Deep Learning

Project

Olive Oil PROJECT REPORT

- 2022 / 2023 -

Written By:

Khawla Soltani

Mghirbi Ilef

Hdhili Samer

Abid Yacine

Balhoudi Helmi

Karoui Med Amine

I- Introduction :

The adulteration of olive oils can be detected through a chemical test. This test is costly and time-consuming. Thus, this study aims to reduce both time and costs. To achieve this, raw data was collected from olive oils.

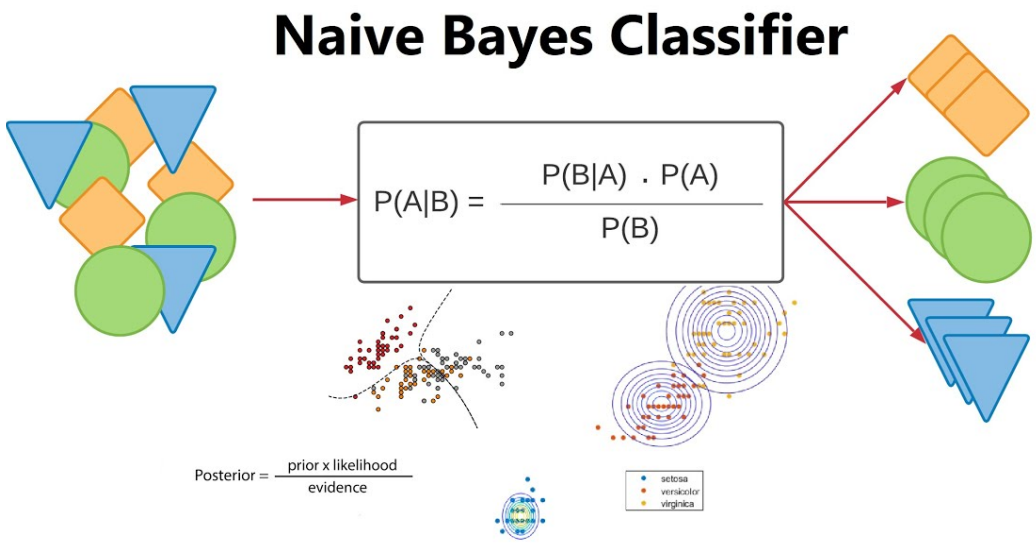
Our objective is to develop a project that takes input data related to a type of olive oil and provides the corresponding target class, or rejects the quality of the oil.

This project aims to compare different machine learning classifiers such as Naive Bayesian, k-Nearest Neighbors (k-NN), Linear Discriminant Analysis (LDA), Decision Tree, Artificial Neural Networks (ANN), and Support Vector Machine (SVM). Subsequently, the performance of these classifiers was compared based on their accuracy.

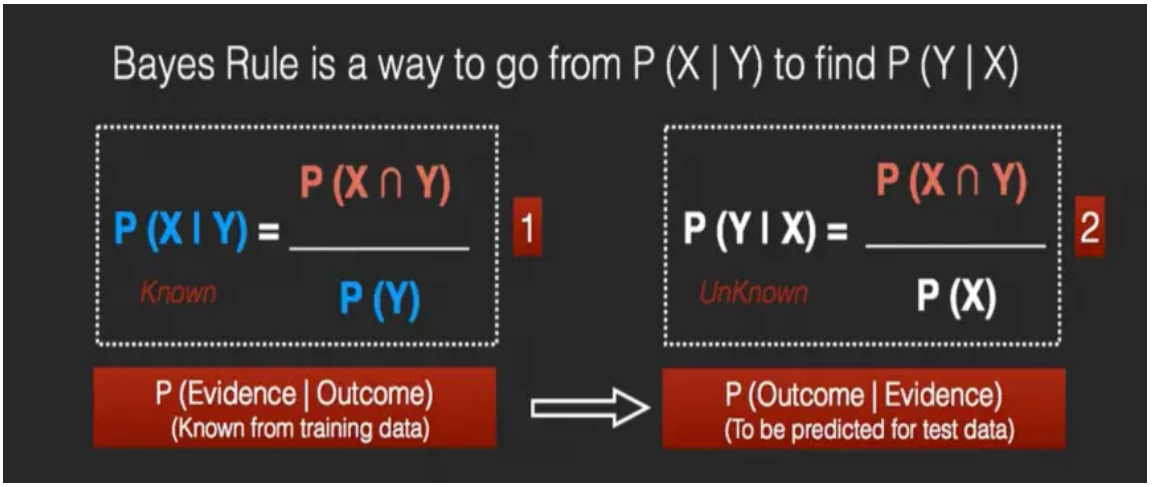
II- Naive Bayesian:

1) *Definition:*

The Naïve Bayes classifier is a supervised machine learning algorithm, which is used for classification tasks, like text classification. It is also part of a family of generative learning algorithms, meaning that it seeks to model the distribution of inputs of a given class or category. Unlike discriminative classifiers, like logistic regression, it does not learn which features are most important to differentiate between classes.



Naïve Bayes is also known as a probabilistic classifier since it is based on Bayes' Theorem. It would be difficult to explain this algorithm without explaining the basics of Bayesian statistics. This theorem, also known as Bayes' Rule, allows us to "invert" conditional probabilities. As a reminder, conditional probabilities represent the probability of an event given some other event has occurred, which is represented with the following formula:



2) *Hyperparameter:*

(**Variance smoothing** in **Naive Bayes** is also called Laplace Smoothing or Additive Smoothing) Benefit of var_smoothing is that when there is missing data or a class is not represented var_smoothing keeps model from breaking down. When feature prior probability is missing (it has never occurred before), we end up with two undesirable cases.

3) *Types of Naïve Bayes classifiers*

Gaussian Naïve Bayes (GaussianNB): This is a variant of the Naïve Bayes classifier, which is used with Gaussian distributions—i.e. normal distributions—and continuous variables. This model is fitted by finding the mean and standard deviation of each class.

Multinomial Naïve Bayes (MultinomialNB): This type of Naïve Bayes classifier assumes that the features are from multinomial distributions. This variant is useful when using discrete data, such as frequency counts, and it is typically applied within natural language processing use cases, like spam classification.

Bernoulli Naïve Bayes (BernoulliNB): This is another variant of the Naïve Bayes classifier, which is used with Boolean variables—that is, variables with two values, such as True and False or 1 and 0.

4) Advantages and disadvantages of the Naïve Bayes classifier:

Advantages:

Less complex: Compared to other classifiers, Naïve Bayes is considered a simpler classifier since the parameters are easier to estimate. As a result, it's one of the first algorithms learned within data science and machine learning courses.

Scales well: Compared to logistic regression, Naïve Bayes is considered a fast and efficient classifier that is fairly accurate when the conditional independence assumption holds. It also has low storage requirements.

Can handle high-dimensional data: Use cases, such document classification, can have a high number of dimensions, which can be difficult for other classifiers to manage.

Disadvantages:

Subject to Zero frequency: Zero frequency occurs when a categorical variable does not exist within the training set. For example, imagine that we're trying to find the maximum likelihood estimator for the word, "sir" given class "spam", but the word, "sir" doesn't exist in the training data. The probability in this case would zero, and since this classifier multiplies all the conditional probabilities together, this also means that posterior probability will be zero. To avoid this issue, laplace smoothing can be leveraged.

