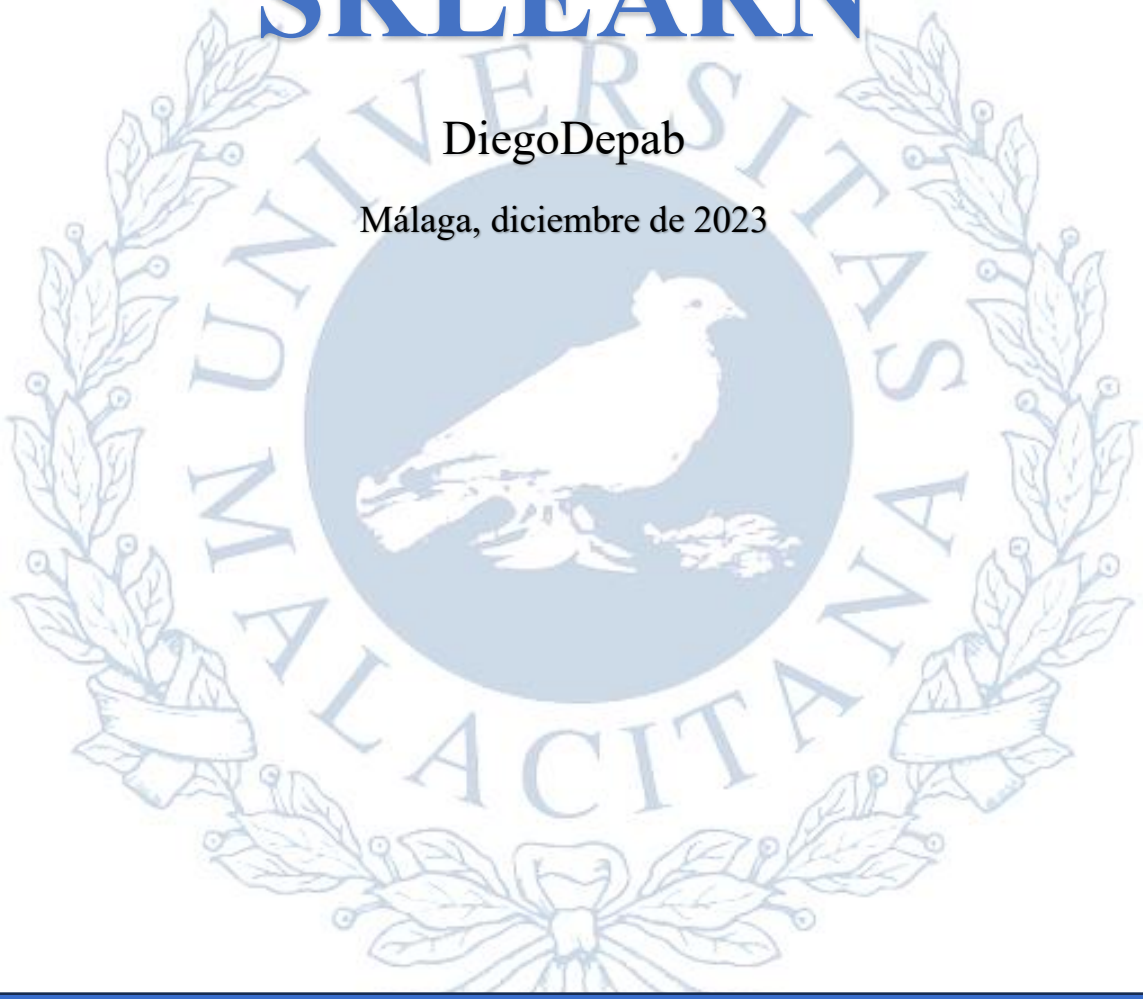
Laboratory Practice 4:

LIBRARY EXERCISES

SKLEARN

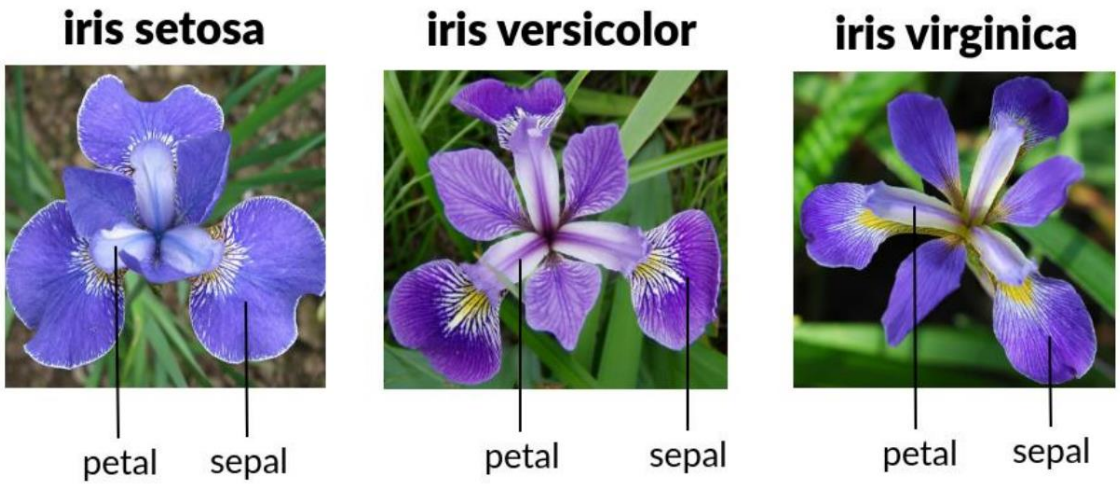
DiegoDepab

Málaga, diciembre de 2023



In this practice, the development of classifiers begins with selecting a classification dataset from the UCI Machine Learning Repository: https://archive.ics.uci.edu/. For simplicity and due to the lack of restrictions, the "**Iris**" dataset will be used in this work. This classic dataset, dating back to Fisher's 1936 paper, is a small and well-known dataset commonly used to evaluate classification methods. Given its widespread use and familiarity, it offers a valuable learning experience.

Dataset structure:



Classes [the chosen dataset has three classes, representing three species of iris plants]:

1. Iris Setosa,
2. Iris Versicolor,
3. Iris Virginica.

Attributes [the chosen dataset contains four features/attributes measured originally in centimeters]:

1. Sepal Length: the length of the iris flower's sepal.
2. Sepal Width: the width of the iris flower's sepal.
3. Petal Length: the length of the iris flower's petal.
4. Petal Width: the width of the iris flower's petal.

This dataset is based on 50 samples of three species of Iris (150 samples in total). These measures were used to create a linear discriminant model to classify the species.

Initial Code:

In the python console install with this command:

```
pip install ucimlrepo
```

We can see the dataset with this initial code:

```
from ucimlrepo import fetch_ucirepo
import pandas as pd
```

```
# fetch dataset
iris = fetch_ucirepo(id=53)

# data (as pandas dataframes)
features = iris.data.features #4 features: sepal length, sepal width,
petal length and petal width
print("Features: \n ", features)
targets = iris.data.targets  #3 classes: Iris Setosa, Iris Versicolor and
Iris Virginica
print("Targets (classes): \n", targets)

'''
#Other information that may be of interest
print(iris.metadata)   #metadata
print(iris.variables) #variable information  (cm in this case)
subset(iris, Species == "setosa")[1:5,]
'''
```

Console:

```
Features:
       sepal length  sepal width  petal length  petal width
0               5.1          3.5           1.4          0.2
1               4.9          3.0           1.4          0.2
2               4.7          3.2           1.3          0.2
3               4.6          3.1           1.5          0.2
4               5.0          3.6           1.4          0.2
..              ...          ...           ...          ...
145             6.7          3.0           5.2          2.3
146             6.3          2.5           5.0          1.9
147             6.5          3.0           5.2          2.0
148             6.2          3.4           5.4          2.3
149             5.9          3.0           5.1          1.8

[150 rows x 4 columns]
Targets (classes):
                class
0         Iris-setosa
1         Iris-setosa
2         Iris-setosa
3         Iris-setosa
4         Iris-setosa
..                ...
145  Iris-virginica
146  Iris-virginica
147  Iris-virginica
148  Iris-virginica
149  Iris-virginica

[150 rows x 1 columns]
```

Exercise 0: Then you must load it with the pandas library. You must clean up and convert any data, as necessary. Once the data is ready for use in a pandas DataFrame object,

The Iris flower dataset is a collection of measurements of the length and width of petals and sepals for several Iris flower species. To make this dataset more compact and user-friendly, it can be discretized, meaning that numerical values in the attributes can be converted into categories. This can be achieved by calculating the 33rd and 66th percentiles and applying thresholds to divide the values into small, medium, and large categories.

