Real or Not? NLP with Disaster Tweets
CS522 Final Project - Final Report

Tanner Fry, Verónica G. Melesse Vergara, Shawn Shen, Miaomiao Zhang
Group 4

Introduction

For our final project, we chose the *Real or Not? NLP with Disaster Tweets* project from Kaggle. This project involves using Natural Language Processing (NLP) to determine if tweets from Twitter contain keywords that categorize the tweet as announcing a natural disaster. In this age of technology, the majority of people get their news or day-to-day information from social media websites. This also means that it has become more common for people to share their own information with their friends. This includes cases when people share information on natural disasters that could be occurring in their area via their social media accounts to spread awareness. The idea of this project is to develop a model that can interpret tweets to determine if the user is in fact sharing information on a natural disaster. The project's dataset includes 10,000 tweets with classifiers. Each data sample includes the following attributes: unique id, text of the tweet, location of the tweet (could be blank), a particular keyword from the tweet (could be blank), and the target/classification (training data only) [2].

Twitter is a social media platform where users can share information with people across the world instantaneously. A tweet is composed of a maximum of 280 characters, and supports attachments such as images, videos, and links.

Pre-processing

In this project, NLP is used to extract keywords within tweets that could imply the tweet is referring to a natural disaster. When using NLP, it is important to preprocess the data that will be used. The following are the techniques that we applied to preprocess the data for this project: punctuation removal, tokenization, stopword removal, stemming. Punctuation removal is the process of removing various punctuation from sentences. For removing punctuation in words, we used the **string** library. Tokenization is used to apply structure to unstructured data, such as split text into separate units such as characters, words, or sentences. For tokenizing data, we used the **re** library. Stopword removal is the process of removing words from text that are common and do not provide much significance for analysis [1]. The stop words we used for this project are found within the **nltk** library. Stemming is the process of removing prefixes or/and suffixes from words such as “pre-”, “post-”, “-ation”, “-ing”, etc [1]. For stemming words in our data, we used the **PorterStemmer** model in the nltk library.

After the data has been preprocessed, the data gets vectorized, where the text is transformed into numeric form for machine learning models to understand [1]. To vectorize the data for this project, we used the **TfidfVectorizer** model in the **sklearn** library. Once the data

has been vectorized, the data is now prepared for the various machine learning models used for classification in this project.

State-of-the-Art Implementation

The Bidirectional Encoder Representations from Transformers (BERT), is one of the state-of-the-art language representation models available today. BERT trains a model using a bidirectional transformer that combines two unsupervised learning techniques: Masked Language Modeling (MaskedLM) and next sentence prediction (NSP). MaskedLM is a process of training a model to be able to predict words within a sentence given the rest of the words of the sentence. Essentially, the model becomes capable of learning statistical properties of word sequencing in natural language [10]. NSP is the process of training a model to predict if a sentence is subsequent to another sentence in the original data. Essentially, during training, the model is provided pairs of sentences, and it predicts whether or not the second sentence within the pair provided is subsequent in the original data [10]. After the pre-training step is complete, BERT can be applied to NLP tasks using a feature-based or a fine-tuned approach [11]. When using the feature-based approach, the output from a BERT model can be used to initialize another BERT model that can be used for further classification. With the fine-tuned approach, a pre-trained BERT model that was trained within a similar domain can be used and the model can be tweaked based on the specific data being used for training.

There are multiple BERT model submissions on Kaggle for this NLP project. The submission that provided the best overall accuracy was able to achieve 85% overall accuracy on the testing data and used the fine-tuning approach of using an existing BERT model and updating the weights based on the data of the project [12]. The first task within this submission was cleaning the data. In the pre-processing steps, the user removed punctuations, fixed common misspellings, removed numbers, and removed spacing. Next, the data was tokenized and split into a training and validation set. Then, the BERT model was configured based on a pre-trained model for text classification. For training the BERT model, the user used a weight decay value of 0.1, and a learning rate of 2e-5 [12]. The weight decay is used to prevent certain attributes within the model from becoming weighted too much, thus keeping the model from becoming skewed. The learning rate is used to specify a degree of adjustment after each epoch based on what the model has learned. From the training, the model was able to achieve an overall accuracy of 88.9% on a randomly selected validation set from the training data. This was the best accuracy found while training and validating, and this trained model was used for submission, which achieved an overall accuracy of 85% on the testing data [12]. While running the model against the testing data, the model took 13382.8 seconds (~223 minutes) to run and generate the results [12].

Implementations

MPP (Case 1, Case 2, and Case 3)

For the maximum posterior probability (MPP) implementation, we modified the source code developed during the semester and provided by Dr. Qi. The implementation for MPP can be found in the **mpp.py** file. The code allows the user to choose whether to apply Case 1 which assumes independent features and does not take variance into account, Case 2 which assumes independent features and same variance, or Case 3 which makes no assumptions regarding dependencies between features or variance.

We used MPP on the cleaned up data and had to make a couple of modifications to take into account the fact that the labels were characters instead of numbers. In addition, because MPP includes information on prior probability, we attempted several pairs of values starting with equal (i.e., 0.5 and 0.5), and favoring one class over the other (i.e., 0.25 and 0.75, or 0.75 and 0.25). The best results were observed when using equal prior probability.

The matrices that resulted from the training data set provided were not invertible. For that reason we were not able to obtain results for Case 2 and Case 3 using the standard MPP procedure. Instead, we investigated using the pseudo-inverse capability in NumPy (i.e., **pinv()**). The method that approximates the inverse takes significantly more time as shown in the **Results** section.

kNN

For implementing the K-Nearest-Neighbors (kNN) algorithm, we used the **KNeighborsClassifier** model in the **sklearn** library [3]. The implementation for the kNN algorithm can be found in the projects repository within the **knn.py** file. To run the file, simply run **knn.py** with **Python3** and no command line arguments.

Within this file, there are three functions: **main()**, **knn()**, and **grid_search_knn()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and can either perform a grid search for kNN with an array of parameters to test, or can run the kNN model with set parameters specified in the function. By default, the grid searching is commented out, and if grid searching for best parameters is desired, the function call will need to be uncommented in the **main()** function, and the parameters dictionary in the **grid_search_knn()** function will need to be updated with the specific parameters that are desired to be tested.

The **grid_search_knn()** function will test all combinations of parameters within the parameters dictionary defined within the function using the **GridSearchCV** model found in the **sklearn** library, and will run combinations in parallel, using all the available cores of the CPU. This function will also return the best parameters found in a dictionary so that it can be used in the **main()** function.

The **knn()** function is a standalone implementation of the kNN model that takes the training data, testing data, and model parameters to use as the input parameters. This function

will also produce a confusion matrix with a heat map, and provide the following performance metrics in the standard out: Precision, Recall, F1-Score, Overall Accuracy, and Execution Time.

For our implementation, only the k-nearest-neighbor value is used in the kNN model as a parameter for both the standalone and the grid searching. From our grid searching, we tested k-nearest-neighbor values ranging from 1 to 50, and found that k equal to 15 provided the best results. The model has the option of changing various other parameters, but from our testing, leaving everything else to default produced the best results. See the **Results** section below for the performance metrics, and to see how well it performed against the other implementations.

BPNN

For implementing the Back Propagation Neural Network (BPNN) algorithm, we used the **MLPClassifier** model in the **sklearn** library [4]. The implementation for the BPNN algorithm can be found in the projects repository within the **bpnn.py** file. To run the file, simply run **bpnn.py** with **Python3** and no command line arguments.

Within this file, there are three functions: **main()**, **bpnn()**, and **grid_search_bpnn()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and can either perform a grid search for BPNN with an array of parameters to test, or can run the BPNN model with set parameters specified in the function. By default, the grid searching is commented out, and if grid searching for best parameters is desired, the function call will need to be uncommented in the **main()** function, and the parameters dictionary in the **grid_search_bpnn()** function will need to be updated with the specific parameters that are desired to be tested.

The **grid_search_bpnn()** function will test all combinations of parameters within the parameters dictionary defined within the function using the **GridSearchCV** model found in the **sklearn** library, and will run combinations in parallel, using all the available cores of the CPU. This function will also return the best parameters found in a dictionary so that it can be used in the **main()** function.

The **bpnn()** function is a standalone implementation of the BPNN model that takes the training data, testing data, and model parameters to use as the input parameters. This function will also produce a confusion matrix with a heat map, and provide the following performance metrics in the standard out: Precision, Recall, F1-Score, Overall Accuracy, and Execution Time.

For our implementation, we tested the following parameters for the BPNN: the activation, function, the solver function, the hidden layer structure, and the learning rate. From our grid searching, we found the best result to be the following: activation function = ‘relu’, solver function = ‘lbfgs’, hidden layer structure = (100,10), and learning rate = 0.0009. The ‘relu’ activation function is the rectified linear unit function [4]. The ‘lbfgs’ solver function is an optimizer in the family of quasi-Newton methods [4]. The model has the option of changing various other parameters, but from our testing, leaving everything else to default produced the best results. See the **Results** section below for the performance metrics, and to see how well it performed against the other implementations.

Random Forest

For implementing the Random Forest (RF) algorithm, we used the **RandomForestClassifier** model in the **sklearn** library [5]. The implementation for the RF algorithm can be found in the projects repository within the **rf.py** file. To run the file, simply run **rf.py** with **Python3** and no command line arguments.

Within this file, there are two functions: **main()**, and **rf()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and calls the **rf()** function to run the model against the project's data.

The **rf()** function is a standalone implementation of the RF model that takes the training data and testing data to use as the input parameters. This function will also produce a confusion matrix with a heat map, and provide the following performance metrics in the standard out: Precision, Recall, F1-Score, Overall Accuracy, and Execution Time.

The RF model used has the option of changing various parameters, but from our testing, leaving everything to default except the max_depth produced the best results. For the max_depth parameter, we set it to None to maximize the potential performance. See the **Results** section below for the performance metrics, and to see how well it performed against the other implementations.

SVM

For implementing the Support Vector Machine (SVM) algorithm, we used the **SVM** model in the **sklearn** library [6]. The implementation for the SVM algorithm can be found in the projects repository within the **svm.py** file. To run the file, simply run **svm.py** with **Python3** and no command line arguments.

Within this file, there are two functions: **main()**, and **svm()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and calls the **svm()** function to run the model against the project's data.

The **svm()** function is a standalone implementation of the SVM model that takes the training data and testing data to use as the input parameters. This function will also produce a confusion matrix with a heat map, and provide the following performance metrics in the standard out: Precision, Recall, F1-Score, Overall Accuracy, and Execution Time.

The SVM model used has the option of changing various parameters, but from our testing, leaving everything to default produced the best results. See the **Results** section below for the performance metrics, and to see how well it performed against the other implementations.

KMeans

For implementing the KMeans algorithm, we used the **KMeans** model in the **sklearn** library. The implementation for the KMeans algorithm can be found in the projects repository within the **kmeans.py** file. To run the file, simply run **kmeans.py** with **Python3** and no command line arguments.

Within this file, there are three functions: **main()**, **kmeans()**, and **grid_search_kmeans()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and can either perform a grid search for Kmeans with an array of parameters to test, or can run the Kmeans model with set parameters specified in the function. By default, the grid searching is commented out, and if grid searching for best parameters is desired, the function call will need to be uncommented in the **main()** function, and the parameters dictionary in the **grid_search_kmeans()** function will need to be updated with the specific parameters that are desired to be tested.

The **grid_search_kmeans()** function will test all combinations of parameters within the parameters dictionary defined within the function using the **GridSearchCV** model found in the **sklearn** library, and will run combinations in parallel, using all the available cores of the CPU. This function will also return the best parameters found in a dictionary so that it can be used in the **main()** function.

The **kmeans()** function is a standalone implementation of the KMeans model that takes the training data, testing data, and model parameters to use as the input parameters. This function will also produce a confusion matrix with a heat map, and provide the following performance metrics in the standard out: Precision, Recall, F1-Score, Overall Accuracy, and Execution Time.

For our implementation, we tested the init and n_init parameters in the model. The number of clusters was kept constant at 2, given the project involved only two possible classification values. From our grid searching, we found the best result to be the following: init = 'k-means++', and n_init = 10. The init parameter is the method used for initialization, and the 'k-means++' does the following: selects initial cluster centers for k-mean clustering is a smart way to speed up convergence [7]. The n_init parameter is the number of times the k-means algorithm will run different centroid seeds [7]. The model has the option of changing various other parameters, but from our testing, leaving everything else to default produced the best results. See the **Results** section below for the performance metrics, and to see how well it performed against the other implementations.

WTA

The winner-take-all algorithm is not included in the sklearn library. To implement this, we used **MiniBatchKMeans()** function from the sklearn library with batch size of 1 [14]. In this way, the classifier searches the distance of one training sample in every iteration and updates parameters, which simulates the behavior of the WTA classifier. The script is in wta.py.

There are two functions in the file: **wta()** and **grid_search_wta()**. When executing the script, **grid_search_wta()** will run first to optimize the parameters. However, the most important parameters, initial cluster center and learning rate, cannot be assigned via function attributes, thus are not optimized. Since the "init" condition is set to "random", the "init_size" needs to be optimized. This parameter determines how many data points are used to assign initial cluster centers. The best init_size are tested to be 13.

PCA

For implementing the Principal Component Analysis (PCA) algorithm, we used the **PCA** model in the **sklearn** library [8]. The implementation for the PCA algorithm can be found in the projects repository within the **pca.py** file. To run the file, simply run **pca.py** with **Python3** and no command line arguments.

Within this file, there are two functions: **main()**, and **pca()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and calls the **pca()** function to run the reduction analysis against the project's data.

The **pca()** function is a standalone implementation of the PCA model that takes the training data, testing data, and number of components to reduce to use as the input parameters. This function will run the training and testing data against the PCA analysis, and return the reduced data.

See the **Results** section below for the performance metrics of the reduced data against the entire dataset across all of the classification algorithms.