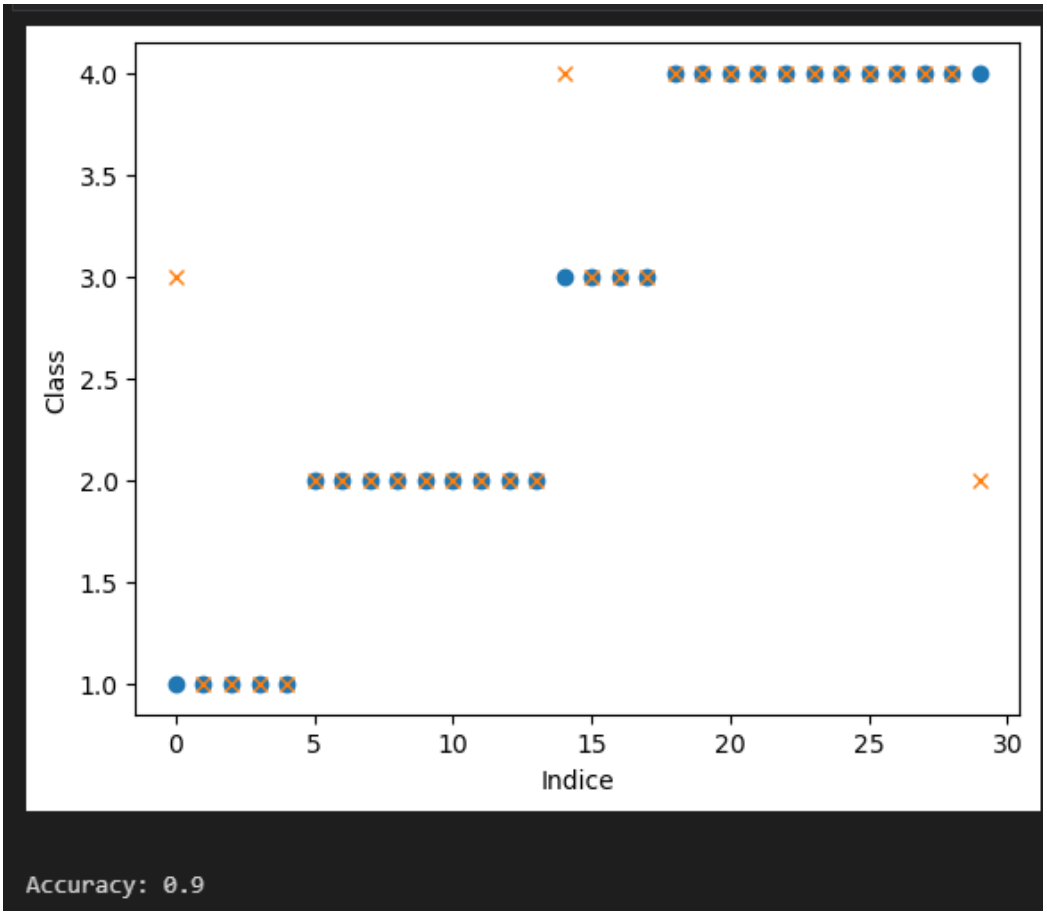
Unrealistic core assumption: While the conditional independence assumption overall performs well, the assumption does not always hold, leading to incorrect classifications.

5) Manipulation:

```
param_grid = {'var_smoothing': [1e-9, 1e-8, 1e-7]}
naive_bayes = GaussianNB()
grid_search = GridSearchCV(naive_bayes, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_nb = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_nb.predict(X_test)
```

```
Best hyperparameters:  {'var_smoothing': 1e-09}
Best accuracy score:  0.8333333333333334
```

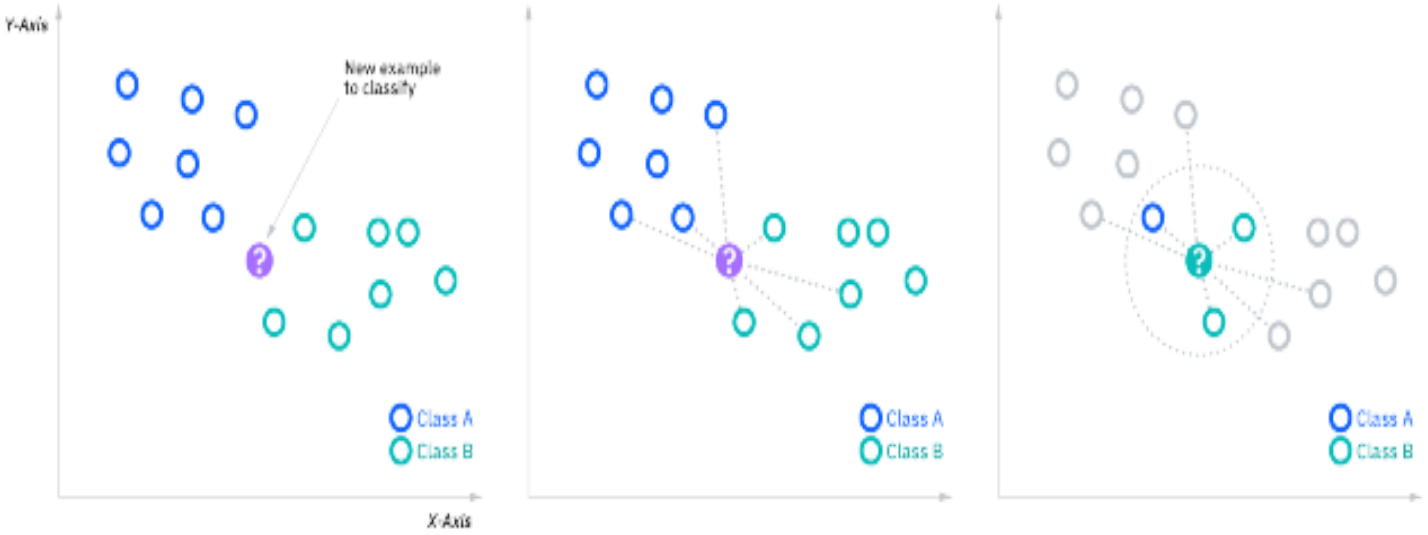
6) Result :

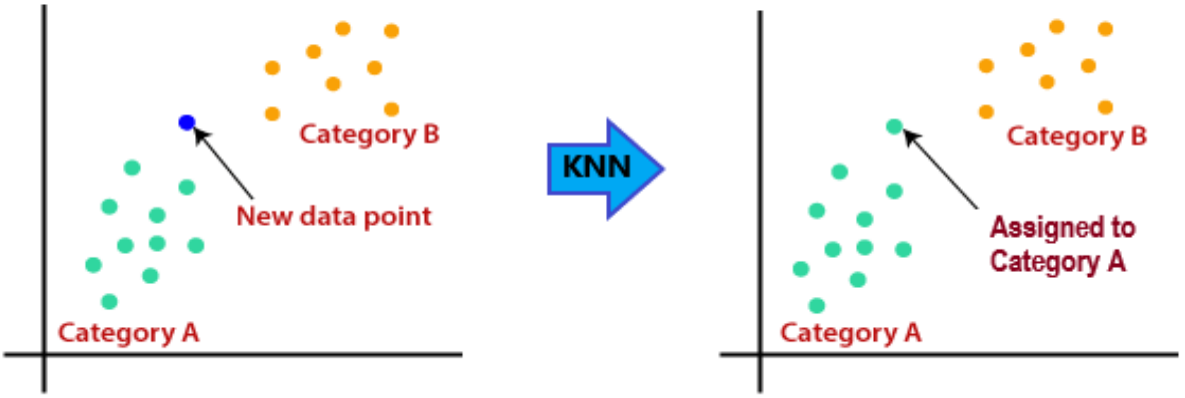


III- Nearest Neighbors:

1) *Definition:*

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another.





2) *Hyperparameters:*

Number of Neighbors (k): The parameter "k" represents the number of nearest neighbors considered for classification. A smaller value of k can make the model more sensitive to noise or outliers, while a larger value can smooth out the decision boundaries but may lead to loss of local details. It is important to choose an appropriate value of k based on the characteristics of your dataset.

3) *Advantages and disadvantages of the KNN algorithm*

Advantages:

- Easy to implement: Given the algorithm's simplicity and accuracy, it is one of the first classifiers that a new data scientist will learn.

