Individual Coursework Submission Form

Specialist Masters Programme

Surname: Chawla	**First Name: Sarthak**
MSc in: Business Analytics	**Student ID number: 220042213**
Module Code: SMM636	
Module Title: Machine Learning	
Lecturer: Dr Rui Zhu	**Submission Date: 24/03/2023**

Declaration:

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the coursework instructions and any other relevant programme and module documentation. In submitting this work, I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

We acknowledge that work submitted late without a granted extension will be subject to penalties, as outlined in the Programme Handbook. Penalties will be applied for a maximum of five days lateness, after which a mark of zero will be awarded.

Marker's Comments (if not being marked on-line):

Deduction for Late Submission:

Final Mark: %

Q1 For the first question, we begin by importing data from GermanCredit.csv into our Jupiter notebook. Furthermore, we,

- Split the data into features (X) and target (Y): The dataset is split into two parts. The 'X' contains all the columns except for the target column 'Class', which includes the input features, and the 'Y' contains the target column, which is what we want to predict.

- Split the data into training and test sets: The dataset is then split into training and test sets, where the test size is set to 30% and the random state is set to 42. This ensures that the same random sample is used each time the code is executed.

(A) In order to Fit a decision tree to the training data with an optimal tree size determined by 5-fold cross-validation, the following steps are performed,

- Defining the decision tree classifier: A **DecisionTreeClassifier** object is created with the random state set to 42.

- Defining the grid of hyperparameters to search: A dictionary of hyperparameters to search is defined, including the criterion for splitting, the maximum depth of the tree, the minimum number of samples required to split an internal node, and the minimum number of samples required to be at a leaf node.

- Defining the grid search object with 5-fold cross-validation: A **GridSearchCV** object is defined with the decision tree classifier, the hyperparameters to search, and 5-fold cross-validation.

- Fitting the grid search object to the training data: The grid search object is then fit to the training data to find the best hyperparameters.

- Printing the best hyperparameters and corresponding cross-validation score: The best hyperparameters found during the grid search are printed along with the corresponding cross-validation score. The cross-validation score represents the mean accuracy of the model on the different cross-validation splits of the training set.

Outcomes:

The output of the code shows that the best hyperparameters for the decision tree classifier on the German credit data were:

```
Best hyperparameters: {'criterion': 'gini', 'max_depth': 7, 'min_samples_leaf': 7, 'min_samples_split': 20}
Cross-validation score: 0.72
```

These hyperparameters were found using grid search with 5-fold cross-validation.
The **criterion** parameter refers to the function used to measure the quality of a split, and in this case, the **gini** criterion was found to be the best. The **max_depth** parameter refers to

the maximum depth of the decision tree, and the best value found was 7. This means that the tree can have a maximum depth of 7 levels. The **min_samples_leaf** parameter refers to the minimum number of samples required to be at a leaf node, and the best value found was 7. The **min_samples_split** parameter refers to the minimum number of samples required to split an internal node, and the best value found was 20.

The cross-validation score of 0.72 indicates that the mean accuracy of the model on the different cross-validation splits of the training set was 72%. This means that, on average, the model predicted the class of 72% of the credit applications correctly. This is a moderate accuracy score, but it may be improved by using other algorithms, increasing the dataset size, or applying other feature engineering techniques.

Next, the decision tree classifier is pruned using grid search. The steps are as follows:

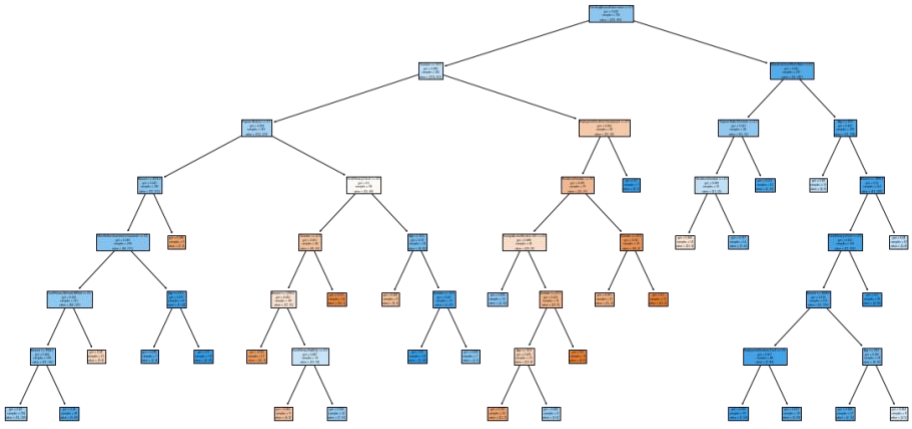
- Prune the decision tree: Once we have identified the best hyperparameters using the grid search method, the decision tree is pruned by creating a new instance of the **DecisionTreeClassifier** class and setting the hyperparameters to the values found by grid search. This new decision tree is then trained using the training set.

