D7054E Project

Sofia padovani plazzi (SOFPAD-3)
Devashish singha roy (DEVSIN-3)

Group 12

March 17, 2024

Abstract

This report describes Group 12's approach to classify the Apple Quality dataset, categorizing apples for their quality (good or bad) based on their features, such as Size, Weight, etc. Especially we used different statistical and Machine Learning techniques to deal with such task, from data pre-processing and cleaning to model implementation. The project focuses on finding the best model to deal with apple quality classification, providing the most accurate estimate.

Introduction

Our project is related to the agricultural industry, as one of the main issues of such a field is ensuring the quality of products, which is critical for producers and consumers. For what concerns apple farmers and distributors, categorizing apples based on their quality can significantly impact sales, consumer satisfaction, and overall profitability. Nevertheless, doing this process manually can be time-consuming, subjective, and sometimes inconsistent. Therefore, there is a need to automate this process to effectively categorize apple quality based on various attributes such as size, color, and defects.

This report introduces and describes our final project. In particular, this project aims to develop a machine-learning model to categorize apple quality based on different features accurately. The hypothesis is that it is possible to build a consistent system that automatically and efficiently classifies apple quality by using a dataset with information on apples' attributes and different machine-learning techniques. Efficient classification implies a classification accuracy higher than 85%-90%. Several algorithms and techniques, such as decision trees and SVM, are implemented and compared and the optimal model is chosen based on its characteristics (e.g., interpretability and efficiency) and performance (e.g., accuracy). Accuracy is also computed using cross validation, which offers a more accurate estimate of the efficacy of the model. Best parameters are searched using the Search Grid algorithm. Our hypothesis is that, with our resources and using Apple Quality dataset, it is possible to find a Machine Learning model with high accuracy test, equal or greater than 85%.
Most of our solution is based on the knowledge gained by studying the course book [1].

Next paragraph will focus on the methodology used to solve the project research problem, from the techniques to ethical considerations. Chapter 2 will show the results of the techniques implementations. The discussion chapter will analyse our work and, finally, the Conclusion chapter will briefly sum up the Project.

1 Methodology

This section provides a structured methodology for conducting each step the Project. Ethical Considerations are then discussed.

1.1 Datasets

The dataset used for this project is an Apple Quality csv file which can be found at [2], in Kaggle platform. It contains information on apple's quality attributes. In particular, the dataset fields are:

- **A_id**: Unique identifier for each fruit;
- **Size**:The size of the fruit;
- **Weight**: Weight of the fruit;
- **Sweetness**:The degree of sweetness of the fruit;
- **Crunchiness**: Texture indicating the crunchiness of the fruit;

- **Juiciness**: Level of the juiciness of the fruit;
- **Ripeness**: Stage of ripeness of the fruit;
- **Acidity**: The acidity level of the fruit;
- **Quality**: Overall quality of the fruit (label: good or bad);

The dataset is already scaled. Nonetheless, some missing values were found and removed. Hence, for better performances of certain models, we scaled the dataset again. All the fields in this data set are of category type 'Continuous Numerical' except for column 'Quality' which is of type 'nominal categorical'. 'Continuous Numerical' means the data represents continuous numeric data while 'nominal categorical' means its categorical data but without any intrinsic order.

1.2 Data Ethics considerations

This project work is based on the data about apples which is a non-human subject, however there are still some data ethical considerations that are important. Here we describe some of them.

Consent: While consent may not be applicable in the traditional sense for non-human subjects like apples, ethical considerations may arise if the data involves information collected from individuals associated with apple cultivation or distribution. It's important to ensure that any data collected from individuals, such as farmers or workers, is done so with their knowledge and consent.

Data Privacy: Even though apples themselves do not have privacy concerns, data privacy may still apply if the dataset includes any information about individuals associated with apple production or distribution. In such cases, personally identifiable information should be handled carefully to protect the privacy of these individuals.

Environmental Impact: Considerations should be made regarding the environmental impact of data collection methods used in gathering information about apples. For example, if data collection involves resource-intensive processes or has adverse environmental effects, efforts should be made to minimize these impacts.

Fair Treatment: Ensure fair treatment and representation of different types of apples and apple producers within the dataset. Biases or preferences towards certain varieties or producers could lead to unfair outcomes or misrepresentation.

Quality Control: Ethical considerations should also extend to the quality and accuracy of the data. Ensuring that data collection methods are rigorous and reliable thus help maintain integrity and trust in the dataset.

Transparency: Transparency in data collection methods, data sources, and any preprocessing steps is important for accountability and reproducibility. Providing clear documentation about how the data was collected and curated enhances trust in the dataset.

Sustainable Practices: Considerations should be made to ensure that data collection practices related to apple cultivation and production are sustainable and environmentally responsible. This includes minimizing waste, reducing resource consumption, and promoting sustainable farming practices.

1.3 Techniques

The techniques employed in this project vary depending on the task we were dealing with. We used object oriented programming (OOPs) to build classes and design methods, as shown in Unified Modeling Language (UML) class diagram 1.

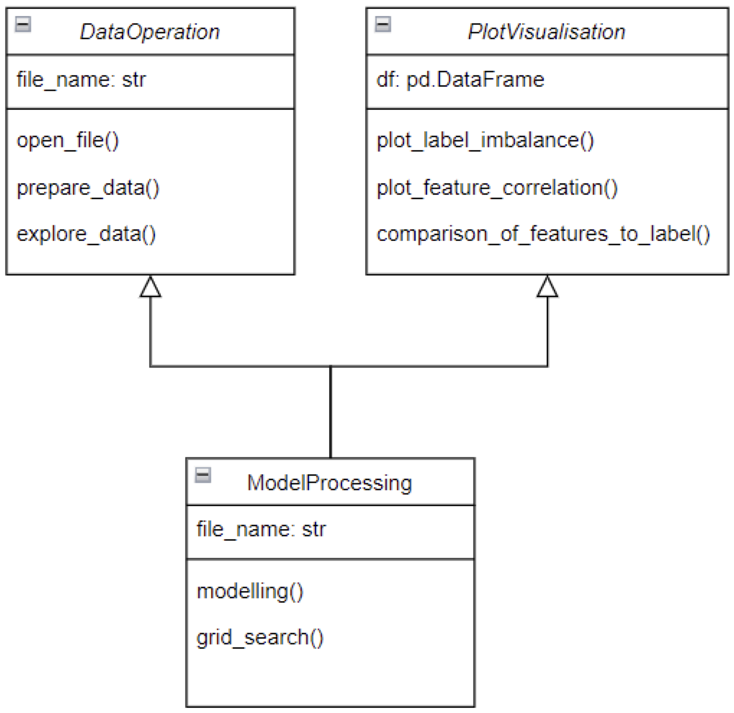
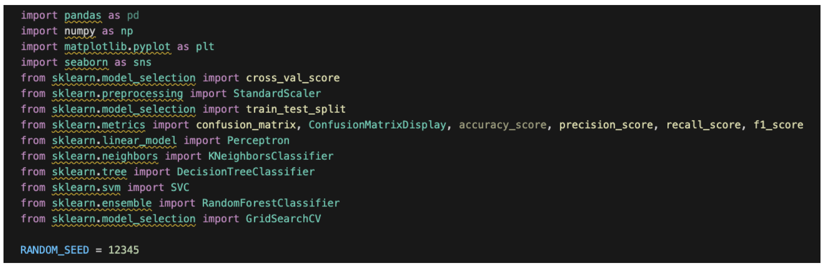


