ECS8051 Final Assignment

Hugo Collins
Student Number: 40446925
Email: hcollins05@qb.ac.uk
MSc in Artificial Intelligence 2024

All code can be found below the report
Please note that there is a lot of code due to the fact that it had to be duplicated for matrix dmem and matrix pbs data. The pbs code is not commented as it follows the exact same procedure as the dmem code.

Abstract

This exploratory report investigates a dataset from a virus detection experiment, exploring the application of various machine learning models and techniques to capture different patterns in the data. The report examines three approaches: regression, classification, and clustering. The regression model aims to predict viral load, while the classification model aims to predict the presence of the virus using binary classification. Additionally, the clustering approach attempts to group the data into clusters and predict viral load based on cluster means. Given the sequential nature of the spectral data, this study addresses the challenges posed by the dataset's complexity. The goal of this report is to evaluate the effectiveness of both simple and complex machine learning models, focusing on their ability to capture underlying relationships in the data using linear and non-linear methods. Rather than being purely results-driven, the report emphasizes understanding the relationships between variables and exploring the reasons behind the models' performance. This approach provides deeper insights into the data and the mechanisms influencing the models' predictions, aiming to guide future improvements in data processing and model development.

Introduction

Spectral data, especially when used for virus detection, is often noisy and difficult to interpret. Various factors such as equipment limitations, environmental conditions, and sample variations can lead to overlapping spectra, making it challenging to identify clear patterns. While portable spectrometers offer the benefits of being affordable, fast, and easy to use, they often produce lower-quality data, which further complicates the analysis. In this report, I work with a dataset consisting of intensity readings at different wavelengths, aiming to predict key aspects like viral load and virus type. To tackle the challenges posed by noisy data, I will apply a broad range of both linear and non-linear machine learning models. These models will be evaluated alongside various feature extraction techniques, with the goal of identifying effective methods for improving prediction accuracy and uncovering meaningful patterns in the data. By combining different models and techniques, I aim to assess the potential for identifying complex, non-linear relationships in the spectral data.

Data/EDA

This dataset, collected as part of a virus detection experiment, consists of 3081 rows and 516 columns, making it highly dimensional. It combines data from four separate sessions, identified by the "SID" column. One concern is that SID 1 has a distribution significantly different from the other sessions, which may pose challenges in a dataset already limited in resources. The dataset includes columns such as "Type" (Virus Type) with values "X" and "Y," "Matrix" (Medium) indicating the environment (e.g., dmem or pbs), "Load" (Serial Dilution Level) reflecting the virus sample dilution, and I001 to I512, which represent light intensity measurements at various wavelengths.

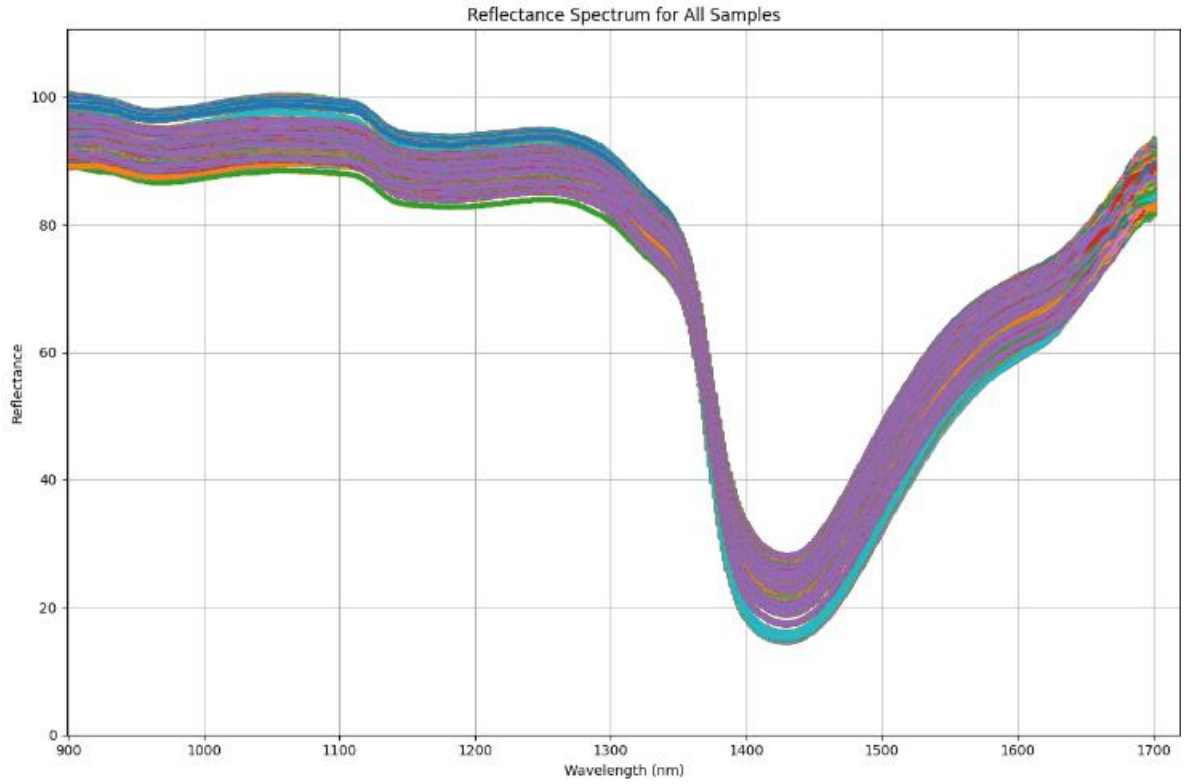

Fig 1. *Spectral Data plotted against wavelengths*

I carried out some exploratory analysis on the data to get some understanding of the relationship between variables:

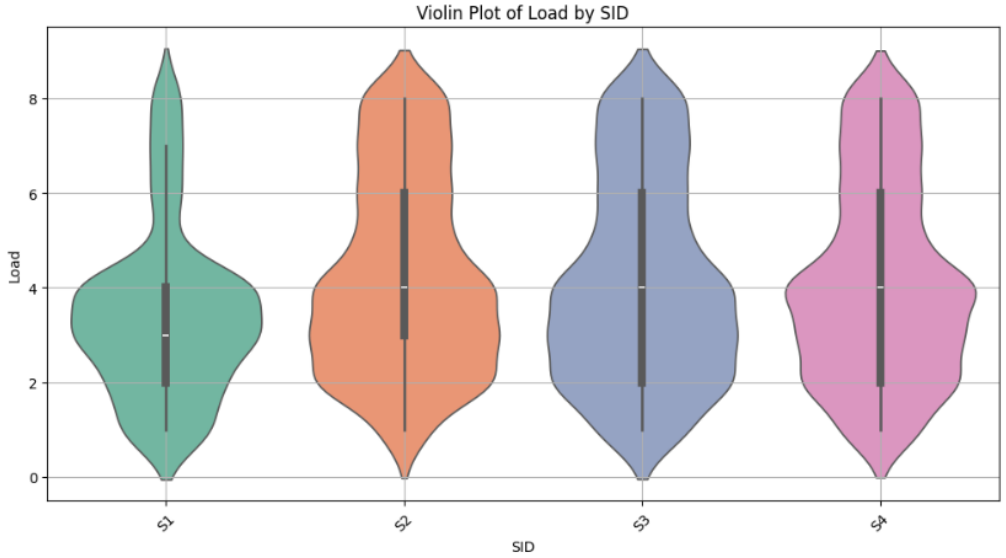

Fig 2 *Violin Plot of Load Distribution by Setup ID*

Fig 2. showed that S2,S3,S4 all followed similar distributions whereas S1 had a far lower median value and lower interquartile range**.**

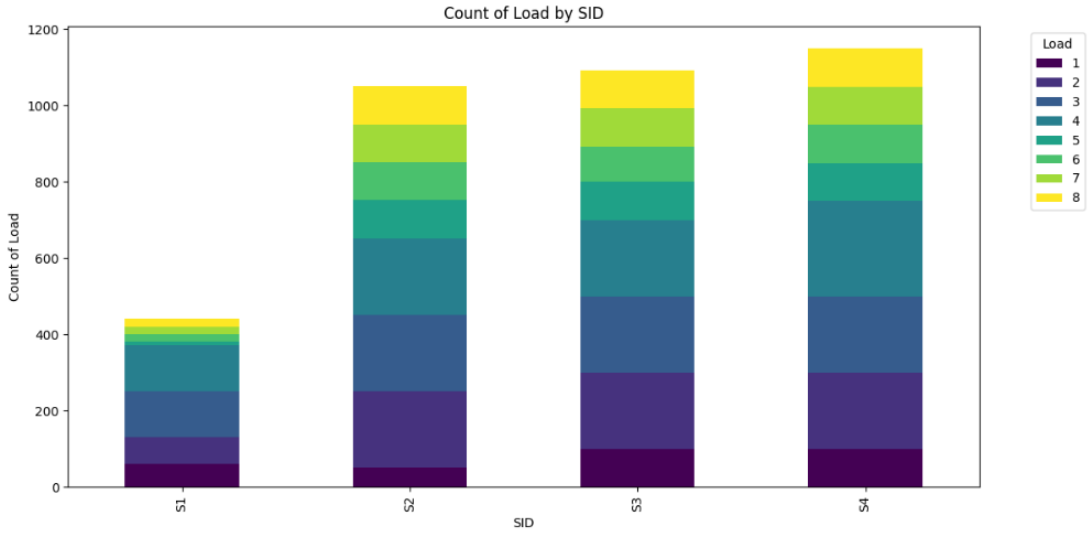

Fig 3. *Count of Loud Type by Session ID*

There is an even spread throughout the sessions with each session appearing to have an increase in load around the central tendency.

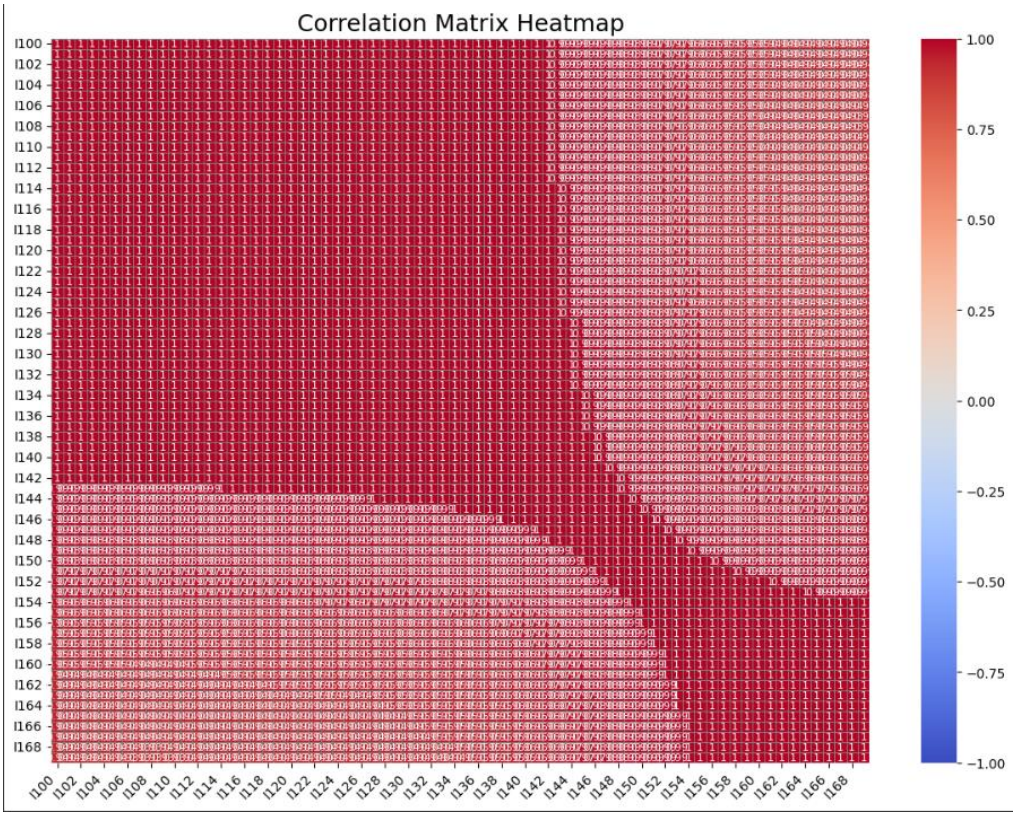

Fig 4. *Correlation Matrix of subset of data from index 100-170*

This correlation matrix was taken from the subset of the data and shows the extremely high correlation between features. This multicollinearity is something that could cause issues including feature redundancy and overfitting.

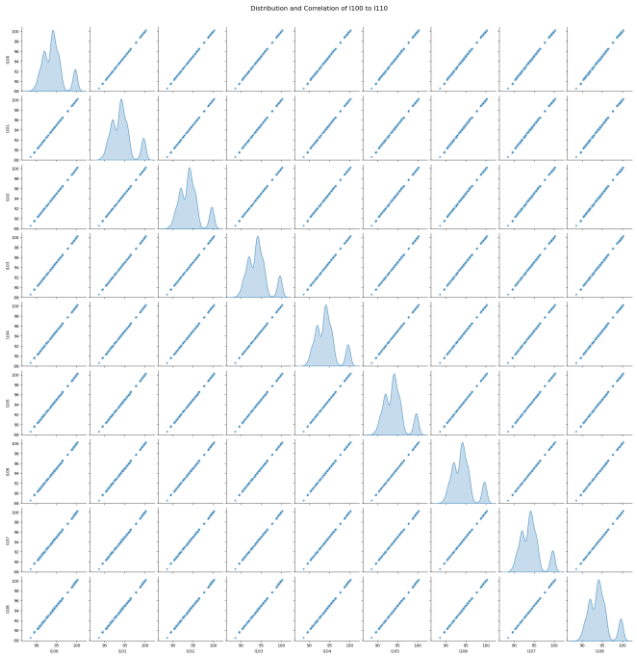

Fig 5. *Pairplot of subset of intensity columns ranging from 100-111*

This plot was again taken from a subset of the data and shows the linear interaction between features in close quarters. The distributions also appear uniform. Methods such as PCA may be effective in capturing this linear relationship.

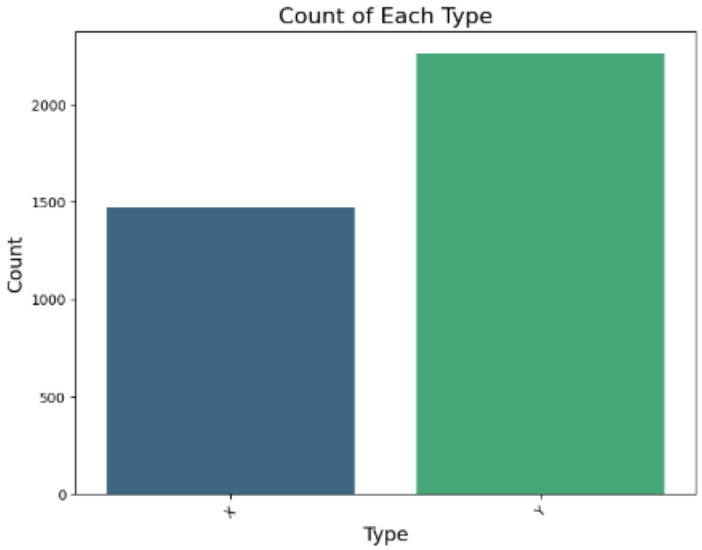

Fig 6. *Countplot by Type*

The above countplot shows that there is a class imbalance in the target variable type for classification. This is something that I will have to account for.

General Train Test Split:

A key challenge in this project was selecting the train/test split due to the sequential nature of the spectral data and the high correlation between intensity columns. A random split caused data leakage, inflating performance metrics by allowing the model to learn patterns from the test data. A split with random=False still didn't ensure proper distribution of target variables, as seen with a 15% test split containing only one class. Ultimately, I chose to use the final 30% of the data for testing to avoid data leakage, although this reduced the amount of training data and led to a slightly different distribution in the test set, which could impact results. However, this was the best approach for assessing the model's generalisation ability.

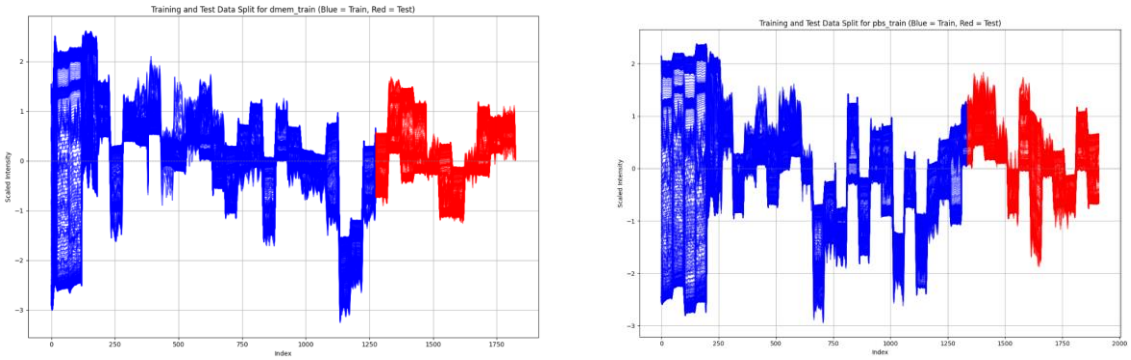

Fig 7, 8 *Train and Test Split of dmem and pbs data*

Classification:

The first step I took was stripping column entries of whitespaces and removed outliers using the z-score method, reducing the dataset from 3801 to 3735 samples. After experimenting with several normalisation methods, I chose StandardScaler() to scale the features. I also used Binary mapping to map X to 0 and Y to 1 to make the target variable compatible with some on my models. During my literature review, I discovered that smoothing techniques could be applied to noisy spectral data to create more informative features. I

played around with methods such as fitting a polynomial using Fourier smoothing, but the best results came when I applied the Savitzky-Golay smoothing filter. This reduced the noise in my data and my aim was that it would allow the model to see relative trends and not fit to the noise. I experimented with a number of different setups but settled on a window size of 13 and poly 2 as the best setup. Figure 9 shows the effects of noise reduction below.

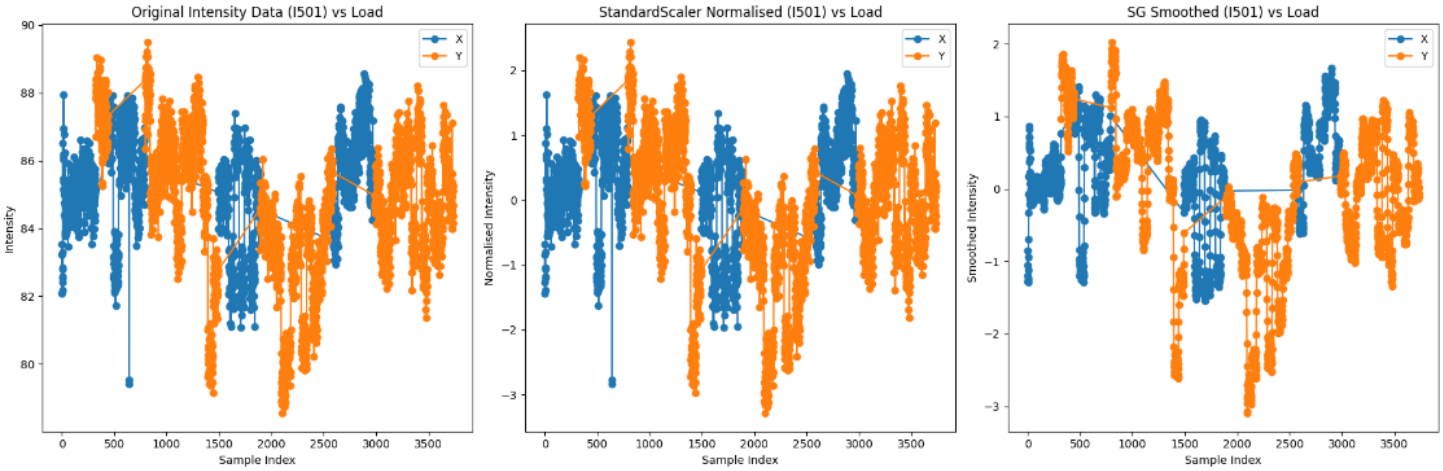

Fig 9: *Data originally, after normalisation, and after smoothing*

The training/test split was done as mentioned above taking the final 30% to ensure a count of both variables in target column Type. This did leave a shortage of training data but will use cross validation to counteract this.

Results and Analysis

Note: These results are from the dmem data as both models returned similar scores and due to word restrictions, I was unable to analyse both.

As a very rough baseline the first step I took was to simply fit the features as they were into a Logistic Regression model and a Support Vector Classifier. As expected, the results were extreme overfitting with the logistic model returning very unstable results and almost perfect F1 and Balanced accuracy scores. These models would not be able to generalise well and would completely fail on test sets, so it was clear a lot of feature engineering was necessary.

