Machine Learning Classification of Gas Chromatography Data

Evan Clark

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Leyla Nazhandali, Chair
A. Lynn Abbott
Hoda Eldardiry

July 21, 2023
Blacksburg, Virginia

Keywords: Gas Chromatography, Machine Learning, Classification

Machine Learning Classification of Gas Chromatography Data

Evan Clark

(ABSTRACT)

Gas Chromatography (GC) is a technique for separating volatile compounds by relying on adherence differences in the chemical components of the compound. As conditions within the GC are changed, components of the mixture elute at different times. Sensors measure the elution and produce data which becomes chromatograms. By analyzing the chromatogram, the presence and quantity of the mixture's constituent components can be determined. Machine Learning (ML) is a field consisting of techniques by which machines can independently analyze data to derive their own procedures for processing it. Additionally, there are techniques for enhancing the performance of ML algorithms. Feature Selection is a technique for improving performance by using a specific subset of the data. Feature Engineering is a technique to transform the data to make processing more effective. Data Fusion is a technique which combines multiple sources of data so as to produce more useful data. This thesis applies machine learning algorithms to chromatograms. Five common machine learning algorithms are analyzed and compared, including K-Nearest Neighbour (KNN), Support Vector Machines (SVM), Convolutional Neural Network (CNN), Decision Tree, and Random Forest (RF). Feature Selection is tested by applying window sweeps with the KNN algorithm. Feature Engineering is applied via the Principal Component Analysis (PCA) algorithm. Data Fusion is also tested. It was found that KNN and RF performed best overall. Feature Selection was very beneficial overall. PCA was helpful for some algorithms, but less so for others. Data Fusion was moderately beneficial.

Machine Learning Classification of Gas Chromatography Data

Evan Clark

(GENERAL AUDIENCE ABSTRACT)

Gas Chromatography is a method for separating a mixture into its constituent components. A chromatogram is a time series showing the detection of gas in the gas chromatography machine over time. With a properly set up gas chromatographer, different mixtures will produce different chromatograms. These differences allow researchers to determine the components or differentiate compounds from each other. Machine Learning (ML) is a field encompassing a set of methods by which machines can independently analyze data to derive the exact algorithms for processing it. There are many different machine learning algorithms which can accomplish this. There are also techniques which can process the data to make it more effective for use with machine learning. Feature Engineering is one such technique which transforms the data. Feature Selection is another technique which reduces the data to a subset. Data Fusion is a technique which combines different sources of data. Each of these processing techniques have many different implementations. This thesis applies machine learning to gas chromatography. ML systems are developed to classify mixtures based on their chromatograms. Five common machine learning algorithms are developed and compared. Some common Feature Engineering, Feature Selection, and Data Fusion techniques are also evaluated. Two of the algorithms were found to be more effective overall than the other algorithms. Feature Selection was found to be very beneficial. Feature Engineering was beneficial for some algorithms but less so for others. Data Fusion was moderately beneficial.

Dedication

To my family and friends.

Acknowledgments

This would not have been possible without the help and support of my colleagues, friends, and family. First, I am extremely grateful to my research advisor and committee chair, Dr. Leyla Nazhandali. My path through graduate school was definitely more circuitous than I had intended, and you helped and gave me the guidance I needed to achieve what seemed impossible. I also wish to thank Dr. Lynn Abbott and Dr. Hoda Eldardiry for agreeing to be on my committee. To Mustahsin Chowdhury, I would like to express my deepest gratitude for your mentorship and advice. With your guidance, I became a better researcher and engineer. I would also like to thank Nipun Tamatham and Divya Korada for your help and guidance with my defense preparation. To Zayeem Zaman, Corwin Warner, Matthew Zong, Rufus Hinton, Hristo Ignatov, and Michael Koch, thank you for your friendship and support. To my parents, thank you for your loving encouragement, and your funding of many a late night Starbucks run. To all my friends and family, your unwavering confidence in me, even when I did not have confidence in myself, gave me the determination I needed, and I will forever be grateful.

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

GC Gas chromatography

ML Machine Learning

Artificial intelligence is a field of computer science encompassing algorithms for enabling machines to process data and make decisions without direct human control.

Gas chromatography is a technique by which compounds are separated using a gaseous mobile phase.

Machine learning is a branch within Artificial Intelligence concerned with developing methods which allow computers to derive their own algorithms for processing data and making decisions.

Chapter 1

Introduction

Gas chromatography is a widely used technique in analytical chemistry for separating mixtures [1]. A detector produces a signal, which is recorded as chromatograms. By analyzing the peaks of the chromatograms, researchers can determine the chemical components of the mixture [2].

Machine Learning is a rapidly expanding field. It allows a machine to automatically find the relation between an input data set and an output data set [3]. Machine Learning can be applied to gas chromatography data. By using labeled data sets of chromatograms, machine learning models can be trained to classify mixtures.

1.1 Gas Chromatography

Chromatography is a standard technique in separation science used for determining the components of a mixture. In column chromatography (the most common type), a mixture is injected into a column. The column contains two distinct solutes, called the *mobile phase* and *stationary phase*. The stationary phase is usually a permeable solid material packed into the column. The mobile phase is usually a fluid which flows through or over the stationary phase. At the start of a measurement, a mixture is injected into the mobile phase at the inlet of the column. As it flows through the column, the components of the mixture will adhere to the stationary phase with different strengths, thereby changing the column travel

speed and causing the components to separate [1].

1.1.1 Conventional Gas Chromatography

The two most common types of gas chromatography utilize either packed columns or capillary columns. Packed columns have a stationary phase coated on the packing in the column. Packing consists of small beads filling the column, which the liquid phase adheres to. In a capillary column, the stationary phase is coated on the internal wall of the column. Because the column is not packed with material, higher flow rates are possible.[1].

1.1.2 Micro Gas chromatography

Micro Gas chromatography (µGC) is a version of GC meant for deployment in the field. By miniaturizing a GC and reducing the cost, analysis can be done in areas not accessible to traditional laboratory devices. The current approach for such miniaturization is to use microelectromechanical systems (MEMS) to replace components with microfabricated devices [1].

1.1.3 Chromatograms

GCs output chromatograms. Chromatograms are generated by a sensor inside the GC, and display a graph proportional to the amount of the mixture leaving the column over time. For a column properly designed to separate the components of the mixture under test, the components will leave the column at different times, resulting in peaks as shown in Figure 1.1 [2]

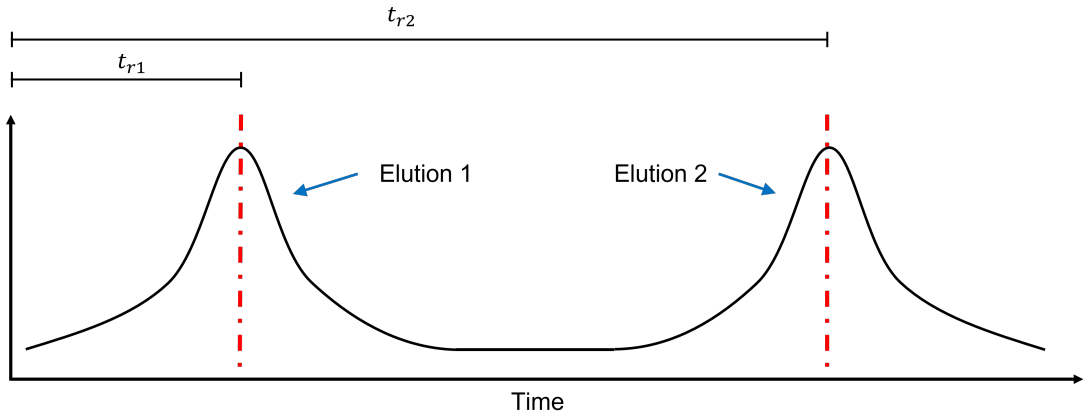


Figure 1.1: Ideal Chromatogram

In an ideal chromatogram, there will be completely separated peaks corresponding with the components of the mixture. The time at which the peak occurs is known as the retention time. The retention time is unique to each combination of chemical and experimental set up [2].

With non ideal chromatograms, the peaks may partially, or even fully overlap (Figure 1.2). This can be the result of a faster analysis, shorter separation column, or non-ideal column configuration [1].

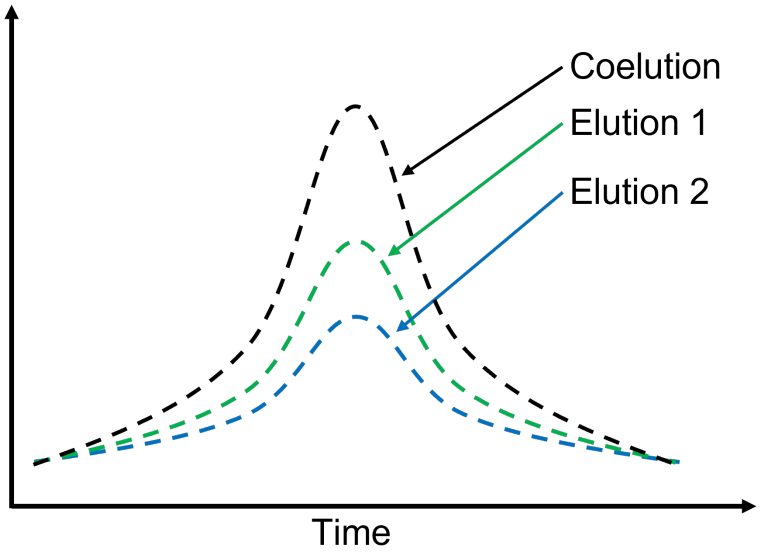


Figure 1.2: Non-ideal Chromatogram

1.2 Machine Learning

Machine learning is a type of artificial intelligence where a machine can analyze data to learn on its own how to process the data. The standard formal definition in the field as written by Tom Mitchell is:,

> A computer program is said to learn from experience **E** with respect to some class of tasks **T** and performance measure **P**, if its performance at tasks in **T**, as measured by **P**, improves with experience **E**. [4]

Typical chromatography data requires careful analysis by experts in order to classify a mixture. By applying machine learning, analysis time can be significantly reduced and automated.

1.3 Motivation

µGCs are a potential way to employ gas chromatography in the field, however, by themselves they require extensive domain-specific knowledge to be able to fully analyze chromatograms. Additionally, they require well separated components in order for researchers to identify components. By finding effective machine-learning techniques and applying them, the chromatograms do not require nearly as much separation, allowing the analysis times to be much shorter. Machine learning algorithms would also allow more automation in the analysis process, thereby opening up its use to individuals with less training.

1.4 Contributions

The following are the major contributions of this thesis:

- We explore the effectiveness of applying machine learning algorithms to one-dimensional gas chromatography data sets. The algorithms investigated are K-Nearest Neighbour, Support Vector Machines, Convolutional Neural Network, Decision Tree, and Random Forest.
- We investigate the effects of applying feature selection, feature engineering, and data fusion.
- We show that after applying feature selection, K-Nearest Neighbour and Random Forest performed optimally compared to other algorithms.
- We provide a qualitative comparison of different algorithms, not only based on their performance but also on how difficult it is to set them up and train them for optimal performance.

Chapter 2

Background

In this chapter, we explore previous work done on applying machine learning to gas chromatography. This will help us to narrow down the focus to certain algorithms. We then provide an overview of the algorithms and techniques tested by this thesis.

2.1 Previous Work on Machine Learning Applied to Gas Chromatography

For determining which algorithms to investigate, multiple fields were searched. Literature was searched for applying machine learning to gas chromatography and chemometrics for time-series analysis.

There are many papers discussing the application of ML to gas chromatography and chemometrics [5], [6], [7], [8], [9], [10], [11] [12], [13], [14], [15], [16]. However, the focus of the majority of the literature is higher resolution techniques, such as gas chromatography with mass spectrometry or multidimensional gas chromatography. These techniques consist of multidimensional data rather than 1-dimensional data. Chemometrics, however, do have much more literature considering the analysis of 1-dimensional data, but the majority of techniques are applied to spectrometry data rather than chromatography data [17], [18], [19], [20].

Machine learning is being applied for a variety of applications, such as disease detection [5], [7], [10], [11] and food analysis [6], [8], [9], [15], [16], [18]. Through all the literature, there is no consensus on a single machine learning method to use for classification. A variety of algorithms are used, the most common of which are k-nearest neighbour, support vector machines, convolutional neural network, decision tree, and random forest. These are the algorithms explored by this thesis.

2.2 Machine Learning

This section will provide an overview of the algorithms tested (k-nearest neighbor, support vector machines, convolutional neural networks, decision trees, and random forests). Throughout this thesis we will discuss 'hyperparameters'. Hyperparameters are variables that the researcher changes, instead of the model changing the parameter itself during training.

2.2.1 K-Nearest Neighbor (KNN)

The K-Nearest Neighbor algorithm classifies points based on the class of the 'K' number of 'nearest' neighbors. A distance calculation is performed between the input data-point and all of the training data points, and the **K** amount of points with the smallest distances are chosen. The method of calculating distance is a hyperparameter left to the user to decide.

Next, the input is labeled based on the class of those nearest points. The method of deciding on class is also a hyperparameter for the designer to use. A straightforward method is to label based on whichever class has the majority. More complex approaches involve weighting the vote based on the distance, thereby allowing closer data points to have more weight on

the decision [21]. Figure 2.1 illustrates the KNN algorithm with $K = 3$.

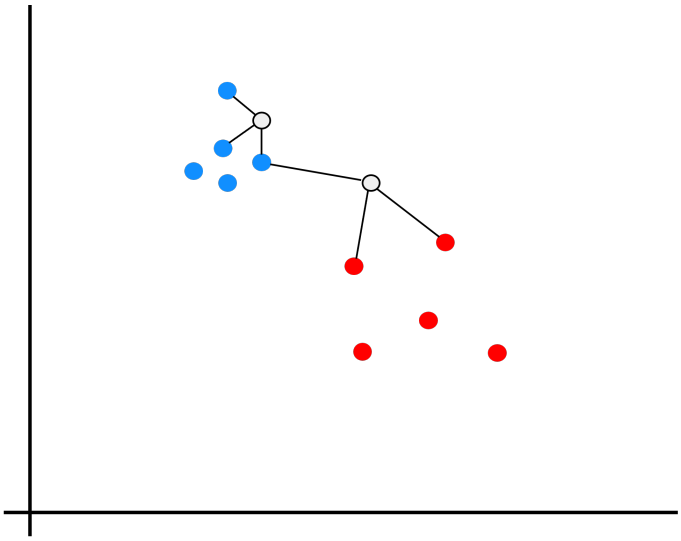


Figure 2.1: KNN Algorithm (K=3)

2.2.2 Support Vector Machines (SVMs)

Support Vector Machines (SVMs) is a method to classify the data set by dividing it with hyperplanes. The method requires that the distances between the hyperplane and the data points are maximized [22]. SVMs have several general advantages. First, they work well with large amounts of dimensions, including cases where there are more dimensions than sample sizes. This fits chromatography data sets well, since each chromatogram usually has thousands of points, while the total number of samples is usually much smaller than that [23]. Second, unlike nearest neighbor algorithms, it only uses a small amount of training points in the model, so it uses much less memory. And third, with the use of kernel functions, the algorithm can have nonlinear decision boundaries. However, the downside of SVMs is that they have a greater risk of overfitting when the there are many more features than samples [23].

Different kernel functions alter the decision boundary (Figures 2.2 and 2.3).

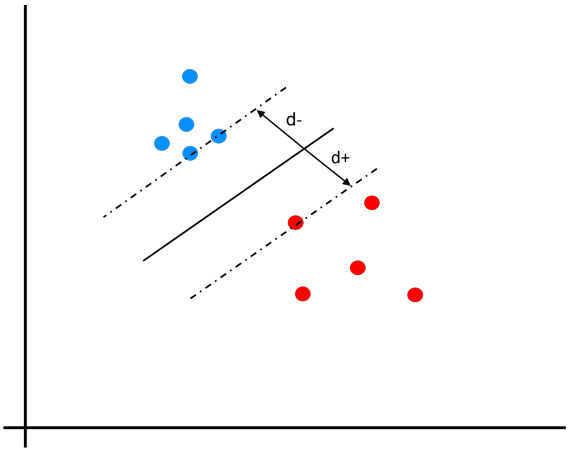


Figure 2.2: Linear SVM

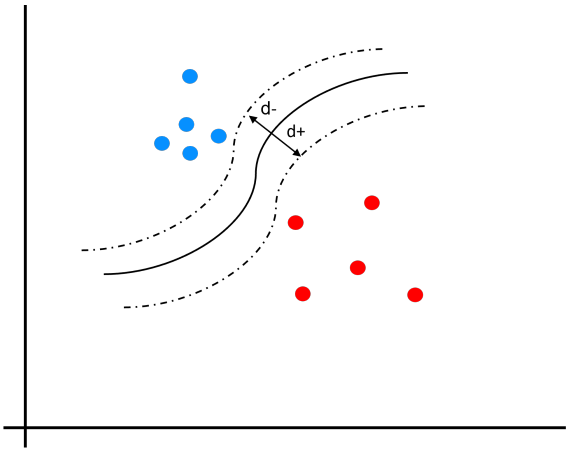


Figure 2.3: Nonlinear SVM

There were 4 different kernel functions tested:

- **linear**: $\langle x, x' \rangle$
- **polynomial**: $(\gamma \langle x, x' \rangle + r)^d$
- **radial basis function (rbf)**: $\exp(-\gamma ||x - x'||^2)$
- **sigmoid**: $\tanh(\gamma \langle x, x' \rangle + r)$

There are several hyperparameters available for tuning the model. The first is the kernel

function used. The second is the parameter c, which exchanges accuracy for simplicity of the decision boundary. The third is the parameter γ which controls how much a single training sample can affect the model.

2.2.3 Decision Trees (DTs)

Decision tree classifiers work by applying a series of logical tests to the input. Each test either leads to a class, or points to an additional logical test. By representing each test as a node, and each class as a leaf node, this classifier forms a tree structure [24]. Figure 2.4 illustrates a decision tree. To create a decision tree, the Gini Impurity is calculated for each feature. The feature with the lowest Gini Impurity is selected. The Gini Impurity at a node in a tree is the probability of an input being classified incorrectly at that node. Formulated mathematically, the Gini Impurity H at a node Q_m is defined as

$$H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$$

where p_{mk} is the proportion of observations of class k at node m [24].

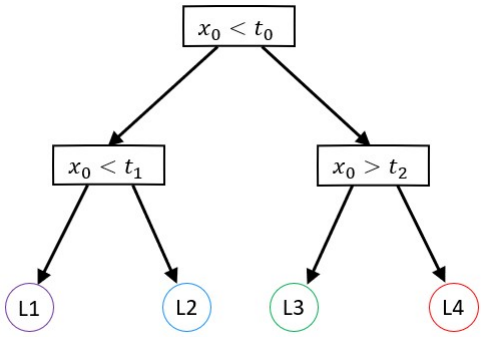


Figure 2.4: Decision tree example.

2.2.4 Random Forests (RFs)

A random forest is a set of decision trees, where each tree is trained on a random subset of the training data. This mitigates overfitting. Predictions are made using a majority vote of the predictions of each tree [25] [26]. Figure 2.5 shows an example of a random forest.

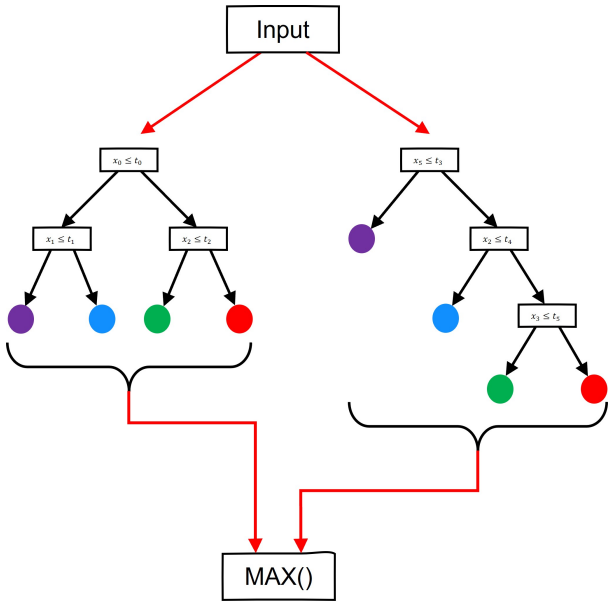


Figure 2.5: Random forest example.