However, given that this practice focuses on evaluating the capabilities of the sklearn library, I believe that discretizing this dataset of only 150 data points would be a misguided approach, as it would result in a loss of information, thereby degrading the quality of the classifications.

If you want to use a database downloaded through Panda you can do it this way:

```
df = pd.read_csv('iris.data') #guarde el .csv en la misma carpeta por
privacidad
```

```
In [2]: df
Out[2]:
     5.1  3.5  1.4  0.2     Iris-setosa
0    4.9  3.0  1.4  0.2     Iris-setosa
1    4.7  3.2  1.3  0.2     Iris-setosa
2    4.6  3.1  1.5  0.2     Iris-setosa
3    5.0  3.6  1.4  0.2     Iris-setosa
4    5.4  3.9  1.7  0.4     Iris-setosa
..   ...  ...  ...  ...             ...
144  6.7  3.0  5.2  2.3  Iris-virginica
145  6.3  2.5  5.0  1.9  Iris-virginica
146  6.5  3.0  5.2  2.0  Iris-virginica
147  6.2  3.4  5.4  2.3  Iris-virginica
148  5.9  3.0  5.1  1.8  Iris-virginica

[149 rows x 5 columns]
```

Exercise 1: Classify the data with a Naïve Bayes classifier:

https://scikit-learn.org/stable/modules/naive_bayes.html

Solution (python source code)

```
#Exercise 1:  Classify the data with a Naïve Bayes classifier:
#pip install ucimlrepo

from ucimlrepo import fetch_ucirepo
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
```

```
from sklearn.preprocessing import LabelEncoder
df = pd.read_csv('iris.data') #use the same file for privacity


iris = fetch_ucirepo(id=53)  #WITH from ucimlrepo import fetch_ucirepo
#If you do not have the database downloaded (iris.data)

label_encoder = LabelEncoder()
for column in df.select_dtypes(include='object').columns:
    df[column] = label_encoder.fit_transform(df[column])
    features = df.drop('Iris-setosa', axis=1)
targets = df['Iris-setosa']
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(features, targets,
test_size=0.15, random_state=50)

# Create a Gaussian Naive Bayes model
model_nb = GaussianNB()
# Fit the model
model_nb.fit(X_train, y_train.values.ravel())

# Make predictions
y_pred = model_nb.predict(X_test)

# Evaluate the model
print("Confusion Matrix:", "\n", confusion_matrix(y_test, y_pred), "\n")
print("Accuracy:", "\n", accuracy_score(y_test, y_pred))
print("Classification report:", "\n",classification_report(y_test,
y_pred))

from sklearn.model_selection import cross_val_score
#All these tasks must be carried out using 10-fold cross-validation
scores = cross_val_score(model_nb, features, targets, cv=10)

print("Scores de validación cruzada:", "\n", scores)
print("Promedio de scores de validación cruzada:", "\n", scores.mean())
```

Code explanation:

1) **Libraries:** This code imports the functions and classes needed to download a data set, preprocess it, split it into training and test sets, train a Naive Bayes classification model, make predictions, and evaluate model performance. However, this code snippet only performs the necessary imports, it does not execute any of these tasks.

```
from ucimlrepo import fetch_ucirepo
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from sklearn.preprocessing import LabelEncoder
```

```
from ucimlrepo import fetch_ucirepo
```

- This line imports the fetch_ucirepo function from the ucimlrepo module, which is used to download datasets from the UCI Machine Learning Repository (in this case, the "iris" dataset).

```
import pandas as pd
```

- Import pandas (I consider that explaining this is redundant having done a practice for this package)

```
from sklearn.model_selection import train_test_split
```

- This line imports the train_test_split function from the sklearn.model_selection module. This function is used to split arrays or arrays into random training and testing subsets.

```
from sklearn.naive_bayes import GaussianNB
```

- This line imports the GaussianNB class from the sklearn.naive_bayes module. GaussianNB implements the Naive Bayes classification algorithm for Gaussian probability distribution classification of features, as specified in the problem statement.

```
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
```

- These functions are imported from the sklearn.metrics module for evaluating the performance of classification models. accuracy_score calculates the accuracy of the model, classification_report provides a detailed breakdown of the classification metrics, and confusion_matrix generates a confusion matrix.

```
from sklearn.preprocessing import LabelEncoder
```

- This line imports the LabelEncoder class from the sklearn.preprocessing module. LabelEncoder is used to convert categorical labels to numbers.

2) **Data preparation:** This stage involves downloading the dataset, extracting relevant features and target classes, and splitting the data into training and testing sets. These are standard initial steps in a machine learning project. However, machine learning models are not yet trained in this stage.

```
df = pd.read_csv('iris.data') #use the same file for privacity

iris = fetch_ucirepo(id=53) )  #WITH from ucimlrepo import fetch_ucirepo
#If you do not have the database downloaded (iris.data)

# data (as pandas dataframes)
label_encoder = LabelEncoder()
for column in df.select_dtypes(include='object').columns:
    df[column] = label_encoder.fit_transform(df[column])
```

```
    features = df.drop('Iris-setosa', axis=1)
targets = df['Iris-setosa']
```

features = df.drop('Iris-setosa', axis=1): Creates a new DataFrame called features that contains all the columns of df except 'Iris-setosa' (when using the df.columns command we see that the program calls ' Iris-setosa' the target column that the model will try to predict),

targets = df['Iris-setosa']: Creates a pandas series called targets that contains the values of the 'Iris-setosa' column of df. These are the target values that the model will try to predict.

```
X_train, X_test, y_train, y_test = train_test_split(features, targets,
test_size=0.15, random_state=50)
```

This last line partitions the features and target classes into training and testing subsets. Eighty five percent of the data will be utilized for training the model, while the remaining thirty percent will be employed for evaluating its performance. The parameter random_state=50 is employed to guarantee the reproducibility of the split.

3) **Gaussian Naive Bayes model:** This code constructs a Naive Bayes Gaussian classification model and then utilizes the trained model for making predictions.

```
# Create a Gaussian Naive Bayes model
model_nb = GaussianNB()
# Fit the model
model_nb.fit(X_train, y_train)
# Make predictions
y_pred = model_nb.predict(X_test)
```

```
model_nb = GaussianNB()
```

- This line instantiates a GaussianNB object, which embodies the Naive Bayes Gaussian classification algorithm. This object is assigned to the variable model_nb for subsequent usage.

```
model_nb.fit(X_train, y_train)
```

- This line equips the model_nb model with the training data, which entails the features embedded in X_train and the corresponding target classes stored in y_train.

```
y_pred = model_nb.predict(X_test)
```

- This line leverages the trained model_nb to generate predictions for the test data, which are stored in the variable y_pred. (This will be useful for the next step where we evaluate the created model.)