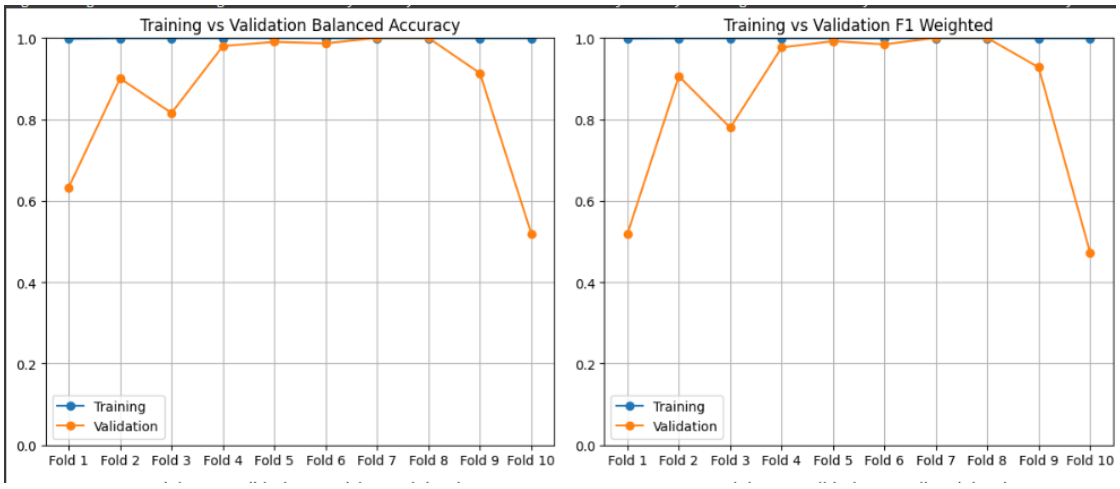

Fig 10 *Logistic Reg – All Features*

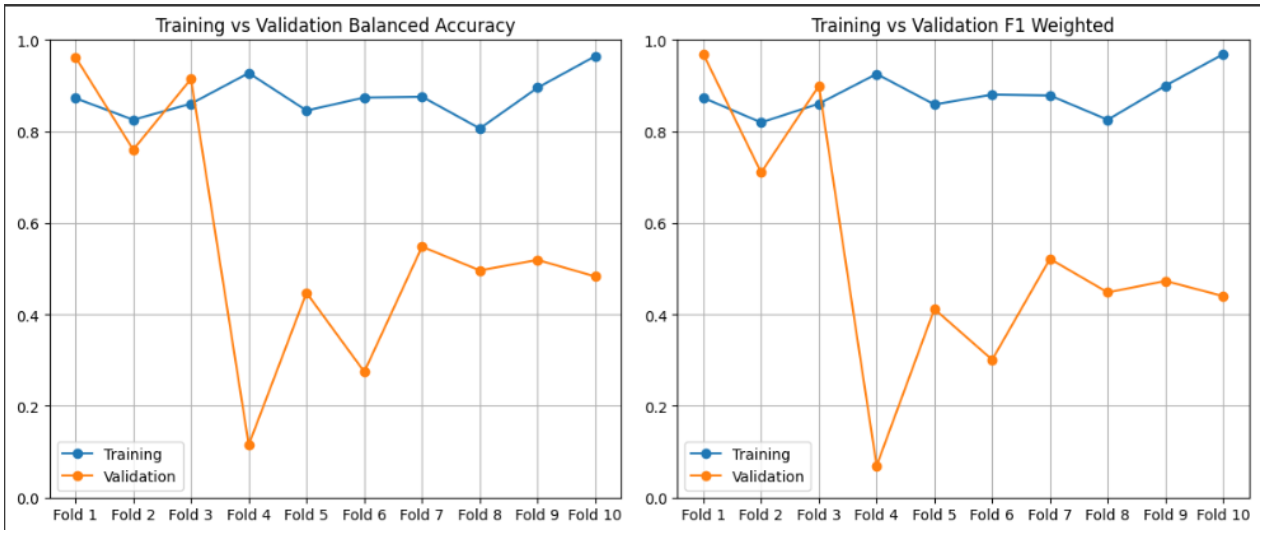

Fig 11 *Support vector classifier all features*

Logistic With C parameter tuned and paired with L1 Regularisation

To handle the large feature space and prevent overfitting, I applied L1 regularisation (Lasso), which shrinks irrelevant features to zero, enhancing both model interpretability and computational efficiency. I chose L1 over L2 specifically for this feature-selection ability. For tuning the penalty parameter C, I used stratified cross-validation without shuffling to prevent data leakage and address class imbalance. Higher C values set more coefficients to zero, improving training performance, but I needed to balance this against overfitting risk. Ultimately, I selected C based on the best validation F1 score, though this tuning process was computationally intensive.

C – Values	Mean Accuracy CV Validation	Mean AUC CV Validation	Mean F1 Score CV Validation
0.1	0.7429	0.7199	0.7680
0.5	0.8394	0.9154	0.8334
1	0.8457	0.9470	0.8392
10	0.8590	0.9704	0.8550

Table 1. *Logistic L1 results from Cross Validation*

PLSDA – Partial Least Squares Discriminant Analysis

The next approach I tried was PLS-DA. The reason behind this choice of model came from the structure of my data. Due to the multicollinearity in my data, I found that PCA was not as suitable dimensionality reduction technique as PLS, as PLS makes use of the Y variables and creates features related to the target. It better captures the complex relationship between variables due to the amount of collinearity.

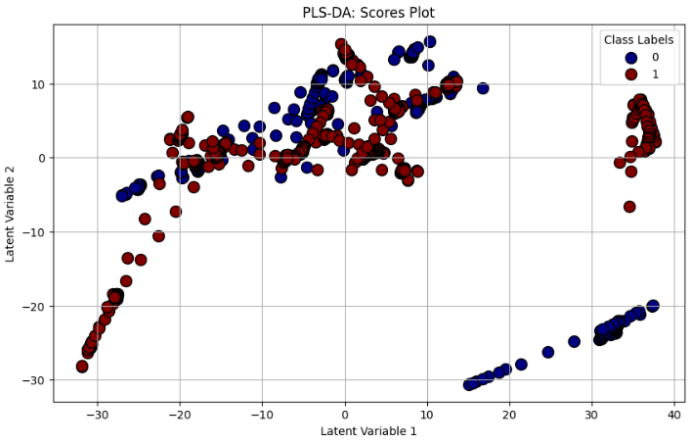

Fig 12 *PLS of Dmem data projected in a 2D space and labelled by class*

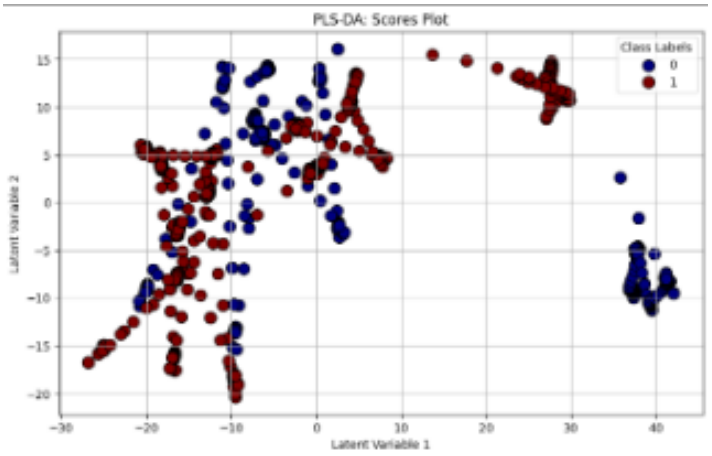

Fig 13. *PLS of pbs data projected in a 2D space and labelled by class*

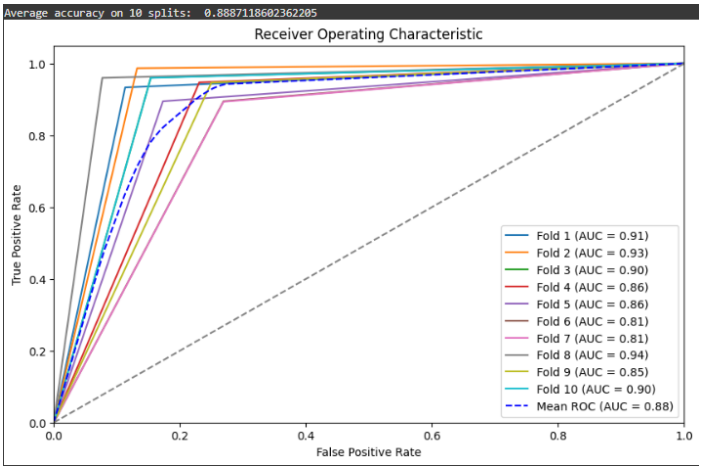

Fig 14 *PLS-DA Dmem ROC Curve*

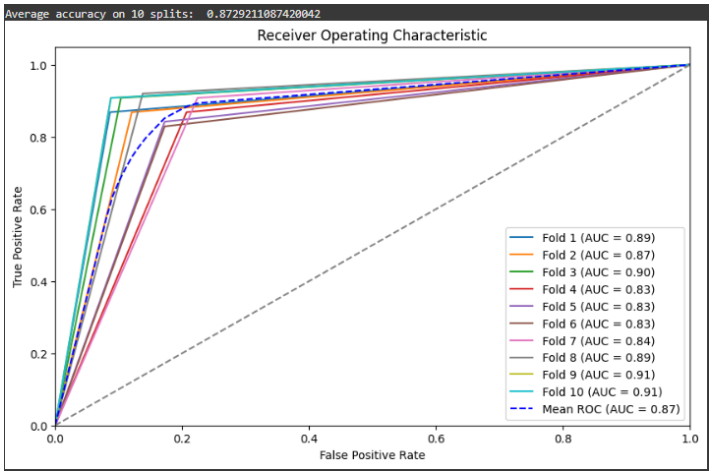

Fig 15 *PLS-DA Pbs Roc Curve*

Here we can see that the PLS data has been relatively successful in reducing the dimensionality of the feature space. We can see from both datasets that the model is stable across the different folds with the average ROC being at .88 and .87.

SVC -

Use stratified cross validation again for C parameter and experimented with different kernels including “Poly”, “Rbf” and “Linear”. Interestingly linear performed the best out of these kernels. The C parameter controls the bias variance tradeoff for SMC. When C is low this can lead to a small amount of support vectors and cause overfitting but when C is too large it creates a model with high bias and low variance which leads to underfitting and guess work. The graph below shows the results of my C tuning through stratified cross validation

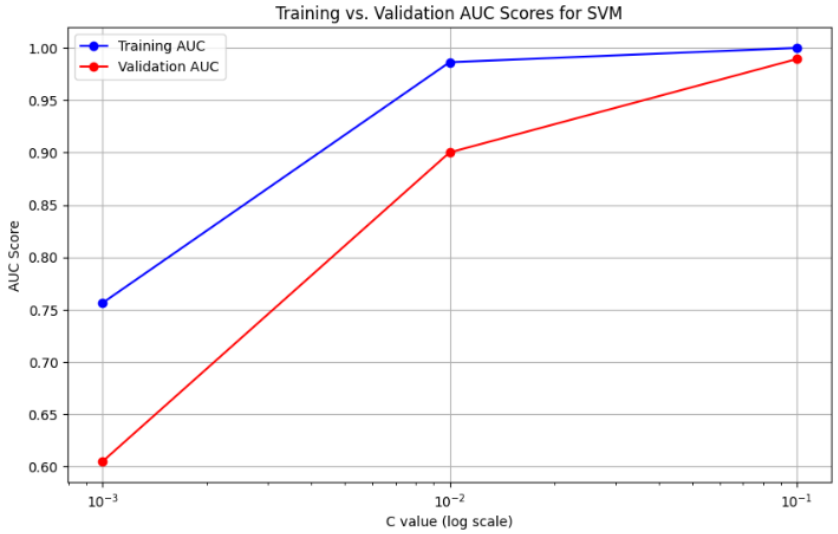

Fig 16. AUC for C values

KNN

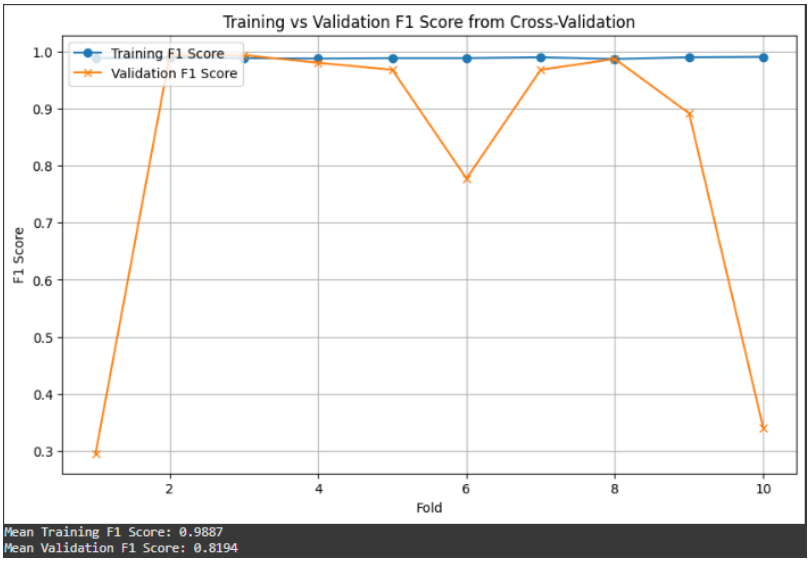

Fig 17 Training and Validation F1 scores KNN

KNN also struggled with overfitting despite tuning the number of nearest neighbours. There is great fluctuation in the F1 across scores which questions the stability of the model.

Tried RFE with Stratified Cross validation:

To improve feature extraction, I then applied Recursive Feature Elimination using sklearn which implements a backward selection process starting with a model of all the predictors and computing importance score for each and selecting top n features. I created a model that checked CV scores at every 20 features.

```
>5 features: Mean CV Score = 0.794 (Std = 0.141)
>25 features: Mean CV Score = 0.710 (Std = 0.169)
>45 features: Mean CV Score = 0.711 (Std = 0.160)
>65 features: Mean CV Score = 0.686 (Std = 0.161)
>85 features: Mean CV Score = 0.712 (Std = 0.161)
>105 features: Mean CV Score = 0.721 (Std = 0.161)
>125 features: Mean CV Score = 0.693 (Std = 0.152)
>145 features: Mean CV Score = 0.715 (Std = 0.150)
>165 features: Mean CV Score = 0.711 (Std = 0.164)
>185 features: Mean CV Score = 0.710 (Std = 0.181)
>205 features: Mean CV Score = 0.707 (Std = 0.169)
```

Fig 18. *RFE mean CV score per feature*

This method was extremely computationally expensive and took over three hours to run. Again, it appears that in their current state the features hold very little importance and contain too much noise.

The final method I tried was an exploratory method that involved using variance as a threshold for feature selection and combining that with Kernal PCA in attempt to combine in a nonlinear fashion.

```
Best Parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 100}
Training F1 Score: 1.0000
Training Accuracy: 1.0000
Validation F1 Score: 0.6167
Validation Accuracy: 0.6599
```

Fig 19 *Variance Thresholding with Kernal PCA*

There was still quite significant overfitting present even after using stratified CV.

Test Set Analysis:

Here I will analyse the results of one model from a dimensionality reduction, regularisation and nonlinear feature extraction approach:

PLS-DA – Test Accuracy 0.83

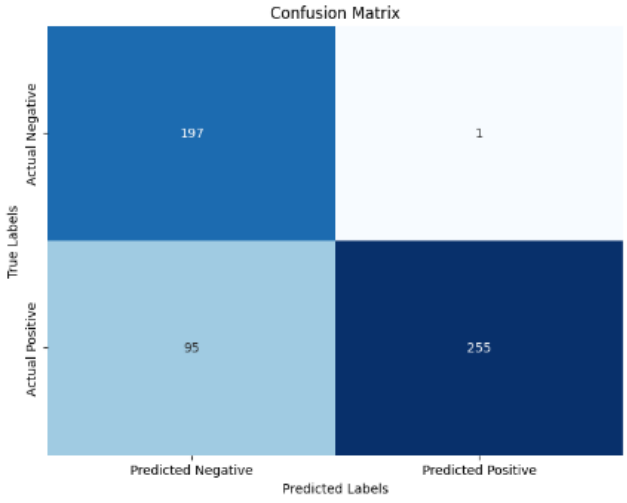

Fig 19 *Confusion Matrix PLS-DA*

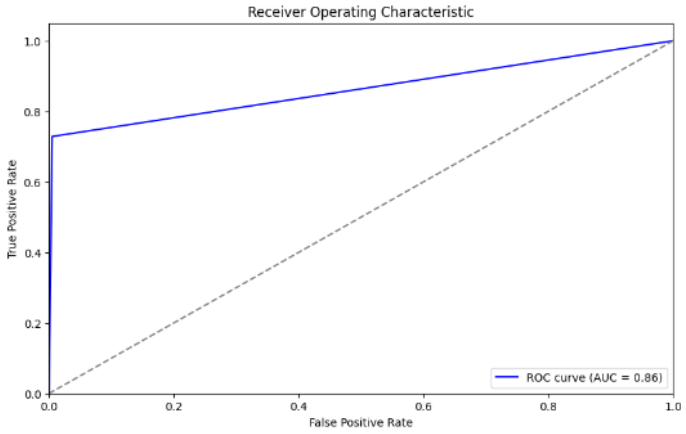

Fig 20 *ROC PLS-DA*

Target	Precision	Recall	F1-Score
X	0.67	0.99	0.81
Y	1	0.73	0.84

Table 2 – *PLS-DA Results*

The model performs well on class X with a high recall, meaning it correctly identifies most X samples, though there's some false positives. For class Y, precision is perfect, but recall is lower meaning some Y samples are missed. The F1-scores show a reasonable fit to the data. Overall, the PLS-DA model performs adequately and outperforms PCA based on how it can handle multicollinearity, especially considering the distribution of the test set being slightly different to the training data overall.

Support Vector Classifier – Test Set Accuracy 0.89

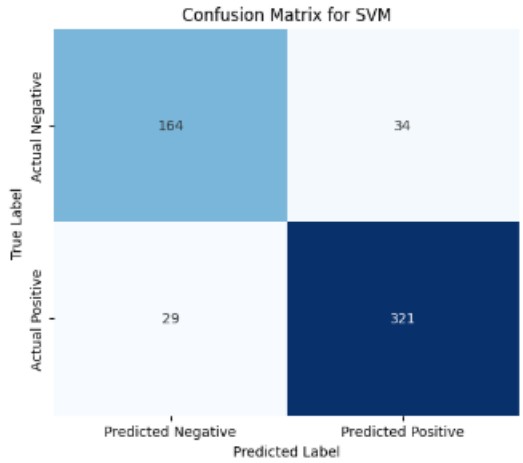

Fig 21 *Confusion Matrix SVM*

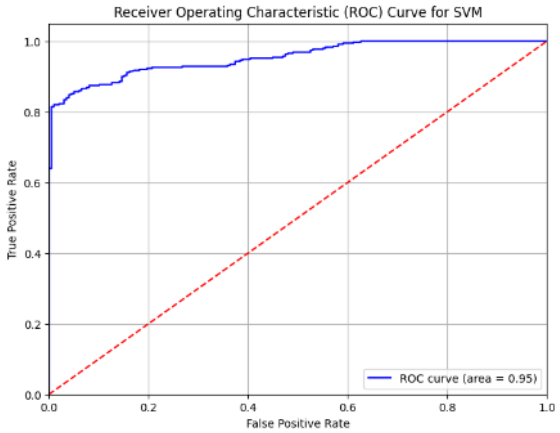

Fig 22 *ROC Curve SVM*

Target	Precision	Recall	F1-Score
X	0.85	0.83	0.84
Y	0.90	0.92	0.91

Table 3 *Test Results SVM*

SVC with a linear kernel and hypertuned C outperformed my other regularisation methods. This was a better-rounded model than my PLS-DA model with improvements in the stability of precision and recall. The ROC curve was close to the left top corner signifying a good model and the AUC was .95 which was high.

Random Forest using Kernal PCA and Variance Thresholding: Test Set accuracy – 0.72

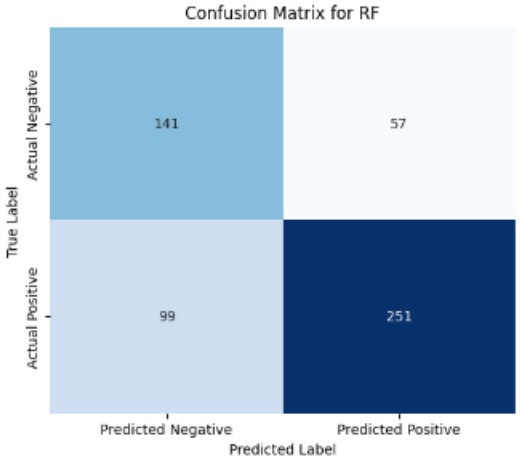

Fig 23. *Confusion Matrix RF with Kernel PCA*

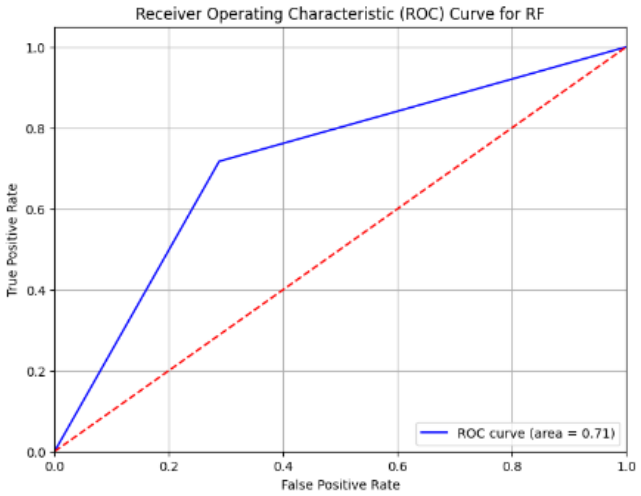

Fig 24 *ROC Curve RF with Kernel PCA*

Target	Precision	Recall	F1-Score
X	0.59	0.71	0.64
Y	0.81	0.72	0.76

Table 3 *Test Results RBF Kernel PCA*

Although these results may seem disappointing at first glance, there were several key takeaways. The RBF kernel with PCA showed a noticeable improvement over the previous, more complex models. The variance thresholding was initially an exploratory step, but with additional domain knowledge, it could become a more effective method. While the results aren't ideal, I believe this is largely due to differences in the test set distribution compared to the training set. With further domain expertise, such as converting readings to absorbance, I believe this approach could be the most effective model overall as it would be able to capture more complex relationships, especially when provided with more insightful training data.

