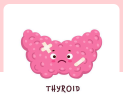


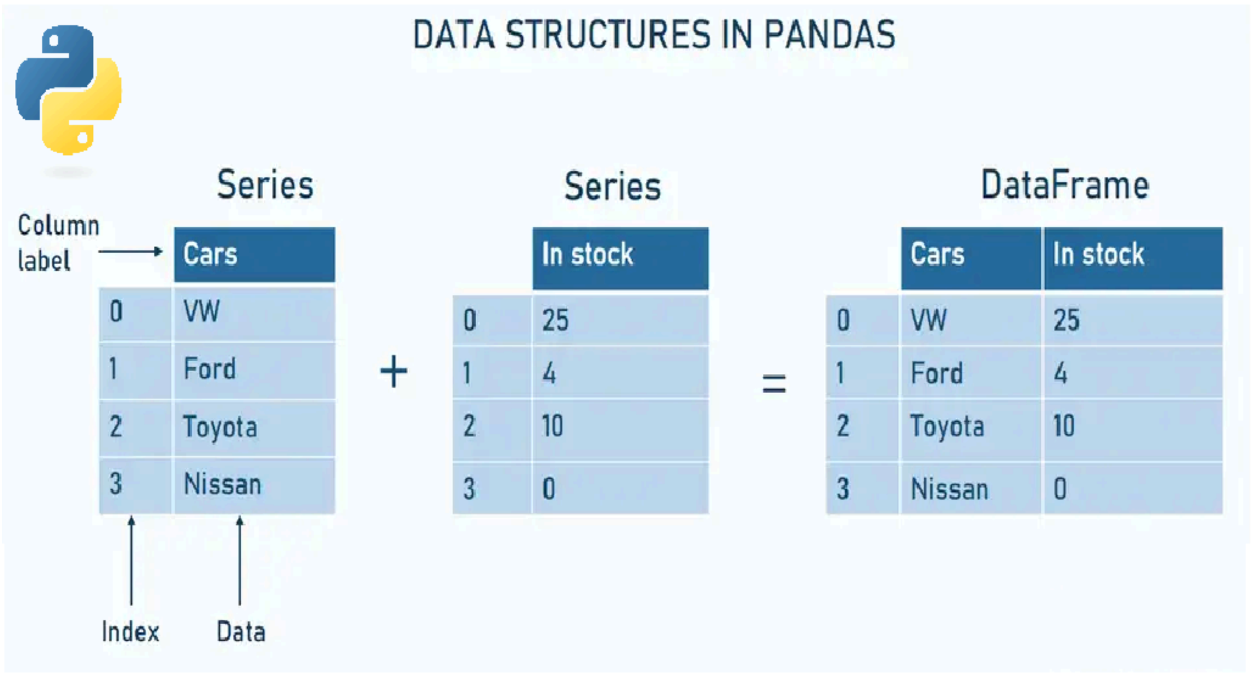
Thyroid Disease Classification

using classification models and ensemble learning to predict the likelihood of thyroid diseases.

1. Importing the Required Libraries/Modules

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

The libraries we used are Pandas and NumPy , the former for reading in the dataset and using it in the form of a Dataframe (which is the Pandas data type that organizes data in rows and columns). NumPy deals with data in the form of Arrays.



We also imported matplotlib.pyplot and Seaborn for visualization purposes.

---

2. Exploratory Data Analysis (EDA)

using Pandas, we took a look at the data using methods like:

▼ **head()**

A method used to show the **first** rows of the dataset (the default value is 5 rows)

▼ **tail()**

A method used to show the **last** rows of the dataset (the default value is 5 rows)

▼ **info()**

A method that prints information about the DataFrame, including the number of columns, column labels, column data types, range index, and the number of data points in each column (non-null values).

```
dataset.info()
```

▼ describe()

A method that returns a description of the data in the DataFrame.

▼ shape

Returns the number of rows and columns of the dataset in the form of a tuple.

▼ isnull()

A method that checks if a value in the dataframe is missing (null). This method is used interchangeably with “isna()”, both performing the same task.

Used with “sum()” to get the total number of missing values.