4) **Evaluate the model:** This code assesses the performance of the trained Naive Bayes Gaussian classification model by employing various metrics, and then presents the evaluation results. These metrics provide a comprehensive overview of the model's performance, allowing for a clear understanding of its strengths and areas for improvement.

```
print("Confusion Matrix:", "\n", confusion_matrix(y_test, y_pred), "\n")
print("Accuracy:", "\n", accuracy_score(y_test, y_pred))
print("Classification report:", "\n",classification_report(y_test,
y_pred))
```

```
print("Confusion Matrix:", "\n", confusion_matrix(y_test, y_pred), "\n")
```

- This line generates and displays the confusion matrix for the model. A confusion matrix is a tabular presentation that summarizes the performance of a classification model on a dataset with known true values. It provides a breakdown of the number of correctly and incorrectly classified instances for each class. The rows of the matrix represent the true classes, while the columns represent the predicted classes.

```
print("Accuracy:", "\n", accuracy_score(y_test, y_pred))
```

- This line computes and presents the accuracy of the model. Accuracy is a crucial evaluation metric, calculated as the proportion of instances that the model correctly classifies.

```
print("Classification report:", "\n",classification_report(y_test,
y_pred))
```

- This line produces and displays a classification report, which encapsulates a comprehensive summary of the key classification metrics, including precision, recall (sensitivity), f1-score, and support (instance count) for each class. Recall measures the proportion of positive instances that were correctly identified. The f1-score combines precision and recall into a single metric, with 1 representing the optimal value and 0 representing the worst-case scenario. Support represents the number of actual instances for each class within the test set.

5) **Using 10-fold cross validation:** To be honest, once I had finished this statement, I read that "All these tasks must be carried out using 10-fold cross-validation, so I will put this section to include this point.

```
from sklearn.model_selection import cross_val_score
#All these tasks must be carried out using 10-fold cross-validation
scores = cross_val_score(model_nb, features, targets, cv=10)

print("Scores de validación cruzada:", "\n", scores)
print("Promedio de scores de validación cruzada:", "\n", scores.mean())
```

```
scores = cross_val_score(model_nb, features, targets, cv=10)

print("Scores de validación cruzada:", "\n", scores)
```

- Cross-validation scores: These are the precision scores that the model obtained in each of the 10 cross-validation folds. Each number is the accuracy (the number of correct predictions divided by the total number of predictions) that the model obtained in a specific cross-validation fold.

```
print("
```

- This is the average of the previous 10 accuracy scores. It is a summary measure of overall model performance in 10-fold cross-validation.

OUTPUT:

Confusion Matrix:

```
Confusion Matrix:
 [[6 0 0]
 [0 8 0]
 [0 0 9]]
```

A **confusion matrix** is a table that is used to evaluate the performance of a classification model. The rows of the matrix represent the actual classes, and the columns represent the predicted classes. The entries in the matrix represent the number of instances that were classified into each class.

In this case we can see how 15% of the data (22.5, that is, 23 rows of the dataset) were used to check its predictions, in the case of using random_state=50 all the predictions will be correct, our model could correctly predict each of the 23 flowers just by knowing their features

In order to learn more about how the confusion matrix works and take advantage and see what happens when the training data changes, I will show what happens **if we use a test_size=0.3, random_state=1, the confusion matrix is as follows:**

```
Confusion Matrix:
 [[16  0  0]
 [ 0 14  2]
 [ 0  1 12]]
```

Actual/predict	Iris-setosa	Iris-versicolor	Iris-virginica
Iris-setosa	16	0	0
Iris-versicolor	0	14	2
Iris-virginica	0	1	12

In layman's terms, we can observe that y_train contained 16 iris-setosas, 16 Iris-versicolor, and 13 Iris-virginica instances (total45 (30%of 150). Our model was trained solely in the features, and we evaluated its performance by counting the number of correct predictions. In this case, all iris-setosa were correctly identified, while one versicolor iris was mistakenly classified as virginica and vice versa. This resulted in a total of 3 errors in the 45 test instances. The confusion matrix indicates that the model performed exceptionally well for iris-setosa but struggled slightly with distinguishing between iris-versicolor and iris-virginica, which share some similar characteristics. Overall, the model achieved impressive results for this particular test set.

Accuracy:

Returning with test_size=0.15, random_state=50)

```
Accuracy:
 1.0
```

Accuracy: This is much simpler to explain, it would be the division of the number of correct answers by the total test data

$$Accuracy: \frac{Correct_{prediction}}{test_{data}} = \frac{45}{45} = 1$$

Classification report:

```
Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      1.00      1.00         8
           2       1.00      1.00      1.00         9

    accuracy                           1.00        23
   macro avg       1.00      1.00      1.00        23
weighted avg       1.00      1.00      1.00        23
```

We can see.

- **Precision:** The proportion of positive predictions that are correct. Precision = TP / (TP + FP)
- **Recall**: The proportion of positive instances that are correctly classified. TP / (TP + FN)
- **F-measure:** The harmonic means of precision and recall. 2 * (Precision * Recall) / (Precision + Recall)
- **Support** is the number of actual instances for each class in the test set.

where:

- TP = True Positive
- FP = False Positive
- FN = False Negative

I consider that this is another way of seeing the results and comparing the quality of the model, but I could not say more information than in the 2 sections, only that it corroborates the ideas said in a more visual way, seeing that the model complies with what is desired.

10-fold cross validation

```
Scores de validación cruzada:
 [0.93333333 0.93333333 1.         0.93333333 0.93333333 0.93333333
 0.86666667 1.         1.         1.        ]
Promedio de scores de validación cruzada:
 0.9533333333333334
```

The first cross-validation score value is 0.93333333, which means that the model had an accuracy of 93.33% in the first fold of cross-validation. The average cross-validation

score is 0.9533333333333334, which means that, on average, the model had an accuracy of 95.33% in the 10-fold cross-validation. which gives good indications that the model is well adapted for the samples we have.

From here on, I will avoid repeating myself so much since we will see many things seen in this exercise in others but applied for a different classification. With the intention of avoiding redundancies, try to make the first exercise the one that has all the explanations, both in code and theoretically. of results, now I will start more by expressing results and things that I have not explained before.

Exercise 2: Classify the data with a nearest neighbours' classifier:

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

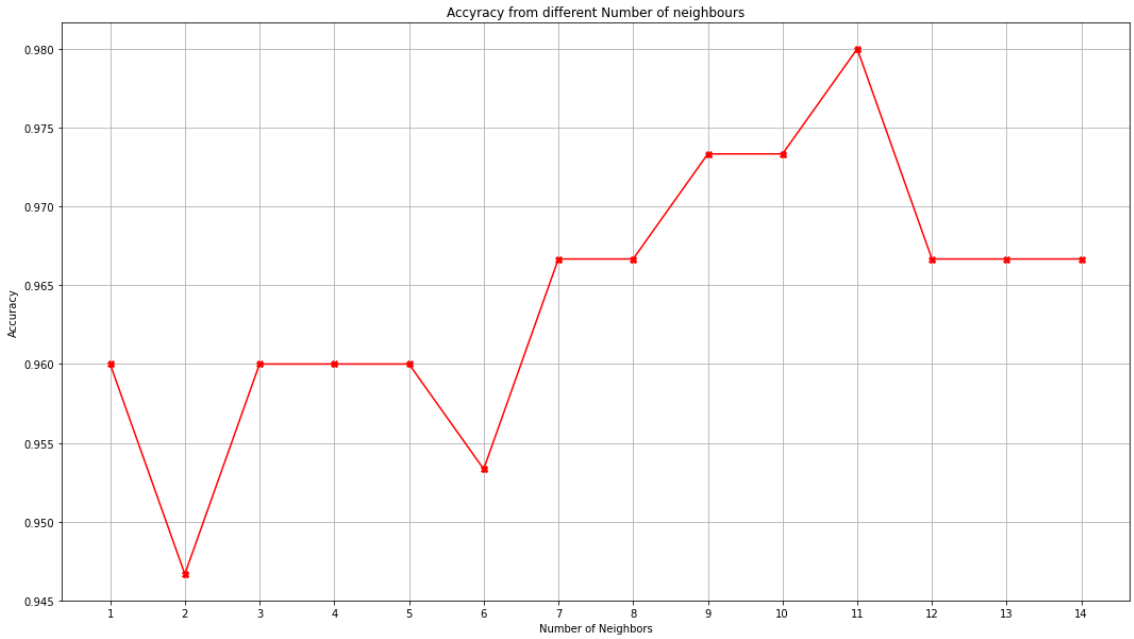
Solution (python source code)

```
#Exercise 2:  Classify the features with a nearest neighbours'
classifier:

from ucimlrepo import fetch_ucirepo
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import model_selection as mdl
from sklearn import neighbors
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('iris.data') #use the same file for privacity


iris = fetch_ucirepo(id=53)  #df = pd.read_csv('C:/Users/Diego De
Pablo/.spyder-py3/Practica 1/iris.features')

# data (as pandas dataframes)
label_encoder = LabelEncoder()
for column in df.select_dtypes(include='object').columns:
    df[column] = label_encoder.fit_transform(df[column])
    features = df.drop('Iris-setosa', axis=1) #print(df.columns)
targets = df['Iris-setosa']
# Split the dataset into training and test sets
features_train, features_test, target_train, target_test =
train_test_split(features, targets,
test_size=0.15, random_state=50)
# Convertion to a numpy array format
arr_target = targets.values
arr_features = features.values

# 10-fold cross validation
skf = mdl.StratifiedKFold(n_splits=10, shuffle=True, random_state=50)
```

```
# Range of neighbors to try
rango = range(1, 30) #poner el numero maximos de vecinos que quieres

# Calculate accuracy scores for each number of neighbors
score = []

for ngb in rango:
    clf = neighbors.KNeighborsClassifier(n_neighbors=ngb)
    clf.fit(features_train, target_train)
    scores = mdl.cross_val_score(clf, arr_features, arr_target, cv=skf,
scoring='accuracy')
    score.append(scores.mean())

# Find the best number of neighbors
best = rango[np.argmax(score)]
print("The best number of neighbors is:", best)

#  Plot the accuracy scores for different numbers of neighbors
plt.figure(figsize=(18, 10))
plt.plot(rango, score, marker='X', color='red')
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.title('Accyracy from different Number of neighbours')
plt.xticks(rango)
plt.grid(True)
plt.show()
```