Overall Discussion:

Linear models performed adequately on this particular test set but would struggle to get extremely accurate results consistently due to the complex relationship between features. Methods such as PLS proved to be somewhat effective, and regularisation definitely helped to improve the models' abilities to generalise. However, to gain a truly accurate model ready for public use, it appears that more in depth feature preprocessing may be necessary. The variance threshold technique applied appears to show promise. Given more time and computational resources, I would investigate some significant peaks algorithm or employ some domain expert knowledge to gain more insights from the features. It is also clear that the dimensionality reduction techniques such as PCA fail to capture the relevant patterns and are not adequate features for this model indicating a beyond linear relationship between features and the target variable type. More complex ensemble models such as random forest struggled here as the features didn't contain enough information and led to overfitting despite the extensive efforts through cross validation and feature extraction. Using a random traintest spilt would have given me great scores for my models but would have been a result of data leakage and despite these results being disappointing I have explored numerous techniques and gained valuable insights into proper model evaluation and with some work these models would be effective.

Regression:

Following on from my classification analysis I can see linear models do not model the relationship between the intensity readings well. As my regression task has a different target variable, I will still explore some simpler models, but I hope to explore a bit more beyond linearity and assess other techniques. For my preprocessing again I followed the same steps as above regarding the SG smoothing and normalisation of the data but used MinMaxScaler() instead this time because it transforms the data into a fixed range. In this case I set Load as my target variable and split based on the methodology used throughout the report.

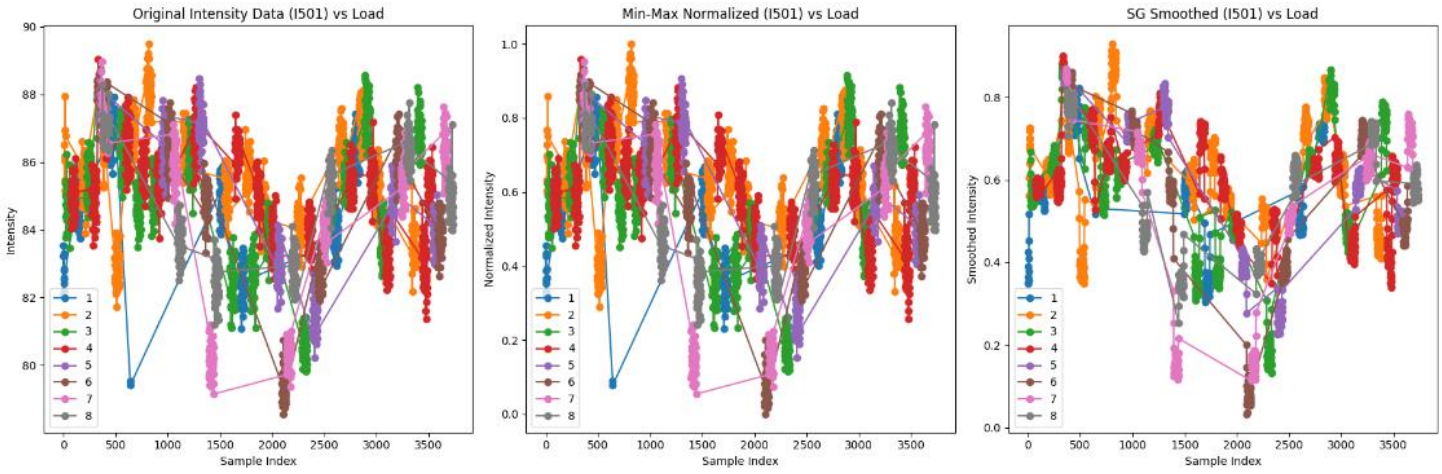

Fig 25 *Data originally, after normalisation, and after smoothing*

Partial Least Squares

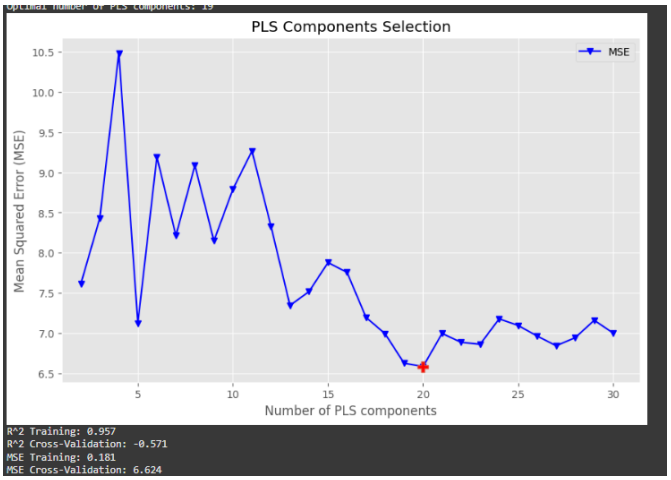

Fig 26 *Optimal Number of PLS Components*

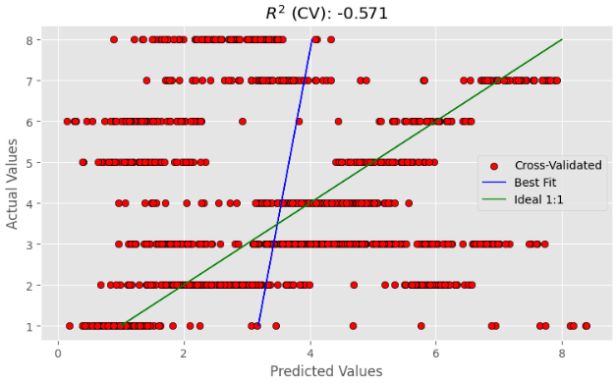

Fig 27 *CV Line vs line of best fit*

Following the success of PLS compared to PCA in the previous classification experimentation, I decided to again use PLS as a feature extraction method. I used a 10-fold cross validation method to determine the optimal number of components and then used these components to create a PLS regression model.

GAM with polynomial

It was clear to me that the features in their current state weren't effective so to allow for non-linear relationships between features and response variable Type, I wanted to fit a GAM to the data. As GAM are additive, feature selection was necessary to allow it to work so I used thresholding as a feature selection process and Poly() at degree three to apply cubic feature expansion.

Lasso Hyperparameter

I then experimented with some Lasso normalisation and tuned the hyperparameter alpha, while avoiding any data leakage by using nested cross validation. The results here were extremely poor with Lasso either sending all the variables to zero or if alpha was reduced it gave very poor predictions on validation and was highly unstable as shown in the figure below

```
Validation MSE scores for each fold:
[28.98917277 10.77053195  1.94741605  5.1884396   4.04408827 10.21451523
  3.72597651  2.03275379  0.48693334 12.88033353]

Mean Training MSE: 2.8598
Standard Deviation of Training MSE: 0.2874

Mean Validation MSE: 8.0200
Standard Deviation of Validation MSE: 8.0256
```

Fig 28 *MSE Scores for training and validation Lasso*

RFE CV with Random Forest and XGBoost

I then decided to try fitting an XGBoost model to the data using RFE. I came across the XGBoost package during my research. It is a decision tree ensemble learning algorithm and holds similarities to RandomForest(). While random forest aims to minimise variance and overfitting through the bagging method, XGBoost uses boosting which involves training an ensemble of shallow decision trees. XGBoost takes this one step further and uses a gradient descent algorithm.

Kernal PCA with optimised components Grid Search CV and XGBoost

Then I decided to try Kernal PCA – I created a reconstruction error plot to help choose the number of components and settled on six.

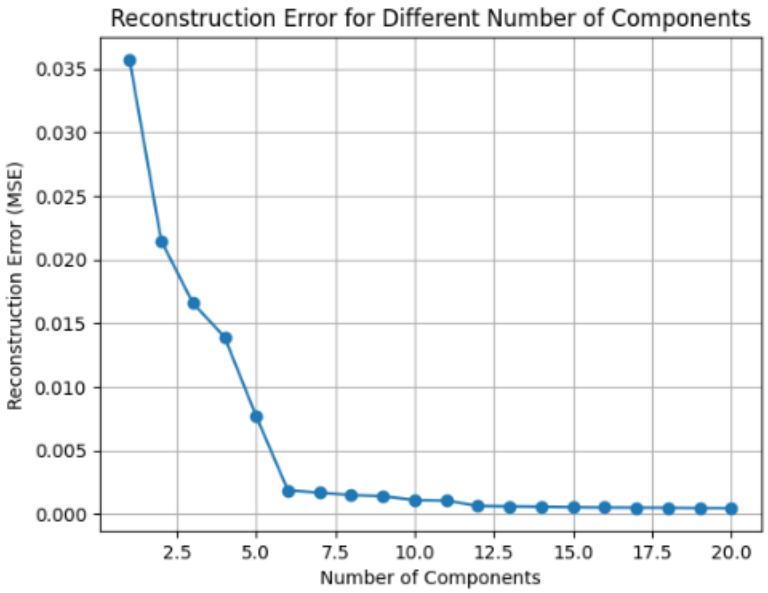

Fig 29 *Component Selection for Kernel PCA*

I then implemented kernel PCA on the training data in the hope that it would improve results and paired it with my XGBoost model. Unfortunately, it returned poor MSE results.

Test Results and Discussion

The metrics were very poor for my models in this section, and I decided to focus on the reasons why in my analysis over the actual results;

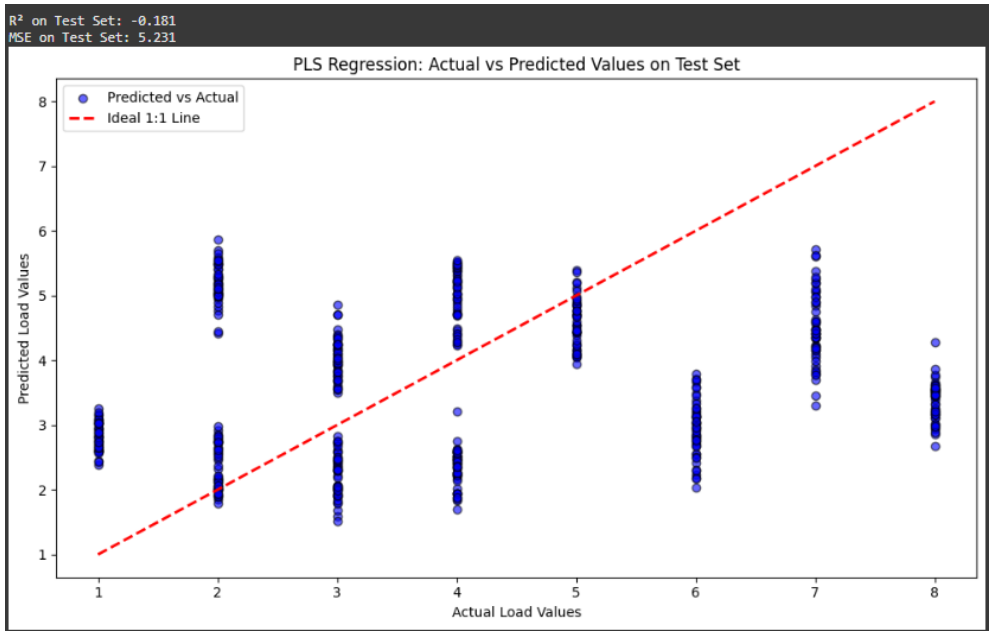

Fig 30 *Test Results for PLS regression*

The figure above shows the plotting of the actual vs the predicted scores for PLS. Lasso and other linear models all had similar results. While these models may have a better Mean Squared Error (MSE) overall due to their tendency to predict values close to the mean (3-5 range), they lack the capacity to accurately model extreme load values. This suggests that the linear models are primarily fitting the central tendency of the data rather than capturing the full variability and the underlying non-linear relationships in the data.

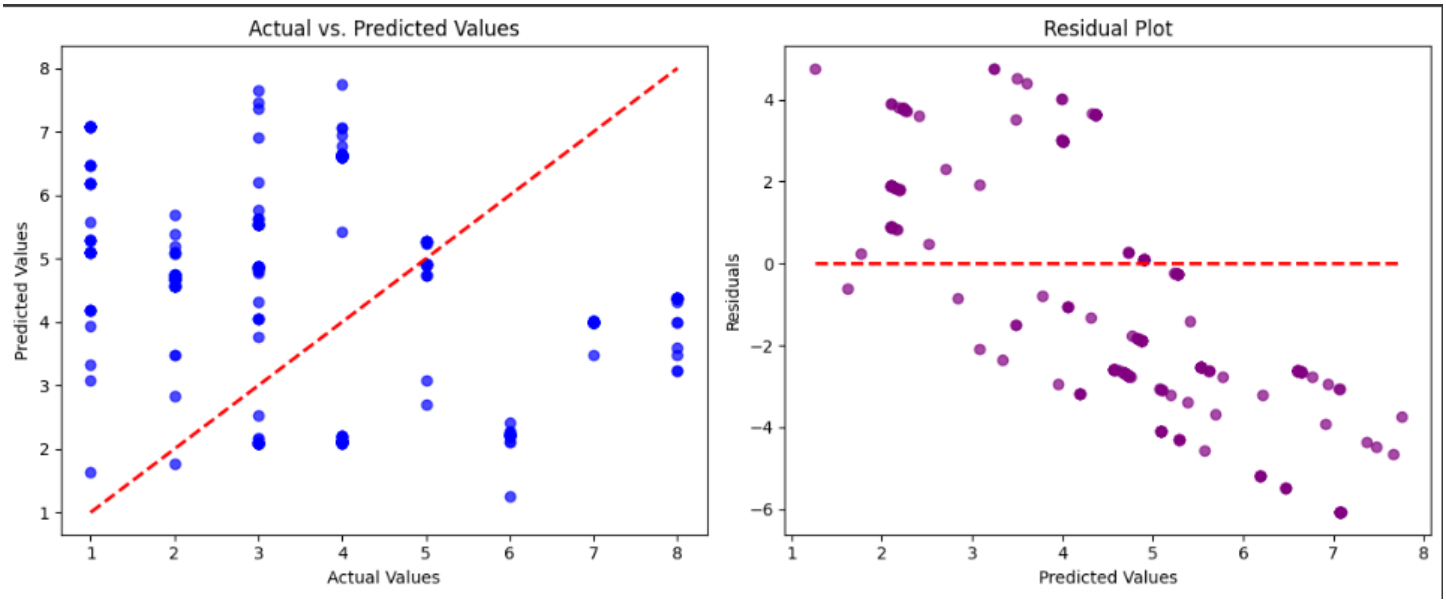

Fig 31 *Residual and Test Result splot*

The XGBoost model with Kernel PCA and other complex models shows negative R2 scores and high MSE, which initially suggests poor performance. However, closer inspection reveals that these models are trying to capture a wider range of predictions, including extreme values, rather than just predicting the mean like linear models. This indicates non-linear relationships between features and the target, which linear models fail to capture. The residual plot further supports this, showing a spread of predictions that, although inaccurate, suggests the models are attempting to map complex patterns in the data.

Takeaways

Despite employing sophisticated techniques like Kernel PCA and tree-based models, the results indicate that there might be limited correlation between the predictors and the viral load. This weak relationship could be due to inherent noise in the data, insufficient feature representation, or potential issues in the data preprocessing phase. For instance, the skewed distribution of features or the imbalance between different setups might be leading to inconsistent model performance. Given more time, some areas I would look further into are addressing potential data issues like distribution imbalances and gathering more data would be extremely beneficial.

In conclusion, while the complex models showed promise in exploring the non-linear nature of the problem, their overall performance raises the question of is it possible to predict the viral load using just the intensity readings. My synopsis would be that it is not the best indicator.

Clustering

As clustering is an unsupervised learning technique there was no need to have the classic train/test split of my data. However, as I wanted to analyse the stability of my predictions, I did withhold a portion of the data to use as unseen data to cluster. I did this by removing the last 30% of the data to avoid data leakage and present completely unseen data. As PCA is highly sensitive to the scale of features the next step was to use StandardScaler() to scale the data to a mean of 0 and standard deviation of 1. I followed the same smoothing technique as before and split it by matrix.

For this section I employed an unsupervised preprocessing technique using Principal Component Analysis due to the high correlation and the significant amount of multicollinearity between the wavelength predictors which was inflating the variance. To combat this, I created new orthogonal features using PCA, resulting in zero correlation between them and therefore reducing the number of features and variance too.

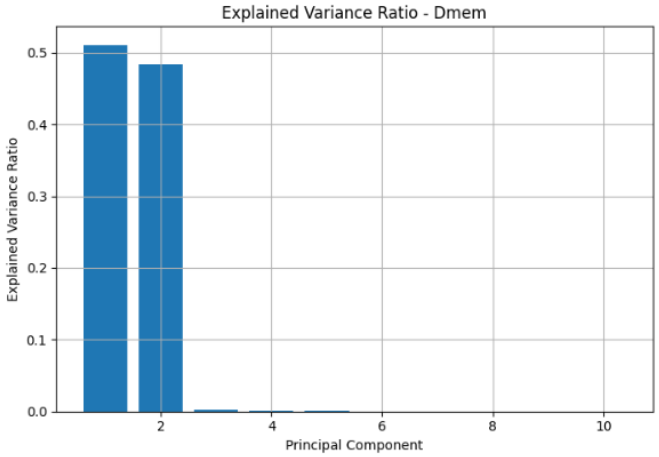

Fig 32 *Cumulative Explained Variance of Dmem 10 Principal Components*

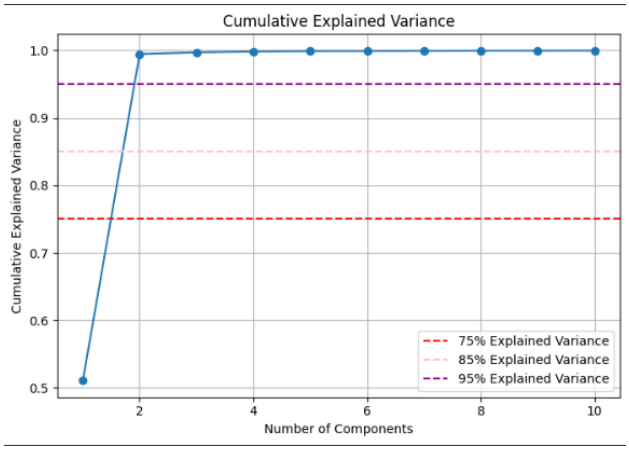

Fig 33 *Explained Variance Ratio of Dmem 10 Principal Components*

```
Total Variance Covered by the first 2 components for Matrix:: 0.9946
```

Fig 34 *Total Variance covered*

As shown in the figures, the first two principal components explained 99.5% of the total variance, reducing the data's dimensionality from 512 to 2-D. Adding more components contributed minimally to the explained variance. However, this reduction may oversimplify the data and potentially lose valuable insights.

I applied the same process to the Pbs data.

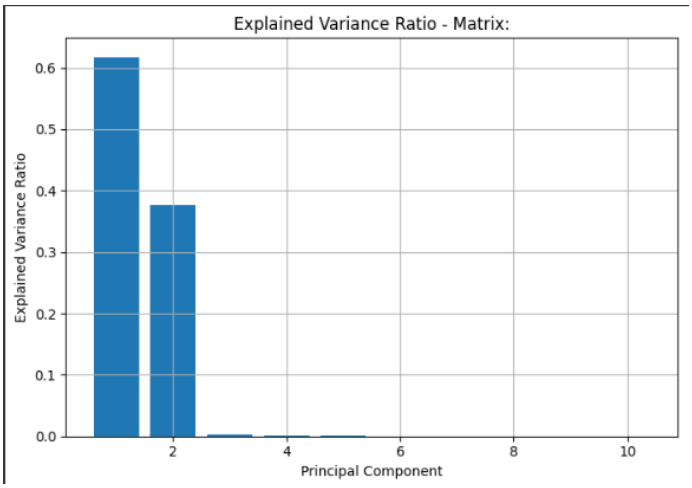

Fig 35 *Cumulative Explained Variance of pbs10 Principal Components*

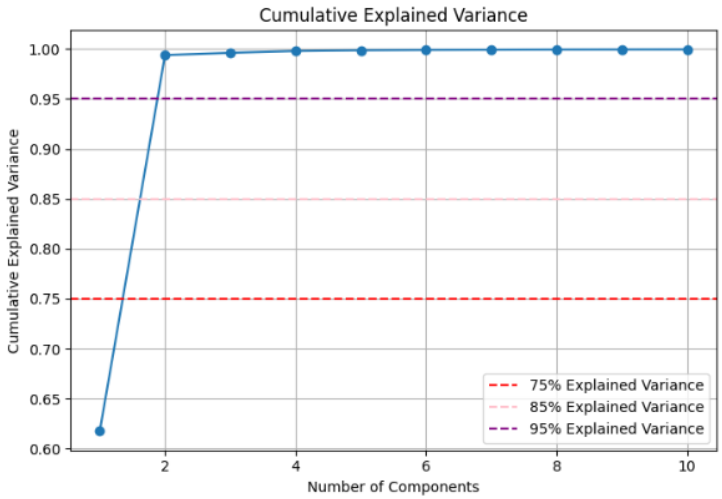

Fig 36 *Explained Variance Ratio of pbs 10 Principal Components*

```
Total Variance Covered by the first 2 components for Matrix:: 0.9939
```

Fig 37 *Total Variance covered*

The pbs data resulted in very similar results and I also selected just two components here based on the cumulative explained variance plot above.

