Sklearn exercices

Gonzalo Mesas Aranda

December 29, 2023

1 Introduction

In the following document, different models and tools from the *sklearn* library will be used in order to obtain performance measures over a clinical data set.

The different models that will be used for the classification problem will be:

- Naive bayes
- Decision tree
- K-nearest neighbors (KNN)
- Support vector machine (SVM)

Along the document, all of them will be discussed and explained. The code and implementation will be provided as well as the code of all the figures used. Some parts have been omitted for not being relevant for the purpose of the report, although if the whole code wants to be checked, it can be found in my GitHub repository 8.

Dataset

About the data set, it is a real data set obtained from Kaggle. It is about heart disease and different parameters about anonymous patients are provided. The target class is binary and tells if the patient has or not a heart condition. No further explanations will be given since it was already explained in other documents from the subject.

2 Data set pre-processing

Before starting with the models, it is important to have a data set with quality data and it is a good practice to pre-process it to avoid wrong data and adapt it so that the machine learnign models can get the most out of it.

In the chosen data set, there are 13 features and 1 target binary class, being all of them numerical. A simple adjustment would be to ensure that there is **no NAN values**, by eliminating the rows that contain at least one.

In addition, taking into account that later PCA will be applied and some models also take advantage of it, **Z-Score standardization** is applied to the whole data set, except for the target class. This adjusts the data in order for each column to have a mean of 0 and a standard deviation of 1.

```
# Load the data set
df_original = pd.read_csv("heart.csv")
# Remove NAN values
df_original.dropna(inplace = True)
df_original.reset_index(inplace = True, drop = True)
# Z-score
std_scaler = StandardScaler().fit(df_original.iloc[:, :-1])
df_std = std_scaler.transform(df_original.iloc[:, :-1])
df = pd.DataFrame(df_std, columns=df_original.columns[:-1])
df['target'] = df_original['target']
```

Listing 1: Code for the NAN values removal and Z-Score.

A reduced version of the data set is represented

age	sex	cp	thalach	exang	oldpeak	slope	ca	thal	target
-0.27	0.66	-0.92	0.82	-0.71	-0.06	1.00	1.21	1.09	0
-0.16	0.66	-0.92	0.26	1.40	1.73	-2.24	-0.73	1.09	0
1.72	0.66	-0.92	-1.05	1.40	1.30	-2.24	-0.73	1.09	0
0.72	0.66	-0.92	0.52	-0.71	-0.91	1.00	0.24	1.09	0
0.83	-1.51	-0.92	-1.87	-0.71	0.71	-0.62	2.18	-0.52	0
0.39	-1.51	-0.92	-1.18	-0.71	-0.06	-0.62	-0.73	-0.52	1
0.39	0.66	-0.92	-0.40	-0.71	2.83	-2.24	2.18	-2.13	0
0.06	0.66	-0.92	-0.18	1.40	-0.23	-0.62	0.24	1.09	0

Table 1: Reduced version of the data set before applying PCA.

PCA

Even though it is not required and maybe not very useful for this particular case, PCA will be applied. The reason is that the finality of the document and the exercises is to give a first approach to ML programming, so the more techniques are used and practiced, the more knowledge and expertise this project will give to the author.

PCA's primary goal is to transform the original features of a data set into a new set of uncorrelated variables called principal components. These components capture the maximum variance in the data, allowing to represent the information with fewer dimensions. The understanding of PCA can be complex, so no deeper information will be provided.

After obtaining the result, the number of principal components needs to be selected. The first components are the most important, adding the last ones the least information. The idea is to find a balance between fewer components, and therefore lower dimension, or more components, which add more information and the approximates the original data set better.

```
# Appply PCA
pca = PCA()
pca.fit(df)
Xpca = pca.transform(df)
# Visualize the graph
explained_var = np.cumsum(np.round(pca.explained_variance_ratio_,
decimals=3) * 100)

plt.figure(figsize=(8,8))
plt.plot(explained_var)
plt.scatter(range(len(explained_var)), explained_var, color='r')
plt.xlabel('Num. of Principal Components')
plt.ylabel('Explained variance (%)')
plt.xticks(range(0, 20, 5))
plt.show()
```

Listing 2: PCA application to the data set.

In the following graph, a representation of the explained variance is shown as a function of the number of PC. The decision taken was 12 out of the 14 PC, which leads to an explained variance of 96.3%.