- Plot the pruned decision tree: The pruned decision tree is then plotted using the **plot_tree()** method from the matplotlib library. This will provide a visual representation of the decision tree with the pruned nodes.

- Compute the test error rate of the pruned decision tree: Finally, the test error rate of the pruned decision tree is computed by calling **the score()** method of the decision tree on the test set. The test error rate is the fraction of misclassified examples in the test set. The lower the test error rate, the better the decision tree classifier is at generalizing to unseen data.

Outcomes:

The test error rate of the pruned decision tree classifier is 0.29, which means that approximately 29% of the test examples were misclassified.



```
Test error rate: 0.29000000000000004
```

(B) In the second part, a random forest is fitted to the training data with the parameter of the number of features to create the splits tuned by 5-fold cross-validation. The number of trees is set to 1000. The following steps were executed to do the same,

- Define the parameter grid: The first step is to define a dictionary of hyperparameters that we want to test. In this case, we define the **max_features** hyperparameter that controls the number of features used to create the splits in the trees of the random forest classifier.

- Define the random forest model: Next, a random forest classifier is defined using the **RandomForestClassifier** class from the scikit-learn library. The number of estimators is set to 1000 and the random state to 42 for reproducibility.

- Tune the hyperparameter using cross-validation: The next step is to tune the hyperparameter of the number of features used to create the splits using 5-fold cross-validation. This is done by creating an instance of the **GridSearchCV** class and passing the random forest classifier, the hyperparameter grid, and the number of folds for cross-validation as inputs. The grid search method finds the best combination of hyperparameters that minimize the cross-validation error.

- Get the best model and its parameters: Once the best hyperparameters are identifies using the grid search method, the best random forest classifier can be generated and its parameters by accessing the **best_estimator_** and **best_params_** attributes of the grid search object.

- Fit the tuned random forest: A new instance of the **RandomForestClassifier** class is created and set the hyperparameters to the best values found by grid search. This new random forest is then trained using the training set.

- Compute the test error rate of the random forest: Finally, the test error rate of the tuned random forest is computed by calling the **score()** method of the random forest on the test set. The test error rate is the fraction of misclassified examples in the test set. The lower the test error rate, the better the random forest classifier is at generalizing to unseen data.

Outcomes:

After tuning the model parameters, the final random forest model was fit to the training data using the optimal max_features value of 5 and 1000 decision trees. The test error rate of this tuned random forest model on the previously unseen test data was 0.24 or 24%, which means that the model correctly classified 76% of the test cases.