2.2.5 Neural Networks

Artificial Neural Networks (ANNs) are a method of machine learning modeled after the nervous systems found in the human brain [27]. Features of the input are passed to neurons, which connect to successive hidden layers. The final layer produces an output. In the context of classification, the output will consist of a layer with the same amount of neurons as the number of labels of the training set. The neuron with the highest value corresponds with the probability of that label. Supervised learning consists of testing what a network produces as a result of known inputs, and then adjusting the weights and biases of the network so that

the actual output matches the correct output [27].

Convolutional Neural Networks (CNNs) are a special case of ANNs. With traditional neural networks, every input connects to every neuron in the first layer. With high dimensional data, such as images, this results in extremely large networks, which are very expensive to train. With a convolutional neural network, a discrete convolution is performed on the input, resulting in a much more manageable output size for that first layer. [28].

2.3 Feature Selection

Feature selection is a technique for increasing classifier accuracy by choosing only a subset of the input features. The selection of data can be based on domain specific knowledge, or can be searched for as part of the hyperparameter tuning process. By removing unnecessary features, this can reduce the model's training time and increase accuracy [29].

2.4 Feature Engineering

Feature engineering is the process of transforming input features in such a way as to increase effectiveness of the model [30].

2.4.1 General Operations

Standard mathematical operations such as addition, subtraction, multiplication, division, and exponentiation can be applied to input features. These operations can be used for multiple purposes. For example, many machine learning models suffer from hardware limitations on the range of acceptable values of internal storage, therefore it is standard practice to scale

and normalize the input[30]. Another common application is with data fusion. Data from different types of sensors will usually have different ranges of values. If this data is merely concatenated, it is entirely likely that one set of features will have a higher impact than the other set, just by having a different variance or average[30].

2.4.2 Principal Component Analysis (PCA)

Principal Component Analysis is a method of dimensionality reduction. An input with a high number of dimensions can be converted to an output with a lower number of dimensions. A linear combination of the input features is used, where the weights for each feature are such that the output explains as much of the variance as possible. This can be used for more easily interpreting high-dimensional data, or for transforming data into a form easier to process[31].

2.5 Data Fusion

Data Fusion is the process of combining multiple data sets to produce a new, more useful, dataset. Data Fusion is quite a large field, with techniques covering many areas, from combining data from sensors with different reliabilities and sampling characteristics, to combining data at different levels of the system [32]. The specifics of how data fusion is applied in this thesis are discussed in Chapter 4.

2.6 Conclusion

In this chapter we gave an overview of machine learning, feature selection, and feature engineering. For machine learning, we discussed the five algorithms used in this thesis: k-nearest neighbor, support vector machines, decision trees, random forests, and neural networks. For feature selection, we discussed its uses and application. For feature engineering we gave a general overview, and discussed principal component analysis, which is also employed later in this thesis.

In the next chapter, we will present the methodology this thesis used. The data used in the analyses will be described. A justification of the algorithms used will be given, and the evaluation process for the algorithms will be described in detail.

Chapter 3

Methodology

This chapter discusses the overall strategies used for this thesis for achieving the research goal. The goal of this thesis is to explore the effectiveness of machine learning algorithms for classifying gas chromatography data. Towards that end, suitable data for analysis was collected; and ML models from Section 2.1 were trained, evaluated, and compared.

3.1 Data

In order to test the effectiveness of the algorithms, it was necessary to gather data. The VT MEMS Lab provided readily accessible data from previous experiments.

There were three sets of data used. Two sets consisted of chromatograms of mixtures of 22 volatile organic compounds (VOCs) with hexanol and 2-butanol, used for testing different column geometries of the lab's GC. The data from these sets were used in Experiments 1-7. The other set consisted of chromatograms of kerosene mixed with diesel, used for a paper on detecting fuel adulteration [33]. This data was used in Experiments 8 & 9. One of the sets had data from two sensors, as well as peak integration data. This data was used for Experiments 6 & 7.

For all of the experiments, a fast GC approach was used, which uses aggressive temperature and pressure programming. For experiments 1-7: the initial temperature was 60 °C with 60

°C/min temperature ramp and a final temperature of 120 °C; the carrier gas had a 20 psi pressure; and a split mode was used for injection with a split ratio of 400:1 and an injection volume of 0.1 µL.

For experiments 8 & 9: the initial temperature was 80 °C with 100 °C/min temperature ramp and a final temperature of 150 °C; the carrier gas had a 40 psi pressure; and a split mode was used for injection with a split ratio of 200:1 and an injection volume of 0.3 µL.

By using already existing data sets, testing with the models could begin immediately. As discussed in Ch. 1, running a traditional gas chromatogramer can be time-consuming.

Additionally, these data sets were also chosen because they had enough samples for the evaluation process discussed later in this chapter in Section 3.2.

3.2 Model Evaluation Process

The selection of models to evaluate was discussed in Section 2.1.

To evaluate the effectiveness of algorithms, the accuracy of those algorithms was tested and compared. Cross-validation was used in the training and evaluation process.

3.2.1 Cross-Validation

Cross-validation is a method for avoiding bias and overfitting during the machine learning process. When models are being trained on data, there is a risk of the model being overfitted. In such a case, rather than learning the true underlying relationships differentiating the classes, the model effectively memorizes the answers. This results in a very high score on the training data set, but extremely poor scores when used with unseen data. To get a more

realistic evaluation of the model, part of the data can be reserved for testing, while the rest is used for training. However, even with this method, there is still a risk of bias. Since the model would be evaluated and modified based on its performance on the test data, the model in effect becomes biased towards the test data set. Cross-validation is a method to mitigate this. In cross-validation, part of the dataset is withheld from the training set. The withheld data is then used for evaluating and tuning the model. A further portion is withheld, and only used for final evaluation[23].

A downside of this approach is that for small data sets this can result in bias from how the data is split. To mitigate this, a variant of cross-validation called k-fold cross-validation is used. With this method, the initial data set is split into training and validation sets k times. Each set is used for training and evaluating the model, and the final score is based on the average of the score for each fold (Figure 3.2).

Additional consideration should be given to how data is split for each fold. The split can be completely random, which helps mitigate bias; however this opens up the possibility of data classes being unevenly allocated between the training and validation sets, resulting in insufficient training on some classes or validation sets with blind spots.

One technique to address this is stratified cross-validation. For stratified cross-validation, data is split evenly for each class, so that each class is equally represented in the training and validation sets. This ensures that no particular class has a disproportionate effect on the results.

Figure 3.1: Stratified K-Fold Cross-Validation (k=5)

3.2.2 Final Model Evaluation Process

After taking into account the issues mentioned above, the final process for tuning and evaluating algorithms used for this work are shown in Figure 3.2. The exact steps are detailed below. For the train-test split, an 80/20 ratio was chosen since it is standard in the field.

1. 20% of the data is withheld for testing. The testing set is used for final evaluation of the model.

2. The data that's left is then used for stratified k-fold cross-validation. The data is split into **k** folds, with each fold containing different splits of the data into training and validation sets.

3. For each fold, the model is trained on the training set, and then evaluated with the validation set.

4. The model is then tuned by changing the model's hyperparameters.

5. The model tuning is repeated for each fold until performance on the validation set is satisfactory.

6. Once the model is tuned, it is evaluated using the testing set.

7. Final model selection is made based on the final testing scores of each model.

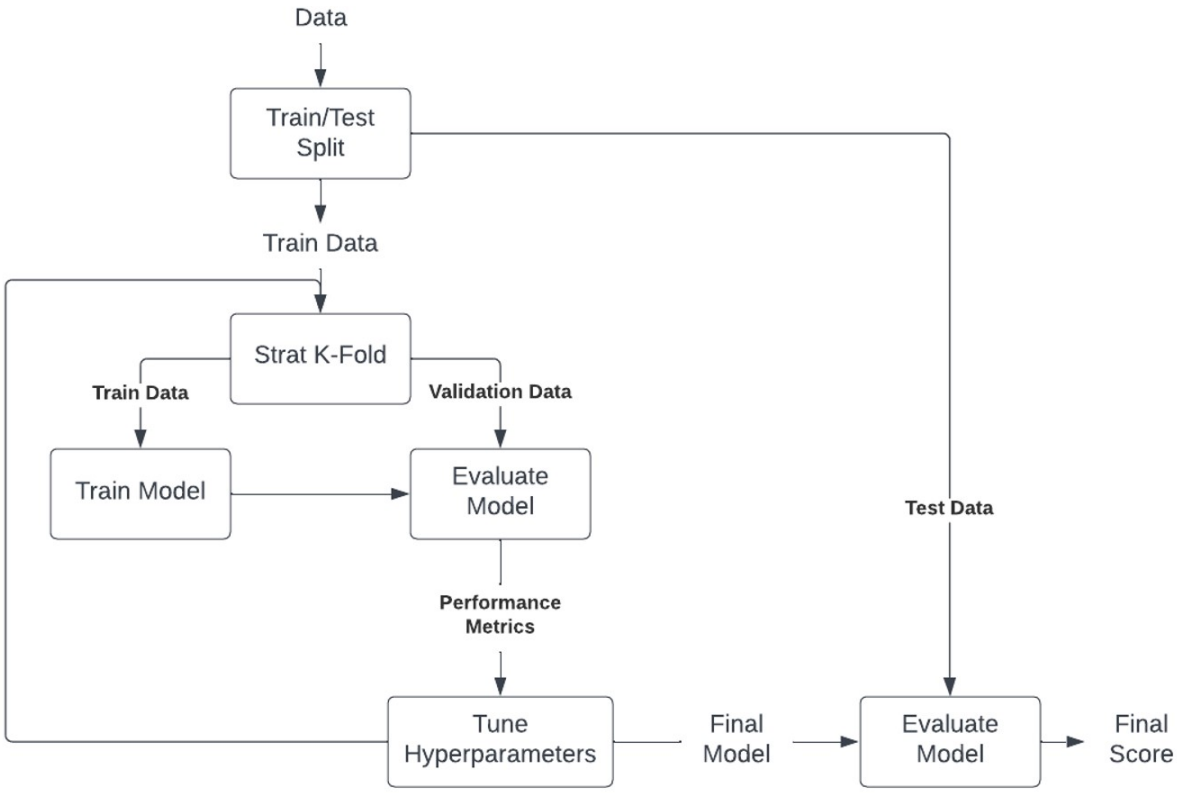


Figure 3.2: Model Evaluation Process

Some data sets consist of an additional blind set. For these data sets, a slightly different methodology was used.

1. The blind data set is used for testing, while the entire non-blind set is used for training and evaluation.

2. The non-blind data is used for stratified k-fold cross-validation. The data is split into **k** folds, with each fold containing different splits of the data into training and validation sets.

3. For each fold, the model is trained on the training set, and then evaluated with the validation set.

4. The model is then tuned by changing the model's hyperparameters.

5. The model tuning is repeated for each fold until performance on the validation set is satisfactory.

6. Once the model is tuned, it is evaluated using the blind set.

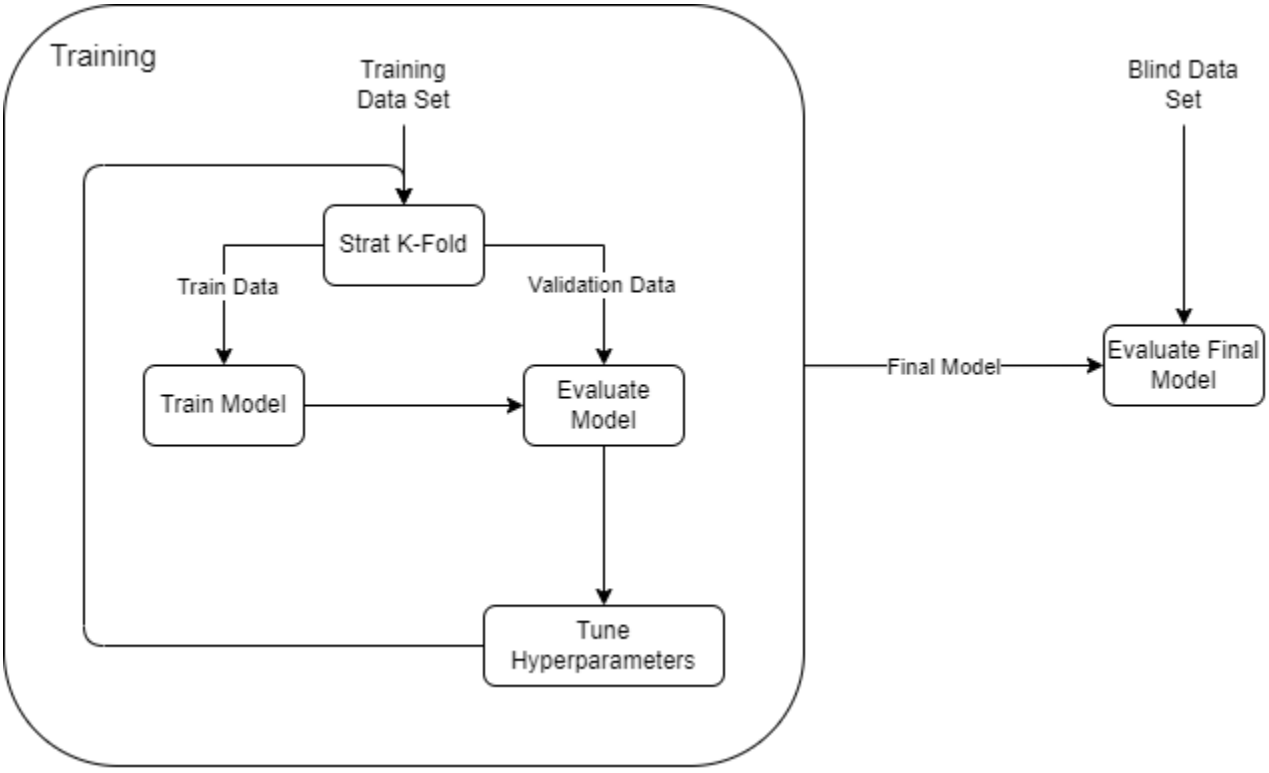


Figure 3.3: Model Evaluation Process for Data Sets with Blind Sets

3.2.3 Model Comparison Process

The final model choice is based on the score with a testing set consisting of unseen data. The score from hyperparameter tuning completely avoids use of the test set in order to avoid

bias. The final score for a model is calculated as follows:

$$\text{Accuracy} = \frac{\#\text{ correct}}{\#\text{ predictions}}$$

Models of each algorithm type and with the different preprocessing steps are compared based on their final score to determine which features offer better performance.

3.3 Conclusion

In this chapter we described the methodology used by this thesis. We provided reasoning behind the data chosen, justification for the ML models analyzed, and a detailed description of the training and comparison process.

In the next chapter, we provide detailed descriptions of the data, and present initial analyses run on the data.

Chapter 4

Data Analysis

This chapter describes the data sets used in this thesis. Principal component analysis is performed on the data, and analysis used for feature selection is also shown.

4.1 Data Overview

A variety of mixtures and columns were used so as to better assess the applicability of the algorithms to different shaped chromatograms. Overall there were three data sets, which shall be called Data Set 1, Data Set 2, and Data Set 3. Data Set 1 and 2 consisted of chromatographs of four mixtures: Mixture 1, Mixture 2, Mixture 3, and Mixture 4. The data sets used Volatile Organic Compounds (VOCs). The components of these four mixtures were as follows:

- **Mixture 1**: 22 VOCs
- **Mixture 2**: 22 VOCs + hexanol
- **Mixture 3**: 22 VOCs + 2-butanol
- **Mixture 4**: 22 VOCs + hexanol + 2-butanol

For Data Set 1, each mixture was analyzed with five different columns. These are denoted Experiments 1-5.

For Data Set 2, only one column was used. However, the column had two sensors: one at the back and one at the front. Each sensor generated chromatograms. Additionally, peak integration was included in the data set, after being calculated from the chromatograms of both sensors. The sensor and peak integration data was gathered into one data set for testing data fusion (Experiment 6). To evaluate the effect without data fusion, another data set was formed just from the front sensor (Experiment 7).

Data Set 3 consisted of three mixtures of diesel adulterated with kerosene, with varying concentrations (Mixture 1, Mixture 2, Mixture 3). The mixtures were:

- **Mixture 1**: 90% diesel and 10% kerosene
- **Mixture 2**: 95% diesel and 5% kerosene
- **Mixture 3**: 100% diesel and 0% kerosene as a control

Each mixture was analyzed by two separate columns. The data from one column was used for Experiment 8, and data from the other column was used for Experiment 9.

Additionally, with ideal chromatographs in traditional GC, there will be distinct differences in peak locations with different mixtures. These chromatographs were generated for testing with µGC, therefore the experiments were run much faster, and resulted in less separation. Thus, for a given experiment, the peaks overlap more, although there are still distinct differences.

Figure 4.1 shows an overview of the data sets and experiments derived from the data sets.

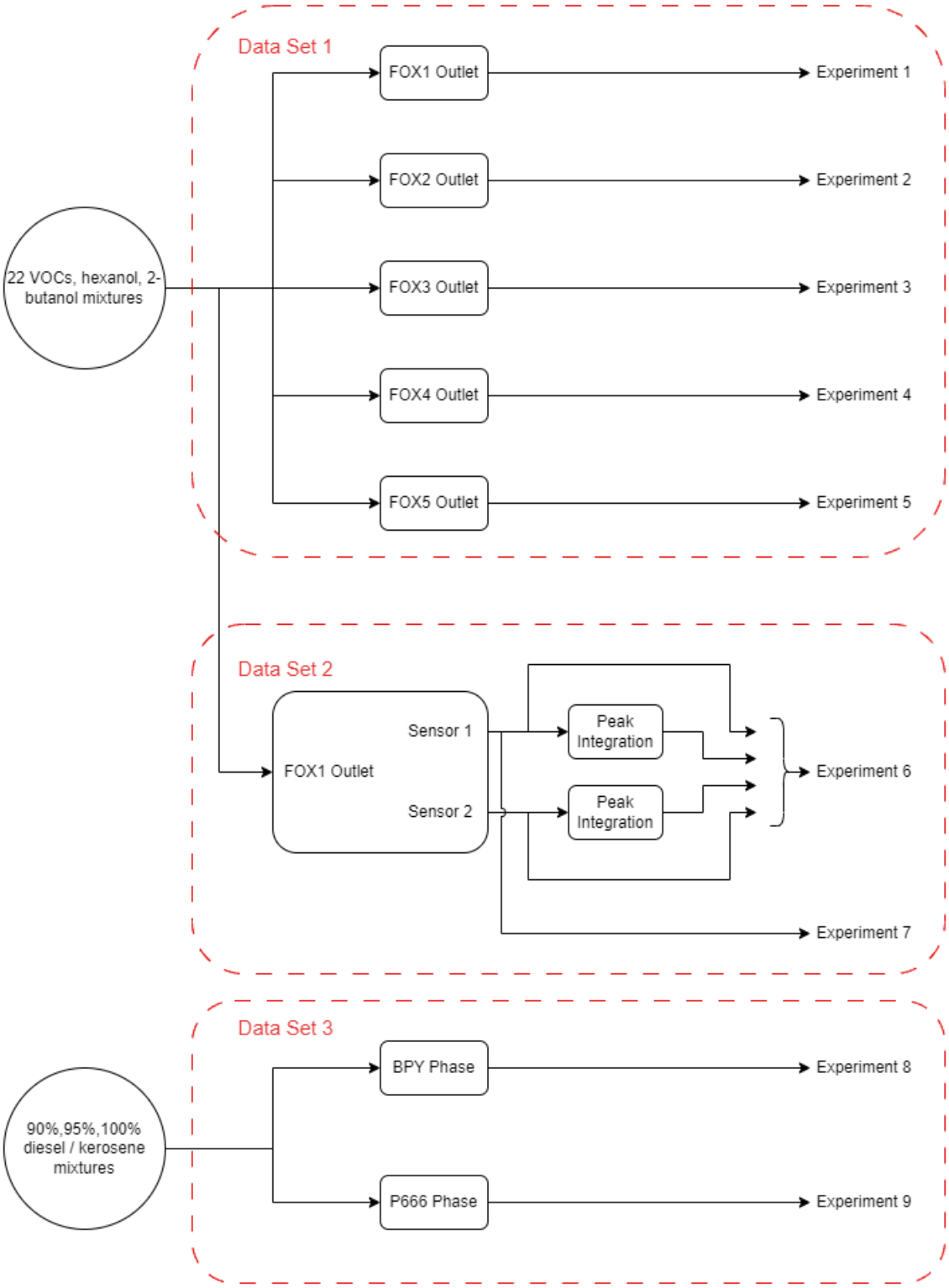


Figure 4.1: Data Overview

4.2 Data Set 1

The 4 mixtures will be referred to as Mixture 1, Mixture 2, and so on. The full data samples include 4500 points, however separation occurs within the first 2500 data points. The remaining points are near-zero for the rest of the chromatogram. The 4 mixtures were separated in 5 different columns, which will be referred to as Experiments 1-5, respectively.

