PARALLELIZATION OF EVOQ SAR MODELING FRAMEWORK AND
GRAPHICAL USER INTERFACE USING PYQT FOR DRUG DESIGN

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computational Science

by
Eric Su
Fall 2015
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the
Thesis of Eric Su:

Parallellization of evoQSAR Modeling Framework and
Graphical User Interface Design using PyQt for Drug Design

Sunil Kumar, Chair
Department of Electrical and Computer Engineering

Rajni Garg
Computational Science Research Center

Mary Thomas
Department of Computer Science

Nov. 19, 2015
Approval Date
DEDICATION

This thesis is dedicated to my parents.
ABSTRACT OF THE THESIS

Parallellization of evoQSAR Modeling Framework and Graphical User Interface Design using PyQt for Drug Design

by

Eric Su

Master of Science in Computational Science
San Diego State University, 2015

Quantitative structure-activity relationship (QSAR) analysis is a methodology that works under the assumption that certain activities of a chemical compound can be related to its structure through a computational model. These models provide a cheaper and quicker means to evaluating drug compounds, when compared to current \textit{in vivo} evaluation methods, and can help expedite screening of large chemical libraries. Unfortunately, these computational models can sometimes be very time consuming to develop due to the complex nature of the problem and lack of simple analytical solutions.

In this thesis, we have significantly enhanced a novel QSAR toolkit developed by Ko et al. called evoQSAR, by adding two additional components, parallel computation and a graphic user interface (GUI). Parallelization helps to decrease computation time by increasing efficiency through the division of independent tasks, normally run sequentially, in parallel. This is achieved by leveraging Python's built-in Multiprocessing module to harness the power of all the processors on one's local machine. Due to architectural constraints, the implementation used in this thesis is limited to Linux and OS X operating systems (OS). A user-friendly GUI was implemented, in lieu of its default python scripting interface, to open the functionalities of evoQSAR to a greater audience, including chemists, other pharmaceutical companies, and fellow researchers. The GUI is implemented through the PyQt framework which leverages off of the popular Qt framework.

The addition of parallelization and a GUI in the evoQSAR toolkit will enable chemists, pharmaceutical companies, academia, and others who could not previously utilize evoQSAR, either due to the time or learning curve associated with using it, the ability to use it for identifying novel drug compounds and expedite screening of large chemical libraries.
# TABLE OF CONTENTS

| ABSTRACT | .......................................................... v |
| LIST OF TABLES | .................................................... viii |
| LIST OF FIGURES | ...................................................... ix |
| ACKNOWLEDGMENTS | ............................................... xii |

## CHAPTER

1 INTRODUCTION .......................................................... 1
   1.1 Parallel Implementation ........................................... 1
   1.2 Graphical User Interface (GUI) ................................. 1
   1.3 Test Set ........................................................... 2

2 RELATED WORK .......................................................... 3
   2.1 evoQSAR .......................................................... 3
   2.2 Parallel Implementation ........................................... 5
   2.3 Graphical User Interface (GUI) ................................. 8

3 RESEARCH METHODS ..................................................... 12
   3.1 Development Environment ......................................... 12
   3.2 Parallelization ...................................................... 12
   3.3 GUI ............................................................... 15

4 RESULTS ................................................................. 19
   4.1 Parallelization of evoQSAR ....................................... 19
      4.1.1 DE-BPSO with MLR ........................................... 24
      4.1.2 DE-BPSO with PLSR .......................................... 28
      4.1.3 Validity ....................................................... 31
   4.2 GUI for evoQSAR .................................................. 40
      4.2.1 Software Dependencies ....................................... 40
      4.2.2 Operation ..................................................... 40

5 Conclusion and Future Direction ......................................... 63
   5.1 Conclusions ....................................................... 63
   5.2 Challenges ....................................................... 63
5.3 Future Direction ................................................................. 64

BIBLIOGRAPHY .............................................................................. 66

APPENDICES

A ADDITIONAL BENCHMARK RESULTS ........................................ 70
B HIV INTEGRASE INHIBITOR DATASET ......................................... 83
LIST OF TABLES

Table 2.1. GUI Comparison Between EvoQSAR and Other QSAR Analysis Software. ... 11

Table 4.1. Abbreviations: || Pop Size = Population Size || $T_g$ = Average Time per Generation || Std Dev = Standard Deviation || CV = Coefficient of Variation || $T_{g,serial}$ = Average Serial Component Time per Generation || $T_{g,parallel}$ = Average Parallel Component Time per Generation || $T_{computation} = $ Average Parallel Component Time Spent in Computation || $T_{computeFitness} = $ Average Time it Takes to Calculate the Fitness of One Subset || $T_{overhead} = $ Average Parallel Component Time Spend on Overhead. ................................................................. 23

Table 4.2. Measured Generation Time vs. Calculated Generation Time. .................. 32

B.1 Standardized Structures of $\beta$-Diketo Acid Compounds. ......................... 84

B.2 Dataset Division and Source Structure of $\beta$-Diketo Acid Compounds. ........... 99
LIST OF FIGURES

Figure 2.1. EvoQSAR modeling framework for drug design. Adapted from Ko et al. [26]. Evolutionary algorithms used for descriptor selection are BPSO, and DE-BPSO. Learning algorithms used are multiple linear regression (MLR), and partial least squares regression (PLSR). ......................... 3

Figure 2.2. An example of chemical descriptors computed from a chemical structure. Chemical structure shown is a representative β-diketo acid template. Adapted from Ko et al. [26]................................................. 4

Figure 2.3. Serial vs. multi-threading. ........................................................ 6

Figure 2.4. Shared memory vs. distributed memory. .................................... 7

Figure 3.1. Serial vs. parallel implementation on multiple processors (4 processors). ...... 14

Figure 3.2. Base template for graphical user interface (GUI). ......................... 16

Figure 3.3. Wizard flow of evoQSAR GUI...................................................... 18

Figure 4.1. DE-BPSO with MLR: average computation time per generation. .............. 26

Figure 4.2. DE-BPSO with MLR: speedup. .................................................. 27

Figure 4.3. DE-BPSO with MLR: efficiency. ................................................ 27

Figure 4.4. DE-BPSO with PLSR: average computation times per generation.............. 29

Figure 4.5. DE-BPSO with PLSR: speedup. .................................................. 29

Figure 4.6. DE-BPSO with PLSR: efficiency. .............................................. 30

Figure 4.7. DE-BPSO with MLR: $T_g$ standard deviation. ................................ 34

Figure 4.8. DE-BPSO with PLSR: $T_g$ standard deviation. ................................ 34

Figure 4.9. DE-BPSO with MLR: $T_{g,serial}$ standard deviation. ....................... 35

Figure 4.10. DE-BPSO with PLSR: $T_{g,serial}$ standard deviation. ................... 35

Figure 4.11. DE-BPSO with MLR: $T_{g,parallel}$ standard deviation. ..................... 36

Figure 4.12. DE-BPSO with PLSR: $T_{g,parallel}$ standard deviation. ................... 36

Figure 4.13. DE-BPSO with MLR: computation time vs. subset size over number of cores. .......................................................... 38

Figure 4.14. DE-BPSO with PLSR: computation time vs. subset size over number of cores. .......................................................... 39

Figure 4.15. GUI - welcome screen. .......................................................... 40
Figure 4.16. GUI - feature selection w/ stepwise QSAR - introduction tab with feature selection w/ stepwise QSAR selected. ........................................ 41
Figure 4.17. GUI - feature selection w/ stepwise QSAR - datasets tab with a successful import. ................................................................. 42
Figure 4.18. GUI - feature selection w/ stepwise QSAR - feature selection tab. ................................................................. 43
Figure 4.19. GUI - feature selection w/ stepwise QSAR - modeling tab. ................................................................. 44
Figure 4.20. GUI - feature selection w/ stepwise QSAR - simulation modeling success. ................................................................. 46
Figure 4.21. GUI - feature selection w/ stepwise QSAR - models found. ................................................................. 46
Figure 4.22. GUI - feature selection w/ stepwise QSAR - results tab with results. ................................................................. 47
Figure 4.23. GUI - feature selection w/ stepwise QSAR - stepwise QSAR. ................................................................. 48
Figure 4.24. GUI - single QSAR - introduction tab. ................................................................. 50
Figure 4.25. GUI - single QSAR - dataset successful import. ................................................................. 51
Figure 4.26. GUI - single QSAR - modeling tab. ................................................................. 52
Figure 4.27. GUI - single QSAR - start simulation. ................................................................. 53
Figure 4.28. GUI - single QSAR - simulation was successful. ................................................................. 53
Figure 4.29. GUI - single QSAR - correlation with results. ................................................................. 54
Figure 4.30. GUI - single QSAR - applicability domain with output filenames. ................................................................. 55
Figure 4.31. GUI - single QSAR - applicability domain with results. ................................................................. 56
Figure 4.32. GUI - single QSAR - y-randomization with results. ................................................................. 57
Figure 4.33. GUI - single QSAR - statistics with results. ................................................................. 58
Figure 4.34. GUI - single QSAR - prediction tab. ................................................................. 60
Figure 4.35. GUI - single QSAR - prediction with a successful run. ................................................................. 60
Figure 4.36. Home, back, and forward buttons. ................................................................. 61
Figure 4.37. Pan/zoom button. ................................................................. 61
Figure 4.38. Zoom-to-rectangle button. ................................................................. 61
Figure 4.39. Subplot configuration button. ................................................................. 62
Figure 4.40. Save button. ................................................................. 62
Figure A.1. Node 16 idle. ................................................................. 72
Figure A.2. EvoQSAR DE-BPSO with MLR on 1 processor. ................................................................. 73
Figure A.3. EvoQSAR DE-BPSO with MLR on 2 processor. ................................................................. 74
Figure A.4. EvoQSAR DE-BPSO with MLR on 4 processor. ................................................................. 75
Figure A.5. EvoQSAR DE-BPSO with MLR on 8 processor. ................................................................. 76
Figure A.6. EvoQSAR DE-BPSO with MLR on 16 processor. ........................................... 77
Figure A.7. EvoQSAR DE-BPSO with PLSR on 1 processor. ........................................... 78
Figure A.8. EvoQSAR DE-BPSO with PLSR on 2 processor. ........................................... 79
Figure A.9. EvoQSAR DE-BPSO with PLSR on 4 processor. ........................................... 80
Figure A.10. EvoQSAR DE-BPSO with PLSR on 8 processor. .......................................... 81
Figure A.11. EvoQSAR DE-BPSO with PLSR on 16 processor. ........................................ 82
ACKNOWLEDGMENTS

I would like to first thank the members of my committee for their service: Professors Sunil Kumar, Rajni Garg, and Mary Thomas. My sincere gratitude to my co-advisors, Professor Kumar and Professor Garg, for their continual guidance and support throughout this thesis. I want to thank Professor Thomas for her assistance in the parallel portion of my research.

I would like to thank several individuals for their contributions: Dr. Gene M. Ko for his mentorship; his willingness to give his time so generously has been very much appreciated, Professor Jose Castillo for his support and guidance as director of the Computational Science Research Center, Dr. James Otto for his assistance in computational resources, Dr. Peter Blomgren for his LaTeX wizardry, and Gregori Clarke and Jorge Dominguez for their undivided support and companionship throughout the program.

Finally, I wish to thank my parents, Chuin-Shui Su and Li-Ying Su, for their unconditional love and support throughout the years.
CHAPTER 1
INTRODUCTION

This thesis builds upon an existing drug discovery toolkit, evoQSAR, developed by Ko et al. [14] [26] to derive computational models for quantitative structure-activity relationship (QSAR) analysis. EvoQSAR, prior to this thesis, could only be run using scripts within the Python programming environment. The enhancements achieved in this thesis include the parallelization of evoQSAR to improve computational performance, and the introduction of a graphic user interface (GUI) to open the toolkit to a wider audience. The work is divided in two parts as discussed below.

1.1 PARALLEL IMPLEMENTATION

Computational models can be very time consuming to develop due to the complex nature of the problem and lack of simple analytical solutions. The first part of this thesis addresses this issue.

EvoQSAR utilizes a novel evolutionary algorithm (EA) called differential evolution-binary particle swarm optimization (DE-BPSO) [26] [24] [13]. This EA is computationally complex due to its reliance on spawning many generations in order to search for optimal solutions [12]. This thesis tackles the problem of time by introducing parallel concepts into the existing evoQSAR toolkit. Parallelization helps decrease computation time and increases efficiency through the division of independent tasks in parallel, which are normally run sequentially. The benefit of parallelization multiplies as the number of available processors increases. This thesis utilizes the Python Multiprocessing module to aggregate and distribute parallel computation tasks to available processors. Concepts touched include symmetric multiprocessing (SMP), data decomposition, multithreading, the use of multiple processors, and resource sharing. The parallelization techniques implemented in this work 1) decrease computation time, thus improving performance, 2) are easily integrable into the evoQSAR toolkit, and 3) maintain coherency and compatibility with the existing evoQSAR toolkit.