Figure 1: UML - Class Diagram

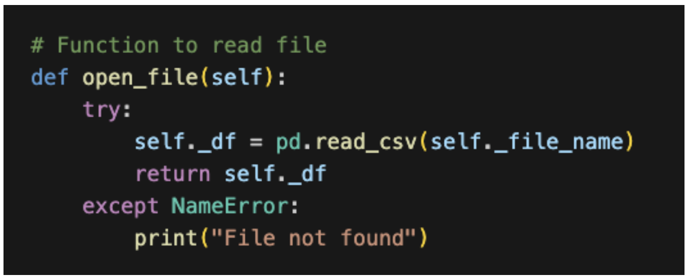
- **Libraries**: the libraries and modules imported for this project can be found in Figure 2. Below libraries are described and its use justified based on their capabilities.
  - **pandas**: Python library for data manipulation and analysis, whose main data structures are series and DataFrame.
  - **numpy**: Python package for mathematical computations, whose main data structures are arrays and matrices.
  - **matplotlib**: Python library for data visualization, providing function for plotting data.
  - **seaborn**: Python library (based on matplotlib) for data visualization, providing function for enhancing matplotlib capabilities. It offers an higher-level of abstraction.
  - **cross_val_score from sklearn.model_selection**: A function extracted from sklearn to estimate models' performance by applying cross validation technique.
  - **StandardScaler from sklearn.preprocessing**: A function extracted from sklearn to pre-process data by removing the mean and scaling to unit variance (standardization).
  - **train_test_split from sklearn.model_selection**: A function extracted from sklearn to divide the given dataset into training, validation and test sets.
  - **sklearn.metrics**: A module of sklearn with main functions to calculate performance metrics like precision or recall, and to record actual and predicted labels into a matrix (confusion matrix).
  - **Perceptron from sklearn.linear_model**: A class in sklearn in which single-neural layer classifier (perceptron) is implemented.
  - **NeighborsClassifier from sklearn.neighbors** : A class in sklearn in which K-nearest neighbors classifier is implemented.
  - **DecisionTreeClassifier from sklearn.tree**: A class ins klearn in which decision tree classifier is implemented.
  - **SVC from sklearn.svm**: A class in sklearn in which SVM classifier is implemented.
  - **RandomForestClassifier from sklearn.ensebmle**: A class in sklearn in which decision tree classifier is implemented.
  - **GridSearchCV from sklearn.model_selection**: A function in sklearn for looking for best parameters of a given classifier among specified parameters.



```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, precision_score, recall_score, f1_score
from sklearn.linear_model import Perceptron
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

RANDOM_SEED = 12345
```

Figure 2: Code for importing modules and libraries.

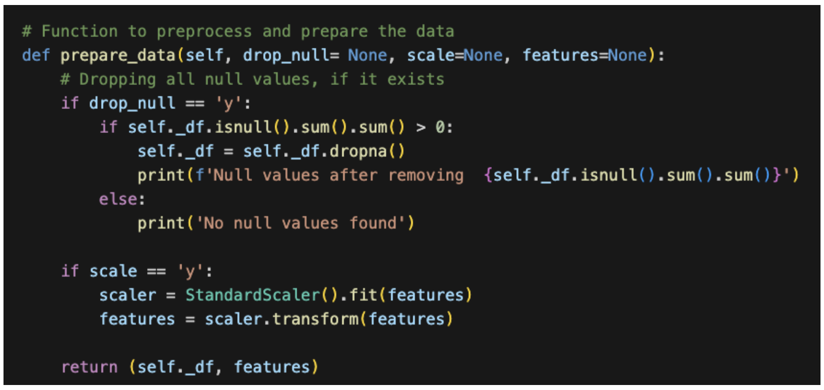
- **Data Loading**: A class called DataOperation is created to deal with data loading, pre-processing and exploration. The dataset was imported by using the pandas library, inside the "open_file" method, hence transformed in a DataFrame. An error is printed if the process fail.



```python
# Function to read file
def open_file(self):
    try:
        self._df = pd.read_csv(self._file_name)
        return self._df
    except NameError:
        print("File not found")
```

Figure 3: Code for loading the dataset using pandas library.

- **Data Pre-processing**: another method of the DataOperation class was created to pre-process data before being used.



```python
# Function to preprocess and prepare the data
def prepare_data(self, drop_null= None, scale=None, features=None):
    # Dropping all null values, if it exists
    if drop_null == 'y':
        if self._df.isnull().sum().sum() > 0:
            self._df = self._df.dropna()
            print(f'Null values after removing  {self._df.isnull().sum().sum()}')
        else:
            print('No null values found')

    if scale == 'y':
        scaler = StandardScaler().fit(features)
        features = scaler.transform(features)

    return (self._df, features)
```

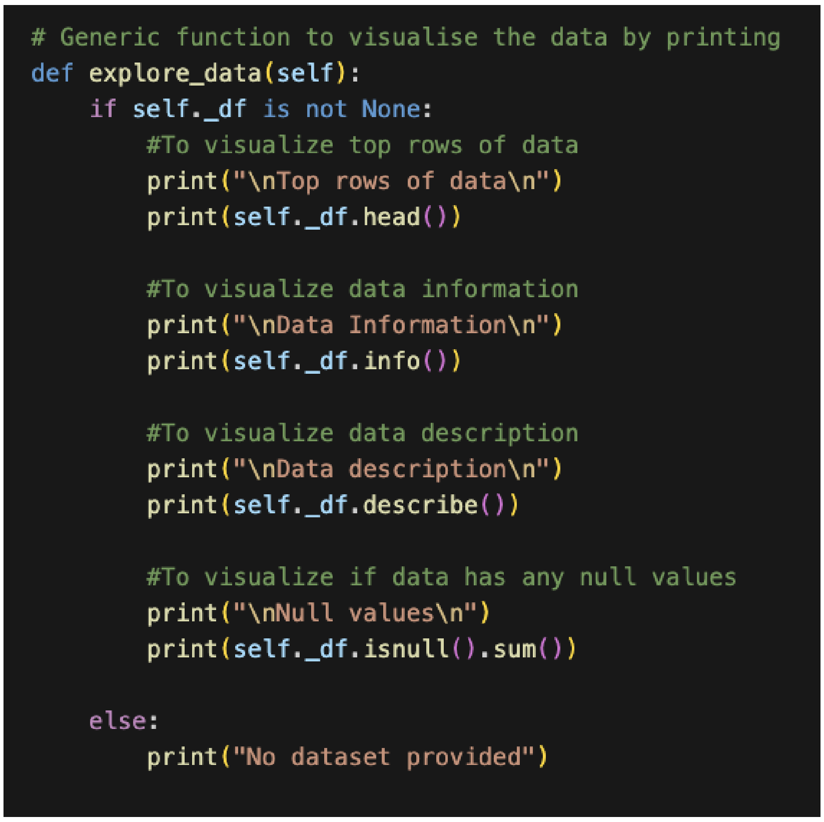
Figure 4: Code for dividing data into features and labels.

Code in Figure 4 was used to:

1. Check the presence of missing values by column.
2. Drop all null values from the dataset, and saving the new dataset in an homonym variable.
3. Check that the total number of missing values in the dataset equals to zero.

As some rows were dropped due to the presence of missing values, we standardized the features dataset.

