchable.

The BGC Prioritization Dashboard allows for the investigation of BGCs in a manner described in Section 3.2.1, in a systematic fashion. After running the associated script, researchers can interact

with their entire BGC dataset to identify BGCs that are sufficiently distinct from their collections as well as public databases. For example, in Figure 3.6, WMMD1102.tig00000001.region013 is highlighted as it shows up separated from all the other Terpene BGCs in the *Micromonosporaceae* dataset. Investigating this further, we can see that the distance to the closest GCF in BiG-SLiCE for this BGC is only 561, well below the threshold of 900 used for the analysis. By shifting the N Closest Distances metric from 1-5, we can see that the 2nd through 5th closest GCFs are well above the threshold of 900. As such, the WMMD1102 BGC is likely similar to only one GCF in BiG-SLiCE and dissimilar to everything else. Any other Terpene BGCs that trend closer towards the center are likely similar to other GCFs, and the closer they overlap, the more similar the distribution of similar GCFs is to each other. In comparison, WMMD406.tig00000001.region014, which is located at (815,400) on the visualization, shows an average distance to the closest 3 GCFs of ~1500. Further investigation revealed that the closest GCF was a distance of 1353 away, well above the threshold of 900. Since the WMMD406 BGC is relatively separate from all the other BGCs, it would likely serve as a good representative to induce expression of for the purposes of novel chemistry or structures.

However, while the BGC Prioritization Dashboard does include the analyses done under differing distance metrics beyond Chebyshev, such as the ones described in Section 3.2.1, I found that the Chebyshev distance metric was the most informative for the dataset I utilized. Specifically, highly overlapping BGCs in Figure 3.6 are likely to be part of the same GCFs in BiG-SCAPE when analyzed through the Chebyshev distance metric, whereas other distance metrics may be teasing other unknown information out of the distance matrixes.

In addition, the BGC Prioritization Dashboard also allows for researchers to choose between two distinct rules for the class a BGC can be referred to as. In Figure 3.6, the “Hybrids Separated?”

section constructs two versions of the dataset to be compared against each other. If “Yes”, this means that a BGC is only considered part of a BGC Class if it exactly contains only that class of BGC. So, for example, a Terpene/RiPP BGC would not be considered a Terpene or RiPP BGC, and would be separated into the remainder category of Hybrid (Not PKS/NRPS). The rationale behind this decision was that the meaning of distance between BGCs improved significantly once the hybrids were separated into their own category. If “No”, then a BGC is considered part of a BGC Class if any of BGC classes were detected in the BGC region using antiSMASH. If a BGC was considered a Terpene/RiPP, it would be included in both the Terpene class and the RiPP class. This improves the ability to interact with these hybrid BGCs that may lead to more interesting chemistry, but at the cost of resolution.

3.3 MATERIALS & METHODS

3.3.1 Strain isolation & extraction

Bacterial strains were isolated using previously published methodologies[16]. The 16S rRNA sequences were extracted using standard methods[16]. DNA sequences were extracted using standard protocols for our laboratory[17].

3.3.2 Strain sequencing, assembly & validation

Genome sequencing used PacBio Sequel platforms using two Sequel single-molecule real-time (SMRT) cells (University of Wisconsin, Madison [UW–Madison], Biotechnology Center). PacBio data were corrected, trimmed, and assembled with Canu v1.8[18] using the parameter "genomeSize=8m" (University of Wisconsin, Madison [UW–Madison], Center for High Throughput Computing). BUSCO v5.4.3[19] and QUAST v5.0.2[20] was used to assess each genome assembly based on completeness and quality, respectively, and the results are listed in Data S4 and S6.

3.3.3 Annotation of biosynthetic gene clusters

To identify BGCs related to secondary metabolism, genome sequences in the Fasta format were annotated by installations of antiSMASH v5.1.1[21] and antiSMASH v6.1.0[22]. For antiSMASH v5.1.1, a minimal run omitting additional features was carried out using the Antibiotic Resistant Target Seeker Version 2 (ARTS)[23,24]. For antiSMASH v6.1.0, BGC prediction was performed with detection strictness relaxed, and the extra features KnownClusterBlast, ActiveSiteFinder, SubClusterBlast, and RREFinder were active. The full parameters for the antiSMASH runs are listed in Data S8. The detailed information for the 39 strains is listed in DataS4, S5, and S6.

3.3.4 Analysis of BGC diversity against terrestrial bacteria

Micromonosporaceae antiSMASH v5.1.1 outputs were queried against the pre-processed dataset of 29,955 Gene Cluster Family (GCF) models in BiG-SLiCE v1.1.1[13] using a clustering threshold value of 900. The 29,955 GCF models had been previously computed by Kautsar et. al.[13] using a clustering threshold distance of 900 to group 1,225,071 BGCs into GCFs. For each BGC-GCF pairing, a membership score (distance) was generated, resulting in 779 BGCs having an associated 29,955 distances for each possible GCF. These distances corresponding to BGC-GCF instances were ranked from lowest to highest to generate the top-X hits, where X represents the X-best GCF hits for each BGC and where a lower distance indicating higher confidence in similarity. The full parameters for the BiG-SLiCE runs are listed in Data S8. The file-based SQL database was accessed in Python using the SQLite3 library. The best-performing GCF for each *Micromonosporaceae* BGC was investigated for the presence of MIBiG BGCs, and the distance values associated with the BGCs and their top-1 GCF pair was plotted in Figure 2.3.

3.3.5 Initial methodology for identification of novel BGCs

The table of membership scores (distances) between 779 *Micromonosporaceae* BGCs and the 29,955 GCFs in BiG-SLiCE was transformed into a pairwise distance matrix using the Chebyshev distance metric, resulting in a 779x779 matrix describing the distance between BGCs in relation to the distribution of membership scores (distances) across the GCFs. Metric (MDS) was performed as a dimensionality reduction technique to reduce the 779x779 matrix into a 2-dimensional space, retaining distances between BGCs as closely as possible[25]. The metric MDS parameters are described in Data S8, and the corresponding metric MDS plots are visualized in Figure 3.2 (Figures S1-S10). The MDS scatter plot marker size and color were scaled based on the average distance of the BGC to the top-3 GCFs (Figures 3.3, S11-S20).

The BGC-GCF table of the 779 BGCs and their membership scores (distances) to the 29,955 GCFs in BiG-SLiCE was split into sub-datasets. These sub-datasets were based on the antiSMASH predicted product type of the BGC, resulting in 10 sub-datasets. Each sub-dataset was transformed into pairwise distance matrices, using 8 different distance metrics: Euclidean, Cosine, Cityblock, Chebyshev, L2, Braycurtis, Canberra, and Correlation pairwise distances. The pairwise distance matrices underwent metric MDS to compare our BGCs against each other (Figures 3.4, S21-S100). The size and color of the markers for each BGC in the metric MDS visualizations were scaled based on the average distance of the BGC to the top-3 GCFs (Figures 3.5, S101-S180). The metric MDS parameters are described in Data S8.

3.3.6 Robust refinements to identifying novel BGCs

The script in Data S11 has been redeveloped, primarily towards minimizing the need for manual categorization and excessive dependencies. The improvements were:

1) Automation of the manual categorization of antiSMASH BGCs into BiG-SCAPE cluster categories[12].
2) Automatic generation of pairwise distance matrices based on rules for the Hybrid (Not PKS/NRPS) category, resulting in either 8 or 9 sub-datasets for metric MDS.
3) Supporting average distance of BGC to closest X GCFs for a range of X between 1 and 5.
4) Removal of code only necessary for figures generated in prior publication[26].
5) Various improvements based on general coding best practices.

3.3.7 Development of visualization dashboard for BGC prioritization

The associated script was re-implemented in Plotly Dash as an application to enhance user capabilities for the visualization and prioritization of BGCs based on the pairwise distance metric

MDS methodology[27,28]. This data visualization dashboard (see Figure 3.6) allows for the customization of the metric MDS scatter plot based on BGC class, distance metric, X closest GCFs used for average distance annotation, color palette, and re-organization of data based on presence of BGC classes or Hybrid (not PKS/NRPS) class separation.

3.4 DATA SUMMARY

Large datasets (mostly as tables) and special files can be found in Zenodo (https://zenodo.org/) and Figshare (https://figshare.com/) under the following links: https://zenodo.org/records/8208940 and https://doi.org/10.6084/m9.figshare.24492253.v1. In particular, the following collections of data for this paper are included:

- Data S1: A folder with all the fasta files, representing the 42 strains (41 *Micromonosporaceae*, 1 *Streptomycetaceae*).
- Data S2: A folder with all the .gbk files for the biosynthetic gene cluster (BGC) regions predicted by antiSMASH v5.1.1. These files were used as inputs for BiG-SCAPE and BiG-SLiCE.
- Data S3: A folder with all the .gbk files for the BGC regions predicted by antiSMASH v6.1.0.
- Data S4: A folder containing all the Quast outputs for the 42 strains.
- Data S5: A folder containing all the BUSCO outputs for the 42 strains. Example scripts are provided for scraping relevant information from the individual BUSCO outputs.
- Data S6: A folder containing GTDB (Genome Taxonomy Database) classification results, and species-level grouping results using FastANI (95% cutoff).

- Data S7: A folder containing an Interactive Tree of Life (iTOL)-compatible bar chart annotation using antiSMASH v5.1.1 BGC region information.
- Data S8: A folder containing a Word document that describes the parameters used with Ubuntu WSL (Windows Subsystem for Linux) on the command line for programs antiSMASH v6.1.0, BiG-SCAPE v1.1.2 and BiG-SLiCE v1.1.1. Also included are parameters for metric MDS in python. An example script is also provided for batch queries of BGCs against BiG-SLiCE v1.1.1's pre-processed dataset of ~1.2 million BGCs.
- Data S9: A folder containing the BiG-SCAPE visualization of the 38 *Micromonosporaceae* (post-QC filtering, excluding WMMA1363, WMMB482, WMMB486 and WMMC500) in Cytoscape.
- Data S10: A folder containing:
    - The pre-processed dataset of 1.2 million BGCs from BiG-SLiCE.
    - All report folders generated by BiG-SLiCE for the 779 *Micromonosporaceae* BGCs queried against the 1.2 million BGCs.
    - The results data.db and associated folders for the pre-processed dataset of 1.2 million BGCs.
- Data S11: A folder containing the scripts necessary to regenerate the figures and perform independent analyses, and the relevant data used for the analyses.
- Data S12: A folder containing the NCBI blast query used to compare WMMD1947 region 12 against WMMD1120 region 14 using antiSMASH v6.1.0.

- Data S13: A folder containing the files pertaining to RiPP subclasses, BiG-SLiCE BGCs annotated with BiG-SCAPE information, and MIBiG BGCs identified as being related to BGCs in our dataset.

Table S1 and Figures S1-181 are located in the Supplementary Information, seen here: https://zenodo.org/records/10952346.

3.5 REFERENCES

(1) Segala, F. V.; Bavaro, D. F.; Di Gennaro, F.; Salvati, F.; Marotta, C.; Saracino, A.; Murri, R.; Fantoni, M. Impact of SARS-CoV-2 Epidemic on Antimicrobial Resistance: A Literature Review. *Viruses* **2021**, *13* (11), 2110. https://doi.org/10.3390/v13112110.

(2) *COVID-19: U.S. Impact on Antimicrobial Resistance, Special Report 2022*; National Center for Emerging and Zoonotic Infectious Diseases, 2022. https://doi.org/10.15620/cdc:117915.

(3) Li, J. W.-H.; Vederas, J. C. Drug Discovery and Natural Products: End of an Era or an Endless Frontier? *Science* **2009**, *325* (5937), 161–165. https://doi.org/10.1126/science.1168243.

(4) Katz, L.; Baltz, R. H. Natural Product Discovery: Past, Present, and Future. *Journal of Industrial Microbiology and Biotechnology* **2016**, *43* (2–3), 155–176. https://doi.org/10.1007/s10295-015-1723-5.

(5) Patridge, E.; Gareiss, P.; Kinch, M. S.; Hoyer, D. An Analysis of FDA-Approved Drugs: Natural Products and Their Derivatives. *Drug Discovery Today* **2016**, *21* (2), 204–207. https://doi.org/10.1016/j.drudis.2015.01.009.

(6) Qi, S.; Gui, M.; Li, H.; Yu, C.; Li, H.; Zeng, Z.; Sun, P. Secondary Metabolites from Marine Micromonospora: Chemistry and Bioactivities. *Chemistry & Biodiversity* **2020**, *17* (4), e2000024. https://doi.org/10.1002/cbdv.202000024.

(7) Ellis, G. A.; Thomas, C. S.; Chanana, S.; Adnani, N.; Szachowicz, E.; Braun, D. R.; Harper, M. K.; Wyche, T. P.; Bugni, T. S. Brackish Habitat Dictates Cultivable Actinobacterial Diversity from Marine Sponges. *PLOS ONE* **2017**, *12* (7), e0176968. https://doi.org/10.1371/journal.pone.0176968.

(8) Selim, M. S. M.; Abdelhamid, S. A.; Mohamed, S. S. Secondary Metabolites and Biodiversity of Actinomycetes. *J Genet Eng Biotechnol* **2021**, *19*, 72. https://doi.org/10.1186/s43141-021-00156-9.

(9) Subramani, R.; Sipkema, D. Marine Rare Actinomycetes: A Promising Source of Structurally Diverse and Unique Novel Natural Products. *Mar Drugs* **2019**, *17* (5), 249. https://doi.org/10.3390/md17050249.

(10) Hifnawy, M. S.; Fouda, M. M.; Sayed, A. M.; Mohammed, R.; Hassan, H. M.; AbouZid, S. F.; Rateb, M. E.; Keller, A.; Adamek, M.; Ziemert, N.; Abdelmohsen, U. R. The Genus *Micromonospora* as a Model Microorganism for Bioactive Natural Product Discovery. *RSC Adv.* **2020**, *10* (35), 20939–20959. https://doi.org/10.1039/D0RA04025H.

(11) Yan, S.; Zeng, M.; Wang, H.; Zhang, H. Micromonospora: A Prolific Source of Bioactive Secondary Metabolites with Therapeutic Potential. *J. Med. Chem.* **2022**, *65* (13), 8735–8771. https://doi.org/10.1021/acs.jmedchem.2c00626.

(12) Navarro-Muñoz, J. C.; Selem-Mojica, N.; Mullowney, M. W.; Kautsar, S. A.; Tryon, J. H.; Parkinson, E. I.; De Los Santos, E. L. C.; Yeong, M.; Cruz-Morales, P.; Abubucker, S.; Roeters, A.; Lokhorst, W.; Fernandez-Guerra, A.; Cappelini, L. T. D.; Goering, A. W.; Thomson, R. J.; Metcalf, W. W.; Kelleher, N. L.; Barona-Gomez, F.; Medema, M. H. A Computational

Framework to Explore Large-Scale Biosynthetic Diversity. *Nat Chem Biol* **2020**, *16* (1), 60–68. https://doi.org/10.1038/s41589-019-0400-9.

(13) Kautsar, S. A.; van der Hooft, J. J. J.; de Ridder, D.; Medema, M. H. BiG-SLiCE: A Highly Scalable Tool Maps the Diversity of 1.2 Million Biosynthetic Gene Clusters. *GigaScience* **2021**, *10* (1), giaa154. https://doi.org/10.1093/gigascience/giaa154.

(14) Blin, K.; Shaw, S.; Augustijn, H. E.; Reitz, Z. L.; Biermann, F.; Alanjary, M.; Fetter, A.; Terlouw, B. R.; Metcalf, W. W.; Helfrich, E. J. N.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 7.0: New and Improved Predictions for Detection, Regulation, Chemical Structures and Visualisation. *Nucleic Acids Research* **2023**, *51* (W1), W46–W50. https://doi.org/10.1093/nar/gkad344.

(15) Kautsar, S. A.; Blin, K.; Shaw, S.; Weber, T.; Medema, M. H. BiG-FAM: The Biosynthetic Gene Cluster Families Database. *Nucleic Acids Research* **2021**, *49* (D1), D490–D497. https://doi.org/10.1093/nar/gkaa812.

(16) Wyche, T. P.; Hou, Y.; Braun, D.; Cohen, H. C.; Xiong, M. P.; Bugni, T. S. First Natural Analogs of the Cytotoxic Thiodepsipeptide Thiocoraline A from a Marine Verrucosispora Sp. *J. Org. Chem.* **2011**, *76* (16), 6542–6547. https://doi.org/10.1021/jo200661n.

(17) Adnani, N.; Chevrette, M. G.; Adibhatla, S. N.; Zhang, F.; Yu, Q.; Braun, D. R.; Nelson, J.; Simpkins, S. W.; McDonald, B. R.; Myers, C. L.; Piotrowski, J. S.; Thompson, C. J.; Currie, C. R.; Li, L.; Rajski, S. R.; Bugni, T. S. Coculture of Marine Invertebrate-Associated Bacteria and Interdisciplinary Technologies Enable Biosynthesis and Discovery of a New Antibiotic, Keyicin. *ACS Chem. Biol.* **2017**, *12* (12), 3093–3102. https://doi.org/10.1021/acschembio.7b00688.

(18) Koren, S.; Walenz, B. P.; Berlin, K.; Miller, J. R.; Bergman, N. H.; Phillippy, A. M. Canu: Scalable and Accurate Long-Read Assembly via Adaptive k-Mer Weighting and Repeat Separation. *Genome Res.* **2017**, *27* (5), 722–736. https://doi.org/10.1101/gr.215087.116.

(19) *BUSCO Update: Novel and Streamlined Workflows along with Broader and Deeper Phylogenetic Coverage for Scoring of Eukaryotic, Prokaryotic, and Viral Genomes | Molecular Biology and Evolution | Oxford Academic*. https://academic.oup.com/mbe/article/38/10/4647/6329644 (accessed 2022-10-24).

(20) Mikheenko, A.; Prjibelski, A.; Saveliev, V.; Antipov, D.; Gurevich, A. Versatile Genome Assembly Evaluation with QUAST-LG. *Bioinformatics* **2018**, *34* (13), i142–i150. https://doi.org/10.1093/bioinformatics/bty266.

(21) Blin, K.; Shaw, S.; Steinke, K.; Villebro, R.; Ziemert, N.; Lee, S. Y.; Medema, M. H.; Weber, T. antiSMASH 5.0: Updates to the Secondary Metabolite Genome Mining Pipeline. *Nucleic Acids Research* **2019**, *47* (W1), W81–W87. https://doi.org/10.1093/nar/gkz310.

(22) Blin, K.; Shaw, S.; Kloosterman, A. M.; Charlop-Powers, Z.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 6.0: Improving Cluster Detection and Comparison Capabilities. *Nucleic Acids Research* **2021**, *49* (W1), W29–W35. https://doi.org/10.1093/nar/gkab335.

(23) Alanjary, M.; Kronmiller, B.; Adamek, M.; Blin, K.; Weber, T.; Huson, D.; Philmus, B.; Ziemert, N. The Antibiotic Resistant Target Seeker (ARTS), an Exploration Engine for Antibiotic

Cluster Prioritization and Novel Drug Target Discovery. *Nucleic Acids Research* **2017**, *45* (W1), W42–W48. https://doi.org/10.1093/nar/gkx360.

(24) Mungan, M. D.; Alanjary, M.; Blin, K.; Weber, T.; Medema, M. H.; Ziemert, N. ARTS 2.0: Feature Updates and Expansion of the Antibiotic Resistant Target Seeker for Comparative Genome Mining. *Nucleic Acids Research* **2020**, *48* (W1), W546–W552. https://doi.org/10.1093/nar/gkaa374.

(25) Cantrell, C. D. *Modern Mathematical Methods for Physicists and Engineers*; Cambridge University Press, 2000.

(26) Alas, I.; Braun, D. R.; Ericksen, S. S.; Salamzade, R.; Kalan, L.; Rajski, S. R.; Bugni, T. S. Micromonosporaceae Biosynthetic Gene Cluster Diversity Highlights the Need for Broad-Spectrum Investigations. *Microbial Genomics* **2024**, *10* (1), 001167. https://doi.org/10.1099/mgen.0.001167.

(27) Inc, P. T. *Collaborative data science*. https://plot.ly.

(28) Hossain, S. Visualization of Bioinformatics Data with Dash Bio. In *Proceedings of the 18th Python in Science Conference*; Calloway, C., Lippa, D., Niederhut, D., Shupe, D., Eds.; 2019; pp 126–133. https://doi.org/10.25080/Majora-7ddc1dd1-012.

Chapter 4

Predicting Tubulin-Binding Compounds Using MS/MS Structurally Representative Fingerprints with Machine Learning

4.1 INTRODUCTION

Previous chapters have highlighted the importance of mining NPs to mitigate the spread of drug resistant infectious diseases. A historical overview of NPs as sources of new drugs from 1981 to 2019 highlighted a significant portion of new approved drugs as rooted in NPs or derived from NP scaffolds[1]. Even though companies had previously shifted synthetic libraries of smaller and easily modifiable structures, companies and researchers are increasingly invested in utilizing NP privileged scaffolds as a basis for drug discovery platforms to identify compounds with enhanced bioactivity[2–4].

However, as described in Chapter 1, several issues are present in the process to uncover NP drug candidates. Most notably, the issue of "rediscovery", which refers to identifying NPs that have already been explored for clinical usage prior by other researchers[5]. Several decades of NP exploration have resulted in the surface-level NPs, with strong bioactivity and often larger quantities, prioritized for healthcare-related purposes[3]. Effective prioritization of NPs requires sifting through years of research to discard already characterized compounds.

To perform this efficient prioritization of NP, researchers must minimize the time spent working on a potentially already characterized compound and maximize time spent working on novel NPs. Bacterial extract high-throughput screening, where bacterial extracts are screened for bioactivity against a panel of pathogens, then undergo bioactivity-guided fractionation to narrow

down which compound specifically is the root of the activity, and finally run through high-resolution liquid chromatography tandem mass spectrometry (HR-LC-MS/MS) and nuclear magnetic resonance (NMR) to identify the structure of the compound, is a time-intensive process[6]. Once a bacterial extract has been identified as interesting, which is approximately 2-5% of extracts in our lab, the only stage where we have a potential idea of the structure is during HR-LC-MS/MS. As NMR is prohibitively time-intensive, requiring possibly weeks to characterize a structure, leveraging HR-LC-MS/MS to determine if a compound is worth exploring further is a must.

Molecular fingerprints, or the encoded structural characteristics of a molecule to a vector, can be generated through MS/MS datasets through software such as SIRIUS5[7,8]. Within these molecular fingerprints is enough information to make an informed guess as to the full structure of the compound of interest, which means that one can identify pre-existing compounds through molecular fingerprint comparisons to known scaffolds. To perform these similarity comparisons in a systematic fashion, machine learning models which perform pattern recognition are necessary.

As a proof of concept, we wanted to take this idea a step further. By leveraging yeast chemical genomics (YCG) as a repository of compounds that are linked to biological processes, could we utilize purely HR-LC-MS/MS-derived information to correlate structures that impact key biological processes to molecular fingerprints, thus allowing for the de-prioritization of structurally analogous compounds without the need for time-intensive NMR[9,10]. Currently, there are no tools to prediction function based on MS2 spectra. Therefore, this represents a new paradigm for understanding the function of natural products. While this is a simple proof of concept, what we learn here can be extended to other mechanistic spaces.

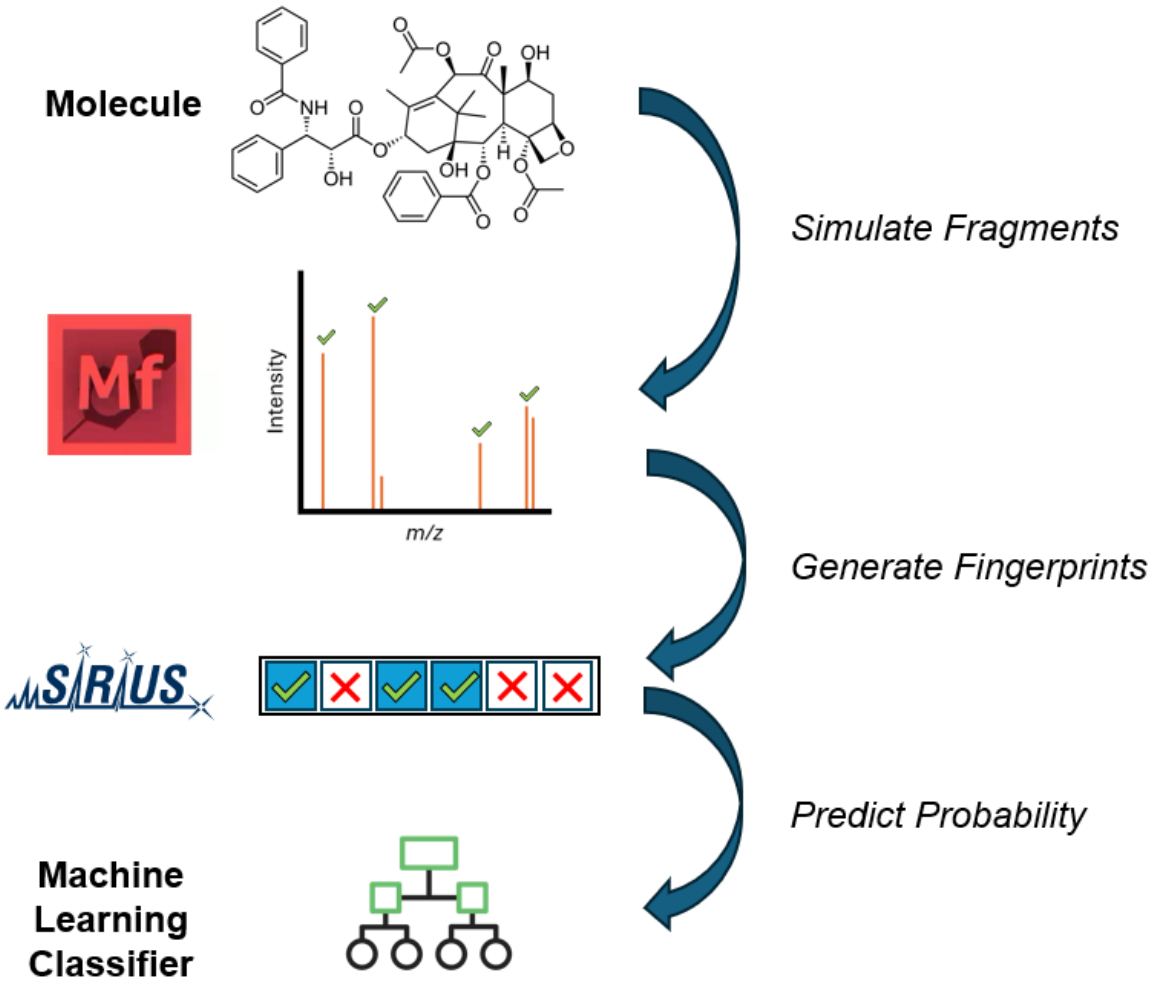


Figure 4.1. Overview of machine learning methodology to predict modulators of tubulin assembly based on structural information. A compound structure undergoes simulated fragmentation. The simulated fragments are reconstructed into a fingerprint. The fingerprint is used to train machine learning classifiers, for the purpose of predicting whether a fingerprint is likely derived from a compound that modulates tubulin assembly. Experimental information from HR-LC-MS/MS is used to generate fingerprints to predict if an unknown compound is a modulator of tubulin assembly or not.

Here, I provide the base of machine learning models to identify modulators of tubulin assembly through purely structural information derived from HR-LC-MS/MS. Our analyses revealed that binary classification models were insufficient using the parameters tested to link impacted biological processes of known structures to molecular fingerprints, but multiclass classification models leveraging structural families could. Most notably, multiclass models resulted in greater

than 96% accuracy on spectra graded A or S rank based on signal-to-noise (S/N) ratio, base peak intensity, and number of peaks observed. This work showcases the merit of leveraging molecular fingerprints from HR-LC-MS/MS as a methodology to link to structural classes, thus allowing for the efficient de-prioritization or prioritization of compounds in those classes with relevant bioactivity.

4.2 RESULTS & DISCUSSION

Additional information on the processes described is given in Section 4.3.

4.2.1 Yeast Chemical Genomics (YCG) as a repository to link bioprocesses with compounds

To link biological processes to compounds in a systematic fashion, we utilized the repository MOSAIC[10,11]. We selected compound classes that were readily known to bind tubulin in literature, for a total of 92 compounds, as previous findings indicated that modulators of mitosis and chromosome segregation were relatively common across NPs in databases such as RIKEN NPDepo[11]. All these compounds were labeled as Positive instances of tubulin modulation, serving as compounds we would leverage to train models to identify core elements potentially related with tubulin modulation. To train against, 10,954 compounds were pulled from RIKEN NPDepo to serve as Negative instances.