1.2 GRAPHICAL USER INTERFACE (GUI)

The script-based interface of evoQSAR introduces significant hurdles for users without a strong background in Python programming. The second part of this thesis addresses this issue.
The solution consists of a stand-alone, cross-platform, and easy to use GUI that can be installed and accessed on several operating systems (OS). The GUI developed in this work allows the evoQSAR program to reside on a local machine and guides the user through evoQSAR using a series of tabbed windows, with navigation similar to an installation wizard. The flow begins by welcoming the user with a start screen introducing evoQSAR and a short description of its features. It then breaks off into the different features, dynamically adjusting to user choices while providing supplementary instruction to eliminate the need of an extensive how-to manual. This GUI delivers a simple, easy to use portal into evoQSAR and allows for easy expansion in the future.

1.3 Test Set

This work leverages a HIV integrase inhibitor dataset compiled by Ko et al. [26] [24] [13] to benchmark the performance improvement resulting from parallelization of the evoQSAR toolkit and verify functionality of the GUI. This dataset comprises of 91 compounds from a class of chemical compounds (β-diketo acids) with inhibitory activity toward the HIV integrase protein [26] [24] [13]. The HIV dataset is optimal for testing since it is the original dataset used by Ko et al. to develop their DE-BPSO algorithm [26], and was used previously as one of the several test sets used to refine evoQSAR. This dataset is given in Appendix B.
CHAPTER 2
RELATED WORK

2.1 EVOQSAR

The QSAR modeling workflow used in evoQSAR is briefly discussed in this section; its details can be found in references [26] [24] [13].

Computational drug design is a multistep process, one of which is the development of computational models for QSAR analysis. These models are used to help identify physical-chemical properties of a class of chemical compounds conducive for its biological (inhibitory) activity towards a specific protein target. One example of a model development workflow used in evoQSAR is given in Figure 2.1.

![Figure 2.1. EvoQSAR modeling framework for drug design. Adapted from Ko et al. [26]. Evolutionary algorithms used for descriptor selection are BPSO, and DE-BPSO. Learning algorithms used are multiple linear regression (MLR), and partial least squares regression (PLSR).](image)

The workflow begins with an initial set of chemical structures that the user would like to analyze. These chemical structures are typically chemical analogs that medicinal chemists use to perform structure-activity relationship (SAR) analysis. SAR is based on the premise that any change in a chemical structure will result in either a positive or negative change in the biological activity. This allows one to identify patterns in the structure which would result in improved biological activities. A chemical structure can be quantified in terms of their physical-chemical properties, called chemical descriptors or features. Some examples of descriptors are a structure’s constitutional, topological, geometrical, electrostatic and...
quantum-chemical properties. A dataset is formed from the chemical structure’s experimental biological activity and its descriptors (Figure 2.2) [9].

![Chemical structure](image)

**Figure 2.2.** An example of chemical descriptors computed from a chemical structure. Chemical structure shown is a representative β-diketo acid template. Adapted from Ko et al. [26].

Next, computational models are developed to identify chemical descriptors which best correlate to the biological activity. As the number of chemical descriptors can be very large, a descriptor selection algorithm can be used to identify the most relevant descriptors. Ko et al. [26] [24] [13] used evolutionary algorithms for descriptor selection to identify optimal subsets of descriptors in order to develop computational models for QSAR analysis.

The evoQSAR toolkit utilizes the power of two EAs: BPSO [26], and DE-BPSO [26] for descriptor selection. Each evolutionary algorithm selects a subset of descriptors in an attempt to build a model to correlate with the biological activity in question [9]. Each subset of descriptors chosen is then used to create a computational model.

Computational models in evoQSAR are developed using regression modeling. Regression modeling is an approach of modeling the relationship between a dependent variable to one or more independent real values [9]. Two regression models used in evoQSAR include multiple linear regression (MLR) [9] and partial least squares regression (PLSR) [9]. Computational models are developed from the subset of descriptors decided by EA. Each model is then ranked according to a fitness function [12]. The chosen EA’s mechanism spawn the next generation of descriptor candidates and repeat until a predefined criteria is met. At the end of the evolutionary descriptor selection, the predictive models are analyzed from which the top ranked descriptors (chosen by frequency of occurrence) are extracted. The final model is determined by stepwise modeling using the top ranked descriptors.

Stepwise QSAR modeling iteratively takes the top ranked descriptors and calculates a model for each combination of descriptors [25]. This process starts with the top two ranked descriptors, creates a model, then adds the third ranked descriptor, creates a model, and
repeats until the list is exhausted. These models are produced along with relevant statistics (Definition 4.29-4.39) to allow the user to perform QSAR analysis.

QSAR analysis can also take the form of single QSAR which is a modified form of stepwise QSAR modeling. Instead of iteratively taking top ranked descriptors to calculate multiple models, one model is calculated from a pre-specified input of descriptors. This allows one to generate a model if the subset of descriptors is known ahead of time.

2.2 PARALLEL IMPLEMENTATION

This section provides a brief overview of widely used techniques and implementations used in parallelization, initially presenting some general computing concepts to better explain parallelization.

In terms of applications, processes and threads as: "An application consists of one or more processes. A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread" [31]. So in essence, a process is a container of threads that act like workers to deliver work to the processor. Threads are used in all processors from laptops to desktop personal computers, to supercomputer centers. Parallel computing is the simultaneous execution of these threads [10].

Multithreading is one type of parallel computing; it is when multiple threads are executed concurrently. Imagine we have a program named ProgA and ProgA only has one process, ProcessA. ProcessA is in charge of obtaining all the necessary system and user resources to run. As ProcessA runs, it creates a set of subtasks (threads) that can be scheduled and concurrently run by the operating system [4]. Each thread has access to ProcessA’s data which eliminates the need for each thread to replicate ProcessA’s resources, while also providing a global memory space for threads to communicate. Throughout the life of the parent process, threads can come and go as they are needed and finished, but the ProcessA will still remain present to ensure the necessary resources are available until the application finishes. A few common implementations of multithreading are POSIX Threads (Pthreads) [3] and Open Multi-Processing (OpenMP) [10] [43].

In the presence of multiple processors, one or more threads may run within the context of a process on a processor. On a single processor system, multiple threads can co-exist on one processor, but only one thread can use the processor at a time. If multiple processors exist, then each thread on its own respective processor can run in parallel with another thread on another processor and achieve full concurrency. An example of the differences can be seen in Figure 2.3. Figure 2.3 shows three computation cases: serial processing, multi-threading on a single processor, and multi-threading on multiple processors. Here one task consists of
one input/output (I/O) time block (in blue) and one computation block (in orange). Each of these cases assumes that the necessary tasks have already been distributed to these processes. In the serial processing case, each task is computed serially, thus taking the most time. Next is the concept of multi-threading on a single processor. One processor is only able to compute one instruction set at a time, but one can gain some level of improvement by eliminating the I/O operation time through the use of multiple threads. Notice how the I/O blocks are being distributed concurrently as the processor is computing a set of instructions, and as soon as the processor is finished with one set of instructions, it begins the next set seamlessly. The final example demonstrates the use of multiple processors to evaluate the same tasks as the other two cases. Here each processor is limited to only one thread for the sake of simplicity. Notice how each processor can process the I/O as well as the computation concurrently; this is because there are four processors now instead of just one as in the previous example, effectively overcoming the single processor bottleneck. In more advanced cases, it is also possible for each processor to spawn its own family of threads to further improve execution time.

![Figure 2.3. Serial vs. multi-threading.](image)

The concurrent use of multiple processors usually takes one of two forms: SMP or distributed computing. SMP systems are tightly-coupled systems with identical processors connected via a bus who all have equal access and access times to a globally shared memory module [4]. Distributed computing systems are loosely-coupled standalone systems that require a communication network to communicate. SMP systems are typically easier to
configure due to its processor’s ability to communicate over the global shared memory module. Distributed computing requires some form of message passing between tasks since each system has its own memory. A brief summary of shared vs. distributed memory can be seen in Figure 2.4.

Shared memory is when all the processors on a system share one global memory module, thus providing a central memory location for all connected processors [10]. Multithreading, mentioned above, uses shared memory. Shared memory allows each thread/process to communicate through the shared memory space, eliminating the need for message passing. Two problems that may limit the use of shared memory is 1) scalability, as more processors have access to the memory, performance degradation will occur; and 2) risks resulting from potential coherency issues, which is the synchronization of memory access among contending processors [10].

Distributed memory exists when memory locations are separate and distinct [10]. Distributed memory allows the system to be easily scalable, but the lack of shared memory requires the processors to communicate through some form of message passing. The use of message passing may also resolve memory contention issues that existed in the shared memory architecture. Message passing is necessary for dividing a group of instructions among several threads/processes and the aggregation of the parallelized instructions when finished. If communication is required between threads/processes during execution, message passing may also serve as a communication mechanism between different threads/processes. Several application program interfaces (API) utilized for distributed memory are the message passing interface (MPI) [30] and the global address space programming interface (GASPI) [15].

![Figure 2.4. Shared memory vs. distributed memory.](image)

A newer concept within the field of parallelization is accelerators, also known as coprocessors. Accelerators are highly specialized processors that are designed to handle
computationally intensive tasks in parallel and work in tandem with traditional CPUs to achieve faster execution times [11] [32] [42] [40]. An example would be the graphic processing units (GPUs) often used in computer graphics. Recently, GPUs have been generalized to perform high-performance computing known as general-purpose graphical processing units (GPGPUs). One commonly used framework is NVIDIA’s CUDA software development kit (SDK) [32], which provides an API to perform computing on NVIDIA’s family of GPUs (Tesla, Quadro, GeForce).

2.3 Graphical User Interface (GUI)

Many programs are available for QSAR analysis [23] [22] [6] [16] [17] [7]. A comparison between four popular software and evoQSAR for QSAR analyses is shown in Table 2.1. Note that the comparison is limited to platform used, workflow, modeling methods, and descriptor selection algorithms. The four software being compared are Roy’s Drug Theoretics and Cheminformatics (DTC) Laboratory software [23] [22], Breneman’s Research Group’s RECCR Online Modeling System (ROMS) [6], Quantitative Structure Activity Relationships Research Unit’s QSARINS software [16] [17], and CHEMBENCH created by the Carolina Exploratory Center for Cheminformatics Research at the University of North Carolina [7].

Roy’s software consists of a web-based tool, DTC MLR Validation [23], and a desktop-based tool, QSAR Tools [22]. DTC MLR Validation develops MLR-based QSAR models using the entire dataset. QSAR Tools offer a full suite of QSAR analysis tools from the calculation of the descriptors to defining the applicability domain once final models have been obtained. Each tool in this suite is its own stand-alone software. Each software is developed and run in the Java Runtime Environment (JRE). The available modeling algorithms are MLR and genetic algorithms (GA). Both web-based and desktop-based software are organized similarly; all configurable options are provided to the user on one screen, and then the user will click on a button to begin the calculation for the associated tool. For comparison, evoQSAR covers a wider breadth of algorithms beyond MLR and GA, but may not offer a suite of analysis tools as comprehensive as Roy’s.

ROMS [6] provides a similar interface to Roy’s web-based DTC MLR Validation. It is accessed through a website and each feature of the software is arranged where all configurable options are provided to the user on one screen and initiated with a single button. ROMS provides a different user experience; instead of different modules for different features, the features are arranged in a tab-like form with each tab serving a specific feature. ROMS offer several modeling methods: PLS, which evoQSAR supports, and two additional modeling
methods: Kernel-PLS and support vector regression (SVR) [9] which evoQSAR does not support. ROMS also lacks descriptor selection capabilities.

QSARINS is a software for the development and validation of MLR models and uses GA for descriptor selection [16] [17]. It is distributed as a desktop-based software and offers multiple windows for users to configure different feature sections such as data setup, parameters for a single model, and database parameters. The user experience here is closer to evoQSAR than the previously mentioned, but still differs in the amount of information presented at one time. The evolutionary algorithm used is limited to GA and the modeling method used is limited to MLR making QSARINS similar to Roy’s QSAR Tools. QSARINS, however, does offer an in-window view of the dataset which is not presently offered in evoQSAR.

CHEMBENCH is a software provided through a web-like portal with different tabs for different features such as a history for past and current jobs, dataset input, modeling parameters, and predictor selection [7]. CHEMBENCH offers some modeling algorithms that evoQSAR doesn’t, which include K-Nearest Neighbor (KNN) [9], KNN-GA [9], KNN-simulated annealing (SA) [9], Random Forest (RF) [9], and support vector machine (SVM) [9]. CHEMBENCH also incorporates a version of GA for descriptor selection which works in conjunction with KNN.

EvoQSAR offers new features in addition to the existing available QSAR analysis tools mentioned above. It introduces two additional descriptor selection algorithms, BPSO and DE-BPSO.