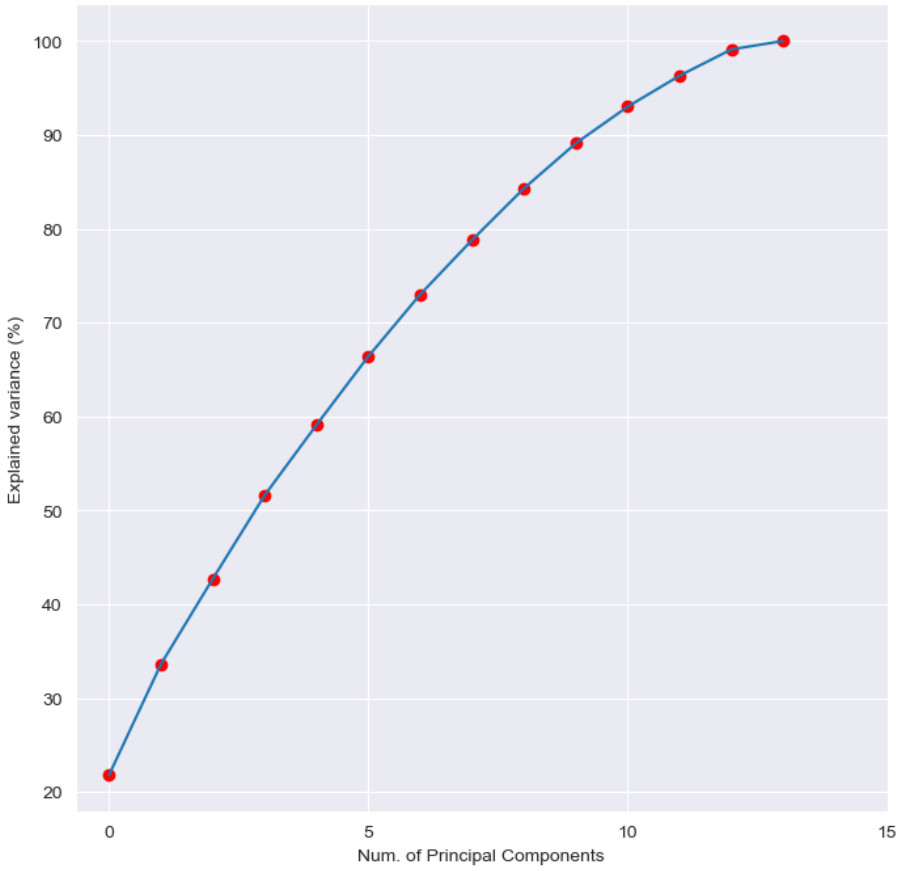


Figure 1: Explained variance for PC analysis.

Dataset segmentation

Now that the pre-processing is done, a few concepts should be clarified in order to understand the methodology in the rest of the document.

The data set will be divided in two, a training set (67%) and a test set (33%). The final metrics will be obtained by training the model with the training set and measuring it with the test set. The adjustment of the hyperparameters though, will be all done with the training set, where a 10-fold cross validation will be applied.

```
from sklearn.model_selection import train_test_split
X_total = pd.DataFrame(Xpca[:, :12])
Y_total = df_original['target']

X_train, X_test, y_train, y_test = train_test_split(X_total,
Y_total, test_size=0.33, random_state=13)
```

```
df_test = pd.DataFrame(columns=X_test.columns, data=X_test)
df_test['target'] = y_test

df_train = pd.DataFrame(columns=X_train.columns, data=X_train)
df_train['target'] = y_train

df_train.to_csv("df_train.csv", index = False)
df_test.to_csv("df_test.csv", index = False)
```

Listing 3: Dataset segmentation.

3 Naive Bayes

In the Naive Bayes model, there are no specified parameters in the constructor of the class. Despite that, the threshold will be adjusted by using an inderect method.

The idea is creating a dataframe where all the data from the metrics is stored and later on used to obtain the best threshold. Different thresholds are used and their performance is measured by obtaining the mean of the metrics for each fold.

The "indirect method" used is $predict_proba$, which returns the probabilities of each label instead of the predicted label. This allows the control over the threshold.

```
from sklearn.naive_bayes import GaussianNB
training_df = pd.read_csv("df_train.csv")
X, Y = training_df.drop(["target"], axis = 1), training_df["target"]

metrics_mean = pd.DataFrame(columns = ["Threshold", "Accuracy",
"F1-Score"])
kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=13)

j = 0
for threshold in [0.5, 0.55, 0.6, 0.65, 0.7]:
    metrics = pd.DataFrame(columns = ["Fold", "Accuracy",
    "F1-Score"])
    i = 1
    for train, validation in kfolds.split(X, Y):
        naive_bayes_model = GaussianNB()
        naive_bayes_model.fit(X.iloc[train], Y.iloc[train])
```

```
        y_pred = (naive_bayes_model.predict_proba(X.iloc[validation])[:, 1]
        >= threshold).astype(int)

        metrics.loc[i-1] = {
            "Fold": i,
            "Accuracy": accuracy_score(Y.iloc[validation], y_pred),
            "F1-Score": f1_score(Y.iloc[validation], y_pred)}
        i += 1

    metrics_mean.loc[j] = {
        "Threshold": threshold,
        "Accuracy": metrics["Accuracy"].mean(),
        "F1-Score": metrics["F1-Score"].mean()}
    j += 1
```

Listing 4: Threshold adjustment in Naive Bayes.

The results obtained are:

Threshold	Accuracy	F1-Score
0.50	0.8500	0.8615
0.55	0.8587	0.8675
0.60	0.8586	0.8637
0.65	0.8426	0.8444
0.70	0.8412	0.8363

Table 2: Threshold adjustment.

As can be seen, the best threshold seems to be **0.55**.

Now, with this threshold, the model is again trained with the training set mentioned before and its performance is measured according to the test set.

```
test_df = pd.read_csv("df_test.csv")
X_Test, Y_Test = test_df.drop(["target"], axis = 1), test_df["target"]
training_df = pd.read_csv("df_train.csv")
X, Y = training_df.drop(["target"], axis=1), training_df["target"]
```

```
naive_bayes_model = GaussianNB()
naive_bayes_model.fit(X, Y)
y_pred = (naive_bayes_model.predict_proba(X_Test)[:, 1] >= 0.55).astype(int)

cm = confusion_matrix(Y_Test, y_pred)

print("Accuracy:", accuracy_score(Y_Test, y_pred))
print("F1-score:", f1_score(Y_Test, y_pred))
print("Precision:", precision_score(Y_Test, y_pred))
print("Recall:", recall_score(Y_Test, y_pred))
plot_confusion_matrix(cm)
```

Listing 5: Code to measure the final performance of Naive Bayes.

Some measures for this model are:

Metric	Value
Accuracy	0.8289
F1-score	0.8371
Precision	0.8098
Recall	0.8663

Table 3: Final performance of Naive Bayes.

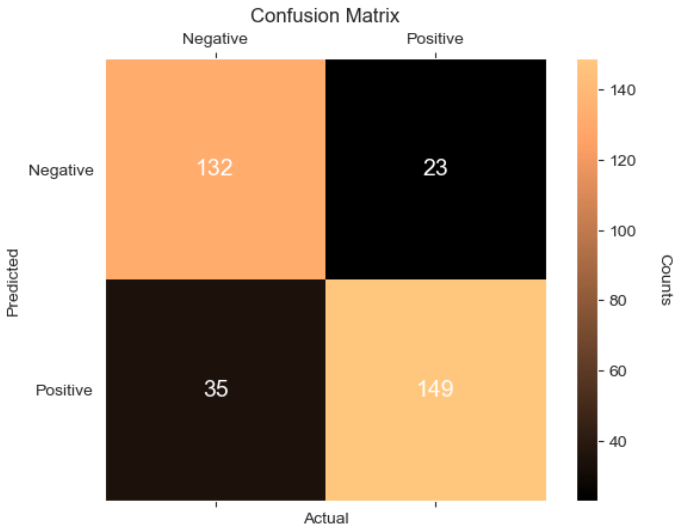


Figure 2: Confusion matrix for Naive Bayes.