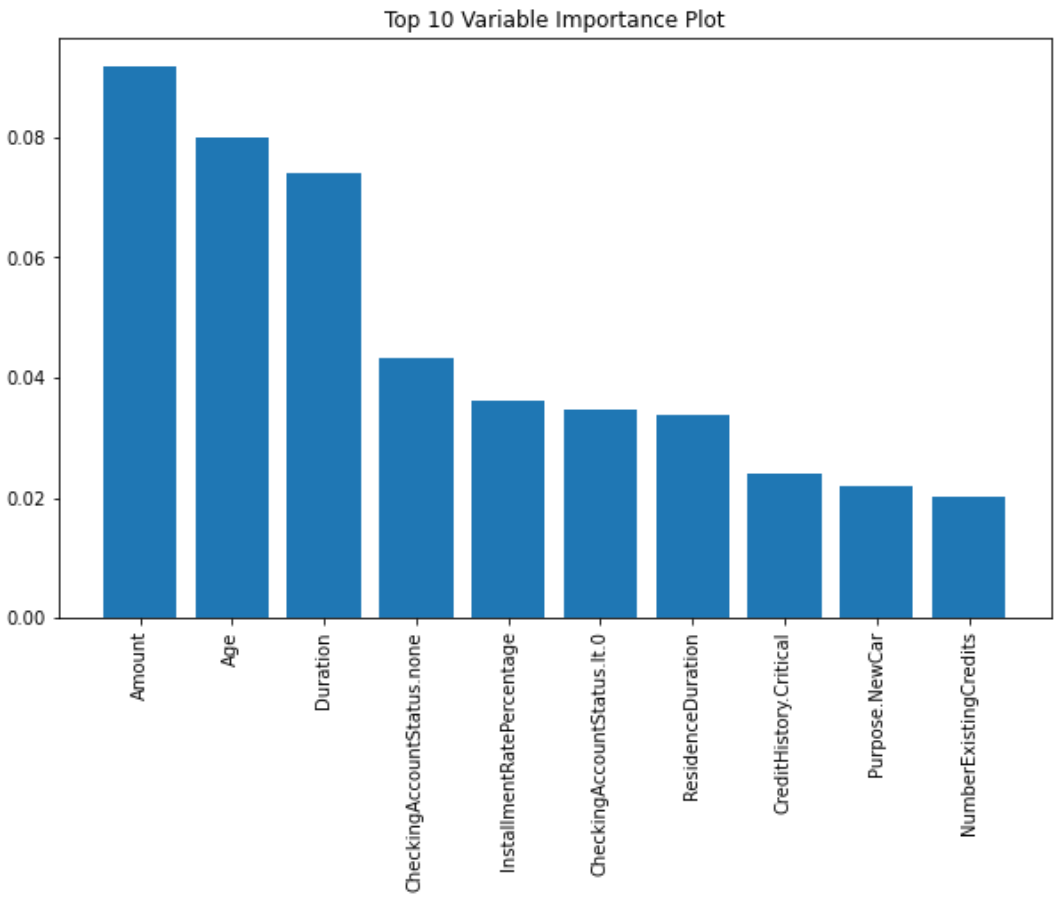
```
Best parameters: {'max_features': 5}
Test error rate of random forest: 0.24
```

Finally, a plot is created showing variable importance for the model with the tuned parameter.

- In order to get the feature importances from the trained random forest model, **rf_tuned** is used.

- The feature importances are sorted in descending order and selected only the top 10 most important features.

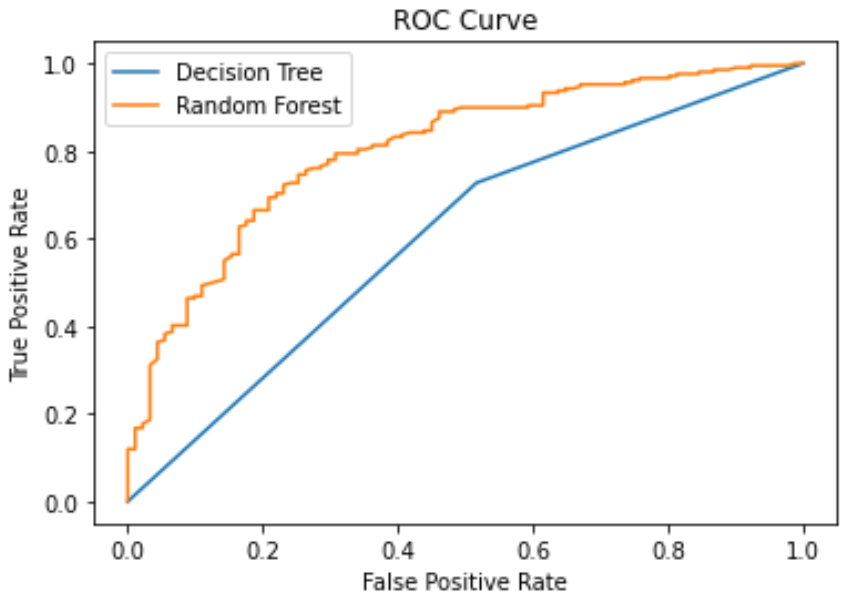
- The y-axis of the bar plot represents the relative importance of each feature, and the x-axis shows the names of the corresponding features.



Based on the plot, it can be seen that the most important feature is the "Amount" followed by "Age" and "Duration". The remaining features also have non-zero importance, but their relative importance is much smaller than the top three features. This plot can be helpful in feature selection, as it allows us to prioritize which features are most important and should be included in further analysis or modelling.

