Final Project: Machine Learning Classification for Cardiovascular Risk Factor Analysis and Detection

Juan Francisco Rodríguez

August 12, 2024

Contents

1. Introducción

In this final project, I will explore some risk factors for cardiovascular problems and apply some of the machine learning models studied in the course to predict or prevent heart failure. The dataset for this work was obtained from the *Kaggle* website and can be found here.

Let's briefly discuss the problem. Cardiovascular diseases (CVD) are the leading cause of death worldwide, accounting for about 31% of all deaths. Of these, 4 out of 5 deaths are due to heart attacks and strokes, and one-third of these deaths occur prematurely in people under 70 years of age. People with cardiovascular disease or who are at high cardiovascular risk need early detection.

This dataset, although small, contains some of the most important features that can be used to predict possible heart disease.

The Work Flow is very clear:

1. Make an EDA
2. Normalize and encode categorical variables
3. Apply some models using a cross-validation method to select the best parameters.
4. Compare the results using visualizations

This work (the pdf document, the notebook and the .csv datase) can be found in my github.

2. Exploratory Data Analysis

2.1. Data Oveview

Before applying any model, it is important to perform exploratory data analysis to understand the nature of the data and make better use of machine learning. The features of our dataset are:

- **Age**: age of the patient in years.
- **Sex**: sex of the patient: male (M) or female (F).
- **Chest Pain Type**: It can be: Typical Angina (TA), Atypical Angina (ATA), Non-Anginal Pain (NAP) and Asymptomatic (ASY).
- **Resting BP**: resting blood presure, in $mmHg$.
- **Cholesterol**: serum cholesterol in mm/dl.
- **Fasting BS**: fasting blood sugar. It is 1 if its over 120 mg/dl; and is 0 otherwise.
- **Resting ECG**: resting electrocardiogram results. It can be:
  - Normal.
  - ST: having ST-T wave abnormality.
  - LVH: showing probable or definite left ventricular hypertrophy by Estes's criteria.
- **Max HR**: maximum heart rate achieved. It a numeric value between 60 and 202.
- **Exercise Angina**: exercise-induced angina, yes (Y) or no (N).
- **Oldepeak**: Numeric value of the ST depression.
- **ST Slope**: the slope of the peak exercise ST segment. It can be:
  - Up: upsloping.
  - Flat.
  - Down: downsloping.
- **Heart Disease**: Output class: 1 if the patient has a heart disease, and 0 if the patient has a normal health.