- Adapts easily: As new training samples are added, the algorithm adjusts to account for any new data since all training data is stored into memory.

- Few hyperparameters: KNN only requires a k value and a distance metric, which is low when compared to other machine learning algorithms.

Disadvantages:

- Does not scale well: Since KNN is a lazy algorithm, it takes up more memory and data storage compared to other classifiers. This can be costly from both a time and money perspective. More memory and storage will drive up business expenses and more data can take longer to compute. While different data structures, such as Ball-Tree, have been created to address the computational inefficiencies, a different classifier may be ideal depending on the business problem.

- Curse of dimensionality: The KNN algorithm tends to fall victim to the curse of dimensionality, which means that it doesn't perform well with high-dimensional data inputs. This is sometimes also referred to as the peaking phenomenon, where after the algorithm attains the optimal number of features, additional features increases the amount of classification errors, especially when the sample size is smaller.

- Prone to overfitting: Due to the "curse of dimensionality", KNN is also more prone to overfitting. While feature selection and dimensionality reduction techniques are leveraged to prevent this from occurring, the value of k can also impact the model's behavior. Lower values of k can overfit the data, whereas higher values of k tend to "smooth out" the prediction values since it is averaging the values over a greater area, or neighborhood. However, if the value of k is too high, then it can underfit the data.

4) Manipulation:

```
param_grid = {'n_neighbors': [2, 3, 4, 5, 6, 7]}
knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_knn = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_knn.predict(X_test)
```

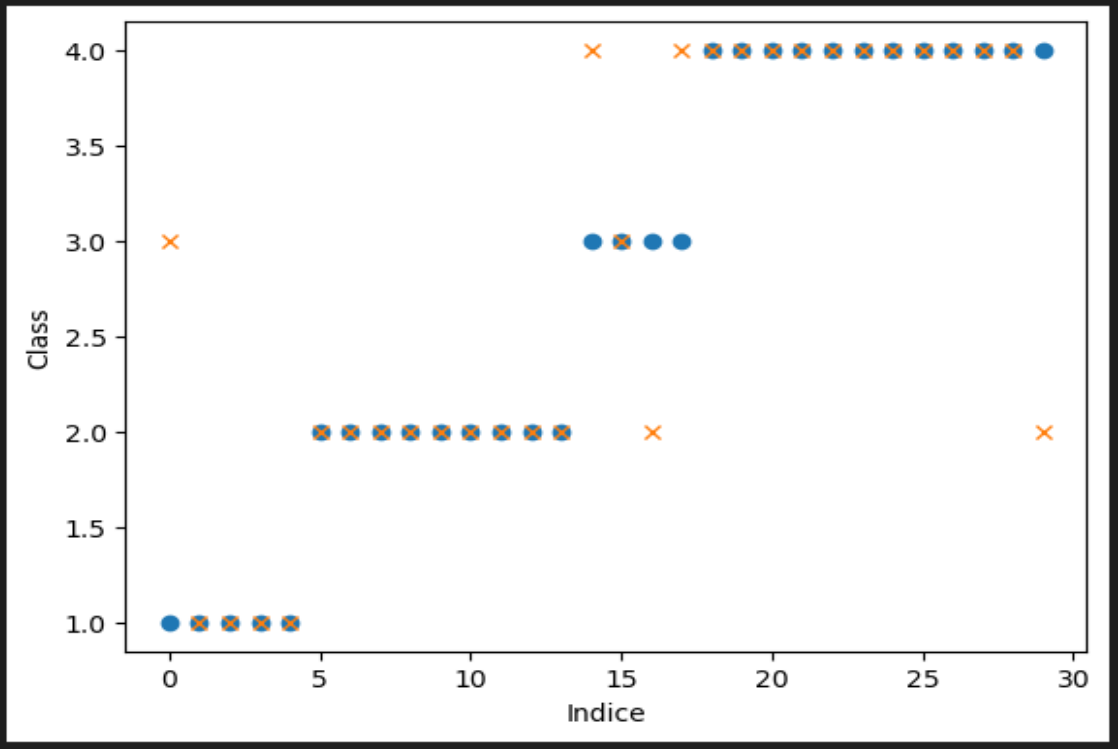
✓ 0.4s

```
Best hyperparameters:  {'n_neighbors': 3}
Best accuracy score:  0.8
```

5) Results:

```
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

✓ 0.2s



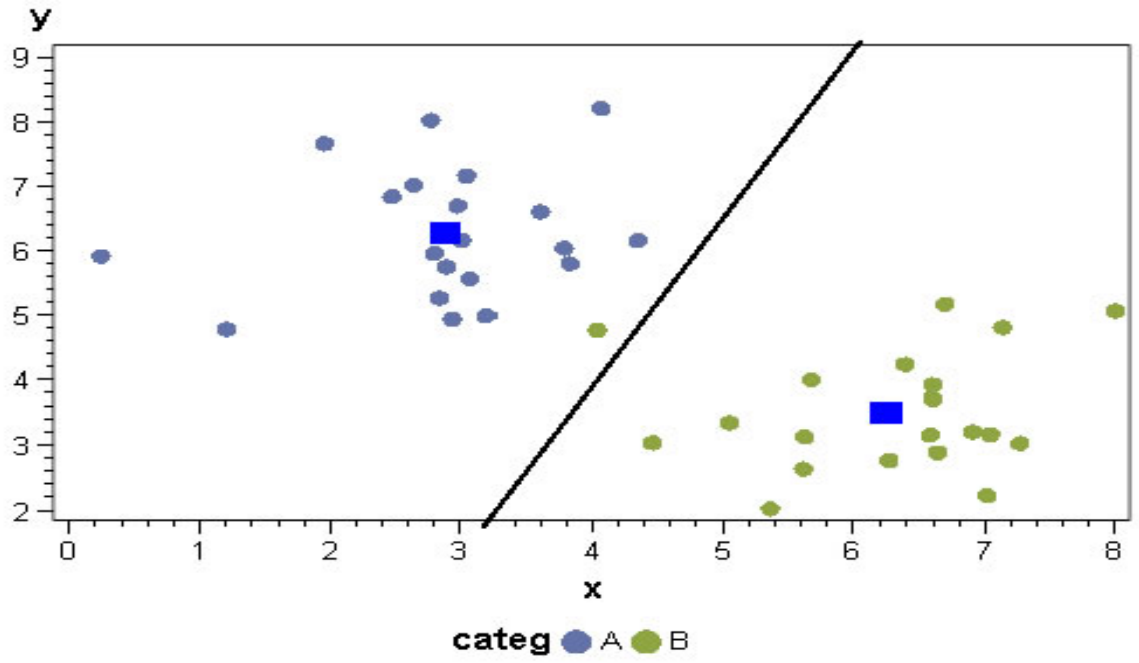