(C) Now, a plot is produced showing two ROC curves for the test predictions of the trained Decision Tree Classifier and Random Forest Classifier. The following steps were performed,

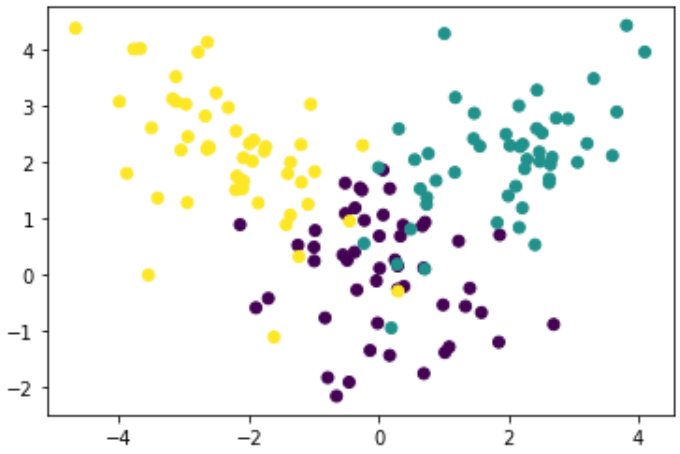
- First, a **LabelEncoder** object was used to encode the binary labels of the test set.
- Next, I have encoded the binary labels of the test set using the **fit_transform** method of the **LabelEncoder** object created in the previous step.
- Now the code predicts the probabilities of the positive class (class 1) using the **predict_proba** method of the decision tree model **dtc** on the test set **X_test**. The second column of the predicted probabilities is extracted using the indexing **[:, 1]**.
- Then we compute the false positive rate (**fpr**), true positive rate (**tpr**), and threshold values for the decision tree model using the **roc_curve** function from **sklearn. metrics**. The function takes in the binary labels of the test set **Y_test_bin** and the predicted probabilities of the positive class **dtc_probs**.
- A plot is created showing ROC curves for the decision tree and the random forest using **matplotlib.pyplot**. The x-axis represents the false positive rate, and the y-axis represents the true positive rate. The **xlabel**, **ylabel**, and **title** functions set the labels and title of the plot, and the **legend** function creates a legend for the two curves.



The ROC curve is a graphical representation of the performance of a binary classifier as the discrimination threshold is varied. Based on the plot, it can be seen that the random forest model outperforms the decision tree model, as its ROC curve is further away from the diagonal line and has a higher AUC score. This suggests that the random forest model has better predictive power than the decision tree model.

Q2 A three-class dataset is established with 50 observations in each class and two features using Gaussian distributions.

(A) In order to demonstrate the dataset is not linearly separable, we take the help of a scatterplot to gain insights.



The scatter plot shows three clusters of data points in different colors, representing the three classes in the dataset. The plot confirms that the dataset is not linearly separable, meaning that it is not possible to draw a straight line that separates the data points from different classes. Instead, a non-linear decision boundary is likely necessary to accurately classify the data. Also, it can be seen from the scatter plot the clusters overlap to some extent, which makes it more challenging to separate the classes.

(B) The dataset is then split into a training set (50%) and a test set (50%). Then,

- Three SVM models are trained on the training set using different kernels: linear, polynomial, and RBF.
- Hyperparameters for each model are tuned using 5-fold cross-validation and grid search.
- The accuracy of each model is calculated on both the training and test sets using the **accuracy score** function from **sklearn**.

Outcomes:

```
Linear SVM train accuracy: 0.893
Linear SVM test accuracy: 0.907
Polynomial SVM train accuracy: 0.867
Polynomial SVM test accuracy: 0.867
RBF SVM train accuracy: 0.920
RBF SVM test accuracy: 0.880
```

The test accuracies of the three SVM models are not close to 100%, indicating that the dataset is not linearly separable. If the dataset were linearly separable, the linear SVM model would have achieved 100% accuracy on both the training and test sets. However, in this case, the linear SVM achieved an accuracy of 89.3% on the training set and 90.7% on the test set, indicating that it is unable to perfectly separate the classes. Similarly, the polynomial SVM achieved an accuracy of 86.7% on both the training and test sets and the RBF SVM achieved an accuracy of 92.0% on the training set and 88.0% on the test set, further confirming that the dataset is not linearly separable.