FLD

For implementing the Fisher's Linear Discriminant (FLD) algorithm, we used the **LDA** model in the **sklearn** library [9]. The implementation for the FLD algorithm can be found in the projects repository within the **fld.py** file. To run the file, simply run **fld.py** with **Python3** and no command line arguments.

Within this file, there are two functions: **main()**, and **fld()**. The **main()** function will run upon the execution of this file, and it loads in the training and testing data, pre-processes the data, and calls the **fld()** function to run the reduction analysis against the project's data.

The **fld()** function is a standalone implementation of the FLD model that takes the training data, training data labels, testing data, and number of components to reduce to use as the input parameters. This function will run the training and testing data against the FLD analysis, and return the reduced data.

See the **Results** section below for the performance metrics of the reduced data against the entire dataset across all of the classification algorithms.

Fusion

To investigate if combining available predictors makes better performance, we use decision level fusion. Several fusion approaches are tested, including majority voting, naive bayes, random forest, logistic regression, and gradient boosting. Since the random forest, logistic regression, and gradient boosting have similar performance as naive bayes, only the results of majority voting and naive bayes are shown in the report.

For the predictors used in fusion, we employ two best performing methods: BPNN and SVM. Parameter optimization and test sample prediction are the same as mentioned above. The predicted result by BPNN and SMV on the training sample are used as training input (x_train);

predictions on the testing sample are used as testing input (x_test). The process is the same for PCA and FLD reduced dataset.

The fusion.ipynb file contains several sections, including functions for predictors, functions for assessing performance, and functions for running the fusion. Not all functions are used in generating the final result. The majority voting function is hard coded. Since two predictors are used, the majority is defined as a uniform agreement that the fusion result is positive only when both predictors give positive feedback. The naive bayes function uses **GaussianNB()** function from sklearn package [13].

Results

For the results section, there will be a confusion matrix and ROC curve (if applicable) shown for each implementation using the best found parameters that were discussed in the **Implementations** section. There will be results shown for each algorithm using the raw dataset, PCA reduced dataset, and FLD reduced dataset. At the end of the results there will be combined bar graphs to showcase each algorithm against each other to compare performance in precision, recall, F1-score, overall accuracy, and execution time. It should be noted that each of us have different machines that run different algorithms so the results will have some form of bias.

MPP

Note: with the raw dataset, only the first case was able to finish successfully.

Raw Dataset (19262 attributes)

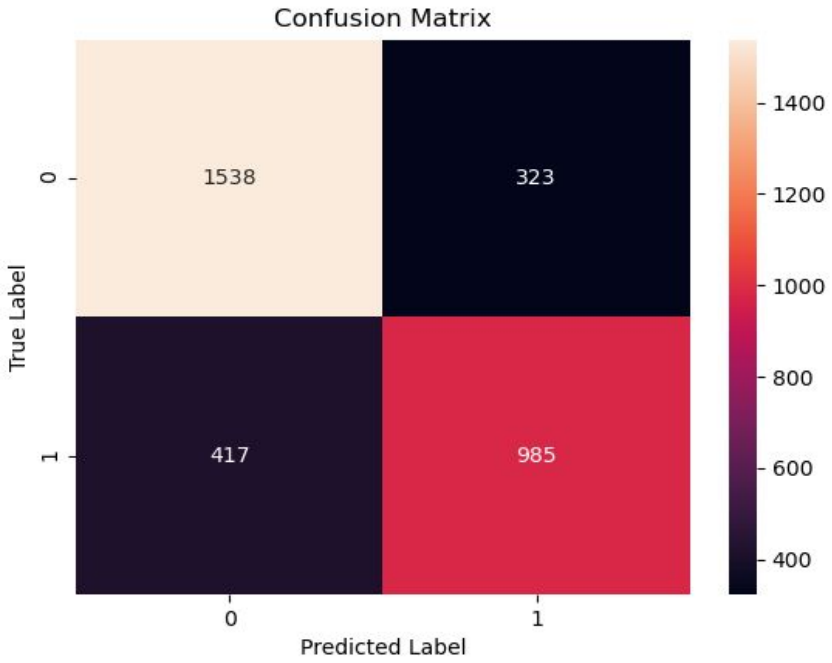


Figure 1. Confusion Matrix for MPP (Case 1) using raw dataset.

PCA Reduced Dataset (2952 attributes)

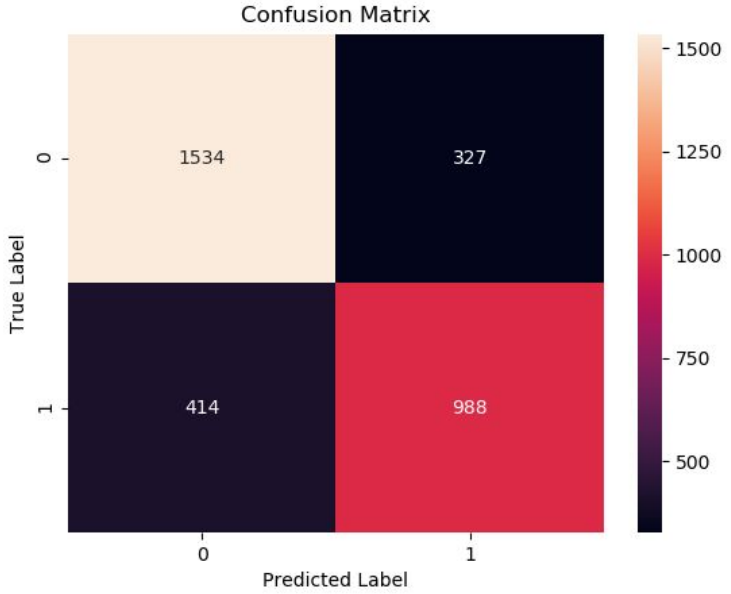


Figure 2. Confusion Matrix for MPP (Case 1) using PCA reduced dataset.

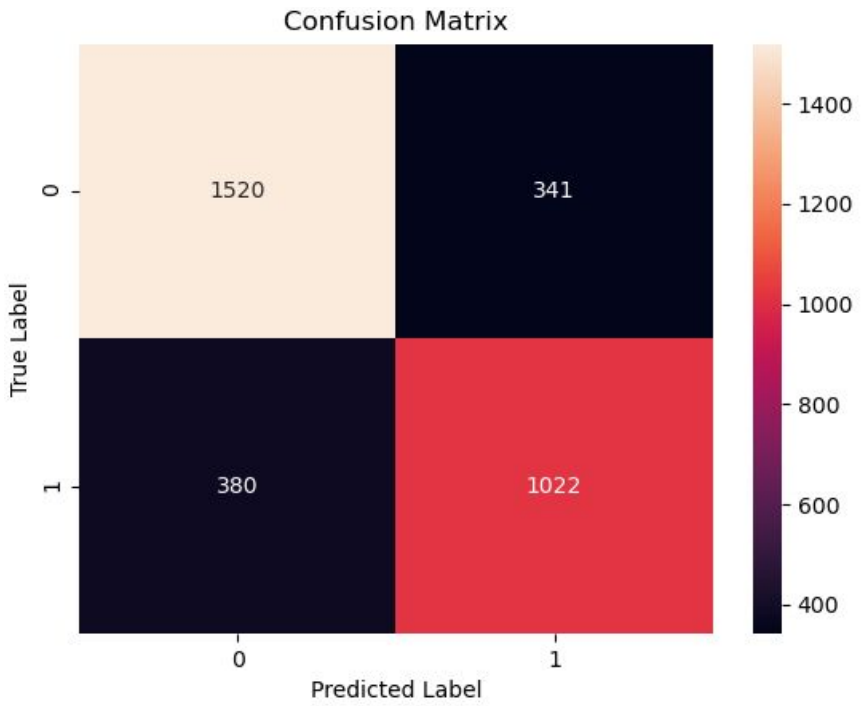


Figure 3. Confusion Matrix for MPP (Case 2) using PCA reduced dataset.

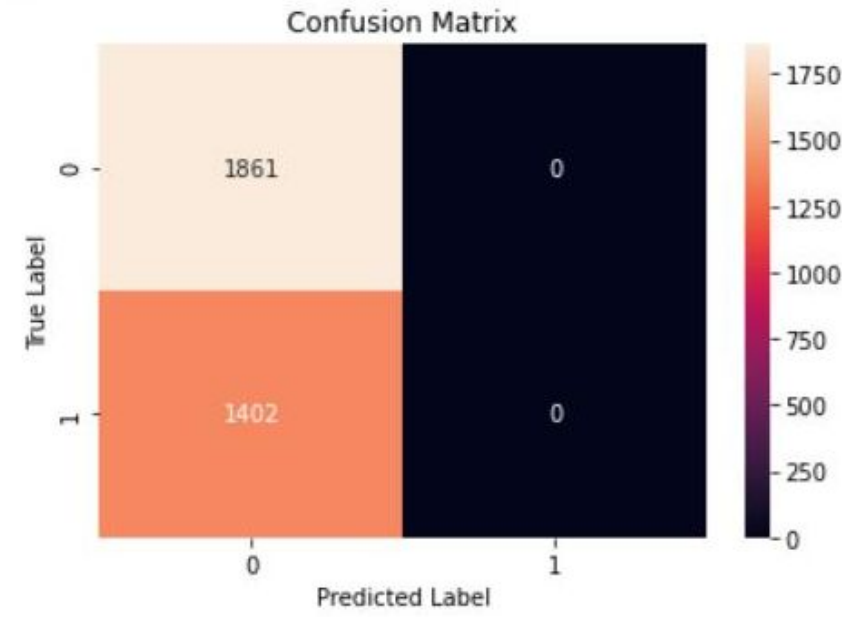


Figure 4. Confusion Matrix for MPP (Case 3) using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

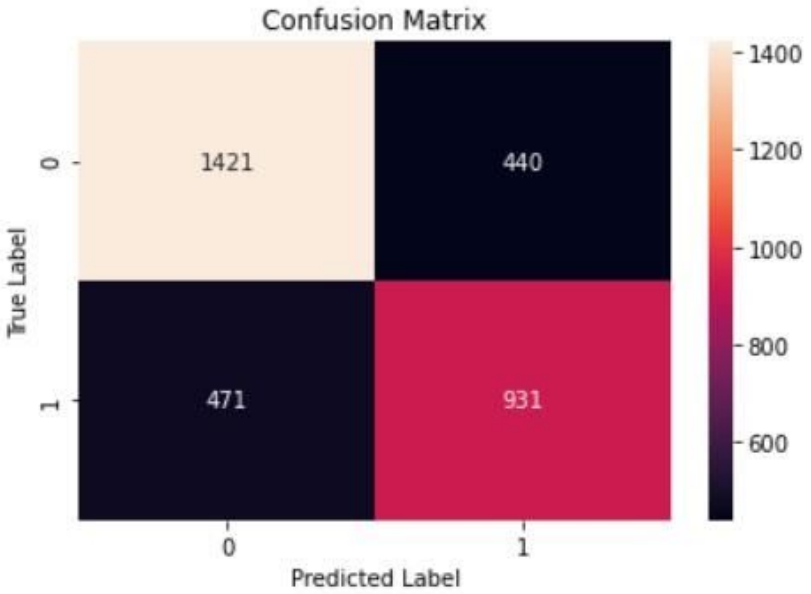


Figure 5. Confusion Matrix for MPP (Case 1) using FLD reduced dataset.

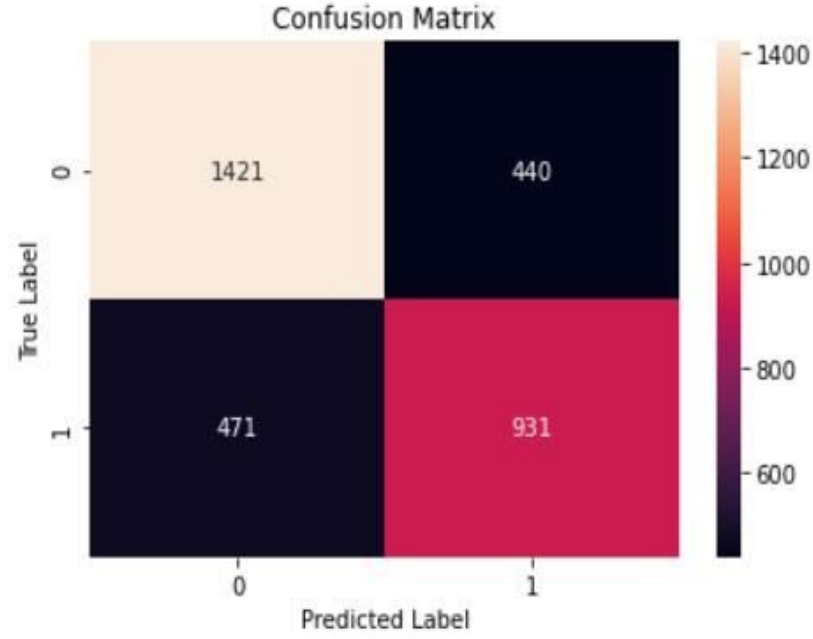


Figure 6. Confusion Matrix for MPP (Case 2) using FLD reduced dataset.