Most of the toolkits previously discussed follow a trend that displays all the configuration parameters on a single screen and separates its software functionality into modules or tabs. The work in this thesis takes a new approach, realizing that a single screen method results in a cluttered and sometime overwhelming user interface, the evoQSAR GUI focuses on simplicity and a user-centric flow. Instead of displaying all parameters on a single page, parameters are separated into several dynamically enabled tabbed pages to reduce clutter, and streamline only what may be important to the user. The wizard-like interface with a 'back' and 'next' navigational feel also allows one to perform all the necessary steps in one coherent fashion instead of having to run separate modules one after another. Tabs are dynamically enabled and disabled based on the user’s input, thus guiding the user to relevant settings for his or her run configuration. Along with breaking up the single screen method, help documentation is also integrated within the GUI to eliminate the need of additional documents. Each step embeds necessary instructions for that specific page, specifically tailoring it to the GUI feature at hand. This effectively eliminates the steep learning curve and traditional process of having to read a comprehensive help file, much of which could be
unnecessary, before using the program itself. A desktop approach is also taken to allow users
to execute the software locally on their machines. This effectively eliminates the reliance on
outside services such as an internet service provider, web host, and remote servers, as well as
alleviates potential delays due to wait-times for data processing on a remote server, and theft
of data [5] [29].
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform</strong></td>
<td>Desktop</td>
<td>Web</td>
<td>Desktop</td>
<td>Web</td>
<td>Desktop</td>
<td>Web</td>
</tr>
<tr>
<td><strong>Flow</strong></td>
<td>Comprehensive Wizard</td>
<td>Single Screen</td>
<td>Single Screen - Module-Based</td>
<td>Single Screen - Tab-Based</td>
<td>Single Screen - Pop-Up Windows</td>
<td>Single Screen - Tab-Based</td>
</tr>
<tr>
<td><strong>Modeling Methods</strong></td>
<td>MLR, PLSR</td>
<td>MLR</td>
<td>MLR</td>
<td>PLSR, K-PLSR, SVR</td>
<td>MLR</td>
<td>KNN, KNN-GA, KNN-SA, Random Forest, SVM</td>
</tr>
<tr>
<td><strong>Descriptor Selection Algorithm</strong></td>
<td>BPSO, DE-BPSO</td>
<td>None</td>
<td>GA</td>
<td>None</td>
<td>GA</td>
<td>GA (only in conjunction with KNN)</td>
</tr>
</tbody>
</table>
CHAPTER 3
RESEARCH METHODS

3.1 DEVELOPMENT ENVIRONMENT

Work conducted for this thesis includes significant enhancements of the evoQSAR toolkit through parallelization and GUI development. Enhancements are coded in cPython 2.7, which is the C implementation of the Python programming language to maintain consistency and coherency with the original evoQSAR toolkit. The original development environment used by Ko et al. [14] included the use of Anaconda which is a “completely free enterprise-ready python distribution for large-scale data processing, predictive analytics, and scientific computing” [8], PyCharm, an integrated development environment (IDE) used to interface, modify, and execute the code [21], Bitbucket, as a Git repository for source control [2] and SourceTree which is a free Git client for windows to interface with Git [1].

In addition to the original development environment of evoQSAR, PyQt [38] and Qt Designer [37] were added to aid in GUI development. PyQt provides python bindings of Qt which is a cross-platform application framework used for developing application software. Qt includes a rich collection of GUI widgets and simplifies the creation of re-usable software components with their signal/slot mechanisms. PyQt allows us to utilize the power of Qt while maintaining coherency with the existing python code. Qt Designer is Qt’s tool for designing and building GUIs from Qt components which comes pre-installed with Qt. Qt Designer allows one to focus more on the design by providing widgets with configuration options to connect back to the programmed code.

3.2 PARALLELIZATION

The parallelization of evoQSAR is implemented using the Python Multiprocessing module [35] and performed on the descriptor selection/computational models section of the evoQSAR modeling framework (Figure 2.1).

A brief demonstration of the parallel implementation can be seen in Figure 3.1. During the descriptor selection/computational models section of evoQSAR, multiple subsets are selected. Each subset then undergoes computations that involve evolving the subset values by the EA and then modeled by a modeling algorithm. Each computation performed on a subset is independent from other subsets making this portion of the code perfect for parallelization. The independent nature of the subset computation is important. The ability to parallelize a problem depends heavily on ability to partition the problem into discrete
chunks [4]. There are two general ways of partitioning the data into parallel tasks: domain decomposition or functional decomposition. Domain decomposition partitions the data associated with the problem while functional decomposition focuses on the work that must be done. The subset computation, being independent, makes it an apt candidate for domain decomposition. The act of solving multiple independent tasks concurrently with little to no coordination is known as embarrassingly parallel [44]. To further explain, instead of each subset computation being performed sequentially, as it is now, each subset computation can be allocated respectively to each available processor and computed in parallel. This decreases the computation time and increase the efficiency of the system. An example of sequential vs. parallel computation and the potential time savings can be seen in Figure 3.1. Definitions of time sequential and time parallel can be found in Definitions 3.1-3.2.

Definition 3.1. Time sequential ($T_{seq}$) is the time it takes a system to compute a set of tasks sequentially, or in serial, usually consisting of only one processor.

Definition 3.2. Time parallel ($T_p$) is the time it takes a system to compute a set of tasks in parallel, usually consisting of more than one processor.

The aforementioned parallel implementation has several hidden requirements: EvoQSAR is designed to be run locally on a user’s machine, which are typically SMP systems. At first glance, there are two contenders, the Python Threading module [34] and the Python Multiprocessing module [35]. Both are capable of leveraging SMP systems, and are already baked into the standard cPython distribution which reduces the installation footprint, ensures compatibility with evoQSAR’s codebase and simplifies integration. The difference lies in how they are implemented.

The fundamental difference is that the Python Threading module uses multiple threads while the Python Multiprocessing module uses multiple processes. Both have their suitable uses. Additional threads are generally more lightweight than additional processes and since the additional threads all reside within one process, the threads all share the same memory space, but due to the nature of how threads operate, the possibility of memory contention issues and race conditions increase dramatically. To prevent these issues from happening, cPython implemented something called the global interpreter lock (GIL) [36] which only allows for one computation-bound thread to execute at one time. The Python Threading module is typically only used for I/O bound tasks since the bottleneck originates from the delays associated with overhead processes and not from the use of the processor. The parallel implementation for evoQSAR is not I/O bound but computation-bound, or CPU-bound, which
Figure 3.1. Serial vs. parallel implementation on multiple processors (4 processors).
means evoQSAR would receive little to no improvement from a multithreading implementation.

The creation of additional processes is typically heavier than threads due to the additional overhead it requires. The Python Multiprocessing module spawns additional processes through the 'fork-join' approach: Forking allows an existing process to spawn an exact copy of itself along with a unique ID and its own copy of the parent’s variables which adds additional overhead when compared to threads. Spawned processes are synchronized with its parent process through the 'join' command which indicates whether a process has been finished or not. All slave processes must be finished before the parent process can continue [44]. Each additional process spawned by the Python Multiprocessing module is another instance of the Python interpreter and this is how the Python Multiprocessing module by-passes the GIL limitation.

Sections of the overhead can be eliminated through the use of shared memory, global variables, and the 'fork-join' approach. The 'fork-join' approach establishes the relationship between child processes and its parent. The use of global variables on a shared memory architecture enables each child process to access its parent’s variables. This eliminates the overhead associated with duplicating the parent’s variables on each child process. Normally separate processes require a communication mechanism to communicate, but since our implementation is embarrassingly parallel, the overhead due to communication is eliminated as well.

This thesis leverages the 'Pool()' class from the Python Multiprocessing module. The 'Pool()' class is often used on domain decomposed problems. It acts as a manager between the number of workers and the tasks at hand. In this case, workers are the number of processors available on the hardware and the tasks at hand are the computations required to develop a computational model from a respective subset. 'Pool()' then distributes the tasks to available workers and as each task is finished, 'Pool()' collects each result and repeats until all tasks are completed and returns the aggregated results in a form of a list. The 'fork-join’ process is only available to Linux and OS X so our parallel implementation is limited to those OS’s as well. The Python Multiprocessing module’s ability to bypass the GIL through the use of the 'fork-join’ approach in conjunction with the use of global variables on a share memory architecture allows evoQSAR to benefit from multiple processors on a system.

### 3.3 GUI

EvoQSAR, in its original state did not have a user-friendly GUI and could only be accessed through a code-based structure written in Python which severely limits the number of potential users. Users would need to have knowledge of the Python programming language.
in order to use evoQSAR. This potentially excludes chemists, other fellow researcher, and industry who are not familiar with Python to use evoQSAR. This thesis addresses this issue by packaging evoQSAR with a simple, user-friendly GUI to open its functionalities to a wider audience.

The GUI serves as a stream-lined guide for one to utilize the full capabilities of evoQSAR. The general flow of the program follows that of an installation wizard with the general template shown in Figure 3.2. The template is designed to be simple and user intuitive. The 'title’ section indicates what the user is currently looking at, while user selections, parameter inputs, and help content are displayed in the 'content' section. Navigation throughout the GUI is controlled through the 'back' and 'next' buttons or through directly clicking through the tabs. In the beginning, the user may find certain tabs disabled and this is done to 1) simplify the user experience and 2) limit potential confusions that arise from an overwhelming influx of information. Tabs are dynamically enabled once the correct selection / input parameters are selected.

Figure 3.2. Base template for graphical user interface (GUI).
The flow of the GUI is shown in Figure 3.3. The program first welcomes the user with an introduction screen explaining what the program is, a short set of instructions notes and allows the user to select one of two modes of evoQSAR, feature selection with stepwise QSAR or single QSAR. Within the GUI, simple and advanced modes are merged through prepopulating default parameters and embedding supplementary help information through side windows and tooltips. This offers users a coherent experience no matter what their knowledge level is. The prepopulating of fields provides basic users indirect feedback on what the minimal necessary parameters are to invoke a simulation while allowing more advanced users to override default parameters to suit their simulation needs. An embedded side window may appear along with tooltips which provides basic users with help documentation without overcrowding the user experience space for more advanced users. Users are also offered the ability to choose input and output files/file locations through the user’s native system prompt to simplify the selection while also preventing potential confusing. A more in-depth description of the GUI’s functionality and operation can be found in Section 4.2.

The implementation of the GUI consists of leveraging a popular GUI development framework called PyQt [38] [41] to create a stand-alone graphical user interface to enable a simple, easy to use, portal into evoQSAR. PyQt provides Python bindings to the Qt framework [37], a widely used, cross-platform application development framework. This enables evoQSAR to be run on multiple OS’s, including mobile and web, which makes this software very expandable in the future. It is important to note that for the scope of this thesis, the GUI is configured to run only on Windows, Linux, or OS X OS’s. PyQt is licensed under the GNU General Public License (GPL) which allows the software to be distributed freely as open-source. This removes the potential financial constraints of licensing fees for the development and use of evoQSAR.
Figure 3.3. Wizard flow of evoQSAR GUI.
CHAPTER 4
RESULTS

4.1 PARALLELIZATION OF EVOQSAR

The parallelization of evoQSAR was decomposed through domain decomposition and optimized to run on a SMP system. Testing was performed on node16 on the computational cluster, cinci.sdsu.edu, available at the Computational Sciences Research Center (CSRC). Node16 houses a dual octa-core configuration consisting of two Intel Xeon E5-2650 processors rated at 2.60 GHz with a total of 64 GB RAM. The cinci cluster contains 17 distinct nodes. Node16 was chosen due to its availability and affinity toward testing requirements at the time of benchmark (process load of Node16 while idle and running can be found in Appendix A).

To characterize the impact of the parallelization, the performance of the feature selection component of evoQSAR was benchmarked. This was the only section parallelized in the parallel implementation and tested against two test cases: DE-BPSO with MLR and DE-BPSO with PLSR, with 1, 2, 4, 8, and 16 cores. Testing was conducted using the following values (defined in Definitions 4.15-4.28):

- Dataset
  - HIV integrase inhibitor dataset (for more information, please refer to Section 1.3)

- Feature Selection Parameters
  - DE-BPSO Parameters
    * Alpha = (0.5, 0.33)
    * Beta = 1.5
    * F = 0.8
    * CR = 0.9
  - Initial Parameters
    * Initial Features = 10
      · Note: For the HIV dataset, the optimum number of selected features ranges between 8-12 features [24] [13].
    * Population Size = 40
      · Note: Since the benchmark testing is performed against a maximum number of 16 processors, a population size of 40 was selected to allow for
an accurate measurement (for more information, please refer
Definition 4.11)

– Stopping Criteria
  ∗ Max Generations = 100
  ∗ Max Models = 10,000
  · Note: The testing should stop at the max number of generations; here 10,000 is selected so that the max models stopping criteria is never reached.

– Penalty Factor = 2

• Modeling Parameters
  – Cross-Validation = Leave-one-out
  – PLSR specific parameters
    ∗ Number of latent components = 6

The goal of the parallelization is to decrease the computation time and increase the efficiency of the evoQSAR toolkit. Additional timing code was strategically inserted to retrieve validation metrics to benchmark the parallelization. Processed results are discussed first followed by specifics and a discussion on the validity of the metrics obtained. Metrics are defined in Definitions 4.1-4.10 and results are shown in Table 4.1. Note that the results are the averaged values over 100 runs.