To get a better understanding of my features I decided to analyse the loading values of the features in these components in greater detail. On closer inspection it seemed that all the feature columns seem to contribute equally with no columns dominating the loading values and makeup of the components. Each feature had a low loading value showing that the model may be lacking significant patterns to distinguish between readings.

Kmeans Clustering

The first model I selected for my analysis was the simple K-means clustering. To get the most effective k-means model I needed to find the optimal number of clusters (K). I implemented a manual grid search over a potential number of clusters ranging from 2-10 and evaluated cluster performance using two key metrics – Silhouette Score and Elbow method which plots the Within-Cluster Sum of Squares scores.

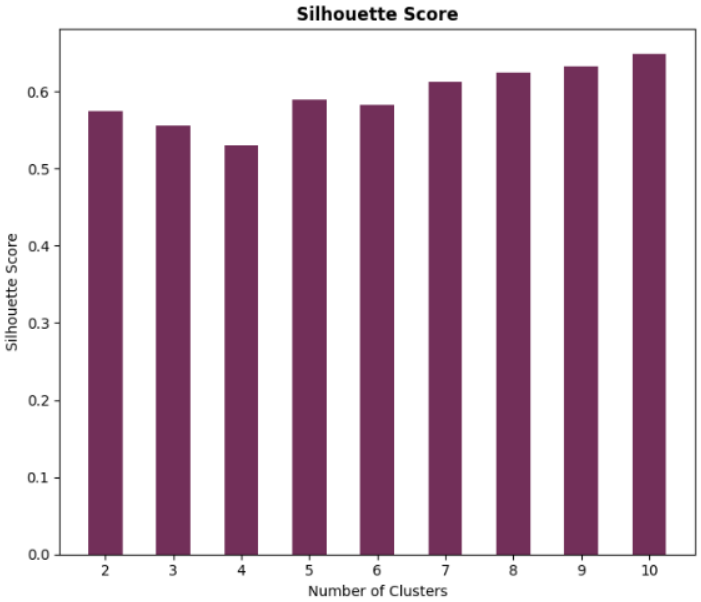

Fig 38 Silhouette Score for Number of clusters

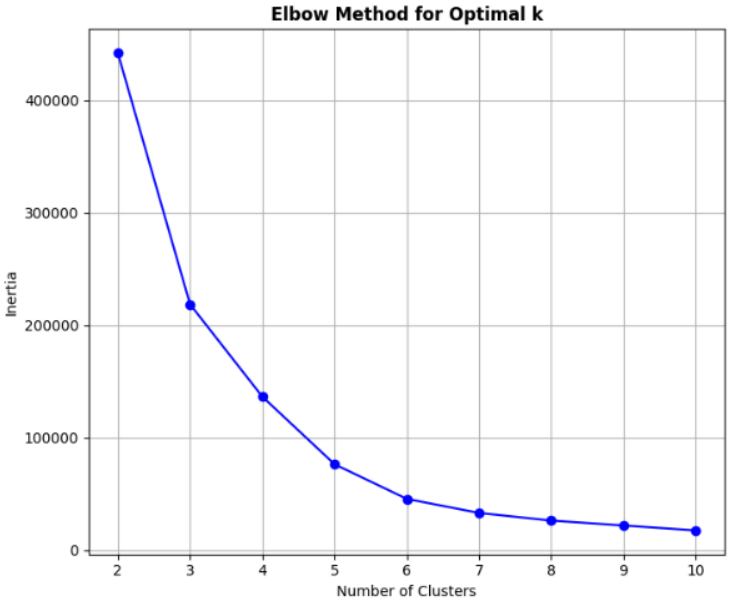

Fig 39 Elbow Method for Number of clusters

The plots above show the process of selecting the optimal K. Since K-means clustering depends on the initial random assignment of clusters, I ran the algorithm over 20 iterations to avoid getting stuck in a local minimum. However, the model performed poorly and failed to cluster based on load, prompting me to try a different method for improved performance.

Spectral Clustering

I chose spectral clustering for a number of different reasons. It is able to handle noisy data better than Kmeans but mainly it can deal with clusters with a non-linear shape, and I wanted to examine if this would improve performance. This clustering algorithm considers the relationship between points which makes it more effective at identifying clusters with a more complex shape.

I selected KNN as the parameter for my spectral clustering and fitted on my PCA components. My hope was that unlike Kmeans, using KNN to create the similarity graph would help identify nonconvex clusters. I iteratively checked using silhouette score to assess the number of clusters.

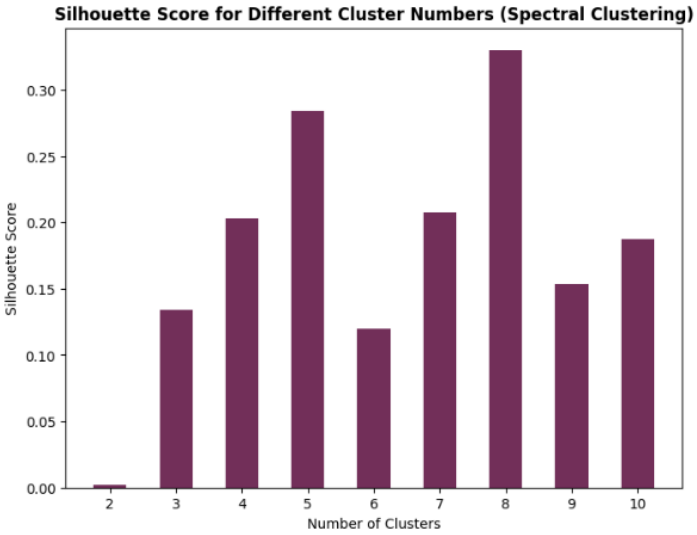

Fig 39 Silhouette Score for Number of clusters

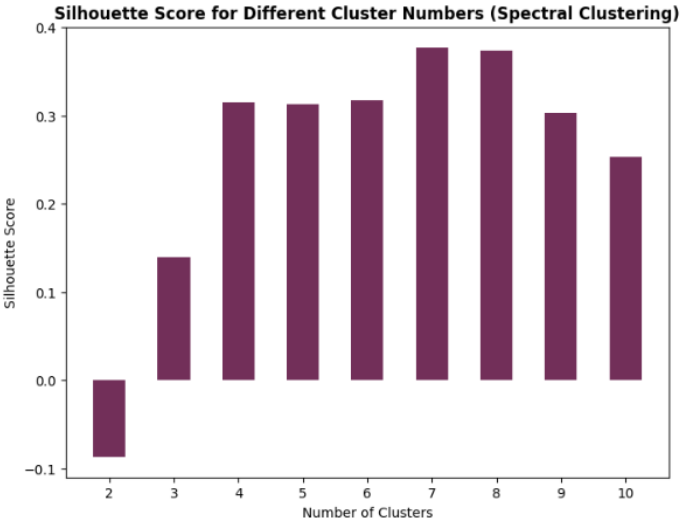

Fig 40 Silhouette Score for Number of clusters

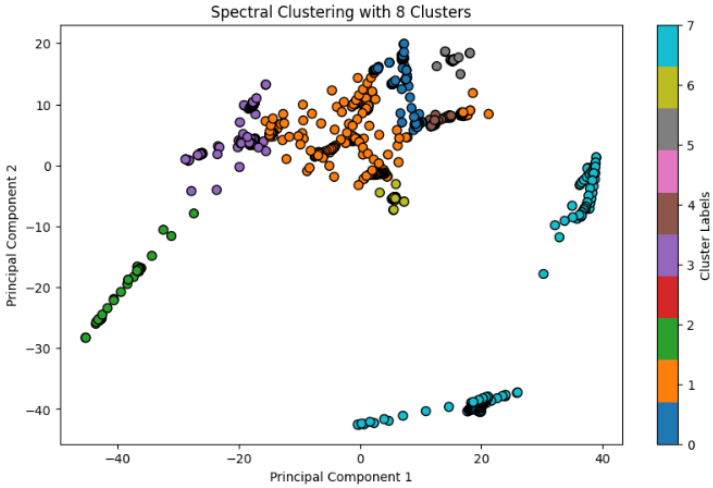

Fig 41 *Spectral clustering of dmem - 8 Clusters*

Fig 42 *Spectral clustering of pbs 7 clusters*

Dmem Data	K-Means (MSE)	Spectral Data (MSE)
Seen Data	2.9384	2.884
Unseen Data	6.1115	6.538

Table 4

Pbs Data	K-Means (MSE)	Spectral Data (MSE)
Seen Data	3.3476	3.3453
Unseen Data	2.8930	3.1298

Table 5

There were a number of interesting observations to be made regarding the results. The mean value of all the clusters fluctuates around three to four as shown in fig 43.

```
Mean viral load for each cluster:
 Cluster
0    4.639394
1    3.500000
2    4.101911
3    1.000000
4    2.622517
5    2.000000
6    5.177489
```

Fig 43

The kMeans algorithm identified 5 clusters for the dmem data, while spectral clustering split into eight clusters, reflecting the ordinal load labels. Poor MSE results on the test set, which contained many extreme label values (7 or 8), suggest that the cluster means are centered around values 4-5, leading to suboptimal predictions. Regardless of the algorithm used, there doesn't appear to be an improvement in results.

This analysis suggests that the data may not naturally separate into distinct groups based on viral loads, with clustering likely influenced more by SID setups than load values. Additionally, PCA may be missing key trends by reducing the data to two components. Kernel PCA could be a better approach to capture non-linear relationships. Overall, the dataset, with varying reflectance readings at different wavelengths, does not separate into distinct clusters based solely on load.

Conclusion

This project evaluated a range of machine learning techniques, including regression, classification, and clustering, to predict viral load and detect virus presence using spectral data. A significant challenge was mitigating data leakage, due to the sequential and highly correlated nature of the dataset, which needed a careful train-test split—though it may have introduced issues with the test set. Linear models, such as PLS and Lasso, predominantly predicted values close to the mean, while more complex models, including XGBoost with Kernel PCA, struggled to capture non-linear relationships using the high-dimensional, noisy data. Clustering analysis revealed that the data grouped more by experimental setup (SID) than by viral load. Overall, the study underscored the complexity of the dataset and the importance of further feature engineering, prioritizing exploration over performance optimisation..

Word Count Excluding Front Page, Titles, Sub-Headings, Tables, Figures, References and Code -

3288 Words

References

Huang, J.-D., Wang, H., Power, U., McLaughlin, J.A., Nugent, C., Rahman, E., Barabas, J. & Maguire, P., 2024. Detecting Respiratory Viruses Using a Portable NIR Spectrometer—A Preliminary Exploration with a Data Driven Approach. *Sensors*, 24(1), p. 308. Available at: https://doi.org/10.3390/s24010308.

Works CitedBorhani, Jeremy. MACHINE LEARNING REFINED: Foundations, Algorithms,and Applications. S.L., Cambridge Univ Press, 2019.

Brownlee, Jason. “Nested Cross-Validation for Machine Learning with Python.” Machine Learning Mastery, 28 July 2020, machinelearningmastery.com/nested-cross-validation-for-machine-learning-with-python/.

Cross. “Cross Validation with Replicates for Spectral Analysis.” Cross Validated, 22 June 2017, stats.stackexchange.com/questions/286698/cross-validation-with-replicates-for-spectral-analysis. Accessed 2 Nov. 2024.

Goodfellow, Ian, et al. Deep Learning. MIT Press, 10 Nov. 2016.How. “How to Choose a Kernel for Kernel PCA?” Cross Validated, 4 Jan. 2015, stats.stackexchange.com/questions/131142/how-to-choose-a-kernel-for-kernel-pca.

Jain, Rahul. “Implementing Spectral Clustering from Scratch: A Step-By-Step Guide.” Medium, Medium, 23 May 2024, rahuljain788.medium.com/implementing-spectral-clustering-from-scratch-a-step-by-step-guide-9643e4836a76

.James, Gareth, et al. An Introduction to Statistical Learning. Springer Nature, 1 Aug. 2023.

Nvidia. “What Is XGBoost?” NVIDIA Data Science Glossary, 2024, www.nvidia.com/en-us/glossary/xgboost/.

Pelliccia, Daniel. “NIRPY Research • Statistical Learning and Chemometrics in Python.” NIRPY Research, 2019, nirpyresearch.com/.

Scikit-learn. “Scikit-Learn: Machine Learning in Python.” Scikit-Learn.org, 2019, scikit-learn.org/stable/.

Shin, Terence. “Top Three Clustering Algorithms You Should Know instead of K-Means Clustering.” Medium, 12 Dec. 2022, terenceshin.medium.com/top-five-clustering-algorithms-you-should-know-instead-of-k-means-clustering-b22f25e5bfb4.

Code – Only included Dmem Code as PBS code is just the same but with pbs data

```
# -*- coding: utf-8 -*-

"""8051 (1).ipynb
```

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1oDzeG64Ht02b5kSsMQz64fMgOtx9V1Wm

Classification

Loading in Data and imports

"""

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')

file_path = '/content/drive/MyDrive/Queens/Module2/2022QUB.xlsx'
df = pd.read_excel(file_path, sheet_name='Data')

from scipy import stats
import numpy as np
import seaborn as sns
import sys
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler,StandardScaler
from scipy.signal import savgol_filter
from sklearn.model_selection import StratifiedKFold, cross_validate,  KFold, cross_val_predict, train_test_split, cross_val_score
from sklearn.metrics import make_scorer, balanced_accuracy_score, f1_score, precision_score, recall_score, accuracy_score, roc_curve, auc, classification_report, confusion_matrix, roc_auc_score
from sklearn.svm import SVC, LinearSVC
from sklearn.linear_model import LogisticRegression,SGDClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
```

```
from sklearn.cross_decomposition import PLSRegression
from sklearn.decomposition import PCA, KernelPCA
from sklearn.tree import DecisionTreeClassifier
from imblearn.over_sampling import SMOTE
from sklearn.feature_selection import RFE
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import VarianceThreshold
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

df

df= df.applymap(lambda x: x.strip() if isinstance(x, str) else x) #Here I am stripping leading and training whitespaces
print("\nCleaned DataFrame:")

df = df.drop(['SID',"Load"], axis=1)  # I am dropping irrelevant columns as they dont contribute to my model

print("\nDataFrame after dropping 'SID':")
print(df)

"""# Exploration and Removal of outliers"""

df.describe()

df.count().sort_values()

z = np.abs(stats.zscore(df._get_numeric_data())) #Converting data to its z-score so that it can be used to detect outliers
print(z)

df= df[(z < 3).all(axis=1)] #Here I am removing any rows outside the threshold of 3 std of the mean
print(df.shape)
```

```
df.reset_index(drop=True, inplace=True)

df

"""## Correlation"""

subset_columns = [f'I{str(i).zfill(3)}' for i in range(100, 111)]
subset_df = df[subset_columns]



sns.pairplot(subset_df,
             kind="scatter",
             diag_kind="kde",
             plot_kws={'alpha':0.5, 's':20},
             diag_kws={'shade':True})

plt.suptitle("Distribution and Correlation of I100 to I110", y=1.02, fontsize=16)
plt.show()

"""# Data Preprocessing"""

df

intensity_columns = df.columns[2:] # I am extracting the wavelength columsn

scaler = StandardScaler()  # I am normalising the data
normalised_data = scaler.fit_transform(df[intensity_columns])

normalised_df = pd.DataFrame(normalised_data, columns=intensity_columns)
window_length = 13 # Here I define a window length and polynomial order for Savitzky-Golay smoothing which must be odd and greater than the polynomial order
```

```
poly_order = 2


smoothed_data = {
    col: savgol_filter(normalised_df[col], window_length, poly_order)
    for col in normalised_df.columns
}


# I apply Savitzky-Golay filter to each column - I chose window length 13 and poly order 2 after a trial and error process
smoothed_df = pd.DataFrame(smoothed_data, columns=intensity_columns)
print(smoothed_df.head())



# Here I am defining a function to plot original, normalised, and smoothed data and compare their distributions to see the effect of the preprocessing on the data
def plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_column):
    plt.figure(figsize=(18, 6))


    plt.subplot(1, 3, 1)
    plt.title(f'Original Intensity Data ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(df[df['Type'] == sample_type].index, df[df['Type'] == sample_type][intensity_column],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Intensity')


    plt.subplot(1, 3, 2)
    plt.title(f'StandardScaler Normalised ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(normalised_df.index[df['Type'] == sample_type],
                 normalised_df[intensity_column][df['Type'] == sample_type],
                 label=sample_type, marker='o', linestyle='-')
```

```
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Normalised Intensity')

    plt.subplot(1, 3, 3)
    plt.title(f'SG Smoothed ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(smoothed_df.index[df['Type'] == sample_type],
                 smoothed_df[intensity_column][df['Type'] == sample_type],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Smoothed Intensity')

    plt.tight_layout()
    plt.show()

for intensity_col in ['I001', 'I101', 'I201', 'I301', 'I401', 'I501']:
    plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_col)

other_columns = df.drop(df.columns[df.columns.str.startswith('I')], axis=1) # Get out the other columsn to add back

final_df = pd.concat([other_columns,smoothed_df], axis=1) #I combine with standarised data
print(final_df.head())

final_df.replace({'Type': {'X': 0, 'Y': 1}}, inplace=True) #Here I am replacing teh target variables so taht they are compatable with the different models
print("Unique values in 'Type' after encoding:", df['Type'].unique())

print(final_df)

df["Type"].value_counts() # can see here that there is a class imbalance that I must account for
```

```
dmem_train = final_df[final_df['Matrix'] == 'dmem'].reset_index(drop=True)  # Split my data based on matrix as otherwise the clustering would make no sense

pbs_train= final_df[final_df['Matrix'] == 'pbs'].reset_index(drop=True)


print("DataFrame for dmem:")

print(dmem_train)

print("\nDataFrame for pbs:")

print(pbs_train)


#In order to get some X and Y target instead of just Y I had to go with a 70/30 splity which isnt ideal as I lose training data but otherise I wouldve had only 1 class

X_dmem = dmem_train.drop(['Type', 'Matrix'], axis=1)

y_dmem = dmem_train['Type']

split_index_dmem = int(0.70 * len(X_dmem))


X_dmem_train, X_dmem_test = X_dmem[:split_index_dmem], X_dmem[split_index_dmem:]

y_dmem_train, y_dmem_test = y_dmem[:split_index_dmem], y_dmem[split_index_dmem:]


X_pbs = pbs_train.drop(['Type', 'Matrix'], axis=1)

y_pbs = pbs_train['Type']


split_index_pbs = int(0.70 * len(X_pbs))


X_pbs_train, X_pbs_test = X_pbs[:split_index_pbs], X_pbs[split_index_pbs:]

y_pbs_train, y_pbs_test = y_pbs[:split_index_pbs], y_pbs[split_index_pbs:]


plt.figure(figsize=(12, 8)) # Essentially I am taking SID 4 as my test set


X_dmem_train_df = pd.DataFrame(X_dmem_train, columns=X_dmem.columns)

X_dmem_test_df = pd.DataFrame(X_dmem_test, columns=X_dmem.columns)

for col in X_dmem_train_df.columns:
```

```
    plt.plot(X_dmem_train_df.index, X_dmem_train_df[col], label=f"Train - {col}", color='blue', alpha=0.6)
for col in X_dmem_test_df.columns:
    plt.plot(X_dmem_test_df.index, X_dmem_test_df[col], label=f"Test - {col}", color='red', alpha=0.6)


plt.xlabel("Index")
plt.ylabel("Scaled Intensity")
plt.title("Training and Test Data Split for dmem_train (Blue = Train, Red = Test)")
plt.axhline(0, color='grey', linestyle='--', linewidth=0.8)
plt.grid(True)


plt.tight_layout()
plt.show()


X_pbs_train_df = pd.DataFrame(X_pbs_train, columns=X_pbs.columns)
X_pbs_test_df = pd.DataFrame(X_pbs_test, columns=X_pbs.columns)


plt.figure(figsize=(12, 8))


for col in X_pbs_train_df.columns:
    plt.plot(X_pbs_train_df.index, X_pbs_train_df[col], color='blue', alpha=0.6)


for col in X_pbs_test_df.columns:
    plt.plot(X_pbs_test_df.index, X_pbs_test_df[col], color='red', alpha=0.6)


plt.xlabel("Index")
plt.ylabel("Scaled Intensity")
plt.title("Training and Test Data Split for pbs_train (Blue = Train, Red = Test)")
plt.axhline(0, color='grey', linestyle='--', linewidth=0.8)
plt.grid(True)


plt.tight_layout()
plt.show()
```

```
"""# Dmem


## Basic models with no feature selection or hyperparameter tuning for DMEM
"""


def evaluate_model(model, X, y, model_name, n_splits=10): #This is a function I iwll use to implement CV and asses training vs validation metrics


    ls_metrics = {
        "balanced_accuracy": "balanced_accuracy", # This gives equal weight to the performance of my model in all classes
        "f1_weighted": "f1_weighted",  # To asses fit
        "precision_weighted": "precision_weighted",
        "recall_weighted": "recall_weighted"
    }


    cv = StratifiedKFold(n_splits=n_splits,shuffle=False) # I used Stratified K-fold cv here to try account for ths class imablance - set shuffle = False to avoid data leakage


    results = cross_validate(model, X, y, cv=cv, return_train_score=True, scoring=ls_metrics) #Here I implement the CV