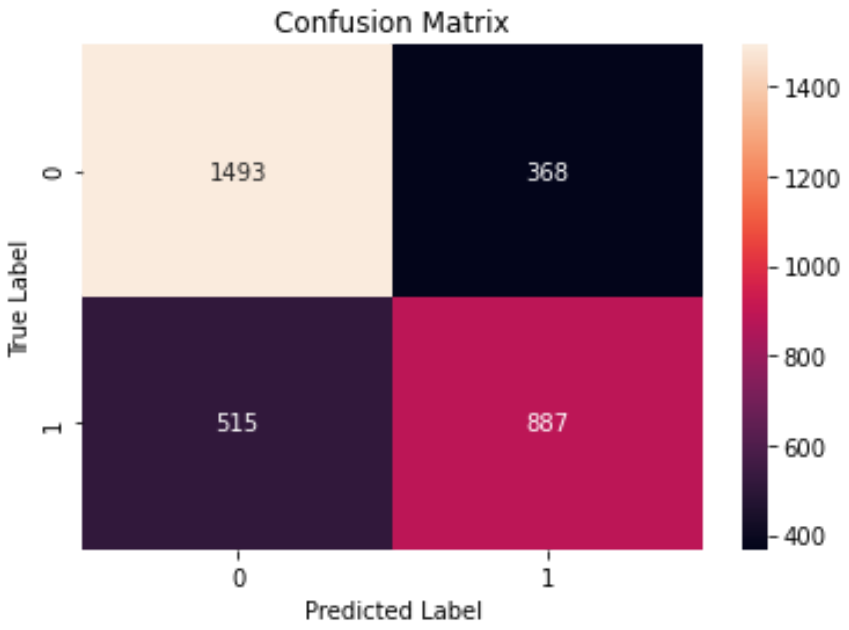


Figure 7. Confusion Matrix for MPP (Case 3) using FLD reduced dataset.

kNN

Raw Dataset (19262 attributes)

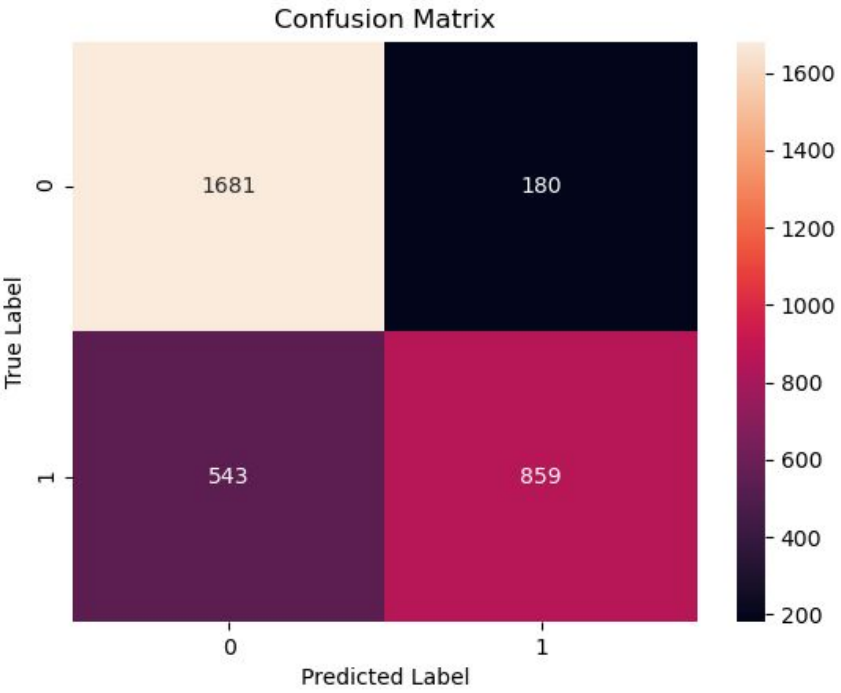


Figure 8. Confusion matrix for kNN using raw dataset.

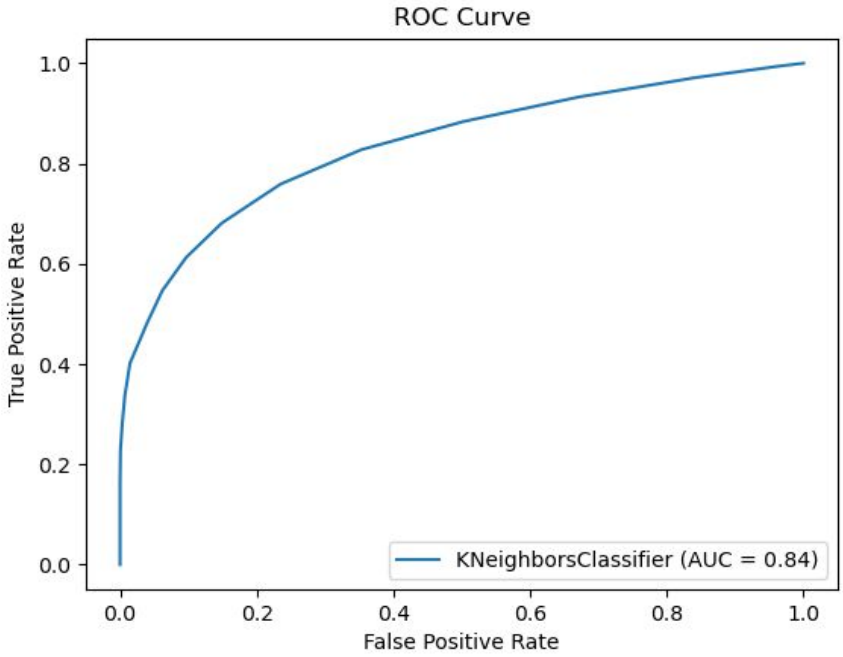


Figure 9. ROC Curve for kNN using raw dataset.

PCA Reduced Dataset (2952 attributes)

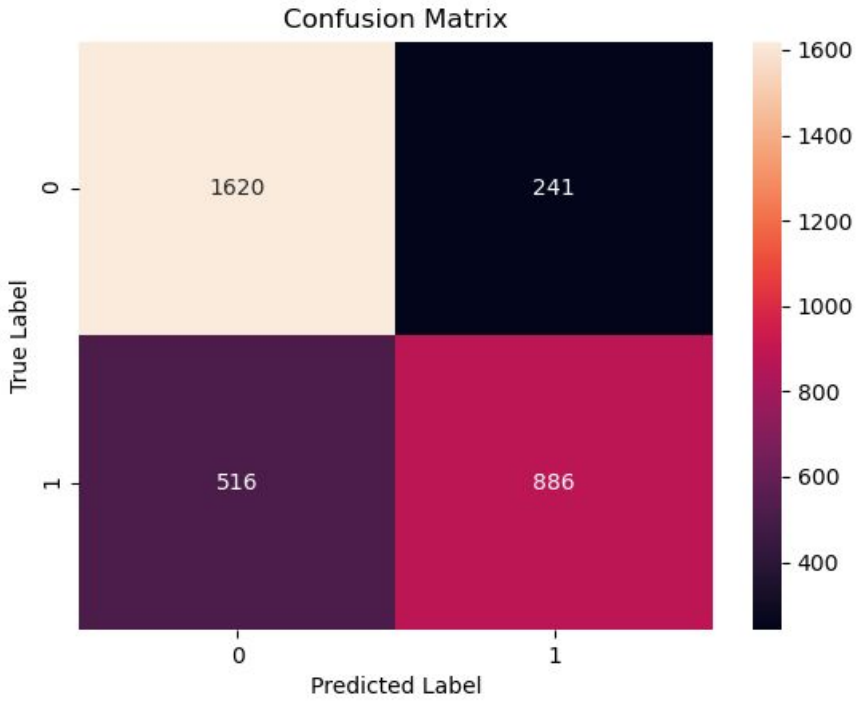


Figure 10. Confusion matrix for kNN using PCA dataset.

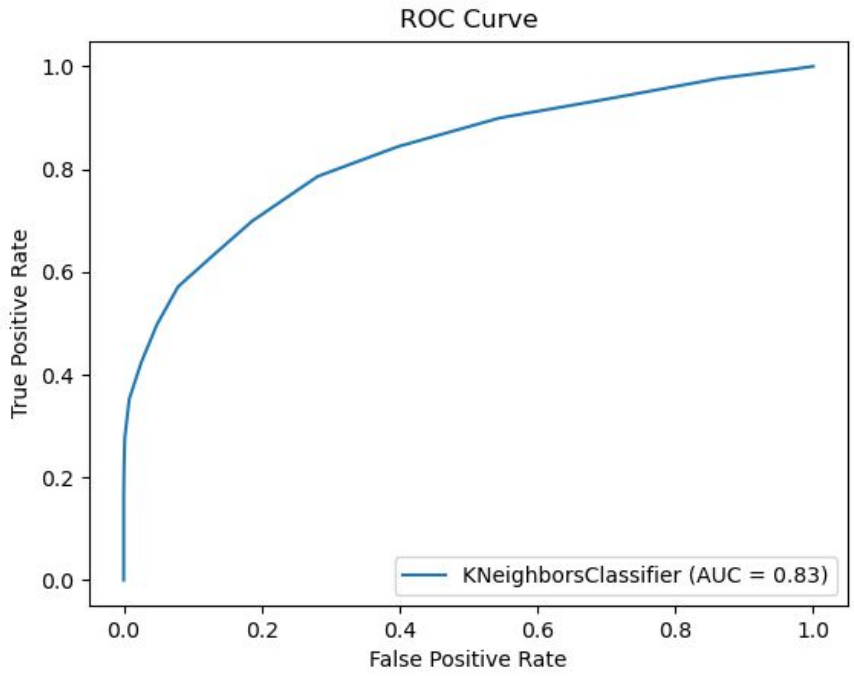


Figure 11. ROC Curve for kNN using PCA dataset.

FLD Reduced Dataset (1 attribute)

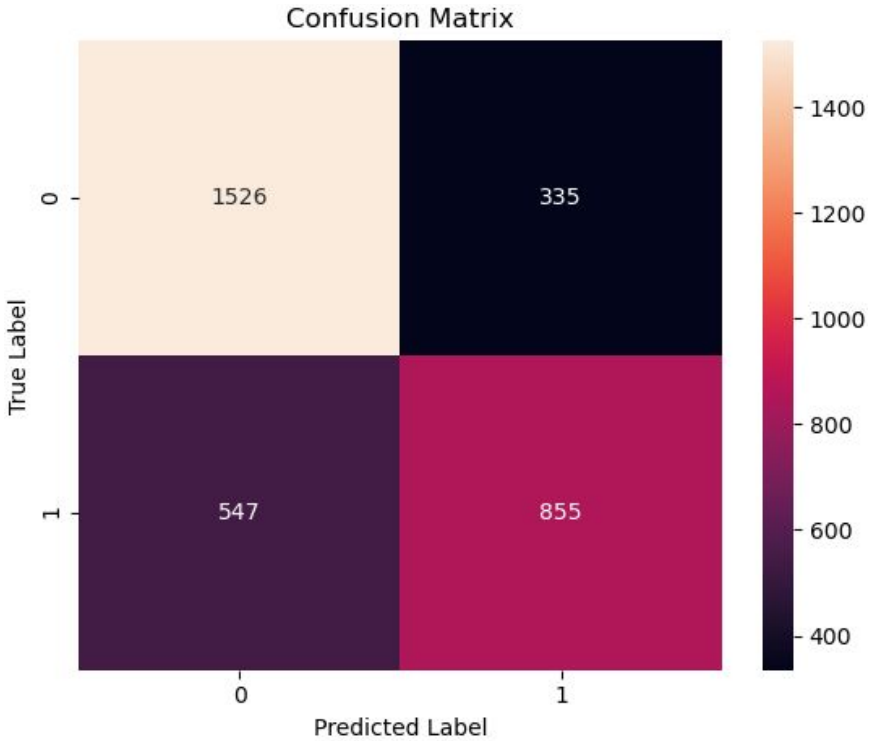


Figure 12. Confusion matrix for kNN using FLD reduced dataset.

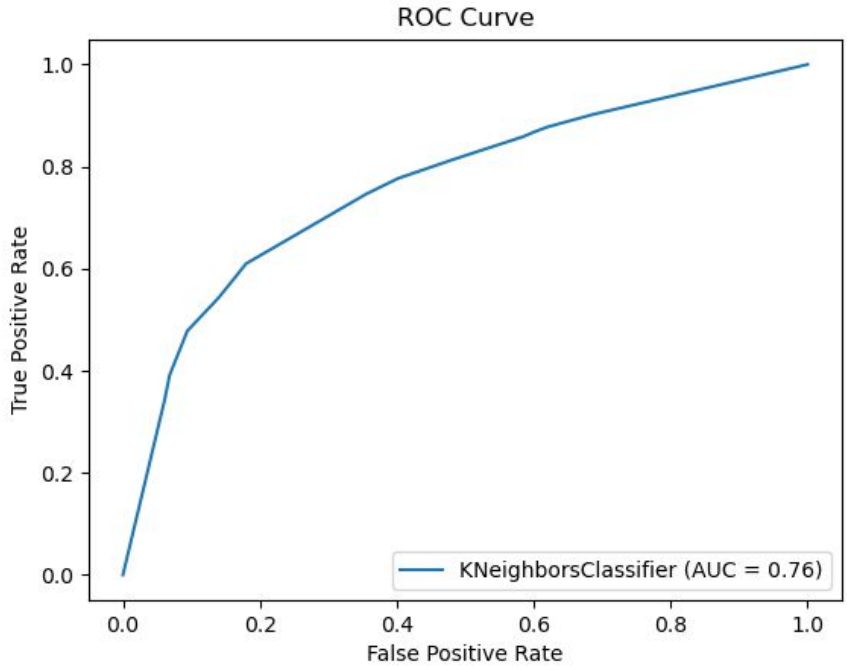


Figure 13. ROC Curve for kNN using FLD reduced dataset.