Experiments 1-5 are shown in Figures 4.2 - 4.6. The sensor initially shows zero while the column is heating up. Additionally, because different combinations of mixtures and columns take different amounts of time to fully elute, the machine was set to gather data for an extended period to cover all of the elutions, resulting in near-zero values at the end of the chromatogram.

The chromatograms vary significantly between each column because of the differences between the columns. Since the column of the chromatogram is known beforehand, and to improve accuracy in the analysis, models were trained for each column individually.

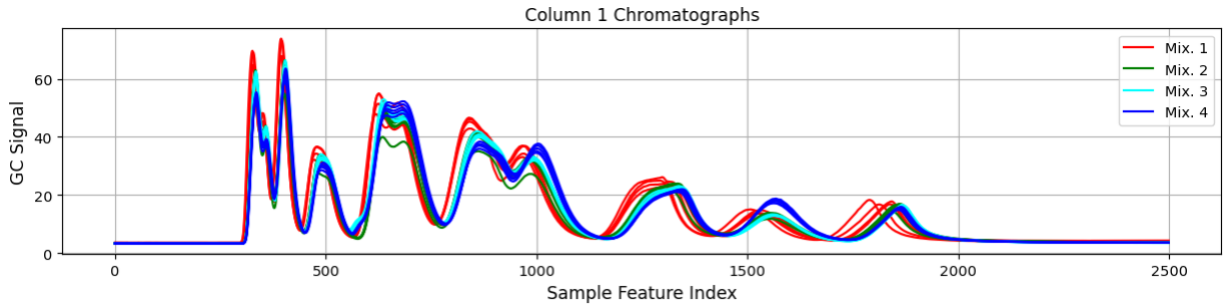


Figure 4.2: Experiment 1

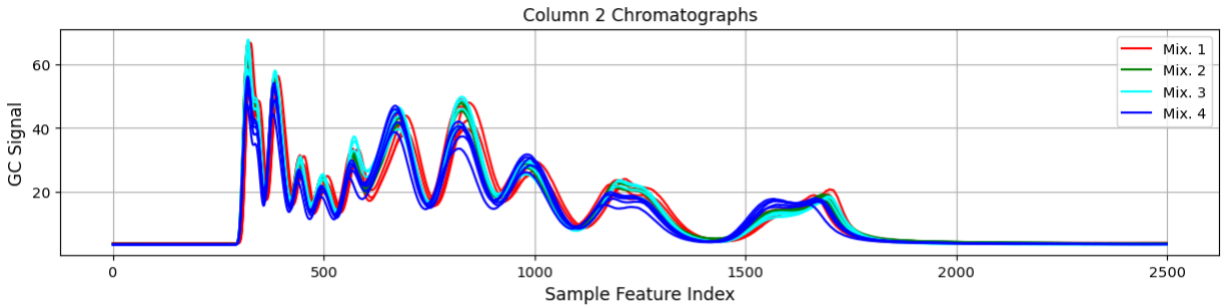


Figure 4.3: Experiment 2

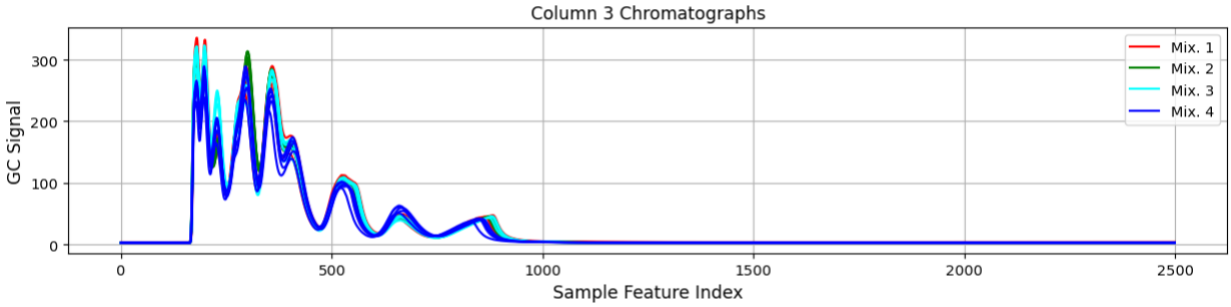


Figure 4.4: Experiment 3

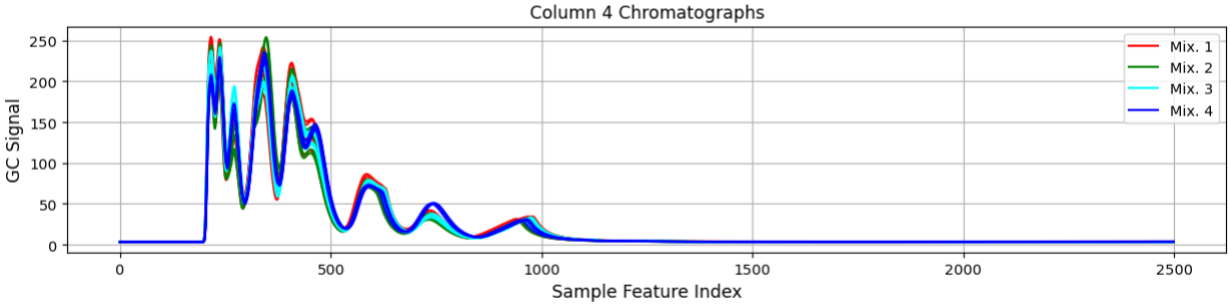


Figure 4.5: Experiment 4

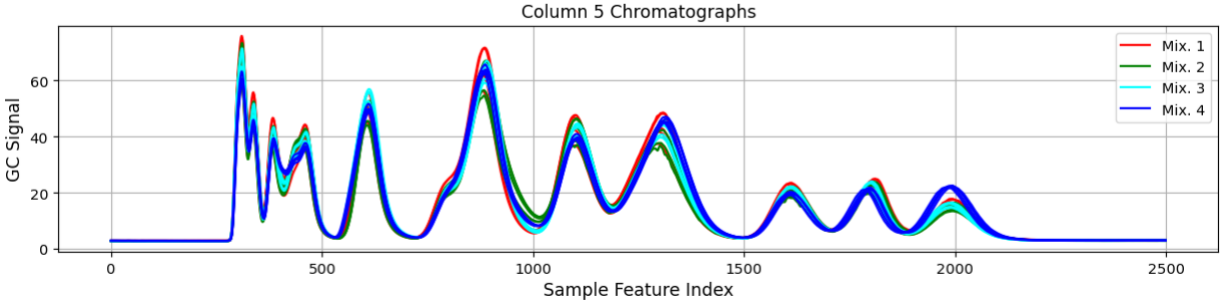
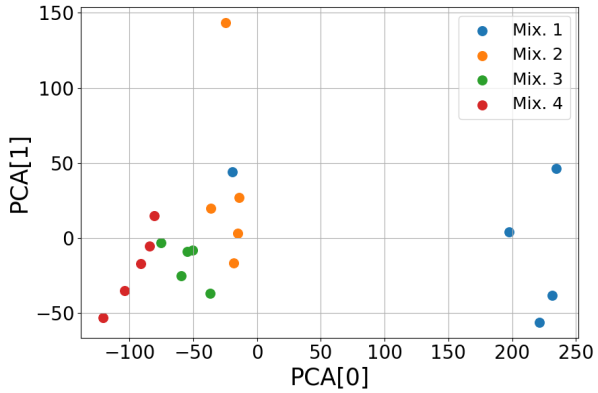


Figure 4.6: Experiment 5

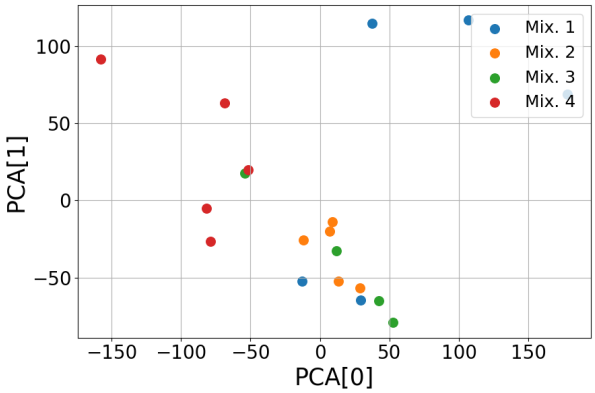
4.2.1 Analysis

Next, the data set was analyzed with PCA. As can be seen from Figure 4.8, the first two components hold the majority of the variance for Experiments 1-5, however for 4 & 5, they hold less than 80% of the variance. Therefore the first two components of PCA are shown in 4.7a - 4.7c, and the first three are shown for 4.7d and 4.7e.

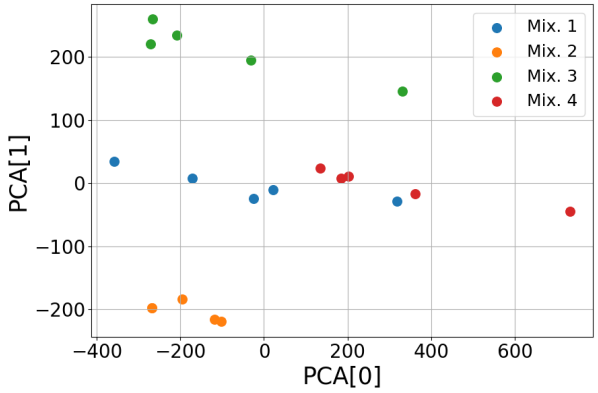
Experiments 1 and 3 show relatively well separated clusters, whereas Experiments 2, 4, and 5 have more overlap.



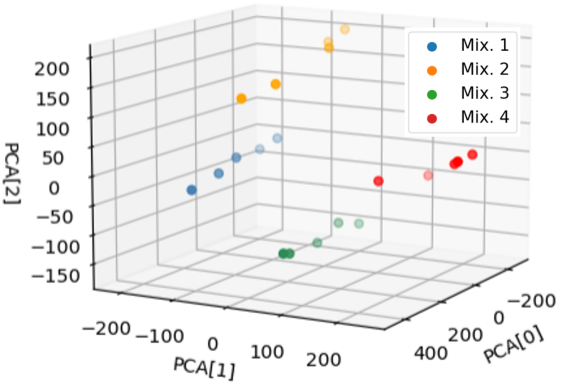
(a) Experiment 1



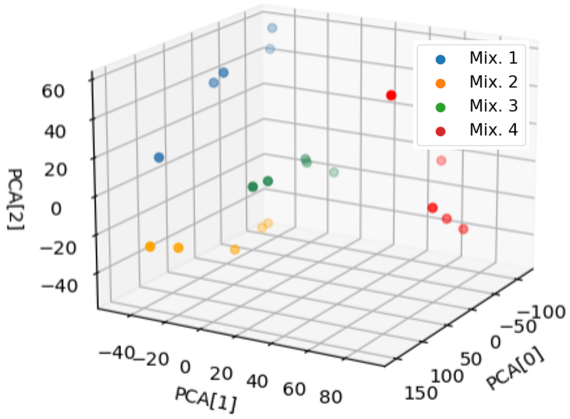
(b) Experiment 2



(c) Experiment 3

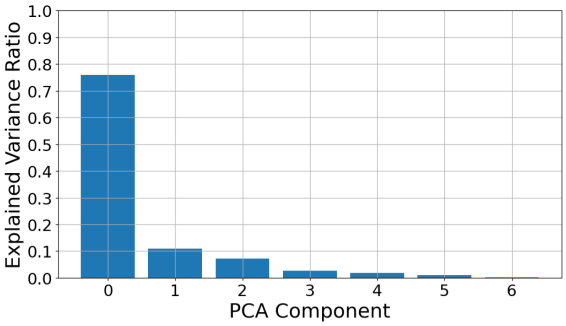


(d) Experiment 4

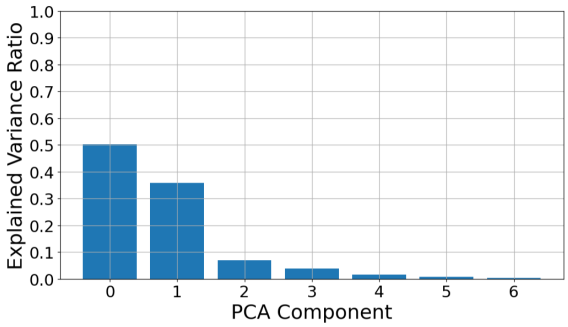


(e) Experiment 5

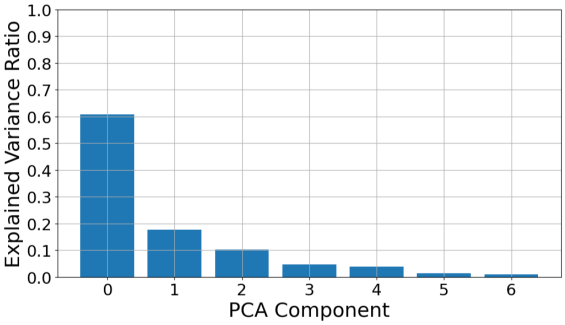
Figure 4.7: Experiments 1-5 PCA



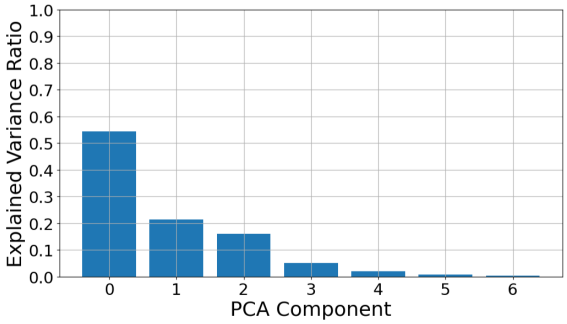
(a) Experiment 1



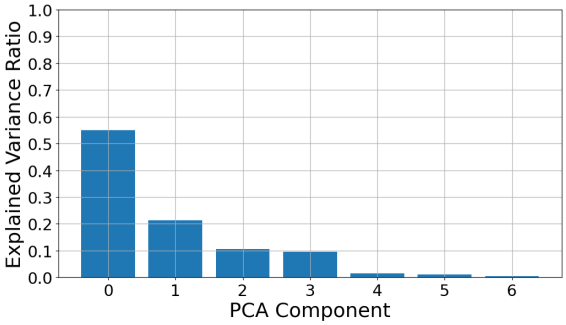
(b) Experiment 2



(c) Experiment 3



(d) Experiment 4



(e) Experiment 5

Figure 4.8: Experiments 1-5 PCA Variance Ratio

4.2.2 Feature Selection

By looking at the data set, it can be seen that this data set is a good candidate for a variance threshold. There are substantial sections of the chromatograms with very little variance.

After applying a variance threshold, it is important to scale and normalize the data so that the mean is zero and the variance is one. Figures 4.9 - 4.13 show the results of applying a variance threshold, scaling, and normalization.