```
Accuracy: 0.8333333333333334
```

1) Definition:

Linear Discriminant Analysis (LDA) is a supervised learning algorithm used for classification tasks in machine learning. It is a technique used to find a linear combination of features that best separates the classes in a dataset.

LDA works by projecting the data onto a lower-dimensional space that maximizes the separation between the classes. It does this by finding a set of linear discriminants that maximize the ratio of between-class variance to within-class variance. In other words, it finds the directions in the feature space that best separate the different classes of data.

LDA assumes that the data has a Gaussian distribution and that the covariance matrices of the different classes are equal. It also assumes that the data is linearly separable, meaning that a linear decision boundary can accurately classify the different classes.



2) Hyperparameters:

Solver: LDA can be solved using different methods or solvers, such as "svd" (singular value decomposition) or "eigen" (eigenvalue decomposition). The choice of solver may depend on the size of the dataset and computational efficiency. The default solver is usually suitable for most cases.

3) Extension to LDA:

Linear Discriminant Analysis is a simple and effective method for classification. Because it is simple and so well understood, there are many extensions and variations to the method. Some popular extensions include:

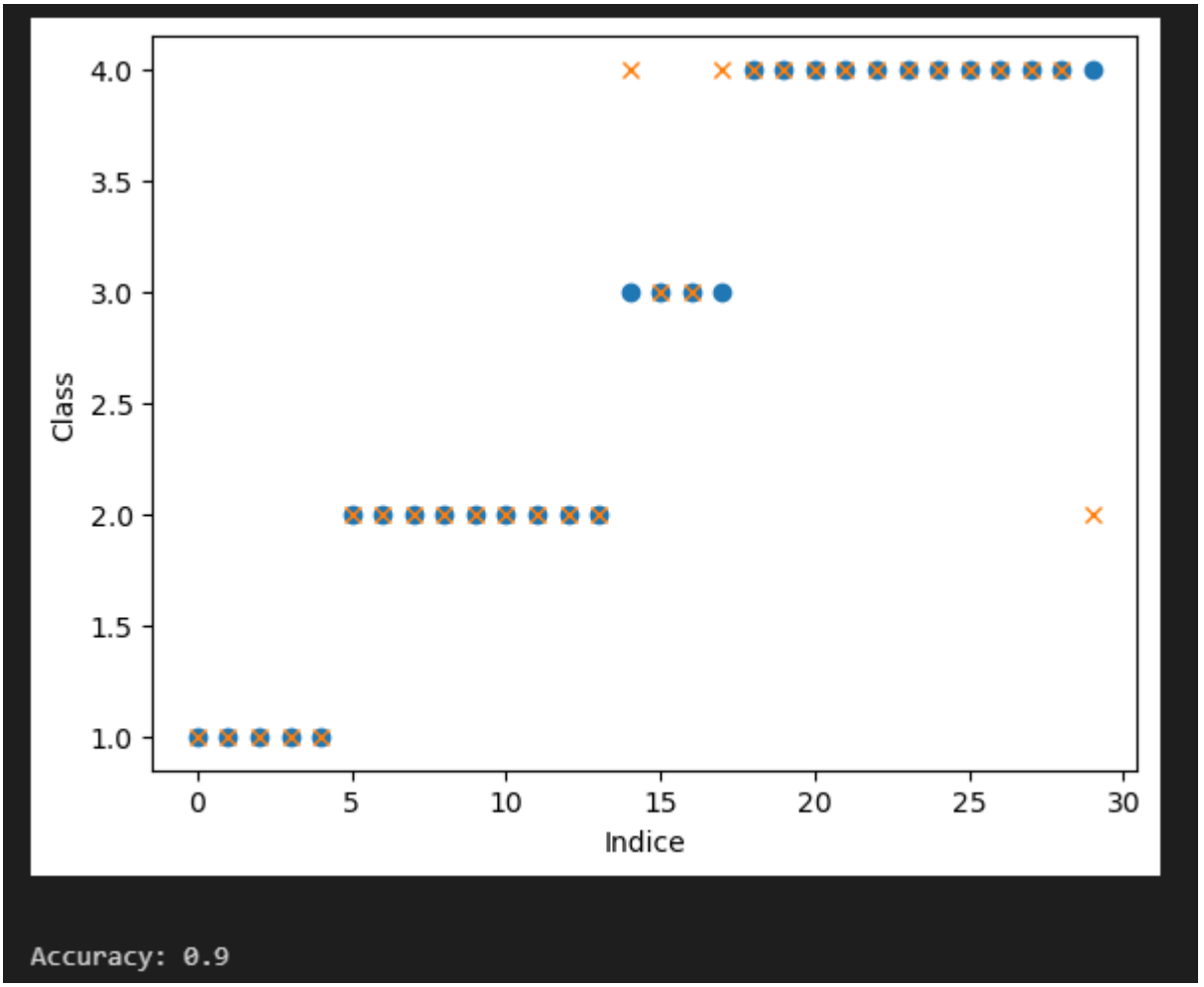
- Quadratic Discriminant Analysis (QDA): Each class uses its own estimate of variance (or covariance when there are multiple input variables).
- Flexible Discriminant Analysis (FDA): Where non-linear combinations of inputs are used such as splines.
- Regularized Discriminant Analysis (RDA): Introduces regularization into the estimate of the variance (actually covariance), moderating the influence of different variables on LDA.

4) Manipulation:

```
param_grid = {'solver': ['svd', 'lsqr']}
lda = LinearDiscriminantAnalysis()
grid_search = GridSearchCV(lda, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_lda = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_lda.predict(X_test)
```

```
Best hyperparameters:  {'solver': 'svd'}
Best accuracy score:  0.8333333333333334
```

5) Results:



Processing Data :

Processing data refers to the steps and techniques used to transform raw data into a format that can be easily analyzed, understood, and utilized for various purposes. This may include cleaning and filtering data, transforming it into a suitable structure, extracting useful information and insights, and preparing it for further analysis or visualization. Processing data is a crucial step in the data analysis pipeline, as it ensures that the data is accurate, consistent, and relevant for the intended analysis or application.

Transfer data from text files to a csv files

```
with open('C:/Users/Helmi Balhoudi/Desktop/OliveOil/OliveOil_TRAIN.txt', 'r') as infile:
    with open('C:/Users/Helmi Balhoudi/Desktop/Train_data.csv', 'w', newline='') as outfile:
        writer = csv.writer(outfile, delimiter=',')
        for line in infile:
            columns = line.strip().split('\t')
            writer.writerow(columns)

with open('C:/Users/Helmi Balhoudi/Desktop/OliveOil/OliveOil_TEST.txt', 'r') as infile:
    with open('C:/Users/Helmi Balhoudi/Desktop/Test_data.csv', 'w', newline='') as outfile:
        writer = csv.writer(outfile, delimiter=',')
        for line in infile:
            columns = line.strip().split('\t')
            writer.writerow(columns)
```

✓ 0.0s Python

Preparing Train & Test Data

```
train = pd.read_csv('C:/Users/Helmi Balhoudi/Desktop/Train_data.csv', sep='  ', header=None, engine='python')
test = pd.read_csv('C:/Users/Helmi Balhoudi/Desktop/Test_data.csv', sep='  ', header=None, engine='python')
```

✓ 0.1s Python

Split data into features & target

```
X_train = train.iloc[:, 1:]
y_train = train.iloc[:, 0]
X_test = test.iloc[:, 1:]
y_test = test.iloc[:, 0]
```

✓ 0.0s Python

Store cleaned data locally