4.2.2 Dataset augmentation scheme to account for low Positive instances

Augmentation of the dataset to correct the imbalance was prioritized due to 92 known modulators compared to 10,954 assumed non-interactors of tubulin. Each of the 92 compounds was structure similarity searched in PubChem for similar compounds using a minimum Tanimoto Structural Similarity Score threshold of 96%[13]. For each of the 92 original compounds, up to 20 similar compounds were selected. These additional compounds were not screened and stored in MOSAIC's CG database, but with a minimum threshold of 96%, it was assumed that any deviations in structure would be relatively minimal and not affect any potential pharmacophore. This augmentation changed the dataset into 1143 potential modulators of tubulin assembly versus the 10,954 assumed non-interactors, resulting in 9.4% Positives compared to the original 0.83% Positives.

However, expansion of the Positive instances through exceedingly similar structures meant that the models would be training on highly similar information, which would cause difficulties when determining the models' ability to generalize on structural information (fingerprints) it had never seen before. To minimize this, I constructed 60% training, 20% validation, and 20% testing splits using the original dataset of 92 Positives and 10,954 Negatives, then augmented each of the splits with the highly similar structures present in only those splits. This strategy allowed for the optimization of hyperparameters for binary classification models based on generalization capabilities, rather than their ability to overfit on nigh-identical information.

4.2.3 Hyperparameter optimization of binary classification models through F1 score and Average Precision (AP)

Each model noted in Table 4.1 had several hyperparameter combinations evaluated while training on the 60% training dataset. In addition, some models used scaled fingerprints such as MinMax Scaler and Standard Scaler as inputs to evaluate the effectiveness of scalers on evaluation. Each hyperparameter combination was ranked based on either F1 score (F1) or Average Precision (AP). The best-performing hyperparameter combinations based on either F1 or AP were evaluated against the 20% validation dataset, as seen in Table 4.1. Metrics used for evaluation were F1-score (binary), F1-score (macro), Precision, Recall, and Area Under the Precision-Recall Curve (AUPR). F1-score (binary) is calculated only using the Positive class, whereas F1-score (macro) is calculated across Positive and Negative classes. The five best-performing models were the Multi-layer Perceptron Classifier (refit using F1 or AP), the Light Gradient-Boosting Machine (LightGBM) (refit using F1), and the Support Vector Machine (SVM) (refit using F1 or AP). The best-performing models generally trended towards increased model complexity, possibly hinting

at difficulties teasing apart Negative and Positive instances. Across the board, evaluation metric scores were relatively low.

Models:	Hyper Opt:	F1 (Binary):	F1 (Macro):	Precision:	Recall:	AUPR:
MLPClassifier	F1	0.398	0.683	0.557	0.31	0.457
MLPClassifier	AP	0.395	0.68	0.526	0.316	0.444
LightGBM	F1	0.354	0.661	0.588	0.253	0.446
SVM	AP	0.35	0.659	0.6	0.247	0.449
SVM	F1	0.324	0.643	0.432	0.259	0.37
Linear Regression: No Scaler	None	0.297	0.632	0.561	0.203	0.409
Random Forest	F1	0.287	0.623	0.37	0.234	0.328
Linear Regression: MinMax Scaler	None	0.284	0.626	0.63	0.184	0.434
LightGBM	AP	0.28	0.624	0.667	0.177	0.45
Bernoulli Naïve Bayes	F1	0.251	0.543	0.154	0.671	0.424
Bernoulli Naïve Bayes	AP	0.251	0.543	0.154	0.671	0.424
Multinomial Naïve Bayes	None	0.224	0.517	0.135	0.658	0.408
Complement Naïve Bayes	F1	0.224	0.508	0.133	0.703	0.428
XGBoost	F1	0.209	0.589	0.792	0.12	0.486
XGBoost	AP	0.202	0.585	0.9	0.114	0.537
Decision Tree: No Scaler	F1	0.182	0.565	0.205	0.165	0.213
KNN	F1	0.168	0.561	0.228	0.133	0.21
Decision Tree: Standard	AP	0.162	0.558	0.225	0.127	0.205
Decision Tree: MinMax Scaler	F1	0.159	0.556	0.215	0.127	0.2
Gaussian Naïve Bayes	None	0.158	0.421	0.09	0.639	0.377
Complement Naïve Bayes	AP	0.154	0.25	0.084	1	0.542
KNN	AP	0.131	0.543	0.211	0.095	0.183
Decision Tree: No Scaler	AP	0.11	0.531	0.165	0.082	0.154
Decision Tree: MinMax	AP	0.11	0.531	0.165	0.082	0.154
Random Forest	AP	0.049	0.508	1	0.025	0.545
Linear Regression: Standard Scaler	None	0	0.483	0	0	0.534
Decision Tree: Standard Scaler	F1	0	0.483	0	0	0.534

Table 4.1. Hyperparameter optimization of models trained on 60% of training data evaluated on 20% validation set. Models are sorted in order of highest F1-score (binary). Models with relevant hyperparameters were optimized based on either F1-score (binary) or Average Precision (AP). Evaluation metrics are: F1-score (binary), F1-score (macro), Precision, Recall, and Area Under the Precision-Recall Curve (AUPR). Each evaluation metric column was colored in order of highest to lowest score.

4.2.4 Application of best-performing binary classifiers reveals poor generalizability

Models:	Hyper Opt:	F1-Score (Binary):	F1-Score (Macro):	Precision:	Recall:	AUPRC:
LightGBM	F1	0.467	0.717	0.656	0.362	0.53
SVM	F1	0.449	0.707	0.622	0.351	0.5
SVM	AP	0.398	0.683	0.716	0.276	0.52
MLPClassifier	AP	0.381	0.673	0.615	0.276	0.47
MLPClassifier	F1	0.321	0.64	0.477	0.241	0.38

Table 4.2. Top 5 models evaluated against 20% testing dataset. The top 5 performing models and associated hyperparameters from Table 1 based on F1-score (binary) were selected. Models are sorted in order of highest F1-score (binary). Models were retrained using 60% training and 20% validation splits. Each evaluation metric column was colored in order of highest to lowest score. Each evaluation metric column was colored in order of highest to lowest score.

Evaluation of the five best-performing models on the held-out 20% testing dataset can be seen in Table 4.2. The models were retrained using the 60% training and 20% validation splits. From this, the three best-performing models using the evaluation metrics were trained on the full 60% training, 20% validation, and 20% testing splits, then evaluated on 419 real experimental GNPS MS/MS spectra of compounds known as modulators of tubulin assembly (Table 4.3). These best-performing models were the LightGBM (refit using F1), SVM (refit using F1), and SVM (refit using AP).

Parent Class:	Number of Spectra:	LightGBM (F1):	SVM (F1):	SVM (AP):
Colchicine	140	0	0	1
Taxanes	113	0	0	0
Vinorelbine	73	0	0	0
Vinblastine	33	0	0	0
Vincristine	23	0	2	2
Vinpocetine	19	0	0	0
Vindoline	18	0	0	0

Table 4.3. GNPS spectra evaluation using best-performing models. The parent classes of the compounds from GNPS are seen in the first column. The Positive predictions for the best-performing models are shown in the last three columns. For each of the models, the evaluation metric used to optimize the hyperparameters are shown in parentheses.

When tasked with classifying the GNPS spectra, the models performed poorly. Of the 419 GNPS spectra, only 2-3 were correctly identified as modulators of tubulin assembly. These exclusively fell within spectra associated with Colchicine and Vincristine. Further investigation revealed the other models performed equally as well. One possible rationale was that the models were trained on effectively perfectly fragmented spectra with no noise. In doing so, the models were trained to assess very minute differences in the spectra that were not found in the GNPS spectra, whose quality depends on the contributor's instrument and methodologies. Another rationale was that the Positive instances, despite being augmented to account for class imbalance, are comprised of structurally diverse compounds. A third possible rationale was differences in the datasets being trained on and evaluated on, resulting in the models being optimized to predict on different distributions of fingerprints than they were evaluated on.

4.2.5 Comparisons between simulated and experimental datasets

t-distributed Stochastic Neighbor Embedding (t-SNE) analysis of a constructed aggregate dataset of 17557 fingerprints, derived from simulated data or experimental GNPS spectra, showed distinct separation in distributions based on the origin of the data (Figure 4.2). Within the simulated dataset (Figure 4.2b), I observed higher concentrations of fingerprints that were similar to each other, visually represented as spatially closer through t-SNE. In the experimental spectra (Figure 4.2b), I observed disagreement between the simulated and experimental fingerprints based on dimensionality reduction, where pockets of fingerprints are spatially distinct across fingerprint origins. Notably, I also found reduced concentration of fingerprints for the experimental dataset compared to the simulated dataset, representing increased variation within the experimental dataset.

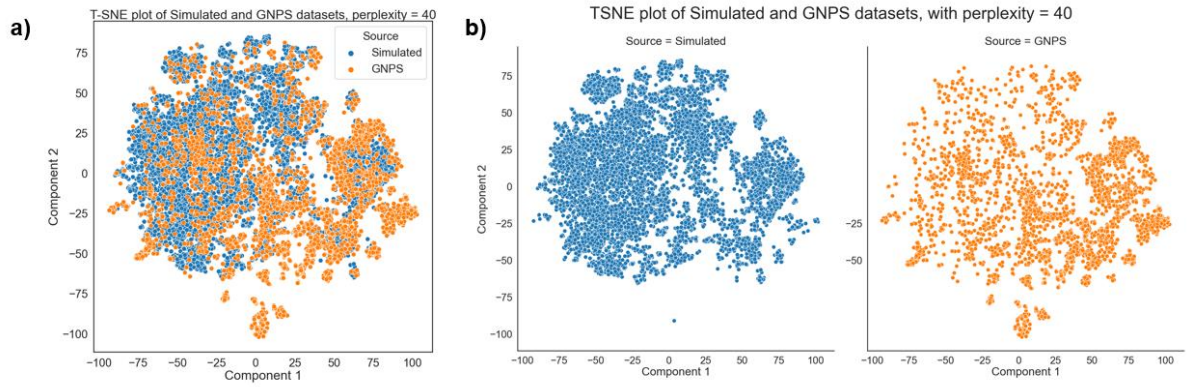


Figure 4.2. t-SNE visualization of simulated and experimental fingerprints. t-SNE was performed on the aggregated dataset of simulated and experimental fingerprints, with the number of components set to 2. **(a)** t-SNE plot shown with a perplexity of 40, each point has been colored based on the source of the fingerprint. **(b)** The t-SNE plot from (a), but each point has been separately plotted based on the source of the fingerprint to highlight differences in distribution.

Further analysis leveraged Principal Component Analysis (PCA) and KMeans Clustering to visualize differences across the simulated and experimental datasets (Figure 4.3). PCA, a linear dimensionality reduction technique that maximizes retained variance within the dataset, was used to show areas of agreement across the simulated and experimental datasets (Figure 4.3a). KMeans Clustering proved more illuminating, breaking down the pockets of fingerprints into distinct clusters (Figure 4.3b). Investigation of the clusters generated by KMeans Clustering (Figure 4.3c) showed two distinct groups as containing primarily fingerprints derived from GNPS experimental spectra (G0, G4), whereas most simulated fingerprints belonged to the same five clusters (G1, G2, G5, G6, G8). This distribution differences across the clusters may be the reason why the binary classification models performed less desirably on when evaluating on the GNPS experimental spectra.

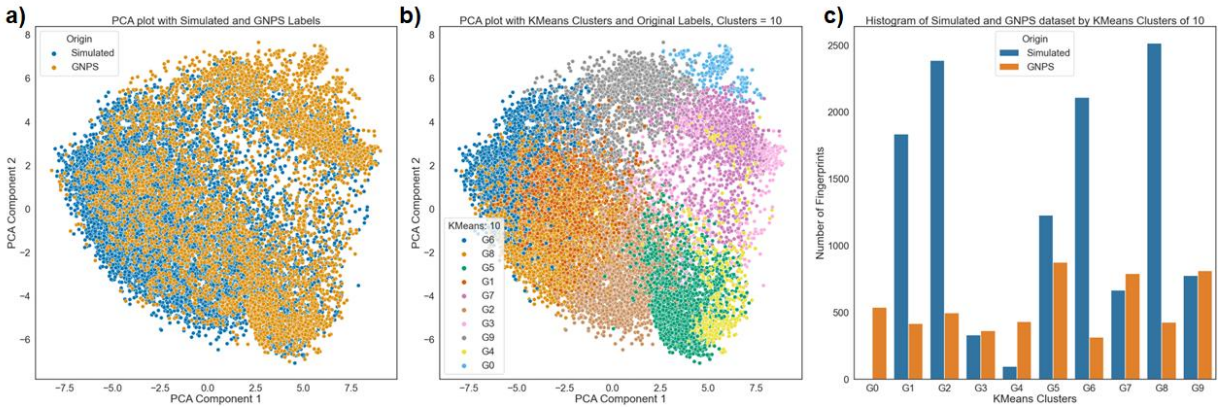


Figure 4.3. PCA and KMeans Clustering exploration of simulated and experimental fingerprints. **(a)** PCA dimensionality reduction on the aggregated dataset of simulated and experimental fingerprints to 2 components. Points are colored based on the origin of the fingerprint (simulated or experimental). **(b)** KMeans Clustering was used on the aggregated dataset, with the number of clusters being defined as 10. Each point was colored based on which cluster it was assigned as. **(c)** Fingerprints belonging to KMeans Clusters seen in (b) were summed per cluster and split by the origin of the fingerprint.

4.2.6 Multiclass classification models show enhanced capabilities

As a subsequent approach, in collaboration with Nathan Brittin, we tried a multiclass classifier. Rather than using a simple Positive/Negative label for predictors of tubulin modulation, 5 representative compound classes were selected as labels to replace the Positive instance, with each representing a unique and disjoint category for labeling. These compound classes can be seen in Table 4.4 and were chosen to provide the models with consistent depictions of specific tubulin modulators, rather than unique one-off structures that reduce the model's ability to infer what a tubulin modulator may look like. This new dataset (Section 4.3.8) constructed by Nathan Brittin was comprised of 1,833 assumed tubulin modulators and 10,954 Negative instances from before.

Class:	Training Spectra:	GNPS Spectra:
Benzimidazoles	35	22
Colchicines	43	144
Podophyllotoxin	176	133
Taxols	1163	113
Vinca Alkaloids	416	166

Table 4.4. Structural class of compounds used for training multiclass and evaluating models. The classes of the compounds used for training multiclass models are comprised of 5 tubulin modulators and one Negative class that contains all other compounds. The total number of spectra used to train the models is 12,792, with 10,959 negative instances.

Multiclass classification models as seen in Table 4.5 underwent hyperparameter optimization on the 80% training splits. The best-performing hyperparameters according to their Weighted F1 score were selected. The optimized models were evaluated on the 20% testing split using the following evaluation metrics: Accuracy, Precision, Recall, F1, and Matthews Correlation Coefficient (MCC). The models were then evaluated against 578 experimental GNPS MS/MS spectra from compounds associated with tubulin modulation (Table 4.6). Metrics were computed based on a model's ability to accurately predict labels, with failure being considered inaccurate labeling. The three best-performing models according to F1 score, which were the K-Nearest Neighbors Classifier, the Support Vector Machine, and the Multi-layer Perceptron Classifier, were compared against a dataset of 5,033 randomly selected experimental GNPS MS/MS spectra comprised of Negatives to calculate False Positive Rate (FPR) (Table 4.7). The FPR is often referred to as the "false alarm rate," applying to instances where a model incorrectly assigns a molecular fingerprint as a modulator of tubulin assembly. Of those three models, the SVM showed the lowest FPR, cementing it as the current best-performing model.

Model	Accuracy	Precision	Recall	F1	MCC
RidgeClassifier	0.996	0.996	0.996	0.996	0.983
LogisticRegression	0.995	0.996	0.995	0.995	0.982
MLPClassifier	0.995	0.995	0.995	0.995	0.982
Perceptron	0.995	0.995	0.995	0.995	0.98
PassiveAggressiveClassifier	0.994	0.994	0.994	0.994	0.975
SVC	0.993	0.993	0.993	0.993	0.974
SGDClassifier	0.992	0.992	0.992	0.991	0.969
KNeighborsClassifier	0.991	0.991	0.991	0.991	0.964

Table 4.5. Best-performing hyperparameters for multiclassification models. Each hyperparameter combination was trained on 80% of the data. Each model's hyperparameter with the best Weighted F1 score was evaluated on the held-out 20% data, using the metrics seen above. Models were sorted by MCC. Each evaluation metric column was colored in order of highest to lowest score.

Model	Accuracy	Precision	Recall	F1	MCC
KNeighborsClassifier	0.488	0.962	0.488	0.623	0.517
RidgeClassifier	0.476	0.968	0.476	0.599	0.507
MLPClassifier	0.446	1	0.446	0.561	0.492
SVC (SVM)	0.448	0.962	0.448	0.572	0.49
LogisticRegression	0.426	1	0.426	0.534	0.479
SGDClassifier	0.41	0.962	0.41	0.523	0.467
PassiveAggressiveClassifier	0.413	0.962	0.413	0.524	0.466
Perceptron	0.391	1	0.391	0.503	0.454

Table 4.6. Evaluation of multiclass models on experimental GNPS MS/MS spectra. ML models seen above were evaluated against 578 GNPS spectra converted to fingerprints. Each model was scored based on: Accuracy, Precision, Recall, F1, MCC. No Negative instances were included in this GNPS MS/MS spectra dataset. Models were sorted by MCC. Each evaluation metric column was colored in order of highest to lowest score.

Model	False Positives	False Positive Rate (FPR)
PassiveAggressiveClassifier	61	1.21
SVC (SVM)	70	1.389
SGDClassifier	76	1.508
KNeighborsClassifier	102	2.023
MLPClassifier	114	2.261
LogisticRegression	116	2.301
Perceptron	128	2.539
RidgeClassifier	634	12.577

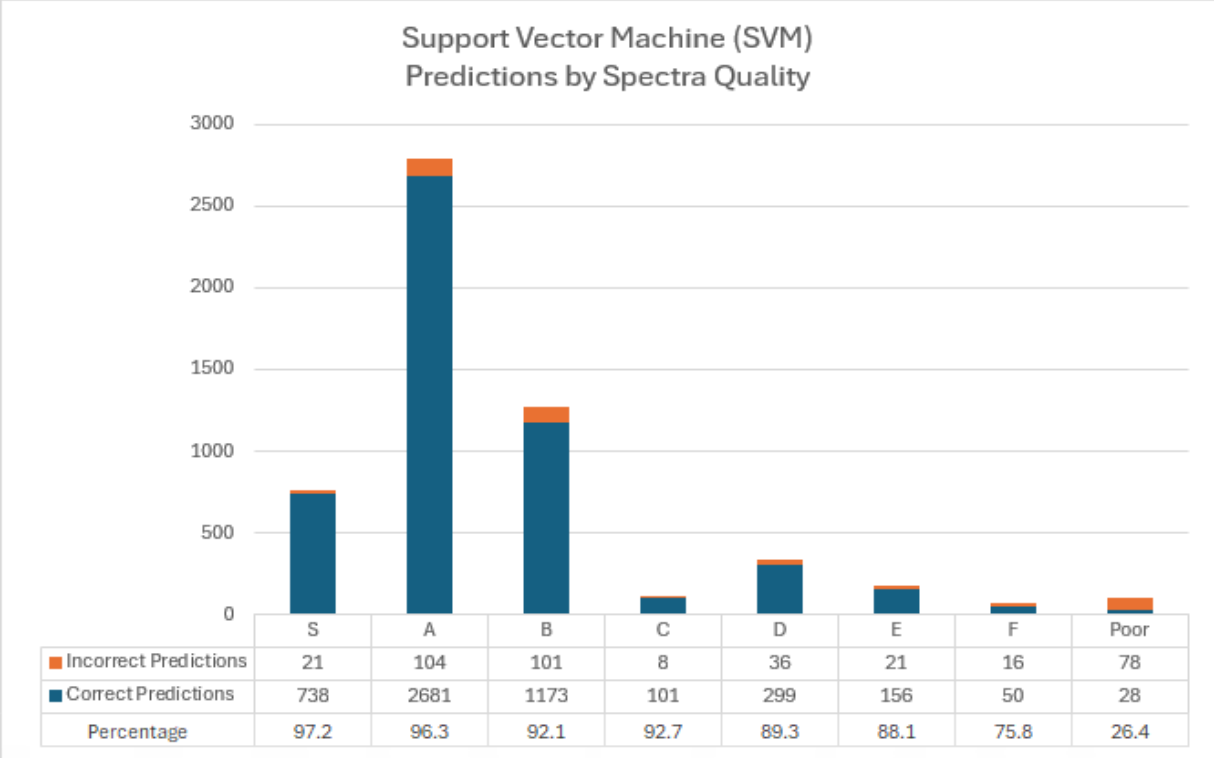
Table 4.7. Evaluation of multiclassification models on experimental GNPS MS/MS spectra. ML models were evaluated against 5033 randomly selected GNPS spectra. False Positive Rate (FPR) was calculated from incorrectly predicted tubulin modulators. The FPs and FPR were ordered from lowest to highest value based on desired outcome.

The experimental GNPS MS/MS dataset of tubulin modulators was combined with the GNPS MS/MS dataset of Negatives, and the SVM model was used to predict whether the fingerprints were associated with modulators of tubulin assembly (Table 4.8). With the SVM, I observed high scores across the board for every evaluation metric, and an associated FPR of 1.4%. Even though other models later scored higher in F1 and other metrics as seen in Table 4.8, no other model scored as consistently well as the SVM model across various datasets.

Model	Accuracy	Precision	Recall	F1	MCC
KNeighborsClassifier	0.929	0.921	0.929	0.92	0.578
SVC (SVM)	0.931	0.918	0.931	0.919	0.579
PassiveAggressiveClassifier	0.929	0.912	0.929	0.914	0.56
MLPClassifier	0.923	0.917	0.923	0.914	0.537
SGDClassifier	0.926	0.917	0.926	0.911	0.542
Perceptron	0.915	0.906	0.915	0.902	0.478
RidgeClassifier	0.834	0.879	0.834	0.848	0.331

Table 4.8. Evaluation of multiclassification models on experimental GNPS MS/MS dataset. Models were evaluated using metrics: Accuracy, Precision, Recall, F1, and MCC. Models were sorted using F1. GNPS MS/MS dataset used contained spectra from Table 6 and 7 combined. SVC (SVM) is bolded to highlight that SVM performed very well on this task, even though the KNeighborsClassifier performed slightly better with regards to F1.

Following the determination of the SVM model as the best-performing model, I looked at the distribution of correct predictions to incorrect predictions as a function of the GNPS MS/MS spectra quality. Spectra quality was graded by the number of peaks observed, the fold difference, and the base peak intensity. In Figure 4.4, we observed a drop in accuracy of the SVM model as spectra quality degrades, with a significant decrease in accuracy associated with the poorest quality spectra. This serves as a possible explanation of poor inference, indicating users should assess their spectra quality before making inferences on possible tubulin interactions.



	S	A	B	C	D	E	F	Poor
Incorrect Predictions	21	104	101	8	36	21	16	78
Correct Predictions	738	2681	1173	101	299	156	50	28
Percentage	97.2	96.3	92.1	92.7	89.3	88.1	75.8	26.4

Figure 4.4. Accuracy of multiclass SVM model decreases based on quality of spectra. Stacked vertical bar plot of SVM model predictions on experimental GNPS MS/MS spectra, across spectra quality grades. In blue is the correct predictions, and in orange is the incorrect predictions.

4.3 MATERIALS & METHODS

Some of the considerations described in Section 4.1 are elaborated on here.

4.3.1 Binary labeling scheme of dataset using CG-Target scores

11419 structures from the RIKEN NPDepo dataset within MOSAIC were selected to serve as the base for our initial dataset[10,12]. MOSAIC compounds within the RIKEN NPDepo with a CG-Target score of less than 0.5 for both *cin1* and *tub3* were labeled as Negative instances. Compounds with a CG-Target score of greater than or equal to 6.0 for either *cin1* or *tub3*, defined as high interaction scores, were labeled as Positive instances. Compounds within the intermediate range were considered edge cases and discarded. The SMILES structures were pulled from MOSAIC, resulting in 92 Positive instances and 10954 Negative instances. These processes were done by Nathan Brittin.

4.3.2 Augmentation of Positive instances using structurally similar compounds

For the 92 compounds from MOSAIC labeled as Positive instances, each structure was structure similarity searched in PubChem for compounds using a minimum Tanimoto Structural Similarity Score threshold of minimum 96%[13]. The SMILES structures of up to 20 similar compounds per original compound were selected, with each one being considered an additional Positive instance for labeling. Each SMILES representation was processed using the RDKit cheminformatics package to strip counterions, canonicalize, and deduplicate. This expanded the Positive label group from 92 to 1143 unique compound structures. These processes were done by Nathan Brittin.

4.3.3 Fingerprint generation through SIRIUS

Mass Frontier 8.1 v8.1.80.8 (Thermo Fisher Scientific, Waltham, Massachusetts) was used to generate realistic compound fragments that would be expected in mass spectrometry with electrospray ionization (ESI) and collision induced dissociation (CID). The fragments were then compiled into an open-source mass spectrometry data format (.mgf) to simulate MS/MS spectra. Following, SIRIUS5 was used to interpret the *in-silico* MS/MS spectra and generate molecular fingerprints, termed MS2FP, of a fixed-length 3878[14–16]. The MS2FP for each compound was then extracted and compiled from SIRIUS5 using Python into a matrix for training. These processes were done by Nathan Brittin.

4.3.4 Train-test-validation split for binary classification

The MS2FP dataset, comprising 92 Positive and 10954 Negative, was split into stratified 60% training, 20% validation, and 20% testing datasets. Each of the training, validation, and testing datasets were augmented with the additional Positive instances from Section 4.3.2 to minimize information bleed-through across splits.

4.3.5 Hyperparameter optimization of binary classification models

The models noted in Table 4.1 were trained using the 60% training dataset, and a selection of manually curated hyperparameters optimized using the GridSearchCV function in the SciKit Learn package[17]. The best hyperparameter combinations were determined based on either F1 (binary) or Average Precision (AP), as noted in Table 4.1. The models were scored against the 20% validation dataset using the evaluation metrics: F1 (binary), F1 (macro), Precision, Recall, Area Under the Precision-Recall Curve (AUPR). The results are seen in Table 4.1.

4.3.6 Identifying the best performing binary classification model

The top five best-performing models were retrained on the 60% training and 20% validation splits and evaluated against the 20% testing dataset (Table 4.2). The best-performing model was retrained using the entirety of the MOSAIC dataset (Section 4.3.1) and the augmented dataset (Section 4.3.2), then evaluated against a set of 419 GNPS spectra. The model's results are seen in Table 4.3.

4.3.7 Compiling and generating MS2FP for experimental GNPS spectra

Compounds from RIKEN NPDepo within the initial *in-silico* generated Positive training set of 92 compounds were name searched within the GNPS MS/MS library for experimentally collected MS/MS spectra. All matching experimental spectra were compiled in python as open source .mgf files and processed using SIRIUS5 with the same parameters as the Positive training examples. The MS2FP were extracted and compiled using python into a matrix for model evaluation on experimentally collected spectra. These processes were done by Nathan Brittin.

4.3.8 Comparing simulated and experimental spectra datasets

The MS2FP dataset comprised of 1143 Positive instances and 10954 Negative instances were compared against the experimental GNPS dataset of 5460 GNPS spectra (419 Positives, 5041 Negatives). T-distributed Stochastic Neighbor Embedding (t-SNE) was used as an unsupervised nonlinear dimensionality reduction technique on the full dataset of 17557 spectra, with a perplexity of 40 (Figure 4.2). Points were colored based on whether the fingerprints were simulated using the methods from Sections 4.3.1-4.3.3 or derived from experimental GNPS spectra using methods from Section 4.3.7. The visualization was separated based on these colors to highlight regions of overlap and separation.

The simulated and experimental spectra datasets were compared through Principal Component Analysis (PCA) and KMeans Clustering. PCA, a linear dimensionality reduction technique, was utilized on the fingerprints to reduce the dimensions to 2 by maximizing the variance present within the dataset to 2 components. KMeans Clustering was run in parallel on the fingerprints. Comparison between these methods is shown in Figure 4.3a and Figure 4.3b. KMeans Clustering information was further investigated by generating a histogram that compared the number of fingerprints in each cluster by whether those fingerprints came from the simulated dataset or the experimental GNPS dataset (Figure 4.3c).