#Have to create lists for my metrics
    all_results = {
        model_name: {
            'train_balanced_accuracy': results['train_balanced_accuracy'],
            'test_balanced_accuracy': results['test_balanced_accuracy'],
            'train_f1_score_weighted': results['train_f1_weighted'],
            'test_f1_score_weighted': results['test_f1_weighted'],
            'train_precision_weighted': results['train_precision_weighted'],
            'test_precision_weighted': results['test_precision_weighted'],
            'train_recall_weighted': results['train_recall_weighted'],
            'test_recall_weighted': results['test_recall_weighted'],
        }
```

```
    }

    # I need to get the average of each metric over the 10 folds. Then I print
    average_results = {
        model_name: {
            metric: np.mean(scores) for metric, scores in all_results[model_name].items()
        }
    }
    for model_name, scores in average_results.items():
        print(f"{model_name} - Training Balanced Accuracy: {scores['train_balanced_accuracy']:.3f}, "
              f"Validation Balanced Accuracy: {scores['test_balanced_accuracy']:.3f}, "
              f"Training F1 Score: {scores['train_f1_score_weighted']:.3f}, "
              f"Validation F1 Score: {scores['test_f1_score_weighted']:.3f}, "
              f"Training Precision: {scores['train_precision_weighted']:.3f}, "
              f"Validation Precision: {scores['test_precision_weighted']:.3f}, "
              f"Training Recall: {scores['train_recall_weighted']:.3f}, "
              f"Validation Recall: {scores['test_recall_weighted']:.3f}")

#This is then me preparing the data for plotting by getting the scpre at each split
    metrics_labels = ['balanced_accuracy', 'f1_weighted', 'precision_weighted', 'recall_weighted']
    x = np.arange(n_splits)

    fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))
    for i, metric in enumerate(metrics_labels):
        train_metric_key = f'train_{metric}'
        test_metric_key = f'test_{metric}'

        # This plotting teachnique was one I found online to help me visualise the way I want
        ax[i // 2, i % 2].plot(x, results[train_metric_key], label='Training', marker='o')
        ax[i // 2, i % 2].plot(x, results[test_metric_key], label='Validation', marker='o')

        ax[i // 2, i % 2].set_xticks(x)
```

```
        ax[i // 2, i % 2].set_xticklabels([f'Fold {j + 1}' for j in x])
        ax[i // 2, i % 2].set_title(f'Training vs Validation {metric.replace("_", " ").title()}')
        ax[i // 2, i % 2].set_ylim(0, 1)
        ax[i // 2, i % 2].legend()
        ax[i // 2, i % 2].grid()

    plt.tight_layout()
    plt.show()

lr_model = LogisticRegression(max_iter = 10000) #Here I am calling a base Logistic function and tetsing use CV above
evaluate_model(lr_model, X_dmem_train, y_dmem_train, "Logistic Regression", n_splits=10)

#Then I applied the same analysis to a support vector classifier
svc_model = SVC(probability=True, max_iter=10000)
evaluate_model(svc_model, X_dmem_train, y_dmem_train, "SVC", n_splits=10)

#Here I wanted to analuse if there was roiughly the same class imbalnce between tarining and test
unique, counts = np.unique(y_dmem_train, return_counts=True)
print("Training data:",dict(zip(unique, counts)))

unique, counts = np.unique(y_dmem_test, return_counts=True)
print("Test data:",dict(zip(unique, counts)))

"""## PCA



"""

pca = PCA(n_components=5)
X_dmem_train_pca = pca.fit_transform(X_dmem_train) #This is me just perfomring PCA on my data with 5 components
pca_df = pd.DataFrame(data=X_dmem_train_pca, columns=['PC1', 'PC2','PC3', 'PC4','PC5'])
```

```
pca_df['Type'] = y_dmem_train #to visiualise by colour


plt.figure(figsize=(8, 6))
sns.scatterplot(data=pca_df, x='PC1', y='PC2', hue='Type', style='Type', palette='viridis', alpha=0.7)


plt.title('PCA of Dmem Training Data Colored by Type')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid()
plt.tight_layout()
plt.legend(title='Type')
plt.show()


explained_variance = pca.explained_variance_ratio_
print(f'Explained variance by each principal component: {explained_variance}')
print(f'Total explained variance: {sum(explained_variance)}')


X_dmem_test_pca = pca.transform(X_dmem_test) # As I transformed the Training data I need to tranform the Test too


#This cell was simply just experimental, as I wanted to examine if the PCA data was performing okay
clf = LogisticRegression(class_weight='balanced')
clf.fit(X_dmem_train_pca, y_dmem_train)
y_pred = clf.predict(X_dmem_test_pca)
score = accuracy_score(y_dmem_test, y_pred)
f1 = f1_score(y_dmem_test, y_pred, average='weighted')
print('Logistic Regression - Accuracy:', score)
print('Logistic Regression - F1 score:', f1)


clf_svc = LinearSVC(class_weight='balanced')
clf_svc.fit(X_dmem_train_pca, y_dmem_train)
y_pred = clf_svc.predict(X_dmem_test_pca)
score = accuracy_score(y_dmem_test, y_pred)
```

```
f1 = f1_score(y_dmem_test, y_pred, average='weighted')
print('Linear SVC - Accuracy:', score)
print('Linear SVC - F1 score:', f1)

clf_dt = DecisionTreeClassifier(random_state=0, class_weight='balanced')
clf_dt.fit(X_dmem_train_pca, y_dmem_train)
y_pred = clf_dt.predict(X_dmem_test_pca)
score = accuracy_score(y_dmem_test, y_pred)
f1 = f1_score(y_dmem_test, y_pred, average='weighted')
print('Decision Tree - Accuracy:', score)
print('Decision Tree - F1 score:', f1)

clf_sgd = SGDClassifier(random_state=42, max_iter=1000, tol=1e-3, class_weight='balanced')
clf_sgd.fit(X_dmem_train_pca, y_dmem_train)
y_pred = clf_sgd.predict(X_dmem_test_pca)
score = accuracy_score(y_dmem_test, y_pred)
f1 = f1_score(y_dmem_test, y_pred, average='weighted')
print('SGD Classifier - Accuracy:', score)
print('SGD Classifier - F1 score:', f1)

"""## PLS

Due to the nature of PLS and the fact it interacts with the target ariable too, I felt it could be an effective way too classify the data using PLS-DA
"""

# I use `[0]` to extract only the transformed predictor variables (X_scores) because `fit_transform`
# returns a tuple where the first element is X_scores and the second is Y_scores.
pls_binary =PLSRegression(n_components=10)
X_pls = pls_binary.fit_transform(X_dmem_train, y_dmem_train)[0]

X_dmem_test.shape
```

```
pls_binary = PLSRegression(n_components=10)
# Fit on my the training set
pls_binary.fit(X_dmem_train, y_dmem_train)

y_pred = pls_binary.predict(X_dmem_test)
# Here I "Force" binary prediction by thresholding
binary_prediction = (y_pred > 0.5).astype('uint8')
print(binary_prediction)
print(y_dmem_test)

"""Basic PLSDA"""

#I initialise a PLS regression model with 4 components and fit it to my training data.
plsr = PLSRegression(n_components=4, scale=False)
plsr.fit(X_dmem_train, y_dmem_train)

#I extract the PLS scores (X_scores) for visualisation.
X_scores = plsr.x_scores_

#I create a scatter plot of the first two PLS scores, using different colors for each unique class in `y_dmem_train`.
plt.figure(figsize=(10, 6))
colors = plt.cm.jet(np.linspace(0, 1, len(np.unique(y_dmem_train))))

for idx, label in enumerate(np.unique(y_dmem_train)):
    plt.scatter(X_scores[y_dmem_train == label, 0], X_scores[y_dmem_train == label, 1],
                color=colors[idx], label=str(label), edgecolors='k', s=100)

plt.title('PLS-DA: Scores Plot')
plt.xlabel('Latent Variable 1')
plt.ylabel('Latent Variable 2')
plt.legend(title='Class Labels')
plt.grid(True)
```

```
plt.show()


def pls_da(X_train, y_train, X_test): # To make my code neater just made a function


    plsda = PLSRegression(n_components=4)
    plsda.fit(X_train, y_train)
    binary_prediction = (plsda.predict(X_test) > 0.5).astype('uint8')


    return binary_prediction


#Lists for my metrics
accuracy = []
fpr_list = []
tpr_list = []
roc_auc_list = []


cval = StratifiedKFold(n_splits=10, shuffle=True, random_state=19)
X_train_np = X_dmem_train.to_numpy() # Needed this so that I could use indexing for nested cross validation
y_train_np = y_dmem_train.to_numpy()


# Implement my CV
for train_idx, test_idx in cval.split(X_train_np, y_train_np):
  #Train and test for each fold
    X_train_fold, X_test_fold = X_train_np[train_idx, :], X_train_np[test_idx, :]
    y_train_fold, y_test_fold = y_train_np[train_idx], y_train_np[test_idx]


    # Uisng my PLS_DA function defined above
    y_pred_proba = pls_da(X_train_fold, y_train_fold, X_test_fold)  # This should give probabilities


    # Calculate accuracy for the current fold
    accuracy.append(accuracy_score(y_test_fold, y_pred_proba))
    fpr, tpr, _ = roc_curve(y_test_fold, y_pred_proba)
```

```
    roc_auc = auc(fpr, tpr)
    fpr_list.append(fpr)
    tpr_list.append(tpr)
    roc_auc_list.append(roc_auc)

# I print the average accuracy across all folds
print("Average accuracy on 10 splits: ", np.array(accuracy).mean())

# Plot my ROC curve
plt.figure(figsize=(10, 6))
for i in range(len(fpr_list)):
    plt.plot(fpr_list[i], tpr_list[i], label=f'Fold {i + 1} (AUC = {roc_auc_list[i]:.2f})')

# Plotting my average ROC curve to show overall performance
mean_fpr = np.linspace(0, 1, 100)
mean_tpr = np.zeros_like(mean_fpr)
#Need to calulate mean for True Positive rate
for i in range(len(fpr_list)):
    mean_tpr += np.interp(mean_fpr, fpr_list[i], tpr_list[i])

mean_tpr /= len(fpr_list)
mean_auc = auc(mean_fpr, mean_tpr)

plt.plot(mean_fpr, mean_tpr, color='b', linestyle='--',
        label=f'Mean ROC (AUC = {mean_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
```

```
plt.show()


#This is applying the same thing as above but testing on my test set
y_pred_test_proba = pls_da(X_train_np, y_train_np, X_dmem_test.to_numpy())
test_accuracy = accuracy_score(y_dmem_test.to_numpy(), (y_pred_test_proba > 0.5).astype('uint8'))
print("Test set accuracy: ", test_accuracy)


print("\nClassification Report:")
print(classification_report(y_dmem_test.to_numpy(), (y_pred_test_proba > 0.5).astype('uint8')))
fpr, tpr, thresholds = roc_curve(y_dmem_test.to_numpy(), y_pred_test_proba)
roc_auc = auc(fpr, tpr)


plt.figure(figsize=(10, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc='lower right')
plt.show()


y_pred_test = (y_pred_test_proba > 0.5).astype('uint8')
conf_matrix = confusion_matrix(y_dmem_test.to_numpy(), y_pred_test)


plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='g', cmap='Blues', cbar=False,
        xticklabels=['Predicted Negative', 'Predicted Positive'],
        yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
```

```
plt.title('Confusion Matrix')
plt.show()


#test set accuracy is solid but there is a clear issue with precision on the 0 (x) class


"""## Logistic with L1


Next I wanted to examine another feature extraction method by examining regualrisation. Using Lasso, it will put some of the features to zero which makes it usitable for my task over ridge
"""


#This is the parameter that tunes the intensity of my Lasso regualrisatin.
C_values = [0.1, 0.5, 1, 10]


cv_accuracies = []
cv_aucs = []
cv_f1_scores = []


# wanted to perform CV using stratified CV to examone the performance across a range of C values - I look at Accuracy. F1 and AUC
for C in C_values:
    log_reg = LogisticRegression(C=C, penalty='l1', solver='saga', max_iter=1000, random_state=42)
    cv_accuracy = cross_val_score(log_reg, X_dmem_train, y_dmem_train, cv=StratifiedKFold(n_splits=10), scoring='accuracy')
    cv_accuracies.append(cv_accuracy)
    cv_auc = cross_val_score(log_reg, X_dmem_train, y_dmem_train, cv=StratifiedKFold(n_splits=10), scoring='roc_auc')
    cv_aucs.append(cv_auc)
    cv_f1 = cross_val_score(log_reg, X_dmem_train, y_dmem_train, cv=StratifiedKFold(n_splits=10), scoring='f1')
    cv_f1_scores.append(cv_f1)


# Get mean validation scores
mean_accuracy = [np.mean(cv_acc) for cv_acc in cv_accuracies]
mean_auc = [np.mean(cv_auc) for cv_auc in cv_aucs]
mean_f1 = [np.mean(cv_f1) for cv_f1 in cv_f1_scores]
```

```
# Print the cross-validation scores for each C value
for idx, C in enumerate(C_values):
    print(f'C = {C}')
    print(f'  Mean Accuracy (CV): {mean_accuracy[idx]:.4f}')
    print(f'  Mean AUC (CV): {mean_auc[idx]:.4f}')
    print(f'  Mean F1 Score (CV): {mean_f1[idx]:.4f}')
    print()

# It appears that a low C is too harsh and results in no parameters at all so a higher C value is more suitable

best_C = C_values[np.argmax(mean_f1)]  # I chose F1 based C so that it would ensure best fit and avoid overfitting
print(f'Best C based on F1 score: {best_C}')

# # Traun on the best C value from CV
final_log_reg = LogisticRegression(C=best_C, penalty='l1', solver='saga', max_iter=5000,random_state=42)
final_log_reg.fit(X_dmem_train, y_dmem_train)

# Evaluate final model
train_accuracy = final_log_reg.score(X_dmem_train, y_dmem_train)
train_auc = roc_auc_score(y_dmem_train, final_log_reg.predict_proba(X_dmem_train)[:, 1])
train_f1 = f1_score(y_dmem_train, final_log_reg.predict(X_dmem_train))

print(f'Final Model Training Accuracy: {train_accuracy:.4f}')
print(f'Final Model Training AUC: {train_auc:.4f}')
print(f'Final Model Training F1 Score: {train_f1:.4f}')

#Evaluate on test set
y_pred_prob = final_log_reg.predict_proba(X_dmem_test)[:, 1]
y_pred = final_log_reg.predict(X_dmem_test)

accuracy = accuracy_score(y_dmem_test, y_pred)
```

```
roc_auc = roc_auc_score(y_dmem_test, y_pred_prob)
conf_matrix = confusion_matrix(y_dmem_test, y_pred)

print(f'Accuracy on Test Set: {accuracy:.2f}')
print(f'AUC-ROC on Test Set: {roc_auc:.2f}')
print(f'Confusion Matrix:\n{conf_matrix}')

#Plot my ROC curve
fpr, tpr, thresholds = roc_curve(y_dmem_test, y_pred_prob)

roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')  # Diagonal line for random guessing
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()

"""## Introduce Support Vector Classifiers and a Kernal
Next I wanted to see if SVM and the kernal feature could perform better
"""

mean_auc_svm = []
#Same as above but for SVM
C_values = [0.1]#,0.01, 0.01, 0.001, ]

# Cross-validation to select the best C value based on AUC
```

```
for C in C_values:
    svm_classifier = SVC(C=C, kernel='linear', probability=True, random_state=42)
    scores = cross_val_score(svm_classifier, X_dmem_train, y_dmem_train, cv=StratifiedKFold(n_splits=10), scoring='roc_auc')
    mean_auc_svm.append(scores.mean())

best_C_svm = C_values[np.argmax(mean_auc_svm)]
print(f'Best C based on AUC for SVM: {best_C_svm}')

final_svm_classifier = SVC(C=best_C_svm, kernel='linear', probability=True, random_state=42)
final_svm_classifier.fit(X_dmem_train, y_dmem_train)

support_vectors = final_svm_classifier.support_vectors_
print(f'Support Vectors: {support_vectors}')

# same CV process
mean_auc_train = []
mean_auc_val = []

C_values = [0.001,0.01,0.1]

X_dmem_train_np = X_dmem_train.to_numpy()
y_dmem_train_np = y_dmem_train.to_numpy()

# StratifiedKFold for consistent splitting
cv = StratifiedKFold(n_splits=10)

for C in C_values:
    train_scores = []
    val_scores = []

    for train_idx, val_idx in cv.split(X_dmem_train_np, y_dmem_train_np):
```

```
        svm_classifier = SVC(C=C, kernel='linear', probability=True,random_state=42)
        svm_classifier.fit(X_dmem_train_np[train_idx], y_dmem_train_np[train_idx])
        train_auc = roc_auc_score(y_dmem_train_np[train_idx], svm_classifier.predict_proba(X_dmem_train_np[train_idx])[:, 1])
        val_auc = roc_auc_score(y_dmem_train_np[val_idx], svm_classifier.predict_proba(X_dmem_train_np[val_idx])[:, 1])

        train_scores.append(train_auc)
        val_scores.append(val_auc)

    mean_auc_train.append(np.mean(train_scores))
    mean_auc_val.append(np.mean(val_scores))

plt.figure(figsize=(10, 6))
plt.plot(C_values, mean_auc_train, label="Training AUC", marker='o', color='blue')
plt.plot(C_values, mean_auc_val, label="Validation AUC", marker='o', color='red')
plt.xscale('log')
plt.xlabel('C value (log scale)')
plt.ylabel('AUC Score')
plt.title('Training vs. Validation AUC Scores for SVM')
plt.legend()
plt.grid()
plt.show()

# Test set evaluation
y_pred_prob_svm = final_svm_classifier.predict_proba(X_dmem_test)[:, 1]
y_pred_svm = final_svm_classifier.predict(X_dmem_test)

accuracy_svm = accuracy_score(y_dmem_test, y_pred_svm)
roc_auc_svm = roc_auc_score(y_dmem_test, y_pred_prob_svm)
conf_matrix_svm = confusion_matrix(y_dmem_test, y_pred_svm)

# I wanted to print the evaluation metrics
print(f'Accuracy on Test Set for SVM: {accuracy_svm:.2f}')
```

```
print(f'AUC-ROC on Test Set for SVM: {roc_auc_svm:.2f}')
print(f'Confusion Matrix for SVM:\n{conf_matrix_svm}')

# Print Classification Report
print("\nClassification Report for SVM:")
print(classification_report(y_dmem_test, y_pred_svm))

fpr, tpr, thresholds = roc_curve(y_dmem_test, y_pred_prob_svm)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (area = {roc_auc_svm:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve for SVM')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# I wanted to plot a confusion matrix
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix_svm, annot=True, fmt="d", cmap="Blues", cbar=False,
        xticklabels=['Predicted Negative', 'Predicted Positive'],
        yticklabels=['Actual Negative', 'Actual Positive'])
plt.title("Confusion Matrix for SVM")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()

"""## Decision Tree with RFE
```

After disappointing results through dimensionality reduction and also regularisation I decided to try a more advanced technique using Recursive feature elimiantion which fits all and then moves backwards one by one. I knew this owuld be computationally expensive but I thought it wwould be inetersing to analyse

"""

```
def get_models(): # This is the implementation of RFE
    models = {}
    for i in range(5,300 , 20):  #Want to try speed up the process so go in steps of 20
        rfe = RFE(estimator=DecisionTreeClassifier(random_state=42), n_features_to_select=i)
        model = DecisionTreeClassifier(random_state=42)
        # I create a pipeline with RFE and the classifier
        models[f'{i} features'] = Pipeline(steps=[('feature_selection', rfe), ('classification', model)])
    return models


#To avoid overfitting I use nested cross validation to avoid overfitting
def evaluate_model_with_nested_cv(model, X, y, n_splits=10):
    skf = StratifiedKFold(n_splits=n_splits)
    outer_scores = []
    X = np.array(X)
    y = np.array(y) #For indexing

    for train_idx, val_idx in skf.split(X, y):
        X_train, X_val = X[train_idx], X[val_idx]
        y_train, y_val = y[train_idx], y[val_idx]
        model.fit(X_train, y_train)

        val_score = model.score(X_val, y_val)
        outer_scores.append(val_score)

    return outer_scores

models = get_models()
```

```
results, names = [], []


# Perform my Nested Cross-Validation and evaluate models
for name, model in models.items():
    cv_scores = evaluate_model_with_nested_cv(model, X_dmem_train, y_dmem_train)
    results.append(cv_scores)
    names.append(name)



    print(f'>{name}: Mean CV Score = {np.mean(cv_scores):.3f} (Std = {np.std(cv_scores):.3f})')


plt.figure(figsize=(12, 6))
plt.boxplot(results, labels=names, showmeans=True)
plt.xticks(rotation=45)
plt.xlabel('Number of Features Selected')
plt.ylabel('Cross-Validated Score (Accuracy)')
plt.title('Model Performance with Different Numbers of Features Selected via RFE (Nested CV)')
plt.grid()
plt.show()