BPNN

Raw Dataset (19262 attributes)

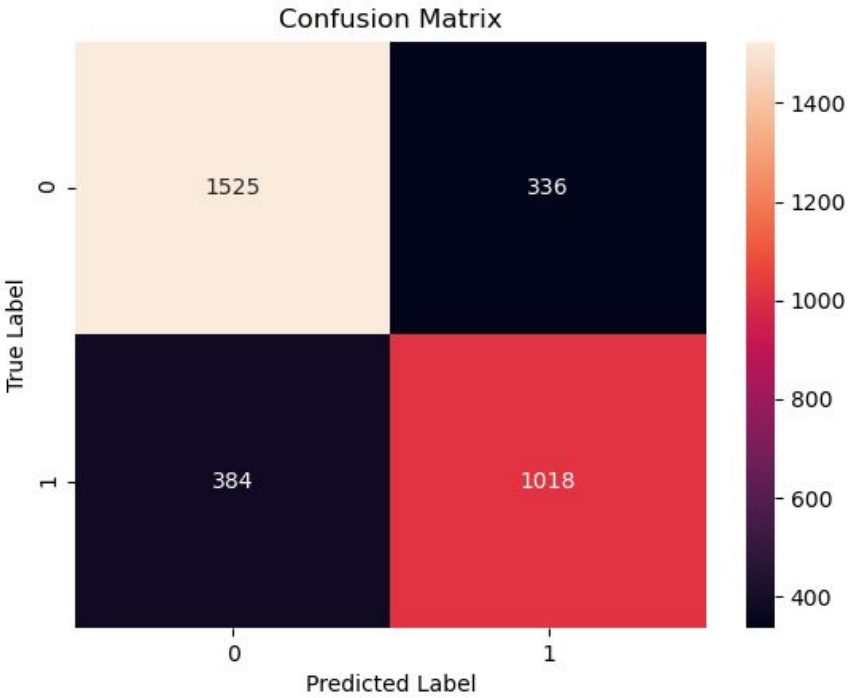


Figure 14. Confusion matrix for BPNN using raw dataset.

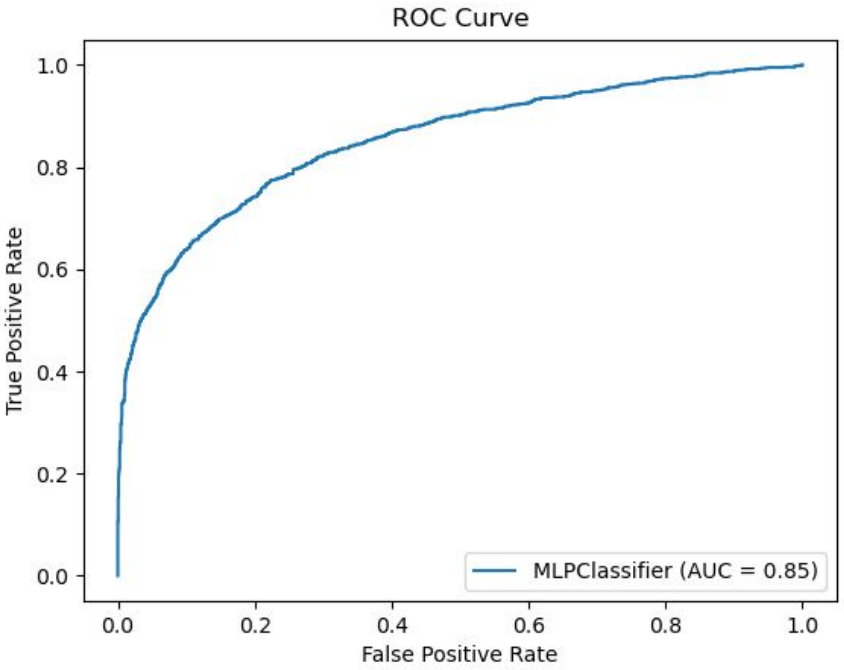


Figure 15. ROC Curve for BPNN using raw dataset.

PCA Reduced Dataset (2952 attributes)

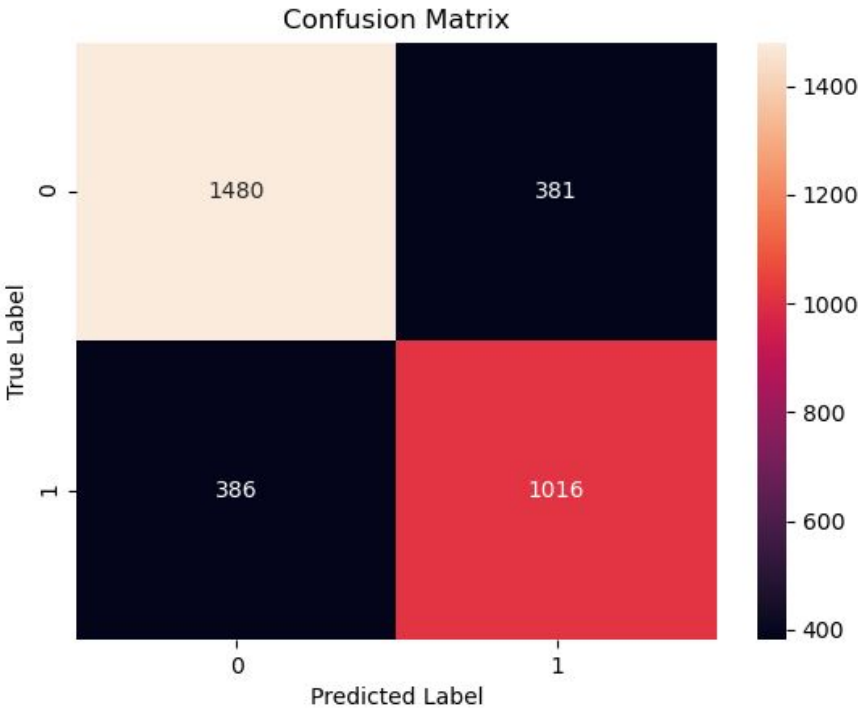


Figure 16. Confusion matrix for BPNN using PCA reduced dataset.

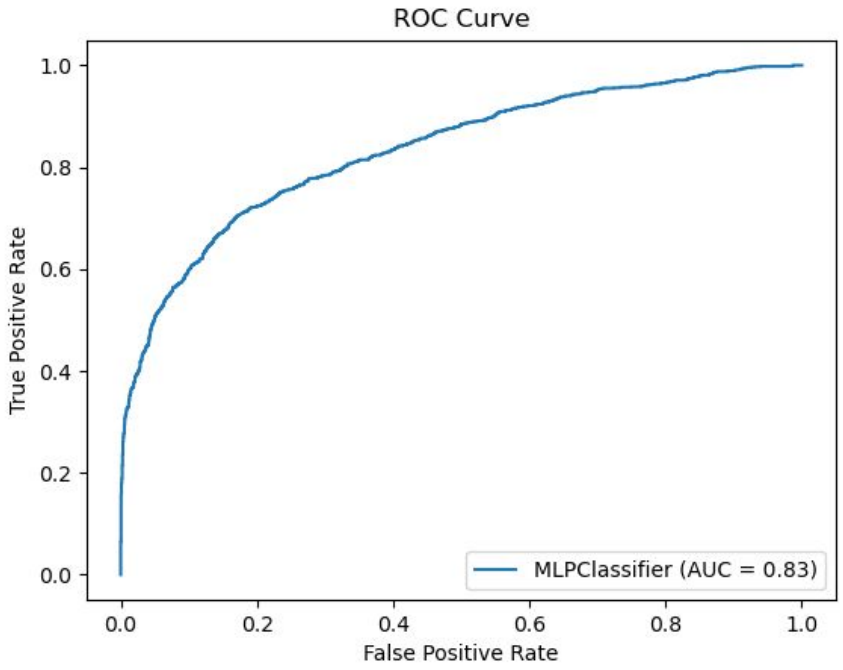


Figure 17. ROC Curve for BPNN using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

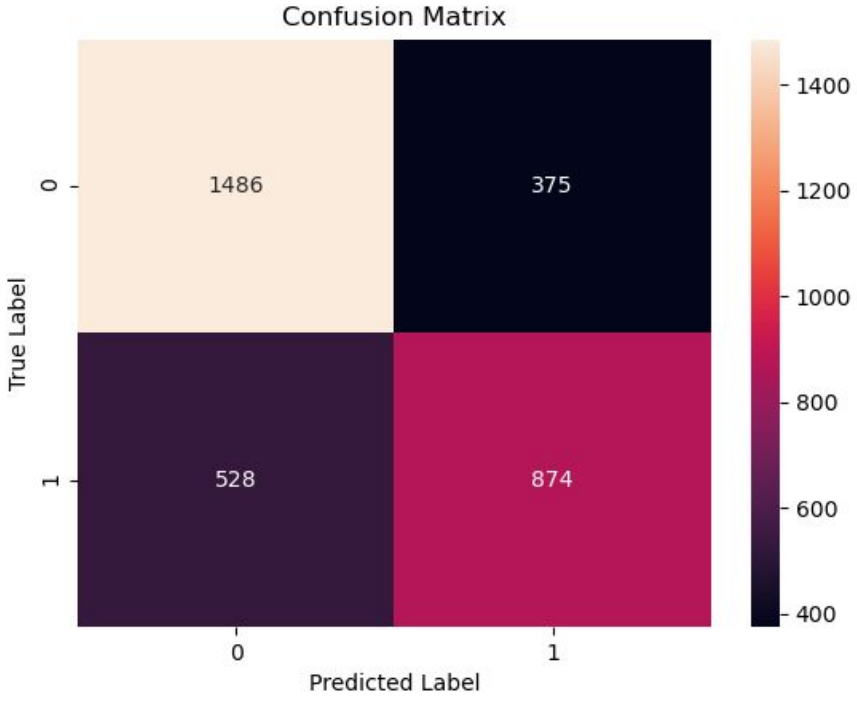


Figure 18. Confusion matrix for BPNN using FLD reduced dataset.

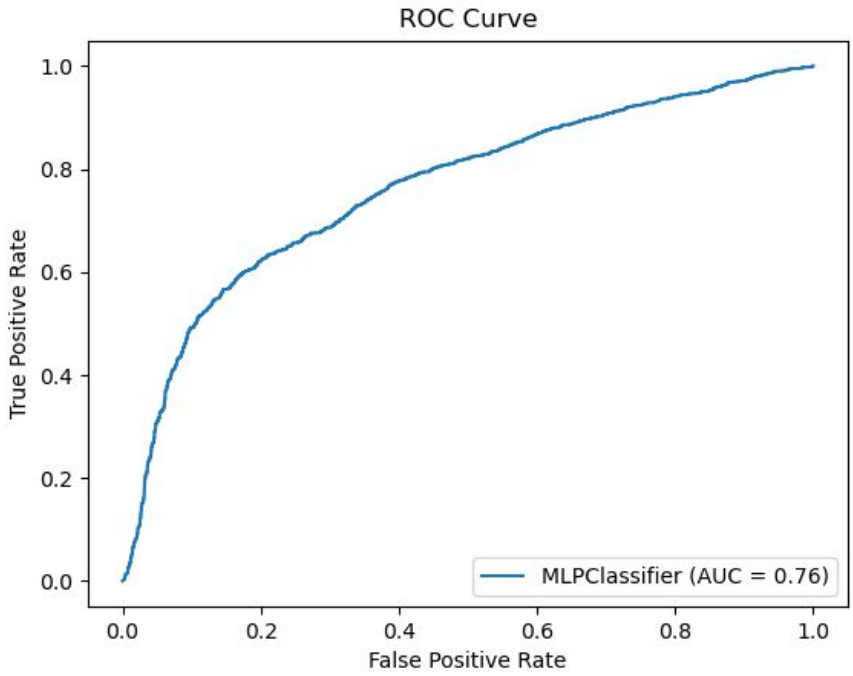


Figure 19. ROC Curve for BPNN using FLD reduced dataset.

Random Forest

Raw Dataset (19262 attributes)

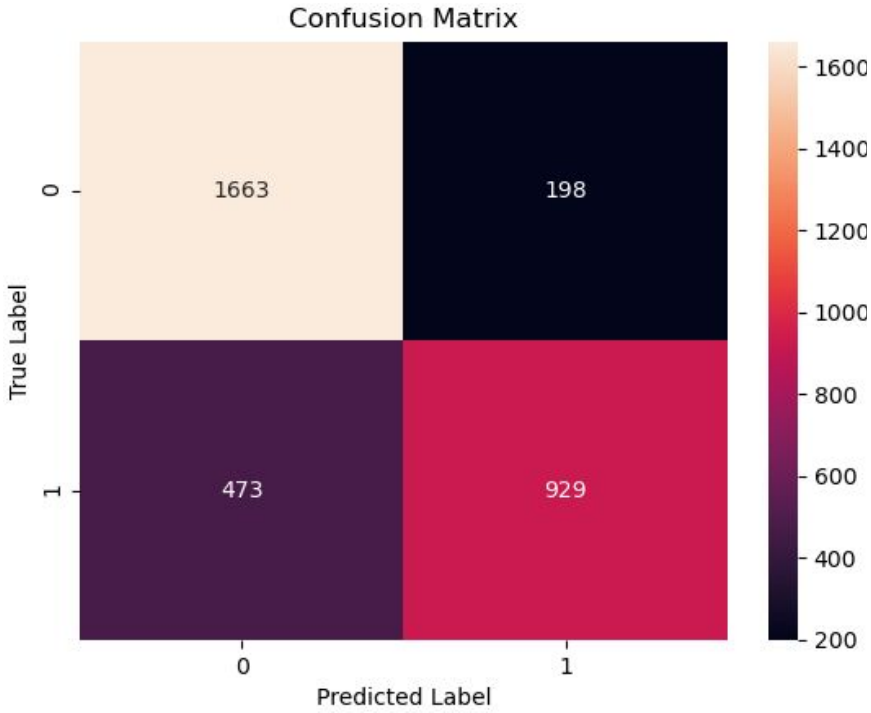


Figure 20. Confusion matrix for RF using raw dataset.