After loading the data with Pandas, we can use the `.info()` method to easily see all features we have. The output is:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 918 entries, 0 to 917
Data columns (total 12 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Age     918 non-null    int64
 1   Sex     918 non-null    object
 2   ChestPainType   918 non-null    object
 3   RestingBP       918 non-null    int64
 4   Cholesterol     918 non-null    int64
 5   FastingBS       918 non-null    int64
 6   RestingECG      918 non-null    object
 7   MaxHR   918 non-null    int64
 8   ExerciseAngina  918 non-null    object
 9   Oldpeak    918 non-null    float64
 10  ST_Slope   918 non-null    object
 11  HeartDisease    918 non-null    int64
dtypes: float64(1), int64(6), object(5)
```

Here, `HeartDisease` and `FastingBS` have `int64` dtype, but they are actually a categorical variables. So we have 12 features, where the numerical ones are `Age`, `RestingBP`, `Cholesterol`, `MaxHR` and `Oldpeak`; and the categorical ones are `Sex`, `ChestPainType`, `FastingBS`, `RestingECG`, `ExerciseAngina`, `ST_Slope` and `HeartDisease`. It obvious that the target variable will be `HeartDisease`.

We note that we have 12 columns (features) and 918 rows (samples). Also, we don't have any `NaN` value, so the data is already clean.

Let's see how the data is distributed. For numerical data, we can use Histograms, while we can look at categorical data with piecharts. For this, we can use `matplotlib.pyplot` and `seaborn` libraries to plot a 4×3 grid, 1.

We can see that numerical features follow a normal type distribution, with a slight skewness in some cases. While categorical variables have majority categories, but they are not very unbalanced.