OUTPUT

```
The best number of neighbors is: 22
```



Code variation where instead of graphing and searching I use the GridSearchCV library.

```
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

df = pd.read_csv('iris.data') #use the same file for privacity


label_encoder = LabelEncoder()
for column in df.select_dtypes(include='object').columns:
    df[column] = label_encoder.fit_transform(df[column])
    features = df.drop('Iris-setosa', axis=1) #print(df.columns)
targets = df['Iris-setosa']
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(features, targets,
test_size=0.10, random_state=50)
# Define the parameter values that should be searched
k_range = list(range(1, 31))

# Create a parameter grid: map the parameter names to the values that
should be searched
param_grid = dict(n_neighbors=k_range)

# Instantiate the grid
grid = GridSearchCV(KNeighborsClassifier(), param_grid, cv=10,
scoring='accuracy')
```

```
# Fit the grid with data
grid.fit(X_train, y_train)

# View the complete results
print(grid.cv_results_)

# Examine the best model

print("grid.best_score_:", "\n", grid.best_score_, "\n")
print("grid.best_params_:", "\n", grid.best_params_)
print("grid.best_estimator_:", "\n",grid.best_estimator_)
```

OUTPUT

```
grid.best_score_:
 0.9870329670329704

grid.best_params_:
  KNeighborsClassifier(n_neighbors=22)
grid.best_estimator_:
 KNeighborsClassifier(n_neighbors=22)
```

Explication

The findings suggest that the K-Nearest Neighbors (KNN) model with 22 neighbors exhibits the highest accuracy, achieving an approximate value of 0.987. This observation aligns with both the generated graph and the results obtained from the second code, where grid.best_score_ returns a precision of 0.98703296703296704 and grid.best_params_ indicates an optimal number of neighbors equal to 22.

The constructed graph illustrates the model's accuracy as a function of the number of neighbors. The Y-axis represents accuracy, while the X-axis represents the number of neighbors. The red line in the graph depicts the accuracy's variation as the number of neighbors increases. Per the graph, accuracy peaks around the 22th neighbor, which corroborates the findings from grid.best_params_.

In conclusion, 22 emerges as the optimal number of neighbors for this specific dataset when employing the KNN classifier. This implies that when classifying a new data point, we must consider the 22 nearest data points in the training set to determine the class of the new data point. Additionally, it's worth noting that if we increase the value of k to 100, we might encounter instances where accuracy equals 22, However, as evident in the code, the model will retain the optimal status until a higher accuracy value is discovered. If we encounter similar accuracy levels, it's preferable to maintain the model as simply as possible to minimize susceptibility to future alterations. Therefore, we maintain the number of neighbors at 22.

Exercise 3: Classify the data with a decision tree classifier:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

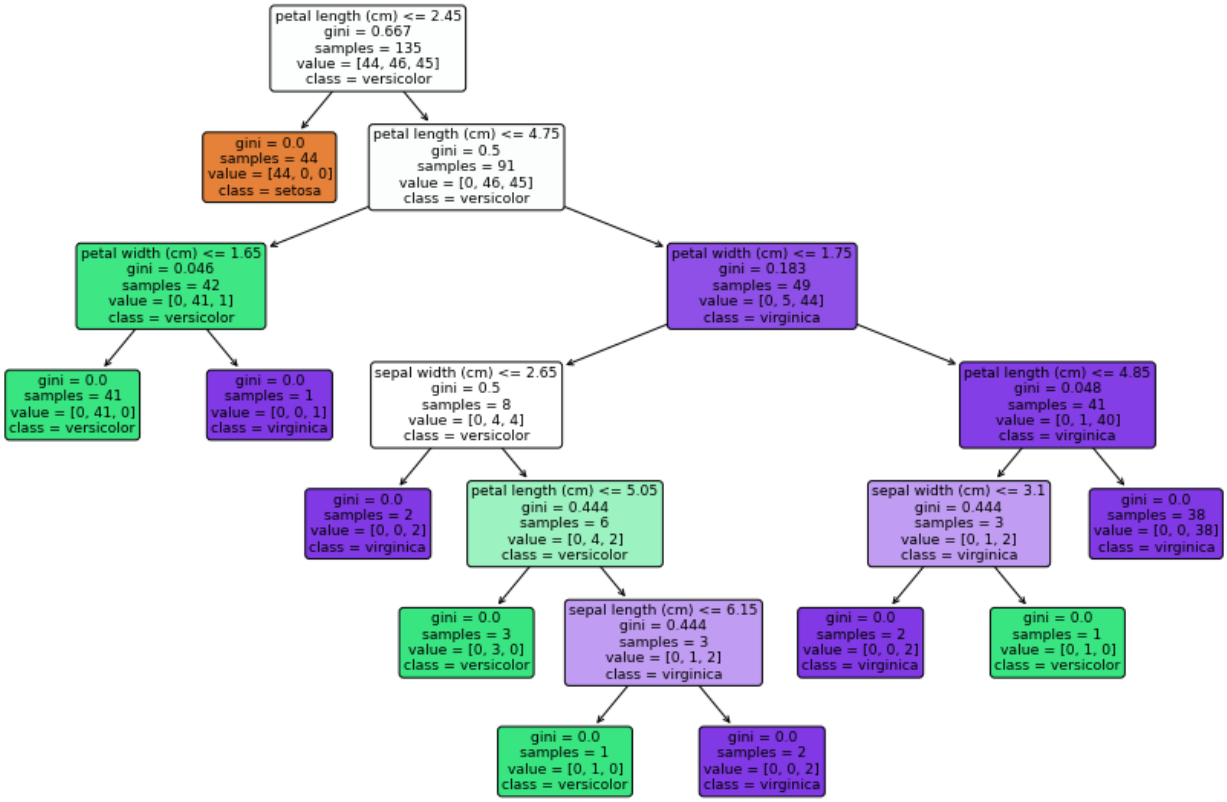
Solution (python source code)

