Applied AI (D7041E) - Project

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

January 9, 2025

Link to GitHub Repository

Introduction

This reports describe our approach to complete the Mini Project. In particular, we followed the grading criteria for Grade 5.

1. We performed an extensive set of experiments and evaluated the performance of 20+ (22) datasets from 121 UCI datasets. See subsection Datasets.
2. We develop and evaluated more than 2 unsupervised (3 unsupervised models: Kmeans, GaussianMixture and Birch) and more than 2 supervised classification models (KNN, SVM, Random Forest, Perceptron, Decision Tree). See subsection Models Implementation.
3. We compared the models' performance to the baseline performance reported in the paper. See section Results, in particular Comparison with Paper.
4. We implemented pre-processing techniques, while keeping the code organized and efficient. See subsection Data Loading and Preprocessing.

In the next paragraphs, we will deep dive into all the steps and corresponding techniques implemented, explaining classes and methods in details (the code will be shown as snapshots of the jupyter notebook). We will also show and discuss the results obtained, as requested by the grading criteria, also reporting the resulting confusion matrices.

1 Methodology

This section provides a structured methodology for conducting each step the Project.

1.1 Datasets

1. **Wine Quality Dataset**
   - Features: Chemical properties like acidity, residual sugar, pH, etc.
   - Target: Wine type (Red/White).
2. **Spambase Dataset**
   - Features: Word frequencies, capital letters usage.
   - Target: Spam or not.
3. **Statlog (Australian Credit Approval)**
   - Features: Applicant attributes like age, income, and credit history.
   - Target: Credit approved or not.
4. **Pima Indians Diabetes Dataset**
   - Features: Glucose level, blood pressure, age, etc.

- Target: Diabetic or not.

5. **Twonorm Dataset**
    - Features: 20 numerical features.
    - Target: Class 0 or 1.
6. **Blood Transfusion Service Center Dataset**
    - Features: Donation frequency, time since last donation, etc.
    - Target: Donated blood or not.
7. **Breast Cancer Wisconsin (Original)**
    - Features: Cell size, shape, and texture.
    - Target: Benign or malignant tumor.
8. **Breast Cancer Wisconsin (Diagnostic)**
    - Features: Texture, area, smoothness of cells.
    - Target: Benign or malignant tumor.
9. **ILPD (Indian Liver Patient Dataset)**
    - Features: Enzyme levels, bilirubin, albumin, etc.
    - Target: Liver disease or not.
10. **Mammographic Mass Dataset**
    - Features: Age, shape, margin, and density of masses.
    - Target: Benign or malignant tumor.
11. **Parkinsons Dataset**
    - Features: Voice measurements like jitter, shimmer, etc.
    - Target: Parkinson's disease or not.
12. **Planning Dataset**
    - Features: Binary features for planning activities.
    - Target: Planning state or relax state.
13. **SPECT Heart Dataset**
    - Features: Binary attributes for heart diagnosis.
    - Target: Normal or abnormal heart condition.
14. **SPECTF Heart Dataset**
    - Features: Quantitative features from heart scans.
    - Target: Normal or abnormal heart condition.
15. **Vertebral Column (2-class)**
    - Features: Pelvic tilt, lumbar angle, etc.
    - Target: Normal or abnormal spine.
16. **Acute Nephritis Dataset (Bladder Inflammation)**
    - Features: Symptoms like nausea, pain, etc.
    - Target: Inflammation present or not.
17. **Acute Nephritis Dataset (Renal Pelvis Nephritis)**
    - Same dataset as above but with a different target.

    - Target: Nephritis present or not.

18. **Breast Cancer Wisconsin (Prognostic)**
    - Features: Tumor size, recurrence time, etc.
    - Target: Recur or not.

19. **Connectionist Bench (Sonar, Mines vs. Rocks)**
    - Features: Frequency-based sonar readings.
    - Target: Rock or mine.

20. **Echocardiogram Dataset**
    - Features: Heart-related attributes like survival time, wall motion score, etc.
    - Target: Alive 1 year later or not.

21. **Fertility Diagnosis Dataset**
    - Features: Age, diseases, habits like smoking.
    - Target: Fertility normal or altered.

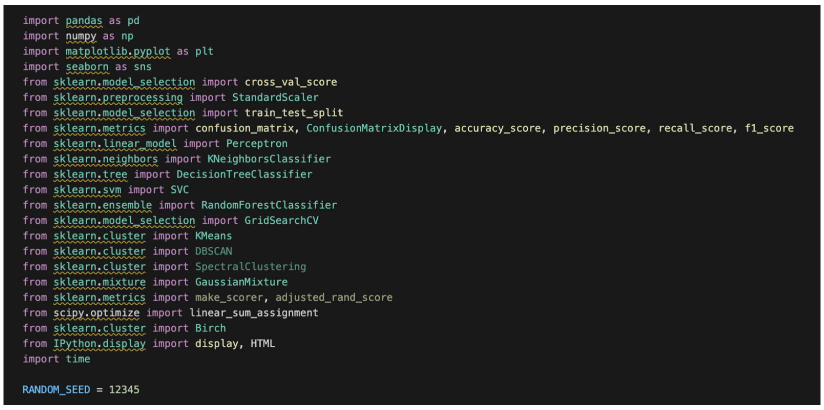
22. **Haberman's Survival Dataset**
    - Features: Age, year of surgery, and positive axillary nodes.
    - Target: Survival 5 years or not.

1.2 Techniques

The techniques employed in this project vary depending on the dataset and models we were dealing with. We mostly used object-oriented programming (OOPs) to build classes and design methods.

- **Libraries**: the libraries and modules imported for this project can be found in Figure 1.
  Below libraries are described and its use are 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 built on matplotlib, designed 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.
  - **confusion_matrix and performance metrics from sklearn.metrics**: A module of sklearn with main functions of calculating performance metrics like precision or recall, and to record actual and predicted labels into a confusion matrix.
  - **Perceptron from sklearn.linear_model**: A class in sklearn in which single-neural layer classifier (perceptron) is implemented.
  - **KNeighborsClassifier from sklearn.neighbors**: A class in sklearn in which K-nearest neighbors classifier is implemented.
  - **DecisionTreeClassifier from sklearn.tree**: A class in sklearn in which decision tree classifier is implemented.
  - **SVC from sklearn.svm**: A class in sklearn in which SVM classifier is implemented.

- **RandomForestClassifier from sklearn.ensemble**: 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.
- **KMeans from sklearn.cluster**: A class in sklearn in which KMeans clustering algorithm is implemented.
- **GaussianMixture from sklearn.mixture**: A function in scikit-for the Gaussian Mixture Model.
- **Birch from sklearn.cluster**: A class in sklearn in which the Birch Clustering algorithm is implemented.
- **linear_sum_assignment from scipy.optimize**: A SciPy function used to optimize task assignments in clustering.
- **IPython.display**: A module for rendering rich content like HTML and images directly within Jupyter notebooks.
- **time**: A module for measuring time.



```
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
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn.cluster import SpectralClustering
from sklearn.mixture import GaussianMixture
from sklearn.metrics import make_scorer, adjusted_rand_score
from scipy.optimize import linear_sum_assignment
from sklearn.cluster import Birch
from IPython.display import display, HTML
import time

RANDOM_SEED = 12345
```

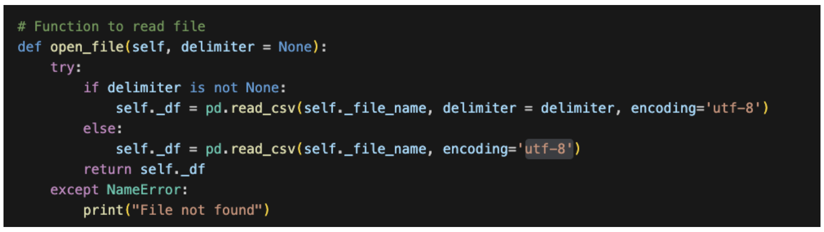
Figure 1: Code for importing modules and libraries.

Furthermore, we used a fixed Random Seed, in this case '12345', to reduce randomness and being able to compare the results of different models and methods.

- **Data Loading and Preprocessing**: As we had to test the chosen models (which will be presented further in the report) with 22 datasets, we tried to handle all cases in the most effective and least time consuming way, writing a code that could be reused to load, and later, to pre-process the dataset by calling a specific method. First of all, the datasets above provided were downloaded from various sources in formats such as CSV, text files, and ARFF. To load the data and use them in our project, we created the 'DataOperation' class, which facilitate the loading, cleaning, and merging of datasets. The class has methods to read files, handle missing values, concatenate datasets, and save processed data.