**Definition 4.1.** *Time generation* ($T_g$) *is the average time it takes to generate and calculate the fitness of each subset for each generation (measured). It can be decomposed into two components: Time serial* ($T_{g,\text{serial}}$) *and Time parallel* ($T_{g,\text{parallel}}$).

\[ T_g = T_{g,\text{serial}} + T_{g,\text{parallel}} \]  

**Definition 4.2.** *Time serial* ($T_{g,\text{serial}}$) *is the serial component of* $T_g$; *the average time spent on serial computation per generation (measured).*

**Definition 4.3.** *Time parallel* ($T_{g,\text{parallel}}$) *is the parallel component of* $T_g$; *the average time spent on parallel computation per generation (measured).*
Definition 4.4. $T_{\text{computation}}$ is the average time it takes to compute a generation in parallel. $T_{\text{computation}}$ can be calculated by:

$$T_{\text{computation}} = (T_{\text{computeFitness}}) \times \text{CEIL}(\frac{\text{Subset Size}}{\# \text{ Processor}})$$  \hspace{1cm} (4.2)

Definition 4.5. $T_{\text{computeFitness}}$ is the average time it takes to calculate the fitness of one subset (measured).

Definition 4.6. $T_{\text{overhead}}$ is the average time spent on overhead per generation in parallel processes. $T_{\text{overhead}}$ can be calculated by:

$$T_{\text{overhead}} = T_{g,\text{parallel}} - T_{\text{computation}}$$  \hspace{1cm} (4.3)

Definition 4.7. Speedup is a quantitative measure of improvement gained through parallelism. Speedup can be calculated by:

$$S_p = \frac{T_{\text{seq}}}{T_p}$$  \hspace{1cm} (4.4)

$S_p = $ Speedup  
$T_{\text{seq}} = $ Execution time without enhancement [T sequential: (p=1)]  
$T_p = $ Execution time with enhancement [T parallel]  
$p = $ Number of processors

Definition 4.8. Efficiency is a quantitative measurement of how well processors were utilized. Efficiency can be calculated by:

$$E_p = \frac{S_p}{p}$$  \hspace{1cm} (4.5)

$S_p = $ Speedup  
$E_p = $ Efficiency  
$p = $ Number of processors

Definition 4.9. Standard Deviation is a quantitative measurement of how spread out the results are. Standard Deviation can be calculated by:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2}$$  \hspace{1cm} (4.6)
Definition 4.10. Coefficient of Variation (CV) [19] is a quantitative measurement of how much variance exists between the standard deviations. As a rule of thumb: \( CV \geq 1 \) indicates relatively high variation; \( CV < 1 \) indicates low variation. \( CV \) can be calculated by:

\[
CV = \frac{\sigma}{\mu}
\]  

(4.7)

\( \sigma = \) Standard Deviation
\( \mu = \) Mean
Table 4.1. Abbreviations:  

- **Pop Size** = Population Size  
- **\( T_g \)** = Average Time per Generation  
- **Std Dev** = Standard Deviation  
- **CV** = Coefficient of Variation  
- **\( T_{g,\text{serial}} \)** = Average Serial Component Time per Generation  
- **\( T_{g,\text{parallel}} \)** = Average Parallel Component Time per Generation  
- **\( T_{\text{computation}} \)** = Average Parallel Component Time Spent in Computation  
- **\( T_{\text{computeFitness}} \)** = Average Time it Takes to Calculate the Fitness of One Subset  
- **\( T_{\text{overhead}} \)** = Average Parallel Component Time Spend on Overhead.

<table>
<thead>
<tr>
<th># Processors</th>
<th>Modeling Method</th>
<th>Pop Size</th>
<th>( T_g ) Std Dev</th>
<th>( T_g ) CV</th>
<th>( T_{g,\text{serial}} ) Std Dev</th>
<th>( T_{g,\text{serial}} ) CV</th>
<th>( T_{g,\text{parallel}} ) Std Dev</th>
<th>( T_{g,\text{parallel}} ) CV</th>
<th>( T_{\text{overhead}} ) Std Dev</th>
<th>( T_{\text{overhead}} ) CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MLR</td>
<td>40</td>
<td>0.9992</td>
<td>0.0321</td>
<td>0.0322</td>
<td>0.0021</td>
<td>0.0209</td>
<td>9.8020</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>MLR</td>
<td>40</td>
<td>0.6110</td>
<td>0.0337</td>
<td>0.0552</td>
<td>0.0072</td>
<td>0.0702</td>
<td>9.8038</td>
<td>0.6035</td>
<td>0.0338</td>
</tr>
<tr>
<td>4</td>
<td>MLR</td>
<td>40</td>
<td>0.3598</td>
<td>0.0490</td>
<td>0.1360</td>
<td>0.0063</td>
<td>0.0616</td>
<td>9.8040</td>
<td>0.3531</td>
<td>0.0490</td>
</tr>
<tr>
<td>8</td>
<td>MLR</td>
<td>40</td>
<td>0.2278</td>
<td>0.0106</td>
<td>0.0464</td>
<td>0.0060</td>
<td>0.0585</td>
<td>9.8057</td>
<td>0.2214</td>
<td>0.0103</td>
</tr>
<tr>
<td>16</td>
<td>MLR</td>
<td>40</td>
<td>0.2334</td>
<td>0.0266</td>
<td>0.1140</td>
<td>0.0051</td>
<td>0.0502</td>
<td>9.8061</td>
<td>0.2278</td>
<td>0.0265</td>
</tr>
<tr>
<td>1</td>
<td>PLSR</td>
<td>40</td>
<td>2.4237</td>
<td>0.0210</td>
<td>0.0088</td>
<td>0.0021</td>
<td>0.0208</td>
<td>9.8020</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>PLSR</td>
<td>40</td>
<td>1.3171</td>
<td>0.0140</td>
<td>0.0106</td>
<td>0.0070</td>
<td>0.0690</td>
<td>9.8043</td>
<td>1.3100</td>
<td>0.0140</td>
</tr>
<tr>
<td>4</td>
<td>PLSR</td>
<td>40</td>
<td>0.8169</td>
<td>0.0202</td>
<td>0.0247</td>
<td>0.0061</td>
<td>0.0596</td>
<td>9.8044</td>
<td>0.8104</td>
<td>0.0199</td>
</tr>
<tr>
<td>8</td>
<td>PLSR</td>
<td>40</td>
<td>0.5130</td>
<td>0.0369</td>
<td>0.0719</td>
<td>0.0062</td>
<td>0.0609</td>
<td>9.8052</td>
<td>0.5064</td>
<td>0.0368</td>
</tr>
<tr>
<td>16</td>
<td>PLSR</td>
<td>40</td>
<td>0.3349</td>
<td>0.0191</td>
<td>0.0571</td>
<td>0.0052</td>
<td>0.0505</td>
<td>9.8067</td>
<td>0.3294</td>
<td>0.0188</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># Processors</th>
<th>Speedup</th>
<th>Efficiency</th>
<th>( T_{\text{computation}} ) Std Dev</th>
<th>( T_{\text{computeFitness}} ) Std Dev</th>
<th>( T_{\text{overhead}} ) Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>0.0249</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>1.6352</td>
<td>0.8176</td>
<td>0.5032</td>
<td>0.0252</td>
<td>0.1003</td>
</tr>
<tr>
<td>4</td>
<td>2.7770</td>
<td>0.6943</td>
<td>0.2626</td>
<td>0.0263</td>
<td>0.0905</td>
</tr>
<tr>
<td>8</td>
<td>4.3864</td>
<td>0.5483</td>
<td>0.1491</td>
<td>0.0298</td>
<td>0.0724</td>
</tr>
<tr>
<td>16</td>
<td>4.2816</td>
<td>0.2676</td>
<td>0.1079</td>
<td>0.0360</td>
<td>0.1200</td>
</tr>
<tr>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>0.0605</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>1.8402</td>
<td>0.9201</td>
<td>1.2159</td>
<td>0.0610</td>
<td>0.0938</td>
</tr>
<tr>
<td>4</td>
<td>2.9670</td>
<td>0.7418</td>
<td>0.6296</td>
<td>0.0630</td>
<td>0.1810</td>
</tr>
<tr>
<td>8</td>
<td>4.7243</td>
<td>0.5905</td>
<td>0.3391</td>
<td>0.0678</td>
<td>0.1673</td>
</tr>
<tr>
<td>16</td>
<td>7.2365</td>
<td>0.4523</td>
<td>0.2100</td>
<td>0.0700</td>
<td>0.1194</td>
</tr>
</tbody>
</table>
4.1.1 DE-BPSO with MLR

Figure 4.1 shows that the parallel implementation was a success; the computation time decreased with the number of processors and scales to approximately 16 processors. The generation runtime ($T_g$) shows that there is significant improvement going from one processor to two, with slightly less improvement from two processors to four processors to eight processors and a decrease in performance going from eight processors to sixteen processors.

Changing our focus a bit, we can see that the serial component of the runtime ($T_{g,\text{serial}}$) stays relatively low compared to parallel component of the runtime ($T_{g,\text{parallel}}$) which demonstrates that the domain decomposition of the problem was done well. Also, the parallel component times stay fairly close to generation time throughout the family of processors, which further supports that the domain decomposition of the problem was done well.

An interesting perspective to point out would be the results from eight processors to sixteen. If one only looked at the generation time, one would see that the time actually increased instead of decreased. This can be explained by breaking down the parallel component into its respective components, the computation time ($T_{\text{computation}}$) and overhead time ($T_{\text{overhead}}$). $T_{\text{computation}}$ is the amount of time spent calculating each subset, while $T_{\text{overhead}}$ accounts all other items not pertaining to computation; including communication time between the master and slave nodes, activating and deactivating processes, or the time it takes to retrieve and write data, so essentially $T_{g,\text{parallel}} = T_{\text{computation}} + T_{\text{overhead}}$. It is important to note that, for each generation, Python processes are activated and killed as part of the parallel implementation and this act of activating and killing may be one of the more significant contributions to $T_{\text{overhead}}$. As we move from eight processors to sixteen, we can see $T_{\text{computation}}$ continue to decrease which indicates that we are still getting performance gains from the computation. However, we also see the start of $T_{\text{overhead}}$ increasing, which effectively negates the gains obtained from $T_{\text{computation}}$ and results in a slightly decrease in overall performance. This phenomenon can be explained through the concept of granularity and Amdahl’s law [10] [4]. Granularity is the qualitative measure of the ratio of computation to communication. Granularity can be either fine-grain or coarse-grain parallelism. Fine-grain parallelism exists when ”relatively small amounts of computational work are done between communication events”, and coarse grain parallelism exists when ”relatively large amounts of computational work are done between communication/synchronization events” [4]. Our benchmark testing fits into the fine-grain parallelism category and ”if the granularity becomes too fine, it becomes possible that the overhead required for communications and synchronization between tasks takes longer than the computation” [4] which results in a decrease in overall performance as we saw earlier.
Amdahl’s law states that one of the reasons that ideal speedup may not be obtained is in part due to the fact that portions of the code can be inherently sequential; the sequential portions of the code also consists of accumulated overhead due to I/O processes, synchronizations, and other communication hindrances [10]. Speedup is quantitative measurement often used to evaluate how much improvement a parallel implementation yields. Speedup is defined in Definition 4.7. Ideal speedup occurs when processors are utilized 100%. In theory, this is possible, but in practice, this is far from reality, but one should be able to see that as the number of processors increases, the potential improvement will decrease and any serial part of the computation will eventually dominate. Speedup results for DE-BPSO with MLR can be seen in Figure 4.2.

While speedup evaluates how much improvement a parallel implementation yields, efficiency evaluates how well each processor is utilized. Efficiency is defined in Definition 4.8. Efficiency, although similar to speedup, provides another perspective on the same results. Speedup focuses on the improvement aspect while efficiency focuses on the utilization of each processor. An example of this may be seen in a business environment. The CEO of the business may only be interested in the speedup metric as he only cares if his investment yielded a positive return, whereas a CFO may be interested in the efficiency metric instead as he/she needs to figure out the cost effectiveness of the implementation, either in the utilization of the additional purchased hardware or the labor cost of the development of the implementation with respect to how well the existing resources were utilized. Efficiency results for DE-BPSO with MLR are shown in Figure 4.3.

If we take a look at the speedup and efficiency results in Figure 4.2 and Figure 4.3, respectively, we can see that the speedup gradually increases from two to eight processors, but tapers off from eight to sixteen processors. From two to four processors, there is about a 57% increase in speedup per processor; from four to eight processors, there is about a 40% increase in speedup per processor; and from eight to sixteen, there is a 1% decrease in speedup. Looking at the efficiency graph, we can see the efficiency starts to decrease at a quicker rate once it hits eight processors; From two to four processors, there is about a 6% decrease in efficiency per processor, from four to eight processors, there is about a 3.7% decrease in efficiency per processor, and from eight to sixteen processors, there is about a 3.5% decrease in efficiency per processor. This falls in line with our findings from Figure 4.1.

The decrease in efficiency, in certain cases, could also be attributed to a non-optimized scaling of the problem. One facet of scaling is the concept of computation cycles.

**Definition 4.11.** A computation cycle can be defined as:

\[
\text{Number of Computation Cycles} = \frac{\text{Subset Size}}{\text{Number of Processors}} \quad (4.8)
\]
Given a population size of 40 (initial benchmark parameter), our results of one, two, four, and eight processors are the only cases where this component of performance is fully optimized. In the case of sixteen processors, we get $40/16 = 2.5$ number of computation cycles, but in theory 3 would be taken since there is no such thing as half a cycle. This means that some efficiency would be wasted when running with sixteen processors.