4.3.9 Multiclass dataset labeling using representative tubulin active structural families

The 92 compounds labeled as Positive in Section 4.3.1 were grouped using hierarchical agglomerative clustering to identify 47 representative structural clusters by Spencer Ericksen. The 4 clusters that contained more than 3 compounds minimum were selected as representative tubulin binders. The 5 structural classes contained within the 4 largest clusters were selected as the base for the multiclass labeled dataset. Additional compounds were incorporated using the scheme described in Section 4.3.2, resulting in a total of 1833 compounds. Each Positive compound was labeled with the class of compound. The classes used are depicted in Table 4.4, along with the number of fingerprints used for training the multiclass models. 10,959 compounds termed previously as Negative in Section 4.3.1 were again labeled Negative. Fingerprints were generated using the same methods as in Section 4.3.3. These processes were done by Nathan Brittin.

4.3.10 Train-test split for multiclass classification

The MS2FP dataset seen in Section 4.3.9 was split into stratified 80% training and 20% testing datasets. The labels were encoded into numbers representing each tubulin modulating class or Negative instance.

4.3.11 Hyperparameter optimization of multiclass models

The models noted in Table 4.5 were trained using the 80% training dataset in Section 4.3.10, and a selection of manually curated hyperparameters were searched using GridSearchCV (Appendix Table B.1). Hyperparameters were scored based on Macro F1 and Weighted F1 scores within the 80% training dataset, and the best-performing hyperparameters according to Weighted F1 were selected (Appendix Table B.2). The models were scored against the 20% testing dataset from Section 4.3.10, and evaluated in Table 4.5 based on the following metrics: Accuracy, Precision, Recall, F1, and Matthews Correlation Coefficient (MCC).

4.3.12 Evaluation of multiclass models on experimental GNPS MS/MS spectra

Models selected with best-performing hyperparameters were evaluated against 578 GNPS MS/MS spectra (Table 4.4) from compounds associated with tubulin modulation as per Section 4.3.7. As seen in Table 4.6, models were evaluated based on: Accuracy, Precision, Recall, F1, and MCC. The 578 experimental GNPS MS/MS categories for structural class are depicted in Table 4.4. The top three best-performing models by F1 were selected for determination of False Positive Rate (FPR).

4.3.13 Determination of False Positive Rate (FPR) for multiclass models

Models were evaluated against 5,033 randomly selected experimental GNPS MS/MS spectra taken from compounds not associated with tubulin modulation. False Positive Rate (FPR) was calculated from determining the number of False Positives predicted by each model in comparison to the total number of spectra evaluated (Table 4.7). The top model was selected by the lowest FPR.

4.3.14 Evaluation of best-performing multiclass model through spectra grade scoring

The best-performing multiclass model according to Sections 4.3.11-4.3.13 was evaluated across the combined experimental GNPS MS/MS spectra dataset from Section 4.3.12 and the experimental GNPS MS/MS spectra dataset from Section 4.3.13 (Table 4.8). Evaluation metrics included the same five metrics: Accuracy, Precision, Recall, F1, and MCC. Spectra quality was assessed using number of peaks, fold difference, and base peak intensity, and separated into 8 grades (S, A, B, C, D, E, F, Poor), with S being the highest quality spectra and Poor being the lowest quality spectra (Figure 4.4). The rules used to assess spectra quality can be seen in appendix Code B.5 and were developed by Nathan Brittin.

4.4 REFERENCES

(1) Newman, D. J.; Cragg, G. M. Natural Products as Sources of New Drugs over the Nearly Four Decades from 01/1981 to 09/2019. *J. Nat. Prod.* **2020**, *83* (3), 770–803. https://doi.org/10.1021/acs.jnatprod.9b01285.

(2) Davison, E. K.; Brimble, M. A. Natural Product Derived Privileged Scaffolds in Drug Discovery. *Current Opinion in Chemical Biology* **2019**, *52*, 1–8. https://doi.org/10.1016/j.cbpa.2018.12.007.

(3) Li, J. W.-H.; Vederas, J. C. Drug Discovery and Natural Products: End of an Era or an Endless Frontier? *Science* **2009**, *325* (5937), 161–165. https://doi.org/10.1126/science.1168243.

(4) Newman, D. J. Natural Products as Leads to Potential Drugs: An Old Process or the New Hope for Drug Discovery? *J. Med. Chem.* **2008**, *51* (9), 2589–2599. https://doi.org/10.1021/jm0704090.

(5) Atanasov, A. G.; Zotchev, S. B.; Dirsch, V. M.; Supuran, C. T. Natural Products in Drug Discovery: Advances and Opportunities. *Nat Rev Drug Discov* **2021**, *20* (3), 200–216. https://doi.org/10.1038/s41573-020-00114-z.

(6) Bugni, T. S.; Richards, B.; Bhoite, L.; Cimbora, D.; Harper, M. K.; Ireland, C. M. Marine Natural Product Libraries for High-Throughput Screening and Rapid Drug Discovery. *J. Nat. Prod.* **2008**, *71* (6), 1095–1098. https://doi.org/10.1021/np800184g.

(7) Capecchi, A.; Probst, D.; Reymond, J.-L. One Molecular Fingerprint to Rule Them All: Drugs, Biomolecules, and the Metabolome. *Journal of Cheminformatics* **2020**, *12* (1), 43. https://doi.org/10.1186/s13321-020-00445-4.

(8) Dührkop, K.; Fleischauer, M.; Ludwig, M.; Aksenov, A. A.; Melnik, A. V.; Meusel, M.; Dorrestein, P. C.; Rousu, J.; Böcker, S. SIRIUS 4: A Rapid Tool for Turning Tandem Mass Spectra into Metabolite Structure Information. *Nat Methods* **2019**, *16* (4), 299–302. https://doi.org/10.1038/s41592-019-0344-8.

(9) Enserink, J. M. Chemical Genetics: Budding Yeast as a Platform for Drug Discovery and Mapping of Genetic Pathways. *Molecules* **2012**, *17* (8), 9258–9273. https://doi.org/10.3390/molecules17089258.

(10) Nelson, J.; Simpkins, S. W.; Safizadeh, H.; Li, S. C.; Piotrowski, J. S.; Hirano, H.; Yashiroda, Y.; Osada, H.; Yoshida, M.; Boone, C.; Myers, C. L. MOSAIC: A Chemical-Genetic Interaction Data Repository and Web Resource for Exploring Chemical Modes of Action. *Bioinformatics* **2018**, *34* (7), 1251–1252. https://doi.org/10.1093/bioinformatics/btx732.

(11) Piotrowski, J. S.; Li, S. C.; Deshpande, R.; Simpkins, S. W.; Nelson, J.; Yashiroda, Y.; Barber, J. M.; Safizadeh, H.; Wilson, E.; Okada, H.; Gebre, A. A.; Kubo, K.; Torres, N. P.; LeBlanc, M. A.; Andrusiak, K.; Okamoto, R.; Yoshimura, M.; DeRango-Adem, E.; van Leeuwen, J.; Shirahige, K.; Baryshnikova, A.; Brown, G. W.; Hirano, H.; Costanzo, M.;

Andrews, B.; Ohya, Y.; Osada, H.; Yoshida, M.; Myers, C. L.; Boone, C. Functional Annotation of Chemical Libraries across Diverse Biological Processes. *Nat Chem Biol* **2017**, *13* (9), 982–993. https://doi.org/10.1038/nchembio.2436.

(12) OSADA, H. Introduction of New Tools for Chemical Biology Research on Microbial Metabolites. *Bioscience, Biotechnology, and Biochemistry* **2010**, *74* (6), 1135–1140. https://doi.org/10.1271/bbb.100061.

(13) Kim, S.; Chen, J.; Cheng, T.; Gindulyte, A.; He, J.; He, S.; Li, Q.; Shoemaker, B. A.; Thiessen, P. A.; Yu, B.; Zaslavsky, L.; Zhang, J.; Bolton, E. E. PubChem 2023 Update. *Nucleic Acids Research* **2023**, *51* (D1), D1373–D1380. https://doi.org/10.1093/nar/gkac956.

(14) *Research in Computational Molecular Biology: 19th Annual International Conference, RECOMB 2015, Warsaw, Poland, April 12-15, 2015, Proceedings*; Przytycka, T. M., Ed.; Lecture Notes in Computer Science; Springer International Publishing: Cham, 2015; Vol. 9029. https://doi.org/10.1007/978-3-319-16706-0.

(15) Hoffmann, M. A.; Nothias, L.-F.; Ludwig, M.; Fleischauer, M.; Gentry, E. C.; Witting, M.; Dorrestein, P. C.; Dührkop, K.; Böcker, S. Assigning Confidence to Structural Annotations from Mass Spectra with COSMIC. bioRxiv March 19, 2021, p 2021.03.18.435634. https://doi.org/10.1101/2021.03.18.435634.

(16) Dührkop, K.; Nothias, L.-F.; Fleischauer, M.; Reher, R.; Ludwig, M.; Hoffmann, M. A.; Petras, D.; Gerwick, W. H.; Rousu, J.; Dorrestein, P. C.; Böcker, S. Systematic Classification of Unknown Metabolites Using High-Resolution Fragmentation Mass Spectra. *Nat Biotechnol* **2021**, *39* (4), 462–471. https://doi.org/10.1038/s41587-020-0740-8.

(17) Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, É. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research* **2011**, *12* (85), 2825–2830.

Chapter 5

Conclusions and Future Work

5.1 CONCLUDING REMARKS

The persistent development of antibacterial resistance has resulted in an immense burden for healthcare systems worldwide[1]. While the financial burden is upwards of 20 billion dollars annually, an associated compromised immune system can develop into complications in patients leveraging modern advances in healthcare such as cancer therapy, organ transplants, and more[2]. As described in the previous chapters, natural products serve as the ideological backbone for many antimicrobial compounds developed to this day, and novel natural products are leveraged to mitigate the spread of antimicrobial resistance in healthcare environments. However, the process to discover these compounds suffers from many problems, chief among them being rediscovery. To identify novel natural products, computational approaches for leverage an ever-increasing amount of public genomic and metabolomic information are necessary.

Within that lens, Chapter 2 and 3 highlight the power of leveraging computational tools to prioritize and establish bacterial families of interest in under-explored environments. Complementary, Chapter 4 describes a computational methodology to enhance novel antibacterial discovery by linking structural information to potential impacted biological processes. These methodologies and tools, while rudimentary, allow for researchers to approach natural product drug discovery holistically, serving as pieces of larger potential workflows to combat antibacterial resistance.

5.2 FUTURE DIRECTIONS

The computational approaches described within this thesis focus on genomic and metabolomic information to improve natural product drug discovery. Therefore, any strides directly building off this material require the incorporation of more comprehensive approaches that leverage further meaningful information. Within Chapter 2, the work presented proposes the further investigation of the family *Micromonosporaceae*, specifically from under-exploited environments such as marine sources, based on the biosynthetic potential present in an untargeted collection of *Micromonosporaceae*. The concept of a BGC continues to be developed, with antiSMASH serving as a resource for the manual detection of regions associated with BGCs[3]. However, other software such as deepBGC leverages higher-order information like positions of distant entities as sources for machine learning models, directly incorporating advances natural language processing[4]. In doing so, they pave the road for further inclusion of machine learning methodologies to identify BGCs and more based on the inhuman eye[5].

Chapter 3 described the Biosynthetic Gene Cluster Prioritization Dashboard, a tool to identify interesting BGCs within datasets based on novelty to existing public databases. Additional steps to improve the BGCPD involve leveraging analytical chemistry techniques such as LC-MS/MS to link observed BGCs to observed metabolites in bacterial extracts. NPLinker and NPOmix serve as modern frameworks to perform that linkage, connecting biosynthetic potential to realized output[6,7]. Direct inclusion and incorporation of these tools within the BGCPD would allow for true prioritization of novel BGCs filtered through produced metabolites.

Chapter 4 leverages structural information through molecular fingerprints to infer impacted biological processes, primarily serving to assist in mitigating rediscovery of natural products relevant for antibacterial resistance. The current model focuses on a multiclass classifier to predict

if a biological process is impacted. Given that this procedure worked, utilizing deep neural network multiclass classifiers to link a system of biological processes to a given compound may be possible[8]. However, investigating the binary classifier to determine if training the binary classifier using experimental spectra rather than simulated fingerprints derived from structural information results in comparable evaluation metrics is still a possibility. Applying that same approach to the multiclass classifier to improve the effectiveness of our multiclass classifier may also be fruitful. Within multiclass classifiers, leveraging modern advances in natural language processing with rediscovery pipelines to generate more sophisticated models may result in more novel compounds to fight antibiotic resistance[9–11].

5.3 REFERENCES

(1) Centers for Disease Control and Prevention (U.S.). *Antibiotic Resistance Threats in the United States, 2019*; Centers for Disease Control and Prevention (U.S.), 2019. https://doi.org/10.15620/cdc:82532.

(2) Dadgostar, P. Antimicrobial Resistance: Implications and Costs. *Infection and Drug Resistance* **2019**, *12*, 3903. https://doi.org/10.2147/IDR.S234610.

(3) Blin, K.; Shaw, S.; Augustijn, H. E.; Reitz, Z. L.; Biermann, F.; Alanjary, M.; Fetter, A.; Terlouw, B. R.; Metcalf, W. W.; Helfrich, E. J. N.; van Wezel, G. P.; Medema, M. H.; Weber, T. antiSMASH 7.0: New and Improved Predictions for Detection, Regulation, Chemical Structures and Visualisation. *Nucleic Acids Research* **2023**, *51* (W1), W46–W50. https://doi.org/10.1093/nar/gkad344.

(4) Hannigan, G. D.; Prihoda, D.; Palicka, A.; Soukup, J.; Klempir, O.; Rampula, L.; Durcak, J.; Wurst, M.; Kotowski, J.; Chang, D.; Wang, R.; Piizzi, G.; Temesi, G.; Hazuda, D. J.; Woelk, C. H.; Bitton, D. A. A Deep Learning Genome-Mining Strategy for Biosynthetic Gene Cluster Prediction. *Nucleic Acids Research* **2019**, *47* (18), e110. https://doi.org/10.1093/nar/gkz654.

(5) Walker, A. S.; Clardy, J. A Machine Learning Bioinformatics Method to Predict Biological Activity from Biosynthetic Gene Clusters. *J Chem Inf Model* **2021**, *61* (6), 2560–2571. https://doi.org/10.1021/acs.jcim.0c01304.

(6) Eldjárn, G. H.; Ramsay, A.; Hooft, J. J. J. van der; Duncan, K. R.; Soldatou, S.; Rousu, J.; Daly, R.; Wandy, J.; Rogers, S. Ranking Microbial Metabolomic and Genomic Links in the NPLinker Framework Using Complementary Scoring Functions. *PLOS Computational Biology* **2021**, *17* (5), e1008920. https://doi.org/10.1371/journal.pcbi.1008920.

(7) Leão, T. F.; Wang, M.; da Silva, R.; Gurevich, A.; Bauermeister, A.; Gomes, P. W. P.; Brejnrod, A.; Glukhov, E.; Aron, A. T.; Louwen, J. J. R.; Kim, H. W.; Reher, R.; Fiore, M. F.; van der Hooft, J. J. J.; Gerwick, L.; Gerwick, W. H.; Bandeira, N.; Dorrestein, P. C. NPOmix: A Machine Learning Classifier to Connect Mass Spectrometry Fragmentation Data to Biosynthetic Gene Clusters. *PNAS Nexus* **2022**, *1* (5), pgac257. https://doi.org/10.1093/pnasnexus/pgac257.

(8) Rácz, A.; Bajusz, D.; Héberger, K. Multi-Level Comparison of Machine Learning Classifiers and Their Performance Metrics. *Molecules* **2019**, *24* (15), 2811. https://doi.org/10.3390/molecules24152811.

(9) Saldívar-González, F. I.; Aldas-Bulos, V. D.; Medina-Franco, J. L.; Plisson, F. Natural Product Drug Discovery in the Artificial Intelligence Era. *Chem. Sci.* **2022**, *13* (6), 1526–1546. https://doi.org/10.1039/D1SC04471K.

(10) Mullowney, M. W.; Duncan, K. R.; Elsayed, S. S.; Garg, N.; van der Hooft, J. J. J.; Martin, N. I.; Meijer, D.; Terlouw, B. R.; Biermann, F.; Blin, K.; Durairaj, J.; Gorostiola González, M.; Helfrich, E. J. N.; Huber, F.; Leopold-Messer, S.; Rajan, K.; de Rond, T.; van Santen, J. A.; Sorokina, M.; Balunas, M. J.; Beniddir, M. A.; van Bergeijk, D. A.; Carroll, L. M.; Clark, C. M.; Clevert, D.-A.; Dejong, C. A.; Du, C.; Ferrinho, S.; Grisoni, F.; Hofstetter, A.; Jespers, W.; Kalinina, O. V.; Kautsar, S. A.; Kim, H.; Leao, T. F.; Masschelein, J.; Rees, E. R.; Reher, R.; Reker, D.; Schwaller, P.; Segler, M.; Skinnider, M. A.; Walker, A. S.; Willighagen, E.

L.; Zdrazil, B.; Ziemert, N.; Goss, R. J. M.; Guyomard, P.; Volkamer, A.; Gerwick, W. H.; Kim, H. U.; Müller, R.; van Wezel, G. P.; van Westen, G. J. P.; Hirsch, A. K. H.; Linington, R. G.; Robinson, S. L.; Medema, M. H. Artificial Intelligence for Natural Product Drug Discovery. *Nat Rev Drug Discov* **2023**, *22* (11), 895–916. https://doi.org/10.1038/s41573-023-00774-7.

(11) Gupta, R.; Srivastava, D.; Sahu, M.; Tiwari, S.; Ambasta, R. K.; Kumar, P. Artificial Intelligence to Deep Learning: Machine Intelligence Approach for Drug Discovery. *Mol Divers* **2021**, *25* (3), 1315–1360. https://doi.org/10.1007/s11030-021-10217-3.

Appendix A

Supplementary data for Chapter 3

Code A.1 Code to generate BGC Prioritization Dashboard.....**92**

Code A.1: Code to generate BGC Prioritization Dashboard

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 18 21:18:42 2023

@author: imraan alas
"""

# The webpage will be at localhost:8002, or 127.0.0.1:8002

#%% Imports

# conda environment: python-class

# AntiSMASH relevant imports
import os
from bs4 import BeautifulSoup # beautiful soup version 4.10.0
from copy import deepcopy
from os import chdir, getcwd
import re
import sys

import pandas as pd # pandas version 1.3.5
import numpy as np # numpy version 1.21.6
import matplotlib.pyplot as plt
import seaborn as sns

# BiG-SLiCE relevant imports
import sqlite3 # sqlite version 3.38.5
import pandas as pd # pandas version 1.3.5
import numpy as np # numpy version 1.21.6
# import sqlalchemy
import matplotlib.pyplot as plt # matplotlib version 3.5.1
# from matplotlib import rcParams
import math
import seaborn as sns # seaborn version 0.11.2
# import networkx as nx
import sklearn # scikit-learn version 1.0.2
# from sklearn import metrics
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
# import graphviz
# import pygraphviz
from sklearn.manifold import TSNE
```

```
from sklearn.manifold import MDS
import scipy # scipy version 1.7.3
import plotly.express as px # plotly express version 0.4.1
import plotly.io as pio # plotly version 5.10.0
# from sklearn.cluster import KMeans, AffinityPropagation, AgglomerativeClustering, Birch, DBSCAN,
MeanShift, OPTICS # k-means clustering
# from sklearn.manifold import TSNE # t-SNE clustering
# from sklearn.model_selection import train_test_split # training/testing splitter
from scipy.cluster.hierarchy import linkage, dendrogram # Hierarchical Clustering
from numpy import unique, where
# from decimal import Decimal
# from pandas import ExcelWriter
# from sklearn import tree
# from mpl_toolkits import mplot3d
from statannotations.Annotator import Annotator
import copy
import plotly.express as px
import plotly.io as pio

# Dash
import dash
from dash import dcc, html, Input, Output, callback

import time
import pickle

#%% Existing Code

startTime = time.time();

# AntiSMASH folders/files:
antiSMASHdirectory = input('What is the filepath to the folder containing the AntiSMASH data? If nothing
is typed, will assume default.\nDirectory: ')
if os.path.exists(antiSMASHdirectory):
    print('AntiSMASH version 5.1.1 directory: ' + antiSMASHdirectory)
else:
    antiSMASHdirectory = r"C:/Users/imraa/Documents/UWM/BugniLab/Genome Assembly/micromonosporaceae-
AntiSmash511-Clean";
    print('Could not find antiSMASH directory, using default directory:\n' + antiSMASHdirectory)

# BiG-SLiCE folders/files:
pathToDataDB = input('Filepath to the data.db for the already run BiG-SLiCE data?\nFilepath: ')
if os.path.exists(pathToDataDB):
    print('The path to the data.db file in the result folder: ' + pathToDataDB)
else:
```

```
    pathToDataDB = "C:/Users/imraa/Downloads/DataS10-bigsliceOutputs-antiSMASH-v511/DataS10-bigsliceOutputs-antiSMASH-v511/full_run_result/result/data.db"; # This is the path to the data.db file, once copied from Linux and put into Windows.
    print('Couldn\'t find the data.db file, here\'s an example location: ' + pathToDataDB)

pathToReportsDB = input('Filepath to the reports folder for the already run BiG-SLiCE data?\nFilepath: ')
if os.path.exists(pathToReportsDB):
    print('The path to the reports folder:\n' + pathToReportsDB)
else:
    pathToReportsDB = "C:/Users/imraa/Downloads/DataS10-bigsliceOutputs-antiSMASH-v511/DataS10-bigsliceOutputs-antiSMASH-v511/full_run_result/result/data.db"; # This is the path to the data.db file, once copied from Linux and put into Windows.
    print('Couldn\'t find the reports folder, here\s an example location: ' + pathToReportsDB)

run_id = int(input('Choose a run_id that was used for querying your BGCs against BiG-SLiCE\'s pre-processed database.\n1 is using no Threshold.\n2 is using Threshold 300.\n'+
               '4 is using Threshold 600.\n6 is using Threshold 900.\n7 is using Threshold 1200.\n8 is using Threshold 1500.\nEnter run_id here: '))
if run_id not in [1,2,4,6,7,8]:
    print('ERROR: Please select a proper run_id, with numbers possible either 1, 2, 4, 6, 7, 8')
    sys.exit()
else:
    runThresholdPair = {1: 0, 2: 300, 4: 600, 6: 900, 7: 1200, 8: 1500};
    thresholdValue = runThresholdPair[run_id];
    print('You have selected run_id ' + str(run_id) + ' with associated threshold value of ' + str(thresholdValue))

reportsRuns = input('Filepath to a text file where each line contains a number that represents the BiG-SLiCE runs used for this analysis.\nFor example, if you wanted to run reports 100-102, the text file would contain:\n100\n101\n102\nFilepath: ')
if os.path.exists(reportsRuns):
    # Read the folder list from the file
    with open('reportsRun', 'r') as file:
        # Assuming each line contains a folder number
        allFolders = [int(line.strip()) for line in file]
        print('The report folders used for this analysis are: ')
        print(allFolders)
else:
    print('ERROR: Couldn\'t find file containing the BiG-SLiCE runs intended to be analyzed...')
    print('Reminder, this should be a simple text file where each line is a number that represents the run you wish to include in the analysis')
    # Default case (for my own files):
    allFolders = np.concatenate([np.arange(111,145,1), np.arange(149,157,1)]); # antismash v5.1.1: Right now, it looks at reports folders 111 to 144, and 150 to 156. The numbers can be found in the BiG-SLiCE visualization.
```

```
    print('(DEFAULT): The report folders used for this analysis are: ')
    print(allFolders)

# A little bit of pre-processing for my own data.
if antiSMASHdirectory == r"C:/Users/imraa/Documents/UWM/BugniLab/Genome Assembly/micromonosporaceae-
AntiSmash511-Clean":
    strainsToRemoveB = "a1363|b482|b486"; # I remove these strains in the BiG-SLICE analysis because they
failed QC.
    print('Strains to remove from the BiG-SLiCE visualizations: ' + strainsToRemoveB)

# Determine script output folder:
scriptOutputs = input('What directory should files generated by this analysis be stored?\nDirectory:')
if os.path.exists(scriptOutputs):
    print('Relevant outputs will be stored in: ' + scriptOutputs)
else:
    scriptOutputs = 'C:/Users/imraa/Documents/UWM/BugniLab/Genome Assembly/scriptOutputs'
    print('The directory does not appear to exist. Here\'s an example one:\n' + scriptOutputs)

# Check for already run files.
print('If you\'ve run this analysis already, this section double-checks to see if there\'s any stored
files it can use.')
# Path to csv file containing BGCs x GCFs (BiG-SLiCE): Used to minimize resource usage after running
Section 5 once.
if os.path.isfile(scriptOutputs + '/' + 'antismash5-strainsQueried-rank.csv'):
    queriedBGC_GCFpath = scriptOutputs + '/' + 'antismash5-strainsQueried-rank.csv';
    print('Found csv file containing BGCs x GCFs, here:\n' + queriedBGC_GCFpath)
else:
    print('No previously analyzed data regarding BiG-SLiCE queries run through this program already
exists, will run and store in:\n' + queriedBGC_GCFpath)

if os.path.isfile(scriptOutputs + '/' + 'mds_data_run.pickle'):
    mdsFile = scriptOutputs + '/' + 'mds_data_run.pickle';
    print('Found pickled file containing MDS results, here:\n' + mdsFile)
else:
    print('Could not find already run MDS data, will generate independently and save into ' +
scriptOutputs + '/mds_run_data.pickle')

#%% Dictionary of antiSMASH product types converted to BiG-SCAPE classes

# Removed Siderophore as a category (they are now Others)

bgSCAPE_conversion  = {
    "t1pks": "PKS I",
    "T1PKS": "PKS I",
    "transatpks": "PKS Other",
```

```
"t2pks": "PKS Other",
"t3pks": "PKS Other",
"otherks": "PKS Other",
"hglks": "PKS Other",
"transAT-PKS": "PKS Other",
"transAT-PKS-like": "PKS Other",
"T2PKS": "PKS Other",
"T3PKS": "PKS Other",
"PKS-like": "PKS Other",
"hglE-KS": "PKS Other",
"prodigiosin": "PKS Other",
"nrps": "NRPS",
"NRPS": "NRPS",
"NRPS-like": "NRPS",
"thioamide-NRP": "NRPS",
"NAPAA": "NRPS",
"lantipeptide": "RiPP",
"thiopeptide": "RiPP",
"bacteriocin": "RiPP",
"linaridin": "RiPP",
"cyanobactin": "RiPP",
"glycocin": "RiPP",
"LAP": "RiPP",
"lassopeptide": "RiPP",
"sactipeptide": "RiPP",
"bottromycin": "RiPP",
"head_to_tail": "RiPP",
"microcin": "RiPP",
"microviridin": "RiPP",
"proteusin": "RiPP",
"guanidinotides": "RiPP",
"lanthipeptide": "RiPP",
"lipolanthine": "RiPP",
"RaS-RiPP": "RiPP",
"fungal-RiPP": "RiPP",
"thioamitides": "RiPP",
"lanthipeptide-class-i": "RiPP",
"lanthipeptide-class-ii": "RiPP",
"lanthipeptide-class-iii": "RiPP",
"lanthipeptide-class-iv": "RiPP",
"lanthipeptide-class-v": "RiPP",
"ranthipeptide": "RiPP",
"redox-cofactor": "RiPP",
"RRE-containing": "RiPP",
"epipeptide": "RiPP",
```

```
    "cyclic-lactone-autoinducer": "RiPP",
    "spliceotide": "RiPP",
    "crocagin": "RiPP",
    "RiPP-like": "RiPP", # was missing from BiG-SCAPE's class document (https://github.com/medema-
group/BiG-SCAPE/wiki/big-scape-classes#hybrids)
    "amglyccycl": "Saccharide",
    "oligosaccharide": "Saccharide",
    "cf_saccharide": "Saccharide",
    "saccharide": "Saccharide",
    "terpene": "Terpene",
    "acyl_amino_acids": "Other",
    "arylpolyene": "Other",
    "aminocoumarin": "Other",
    "ectoine": "Other",
    "butyrolactone": "Other",
    "nucleoside": "Other",
    "melanin": "Other",
    "phosphoglycolipid": "Other",
    "phenazine": "Other",
    "phosphonate": "Other",
    "other": "Other",
    "cf_putative": "Other",
    "resorcinol": "Other",
    "indole": "Other",
    "ladderane": "Other",
    "PUFA": "Other",
    "furan": "Other",
    "hserlactone": "Other",
    "fused": "Other",
    "cf_fatty_acid": "Other",
    "siderophore": "Other", # Siderophore is its own category for now.
    "blactam": "Other",
    "fatty_acid": "Other",
    "PpyS-KS": "Other",
    "CDPS": "Other",
    "betalactone": "Other",
    "PBDE": "Other",
    "tropodithietic-acid": "Other",
    "NAGGN": "Other",
    "halogenated": "Other",
    "pyrrolidine": "Other",
    "mycosporine-like": "Other",
    "TfuA-related": "RiPP",
}
```