  Below the DataOperation class' methods are listed and described:

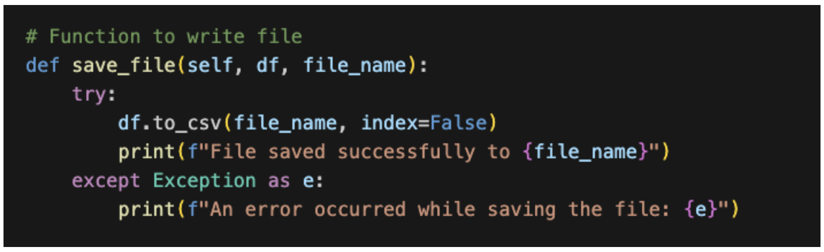
  1. `open_file(delimiter=None)`
     - Reads the dataset file.
     - Deals with delimiters, if present, and encode to utf-8 .
     - Returns the dataframe.
     - Prints error if it fails.



```
# Function to read file
def open_file(self, delimiter = None):
    try:
        if delimiter is not None:
            self._df = pd.read_csv(self._file_name, delimiter = delimiter, encoding='utf-8')
        else:
            self._df = pd.read_csv(self._file_name, encoding='utf-8')
        return self._df
    except NameError:
        print("File not found")
```

Figure 2: Open File function.

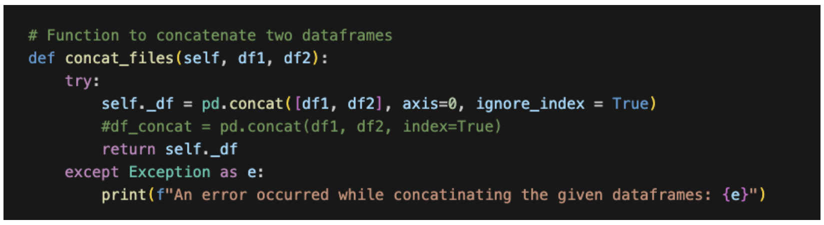
2. `save_file(df, file_name)`
   - Saves a dataframe to the specified file as csv.
   - Prints error if it fails.



```python
# Function to write file
def save_file(self, df, file_name):
    try:
        df.to_csv(file_name, index=False)
        print(f"File saved successfully to {file_name}")
    except Exception as e:
        print(f"An error occurred while saving the file: {e}")
```

Figure 3: Save File function.

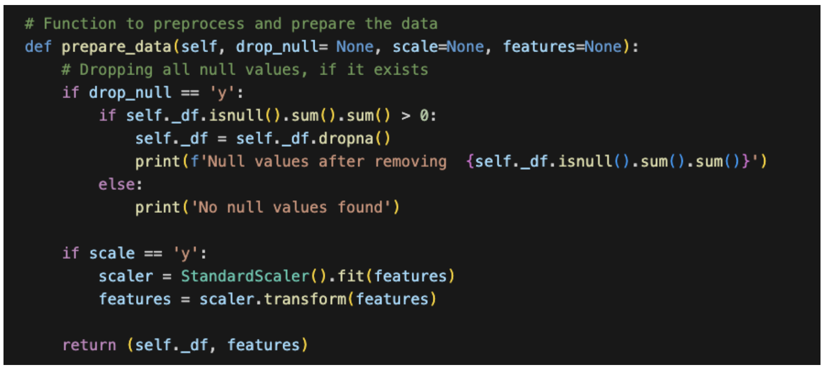
3. `concat_files(df1, df2)`
   - Combines two dataframes on rows, while removing redundant indices
   - Returns the resulting dataframe.
   - Prints error if it fails.



```python
# Function to concatenate two dataframes
def concat_files(self, df1, df2):
    try:
        self._df = pd.concat([df1, df2], axis=0, ignore_index = True)
        #df_concat = pd.concat(df1, df2, index=True)
        return self._df
    except Exception as e:
        print(f"An error occurred while concatinating the given dataframes: {e}")
```

Figure 4: Concat Files function.

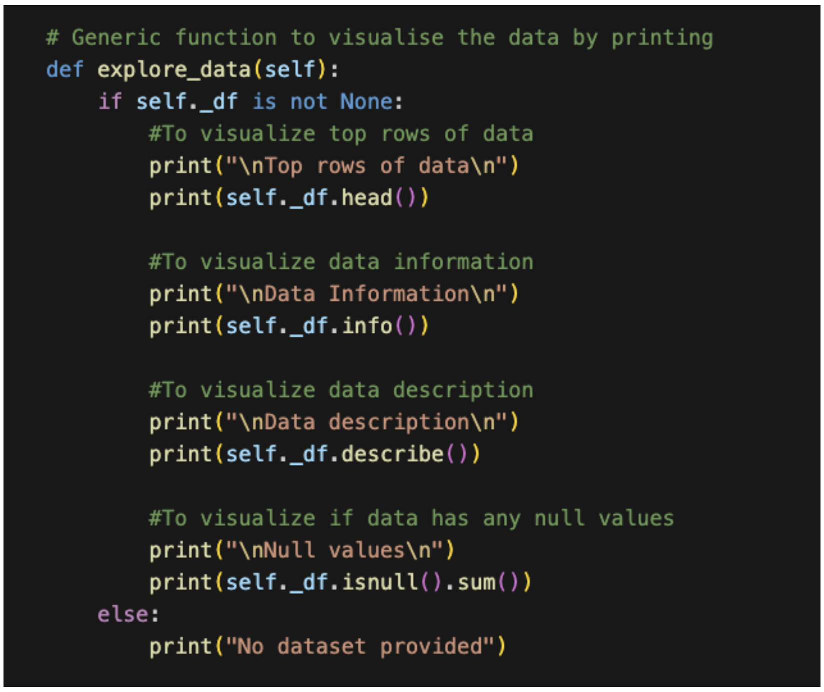
4. `prepare_data(drop_null=None, scale=None, features=None)`
   - Handles different preprocessing tasks, e.g.:
     * Drops rows with missing values (`drop_null`).
     * Scales numerical features (`StandardScaler`).
   - Returns the processed dataframe and features.



```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 5: Concat Files function.

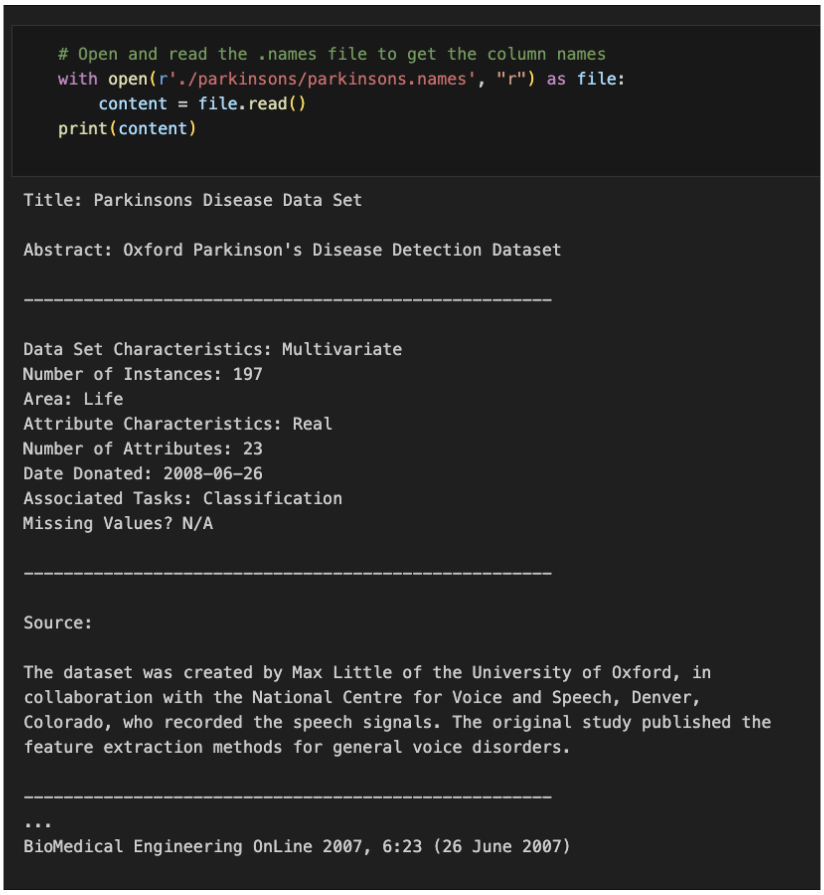
5. `explore_data()`
   - Print the first rows of the dataset.
   - Print information about the dataset,
   - Print the description of the dataset (simple statistics of each features).
   - Print the total number of missing values in the dataset.



```
# 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 6: Explore Data function.