4 Decision Tree

In the decision tree classifier there are many parameters. Since it would add too much complexity to the document and many of the parameters were not studied in the theory, only two will be adjusted. The methodology to adjust more hyperparameters is the same as the one that will be shown, so if the following is understood, for the rest no additional knowledge is needed.

The two parameters to be adjusted are:

- **Criterion:** it is the function used to measure the quality of a split.
- **Maximum depth:** maximum depth permitted of the tree.

The idea is the same as for the Naive Bayes, measures will be stored in a dataframe for posterior comparison. Additionally, graphs that show the performance for each criteria are shown.

```
from sklearn.tree import DecisionTreeClassifier
training_df = pd.read_csv("df_train.csv")
X, Y = training_df.drop(["target"], axis=1), training_df["target"]

metrics_mean = pd.DataFrame(columns=["Criterion", "Max_depth",
"Accuracy", "F1-Score"])
kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=13)

j = 0
for criterion in ['gini', 'entropy', 'log_loss']:
    metrics = pd.DataFrame(columns=["Fold", "Accuracy", "F1-Score"])
    i = 1
    fig, ax = plt.subplots(figsize=(10,6))
    accuracies = []
    f1_scores = []
    for depth in np.arange(1, 20).tolist():
        acc = []
        f1 = []
        for train, validation in kfolds.split(X, Y):
            decision_tree_model = DecisionTreeClassifier(criterion=criterion,
            max_depth=depth)
            decision_tree_model.fit(X.iloc[train], Y.iloc[train])
```

```
            y_pred = decision_tree_model.predict(X.iloc[validation])

            acc.append(accuracy_score(Y.iloc[validation], y_pred))
            f1.append(f1_score(Y.iloc[validation], y_pred))

            metrics.loc[i - 1] = {
                "Fold": i,
                "Accuracy": accuracy_score(Y.iloc[validation], y_pred),
                "F1-Score": f1_score(Y.iloc[validation], y_pred)}
            i += 1

        metrics_mean.loc[j] = {
            "Criterion": criterion,
            "Max_depth": depth,
            "Accuracy": metrics["Accuracy"].mean(),
            "F1-Score": metrics["F1-Score"].mean()}
        j += 1
        accuracies.append(np.mean(acc))
        f1_scores.append(np.mean(f1))

    ax.plot(range(1,20), accuracies, color = "cyan")
    ax.plot(range(1,20), f1_scores, color = "green")
    ax.set_xlabel('Depth')
    ax.set_ylabel("Metric's value")
    ax.set_title(f'Metrics using {criterion} criterion')
    ax.legend(['F1-Score', 'Accuracy'])
```

Listing 6: Training of the decision tree.

Now the the performance, accuracy and f1-measure, will be visualized by plotting score as a function of the maximum depth parameter, one for each criteria used.

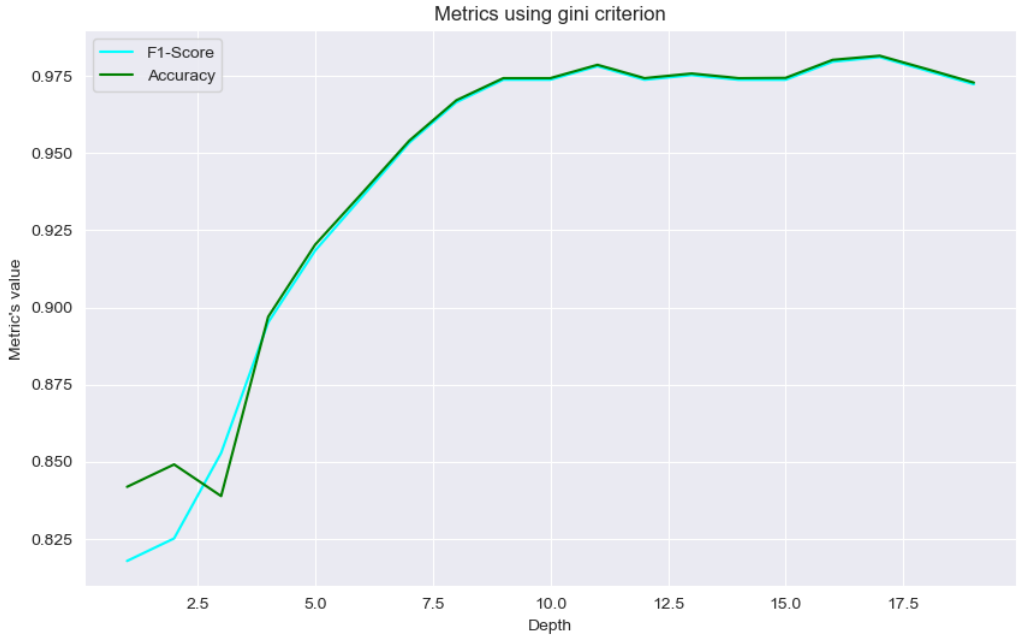


Figure 3: Performance for the gini criteria.