```
X_train.to_csv('C:/Users/Helmi Balhoudi/Desktop/X_train.csv', index=False)
X_test.to_csv('C:/Users/Helmi Balhoudi/Desktop/X_test.csv', index=False)
y_test.to_csv('C:/Users/Helmi Balhoudi/Desktop/y_test.csv', index=False)
```

✓ 0.0s Python

Display Data

X_train

✓ 0.1s

Pytho

	1	2	3	4	5	6	7	8	9	10	...	561	562
0	-0.611375	-0.610586	-0.606557	-0.601132	-0.594315	-0.585762	-0.577419	-0.570175	-0.563285	-0.557407	...	-0.979553	-0.980385
1	-0.615392	-0.613729	-0.609228	-0.604315	-0.598768	-0.590507	-0.581617	-0.572926	-0.565374	-0.559907	...	-0.979210	-0.979335
2	-0.611999	-0.610500	-0.606374	-0.600445	-0.593084	-0.585245	-0.577118	-0.568827	-0.561596	-0.556093	...	-0.979514	-0.979663
3	-0.622784	-0.622222	-0.619049	-0.613251	-0.605889	-0.597508	-0.589047	-0.580697	-0.572822	-0.566727	...	-0.968819	-0.969902
4	-0.621793	-0.621272	-0.617298	-0.612074	-0.605472	-0.597607	-0.589165	-0.581424	-0.574620	-0.568411	...	-0.977420	-0.977831
5	-0.600229	-0.599577	-0.596144	-0.590664	-0.583358	-0.574398	-0.565483	-0.557200	-0.549211	-0.543827	...	-0.986319	-0.987329
6	-0.615430	-0.614164	-0.610041	-0.603962	-0.596794	-0.588885	-0.579624	-0.571410	-0.564899	-0.559462	...	-0.978359	-0.978609
7	-0.614015	-0.612369	-0.608472	-0.602971	-0.596025	-0.588022	-0.579095	-0.570355	-0.562975	-0.557196	...	-0.977127	-0.977156
8	-0.596650	-0.595023	-0.591794	-0.586665	-0.579232	-0.571202	-0.562909	-0.554685	-0.547703	-0.541407	...	-0.986295	-0.986149
9	-0.620747	-0.620111	-0.616434	-0.611036	-0.603726	-0.595674	-0.587556	-0.579830	-0.573194	-0.566914	...	-0.975419	-0.975879
10	-0.611314	-0.610259	-0.607057	-0.601289	-0.593749	-0.585300	-0.576988	-0.569588	-0.562598	-0.556409	...	-0.977966	-0.978579
11	-0.622280	-0.620847	-0.617238	-0.612637	-0.606484	-0.598921	-0.590348	-0.582025	-0.574668	-0.568545	...	-0.976452	-0.976473
12	-0.610725	-0.610314	-0.606940	-0.601189	-0.593774	-0.585030	-0.577836	-0.570249	-0.562370	-0.555958	...	-0.976564	-0.976903

y_train

✓ 0.0s

Output exceeds the size limit. Open the full output data in a text editor

```
0     1.0
1     1.0
2     1.0
3     1.0
4     1.0
5     2.0
6     2.0
7     2.0
8     2.0
9     2.0
10    2.0
11    2.0
12    2.0
13    3.0
14    3.0
15    3.0
16    3.0
17    4.0
18    4.0
19    4.0
20    4.0
21    4.0
22    4.0
23    4.0
```

```
X_test
```

✓ 0.0s

	1	2	3	4	5	6	7	8	9	10	...	561	562
0	-0.621957	-0.620677	-0.616684	-0.611012	-0.603769	-0.595267	-0.586477	-0.577899	-0.570147	-0.564274	...	-0.966336	-0.967106
1	-0.618925	-0.618361	-0.614664	-0.608693	-0.601446	-0.593516	-0.585786	-0.577708	-0.569952	-0.563845	...	-0.971519	-0.971990
2	-0.618169	-0.616704	-0.612678	-0.607578	-0.600574	-0.592209	-0.584307	-0.576085	-0.567684	-0.561432	...	-0.976493	-0.976940
3	-0.613445	-0.611719	-0.607739	-0.602477	-0.595154	-0.587271	-0.579135	-0.570903	-0.563252	-0.557242	...	-0.976533	-0.977072
4	-0.616338	-0.615301	-0.611505	-0.605503	-0.597522	-0.589108	-0.580610	-0.572432	-0.565709	-0.559617	...	-0.974940	-0.975242
5	-0.618782	-0.617716	-0.613997	-0.608234	-0.601079	-0.593254	-0.585160	-0.577374	-0.569800	-0.563630	...	-0.974467	-0.974664
6	-0.618084	-0.615795	-0.611206	-0.605597	-0.598689	-0.591469	-0.583601	-0.574438	-0.567063	-0.561506	...	-0.974884	-0.975553
7	-0.611166	-0.610133	-0.606317	-0.600795	-0.593912	-0.585110	-0.576604	-0.569136	-0.562062	-0.555740	...	-0.981540	-0.981895
8	-0.611302	-0.610611	-0.607795	-0.601815	-0.594525	-0.586312	-0.577069	-0.568377	-0.561396	-0.555672	...	-0.977748	-0.978070
9	-0.622387	-0.621123	-0.617521	-0.612813	-0.606186	-0.598040	-0.590471	-0.582751	-0.574777	-0.568147	...	-0.977619	-0.978230
10	-0.611277	-0.610722	-0.606220	-0.599433	-0.592361	-0.584520	-0.576297	-0.567895	-0.560001	-0.554045	...	-0.978730	-0.979315
11	-0.617842	-0.616531	-0.611362	-0.605863	-0.599935	-0.592597	-0.584316	-0.576853	-0.569044	-0.562257	...	-0.983337	-0.984195
12	-0.621060	-0.620078	-0.615612	-0.609895	-0.602286	-0.594615	-0.587167	-0.578923	-0.571076	-0.564738	...	-0.974408	-0.974927
13	-0.615062	-0.613599	-0.609677	-0.604425	-0.597571	-0.589679	-0.581252	-0.573410	-0.565802	-0.559426	...	-0.980443	-0.980485
14	-0.611671	-0.610023	-0.606539	-0.600833	-0.593155	-0.584527	-0.576334	-0.568382	-0.560159	-0.553506	...	-0.969864	-0.970387
15	-0.601164	-0.601228	-0.598055	-0.592769	-0.586128	-0.578572	-0.570070	-0.564010	-0.557645	-0.550484	...	-0.984016	-0.984764
16	-0.582303	-0.581097	-0.576991	-0.570507	-0.563693	-0.556130	-0.547596	-0.539889	-0.532386	-0.527216	...	-0.995655	-0.996061

```
y_test
```

✓ 0.0s

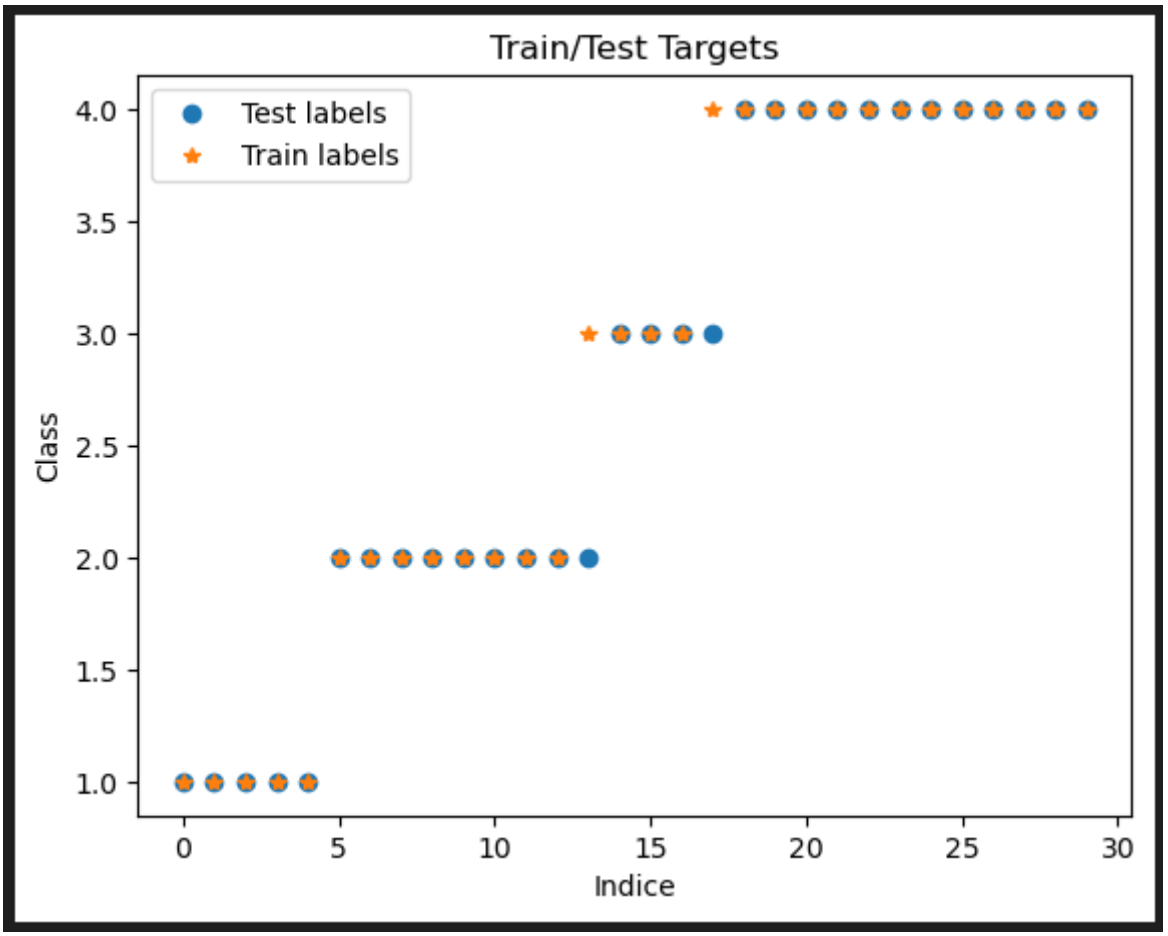
Output exceeds the size limit. Open the full output data in a text editor