One should also note the configuration of the cinci.sdsu.edu cluster and node16. How a node is configured along with how processes are distributed onto available cores may also play a part in how efficiently one can utilize a system. In this thesis, we assume that the Portable Batch System (PBS) interface on the cinci cluster will allow us to utilize the number of processors, among other configurations, pre-specified in our batch script for benchmarking. However, aside from monitoring the process load on the node (see Appendix A), we did not have an explicit way of verifying what ran on what core.

Figure 4.1. DE-BPSO with MLR: average computation time per generation.
Figure 4.2. DE-BPSO with MLR: speedup.

Figure 4.3. DE-BPSO with MLR: efficiency.
4.1.2 DE-BPSO with PLSR

DE-BPSO with PLSR displayed similar results to DE-BPSO with MLR as seen in Figure 4.4; both test cases did benefit from the parallelization, and also experienced a slight tapering of efficiency as the number of processors grew. The parallel component of the runtime ($T_{g,parallel}$) stayed close to the generation runtime ($T_g$) and serial component of the runtime ($T_{g,serial}$) remains relatively low when compared to parallel component of the runtime ($T_{g,parallel}$) which further supports that the domain decomposition of the problem was done well. PLSR, when compared to MLR is a more complicated modeling method which results in a longer computation time. Comparing the serial component runtimes, PLSR took 1.42 seconds or 143% longer than MLR. Throughout the family of processors, PLSR took a little over two times the time it took to compute the parallel portion of the code ($T_{g,parallel}$ and $T_{computation}$), while maintaining roughly the same serial component runtimes ($T_{g,serial}$). To further support this claim, one can take a look at $T_{computeFitness}$ in Table 4.1 and see that the average time it took to compute an individual subset is roughly a little under 1.5 times longer in PLSR as compared to MLR. This longer computation time effectively provides a coarser-grain parallelism run that should delay the substantial increase in $T_{overhead}$, as seen in MLR.

Figure 4.4 shows the results of DE-BPSO with PLSR. Here we can see an overall decrease in computation time while an increase in efficiency throughout the entire family of processors tested, from one to sixteen processors. We can also notice that the performance gain did not stop at eight processors like in the MLR test case. Instead of a 1% decrease in speedup, there was a 31% increase in speedup per processor which means that there is still room for improvement if more processors were available.

Speedup results are shown in Figure 4.5 and efficiency results in Figure 4.6. Here we can see that the speedup is improving consistently as the number or processors increases; from two to four processors, there is about a 56% increase in speedup per processor, from four to eight processors, there is about a 44% increase in speedup per processor, and from eight to sixteen, there is about a 31% increase in speedup. From the efficiency standpoint, the graph may show a linear progression but one should keep in mind that the incremental marks on the x-axis are not evenly spaced apart, jumping from one to two to four to eight to sixteen processors per step size. With that in mind, the decrease in efficiency per processor is 8.9%, 3.8% and 1.7% respectively. The gaps between ideal and experimental results for speedup and efficiency can be attributed to the aforementioned Amdahl’s law, overhead, granularity, and non-optimized scaling.
Figure 4.4. DE-BPSO with PLSR: average computation times per generation.

Figure 4.5. DE-BPSO with PLSR: speedup.
Figure 4.6. DE-BPSO with PLSR: efficiency.
4.1.3 Validity

The results presented in this section validate our results displayed in Table 4.1 and Figures 4.1-4.6, through comparing measured vs. calculated values, standard deviation and coefficient of variation, and addressing variability within each run.

4.1.3.1 Measured vs. Calculated Values

In this section, we will discuss the difference between measured and calculated values as one way to validate our results. Additional metrics used are defined in Definition 4.12-4.14.

Definition 4.12. The measured generation runtime should be equivalent to the measured parallel component of the runtime plus the measured serial component of the runtime.

\[ T_{g,\text{calc}} = T_{g,\text{parallel}} + T_{g,\text{serial}} \]  (4.9)

- \( T_{g,\text{calc}} \) = Calculated generation runtime
- \( T_{g,\text{parallel}} \) = Measured parallel component of the runtime
- \( T_{g,\text{serial}} \) = Measured serial component of the runtime

Definition 4.13. The error is calculated by taking the difference between the measured value and the calculated value.

\[ Error = T_g - T_{g,\text{calc}} \]  (4.10)

- \( Error \) = Error between measured and calculated generation runtime
- \( T_g \) = Measured generation runtime
- \( T_{g,\text{calc}} \) = Calculated generation runtime

Definition 4.14. The percent error is calculated by the error divided by the measured value multiplied by 100.

\[ \text{Percent Error} = \frac{Error}{T_g} \times 100 \]  (4.11)

- \( \text{Percent Error} \) = Percent error between measured and calculated generation runtime
- \( Error \) = Error between measured and calculated generation runtime
- \( T_g \) = Measured generation runtime
Table 4.2. Measured Generation Time vs. Calculated Generation Time.

<table>
<thead>
<tr>
<th># Proc</th>
<th>$T_g$</th>
<th>$T_{g,\text{calc}}$</th>
<th>Err.</th>
<th>Percent Err.</th>
<th>$T_g$</th>
<th>$T_{g,\text{calc}}$</th>
<th>Err.</th>
<th>Percent Err.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.6110</td>
<td>0.6106</td>
<td>0.0004</td>
<td>0.0662</td>
<td>1.3171</td>
<td>1.3167</td>
<td>0.0004</td>
<td>0.03034</td>
</tr>
<tr>
<td>4</td>
<td>0.3598</td>
<td>0.3594</td>
<td>0.0004</td>
<td>0.1106</td>
<td>0.8169</td>
<td>0.8165</td>
<td>0.0004</td>
<td>0.0469</td>
</tr>
<tr>
<td>8</td>
<td>0.2278</td>
<td>0.2274</td>
<td>0.0004</td>
<td>0.1838</td>
<td>0.5130</td>
<td>0.5126</td>
<td>0.0004</td>
<td>0.0850</td>
</tr>
<tr>
<td>16</td>
<td>0.2334</td>
<td>0.2330</td>
<td>0.0004</td>
<td>0.1780</td>
<td>0.3349</td>
<td>0.3345</td>
<td>0.0004</td>
<td>0.1249</td>
</tr>
</tbody>
</table>

Table 4.2 shows the measured generation runtime, the calculated generation runtime and the error and percent error for each test case. Error is defined in Definition 4.13 while percent error is defined in Definition 4.14. From the calculations shown in the percent error column of Table 4.2, we can see that the percent error is well within an acceptable range with a max percent error of 0.18% showing the average results used in this benchmark is valid.

4.1.3.2 Standard Deviation and Coefficient of Variation

The standard deviation is a quantitative measurement of how widely dispersed the results are; in essence it helps in determining how realistic the calculated mean is with respect to the results. Low values of standard deviation means that all of individual results were tightly grouped around the mean, whereas high standard deviation means that the individual results would be widely dispersed potentially indicating that the confidence value of the respective mean is fairly low. The standard deviation is defined in Definition 4.9.

Figures 4.7-4.12 show the respective calculated average mean with error bars that indicate +/- one standard deviation away from the mean for the respective metric. One standard deviation is chosen as a simple way to compare the validity of using the average calculated mean values of $T_g, T_{g,\text{serial}},$ and $T_{g,\text{parallel}}$ for analysis. According to the ‘three-sigma rule of thumb’, one standard deviation away from the mean roughly indicates that approximately 68% or two-thirds of the data values used to calculate the mean should reside within one standard deviation away from the mean.

The coefficient of variation (CV) can be used as another metric to compare how much variance exists between the standard deviations and can also be considered as a normalized version of the standard deviation allowing direct comparisons between different standard deviations. As a rule of thumb $CV \geq 1$ indicates relatively high variation, while $CV < 1$ indicates low variation. CV is defined in Definition 4.10.

Figure 4.7 and Figure 4.8 show the standard deviation of the time it took to calculate one generation ($T_g$) over the span of 100 generations, or data points, for its respective modeling method. Looking at the results, we can see that the standard deviation overall
remains fairly small indicating that the runtimes for each generation for its respective modeling method were tightly grouped around the calculated average mean of the generation runtime, i.e., the mean is an accurate representation of the raw data. A general improvement can still be seen even in the worst case scenarios i.e., in Figure 4.7, there is still an improvement when comparing the quickest time for four processors vs. the slowest time for eight processors. CV for this group of data are all well below the threshold of 1, with the max coefficient of variation being 0.136 which also happens to be the worst standard deviations for $T_g$ with a value of 0.0489 which is fairly small and well within the acceptable standards of the model (MLR $T_g$ with four processors). This shows that the calculated results used to show the improvement of the parallel implementation are accurate.

Figure 4.9 and Figure 4.10 shows the standard deviation of the serial component of the runtime ($T_g_{\text{serial}}$) over the span of 100 generations, or data points, for its respective modeling method. The serial component of the runtime’s standard deviations indicate that the data values used to obtain the average calculated mean time are widely dispersed. This is somewhat expected as the serial component times all existed within the 1000’ths of a second. At this granularity level, even an additional process or query on the computation node may skew the results severely (snapshots of processor load during benchmark testing can be seen in Appendix A). It could have also exceeded the level of granularity that Python’s time function offers. Nevertheless, the serial component time only makes up less than 3% of a generation run so the impact of the variance of the results is minimal. The CV for these results hovered around 9.8 which indicates a high variation exists.

Figure 4.11 and Figure 4.12 shows the standard deviation of the parallel component of the runtime ($T_g_{\text{parallel}}$) over the span of 100 generations, or data points, for its respective modeling method. The standard deviation of the parallel component ($T_g_{\text{parallel}}$) complimented that of the standard deviation of the generation runtime ($T_g$) with a low standard deviation and improvements even in the worst case scenarios. The CV were all well below the threshold of 1 with a max being 0.139.

Overall the standard deviations and CV show that the computed average mean time per generation per metric is an accurate benchmark of the system.
Figure 4.7. DE-BPSO with MLR: $T_g$ standard deviation.

Figure 4.8. DE-BPSO with PLSR: $T_g$ standard deviation.
Figure 4.9. DE-BPSO with MLR: $T_{g_{\text{serial}}}$ standard deviation.

Figure 4.10. DE-BPSO with PLSR: $T_{g_{\text{serial}}}$ standard deviation.
Figure 4.11. DE-BPSO with MLR: $T_{g, \text{parallel}}$ standard deviation.

Figure 4.12. DE-BPSO with PLSR: $T_{g, \text{parallel}}$ standard deviation.
4.1.3.3 Subset Sizes

Benchmark results included in this thesis are run against 100 generation with a population size of 40. The population size is equivalent to the number of subsets run per generation. Each subset will hold a random number of descriptors that it runs, which is known as the subset size. In order to further verify that the obtained and calculated results are an accurate representation of the system, one must also be able to prove that the variable subset sizes do not impact the results significantly.

Figure 4.13 shows the computation time vs. subset size over the number of cores for the test case: DE-BPSO with MLR. Two notable behaviors come to mind: how come the computation time increases as the number of processors increases and why does the computation time increase as the subset size increases. For these answers, we investigate how the computation was performed and the intricacies of the different modeling methods tested.

The benchmark results obtained within this thesis are run against an SMP model with identical processors running on a shared memory architecture. A common architecture for small SMP systems is the single bus architecture where all the processors and memory modules are attached to the same set of wires, known as the bus. This architecture decreases complexity of the system while also allowing systems to be very cost-effective for the end user. A major crux of this design is that only one processor is allowed to use the bus at a time. For a miniscule number of processors, this limitation is hidden, but nevertheless the limitation of a max bandwidth that the bus can handle still exists. Typically, contention increases with the increase in the number of processors which eventually leads to the saturation of the bus. Several enhancements, such as introducing a cache memory module for each processor to reduce the frequency of access to the main memory and the addition of interconnects that provides additional connectivity between modules, have helped relieve the impact of the aforementioned crux, but the limitation still exists [44]. This contention between processors may be one explanation for why the computation time increases as the number of processors increases.

MLR is a modeling method that is used to assess the association between two or more independent variables and a single continuous dependent variable. In this thesis, the two or more independent variables are the descriptors and the dependent variable is how effective the combination of the set of descriptors is. As the number of descriptors increases, the subset size will increase, which means the MLR modeling method will have an increased number of independent variable to use in its calculation to derive a relationship with the single continuous dependent variable. Knowing this, an increase in computation time as the subset size increases for MLR is expected.
Figure 4.14 shows the computation time vs. subset size over the number of cores for the test case: DE-BPSO with PLSR. Here we observe that PLSR behaves similarly to MLR in terms of the computation time increasing as the number of subsets increase, but the change is minimal. This can be attributed to the additional complexity of PLSR since PLSR is not just a function of the number of variables, but a function of the number of latent variables as well.