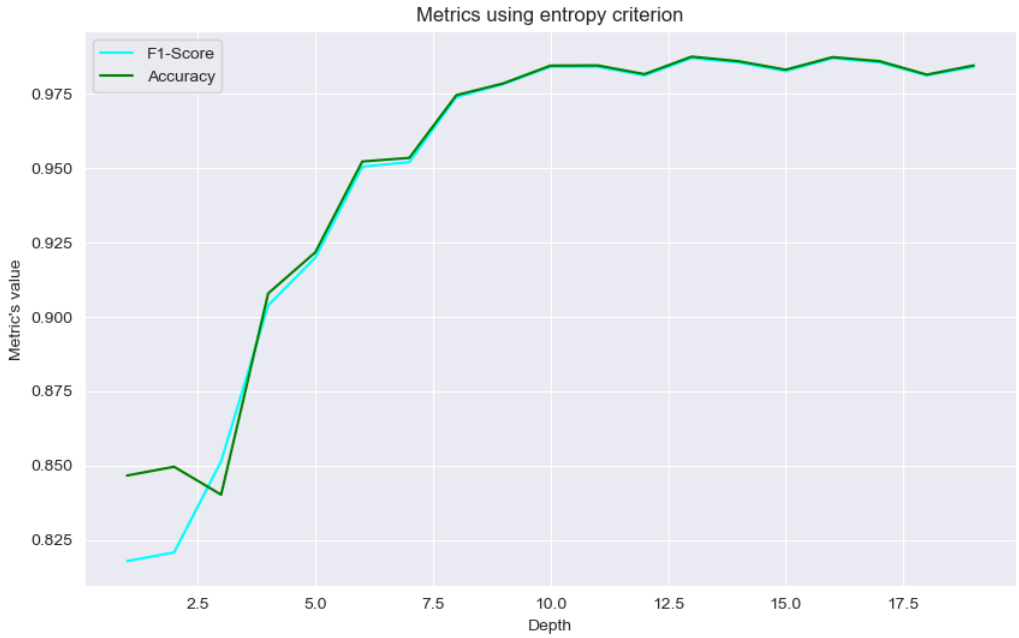


Figure 4: Performance for the entropy criteria.

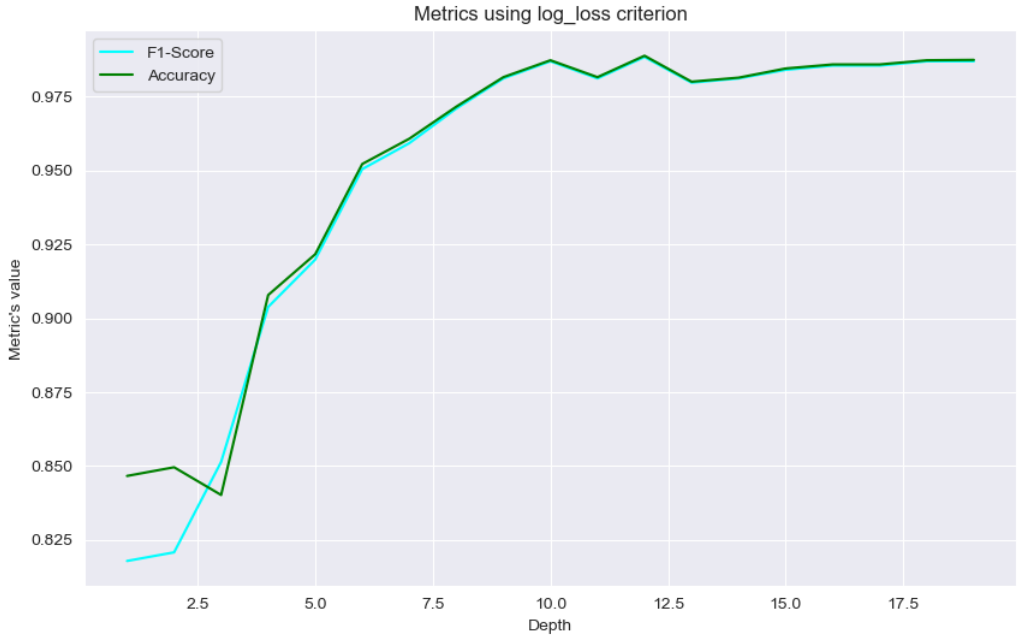


Figure 5: Performance for the log_loss criteria.

Looking up the dataframe with the performace measures, both the accuracy and f1-measure are optimal for the log_loss criteria and a maximum depth of 19.

Therefore, the model is trained using this hyperparameters, using the sets mentioned before. The final performance scores for the decision tree classifier are:

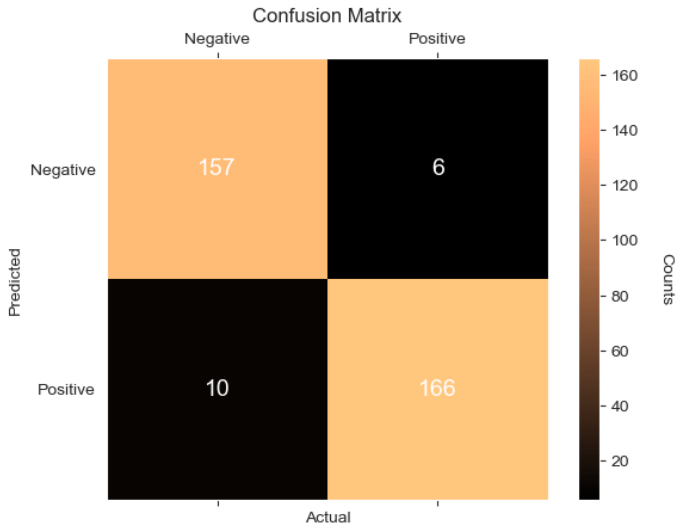


Figure 6: Confusion matrix for Decision Tree Classifier.

Measure	Value
Accuracy	0.9528
F1-score	0.9540
Precision	0.9432
Recall	0.9651

Table 4: Final performance measure for the decision tree.