#I then wanted to experiment with another possible feature extraction method using variance thresholding - my thought process behind this is I could get substancial peaks as features
variance_threshold = 0.05


selector = VarianceThreshold(threshold=variance_threshold)
X_reduced = selector.fit_transform(X_dmem_train)


selected_features = X_dmem_train.columns[selector.get_support()]
print("Original", X_dmem_train.shape)
print("Reduced feature matrix shape:", X_reduced.shape)
print("Selected features:", selected_features)
#This reduces the dimensionality but I have worries as the setups follow different distributions taht this threhold may be resulting in bias in my data
```

```
#Number of features from my RFE
param_grid = {
    'rfe__n_features_to_select': [8]
}

dt_classifier = RandomForestClassifier(
    criterion='gini',
    max_depth=None,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=10,
    random_state=42
)

# Train my final model with the best parameters
final_dt_classifier = dt_classifier
final_rfe = RFE(estimator=final_dt_classifier, n_features_to_select=8, step=1)
final_rfe.fit(X_dmem_train, y_dmem_train)

selected_features = X_dmem_train.columns[final_rfe.support_]
feature_importances = final_rfe.estimator_.feature_importances_

print(f'Selected Features: {selected_features}')
print(f'Feature Importances of Selected Features: {feature_importances}')

#check on test set
final_dt_classifier.fit(X_dmem_train, y_dmem_train)

y_pred_prob_dt = final_dt_classifier.predict_proba(X_dmem_test)[:, 1]
y_pred_dt = final_dt_classifier.predict(X_dmem_test)
accuracy_dt = accuracy_score(y_dmem_test, y_pred_dt)
```

```
roc_auc_dt = roc_auc_score(y_dmem_test, y_pred_prob_dt)
conf_matrix_dt = confusion_matrix(y_dmem_test, y_pred_dt)

print(f'Accuracy on Test Set for Decision Tree: {accuracy_dt:.2f}')
print(f'AUC-ROC on Test Set for Decision Tree: {roc_auc_dt:.2f}')
print(f'Confusion Matrix for Decision Tree:\n{conf_matrix_dt}')

fpr, tpr, thresholds = roc_curve(y_dmem_test, y_pred_prob_dt)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (area = {roc_auc_dt:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()

"""## Logistic AND RandomForest Using PCA

I wanted to examine the effects of PCA and see if taht was any more sucessful
"""

X_train = X_dmem_train.to_numpy()
y_train = y_dmem_train.to_numpy()

k_values = range(1, 10)
models = {
    'Logistic Regression': LogisticRegression(),
```

```
    'Random Forest': RandomForestClassifier()
}

stratified_kfold = StratifiedKFold(n_splits=10)
mean_fpr = np.linspace(0, 1, 100)
results = {}

for model_name, model in models.items():
    best_k = None
    best_score = 0
    mean_tpr_list = []



    for k in k_values:
        fold_scores = []
        mean_tpr = np.zeros_like(mean_fpr)

        for train_idx, test_idx in stratified_kfold.split(X_train, y_train):

            pipeline = Pipeline([
                ('pca', PCA(n_components=k)),  # PCA step
                ('classifier', model)          # Classifier step
            ])

            pipeline.fit(X_train[train_idx], y_train[train_idx])
            y_pred_proba = pipeline.predict_proba(X_train[test_idx])[:, 1]
            fpr, tpr, _ = roc_curve(y_train[test_idx], y_pred_proba)
            roc_auc = auc(fpr, tpr)
            fold_scores.append(roc_auc)
            mean_tpr += np.interp(mean_fpr, fpr, tpr)

        mean_tpr /= stratified_kfold.get_n_splits()
```

```
        mean_tpr[-1] = 1.0

        avg_score = np.mean(fold_scores)
        print(f'Average AUC for {model_name} with k={k}: {avg_score:.4f}')

        if avg_score > best_score:
            best_score = avg_score
            best_k = k
            best_mean_tpr = mean_tpr

    results[model_name] = (best_k, best_score, best_mean_tpr)

plt.figure(figsize=(8, 6))
for model_name, (best_k, best_score, mean_tpr) in results.items():
    plt.plot(mean_fpr, mean_tpr, label=f'{model_name} (Best k={best_k}, AUC={best_score:.2f})')

plt.plot([0, 1], [0, 1], color='grey', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.grid()
plt.show()

for model_name, (best_k, best_score, _) in results.items():
    print(f'Best k for {model_name}: {best_k} with average AUC: {best_score:.4f}')

#My fucntion to plot Roc curve and perform PCA

def plot_roc_curve(X_train, y_train, X_test, y_test, k, model):
```

```
    pca = PCA(n_components=k)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)

    model.fit(X_train_pca, y_train)

    y_pred_proba = model.predict_proba(X_test_pca)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
    roc_auc = auc(fpr, tpr)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='red', linestyle='--')  # Diagonal line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc='lower right')
    plt.grid()
    plt.show()

plot_roc_curve(X_dmem_train, y_dmem_train, X_dmem_test, y_dmem_test, k=10, model=LogisticRegression())

"""## KNN"""

#using my PLS componets to reduce dimensionality as it was the bettwe performer over PCA
pls = PLSRegression(n_components=5)
X_dmem_train_pls = pls.fit_transform(X_dmem_train, y_dmem_train)[0]
X_dmem_test_pls = pls.fit_transform(X_dmem_test,y_dmem_test)[0]

#Using a KNN will iintroduce the non linear aspect whcih I want to inspect
```

```
knn_cv = KNeighborsClassifier(n_neighbors=5)

cv_scores = cross_val_score(knn_cv, X_dmem_train_pls, y_dmem_train, cv=StratifiedKFold(n_splits=10), scoring='f1')

# Validation scores
print("Cross-validation scores:", cv_scores)
print("Mean CV score (F1 score):", np.mean(cv_scores))

#uisng the same cross validation approach as always but for KNN
knn_cv = KNeighborsClassifier(n_neighbors=5)

train_scores = []
val_scores = []

X_dmem_train_np = np.array(X_dmem_train_pls)
y_dmem_train_np = np.array(y_dmem_train)

cv = StratifiedKFold(n_splits=10)

for train_index, val_index in cv.split(X_dmem_train_np, y_dmem_train_np):
    X_train, X_val = X_dmem_train_np[train_index], X_dmem_train_np[val_index]
    y_train, y_val = y_dmem_train_np[train_index], y_dmem_train_np[val_index]

    knn_cv.fit(X_train, y_train)

    train_pred = knn_cv.predict(X_train)
    val_pred = knn_cv.predict(X_val)

    train_f1 = f1_score(y_train, train_pred)
    val_f1 = f1_score(y_val, val_pred)
```

```
    train_scores.append(train_f1)
    val_scores.append(val_f1)

plt.figure(figsize=(10, 6))
plt.plot(range(1, len(train_scores) + 1), train_scores, label='Training F1 Score', marker='o')
plt.plot(range(1, len(val_scores) + 1), val_scores, label='Validation F1 Score', marker='x')
plt.title('Training vs Validation F1 Score from Cross-Validation')
plt.xlabel('Fold')
plt.ylabel('F1 Score')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()

print(f'Mean Training F1 Score: {np.mean(train_scores):.4f}')
print(f'Mean Validation F1 Score: {np.mean(val_scores):.4f}')

"""Tuning the number of neighbours as a hyperparameter"""

knn2 = KNeighborsClassifier()
param_grid = {'n_neighbors': np.arange(1, 25)}
#usING gridsearch to test all values for n_neighbors
knn_gscv = GridSearchCV(knn2, param_grid, cv=StratifiedKFold(n_splits=10),scoring="f1")
#fit model to data
knn_gscv.fit(X_dmem_train, y_dmem_train)

print(knn_gscv.best_params_)
print(knn_gscv.best_score_)

knn_gscv.fit(X_dmem_train_pls, y_dmem_train)
y_pred = knn_gscv.predict(X_dmem_test_pls)

accuracy_knn= accuracy_score(y_dmem_test, y_pred)
```

```
roc_auc_knn= roc_auc_score(y_dmem_test, y_pred)
conf_matrix_knn = confusion_matrix(y_dmem_test, y_pred)

print(f'Accuracy on Test Set for KNN: {accuracy_knn:.2f}')
print(f'AUC-ROC on Test Set for KNN: {roc_auc_knn:.2f}')
print(f'Confusion Matrix for KNN:\n{conf_matrix_knn}')

fpr, tpr, thresholds = roc_curve(y_dmem_test, y_pred)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (area = {roc_auc_knn:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve for SVM')
plt.legend(loc='lower right')
plt.grid()
plt.show()

#Disappointng reuslts hwo the test set is not reflective of the trainig data and is overfitting also

"""## Using Varianace Thresholding, Kernal PCA and SNV"""

#Here I wanted to examine some more in depth preporcessing as the previous werent effective
# I came across SNV which removes variabolity due to scatter effects
intensity_columns = df.columns[2:]

def apply_snv(df, intensity_columns):
    snv_data = df[intensity_columns].apply(lambda row: (row - row.mean()) / row.std(), axis=1)
    return snv_data
```

```
snv_df = apply_snv(df, intensity_columns)

window_length = 13
poly_order = 2

smoothed_data = {
    col: savgol_filter(snv_df[col], window_length, poly_order)
    for col in snv_df.columns
}

smoothed_df = pd.DataFrame(smoothed_data, columns=intensity_columns)

print(smoothed_df.head())

def plot_intensity_vs_type(df, snv_df, smoothed_df, intensity_column):
    plt.figure(figsize=(18, 6))

    plt.subplot(1, 3, 1)
    plt.title(f'Original Intensity Data ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(df[df['Type'] == sample_type].index, df[df['Type'] == sample_type][intensity_column],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Intensity')

    plt.subplot(1, 3, 2)
    plt.title(f'SNV Transformed ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(snv_df.index[df['Type'] == sample_type],
                 snv_df[intensity_column][df['Type'] == sample_type],
```

```
            label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('SNV Intensity')

    plt.subplot(1, 3, 3)
    plt.title(f'SG Smoothed ({intensity_column}) vs Load')
    for sample_type in df['Type'].unique():
        plt.plot(smoothed_df.index[df['Type'] == sample_type],
            smoothed_df[intensity_column][df['Type'] == sample_type],
            label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Smoothed Intensity')

    plt.tight_layout()
    plt.show()

for intensity_col in ['I001', 'I101', 'I201', 'I301', 'I401', 'I501']:
    plot_intensity_vs_type(df, snv_df, smoothed_df, intensity_col)

variance_threshold = 0.003 # Far smaller varinavce now that baseline is corrected
selector = VarianceThreshold(threshold=variance_threshold)

X_reduced = selector.fit_transform(X_dmem_train)

selected_features = X_dmem_train.columns[selector.get_support()]
print("Original", X_dmem_train.shape)
print("Reduced feature matrix shape:", X_reduced.shape)
print("Selected features:", selected_features)

#I experiment with kernal PCA in an attempt to try capture the non linear relationship between the intenity points
```

```
pca = KernelPCA(n_components=5)
X_dmem_train_pca = pca.fit_transform(X_reduced)
X_dmem_test_pca = pca.fit_transform(X_dmem_test)
pca_df = pd.DataFrame(data=X_dmem_train_pca, columns=['PC1', 'PC2','PC3', 'PC4','PC5'])


pca_df['Type'] = y_dmem_train


plt.figure(figsize=(8, 6))
sns.scatterplot(data=pca_df, x='PC1', y='PC2', hue='Type', style='Type', palette='viridis', alpha=0.7)
plt.title('PCA of Training Data Colored by Type')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid()
plt.tight_layout()
plt.legend(title='Type')
plt.show()


"""Apply my Kernal Pca data and snv data to RF using hyperparameter tuning"""


param_grid = {
    'n_estimators': [50, 100, 200],  # Number of trees in the forest
    'max_depth': [None, 10, 20, 30],  # Maximum depth of each tree
    'min_samples_split': [2, 5, 10]  # Minimum number of samples required to split a node
}


rf = RandomForestClassifier(random_state=42)
cv = StratifiedKFold(n_splits=5)
scorers = {
    'f1_score': make_scorer(f1_score, average='weighted'),
    'accuracy': make_scorer(accuracy_score)
}
```

```
grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    scoring=scorers,
    refit='f1_score',
    cv=cv,
    return_train_score=True,
    n_jobs=-1
)



grid_search.fit(X_dmem_train_pca, y_dmem_train)
best_rf = grid_search.best_estimator_
#Cv sxcores
cv_results = grid_search.cv_results_
train_f1 = cv_results['mean_train_f1_score'][grid_search.best_index_]
train_acc = cv_results['mean_train_accuracy'][grid_search.best_index_]
val_f1 = cv_results['mean_test_f1_score'][grid_search.best_index_]
val_acc = cv_results['mean_test_accuracy'][grid_search.best_index_]
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Training F1 Score: {train_f1:.4f}")
print(f"Training Accuracy: {train_acc:.4f}")
print(f"Validation F1 Score: {val_f1:.4f}")
print(f"Validation Accuracy: {val_acc:.4f}")

# Train the best Random Forest model
best_rf.fit(X_reduced, y_dmem_train)

# Predict on the test set
y_pred = best_rf.predict(X_dmem_test[selected_features])

accuracy_rf = accuracy_score(y_dmem_test, y_pred)
```

```
roc_auc_rf = roc_auc_score(y_dmem_test, y_pred)
conf_matrix_rf = confusion_matrix(y_dmem_test, y_pred)

print(f'Accuracy on Test Set for RF: {accuracy_rf:.2f}')
print(f'AUC-ROC on Test Set for RF: {roc_auc_rf:.2f}')
print(f'Confusion Matrix for RF:\n{conf_matrix_rf}')

# Classification Report for RF
print("\nClassification Report for RF:")
print(classification_report(y_dmem_test, y_pred))

fpr, tpr, thresholds = roc_curve(y_dmem_test, y_pred)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', label=f'ROC curve (area = {roc_auc_rf:.2f})')
plt.plot([0, 1], [0, 1], color='red', linestyle='--')  # Diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve for RF')
plt.legend(loc='lower right')
plt.grid()
plt.show()

# Confusion Matrix Heatmap for RF
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix_rf, annot=True, fmt="d", cmap="Blues", cbar=False,
        xticklabels=['Predicted Negative', 'Predicted Positive'],
        yticklabels=['Actual Negative', 'Actual Positive'])
plt.title("Confusion Matrix for RF")
plt.xlabel("Predicted Label")
```

```
plt.ylabel("True Label")
plt.show()


"""# Regression


#Data Loading and imports
"""


import pandas as pd
from google.colab import drive
drive.mount('/content/drive')


file_path = '/content/drive/MyDrive/Queens/Module2/2022QUB.xlsx'
df = pd.read_excel(file_path, sheet_name='Data')


pip install pygam


from scipy import stats
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.signal import savgol_filter
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.cross_decomposition import PLSRegression
from sklearn.decomposition import PCA, KernelPCA
from sklearn.model_selection import cross_val_predict,KFold, cross_validate, GridSearchCV, cross_val_score
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.linear_model import LinearRegression, LassoCV ,Lasso
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import PolynomialFeatures
from pygam import GAM, s, LinearGAM
```

```
import sys
from sklearn.feature_selection import RFECV
from sklearn.ensemble import RandomForestRegressor
from sklearn.pipeline import make_pipeline
from xgboost import XGBRegressor
from sklearn.pipeline import Pipeline

df

df= df.applymap(lambda x: x.strip() if isinstance(x, str) else x)
print("\nCleaned DataFrame:")

"""# Exploration"""

df.describe()

df.count().sort_values()

z = np.abs(stats.zscore(df._get_numeric_data()))
print(z)

df= df[(z < 3).all(axis=1)]
print(df.shape)
df.reset_index(drop=True, inplace=True)

df

"""#EDA against Load

"""
```

```
df

sid_load_counts = df.groupby(['SID', 'Load']).size().unstack(fill_value=0) # Group by SID so that I can visulsie Load counts

print("Count of Load values by SID:")
print(sid_load_counts)

sid_load_counts.plot(kind='bar', stacked=True, figsize=(12, 6), colormap="viridis")
plt.title('Count of Load by SID')
plt.xlabel('SID')
plt.ylabel('Count of Load')
plt.legend(title='Load', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

#Boxplot to examine Type distribution by load
plt.figure(figsize=(12, 6))
df.boxplot(column='Load', by='Type', grid=False)
plt.title('Load Distribution by Type')
plt.suptitle('')
plt.xlabel('Type')
plt.ylabel('Load')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Boxplot of load by Matrix
plt.figure(figsize=(12, 6))
df.boxplot(column='Load', by='Matrix', grid=False)
plt.title('Load Distribution by Matrix')
plt.suptitle('')
plt.xlabel('Type')
```

```
plt.ylabel('Load')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()


plt.figure(figsize=(12, 6))
sns.countplot(data=df, x='Load', order=df['Load'].value_counts().index)
load_counts = df['Load'].value_counts(normalize=True) * 100

for i, count in enumerate(load_counts):
    plt.text(i, count + 1, f'{count:.1f}%', ha='center') #Adding the text elemnt to make my graoh clearer

plt.title('Count and Percentage of Load Values')
plt.xlabel('Load')
plt.ylabel('Count')

plt.tight_layout()
plt.show()

plt.figure(figsize=(12, 6))
sns.violinplot(data=df, x='SID', y='Load', palette='Set2')
plt.title('Violin Plot of Load by SID')
plt.xlabel('SID')
plt.ylabel('Load')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

plt.figure(figsize=(12, 6))
sns.countplot(data=df, x='Load', hue='SID', order=df['Load'].value_counts().index, palette='viridis')
plt.title('Count of Load Values by SID')
plt.xlabel('Load')
```

```
plt.ylabel('Count')
plt.legend(title='SID', bbox_to_anchor=(1.05, 1), loc='upper left')

plt.tight_layout()
plt.show()

plt.figure(figsize=(12, 6))
sns.violinplot(data=df, x='Type', y='Load', palette='Set2')
plt.title('Violin Plot of Load by SID')
plt.xlabel('SID')
plt.ylabel('Load')
plt.xticks(rotation=45)
plt.grid(True)
plt.show()

"""# Preprocessing

## Exploring shape
"""

df = df.drop(['SID',"Type"], axis=1)
print("\nDataFrame after dropping 'SID':")
print(df)

sample_indices = range(len(df))  # Use indexes for x axis values
num_I_columns = df.shape[1] - 4
wavelengths = np.linspace(900, 1700, num=num_I_columns) # I found out these numbers from the original paper which allowed me to plot the shape of the data

plt.figure(figsize=(12, 8))
for sample_index in sample_indices:
    reflectance_values = df.iloc[sample_index, 4:].values  # All intensity columns
```

```
    plt.plot(wavelengths, reflectance_values, marker='o', linestyle='-', markersize=3, label=f'Sample {sample_index}')


plt.title('Reflectance Spectrum for All Samples')
plt.xlabel('Wavelength (nm)')
plt.ylabel('Reflectance')
plt.grid()
plt.xlim([899, 1721])  # Limit my x-axis to the wavelength range
plt.ylim([0, np.max(df.iloc[:, 4:].values) * 1.1])
plt.tight_layout()
plt.show()


#Here I am doing the same but visualusing the data after smoothong
num_I_columns = df.shape[1] - 4
wavelengths = np.linspace(900, 1700, num=num_I_columns)


sample_indices = range(0, len(df), 50)


plt.figure(figsize=(12, 8))


for sample_index in sample_indices:
    reflectance_values = df.iloc[sample_index, 4:].values
    sg = savgol_filter(reflectance_values, window_length=13, polyorder=2) #Applying my smmothing
    plt.plot(wavelengths, sg, linestyle='--', label=f'Derivative Sample {sample_index}')


plt.title('Savitzky-Golay Smoothing on Data')
plt.xlabel('Wavelength (nm)')
plt.ylabel('Reflectance')
plt.grid()
plt.xlim([899, 1721])
plt.ylim(np.min(sg) * 0.5)
plt.tight_layout()
plt.show()
```

```
intensity_columns = df.columns[2:]

scaler = MinMaxScaler() #Use min max sclaer instead
normalised_data = scaler.fit_transform(df[intensity_columns])
normalised_df = pd.DataFrame(normalised_data, columns=intensity_columns)

window_length = 13
poly_order = 2

# Apply Savitzky-Golay filter tmy columns
smoothed_data = {
    col: savgol_filter(normalised_df[col], window_length, poly_order)
    for col in normalised_df.columns
}

smoothed_df = pd.DataFrame(smoothed_data, columns=intensity_columns)
print(smoothed_df.head())

def plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_column):
    plt.figure(figsize=(18, 6))

    plt.subplot(1, 3, 1)
    plt.title(f'Original Intensity Data ({intensity_column}) vs Load')
    for sample_type in df['Load'].unique():
        plt.plot(df[df['Load'] == sample_type].index, df[df['Load'] == sample_type][intensity_column],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Intensity')

    plt.subplot(1, 3, 2)
```

```
    plt.title(f'Min-Max Normalised ({intensity_column}) vs Load')
    for sample_type in df['Load'].unique():
        plt.plot(normalised_df.index[df['Load'] == sample_type],
                 normalised_df[intensity_column][df['Load'] == sample_type],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Normalised Intensity')

    plt.subplot(1, 3, 3)
    plt.title(f'SG Smoothed ({intensity_column}) vs Load')
    for sample_type in df['Load'].unique():
        plt.plot(smoothed_df.index[df['Load'] == sample_type],
                 smoothed_df[intensity_column][df['Load'] == sample_type],
                 label=sample_type, marker='o', linestyle='-')
    plt.legend()
    plt.xlabel('Sample Index')
    plt.ylabel('Smoothed Intensity')

    plt.tight_layout()
    plt.show()

for intensity_col in ['I001', 'I101', 'I201', 'I301', 'I401', 'I501']:
    plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_col)