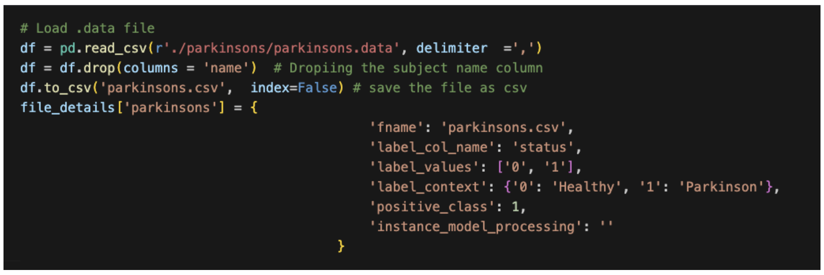
- **More Data Handling**: Other than the function above described, each dataset was handled based on their content and specific pre-processing requirements, e.g., handling columns names, dropping specific columns, etc. Figures 7 and 8, it can be seen that the .names file was first open and its content printed to get the column names. Hence, the .data file (csv) was opened as a dataframe using the pandas library, the column 'name' was dropped and then the obtained dataframe was saved as a csv.



```
# Open and read the .names file to get the column names
with open(r'./parkinsons/parkinsons.names', "r") as file:
    content = file.read()
print(content)
```

```
Title: Parkinsons Disease Data Set

Abstract: Oxford Parkinson's Disease Detection Dataset

-----------------------------------------------------

Data Set Characteristics: Multivariate
Number of Instances: 197
Area: Life
Attribute Characteristics: Real
Number of Attributes: 23
Date Donated: 2008-06-26
Associated Tasks: Classification
Missing Values? N/A

-----------------------------------------------------

Source:

The dataset was created by Max Little of the University of Oxford, in
collaboration with the National Centre for Voice and Speech, Denver,
Colorado, who recorded the speech signals. The original study published the
feature extraction methods for general voice disorders.

-----------------------------------------------------
...
BioMedical Engineering OnLine 2007, 6:23 (26 June 2007)
```

Figure 7: Example of one of the 22 dataset pre-processed (1).



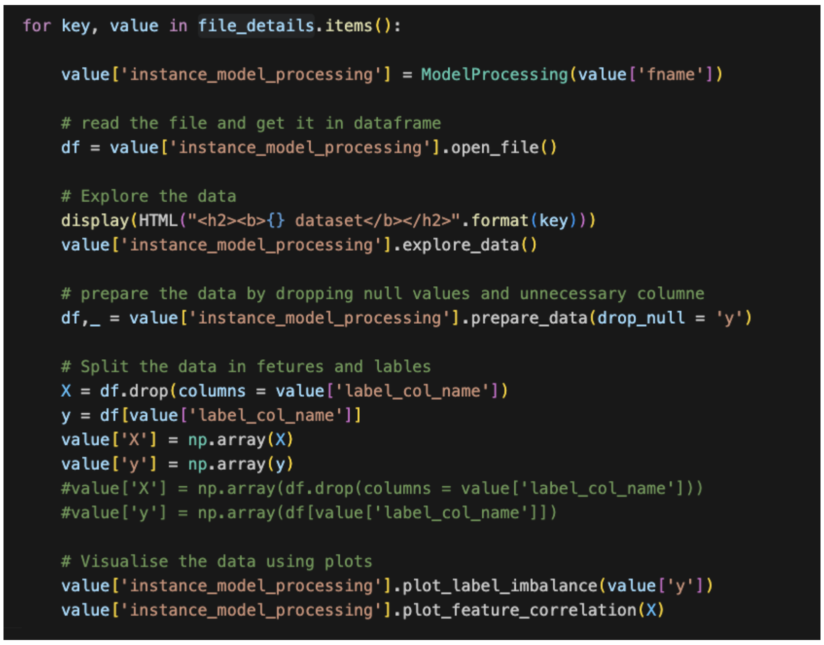
```
# Load .data file
df = pd.read_csv(r'./parkinsons/parkinsons.data', delimiter  =',')
df = df.drop(columns = 'name')  # Dropiing the subject name column
df.to_csv('parkinsons.csv',  index=False) # save the file as csv
file_details['parkinsons'] = {
                                           'fname': 'parkinsons.csv',
                                           'label_col_name': 'status',
                                           'label_values': ['0', '1'],
                                           'label_context': {'0': 'Healthy', '1': 'Parkinson'},
                                           'positive_class': 1,
                                           'instance_model_processing': ''
                                       }
```

Figure 8: Example of one of the 22 dataset pre-processed (2).

Each file details, as shown in the example in Figure **??**, is stored in a dictionary with keys:

- fname
- label_col_name
- label_values
- label_context
- positive_class
- instance_model_processing

This dictionary is used to organize metadata and processing instances for datasets. It associates each dataset with a 'ModelProcessing' instance for preprocessing and visualization tasks. This allows a unified and consistent pipeline for efficiently handling and exploring different datasets (Figure 9).

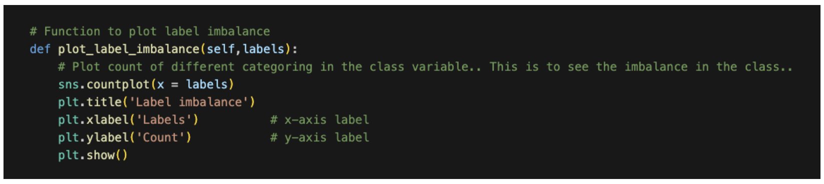


```python
for key, value in file_details.items():

    value['instance_model_processing'] = ModelProcessing(value['fname'])

    # read the file and get it in dataframe
    df = value['instance_model_processing'].open_file()

    # Explore the data
    display(HTML("<h2><b>{} dataset</b></h2>".format(key)))
    value['instance_model_processing'].explore_data()

    # prepare the data by dropping null values and unnecessary columne
    df,_ = value['instance_model_processing'].prepare_data(drop_null = 'y')

    # Split the data in fetures and lables
    X = df.drop(columns = value['label_col_name'])
    y = df[value['label_col_name']]
    value['X'] = np.array(X)
    value['y'] = np.array(y)
    #value['X'] = np.array(df.drop(columns = value['label_col_name']))
    #value['y'] = np.array(df[value['label_col_name']])

    # Visualise the data using plots
    value['instance_model_processing'].plot_label_imbalance(value['y'])
    value['instance_model_processing'].plot_feature_correlation(X)
```

Figure 9: Code example of using file details to associate each dataset with a ‘ModelProcessing‘ instance for preprocessing.

- **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.

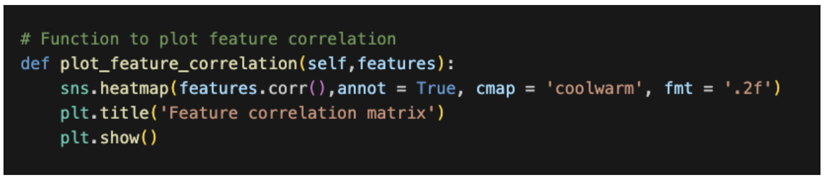
  1. `plot_label_imbalance(self,labels)`
     Figure 10 is a method created to show how many times a label appear in the dataset, so how many rows we have about a specific label. This is very useful to check for data imbalance, which can hinder the training process and prediction accuracy of a classification model.



```python
# Function to plot label imbalance
def plot_label_imbalance(self,labels):
    # Plot count of different categoring in the class variable.. This is to see the imbalance in the class..
    sns.countplot(x = labels)
    plt.title('Label imbalance')
    plt.xlabel('Labels')          # x-axis label
    plt.ylabel('Count')           # y-axis label
    plt.show()
```

Figure 10: Code for Bar plot showing the number of rows per classes.

  2. `plot_feature_correlation(self,features)`
     Figure 11 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.

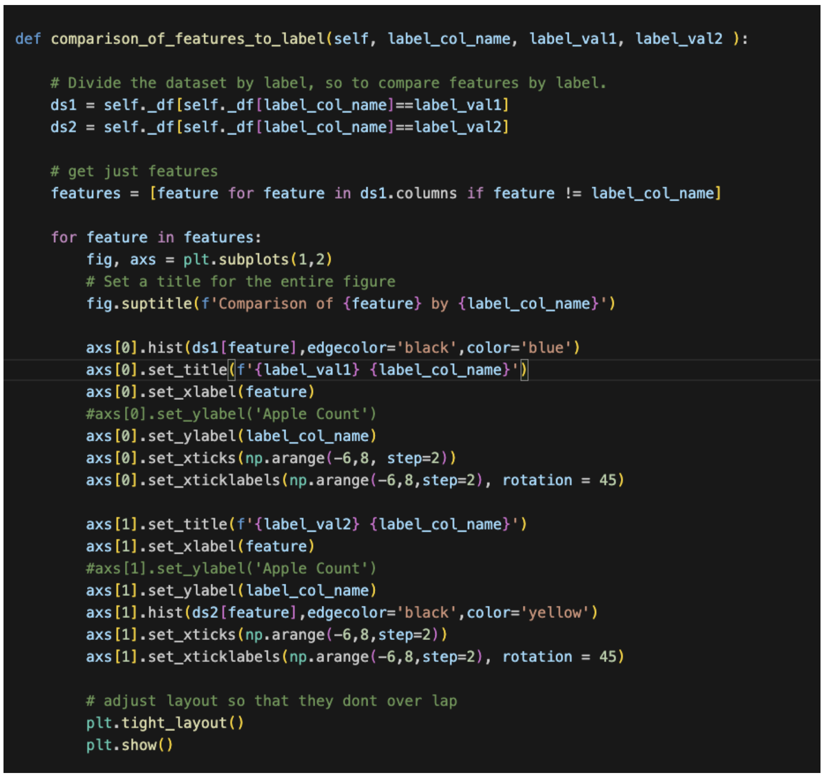


```python
# Function to plot feature correlation
def plot_feature_correlation(self,features):
    sns.heatmap(features.corr(),annot = True, cmap = 'coolwarm', fmt = '.2f')
    plt.title('Feature correlation matrix')
    plt.show()
```

Figure 11: Code for Correlation Matrix.