For the numerical features, we can also use the `.describe()` method to obtain a statistical summary:

	Age	**RestingBP**	**Cholesterol**	**MaxHR**	**Oldpeak**
count	918.0	918.0	918.0	918.0	918.0
mean	53.51089	132.39651	198.79956	136.80937	0.88736
std	9.43262	18.51415	109.38414	25.46033	1.06657
min	28.0	0.0	0.0	60.0	-2.6
25%	47.0	120.0	173.25	120.0	0.0
50%	54.0	130.0	223.0	138.0	0.6
75%	60.0	140.0	267.0	156.0	1.5
max	77.0	200.0	603.0	202.0	6.2

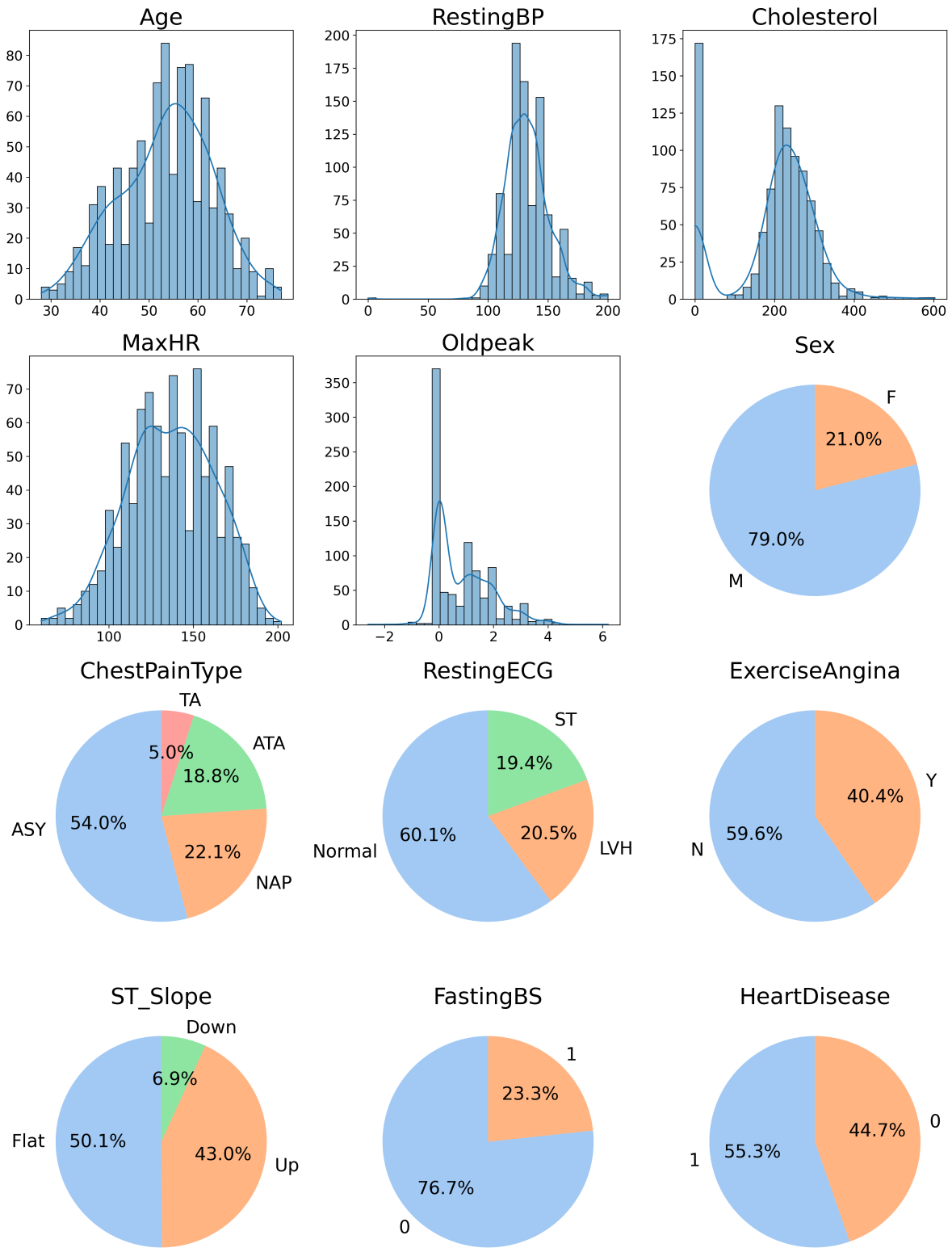

Figure 1: Features Distribution

2.2. Formating Data

Now, let's normalize the numerical data and encode the categorical features into numerical values. This is an important step to be able to compare different features.

Normalizing data is an important step in order to make comparisons and extract correlations between the features. Since most of the numerical values are positive, we can use the Minimum-Maximum scaler to scale the data into the $[0, 1]$ range. So we use the `MinMaxScaler` from the `preprocessing` module of `sklearn`.

In order to compare categorical features, is necesary to transform them into numerical values. The library `sklearn` provides many encoders based on different criteria. It's important to understand the nature of this categories.

Looking at the data, we can see that Sex and ExerciseAngina are binary and cannot be ordered, so we can use `OneHotEncoder`. The features ChestPainType and RestingECG have 4 and 3 categories respectively, but them can't be ordered, so it is not a good idea to encode them using numerical values $0, 1, 2, \ldots$; the best way to proceed is using the `OneHotEncoder` again, but it will create new features, one for each category. Finally, ST_Slope have more than 2 categories, but the can be sorted as Down $<$ Flat $<$ Up. So we can use an ordinal encoder for it, such as `OrdinalEncoder` from the `preprocessing` module. Giving the following DataFrame:

Age	**RestingBP**	$\cdots$	**RestingECG_ST**	**STSlope**	**HeartDisease**
0.2449	0.7	$\cdots$	0.0	2.0	0.0
0.4286	0.8	$\cdots$	0.0	1.0	1.0
0.1837	0.65	$\cdots$	1.0	2.0	0.0
0.4082	0.69	$\cdots$	0.0	1.0	1.0
0.5306	0.75	$\cdots$	0.0	2.0	0.0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

2.3. Correlations between features

We can calculate the correlations between the independent variables and show them using different plots. Also, we can calculate their correlation with the target variable. This can give us an overview of how well-chosen the features are and how prediction models might behave.

`pandas` provides the `.corr()` method to calculate the matrix directly from the DataFrame. We can plot the results into a heatmap graph, 2. Also, we can plot the distribution of the correlations, 3.

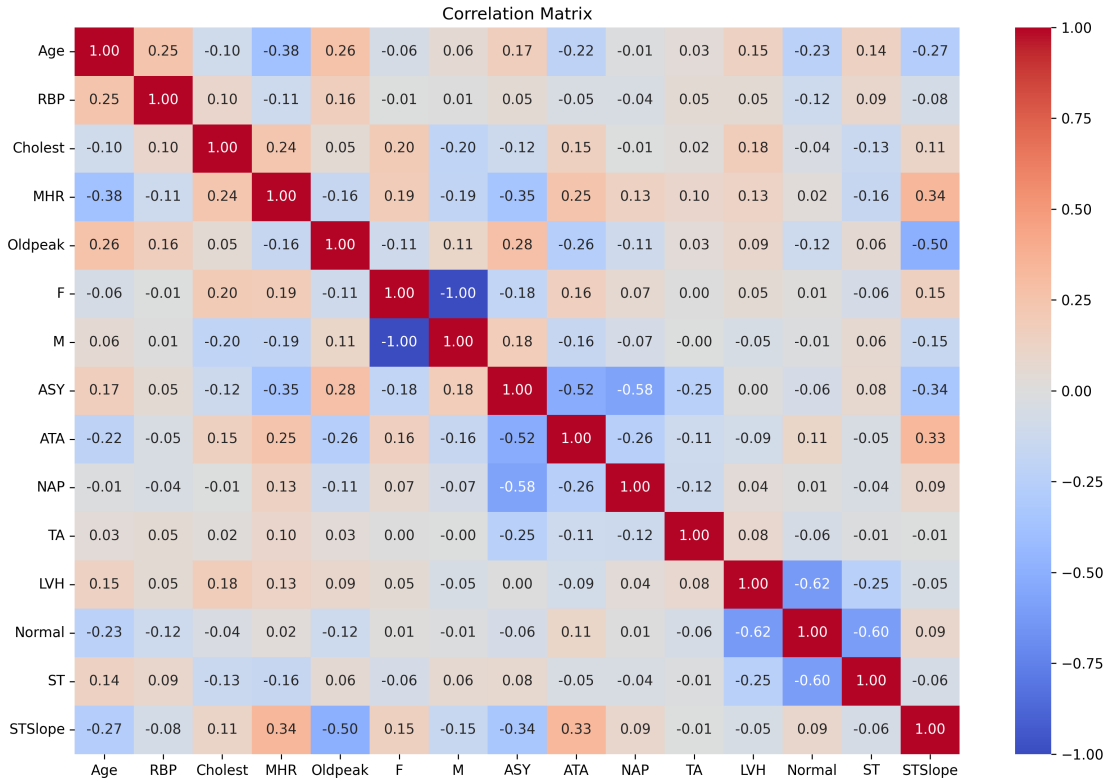

Figure 2: Correlation Heatmap

Looking at the heatmap, it's obvious that there is a low correlation between all features, which is a good sign because it means that our variables are, in fact, independent.

Also we can get the correlations between each feature and the target variable, 4.

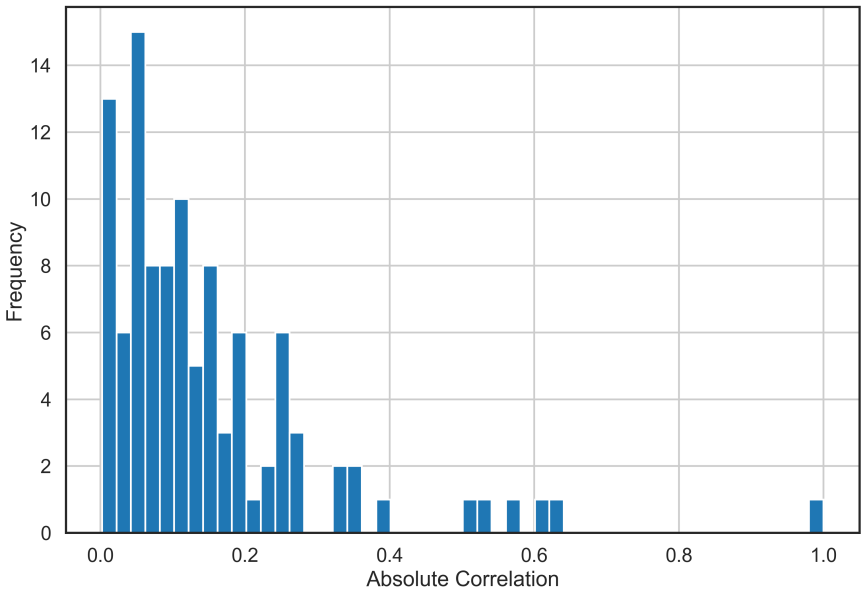

Figure 3: Correlation Distribution

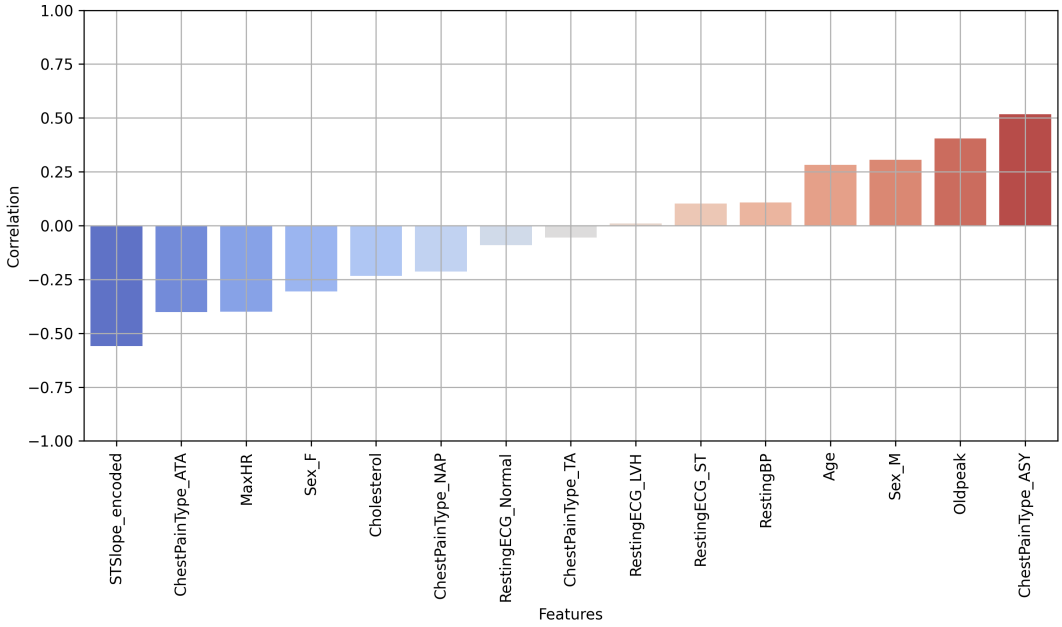

Figure 4: Correlation Distribution

2.4. Conclusion

To conclude this EDA, we can say that:

- The chosen dataset is very clean, with balanced data and no outliers.
- The differentiation of the categorial variables and the numerical features is very clear.
- The independent variables are very well-chose because they have a very low correlation between them, which shows a good variety of features.