Q3

(A) In this part, the **newthyroid.txt** data is classified using KNN and LDA. The data is randomly split into a training set (70%) and a test set (30%) 10 times. Further, the code performs the following steps,

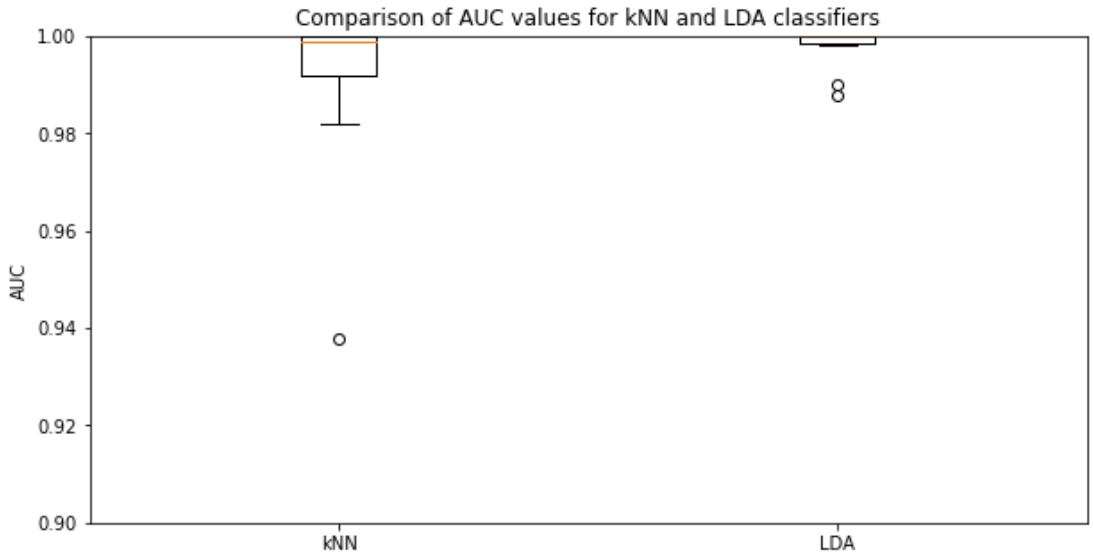
- First, it separates the features and target variables from the input data.
- Performs a loop over a range of 10 random splits of the data into training and test sets.
- For each split, loop over a range of k values (3 to 15 in steps of 2) for the k-nearest neighbors (kNN) algorithm.
- Computes the cross-validation scores and AUC scores for each k value in the range.
- Chooses the k with the largest AUC score.
- Trains a kNN model with the best k value and computes the AUC score on the test set.
- Trains a Linear Discriminant Analysis (LDA) model and computes the AUC score on the test set.
- Stores the AUC scores for each kNN and LDA model in separate lists.
- Prints the best k value and tests AUC scores for each fold.

Outcomes:

The results show that the kNN and LDA models perform very well on the dataset, with most test AUC scores being close to or equal to 1.0. The best k values for the kNN model range from 3 to 7 across the 10 random splits. It is worth noting that the AUC score for the kNN model in fold 4 is lower than the others, at 0.938, which may suggest that the model is overfitting to the training data in that fold. Overall, the results indicate that both the kNN and LDA models are effective for this particular dataset.

```
Best k for fold 1 is 3
Test AUC for kNN in fold 1 is 1.000
Test AUC for LDA in fold 1 is 1.000
Best k for fold 2 is 3
Test AUC for kNN in fold 2 is 1.000
Test AUC for LDA in fold 2 is 1.000
Best k for fold 3 is 3
Test AUC for kNN in fold 3 is 0.991
Test AUC for LDA in fold 3 is 1.000
Best k for fold 4 is 7
Test AUC for kNN in fold 4 is 0.938
Test AUC for LDA in fold 4 is 0.998
Best k for fold 5 is 5
Test AUC for kNN in fold 5 is 0.998
Test AUC for LDA in fold 5 is 1.000
Best k for fold 6 is 3
Test AUC for kNN in fold 6 is 0.994
Test AUC for LDA in fold 6 is 0.988
Best k for fold 7 is 3
Test AUC for kNN in fold 7 is 1.000
Test AUC for LDA in fold 7 is 1.000
Best k for fold 8 is 5
Test AUC for kNN in fold 8 is 1.000
Test AUC for LDA in fold 8 is 0.990
Best k for fold 9 is 3
Test AUC for kNN in fold 9 is 1.000
Test AUC for LDA in fold 9 is 1.000
Best k for fold 10 is 3
Test AUC for kNN in fold 10 is 0.982
Test AUC for LDA in fold 10 is 1.000
```

(B) For further analysis and interpretation of 10 AUC values of kNN and LDA, a plot is constructed displaying two boxplots with independent values of kNN and LDA.