Figure 4.13. DE-BPSO with MLR: computation time vs. subset size over number of cores.
Figure 4.14. DE-BPSO with PLSR: computation time vs. subset size over number of cores.
4.2 GUI FOR EVOQSAR

4.2.1 Software Dependencies

The GUI was designed and tested against the following software:

- evoQSAR 0.6.2
- Python 2.7.10
- Anaconda 2.3.0
- Conda 3.18.1
- Matplotlib 1.4.3
- Numpy 1.9.3
- Pandas 0.16.2
- PyQt 4.10.4

4.2.2 Operation

This section provides the user with an overview of the GUI and its functionalities. EvoQSAR begins with an introduction screen with two modes: Feature Selection with Stepwise QSAR, or Single QSAR. Notice how the 'Introduction' tab is the only one enabled in Figure 4.15, this is by design. Tabs are dynamically enabled when proper conditions are met.

![Figure 4.15. GUI - welcome screen.](image-url)
4.2.2.1 **evoQSAR Feature Selection with Stepwise QSAR**

When the user selects a mode, the 'Datasets' tab becomes dynamically enabled. The user is then given the option to navigate to the 'Datasets' tab through either the 'Next' button or directly clicking on the 'Datasets' tab.

![GUI - feature selection w/ stepwise QSAR - introduction tab with feature selection w/ stepwise QSAR selected.](image)

**Figure 4.16.** GUI - feature selection w/ stepwise QSAR - introduction tab with feature selection w/ stepwise QSAR selected.

4.2.2.1.1 **Datasets**

The 'Datasets' tab is used for importing datasets to be modeled after. For feature selection, the training and validation set are the only ones needed. Notice how the test set’s path location box and button is greyed out in Figure 4.17, this is because feature selection does not use the test set so it is disabled (it is included here because mode: Single QSAR uses it and shares the 'Datasets' tab with mode: Feature Selection with Stepwise QSAR). Only relevant features are enabled and allowed. A brief description of how the dataset should be formatted, along with descriptions of the configurations can be found on the right hand side help box. Help boxes are embedded throughout the GUI to remind the user of key facts and pertinent instructions. Figure 4.17 also shows the 'Feature Selection' and 'Modeling' tabs enabled, this is because the screen shot depicts the result of a successful import. Prompts are provided throughout the GUI to notify the user if the imports and simulation runs are successful or not. If the import was not successful, an error prompt with an error message will ask the user to re-enter a proper dataset for import.
Datasets should be in comma separated values (.csv) format and the user can input datasets using the '...' button which will prompt the user with an 'open file' dialog. The GUI uses the user’s operating system’s native file dialogs to provide a seamless experience with what the user is used to. The GUI offers the user the flexibility of inserting a header column or an index column and provides two normalization options, standardize and rescale, to reduce the work the user needs to do.

![Figure 4.17. GUI - feature selection w/ stepwise QSAR - datasets tab with a successful import.](image)

### 4.2.2.1.2 Feature Selection

Once acceptable datasets are successfully entered, the next tab is 'Feature Selection'. Here the user is presented with a variety of options to configure their feature selection run. Notice how all the fields are populated already in Figure 4.18. This is because fields are pre-populated with the default values to allow novice users and advanced users alike to run simulations with default parameters. All the pre-populated fields are user-programmable as well thus offering advanced users the ability to customize their runs as well. Tips and short descriptions are again offered in the right hand side help box. Users can also access a tooltip which offers quick guidance on a specific field if they hover their mouse pointer over the respective field.

For 'Feature Selection’, the user is provided with the option to choose either BPSO or DE-BPSO. Figure 4.18 shows the default selection of DE-BPSO. For BPSO, DE-BPSO
specific parameters, 'F', and 'CR', will be grayed out. The user selectable parameters are described in Definitions 4.15-4.24.

**Definition 4.15.** *Feature Selection Parameter:* \( \text{ALPHA (INITIAL, FINAL)} \) is the initial and final static probability of feature selection.

**Definition 4.16.** *Feature Selection Parameter:* \( \text{BETA} \) is the mutation rate.

**Definition 4.17.** *Feature Selection Parameter:* \( \text{F (DE-BPSO ONLY)} \) is the scaling factor.

**Definition 4.18.** *Feature Selection Parameter:* \( \text{CR (DE-BPSO ONLY)} \) is the cross-over rate.

**Definition 4.19.** *Feature Selection Parameter:* \( \text{INITIAL FEATURES} \) is the average number of features selected in the initial population.

**Definition 4.20.** *Feature Selection Parameter:* \( \text{POPULATION SIZE} \) is the population size.

**Definition 4.21.** *Feature Selection Parameter:* \( \text{MAX GENERATIONS} \) is the maximum number of generations.

**Definition 4.22.** *Feature Selection Parameter:* \( \text{MAX MODELS} \) is the maximum number of models.
**Definition 4.23.** *Feature Selection Parameter:* **PENALTY FACTOR (C)** encourages fewer number of descriptors selected to prevent overfitting (if higher).

**Definition 4.24.** *Feature Selection Parameter:* **PARALLEL** enables / disables parallel functionality (Linux and OS X only).

### 4.2.2.1.3 Modeling

The next tab after 'Feature Selection' is 'Modeling'. Along with the 'Datasets' tab, this tab is also shared with mode: Single QSAR. Here the user is presented with either MLR or PLSR for the modeling method and either leave-one-out or k-fold for cross validation options. The default selection is MLR with a cross-validation option of leave-one-out. Latent variables are a PLSR specific parameter and will be enabled if the user selects PLSR as the modeling method. The number of folds is enabled for only a cross validation option of k-fold. The number of runs is enabled only if shuffle is checked which is enabled by selecting k-fold. The user selectable parameters are described in Definitions 4.25-4.28.

Once the user has entered all of the parameters, the user can click 'Run' and it will begin the feature selection simulation.

![GUI - feature selection w/ stepwise QSAR - modeling tab.](image)

**Definition 4.25.** *Modeling Parameter:* **LATENT VARIABLES (PLSR ONLY)** is the number of latent variables for PLSR.
Definition 4.26. **Modeling Parameter: NUMBER OF FOLDS (K-FOLD ONLY)** is the number of partitions for k-fold.

Definition 4.27. **Modeling Parameter: SHUFFLE (K-FOLD ONLY)** enables/disables shuffling the dataset before each cross-validation run.

Definition 4.28. **Modeling Parameter: NUMBER OF RUNS (K-FOLD ONLY)** is the number of cross-validated runs.

4.2.2.1.4 Results

Once the simulation finishes, the user is greeted with another prompt indicating whether the run was successful or not. Next, the user is greeted with whether viable models were found or not. Figure 4.20 and Figure 4.21 assumes that the simulation was successful and models were found. This leads us to Figure 4.22 with the 'Results' tab. Here the user is given the ability to display the top 20 ranked descriptors found through the feature selection simulation. To display the figure, the user must first click 'Initialize Plot' and then the 'Plot' button to display the chart. The user is presented with a few plot modification tools and the ability to save the plot (more information on the plot tools can be found in Section 4.2.2.3). The user is also able to export the models and importance derived from the simulation to .csv format. The models output provides the user with information on the models found including the descriptors that compose the model, its fitness and some model specific metrics, $R^2$, $Q^2$ and $R_{pred}^2$ (defined in Definitions 4.30-4.33). Importance provides the user with a list of how frequent each descriptor was used in the modeling process ranked from most used to least used.
Figure 4.20. GUI - feature selection w/ stepwise QSAR - simulation modeling success.

Figure 4.21. GUI - feature selection w/ stepwise QSAR - models found.
Figure 4.22. GUI - feature selection w/ stepwise QSAR - results tab with results.
4.2.2.1.5 Stepwise QSAR

The last tab and functionality for mode: Feature Selection with Stepwise QSAR is the ‘Stepwise QSAR’ functionality seen in Figure 4.23. Here the user can upload a test set which contains values that the user would like to test against the models derived from the simulation run. The dataset should be massaged to the same format as the training and validation sets formerly imported (follows the same normalized option: None, Standardize, Rescale), but flexibility is given in the form of headers and index columns. Like the ‘Datasets’ tab, the user will click ‘Import Test Set’ to import the test set and the user will be prompted whether it was successfully or not. The user is then able to select the number of descriptors they would like to run the stepwise QSAR over in the field: ‘Max Number of Descriptors’, and select and output location and filename for the output file. The output file will contain the results of stepwise QSAR in a text file ran against the recently uploaded test set. A brief description of the calculated metrics can be found in Definitions 4.29-4.39.

Figure 4.23. GUI - feature selection w/ stepwise QSAR - stepwise QSAR.

**Definition 4.29.** QSAR Output Statistics: \textbf{INDICES} is the feature ID number.

**Definition 4.30.** QSAR Output Statistics: $R^2$ is the coefficient of determination.

A metric to determine the quality of fit of a regression model - assess how well the trained
model fits the training set.

\[ R^2 > 0.6 \] is considered an acceptable model.

**Definition 4.31.** QSAR Output Statistics: \( Q^2 \) is the model predictivity coefficient
A metric used to estimate the predictivity of the model using the training set data.
\( Q^2 > 0.5 \) is considered an acceptable model.

**Definition 4.32.** QSAR Output Statistics: \( R^2_{pred}(v) \) is the external predictivity coefficient for the validation set.
A metric that measures the degree of correlation between the observed and predicted biological activities.
\( R^2_{pred}(v) > 0.5 \) is considered to be predictive.

**Definition 4.33.** QSAR Output Statistics: \( R^2_{pred}(t) \) is the external predictivity coefficient for the test set.
A metric that measures the degree of correlation between the observed and predicted biological activities.
\( R^2_{pred}(t) > 0.5 \) is considered to be predictive.

**Definition 4.34.** QSAR Output Statistics: \( \frac{(r^2 - r_o^2)}{r^2} \) is the calculated metric used to determine if a model is predictive.

**Definition 4.35.** QSAR Output Statistics: \( r_0^2 \) is the correlation of the regression line.

**Definition 4.36.** QSAR Output Statistics: \( r^2 \) is the correlation coefficient.

**Definition 4.37.** QSAR Output Statistics: \( k \) is the slope of the regression line.

**Definition 4.38.** QSAR Output Statistics: \( \frac{(r^2 - r_o^2)}{r^2} \) is a calculated metric used to determine if a model is predictive.

**Definition 4.39.** QSAR Output Statistics: \( |r_o^2 - r_o'^2| \) is a calculated metric used to determine if a model is predictive.

### 4.2.2.2 EVOQSAR SINGLE QSAR

This section goes over the mode: Single QSAR feature of evoQSAR. Similarly to mode: Feature Selection with Stepwise QSAR, when mode: Single QSAR is clicked, the
Datasets’ tab is enabled. The user can then navigate to the datasets tab either through using the ’Next’ button or directly clicking on the ’Datasets’ tab.

![Figure 4.24. GUI - single QSAR - introduction tab.](image)

4.2.2.1 Datasets

The ’Datasets’ tab is shared between mode: Feature Selection with Stepwise QSAR and mode: Single QSAR. The difference is the enabled functionality: For mode: Feature Selection with Stepwise QSAR, the training set and validation set were enabled, but for the mode: Single QSAR, the training set and test set is enabled, with the validation set disabled. Similar to mode: Feature Selection with Stepwise QSAR, the user must import the dataset correctly in order to move on to the next tab, in this case, ‘Modeling’. If a dataset was imported incorrectly, the prompt shown in Figure 4.25 would change from a Success to an Error.
Figure 4.25. GUI - single QSAR - dataset successful import.
4.2.2.2 Modeling

The 'Modeling' tab for mode: Single QSAR is the same as it was for mode: Feature Selection with Stepwise QSAR. The user is able to choose a modeling method of either MLR or PLSR, and cross validation options of either leave-one-out or k-fold. The default selection is MLR with leave-one-out. Other options are enabled when certain criteria are met such as latent variables for PLSR and number of folds when k-fold cross validation options is selected. Options are defined in Definitions 4.25-4.28.

![Figure 4.26. GUI - single QSAR - modeling tab.](image)

After parameters have been entered, the user initiates the simulation by clicking 'Run'. A pop up dialog will inform the user that the simulation was started successfully, Figure 4.27, and another one will inform the user if the simulation was successfully or not, Figure 4.28.
Figure 4.27. GUI - single QSAR - start simulation.

Figure 4.28. GUI - single QSAR - simulation was successful.
Once a successful run has been completed, five other tabs then becomes available: 'Correlation' (Figure 4.29), 'Applicability Domain' (Figure 4.30-4.31), 'y-Randomization' (Figure 4.32), 'Statistics' (Figure 4.33), and 'Prediction' (Figure 4.34-4.35).

### 4.2.2.2.3 Correlation

The 'Correlation' tab demonstrates how well the model correlates with the predicted and observed values of the training and test set by plotting the training set values (in red) and the test set values (in blue) against predicted and observed axis’s.

![Figure 4.29. GUI - single QSAR - correlation with results.](image)

### 4.2.2.2.4 Applicability Domain

The applicability domain is defined as the chemical space in which a QSAR model can make reliable predictions for a given chemical compound. This tab provides the user with the functionality to output two applicability domain files in .csv format, one for the training set and another for the test set, and a visual representation of the aforementioned file statistics. A description of the calculated statistics can be found in Definitions 4.40-4.46.

**Definition 4.40.** Applicability Domain Metrics: STRUCTURE NAME is the name of the compound.

**Definition 4.41.** Applicability Domain Metrics: OBSERVED is the observed values.

**Definition 4.42.** Applicability Domain Metrics: PREDICTED is the model predicted values.
**Definition 4.43.** Applicability Domain Metrics: **RESIDUAL** is the predicted value subtracted from the observed value.