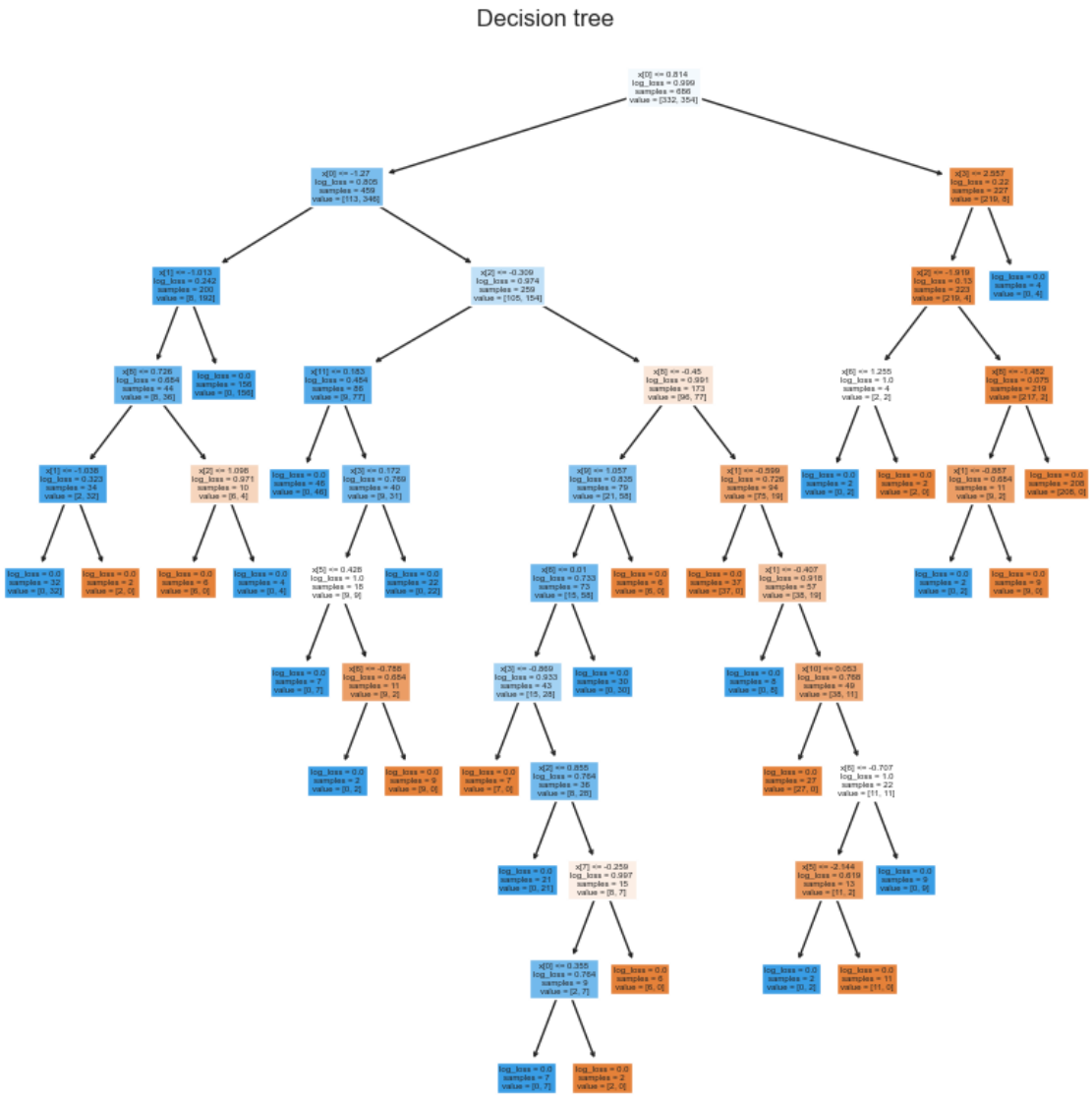


Figure 7: Decision tree of the final model.

5 Nearest Neighbors

In the nearest neighbors classifier, the only hyperparameter that needs to be adjusted is the number of neighbors that are considered in order to classify. Since the target class is binary, an odd number will be chosen to avoid ties.

The same procedure as in the other models is followed, so no explanation is given to avoid repetition.

```
from sklearn.neighbors import KNeighborsClassifier
training_df = pd.read_csv("df_train.csv")
X, Y = training_df.drop(["target"], axis=1), training_df["target"]
metrics_mean = pd.DataFrame(columns=["K", "Accuracy", "F1-Score"])
kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=13)

_, ax = plt.subplots(figsize=(10,6))
j = 0
accuracies = []
f1_scores = []

for k in np.arange(1,21,2).tolist():
    metrics = pd.DataFrame(columns=["Fold", "Accuracy", "F1-Score"])
    acc = []
    f1 = []
    i = 1
    for train, validation in kfolds.split(X, Y):
        knn_model = KNeighborsClassifier(n_neighbors=k)
        knn_model.fit(X.iloc[train], Y.iloc[train])

        y_pred = knn_model.predict(X.iloc[validation])

        acc.append(accuracy_score(Y.iloc[validation], y_pred))
        f1.append(f1_score(Y.iloc[validation], y_pred))

        metrics.loc[i - 1] = {
            "Fold": i,
            "Accuracy": accuracy_score(Y.iloc[validation], y_pred),
            "F1-Score": f1_score(Y.iloc[validation], y_pred)}
        i += 1
```

```
        metrics_mean.loc[j] = {
            "K": k,
            "Accuracy": metrics["Accuracy"].mean(),
            "F1-Score": metrics["F1-Score"].mean()}
        j += 1
        accuracies.append(np.mean(acc))
        f1_scores.append(np.mean(f1))

ax.plot(range(1,21,2), accuracies, color = "cyan")
ax.plot(range(1,21,2), f1_scores, color = "green")
ax.set_xlabel('Number of neighbours K')
ax.set_ylabel("Metric's value")
ax.set_title("Metrics for KNN")
ax.legend(['F1-Score', 'Accuracy'])
ax.set_xticks(np.arange(1,21,2))
plt.show()
```

Listing 7: Training of the KNN.