```
#%% AntiSMASH analysis:

chdir(antiSMASHdirectory); # change the directory the antiSMASH directory specified previously.
wd = getcwd(); # get the current path, it should match the antiSMASh directory
# check if the directories match
if wd == antiSMASHdirectory:
    print('Currently in the antiSMASH directory...')

asmStorage = {};
htmlStorageFull = []; # container for the html file path names
htmlStorage = []; # container for mini html file path names
genomeNameS = []; # container for the individual genome names
titleStorage = [];

for filename in os.listdir(wd): # look at each folder/file within the main folder
    newF = os.path.join(wd, filename); # store the folder/file name, append, so we can access further
    genomeNameS.append(newF); # store genome names "generally speaking" here.
    for filename2 in os.listdir(filename): # look at each folder/file in the subfolder
        if filename2.endswith('x.html'): # find only the index.html file
            fname = os.path.join(newF, filename2) # find the filename/path
            htmlStorageFull.append(fname); # store the file name path
            htmlStorage.append(os.path.join(filename, filename2));
            # print("Current file name ..", os.path.abspath(fname)) # print statement to see if we found
the right files

# antiSMASH v5.1.1 specific processing

# Fix the "a998" to "WMMA998":
def transform_string(input_str):
    # Extract the first letter
    first_letter = input_str[0].upper()
    # Add the relevant stuff & replace the first letter.
    result_str = "WMM" + first_letter + input_str[1:]
    return result_str
# Scrape BGC names for later:
bgc_names = {}; # dictionary of dataframes
counter = 1;
for filename in os.listdir(wd): # look at each folder/file within the main folder
    newF = os.path.join(wd, filename); # store the folder/file name, append, so we can access further
    bgc_namesDF = pd.DataFrame();
    for filename2 in os.listdir(filename): # look at each folder/file in the subfolder
        if (('region' in filename2) and ('regions.js' not in filename2)): # find only the files containing
'region'
            bgc_namesDF = bgc_namesDF.append({'BGC Name':filename2.split('.gbk')[0]}, ignore_index = True)
    bgc_names[transform_string(filename)] = bgc_namesDF;
```

```
counter = 0; # useful counter
print('Iterating through ' + str(len(htmlStorage)) + ' folders in the antiSMASH directory...')
for d in htmlStorage: # for each html file found
    htmlfile = open(d) # don't forget to close it at the very end when we're done with it
    contents = htmlfile.read(); # read it out
    beautifulSoupT = BeautifulSoup(contents, 'html.parser') # can update to a later parser eventually
    titleStorage.append(beautifulSoupT.head.title); # this contains the # of regions found, might be
useful

    regList = list(beautifulSoupT.find_all('tbody')); # only take the first one
    regList = list(regList[-1]); # this contains all the info we could want

    asmStorageT = []; # storage for a singular html files import values (all regions)

    for i in np.arange(1, len(regList), 2): # 0, 2, 4,... all are empty headers '\n'
        poolT = regList[i].text.splitlines(); # get only the names, remove the \n (keeps spaces and +)
        asmStorageT.append(poolT); # take the split characters, put the list into the bigger list
    genomeInfo = pd.DataFrame(asmStorageT); # convert the big list into a dataFrame
    # this section is hardcoded, very bad idea, but works for now
    genomeInfo = genomeInfo.drop(columns = [0,1,3,4,6,9]); # remove the columns that are consistently bad
under my current naming scheme
    genomeInfo.columns = ['Region', 'Type', 'From', 'To'];
    nameP = genomeNameS[counter].split('\\')[-1]; # Assuming that the folder name is:
'\path\to\folder\genomeName', this will take just the 'genomeName'
    nameUpd = transform_string(nameP);
    asmStorage[nameUpd] = genomeInfo; # adds to dict the key (genome name), and the dataframe (genome
info)
    htmlfile.close(); # close the file
    counter += 1

# If there was any contamination that you are able to identify in your genome, and want to filter it out
from the data here.
    # Example: If there's contamination in my bacteria strain d975, and I found that it was in antiSMASH
regions 25 through 38 (counting down from the top in antismash), this is how I would remove those data
pieces.
print('The code will now attempt to remove extraneous elements in my data...\nIf they are not present, you
will see some print statements indicating that...')
# Remove stuff from asmStorage
try:
    asmStorage['WMMD975'].drop(list(np.arange(25,39,1)),inplace=True); # Because python is 0 indexed,
region 1 is actually the 0th region in python.
except:
    print('WMMD975 extraneous information was not present...')
try:
```

```
    asmStorage.pop("WMMA1363");
except:
    print('WMMA1363 was not present...')
try:
    asmStorage.pop("WMMB482");
except:
    print('WMMB482 was not present...')
try:
    asmStorage.pop("WMMB486");
except:
    print('WMMB486 is not present...')
try:
    asmStorage.pop('WMMC500');
except:
    print('WMMC500 is not present...')
# Remove stuff from bgc_names
try:
    bgc_names.pop("WMMA1363")
except:
    pass
try:
    bgc_names.pop("WMMB482")
except:
    pass
try:
    bgc_names.pop("WMMB486")
except:
    pass
try:
    bgc_names.pop('WMMC500')
except:
    pass
try:
    bgc_names['WMMD975'].drop(list(np.arange(25,39,1)),inplace=True); # Because python is 0 indexed,
region 1 is actually the 0th region in python.
except:
    pass

print('Finished iterating through the ' + str(counter) + ' folders in the antiSMASH directory, see
asmStorage dictionary for output...')
print('Final amount of antiSMASH results used: ' + str(len(asmStorage)))
asmStorageTemp = asmStorage.copy();

#%% Convert BGC product type to BiG-SCAPE classes:
```

```
print('Replacing antiSMASH product types with BiG-SCAPE classes...')
asmStorageClean = deepcopy(asmStorageTemp);
keyList = list(asmStorageClean.keys());
counter1 = 0;
# Replace antiSMASH product types with BiG-SCAPE classifications.
    # asmStorageClean will contain the replaced information (replaced inplace)
    # asmStorageTemp contains the old antiSMASH product types
for i in keyList: # access each dataframe
    # print(i);
    counter1 +=1;
    # print(counter1);
    dFClean = asmStorageClean[i];
    dfClean_fix = dFClean['Type'].apply(lambda x: ', '.join([bgSCAPE_conversion[word] for word in
x.split(',')]));
    dFClean.Type = dfClean_fix;

# Get all relevant antiSMASH info, stored here. (Unnecessary?)
typeDF_List = [];
for key, df in asmStorageClean.items():
    # print(key)
    df['Genome'] = key;
    if key != 'WMMC500': # Hardcoded because I don't want WMMC500
        typeDF_List.append(df)
combinedTypeDF_List = pd.concat(typeDF_List, ignore_index=True)

# Map the BGC names to the Product Types:
bgcClassStorage = {};
for key, df in asmStorageClean.items():
    mixStorageDF = pd.DataFrame({'name 1': bgc_names[key]['BGC Name'].values, 'bgc product':
df.Type.values.tolist()});
    bgcClassStorage[key] = mixStorageDF;

bgSCAPEclasses = pd.concat(bgcClassStorage, ignore_index = True); # matches the old bgSCAPEclasses excel
document


#%% BiG-SLiCE (Data Generation): Unnecessary to run in future attempts.

try:
    chdir(scriptOutputs)
    # See if the MDS data is already present
    # Load pickle: (contains already analyzed MDS data)
    with open(mdsFile, 'rb') as handle:
        dictDFMDS_ALL = pickle.load(handle)
except:
```

```
    chdir(scriptOutputs)
    # Check to see if the work has been done already.
    try:
        superDF3 = pd.read_csv(queriedBGC_GCFpath);
        print('Taking already run BiG-SLiCE analysis...')
    except:
        print('Running full BiG-SLiCE analysis...')

        # Computationally-intensive. (Generates a BGC x GCF dataframe)
        connMib = sqlite3.connect(pathToDataDB);
        currMib = connMib.cursor();
        # allFolders = np.concatenate([np.arange(111,145,1), np.arange(149,157,1)]); # antismash v5.1.1 #
DEFINED IN SECTION 3
        storageDict = {};
        counterN=0;
        for j in allFolders: # whatever the report runs i want to look at are
            print('Analyzing folder: ' + str(j) + ' and scraping all queried distances associated with
this strain...')
            connF = sqlite3.connect(pathToReportsDB + str(j) + "/data.db") # open a connection
            curF = connF.cursor();
            dfGCF = pd.read_sql_query("SELECT * FROM gcf_membership", connF);
            dfSliceGCF = dfGCF[(dfGCF.iloc[:,3] == 0)]; # get only the highest rank values
            listBGC = list(dfSliceGCF.gcf_id);
            numBGCs = len(dfGCF.bgc_id.value_counts());
            list1, list2, list3, list4 = list(), list(), list(), list();
            count12 = 0;
            for i in np.arange(1,numBGCs+1,1):
                name1 = curF.execute("SELECT name FROM bgc WHERE id=" + str(i)).fetchall()[0]; # modified
for this new version
                dist1 = curF.execute("SELECT membership_value, gcf_membership.gcf_id FROM gcf_membership
WHERE bgc_id=" + str(i)).fetchall(); # modded to grab all ranks
                # gcf1 = currMib.execute("SELECT id_in_run FROM gcf, clustering WHERE gcf.clustering_id=4
AND gcf.id=" + str(dist1[1]) + " AND clustering.run_id=4").fetchall()[0];
                name2 = name1[0].split('/')[0]; # genome name folder
                name22 = name1[0].split('/')[1]; # tig.region
                # I HARDCODED THIS SECTION because it was easier.
                # What this does is it scrapes the genome name associated with this run, and then the
contig and the region for the BGC. You can re-implement this how you see fit.
                if name2 != 'm6_a1363': # hard code it because i messed up.
                    name3 = name2.split('.')[-2]; # bc_genomeName
                    if len(name3.split('_')) < 2: # a rule for if its bc12_genomeName or bc-genomeName
                        name4 = name3.split('-')[-1];
                        name42 = name4 + '.' + name22;
                    else:
                        name4 = name3.split('_')[-1]; # just genomeName
```

```
                        name42 = name4 + '.' + name22; # genomeName.tig00.region00
                else:
                    name4 = 'a1363';
                    name42 = name4 + '.' + name22;
                for k in np.arange(0,len(dist1),1):
                    gcf1 = currMib.execute("SELECT id_in_run FROM gcf, clustering WHERE
gcf.clustering_id=4 AND gcf.id=" + str(dist1[k][1]) + " AND clustering.run_id=4").fetchall()[0];
                    list1.append(name42);
                    list2.append(dist1[k][0]);
                    list3.append(dist1[k][1]);
                    list4.append(gcf1[0]);
                count12 += 1;
            counterN +=1;
            print('Completed folder: ' + str(counterN) + ' of ' + str(len(allFolders)))
            genomeName = name4; # name1[0].split('.')[0];
            nameDistDict = {'BGC': list1, 'Distance': list2, 'gcf_ID': list3, 'GCF_Value':list4};
            nameDist = pd.DataFrame(nameDistDict);
            storageDict[genomeName] = nameDist;
            connF.close();

        # REMOVE IF ERROR (Hardcoded for my own data)
        # Remove the extra stuff in WMMD975 that might be contamination.
        try:
            d975Stuff = storageDict['d975'];
            d975StuffKeep = d975Stuff[d975Stuff['BGC'].str.contains('tig00000001')];
            d975StuffRemove = d975Stuff[~d975Stuff['BGC'].str.contains('tig00000001')];
            storageDict['d975'] = d975StuffKeep;
        except:
            print('WMMD975 was not found in the BiG-SLiCE files...')

        print('Finished scraping all of the BiG-SLiCE report folders...')
        print('Construcing file for exporting...')
        # Merge all of the information into one big dataframe. (X, where X = GCF*BGC, by 4)
        newDF = pd.DataFrame(columns = list(storageDict['a1363'].columns));
        dfs_list = [];
        for i in list(storageDict.keys()):
            print(i)  # Tracker: outputs to terminal
            oldDF = storageDict[i].sort_values(by=['BGC', 'gcf_ID'])
            dfs_list.append(oldDF)
        newDF = pd.concat(dfs_list, ignore_index=True)
        testDF = newDF;

        # Dataframe (rows are BGCs, columns are GCF values)
        testT1 = list(testDF.BGC); # get the BGC names.
        testT1 = list(set(testT1)); # get the unique BGC names.
```

```
        testT2 = list(testDF.gcf_ID); # get the gcf id
        testT2 = list(set(testT2)); # get the unique GCF ids

        # Get the individual data into lists, ordered and sorted.
        superList1 = [];
        counterT = 0;
        counterT2 = 0;
        for i in testT1:
            placeHolder1 = testDF.loc[(testDF['BGC'] == i)]; # returns rows of only the BGC (sorted by
gcf_ID)
            pH2 = list(placeHolder1.Distance);
            pH3 = list(placeHolder1.gcf_ID);
            superList1.append(pH2)
            counterT2 += 1;
            if (counterT2 - counterT) >= 30: # Tracker (outputs to console)
                # print(counterT2);
                counterT = counterT2;

        print("Making a mega dataframe, takes awhile...")
        # Make a super dataframe, with the columns as GCFs-200946 to the end.
        superDF2 = pd.DataFrame([i for i in superList1], columns = testT2);
        superDF2.index = testT1; # change the row names to be the BGC names.

        superDFbackup = superDF2.copy(deep=True);

        # REMOVE IF ERROR, BASED ON MY ORIGINAL DATA: Removes the Streptomycetaceae (c500) and the ones
that failed QC.
        superDF2 = superDF2[~superDF2.index.str.contains('c500|a1363|b482|b486')];

        # print('Exporting the BGC x GCF file for future re-use to ' + resultsDirectory + r'\antismash5-
strainsQueried-rank.csv')

        # Must swap the rows and columns, as excel doesn't like having 29,955 columns.
        superDF2T = superDF2.T;

        # Export the superDF2 (GCF x BGC file) to CSV for ease of access later.
        if not os.path.isfile('antismash5-strainsQueried-rank.csv'):
            superDF2T.to_csv('antismash5-strainsQueried-rank.csv');

    #%% BiG-SLiCE (can run without previous section, if it's been run before):

    chdir(scriptOutputs)

    # ---- If re-using a csv file containing all BGCs x GCFs queried, can load here.
    print('Loading file at ' + queriedBGC_GCFpath)
```

```
    # Load file.
    try:
        superDF3 = pd.read_csv(queriedBGC_GCFpath);
    except:
        print('You need to run the previous section, titled:\nBiG-SLiCE (Data Generation)')
    # Get the index.
    superDF3.index = superDF3.iloc[:,0]; # replace index with GCFs
    # Remove the first column (which previously contained the index)
    superDF3.drop(columns = superDF3.columns[0], axis=1, inplace=True); # remove first column
    # Transpose so it is in the shaep of BGC x GCF.
    superDF3=superDF3.T;
    # Scrape the index.
    testT1 = list(superDF3.index);

    superDF2 = superDF3;

    # ---- Data necessary for the Figures.

    #%% Generate Data for the non-Hybrid version

    # Load bigscape class product types in relation to bgc names.
    # bgSCAPEclasses = pd.read_excel(bigSCAPEclassesAllpath);
    bgSCAPEclasses = bgSCAPEclasses

    # Create filters for eventually splitting into the relevant datasets:

    # List of substrings to check
    substrings = ['Terpene', 'PKS I', 'RiPP', 'Other', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Saccharide']
    # Create a new column for each category and mark with True or False
    for sub in substrings:
        bgSCAPEclasses[sub] = bgSCAPEclasses['bgc product'].str.contains(sub, case=False, na=False,
regex=True)
    # Create a new column for 'PKS/NRPS'
    bgSCAPEclasses['PKS/NRPS'] = bgSCAPEclasses['bgc product'].str.contains('PKS I') & bgSCAPEclasses['bgc
product'].str.contains('NRPS')
    bgSCAPEclasses['PKS Other'] = bgSCAPEclasses['bgc product'].str.contains(r'PKS Other\b', case=False,
na=False, regex=True)
    bgSCAPEclasses['Other'] = bgSCAPEclasses['bgc product'].str.contains(r'(?<!PKS )Other', case=False,
na=False, regex=True)

    # Filter the DataFrame based on the presence of substrings
    filtered_df = bgSCAPEclasses[bgSCAPEclasses[substrings].any(axis=1)]
    missing_df = bgSCAPEclasses[~bgSCAPEclasses[substrings].any(axis=1)]
```

```
    colors = ['#a6cee3','#1f78b4','#b2df8a','#33a02c','#e31a1c','#fdbf6f','#ff7f00','#cab2d6']; # colors
match the ITOL TREE
    classes1 = ['Terpene', 'PKS I', 'RiPP', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Other', 'Saccharide']; #
matches the colors indices.

    # Construct a dictionary of palettes.
    palDict = {};
    for i in range(0,len(colors),1):
        palDict[classes1[i]] = colors[i];

    classDict_Full = {};
    for i in classes1:
        classDict_Full[i] = bgSCAPEclasses[bgSCAPEclasses[i]];

    # # Find which indices in the BGCs map to which classes.
    # classDict = {}; # Keys are the classes, each key links to a list of the indices where it was found
in classList
    # for i in classes1:
    #     classDict[i] = [a for a, x in enumerate(classList) if x == i];
    # Copy superDF2 just incase.
    superCatDF2 = superDF2.copy(deep=True);

    dictDistMetrics = {}; # initialize empty dictionary for storage
    distMets = ['euclidean', 'cosine', 'cityblock', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra',
'chebyshev', 'correlation', 'hamming'] # this is where the different strings for distance metrics go
    superDict0 = {}; # dictionary of the different simplified datasets I want to play with.
    # Create the datasets (sliced by bigSCAPE categories)
    for b in classes1:
        # Store the relevant data from superDF2 based on the bigscape class the BGC contains.
        superDict0[b+"_orig"] = superCatDF2[superCatDF2.index.isin(classDict_Full[b]['name 1'])]

    # Initialize an empty dictionary
    dictDistMetrics = {};
    # Run the distance metrics, the structure of dictDistMetrics is as follows:
        # 'Terpene_orig' -> 'Terpene_orig_euclidean", "Terpene_orig_cosine", etc (each is a dataframe)
    for j in list(superDict0.keys()):
        dictDistMetrics[j] = {};
        print('Iterating over BGC class: ' + j)
        for i in distMets:
            print('Computing distance for: ' + i)
            outputDist0 = pd.DataFrame(sklearn.metrics.pairwise_distances(superDict0[j], Y=None, metric=i,
n_jobs=None, force_all_finite=True), columns = superDict0[j].index.tolist());
            outputDist0.index = superDict0[j].index.tolist();
            dictDistMetrics[j][j+"_"+i] = outputDist0; # pop it back in
    # Convert the distance metrics from dataframes to arrays for MDS.
```

```
    print('Converting dataframes to arrays...')
    dictArray = {};
    for j in list(superDict0.keys()):
        dictArray[j] = {};
        for i in distMets:
            outputArray = dictDistMetrics[j][j+"_"+i].to_numpy();
            dictArray[j][j+"_"+i] = outputArray;

    # Perform MDS
    print('Performing MDS...')
    dictMDS = {};
    nComps = 2; # number of components
    for j in list(superDict0.keys()):
        print('Iterating over BGC class: ' + j)
        dictMDS[j] = {};
        for i in distMets:
            print('Metric used for MDS currently: ' + i)
            mdsModelA = MDS(n_components = nComps, dissimilarity='precomputed', random_state=0);
            dataTrans = mdsModelA.fit_transform(dictArray[j][j+"_"+i]);
            dictMDS[j][j+"_"+i] = dataTrans;

    print('Converting to dataframes for ease of usage...')
    # aveDist = 3; # No longer necessary. I set hardcoded values of 1,2,3,4,5 as k in the below loop.
    dictDFMDS = {};
    for j in list(superDict0.keys()):
        dictDFMDS[j] = {};
        superDict0_j = superDict0[j];
        print('Iterating through: ' + j)
        for i in distMets:
            dfMDStemp = pd.DataFrame(dictMDS[j][j+"_"+i], index = superDict0_j.index.tolist())
            dfMDStemp['Category'] = j.split("_")[0]; # bigscape product types
            # grab the <900 >900 information from bigslice
            # list comprehension (if/else statement). GOal: '> 900' if the BGC fell above 900 in BiGSLICE,
otherwise '< 900'
            # bigSLICEinfo = ['> 900' if storage2DF.loc[storage2DF['BGC'] == iter1].Distance.tolist()[0]
>= 900 else '< 900' for iter1 in superDict0[j].index.tolist()];

            # Removed dependency on storageDict
            bigSLICEinfo_v2 = ['> 900' if min(superDF2.loc[iter1]) >= 900 else '< 900' for iter1 in
superDict0_j.index.tolist()]
            # find the distance to the closest GCF
            # closestGCFDist = [storage2DF.loc[storage2DF['BGC'] == iter1].Distance.tolist()[0] for iter1
in superDict0[j].index.tolist()];

            # Removed dependency on storageDict
```

```
            closestGCFDist_v2 = [min(superDF2.loc[iter1]) for iter1 in superDict0_j.index.tolist()]

            # Do some average distance calculations.
            for k in [1,2,3,4,5]:
                # find the distance to the X closest GCFs, then average them. Where X is set above as
aveDist
                # aveDistStorage = [sorted(superDict0_j.loc[iter2])[:k] for iter2 in superDict0_j.index];
# list of lists
                # averageDist = [np.mean(entry1) for entry1 in aveDistStorage]; # average each list inside
the list of lists
                # dfMDStemp['Average of ' + str(k) + ' Distances'] = averageDist;
                dfMDStemp[f'Average of {k} Distances'] = [np.mean(sorted(superDict0_j.loc[iter2])[:k]) for
iter2 in superDict0_j.index]



            dfMDStemp['BIG-SLICE'] = bigSLICEinfo_v2; # not dependent on storageDict anymore
            dfMDStemp['Distance'] = closestGCFDist_v2; # not dependent on storageDict anymore
            # dfMDStemp['Average Distance'] = averageDist;
            # dfMDStemp['# of GCF Distances used'] = aveDist;
            # Add metadata
            # bgSCAPEinfo = [bgSCAPEclasses.loc[bgSCAPEclasses['name 1'] == i]['bgc product'].values for i
in dfMDStemp.index];
            # dfMDStemp['Metadata'] = bgSCAPEinfo
            dfMDStemp['Metadata'] = dfMDStemp.index.map(lambda x: bgSCAPEclasses.loc[bgSCAPEclasses['name
1'] == x]['bgc product'].values[0])
            dfMDStemp.rename(columns={0:'x', 1:'y'}, inplace=True) # clean up column names for ease of
reference later

            dictDFMDS[j][j+"_"+i] = dfMDStemp;

    specificMets = distMets # this is where the different strings for distance metrics go
    dictDFMDS_ALL = {};
    dictDFMDS_ALL['dictDFMDS_noHybridCategory'] = dictDFMDS;

    #%% Generate data necessary for the Hybrid dataset.

    bgSCAPEclasses = pd.concat(bgcClassStorage, ignore_index = True); # matches the old bgSCAPEclasses
excel document

    # Create filters for eventually splitting into the relevant datasets:

    # List of substrings to check
    substrings = ['Terpene', 'PKS I', 'RiPP', 'Other', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Saccharide']
    # Create a new column for each category and mark with True or False
    for sub in substrings:
```

```
        bgSCAPEclasses[sub] = bgSCAPEclasses['bgc product'] == sub;
        # bgSCAPEclasses['bgc product'].str.match(sub, case=True, na=False)
    # Create a new column for 'PKS/NRPS'
    pks_nrps_substrings = ['PKS I, NRPS', 'NRPS, PKS I'];
    bgSCAPEclasses['PKS/NRPS'] = [value in pks_nrps_substrings for value in bgSCAPEclasses['bgc product']]

    new_substrings = ['Terpene', 'PKS I', 'RiPP', 'Other', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Saccharide']

    # Filter the DataFrame based on the presence of substrings
    filtered_df = bgSCAPEclasses[bgSCAPEclasses[new_substrings].any(axis=1)]
    missing_df = bgSCAPEclasses[~bgSCAPEclasses[new_substrings].any(axis=1)]

    # Currently, the Hybrid rule is this:
        # The BGCs that are in each category, they are purely just 1 BGC. Meaning, 'PKS I' only shows up
in the category 'PKS I'.
        # So, 'PKS I, PKS I' would show up only in the Hybrid category. 'PKS I, Other' would show up only
in the Hybrid category.
        # However, 'PKS/NRPS' category includes only BGCs that have 'PKS I, NRPS' or 'NRPS, PKS I'.
    # Create the Hybrid category.
    bgSCAPEclasses['Hybrid (Not PKS/NRPS)'] = ~bgSCAPEclasses[new_substrings].any(axis=1);

    colors = ['#a6cee3','#1f78b4','#b2df8a','#33a02c','#e31a1c','#fdbf6f','#ff7f00','#cab2d6','#6a3d9a'];
# colors match the ITOL TREE
    classes1 = ['Terpene', 'PKS I', 'RiPP', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Other', 'Saccharide',
'Hybrid (Not PKS/NRPS)']; # matches the colors indices.