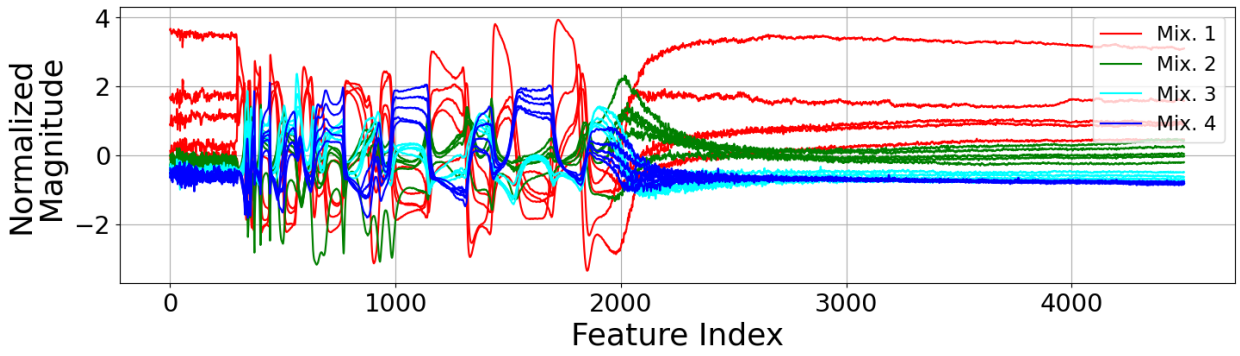


Figure 4.9: Experiment 1 Thresholded and Normalized

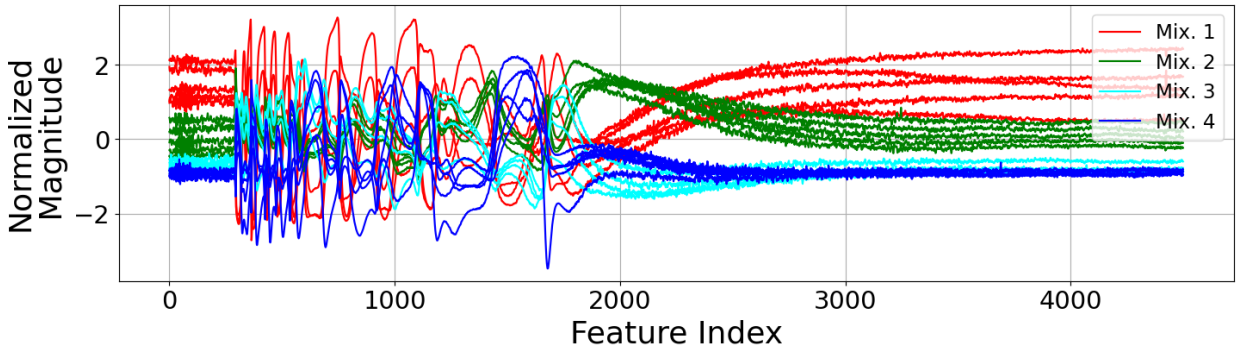


Figure 4.10: Experiment 2 Thresholded and Normalized

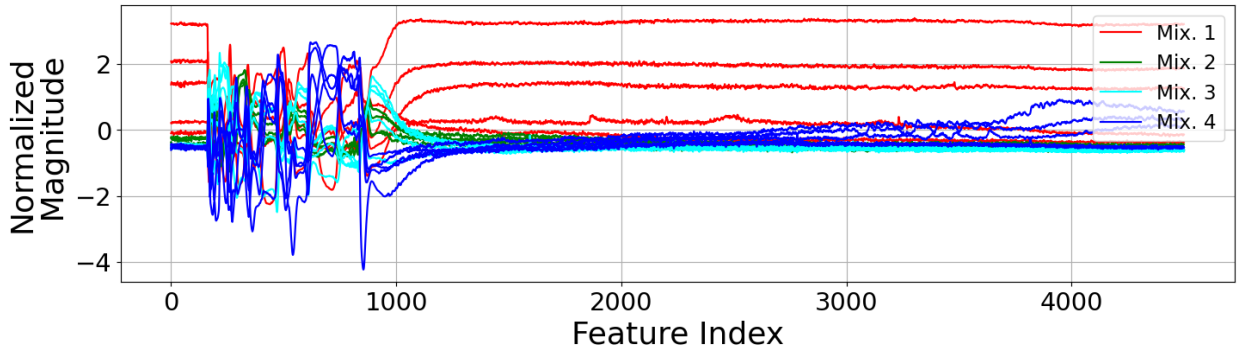


Figure 4.11: Experiment 3 Thresholded and Normalized

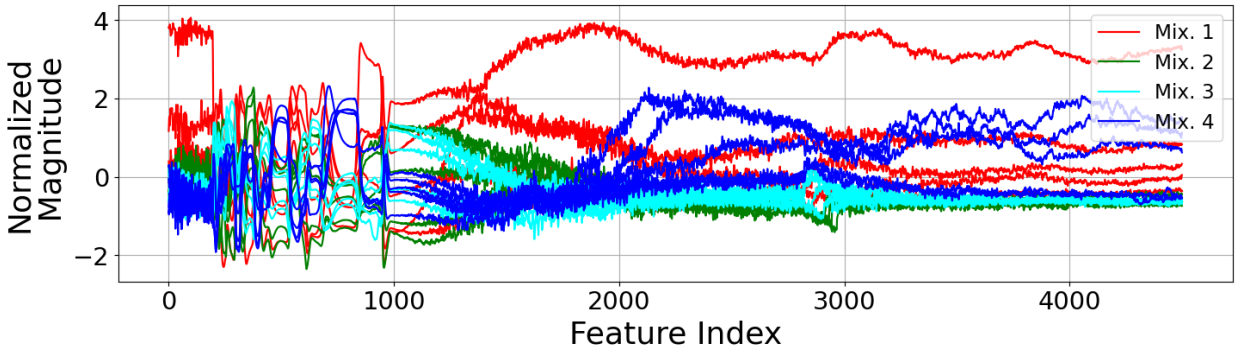


Figure 4.12: Experiment 4 Thresholded and Normalized

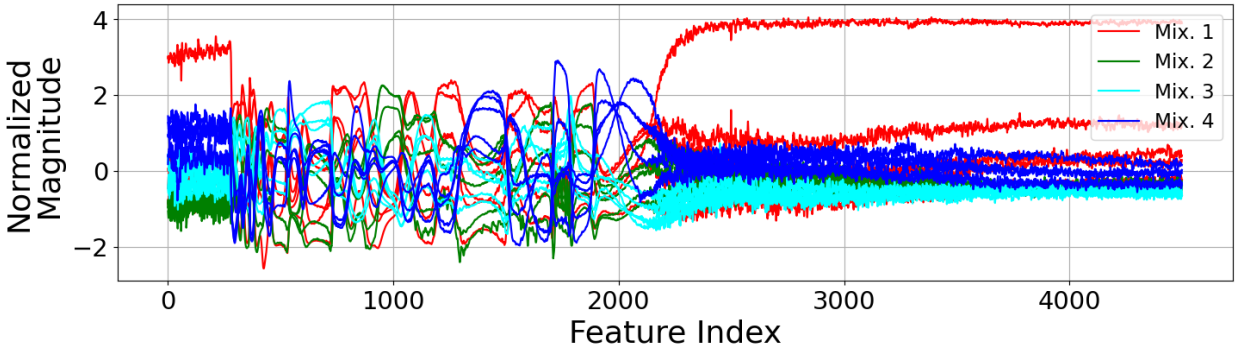


Figure 4.13: Experiment 5 Thresholded and Normalized

An analysis was performed to determine the optimal features to use. A window sweep of the chromatogram features was performed, within which the window size and window location were varied. The performance of the window was evaluated with the KNN algorithm.

The parameters tuned for the KNN algorithm by itself were the number of points used for the nearest neighbor calculations, and the weighting system used with voting. A distance-based weighting system was used for Experiments 1-5 because the distance between clusters was typically greater than the distance between samples within the same cluster. Aside from the algorithm itself, the window size and location were also swept. The location of the window refers to the index of the last feature within the window. The system was tested with and without PCA. For PCA, the system was tested with variable amounts of PCA components, with the highest variance components always being used.

Once this analysis was performed, it was found that reducing the feature set reduced performance. Best performance was achieved by including all of the features of the sample.

4.3 Data Set 2

Data Set 2 had two sensors, each generating chromatograms. For each chromatogram, the system calculated peak integration data. Peak integration data consisted of the peak-location, area, height, width, type, peak-start, and peak-end for peaks in a chromatogram. The same mixtures were used for this data set as for Data Set 1 (Mixture 1 ... Mixture 4). This data set was used for testing the effectiveness of data fusion. Figure 4.14 shows the data set after data fusion occurs. Low-level data fusion was used, wherein the data is concatenated. The chromatogram data from the front sensor (Figure 4.15) was used as a control. From here on, the fused data will be referred to as Experiment 6, and the control data will be referred to as Experiment 7.

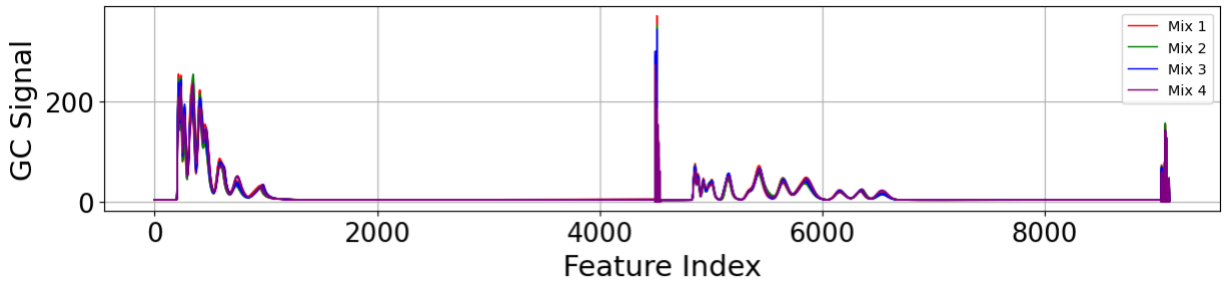


Figure 4.14: Experiment 6

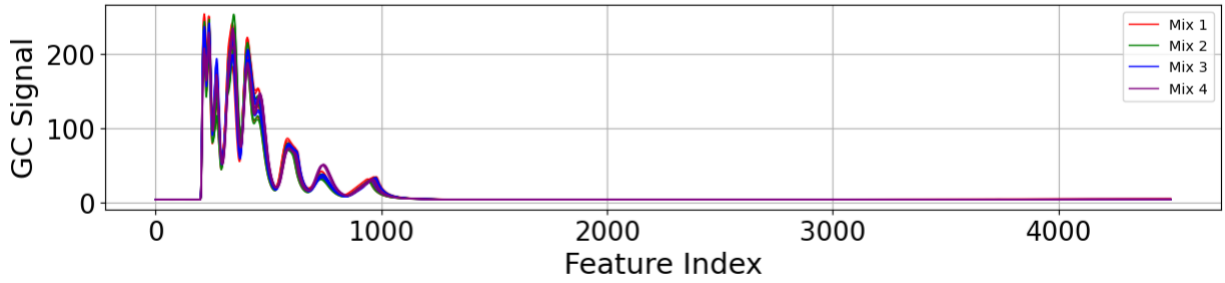
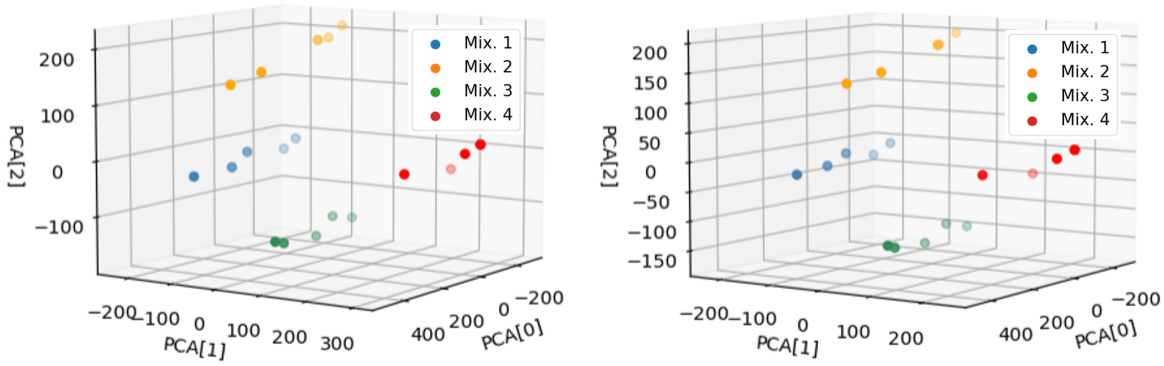


Figure 4.15: Experiment 7

4.3.1 Analysis

Next, the data set was analyzed with PCA (Figures 4.16, 4.17, and 4.18). PCA of Experiment 6 shows relatively poor clustering. Additionally, PCA characteristics are very similar between Experiment 6 and Experiment 7. This is most likely because the chromatogram data has many more features than the peak integration data.



(a) Experiment 6 PCA (b) Experiment 7 PCA (Front Signal)

Figure 4.16: Experiment 6-7 PCA

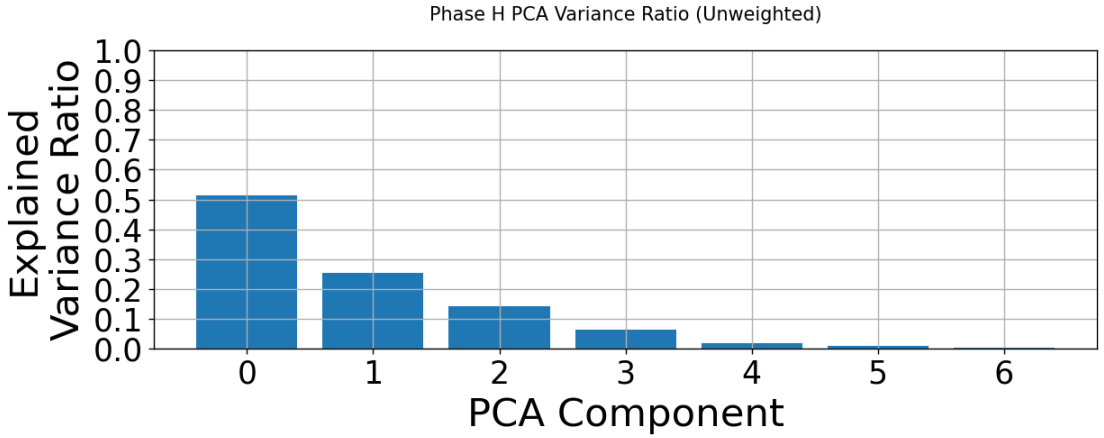


Figure 4.17: Experiment 6 Variance Ratio

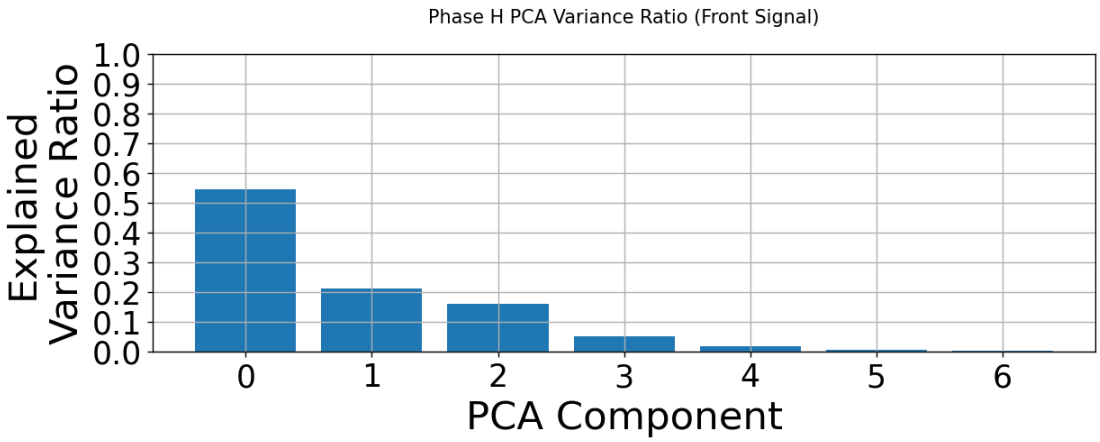


Figure 4.18: Experiment 7 Variance Ratio

4.3.2 Feature Selection

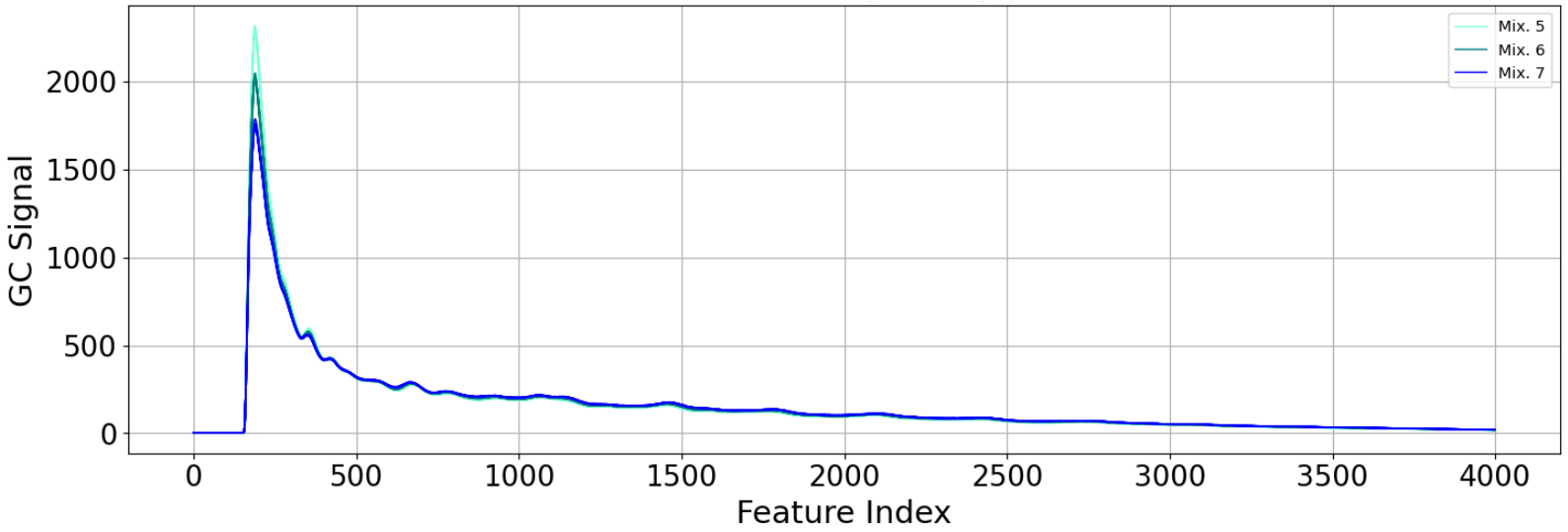
Since the chromatogram data is very similar to the data from Data Set 1, similar steps are followed for preprocessing and feature selection. A variance threshold is applied, and the features are scaled to a mean of zero and a variance of one.

For performing window sweeps for feature selection, the same procedure was followed as with Data Set 1. We performed a window sweep of the features, using the KNN algorithm to evaluate performance of each window. A distance-based weighting was used for the KNN algorithm. The number of points used by the KNN algorithm, the size of the window, and the location of the window used for hypertuning. Additionally, the data set was tested both with and without PCA. All of this was done for the fused data as well as the control data.

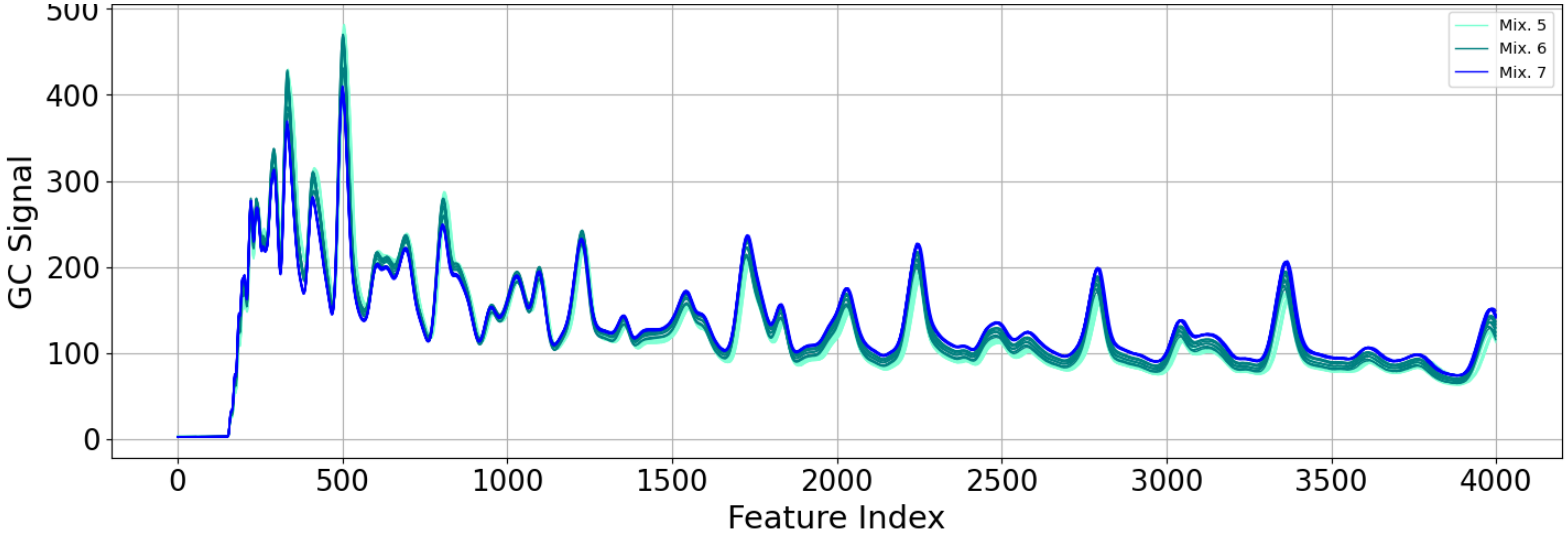
Similar to Data Set 1, feature selection showed no benefits. The best KNN accuracy was achieved by using the full data set.

4.4 Data Set 3

For this data set, aside from the initial sample set, a blind sample set was also taken. The blind set was used as the test set for final evaluation of the model. As mentioned, two sets of samples were taken for each column, one for training and one blind set for testing. The blind sets were used only for testing, not for validation or training. Only the first 4000 features are shown for clarity, however the actual samples have 11249 features.