  3. `comparison_of_features_to_label(self,label_col_name, label_val1, label_val2)`
     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 and we used histogram plots to see how features were distributed (Figure 12).
The colours of the two plots (blue and yellow) were chosen by taking into consideration colorblindness syndrome.

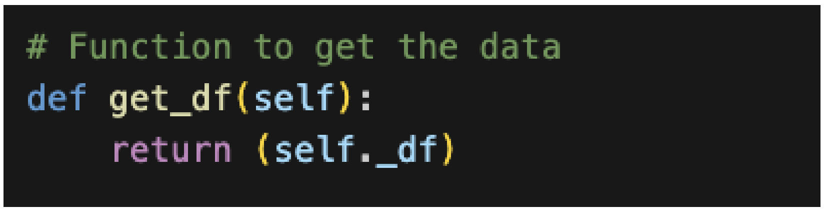


```python
def comparison_of_features_to_label(self, label_col_name, label_val1, label_val2 ):

    # 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_ylabel(label_col_name)
        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].set_ylabel(label_col_name)
        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 12: Code for Histogram visualization function.

- **Models Implementation**: We built the 'ModelProcessing' class, which is the child class of 'DataOperation' and 'PlotVisualisation', to apply the machine learning models to our datasets. In particular, it handles different tasks from preparing data for analysis to training models, as well as providing the models' performance. This class was designed to be used with both classification (supervised) and clustering (unsupervised) models and it adapts how the performance is evaluated based on the type of model used. Furthermore, it also prints the the confusion matrix for a better understanding of the results obtained.
Below, the class' methods are listed and described with their corresponding code:

  1. `get_df(self)` The method retrieves and returns the DataFrame (`_df`) associated with the class object.

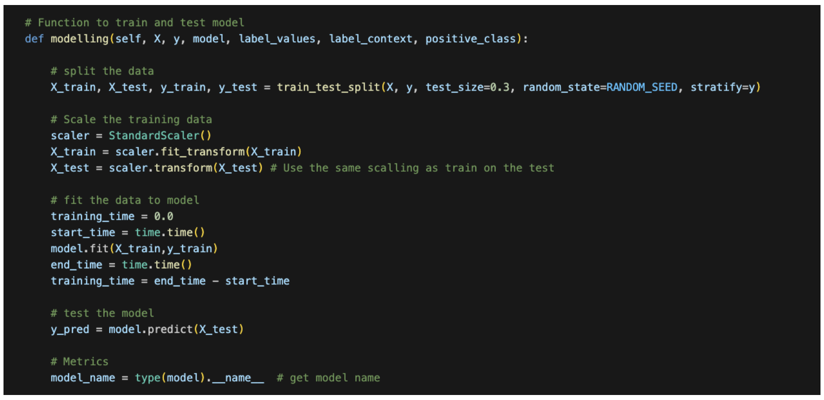


```python
# Function to get the data
def get_df(self):
    return (self._df)
```

Figure 13: Code for Get DataFrame Function.

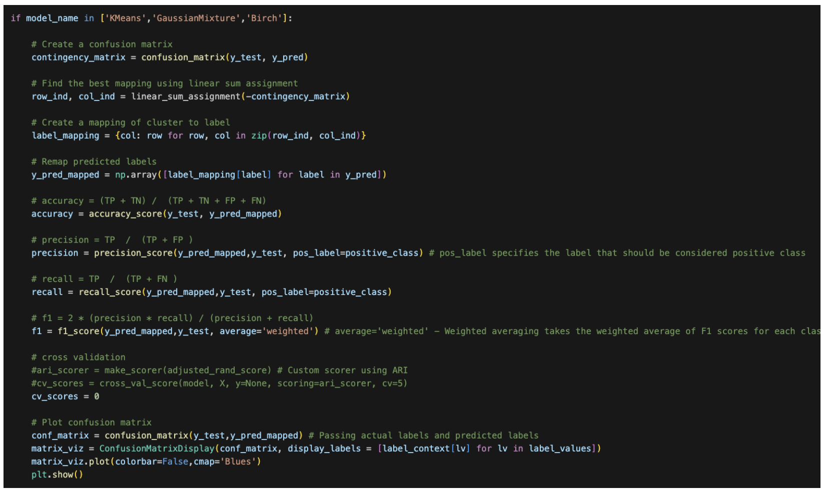
  2. `modelling(self, X, y, model, label_values, label_context, positive_class)`
  The method:
     - Splits the data into training and testing sets (with a 70-30 ratio).
     - Scales the features using the `StandardScaler`.
     - Trains the model while calculating its training time.
     - Makes predictions by passing the test set to the trained model.

- If it is an Unsupervised Model: it deals with label mapping and calculate accuracy, precision, recall, and F1 score.
- If it is a Supervised Model: it calculates accuracy, precision, recall, and F1 score.
- Display the resulting confusion matrix.
- Returns the accuracy, precision, recall, F1 score, cross-validation scores, and training time.



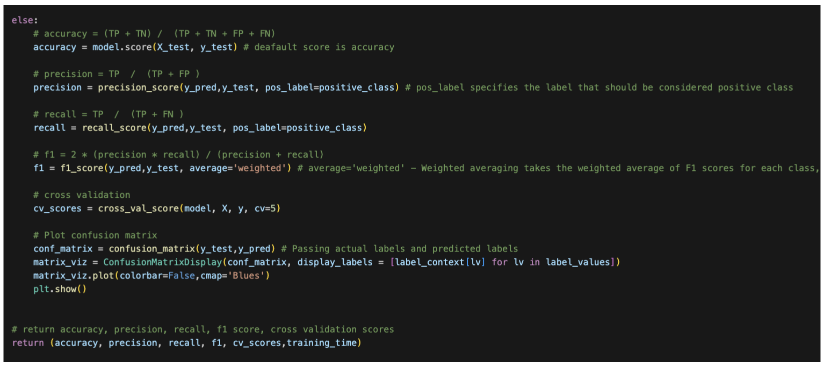
```
# Function to train and test model
def modelling(self, X, y, model, label_values, label_context, positive_class):

    # 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)

    # Scale the training data
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test) # Use the same scalling as train on the test

    # fit the data to model
    training_time = 0.0
    start_time = time.time()
    model.fit(X_train,y_train)
    end_time = time.time()
    training_time = end_time - start_time

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

    # Metrics
    model_name = type(model).__name__  # get model name
```

Figure 14: First Part of the Modelling Function.



```
if model_name in ['KMeans','GaussianMixture','Birch']:

    # Create a confusion matrix
    contingency_matrix = confusion_matrix(y_test, y_pred)

    # Find the best mapping using linear sum assignment
    row_ind, col_ind = linear_sum_assignment(-contingency_matrix)

    # Create a mapping of cluster to label
    label_mapping = {col: row for row, col in zip(row_ind, col_ind)}

    # Remap predicted labels
    y_pred_mapped = np.array([label_mapping[label] for label in y_pred])

    # accuracy = (TP + TN) /  (TP + TN + FP + FN)
    accuracy = accuracy_score(y_test, y_pred_mapped)

    # precision = TP  /  (TP + FP )
    precision = precision_score(y_pred_mapped,y_test, pos_label=positive_class) # pos_label specifies the label that should be considered positive class

    # recall = TP  /  (TP + FN )
    recall = recall_score(y_pred_mapped,y_test, pos_label=positive_class)

    # f1 = 2 * (precision * recall) / (precision + recall)
    f1 = f1_score(y_pred_mapped,y_test, average='weighted') # average='weighted' - Weighted averaging takes the weighted average of F1 scores for each clas

    # cross validation
    #ari_scorer = make_scorer(adjusted_rand_score) # Custom scorer using ARI
    #cv_scores = cross_val_score(model, X, y=None, scoring=ari_scorer, cv=5)
    cv_scores = 0