    # Construct a dictionary of palettes.
    palDict = {};
    for i in range(0,len(colors),1):
        palDict[classes1[i]] = colors[i];

    classDict_Full = {};
    for i in classes1:
        classDict_Full[i] = bgSCAPEclasses[bgSCAPEclasses[i]];

    # # Find which indices in the BGCs map to which classes.
    # classDict = {}; # Keys are the classes, each key links to a list of the indices where it was found
in classList
    # for i in classes1:
    #     classDict[i] = [a for a, x in enumerate(classList) if x == i];
    # Copy superDF2 just incase.
    superCatDF2 = superDF2.copy(deep=True);

    dictDistMetrics = {}; # initialize empty dictionary for storage
```

```
    distMets = ['euclidean', 'cosine', 'cityblock', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra',
'chebyshev', 'correlation', 'hamming'] # this is where the different strings for distance metrics go
    superDict0 = {}; # dictionary of the different simplified datasets I want to play with.
    # Create the datasets (sliced by bigSCAPE categories)
    for b in classes1:
        # Store the relevant data from superDF2 based on the bigscape class the BGC contains.
        superDict0[b+"_orig"] = superCatDF2[superCatDF2.index.isin(classDict_Full[b]['name 1'])]

    # Initialize an empty dictionary
    dictDistMetrics = {};
    # Run the distance metrics, the structure of dictDistMetrics is as follows:
        # 'Terpene_orig' -> 'Terpene_orig_euclidean", "Terpene_orig_cosine", etc (each is a dataframe)
    for j in list(superDict0.keys()):
        dictDistMetrics[j] = {};
        print('Iterating over BGC class: ' + j)
        for i in distMets:
            print('Computing distance for: ' + i)
            outputDist0 = pd.DataFrame(sklearn.metrics.pairwise_distances(superDict0[j], Y=None, metric=i,
n_jobs=None, force_all_finite=True), columns = superDict0[j].index.tolist());
            outputDist0.index = superDict0[j].index.tolist();
            dictDistMetrics[j][j+"_"+i] = outputDist0; # pop it back in
    # Convert the distance metrics from dataframes to arrays for MDS.
    print('Converting dataframes to arrays...')
    dictArray = {};
    for j in list(superDict0.keys()):
        dictArray[j] = {};
        for i in distMets:
            outputArray = dictDistMetrics[j][j+"_"+i].to_numpy();
            dictArray[j][j+"_"+i] = outputArray;

    # Perform MDS
    print('Performing MDS...')
    dictMDS = {};
    nComps = 2; # number of components
    for j in list(superDict0.keys()):
        print('Iterating over BGC class: ' + j)
        dictMDS[j] = {};
        for i in distMets:
            print('Metric used for MDS currently: ' + i)
            mdsModelA = MDS(n_components = nComps, dissimilarity='precomputed', random_state=0);
            dataTrans = mdsModelA.fit_transform(dictArray[j][j+"_"+i]);
            dictMDS[j][j+"_"+i] = dataTrans;

    print('Converting to dataframes for ease of usage...')
    # aveDist = 3; # No longer necessary. I set hardcoded values of 1,2,3,4,5 as k in the below loop.
```

```
    dictDFMDS = {};
    for j in list(superDict0.keys()):
        dictDFMDS[j] = {};
        superDict0_j = superDict0[j];
        print('Iterating through: ' + j)
        for i in distMets:
            dfMDStemp = pd.DataFrame(dictMDS[j][j+"_"+i], index = superDict0_j.index.tolist())
            dfMDStemp['Category'] = j.split("_")[0]; # bigscape product types
            # grab the <900 >900 information from bigslice
            # list comprehension (if/else statement). GOal: '> 900' if the BGC fell above 900 in BiGSLICE,
otherwise '< 900'
            # bigSLICEinfo = ['> 900' if storage2DF.loc[storage2DF['BGC'] == iter1].Distance.tolist()[0]
>= 900 else '< 900' for iter1 in superDict0[j].index.tolist()];

            # Removed dependency on storageDict
            bigSLICEinfo_v2 = ['> 900' if min(superDF2.loc[iter1]) >= 900 else '< 900' for iter1 in
superDict0_j.index.tolist()]
            # find the distance to the closest GCF
            # closestGCFDist = [storage2DF.loc[storage2DF['BGC'] == iter1].Distance.tolist()[0] for iter1
in superDict0[j].index.tolist()];

            # Removed dependency on storageDict
            closestGCFDist_v2 = [min(superDF2.loc[iter1]) for iter1 in superDict0_j.index.tolist()]

            # Do some average distance calculations.
            for k in [1,2,3,4,5]:
                # find the distance to the X closest GCFs, then average them. Where X is set above as
aveDist
                # aveDistStorage = [sorted(superDict0_j.loc[iter2])[:k] for iter2 in superDict0_j.index];
# list of lists
                # averageDist = [np.mean(entry1) for entry1 in aveDistStorage]; # average each list inside
the list of lists
                # dfMDStemp['Average of ' + str(k) + ' Distances'] = averageDist;
                dfMDStemp[f'Average of {k} Distances'] = [np.mean(sorted(superDict0_j.loc[iter2])[:k]) for
iter2 in superDict0_j.index]


            dfMDStemp['BIG-SLICE'] = bigSLICEinfo_v2; # not dependent on storageDict anymore
            dfMDStemp['Distance'] = closestGCFDist_v2; # not dependent on storageDict anymore
            # dfMDStemp['Average Distance'] = averageDist;
            # dfMDStemp['# of GCF Distances used'] = aveDist;
            # Add metadata
            # bgSCAPEinfo = [bgSCAPEclasses.loc[bgSCAPEclasses['name 1'] == i]['bgc product'].values for i
in dfMDStemp.index];
            # dfMDStemp['Metadata'] = bgSCAPEinfo
```

```
            dfMDStemp['Metadata'] = dfMDStemp.index.map(lambda x: bgSCAPEclasses.loc[bgSCAPEclasses['name
1'] == x]['bgc product'].values[0])
            dfMDStemp.rename(columns={0:'x', 1:'y'}, inplace=True) # clean up column names for ease of
reference later

            dictDFMDS[j][j+"_"+i] = dfMDStemp;

    specificMets = distMets # this is where the different strings for distance metrics go
    dictDFMDS_ALL['dictDFMDS_yesHybridCategory'] = dictDFMDS;
    # Save to Pickle (in current directory, which should be scriptOutputs)
    if not os.path.isfile('mds_data_run.pickle'):
        with open('mds_data_run.pickle', 'wb') as handle:
            pickle.dump(dictDFMDS_ALL, handle, protocol = pickle.HIGHEST_PROTOCOL)

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nCompleted processing in {}'.format(t_str))

#%% Dash App
chdir(scriptOutputs)

colorSelectionScale = px.colors.named_colorscales()
classes1 = ['Terpene', 'PKS I', 'RiPP', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Other', 'Saccharide', 'Hybrid
(Not PKS/NRPS)']; # matches the colors indices.
substrings = ['Terpene', 'PKS I', 'RiPP', 'Other', 'PKS Other', 'PKS/NRPS', 'NRPS', 'Saccharide']
distMets = ['euclidean', 'cosine', 'cityblock', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra',
'chebyshev', 'correlation', 'hamming'] # this is where the different strings for distance metrics go


#%% Initialize App

# Sample empty plot
empty_plot = px.scatter()

app = dash.Dash(__name__)
app.title = "BGC Prioritization App"

# Define the layout of the app
app.layout = html.Div([
    # Left column with dropdowns and search bar
    html.Div([
        html.H1("BGC Prioritization Display", style={'textAlign': 'center'}),
        html.H3("Welcome to the Dash Tool to Assist with Prioritizing BGCs Based on Potential Novelty",
```

```
                style={'textAlign': 'center'}),
        html.H5("Developer: Imraan Alas (Bugni Lab)", style={'textAlign':'center'}),
        html.Div(id="intro",
                 children="Manipulate the various dropdown bars to customize which data is being
presented, "
                          "the metadata used to color the plots, and the associated color scales.",
                 style={'textAlign': 'center'}),
        html.P("BGC Class: Choose from pre-defined BGC classes. You can't select Hybrid if Hybrids are not
separated.", style={'textAlign': 'center'}),
        dcc.Dropdown(
            id='bigScapeClass-dropdown',
            options=[{'label': key, 'value': key} for key in classes1],
            value=substrings[0],  # Default value
            style={'width': '48%', 'margin': 'auto', 'text-align': 'center'},
            clearable=False,
            multi=False
        ),
        html.P("Distance Metric: Choose the distance metric used for metric dimensional scaling.",
style={'textAlign': 'center'}),
        dcc.Dropdown(
            id='distanceMetric-dropdown',
            options=[{'label': metric, 'value': metric} for metric in distMets],
            value= 'chebyshev', # distMets[0],  # Default value
            style={'width': '48%', 'margin': 'auto', 'text-align': 'center'},
            clearable=False,
            multi=False
        ),
        html.P("N Closest Distances: Choose the # of closest GCFs to a BGC that are considered for the
average distance metadata.", style={'textAlign': 'center'}),
        dcc.Dropdown(
            id='averageDistance-dropdown',
            options=[{'label': metric, 'value': metric} for metric in [1, 2, 3, 4, 5]],
            value=3,  # Default value
            style={'width': '48%', 'margin': 'auto', 'text-align': 'center'},
            clearable=False,
            multi=False
        ),
        html.P("Color Palette: Choose a relevant color palette based on Plotly's inbuilt color scales.",
style={'textAlign': 'center'}),
        dcc.Dropdown(
            id='colorPalette-dropdown',
            options=colorSelectionScale,
            value='rdpu',  # Default value
            style={'width': '48%', 'margin': 'auto', 'text-align': 'center'},
            clearable=False
```

```
        ),
        html.P("Hybrids Separated? Selecting 'No' will make BGC evaluations through the BGCPD more
difficult.", style={'textAlign':'center'}),
        dcc.Dropdown(
            id='hybridSelection-dropdown',
            options=[{'label': metric, 'value': metric} for metric in ['Yes', 'No']],
            value='Yes',  # Default value
            style={'width': '48%', 'margin': 'auto', 'text-align': 'center'},
            clearable=False
        ),
        html.P("Search BGC: Enter BGC name to filter the scatter plot.", style={'textAlign':'center'}),
        dcc.Input(
            id='search-input',
            type='text',
            value='',  # Default value
            placeholder='Enter BGC name...',
            style={'width': '99%', 'margin': 'auto', 'text-align': 'center'},
        ),
    ], style={'width': '48%', 'float': 'left'}),

    # Right column with the scatter plot
    html.Div([
        dash.html.Center(dcc.Graph(id='scatter-plot', figure=empty_plot)),
    ], style={'width': '48%', 'float': 'right'})

], style={"backgroundColor": "#f0f0f0", "display": "flex"})

# Define callback to update the scatter plot
@app.callback(
    Output('scatter-plot', 'figure'),
    [Input('bigScapeClass-dropdown', 'value'),
     Input('distanceMetric-dropdown', 'value'),
     Input('averageDistance-dropdown', 'value'),
     Input('colorPalette-dropdown', 'value'),
     Input('hybridSelection-dropdown', 'value'),
     Input('search-input', 'value')]
)
def update_scatter_plot(selected_big_scape_class, selected_distance_metric, selected_average_distance,
                        selected_color_scale, selected_hybrids, search_value):
    try:
        selected_key = f"{selected_big_scape_class}_orig_{selected_distance_metric}"
        if selected_hybrids == 'Yes':
            selectedHybridVal = 'yes'
        else:
            selectedHybridVal = 'no'
```

```
    except:
        return empty_plot

    try:
        if selected_key in dictDFMDS_ALL['dictDFMDS_' + selectedHybridVal +
'HybridCategory'][selected_big_scape_class + '_orig']:
            bgcTitleName = selected_key.split('_')[0]
            metricTitleName = selected_key.split('_')[-1].capitalize()
            df = dictDFMDS_ALL['dictDFMDS_' + selectedHybridVal +
'HybridCategory'][selected_big_scape_class + '_orig'][selected_key]

            selected_aveDist = 'Average of ' + str(selected_average_distance) + ' Distances'
            fig = px.scatter(df, x='x', y='y', color_continuous_scale=selected_color_scale,
color=selected_aveDist,
                             size=selected_aveDist, hover_data=['Metadata', 'Distance'],
                             hover_name=['WMM' + i.capitalize() for i in df.index.tolist()])
            fig.update_layout(height=900, width=1000)
            fig.update_layout(title_text=f'{bgcTitleName} BGCs Analyzed with {metricTitleName} Distance
Metric '
                                         f'<br><sup>Size & Color Described by Average Distance of BGC to
Closest {selected_average_distance} GCFs</sup>',
                              title_x=0.5)
            fig.update_layout(xaxis_title='Dimension 1', yaxis_title='Dimension 2')

            if search_value:
                df['IsSearchMatch'] = df.index.str.contains(search_value, case=False)
                fig.update_traces(
                    marker=dict(line=dict(color='black', width=[3 if is_search_match and search_value else
0.5 for is_search_match in df['IsSearchMatch']])))
            else:
                fig.update_traces(marker=dict(line=dict(color='white', width=1)))
            return fig
        else:
            return empty_plot
    except:
        return empty_plot