(a) Experiment 8 Raw Chromatogram Train & Test Data

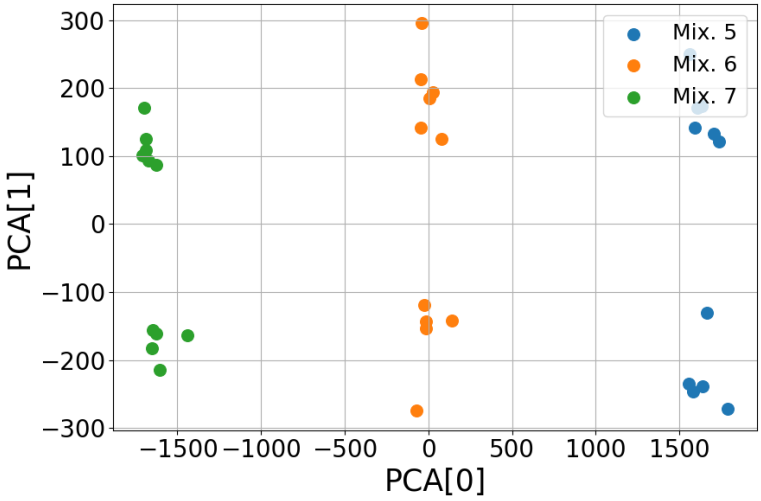


(b) Experiment 9 Raw Chromatogram Train & Test Data

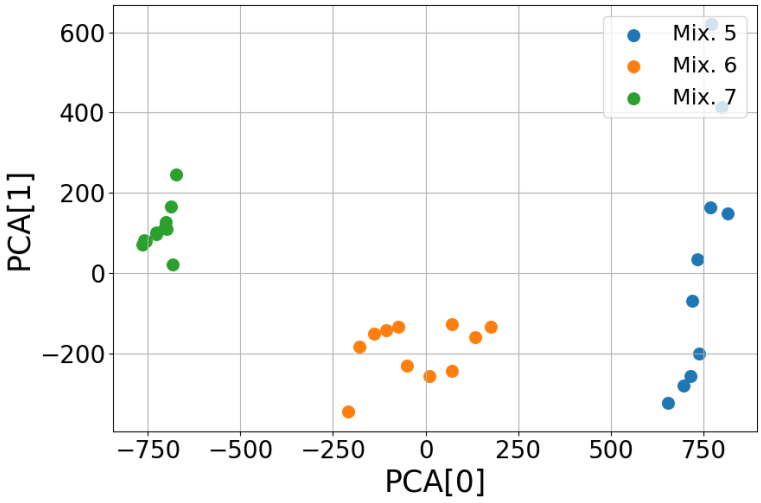
Figure 4.19: Fuel Adulteration Data Set

4.4.1 Analysis

Just as with Data Sets 1 & 2, PCA analysis was performed on Data Set 3. From PCA, it can be seen that Experiments 8 and 9 have well defined clusters. The effectiveness of using PCA is evaluated in later sections.

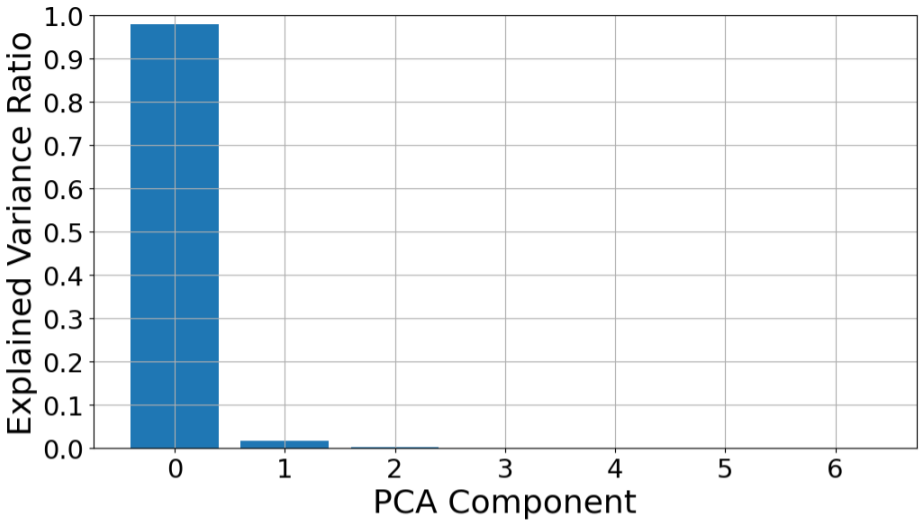


(a) Experiment 8

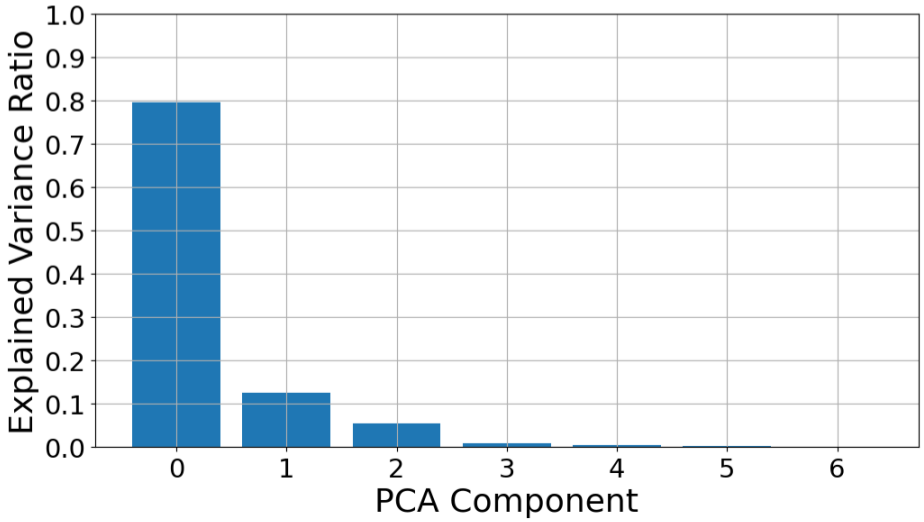


(b) Experiment 9

Figure 4.20: Experiment 8 and 9 PCA



(a) Experiment 8



(b) Experiment 9

Figure 4.21: Experiment 8-9 PCA Variance Ratio

4.4.2 Feature Selection

From the raw chromatograms (Figure 4.19), there are obviously sections with zero variance, that have no useful features for analysis. A simple variance threshold is applied for filtering them out. Once a variance threshold is applied, the features are then scaled to have a mean of zero and a variance of one (Figures 4.22 & 4.23).

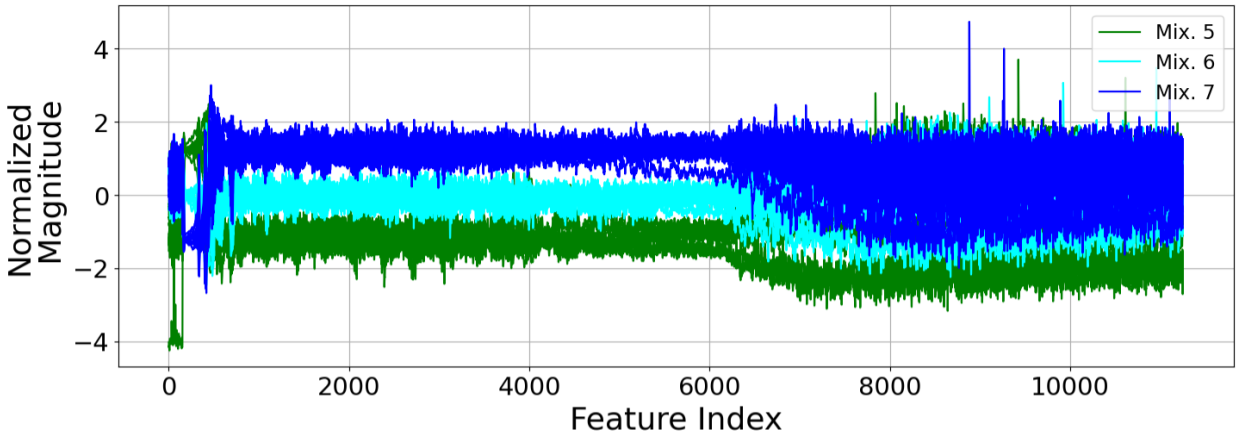


Figure 4.22: Experiment 8 Thresholded and Normalized

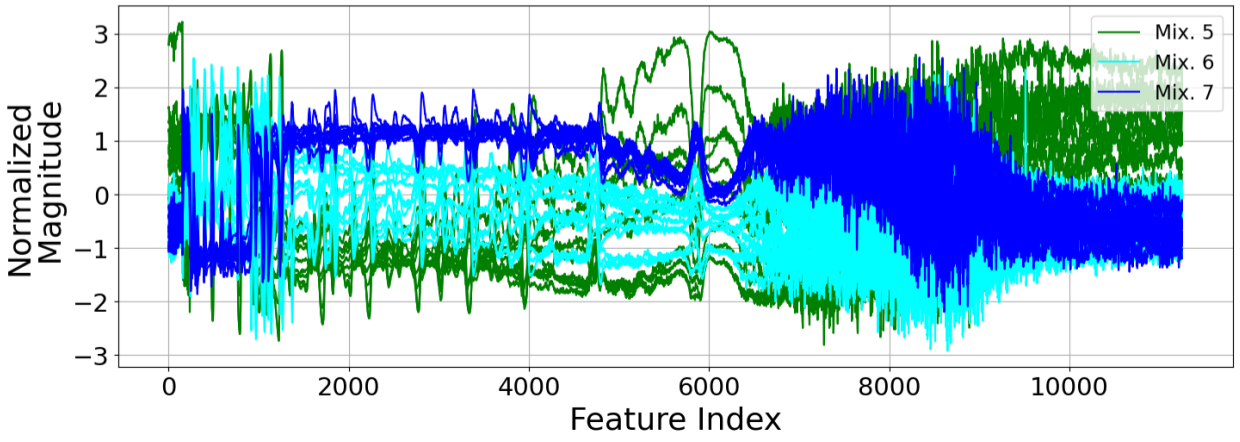


Figure 4.23: Experiment 9 Thresholded and Normalized

Preliminary testing with Experiment 8 was good, due to the good clustering with the data. However, the Experiment 9 data performed very poorly with the same tests. Therefore, similar to Experiments 1-5, an analysis was performed to determine the best features to use.

A window sweep of the features was performed with the KNN algorithm. The two parameters available for the KNN algorithm, by itself, are the number of points used for the nearest neighbor calculations and the weighting system used for voting. A distance based weighting was used.

There were several hyperparameters swept for tuning the algorithm. The first parameter was the number of points used for the nearest neighbor algorithm. For the second and third parameter, a window sweep of the input data points was used to find the best features for the model. The second parameter is the size of the window, while the third parameter is the location of the window. The index of the last feature within the window was used as the location. This was tested both with and without PCA. When it was tested with PCA, the number of components was varied, with the components having the most variance being used each time.

There were multiple parameter sets achieving 100% accuracy on the validation set. Out of the models achieving perfect scores on the validation set, the final window for feature selection was selected based on performance on the testing set.

Table 4.1: Experiment 8 Optimal Parameters

Parameter	Values
k	1 - 7
PCA	1 - 7
Win Size	703 - 7030
Win Loc	1406 - 7574

Based on this data, the most useful features for Experiment 8 are between index 544 and 7574, and the most useful features for Experiment 9 are between index 340 and 1043. This corresponds mostly with the data not being filtered out by the variance threshold, however that alone does not explain why the features between 7574 and 11249 and 1043 and 11249 result in worse performance for Experiments 8 & 9 respectively.

Table 4.2: Experiment 9 Optimal Parameters

Parameter	Values
k	1 - 7
PCA	1 - 7
Win Size	703
Win Loc	1043

To explain this, it is necessary to reexamine the overall model used and apply knowledge of the domain. When a chromatogramer separates a mixture, the components elute into the mobile phase and exit the column. After the majority of the mixture has exited the column, the amount detected by the sensor exponentially decreases, but will still have some residual amounts that are not useful for the chromatograms and are essentially just noise. For the model, a variance threshold filters out parameters with low variance, after which the data is scaled to a zero mean and variance of one. For features with extremely low variance, they have no effect. However, for features with a variance just large enough to exceed the threshold, but holding no useful data from the chromatogramer, they get scaled to have the same variance as all the other features.

The variance threshold was useful for eliminating features with a low variance. Aside from reducing model complexity and computation time, this filtering was also important for accuracy. Because the features are scaled to have a mean of 0 and a variance of 1, features with extremely low variances would be scaled to large magnitudes, and would have a much higher impact on the model than they should.

4.5 Conclusion

In this chapter, we analyzed the data used in this thesis using PCA and feature selection. In the next chapter we compare and analyze the models.

Chapter 5

Model Comparison and Analysis

In this chapter, we discuss the preprocessing steps used with the models and the hyperparameters tuned for each model.

5.1 Training and Preprocessing

For the following models, the data is preprocessed with a variance threshold, before being normalized to zero mean and unit variance. The models are tested both with and without feature selection, and with and without PCA (Figure 5.1).

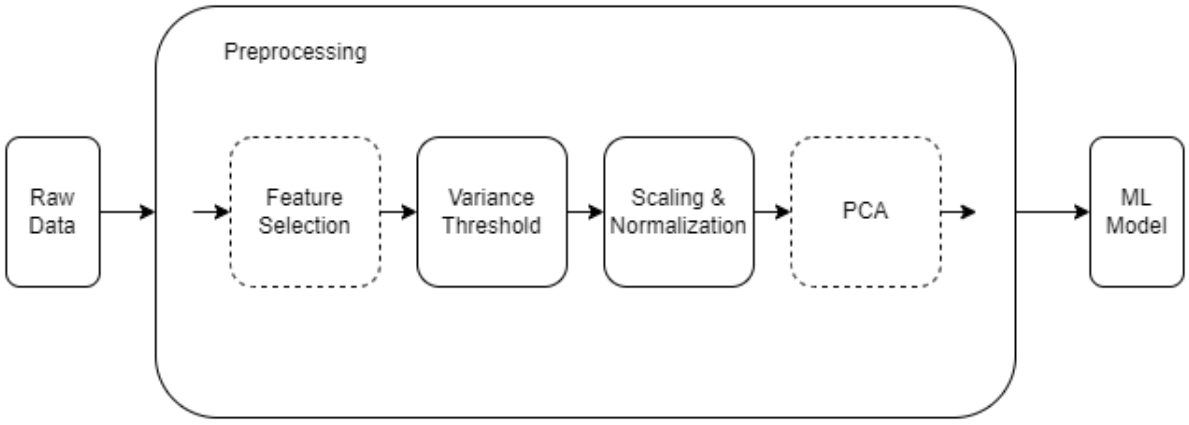


Figure 5.1: Training and Preprocessing Steps

5.2 ML Models

In this section we address the parameters tuned for each model, individually.

5.2.1 KNN

The parameters varied for the KNN algorithm were the amount of points used for the calculation and the weighting system used during voting. Distance based weighting, where the weight of a vote is inversely proportional to the distance, was chosen.

Hyperparameter tuning consisted of tuning the following parameters: one, the k value for the nearest neighbor algorithm; two, the size of a sliding window used for selecting the input dataset; and three, the location of said window. The k value was limited to testing values of 1 to 7 based on limitations in the amount of samples per class.

Along with the hyperparameter tuning was a comparison of the performance of the model with and without PCA. For the analysis with PCA, the performance of the model was compared with different amounts of PCA components, with the components with the highest variance being used each time.

From the data, the model can achieve 100% accuracy with any value of window size and window location. However, the value of k affects which window sizes and locations should be used. Small values of k achieve perfect accuracy for any size and location. However, k values of 4 and above result in less accuracy when using a size and location above 2000. This is most likely due to limitations in the training set, since for values of k larger than the number of samples per class, the nearest points will include points of the wrong class, throwing off the vote.

5.2.2 SVM

For SVM models, the model was initially evaluated with a radial basis function, while sweeping gamma and C. Results were poor, so the search was expanded to other kernel functions, including linear, polynomial, and sigmoid. The C parameter, which trades the accuracy of the model for simplicity, was swept for all kernel functions. For rbf, an additional parameter, gamma, was swept.

5.2.3 CNN

While CNNs are often used for image data, they can also be used for time-series data. A one-dimensional convolution was used in the convolution layer rather than two-dimensional. The general process for developing a CNN is mostly a matter of trial and error [34]. In order to avoid overfitting, a simple model with as few layers as possible was used to start with. To increase accuracy, the complexity was gradually increased by adding layers, until the model started to overfit. Some additional steps were taken to address overfitting by adding dropout layers, which randomly set inputs to 0. The output layer uses the softmax activation function.

Because of how the software for the ML networks works, inputs must be at least two-dimensional, even for 1-dimensional data. This is to allow for multiple vectors to be fed into the model at one time. The input to the model is of shape $(k, 1)$ where k is the length of the chromatogram, and the output is of size $(1, c)$, where c is the number of categories.

Furthermore, the layers are scaled based on the size of the input. Literature surveys show that a gradual reduction in size, rather than immediate decrease, generally results in better performance[34].

When using PCA, the same layer types can be used, and in the same order, however their size must be reduced to take into account the different input size. Testing showed that the CNN with PCA performed significantly worse than without PCA.

5.2.4 Decision Tree

Decision trees were trained and evaluated for the data sets using the methodology in Chapter 3. A hyperparameter sweep was performed where the effect of varying the max depth and the number of PCA components was evaluated.

5.2.5 Random Forest

Using the methods described in Chapter 3, random forests were tested. Similar to decision trees, the max depth of the trees and the number of PCA components used was varied. Unlike with decision trees, the maximum amount of features used for finding the best split was also varied. Since random forests consist of multiple decision trees, they take longer to train and use more resources. However, by training off of random subsets of the sample data, they mitigate overfitting.

5.3 Conclusion

In this chapter, we described the preprocessing steps used with the models, as well as the hyperparameters adjusted for training the models. In the next chapter, we provide the evaluation scores for models after tuning and discuss the results.

Chapter 6

Results

In the previous chapters, we discussed the research methodology used by this thesis. We analyzed the data, and then applied machine learning algorithms to them.

In this chapter, the evaluation scores of the algorithms are collated and analysed. Evaluation was done using cross-validation as discussed in Chapter 3. The scores both with and without PCA, and with and without feature selection are given and compared.

6.1 KNN and KNN Parameter Sweeps

The results of the window sweeps of the data sets with the KNN model are shown in Table 6.1. The Optimal Window column shows the range of feature indexes that result in the best score. For Experiments 1-7; feature selection was unable to improve the score. The optimal windows include all of the features for those cases.

For Experiment 8 and Experiment 9, however, feature selection did have a significant impact. For the results in the following sections, scores with feature selection used the windows from this table. Since the optimal windows include the entire data set for Experiments 1-7, feature selection makes no difference, so there is only one set of scores.

Table 6.1: Optimal Feature Selection Windows

Experiment	Optimal Window (Indexes)	Highest Accuracy with Feature Selection	Highest Accuracy without Feature Selection
1	0-4498	1.0	1.0
2	0-4498	1.0	1.0
3	0-4498	1.0	1.0
4	0-4498	1.0	1.0
5	0-4498	1.0	1.0
6	0-9123	1.0	1.0
7	0-4498	1.0	1.0
8	544-7574	0.97	0.58
9	340-1043	1.0	0.67

6.2 SVM

The results of SVM training are shown in Tables 6.2, 6.4, and 6.5. The table shows the best scores after hyperparameter tuning, while differentiating between models that used PCA and models that didn't. For example, for the Experiment 4 data set with the RBF kernel, the highest score without PCA is 0.75, and the highest score with PCA is 1.0. For those that did use PCA, the models were evaluated using a variable amount of PCA components, as discussed in Ch 5.

The best kernel overall was linear, which performed as well or better than any other kernel in every case, except for Experiment 5 with PCA.