    # Plot confusion matrix
    conf_matrix = confusion_matrix(y_test,y_pred_mapped) # Passing actual labels and predicted labels
    matrix_viz = ConfusionMatrixDisplay(conf_matrix, display_labels = [label_context[lv] for lv in label_values])
    matrix_viz.plot(colorbar=False,cmap='Blues')
    plt.show()
```

Figure 15: Special handling of predictions from Unsupervised models to get metrics.

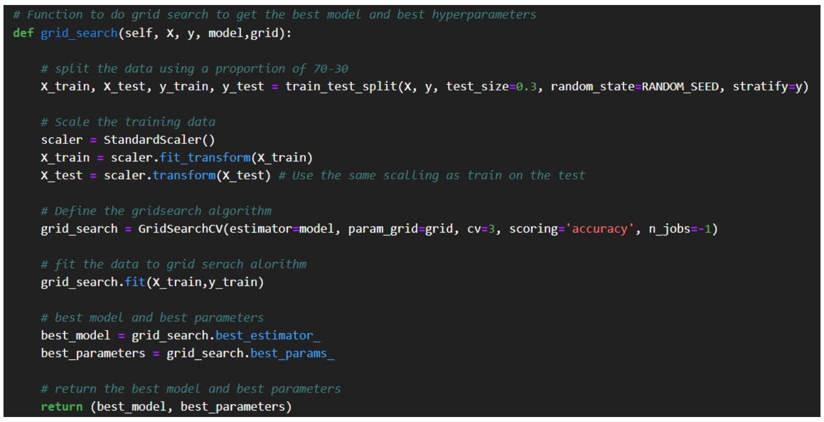


```
else:
    # accuracy = (TP + TN) /  (TP + TN + FP + FN)
    accuracy = model.score(X_test, y_test) # deafault score is accuracy

    # precision = TP  /  (TP + FP )
    precision = precision_score(y_pred,y_test, pos_label=positive_class) # pos_label specifies the label that should be considered positive class

    # recall = TP  /  (TP + FN )
    recall = recall_score(y_pred,y_test, pos_label=positive_class)

    # f1 = 2 * (precision * recall) / (precision + recall)
    f1 = f1_score(y_pred,y_test, average='weighted') # average='weighted' - Weighted averaging takes the weighted average of F1 scores for each class,

    # cross validation
    cv_scores = cross_val_score(model, X, y, cv=5)

    # Plot confusion matrix
    conf_matrix = confusion_matrix(y_test,y_pred) # Passing actual labels and predicted labels
    matrix_viz = ConfusionMatrixDisplay(conf_matrix, display_labels = [label_context[lv] for lv in label_values])
    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,training_time)
```

Figure 16: Getting metrics for Supervised models.

3. `grid_search(self, X, y, model,grid)`
   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.

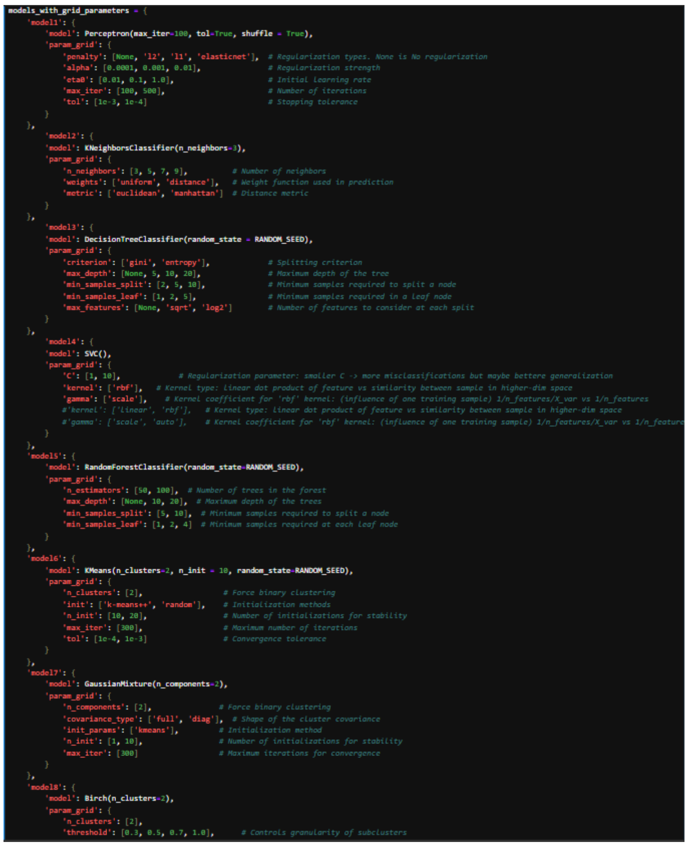
  - Splits the data into training and testing subsets (70-30 split).
  - Scales the data with `StandardScaler`.
  - Use grid search by using `GridSearchCV` object, previously imported from sklearn.model_selection, to identify the best hyperparameters.
  - Returns the best model and parameters.



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

    # split the data using a proportion of 70-30
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=RANDOM_SEED, stratify=y)

    # Scale the training data
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test) # Use the same scalling as train on the test

    # Define the gridsearch algorithm
    grid_search = GridSearchCV(estimator=model, param_grid=grid, cv=3, scoring='accuracy', n_jobs=-1)

    # 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 17: Code for Grid Search Function.

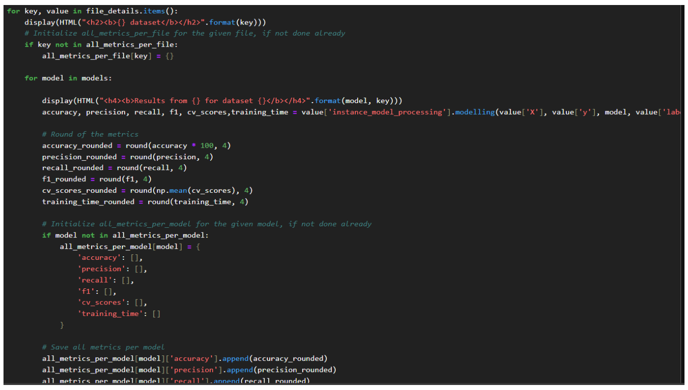
For a better understanding, the search space used to find the best hyperparameter for each model is as shown in 18



```python
models_with_grid_parameters = {
    'model1': {
        'model': Perceptron(max_iter=100, tol=True, shuffle = True),
        'param_grid': {
            'penalty': [None, 'l2', 'l1', 'elasticnet'],  # Regularization types. None is No regularization
            'alpha': [0.0001, 0.001, 0.01],               # Regularization strength
            'eta0': [0.01, 0.1, 1.0],                     # Initial learning rate
            'max_iter': [100, 500],                       # Number of iterations
            'tol': [1e-3, 1e-4]                           # Stopping tolerance
        }
    },
        'model2': {
        'model': KNeighborsClassifier(n_neighbors=3),
        'param_grid': {
            'n_neighbors': [3, 5, 7, 9],          # Number of neighbors
            'weights': ['uniform', 'distance'],   # Weight function used in prediction
            'metric': ['euclidean', 'manhattan']  # Distance metric
        }
    },
        'model3': {
        'model': DecisionTreeClassifier(random_state = RANDOM_SEED),
        'param_grid': {
            'criterion': ['gini', 'entropy'],             # Splitting criterion
            'max_depth': [None, 5, 10, 20],               # Maximum depth of the tree
            'min_samples_split': [2, 5, 10],              # Minimum samples required to split a node
            'min_samples_leaf': [1, 2, 5],                # Minimum samples required in a leaf node
            'max_features': [None, 'sqrt', 'log2']        # Number of features to consider at each split
        }
    },
        'model4': {
        'model': SVC(),
        'param_grid': {
            'C': [1, 10],             # Regularization parameter: smaller C -> more misclassifications but maybe bettere generalization
            'kernel': ['rbf'],   # Kernel type: linear dot product of feature vs similarity between sample in higher-dim space
            'gamma': ['scale'],    # Kernel coefficient for 'rbf' kernel: (influence of one training sample) 1/n_features/X_var vs 1/n_features
            #'kernel': ['linear', 'rbf'],   # Kernel type: linear dot product of feature vs similarity between sample in higher-dim space
            #'gamma': ['scale', 'auto'],    # Kernel coefficient for 'rbf' kernel: (influence of one training sample) 1/n_features/X_var vs 1/n_feature
        }
    },
    'model5': {
        'model': RandomForestClassifier(random_state=RANDOM_SEED),
        'param_grid': {
            'n_estimators': [50, 100],  # Number of trees in the forest
            'max_depth': [None, 10, 20],  # Maximum depth of the trees
            'min_samples_split': [5, 10],  # Minimum samples required to split a node
            'min_samples_leaf': [1, 2, 4]  # Minimum samples required at each leaf node
        }
    },
    'model6': {
        'model': KMeans(n_clusters=2, n_init = 10, random_state=RANDOM_SEED),
        'param_grid': {
            'n_clusters': [2],                    # Force binary clustering
            'init': ['k-means++', 'random'],      # Initialization methods
            'n_init': [10, 20],                   # Number of initializations for stability
            'max_iter': [300],                    # Maximum number of iterations
            'tol': [1e-4, 1e-3]                   # Convergence tolerance
        }
    },
    'model7': {
        'model': GaussianMixture(n_components=2),
        'param_grid': {
            'n_components': [2],                # Force binary clustering
            'covariance_type': ['full', 'diag'],  # Shape of the cluster covariance
            'init_params': ['kmeans'],          # Initialization method
            'n_init': [1, 10],                  # Number of initializations for stability
            'max_iter': [300]                   # Maximum iterations for convergence
        }
    },
    'model8': {
        'model': Birch(n_clusters=2),
        'param_grid': {
            'n_clusters': [2],
            'threshold': [0.3, 0.5, 0.7, 1.0],      # Controls granularity of subclusters
```

Figure 18: Parameter grid for each model

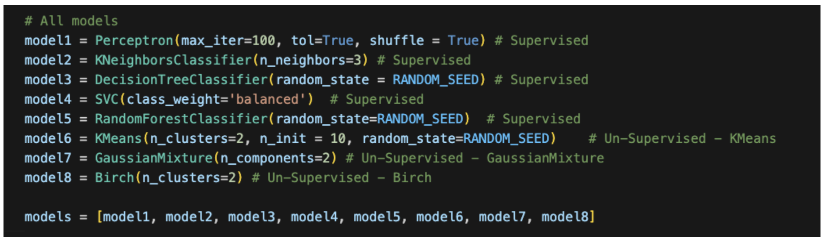
- **Training and testing loop**: A comprehensive training and test loop was developed to seamlessly train and test all 22 files on all the 8 models in an automatic way. Metrics such as Accuracy, Precision, Recall, F1-score, Training time and accuracy from using cross validation was recorded. These metrics were stored in dictionaries so as to perform comparative result analysis as described in the result section.