```
0     1.0
1     1.0
2     1.0
3     1.0
4     1.0
5     2.0
6     2.0
7     2.0
8     2.0
9     2.0
10    2.0
11    2.0
12    2.0
13    2.0
14    3.0
15    3.0
16    3.0
17    3.0
18    4.0
19    4.0
20    4.0
21    4.0
22    4.0
23    4.0
```

Plotting Data

```
plt.plot(range(len(y_test)), y_test, 'o', label='Test labels')
plt.plot(range(len(y_train)), y_train, '*', label='Train labels')
plt.legend(loc='upper right', fontsize='medium')
plt.title("Train/Test Targets")
plt.legend()
plt.xlabel('Indice')
plt.ylabel('Class')
plt.show()
```

✓ 0.2s



Decision Tree :

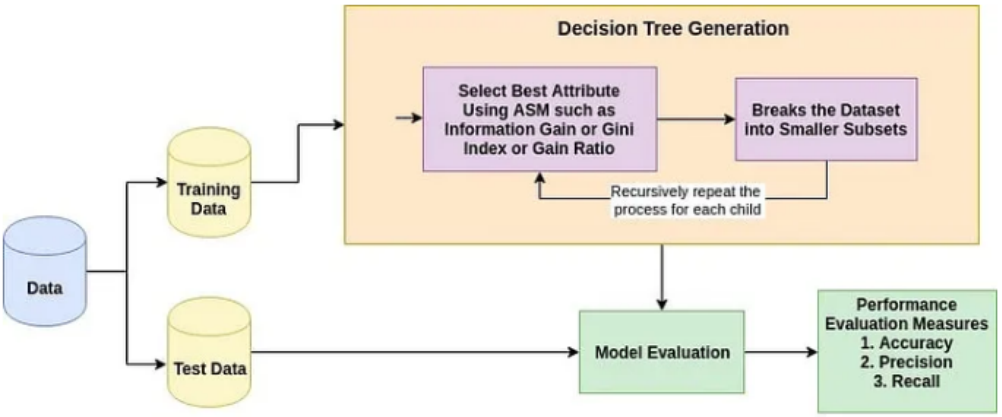
1) *Définition:*

- In this kind of decision trees, the decision variable is **categorical**. The above decision tree is an example of a classification decision tree.
- Such a tree is built through a process known as **binary recursive partitioning**. This is an iterative process of **splitting the data into partitions**, and then splitting it up further on each of the branches.
- Decisions are based on some conditions. Decisions made can be easily explained.
- Decision trees can handle high dimensional data with good accuracy.
- The tree contains **decision nodes** and **leaf nodes**.

→**Decision nodes** are those nodes that represent the value of the input variable(x). It has two or more than two branches.
→**Leaf nodes** contain the decision or the output variable(y).

2) *Implementation:*

1. Root Node: It represents the entire population or sample and this further gets divided into two or more homogeneous sets.
2. Splitting: It is a process of dividing a node into two or more sub-nodes.
3. Decision Node: When a sub-node splits into further sub-nodes, then it is called decision node.
4. Leaf/ Terminal Node: Nodes that do not split are called Leaf or Terminal nodes.
5. Pruning: When we remove sub-nodes of a decision node, this process is called pruning. You can say the opposite process of splitting.
6. Branch / Sub-Tree: A subsection of an entire tree is called a branch or sub-tree.
7. Parent and Child Node: A node, which is divided into sub-nodes is called parent node of sub-nodes whereas sub-nodes are the child of parent node.

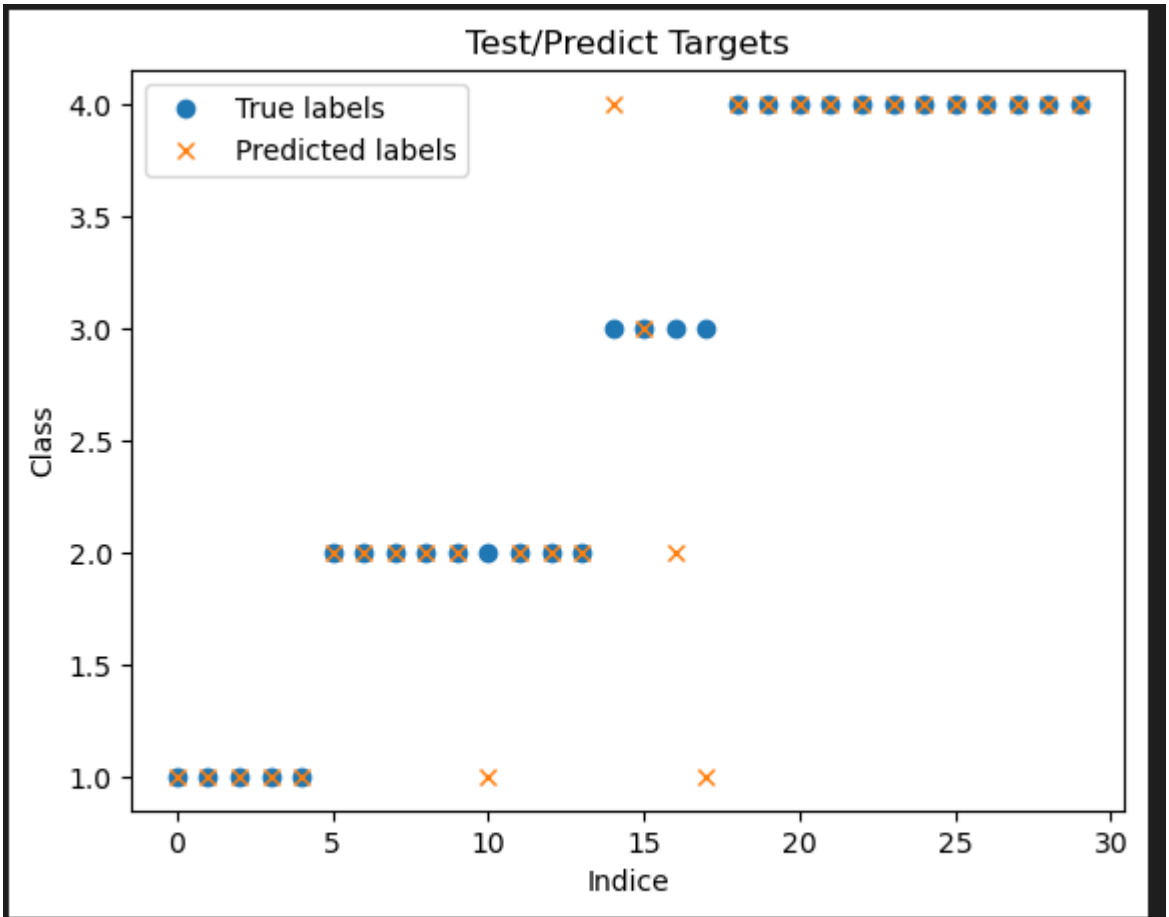


Manipulation :