Table 6.2: SVM Scores (Experiments 1-5)

Data Set	Accuracy (no-PCA/PCA)			
Kernel	Linear	RBF	Polynomial	Sigmoid
Experiment 1	1.0 / 1.0	0.75 / 0.75	1.0 / 1.0	0.0 / 0.0
Experiment 2	1.0 / 1.0	1.0 / 1.0	1.0 / 1.0	0.25 / 0.25
Experiment 3	1.0 / 1.0	1.0 / 1.0	0.75 / 1.0	0.75 / 0.75
Experiment 4	1.0 / 1.0	0.75 / 1.0	0.5 / 0.75	0.25 / 0.5
Experiment 5	0.75 / 0.75	1.0 / 1.0	0.75 / 1.0	0.25 / 0.25

Table 6.3: SVM Scores (Experiments 6-7)

Data Set	Accuracy (no-PCA/PCA)			
Kernel	Linear	RBF	Polynomial	Sigmoid
Experiment 6	1.0 / 1.0	1.0 / 1.0	0.75 / 1.0	0.75 / 0.75
Experiment 7	1.0 / 1.0	1.0 / 1.0	0.75 / 0.75	0.25 / 0.5

Table 6.4: SVM Scores (Experiments 8-9) With Feature Selection

Data Set	Accuracy (no-PCA/PCA) With Feature Selection			
Kernel	Linear	RBF	Polynomial	Sigmoid
Experiment 8	0.97 / 0.97	0.7 / 0.88	0.91 / 0.91	0.66 / 0.94
Experiment 9	1.0 / 1.0	0.67 / 1.0	0.91 / 0.97	0.70 / 0.70

Table 6.5: SVM Scores (Experiments 8-9) Without Feature Selection

Data Set	Accuracy (no-PCA/PCA) Without Feature Selection			
Kernel	Linear	RBF	Polynomial	Sigmoid
Experiment 8	0.94 / 0.97	0.67/ 0.94	0.67 / 0.67	0.09 / 0.33
Experiment 9	0.94 / 0.91	0.88/ 0.82	0.70 / 0.67	0.33 / 0.52

6.3 CNN

For CNN, an architecture was derived which is applicable to all data sets. The layer sizes are scaled based on the size of the input, as discussed in section 5.2.3. The final architecture is as follows.

1. A 1-dimensional convolution layer with a kernel size of 5.
2. A max pooling layer for downsampling the output of the convolution layer.
3. A second convolution layer.

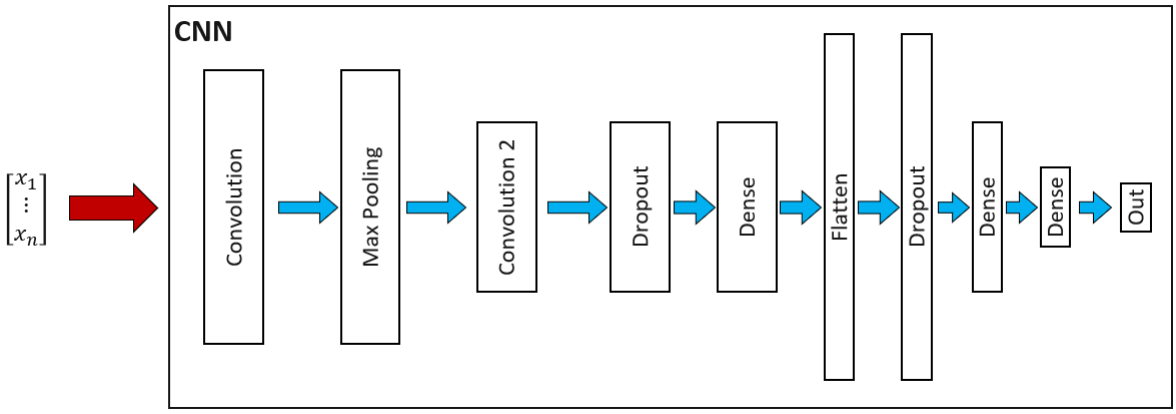


Figure 6.1: CNN Architecture

4. A dropout layer for reducing overfitting.

5. A dense layer for reducing the size.

6. A flattening layer for changing the layer shape to a 1-dimensional vector.

7. An additional dropout layer for reducing overfitting.

8. A dense layer for reducing the size.

9. A final dense layer for reducing the size to the number of categories.

10. The output layer, which uses the softmax activation function to normalize the output.

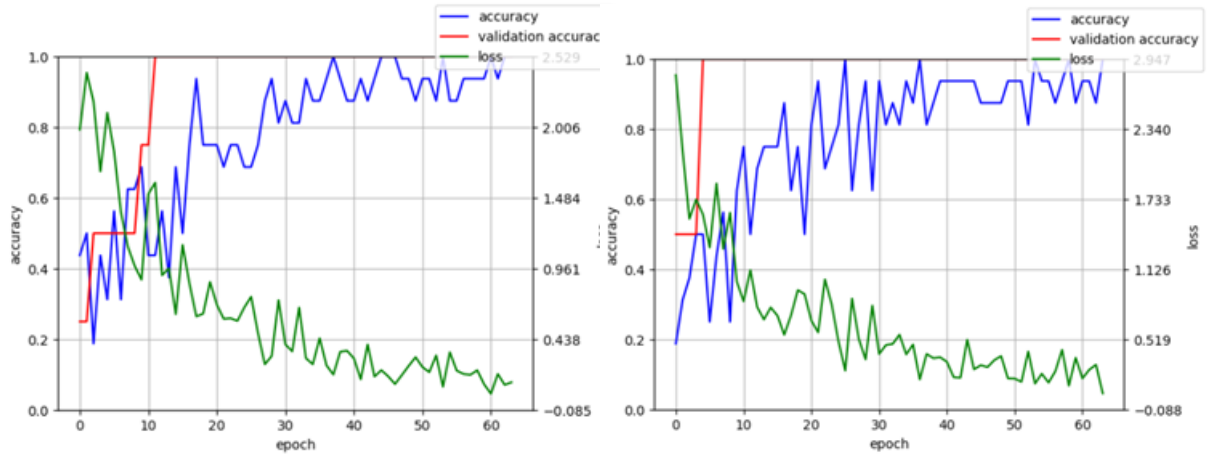
The results of testing Experiments 1 - 5 are shown in figures 6.2 - 6.6. Each epoch represents a single iteration of training the model. The loss, training score, and validation score are shown.

Because parts of the CNN model incorporate randomness to prevent overfitting, the results are not deterministic. The same model, when trained on the same data, can still result in different outcomes.

For Experiments 1-5, there were some small variations, but the results were fairly consistent with the examples shown. The model was able to achieve a high accuracy within the first 30 epochs. Because the testing set was significantly smaller than the training set, the validation score reaches a ceiling faster than the training score. In order to have a model with a real world accuracy matched by the training accuracy, one should choose the number of training epochs such that the training score matches the validation score. This consistently occurs after approximately 60 training epochs.

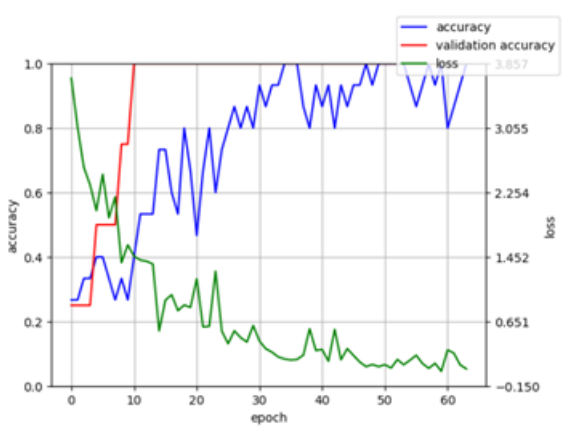
Experiments 6 and 7, without PCA, performed very similar to Experiments 1-5. With PCA, however, performance fluctuated too much to be able to assign a score.

For Experiments 8 and 9, training and validation were done similar to 1-7, however final evaluation was done with the blind sets taken for those experiments. The results were much less consistent. Experiment 8's model fluctuated between 0.33 - 0.67 final accuracy, whereas Experiment 9 fluctuated between 0.33 and 0.5. This is likely due to overfitting of the model.

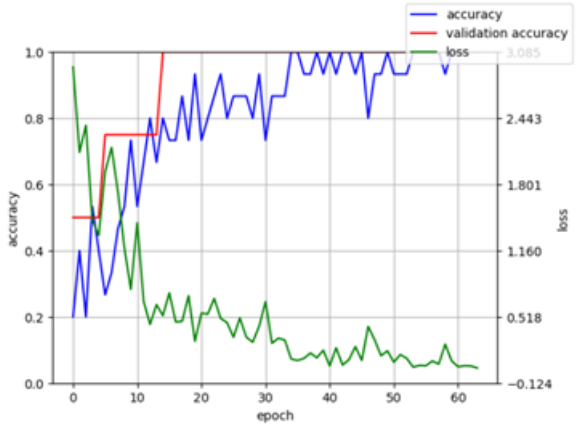


(a) Experiment 1 (b) Experiment 1 with PCA

Figure 6.2: Experiment 1 CNN Results

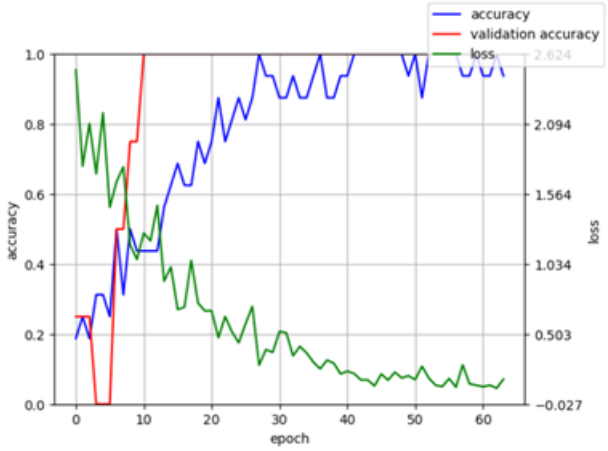


(a) Experiment 2

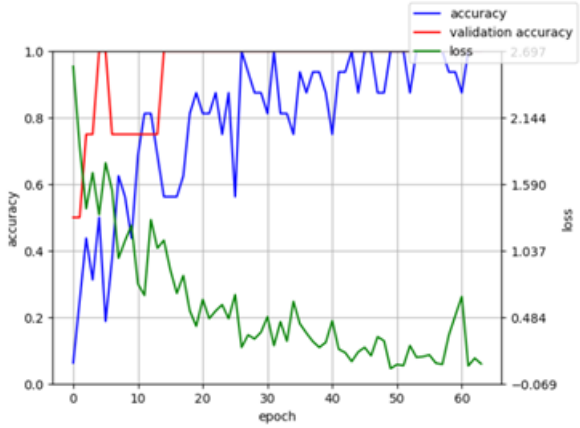


(b) Experiment 2 with PCA

Figure 6.3: Experiment 2 CNN Results

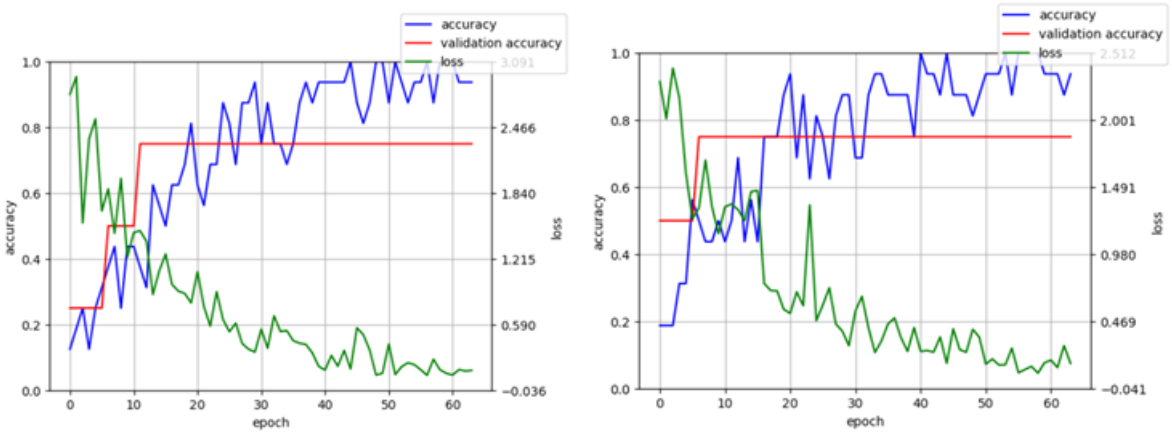


(a) Experiment 3



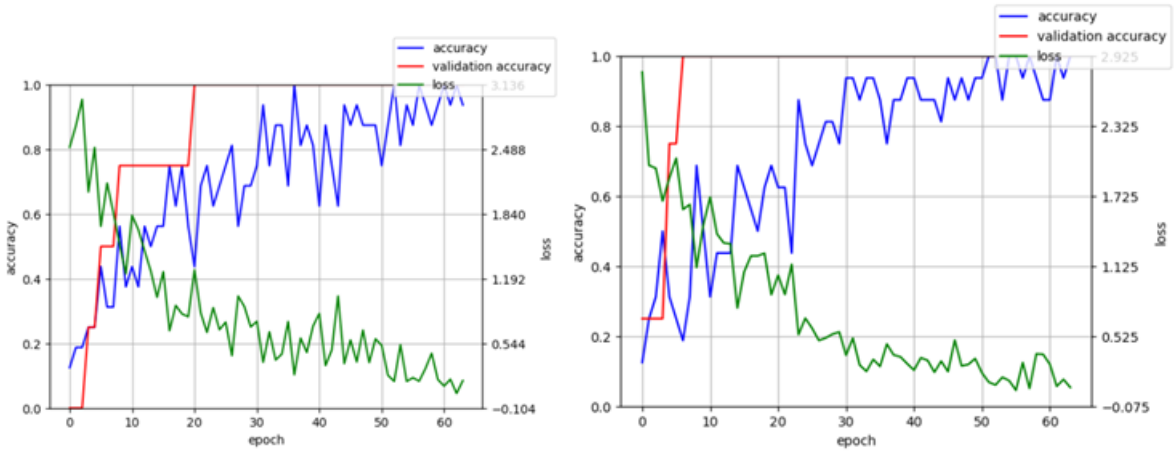
(b) Experiment 3 with PCA

Figure 6.4: Experiment 3 CNN Results



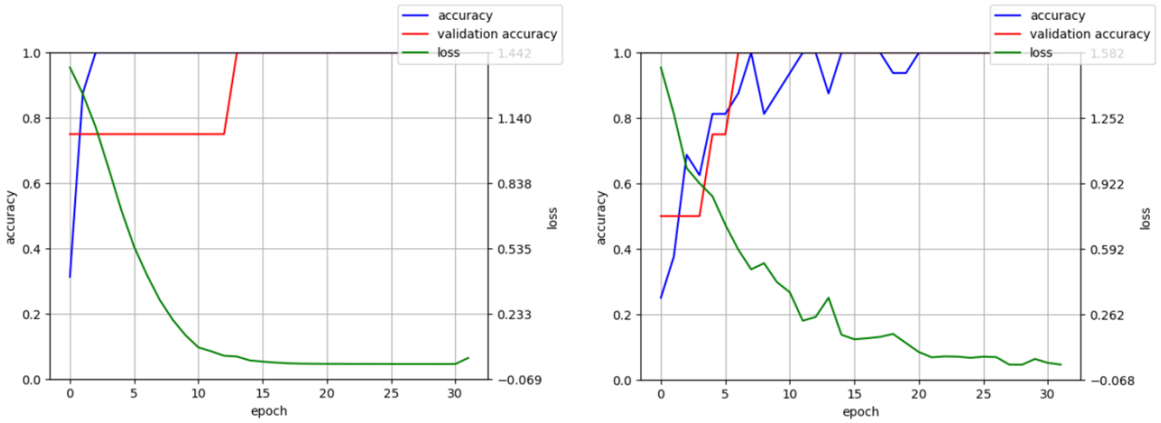
(a) Experiment 4 (b) Experiment 4 with PCA

Figure 6.5: Experiment 4 CNN Results



(a) Experiment 5 (b) Experiment 5 with PCA

Figure 6.6: Experiment 5 CNN Results



(a) Experiment 6 (b) Experiment 7

Figure 6.7: Experiment 6-7 CNN Results (No PCA)

6.4 DT

For DTs, the parameter for the max depth of the tree was swept. Models were tested with and without PCA, and with or without feature selection.

PCA had no overall advantage, with Experiment 5 having better accuracy with PCA, but Experiment 4 having worse accuracy.

Feature selection was very beneficial. While it had no noticeable effect on Experiment 8, it significantly improved the model accuracy with Experiment 9.

Table 6.6: DT Scores (Experiments 1-5)

Data Set	Accuracy	
	No PCA	PCA
Experiment 1	0.75	0.75
Experiment 2	1.0	1.0
Experiment 3	1.0	1.0
Experiment 4	0.5	0.25
Experiment 5	0.5	0.75

Table 6.7: DT Scores (Experiments 6-7)

Data Set	Accuracy	
	No PCA	PCA
Experiment 6	0.5	0.5
Experiment 7	0.5	0.75

Table 6.8: DT Scores (Experiments 8-9)

Data Set	Accuracy With Feature Selection		Accuracy Without Feature Selection	
	No PCA	PCA	No PCA	PCA
Experiment 8	1.0	1.0	1.0	1.0
Experiment 9	1.0	1.0	0.67	0.67

6.5 RF

For RF, the max depth of the trees and the maximum number of features considered were swept. Models were tested with and without PCA, and with or without feature selection.

Overall the results were somewhat similar to the results for DT. Feature selection was very helpful for this model as well. However, unlike with DT, PCA consistently performed better or as well, with the only exception being Experiment 7.

Table 6.9: RF Scores (Experiments 1-5)

Data Set	Accuracy	
	No PCA	PCA
Experiment 1	0.75	1.0
Experiment 2	1.0	1.0
Experiment 3	1.0	1.0
Experiment 4	1.0	1.0
Experiment 5	0.75	1.0

Table 6.10: RF Scores (Experiments 6-7)

Data Set	Accuracy	
	No PCA	PCA
Experiment 6	0.75	0.75
Experiment 7	0.75	0.5

Table 6.11: RF Scores (Experiments 8-9)

Data Set	Accuracy With Feature Selection		Accuracy Without Feature Selection	
	No PCA	PCA	No PCA	PCA
Experiment 8	1.0	1.0	1.0	1.0
Experiment 9	1.0	1.0	0.67	0.67

6.6 Discussion

Overall, every algorithm was able to achieve a perfect score on some experiments, but no algorithm was able to achieve a perfect score on all experiments. KNN came close to a perfect score, with a perfect score for every Experiment except 8.

Most algorithms performed well on Experiments 1-5 and 8 and struggled on Experiment 9. Figures 6.8 and 6.9 show the main dataset along with the blind sets. Examination of the blind set for Experiment 9 showed showed a slight shift of the chromatograms from the training set, most likely because the training set and blind set were taken on different days. Figure 6.10 illustrates this clearly, with the test data for a 95% mixture overlapping with the training data for a 90% mixture. While this made it harder for algorithms to perform well with this data set, this also makes results from this data set more valuable. Algorithms that are able to perform well on this data set, despite the difficulties, will be more tolerant to variations in testing conditions.