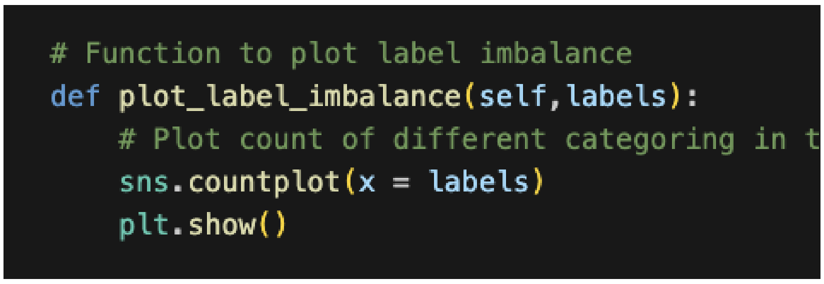
- **Data Exploration**: the "explore_data" method of the DataOperation (Figure 5 was created to:
  1. Print the first rows of the dataset.
  2. Print information about the dataset,
  3. Print the description of the dataset (so simple statistics of each features).
  4. Print the total number of missing values in the dataset.



```python
# Generic function to visualise the data by printing
def explore_data(self):
    if self._df is not None:
        #To visualize top rows of data
        print("\nTop rows of data\n")
        print(self._df.head())

        #To visualize data information
        print("\nData Information\n")
        print(self._df.info())

        #To visualize data description
        print("\nData description\n")
        print(self._df.describe())

        #To visualize if data has any null values
        print("\nNull values\n")
        print(self._df.isnull().sum())

    else:
        print("No dataset provided")
```

Figure 5: Code for exploring data.

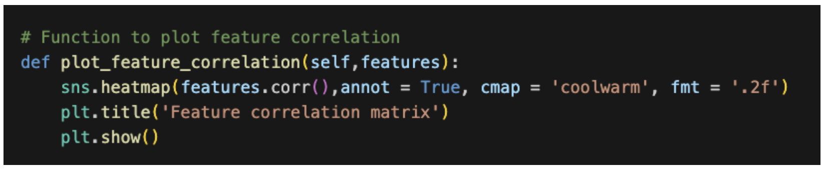
- **Data Visualization**: Visualization helps us understanding the data and which approaches would be better for our task. It is always better having an idea and making hypothesis on your data before starting implementing your models.
  In this project we create a class PlotVisualisation were different methods were created to build different plots to visualize the data.
  Figure 6 is a function created to show how many times a label appear in the dataset, so how many rows we have about a specific label. This is very useful to check for data imbalance, which can hinder the training process and prediction accuracy of a classification model.



```python
# Function to plot label imbalance
def plot_label_imbalance(self,labels):
    # Plot count of different categoring in t
    sns.countplot(x = labels)
    plt.show()
```

Figure 6: Bar plot showing the number of rows per classes.

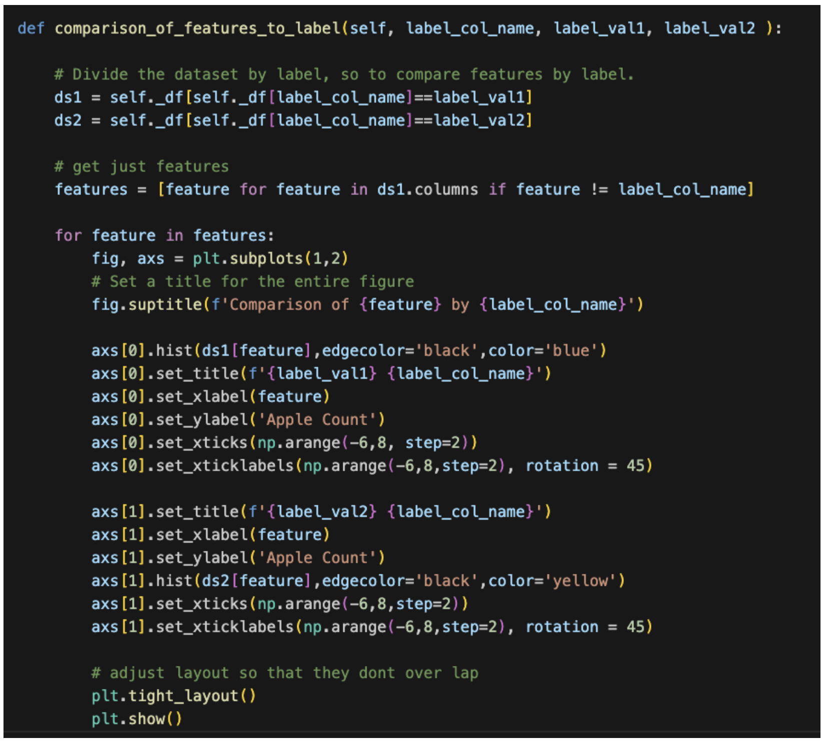
Figure 7 is a method created to show the pairwise correlation between features. Correlation between independent variables can lead to to less interpretable models, instability, overfitting, and reduced generalization performance.



```python
# Function to plot feature correlation
def plot_feature_correlation(self,features):
    sns.heatmap(features.corr(),annot = True, cmap = 'coolwarm', fmt = '.2f')
    plt.title('Feature correlation matrix')
    plt.show()
```

Figure 7: Correlation Matrix.

Finally, the "comparison_of_features_to_label" function of the class PlotVisualisation was created to compare features distribution by label, we divided the dataset by label (hence in two datasets, one with labels 'good' and the other with label 'bad') and we used histogram plots to see how features were distributed (Figure 8). Nonetheless, as the dataset was already scaled, it is difficult to fully extract insights from the plots.
The colours of the two plots (bluie and yellow) were chosen by taking into consideration colorblindness syndrome.

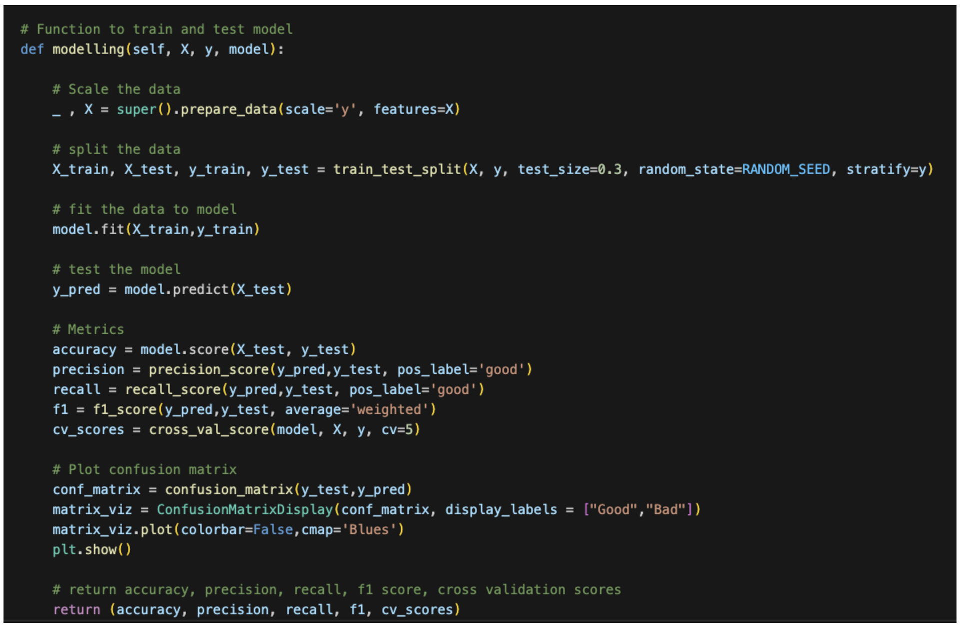


```python
def comparison_of_features_to_label(self, label_col_name, label_val1, label_val2 ):

    # Divide the dataset by label, so to compare features by label.
    ds1 = self._df[self._df[label_col_name]==label_val1]
    ds2 = self._df[self._df[label_col_name]==label_val2]

    # get just features
    features = [feature for feature in ds1.columns if feature != label_col_name]

    for feature in features:
        fig, axs = plt.subplots(1,2)
        # Set a title for the entire figure
        fig.suptitle(f'Comparison of {feature} by {label_col_name}')

        axs[0].hist(ds1[feature],edgecolor='black',color='blue')
        axs[0].set_title(f'{label_val1} {label_col_name}')
        axs[0].set_xlabel(feature)
        axs[0].set_ylabel('Apple Count')
        axs[0].set_xticks(np.arange(-6,8, step=2))
        axs[0].set_xticklabels(np.arange(-6,8,step=2), rotation = 45)

        axs[1].set_title(f'{label_val2} {label_col_name}')
        axs[1].set_xlabel(feature)
        axs[1].set_ylabel('Apple Count')
        axs[1].hist(ds2[feature],edgecolor='black',color='yellow')
        axs[1].set_xticks(np.arange(-6,8,step=2))
        axs[1].set_xticklabels(np.arange(-6,8,step=2), rotation = 45)

        # adjust layout so that they dont over lap
        plt.tight_layout()
        plt.show()
```

Figure 8: Histogram visualization function.