For this exercise, there are various approaches, from quantifying the data to manually assigning categories based on percentiles. To provide a comprehensive comparison, I've created two codes: one that prioritizes achieving reasonable accuracy and another that involves hyperparameter optimization and visualizing the optimal decision tree.

```
#from sklearn.preprocessing import LabelEncoder
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df['target'] = iris.target

features = df.drop('target', axis=1)
targets = df['target']

# Split the dataset into training and test sets
features_train, features_test, target_train, target_test =
train_test_split(features, targets, test_size=0.10, random_state=50)

# Create a Decision Tree model
model = DecisionTreeClassifier()

# Fit the model
model.fit(features_train, target_train)

# Create a figure
plt.figure(figsize=(15,10))

# Draw the decision tree
plot_tree(model, filled=True, rounded=True,
class_names=iris.target_names.tolist(), feature_names=iris.feature_names)
# Show the figure
plt.show()

target_pred = model.predict(features_test)
print("Accuracy:", accuracy_score(target_test, target_pred))
```

OUTPUT:

```
Accuracy: 1.0
```

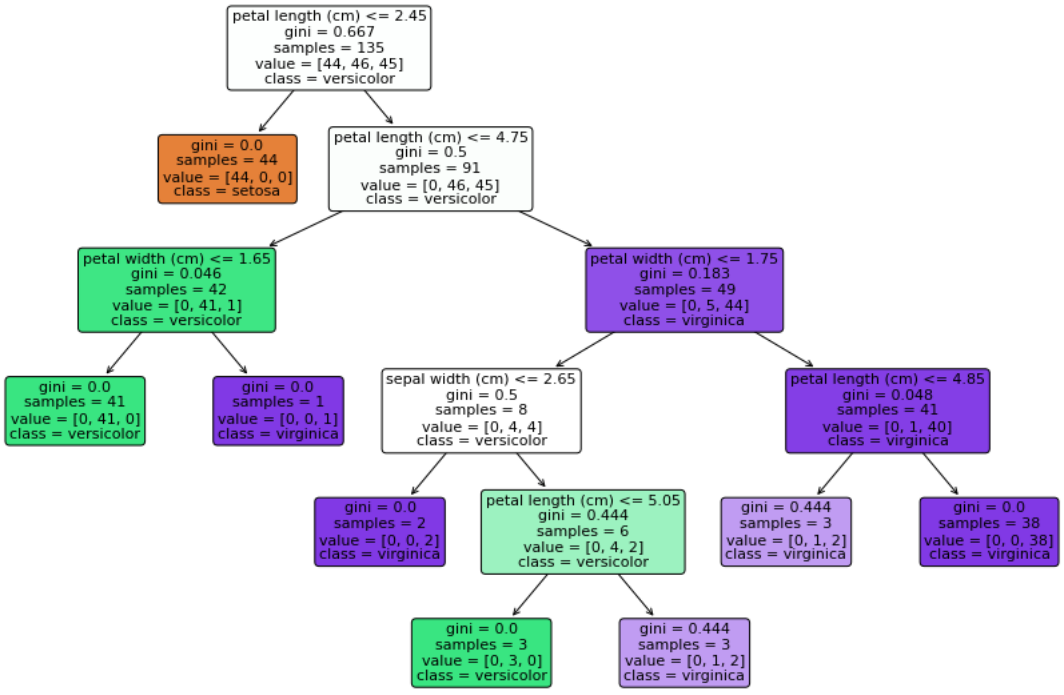


Solution (python source code)

```
print("----------You must find out the best hyperparameters and plot the
best decision tree. ---------------")


# New code for hyperparameter tuning
from sklearn.model_selection import GridSearchCV

rangeX = list(range(2, 50))

# Create a parameter grid: map the parameter names to the values that
should be searched
param_grid = dict(min_samples_split=rangeX)

# Instantiate the grid
grid = GridSearchCV(model, param_grid, cv=10, scoring='accuracy')

# Fit the grid with data
grid.fit(features_train, target_train)

# View the complete results: print(grid.cv_results_)

# Examine the best model
print("\nBest score: ", grid.best_score_)
print("Best params: ", grid.best_params_)
```

```
# Create a Decision Tree model with best parameters
best_model =
DecisionTreeClassifier(min_samples_split=grid.best_params_['min_samples_s
plit'])

# Fit the model
best_model.fit(features_train, target_train)

# Create a figure
plt.figure(figsize=(15,10))

# Draw the decision tree
plot_tree(best_model, filled=True, rounded=True,
class_names=iris.target_names.tolist(), feature_names=iris.feature_names)

# Show the figure
plt.show()
target_pred = best_model.predict(features_test)
print("Accuracy:", accuracy_score(target_test, target_pred))
```

OUTPUT:




```
----------You must find out the best hyperparameters and plot the best decision tree. ---------------

Best score:  0.9472527472527472
Best params:  {'min_samples_split': 6}
Accuracy: 0.9333333333333333
```

Explain:

As we can observe, these two decision trees, though seemingly similar, provide valuable insights into the actual process of decision tree construction. Unlike the previous examples, these trees demonstrate an efficient data saving mechanism. The first tree prioritizes achieving the highest possible accuracy, even if it comes at a higher computational and spatial cost. With this dataset, it achieved a perfect accuracy of 1, classifying all training data instances correctly. However, it comes at the cost of a deeper tree with seven levels and nineteen leaves. The second approach, which emphasizes hyperparameter optimization, achieves a very good accuracy of 0.933 while maintaining a more compact tree structure with six levels and fifteen leaves. This involves a trade-off between accuracy and computational efficiency. While the difference may seem insignificant for this small dataset with 150 instances, it becomes more pronounced when dealing with datasets containing thousands or even millions of rows. In such cases, eliminating even a single level of depth can significantly reduce computational overhead."

Exercise 4: Classify the data with a Support Vector Machine classifier:

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

You must find out the best kernel and hyperparameters.

Solution (python source code)

```
#4) Classify the data with a Support Vector Machine classifier:
    #You must find out the best kernel and hyperparameters.
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from sklearn.preprocessing import LabelEncoder
from sklearn import svm
#Prepare the features and target variable
df = pd.read_csv('iris.data')
features = df.drop('Iris-setosa', axis=1) #print(df.columns)
targets = df['Iris-setosa']

#encode the target variable
label_encoder = LabelEncoder()
targets = label_encoder.fit_transform(targets)

#split the dataset in train and test (10% test)
features_train, features_test, targets_train, targets_test =
train_test_split(features, targets, test_size=0.1, random_state=50)
svm_classifier = svm.SVC(random_state=50)
```

```
#param_grid is defined where the keys are the names of the
hyperparameters
# be tuned and the values are the lists of values to try. dictionary that
specifies which
#hyperparameters you'd like to tune for the Support Vector Machine (SVM)
classifier.
param_grid = {
 'C': [0.1, 1, 10, 20],
 'kernel': ['linear', 'poly', 'rbf', 'sigmoid']
}

# Perform grid search to find the best hyperparameters
from sklearn import model_selection as mdl
grid_search = mdl.GridSearchCV(svm_model, grid, cv=10,
scoring='accuracy')
grid_search.fit(data_train, target_train)

# Best hyperparameters
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

# Best model found
best_svm = grid_search.best_estimator_

# Predictions on the test set with the best model
predictions = best_svm.predict(data_test)

# Evaluation
from sklearn import metrics
accuracy = metrics.accuracy_score(target_test, predictions)
print("Best hyperparameters accuracy:", accuracy)
classif = metrics.classification_report(target_test, predictions)
print(classif)
```

Output:

```
find out the best kernel and hyperparameters.: {'C': 20, 'kernel': 'rbf'}

Confusion matrix:
 [[ 6  0  0]
 [ 0 13  0]
 [ 0  0 11]]

Accuracy: 1.0

Classification Report;                precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Output if I use randomstate=1

```
train_test_split(features, targets, test_size=0.1, random_state=1)
svm_classifier = svm.SVC(random_state=1)
```

```
cambiando el random_state a '1'
find out the best kernel and hyperparameters.: {'C': 1, 'kernel': 'rbf'}

Confusion matrix:
 [[10  0  0]
 [ 0 13  0]
 [ 0  1  6]]