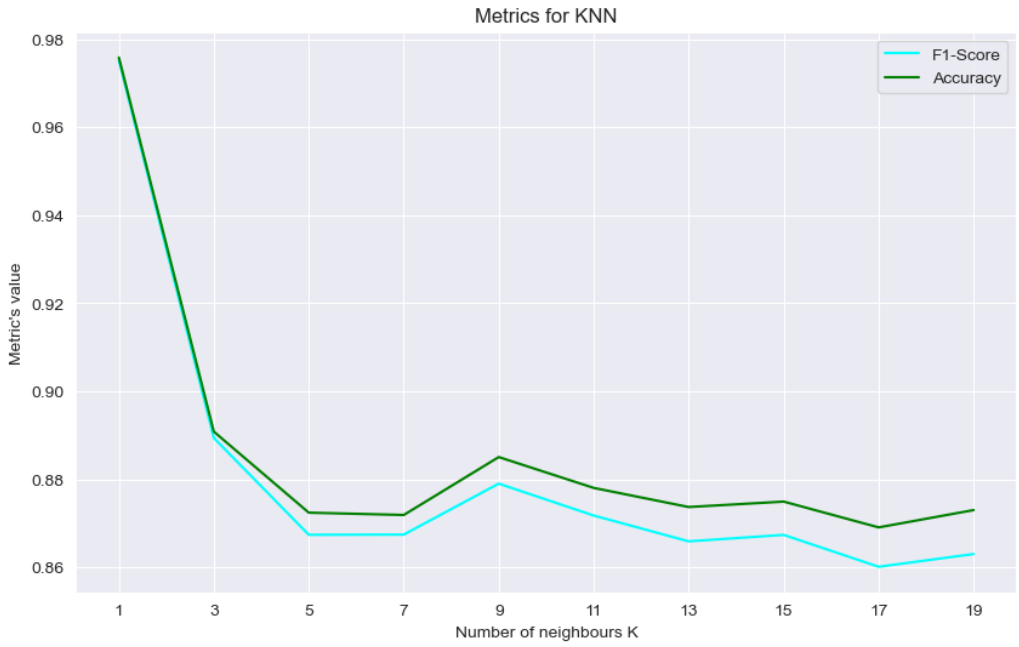


Figure 8: Scores as a function of the number of neighbors.

In the previous graph, trivially the best result is produced when only one neighbor is selected. This could mislead to an overfitting, which sometimes occurs when a very small number of neighbors is chosen.

In awareness of this issue, the final performance metrics were computed for the first three odd numbers. Still, 1 neighbor was the best decision. The overfitting did not seem to happen in this case, maybe because of the 10-fold cross validation, which one of its advantages is the overfitting avoidance.

Measure	**Value**
Accuracy	0.9617
F1-score	0.9630
Precision	0.9441
Recall	0.9826

Table 5: Final performance of the KNN with 1 neighbor

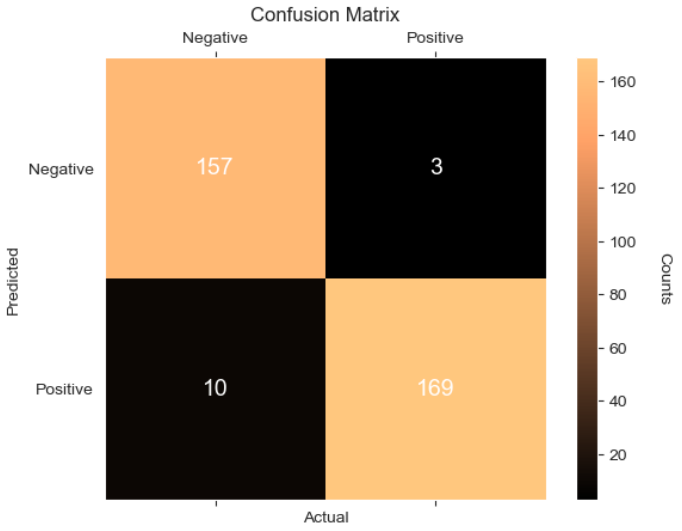


Figure 9: Confusion matrix for the final KNN model.

6 Support Vector Machine

In the support vector machine (SVM), there are again plenty of hyperparameters. Therefore, the following ones were chosen for adjustment:

- **Kernel:** specifies the kernel type to be used in the algorithm.
- **C:** it is a regularization parameter, where the strength of the regularization is inversely proportional to C.
- **Degree:** degree of the polynomial kernel function 'poly'.

The methodology is the same, but there is a difference. Since the degree parameter only makes a difference when the kernel is *poly*, this kernel is left aside from the main loop and checked later, including the degree. With this small modification, the code looks like this:

```
from sklearn.svm import SVC
training_df = pd.read_csv("df_train.csv")
X, Y = training_df.drop(["target"], axis=1), training_df["target"]
metrics_mean = pd.DataFrame(columns=["C", "Kernel", "Degree", "Accuracy",
"F1-Score"])
kfolds = StratifiedKFold(n_splits=10, shuffle=True, random_state=13)

j = 0
accuracies = []
f1_scores = []
C = [0.1, 1, 10, 100]
Kernel = ['linear', 'rbf', 'sigmoid']

for kernel in Kernel:
    for c in C:
        metrics = pd.DataFrame(columns=["Fold", "Accuracy", "F1-Score"])
        acc = []
        f1 = []
        i = 1
        for train, validation in kfolds.split(X, Y):
            svc_model = SVC(C=c, kernel=kernel)
            svc_model.fit(X.iloc[train], Y.iloc[train])

            y_pred = svc_model.predict(X.iloc[validation])
```

```
            acc.append(accuracy_score(Y.iloc[validation], y_pred))
            f1.append(f1_score(Y.iloc[validation], y_pred))

            metrics.loc[i - 1] = {
                "Fold": i,
                "Accuracy": accuracy_score(Y.iloc[validation], y_pred),
                "F1-Score": f1_score(Y.iloc[validation], y_pred)}
            i += 1

        metrics_mean.loc[j] = {
            "C": c,
            "Kernel": kernel,
            "Degree": -1,
            "Accuracy": metrics["Accuracy"].mean(),
            "F1-Score": metrics["F1-Score"].mean()}
        j += 1
        accuracies.append(np.mean(acc))
        f1_scores.append(np.mean(f1))

kernel = 'poly'
for c in C:
    for degree in [2,3,4,5,6,7,8]:
        metrics = pd.DataFrame(columns=["Fold", "Accuracy", "F1-Score"])
        acc = []
        f1 = []
        i = 1
        for train, validation in kfolds.split(X, Y):
            svc_model = SVC(C=c, kernel=kernel, degree=degree)
            svc_model.fit(X.iloc[train], Y.iloc[train])

            y_pred = svc_model.predict(X.iloc[validation])

            acc.append(accuracy_score(Y.iloc[validation], y_pred))
            f1.append(f1_score(Y.iloc[validation], y_pred))

            metrics.loc[i - 1] = {
                "Fold": i,
                "Accuracy": accuracy_score(Y.iloc[validation], y_pred),
                "F1-Score": f1_score(Y.iloc[validation], y_pred)}
            i += 1
```

```
metrics_mean.loc[j] = {
    "C": c,
    "Kernel": kernel,
    "Degree": degree,
    "Accuracy": metrics["Accuracy"].mean(),
    "F1-Score": metrics["F1-Score"].mean()}
j += 1
accuracies.append(np.mean(acc))
f1_scores.append(np.mean(f1))
```