**Definition 4.44.** Applicability Domain Metrics: **NORMALIZED RESIDUAL** is the normalized residual with respect to all the residuals calculated within the respective file.

**Definition 4.45.** Applicability Domain Metrics: **LEVERAGE** is a common metric used to define the applicability domain that is a distance based metric that defines the boundaries of which the QSAR model can make reliable predictions.

**Definition 4.46.** Applicability Domain Metrics: **WARNING** is an indicative measure of whether the compound is inside or outside the applicability domain. 
0 = Inside applicability domain.
1 = outside applicability domain: compounds that fall outside the applicability domain require significant extrapolation to make a reliable prediction.

![Figure 4.30. GUI - single QSAR - applicability domain with output filenames.](image-url)
Figure 4.31. GUI - single QSAR - applicability domain with results.
4.2.2.2.5  y-Randomization

The ‘y-Randomization’ tab allows the user to visually view a user selectable number of runs displayed on a $Q^2$ by $R^2$ graph. Each run estimates one $Q^2$ and one $R^2$ value which is then plotted on the graph. The y-Randomization test estimates the robustness of the model by determining whether or not highly correlated computational models were developed due to chance. $R^2$ is the coefficient of determination which helps in determining the quality of fit of a QSAR model. $Q^2$ is the cross-validated $R^2$ which helps determine the model’s predictivity. The randomization process is repeated many times and it is expected that the majority of randomization runs do not exceed a threshold $R^2 > 0.4$ and $Q^2 > 0.4$, indicating no recognized chance correlation [13].

Figure 4.32. GUI - single QSAR - y-randomization with results.
4.2.2.2.6 Statistics

The 'Statistics' tab allows the user to output the model’s relevant statistics to a text file (output statistics are defined in Definitions 4.29-4.39). The contents of the text file are also displayed within the GUI for the user’s convenience.

Figure 4.33. GUI - single QSAR - statistics with results.
4.2.2.2.7 Prediction

The 'Prediction' tab allows the user to upload a prediction set (or test set) to run against the recently computed model. The user must first import the prediction dataset successfully before a run can be ran. Notice in Figure 4.34 that the 'Run' button is greyed out; this is because the user has not imported the prediction dataset yet. Figure 4.35 shows a successful run with the 'Run' button un-greyed. The output .csv contains the compound name (Definition 4.40), predicted activity value for the compound (Definition 4.42), and a leverage metric (Definition 4.45). The leverage metric here is used to predict if the compound is within the applicability domain or not (0, and 1 respectively).
Figure 4.34. GUI - single QSAR - prediction tab.

Figure 4.35. GUI - single QSAR - prediction with a successful run.
4.2.2.3 **Plot Toolbar**

The plot toolbar can be seen on the top of every plot within the GUI. The functionalities shown below are incorporated into the GUI to provide the user with another level of customization. The plot toolbar is part of the matplotlib module [20].

4.2.2.3.1 **Home/Forward/Back Buttons**

The 'Home' button brings the graph back to its initial state. The 'Forward' and 'Back' buttons are analogous to your typical redo and undo buttons for modifications to the graph.

![Home, back, and forward buttons](image)

*Figure 4.36. Home, back, and forward buttons.*

4.2.2.3.2 **Pan/Zoom Button**

This button allows the user to 'Pan' and 'Zoom' through the plot; Left click to pan through the graph, and right click to zoom through the graph.

![Pan/zoom button](image)

*Figure 4.37. Pan/zoom button.*

4.2.2.3.3 **Zoom-to-Rectangle Button**

This button allows the user to zoom in on a section of the plot; Left click to drag a box around an area to be zoomed in, and right click to drag a box around an area to be zoomed out of.

![Zoom-to-rectangle button](image)

*Figure 4.38. Zoom-to-rectangle button.*
4.2.2.3.4 Subplot Configuration Button
(Not Used)

This button allows the user to configure the parameters of the subplot. This functionality is not used in this GUI.

Figure 4.39. Subplot configuration button.

4.2.2.3.5 Save Button

This button launches a file save dialog to save the image. Valid extensions are .png, .ps, .eps, .svg, and .pdf.

Figure 4.40. Save button.
CHAPTER 5
CONCLUSION AND FUTURE DIRECTION

5.1 CONCLUSIONS

In this thesis, we implemented two significant enhancements to evoQSAR, parallelization and a GUI. These enhancements maximize the efficiency of the user’s system when computing models and an easy to use interface to run simulations.

The parallelization of evoQSAR leveraged the Python Multiprocessing module for its coherency and compatibility with the existing evoQSAR toolkit, written and distributed in Python, and ease of use, through using its ‘Pool’ class. The parallelization enabled evoQSAR to utilize all processors on a local system for computation. It was designed to run on a SMP system where processes can be spawned using the ‘fork-join’ approach. The use of shared memory, global variables, and the ‘fork-join’ approach has allowed us to eliminate the overhead associated with the duplication of the parent’s variables on each child process. Unfortunately, this approach only allows parallelization to run on Linux and OS X.

The GUI development of evoQSAR used PyQt to leverage the tried and true Qt framework while still staying pythonic in nature and Qt Designer for its GUI design functionality. The GUI provides another portal into evoQSAR, in lieu of default python scripts, by providing a stand-alone, cross-platform, and intuitive interface. Default values are pre-populated throughout the GUI to reduce the learning curve needed for novice and advanced users alike. This approach simplifies the user experience and limits potential confusion that arise from an overwhelming influx of information.

The work covered in this thesis was verified using a HIV integrase inhibitor dataset (given in Appendix B) tested against two benchmarks: DE-BPSO with MLR and DE-BPSO with PLSR. Each test case was tested on a family of processors, 1, 2, 4, 8, and 16, to characterize the parallelization improvement. Overall, the parallelization of evoQSAR decreased the computation time needed to develop predictive computational models for QSAR analysis that can aid in drug design. In addition, the GUI enabled the functionality of evoQSAR without the need of complicated python scripts. These enhancements to the existing evoQSAR toolkit will now allow chemists, pharmaceutical companies, academia, and others to use evoQSAR which was not possible previously due to usability concerns.
5.2 CHALLENGES

The challenges encountered in the completion of the research presented in this thesis included:

- Background knowledge of biology and chemistry to understand the dataset, the mechanics of how evoQSAR works, QSAR analysis, and basic drug design principles.
- Knowledge of the MLR and PLSR modeling methods.
- Knowledge of the BPSO and DE-BPSO evolutionary algorithms.
- Knowledge of parallel concepts for parallel implementation.
- Learning PyQt and the Qt development framework for the GUI implementation.
- Python to understand the existing evoQSAR codebase and to implement the parallelization and GUI.

5.3 FUTURE DIRECTION

Parallelization and GUI developed in evoQSAR can be extended on multiple fronts. A few future work directions are explained below.

The parallel aspect of evoQSAR currently only works on Linux and OS X due to the lack of ‘fork’ in the Windows operating system. In order to make evoQSAR truly cross-platform, this additional functionality would be needed. This thesis designed evoQSAR to be run locally on one’s computer, but future enhancements may include the migration of evoQSAR to a web-based service located on high performance computing hardware interfaced through the web, while leveraging faster hardware, thus removing the physical hardware limitations of the host computer. This migration may also include the need to migrate evoQSAR’s parallel implementation from a shared memory architecture to a distributed memory architecture depending on the intended hardware. This may also involve the use of GPU accelerators.

The Qt framework used to create the GUI of evoQSAR allows the ability to port the existing GUI to either a web, as previously mentioned, or to a mobile environment, thus increasing the number of mediums one can use to access evoQSAR. Within the GUI, one could also include the ability to dynamically view and modify the datasets within the GUI, and potentially even perform single QSAR analysis by manually selecting the descriptors from a general dataset.
EvoQSAR, as it is, is limited to two feature selection algorithms, BPSO and DE-BPSO, and two modeling methods, MLR and PLSR. Future work may include extending EvoQSAR’s functionality to include additional feature selection algorithms such as GA, artificial neural networks (ANN), and support vector machines (SVM). The implemented modeling methods all belonged to the linear regression family; future work may include additional modeling methods such as logistic regression (LR), random forests (RF), and k-nearest neighbor (KNN).

In addition to the aforementioned potential enhancements, one could also tailor the EvoQSAR modeling framework for other fields as the algorithms and modeling methods are not limited to drug design only.
BIBLIOGRAPHY


http://hisham.hm/htop/.


http://teqip.jdvu.ac.in/QSAR_Tools/.


[30] MESSAGE PASSING INTERFACE FORUM, 

[31] MICROSOFT, Processes and Threads (Windows), 2015. 


[34] PYTHON SOFTWARE FOUNDATION, 

[35] PYTHON SOFTWARE FOUNDATION, 


http://openmp.org/wp/.


APPENDIX A
ADDITIONAL BENCHMARK RESULTS
ADDITIONAL BENCHMARK RESULTS

This section shows the processing loads on Node 16 during the benchmark tests using htop, an interactive process viewer for Linux [18].
Figure A.1. Node 16 idle.
Figure A.2. EvoQSAR DE-BPSO with MLR on 1 processor.
Figure A.3. EvoQSAR DE-BPSO with MLR on 2 processor.
Figure A.4. EvoQSAR DE-BPSO with MLR on 4 processor.
Figure A.5. EvoQSAR DE-BPSO with MLR on 8 processor.
Figure A.6. EvoQSAR DE-BPSO with MLR on 16 processor.
Figure A.7. EvoQSAR DE-BPSO with PLSR on 1 processor.
Figure A.8. EvoQSAR DE-BPSO with PLSR on 2 processor.
Figure A.9. EvoQSAR DE-BPSO with PLSR on 4 processor.
Figure A.10. EvoQSAR DE-BPSO with PLSR on 8 processor.
Figure A.11. EvoQSAR DE-BPSO with PLSR on 16 processor.
APPENDIX B
HIV INTEGRASE INHIBITOR DATASET
HIV INTEGRASE INHIBITOR DATASET

Table B.1. Standardized Structures of β-Diketo Acid Compounds.