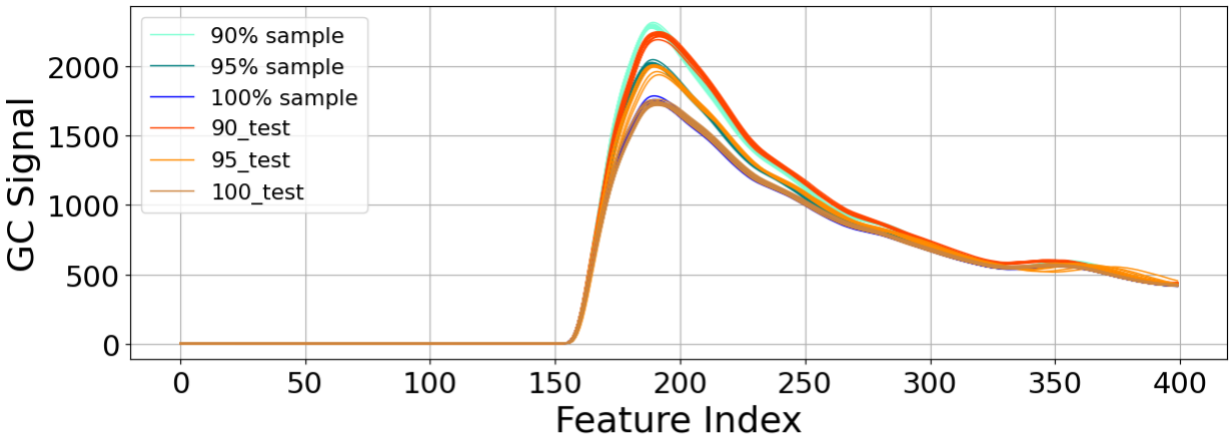


Figure 6.8: Experiment 8 Train and Blind Set

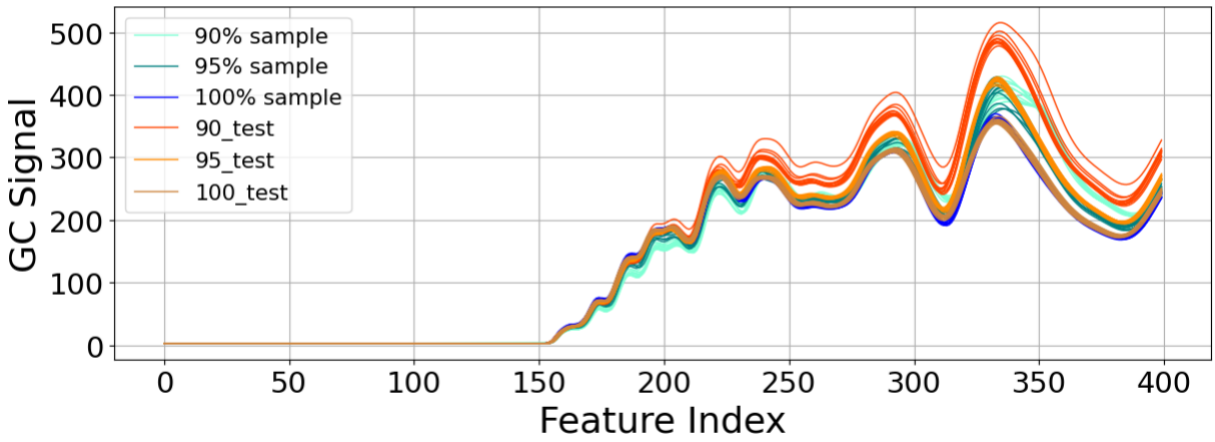


Figure 6.9: Experiment 9 Train and Blind Set

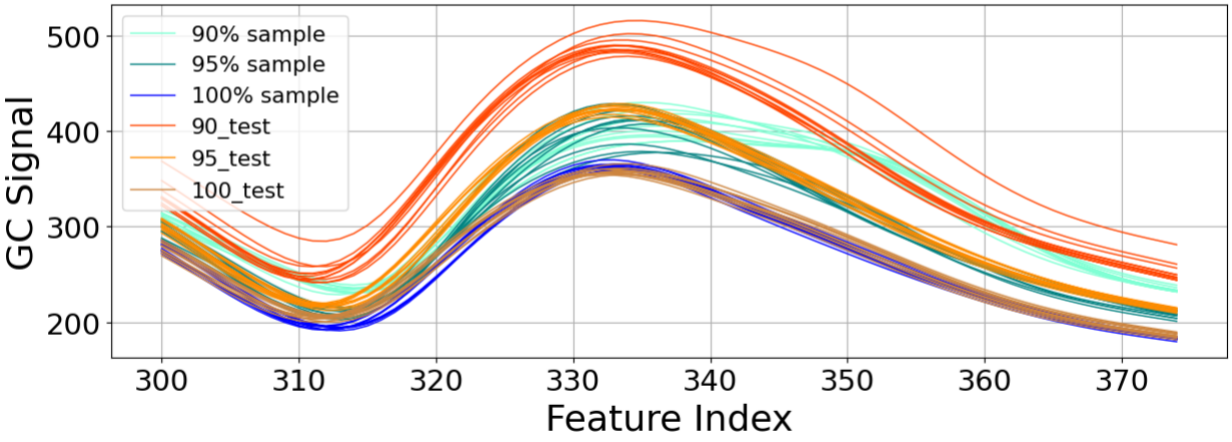


Figure 6.10: Experiment 9 Train and Blind Set with Data Shifting

The performance of Experiments 6 and 7 were comparable to the performance for Experiments 1-5. This is as expected, since they consisted of the same mixtures. KNN and SVM performed well. The CNN without PCA performed well, but with PCA performed much worse. RF consistently performed better than DT.

For examining the effect of data fusion, we compare the results of Experiment 6 (the data fusion set) to Experiment 7 (the control set consisting of only the front sensor chromatograms). The performance was similar for most algorithms except SVM. For SVM, Experiment 6 performed much better for the polynomial and signmoid kernels.

Some preprocessing steps proved valuable. The variance threshold discussed in Ch. 5 helped to reduce data set size, which in theory should result in speedier model training. However, very few features were filtered out, so the impact was minimal. PCA was beneficial in some cases, and ineffective in others. Feature selection proved to be the most effective preprocessing step. In the majority of cases it resulted in significant performance improvements, especially with Experiment 9.

In terms of comparing the algorithms from a user's perspective, we define two metrics.

- **setup difficulty:** how hard it is to implement an instance of an algorithm (i.e. how hard it is to set up a model in software such that the user can start tuning the model)
- **tuning difficulty:** how difficult or expensive tuning is

Setup difficulty takes into account that software libraries already exist for implementing these algorithms. For implementing cross-validation, KNN, SVM, DT, and RF, the Scikit-Learn software library for Python was used [35]. Appendix A shows some of the Python analysis code, to demonstrate software implementation. For analysing CNNs, the Keras software library was used [36]. Table 6.12 summarizes the hyperparameters for each model.

The setup difficulty and tuning difficulty were low for KNN. The model was very easy to get up and running with the software, and there were only two hyperparameters. For the purpose of comparing the KNN algorithm to the others, the window sweeps were not considered hyperparameters. For SVM, the setup difficulty was also low, however the tuning difficulty was classified as medium. Using the library was as easy as for KNN, however tuning was more difficult. There were more hyperparameters, two of which required a logarithmic search of an exponentially large design space. This caused training and tuning to take considerably longer.

For CNN, the setup difficulty and tuning difficulty were classified as high. When compared to the other algorithms, this algorithm was by far the most difficult to setup and the most time-consuming to train and tune. The layers of the CNN had to be individually specified for setup, and had to be adjusted as part of the tuning process. The tuning process itself was far more time-consuming than the other algorithms.

For DT and RF, the setup difficulty was low, as setting them up with the Scikit-Learn library was as easy as for KNN and SVM. For DT, tuning difficulty was low, since there were few parameters and the tuning parameter process took an amount of time comparable to KNN. However, for RF, tuning difficulty was classified as medium, since there were more hyperparameters, and the tuning process took considerably longer, although still not nearly as much as for CNN.

Table 6.12: Algorithm Hyperparameters

Algorithm	Hyperparameters
KNN	k, PCA
SVM	kernel, gamma, C, PCA
CNN	layers, PCA
DT	max depth, PCA
RF	max depth, max features, PCA

Ultimately, the KNN algorithm performed best overall. It was the only algorithm able to achieve >95% accuracy on all data sets. However, it was only able to achieve this performance through the use of feature selection and PCA.

6.7 Limitations

One factor that majorly impacted the findings was the data quantity. For the Experiment 1-7 data sets, there were only 5 samples per mixture. Since the train-test split was 80/20, this only left one sample per mixture in the testing set used for final evaluation. With such a small testing set, the test score had a large variation, and may be less reflective of actual performance.

For this reason, more weight is given to the Experiment 8 and 9 data sets. Since those data sets had blind testing sets along with the training data, testing was able to be much more thorough.

For future applications, one limiting factor is testing conditions. In the case of the Experiment 9 data set, the blind set was taken on a different day from the training set. This resulted in shifting them a little bit from the training data, making the testing step much more difficult for the algorithms . However, algorithms that were able to perform well, despite this difficulty, will have more tolerance to noise. Thus, although most of the algorithms performed significantly worse on the Experiment 9 data set, the scores are more valuable for determining an algorithm's effectiveness.

Chapter 7

Conclusion

In this thesis, machine learning models were applied to gas chromatography data. Multiple machine learning algorithms were investigated, inluding K-Nearest Neighbor (KNN), Support Vector Machine (SVM), Convolutional Neural Network (CNN), Decision Tree (DT), and Random Forest (RF). Pre-processing steps were tested and applied, including feature selection and feature engineering. The effect of data fusion was tested. The performance of the algorithms was evaluated using k-fold cross-validation to create more relevant evaluations. And finally, the algorithms were compared against each other in terms of performance, and usefulness for application to additional chromatography data sets. All models performed well with some data sets, however it was ultimately found that KNN was able to perform consistently well for all data sets, if feature selection was used, followed by RF.

7.1 Future Work

Although this work examined the most common algorithms, it could be expanded in several ways.

- Gathering more data sets from additional mixture types would provide a more complete evaluation of the algorithms.
- Synthesizing noise in the chromatography systems and evaluating the noisy chro-

matograms would better test the algorithms' noise tolerance. While Experiment 9 did test this, more data is needed to conclusively prove effectiveness against noise.

- Window sweeps with the KNN algorithm were used for feature selection for this work. Evaluating the windows with other ML models could potentially find better feature sets. Additionally, other feature selection algorithms could prove more beneficial.
- Other dimensionality reduction techniques could be tested.
- Additional feature engineering techniques could be tested. Experiment 6 used a data set consisting of data fusion of chromatographs with peak integration data. This resulted in some accuracy improvement. Tailoring a peak detection model as part of feature engineering for a machine learning algorithm could potentially improve performance.

Bibliography

[1] M. M. Rahman, A. Abd El-Aty, J.-H. Choi, H.-C. Shin, S. C. Shin, and J.-H. Shim, *Basic Overview on Gas Chromatography Columns*. John Wiley & Sons, Ltd, 2015, ch. 3, pp. 823–834. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9783527678129.assep024

[2] T. Wenzel. [Online]. Available: https://chem.libretexts.org/Bookshelves/Analytical_Chemistry/Supplemental_Modules_(Analytical_Chemistry)/Analytical_Sciences_Digital_Library/In_Class_Activities/Separation_Science/2%3A_Chromatography_%E2%80%93_Background/01_Introduction

[3] W.-L. Chao, "Machine learning tutorial," *Digital Image and Signal Processing*, 2011.

[4] T. M. Mitchell, "Machine learning," 1997.

[5] M. Beccaria, T. R. Mellors, J. S. Petion, C. A. Rees, M. Nasir, H. K. Systrom, J. W. Sairistil, M.-A. Jean-Juste, V. Rivera, K. Lavoile *et al.*, "Preliminary investigation of human exhaled breath for tuberculosis diagnosis by multidimensional gas chromatography–time of flight mass spectrometry and machine learning," *Journal of Chromatography B*, vol. 1074, pp. 46–50, 2018.

[6] L. Lebanov, L. Tedone, A. Ghiasvand, and B. Paull, "Random forests machine learning applied to gas chromatography – mass spectrometry derived average mass spectrum data sets for classification and characterisation of essential oils," *Talanta*, vol. 208, p. 120471, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S003991401931104X

[7] S. Gu, W. Chen, Z. Wang, J. Wang, and Y. Huo, "Rapid detection of aspergillus spp. infection levels on milled rice by headspace-gas chromatography ion-mobility spectrometry (hs-gc-ims) and e-nose," *LWT*, vol. 132, p. 109758, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0023643820307477

[8] N. Gerhardt, M. Birkenmeier, S. Schwolow, S. Rohn, and P. Weller, "Volatile-compound fingerprinting by headspace-gas-chromatography ion-mobility spectrometry (hs-gc-ims) as a benchtop alternative to 1h nmr profiling for assessment of the authenticity of honey," *Analytical Chemistry*, vol. 90, no. 3, pp. 1777–1785, 2018, pMID: 29298045. [Online]. Available: https://doi.org/10.1021/acs.analchem.7b03748

[9] X. Ju, F. Lian, H. Ge, Y. Jiang, Y. Zhang, and D. Xu, "Identification of rice varieties and adulteration using gas chromatography-ion mobility spectrometry," *IEEE Access*, vol. 9, pp. 18 222–18 234, 2021.

[10] N. I. Hadi, Q. Jamal, A. Iqbal, F. Shaikh, S. Somroo, and S. G. Musharraf, "Serum metabolomic profiles for breast cancer diagnosis, grading and staging by gas chromatography-mass spectrometry," *Scientific reports*, vol. 7, no. 1, p. 1715, 2017.

[11] Y. Sakumura, Y. Koyama, H. Tokutake, T. Hida, K. Sato, T. Itoh, T. Akamatsu, and W. Shin, "Diagnosis by volatile organic compounds in exhaled breath from lung cancer patients using support vector machine algorithm," *Sensors*, vol. 17, no. 2, 2017. [Online]. Available: https://www.mdpi.com/1424-8220/17/2/287

[12] G.-H. Fu, B.-Y. Zhang, H.-D. Kou, and L.-Z. Yi, "Stable biomarker screening and classification by subsampling-based sparse regularization coupled with support vector machines in metabolomics," *Chemometrics and Intelligent Laboratory Systems*, vol. 160, pp. 22–31, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016974391630466X

[13] R. Houhou and T. Bocklitz, "Trends in artificial intelligence, machine learning, and chemometrics applied to chemical data," *Analytical Science Advances*, vol. 2, no. 3-4, pp. 128–141, 2021. [Online]. Available: https://chemistry-europe.onlinelibrary.wiley.com/doi/abs/10.1002/ansa.202000162

[14] A. Skarysz, Y. Alkhalifah, K. Darnley, M. Eddleston, Y. Hu, D. B. McLaren, W. H. Nailon, D. Salman, M. Sykora, C. L. P. Thomas, and A. Soltoggio, "Convolutional neural networks for automated targeted analysis of raw gas chromatography-mass spectrometry data," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8.

[15] K. Bi, D. Zhang, T. Qiu, and Y. Huang, "Gc-ms fingerprints profiling using machine learning models for food flavor prediction," *Processes*, vol. 8, no. 1, 2020. [Online]. Available: https://www.mdpi.com/2227-9717/8/1/23

[16] N. L. da Costa, M. S. da Costa, and R. Barbosa, "A review on the application of chemometrics and machine learning algorithms to evaluate beer authentication," *Food Analytical Methods*, vol. 14, no. 1, p. 136–155, 2020.

[17] S. Guo, J. Popp, and T. Bocklitz, "Chemometric analysis in raman spectroscopy from experimental design to machine learning–based modeling," *Nature protocols*, vol. 16, no. 12, pp. 5426–5459, 2021.

[18] Y. Zou, M. Gaida, F. A. Franchina, P.-H. Stefanuto, and J.-F. Focant, "Distinguishing between decaffeinated and regular coffee by hs-spme-gc× gc-tofms, chemometrics, and machine learning," *Molecules*, vol. 27, no. 6, p. 1806, 2022.

[19] M. M. Tomazzoli, R. D. Pai Neto, R. Moresco, L. Westphal, A. R. Zeggio, L. Specht, C. Costa, M. Rocha, and M. Maraschin, "Discrimination of brazilian propolis according

to the seasoning using chemometrics and machine learning based on uv-vis scanning data," *Journal of integrative bioinformatics*, vol. 12, no. 4, pp. 15–26, 2015.

[20] P. B. Joshi, "Navigating with chemometrics and machine learning in chemistry," *Artificial Intelligence Review*, pp. 1–26, 2023.

[21] P. Cunningham and S. J. Delany, "K-nearest neighbour classifiers - a tutorial," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021. [Online]. Available: https://doi.org/10.1145/3459665

[22] L. Nguyen, "Tutorial on support vector machine," *Applied and Computational Mathematics*, vol. 6, p. 1–15, Jun 2016.

[23] [Online]. Available: https://scikit-learn.org/stable/modules/cross_validation.html

[24] [Online]. Available: https://scikit-learn.org/stable/modules/tree.html#tree

[25] J. Gall, N. Razavi, and L. Van Gool, "An introduction to random forests for multi-class object detection," in *Outdoor and Large-Scale Real-World Scene Analysis*, F. Dellaert, J.-M. Frahm, M. Pollefeys, L. Leal-Taixé, and B. Rosenhahn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 243–263.

[26] J. Ali, R. Khan, N. Ahmad, and I. Maqsood, "Random forests and decision trees," *International Journal of Computer Science Issues*, vol. 9, no. 5, Sep 2012.

[27] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, p. e00938, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405844018332067

[28] Q. V. Le *et al.*, "A tutorial on deep learning part 2: Autoencoders, convolutional neural networks and recurrent neural networks," *Google Brain*, vol. 20, pp. 1–20, 2015.

[29] S. H. Huang, "Supervised feature selection: A tutorial." *Artif. Intell. Res.*, vol. 4, no. 2, pp. 22–37, 2015.

[30] A. Zheng and A. Casari, *Feature engineering for machine learning: principles and techniques for data scientists.* " O'Reilly Media, Inc.", 2018.

[31] R. Bro and A. K. Smilde, "Principal component analysis," *Analytical methods*, vol. 6, no. 9, pp. 2812–2831, 2014.

[32] F. Castanedo *et al.*, "A review of data fusion techniques," *The scientific world journal*, vol. 2013, 2013.

[33] M. Chowdhury, A. Gholizadeh, and M. Agah, "Rapid detection of fuel adulteration using microfabricated gas chromatography," *Fuel*, vol. 286, p. 119387, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0016236120323838

[34] B. Debus, H. Parastar, P. Harrington, and D. Kirsanov, "Deep learning in analytical chemistry," *TrAC Trends in Analytical Chemistry*, vol. 145, p. 116459, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016599362100282X

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[36] F. Chollet *et al.* (2015) Keras. [Online]. Available: https://github.com/fchollet/keras

[37] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy,

W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

Appendices

Appendix A

Analysis Code

This appendix contains some of the code used for performing the analysis. The purpose is to show how the window sweep is performed (Section A.1) and how the hyperparameter tuning was done (Section A.2).