```
param_grid = {'criterion': ['gini', 'entropy'], 'max_depth': [2, 5, 10]}
dt = DecisionTreeClassifier()
grid_search = GridSearchCV(dt, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_dt = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_dt.predict(X_test)
```

✓ 0.4s

```
Best hyperparameters:  {'criterion': 'entropy', 'max_depth': 5}
Best accuracy score:  0.8333333333333334
```



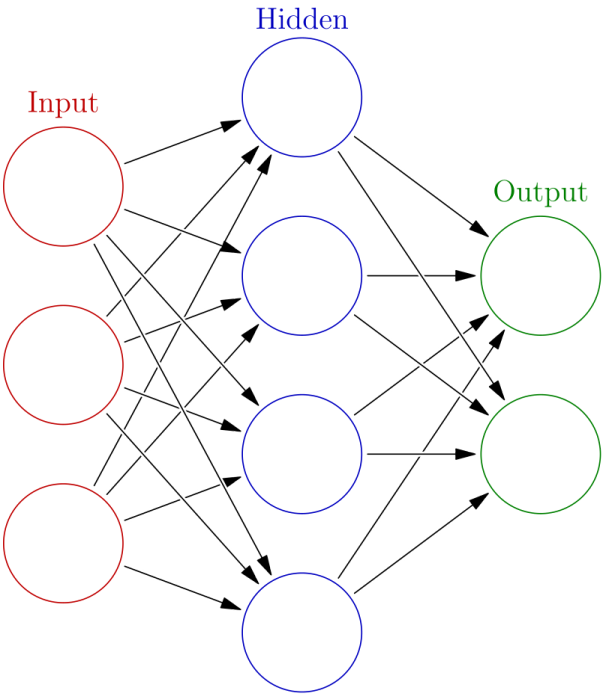

```
Accuracy: 0.8666666666666667
```

Artificial Neural Networks :

Définition:

Artificial neural networks (ANNs), usually simply called neural networks (NNs) or neural nets,[1] are computing systems inspired by the biological neural networks that constitute animal brains.[2] An ANN is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. An artificial neuron receives signals then processes them and can signal neurons connected to it. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold.

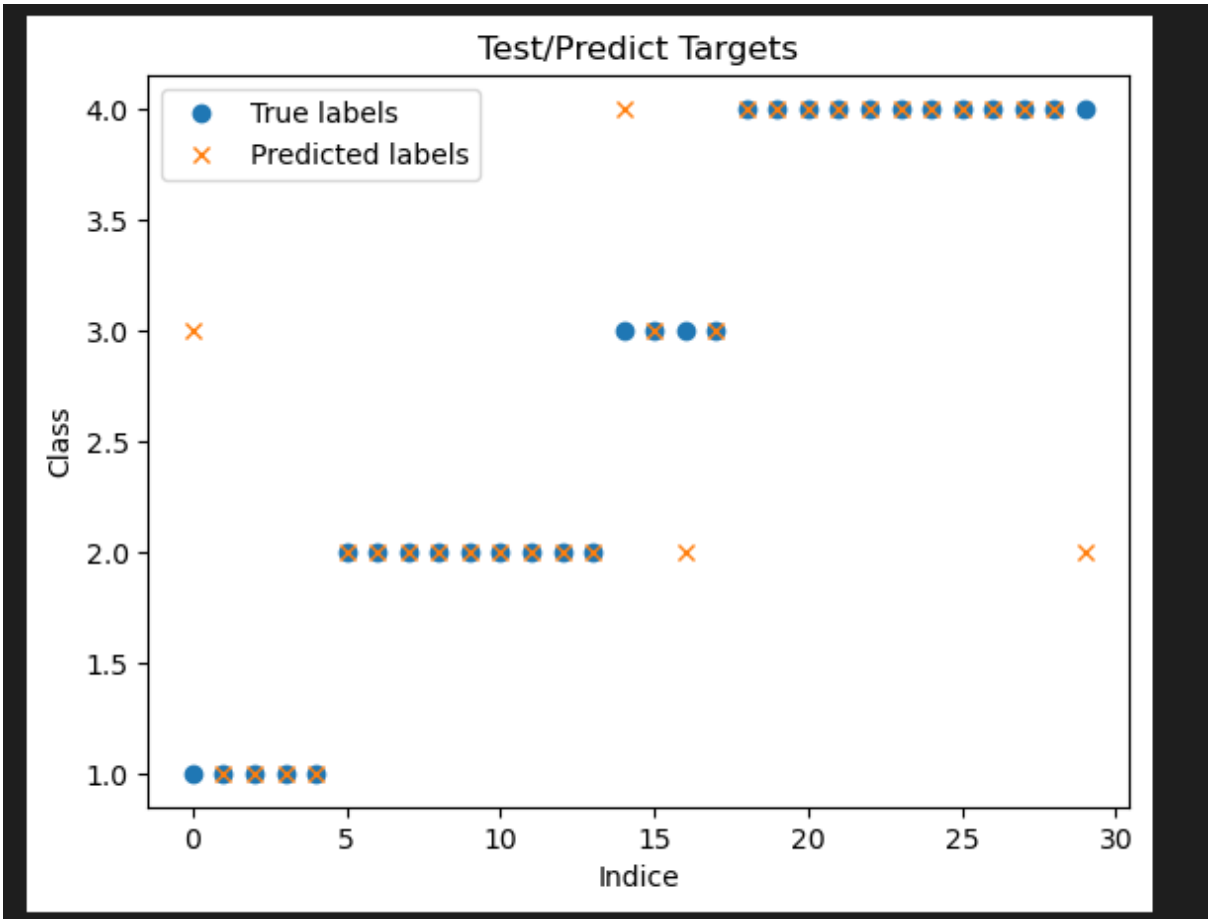
Implementation



Manipulation :

```
param_grid = {'hidden_layer_sizes': [(100,), (50,50)], 'activation': ['relu', 'tanh'], 'max_iter': [100, 500, 1000]}
mlp = MLPClassifier()
grid_search = GridSearchCV(mlp, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_mlp = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_mlp.predict(X_test)
```

```
Best hyperparameters:  {'activation': 'relu', 'hidden_layer_sizes': (100,), 'max_iter': 100}
Best accuracy score:  0.9666666666666666
```