Accuracy: 0.9666666666666667

Classification Report;              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       0.93      1.00      0.96        13
           2       1.00      0.86      0.92         7

    accuracy                           0.97        30
   macro avg       0.98      0.95      0.96        30
weighted avg       0.97      0.97      0.97        30
```

In this instance, we can compare the output to that of the first program, where the chosen random state yielded perfect results. However, I also implemented a different random state for better comparison and to elucidate the obtained outcomes.

The first printout reveals "Best hyperparameters: {'C': 20, 'kernel': 'rbf'}", indicating that the optimal hyperparameters for the SVM classifier, as determined by the grid search, are a C value of 20 and a kernel of type 'rbf' (Radial basis function). Using random_state=1 resulted in {'C': 1, 'kernel': 'rbf'}, which implies a C value of 1 instead of 20 while retaining the same kernel type.

The confusion matrix, as before, reflects perfect classification in the initial instance. However, when utilizing random_state=1, the matrix exhibits a single misclassified instance.

Accuracy is a metric for assessing classification models. In this case, the accuracy is 1.0, which signifies that the model accurately classified all instances in the test set.

The classification report provides precision, recall, and f1-score metrics for each class, along with the weighted average of these metrics. In your case, all these metrics are 1.0 for all classes, demonstrating that the model exhibits flawless performance.

Exercise 5: Compare the performance of the previous classifiers. You must report several classification performance measures. You should show the Receiver Operating Characteristic plots for the compared models:

https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html

You must find out the best kernel and hyperparameters.

Solution (python source code)

```
# 5. Compare the performance of the previous classifiers.
#You must report several classification performance measures.
#You should show the Receiver Operating Characteristic plots for the
compared models:
import pandas as pd
from sklearn import model_selection as mdl
from sklearn import metrics
from matplotlib import pyplot as plt
from sklearn import naive_bayes
from sklearn import neighbors
from sklearn import tree
from sklearn import svm
from sklearn import preprocessing

df = pd.read_csv('iris.data', names=['sepal_length', 'sepal_width',
'petal_length', 'petal_width', 'class'])

label=preprocessing.LabelEncoder()
for column in df.select_dtypes(include='object').columns:
    df[column]=label.fit_transform(df[column])
target=df.iloc[:,-1] # Use the last column as the target variable
data=df.iloc[:,:-1] # Use all but the last column as the features

# models
nb_model = naive_bayes.GaussianNB()
knn_model = neighbors.KNeighborsClassifier(n_neighbors=22)
dtc_model = tree.DecisionTreeClassifier(random_state=50)
svm_model = svm.SVC(probability=True)

#models name:
models = [nb_model, knn_model, dtc_model, svm_model]
names = ['Naive Bayes', 'nearest neighbors', 'Decision Tree', ' Support
Vector Machine']

#  10-fold cross-validation
cv = mdl.StratifiedKFold(n_splits=10, shuffle=True, random_state=50)

# Initialize lists
accuracies = []
```

```
precisions = []
recalls = []
f1_scores = []
roc_auc_scores = []

# Function to calculate ROC curve and AUC
def calculate_automatic(x, y, modelo):
    probabilities = model.predict_proba(data)
    # Calculate ROC curve with class probabilities
    fpr, tpr, _ = metrics.roc_curve(y, probabilities[:, 1], pos_label=1)

    roc_auc = metrics.auc(fpr, tpr)
    return fpr, tpr, roc_auc

# Iterate through models
for model, name in zip(models, names):
    model.fit(data, target) # Fit the model

    # Cross-validation
    predictions = mdl.cross_val_predict(model, data, target, cv=cv)

    # Calculate performance metrics
    accuracy = metrics.accuracy_score(target, predictions)

    precision = metrics.precision_score(target, predictions,
average='micro')

    recall = metrics.recall_score(target, predictions, average='micro')

    f1 = metrics.f1_score(target, predictions, average='micro')

    # Calculate class probabilities
    probabilities = mdl.cross_val_predict(model, data, target, cv=cv,
method='predict_proba')

    # Calculate ROC AUC score with class probabilities
    roc_auc = metrics.roc_auc_score(target, probabilities,
multi_class='ovr')

    # Add values to lists
    accuracies.append(accuracy)
    precisions.append(precision)
    recalls.append(recall)
    f1_scores.append(f1)
    roc_auc_scores.append(roc_auc)

    # Plot ROC curve
    fpr, tpr, roc_auc = calculate_automatic(data, target, model)
    plt.figure(figsize=(12, 12))
```

```
    plt.plot(fpr, tpr, label=f'{name} (AUC = {roc_auc:.3f})')
    plt.plot([0, 1], [0, 1], linestyle='-', color='black',
label='Ramdod')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(f'ROC Curve - {name}')
    plt.legend()
    plt.show()