- **Models Implementation and Cross Validation**: The models implemented are:
  - **Perceptron**: single-layer neural network used for binary classification. It divides classes based on a linear decision boundary, and because of that, it does not handle well non-linear relationships.
  - **KNN**: classifier that makes predictions depending on the K majority points nearest the new point to label.
  - **Decision Tree**: classifier that recursively divide data into subsections based on the features' value, creating a boolean-logic map.
  - **SVM**: classifier that identifies a hyperplane in high-dimensional space to separate data into distinct classes. It maximizes the margin between nearest data points of each classes to the hyperplane (decision boundary).
  - **Random Forest**: ensemble learning classifier that combines multiple decision trees' prediction to improve its performance.

To implement all these models, a class called ModelProcessing was implemented, inheriting from parent class DataOperation, PlotVisualisation. In Figure 9, it can be seen how data are scaled by using the inherited method "prepare_data". Then data are divided into training and test sets. The classifier object, passed as a parameter of the method, is then fitted to training features and labels and predictions are instead made on unseen data, so on the test set. Accuracy is then calculated and also other metrics are calculated in order to enable a better interpretation of the model's performance.

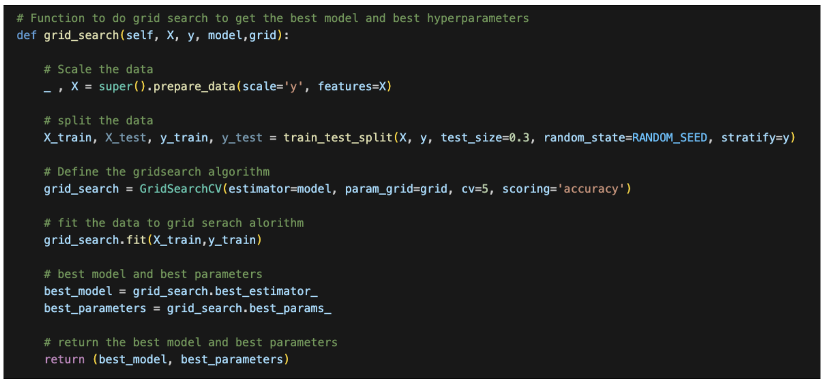


```python
# Function to train and test model
def modelling(self, X, y, model):

    # Scale the data
    _ , X = super().prepare_data(scale='y', features=X)

    # split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=RANDOM_SEED, stratify=y)

    # fit the data to model
    model.fit(X_train,y_train)

    # test the model
    y_pred = model.predict(X_test)

    # Metrics
    accuracy = model.score(X_test, y_test)
    precision = precision_score(y_pred,y_test, pos_label='good')
    recall = recall_score(y_pred,y_test, pos_label='good')
    f1 = f1_score(y_pred,y_test, average='weighted')
    cv_scores = cross_val_score(model, X, y, cv=5)

    # Plot confusion matrix
    conf_matrix = confusion_matrix(y_test,y_pred)
    matrix_viz = ConfusionMatrixDisplay(conf_matrix, display_labels = ["Good","Bad"])
    matrix_viz.plot(colorbar=False,cmap='Blues')
    plt.show()

    # return accuracy, precision, recall, f1 score, cross validation scores
    return (accuracy, precision, recall, f1, cv_scores)
```

Figure 9: Code for model implementation.

Finally, the confusion matrix is plotted to give a better idea of the number of right and wrong class predicted, compared to the actual labels.
Finally, to compute an even more accurate accuracy score, we used cross validation. In general, cross validation provides a more robust estimate of model performance and helps in assessing how well the model generalizes to unseen data. This technique can be used for model selection or tuning hyperparameters. It is, in fact, usually used to train and evaluate models on different subsets of the dataset. In this project, however, cross-validation is applied to the model after being fitted to training data. This keeps the model's structure unchanged while a further evaluation is provided.

- **Hyperparameters tuning**: when it comes to parameters optimization, there are different paths that a data scientist can choose. In this project we opted for Grid Search, which is an algorithm that find the optimal hyperparameters of a model from a dictionary of specified parameters. The code for implementing such a technique can be seen in Figure 10. As such a algorithm is usually time demanding when training, we implemented such technique only with the three models with higher accuracy score. The function is a method of the ModelProcessing class.



```python
# Function to do grid search to get the best model and best hyperparameters
def grid_search(self, X, y, model,grid):

    # Scale the data
    _ , X = super().prepare_data(scale='y', features=X)

    # split the data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=RANDOM_SEED, stratify=y)

    # Define the gridsearch algorithm
    grid_search = GridSearchCV(estimator=model, param_grid=grid, cv=5, scoring='accuracy')

    # fit the data to grid serach alorithm
    grid_search.fit(X_train,y_train)

    # best model and best parameters
    best_model = grid_search.best_estimator_
    best_parameters = grid_search.best_params_

    # return the best model and best parameters
    return (best_model, best_parameters)
```

Figure 10: Code for Grid Search implementation.

2 Results

2.1 Data Visualization

:

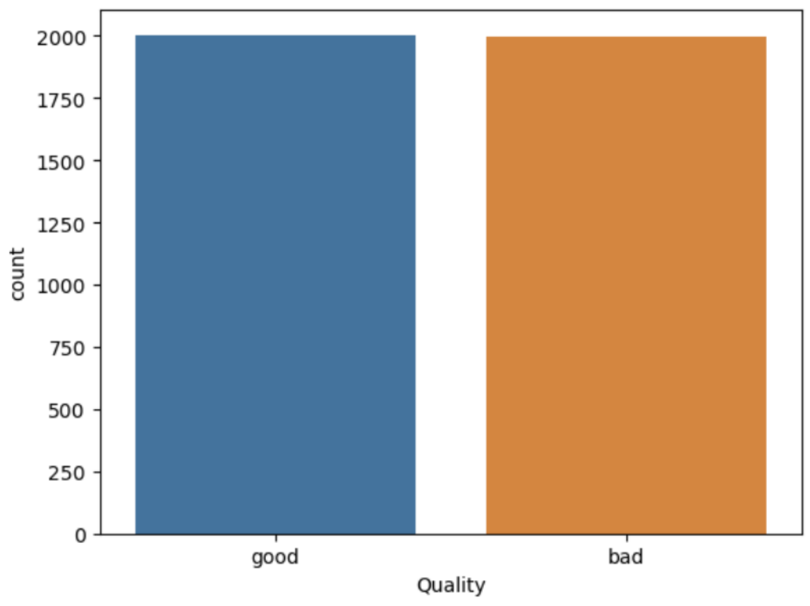
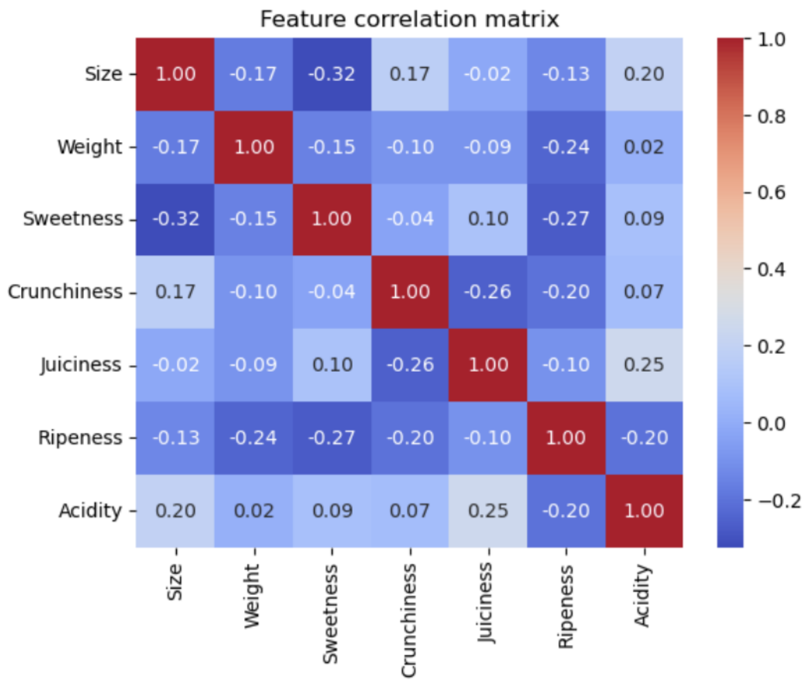


Figure 11: Plot checking for data imbalance.



	Size	Weight	Sweetness	Crunchiness	Juiciness	Ripeness	Acidity
Size	1.00	-0.17	-0.32	0.17	-0.02	-0.13	0.20
Weight	-0.17	1.00	-0.15	-0.10	-0.09	-0.24	0.02
Sweetness	-0.32	-0.15	1.00	-0.04	0.10	-0.27	0.09
Crunchiness	0.17	-0.10	-0.04	1.00	-0.26	-0.20	0.07
Juiciness	-0.02	-0.09	0.10	-0.26	1.00	-0.10	0.25
Ripeness	-0.13	-0.24	-0.27	-0.20	-0.10	1.00	-0.20
Acidity	0.20	0.02	0.09	0.07	0.25	-0.20	1.00

Figure 12: Correlation Matrix.

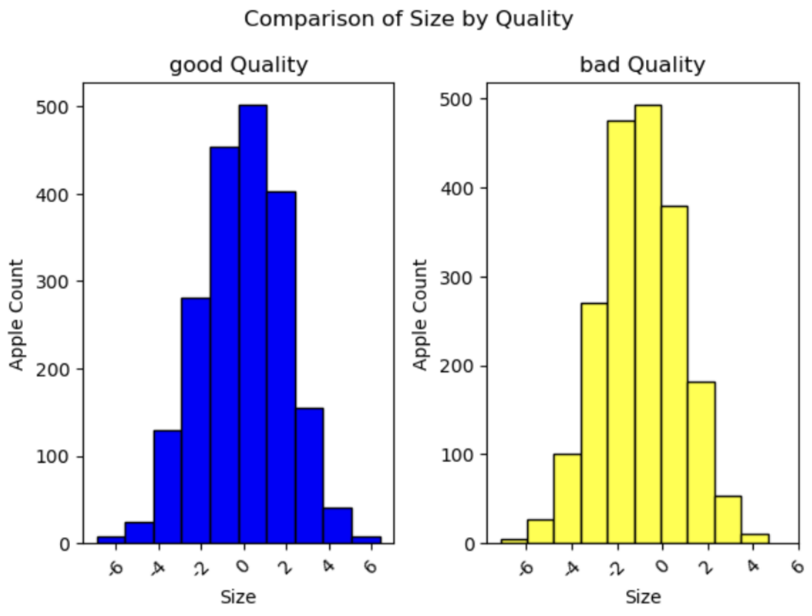


Figure 13: Size distribution compared by Quality class.

2.2 Perceptron

- **Accuracy**: 70.33%
- **Cross-Validation Accuracy**: 66.07%
- **Recall**: 70.59%
- **Precision**: 57.90%
- **F1 score**: 67.10%

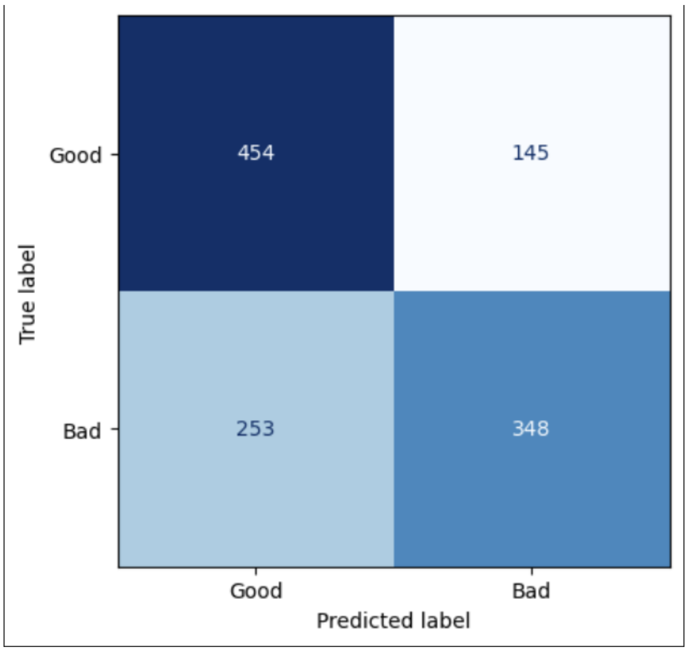


Figure 14: Confusion Matrix of Perceptron model.

2.3 KNN

- **Accuracy**: 86.83%
- **Cross-Validation Accuracy**: 89.15%
- **Recall**: 86.61%

- **Precision**: 87.19%
- **F1 score**: 86.83%

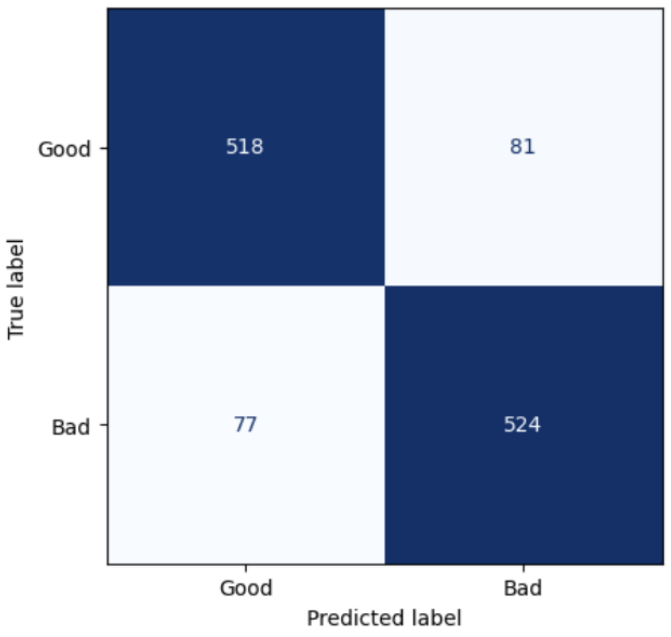


Figure 15: Confusion Matrix of KNN model.