```python
for key, value in file_details.items():
    display(HTML("<h2><b>{} dataset</b></h2>".format(key)))
    # Initialize all_metrics_per_file for the given file, if not done already
    if key not in all_metrics_per_file:
        all_metrics_per_file[key] = {}

    for model in models:

        display(HTML("<h4><b>Results from {} for dataset {}</b></h4>".format(model, key)))
        accuracy, precision, recall, f1, cv_scores,training_time = value['instance_model_processing'].modelling(value['X'], value['y'], model, value['lab

        # Round of the metrics
        accuracy_rounded = round(accuracy * 100, 4)
        precision_rounded = round(precision, 4)
        recall_rounded = round(recall, 4)
        f1_rounded = round(f1, 4)
        cv_scores_rounded = round(np.mean(cv_scores), 4)
        training_time_rounded = round(training_time, 4)

        # Initialize all_metrics_per_model for the given model, if not done already
        if model not in all_metrics_per_model:
            all_metrics_per_model[model] = {
                'accuracy': [],
                'precision': [],
                'recall': [],
                'f1': [],
                'cv_scores': [],
                'training_time': []
            }

        # Save all metrics per model
        all_metrics_per_model[model]['accuracy'].append(accuracy_rounded)
        all_metrics_per_model[model]['precision'].append(precision_rounded)
        all_metrics_per_model[model]['recall'].append(recall_rounded)
```

Figure 19: Comprehensive training and test loop

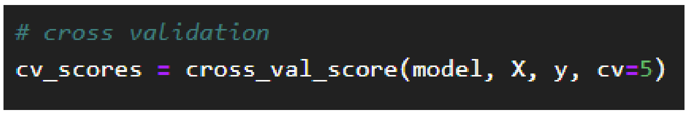
- **The Models Implemented**:
  The models implemented are:
  - **Perceptron**: single-layer neural network used for binary classification. It divides classes based on a linear decision boundary, and because of that, it does not handle well non-linear relationships.
  - **KNN**: classifier that makes 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.
  - **KMeans**: clustering algorithm that divide data into K clusters by minimizing the variance within each cluster (increasing the variance between clusters). Each data point is assigned to the cluster with the nearest centroid.
  - **Gaussian Mixture**: probabilistic clustering algorithm that divide data based on different Gaussian distributions. Each data point is assigned to the cluster depending on their probability to belong to one or another.
  - **Birch**: clustering algorithm with a hierarchical tree structure. Good for large datasets, it identifies dense regions and clusters data points while compressing sparse regions.



```python
# All models
model1 = Perceptron(max_iter=100, tol=True, shuffle = True) # Supervised
model2 = KNeighborsClassifier(n_neighbors=3) # Supervised
model3 = DecisionTreeClassifier(random_state = RANDOM_SEED) # Supervised
model4 = SVC(class_weight='balanced')  # Supervised
model5 = RandomForestClassifier(random_state=RANDOM_SEED)  # Supervised
model6 = KMeans(n_clusters=2, n_init = 10, random_state=RANDOM_SEED)    # Un-Supervised - KMeans
model7 = GaussianMixture(n_components=2) # Un-Supervised - GaussianMixture
model8 = Birch(n_clusters=2) # Un-Supervised - Birch

models = [model1, model2, model3, model4, model5, model6, model7, model8]
```

Figure 20: Code showing the models implemented.

- **Cross validation**:
  Cross validation has been implemented using cross_val_score function provided by model_selection module of scikit-learn. Here we pass the model together with the input and targets and cross-validation folds as 5 which results in the dataset to be split into 5 equal parts (folds), and the model is trained on 4 folds while the 5th is used for testing. This is repeated 5 times where each fold is used once as the test set.