A.1 KNN Evaluation and Window Sweep

This is the Python code written for performing feature selection of data using the KNN algorithm. Aside from the standard Python libraries, this code also uses the Scikit-Learn code library [35] and Numpy library [37]. Custom code was made for loading raw data from files, via the custom_dataset, fuel_dataset, and fox_sig_int modules. Please note, some of the code makes use of absolute paths. Those sections of code have been altered to remove sensitive information.

```
# Lib Imports
import multiprocessing as mp
from sklearn.covariance import EllipticEnvelope
import numpy as np
import sys
import importlib
```

```
# import matplotlib
# import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import sklearn.feature_selection as feature_selection
from sklearn.neighbors import KNeighborsClassifier
from sklearn.utils import shuffle
from sklearn.model_selection import StratifiedKFold
from sklearn.pipeline import Pipeline
sys.path.insert(0, "absolute\\path\\to\\python_files")
from scipy.signal import savgol_filter
import datetime
#save history
import pickle
# import hickle
# Profiling
import cProfile
from pstats import SortKey, Stats
import io
import pandas as pd
import io

# Local Imports
```

```
import custom_dataset
import fuel_dataset
import pca as custom_pca
import outliers as custom_outliers
import fox_sig_int


def run_sweep_iteration(pipe, x_train, y_train, x_val, y_val, x_test,
↪  y_test) -> list:
    """This runs a single iteration within the sweep of a loop,
    ↪  collects the data,
       and returns the data
    """
    # profile the training for comparing runtime/memory data
    # prof = cProfile.Profile()
    # prof.enable()

    pipe.fit(x_train, y_train)
    val_score = pipe.score(x_val, y_val)
    val_predictions = pipe.predict(x_val)
    test_score = pipe.score(x_test, y_test)
    test_predictions = pipe.predict(x_test)

    # prof.disable()
    # sec = io.StringIO()
```

```
        # sortby = SortKey.CUMULATIVE
        # stats = Stats(prof, stream=sec).sort_stats(sortby)

        results = {'val_score': val_score,
                   'val_predictions': val_predictions,
                   'test_score': test_score,
                   'test_predictions': test_predictions}
                #    'prof_stats': stats}

        return results


    def task(params):
        ds = params['dataset']
        dsname = params['dsname']
        fidx = params['fold_idx']
        train_idx = params['folds'][fidx][0]
        val_idx = params['folds'][fidx][1]
        use_pca = params['use_pca']

        # x_train = ds.x_train[train_idx,:]
        # y_train = ds.y_train[train_idx]
        # x_val = ds.x_train[val_idx, :]
        # y_val = ds.y_train[val_idx]
        # split training data into training and validation sets
```

```
# test sets were already split before k-fold process
x_train = params['train_data']['x_train'][train_idx,:]
y_train = params['train_data']['y_train'][train_idx]
x_val = params['train_data']['x_train'][val_idx, :]
y_val = params['train_data']['y_train'][val_idx]
x_test = params['test_data']['x_test']
y_test = params['test_data']['y_test']

label_amts = {label: len(y_train[y_train==label]) for label in
↪  np.unique(y_train)}

# print(f"fold id: {fidx}\ntrain_idx: {train_idx}\ntrain_val:
↪  {y_train}\nval_idx: {val_idx}\ntrain_val: {val_idx}\n\n")

# data to be saved to file



# max_pca = min(label_amts.values())
if dsname.lower() != 'fsi_preprocessed':
    if use_pca == True:
        max_pca = 7
        pipe = Pipeline([
                ('varthresh',
                ↪  feature_selection.VarianceThreshold()),
                ('scaler', StandardScaler()),('pca', PCA()),
```

```
                    ('knn', KNeighborsClassifier(n_neighbors=7,
                    ↪  weights='distance'))
                ])
            pca_range = range(1,max_pca+1)
        else:
            max_pca = 0
            pca_range = [None]
            pipe = Pipeline([
                    ('varthresh',
                    ↪  feature_selection.VarianceThreshold()),
                    ('scaler', StandardScaler()),('pca', PCA()),
                    ('knn', KNeighborsClassifier(n_neighbors=7,
                    ↪  weights='distance'))
                ])
    else:
        # using fsi_preprocessed. data is already scaled, so don't use
        ↪  scaler
        if use_pca == True:
            max_pca = 7
            pipe = Pipeline([
                    ('varthresh',
                    ↪  feature_selection.VarianceThreshold()),
                    ('pca', PCA()),
                    ('knn', KNeighborsClassifier(n_neighbors=7,
                    ↪  weights='distance'))
```

```
            ])
        pca_range = range(1,max_pca+1)
    else:
        max_pca = 0
        pca_range = [None]
        pipe = Pipeline([
                ('varthresh',
                ↪  feature_selection.VarianceThreshold()),
                ('pca', PCA()),
                ('knn', KNeighborsClassifier(n_neighbors=7,
                ↪  weights='distance'))
            ])



x_len = max(x_train.shape)
window_size_range = np.linspace(int(x_len/16), x_len,
↪  16).astype(np.int32)
# window_size_range = np.linspace(int(x_len/4), x_len,
↪  4).astype(np.int32)
# window_step = 32
num_steps = 32
# num_steps = 2
        # {'val_score': val_score,
        #    'val_predictions': val_predictions,
```

```
            #     'test_score': test_score,
            #     'test_predictions': test_predictions,
            #     'prof_stats': stats}

krange = range(1,8)

individual_result_dt = np.dtype([
    ('val_score', np.float32, (1,)),
    ('val_predictions', y_val.dtype, y_val.shape),
    ('test_score', np.float32, (1,)),
    ('test_predictions', y_test.dtype, y_test.shape),
    ('prof_stats', Stats, (1,))
])

result_dt = np.dtype([
    ('k', np.int32, (1,)),
    ('pca_n_components', np.int32, (1,)),
    ('win_size', np.int32, (1,)),
    ('win_loc', np.int32, (1,)),
    ('result', object, (1,))
])

# sweep_data = np.ndarray(shape=(len(pca_range),
↪  len(window_size_range), num_steps), dtype=result_dt)
```

```
train_val_dt = np.dtype([('train_idx', train_idx.dtype,
↪  train_idx.shape),('val_idx', val_idx.dtype, val_idx.shape),
                         ('x_train', x_train.dtype,
                         ↪  x_train.shape),('y_train',
                         ↪  y_train.dtype, y_train.shape),
                         ('x_val', x_val.dtype,
                         ↪  x_val.shape),('y_val', y_val.dtype,
                         ↪  y_val.shape),
                         ('x_test', x_test.dtype,
                         ↪  x_test.shape),('y_test', y_test.dtype,
                         ↪  y_test.shape)],(1,))


train_val = np.array([(train_idx, val_idx,
        x_train, y_train,
        x_val, y_val,
        x_test, y_test)], dtype=train_val_dt)


#                    x                    |    y
# df cols: pca, window size, window loc, | result
shape = (len(krange), len(pca_range), len(window_size_range),
↪  num_steps)

# used to store result_dt
```

```
dataframe =
↪  pd.DataFrame(np.zeros(shape=(np.cumprod(shape)[-1],len(shape)+1)),
↪  dtype=object, columns=['k', 'pca_n', 'win_size', 'win_loc',
↪  'result'])


dt = np.dtype([
    ('alg', np.unicode_, 32),
    ('sweep_data', object, (1,)),# ('sweep_data', sweep_data.dtype,
    ↪  sweep_data.shape),
    ('train/val data', train_val_dt, (1,))
])


data = np.array( [('knn',
                   0,
                   train_val)], dtype=dt)


# data = {'alg': 'knn',
#         'sweep_data': None,
#         'train/val data': {'train_idx': train_idx, 'val_idx':
↪  val_idx,
#                            'x_train': x_train,'y_train':
↪  y_train,
#                            'x_val': x_val, 'y_val': y_val,
#                            'x_test': x_test,'y_test': y_test
#                            },
```

```
#         }


index = 0
# run knn sweep of PCA
for k_idx, kval in enumerate(krange):
    print(f'starting k={kval}')
    pipe['knn'].n_neighbors = kval
    for i, pca_n in enumerate(pca_range):
        pipe['pca'].n_components = pca_n
        for j, size in enumerate(window_size_range):
            # window_loc_range = np.arange(start= size, stop=
            ↪  x_len, step=window_step)
            window_loc_range = np.linspace(start=size, stop=x_len,
            ↪  num= num_steps).astype(np.int32)
            # results = []
            for k, loc in enumerate(window_loc_range):
                start_idx = loc - size
                stop_idx = loc

                result = run_sweep_iteration(pipe,
                                             x_train[:, start_idx:
                                             ↪  stop_idx], y_train,
                                             x_val[:, start_idx:
                                             ↪  stop_idx], y_val,
```

```
                                        x_test[:, start_idx:
                                        ↪ stop_idx], y_test)
                    dataframe.iloc[index] = {'k': kval,'pca_n': pca_n,
                    ↪ 'win_size': size, 'win_loc': loc, 'result':
                    ↪ result}
                    index += 1
                    # results.append({'pca_n_components': pca_n,
                    #                 'win_size': size,
                    #                 'win_loc': loc,
                    #                 'result': result})

                    # first index is 0 because for some reason numpy is
                    ↪ making sweep_data shape
                    # (1, len(pca_range), len(window_size_range),
                    ↪ num_steps)
                    # data['sweep_data'][0, i, j, k] =
                    ↪ {'pca_n_components': pca_n,
                    #                       'win_size': size,
                    #                       'win_loc': loc,
                    #                       'result': result}
                    # data['sweep_data'][0, i, j, k] =
                    ↪ np.array([(pca_n, size, loc, result)],
                    ↪ dtype=result_dt)
                # sweep_data[i, j] = results
```

```
                print(f" finished fold {fidx}:\n\tpca:
                ↪  {pca_n}\n\twin_size: {size}\n\n")

            # print(f"run ({n})")
            # print(stats.__str__())
            # print("")

    iobytes = io.BytesIO()
    print(f"index: {index}\nlength: {len(iobytes.getvalue())}")
    dataframe.to_pickle(iobytes)
    data['sweep_data'] = iobytes.getvalue()

    if use_pca:
        data_dir = 'sweep_data/knn_sweeps/'
        with open( data_dir +
        ↪  f'{dsname}/knn_dataframe_{dsname}_fid{fidx}.pickle', 'wb')
        ↪  as file:
            pickle.dump(data, file)
    else:
        data_dir = 'sweep_data/knn_sweeps/'
        with open(data_dir +
        ↪  f'{dsname}/knn_dataframe_nopca_{dsname}_fid{fidx}.pickle',
        ↪  'wb') as file:
            pickle.dump(data, file)
```

```
        # with open(f'sweep_data/knn_sweeps/knn_data{fidx}.hkl', 'wb') as
        ↪  file:
        #     hickle.dump(data, file)
        # fname = r"absolute\path\to\sweep_data\knn_sweeps" +
        ↪  f"\knn_data{fidx}.hkl"
        # hickle.dump(data, fname, mode='wb')
```

```
def load_dataset(ds_name: str):
    match ds_name.lower():
        case 'bpy':
            ds = fuel_dataset.FuelDataset(data_dir =
            ↪  'absolute/path/to/Fuel_Dataset/files/',
                        outlet = 'bpy')
        case 'p666':
            ds = fuel_dataset.FuelDataset(data_dir =
            ↪  'absolute/path/to/Fuel_Dataset/files/',
                        outlet = 'p666')
        case 'fox1' | 'fox2' | 'fox3' | 'fox4' | 'fox5':
            outlet=ds_name.lower()[-1]
            fname = 'COMPILED_AND_NORMALIZED.xlsx'
            ds = custom_dataset.CustomDataset(filename=fname,
```

```
                    data_dir='absolute/path/to/fox/data/',
                    labels=[f"FOX_{outlet}_22", f"FOX_{outlet}_23A",
                    ↪  f"FOX_{outlet}_23B", f"FOX_{outlet}_24"])
        case 'fsi_unweighted' | 'fsi_preprocessed' | 'fsi_fr_sig':
            ds = fox_sig_int.FoxSigIntDataset()
        case _:
            raise ValueError
    return ds


def main():
    n_splits = 5
    num_processes = n_splits if n_splits < 8 else 8
    pool = mp.Pool(processes=num_processes)

    dsname = 'fsi_fr_sig'
    use_pca = True
    match dsname.lower():
        case 'fox1' | 'fox2' | 'fox3' | 'fox4' | 'fox5':
            ds = load_dataset(dsname)
            x_train, x_test, y_train, y_test = train_test_split(ds.x,
            ↪  ds.y, test_size=0.2, random_state=42)
        case 'bpy' | 'p666':
            ds = load_dataset(dsname)
            x_train = ds.x_train
            y_train = ds.y_train
```

```
            x_test = ds.x_test
            y_test = ds.y_test
        case 'fsi_unweighted':
            ds = load_dataset(dsname)
            x_train, x_test, y_train, y_test =
            ↪ train_test_split(ds.data['fused_unweighted'],
            ↪ ds.data['y'], test_size=0.2, random_state=42)
        case 'fsi_preprocessed':
            ds = load_dataset(dsname)
            x_train, x_test, y_train, y_test =
            ↪ train_test_split(ds.data['fused_preprocessed'],
            ↪ ds.data['y'], test_size=0.2, random_state=42)
        case 'fsi_fr_sig':
            ds = load_dataset(dsname)
            x_train, x_test, y_train, y_test =
            ↪ train_test_split(ds.data['fr_sig'], ds.data['y'],
            ↪ test_size=0.2, random_state=42)
        case _:
            raise ValueError('Unknown data set name')

    # shuffle data since KFold won't shuffle for some reason
    # x_train_shuf, y_train_shuf = shuffle(x_train, y_train,
    ↪ random_state=42)
    # x_test_shuf, y_test_shuf = shuffle(x_test, y_test,
    ↪ random_state=42)
```

```
    # kfold
    skf = StratifiedKFold(n_splits=n_splits, shuffle=True,
    ↪  random_state=42)
    folds = [(shuffle(fold[0], random_state=42), shuffle(fold[1],
    ↪  random_state=42)) for fold in skf.split(x_train, y_train)]

    # have each process do a fold
    pool.map(task,
             [{'fold_idx': i,
               'dataset': ds,
               'dsname': dsname,
               'use_pca': use_pca,
               'folds': folds,
               'train_data': {'x_train': x_train, 'y_train': y_train},
               'test_data': {'x_test': x_test, 'y_test': y_test}} for i
               ↪  in range(n_splits)])


if __name__ == '__main__':
    main()
```

A.2 SVM Hyperparameter Tuning and Evaluation

Similar to the previous section, the code has been altered to remove any sensitive information.

```
# from sklearnex import patch_sklearn
# patch_sklearn()



# Lib Imports
import multiprocessing as mp
from sklearn.covariance import EllipticEnvelope
import numpy as np
import sys
import importlib
# import matplotlib
# import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.tree import DecisionTreeClassifier
import sklearn.feature_selection as feature_selection
from sklearn.neighbors import KNeighborsClassifier
from sklearn.utils import shuffle
```

```
from sklearn.model_selection import StratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
sys.path.insert(0, "absolute\\path\\to\\python_files")
import fox_sig_int
from scipy.signal import savgol_filter
import datetime
#save history
import pickle
# import hickle
# Profiling
import cProfile
from pstats import SortKey, Stats
import io
import pandas as pd
import io

# Local Imports
# import custom_dataset
import fuel_dataset
from custom_dataset import CustomDataset, LabelData
import pca as custom_pca
import outliers as custom_outliers
```

```
def load_data(dsname: str):
    ds = None
    # with open(r"absolute\path\to\sweep_data\knn_sweeps" +
    ↪  f"\knn_data{num}.pickle", 'rb') as file:
    #     result = pickle.load(file)
    match dsname.lower():
        case 'fox1' | 'fox2' | 'fox3' | 'fox4' | 'fox5':
            fox_num = int(dsname.lower()[-1])
            ds =
            ↪  CustomDataset(filename='COMPILED_AND_NORMALIZED.xlsx',
                        data_dir='absolute/path/to/dox/dataset/',
                        labels=[f"FOX_{fox_num}_22",
                        ↪  f"FOX_{fox_num}_23A",
                        ↪  f"FOX_{fox_num}_23B",
                        ↪  f"FOX_{fox_num}_24"])
        case 'bpy':
            ds = fuel_dataset.FuelDataset(data_dir =
            ↪  'absolute/path/to/Fuel_Dataset/90-100/',
                    outlet = 'bpy')
        case 'p666':
            ds = fuel_dataset.FuelDataset(data_dir =
            ↪  'absolute/path/to/Fuel_Dataset/90-100/',
                    outlet = 'p666')
        case 'fsi_unweighted' | 'fsi_preprocessed' | 'fsi_fr_sig':
            ds = fox_sig_int.FoxSigIntDataset()
```

```
            case _ :
                raise ValueError('Unknown Data Set Name')

        return ds


def main(dsname: str, params: dict):
    # use_pca = params['use_pca']
    ds = load_data(dsname)

    # if use_pca:
    #     n_components = 4
    # else:
    #     # if this is None PCA uses all the features
    #     n_components = None

    x_train = None
    y_train = None
    # get x_train and x_test
    match dsname.lower():
        case 'p666' | 'bpy':
            x_train = ds.x_train
            y_train = ds.y_train
            x_test = ds.x_test
            y_test = ds.y_test
```

```
        case 'fox1' | 'fox2' | 'fox3' | 'fox4' | 'fox5':
            X = ds.x
            Y = ds.y
            x_train, x_test, y_train, y_test = train_test_split(X,
            ↪  Y, test_size=0.2, random_state=42)
        case 'fsi_unweighted':
            x_train, x_test, y_train, y_test =
            ↪  train_test_split(ds.data['fused_unweighted'],
            ↪  ds.data['y'], test_size=0.2, random_state=42)
        case 'fsi_preprocessed':
            x_train, x_test, y_train, y_test =
            ↪  train_test_split(ds.data['fused_preprocessed'],
            ↪  ds.data['y'], test_size=0.2, random_state=42)
        case 'fsi_fr_sig':
            x_train, x_test, y_train, y_test =
            ↪  train_test_split(ds.data['fr_sig'], ds.data['y'],
            ↪  test_size=0.2, random_state=42)
        case _:
            raise ValueError(f'Unknown dataset {dsname}')

    if dsname != 'fsi_preprocessed':
        pipe = Pipeline([
            ('varthresh', feature_selection.VarianceThreshold()),
            ('scaler', StandardScaler()),
            ('pca', PCA()),
```

```
            ('svc', SVC(kernel=params['kernel'])),
        ])
    else:
        pipe = Pipeline([
            ('varthresh', feature_selection.VarianceThreshold()),
            # ('scaler', StandardScaler()),
            ('pca', PCA()),
            ('svc', SVC(kernel=params['kernel'])),
        ])


    # if use_pca:
    #     pca_list = [1,2,3,4,5,6,7]
    # else:
    #     pca_list = [None]
    # pca_list = [1,2,3,4,5,6,7]
    pca_list = params['pca_list']
    if params['kernel'].lower() != 'rbf':
        parameters = {'pca__n_components': pca_list,
                      'svc__C': np.logspace(start=-10, stop=2,
                      ↪  base=10, num=16),
                     'svc__gamma': np.logspace(start=0, stop=3,
                     ↪  base=10, num=16),
                     }
    else:
        parameters = {'pca__n_components': pca_list,
```

```
                    'svc__C': np.logspace(start=-2, stop=10,
                    ↪  base=10, num=16),
                    'svc__gamma': np.logspace(start=-9, stop=3,
                    ↪  base=10, num=16),
                    }

grid = GridSearchCV(estimator=pipe,
                        param_grid=parameters,
                        cv = 5, # specifies 5 folds for stratified
                        ↪  k-fold
                        n_jobs=-1,
                        return_train_score=True,
                        error_score='raise',
                        verbose=1)

if 'lbound' in params.keys() and 'rbound' in params.keys():
    lbound = params['lbound']
    rbound = params['rbound']

    grid.fit(X=x_train[:, lbound:rbound], y=y_train)
    # print(grid.cv_results_)
    # apply best params
    pipe.set_params(**grid.best_params_)
    pipe.fit(X=x_train[:, lbound:rbound], y=y_train)
```

```
        final_score = pipe.score(X=x_test[:, lbound:rbound],
        ↪  y=y_test)
    else:
        grid.fit(X=x_train[:, :], y=y_train)
        # print(grid.cv_results_)
        # apply best params
        pipe.set_params(**grid.best_params_)
        pipe.fit(X=x_train[:, :], y=y_train)
        final_score = pipe.score(X=x_test[:, :], y=y_test)

    print('\n\n-------------------------------------------------' +
    ↪  '-------------------------------------------------------')
    print(f"Data Set: '{dsname}'")
    print(f'best_params: {grid.best_params_}')
    print(f'final score: {final_score}\n')
    print(f'\ntested {dsname} with params {params} and grid params
    ↪  {parameters}')


    # scoring = ['validation', 'testing']

    # cv_results = cross_validate(pipe, X=x_train, y=y_train)

    # pipe.fit(X=x_test, y=y_test)
    # test_results = pipe.score()
```

```
        # print(f"validation scores: {cv_results['test_score']}")


if __name__ == "__main__":
    dsname = 'fsi_preprocessed'
    # for FOX, use entire dataset
    # params = {'lbound': 0, 'rbound': 1000, 'kernel':'poly',
    ↪  'pca_list': [None]}
    for pca in [None,1,2,3,4,5,6,7]:
        params = {'kernel':'linear', 'pca_list': [pca]}
        # params = {'lbound': 0, 'rbound': 1043,'kernel':'rbf',
        ↪  'pca_list': [pca]}
        main(dsname, params)
```