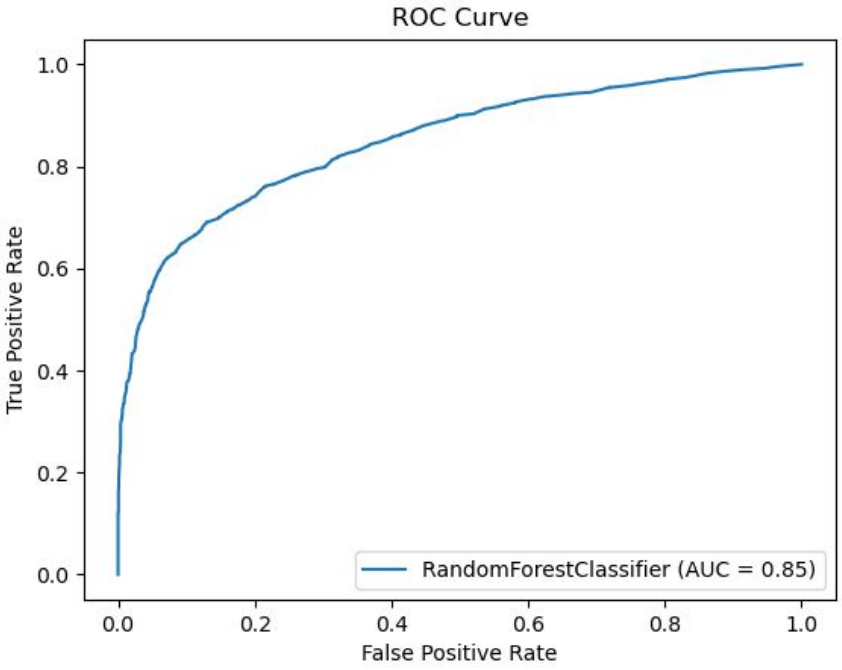


Figure 21. ROC Curve for RF using raw dataset.

PCA Reduced Dataset (2952 attributes)

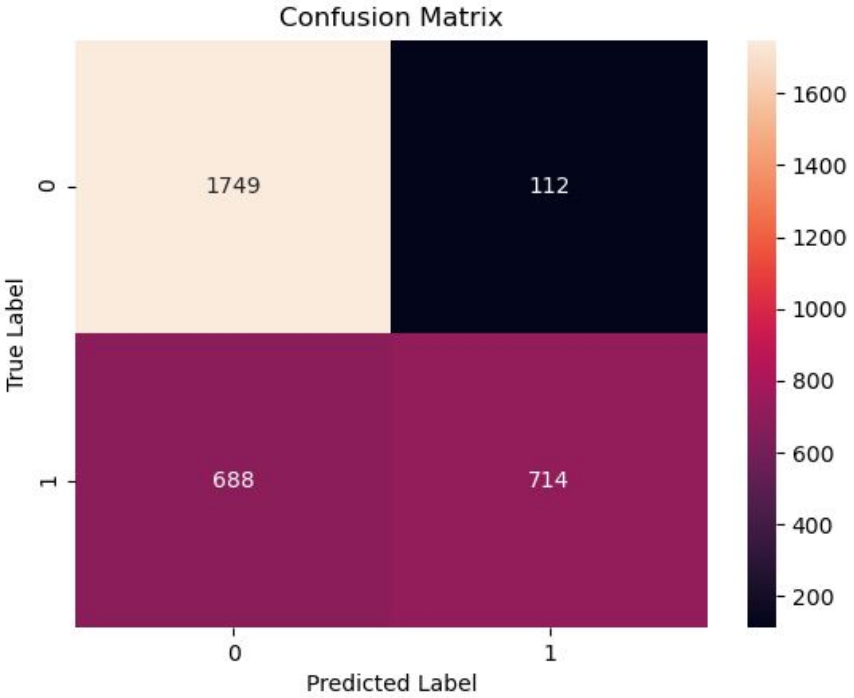


Figure 22. Confusion matrix for RF using PCA reduced dataset.

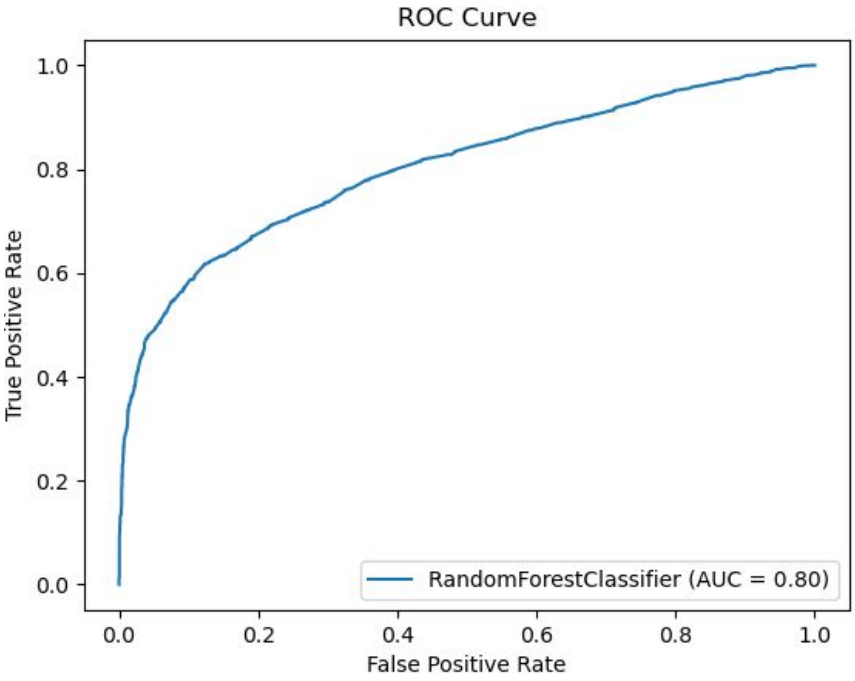


Figure 23. ROC Curve for RF using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

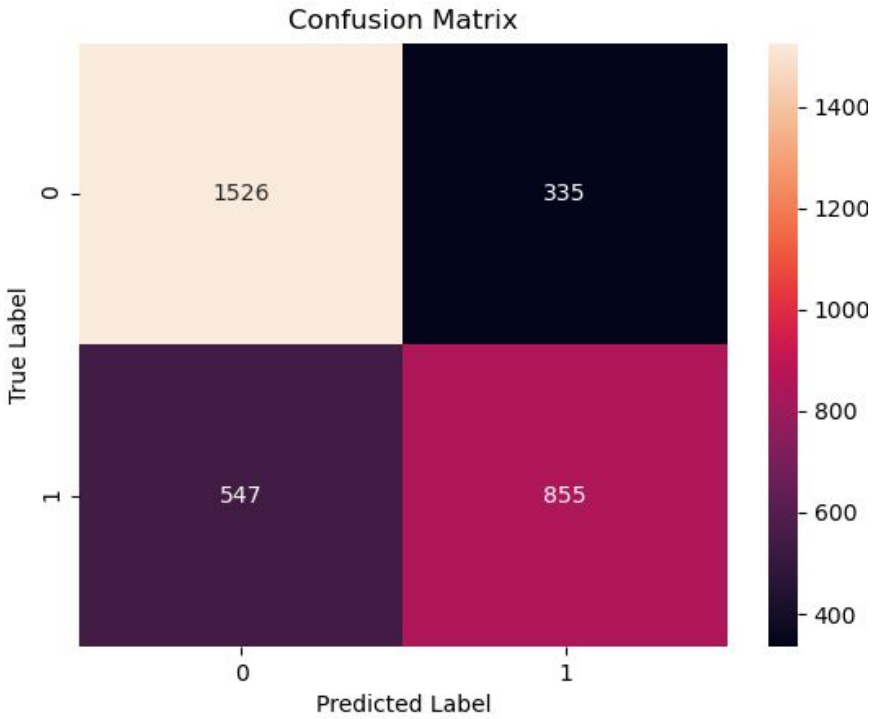


Figure 24. Confusion matrix for RF using FLD reduced dataset.

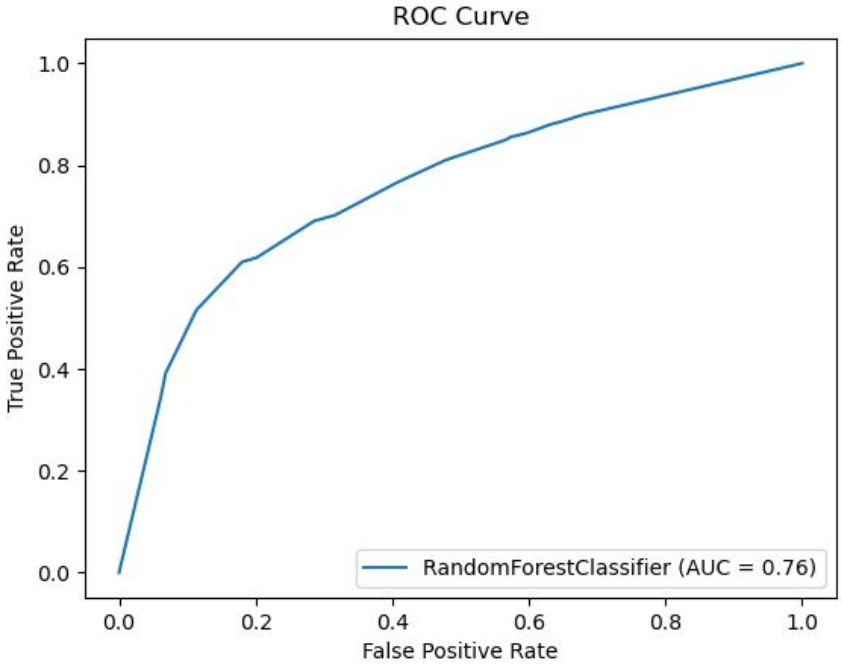


Figure 25. ROC Curve for RF using FLD reduced dataset.

SVM

Raw Dataset (19262 attributes)

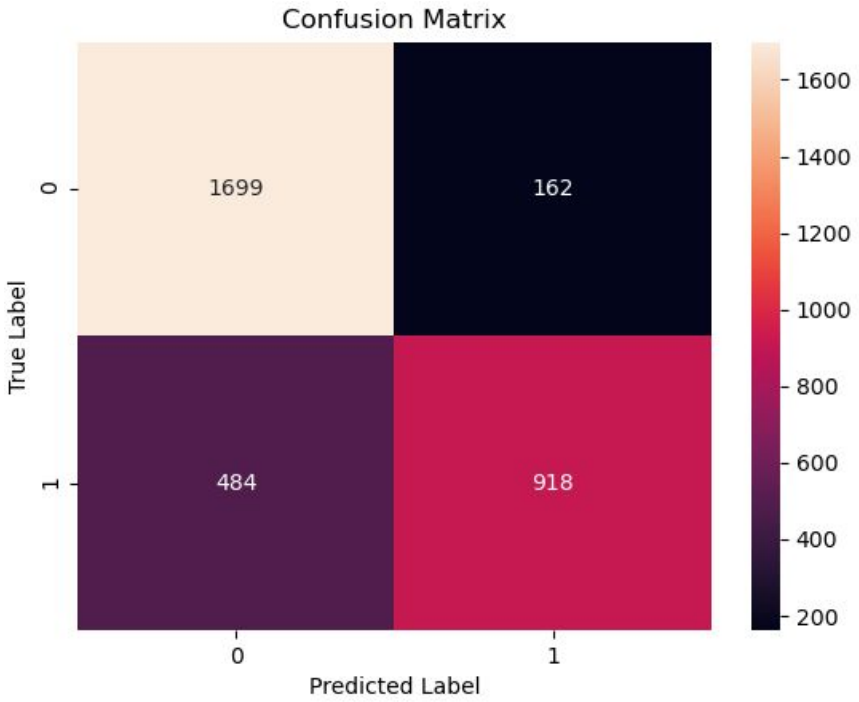


Figure 26. Confusion matrix for SVM using raw dataset.

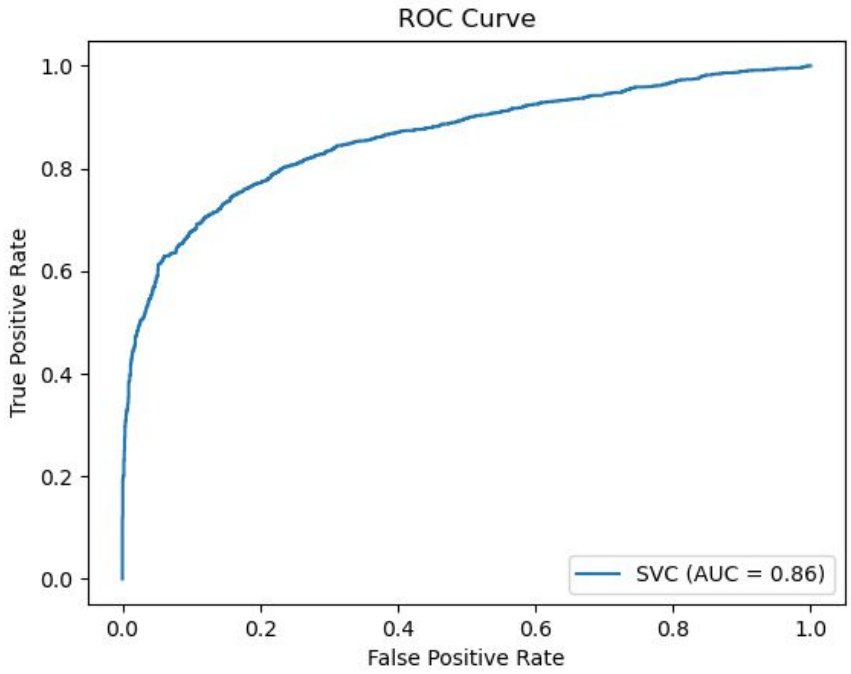


Figure 27. ROC Curve for SVM using raw dataset.