```python
# cross validation
cv_scores = cross_val_score(model, X, y, cv=5)
```

Figure 21: Cross Validation.

2 Results

In the chapter, models are evaluated based on these metrics:

- **Accuracy**: This represents the percentage of correctly classified instances out of the total number of instances.
- **Recall**: 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**: 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 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.
- **CV Score**: Cross-validation accuracy provides an estimate of how well the model is likely to perform on unseen data. Precisely, it is the accuracy obtained by calculating the average accuracy across the different folds.
- **Training Time**: It represents the duration of the model's training phase.

2.1 Models Performance

Table 1: Average Model Performance Metrics across 22 Datasets before GridSearch

Model	**Accuracy (%)**	**Precision**	**Recall**	**F1-Score**	**CV Scores**	**Training Time (s)**
RandomForestClassifier(random_state=12345)	85.32	0.7769	0.8312	0.8666	0.8330	0.3021
SVC(class_weight='balanced')	82.97	0.8180	0.8036	0.8257	0.7160	0.024
KNeighborsClassifier(n_neighbors=3)	82.07	0.7451	0.7988	0.8296	0.7721	0.0015
Perceptron(max_iter=100, tol=True)	80.24	0.7706	0.7515	0.8054	0.7200	0.0018
DecisionTreeClassifier(random_state=12345)	78.47	0.7257	0.7438	0.7876	0.7704	0.014
KMeans(n_clusters=2, n_init=10, random_state=12345)	71.46	0.6760	0.6998	0.7267	0.0000	0.1057
Birch(n_clusters=2)	71.46	0.5965	0.6188	0.7563	0.0000	0.1284
GaussianMixture(n_components=2)	70.86	0.6093	0.6548	0.7202	0.0000	0.084

Based on the Table 1 we can conclude that:

- **Supervised Models:** The `RandomForestClassifier` was the model with the best performance, with the highest accuracy (85.32%) and F1-score (86.66%). Its training time, however, is relatively higher compared to other models, such as `SVC`.
- **Unsupervised Models:** `KMeans` and `Birch` achieved a moderate performance, with accuracy scores around 71.5%. Their precision and recall scores indicate a lower ability to differentiate classes effectively compared to supervised models.
- **Efficiency (Running Time):** `KNeighborsClassifier` registered the lowest training time (0.0015s). This was to be expected, as KNN belongs to the lazy learning models category, which has no training phase. However, it needs to be taken into account that it may require more storage.

Table 2: Average Model Performance Metrics across 22 Datasets after GridSearch

Model	**Accuracy (%)**	**Precision**	**Recall**	**F1-Score**	**CV Scores**	**Training Time (s)**
RandomForestClassifier(random_state=12345)	85.9994	0.7776	0.8635	0.8738	0.8391	0.2824
SVC()	85.8228	0.7769	0.7766	0.8803	0.7745	0.0248
KNeighborsClassifier(n_neighbors=3)	83.0913	0.7267	0.8099	0.8464	0.7803	0.0016
Perceptron(max_iter=100, tol=True)	82.57	0.7849	0.7532	0.8401	0.7198	0.0016
DecisionTreeClassifier(random_state=12345)	81.5533	0.7379	0.7534	0.8282	0.7908	0.0071
GaussianMixture(n_components=2)	73.6887	0.6427	0.6666	0.7587	0.0000	0.0716
Birch(n_clusters=2)	71.7764	0.6022	0.6201	0.7595	0.0000	0.1274
KMeans(n_clusters=2, n_init=10, random_state=12345)	71.4832	0.6763	0.6999	0.7270	0.0000	0.1033

Based on the Table 2 we can conclude that:

- **Supervised Models:** As before implementing Grid Search for Hyperparameter tuning, the `RandomForestClassifier` was the best-performing model having the highest accuracy (85.99%) and F1-score (87.38%).
- **Unsupervised Models:** Again, `KMeans` and `Birch` maintained moderate performance, with accuracy scores of 71.48% and 71.78%, respectively.
- **Efficiency (Running Time):** As before Grid Search, `KNeighborsClassifier` recorded the lowest training time (0.0016s). It also achieved a better accuracy (83.09%) and F1-score (84.64%).

  It needs to be noted, that even though `KNeighborsClassifier` and `RandomForestClassifier` achieved the best running time and accuracy respectively, `SVClassifier` after Grid Search provided a good performance (85.82%), very cloased to the one registered by the `RandomForestClassifier`, and a moderately low running time (0.02s). Hence, `SVClassifier` can be considered an excellent compromise.

2.2 Comparison with Paper

We will now compare the performance registered by our models with the one reported in the paper of Fernández-Delgado, Manuel, et al. Below, it is possible to notice the accuracy of **the most similar** model listed in the paper.

Table 3: Performance and Rank of Supervised and Unsupervised Models based on Paper

Model Type	**Model**	**Rank**	**Accuracy (%)**
	Supervised Models		
Supervised	k-Nearest Neighbors (kkn_R)	65.0	79.0
Supervised	Random Forest (rf_t)	33.1	82.3
Supervised	Perceptron (dkp_C)	59.9	79.7
Supervised	Support Vector Machine (svmRadial_t)	42.5	81.0
Supervised	Decision Tree (J48)	87.6	76.4
	Unsupervised Models		
Unsupervised	k-Means (ClassificationViaClustering_w)	157.4	52.1
Unsupervised	BIRCH	—	—
Unsupervised	Gaussian Mixture	—	—

All our models achieved a better performance compared to the models reported in the paper. In particular:

- **Supervised Models:**
  - The `RandomForestClassifier` achieved the highest accuracy (**85.99%**) and F1-score (**87.38%**), outperforming the paper's Random Forest model (**82.3%**). And as stated also in the paper, it is one of the most-performing models.
  - The `Support Vector Classifier (SVC)`, which balances accuracy (**85.82%**) and efficiency, training time (**0.02s**). It performed better than the one in the paper (in this case, Radial SVM) which ahieved an accuracy of **81.0%**
  - The `KNeighborsClassifier` registered a better accuracy (**83.09%**) compared to the paper's k-NN variant (**79.0%**).
  - The `Perceptron` with an accuracy of **82.57%** outperformed the paper's Perceptron variant (**79.7%**).
- **Unsupervised Models:**
  - The `KMeans` algorithm achieved an accuracy of **71.48%**, significantly higher than the paper's result (**52.1%**).

2.3 Performance of Models per Dataset

For each dataset, we wanted to see which models performed the best. So in the table below, we can see the models arranged in descending order of accuracy for each dataset. For e.g. For PIMA dataset, SVC performed the best with the accuracy of 77.05% followed by KNeighboursClassifier and so on.

File Name	Model	Accuracy(%)	Precision	Recall	F1	CV	Train time(s)
Pima	SVC	77.0563	0.5926	0.7059	0.7758	0.7591	0.008
Pima	KNeighborsClassifier	75.7576	0.5802	0.6812	0.7626	0.7136	0.0011
Pima	RandomForestClassifier	74.4589	0.5556	0.6618	0.7504	0.7618	0.1613
Pima	DecisionTreeClassifier	71.8615	0.4568	0.6379	0.7319	0.7462	0.002
Pima	KMeans	68.8312	0.6914	0.5437	0.682	0.0	0.1222
Pima	Birch	64.9351	0.4568	0.5	0.6533	0.0	0.0529
Pima	Perceptron	59.7403	0.4074	0.4231	0.5992	0.4622	0.001
Pima	GaussianMixture	59.7403	0.3827	0.4189	0.6019	0.0	0.1248
Planning	RandomForestClassifier	72.7273	0.0625	1.0	0.8256	0.7146	0.092
Planning	GaussianMixture	72.7273	0.25	0.5714	0.7665	0.0	0.0992
Planning	Perceptron	70.9091	0.0	0.0	0.8298	0.5715	0.001
Planning	KNeighborsClassifier	70.9091	1.25	0.5	0.777	0.6758	0.0011
Planning	DecisionTreeClassifier	70.9091	0.0	0.0	0.8298	0.6159	0.001
Planning	SVC	70.9091	0.0	0.0	0.8298	0.7089	0.0026
Planning	KMeans	56.3636	0.8125	0.3824	0.5505	0.0	0.0963
Planning	Birch	56.3636	0.5	0.3333	0.5449	0.0	0.015
SPECT	DecisionTreeClassifier	70.8333	0.5833	0.7778	0.713	0.7467	0.002
SPECT	Perceptron	66.6667	0.5	0.75	0.6762	0.6967	0.001
SPECT	SVC	66.6667	0.5	0.75	0.6762	0.72	0.001
SPECT	KMeans	66.6667	0.3333	1.0	0.7083	0.0	0.0946
SPECT	KNeighborsClassifier	62.5	0.5	0.6667	0.631	0.495	0.0
SPECT	RandomForestClassifier	62.5	0.5	0.6667	0.631	0.6833	0.1326
SPECT	GaussianMixture	62.5	0.25	1.0	0.6864	0.0	0.0206
SPECT	Birch	58.3333	0.1667	1.0	0.6709	0.0	0.009
SPECTF	SVC	79.17	0.75	0.8182	0.792	0.785	0.001
SPECTF	RandomForestClassifier	79.17	0.75	0.8182	0.792	0.7983	0.1442
SPECTF	Perceptron	75.00	1.00	0.6667	0.7667	0.4933	0.002