Listing 8: PCA application to the data set.

From all of this models, the one that had the best performance, both in accuracy and f1-measure, was the one for C = 100, kernel = 'ploy' and degree = 5.

Again, the model is trained with the sets mentioned in the beginning of the document and the final performance is:

Measure	**Value**
Accuracy	0.9617
F1-score	0.9623
Precision	0.9595
Recall	0.9651

Table 6: Final performance for the SVM.

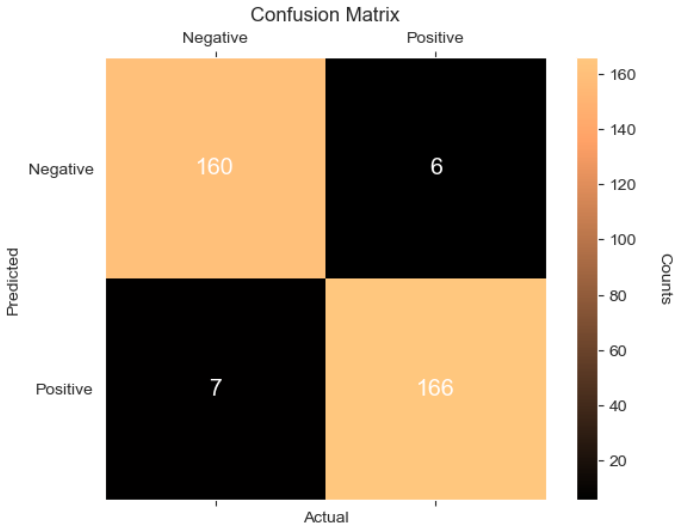
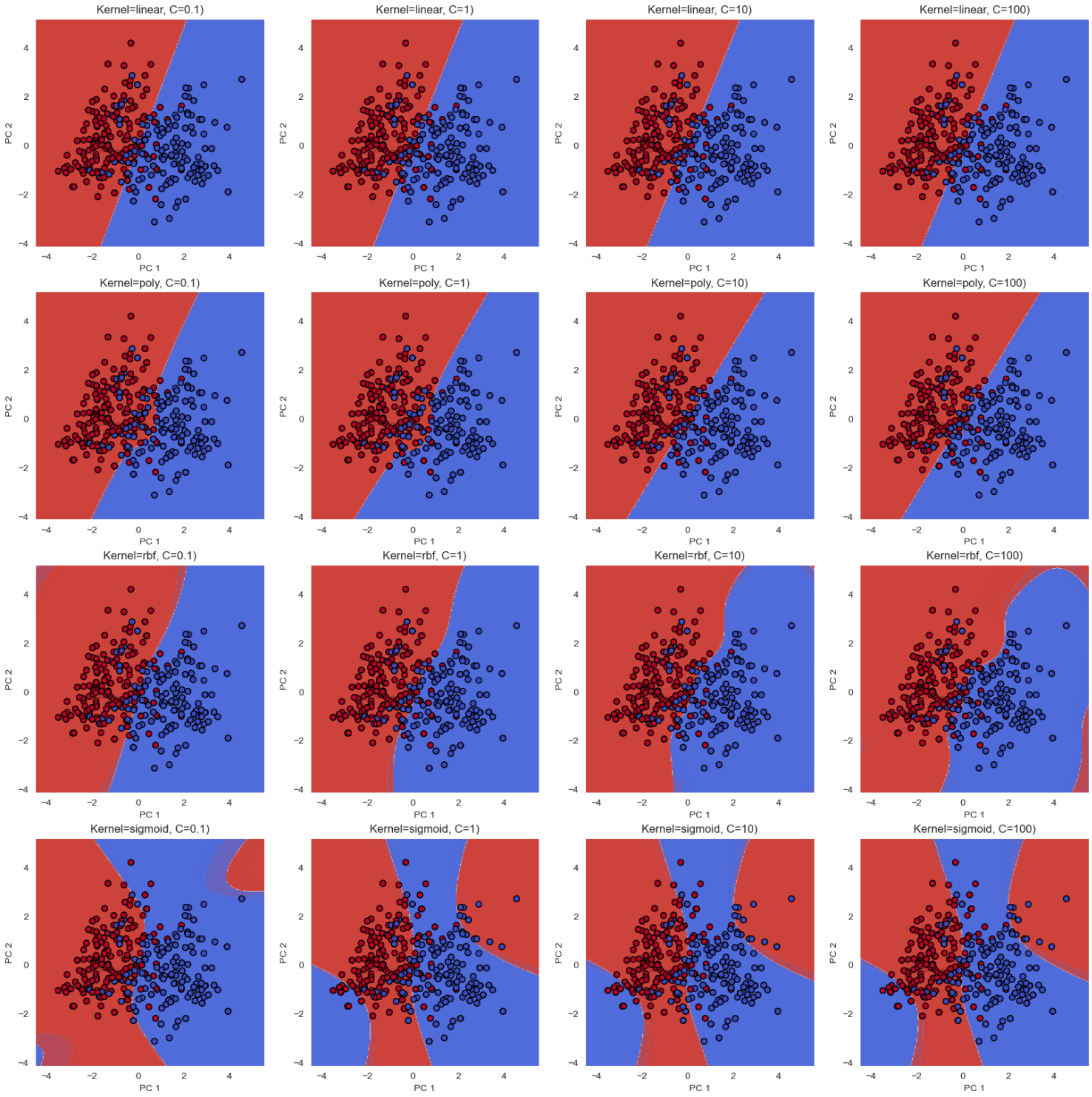


Figure 10: Confusion matrix for the SVM final model.