Outcomes:

The boxplot shows the comparison of AUC values for kNN and LDA classifiers across 10 random splits of the dataset. The plot indicates that both classifiers perform well with AUC values generally above 0.95 and with no significant outliers. However, kNN appears to have slightly more variability in performance than LDA, with a wider range of AUC values and a larger spread of data points. Overall, the plot suggests that both classifiers are effective for this particular dataset, but LDA may be more consistent in its performance.

Q4 This code is performing a classification task on the GermanCredit dataset using Fisher's Discriminant Analysis (FDA) and Principal Component Analysis (PCA).

The **myFDA** function defined at the beginning of the code performs Fisher's Discriminant Analysis on the input data X and output labels y. It first splits the data into two subsets based on the output labels: "good" and "bad". Then it computes the mean vectors for each subset and the pooled within-class covariance matrix Sw. Finally, it computes the Fisher discriminant vector w by solving the equation **w = Sw^-1 (mean(good) - mean(bad))**.

```
> myFDA <- function(X, y) {
+     good <- X[y == 1,]
+     bad <- X[y == 0,]
+     ngood <- nrow(good)
+     nbad <- nrow(bad)
+     gmean <- colMeans(good)
+     bmean <- colMeans(bad)
+     Sw <- ((ngood-1)*cov(good) + (nbad-1)*cov(bad))/(ngood+nbad-2)
+     w <- solve(Sw, gmean - bmean)
+     return(w)
+ }
```

The next part of the code reads in the GermanCredit dataset, removes two variables that have the same value for both classes and converts the class variable to numeric (0 or 1). It then splits the data into training and testing sets using the **createDataPartition** function from the **'caret'** package.

The training set is then used to perform Principal Component Analysis using the **prcomp** function. The resulting principal components are used as input features for the myFDA function, along with the corresponding output labels. The Fisher discriminant vector w obtained from myFDA is then used to classify the testing set.

Gaining insights on ‘w’ :

```
          PC1           PC2           PC3
-0.0001580893 -0.0255196205  0.0441598767
          PC4           PC5           PC6
 0.0757385820 -0.1253502634 -0.7470631039
          PC7           PC8           PC9
-1.2663279283 -0.0065432533  0.0056681570
         PC10          PC11          PC12
 0.2441342935 -0.6019661820  0.0297915250
         PC13          PC14          PC15
-0.4908769426  0.6617243869  0.3540096708
         PC16          PC17          PC18
-0.5139842477 -0.5434165289 -0.2758718891
         PC19          PC20          PC21
-0.3723833537 -0.4554248865  0.1605535178
         PC22          PC23          PC24
 0.1770919894  0.3654197549 -0.1991052523
         PC25          PC26          PC27
-0.1621575409  0.0801959404  0.2091928180
         PC28          PC29          PC30
 0.1771331711  0.5225647160  0.1341253404
         PC31          PC32          PC33
-0.6772710599  0.8428783301  0.0289131169
         PC34          PC35          PC36
-0.1470748641 -0.2792534542 -0.5057794683
         PC37
-1.0075817523
```

The output of the function **myFDA** is a vector **w** that contains the linear discriminant coefficients that can be used to project the data onto a lower-dimensional space that maximizes the separation between the two classes.

In this case, **w** is a vector with 37 elements corresponding to the 37 principal components obtained from the PCA. Each element represents the contribution of the corresponding principal component to the linear combination that maximizes the separation between the two classes.

The first few elements of **w** have small values close to zero, which suggests that these components do not contribute much to the discrimination between the classes. The largest absolute values are found in the 6th and 7th elements, which correspond to the 5th and 6th principal components obtained from the PCA.

These components may have a stronger influence on the separation between the two classes. The negative sign of the coefficients in the first two principal components suggests that they contribute negatively to the separation between the classes. Conversely, the positive sign of the coefficients in the 6th and 7th principal components suggests that they contribute positively to the separation between the classes.

Overall, **w** can be used to project new data onto a lower-dimensional space that maximizes the separation between the two classes. The projections can then be used to classify the new data as either belonging to the "Good" or "Bad" credit risk class.

References

1. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

2. https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html

3. Chapter 4 of "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman (2009): https://web.stanford.edu/~hastie/Papers/ESLII.pdf

4. Chapter 14 of "The Elements of Statistical Learning" by Hastie, Tibshirani, and Friedman (2009): https://web.stanford.edu/~hastie/Papers/ESLII.pdf

5. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html