# Run the app
if __name__ == '__main__':
    print('Open localhost:8002 or 127.0.0.1:8002 to see the dash visualization')
    app.run_server(port=8002)  # Open 127.0.0.1:8002 or localhost:8002 on your local web browser.
```

Appendix B

Supplementary data for Chapter 4

Code B.1 Binary classification model hyperparameter optimization **117**
Code B.2 Binary classification model evaluation on 20% testing data **148**
Code B.3 Binary classification model evaluation on GNPS spectra **154**
Code B.4 Multiclass classification model hyperparameter optimization and evaluation **158**
Code B.5 Function to determine spectra quality grades **167**
Table B.1 Hyperparameters for multiclass classification **168**
Table B.2 Hyperparameters selected for multiclass classification **170**

Code B.1: Binary classification model hyperparameter optimization

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 25 20:56:40 2023

@author: alas
"""
#%% Imports

import os
from os import chdir

import numpy as np
import pandas as pd
import seaborn as sns
import sklearn
import pickle
import sys
import lightgbm as lgb
import matplotlib.pyplot as plt
import time
import catboost
import xgboost as xgb


from rdkit import Chem
from rdkit.Chem import Descriptors
from rdkit.Chem import PandasTools
from rdkit.Chem import AllChem

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_validate
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score, average_precision_score
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
confusion_matrix
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.feature_selection import VarianceThreshold,SelectKBest, f_classif, RFECV
from scipy.signal import find_peaks
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB, ComplementNB

from catboost import CatBoostClassifier, Pool

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, average_precision_score, precision_recall_curve, auc,
PrecisionRecallDisplay


#%% Directory

# chdir(r"C:\Users\imraa\Documents\UWM\BugniLab\Tubulin-Binders")
chdir(r"C:\Users\alas\Documents\Python")

results_df = None;

#%% Load Data:

## LOAD DATA
```

```
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)


#%% The Simplest Classification Model: Logistic Regression

# Retry: This time, train on the entire training dataset. Then, evaluate on the validation data.
    # Expectation: Everything decreases across the board because there's little to no spillover of
compounds across training to prediction.

### Load & process the validation data.
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

### Models:
# Train on training data. Evaluate on validation data.
# Default model
clf_def = LogisticRegression(random_state=42,max_iter=10000).fit(X,y);

# Standard Scaler Model
scaler = StandardScaler();
X_standard = scaler.fit_transform(X);
X_val_standard = scaler.transform(X_val);
clf_stand = LogisticRegression(random_state=42, max_iter=10000).fit(X_standard,y);
# Min Max Scaler Model
min_max_scaler = MinMaxScaler()
X_min_max = min_max_scaler.fit_transform(X)
X_val_min_max = min_max_scaler.transform(X_val)
clf_minmax = LogisticRegression(random_state=42,max_iter=10000).fit(X_min_max,y)

### Evaluate Models:
dict_yPred = {};
y_pred_def = clf_def.predict(X_val);
dict_yPred['No Scaler'] = y_pred_def;
report_def = classification_report(y_val,y_pred_def,zero_division=0);
print('Classification report: no Scaler:\n', report_def);
y_pred_standard = clf_stand.predict(X_val);
```

```
dict_yPred['Standard Scaler'] = y_pred_standard;
report_stand = classification_report(y_val, y_pred_standard,zero_division=0);
print('Classification report: StandardScaler:\n', report_stand);
y_pred_minmax = clf_minmax.predict(X_val);
dict_yPred['MinMax Scaler'] = y_pred_minmax;
report_minmax = classification_report(y_val,y_pred_minmax,zero_division=0);
print('Classification report: MinMaxScaler:\n',report_minmax);

# # Store models.
# clf_storage = {};
# clf_storage['No Scaler'] = clf_def;
# clf_storage['Standard Scaler'] = clf_stand;
# clf_storage['MinMax Scaler'] = clf_minmax;

# # Generate confusion matrices.
# for i in list(dict_yPred.keys()):
#     cm = confusion_matrix(y_val,dict_yPred[i])
#     plt.figure(figsize=(6, 4))
#     sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
#     plt.xlabel('Predicted')
#     plt.ylabel('Actual')
#     plt.title('Confusion Matrix: Logistic Regression with ' + str(i))
#     plt.show()

# # Generate precision-recall curves.
# for i in list(dict_yPred.keys()):
#     precision1, recall1, threshold1 = precision_recall_curve(y_val,dict_yPred[i])
#     display = PrecisionRecallDisplay.from_predictions(y_val,dict_yPred[i], name = i + ' Linear
Regression')
#     _ = display.ax_.set_title(i + ' Linear Regression: Precision-Recall Curve')
#     auc_score = auc(recall1, precision1);
#     print(i + ' Linear Regression AUPRC Score: ' + str(auc_score))

# f1_types = ['binary','macro','micro','weighted']
# for j in f1_types:
#     print('\n'+j)
#     for i in list(dict_yPred.keys()):
#         f1SCORE1 = f1_score(y_val, dict_yPred[i], average = j);
#         print(i + ' Linear Regression F1 Score: ' + j + ' ' + str(f1SCORE1))

# Summarize Everything:
# Create Results DF
results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

for i in list(dict_yPred.keys()):
    cm = confusion_matrix(y_val, dict_yPred[i]);
    tn, fp, fn, tp = cm.ravel();
    precision1, recall1, threshold1 = precision_recall_curve(y_val, dict_yPred[i]) # Precision, Recall
    auc_score = auc(recall1, precision1) # AUPRC
    f1_binary = f1_score(y_val, dict_yPred[i], average='binary') # F1-binary
    f1_macro = f1_score(y_val, dict_yPred[i], average='macro') # F1-macro
    temp_df = pd.DataFrame({
        'Model': [f'Linear Regression ({i})'],
        'True Negative': [tn],
        'False Positive': [fp],
        'False Negative': [fn],
        'True Positive': [tp],
        'F1-Score (Binary)': [f1_binary],
        'F1-Score (Macro)': [f1_macro],
        'Precision': [precision_score(y_val, dict_yPred[i])],
        'Recall': [recall_score(y_val, dict_yPred[i])],
        'AUPRC': [auc_score]
    })
    results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% SECOND MODEL: Decision Tree
```

```
### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }
# TIMER:
startTime = time.time();

### Define param grid:
param_grid_DT = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30, 40,50,60],
    'min_samples_split': [2, 5, 10,20,50],
    'min_samples_leaf': [1, 2, 4,6,10],
    'max_features': [None, 'sqrt', 'log2']
}
print('Starting gridsearch!...')
# GridsearchCV setup:
grid_search_DT = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42), param_grid=param_grid_DT,
                              cv=5, scoring=scoring, refit = 'F1-score', n_jobs=-1)
# Perform the grid search model generation
grid_search_DT.fit(X,y);
# Evaluate the best model on the validation data.
best_model_DT = grid_search_DT.best_estimator_
y_pred_val_DT = best_model_DT.predict(X_val);
# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))
```

```
# Reports
report_DT = classification_report(y_val,y_pred_val_DT,zero_division=0);
print('Classification report: Decision Tree (GridSearch 1):\n', report_DT);

cm_DT = confusion_matrix(y_val,y_pred_val_DT)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_DT, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Decision Tree (Default)')
plt.show()

### Standard Scaler
print('Starting gridsearch2!...')
scaler = StandardScaler();
X_standard = scaler.fit_transform(X);
X_val_standard = scaler.transform(X_val);
grid_search_DT_stand = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42),
param_grid=param_grid_DT,
                           cv=5, scoring=scoring, refit = 'F1-score', n_jobs=-1)
grid_search_DT_stand.fit(X_standard,y);
best_model_DT_stand = grid_search_DT_stand.best_estimator_
y_pred_val_DT_stand = best_model_DT_stand.predict(X_val);
# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))
report_DT_stand = classification_report(y_val,y_pred_val_DT_stand,zero_division=0);
print('Classification report: Decision Tree (GridSearch: Standard Scaler):\n', report_DT_stand);

cm_DT_stand = confusion_matrix(y_val,y_pred_val_DT_stand)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_DT_stand, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Decision Tree (StandardScaler)')
plt.show()

### MinMax Scaler
print('Starting gridsearch3!...')
min_max_scaler = MinMaxScaler()
X_min_max = min_max_scaler.fit_transform(X)
X_val_min_max = min_max_scaler.transform(X_val)
grid_search_DT_mm = GridSearchCV(estimator=DecisionTreeClassifier(random_state=42),
param_grid=param_grid_DT,
                           cv=5, scoring=scoring, refit = 'F1-score', n_jobs=-1)
grid_search_DT_mm.fit(X_min_max,y);
best_model_DT_mm = grid_search_DT_mm.best_estimator_
y_pred_val_DT_mm = best_model_DT_mm.predict(X_val);
# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))
report_DT_mm = classification_report(y_val,y_pred_val_DT_mm,zero_division=0);
print('Classification report: Decision Tree (GridSearch: MinMax Scaler):\n', report_DT_mm);

cm_DT_mm = confusion_matrix(y_val,y_pred_val_DT_mm)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_DT_mm, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Decision Tree (MinMax Scaler)')
plt.show()
```

```
# Summarize Results:

dict_yPred = {};
dict_yPred['No Scaler'] = y_pred_val_DT;
dict_yPred['Standard Scaler'] = y_pred_val_DT_stand;
dict_yPred['MinMax Scaler'] = y_pred_val_DT_mm

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate the best-performing models (as determined through highest F1 during the gridsearch for ideal
hyperparameters). Store metrics.
for i in list(dict_yPred.keys()):
    cm = confusion_matrix(y_val, dict_yPred[i]);
    tn, fp, fn, tp = cm.ravel();
    precision1, recall1, threshold1 = precision_recall_curve(y_val, dict_yPred[i]) # Precision, Recall
    auc_score = auc(recall1, precision1) # AUPRC
    f1_binary = f1_score(y_val, dict_yPred[i], average='binary') # F1-binary
    f1_macro = f1_score(y_val, dict_yPred[i], average='macro') # F1-macro
    temp_df = pd.DataFrame({
        'Model': [f'Decision Tree ({i})'],
        'True Negative': [tn],
        'False Positive': [fp],
        'False Negative': [fn],
        'True Positive': [tp],
        'F1-Score (Binary)': [f1_binary],
        'F1-Score (Macro)': [f1_macro],
        'Precision': [precision_score(y_val, dict_yPred[i])],
        'Recall': [recall_score(y_val, dict_yPred[i])],
        'AUPRC': [auc_score]
    })
    results_df = pd.concat([results_df, temp_df], ignore_index = True)

# Grab the best performing hyperparameters if I had refit based on Average Precision (AUPRC?).
DT_search_results = pd.DataFrame(grid_search_DT.cv_results_);
DT_standard_search_results = pd.DataFrame(grid_search_DT_stand.cv_results_);
DT_mm_search_results = pd.DataFrame(grid_search_DT_mm.cv_results_)

DT_averagePrecision = {};
DT_averagePrecision['No Scaler'] = DT_search_results;
DT_averagePrecision['Standard Scaler'] = DT_standard_search_results;
DT_averagePrecision['MinMax Scaler'] = DT_mm_search_results;

X_scales = {};
X_scales['No Scaler'] = X;
X_scales['Standard Scaler'] = X_standard;
X_scales['MinMax Scaler'] = X_min_max;

X_val_scales = {};
X_val_scales['No Scaler'] = X_val;
X_val_scales['Standard Scaler'] = X_val_standard;
X_val_scales['MinMax Scaler'] = X_val_min_max;

# Generate results_df for best performing hyperparameters based on Average Precision.
for i in list(DT_averagePrecision.keys()):
    # Search for best performing Average Precision Model.
    best_row = DT_averagePrecision[i][DT_averagePrecision[i]['rank_test_Average_Precision'] == 1]
    # Scrape hyperparameters.
    best_params = {
    'criterion': best_row['param_criterion'].values[0],
    'max_depth': best_row['param_max_depth'].values[0],
    'min_samples_split': best_row['param_min_samples_split'].values[0],
    'min_samples_leaf': best_row['param_min_samples_leaf'].values[0],
    'max_features': best_row['param_max_features'].values[0]
    };
```

```
    # Generate model.
    best_model_DT = DecisionTreeClassifier(random_state=42, **best_params)
    best_model_DT.fit(X_scales[i], y); # Fit using the scaled X and the actual y.
    best_y_pred_DT = best_model_DT.predict(X_val_scales[i]) # Predict using the scaled y.
    cm = confusion_matrix(y_val,best_y_pred_DT); # Generate confusion matrix.
    tn, fp, fn, tp = cm.ravel(); # Get relevant values.
    precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_DT) # Precision, Recall
    auc_score = auc(recall1, precision1) # AUPRC
    f1_binary = f1_score(y_val, best_y_pred_DT, average='binary') # F1-binary
    f1_macro = f1_score(y_val, best_y_pred_DT, average='macro') # F1-macro
    temp_df = pd.DataFrame({
        'Model': [f'Decision Tree ({i}), refit using Average Precision'],
        'True Negative': [tn],
        'False Positive': [fp],
        'False Negative': [fn],
        'True Positive': [tp],
        'F1-Score (Binary)': [f1_binary],
        'F1-Score (Macro)': [f1_macro],
        'Precision': [precision_score(y_val, best_y_pred_DT)],
        'Recall': [recall_score(y_val, best_y_pred_DT)],
        'AUPRC': [auc_score]
    })
    results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% Random Forest: Many Hyperparameters. (GridSearch v1)

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
```

```
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

param_searchSpace_RF = {'n_estimators':[100,500,1000,2000],
                        'max_depth': [None,10,30,100],
                        'max_samples': [1.0],
                        'min_samples_split':[2,5,15],
                        'min_samples_leaf':[1,3,5],
                        'criterion':['gini','entropy'],
                        'bootstrap':[True,False],
                        'max_features': ['sqrt','log2']};

# TIMER:
startTime = time.time();

print('\nStarting hyperparameter optimization! Wish me luck!')
modelRF =
RandomForestClassifier(n_estimators=800,criterion='gini',max_depth=6,max_samples=1.0,min_samples_split=2,

min_samples_leaf=1,max_features=0.15,max_leaf_nodes=None,min_impurity_decrease=0.0,
                                    bootstrap=True,oob_score=False,n_jobs=-
1,class_weight='balanced',verbose=1)

# GridsearchCV setup:
grid_search_RF =
GridSearchCV(estimator=RandomForestClassifier(max_leaf_nodes=None,min_impurity_decrease=0.0,oob_score=Fals
e,class_weight='balanced',random_state=42),
                                param_grid=param_searchSpace_RF,cv=5, scoring=scoring, refit = 'F1-score',
n_jobs=-1)

# Perform the grid search model generation
grid_search_RF.fit(X,y);
# Evaluate the best model on the validation data.
best_model_RF = grid_search_RF.best_estimator_
y_pred_val_RF = best_model_RF.predict(X_val);
# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))
# Reports
report_RF = classification_report(y_val,y_pred_val_RF,zero_division=0);
print('Classification report: Random Forest Classifier (GridSearch 7):\n', report_RF);

cm_RF = confusion_matrix(y_val,y_pred_val_RF)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_RF, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Random Forest (Gridsearch v7)')
plt.show()

df_RF1 = pd.DataFrame(grid_search_RF.cv_results_)
df_RF1['gsearch'] = 1;
f = open('scriptOutputs/20240307-
script_SpartanComp/Hyperparameter_for_{}_RandomForest_result_opt_v7.txt'.format( feat_type ), 'w' )
f.write("Hyperparameter tuning for RandomForest (GridSearch v7):\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_RF.best_params_)
f.write("The best_score is %s \n" % grid_search_RF.best_score_)
f.write("\n")
modelRF.set_params(**grid_search_RF.best_params_)
f.write("The optimized model parameters are %s \n" % modelRF.get_params())
f.close()

df_RF1.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_RForest_defaultParam_v1.csv',
index=False)

# See if results_df exists already. If not, create it.
```

```
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_RF);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_RF) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_RF, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_RF, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Random Forest'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_RF)],
    'Recall': [recall_score(y_val, y_pred_val_RF)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

# Generate results_df for best performing hyperparameters based on Average Precision.
df_rf = pd.DataFrame(grid_search_RF.cv_results_);


# Search for best performing Average Precision Model.
best_row_RF = df_rf[df_rf['rank_test_Average_Precision'] == 1];
# Scrape hyperparameters.
# Pulled it out manually
best_paramsRF = {'bootstrap': True, 'criterion': 'entropy', 'max_depth': None, 'max_features': 'sqrt',
'max_samples': 1.0, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 2000};
# Generate model.
best_model_RF =
RandomForestClassifier(random_state=42,max_leaf_nodes=None,min_impurity_decrease=0.0,oob_score=False,class
_weight='balanced', **best_paramsRF)
best_model_RF.fit(X, y); # Fit using the scaled X and the actual y.
best_y_pred_RF = best_model_RF.predict(X_val) # Predict using X
cm = confusion_matrix(y_val,best_y_pred_RF); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_RF) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, best_y_pred_RF, average='binary') # F1-binary
f1_macro = f1_score(y_val, best_y_pred_RF, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': [f'Random Forest ({i}), refit using Average Precision'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, best_y_pred_RF)],
    'Recall': [recall_score(y_val, best_y_pred_RF)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% Support Vector Machine (GridSearch v2)

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
```

```
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

svm_model = SVC();

param_grid = {
    'C': [0.1, 1, 10,100,200,300,400,500,1000,1500,2000],        # Regularization parameter
    'kernel': ['linear', 'poly', 'rbf'],  # Kernel type
    'gamma': ['scale', 'auto', 0.1, 1,10],  # Kernel coefficient
}

# TIMER:
startTime = time.time();

print('\nStarting hyperparameter optimization! Wish me luck!')

grid_search_SVM = GridSearchCV(estimator=svm_model, param_grid=param_grid, scoring = scoring, refit = 'F1-
score',cv=5,n_jobs=-1)
grid_search_SVM.fit(X,y); # fixed to X,y in v3_2
best_svm_model = grid_search_SVM.best_estimator_
y_pred_val_SVM = best_svm_model.predict(X_val);

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))

# Reports
report_SVM = classification_report(y_val,y_pred_val_SVM,zero_division=0);
print('Classification report: SVM (Gridsearch v3_2)):\n', report_SVM);
```

```
cm_SVM = confusion_matrix(y_val,y_pred_val_SVM)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_SVM, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: SVM (Gridsearch v3)')
plt.show()

# Hyperparameter optimization (individual results)
df_SVM = pd.DataFrame(grid_search_SVM.cv_results_)
df_SVM['gsearch'] = 2;
f = open('scriptOutputs/20240307-script_SpartanComp/Hyperparameter_for_{}_SVM_result_opt_v1.txt'.format(
feat_type ), 'w' )
f.write("Hyperparameter tuning for SVM:\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_SVM.best_params_)
f.write("The best_score is %s \n" % grid_search_SVM.best_score_)
f.write("\n")
svm_model.set_params(**grid_search_SVM.best_params_)
f.write("The optimized model parameters are %s \n" % svm_model.get_params())
f.close()

df_SVM.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_SVM_defaultParam_v1.csv', index=False)

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_SVM);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_SVM) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_SVM, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_SVM, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['SVM'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_SVM)],
    'Recall': [recall_score(y_val, y_pred_val_SVM)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)


# Generate results_df for best performing hyperparameters based on Average Precision.
df_SVM = pd.DataFrame(grid_search_SVM.cv_results_);


# Search for best performing Average Precision Model.
best_row_SVM = df_SVM[df_SVM['rank_test_Average_Precision'] == 1];
# Scrape hyperparameters.
# Pulled it out manually
best_paramsSVM = {'C': 10, 'gamma': 'scale', 'kernel': 'poly'};
# Generate model.
best_model_SVM = SVC(random_state=42,**best_paramsSVM)
best_model_SVM.fit(X, y); # Fit using the scaled X and the actual y.
best_y_pred_SVM = best_model_SVM.predict(X_val) # Predict using X
cm = confusion_matrix(y_val,best_y_pred_SVM); # Generate confusion matrix.
```

```
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_SVM) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, best_y_pred_SVM, average='binary') # F1-binary
f1_macro = f1_score(y_val, best_y_pred_SVM, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['SVM, refit using Average Precision'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, best_y_pred_SVM)],
    'Recall': [recall_score(y_val, best_y_pred_SVM)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% KNN: GridSearch v4

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

knn_model = KNeighborsClassifier();

param_grid = {
    'n_neighbors': [2,3,4,5,6,7,8,9,10],  # Number of neighbors to consider
```

```
    'weights': ['uniform', 'distance'],  # Weighting of neighbors
    'p': [1, 2],  # Minkowski distance metric (1 for Manhattan, 2 for Euclidean)
}

# TIMER:
startTime = time.time();

print('\nStarting hyperparameter optimization! Wish me luck!')

grid_search_KNN = GridSearchCV(estimator=knn_model, param_grid=param_grid, scoring = scoring, refit = 'F1-
score',cv=5,n_jobs=-1)
grid_search_KNN.fit(X,y);
best_KNN_model = grid_search_KNN.best_estimator_
y_pred_val_KNN = best_KNN_model.predict(X_val);

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))

# Reports
report_KNN = classification_report(y_val,y_pred_val_KNN,zero_division=0);
print('Classification report: KNN (Gridsearch v1):\n', report_KNN);

cm_KNN = confusion_matrix(y_val,y_pred_val_KNN)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_KNN, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: KNN (Gridsearch v1)')
plt.show()

# Hyperparameter optimization (individual results)
df_KNN = pd.DataFrame(grid_search_KNN.cv_results_)
df_KNN['gsearch'] = 4;
f = open('scriptOutputs/20240307-script_SpartanComp/Hyperparameter_for_{}_KNN_result_opt_v1.txt'.format(
feat_type ), 'w' )
f.write("Hyperparameter tuning for KNN:\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_KNN.best_params_)
f.write("The best_score is %s \n" % grid_search_KNN.best_score_)
f.write("\n")
knn_model.set_params(**grid_search_KNN.best_params_)
f.write("The optimized model parameters are %s \n" % knn_model.get_params())
f.close()

df_KNN.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_KNN_defaultParam_v1.csv', index=False)

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_KNN);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_KNN) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_KNN, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_KNN, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['KNN'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
```

```
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_KNN)],
    'Recall': [recall_score(y_val, y_pred_val_KNN)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)


# Generate results_df for best performing hyperparameters based on Average Precision.
df_KNN = pd.DataFrame(grid_search_KNN.cv_results_);


# Search for best performing Average Precision Model.
best_row_KNN = df_KNN[df_KNN['rank_test_Average_Precision'] == 1];
# Scrape hyperparameters.
# Pulled it out manually
best_paramsKNN = {'n_neighbors': 10, 'p': 2, 'weights': 'distance'};
# Generate model.
best_model_KNN = KNeighborsClassifier(**best_paramsSVM)
best_model_KNN.fit(X, y); # Fit using the scaled X and the actual y.
best_y_pred_KNN = best_model_KNN.predict(X_val) # Predict using X
cm = confusion_matrix(y_val,best_y_pred_KNN); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_KNN) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, best_y_pred_KNN, average='binary') # F1-binary
f1_macro = f1_score(y_val, best_y_pred_KNN, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['KNN, refit using Average Precision'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, best_y_pred_KNN)],
    'Recall': [recall_score(y_val, best_y_pred_KNN)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Naive Bayes (GridSearch v5)

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
```

```
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

nb_model = GaussianNB();

print('\nNo Hyperparameters for Gaussian Naive Bayes')

nb_model.fit(X,y);
y_pred_val_nb = nb_model.predict(X_val);
# Histogram
y_pred_val_nb_range = nb_model.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_nb_range)

# Reports
report_nb = classification_report(y_val,y_pred_val_nb,zero_division=0);
print('Classification report: Naive Bayes (Gridsearch v5):\n', report_nb);

cm_nb = confusion_matrix(y_val,y_pred_val_nb)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_nb, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Naive Bayes (Gridsearch v5)')
plt.show()

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_nb);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_nb) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_nb, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_nb, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Gaussian Naive Bayes'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_nb)],
    'Recall': [recall_score(y_val, y_pred_val_nb)],
    'AUPRC': [auc_score]
})
```

```
results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% Naive Bayes (Multinomial, Complement, and Bernoulli):

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

# Multinomial Naive Bayes:
mb_model = MultinomialNB();
mb_model.fit(X,y);
y_pred_val_mb = mb_model.predict(X_val);
y_pred_val_mb_range = mb_model.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_mb_range); # Check value range.
# Reports (Multinomial Naive Bayes)
report_mb = classification_report(y_val,y_pred_val_mb,zero_division=0);
print('Classification report: Multinomial Naive Bayes (Gridsearch v6):\n', report_mb);

cm_mb = confusion_matrix(y_val,y_pred_val_mb)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_mb, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Multinomial Naive Bayes (Gridsearch v6)')
plt.show()

# Complement Naive Bayes:
```

```
param_grid = {
    'alpha': [0.1, 0.5, 1.0],  # Values for Laplace smoothing
    'fit_prior': [True, False],  # Whether to learn class priors
    'class_prior': [None, [0.3, 0.7]],  # Prior probabilities of classes (if fit_prior is False)
    'norm': [True, False]  # Whether to normalize term counts
}
startTime = time.time();
print('\nStarting hyperparameter optimization! Wish me luck!')
cnb_model = ComplementNB()
grid_search_cnb = GridSearchCV(estimator=cnb_model, param_grid=param_grid, scoring = scoring, refit = 'F1-
score',cv=5,n_jobs=-1)
grid_search_cnb.fit(X,y);
best_cnb_model = grid_search_cnb.best_estimator_
y_pred_val_cnb = best_cnb_model.predict(X_val);
y_pred_val_cnb_range = best_cnb_model.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_cnb_range)

#Plot the histogram of predicted probabilities
plt.figure(figsize=(10, 6))
sns.histplot(y_pred_val_cnb_range, bins=50, kde=True, color='blue', label='Predicted Probabilities')

# Overlay the histogram with vertical lines at the actual class values (0 or 1)
plt.axvline(x=y_pred_val_cnb_range[y_val == 0].mean(), color='red', linestyle='dashed', linewidth=2,
label='Actual 0 Mean')
plt.axvline(x=y_pred_val_cnb_range[y_val == 1].mean(), color='green', linestyle='dashed', linewidth=2,
label='Actual 1 Mean')

# Add labels and legend
plt.xlabel('Predicted Probabilities')
plt.ylabel('Frequency')
plt.title('Histogram of Predicted Probabilities with Actual Class Means')
plt.legend()

# Show the plot
plt.show()

# THIS PLOT SHOWS CONFUSION MATRIX AS A FUNCTION OF THRESHOLD

# Define threshold values
# thresholds = np.arange(0.499, 0.511, 0.001)
thresholds = np.linspace(min(y_pred_val_cnb_range), max(y_pred_val_cnb_range), 100)

# Initialize lists to store results
tn_percentages = []
fp_percentages = []
fn_percentages = []
tp_percentages = []

# Calculate metrics for each threshold
for threshold in thresholds:
    # Convert probabilities to binary predictions based on the threshold
    y_pred_binary = (y_pred_val_cnb_range > threshold).astype(int)

    # Calculate confusion matrix
    conf_matrix = confusion_matrix(y_val, y_pred_binary)

    # Calculate percentages
    total_samples = len(y_val)
    tn_percentages.append(conf_matrix[0, 0] / total_samples * 100)
    fp_percentages.append(conf_matrix[0, 1] / total_samples * 100)
    fn_percentages.append(conf_matrix[1, 0] / total_samples * 100)
    tp_percentages.append(conf_matrix[1, 1] / total_samples * 100)

# Plot the bar graph
plt.figure(figsize=(10, 6))
plt.bar(thresholds, tn_percentages, width=0.002, label='True Negatives')
plt.bar(thresholds, fp_percentages, width=0.002, label='False Positives', bottom=tn_percentages)
```

```
plt.bar(thresholds, fn_percentages, width=0.002, label='False Negatives', bottom=np.array(tn_percentages)
+ np.array(fp_percentages))
plt.bar(thresholds, tp_percentages, width=0.002, label='True Positives', bottom=np.array(tn_percentages) +
np.array(fp_percentages) + np.array(fn_percentages))

plt.xlabel('Threshold')
plt.ylabel('Percentage')
plt.title('Percentage of Confusion Matrix Elements as a Function of Threshold')
plt.legend()
plt.show()

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))

# Reports (CNB)
report_cnb = classification_report(y_val,y_pred_val_cnb,zero_division=0);
print('Classification report: Complement Naive Bayes (Gridsearch v6):\n', report_cnb);

cm_cnb = confusion_matrix(y_val,y_pred_val_cnb)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_cnb, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Complement Naive Bayes (Gridsearch v6)')
plt.show()

df_cnb = pd.DataFrame(grid_search_cnb.cv_results_)
df_cnb['gsearch'] = 6;
f = open('scriptOutputs/20240307-
script_SpartanComp/Hyperparameter_for_{}_CNBayes_result_opt_v1.txt'.format( feat_type ), 'w' )
f.write("Hyperparameter tuning for CNBayes:\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_cnb.best_params_)
f.write("The best_score is %s \n" % grid_search_cnb.best_score_)
f.write("\n")
cnb_model.set_params(**grid_search_cnb.best_params_)
f.write("The optimized model parameters are %s \n" % cnb_model.get_params())
f.close()

df_cnb.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_CNBayes_defaultParam_v1.csv',
index=False)

# Bernoulli Naive Bayes:

param_grid = {
    'alpha': [0.1, 0.5, 1.0],  # Values for Laplace smoothing
    'binarize': [0.0, 0.2, 0.5],  # Thresholds for binarizing features
    'fit_prior': [True, False],  # Whether to learn class priors
    'class_prior': [None, [0.3, 0.7]],  # Prior probabilities of classes (if fit_prior is False)
}
startTime = time.time();
print('\nStarting hyperparameter optimization! Wish me luck!')
bnb_model = BernoulliNB()
grid_search_bnb = GridSearchCV(estimator=bnb_model, param_grid=param_grid, scoring = scoring, refit = 'F1-
score',cv=5,n_jobs=-1)
grid_search_bnb.fit(X,y);
best_bnb_model = grid_search_bnb.best_estimator_
y_pred_val_bnb = best_bnb_model.predict(X_val);
y_pred_val_bnb_range = best_bnb_model.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_bnb_range)

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
```

```
print('\nFinished this hyperparameter optimization in {}'.format(t_str))

# Reports (CNB)
report_bnb = classification_report(y_val,y_pred_val_bnb,zero_division=0);
print('Classification report: Bernoulli Naive Bayes (Gridsearch v6):\n', report_bnb);

cm_bnb = confusion_matrix(y_val,y_pred_val_bnb)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_bnb, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: Bernoulli Naive Bayes (Gridsearch v6)')
plt.show()

df_bnb = pd.DataFrame(grid_search_bnb.cv_results_)
df_bnb['gsearch'] = 6_1;
f = open('scriptOutputs/20240307-
script_SpartanComp/Hyperparameter_for_{}_BNBayes_result_opt_v1.txt'.format( feat_type ), 'w' )
f.write("Hyperparameter tuning for BNBayes:\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_bnb.best_params_)
f.write("The best_score is %s \n" % grid_search_bnb.best_score_)
f.write("\n")
bnb_model.set_params(**grid_search_bnb.best_params_)
f.write("The optimized model parameters are %s \n" % bnb_model.get_params())
f.close()

df_bnb.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_BNBayes_defaultParam_v1.csv',
index=False)


# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

y_pred_NaiveBayes = {};
y_pred_NaiveBayes['Multinomial'] = y_pred_val_mb
y_pred_NaiveBayes['Complement'] = y_pred_val_cnb
y_pred_NaiveBayes['Bernoulli'] = y_pred_val_bnb

for i in list(y_pred_NaiveBayes.keys()):
    # Get model results (evaluation metrics)
    cm = confusion_matrix(y_val, y_pred_NaiveBayes[i]);
    tn, fp, fn, tp = cm.ravel();
    precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_NaiveBayes[i]) # Precision,
Recall
    auc_score = auc(recall1, precision1) # AUPRC
    f1_binary = f1_score(y_val, y_pred_NaiveBayes[i], average='binary') # F1-binary
    f1_macro = f1_score(y_val, y_pred_NaiveBayes[i], average='macro') # F1-macro
    temp_df = pd.DataFrame({
        'Model': [i + ' Naive Bayes'],
        'True Negative': [tn],
        'False Positive': [fp],
        'False Negative': [fn],
        'True Positive': [tp],
        'F1-Score (Binary)': [f1_binary],
        'F1-Score (Macro)': [f1_macro],
        'Precision': [precision_score(y_val, y_pred_NaiveBayes[i])],
        'Recall': [recall_score(y_val, y_pred_NaiveBayes[i])],
        'AUPRC': [auc_score]
    })
    results_df = pd.concat([results_df, temp_df], ignore_index = True)

# Generate results_df for best performing hyperparameters based on Average Precision.
df_CNB = pd.DataFrame(grid_search_cnb.cv_results_);
```

```
# Search for best performing Average Precision Model.
best_row_CNB = df_CNB[df_CNB['rank_test_Average_Precision'] == 1];
# Scrape hyperparameters.
# Pulled it out manually
best_paramsCNB = {'alpha': 0.1, 'class_prior': None, 'fit_prior': True, 'norm': True};
# Generate model.
best_model_CNB = ComplementNB(**best_paramsCNB)
best_model_CNB.fit(X, y); # Fit using the scaled X and the actual y.
best_y_pred_CNB = best_model_CNB.predict(X_val) # Predict using X
y_pred_val_CNB_range_AP = best_model_CNB.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_CNB_range_AP)
cm = confusion_matrix(y_val,best_y_pred_CNB); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_CNB) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, best_y_pred_CNB, average='binary') # F1-binary
f1_macro = f1_score(y_val, best_y_pred_CNB, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Complement Naive Bayes, refit using Average Precision'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, best_y_pred_CNB)],
    'Recall': [recall_score(y_val, best_y_pred_CNB)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

thresholds = np.linspace(min(y_pred_val_CNB_range_AP), max(y_pred_val_CNB_range_AP), 100)

# Initialize lists to store results
tn_percentages = []
fp_percentages = []
fn_percentages = []
tp_percentages = []

# Calculate metrics for each threshold
for threshold in thresholds:
    # Convert probabilities to binary predictions based on the threshold
    y_pred_binary = (y_pred_val_CNB_range_AP > threshold).astype(int)

    # Calculate confusion matrix
    conf_matrix = confusion_matrix(y_val, y_pred_binary)

    # Calculate percentages
    total_samples = len(y_val)
    tn_percentages.append(conf_matrix[0, 0] / total_samples * 100)
    fp_percentages.append(conf_matrix[0, 1] / total_samples * 100)
    fn_percentages.append(conf_matrix[1, 0] / total_samples * 100)
    tp_percentages.append(conf_matrix[1, 1] / total_samples * 100)

# Plot the bar graph
plt.figure(figsize=(10, 6))
plt.bar(thresholds, tn_percentages, width=0.002, label='True Negatives')
plt.bar(thresholds, fp_percentages, width=0.002, label='False Positives', bottom=tn_percentages)
plt.bar(thresholds, fn_percentages, width=0.002, label='False Negatives', bottom=np.array(tn_percentages)
+ np.array(fp_percentages))
plt.bar(thresholds, tp_percentages, width=0.002, label='True Positives', bottom=np.array(tn_percentages) +
np.array(fp_percentages) + np.array(fn_percentages))

plt.xlabel('Threshold')
plt.ylabel('Percentage')
plt.title('Percentage of Confusion Matrix Elements as a Function of Threshold')
plt.legend()
```

```
plt.show()


#
df_BNB = pd.DataFrame(grid_search_bnb.cv_results_);


# Search for best performing Average Precision Model.
best_row_BNB = df_BNB[df_BNB['rank_test_Average_Precision'] == 1];
# Scrape hyperparameters.
# Pulled it out manually
best_paramsBNB = {'alpha': 0.1, 'binarize': 0.5, 'class_prior': None, 'fit_prior': True};
# Generate model.
best_model_BNB = BernoulliNB(**best_paramsBNB)
best_model_BNB.fit(X, y); # Fit using the scaled X and the actual y.
best_y_pred_BNB = best_model_BNB.predict(X_val) # Predict using X
y_pred_val_BNB_range_AP = best_model_BNB.predict_proba(X_val)[:,1]
sns.histplot(y_pred_val_BNB_range_AP)
cm = confusion_matrix(y_val,best_y_pred_BNB); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_val,best_y_pred_BNB) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, best_y_pred_BNB, average='binary') # F1-binary
f1_macro = f1_score(y_val, best_y_pred_BNB, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Bernoulli Naive Bayes, refit using Average Precision'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, best_y_pred_BNB)],
    'Recall': [recall_score(y_val, best_y_pred_BNB)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Gradient Boost (XGBoost):

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
```

```
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

param_searchSpace_XGB = {'n_estimators': [800, 1000, 1200, 1600],
                         'max_depth': [6, 8, 10],
                         'learning_rate': [0.01, 0.1, 0.2],
                         'reg_alpha': [0, 0.1, 0.5],
                         'reg_lambda': [0,0.1, 0.5]}

modelXGB = xgb.XGBClassifier(n_estimators=800,
                         max_depth=6,
                         learning_rate=0.01,
                         objective='binary:logistic',
                         booster='gbtree',
                         eval_metric='logloss',  # Set the appropriate evaluation metric
                         n_jobs=-1,
                         verbosity=2)

startTime = time.time()
print('Starting hyper parameter optimization...')

grid_search_XGB = GridSearchCV(estimator=modelXGB, param_grid=param_searchSpace_XGB,
scoring=scoring,refit='F1-score', cv=5, n_jobs=-1)
grid_search_XGB.fit(X, y)
best_XGB_model = grid_search_XGB.best_estimator_
y_pred_val_XGB = best_XGB_model.predict(X_val);

# Timer
endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished this hyperparameter optimization in {}'.format(t_str))

# Reports (CNB)
report_XGB = classification_report(y_val,y_pred_val_XGB,zero_division=0);
print('Classification report: XGBoost (Gridsearch v1):\n', report_XGB);

cm_XGB = confusion_matrix(y_val,y_pred_val_XGB)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_XGB, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: XGBoost (Gridsearch v1)')
plt.show()

df_XGB = pd.DataFrame(grid_search_XGB.cv_results_)
df_XGB['gsearch'] = 7;
f = open('scriptOutputs/20240307-
script_SpartanComp/Hyperparameter_for_{}_XGBoost_result_opt_v7.txt'.format( feat_type ), 'w' )
f.write("Hyperparameter tuning for XGBoost:\n")
f.write("Tuned several hyperparameters\n")
f.write("The best_params are %s \n" % grid_search_XGB.best_params_)
f.write("The best_score is %s \n" % grid_search_XGB.best_score_)
f.write("\n")
modelXGB.set_params(**grid_search_XGB.best_params_)
f.write("The optimized model parameters are %s \n" % modelXGB.get_params())
```

```
f.close()

df_XGB.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_XGBoost_defaultParam_v7.csv',
index=False)

#%% XGBoost with optimized parameters (computer broke):

modelXGB = xgb.XGBClassifier(n_estimators=800,
                          max_depth=6,
                          learning_rate=0.02,
                          objective='binary:logistic',
                          booster='gbtree',
                          reg_alpha =0,
                          reg_lambda = 0,
                          eval_metric='logloss',  # Set the appropriate evaluation metric
                          n_jobs=-1,
                          verbosity=2)
xgbParams_F1 = {'learning_rate': 0.2, 'max_depth': 6, 'n_estimators': 800, 'reg_alpha': 0, 'reg_lambda':
0}

modelXGB = xgb.XGBClassifier(objective='binary:logistic', eval_metric='logloss', booster='gbtree',
**xgbParams_F1)
modelXGB.fit(X,y);
y_pred_val_XGB = modelXGB.predict(X_val);
# Reports (CNB)
report_XGB = classification_report(y_val,y_pred_val_XGB,zero_division=0);
print('Classification report: XGBoost (Gridsearch v7):\n', report_XGB);

cm_XGB = confusion_matrix(y_val,y_pred_val_XGB)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_XGB, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: XGBoost (Gridsearch v7)')
plt.show()

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_XGB);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_XGB) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_XGB, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_XGB, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['XGBoost (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_XGB)],
    'Recall': [recall_score(y_val, y_pred_val_XGB)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

# Based on Average Precision

xgbParams_AP = {'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 1200, 'reg_alpha': 0.1,
'reg_lambda': 0};
```

```
modelXGB_AP = xgb.XGBClassifier(objective='binary:logistic', eval_metric='logloss', booster='gbtree',
**xgbParams_AP)
modelXGB_AP.fit(X,y);
y_pred_val_XGB_AP = modelXGB_AP.predict(X_val);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_XGB_AP);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_XGB_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_XGB_AP, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_XGB_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['XGBoost (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_XGB_AP)],
    'Recall': [recall_score(y_val, y_pred_val_XGB_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% LightGBM

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
```

```
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

# param_base_LGB = {'n_estimators': 500, 'subsample':0.8, 'colsample_bytree':0.8,'subsample_freq':1}
# param_search1_LGB = {'n_estimators':[100, 250, 500, 750, 1000]} # 5
# param_search2_LGB = {'num_leaves':[15, 31, 45, 60, 75],'min_child_samples':[10, 20, 30, 40]}  # 5* 4 =
20
# param_search3_LGB = {'subsample':[0.6, 0.7, 0.8, 0.9, 1.0], 'colsample_bytree':[0.6, 0.7, 0.8, 0.9,
1.0], 'subsample_freq': [0,1,3,5]} # 5*5*4 =100
# param_search4_LGB = {'reg_alpha':[0, 0.2, 0.5, 0.8], 'reg_lambda':[0, 0.2, 0.5, 0.8]}  # 4*4 = 16

# param_search_LGB = {'n_estimators':[100, 250, 500, 750, 1000],
#                       'num_leaves':[15, 31, 45, 60, 75],'min_child_samples':[10, 20, 30, 40],
#                       'subsample':[0.6, 0.7, 0.8, 0.9, 1.0],'colsample_bytree':[0.6, 0.7, 0.8, 0.9,
1.0],'subsample_freq': [0,1,3,5],
#                       'reg_alpha':[0, 0.2, 0.5, 0.8], 'reg_lambda':[0, 0.2, 0.5, 0.8]}

param_search_LGB = {'n_estimators':[100,1500,2000,2500],
                     'num_leaves':[15,30,50],'min_child_samples':[210, 1000,2000],
                     'subsample':[0.6],'colsample_bytree':[1.0],'subsample_freq': [5],
                     'reg_alpha':[0,1], 'reg_lambda':[0.8,1]}
startTime = time.time();
print('Starting hyper parameter optimization...')
# IA: the lightGBM documents say you shouldn't use class_weight and is_unbalance at the same time??
# model = lgb.LGBMClassifier( n_estimators=500, objective='binary', is_unbalance='true',
class_weight='balanced', subsample=0.8, colsample_bytree=0.8, subsample_freq=1, n_jobs=20 )
model = lgb.LGBMClassifier( n_estimators=500, objective='binary', is_unbalance='true', subsample=0.8,
colsample_bytree=0.8, subsample_freq=1, n_jobs=-1 )

# Sequential Grid search with stratified 5-fold cross-validation
# will use stratified k-fold by default if integer supplied to 'cv' argument
gsearch1 = GridSearchCV( estimator=model, param_grid=param_search_LGB, scoring=scoring, refit='F1-score',
cv=5, n_jobs=-1, verbose=1 )
#gsearch1 = GridSearchCV( estimator=model, param_grid=param_search1_LGB, scoring='f1', cv=3, n_jobs=20 )
gsearch1.fit(X, y)
df1 = pd.DataFrame( gsearch1.cv_results_ )
df1['gsearch'] = 8
f = open( 'scriptOutputs/20240307-script_SpartanComp/Hyperparameter_for_{}_LightGBM_opt_v2.txt'.format(
feat_type ), 'w' )
f.write("Hyperparameter tuning for LightGBM:\n")
f.write("The 1st round of tuning n_estimators\n")
f.write("The best_params are %s \n" % gsearch1.best_params_)
f.write("The best_score is %s \n" % gsearch1.best_score_)
f.write("\n")

model.set_params(**gsearch1.best_params_)

f.write("The optimized model parameters are %s \n" % model.get_params())
f.close()

# df = pd.concat( [df1, df2, df3, df4], axis=0 )
df1.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_LGBM_v2.csv', index=False )

endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished hyperparameter optimization in {}'.format(t_str))

best_LGBM_model = gsearch1.best_estimator_;
y_pred_val_LGBM = best_LGBM_model.predict(X_val)

# Reports (CNB)
```

```
report_LGBM = classification_report(y_val,y_pred_val_LGBM,zero_division=0);
print('Classification report: LightGBM (Gridsearch v2):\n', report_LGBM);

cm_LBGM = confusion_matrix(y_val,y_pred_val_LGBM)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_LBGM, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: LightGBM (Gridsearch v12)')
plt.show()


# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_LGBM);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_LGBM) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_LGBM, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_LGBM, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['LightGBM (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_LGBM)],
    'Recall': [recall_score(y_val, y_pred_val_LGBM)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)



# Based on Average Precision

lgbParams_AP = {'colsample_bytree': 1.0, 'min_child_samples': 210, 'n_estimators': 2000, 'num_leaves': 15,
'reg_alpha': 0, 'reg_lambda': 1, 'subsample': 0.6, 'subsample_freq': 5};
modelLGB_AP = lgb.LGBMClassifier(objective='binary', is_unbalance='true', **lgbParams_AP)
modelLGB_AP.fit(X,y);
y_pred_val_LGB_AP = modelLGB_AP.predict(X_val);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_LGB_AP);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_LGB_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_LGB_AP, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_LGB_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['LightGBM (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
```

```
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_LGB_AP)],
    'Recall': [recall_score(y_val, y_pred_val_LGB_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% MLPClassifier

### Load the training data:
## LOAD DATA
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()

# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values

# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load the validation data
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

# Define a grid of hyperparameters to search over
param_grid = {
    'hidden_layer_sizes': [(100,),(256,128),(256,64),(256,32),(256,16),(64, 32), (128, 64), (32, 16),
(16,8)],
    'activation': ['relu', 'tanh','logistic'],
    'solver': ['adam', 'sgd', 'lbfgs'],
    'alpha': [0.0001, 0.001, 0.01, 0.1],
}

param_grid2 = {
    'hidden_layer_sizes': [(512,), (1024,), (1536,), (2048,), (256,256), (512,512), (1024,1024),
(1536,1536), (2048,2048)],
    'activation': ['relu', 'tanh','logistic'],
```

```
    'solver': ['adam', 'sgd', 'lbfgs'],
    'alpha': [0.0001, 0.001, 0.01, 0.1],
    'early_stopping': [True],
}
param_grid = param_grid2;

startTime = time.time();
print('Starting hyper parameter optimization...')
# Create GridSearchCV
clf = MLPClassifier(random_state=42, max_iter=2000)
gsearch1 = GridSearchCV(clf, param_grid, cv=5, scoring=scoring, refit='F1-score', n_jobs=-1,verbose=2)

# Fit the model to the training data
gsearch1.fit(X, y)

df1 = pd.DataFrame(gsearch1.cv_results_ )
df1['gsearch'] = 13
f = open( 'scriptOutputs/20240307-script_SpartanComp/Hyperparameter_for_{}_MLPClass_opt_v2.txt'.format(
feat_type ), 'w' )
f.write("Hyperparameter tuning for MLPClassifier:\n")
f.write("The 1st round of tuning n_estimators\n")
f.write("The best_params are %s \n" % gsearch1.best_params_)
f.write("The best_score is %s \n" % gsearch1.best_score_)
f.write("\n")

clf.set_params(**gsearch1.best_params_)

f.write("The optimized model parameters are %s \n" % clf.get_params())
f.close()

# df = pd.concat( [df1, df2, df3, df4], axis=0 )
df1.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_MLPClass_v2.csv', index=False )

endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished hyperparameter optimization in {}'.format(t_str))

best_MLPC_model = gsearch1.best_estimator_;
y_pred_val_MLPC = best_MLPC_model.predict(X_val)

# Reports (CNB)
report_MLPC = classification_report(y_val,y_pred_val_MLPC,zero_division=0);
print('Classification report: MLPClassifier (Gridsearch v2):\n', report_MLPC);

cm_MLPC = confusion_matrix(y_val,y_pred_val_MLPC)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_MLPC, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: MLPClassifier (Gridsearch v2)')
plt.show()

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_MLPC);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_MLPC) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_MLPC, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_MLPC, average='macro') # F1-macro
temp_df = pd.DataFrame({
```

```
    'Model': ['MLPClassifier (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_MLPC)],
    'Recall': [recall_score(y_val, y_pred_val_MLPC)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

# Based on Average Precision

# mlpParams_AP = {'activation': 'relu', 'alpha': 0.01, 'hidden_layer_sizes': (32, 16), 'solver': 'adam'};
# modelMLP_AP = MLPClassifier(random_state=42, max_iter=500, **mlpParams_AP)
# modelMLP_AP.fit(X,y);
# y_pred_val_MLP_AP = modelMLP_AP.predict(X_val);

# # See if results_df exists already. If not, create it.
# try:
#     results_df
# except:
#     results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative',
'True Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# # Get model results (evaluation metrics)
# cm = confusion_matrix(y_val, y_pred_val_MLP_AP);
# tn, fp, fn, tp = cm.ravel();
# precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_MLP_AP) # Precision, Recall
# auc_score = auc(recall1, precision1) # AUPRC
# f1_binary = f1_score(y_val, y_pred_val_MLP_AP, average='binary') # F1-binary
# f1_macro = f1_score(y_val, y_pred_val_MLP_AP, average='macro') # F1-macro
# temp_df = pd.DataFrame({
#     'Model': ['MLPClassifier (AP)'],
#     'True Negative': [tn],
#     'False Positive': [fp],
#     'False Negative': [fn],
#     'True Positive': [tp],
#     'F1-Score (Binary)': [f1_binary],
#     'F1-Score (Macro)': [f1_macro],
#     'Precision': [precision_score(y_val, y_pred_val_MLP_AP)],
#     'Recall': [recall_score(y_val, y_pred_val_MLP_AP)],
#     'AUPRC': [auc_score]
# })
# results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% Ridge Classifier:

### Load & process the validation data.
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)
```

```
# Define scoring:
scoring = { "ROCAUC": "roc_auc", "Average_Precision":"average_precision", "F1-score":"f1" }

# Define a grid of hyperparameters to search over
param_grid = {
    'solver': ['auto', 'svd', 'cholesky', 'lsqr','sparse_cg','sag','saga','lbfgs'],
    'alpha': [0.0001, 0.001, 0.01, 0.1, 0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0],
    'class_weight':[None, 'balanced']}

startTime = time.time();
print('Starting hyper parameter optimization...')
# Create GridSearchCV
ridgeCLF = RidgeClassifier(random_state=42, max_iter=2000)
gsearch1 = GridSearchCV(ridgeCLF, param_grid, cv=5, scoring=scoring, refit='F1-score', n_jobs=-
1,verbose=1)

# Fit the model to the training data
gsearch1.fit(X, y)

df1 = pd.DataFrame(gsearch1.cv_results_ )
df1['gsearch'] = 13
f = open( 'scriptOutputs/20240307-script_SpartanComp/Hyperparameter_for_{}_RidgeClass_opt_v1.txt'.format(
feat_type ), 'w' )
f.write("Hyperparameter tuning for RidgeClassifier:\n")
f.write("The 1st round of tuning n_estimators\n")
f.write("The best_params are %s \n" % gsearch1.best_params_)
f.write("The best_score is %s \n" % gsearch1.best_score_)
f.write("\n")

ridgeCLF.set_params(**gsearch1.best_params_)

f.write("The optimized model parameters are %s \n" % ridgeCLF.get_params())
f.close()

# df = pd.concat( [df1, df2, df3, df4], axis=0 )
df1.to_csv('scriptOutputs/20240307-script_SpartanComp/grid_opt_RidgeClass_v1.csv', index=False )

endTime = time.time()
lengthTime = endTime - startTime;
t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
print('\nFinished hyperparameter optimization in {}'.format(t_str))

best_RC_model = gsearch1.best_estimator_;
y_pred_val_RC = best_RC_model.predict(X_val)

# Reports (CNB)
report_RC = classification_report(y_val,y_pred_val_RC,zero_division=0);
print('Classification report: RidgeClassifier (Gridsearch v1):\n', report_RC);

cm_RC = confusion_matrix(y_val,y_pred_val_RC)
plt.figure(figsize=(6, 4))
sns.heatmap(cm_RC, annot=True, fmt='d', cmap='Blues', xticklabels=['Predicted 0', 'Predicted 1'],
yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix: RidgeClassifier (Gridsearch v13)')
plt.show()

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_RC);
tn, fp, fn, tp = cm.ravel();
```

```
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_RC) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_RC, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_RC, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Ridge Classifier (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_RC)],
    'Recall': [recall_score(y_val, y_pred_val_RC)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

rcParams_AP = {'alpha': 1.0, 'class_weight': 'balanced', 'solver': 'auto'}

modelRC_AP = RidgeClassifier(random_state=42, max_iter=2000, **rcParams_AP)
modelRC_AP.fit(X,y);
y_pred_val_RC_AP = modelRC_AP.predict(X_val);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Get model results (evaluation metrics)
cm = confusion_matrix(y_val, y_pred_val_RC_AP);
tn, fp, fn, tp = cm.ravel();
precision1, recall1, threshold1 = precision_recall_curve(y_val, y_pred_val_RC_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_val, y_pred_val_RC_AP, average='binary') # F1-binary
f1_macro = f1_score(y_val, y_pred_val_RC_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Ridge Classifier (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_val, y_pred_val_RC_AP)],
    'Recall': [recall_score(y_val, y_pred_val_RC_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)
```