# Performance metrics
perf_metrics = pd.DataFrame({'Model': names,
                             'Accuracy': accuracies,
                             'Precision': precisions,
                             'Recall': recalls,
                             'F1 Score': f1_scores,
                             'ROC AUC Score': roc_auc_scores})
print(perf_metrics)
```

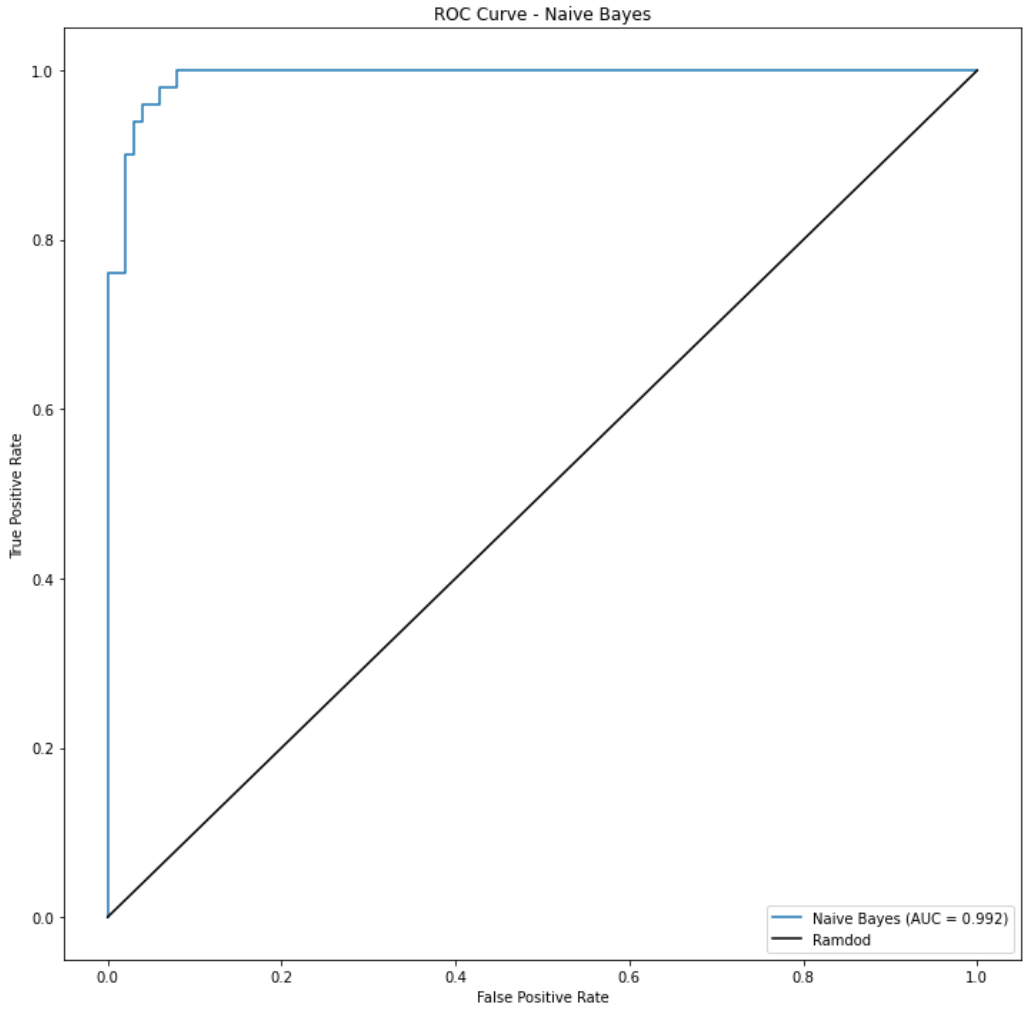
OUTPUT WITH ITS EXPLANATION:

	Model	Accuracy	...	F1 Score	ROC AUC Score
0	Naive Bayes	0.966667	...	0.966667	0.994000
1	nearest neighbors	0.973333	...	0.973333	0.996067
2	Decision Tree	0.973333	...	0.973333	0.980000
3	Support Vector Machine	0.966667	...	0.966667	0.997600

I believe the key points have been thoroughly explained in this work. However, it's worth emphasizing the AUC ROC score of the model, which represents the area under the ROC (Receiver Operating Characteristic) curve. This metric provides an overall assessment of model performance across the range of possible classification thresholds. As we've seen, altering the random state can lead to different values. For instance, using random_state=1 yielded values that weren't as close to 100.

Now I will move on to the generated images, they will all follow the same trail of algorithms that reach true positive read high very quickly due to the simplicity of the study material and the limitation of being such a small dataset.

Naïve bayes:



The provided image depicts the Receiver Operating Characteristic Curve (ROC) for the Naive Bayes classification model. It visually represents the performance of the model in distinguishing between positive and negative instances as the classification threshold varies.

The x-axis represents the False Positive Rate (FPR), which signifies the proportion of negative observations incorrectly classified as positive. It ranges from 0 to 1.

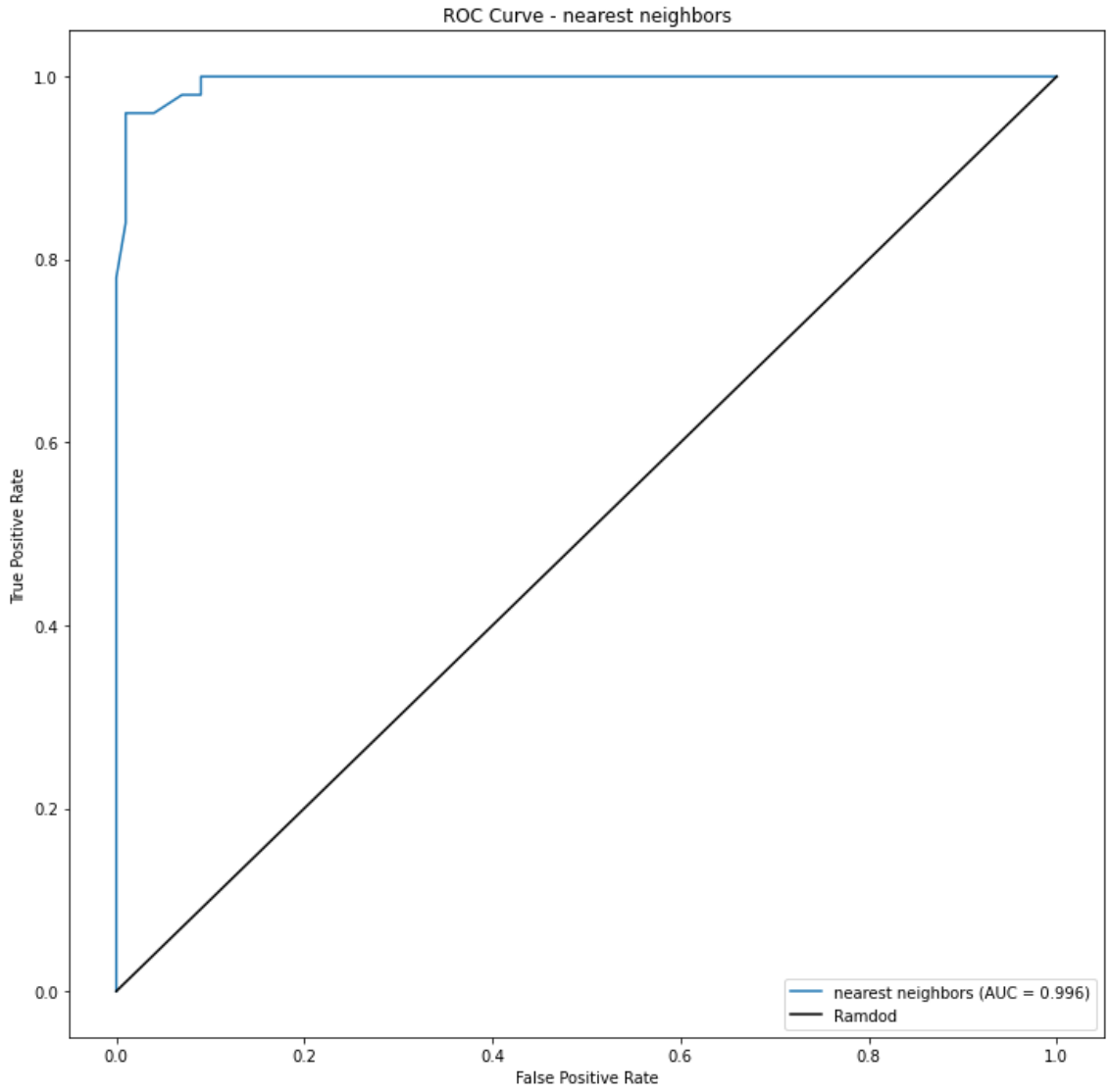
The y-axis represents the True Positive Rate (TPR), representing the proportion of positive observations correctly classified. It also ranges from 0 to 1.

The blue line on the graph represents the ROC curve for the Naive Bayes model. A closer position of this line towards the top-left corner of the graph indicates superior model performance.

The black diagonal line represents a random classifier, providing a benchmark for comparison. An effective classification model should exhibit an ROC curve significantly above this line.

The area under the curve (AUC) serves as a metric to assess overall model performance. The model's AUC of 0.992 highlights its exceptional performance.

Nearest Neighbors:



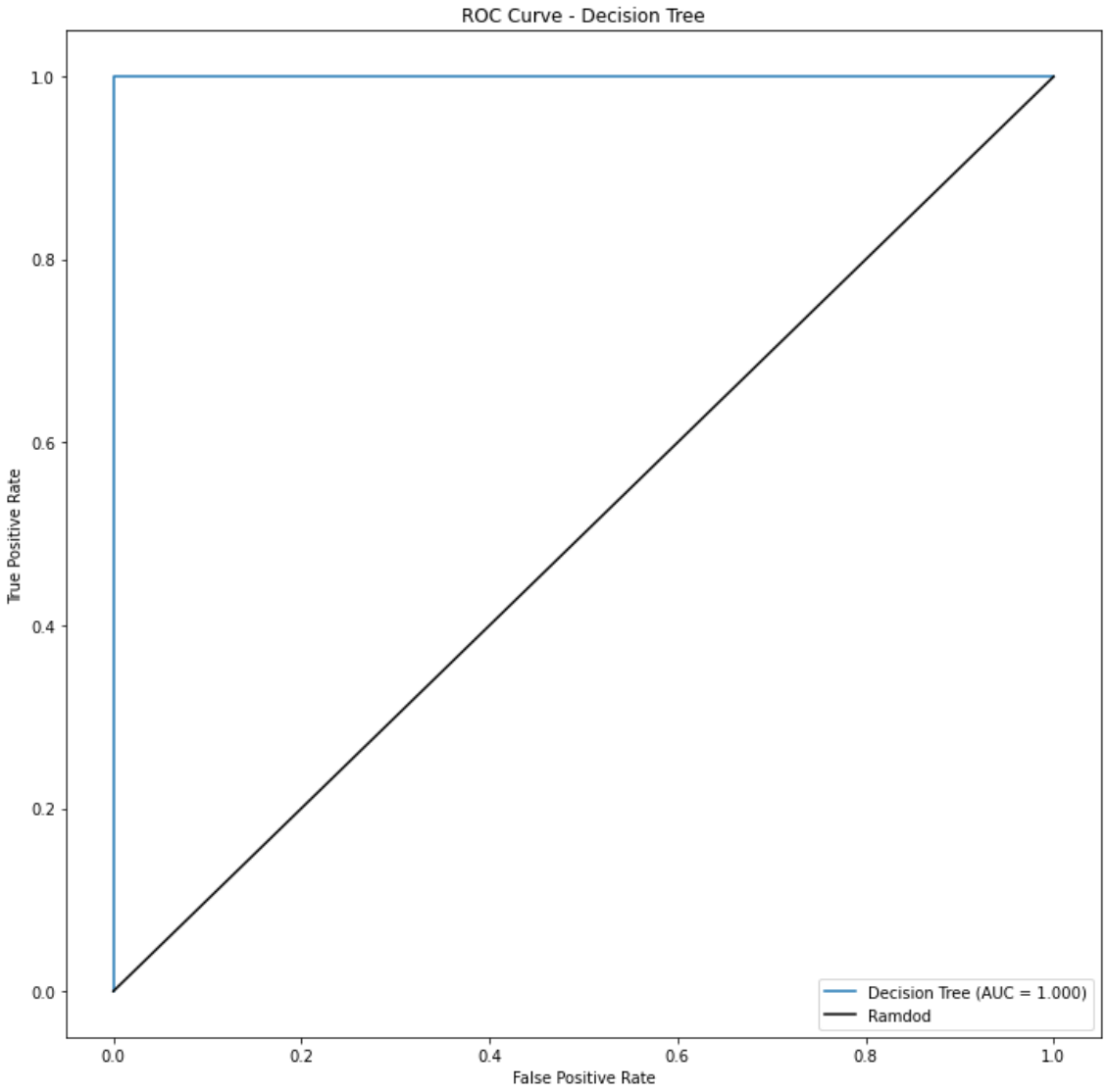