"""## With Split"""

other_columns = df.drop(df.columns[df.columns.str.startswith('I')], axis=1)
other_columns

final_df = pd.concat([other_columns,smoothed_df], axis=1)
print(final_df.head())
```

```
print(final_df)


"""## Test/Train Split"""

dmem_train = final_df[final_df['Matrix'] == 'dmem'].reset_index(drop=True)  # DataFrame for 'dmem'
pbs_train= final_df[final_df['Matrix'] == 'pbs'].reset_index(drop=True)    # DataFrame for 'pbs'

print("DataFrame for dmem:")
print(dmem_train)

print("\nDataFrame for pbs:")
print(pbs_train)

# Same split as the rest of my analysis just using Load as target
X_dmem = dmem_train.drop(['Load', 'Matrix'], axis=1)
y_dmem = dmem_train['Load']

split_index_dmem = int(0.70 * len(X_dmem))

X_dmem_train, X_dmem_test = X_dmem[:split_index_dmem], X_dmem[split_index_dmem:]
y_dmem_train, y_dmem_test = y_dmem[:split_index_dmem], y_dmem[split_index_dmem:]

X_pbs = pbs_train.drop(['Load', 'Matrix'], axis=1)
y_pbs = pbs_train['Load']
split_index_pbs = int(0.70 * len(X_pbs))
X_pbs_train, X_pbs_test = X_pbs[:split_index_pbs], X_pbs[split_index_pbs:]
y_pbs_train, y_pbs_test = y_pbs[:split_index_pbs], y_pbs[split_index_pbs:]

X_dmem_train
```

```
"""# Dmem

## Partial Least Squares
"""

X_dmem_train.shape

def pls_mse_analysis(X, y, max_components=10, plot_components=True):
    mse_values = []
    components = np.arange(2, max_components + 1) #I dont want to start at 1

    for comp in components:
     #using CV
        pls = PLSRegression(n_components=comp)
        y_cv = cross_val_predict(pls, X, y, cv=10)
        mse_values.append(mean_squared_error(y, y_cv))

        # Print my progress
        percent_complete = 100 * (comp) / max_components
        sys.stdout.write("\r%d%% completed" % percent_complete)
        sys.stdout.flush()
    sys.stdout.write("\n")

    # Find the location of smallest MSE so I can print
    optimal_index = np.argmin(mse_values)
    print("Optimal number of PLS components:", optimal_index + 1)

    # Plot MSE vs number of components
    if plot_components:
        plt.figure(figsize=(10, 6))
        with plt.style.context('ggplot'):
            plt.plot(components, mse_values, '-v', color='blue', mfc='blue', label="MSE")
```

```
        plt.plot(components[optimal_index], mse_values[optimal_index], 'P', ms=10, mfc='red')
        plt.xlabel('Number of PLS components')
        plt.ylabel('Mean Squared Error (MSE)')
        plt.title('PLS Components Selection')
        plt.legend()
        plt.xlim(left=1)
        plt.show()

# Now I fit PLS model with optimal number of components and calcualte trainingand validtion scores
optimal_pls = PLSRegression(n_components=optimal_index + 1)
optimal_pls.fit(X, y)
y_trained= optimal_pls.predict(X)
y_cross_validated = cross_val_predict(optimal_pls, X, y, cv=10)
calibration_score = r2_score(y, y_trained)
cross_validation_score = r2_score(y, y_cross_validated)

mse_calibration = mean_squared_error(y, y_trained)
mse_cross_validation = mean_squared_error(y, y_cross_validated)

print('R^2 Training: %5.3f' % calibration_score)
print('R^2 Cross-Validation: %5.3f' % cross_validation_score)
print('MSE Training: %5.3f' % mse_calibration)
print('MSE Cross-Validation: %5.3f' % mse_cross_validation)

#Plot my cross validated bets model load values vs predictions
y_range = max(y) - min(y)
y_cal_range = max(y_trained) - min(y_trained)

fit_line = np.polyfit(y, y_cross_validated.ravel(), 1)

with plt.style.context('ggplot'):
    fig, ax = plt.subplots(figsize=(9, 5))
```

```
        ax.scatter(y_cross_validated, y, c='red', edgecolors='k', label="Cross-Validated")
        ax.plot(np.polyval(fit_line, y), y, c='blue', linewidth=1, label="Best Fit")
        # Plot my ideal 1:1 line
        ax.plot(y, y, color='green', linewidth=1, label="Ideal 1:1")

        plt.title(f'$R^{2}$ (CV): {cross_validation_score:.3f}')
        plt.xlabel('Predicted Values')
        plt.ylabel('Actual Values')
        plt.legend()
        plt.show()

#Call my function
pls_mse_analysis(X_dmem_train,y_dmem_train, 30, plot_components=True)

#fit on training and preduct on test
pls = PLSRegression(n_components=4)
pls.fit(X_dmem_train, y_dmem_train)
y_test_pred = pls.predict(X_dmem_test)

r2_test = r2_score(y_dmem_test, y_test_pred)
mse_test = mean_squared_error(y_dmem_test, y_test_pred)

print(f"R² on Test Set: {r2_test:.3f}")
print(f"MSE on Test Set: {mse_test:.3f}")

plt.figure(figsize=(10, 6))
plt.scatter(y_dmem_test, y_test_pred, color='blue', edgecolor='k', alpha=0.6, label="Predicted vs Actual")
plt.plot([y_dmem_test.min(), y_dmem_test.max()], [y_dmem_test.min(), y_dmem_test.max()], 'r--', lw=2, label="Ideal 1:1 Line")
plt.xlabel("Actual Load Values")
plt.ylabel("Predicted Load Values")
plt.title("PLS Regression: Actual vs Predicted Values on Test Set")
plt.legend()
```

```
plt.tight_layout()
plt.show()

"""## Polynomial Features using variance thresholding

Here I wanted to examine the effcets of tranforming the features to polynomial
"""

X_train_pls = pls.transform(X_dmem_train) #use my PLS for diumension reduction to use later
X_test_pls = pls.transform(X_dmem_test)

variance_threshold = 0.065 #use variancve thresholding for reduction
selector = VarianceThreshold(threshold=variance_threshold)
X_reduced = selector.fit_transform(X_dmem_train)
selected_features = X_dmem_train.columns[selector.get_support()]

print("Original", X_dmem_train.shape)
print("Reduced feature matrix shape:", X_reduced.shape)
print("Selected features:", selected_features)

X_reduced_test = selector.transform(X_dmem_test)
X_reduced_test.shape

#Here I do a cubic tranformation of the reduced feature set
poly = PolynomialFeatures(degree=3, include_bias=False)

X_poly = poly.fit_transform(X_reduced)
X_poly_test = poly.transform(X_reduced_test)

X_poly.shape
```

```
X_poly_df = pd.DataFrame(X_poly)
X_poly_df

#Using Lasso to extract best features as my GAM model can only take so many
c\v_strategy = KFold(n_splits=10)
lasso_cv = LassoCV(alphas=([0.001]), cv=cv_strategy, random_state=42)
lasso_cv.fit(X_poly, y_dmem_train)
lasso_selected_features = X_poly_df.columns[lasso_cv.coef_ != 0]

print("Best Alpha from LassoCV:", lasso_cv.alpha_)
print("Lasso Selected Features for GAM:\n", lasso_selected_features)

coef_df = pd.DataFrame(lasso_cv.coef_, index=X_poly_df.columns, columns=['Coefficient'])
print("Lasso Coefficients:\n", coef_df[coef_df['Coefficient'] != 0])

"""GAM model"""

pip install pygam

#As GAM is addidative need to define indices
selected_features_indices = [  1,   2,   3,   7,   8,   9,  10,  11,  13,  16,  17,  18,  20,  21,
       24,  32,  39,  41,  42,  51,  56,  58,  59,  63,  65,  66,  69,  71,
       72,  80,  81,  84,  90,  91, 102, 141, 147, 168, 213, 268, 349, 371,
      377, 398, 433, 443, 449, 452, 453]


X_selected = X_poly_df.iloc[:, selected_features_indices]
print(X_selected.head())
#Define my addiditve GAM
gam = GAM(s(0) + s(1) + s(2) + s(3) + s(4) + s(5) + s(6) + s(7) +
        s(8) + s(9) + s(10) + s(12) + s(13) + s(14) + s(15) +
        s(16) + s(16) + s(18) +s(19) + s(20) + s(21) + s(22)+
```

```
        s(23) + s(24) + s(25) + s(26) + s(27) + s(28) + s(29) + s(30) + s(31) + s(32) + s(33),
        link='identity')
gam.fit(X_dmem_train, y_dmem_train)

y_pred = gam.predict(X_dmem_test)
#EEvaluate on test set
mse = mean_squared_error(y_dmem_test, y_pred)
print(f'Mean Squared Error: {mse}')

plt.figure(figsize=(8, 6))
plt.scatter(y_dmem_test, y_pred, alpha=0.6, color='blue', edgecolors='w', s=80)
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Predicted vs Actual Values')

max_value = max(max(y_dmem_test), max(y_pred))
min_value = min(min(y_dmem_test), min(y_pred))
plt.plot([min_value, max_value], [min_value, max_value], 'r--')

plt.grid()

plt.xlim([min_value, max_value])
plt.ylim([min_value, max_value])
plt.show()

"""## Lasso"""

# same as above
alphas = np.logspace(-2, 0, 50)
lasso_cv = LassoCV(alphas=alphas, cv=10, random_state=42)
lasso_cv.fit(X_dmem_train, y_dmem_train)
```

```
best_alpha = lasso_cv.alpha_
print(f'Best Alpha: {best_alpha}')
lasso_coefficients = lasso_cv.coef_
print(f'Lasso Coefficients: {lasso_coefficients}')

#here I am using cross-validation  on my lasso model
lasso_model = Lasso(alpha=.01, random_state=42)
cv_results = cross_validate(lasso_model, X_dmem_train, y_dmem_train, cv=10, scoring='neg_mean_squared_error', return_train_score=True)

train_mse_scores = -cv_results['train_score']
val_mse_scores = -cv_results['test_score'] #converting the negagtive mse to positive

print("Training MSE scores for each fold:")
print(train_mse_scores)
print("\nValidation MSE scores for each fold:")
print(val_mse_scores)

mean_train_mse = np.mean(train_mse_scores)
std_train_mse = np.std(train_mse_scores)
mean_val_mse = np.mean(val_mse_scores)
std_val_mse = np.std(val_mse_scores)

print(f"\nMean Training MSE: {mean_train_mse:.4f}")
print(f"Standard Deviation of Training MSE: {std_train_mse:.4f}")
print(f"\nMean Validation MSE: {mean_val_mse:.4f}")
print(f"Standard Deviation of Validation MSE: {std_val_mse:.4f}")

lasso_model.fit(X_dmem_train, y_dmem_train)
feature_importances = lasso_model.coef_

# Print only non-zero features
non_zero_features = np.where(feature_importances != 0)[0]  # Get indices of non-zero features
```

```
print("\nNon-Zero Feature Indices and Coefficients:")
for idx in non_zero_features:
    print(f"Feature index: {idx}, Coefficient: {feature_importances[idx]:.4f}")


y_test_pred = lasso_model.predict(X_dmem_test)


mse_test = mean_squared_error(y_dmem_test, y_test_pred)
mae_test = mean_absolute_error(y_dmem_test, y_test_pred)
r2_test = r2_score(y_dmem_test, y_test_pred)

print(f"Test Set Evaluation:")
print(f"Mean Squared Error (MSE): {mse_test:.4f}")
print(f"Mean Absolute Error (MAE): {mae_test:.4f}")
print(f"R^2 Score: {r2_test:.4f}")


import matplotlib.pyplot as plt


plt.figure(figsize=(10, 6))
plt.scatter(y_dmem_test, y_test_pred, alpha=0.7, color='blue')
plt.plot([y_dmem_test.min(), y_dmem_test.max()], [y_dmem_test.min(), y_dmem_test.max()], 'r--')
plt.title('Predicted vs Actual')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.grid()
plt.show()


"""## With Hyperparameters"""


#Here I tune my lasso Alpha to regulate the amount of regulatsation
```

```
pipeline = make_pipeline(Lasso(random_state=42))



param_grid = {
    'lasso__alpha': [.001,.01]
}

#Using nested CV to avoid data leakage when tuning
inner_cv = GridSearchCV(pipeline, param_grid, cv=5, scoring='neg_mean_squared_error', n_jobs=-1)

outer_cv_results = cross_validate(
    inner_cv,
    X_dmem_train,
    y_dmem_train,
    cv=10,
    scoring='r2',
    return_train_score=True
)

outer_train_mse_scores = -outer_cv_results['train_score']
outer_val_mse_scores = -outer_cv_results['test_score']

print("Training MSE scores for each outer fold:")
print(outer_train_mse_scores)
print("\nValidation MSE scores for each outer fold:")
print(outer_val_mse_scores)

mean_outer_train_mse = np.mean(outer_train_mse_scores)
std_outer_train_mse = np.std(outer_train_mse_scores)
mean_outer_val_mse = np.mean(outer_val_mse_scores)
std_outer_val_mse = np.std(outer_val_mse_scores)
```

```
print(f"\nMean Outer Training MSE: {mean_outer_train_mse:.4f}")
print(f"Standard Deviation of Outer Training MSE: {std_outer_train_mse:.4f}")
print(f"\nMean Outer Validation MSE: {mean_outer_val_mse:.4f}")
print(f"Standard Deviation of Outer Validation MSE: {std_outer_val_mse:.4f}")

# I fit the final model on the entire training data with the best alpha found in the inner CV
inner_cv.fit(X_dmem_train, y_dmem_train)
best_alpha = inner_cv.best_params_['lasso__alpha']
print(f"\nBest alpha found: {best_alpha:.4f}")

# I get the coefficients from the best estimator in the inner CV
final_model = inner_cv.best_estimator_
feature_importances = final_model.named_steps['lasso'].coef_

non_zero_features = np.where(feature_importances != 0)[0]  # Get indices of non-zero features
print("\nNon-Zero Feature Indices and Coefficients:")
for idx in non_zero_features:
    print(f"Feature index: {idx}, Coefficient: {feature_importances[idx]:.4f}")

lasso_model = Lasso(alpha=best_alpha, random_state=42)
lasso_model.fit(X_dmem_train, y_dmem_train)
y_test_pred = lasso_model.predict(X_dmem_test)


mse_test = mean_squared_error(y_dmem_test, y_test_pred)
mae_test = mean_absolute_error(y_dmem_test, y_test_pred)
r2_test = r2_score(y_dmem_test, y_test_pred)


print(f"Test Set Evaluation:")
print(f"Mean Squared Error (MSE): {mse_test:.4f}")
print(f"Mean Absolute Error (MAE): {mae_test:.4f}")
```

```
print(f"R^2 Score: {r2_test:.4f}")


plt.figure(figsize=(10, 6))
plt.scatter(y_dmem_test, y_test_pred, alpha=0.7, color='blue')
plt.plot([y_dmem_test.min(), y_dmem_test.max()], [y_dmem_test.min(), y_dmem_test.max()], 'r--')
plt.title('Predicted vs Actual')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.grid()
plt.show()


"""## GAM with best predictors"""


#Taken from above
non_zero_features = np.array([69, 71, 72, 73, 349,492,503,511])


print(non_zero_features)


non_zero_features


X_train_gam = X_dmem_train.iloc[:, non_zero_features]
X_test_gam = X_dmem_test.iloc[:, non_zero_features]
# I f a GAM model; using s() for each selected feature
gam = LinearGAM(s(0) + s(1) + s(2) + s(3) + s(4)+s(5)+s(6)+s(7))
gam.fit(X_train_gam, y_dmem_train)


y_pred_gam = gam.predict(X_test_gam)


mse_gam = mean_squared_error(y_dmem_test, y_pred_gam)
mae_gam = mean_absolute_error(y_dmem_test, y_pred_gam)
r2_gam = r2_score(y_dmem_test, y_pred_gam)
```

```
# Print results
print(f'MSE of GAM model: {mse_gam:.4f}')
print(f'MAE of GAM model: {mae_gam:.4f}')
print(f'R² of GAM model: {r2_gam:.4f}')

plt.figure(figsize=(8, 6))
plt.scatter(y_dmem_test, y_pred_gam, alpha=0.6, color="b", edgecolor='k')
plt.plot([y_dmem_test.min(), y_dmem_test.max()], [y_dmem_test.min(), y_dmem_test.max()], 'r--', lw=2)
plt.xlabel("Actual values (y_test)")
plt.ylabel("Predicted values (y_pred_gam)")
plt.title("Actual vs. Predicted Values (Test Set)")
plt.show()
#Made more daring predictions than Lasso

"""## RFE Random Forest"""

df_train = pd.DataFrame(X_dmem_train)
df_train

X_train_pls.shape

y_dmem_train

"""## RFE Random Forest"""

pca = PCA()
X_pca = pca.fit_transform(X_dmem_train)

cumulative_explained_variance = np.cumsum(pca.explained_variance_ratio_)
```

```
# I determine the number of components to reach 99% variance
n_components_99 = np.argmax(cumulative_explained_variance >= 0.99) + 1
print(f"Number of components needed to explain 99% of the variance: {n_components_99}")
# Plotting the explained variance
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1), pca.explained_variance_ratio_, marker='o')
plt.title('Explained Variance Ratio by Principal Components')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance Ratio')
plt.xticks(range(1, len(pca.explained_variance_ratio_) + 1))
plt.grid()
plt.axhline(y=0.1, color='r', linestyle='--')
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.7)
plt.title('PCA Result: First Two Principal Components')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid()
plt.show()

n_components = 2
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_dmem_train)
X_test_pca = pca.fit_transform(X_dmem_test)

df_train_pca = pd.DataFrame(X_pca)
df_train_pca
df_test_pca = pd.DataFrame(X_test_pca)
df_test_pca

model = RandomForestRegressor()
```

```
rfecv = RFECV(estimator=model, step=1, cv=5, scoring='r2') #I use RFE cross valdiation focuing on best fit to avoid overfitting
rfecv.fit(X_pca, y_dmem_train)

selected_features = df_train_pca.columns[rfecv.support_]
print("Selected features:", selected_features)

print("Cross-validation scores for each step:")
mean_test_scores = -rfecv.cv_results_['mean_test_score']
std_test_scores = rfecv.cv_results_['std_test_score']
print(mean_test_scores)
print(std_test_scores)

mean_mse = np.mean(mean_test_scores)
avg_std = np.mean(std_test_scores)
print(f"\nMean MSE across folds: {mean_mse:.4f}")
print(f"Average Std Dev of MSE across folds: {avg_std:.4f}")

"""# XGboost"""

model = XGBRegressor(random_state=42)

selected_X_train = df_train_pca.iloc[:, rfecv.support_]

outer_cv_results = cross_validate(
    model,
    selected_X_train,
    y_dmem_train,
    cv=5,
    scoring='neg_mean_squared_error',
    return_train_score=True
)
```

```
# O convert scores from negative to positive MSE for readability
outer_train_mse_scores = -outer_cv_results['train_score']
outer_val_mse_scores = -outer_cv_results['test_score']

print("Training MSE scores for each outer fold:")
print(outer_train_mse_scores)
print("\nValidation MSE scores for each outer fold:")
print(outer_val_mse_scores)

mean_outer_train_mse = np.mean(outer_train_mse_scores)
std_outer_train_mse = np.std(outer_train_mse_scores)
mean_outer_val_mse = np.mean(outer_val_mse_scores)
std_outer_val_mse = np.std(outer_val_mse_scores)

print(f"\nMean Outer Training MSE: {mean_outer_train_mse:.4f}")
print(f"Standard Deviation of Outer Training MSE: {std_outer_train_mse:.4f}")
print(f"\nMean Outer Validation MSE: {mean_outer_val_mse:.4f}")
print(f"Standard Deviation of Outer Validation MSE: {std_outer_val_mse:.4f}")

"""XGBoost with Kernal PCA"""

# Using Kernal PCA - found out that cannot use explained variance so this is the method instead - looks at reconstrcution error
def kernel_pca_reconstruction_error(data, max_components, kernel='rbf', gamma=None):
    errors = []
    for n in range(1, max_components + 1):
        kpca = KernelPCA(n_components=n, kernel=kernel, gamma=gamma, fit_inverse_transform=True)
        transformed_data = kpca.fit_transform(data)
        reconstructed_data = kpca.inverse_transform(transformed_data)

        # Compute the reconstruction error (Mean Squared Error)
        error = mean_squared_error(data, reconstructed_data)
        errors.append(error)
```

```
    return errors

errors = kernel_pca_reconstruction_error(X_dmem_train, max_components=20, kernel='rbf', gamma=0.1)

plt.plot(range(1, 21), errors, marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Reconstruction Error (MSE)')
plt.title('Reconstruction Error for Different Number of Components')
plt.grid(True)
plt.show()

X_train

y_train

def perform_kernel_pca(train_data, test_data, num_components=8, kernel='rbf', gamma=0.1):
    kpca = KernelPCA(n_components=num_components, kernel=kernel, gamma=gamma, fit_inverse_transform=True)
    train_data_kpca = kpca.fit_transform(train_data)
    test_data_kpca = kpca.transform(test_data)
    return train_data_kpca, test_data_kpca

# Kernel PCA to the training and test data with 6 components
kpca_data_8_train, kpca_data_8_test = perform_kernel_pca(X_dmem_train, X_dmem_test, num_components=6, kernel='rbf', gamma=0.1)
xgb_reg = XGBRegressor(objective='reg:squarederror', random_state=0)

xgb_reg.fit(kpca_data_8_train, y_dmem_train)

y_pred = xgb_reg.predict(kpca_data_8_test)

mse = mean_squared_error(y_dmem_test, y_pred)
```

```
mae = mean_absolute_error(y_dmem_test, y_pred)
r2 = r2_score(y_dmem_test, y_pred)

print(f"Mean Squared Error: {mse:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")
print(f"R^2 Score: {r2:.4f}")

def create_pipeline(num_components=8, kernel='rbf', gamma=0.1):
    return Pipeline([
        ('kpca', KernelPCA(n_components=num_components, kernel=kernel, gamma=gamma)),
        ('xgb', XGBRegressor(objective='reg:squarederror', random_state=0))
    ])

param_grid = {
    'kpca__n_components': [5, 8, 10],
    'xgb__learning_rate': [0.01, 0.1, 0.2],
    'xgb__max_depth': [3, 5, 7],
}



pipeline = create_pipeline()
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_dmem_train, y_dmem_train)
best_model = grid_search.best_estimator_

y_pred = best_model.predict(X_dmem_test)
mse = mean_squared_error(y_dmem_test, y_pred)
mae = mean_absolute_error(y_dmem_test, y_pred)
r2 = r2_score(y_dmem_test, y_pred)

print(f"Best Hyperparameters: {grid_search.best_params_}")
print(f"Mean Squared Error: {mse:.4f}")
```

```
print(f"Mean Absolute Error: {mae:.4f}")
print(f"R^2 Score: {r2:.4f}")

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(y_dmem_test, y_pred, alpha=0.7, color='blue')
plt.plot([y_dmem_test.min(), y_dmem_test.max()], [y_dmem_test.min(), y_dmem_test.max()], 'r--', lw=2)
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")

residuals = y_dmem_test - y_pred

plt.subplot(1, 2, 2)
plt.scatter(y_pred, residuals, alpha=0.7, color='purple')
plt.hlines(y=0, xmin=y_pred.min(), xmax=y_pred.max(), colors='red', linestyles='--', lw=2)
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residual Plot")

plt.tight_layout()
plt.show()

"""# Clustering

In this notebook I aim to explore if there is a natural seperation within the data and if this reflects the viral load.