2.4 Decision Tree

- **Accuracy**: 79.58%
- **Cross-Validation Accuracy**: 80.95%
- **Recall**: 80.07%
- **Precision**: 78.87%
- **F1 score**: 79.58%

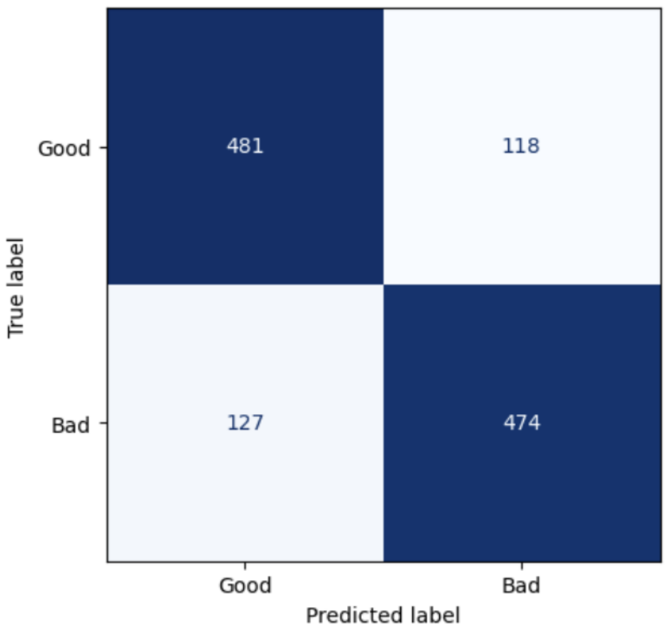


Figure 16: Confusion Matrix of Decision Tree model.

2.5 SVM

- **Accuracy**: 87.17%
- **Cross-Validation Accuracy**: 89.15%
- **Recall**: 86.70%
- **Precision**: 87.85%
- **F1 score**: 87.17%

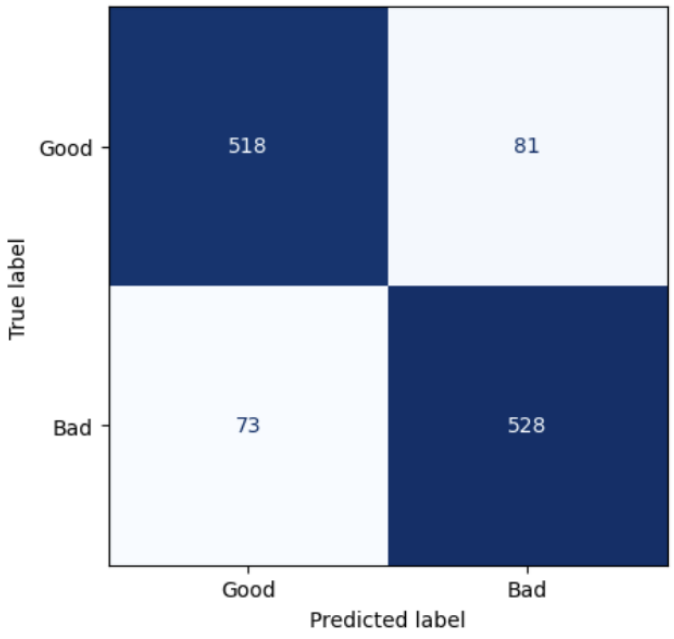


Figure 17: Confusion Matrix of SVM model.

2.6 Random Forest

- **Accuracy**: 86.58%
- **Cross-Validation Accuracy**: 88.33%
- **Recall**: 85.95%
- **Precision**: 87.52%
- **F1 score**: 86.58%

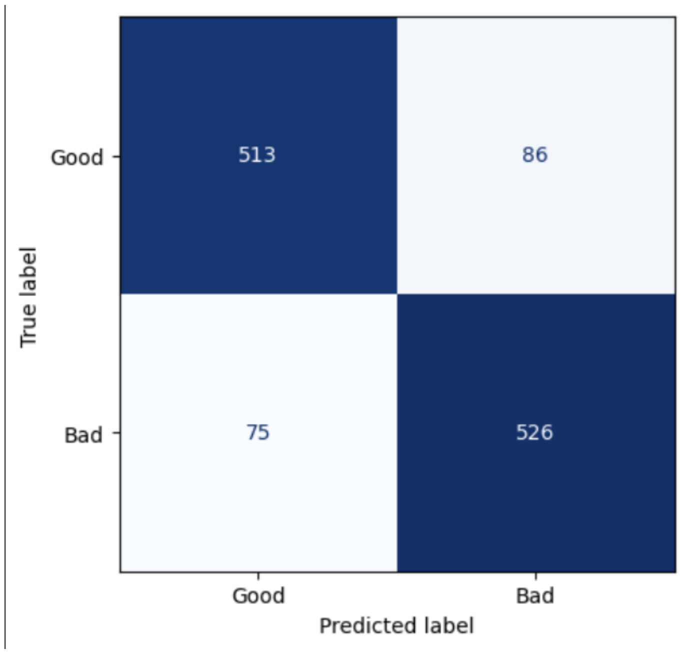


Figure 18: Confusion Matrix of Random Forest model.

2.7 Grid Search

2.7.1 Random Forest

- **Accuracy**: 86.42%
- **Cross-Validation Accuracy**: 88.6%
- **Recall**: 86.02%
- **Precision**: 87.02%
- **F1 score**: 86.42%

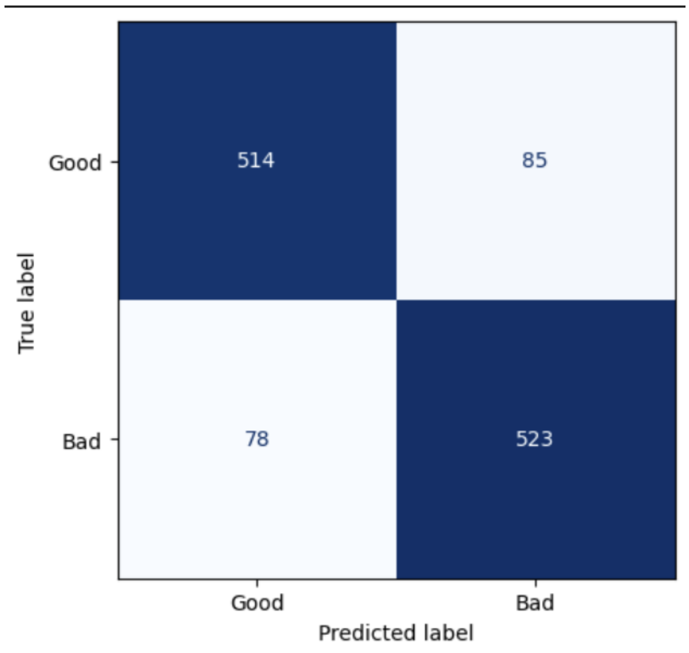


Figure 19: Confusion Matrix of Random Forest model with best hyperparameters.