Reiterating the same analysis would not be particularly insightful at this stage, so let's focus on highlighting the key differences between the two graphs: the first representing the Naive Bayes model's performance and the second depicting the Nearest Neighbors model's performance.

Area Under the Curve (AUC): The AUC for the Naive Bayes model is 0.992, while the AUC for the Nearest Neighbors model is 0.996. This indicates a slight edge for the Nearest Neighbors model in terms of overall performance on your dataset.

ROC Curve: Both ROC curves closely approach the top-left corner of the graph, suggesting high performance for both models. Nevertheless, the ROC curve for the Nearest Neighbors model is slightly closer to the top-left corner, implying a higher True

Positive Rate (TPR) and a lower False Positive Rate (FPR) compared to the Naive Bayes model."

DecisionTree:



While a perfect ROC curve may seem desirable, it can raise concerns in practice. Here's why:

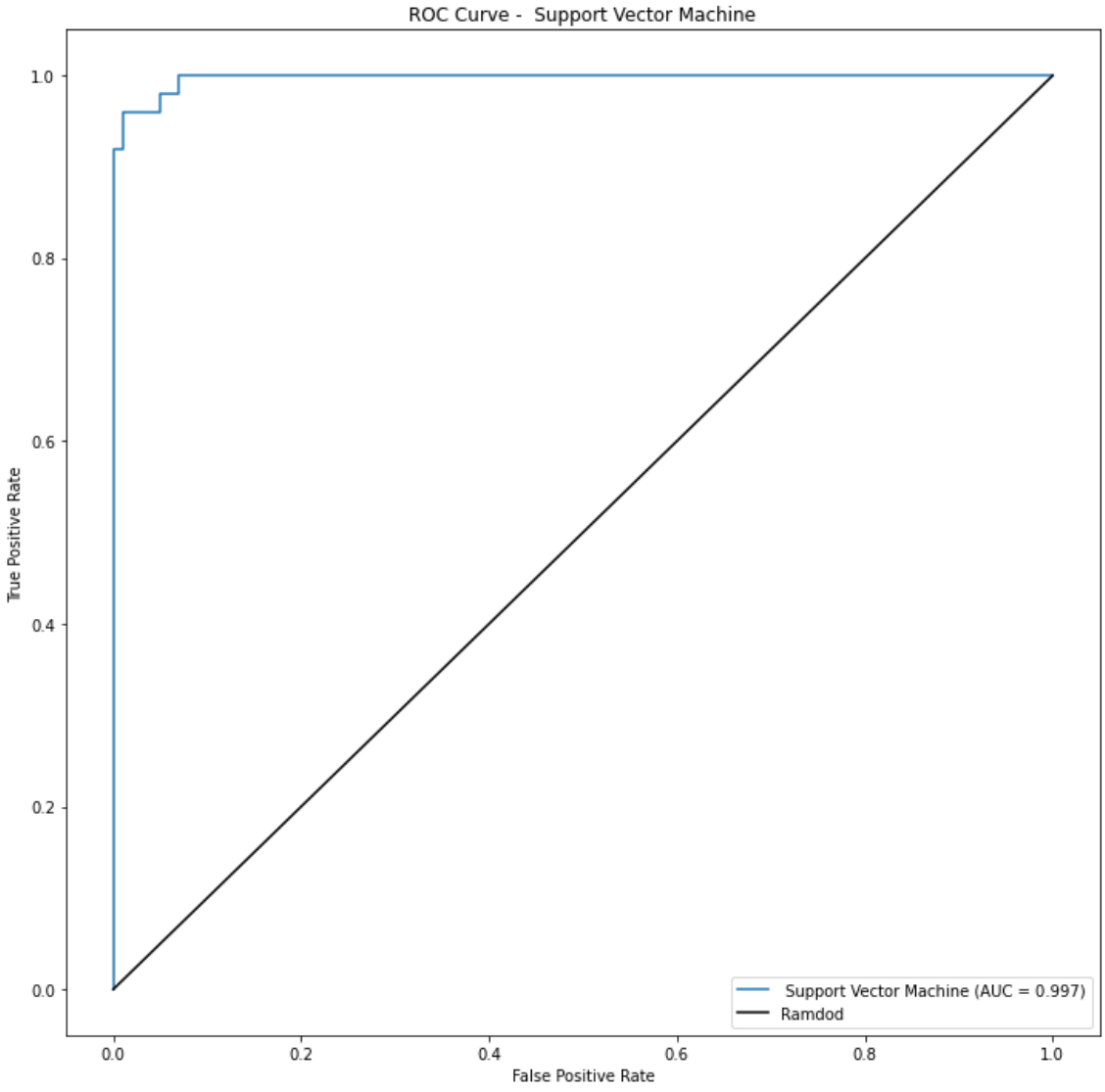
Overfitting: A model that performs flawlessly on training data may be overfitting the data. This means it may be memorizing specific details or noise from the training set instead of learning generalizable patterns. As a result, the model may not perform well on unseen data.

Data Issues: A perfect ROC curve can also signal underlying data problems. For instance, there might be data leakage, where the target variable has somehow been incorporated into the features. Additionally, there could be errors in data collection or processing.

Inappropriate Validation: Since we're utilizing cross-validation, it's crucial to ensure it's implemented correctly. If test data is inadvertently used during training, it could lead to an inflated assessment of the model's effectiveness.

I spent some time checking what was happening and I did not find the error, it could be that by reusing so much code a variable was saved incorrectly or I used that model that prioritized reaching accuracy 100 instead of obtaining the best hyperparameters

Support Vector Machine:



Honestly, I think it would be repeating myself too much to analyze another similar graph again. It shows good performance. Having already talked about the problem of perfection, I consider that the best model is this one, having good values from the beginning and immediately adjusting to perfection.

Note: I understand that this document only requested Word, but I did this work both in .py files from which I took the screenshots and also in a Jupyter notebook which I uploaded on my github page, if necessary, corroborate part of the code here I attach the link to see the code

https://github.com/Diegodepab/Useful_python_libraries/blob/main/SKLEARN/SKLEARN_example_Diegodepab.ipynb