Kernel=linear, C=0.1)
Kernel=linear, C=1)
Kernel=linear, C=10)
Kernel=linear, C=100)
Kernel=poly, C=0.1)
Kernel=poly, C=1)
Kernel=poly, C=10)
Kernel=poly, C=100)
Kernel=rbf, C=0.1)
Kernel=rbf, C=1)
Kernel=rbf, C=10)
Kernel=rbf, C=100)
Kernel=sigmoid, C=0.1)
Kernel=sigmoid, C=1)
Kernel=sigmoid, C=10)
Kernel=sigmoid, C=100)
PC 1
PC 2

In the previous page, a visualization of different decision boundaries for different models is shown. This is just an approximation since the real space has 12 dimensions. The first two principal components were chosen for the representation for being the most important ones.

The plot is interesting since it helps to understand what the variation of the kernels and the C parameter actually does.

7 Comparison

In this final part, a comparison of the models will be given and discussed. First, a summary table with the performance measure is provided.

Model	**Metric**	**Value**
Naive Bayes	Accuracy	0.8289
Naive Bayes	F1-score	0.8371
Naive Bayes	Precision	0.8098
Naive Bayes	Recall	0.8663
Decision Tree	Accuracy	0.9528
Decision Tree	F1-score	0.9540
Decision Tree	Precision	0.9432
Decision Tree	Recall	0.9651
KNN	Accuracy	0.9617
KNN	F1-score	0.9630
KNN	Precision	0.9441
KNN	Recall	0.9826
SVM	Accuracy	0.9617
SVM	F1-score	0.9623
SVM	Precision	0.9595
SVM	Recall	0.9651

Table 7: Final performance of all the models.

According to the measurements, the Decision tree, KNN and SVM got very similar results, with a very high performance. For the Naive bayes, the performance is reasonably well, although not as good as the rest.

Before giving the final conclusion, the ROC curve for the different models is shown.

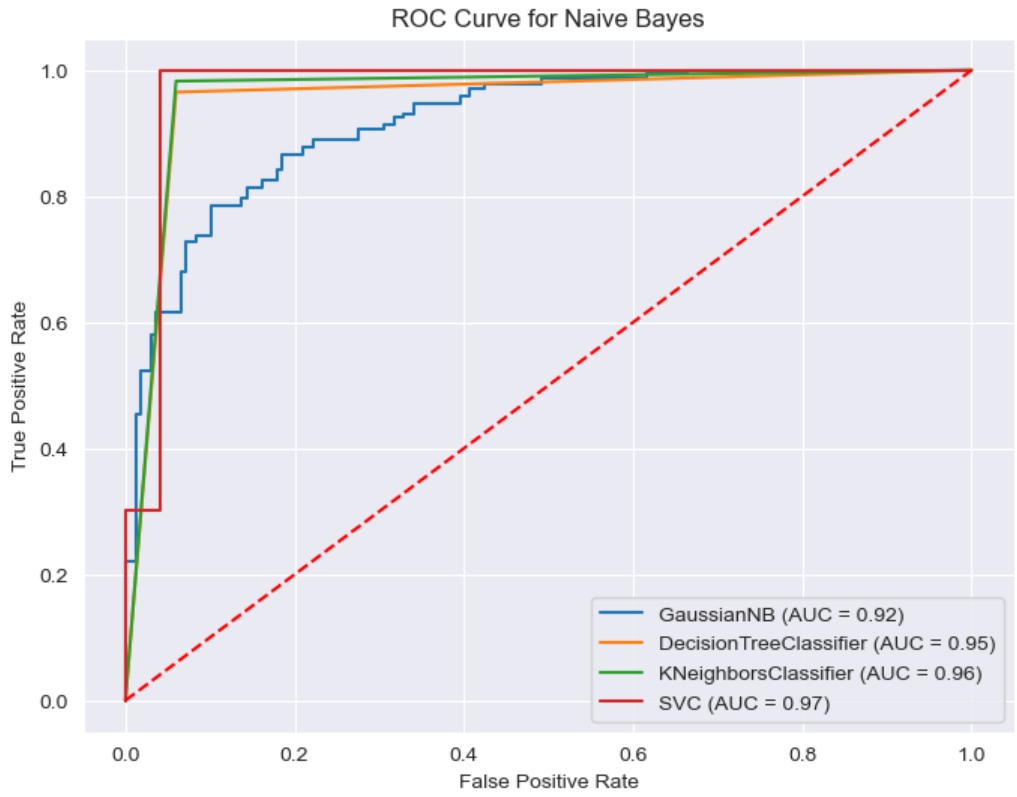


Figure 11: ROC curve for the different models.

The low resolution of the Decision tree, KNN and SVM curves is due to a small number of thresholds used for the plotting.

In the ROC, the true positive rate versus the false positive rate is represented, being a model better the greater the area under the curve (AUC). This can be understood as a representation of many confusion matrices, where the threshold is varied, for each model. The optimal point is the one that makes the false positive rate zero and the true positive rate 1.

Taking this into account, the greater the AUC, the more "stable" or the better proportion between TPR and FPR there is with different thresholds.

Once again, the Decision tree, KNN and SVM got a very good and similar result, leaving the naive bayes a little bit in the back.

To conclude, even though there were better models than others, which is as well a very important factor, the final decision of which model to use depends on the specific requirements and characteristics of the problem being solved. For example the computational efficiency could be critical, which is often a drawback for some models in the medicine field.

8 License and repository.

The full code can be found in my GitHub repository.

License: MIT License