- Although each features is slightly correlated to the target variable independentlyng, it does not mean that they cannot predict it.

Therefore, we are in a good position to apply some of the machine learning classification models we have seen throughout the course.

3. Applying Machine Learnig Models

3.1. How to proceed

Now, we are going to apply some machine learning models to our dataset. Specifically, we will apply the main basic models seen during the course: Logistic Regression, K Nearest Neighbors, Support Vector Machines and Decision Trees. We will apply the same process to each one of these model in order to compare them.

There are several ways to proceed when applying machine learning models. I will use a cross-validation method to find the best hyperparameters for each model. Concretely, I will use `gridsearchcv`, that do it automatically, combined with `StratifiedShuffleSplit`. After that, I can fit the model and get its scores.

With `StratifiedShuffleSplit` let's create an iterator that generates differents data splits that will be used in the cross-validation process to select the best parameters. With `train_test_split` let's create train and test splits that will be used to fit the final model with these best parameters.

The code for `KNeighborsClassifier`, for example, is the following:

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier()
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan'] }

# setting the grid search
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
                           cv=strat_shuf_split, scoring='f1')
# Fitting the model with the whole dataset
grid_search.fit(X, y)
# Params
best_params = grid_search.best_params_
# Final model with the best parameters
KNN = KNeighborsClassifier(**best_params)
# Fitting it with all the data
KNN.fit(X_train, y_train)
```