PCA Reduced Dataset (2952 attributes)

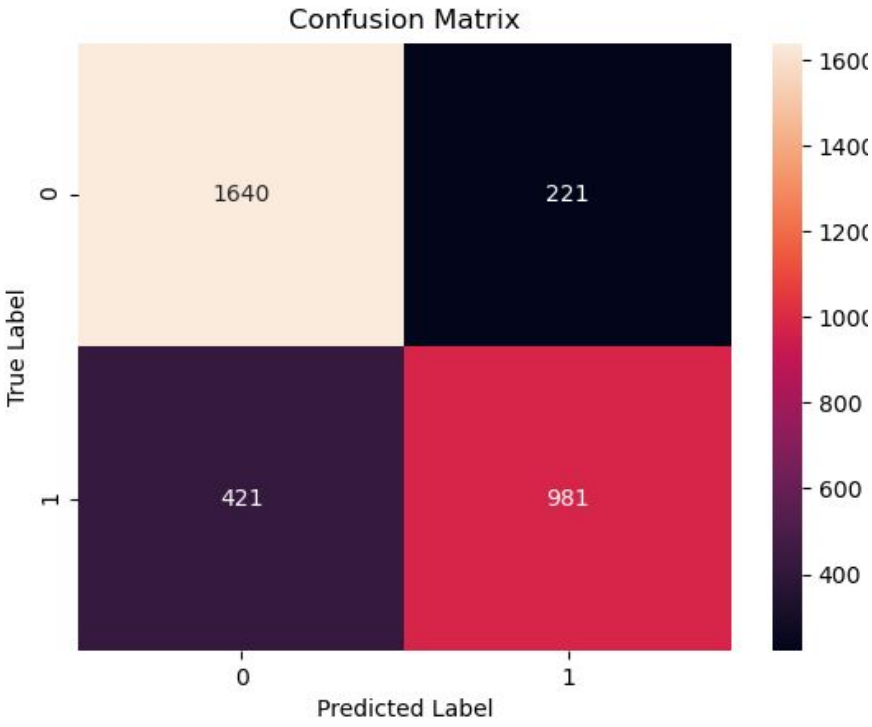


Figure 28. Confusion matrix for SVM using PCA reduced dataset.

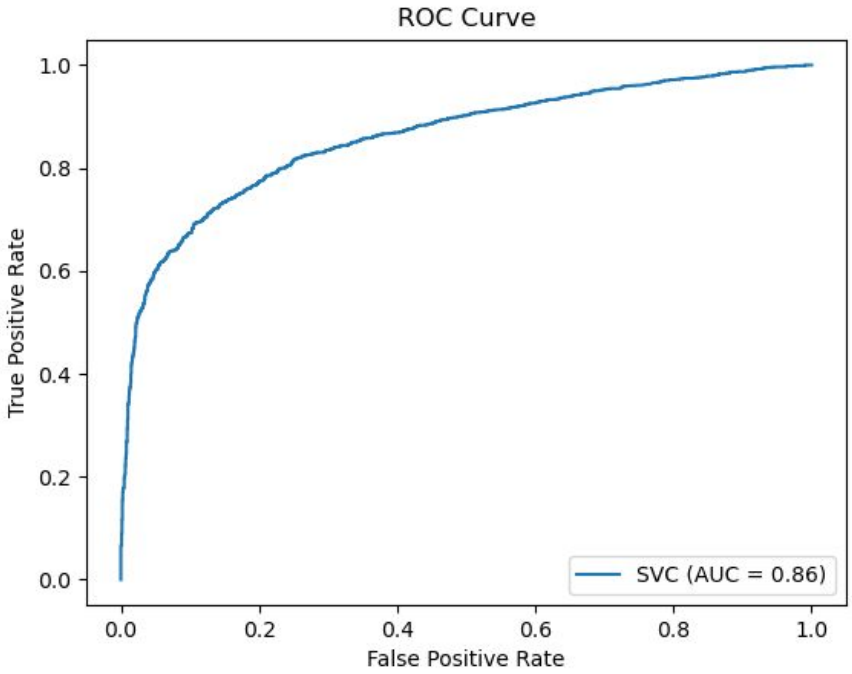


Figure 29. ROC Curve for SVM using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

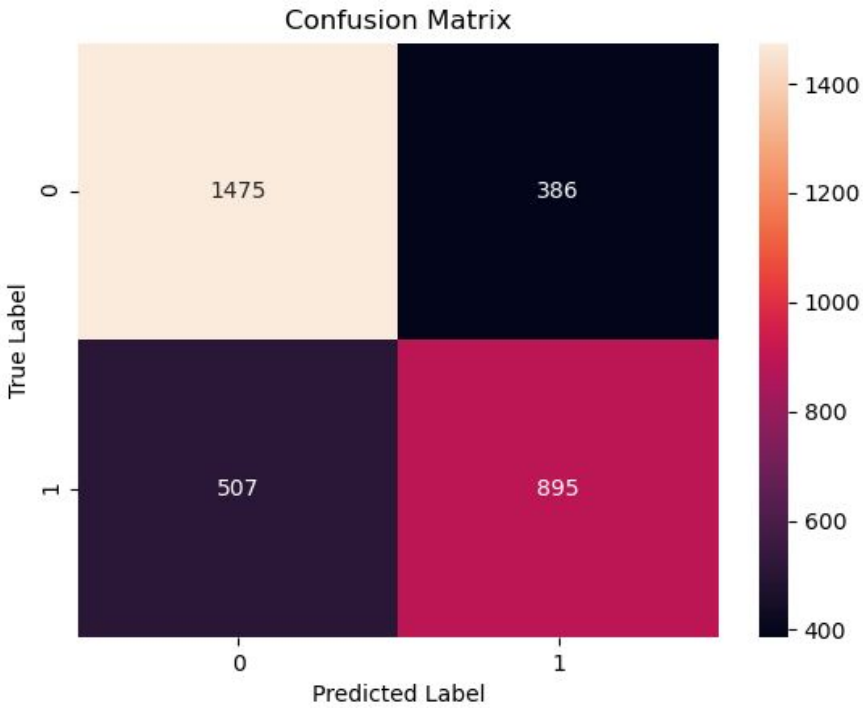


Figure 30. Confusion matrix for SVM using FLD reduced dataset.

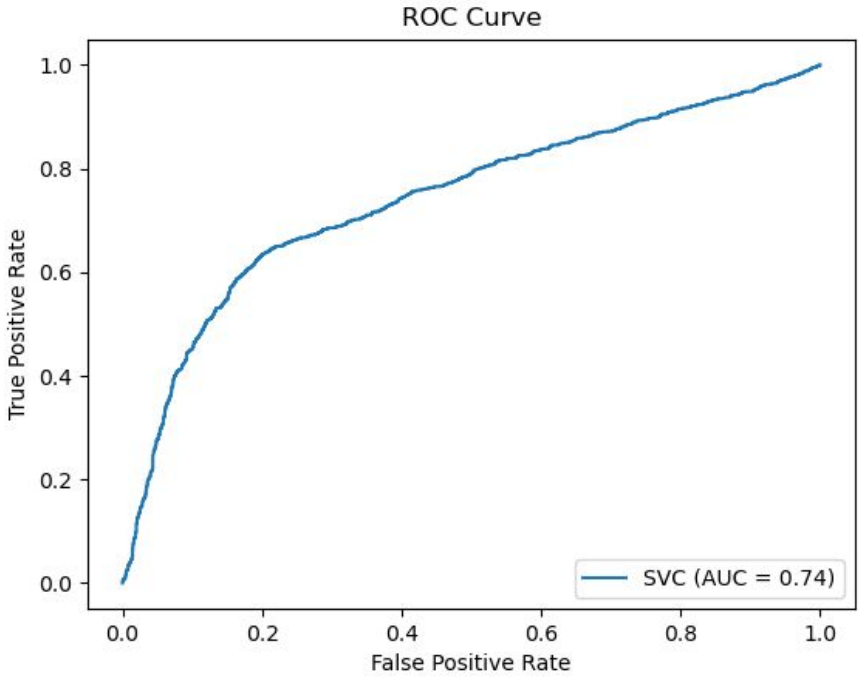


Figure 31. ROC Curve for SVM using FLD reduced dataset.

KMeans

Raw Dataset (19262 attributes)

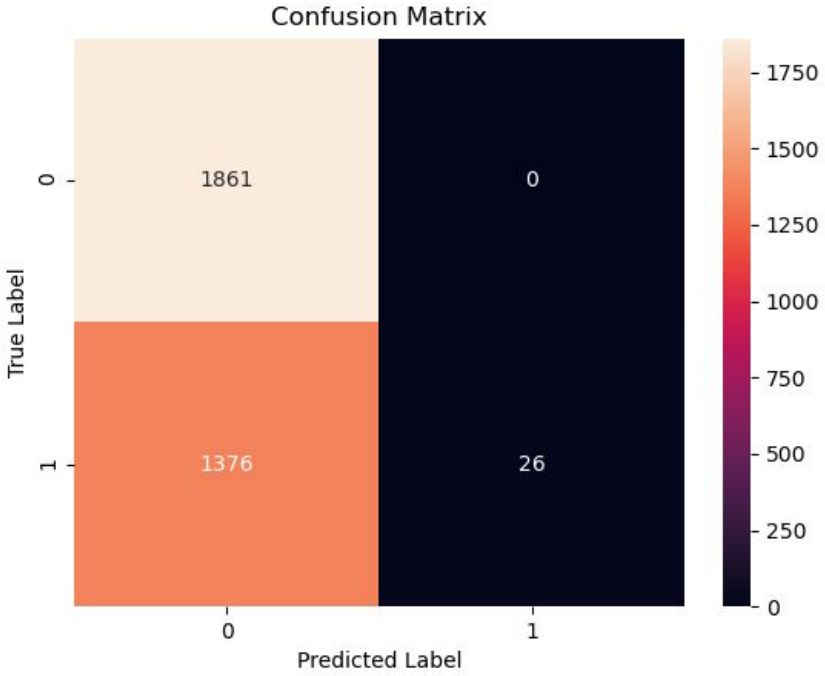


Figure 32. Confusion matrix for KMeans using raw dataset.

PCA Reduced Dataset (2952 attributes)

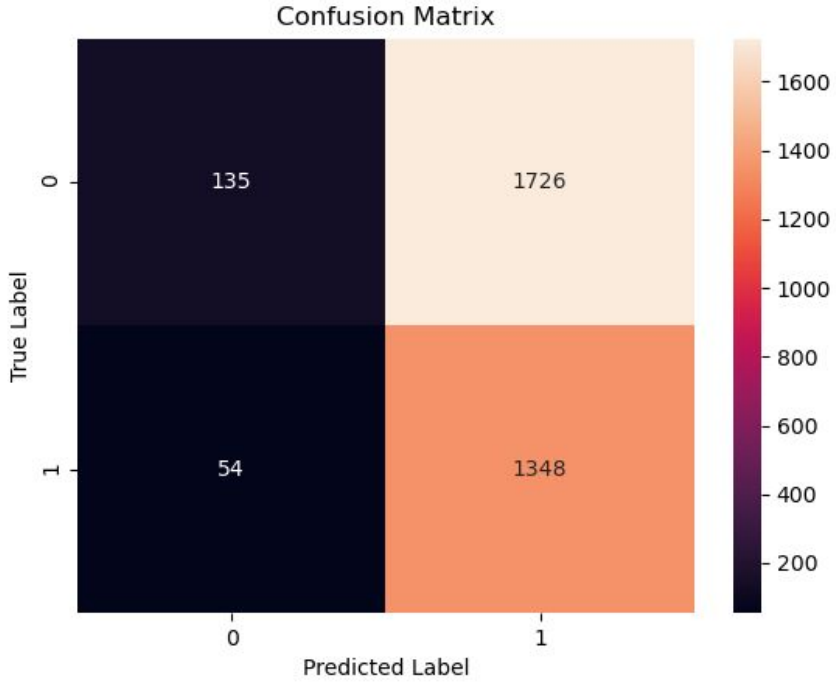


Figure 33. Confusion matrix for KMeans using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

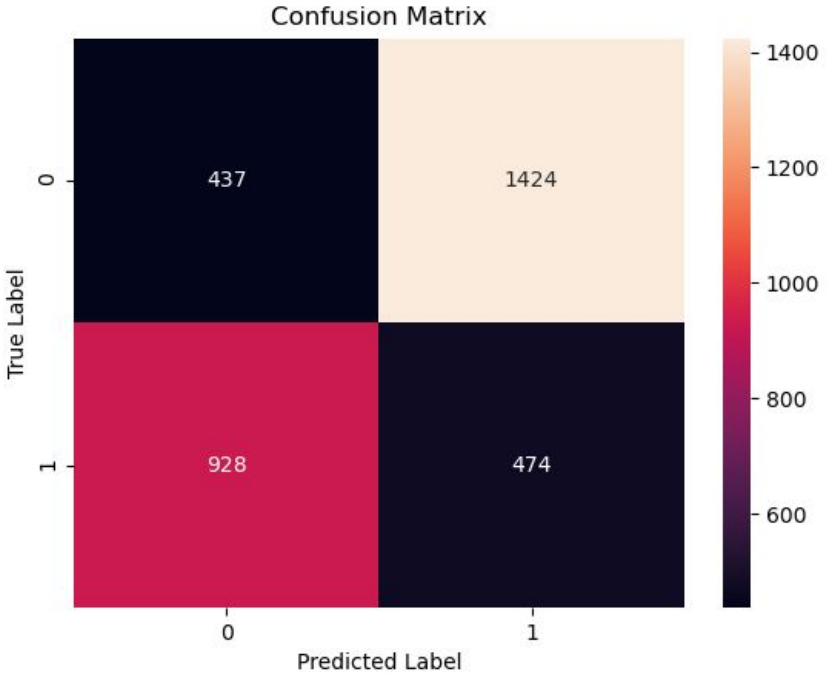


Figure 34. Confusion matrix for KMeans using FLD reduced dataset.