SPECTF	KNeighborsClassifier	62.50	0.25	1.00	0.6864	0.7208	0.001
SPECTF	DecisionTreeClassifier	62.50	0.5833	0.6364	0.6257	0.6483	0.004
SPECTF	KMeans	54.17	0.0833	1.00	0.6636	0.0	0.0959
SPECTF	GaussianMixture	50.00	0.0	0.0	0.6667	0.0	0.0227
SPECTF	Birch	50.00	0.0	0.0	0.6667	0.0	0.014
Spambase	RandomForestClassifier	95.66	0.9283	0.9601	0.9567	0.9294	0.6629
Spambase	SVC	93.19	0.8934	0.9310	0.9322	0.7322	0.2084
Spambase	KNeighborsClassifier	91.96	0.8566	0.9339	0.9203	0.8127	0.001
Spambase	DecisionTreeClassifier	91.60	0.8768	0.9068	0.9163	0.9028	0.0472
Spambase	Perceptron	85.16	0.8015	0.8180	0.8518	0.7205	0.003
Spambase	GaussianMixture	76.25	0.9026	0.6410	0.7605	0.0	0.1345
Spambase	Birch	60.32	0.0	0.0	0.7503	0.0	0.6047
Spambase	KMeans	59.96	0.0	0.0	0.7448	0.0	0.1848
Twonorm	SVC	98.06	0.9811	0.9802	0.9806	0.9777	0.2014
Twonorm	KMeans	97.97	0.9793	0.9801	0.9797	0.0	0.0843
Twonorm	GaussianMixture	97.93	0.9766	0.9819	0.9793	0.0	0.0624
Twonorm	RandomForestClassifier	97.57	0.9748	0.9765	0.9757	0.9705	2.4313
Twonorm	KNeighborsClassifier	97.30	0.9802	0.9662	0.9730	0.9649	0.0011
Twonorm	Perceptron	96.94	0.9684	0.9702	0.9694	0.9646	0.004
Twonorm	Birch	91.53	0.9657	0.8771	0.9155	0.0	1.2735
Twonorm	DecisionTreeClassifier	85.36	0.8557	0.8519	0.8536	0.8520	0.0397
Winequality	RandomForestClassifier	99.641	0.9896	0.9958	0.9964	0.9942	0.5921
Winequality	SVC	99.4872	0.9854	0.9937	0.9949	0.9358	0.0567
Winequality	KNeighborsClassifier	99.3846	0.9854	0.9895	0.9939	0.9292	0.0129
Winequality	Perceptron	99.3333	0.9812	0.9916	0.9933	0.9226	0.003
Winequality	DecisionTreeClassifier	98.6154	0.9792	0.9651	0.9861	0.9815	0.0286
Winequality	KMeans	98.0513	0.9833	0.9402	0.9804	0.0	0.0826
Winequality	Birch	96.6667	0.9854	0.8908	0.9661	0.0	0.5138
Winequality	GaussianMixture	94.7692	0.9854	0.8327	0.9463	0.0	0.1194

File Name	**Model**	**Accuracy(%)**	**Precision**	**Recall**	**F1**	**CV**	**Train time(s)**
blood_transfusion	SVC	79.5556	0.3148	0.6538	0.8236	0.7594	0.0096
blood_transfusion	RandomForestClassifier	78.6667	0.3333	0.6	0.8101	0.7274	0.049
blood_transfusion	Perceptron	76.8889	0.1481	0.5714	0.8248	658	0.0019
blood_transfusion	KNeighborsClassifier	76.8889	0.2778	0.5357	0.7974	0.6673	0.001
blood_transfusion	Birch	76.0	37	0.5	0.8481	0.0	0.021
blood_transfusion	DecisionTreeClassifier	74.2222	0.2222	0.4286	774	0.7127	0.001
blood_transfusionr	KMeans	69.3333	0.3333	0.3529	0.6964	0.0	0.1174
blood_transfusion	GaussianMixture	58.6667	0.6481	0.3211	0.5565	0.0	0.0206
breast_cancer	SVC	98.8304	0.9844	0.9844	0.9883	0.9122	0.005
breast_cancer	Perceptron	96.4912	0.9688	0.9394	0.9648	0.8278	0.001
breast_cancer	KNeighborsClassifier	96.4912	0.9219	0.9833	0.9652	0.9191	0.001
breast_cancer	RandomForestClassifier	96.4912	0.9531	0.9531	0.9649	0.9614	0.2091
breast_cancer	DecisionTreeClassifier	92.3977	0.9062	0.8923	0.9239	0.9191	0.008
breast_cancer	KMeans	92.3977	0.8594	0.9322	0.9247	0.0	0.1031
breast_cancer	Birch	92.3977	0.8438	0.9474	925	0.0	0.052
breast_cancer	GaussianMixture	91.8129	0.9688	0.8378	0.9172	0.0	0.0196
breast_cancer	RandomForestClassifier	79.3103	0.2143	0.75	0.84	0.7665	0.1623
breast_cancer	Perceptron	75.8621	0.7143	0.5	0.7462	0.5067	0.0011
breast_cancer	KNeighborsClassifier	75.8621	0.0714	0.5	0.8347	0.7043	0.0009
breast_cancer	SVC	75.8621	0.0	0.0	0.8627	0.7617	0.001
breast_cancer	GaussianMixture	75.8621	0.2143	0.5	0.7968	0.0	0.0301
breast_cancer	Birch	72.4138	0.2857	0.4	0.7411	0.0	0.0193
breast_cancer	KMeans	68.9655	0.7857	0.4231	0.6676	0.0	0.09
breast_cancer	DecisionTreeClassifier	58.6207	0.2857	0.2222	0.5702	673	0.004
connectionist	KNeighborsClassifier	87.3016	0.7931	0.92	0.8742	0.5304	0.001
connectionist	SVC	85.7143	0.7931	0.8846	858	0.6331	0.003
connectionist	RandomForestClassifier	80.9524	0.6897	0.8696	0.8128	0.7294	0.1674
connectionist	Perceptron	74.6032	0.6207	0.7826	0.7504	0.5761	0.002
connectionist	DecisionTreeClassifier	68.254	0.5172	0.7143	0.6912	0.5945	0.0011
connectionist	KMeans	55.5556	0.8966	0.5098	0.6007	0.0	0.0926
connectionist	GaussianMixture	53.9683	0.0	0.0	701	0.0	0.0335
connectionist	Birch	50.6329	0.5106	0.6129	0.5893	0.0	0.0189
diagnosis	Perceptron	100.0	1.0	1.0	1.0	1.0	0.001
diagnosis	KNeighborsClassifier	100.0	1.0	1.0	1.0	0.9917	0.001
diagnosis	DecisionTreeClassifier	100.0	1.0	1.0	1.0	1.0	0.001
diagnosis	SVC	100.0	1.0	1.0	1.0	0.4917	0.001
diagnosis	RandomForestClassifier	100.0	1.0	1.0	1.0	1.0	0.1289
diagnosis	GaussianMixture	75.0	0.5556	0.9091	0.7598	0.0	0.0635
diagnosis	KMeans	63.8889	0.4444	0.7273	0.6531	0.0	0.0935
diagnosis	Birch	63.8889	0.4444	0.7273	0.6531	0.0	0.0085
diagnosis_n	KNeighborsClassifier	100.0	1.0	1.0	1.0	875	0.001
diagnosis_n	DecisionTreeClassifier	100.0	1.0	1.0	1.0	0.8583	0.0011
diagnosis_n	SVC	100.0	1.0	1.0	1.0	0.5833	0.001
diagnosis_n	RandomForestClassifier	100.0	1.0	1.0	1.0	1.0	0.1291
diagnosis_n	Perceptron	97.2222	1.0	0.9375	0.9721	1.0	0.002
diagnosis_n	Birch	86.1111	0.6667	1.0	0.8676	0.0	0.0094
diagnosis_n	KMeans	61.1111	0.6667	0.5263	0.6087	0.0	0.0956
diagnosis_n	GaussianMixture	61.1111	0.6667	0.5263	0.6087	0.0	0.0214
echocardiogram	DecisionTreeClassifier	100.0	1.0	1.0	1.0	0.9513	0.001
echocardiogram	RandomForestClassifier	100.0	1.0	1.0	1.0	0.9846	0.1225
echocardiogram	KMeans	94.7368	1.0	0.8333	946	0.0	0.0886
echocardiogram	SVC	89.4737	1.0	0.7143	0.89	0.9526	0.001
echocardiogram	GaussianMixture	89.4737	1.0	0.7143	0.89	0.0	0.02
echocardiogram	Birch	89.4737	1.0	0.7143	0.89	0.0	0.0071
echocardiogram	Perceptron	84.2105	1.0	625	0.8334	0.9846	0.002
echocardiogram	KNeighborsClassifier	78.9474	0.8	0.5714	0.78	1.0	0.0

File Name	Model	Accuracy(%)	Precision	Recall	F1	CV	Train time(s)
fertility	Perceptron	86.6667	1.0	0.8667	0.9286	0.68	0.0
fertility	SVC	86.6667	1.0	0.8667	0.9286	0.88	0.001
fertility	KNeighborsClassifier	83.3333	0.9615	0.8621	0.8788	0.82	0.001
fertility	RandomForestClassifier	83.3333	0.9615	0.8621	0.8788	0.82	0.1296
fertility	DecisionTreeClassifier	80.0	0.8846	0.8846	0.8	0.73	0.001
fertility	GaussianMixture	80.0	0.9231	0.8571	0.8296	0.0	0.0606
fertility	Birch	66.6667	0.7692	0.8333	0.64	0.0	0.0115
fertility_Diagnosis	KMeans	56.6667	0.6538	0.8095	0.5064	0.0	0.095
haberman	SVC	78.2609	0.9559	0.7927	0.8172	0.7288	0.002
haberman	KNeighborsClassifier	77.1739	0.9559	0.7831	0.8123	0.7254	0.001
haberman	Birch	77.1739	1.0	764	0.8452	0.0	0.0145
haberman	RandomForestClassifier	76.087	0.9412	0.7805	0.7989	0.6796	0.1388
haberman	Perceptron	73.913	0.7941	0.8438	733	0.7352	0.0011
haberman	DecisionTreeClassifier	72.8261	0.8971	0.7722	0.7585	0.6566	0.002
haberman	GaussianMixture	63.0435	0.6324	0.8269	0.6089	0.0	0.1322
haberman	KMeans	53.2609	0.5294	766	0.5042	0.0	0.0945
indian_liver	RandomForestClassifier	76.4368	0.9274	0.7823	0.7896	0.6961	0.0752
indian_liver	Perceptron	71.2644	0.9597	0.7256	0.7885	0.5426	0.002
indian_liver	SVC	71.2644	1.0	0.7126	0.8322	715	0.0091
indian_liver	DecisionTreeClassifier	70.6897	0.9435	0.7267	0.7739	0.6961	0.002
indian_liver	KNeighborsClassifier	66.092	0.7903	0.7481	669	0.6478	0.0009
indian_liver	GaussianMixture	64.9425	0.5403	0.9437	0.6363	0.0	0.1623
indian_liver	KMeans	62.069	0.5887	0.8295	0.6027	0.0	0.1061
indian_liver	Birch	58.046	0.8065	0.6711	0.6312	0.0	0.0386

3 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.

Working on a project with multiple files and multiple models was a completely new experience for us. It was fun, challenging, and full of learning. As we worked with several files, we gained different domain knowledge (for e.g. we worked with many health-related files). We also had to figure out how to automate the process so that 22 files could be used to train and test with 8 models. It wasn't easy, and while our solution might not be perfect, it's something we're proud of for now and can build upon in future. Another great takeaway was learning how to organize results from the models so that it gives a good ground for us to compare models and understand what worked best. In summary, it has been rewarding experience, and all the learnings from this project will help us in our future work.