We need to apply the same process to `LogisticRegressin`, `SVC` and `DecisionTreeClassifier`, selecting the corresponding parameters in the `param_grid` dictionary.

3.2. Comparing Models

Having all models ready, we can compare the using different metrics.

Error Metrics

The main error metrics we know are Accuracy, Recall, Precision and F_1-Score. All of them are between 0 and 1, and near to 0 means a better score. The most important of them is F_1-Score, but all will be compared.

We can calculate this metrics using the following funciton:

```
def evaluate_metrics(model, X_test, y_test):
    yt = y_test
    yp = model.predict(X_test)
    results_pos = {}
    results_pos['Accuracy'] = accuracy_score(yt, yp)
    precision, recall, f_beta, _ =
                precision_recall_fscore_support(yt, yp, average='binary')
    results_pos['Recall'] = recall
    results_pos['Precision'] = precision
    results_pos['F1 Score'] = f_beta
    return results_pos
```

where `model` is the final model after cross-validation.

Finally, we get the following statistics:

	Accuracy	Recall	Precision	F1 Score
Logistic Regression	0.8652	0.9134	0.8529	0.8821
K Nearest Neighbors	0.9783	0.9843	0.9766	0.9804
Support Vector Machines	0.8609	0.9213	0.8417	0.8797
Decision Trees	0.8478	0.8268	0.8898	0.8571

We can plot it in some bar charts as in the Figure 5.

Confusion Matrix

Also we can plot the confusion matrix that shows the proportion of labels predicted correctly and incorrectly. The main diagonal contains the correct ones, so a higher proportion of the numbers here means a better behaviour of our model.

To obtain the confusion matrix, `sklearn` has the `confusion_matrix` built-in funciton and the `ConfusionMatrixDisplay` class for plotting. So we can do something like this:

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
models = [LR, KNN, SVM, DT]
model_names = ['Logistic Regression', 'K Nearest Neighbors',
               'Support Vector Machines', 'Decision Trees']

fig, axes = plt.subplots(2, 2, figsize=(8, 8))

# Iterate with each model
for model, name, ax in zip(models, model_names, axes.flatten()):
    y_pred = model.predict(X_test)
    conf_matrix = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
                                  display_labels=model.classes_)
    disp.plot(cmap=plt.cm.Blues, ax=ax, colorbar = False)
    ax.set_title(name, fontsize = 15)
    for labels in ax.texts:
        labels.set_fontsize(20)

plt.tight_layout(pad=2.0)
plt.show()
```

And finally, plot it in the Fig 6.

ROC Curves

Finally, another good way to compare some models is to plot the Receiver Operating Characteristic (ROC) curves, that shows the relationship between the true positive rate (sensitivity) and the false positive rate (1 - specificity) at different decision thresholds. The area under the curve (AUC) allows us to measure the quality of the model. The closer to 1, the better the model.

With `sklearn` we can use `roc_curve` and `aun` built-in functions to get the curves.

```
from sklearn.metrics import roc_curve, roc_auc_score, auc

X_test_df = pd.DataFrame(X_test, columns=feature_names)

plt.figure(figsize=(7, 7))