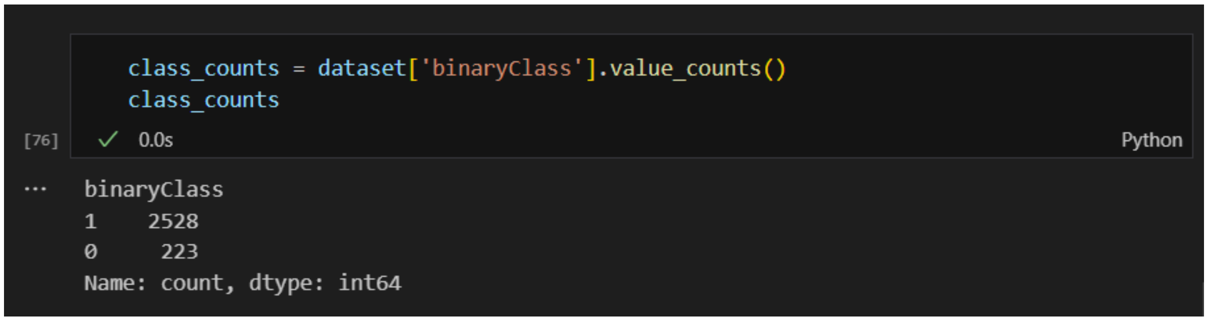
▼ nunique()

A method that returns the number of unique values(classes) in each column.

▼ duplicated()

A method that is used to check if each row in the dataframe is duplicated or not. Also used with “sum()” to get the total number of duplicated rows.

one thing we noticed is that the dataset appears to be imbalanced:


```
class_counts = dataset['binaryClass'].value_counts()
class_counts
```
[76] ✓ 0.0s Python

```
binaryClass
1    2528
0     223
Name: count, dtype: int64
```

where the majority of the instances belong to class 1

---

3. Data Preprocessing

using Pandas, we started addressing some of the problems in the dataset by handling missing values, duplicated rows, and more using methods like:

▼ dropna()

A method used to delete rows that include missing values.

▼ fillna()

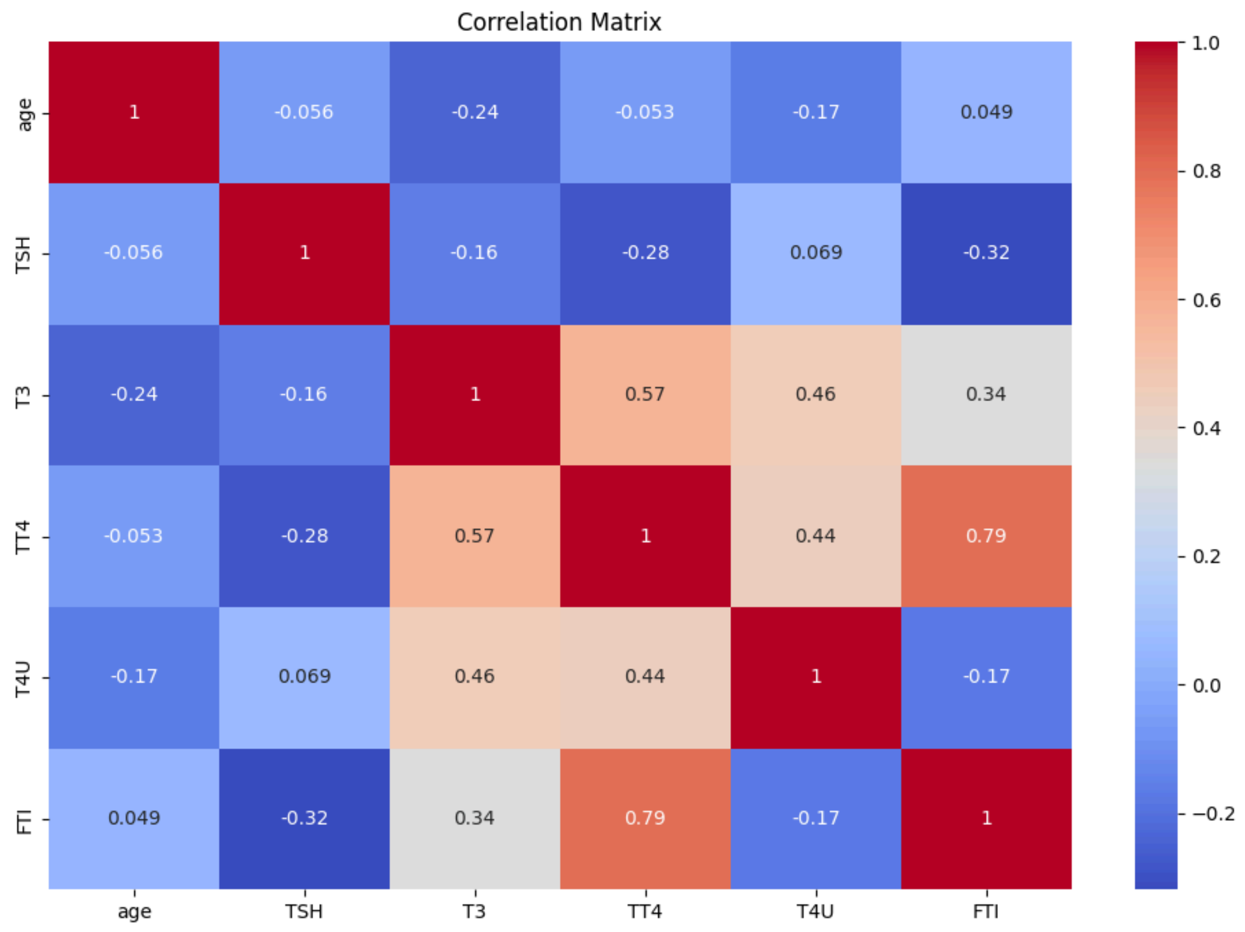
A method used to fill the missing values with a predefined value.