2.7.2 SVM

- **Accuracy**: 89.08%

- **Cross-Validation Accuracy**: 90.68%
- **Recall**: 90.10%
- **Precision**: 87.85%
- **F1 score**: 89.08%

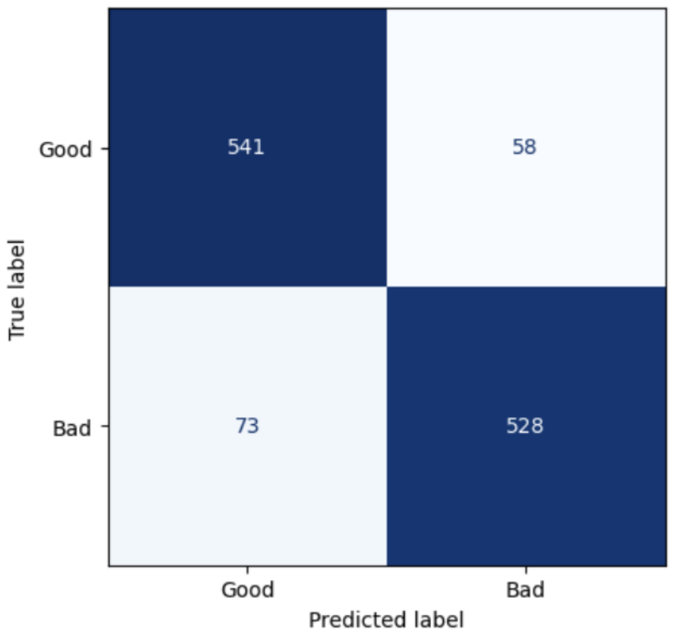


Figure 20: Confusion Matrix of SVM model with best hyperparameters.

2.7.3 KNN

- **Accuracy**: 89.08%
- **Cross-Validation Accuracy**: 89.83%
- **Recall**: 88.52%
- **Precision**: 89.85%%
- **F1 score**: 89.08%%

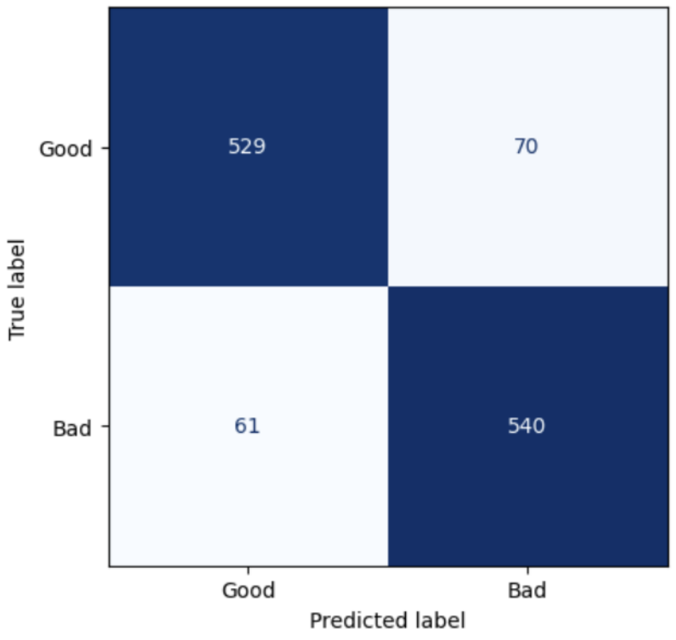


Figure 21: Confusion Matrix of KNN model with best hyperparameters.

3 Discussion

In this chapter we will focus on discussion the results reported above. We are going to explain performance metrics used to understand a model efficacy to correctly predict classes. We are going to mention positive and negative instances, where for positive we refer to the label 'good' and negative to the label 'bad'.

3.1 Data Visualization

As Figure 11 shows an equal quantity of 'good' and 'bad' labels, no corrective actions need to be taken.
Figure 12 shows the pairwise correlation between features. Correlation between independent variables can lead to to less interpretable models, instability, overfitting, and reduced generalization performance. In this case, there is no high correlation, so no corrective actions need to be taken.
Figure 13 shows an example of the feature 'Size' in both dataset with 'good' quality apples and 'bad' quality apples. It can be noticed that the average size of good quality apples is slightly greater than the ones of bad quality. Also worth mentioning that we have used high contrast colors like blue and yellow which are considered to be good for color blind individuals.

3.2 Perceptron

Accuracy: The accuracy of the Perceptron model is 70.33%. This represents the percentage of correctly classified instances out of the total number of instances.

Cross-Validation Accuracy: The cross-validation accuracy of the model is 66.07%. Cross-validation is a technique used to assess how model will generalize to an independent dataset. The low cross-validation accuracy suggests that the model may have some variability in its performance on different subsets of the data.

Recall: The recall of the model is 70.59%. Recall, also known as sensitivity, is the ratio of true positives to the sum of true positives and false negatives. It measures the model's ability to correctly identify positive instances out of all actual positive instances. A high recall indicates that the model is good at capturing positive instances.

Precision: The precision of the model is 57.90%. Precision is the ratio of true positives to the sum of true positives and false positives. It measures the model's ability to correctly identify positive instances out of all instances that the model classified as positive. A high precision indicates that when the model predicts a positive instance, it is likely to be correct.

F1 Score: The F1 score of the model is 67.10%. The F1 score is the harmonic mean of precision and recall. It provides a balance between precision and recall, making it a useful metric for evaluating model performance, especially when there is an imbalance between the classes or when both false positives and false negatives are important.

Conclusion: The above metrics show a good performance by the Perceptron model. However, better result can be achieved using different models.

3.3 KNN

Accuracy: The accuracy of the KNN model is 86.83%. This indicates that the model correctly predicts the class labels for approximately 86.83% of the instances in the dataset. High accuracy suggests that the model is generally effective at distinguishing between different classes.

Cross-Validation Accuracy: The cross-validation accuracy of the model is slightly higher at 89.15% compared to the overall accuracy 86.83%. Cross-validation accuracy provides an estimate of how well the model is likely to perform on unseen data. The fact that cross-validation accuracy is close to overall accuracy indicates that the model generalizes well to new data.

Recall: The recall of the model is 86.61%. Recall measures the model's ability to correctly identify all relevant instances from a given class. In the context of a classification task, it represents the ratio of true positives to the sum of true positives and false negatives. A high recall indicates that the model effectively captures positive instances.

Precision: The precision of the model is 87.19%. Precision measures the model's ability to correctly identify relevant instances among all instances predicted as positive. It is the ratio of true positives to the sum of true positives and false positives. A high precision suggests that the model makes few false positive predictions.

F1 Score: The F1 score of the model is 86.83%. The F1 score is the harmonic mean of precision and recall, providing a balance between the two metrics. The fact that the F1 score is close to both precision and recall indicates that the model achieves a good balance between them.

Conclusion: The above metrics show a great performance by the KNN model, both in predicting good and bad classes. As this model was one of the three models that performed better, grid search will be implemented to find the optimal parameters to use in the KNN classifier.

3.3.1 KNN after Hyperparameter Tuning

After applying GridSearchCV from module model_selection of sklearn, the best Hyperparameters for the KNN model was 'metric': 'euclidean', 'n_neighbors': 9, 'weights': 'distance'. On using these hyperparameter values we get the following result.

The accuracy improves to 89.08%. The cross-validation accuracy of 89.83% is similar to original model. The fact that cross-validation accuracy is close to overall accuracy indicates that the model generalizes well to new data, even with the selected hyperparameters. The recall of the model with the best parameters is 88.52% which is better than the original model without the hyperparameter tuning. Also the precision has improved to 89.85% in comparison to the precision 87.19% of the original model without hyperparameter tuning. The F1 score is 89.08% which is also better than the original model.

Overall we see significant improvement in performance across various evaluation metrics, showing effectiveness of using grid search in hyper parameter tuning.

3.4 Decision Tree

Accuracy: The accuracy of the Decision Tree model is 79.58% which suggests that the model's predictions are accurate for approximately 79.58% of the dataset.