for model, name in zip(models, model_names):
    y_pred_proba = model.predict_proba(X_test_df)[:, 1]
    fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.4f})')
```

This gives the Fig 7 as result.

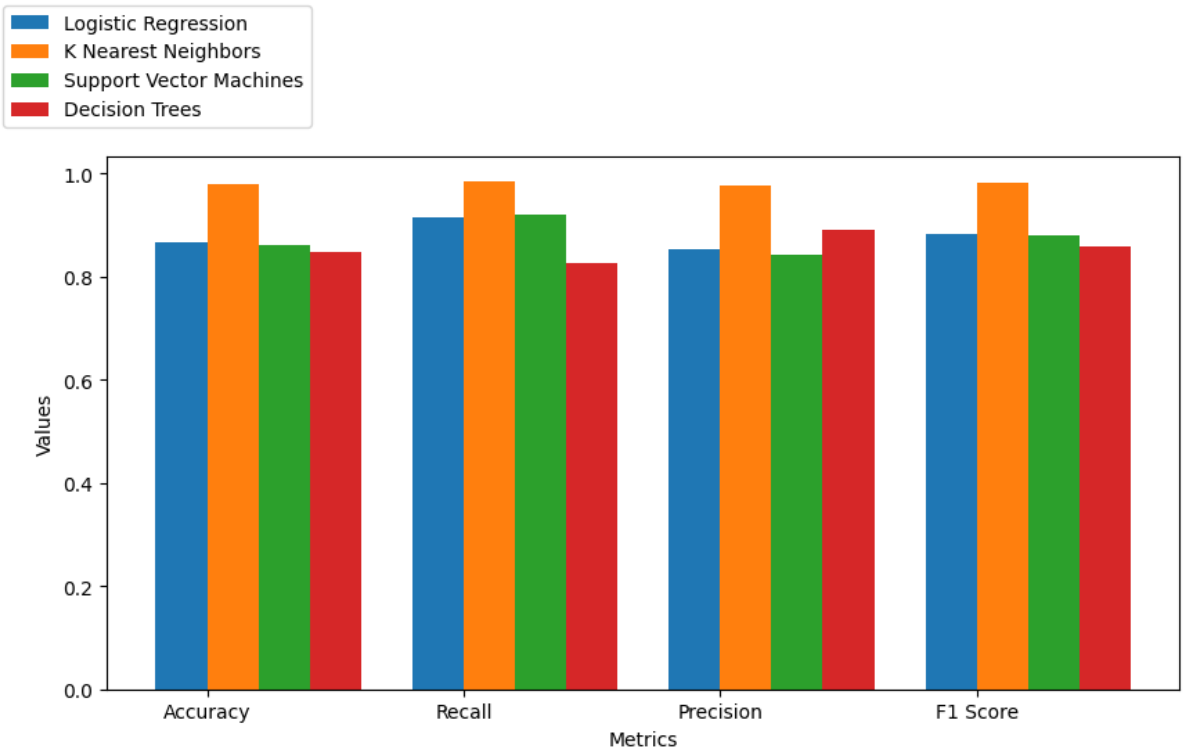

Figure 5: Error Metrics Comparison

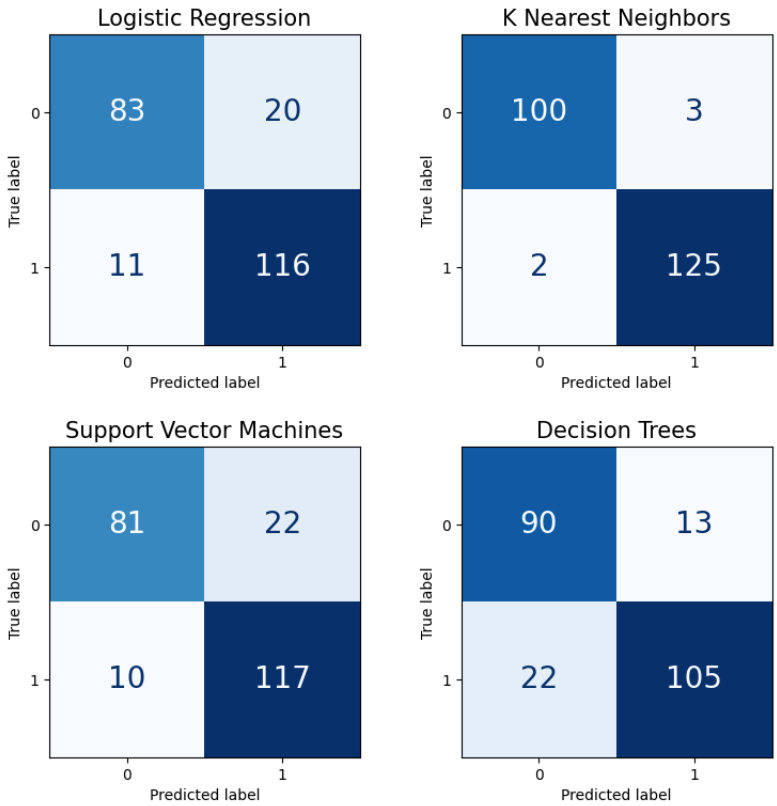

Figure 6: Confusion Matrix Comparison

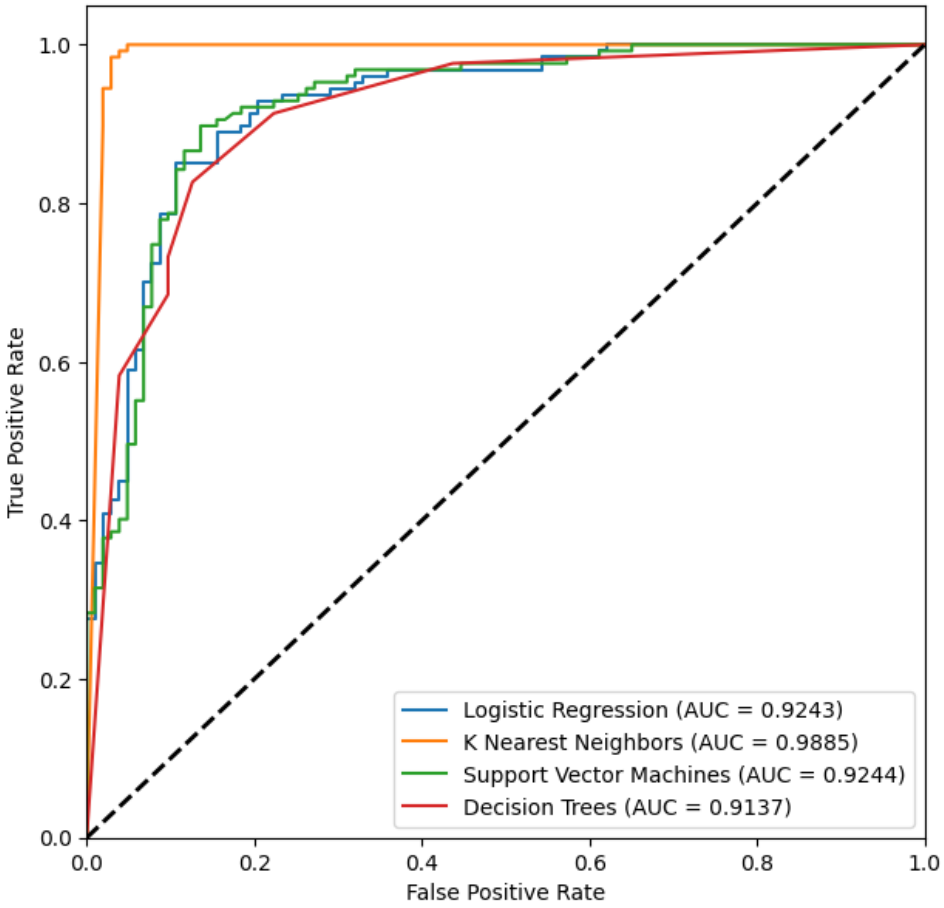

Figure 7: ROC Curves Comparison

3.3. Results

Looking at these three graphs, we can say that **KNN** is the one that has made the best predictions, while the other three models are very evenly matched.

- Looking at the ROC curves (Fig 7), we see that the area for KNN is very close to 1, for the others it is around 0.92.
- Regarding the confusion matrix (Fig 6), KNN is the best at predicting both positive and negative outcomes. Logistic Regression and SVM are practically the same; and comparing them with Decision Trees, we see that the first two are more successful at predicting negative outcomes, while Decision Trees is better at predicting positive outcomes.
- Looking at the error metrics (Fig 5). KNN is better in all of them, while the other models are around the same values.

4. How to continue with the Analysis

Other ways we could have proceed are:

- **Other type of cross-validation.** Here I decided to use `GridSearchCV` combined with `StratifiedShuffleSplit` to find the best parameters for differents models.

  Other cross-validation methods could have been selected.. For example, we could use the `cross_val_score` built-in function to do a manual hyperparameter search.

  Or simply setting a training/test set for all models and comparing their performance. It is something much more basic but it reduces the cost of execution and would have allowed more models to be applied, for example.

- **Othe models.** Some kind of models like Decision Trees or Logistic Regresion does not have more than one or two clases to perform the method, `LogisticRegression` and `LogisticRegressionCV` for Logistic Regression and `DecisionTreeClassifier` for Decision Trees.

  But other kind of models has much more built-in classes that we could have applied. For exampe, for SVM I chose `SVC`, but also exists `LinearSVC` and `NuSVC` for classification.

- **Othe hyperparameters**. In the cross-validation process, I set the `best_params` dictionary what I thought were the most important parameters. For example, for the `KNeighborsClassifier` class, I only try with Euclidean and Manhattan distances. We could have tried also with a Minkowski distance for $p > 2$.

To continue with this Analysis, could be interesting to try with ensemble models such as Bagging, and compare the results with the basic models already implemented in order to see if there is a substantial improvement.