Code B.2: Binary classification model evaluation on 20% testing data.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 25 20:56:40 2023

@author: alas
"""
#%% Imports

import os
from os import chdir

import numpy as np
import pandas as pd
import seaborn as sns
import sklearn
import pickle
import sys
import lightgbm as lgb
import matplotlib.pyplot as plt
import time
import catboost
import xgboost as xgb


from rdkit import Chem
from rdkit.Chem import Descriptors
from rdkit.Chem import PandasTools
from rdkit.Chem import AllChem

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_validate
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score, average_precision_score
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
confusion_matrix
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.feature_selection import VarianceThreshold,SelectKBest, f_classif, RFECV
from scipy.signal import find_peaks
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB, ComplementNB

from catboost import CatBoostClassifier, Pool

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, average_precision_score, precision_recall_curve, auc,
PrecisionRecallDisplay


#%% Directory

# chdir(r"C:\Users\imraa\Documents\UWM\BugniLab\Tubulin-Binders")
chdir(r"C:\Users\alas\Documents\Python")

results_df = None;

#%% Data Preparation

### Load & process the training data.
```

```
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()
# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values
# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load & process the validation data.
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

### Load & process the testing data
feat_type = 'ms2fp';
# load data, get features
test_df = pd.read_pickle('scriptOutputs/testing_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
test_df = test_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
test_df['CID'] = test_df['CID'].astype(str)
# get molids
molid_list_test = test_df.CID.tolist()
# features
test_col_list = test_df.columns.tolist()
test_feat_cols = [ c for c in test_col_list[2:] if 'ms2fp_' in str(c) ]
X_test = test_df[test_feat_cols].values
# labels
y_test = test_df.Positive.values
# weird issue with int type i
y_test = np.array(y_test, np.int64)

### Group the 60% Training & 20% Validation

X_train = np.concatenate((X, X_val));
y_train = np.concatenate((y,y_val));

#%% Model #5: SVM, with hyperparameters optimized based on Average Precision.

svm_param_AP = {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}
svm_model_AP = SVC(random_state = 42, **svm_param_AP)
svm_model_AP.fit(X_train, y_train);
y_pred_SVM_AP = svm_model_AP.predict(X_test);

# See if results_df exists already. If not, create it.
```

```
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_SVM_AP); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_SVM_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_SVM_AP, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_SVM_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['SVM (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_SVM_AP)],
    'Recall': [recall_score(y_test, y_pred_SVM_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Model #4: SVM, with hyperparameters optimized based on F1.

svm_param_F1 = {'C': 1, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0,
'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'scale', 'kernel': 'linear', 'max_iter': -1,
'probability': False, 'shrinking': True, 'tol': 0.001, 'verbose': False}
svm_model_F1 = SVC(random_state = 42, **svm_param_F1)
svm_model_F1.fit(X_train, y_train);
y_pred_SVM_F1 = svm_model_F1.predict(X_test);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_SVM_F1); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_SVM_F1) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_SVM_F1, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_SVM_F1, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['SVM (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_SVM_F1)],
    'Recall': [recall_score(y_test, y_pred_SVM_F1)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Model #3: LightGBM, with hyperparameters optimized based on F1.

lgb_param_F1 = {'boosting_type': 'gbdt', 'class_weight': None, 'colsample_bytree': 1.0, 'importance_type':
'split', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_samples': 1000, 'min_child_weight': 0.001,
'min_split_gain': 0.0, 'n_estimators': 2500, 'n_jobs': -1, 'num_leaves': 15, 'objective': 'binary',
```

```
'reg_alpha': 1, 'reg_lambda': 0.8, 'silent': 'warn', 'subsample': 0.6, 'subsample_for_bin': 200000,
'subsample_freq': 5, 'is_unbalance': 'true'};
lgb_model_F1 = lgb.LGBMClassifier(random_state=42, **lgb_param_F1)
lgb_model_F1.fit(X_train, y_train);
y_pred_LGB_F1 = lgb_model_F1.predict(X_test);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_LGB_F1); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_LGB_F1) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_LGB_F1, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_LGB_F1, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['LightGBM (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_LGB_F1)],
    'Recall': [recall_score(y_test, y_pred_LGB_F1)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)


#%% Model #2: MLPClassifier, with hyperparameters optimized based on AP.

mlp_param_AP = {'activation': 'relu', 'alpha': 0.01, 'hidden_layer_sizes': (32, 16), 'solver': 'adam'};
mlp_model_AP = MLPClassifier(random_state=42, max_iter=500, **mlp_param_AP)
mlp_model_AP.fit(X_train, y_train);
y_pred_MLP_AP = mlp_model_AP.predict(X_test);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_MLP_AP); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_MLP_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_MLP_AP, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_MLP_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['MLPClassifier (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_MLP_AP)],
    'Recall': [recall_score(y_test, y_pred_MLP_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)
```

```
#%% Model #1: MLPClassifier, with hyperparameters optimized based on F1.

mlp_param_F1 = {'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1': 0.9, 'beta_2':
0.999, 'early_stopping': False, 'epsilon': 1e-08, 'hidden_layer_sizes': (128, 64), 'learning_rate':
'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 500, 'momentum': 0.9,
'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 42, 'shuffle': True,
'solver': 'lbfgs', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
mlp_param_F1_v2 = {'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1': 0.9,
'beta_2': 0.999, 'early_stopping': True, 'epsilon': 1e-08, 'hidden_layer_sizes': (256, 256),
'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 2000, 'momentum':
0.9, 'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 42, 'shuffle':
True, 'solver': 'lbfgs', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
mlp_param_F1 = mlp_param_F1_v2
mlp_model_F1 = MLPClassifier(**mlp_param_F1) # max_iter & random_state already included in mlp_param_F1
mlp_model_F1.fit(X_train, y_train);
y_pred_MLP_F1 = mlp_model_F1.predict(X_test);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_MLP_F1); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_MLP_F1) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_MLP_F1, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_MLP_F1, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['MLPClassifier v2 (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_MLP_F1)],
    'Recall': [recall_score(y_test, y_pred_MLP_F1)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Ridge Classifier (F1)

rc_param_F1 = {'alpha': 0.6, 'class_weight': 'balanced', 'copy_X': True, 'fit_intercept': True,
'max_iter': 2000, 'positive': False, 'random_state': 42, 'solver': 'sparse_cg', 'tol': 0.0001}
rc_model_F1 = RidgeClassifier(**rc_param_F1);
rc_model_F1.fit(X_train,y_train);
y_pred_RC_F1 = rc_model_F1.predict(X_test)

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_RC_F1); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_RC_F1) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_RC_F1, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_RC_F1, average='macro') # F1-macro
temp_df = pd.DataFrame({
```

```
    'Model': ['Ridge Classifier (F1)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_RC_F1)],
    'Recall': [recall_score(y_test, y_pred_RC_F1)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)

#%% Ridge Classifier (AP)

rc_param_AP = rcParams_AP = {'alpha': 1.0, 'class_weight': 'balanced', 'solver': 'auto'}
rc_model_AP = RidgeClassifier(random_state = 42, max_iter = 2000, **rc_param_AP);
rc_model_AP.fit(X_train,y_train);
y_pred_RC_AP = rc_model_AP.predict(X_test)

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

# Evaluate results
cm = confusion_matrix(y_test,y_pred_RC_AP); # Generate confusion matrix.
tn, fp, fn, tp = cm.ravel(); # Get relevant values.
precision1, recall1, threshold1 = precision_recall_curve(y_test,y_pred_RC_AP) # Precision, Recall
auc_score = auc(recall1, precision1) # AUPRC
f1_binary = f1_score(y_test, y_pred_RC_AP, average='binary') # F1-binary
f1_macro = f1_score(y_test, y_pred_RC_AP, average='macro') # F1-macro
temp_df = pd.DataFrame({
    'Model': ['Ridge Classifier (AP)'],
    'True Negative': [tn],
    'False Positive': [fp],
    'False Negative': [fn],
    'True Positive': [tp],
    'F1-Score (Binary)': [f1_binary],
    'F1-Score (Macro)': [f1_macro],
    'Precision': [precision_score(y_test, y_pred_RC_AP)],
    'Recall': [recall_score(y_test, y_pred_RC_AP)],
    'AUPRC': [auc_score]
})
results_df = pd.concat([results_df, temp_df], ignore_index = True)
```

Code B.3: Binary classification model evaluation on GNPS spectra

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 25 20:56:40 2023

@author: alas
"""
#%% Imports

import os
from os import chdir

import numpy as np
import pandas as pd
import seaborn as sns
import sklearn
import pickle
import sys
import lightgbm as lgb
import matplotlib.pyplot as plt
import time
import catboost
import xgboost as xgb


from rdkit import Chem
from rdkit.Chem import Descriptors
from rdkit.Chem import PandasTools
from rdkit.Chem import AllChem

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_validate
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_auc_score, average_precision_score
from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
confusion_matrix
from sklearn.linear_model import LogisticRegression, RidgeClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.feature_selection import VarianceThreshold,SelectKBest, f_classif, RFECV
from scipy.signal import find_peaks
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB, ComplementNB

from catboost import CatBoostClassifier, Pool

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, average_precision_score, precision_recall_curve, auc,
PrecisionRecallDisplay


#%% Directory

# chdir(r"C:\Users\imraa\Documents\UWM\BugniLab\Tubulin-Binders")
chdir(r"C:\Users\alas\Documents\Python")

results_df = None;

#%% Data Preparation

### Load & process the training data.
```

```
feat_type = 'ms2fp';
# load data, get features
df = pd.read_pickle('scriptOutputs/training_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
df = df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
df['CID'] = df['CID'].astype(str)
# get molids
molid_list = df.CID.tolist()
# features
col_list = df.columns.tolist()
feat_cols = [ c for c in col_list[2:] if 'ms2fp_' in str(c) ]
X = df[feat_cols].values
# labels
y = df.Positive.values
# weird issue with int type in y, had to set type to match X
y = np.array(y, np.int64)

### Load & process the validation data.
feat_type = 'ms2fp';
# load data, get features
val_df = pd.read_pickle('scriptOutputs/validation_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
val_df = val_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
val_df['CID'] = val_df['CID'].astype(str)
# get molids
molid_list_val = val_df.CID.tolist()
# features
val_col_list = val_df.columns.tolist()
val_feat_cols = [ c for c in val_col_list[2:] if 'ms2fp_' in str(c) ]
X_val = val_df[val_feat_cols].values
# labels
y_val = val_df.Positive.values
# weird issue with int type i
y_val = np.array(y_val, np.int64)

### Load & process the testing data
feat_type = 'ms2fp';
# load data, get features
test_df = pd.read_pickle('scriptOutputs/testing_data_gitter_v1_20231025.pkl') # already split
# in descriptors, had to replace infinite values with NaNs
test_df = test_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
test_df['CID'] = test_df['CID'].astype(str)
# get molids
molid_list_test = test_df.CID.tolist()
# features
test_col_list = test_df.columns.tolist()
test_feat_cols = [ c for c in test_col_list[2:] if 'ms2fp_' in str(c) ]
X_test1 = test_df[test_feat_cols].values
# labels
y_test1 = test_df.Positive.values
# weird issue with int type i
y_test1 = np.array(y_test1, np.int64)

### Group the 60% Training & 20% Validation & 20% testing data.

X_train = np.concatenate((X, X_val, X_test1));
y_train = np.concatenate((y,y_val,y_test1));

### Load & process the GNPS data
feat_type = 'ms2fp';
# load data, get features
gnps_df = pd.read_csv('scriptOutputs/GNPS_tubulin_binding_fingerprints.tsv', sep='\t') # already split
# in descriptors, had to replace infinite values with NaNs
gnps_df = gnps_df.replace( [np.inf, -np.inf], np.nan )
# Force all elements in CID to be strings.
gnps_df['CID'] = gnps_df['CID'].astype(str)
```

```
# Create a dictionary to map old column names to new ones
column_rename_map = {str(i): f'ms2fp_{i}' for i in range(3878)}  # Assuming you want columns up to 3877
# Rename the columns using the dictionary
gnps_df = gnps_df.rename(columns=column_rename_map)

# get molids
molid_list_gnps = gnps_df.CID.tolist()
# Get parent classes
parent_classes_gnps = gnps_df['Parent Class'].tolist()
# features
gnps_col_list = gnps_df.columns.tolist()
gnps_feat_cols = [ c for c in gnps_col_list[2:] if 'ms2fp' in str(c) ]
X_test = gnps_df[gnps_feat_cols].values
# I have no labels for the GNPS data.
# There is no y_test.



#%% Model #5: SVM, with hyperparameters optimized based on Average Precision.

svm_param_AP = {'C': 10, 'gamma': 'scale', 'kernel': 'poly'}
svm_model_AP = SVC(random_state = 42, **svm_param_AP)
svm_model_AP.fit(X_train, y_train);
y_pred_SVM_AP = svm_model_AP.predict(X_test);

#%% Model #4: SVM, with hyperparameters optimized based on F1.

svm_param_F1 = {'C': 1, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0': 0.0,
'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'scale', 'kernel': 'linear', 'max_iter': -1,
'probability': False, 'shrinking': True, 'tol': 0.001, 'verbose': False}
svm_model_F1 = SVC(random_state = 42, **svm_param_F1)
svm_model_F1.fit(X_train, y_train);
y_pred_SVM_F1 = svm_model_F1.predict(X_test);

# See if results_df exists already. If not, create it.
try:
    results_df
except:
    results_df = pd.DataFrame(columns=['Model', 'True Negative', 'False Positive', 'False Negative', 'True
Positive', 'F1-Score (Binary)', 'F1-Score (Macro)', 'Precision', 'Recall', 'AUPRC'])

#%% Model #3: LightGBM, with hyperparameters optimized based on F1.

lgb_param_F1 = {'boosting_type': 'gbdt', 'class_weight': None, 'colsample_bytree': 1.0, 'importance_type':
'split', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_samples': 1000, 'min_child_weight': 0.001,
'min_split_gain': 0.0, 'n_estimators': 2500, 'n_jobs': -1, 'num_leaves': 15, 'objective': 'binary',
'reg_alpha': 1, 'reg_lambda': 0.8, 'silent': 'warn', 'subsample': 0.6, 'subsample_for_bin': 200000,
'subsample_freq': 5, 'is_unbalance': 'true'};
lgb_model_F1 = lgb.LGBMClassifier(random_state=42, **lgb_param_F1)
lgb_model_F1.fit(X_train, y_train);
y_pred_LGB_F1 = lgb_model_F1.predict(X_test);

#%% Model #2: MLPClassifier, with hyperparameters optimized based on AP.

mlp_param_AP = {'activation': 'relu', 'alpha': 0.01, 'hidden_layer_sizes': (32, 16), 'solver': 'adam'};
mlp_model_AP = MLPClassifier(random_state=42, max_iter=500, **mlp_param_AP)
mlp_model_AP.fit(X_train, y_train);
y_pred_MLP_AP = mlp_model_AP.predict(X_test);

#%% Model #1: MLPClassifier, with hyperparameters optimized based on F1.

mlp_param_F1 = {'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1': 0.9, 'beta_2':
0.999, 'early_stopping': False, 'epsilon': 1e-08, 'hidden_layer_sizes': (128, 64), 'learning_rate':
'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 500, 'momentum': 0.9,
'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 42, 'shuffle': True,
'solver': 'lbfgs', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
# mlp_param_F1_v2 = {'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1': 0.9,
'beta_2': 0.999, 'early_stopping': True, 'epsilon': 1e-08, 'hidden_layer_sizes': (256, 256),
```

```
'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 2000, 'momentum':
0.9, 'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 42, 'shuffle':
True, 'solver': 'lbfgs', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
# mlp_param_F1 = mlp_param_F1_v2;
mlp_model_F1 = MLPClassifier(**mlp_param_F1) # max_iter & random_state already included in mlp_param_F1
mlp_model_F1.fit(X_train, y_train);
y_pred_MLP_F1 = mlp_model_F1.predict(X_test);

#%% Ridge Classifier (F1)

rc_param_F1 = {'alpha': 0.6, 'class_weight': 'balanced', 'copy_X': True, 'fit_intercept': True,
'max_iter': 2000, 'positive': False, 'random_state': 42, 'solver': 'sparse_cg', 'tol': 0.0001}
rc_model_F1 = RidgeClassifier(**rc_param_F1);
rc_model_F1.fit(X_train,y_train);
y_pred_RC_F1 = rc_model_F1.predict(X_test)

#%% Ridge Classifier (AP)

rc_param_AP = rcParams_AP = {'alpha': 1.0, 'class_weight': 'balanced', 'solver': 'auto'}
rc_model_AP = RidgeClassifier(random_state = 42, max_iter = 2000, **rc_param_AP);
rc_model_AP.fit(X_train,y_train);
y_pred_RC_AP = rc_model_AP.predict(X_test)

#%% MLPCLassifier, v2 hyperparameters (F1)

mlp_param_F1_v2 = {'activation': 'logistic', 'alpha': 0.0001, 'batch_size': 'auto', 'beta_1': 0.9,
'beta_2': 0.999, 'early_stopping': True, 'epsilon': 1e-08, 'hidden_layer_sizes': (256, 256),
'learning_rate': 'constant', 'learning_rate_init': 0.001, 'max_fun': 15000, 'max_iter': 2000, 'momentum':
0.9, 'n_iter_no_change': 10, 'nesterovs_momentum': True, 'power_t': 0.5, 'random_state': 42, 'shuffle':
True, 'solver': 'lbfgs', 'tol': 0.0001, 'validation_fraction': 0.1, 'verbose': False, 'warm_start': False}
mlp_model_F1_v2 = MLPClassifier(**mlp_param_F1_v2) # max_iter & random_state already included in
mlp_param_F1
mlp_model_F1_v2.fit(X_train, y_train);
y_pred_MLP_F1_v2 = mlp_model_F1_v2.predict(X_test);

#%% MLPClassifier, v2 hyperparameters (AP)

mlp_param_AP_v2 = {'activation': 'tanh', 'alpha': 0.01, 'early_stopping': True, 'hidden_layer_sizes':
(1536,), 'solver': 'adam'}
mlp_model_AP_v2 = MLPClassifier(random_state = 42, max_iter = 2000, **mlp_param_AP_v2)
mlp_model_AP_v2.fit(X_train, y_train)
y_pred_MLP_AP_v2 = mlp_model_AP_v2.predict(X_test)

#%% Investigate results

# Store data.
y_values = pd.DataFrame(data={'SVM (AP)': y_pred_SVM_AP,
                              'SVM (F1)': y_pred_SVM_F1,
                              'LGB (F1)': y_pred_LGB_F1,
                              'MLP (AP)': y_pred_MLP_AP,
                              'MLP (F1)': y_pred_MLP_F1,
                              'RC (F1)': y_pred_RC_F1,
                              'RC (AP)': y_pred_RC_AP,
                              'MLP v2 (F1)': y_pred_MLP_F1_v2,
                              'MLP v2 (AP)': y_pred_MLP_AP_v2});
# Add metadata
y_values['Parent Class'] = parent_classes_gnps;
# Check sums (if predicted No, has a 0. If predicted 'yes', has a 1).
print(y_values.sum())
# Select rows where at least one column has a value of 1.
positive_guesses = y_values[(y_values == 1).any(axis=1)];
# Get the number of unique parent compound classes:
unique_entries = y_values['Parent Class'].value_counts();
print(unique_entries)

# See summed_df for number of positive guesses by parent class.
summed_df = y_values.groupby(by='Parent Class').sum()
```

Code B.4: Multiclass classification model hyperparameter optimization and evaluation

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  4 20:24:19 2024

@author: alas
"""

#%% Imports

import os
from os import chdir

#%% Directory

chdir('C:/Users/alas/Documents/Python/20240404-tubulin_binding_work')

#%% Imports (Nathan)

"""
Goal: this notebook will be the experimental notebook for the training and testing metrics of each model
type against the training/testing data as well as the GNPS testing data.

Steps:
1. Import data for training/testing and GNPS testing
2. Initiate a variety of model types
3. Train each model type
4. Test each model type
5. Get the metrics for each model type on the training/testing data and GNPS testing data
6. Evaluate the false positive rate for each model type on the random spectra
7. Sort the models by their metrics

Author: Nathan Brittin
Date: 12 - 08 - 23

"""
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from tqdm import tqdm
import warnings
from sklearn.model_selection import train_test_split
from sklearn import linear_model
from sklearn import neural_network
from sklearn import svm
from sklearn import neighbors
from sklearn import tree
from sklearn.metrics import mean_squared_error, r2_score, accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, matthews_corrcoef, make_scorer, average_precision_score
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import LabelEncoder
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import GridSearchCV
import time
import copy


#%% Pre-process data

positive_filepath = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/Training and
Testing/Fingerprints/Tubulin_Binding_Compounds_Positive_Fingerprint_Matrix.tsv"
negative_filepath = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/Training and
Testing/Fingerprints/Negative_Fingerprint_Matrix.tsv"

"""
Importing the datasets
```

```
"""

# Get the postive and negative dataframes and read in the pickle files
positive_df = pd.read_csv(positive_filepath, sep="\t", header=0)
negative_df = pd.read_csv(negative_filepath, sep="\t", header=0)

print("Positive dataframe shape: ", positive_df.shape)
print("Negative dataframe shape: ", negative_df.shape)

combined_df = pd.concat([positive_df, negative_df], axis=0)
print("Combined dataframe shape: ", combined_df.shape)
# display(combined_df.head())

X = combined_df.copy()
X.set_index("CID", inplace=True)
smiles = X.pop("SMILES")
family = X.pop("Positive")
name = X.pop("Name")
y = X.pop("Family")

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
# Encode the labels
le = LabelEncoder()
y_train_encoded = le.fit_transform(y_train) # Converts the text labels into numbers representing each
unique label
y_test_encoded = le.transform(y_test)

print(f"X_train shape: {X_train.shape}, y_train shape: {y_train.shape}, X_test shape: {X_test.shape},
y_test shape: {y_test.shape}")

#%% Once done with Hyperparameter Optimization:

# If you run this, do not run the section titled: Model: Test
    # That section was previously used to evaluate the 80% trained model on the 20% held out.
    # This section re-designs the data such that you're now training on 100% of the data.