WTA

Raw Dataset (19262 attributes)

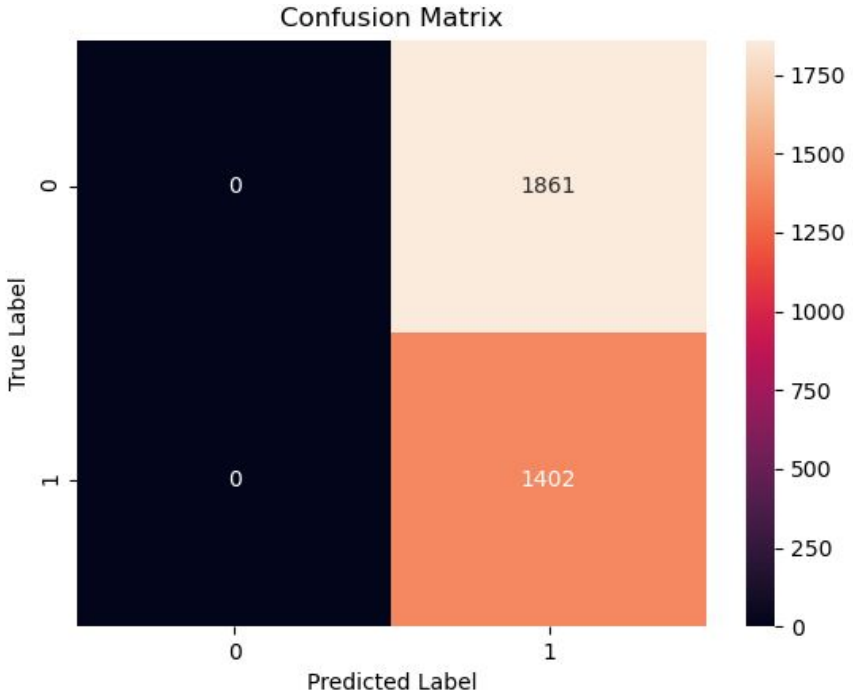


Figure 35. Confusion matrix for WTA using raw dataset.

PCA Reduced Dataset (2952 attributes)

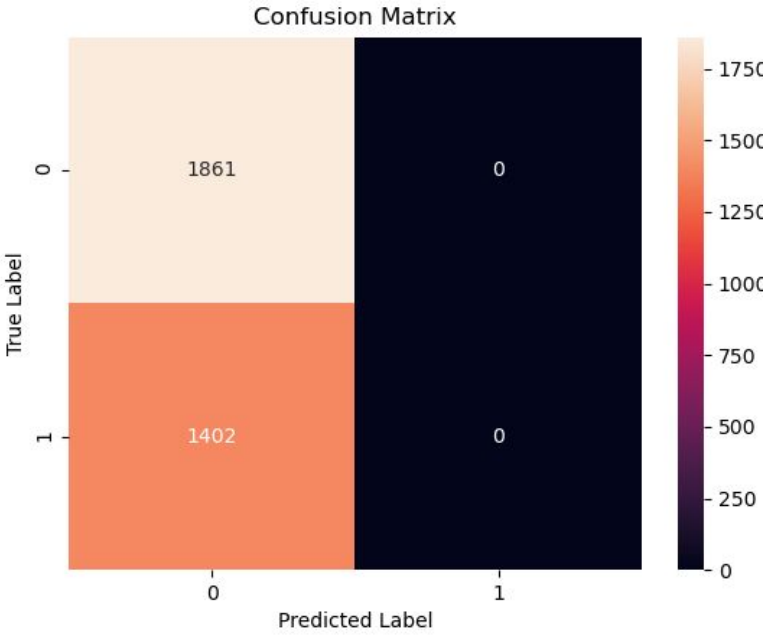


Figure 36. Confusion matrix for WTA using PCA reduced dataset.

FLD Reduced Dataset (1 attribute)

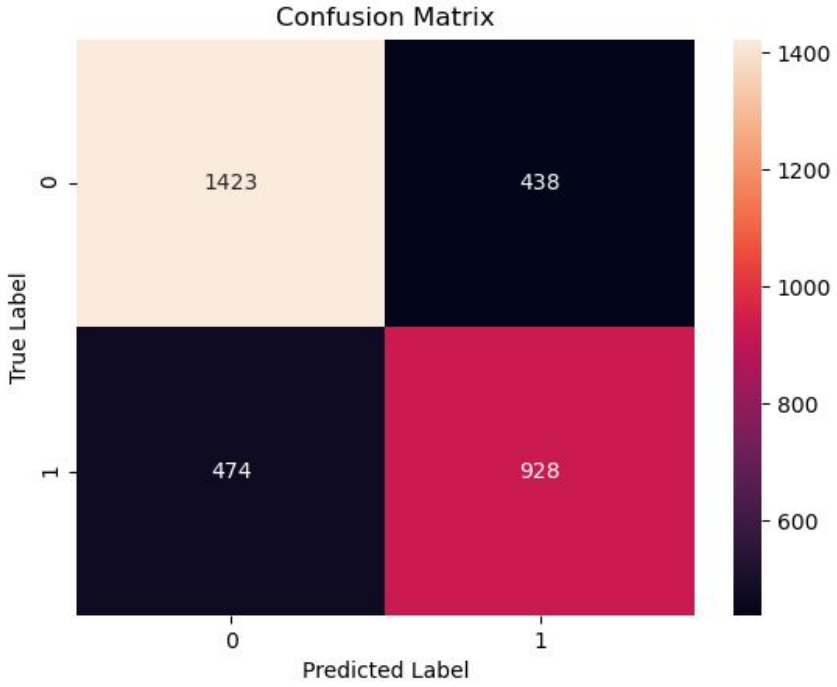
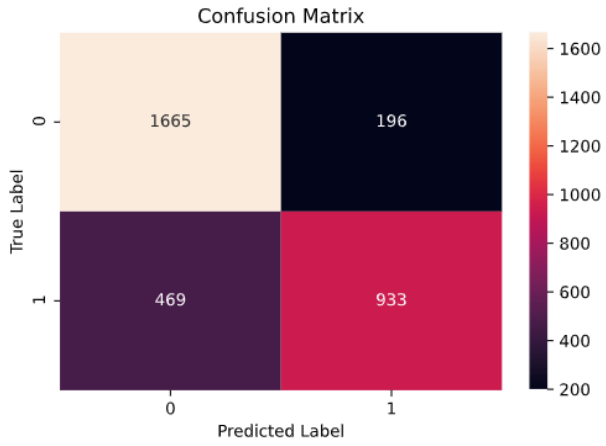


Figure 37. Confusion matrix for WTA using FLD reduced dataset.

Fusion

Raw Dataset (19262 attributes)

Majority Voting



Naive Bayes

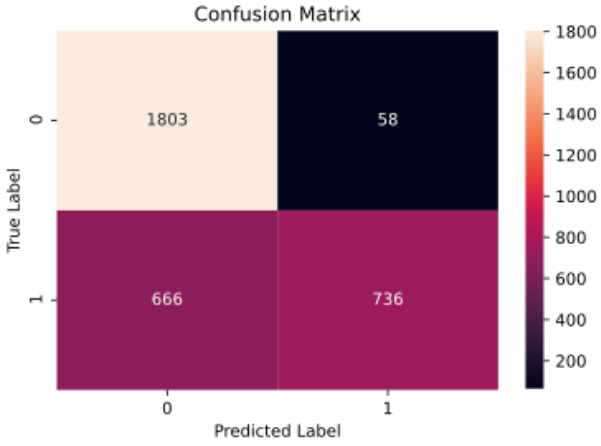


Figure 38. Confusion matrix for Fusion using Majority Voting and NB using raw dataset.

PCA Reduced Dataset (2952 attributes)

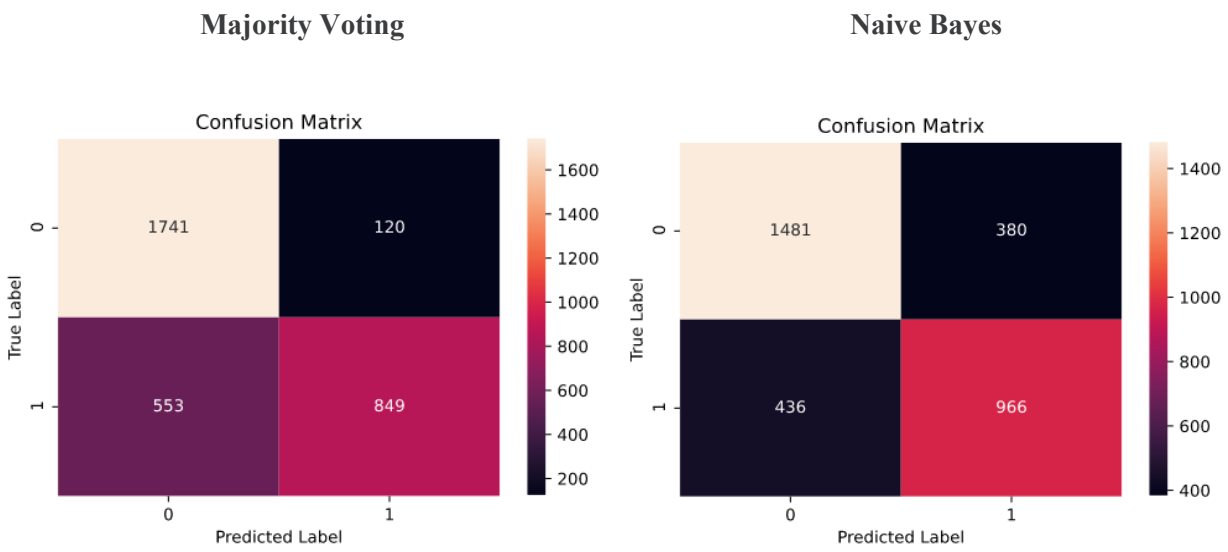


Figure 39. Confusion matrix for Fusion using Majority Voting and NB on PCA reduced data set.

FLD Reduced Dataset (1 attribute)

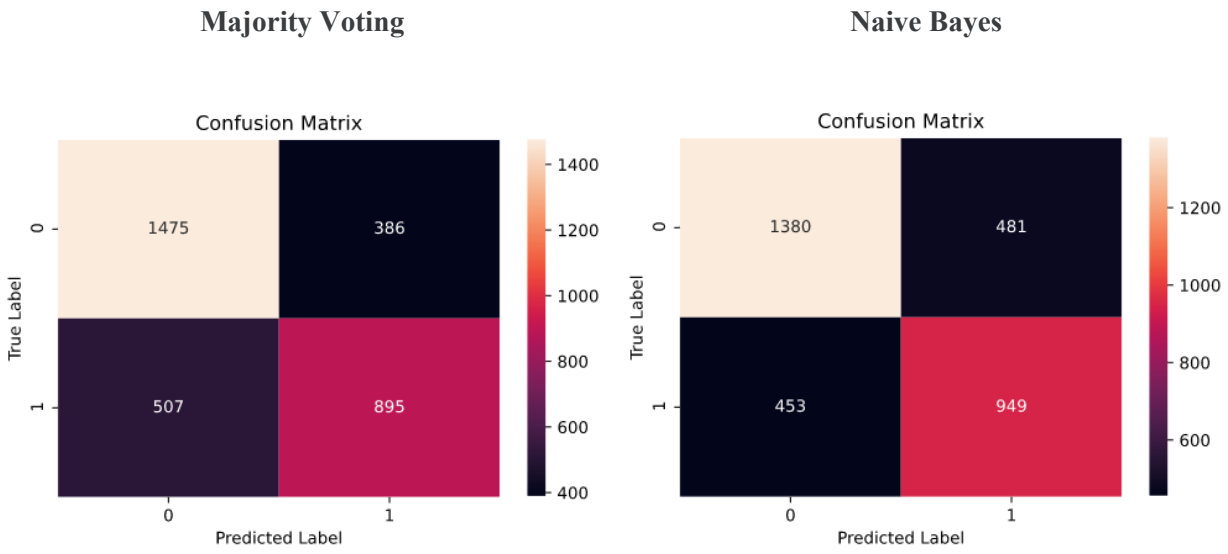


Figure 40. Confusion matrix for Fusion using Majority Voting and NB on FLD reduced data set.

Overall Results

Precision

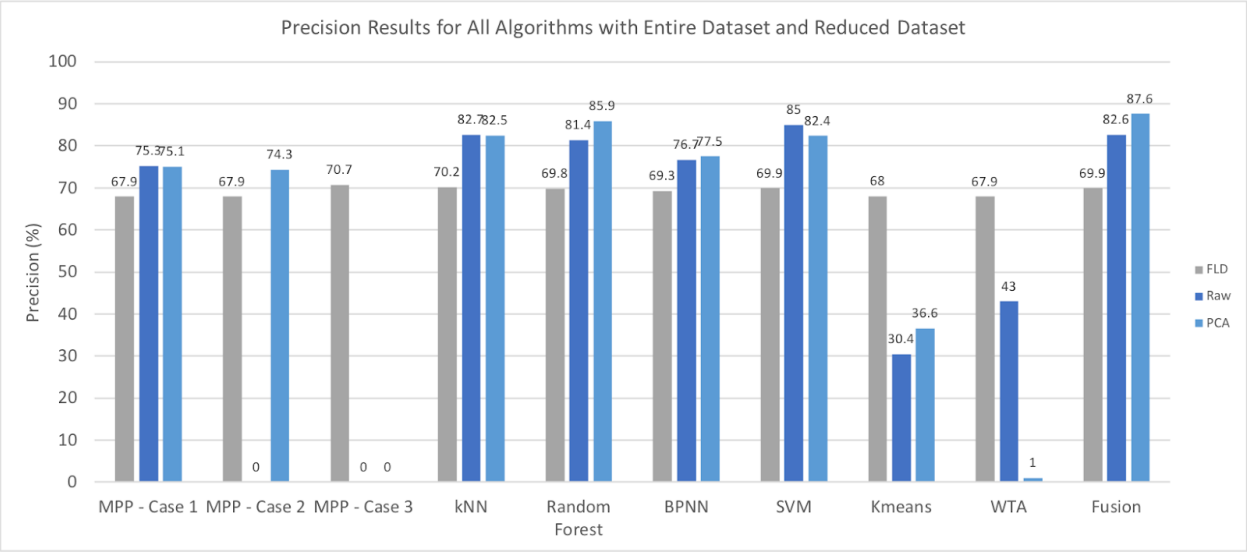


Figure 41. Precision obtained from all learning models applied to the raw data set and reduced versions of the dataset obtained using FLD and PCA.