```
Accuracy: 0.8666666666666667
```

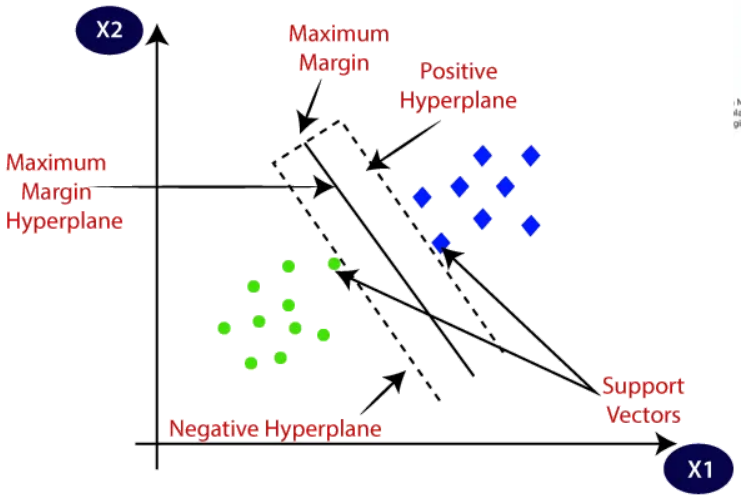
Support Vector Machine (SVM) :

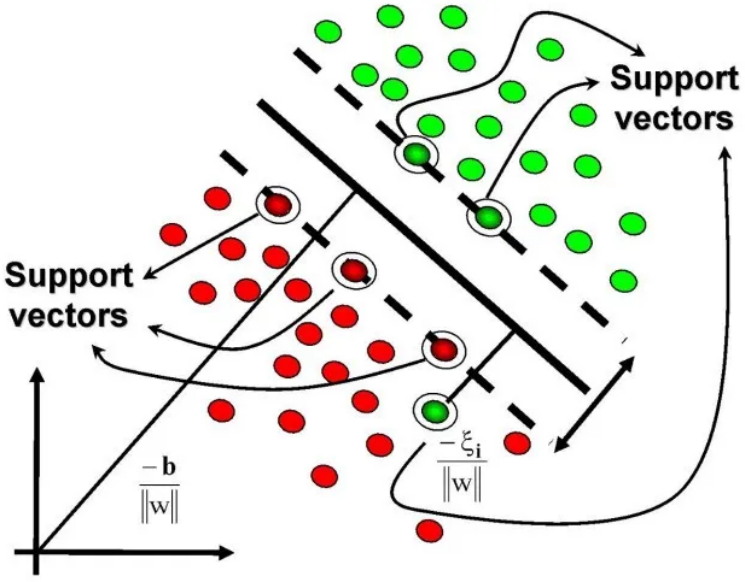
Définition:

Support vector machines, so called as SVM, is a supervised learning algorithm which can be used for classification and regression problems as support vector classification (SVC) and support vector regression (SVR). It is used for smaller datasets as it takes too long to process. In this set, we will be focusing on SVC.

Implementation

- ***Support Vector Points:*** The points closest to the hyperplane are called as the support vector points.
- ***Margins:*** The distance of the vectors from the hyperplane are called the margins.
- ***Kernel***: Kernel is a function that is used to map lower-dimensional data points into higher-dimensional data points. As SVR performs linear regression in a higher dimension, this function is crucial. There are many types of kernel such as **Polynomial Kernel, Gaussian Kernel, Sigmoid Kernel**, etc.
- ***Hyper Plane***: In Support Vector Machine, a hyperplane is a line used to separate two data classes in a higher dimension than the actual dimension. In SVR, a hyperplane is a line that is used to predict continuous value.
- ***Boundary Line***: Two parallel lines drawn to the two sides of Support Vector with the **error threshold** value, ***ε***(epsilon) are known as the boundary line. Boundary line situated at the positive region is known as the Positive **Hyperplane** and the boundary line situated at the negative region is known as the Negative **Hyperplane** .These lines create a margin between the data points.





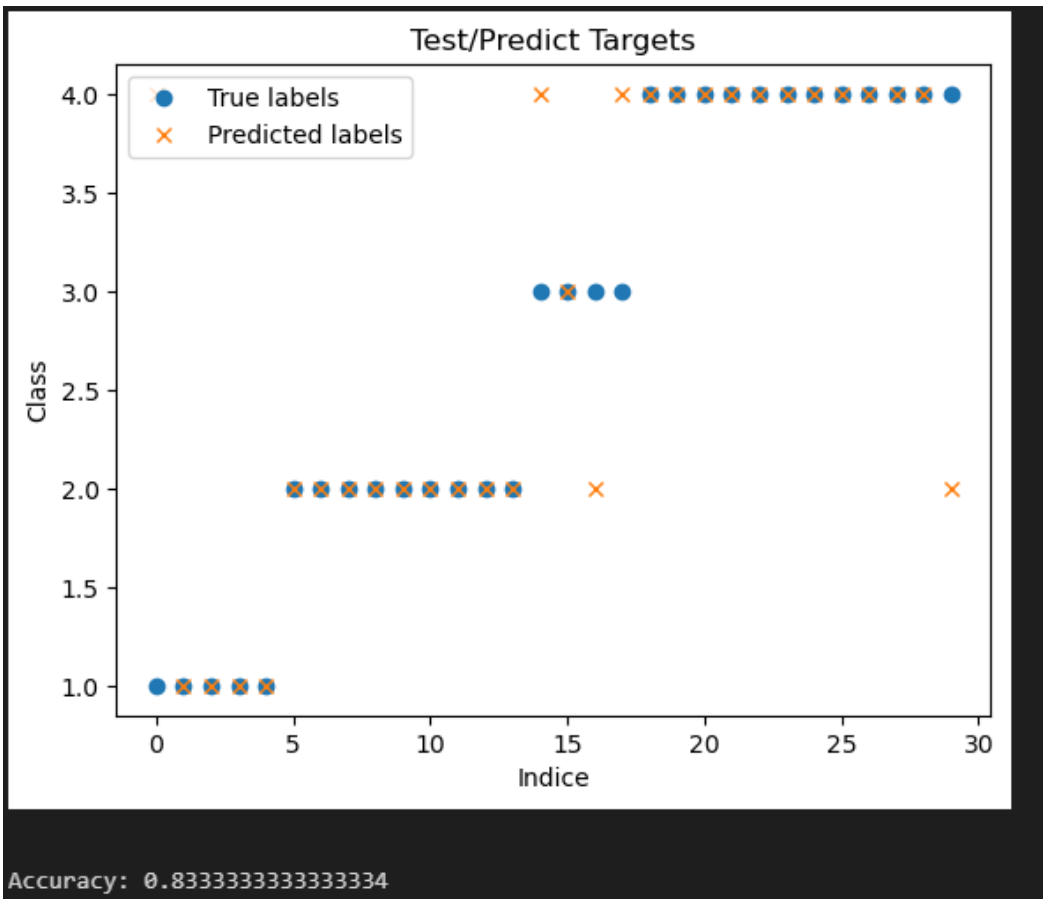
Support Vector machines create n-dimensional hyperplanes to segregate the data based on parameters. If the data has 2 variables then it's a line, 3, it's a plane and so on.

Manipulation :

```
param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'poly', 'rbf', 'sigmoid'], 'gamma': ['scale', 'auto']}
svm = SVC()
grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)
best_svm = grid_search.best_estimator_
print("Best hyperparameters: ", grid_search.best_params_)
print("Best accuracy score: ", grid_search.best_score_)
y_pred = best_svm.predict(X_test)
```

✓ 0.2s

```
Best hyperparameters:  {'C': 10, 'gamma': 'scale', 'kernel': 'rbf'}
Best accuracy score:  0.9333333333333333
```



Conclusion :

In conclusion, this research project aimed to address the challenges associated with the detection of olive oil adulteration, which typically involves expensive and time-consuming chemical tests. The primary goal was to develop a cost-effective and time-efficient solution.

To achieve this objective, a comprehensive dataset of raw olive oil samples was collected. The project focused on developing a classification system capable of accurately predicting the quality of a given olive oil type based on input data. In cases where an oil sample did not meet the desired quality standards, it would be rejected.

The project compared the performance of various machine learning classifiers, including Naïve Bayesian, k-Nearest Neighbors (k-NN), Linear Discriminant Analysis (LDA), Decision Tree, Artificial Neural Networks (ANN), and Support Vector Machine (SVM). These classifiers were assessed based on their precision, which allowed for a thorough evaluation of their effectiveness in detecting adulteration.

The findings of this study are instrumental in identifying the most robust classifier for olive oil adulteration detection. By implementing the selected classifier, significant advancements can be made in reducing both the time and financial resources

required for quality control processes. This outcome has substantial implications for the olive oil industry, benefitting producers and consumers alike.

In conclusion, this research project provides invaluable insights into the application of machine learning techniques for the detection of olive oil adulteration. The results have the potential to revolutionize quality control practices, resulting in improved product integrity, enhanced consumer confidence, and cost savings across the olive oil supply chain.