# # Store old X_train, y_train
# old_X_train = X_train;
# old_y_train = y_train;
# old_y_train_encoded = y_train_encoded;
# old_X_test = X_test;
# old_y_test = y_test;
# old_y_test_encoded = y_test_encoded;

# # Re-use X_train to keep code functional
# X_train = pd.concat([X_train, X_test], sort=False)
# # Re-use y_train to keep code function
# y_train = y;
# y_train_encoded = le.fit_transform(y_train);


#%% Models: List

models_list = [
            linear_model.RidgeClassifier(random_state=42),
            linear_model.Perceptron(random_state=42, n_jobs=-1),
            linear_model.PassiveAggressiveClassifier(random_state=42, n_jobs=-1),
            linear_model.SGDClassifier(random_state=42, n_jobs=-1),
            # linear_model.LogisticRegression(random_state=42, n_jobs=-1),
            neighbors.KNeighborsClassifier(n_jobs=-1),
            svm.SVC(random_state=42),
            neural_network.MLPClassifier(random_state=42, hidden_layer_sizes=(128,)),
            ]

#%% Models: Previously Attempted Hyperparameters

# param_grid = {'RidgeClassifier': {'solver': ['auto', 'svd', 'cholesky',
'lsqr','sparse_cg','sag','saga','lbfgs'],
```

```
#                                             'alpha': [0.0001, 0.001, 0.01, 0.1,
0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0],
#                                             'class_weight':[None, 'balanced']},
#               'Perceptron': {'penalty':[None,'l2','l1','elasticnet'],
#                              'alpha': [0.001,0.001,0.1,0.5],
#                              'early_stopping': [False, True],
#                              'class_weight': [None, 'balanced'],
#                              'max_iter': [1000, 2000]},
#               'PassiveAggressiveClassifier': {'max_iter': [1000,2000],
#                                               'early_stopping': [False, True],
#                                               'class_weight': [None, 'balanced'],
#                                               'C': [0.1,0.5,0.8,0.9]},
#               'SGDClassifier': {'loss': ['hinge', 'log_loss', 'modified_huber', 'squared_hinge',
'perceptron', 'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'],
#                                 'penalty': ['l2','l1','elasticnet',None],
#                                 'alpha': [0.001,0.1,0.5],
#                                 'max_iter': [1000,2000],
#                                 'early_stopping': [False, True],
#                                 'class_weight': [None, 'balanced']},
#               'LogisticRegression': {'penalty': ['l1','l2','elasticnet',None],
#                                      'C': [0.1,0.5,0.8,0.9],
#                                      'class_weight': [None, 'balanced'],
#                                      'solver': ['lbfgs','liblinear','newton-cg','newton-
cholesky','sag','saga'],
#                                      'max_iter': [1000,2000],
#                                      'multi_class': ['auto','ovr','multinomial']},
#               'KNeighborsClassifier': {'n_neighbors': [3,5,10,15,20,50,100],
#                                        'weights': ['uniform','distance'],
#                                        'algorithm': ['ball_tree','kd_tree','brute'],
#                                        'p': [1,2],
#                                        'leaf_size': [15, 30, 45, 60]},
#               'SVC': {'C': [0.1,0.5,0.8,0.9],
#                       'kernel': ['linear', 'poly', 'rbf','sigmoid'],
#                       'gamma': ['scale','auto'],
#                       'class_weight': [None, 'balanced'],
#                       'decision_function_shape': ['ovr','ovo']},
#               'MLPClassifier': {'hidden_layer_sizes': [(512,), (1024,), (1536,), (2048,), (256,256),
(512,512), (1024,1024), (1536,1536), (2048,2048)],
#                                 'activation': ['relu', 'tanh','logistic'],
#                                 'solver': ['adam', 'sgd', 'lbfgs'],
#                                 'alpha': [0.0001, 0.001, 0.01, 0.1],
#                                 'early_stopping': [True]},
#               }

#%% Models: Current Hyperparameters (Storage For Later)

param_grid = {};
# Weighted_F1
param_grid = {'RidgeClassifier': {'alpha': [0.01], 'class_weight': ['balanced'], 'solver': ['sparse_cg']},
              'Perceptron': {'alpha': [0.001], 'class_weight': [None], 'early_stopping': [False],
'max_iter': [1000], 'penalty': [None]},
              'PassiveAggressiveClassifier': {'C': [0.1], 'class_weight': [None], 'early_stopping':
[False], 'max_iter': [1000]},
              'SGDClassifier': {'alpha': [0.001], 'class_weight': [None], 'early_stopping': [False],
'loss': ['hinge'], 'max_iter': [1000], 'penalty': ['l2']},
              'LogisticRegression': {'C': [0.1], 'class_weight': [None], 'max_iter': [1000],
'multi_class': ['ovr'], 'penalty': [None], 'solver': ['newton-cg']},
              'KNeighborsClassifier': {'algorithm': ['ball_tree'], 'leaf_size': [15], 'n_neighbors': [3],
'p': [1], 'weights': ['distance']},
              'SVC': {'C': [0.1], 'class_weight': [None], 'decision_function_shape': ['ovr'], 'gamma':
['auto'], 'kernel': ['linear']},
              'MLPClassifier': {'activation': ['tanh'], 'alpha': [0.1], 'early_stopping': [True],
'hidden_layer_sizes': [(256, 256)], 'solver': ['lbfgs'], 'max_iter':[10000]}
              }

# Note: Logistic Regression with these hyperparameters doesn't converge with 1000 iterations.
# MLPClassifier had to be bumped to 10,000 max_iters to converge.
```

```
#%% Scoring

scoring = {'macro_f1': make_scorer(f1_score, average='macro'),
           'weighted_f1': make_scorer(f1_score, average='weighted')# ,
           # 'average_precision': make_scorer(average_precision_score, average = 'macro',
multi_class='ovr'),
           # 'macro_rocauc': make_scorer(roc_auc_score, average = 'macro', multi_class='ovo')
           }
optimized_models = [];
grid_search_storage = [];

#%% Models: Train (Hyperparameters)

# for model in tqdm(models_list):
#     model.fit(X_train, y_train_encoded)

# If using optimized hyperparameters:
for model in tqdm(models_list):
    model_name = str(model).split('(')[0]
    print('\n' + model_name)
    startTime = time.time(); # Check Time
    params = param_grid[model_name]
    single_val_params = {key: val[0] for key, val in params.items()}
    model.set_params(**single_val_params)
    model.fit(X_train, y_train_encoded)
    print('Completed training for ' + model_name)
    endTime = time.time()
    lengthTime = endTime - startTime;
    t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
    print('\nFinished this model in {}'.format(t_str))
    optimized_models.append(model)

# If searching hyperparameters
# for model in tqdm(models_list):
#     model_name = str(model).split('(')[0]
#     print('\n' + model_name)
#     startTime = time.time(); # Check Time
#     if model_name in param_grid:
#         params = param_grid[model_name]
#         # print("here")
#         grid_search = GridSearchCV(model,params,scoring=scoring,refit='weighted_f1',cv=5,n_jobs=-1,
verbose=0)
#         # print("here 2")
#         grid_search.fit(X_train, y_train_encoded);
#         # print("here 3")
#         best_model = grid_search.best_estimator_;
#         # print("here 4")
#         optimized_models.append(best_model)
#         grid_search_storage.append(grid_search);
#         print('\nCompleted Hyperparameter Search for: ' + model_name)
#     else:
#         model.fit(X_train, y_train_encoded)
#         print('Completed training for ' + model_name)
#         optimized_models.append(model)
#     endTime = time.time()
#     lengthTime = endTime - startTime;
#     t_str = '{}h{}m{}s'.format(int(lengthTime/3600),int(lengthTime%3600/60),int(lengthTime%3600%60))
#     print('\nFinished this model in {}'.format(t_str))

# If searching hyperparameters, save the hyperparameters somewhere.
# for grid_search_result in grid_search_storage:
#     counter = 1; # Always start from 1.
#     gridSearchDF = pd.DataFrame(grid_search_result.cv_results_)
#     model_name_pull = str(grid_search_result).split('estimator=')[1].split('(')[0] # assumptions: the
model name is referenced by estimator, and the model name always does model()
#     possible_file_name = 'C:/Users/alas/Documents/Python/20240404-
tubulin_binding_work/Scripts_Imraan/scriptOutputs/grid_opt_' + model_name_pull + '_v' + str(counter) +
'.csv';
#     while os.path.isfile(possible_file_name):
```

```
#         # if file exists, increment the counter
#         counter += 1;
#         # update the filename
#         possible_file_name = 'C:/Users/alas/Documents/Python/20240404-
tubulin_binding_work/Scripts_Imraan/scriptOutputs/grid_opt_' + model_name_pull + '_v' + str(counter) +
'.csv';
#     gridSearchDF.to_csv(possible_file_name, index=False)

print('Finished training...')

#%% Model: Test

training_results_df = pd.DataFrame(columns=['Model', 'Accuracy', 'Precision', 'Recall', 'F1', 'MCC'])
for model in tqdm(optimized_models):
    predictions = model.predict(X_test)
    predictions = np.rint(predictions).astype(int)
    accuracy = accuracy_score(y_test_encoded, predictions)
    precision = precision_score(y_test_encoded, predictions, average='weighted', zero_division=0)
    recall = recall_score(y_test_encoded, predictions, average='weighted', zero_division=0)
    f1 = f1_score(y_test_encoded, predictions, average='weighted', zero_division=0)
    mcc = matthews_corrcoef(y_test_encoded, predictions)
    model_name = str(model).split('(')[0]
    temp_df = pd.DataFrame({'Model': model_name, 'Accuracy': accuracy, 'Precision': precision, 'Recall':
recall, 'F1': f1, 'MCC': mcc}, index=[0])
    training_results_df = pd.concat([training_results_df, temp_df], axis=0, ignore_index=True)
training_results_df = training_results_df.sort_values(by=['MCC'], ascending=False)
print("Testing results for each model type:")
print(training_results_df)

#%% Find duplicate CIDs across FPR dataset and GNPS dataset


# FPR dataset contains all negatives, we expect.
fpr_filepath = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/False Positive
Testing/Random_Fingerprints/Random_GNPS_fingerprints.tsv"
fpr_fingerprints = pd.read_csv(fpr_filepath, sep="\t", header=0)
fpr_fingerprints.set_index("CID", inplace=True)

# GNPS dataset contains all positives, as we expect.

# Import GNPS fingerprints
gnps_file = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/GNPS Spectra
Testing/Fingerprints/GNPS_tubulin_binding_fingerprints.tsv"
gnps_fingerprints = pd.read_csv(gnps_file, sep="\t", header=0)
print(gnps_fingerprints["Family"].value_counts())

# save original gnps df
original_gnps = copy.deepcopy(gnps_fingerprints);

y_gnps = gnps_fingerprints.pop("Family")
# Replace the Taxoles label with Taxols
y_gnps = y_gnps.replace("Taxoles", "Taxols")
# Didn't realize that Podophyllotoxin (the label that was used for training) had a space at the end. Make
this match, can fix later.
y_gnps = y_gnps.replace("Podophyllotoxin", "Podophyllotoxin ")
y_gnps_encoded = le.transform(y_gnps)
print(pd.DataFrame(y_gnps_encoded).value_counts())
gnps_fingerprints.set_index("CID", inplace=True)

# If we combine the two, we get a picture of the models effectiveness.
y_FPR = pd.Series(["Negative"]*5041, name="Family")
y_all = pd.concat([y_gnps, y_FPR], ignore_index=True)
fingerprints_all = pd.concat([gnps_fingerprints, fpr_fingerprints])
print("Shape of concatenated DataFrame:", fingerprints_all.shape)
print("Unique index values in concatenated DataFrame:", len(fingerprints_all.index.unique()))
# Determine overlapping values?
duplicated_indexes = fingerprints_all[fingerprints_all.index.duplicated(keep=False)]
stored_duplicated_families = original_gnps.iloc[:0,:].copy();
# See which families the duplicated indexes belong to
```

```
for i in set(duplicated_indexes.index.tolist()): # only get 1 of the 2 duplicated values for each
duplicated instance
    stored_duplicated_families = pd.concat([stored_duplicated_families, original_gnps[original_gnps['CID']
== i]]);

# Check if all the values of the rows with duplicated CIDs match each other
all_rows_match = [];
for index in set(duplicated_indexes.index.tolist()):
    doubled_rows = fingerprints_all.loc[index]
    first_row = doubled_rows.iloc[0];
    second_row = doubled_rows.iloc[1];
    rows_match = first_row.equals(second_row)
    all_rows_match.append(rows_match)

# Convert the list to a Series
all_rows_match_series = pd.Series(all_rows_match, index=set(duplicated_indexes.index.tolist()))
print(all_rows_match_series)

# Note: Remove duplicates from FPR, not GNPS. Current assumption: They're actually positives, so they
should not be a part of the FPR dataset.
duplicated_CIDs = list(set(duplicated_indexes.index.tolist()));

#%% Models: Evaluate on GNPS.

# Import GNPS fingerprints
gnps_file = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/GNPS Spectra
Testing/Fingerprints/GNPS_tubulin_binding_fingerprints.tsv"
gnps_fingerprints = pd.read_csv(gnps_file, sep="\t", header=0)
print(gnps_fingerprints["Family"].value_counts())

# save original gnps df
original_gnps = copy.deepcopy(gnps_fingerprints);

y_gnps = gnps_fingerprints.pop("Family")
# Replace the Taxoles label with Taxols
y_gnps = y_gnps.replace("Taxoles", "Taxols")
# Didn't realize that Podophyllotoxin (the label that was used for training) had a space at the end. Make
this match, can fix later.
y_gnps = y_gnps.replace("Podophyllotoxin", "Podophyllotoxin ")
y_gnps_encoded = le.transform(y_gnps)
print(pd.DataFrame(y_gnps_encoded).value_counts())
gnps_fingerprints.set_index("CID", inplace=True)

storage_accuracy = [];
gnps_results = pd.DataFrame(columns=["Model", "Accuracy","Precision","Recall","F1","MCC"])
for model in tqdm(optimized_models):
    model_name = str(model).split("(")[0]
    y_pred = model.predict(gnps_fingerprints)
    y_pred = np.round(y_pred)
    y_pred = y_pred.astype(int)
    # Decode the labels
    y_pred = le.inverse_transform(y_pred)
    # print(pd.DataFrame(y_pred).value_counts())
    accuracy = accuracy_score(y_gnps, y_pred)
    storage_accuracy.append(accuracy)
    precision = precision_score(y_gnps, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_gnps, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_gnps, y_pred, average='weighted', zero_division=0)
    mcc = matthews_corrcoef(y_gnps, y_pred)
    temp_df = pd.DataFrame({'Model': model_name, 'Accuracy': accuracy, 'Precision': precision, 'Recall':
recall, 'F1': f1, 'MCC': mcc}, index=[0])
    gnps_results = pd.concat([gnps_results, temp_df], axis=0)

gnps_results.sort_values(by="MCC", ascending=False, inplace=True)

#%% Models: Calculate FPR

# Import FPR fingerprint dataset.
```

```
fpr_filepath = "C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/False Positive
Testing/Random_Fingerprints/Random_GNPS_fingerprints.tsv"
fpr_fingerprints = pd.read_csv(fpr_filepath, sep="\t", header=0)
fpr_fingerprints.set_index("CID", inplace=True)

# Save original fpr dataset
original_fpr = copy.deepcopy(fpr_fingerprints);

# Remove duplicate CIDs that were in the GNPS dataset.
fpr_fingerprints.drop(duplicated_CIDs, inplace=True);

# Get the predictions for each model on the FPR dataset
fpr_results = pd.DataFrame(columns=["Model", "Number of False Positives", "False Positive Rate"])
for model in tqdm(optimized_models):
    y_pred = model.predict(fpr_fingerprints)
    y_pred = np.round(y_pred)
    y_pred = y_pred.astype(int)
    # Decode the labels
    y_pred = le.inverse_transform(y_pred)
    negative_count = y_pred.tolist().count("Negative")
    num_fp = len(y_pred) - negative_count
    fpr = num_fp / len(y_pred)
    fpr = fpr * 100
    model_name = str(model).split("(")[0]
    temp_df = pd.DataFrame({"Model": [model_name], "Number of False Positives": [num_fp], "False Positive
Rate": [fpr]})
    fpr_results = pd.concat([fpr_results, temp_df], axis=0)

fpr_results.sort_values(by="False Positive Rate", ascending=True, inplace=True)
# display(fpr_results)

#%% Evaluate across FPR dataset & GNPS dataset

# If we combine the two, we get a picture of the models effectiveness.
y_FPR = pd.Series(["Negative"]*5033, name="Family") # 5033 after removing the 8 from 5041
y_all = pd.concat([y_gnps, y_FPR], ignore_index=True)
fingerprints_all = pd.concat([gnps_fingerprints, fpr_fingerprints]) # Combine the gnps_fingerprint & the
fpr (de-duplicated) fingerprint

all_results = pd.DataFrame(columns=["Model", "Accuracy", "Precision","Recall","F1","MCC"])
for model in tqdm(optimized_models):
    model_name = str(model).split("(")[0]
    y_pred = model.predict(fingerprints_all)
    y_pred = np.round(y_pred)
    y_pred = y_pred.astype(int)
    # Decode the labels
    y_pred = le.inverse_transform(y_pred)
    # print(pd.DataFrame(y_pred).value_counts())
    accuracy = accuracy_score(y_all, y_pred)
    precision = precision_score(y_all, y_pred, average='weighted', zero_division=0)
    recall = recall_score(y_all, y_pred, average='weighted', zero_division=0)
    f1 = f1_score(y_all, y_pred, average='weighted', zero_division=0)
    mcc = matthews_corrcoef(y_all, y_pred)
    temp_df = pd.DataFrame({'Model': model_name, 'Accuracy': accuracy, 'Precision': precision, 'Recall':
recall, 'F1': f1, 'MCC': mcc}, index=[0])
    all_results = pd.concat([all_results, temp_df], axis=0)

#%% Check Results:

# See the effectiveness of the models on the GNPS data (all positives).
print(gnps_results)
# See the effectiveness of the models on the GNPS data (all negatives).
print(fpr_results)
# See the effectiveness of the models on all the data (GNPS & FPR datasets).
print(all_results)

collected_results = {};
collected_results['GNPS'] =  gnps_results
collected_results['FPR'] =  fpr_results
```

```
collected_results['All'] =  all_results

#%% Evaluate within Spectra Categories

# Load the FPR dataset & GNPS dataset, specifically connecting CIDs to Spectra Quality.
quality_FPR_df = pd.read_csv('C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/Assessing
Spectral Quality/FPR_set_spectral_quality_summary.tsv', sep='\t');
quality_GNPS_df = pd.read_csv('C:/Users/alas/Documents/Python/20240404-tubulin_binding_work/Assessing
Spectral Quality/tubulin_binding_GNPS_spectral_quality_summary.tsv', sep='\t');

# Pull the combined FPR/GNPS dataset (duplicate CIDs across the two types removed)
fingerprints_all;

# For each CID in fingerprints_all, store information regarding Spectra Quality.
columnsSpectraQuality = ['CID', 'Family', 'Quality Grade', 'Number of Peaks', 'Fold Difference', 'Base
Peak Intensity'];
model_names = [str(i).split("(")[0] for i in optimized_models]
# columnsSpectraQuality.extend(model_names);
dict_storage = {};
spectra_quality_DF = pd.DataFrame(columns = columnsSpectraQuality);
# For each CID, get the relevant 'Family', 'Grade', # of Peaks', 'Fold Difference', 'Base Peak Intensity'
for i in fingerprints_all.index.tolist():
    # Find the correct dataset.
    if i in list(quality_GNPS_df['CID']):
        relevant_df = quality_GNPS_df;
    elif i in list(quality_FPR_df['CID']):
        relevant_df = quality_FPR_df
    else:
        print('Didn\'t find a match')
        continue
    # Scrape the relevant content
    relevantRow = relevant_df[relevant_df['CID'] == i]
    quality_data = relevantRow[['Quality Grade', 'Number of Peaks', 'Fold Difference', 'Base Peak
Intensity']].values.flatten()
    qData_dict = {'CID': i};
    qData_dict.update(zip(columnsSpectraQuality[2:], quality_data))
    # Put into dictionary of dictionaries
    dict_storage[i] = qData_dict

# Create dataframe of links.
qualityDF = pd.DataFrame.from_dict(dict_storage, orient='index');

# Take the best-performing model.
final_model = optimized_models[5];
y_pred = final_model.predict(fingerprints_all)
y_pred = np.round(y_pred)
y_pred = y_pred.astype(int)
# Decode the labels
y_pred = le.inverse_transform(y_pred)
accuracy = accuracy_score(y_all, y_pred)
precision = precision_score(y_all, y_pred, average='weighted', zero_division=0)
recall = recall_score(y_all, y_pred, average='weighted', zero_division=0)
f1 = f1_score(y_all, y_pred, average='weighted', zero_division=0)
mcc = matthews_corrcoef(y_all, y_pred)
final_df = pd.DataFrame({'Model': model_name, 'Accuracy': accuracy, 'Precision': precision, 'Recall':
recall, 'F1': f1, 'MCC': mcc}, index=[0])

# Note: qualityDF is in order of gnps_fingerprints then fpr_fingerprints
# This is the same order as fingerprints_all
# This is the same order as y_all and y_pred

qualityDF['Predicted'] = y_pred;
qualityDF['Truth'] = y_all.values;

# Split dataset up by Grades.
gradeDict = {};
gradeList = ['S', 'A', 'B', 'C', 'D', 'E', 'F', 'Really Bad!'];
for i in gradeList:
    slicedDF = qualityDF[qualityDF['Quality Grade'] == i];
```

```
    slicedPred = slicedDF['Predicted'].tolist();
    slicedTruth = slicedDF['Truth'].tolist();
    accuracyGrade = accuracy_score(slicedTruth, slicedPred);
    slicedDict = {'Grade': i, 'Accuracy': accuracyGrade}
    gradeDict[i] = slicedDict;

# Accuracy by Grade.
accuracyGradeDF = pd.DataFrame.from_dict(gradeDict, orient='index')
```

Code B.5: Function to determine spectra quality grades

```
def assign_quality_group(num_peaks, fold_diff, base_peak_intensity):
    """
    Assign a quality group based on the number of peaks, fold difference, and base peak intensity
    """
    # Initialize the quality group
    quality_group = 1
    # Assign the quality group based on the number of peaks, fold difference, and base peak intensity
    if num_peaks < 20 and fold_diff < 30 and base_peak_intensity < 20000:
        quality_group = 4
        if num_peaks < 15:
            quality_group = 5
    elif num_peaks < 30 and fold_diff < 50 and base_peak_intensity < 50000:
        quality_group = 3
        if num_peaks < 25:
            quality_group = 4
    elif num_peaks < 40 and fold_diff < 200 and base_peak_intensity < 100000:
        quality_group = 2
        if num_peaks < 35:
            quality_group = 3
    if num_peaks < 50:
        quality_group = quality_group + 1
    if fold_diff < 100:
        quality_group = quality_group + 1
    if base_peak_intensity < 10000:
        quality_group = quality_group + 1
    grade_dict = {1: "S", 2: "A", 3: "B", 4: "C", 5: "D", 6: "E", 7: "F", 8: "Poor"}
    quality_grade = grade_dict[quality_group]
    return quality_grade
```

Table B.1: Hyperparameters for multiclass classification

Models:	Hyperparameters Searched:
Ridge Classifier	'solver': ['auto', 'svd', 'cholesky', 'lsqr','sparse_cg','sag','saga','lbfgs'], 'alpha': [0.0001, 0.001, 0.01, 0.1, 0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0], 'class_weight':[None, 'balanced']
Perceptron	'penalty':[None,'l2','l1','elasticnet'], 'alpha': [0.001,0.001,0.1,0.5], 'early_stopping': [False, True], 'class_weight': [None, 'balanced'], 'max_iter': [1000, 2000]
Passive Aggressive Classifier	'max_iter': [1000,2000], 'early_stopping': [False, True], 'class_weight': [None, 'balanced'], 'C': [0.1,0.5,0.8,0.9]
SGD Classifier	'loss': ['hinge', 'log_loss', 'modified_huber', 'squared_hinge', 'perceptron', 'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'], 'penalty': ['l2','l1','elasticnet',None], 'alpha': [0.001,0.1,0.5], 'max_iter': [1000,2000], 'early_stopping': [False, True], 'class_weight': [None, 'balanced']
Logistic Regression	'penalty': ['l1','l2','elasticnet',None], 'C': [0.1,0.5,0.8,0.9], 'class_weight': [None, 'balanced'], 'solver': ['lbfgs','liblinear','newton-cg','newton-cholesky','sag','saga'], 'max_iter': [1000,2000], 'multi_class': ['auto','ovr','multinomial']
K Neighbors Classifier	'n_neighbors': [3,5,10,15,20,50,100], 'weights': ['uniform','distance'], 'algorithm': ['ball_tree','kd_tree','brute'], 'p': [1,2], 'leaf_size': [15, 30, 45, 60]
SVC (Support Vector Machine)	'C': [0.1,0.5,0.8,0.9], 'kernel': ['linear', 'poly', 'rbf','sigmoid'], 'gamma': ['scale','auto'], 'class_weight': [None, 'balanced'], 'decision_function_shape': ['ovr','ovo']
MLP Classifier	'hidden_layer_sizes': [(512,), (1024,), (1536,), (2048,), (256,256), (512,512), (1024,1024), (1536,1536), (2048,2048)], 'activation': ['relu', 'tanh','logistic'], 'solver': ['adam', 'sgd', 'lbfgs'], 'alpha': [0.0001, 0.001, 0.01, 0.1], 'early_stopping': [True]

Table B.1. Hyperparameters explored for multiclass classification models. Parameter grids were searched using GridSearchCV. Hyperparameters were trained and evaluated on 5-fold

splits using 80% training datasets, scored using F1-macro and F1-weighted, and the best performing hyperparameters were selected using F1-weighted. Any hyperparameters not shown used default values.

Table B.2: Hyperparameters selected for multiclass classification

Models:	Hyperparameters Selected:
Ridge Classifier	'alpha': [0.01], 'class_weight': ['balanced'], 'solver': ['sparse_cg']
Perceptron	'alpha': [0.001], 'class_weight': [None], 'early_stopping': [False], 'max_iter': [1000], 'penalty': [None]
Passive Aggressive Classifier	'C': [0.1], 'class_weight': [None], 'early_stopping': [False], 'max_iter': [1000]
SGD Classifier	'alpha': [0.001], 'class_weight': [None], 'early_stopping': [False], 'loss': ['hinge'], 'max_iter': [1000], 'penalty': ['l2']
Logistic Regression	'C': [0.1], 'class_weight': [None], 'max_iter': [1000], 'multi_class': ['ovr'], 'penalty': [None], 'solver': ['newton-cg']
K Neighbors Classifier	'algorithm': ['ball_tree'], 'leaf_size': [15], 'n_neighbors': [3], 'p': [1], 'weights': ['distance']
SVC (Support Vector Machine)	'C': [0.1], 'class_weight': [None], 'decision_function_shape': ['ovr'], 'gamma': ['auto'], 'kernel': ['linear']
MLP Classifier	'activation': ['tanh'], 'alpha': [0.1], 'early_stopping': [True], 'hidden_layer_sizes': [(256, 256)], 'solver': ['lbfgs'], 'max_iter':[10000]
Table B.2. Hyperparameters selected for multiclass classification models. Parameter grids were searched using GridSearchCV. Hyperparameters were trained and evaluated on 5-fold splits using 80% training datasets, scored using F1-macro and F1-weighted, and the best performing hyperparameters were selected using F1-weighted.	