Recall

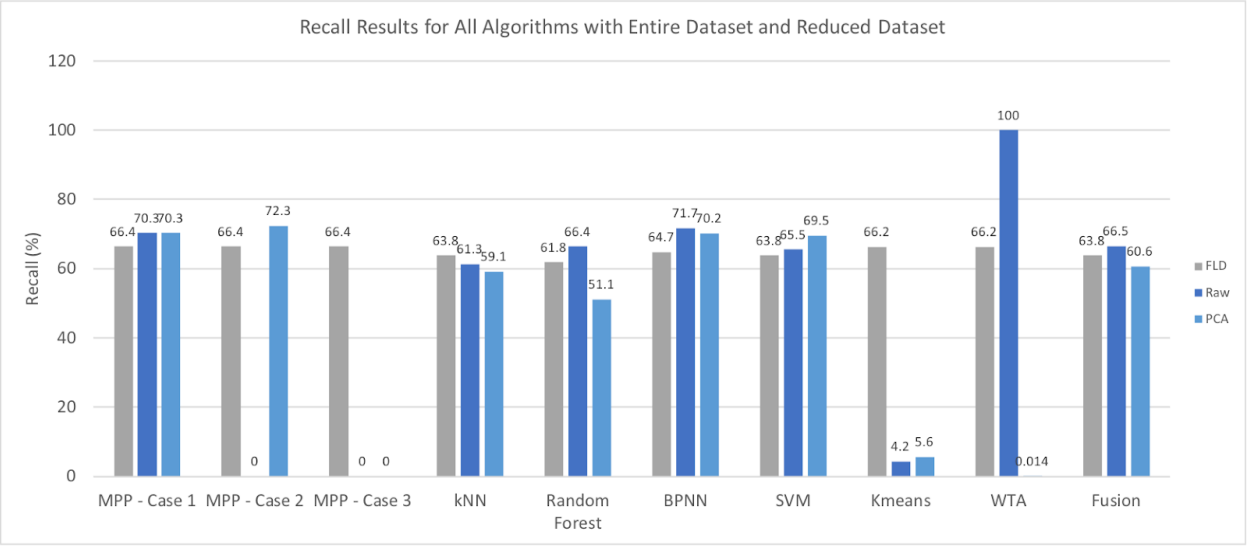


Figure 42. Recall obtained from all learning models applied to the raw data set and reduced versions of the dataset obtained using FLD and PCA.

F1-Score

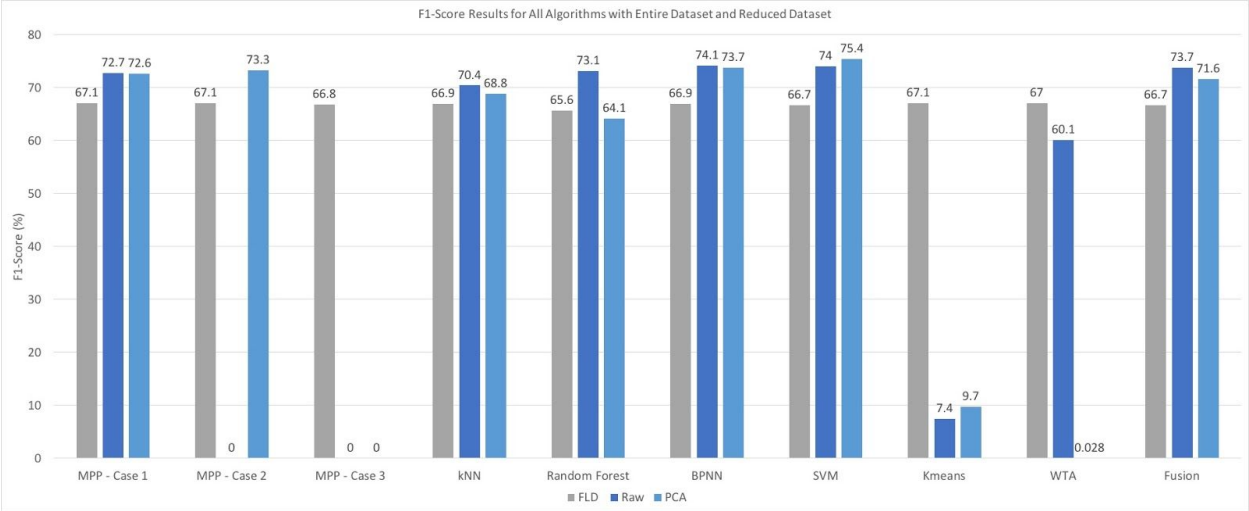


Figure 43. F-1 score for all learning models using the raw data set and reduced versions of the dataset obtained using FLD and PCA.

Overall Accuracy

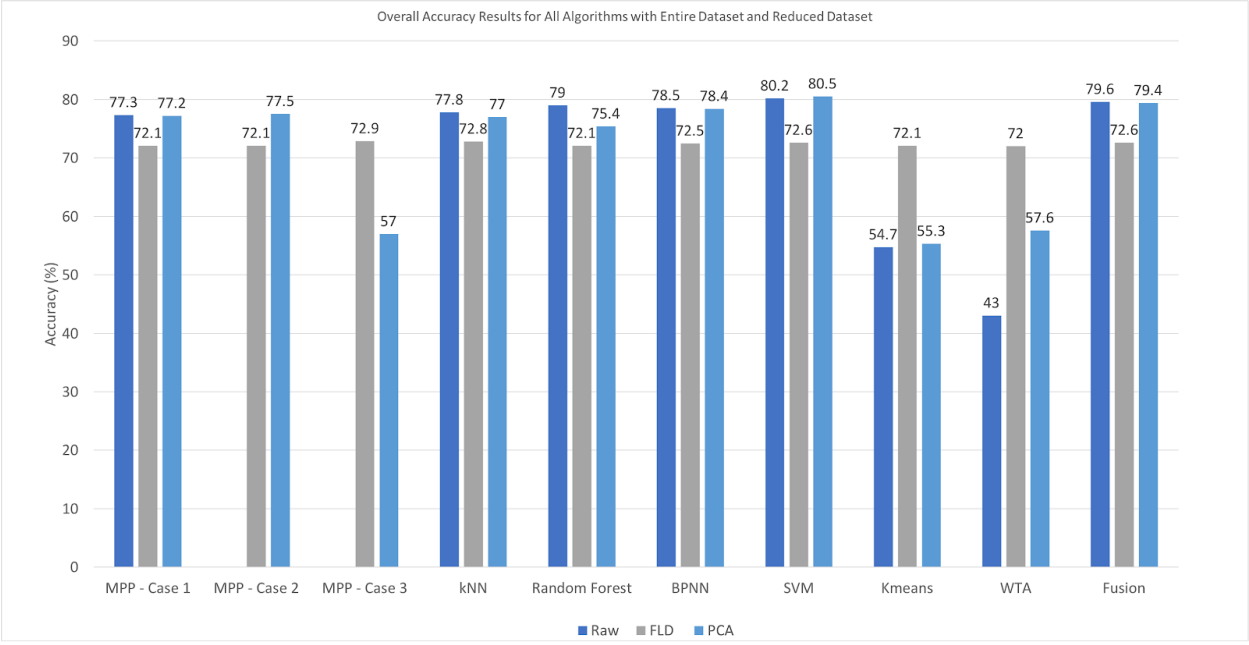


Figure 44. Overall accuracy for all learning models using the raw data set and reduced versions of the dataset obtained using FLD and PCA.

Execution Time

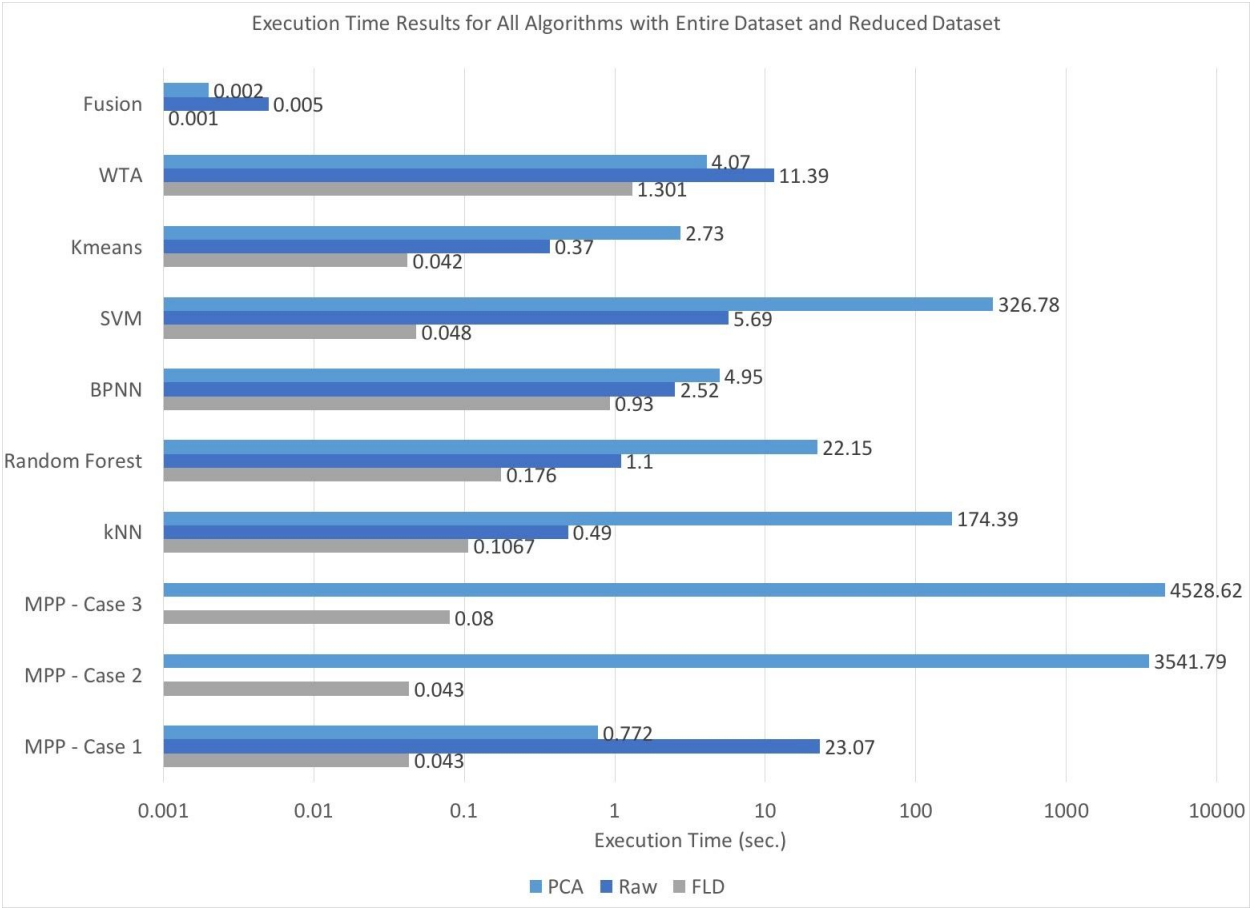


Figure 45. Execution time for each learning model when applied to the raw, PCA-reduced, and FLD-reduced data sets. The *x*-axis is in log scale to better visualize the magnitude differences.

Conclusion

As observed in the previous section, the highest accuracy was achieved using SVM on the reduced data set closely followed by SVM on the raw data set. The overall accuracy was 80.5% and 80.2%, respectively. There are significant performance differences between these two learning models. While SVM on the raw data set provides slightly better accuracy, it also takes approximately 57.4x more time to complete. Winner-takes-all (WTA) resulted in the worst overall accuracy reaching only 43%. In addition, the performance of the algorithm was also slower than models that provide better accuracy. For instance, WTA using the raw data set was 2x slower than SVM. The BERT model was capable of achieving 85% accuracy with an execution time of roughly 223 minutes [12]. Although our implementations were not able to outperform the state-of-the-art model, SVM does provide fairly close results, and with more tweaking and more fusion testing, could potentially perform as well as the BERT model.

Source Code

https://github.com/TannerFry/CS522-Final-Project

References

[1] Shetty, Badreesh. "Natural Language Processing(NLP) for Machine Learning." *Medium*, Towards Data Science, 24 Nov. 2018, towardsdatascience.com/natural-language-processing-nlp-for-machine-learning-d44498845d5b

[2] "Real or Not? NLP with Disaster Tweets." *Kaggle*, www.kaggle.com/c/nlp-getting-started/data.

[3] "Sklearn.neighbors.KNeighborsClassifier." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[4] "Sklearn.neural_network.MLPClassifier." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[5] "3.2.4.3.1. Sklearn.ensemble.RandomForestClassifier." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[6] "Sklearn.svm.SVC." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[7] "Sklearn.cluster.KMeans." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[8] "Sklearn.decomposition.PCA." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[9] "Sklearn.discriminant_analysis.LinearDiscriminantAnalysis." *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html.

[10] Horev, Rani. "BERT Explained: State of the Art Language Model for NLP." *Medium*, Towards Data Science, 17 Nov. 2018, towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

[11] Devlin, J., Chang, M., Lee, K., Toutanova, K.,"BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018, http://arxiv.org/abs/1810.04805

[12] nxhong93. “tweet_predict1.” *Kaggle*, Kaggle, 23 Mar. 2020, www.kaggle.com/nxhong93/tweet-predict1

[13] “Sklearn.naive_bayes.GaussianNB¶.” *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html.

[14] “Sklearn.cluster.MiniBatchKMeans¶.” *Scikit*, scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html.