## Preprocessing
"""
```

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')

file_path = '/content/drive/MyDrive/Queens/Module2/2022QUB.xlsx' #Load in my data
df = pd.read_excel(file_path, sheet_name='Data')

from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error
from sklearn.cluster import SpectralClustering
from sklearn.preprocessing import StandardScaler, normalize
from sklearn.metrics import silhouette_score

df

df= df.applymap(lambda x: x.strip() if isinstance(x, str) else x) #Here I am stripping leading and training whitespaces
print("\nCleaned DataFrame:")

df = df.drop(['SID',"Type"], axis=1) # I am dropping irrelevant columns as they dont contribute to my model

print("\nDataFrame after dropping 'SID' and 'Type':")
print(df)

"""## Data Preparation
```

```
In this section I am removing outliers and normalising my data and apply the smoothing techniques mentioned in my report.
"""


z = np.abs(stats.zscore(df._get_numeric_data())) #Converting data to its z-score so that it can be used to detect outliers
print(z)


df= df[(z < 3).all(axis=1)] #Here I am removing any rows outside the threshold of 3 std of the mean
  print(df.shape)
  df.reset_index(drop=True, inplace=True)


intensity_columns = df.columns[2:]  # I am removing the wavelength columsn


scaler = StandardScaler() # I am normalising the data
normalised_data = scaler.fit_transform(df[intensity_columns])
normalised_df = pd.DataFrame(normalised_data, columns=intensity_columns)


window_length = 13  # Here I define a window length and polynomial order for Savitzky-Golay smoothing which must be odd and greater than the polynomial order
poly_order = 2


# I apply Savitzky-Golay filter to each column - I chose window length 13 and poly order 2 after a trial and error process
smoothed_data = {
    col: savgol_filter(normalised_df[col], window_length, poly_order)
    for col in normalised_df.columns
}


smoothed_df = pd.DataFrame(smoothed_data, columns=intensity_columns)
print(smoothed_df.head())


# Here I am defining a function to plot original, normalised, and smoothed data and compare their distributions to see the effect of the preprocessing on the data
def plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_column):
```

```
plt.figure(figsize=(18, 6))
plt.subplot(1, 3, 1)
plt.title(f'Original Intensity Data ({intensity_column}) vs Load')
for sample_type in df['Load'].unique():
    plt.plot(df[df['Load'] == sample_type].index, df[df['Load'] == sample_type][intensity_column],
             label=sample_type, marker='o', linestyle='-')
plt.legend()
plt.xlabel('Sample Index')
plt.ylabel('Intensity')

plt.subplot(1, 3, 2)
plt.title(f'Standardised ({intensity_column}) vs Load')
for sample_type in df['Load'].unique():
    plt.plot(normalised_df.index[df['Load'] == sample_type],
             normalised_df[intensity_column][df['Load'] == sample_type],
             label=sample_type, marker='o', linestyle='-')
plt.legend()
plt.xlabel('Sample Index')
plt.ylabel('Normalised Intensity')

plt.subplot(1, 3, 3)
plt.title(f'SG Smoothed ({intensity_column}) vs Load')
for sample_type in df['Load'].unique():
    plt.plot(smoothed_df.index[df['Load'] == sample_type],
             smoothed_df[intensity_column][df['Load'] == sample_type],
             label=sample_type, marker='o', linestyle='-')
plt.legend()
plt.xlabel('Sample Index')
plt.ylabel('Smoothed Intensity')

plt.tight_layout()
plt.show()
```

```
for intensity_col in ['I001', 'I101', 'I201', 'I301', 'I401', 'I501']:
    plot_intensity_vs_type(df, normalised_df, smoothed_df, intensity_col)


other_columns = df.drop(df.columns[df.columns.str.startswith('I')], axis=1) # Get out the other columsn to add back


final_df = pd.concat([other_columns,smoothed_df], axis=1) #I combine with standarised data
print(final_df.head())


print(final_df)


dmem_train = final_df[final_df['Matrix'] == 'dmem'].reset_index(drop=True)  # Split my data based on matrix as otherwise the clustering would make no sense
pbs_train= final_df[final_df['Matrix'] == 'pbs'].reset_index(drop=True)


print("DataFrame for dmem:")
print(dmem_train)
print("\nDataFrame for pbs:")
print(pbs_train)


#Even tho this is a supervised task I still want to withold a unseen dataset that I can use to measure the effectiveness of using mean viral loads to predict
X_dmem = dmem_train.drop(['Load', 'Matrix'], axis=1)
y_dmem = dmem_train['Load']
split_index_dmem = int(0.70 * len(X_dmem))
X_dmem_seen, X_dmem_unseen = X_dmem[:split_index_dmem], X_dmem[split_index_dmem:]
y_dmem_seen, y_dmem_unseen= y_dmem[:split_index_dmem], y_dmem[split_index_dmem:]


X_pbs = pbs_train.drop(['Load', 'Matrix'], axis=1)
y_pbs = pbs_train['Load']
split_index_pbs = int(0.70 * len(X_pbs))
X_pbs_seen, X_pbs_unseen = X_pbs[:split_index_pbs], X_pbs[split_index_pbs:]
y_pbs_seen, y_pbs_unseen = y_pbs[:split_index_pbs], y_pbs[split_index_pbs:]
```

```
X_dmem_seen


X_dmem_unseen


"""## PCA Dimensionality Reduction - Dmem"""


# My function to perform PCA
def perform_pca(data, num_components=10):
    pca = PCA(n_components=num_components)
    pca_data = pca.fit_transform(data)
    return pca, pca_data


pca_10, pca_data_10 = perform_pca(X_dmem_seen, num_components=10) #I experiment with 10 componets and then with 2 to examine the total bvaraince covered
pca_2, pca_data_2 = perform_pca(X_dmem_seen, num_components=2)




# Here I am creating a plot for Explained Variance Ratio
def plot_variance_analysis(pca, title):
    explained_variance_ratio = pca.explained_variance_ratio_
    cumulative_variance_ratio = np.cumsum(explained_variance_ratio)


    fig, axes = plt.subplots(1, 2, figsize=(14, 5))


    # Explained Variance Ratio Plot
    axes[0].bar(range(1, len(explained_variance_ratio) + 1), explained_variance_ratio, align='center')
    axes[0].set_xlabel('Principal Component')
    axes[0].set_ylabel('Explained Variance Ratio')
    axes[0].set_title(f'Explained Variance Ratio - Dmem')
    axes[0].grid()
```

```
    # Cumulative Explained Variance Plot
    axes[1].plot(range(1, len(cumulative_variance_ratio) + 1), cumulative_variance_ratio, marker='o')
    axes[1].axhline(y=0.75, color='red', linestyle='--', label='75% Explained Variance') #Put these lines in as markers to make my graph clearer
    axes[1].axhline(y=0.85, color='pink', linestyle='--', label='85% Explained Variance')
    axes[1].axhline(y=0.95, color='purple', linestyle='--', label='95% Explained Variance')
    axes[1].set_xlabel("Number of Components")
    axes[1].set_ylabel("Cumulative Explained Variance")
    axes[1].set_title("Cumulative Explained Variance")
    axes[1].grid()
    axes[1].legend()

    plt.tight_layout()
    plt.show()

    total_variance_covered = np.sum(explained_variance_ratio) # I calculate total variance covered by the components
    print(f"Total Variance Covered by the first {len(explained_variance_ratio)} components for {title}: {total_variance_covered:.4f}")



plot_variance_analysis(pca_10, 'Matrix:')
plot_variance_analysis(pca_2, 'Matrix:')

components = pd.DataFrame(pca_2.components_, columns=X_dmem_seen.columns)
importance = components.abs() # Get the absolute values of the components so that I can extract the laoding values of each components

# Lopp for each principal component so that I can get the top 10 features
top_features = {}
for i in range(2):  # My first 2 components
    top_features[f'PC{i+1}'] = importance.iloc[i].nlargest(10)

# I am putting the top 10 features for each principal component in table format and outpuuting
for pc, features in top_features.items():
```

```
    print(f"\nTop 10 Features for {pc}:")
    print(features)


#Plotting the table - am able to see that all fetaures have roughly equal loading - no one feature dominating which may point to noise being captured
plt.figure(figsize=(10, 6))
importance.iloc[0].nlargest(10).plot(kind='barh', color='skyblue')
plt.title('Top 10 Important Features for Principal Component 1')
plt.xlabel('Absolute Value of Loadings')
plt.ylabel('Features')
plt.grid(axis='x')
plt.show()


# Possible list for number of clusters to expermient with
parameters = [2, 3, 4, 5, 6, 7, 8, 9, 10]
# Instantiate ParameterGrid with the number of clusters as input - found this technique during my research
parameter_grid = ParameterGrid({'n_clusters': parameters})


best_score = -1
silhouette_scores = []
inertia_scores = []


# Perform multiple initialisations for stability - this avoids settling at a local minimum which kmeans can be prone to do
for p in parameter_grid:
    # Running KMeans with multiple random (50 runs per configuration). This means that the clusters are starting at random points and avoid it being stuck
    kmeans_model = KMeans(n_clusters=p['n_clusters'], n_init=50, random_state=42)
    kmeans_model.fit(pca_data_2)  # Fit model on my PCA data


    # Calculate silhouette score - which shows how effective clusteres are
    ss = metrics.silhouette_score(pca_data_2, kmeans_model.labels_)
    silhouette_scores.append(ss)  # Store silhouette score
```

```
    # Calculate my within sum of squared distances to the closest cluster center aim is to mimise this
    inertia = kmeans_model.inertia_
    inertia_scores.append(inertia)  # Store inertia

    print(f'Parameter: {p["n_clusters"]}, Silhouette Score: {ss:.4f}, Inertia: {inertia:.4f}')

    # Check which parameter has the best silhouette score
    if ss > best_score:
        best_score = ss
        best_grid = p
#My silhouette scores and inertia (Elbow Method)
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.bar(range(len(silhouette_scores)), silhouette_scores, align='center', color='#722f59', width=0.5)
plt.xticks(range(len(silhouette_scores)), parameters)
plt.title('Silhouette Score', fontweight='bold')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')

plt.subplot(1, 2, 2)
plt.plot(parameters, inertia_scores, marker='o', color='b')
plt.title('Elbow Method for Optimal k', fontweight='bold')
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.grid()

plt.tight_layout()
plt.show()

print(f'Best Number of Clusters based on Silhouette Score: {best_grid["n_clusters"]} with a score of {best_score:.4f}')
```

```
pca_data_2.shape


"""Again I am running a number of models a numeber of times to ensure that the random creation of clusters doesnt result in the model getting stuck at a local minimum"""


kmeans1 = KMeans(n_clusters=5,
                 random_state=3,
                 n_init=1).fit(pca_data_2)
kmeans20 = KMeans(n_clusters=5,
                  random_state=3,
                  n_init=20).fit(pca_data_2);
kmeans50 = KMeans(n_clusters=5,
                  random_state=3,
                  n_init=50).fit(pca_data_2);
kmeans1.inertia_, kmeans20.inertia_, kmeans50.inertia_


kmeans20.fit(pca_data_2)


n_clusters = 5


labels = kmeans20.labels_ #This is just extracting the labels for cluster colouring
centroids = kmeans20.cluster_centers_
plt.figure(figsize=(10, 6))
plt.scatter(pca_data_2[:, 0], pca_data_2[:, 1], c=labels, cmap='viridis', marker='o', alpha=0.6, edgecolor='k')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='X', s=200, label='Centroids')



plt.title(f'KMeans Clustering (n_clusters={n_clusters})', fontsize=16)
plt.xlabel('Principal Component 1', fontsize=14)
plt.ylabel('Principal Component 2', fontsize=14)
plt.grid(True)
plt.legend()
```

```
plt.tight_layout()

plt.show()


X_dmem_seen = X_dmem_seen.copy()  # Avoids SettingWithCopyWarning by working on a copy - this error kept coming up so I found this solution online

X_dmem_seen['Cluster'] = labels #This is extracting the clusters from the seen data which i will use to predict


# Adding the true viral load data to 'X_dmem_seen'

X_dmem_seen['Load'] = y_dmem_seen.values


# Calculate mean viral load for each cluster

cluster_viral_loads = X_dmem_seen.groupby('Cluster')['Load'].mean() #Here I am calculating the mean of my clsuters to use for prediction

print("Mean viral load for each cluster:\n", cluster_viral_loads)


# I Predict viral load for each data point based on cluster mean

X_dmem_seen['Predicted Viral Load'] = X_dmem_seen['Cluster'].map(cluster_viral_loads)


mse_seen = mean_squared_error(X_dmem_seen['Load'], X_dmem_seen['Predicted Viral Load'])

print(f"Mean Squared Error for the 'seen' dataset: {mse_seen:.4f}")


X_dmem_seen


"""## Unseen - Dmem"""


pca_data_unseen = pca_2.transform(X_dmem_unseen)

X_dmem_unseen['Cluster'] = kmeans20.predict(pca_data_unseen)


# I am predicting viral load for each data point in unseen data based on cluster mean from seen data

X_dmem_unseen['Predicted Viral Load'] = X_dmem_unseen['Cluster'].map(cluster_viral_loads)

X_dmem_unseen['Load'] = y_dmem_unseen

mse_unseen = mean_squared_error(X_dmem_unseen['Load'], X_dmem_unseen['Predicted Viral Load'])

print("Predicted Viral Loads for the 'unseen' dataset:")
```

```
print(X_dmem_unseen[['Cluster', 'Load', 'Predicted Viral Load']])

print(f"Mean Squared Error for the 'unseen' dataset: {mse_unseen:.4f}")


"""## Spectral Clustering


Results were slighly disappointing so I decided to examien another clustering technique

"""


#I had to reset this as new columns had been added in the last clustering

X_dmem = dmem_train.drop(['Load', 'Matrix'], axis=1)

y_dmem = dmem_train['Load']


split_index_dmem = int(0.70 * len(X_dmem))

X_dmem_seen, X_dmem_unseen = X_dmem[:split_index_dmem], X_dmem[split_index_dmem:]

y_dmem_seen, y_dmem_unseen= y_dmem[:split_index_dmem], y_dmem[split_index_dmem:]


X_pbs = pbs_train.drop(['Load', 'Matrix'], axis=1)

y_pbs = pbs_train['Load']

split_index_pbs = int(0.70 * len(X_pbs))

X_pbs_seen, X_pbs_unseen = X_pbs[:split_index_pbs], X_pbs[split_index_pbs:]

y_pbs_seen, y_pbs_unseen = y_pbs[:split_index_pbs], y_pbs[split_index_pbs:]


#Same process as above but with the Spectral Model

parameters = [2, 3, 4, 5, 6, 7, 8, 9, 10]

parameter_grid = ParameterGrid({'n_clusters': parameters})


best_score = -1

silhouette_scores = []


for p in parameter_grid:

    spectral_model = SpectralClustering(n_clusters=p['n_clusters'], affinity='nearest_neighbors', n_neighbors=10, random_state=42) #The NN introduces the non-linear aspect which I wanted to examine
```

```
    labels = spectral_model.fit_predict(pca_data_2)

    ss = metrics.silhouette_score(pca_data_2, labels)
    silhouette_scores.append(ss)

    print(f'Number of Clusters: {p["n_clusters"]}, Silhouette Score: {ss:.4f}')

    if ss > best_score:
        best_score = ss
        best_grid = p

plt.figure(figsize=(8, 6))

plt.bar(range(len(silhouette_scores)), silhouette_scores, align='center', color='#722f59', width=0.5)
plt.xticks(range(len(silhouette_scores)), parameters)
plt.title('Silhouette Score for Different Cluster Numbers (Spectral Clustering)', fontweight='bold')
plt.xlabel('Number of Clusters')
plt.ylabel('Silhouette Score')

plt.show()

print(f'Best Number of Clusters based on Silhouette Score: {best_grid["n_clusters"]} with a score of {best_score:.4f}')

n_clusters = 8
spectral_model = SpectralClustering(n_clusters=n_clusters, affinity='nearest_neighbors', n_neighbors=10, random_state=42)
labels = spectral_model.fit_predict(pca_data_2)

plt.figure(figsize=(10, 6))

scatter = plt.scatter(pca_data_2[:, 0], pca_data_2[:, 1], c=labels, cmap='tab10', edgecolor='k', s=50)
plt.colorbar(scatter, ticks=range(n_clusters), label="Cluster Labels")
```

```
plt.title(f"Spectral Clustering with {n_clusters} Clusters")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")

plt.show()
# Can see that the PCA plotting is the exact same but the clustering is different so interesting to analyse now

print(f"Shape of pca_data_2: {pca_data_2.shape}")
print(f"Shape of X_dmem_seen: {X_dmem_seen.shape}")

# Ensure they have the same number of rows - I put in some error code here to avoid misanalysis
if pca_data_2.shape[0] != X_dmem_seen.shape[0]:
    print("Warning: Number of rows in pca_data_2 and X_dmem_seen are different.")
else:
    labels = spectral_model.fit_predict(pca_data_2)  # Fit model on my PCA data
    X_dmem_seen['Cluster'] = labels
    X_dmem_seen['Load'] = y_dmem_seen.values
    cluster_viral_loads = X_dmem_seen.groupby('Cluster')['Load'].mean()
    print("Mean viral load for each cluster:\n", cluster_viral_loads)

    X_dmem_seen['Predicted Viral Load'] = X_dmem_seen['Cluster'].map(cluster_viral_loads)
    mse = mean_squared_error(X_dmem_seen['Load'], X_dmem_seen['Predicted Viral Load'])
    print(f"Mean Squared Error of Spectral Clustering-Based Prediction: {mse:.4f}")

#Predicting the Unseen data using the mean from the seen
pca_data_unseen = pca_2.transform(X_dmem_unseen)
X_dmem_unseen['Cluster'] = spectral_model.fit_predict(pca_data_unseen)
X_dmem_unseen['Predicted Viral Load'] = X_dmem_unseen['Cluster'].map(cluster_viral_loads)

X_dmem_unseen['Load'] = y_dmem_unseen
mse_unseen = mean_squared_error(X_dmem_unseen['Load'], X_dmem_unseen['Predicted Viral Load'])
r2_unseen = metrics.r2_score(X_dmem_unseen['Load'], X_dmem_unseen['Predicted Viral Load'])
```

```
print("Predicted Viral Loads for the 'unseen' dataset:")
print(X_dmem_unseen[['Cluster', 'Load', 'Predicted Viral Load']])
print(f"Mean Squared Error for the 'unseen' dataset: {mse_unseen:.4f}")
```