Cross-Validation Accuracy: The cross-validation accuracy of the model is slightly higher at 80.95% compared to the overall accuracy which indicates that the model's performance is consistent across different subsets of the data.

Recall: The recall of the model is 80.07% which suggests that the model is effective at capturing a large portion of the positive instances.

Precision: The precision of the model is 78.87% indicates that when the model predicts an instance as positive, it is correct approximately 78.87% of the time.

F1 Score: The F1 score of the model is 79.58% can be considered a good score which suggests that the model achieves a good balance between precision and recall.

Conclusion: By looking at the metrics above, the decision tree model performed well. However, a better performance can be achieved with other models.

3.5 SVM

Accuracy: The accuracy of the SVM model is 89.08% suggests that the model's predictions are accurate for approximately 89.08% of the dataset.

Cross-Validation Accuracy: The cross-validation accuracy of the model is slightly higher at 90.68% compared to the overall accuracy indicates that the model's performance is consistent across different subsets of the data.

Recall: The recall of the model is 90.10% suggests that the model is highly effective at capturing a large portion of the positive instances, which means that out of 100 'good' classes, it correctly predicts at least 90 of them.

Precision: The precision of the model is 87.85% indicates that when the model predicts an instance as positive, it is correct approximately 87.85% of the time.

F1 Score: The F1 score of the model is 89.08%. A high F1 score suggests that the model achieves a good balance between precision and recall.

Conclusion: Support Vector Classifier demonstrated a great performance, with high scores in all metrics. Because of that, we decided to find the optimal parameters through grid search algorithm.

3.5.1 SVM after Hyperparameter Tuning

By applying GridSearch the best Hyperparameters for the SVC model was found to be 'C': 10, 'gamma': 'scale', 'kernel': 'rbf'. On using these hyperparameter values to the SVC model we get the following result.

Accuracy: The accuracy of the SVM model after hyperparameter tuning is 89.08% which is slightly better than the original model 87.17% without any hyper paremetr tuning.

Cross-Validation Accuracy: The cross-validation accuracy of the model after hyperparameter tuning is 90.68% which is also better as compared to model without the hyper parameter tuning which was 89.15%.

Recall: The recall is 90.10% whereas with the original model it was 86.70% which is a good improvement.

Precision: The precision with and without hyperparameter tuning stands to be same at 87.85%.

F1 Score: The F1 score with hyperparameter tuning is 89.08% while without the tuning is slightly less at 87.17%.

Conclusion: In summary, the SVM model with hyperparameter tuning using grid search demonstrates strong performance across various evaluation metrics. Its high accuracy, cross-validation accuracy, recall and F1 score indicates its effectiveness in accurately classifying instances. The model appears to be well-suited for the classification task at hand, with high generalization ability and robustness to unseen data, thanks to the optimized hyperparameters obtained through grid search.

3.6 Random Forest

Accuracy: The accuracy of the Random Forest model is 86.58%, which suggests that the model's predictions are accurate for approximately 86.58% of the dataset.

Cross-Validation Accuracy: The cross-validation accuracy of the model is slightly higher at 88.32% compared to the overall accuracy indicates that the model's performance is consistent across different subsets of the data.

Recall: The recall of the model is 85.95% suggests that the model is good at capturing a large portion of the positive instances, which means that out of 100 'good' classes, it correctly predicts almost 86 of them.

Precision: The precision of the model is 87.52% indicates that when the model predicts an instance as positive, it is correct approximately 87.52% of the time.

F1 Score: The F1 score of the model is 86.58%. A high F1 score suggests that the model achieves a good balance between precision and recall.

Conclusion: Random Forest Classifier demonstrated a good performance, with a good balance for predicting both classes, as showed by the metrics. To try to increase the its performance, we tried to use Grid Search algorithm to find the best parameters.

3.6.1 Random Forest after Hyperparameter Tuning

: By applying GridSearch the best Hyperparameters for the Random Forest model was found to be 'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200 RandomForestClassifier(n_estimators=200, random_state=12345). On using these hyperparameter values to the Random Forest model we get the following result.

Accuracy: Grid Search (86.42%) vs. Without Grid Search (86.58%): The accuracy achieved after hyperparameter tuning using grid search is slightly lower than the accuracy without tuning. While the difference is small, it suggests that the default parameters used in the Random Forest model might have been close to optimal for this dataset.

Cross-Validation Accuracy: Grid Search (88.6%) vs. Without Grid Search (88.33%): The cross-validation accuracy after hyperparameter tuning is slightly higher than the accuracy without tuning. This indicates that hyperparameter tuning has slightly improved the model's generalization performance.

Recall: Grid Search (86.02%) vs. Without Grid Search (85.95%): The recall scores are quite similar between the two approaches, suggesting that hyperparameter tuning did not significantly affect the model's ability to capture positive instances.

Precision: Grid Search (87.02%) vs. Without Grid Search (87.52%): The precision achieved after hyperparameter tuning is slightly lower than the precision without tuning. This indicates that the model after tuning may be slightly less conservative in predicting positive instances.

F1 Score: Grid Search (86.42%) vs. Without Grid Search (86.58%): The F1 scores are similar between the two approaches, indicating a balanced performance between precision and recall.

Conclusion: Overall, while hyperparameter tuning using grid search has led to slight improvements in cross-validation accuracy and has resulted in a more generalized model, the differences in performance metrics between the tuned and un-tuned models are relatively small. This suggests that the default parameters used in the Random Forest model were already quite effective for this dataset, and the additional tuning did not lead to significant improvements. However, hyperparameter tuning is still a valuable technique to explore as it can potentially lead to better performance on other datasets or with different model configurations.

3.7 Result Summary

Overall, SVM and KNN with hyperparameter tuning stand out as the top-performing models across multiple metrics, showcasing their effectiveness in accurately classifying instances while maintaining a good balance between precision and recall.

Model	Accuracy	Cross validation accuracy	Recall	Precision	F1 score
Perceptron	70.33	66.07	70.59	57.90	67.10
KNN	86.83	89.15	86.61	87.19	86.83
KNN with hyperparameter tuning	89.08	89.83	88.52	89.85	89.08
Decision tree	79.58	80.95	80.07	78.87	79.58
SVM	87.17	89.15	86.70	87.85	87.17
SVM with hyperparameter tuning	89.08	90.68	90.10	87.85	89.08
Random Forest	86.58	88.33	85.95	87.52	86.58
Random Forest with hyperparameter tuning	86.42	88.6	86.02	87.02	86.42

Table 1: Result summary

As seen above, SVM and KNN classifiers have very similar performance scores. Therefore, we should consider several factors to choose which is best for this project. SVM is usually more computationally efficient, especially when dealing with larger datasets and higher dimensions, whereas KNN's inference time increases with dataset size. KNN is more interpretable since it directly uses nearest neighbors, while SVM might require tuning parameters like kernel choice and regularisation. SVM can be more robust to noisy data and class imbalance, but its training time is longer. Considering the characteristics of this project, we recommend using the SVM algorithm as it is more computationally efficient and robust. However, KNN would still be an optimal solution thanks to its high interpretability.

4 Conclusion

Overall, this project helped us to actively think and implement solutions to reach our goals. It was useful to train in datasets pre-processing and models implementation. Comparing models using different metrics and trying to get higher accuracy by implementing an hyperparameters tuning technique was essential to better understand the important concepts of data science.

References

[1] Joel Grus. Data science from scratch: First principles with python, 2015.

[2] Nelgiri Withana. Apple quality dataset. https://www.kaggle.com/datasets/nelgiriyewithana/apple-quality, 2024. Accessed: 15 March 2024.