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image" alt="Structure 1" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image" alt="Structure 2" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image" alt="Structure 3" /></td>
</tr>
<tr>
<td>4</td>
<td><img src="image" alt="Structure 4" /></td>
</tr>
<tr>
<td>5</td>
<td><img src="image" alt="Structure 5" /></td>
</tr>
<tr>
<td>6</td>
<td><img src="image" alt="Structure 6" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td><img src="image" alt="Structure 7" /></td>
</tr>
<tr>
<td>8</td>
<td><img src="image" alt="Structure 8" /></td>
</tr>
<tr>
<td>9</td>
<td><img src="image" alt="Structure 9" /></td>
</tr>
<tr>
<td>10</td>
<td><img src="image" alt="Structure 10" /></td>
</tr>
<tr>
<td>11</td>
<td><img src="image" alt="Structure 11" /></td>
</tr>
<tr>
<td>12</td>
<td><img src="image" alt="Structure 12" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>![Structure 13]</td>
</tr>
<tr>
<td>14</td>
<td>![Structure 14]</td>
</tr>
<tr>
<td>15</td>
<td>![Structure 15]</td>
</tr>
<tr>
<td>16</td>
<td>![Structure 16]</td>
</tr>
<tr>
<td>17</td>
<td>![Structure 17]</td>
</tr>
<tr>
<td>18</td>
<td>![Structure 18]</td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td><img src="image" alt="Structure 19" /></td>
</tr>
<tr>
<td>20</td>
<td><img src="image" alt="Structure 20" /></td>
</tr>
<tr>
<td>21</td>
<td><img src="image" alt="Structure 21" /></td>
</tr>
<tr>
<td>22</td>
<td><img src="image" alt="Structure 22" /></td>
</tr>
<tr>
<td>23</td>
<td><img src="image" alt="Structure 23" /></td>
</tr>
<tr>
<td>24</td>
<td><img src="image" alt="Structure 24" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td><img src="image" alt="Structure 25" /></td>
</tr>
<tr>
<td>26</td>
<td><img src="image" alt="Structure 26" /></td>
</tr>
<tr>
<td>27</td>
<td><img src="image" alt="Structure 27" /></td>
</tr>
<tr>
<td>28</td>
<td><img src="image" alt="Structure 28" /></td>
</tr>
<tr>
<td>29</td>
<td><img src="image" alt="Structure 29" /></td>
</tr>
<tr>
<td>30</td>
<td><img src="image" alt="Structure 30" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td><img src="image" alt="Structure 31" /></td>
</tr>
<tr>
<td>32</td>
<td><img src="image" alt="Structure 32" /></td>
</tr>
<tr>
<td>33</td>
<td><img src="image" alt="Structure 33" /></td>
</tr>
<tr>
<td>34</td>
<td><img src="image" alt="Structure 34" /></td>
</tr>
<tr>
<td>35</td>
<td><img src="image" alt="Structure 35" /></td>
</tr>
<tr>
<td>36</td>
<td><img src="image" alt="Structure 36" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td><img src="image" alt="Structure 37" /></td>
</tr>
<tr>
<td>38</td>
<td><img src="image" alt="Structure 38" /></td>
</tr>
<tr>
<td>39</td>
<td><img src="image" alt="Structure 39" /></td>
</tr>
<tr>
<td>40</td>
<td><img src="image" alt="Structure 40" /></td>
</tr>
<tr>
<td>41</td>
<td><img src="image" alt="Structure 41" /></td>
</tr>
<tr>
<td>42</td>
<td><img src="image" alt="Structure 42" /></td>
</tr>
<tr>
<td>43</td>
<td><img src="image" alt="Structure 43" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td><img src="image" alt="Structure 44" /></td>
</tr>
<tr>
<td>45</td>
<td><img src="image" alt="Structure 45" /></td>
</tr>
<tr>
<td>46</td>
<td><img src="image" alt="Structure 46" /></td>
</tr>
<tr>
<td>47</td>
<td><img src="image" alt="Structure 47" /></td>
</tr>
<tr>
<td>48</td>
<td><img src="image" alt="Structure 48" /></td>
</tr>
<tr>
<td>49</td>
<td><img src="image" alt="Structure 49" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td><img src="image1" alt="Structure" /></td>
</tr>
<tr>
<td>51</td>
<td><img src="image2" alt="Structure" /></td>
</tr>
<tr>
<td>52</td>
<td><img src="image3" alt="Structure" /></td>
</tr>
<tr>
<td>53</td>
<td><img src="image4" alt="Structure" /></td>
</tr>
<tr>
<td>54</td>
<td><img src="image5" alt="Structure" /></td>
</tr>
<tr>
<td>55</td>
<td><img src="image6" alt="Structure" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>![Structure 56]</td>
</tr>
<tr>
<td>57</td>
<td>![Structure 57]</td>
</tr>
<tr>
<td>58</td>
<td>![Structure 58]</td>
</tr>
<tr>
<td>59</td>
<td>![Structure 59]</td>
</tr>
<tr>
<td>60</td>
<td>![Structure 60]</td>
</tr>
<tr>
<td>61</td>
<td>![Structure 61]</td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td><img src="image" alt="Structure 62" /></td>
</tr>
<tr>
<td>63</td>
<td><img src="image" alt="Structure 63" /></td>
</tr>
<tr>
<td>64</td>
<td><img src="image" alt="Structure 64" /></td>
</tr>
<tr>
<td>65</td>
<td><img src="image" alt="Structure 65" /></td>
</tr>
<tr>
<td>66</td>
<td><img src="image" alt="Structure 66" /></td>
</tr>
<tr>
<td>67</td>
<td><img src="image" alt="Structure 67" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td><img src="image1" alt="Image" /></td>
</tr>
<tr>
<td>69</td>
<td><img src="image2" alt="Image" /></td>
</tr>
<tr>
<td>70</td>
<td><img src="image3" alt="Image" /></td>
</tr>
<tr>
<td>71</td>
<td><img src="image4" alt="Image" /></td>
</tr>
<tr>
<td>72</td>
<td><img src="image5" alt="Image" /></td>
</tr>
<tr>
<td>73</td>
<td><img src="image6" alt="Image" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td><img src="image1" alt="Structure 74" /></td>
</tr>
<tr>
<td>75</td>
<td><img src="image2" alt="Structure 75" /></td>
</tr>
<tr>
<td>76</td>
<td><img src="image3" alt="Structure 76" /></td>
</tr>
<tr>
<td>77</td>
<td><img src="image4" alt="Structure 77" /></td>
</tr>
<tr>
<td>78</td>
<td><img src="image5" alt="Structure 78" /></td>
</tr>
<tr>
<td>79</td>
<td><img src="image6" alt="Structure 79" /></td>
</tr>
</tbody>
</table>

(table continues)
Table B.1 (Continued)

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td><img src="image" alt="Structure 80" /></td>
</tr>
<tr>
<td>81</td>
<td><img src="image" alt="Structure 81" /></td>
</tr>
<tr>
<td>82</td>
<td><img src="image" alt="Structure 82" /></td>
</tr>
<tr>
<td>83</td>
<td><img src="image" alt="Structure 83" /></td>
</tr>
<tr>
<td>84</td>
<td><img src="image" alt="Structure 84" /></td>
</tr>
<tr>
<td>85</td>
<td><img src="image" alt="Structure 85" /></td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Standardized Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>86</td>
<td><img src="image" alt="Structure 86" /></td>
</tr>
<tr>
<td>87</td>
<td><img src="image" alt="Structure 87" /></td>
</tr>
<tr>
<td>88</td>
<td><img src="image" alt="Structure 88" /></td>
</tr>
<tr>
<td>89</td>
<td><img src="image" alt="Structure 89" /></td>
</tr>
<tr>
<td>90</td>
<td><img src="image" alt="Structure 90" /></td>
</tr>
<tr>
<td>91</td>
<td><img src="image" alt="Structure 91" /></td>
</tr>
</tbody>
</table>
Table B.2. Dataset Division and Source Structure of β-Diketo Acid Compounds.

<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Dataset</th>
<th>Source No.¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Training</td>
<td>4a</td>
</tr>
<tr>
<td>2</td>
<td>Training</td>
<td>4b</td>
</tr>
<tr>
<td>3</td>
<td>Training</td>
<td>4c</td>
</tr>
<tr>
<td>4</td>
<td>Training</td>
<td>4d</td>
</tr>
<tr>
<td>5</td>
<td>Training</td>
<td>4e</td>
</tr>
<tr>
<td>6</td>
<td>Training</td>
<td>7a</td>
</tr>
<tr>
<td>7</td>
<td>Test</td>
<td>7b</td>
</tr>
<tr>
<td>8</td>
<td>Validation</td>
<td>7c</td>
</tr>
<tr>
<td>9</td>
<td>Training</td>
<td>10a</td>
</tr>
<tr>
<td>10</td>
<td>Training</td>
<td>10b</td>
</tr>
<tr>
<td>11</td>
<td>Training</td>
<td>10e</td>
</tr>
<tr>
<td>12</td>
<td>Validation</td>
<td>12b</td>
</tr>
<tr>
<td>13</td>
<td>Test</td>
<td>13a</td>
</tr>
<tr>
<td>14</td>
<td>Test</td>
<td>13b</td>
</tr>
<tr>
<td>15</td>
<td>Test</td>
<td>13f</td>
</tr>
<tr>
<td>16</td>
<td>Validation</td>
<td>13h</td>
</tr>
<tr>
<td>17</td>
<td>Training</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>Training</td>
<td>8</td>
</tr>
<tr>
<td>19</td>
<td>Validation</td>
<td>9</td>
</tr>
<tr>
<td>20</td>
<td>Training</td>
<td>10</td>
</tr>
<tr>
<td>21</td>
<td>Training</td>
<td>11</td>
</tr>
<tr>
<td>22</td>
<td>Training</td>
<td>12</td>
</tr>
<tr>
<td>23</td>
<td>Training</td>
<td>13</td>
</tr>
<tr>
<td>24</td>
<td>Training</td>
<td>14</td>
</tr>
<tr>
<td>25</td>
<td>Training</td>
<td>15</td>
</tr>
<tr>
<td>26</td>
<td>Training</td>
<td>16</td>
</tr>
<tr>
<td>27</td>
<td>Training</td>
<td>4a</td>
</tr>
<tr>
<td>28</td>
<td>Training</td>
<td>4b</td>
</tr>
<tr>
<td>29</td>
<td>Training</td>
<td>4c</td>
</tr>
<tr>
<td>30</td>
<td>Training</td>
<td>4d</td>
</tr>
<tr>
<td>31</td>
<td>Training</td>
<td>4e</td>
</tr>
<tr>
<td>32</td>
<td>Training</td>
<td>4f</td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Dataset</th>
<th>Source No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>Training</td>
<td>5a</td>
</tr>
<tr>
<td>34</td>
<td>Training</td>
<td>5b</td>
</tr>
<tr>
<td>35</td>
<td>Training</td>
<td>5c</td>
</tr>
<tr>
<td>36</td>
<td>Training</td>
<td>5d</td>
</tr>
<tr>
<td>37</td>
<td>Training</td>
<td>5e</td>
</tr>
<tr>
<td>38</td>
<td>Validation</td>
<td>4a</td>
</tr>
<tr>
<td>39</td>
<td>Validation</td>
<td>4b</td>
</tr>
<tr>
<td>40</td>
<td>Test</td>
<td>4c</td>
</tr>
<tr>
<td>41</td>
<td>Validation</td>
<td>4d</td>
</tr>
<tr>
<td>42</td>
<td>Training</td>
<td>4e</td>
</tr>
<tr>
<td>43</td>
<td>Test</td>
<td>5a</td>
</tr>
<tr>
<td>44</td>
<td>Validation</td>
<td>5b</td>
</tr>
<tr>
<td>45</td>
<td>Validation</td>
<td>5c</td>
</tr>
<tr>
<td>46</td>
<td>Test</td>
<td>5d</td>
</tr>
<tr>
<td>47</td>
<td>Training</td>
<td>1a</td>
</tr>
<tr>
<td>48</td>
<td>Training</td>
<td>1b</td>
</tr>
<tr>
<td>49</td>
<td>Test</td>
<td>1c</td>
</tr>
<tr>
<td>50</td>
<td>Test</td>
<td>1d</td>
</tr>
<tr>
<td>51</td>
<td>Validation</td>
<td>1e</td>
</tr>
<tr>
<td>52</td>
<td>Test</td>
<td>1f</td>
</tr>
<tr>
<td>53</td>
<td>Training</td>
<td>2a</td>
</tr>
<tr>
<td>54</td>
<td>Training</td>
<td>2b</td>
</tr>
<tr>
<td>55</td>
<td>Training</td>
<td>3b</td>
</tr>
<tr>
<td>56</td>
<td>Test</td>
<td>3c</td>
</tr>
<tr>
<td>57</td>
<td>Test</td>
<td>4c</td>
</tr>
<tr>
<td>58</td>
<td>Validation</td>
<td>4d</td>
</tr>
<tr>
<td>59</td>
<td>Test</td>
<td>4e</td>
</tr>
<tr>
<td>60</td>
<td>Training</td>
<td>5a</td>
</tr>
<tr>
<td>61</td>
<td>Training</td>
<td>6a</td>
</tr>
<tr>
<td>62</td>
<td>Training</td>
<td>8b</td>
</tr>
<tr>
<td>63</td>
<td>Training</td>
<td>9b</td>
</tr>
<tr>
<td>64</td>
<td>Validation</td>
<td>9c</td>
</tr>
</tbody>
</table>

(table continues)
<table>
<thead>
<tr>
<th>Compound No.</th>
<th>Dataset</th>
<th>Source No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>Test</td>
<td>9d</td>
</tr>
<tr>
<td>66</td>
<td>Training</td>
<td>10b</td>
</tr>
<tr>
<td>67</td>
<td>Test</td>
<td>10c</td>
</tr>
<tr>
<td>68</td>
<td>Validation</td>
<td>10e</td>
</tr>
<tr>
<td>69</td>
<td>Training</td>
<td>11b</td>
</tr>
<tr>
<td>70</td>
<td>Validation</td>
<td>11c</td>
</tr>
<tr>
<td>71</td>
<td>Validation</td>
<td>11d</td>
</tr>
<tr>
<td>72</td>
<td>Test</td>
<td>11e</td>
</tr>
<tr>
<td>73</td>
<td>Training</td>
<td>12b</td>
</tr>
<tr>
<td>74</td>
<td>Test</td>
<td>13b</td>
</tr>
<tr>
<td>75</td>
<td>Training</td>
<td>4a</td>
</tr>
<tr>
<td>76</td>
<td>Validation</td>
<td>4b</td>
</tr>
<tr>
<td>77</td>
<td>Training</td>
<td>4c</td>
</tr>
<tr>
<td>78</td>
<td>Training</td>
<td>4d</td>
</tr>
<tr>
<td>79</td>
<td>Validation</td>
<td>4e</td>
</tr>
<tr>
<td>80</td>
<td>Test</td>
<td>4f</td>
</tr>
<tr>
<td>81</td>
<td>Training</td>
<td>4g</td>
</tr>
<tr>
<td>82</td>
<td>Training</td>
<td>4h</td>
</tr>
<tr>
<td>83</td>
<td>Training</td>
<td>4i</td>
</tr>
<tr>
<td>84</td>
<td>Training</td>
<td>4j</td>
</tr>
<tr>
<td>85</td>
<td>Validation</td>
<td>4k</td>
</tr>
<tr>
<td>86</td>
<td>Training</td>
<td>4l</td>
</tr>
<tr>
<td>87</td>
<td>Training</td>
<td>4m</td>
</tr>
<tr>
<td>88</td>
<td>Training</td>
<td>4n</td>
</tr>
<tr>
<td>89</td>
<td>Training</td>
<td>4o</td>
</tr>
<tr>
<td>90</td>
<td>Training</td>
<td>4p</td>
</tr>
<tr>
<td>91</td>
<td>Training</td>
<td>4q</td>
</tr>
</tbody>
</table>

\(^1\text{Source No. refers to the compound number in the original paper from where it is taken: compounds 1-16 [33], 17-26 [46], 27-37 [39], 38-46 [28], 47-74 [45], 75-91 [27].}\)