▼ drop_duplicates()

A method used to delete duplicated rows.

▼ replace()

A method used to replace one value with another (we used it to replace binary classes like “N” and ”P” with 0 and 1) we also used seaborn to apply a correlation matrix to the features of the dataset in order to check for the presence of highly correlated columns.



4. Splitting the dataset into train and test:

first we split the data into x and y, where y is the target (output variable) and x is the features (input variables)

```
x= dataset.drop("binaryClass",axis=1)
y=dataset["binaryClass"]
```

then, we use import train_test_split() from the model_selection module in sklearn

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

lastly, we use train_test_split() to further split x and y into x_train , x_test , y_trian , and y_test

```
x_train,x_test,y_train,y_test= train_test_split(x,y,test_size=0.2,random_state=50)
```

There are 2 parameters to note here:

▼ test_size = 0.2

This means that the percentage of test data is 20% of the entire data (and, as a result, the percentage of train data is 80%).

▼ random_state = 50

This ensures that we get the same rows in test and train across different executions. 50 is the chosen random seed.

now, the x_train is 2200 rows and 20 column, while the x_test is 551 rows and 20 columns.

---

5. Applying SMOTE to balance the dataset

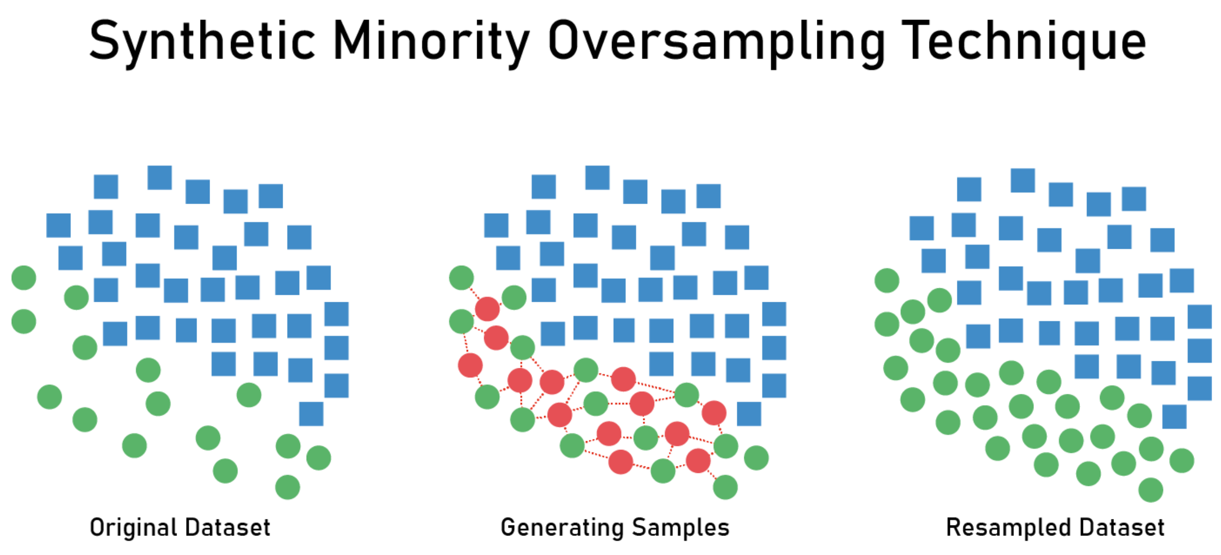
We start by importing SMOTE from the over_sampling module in imblearn

```
from imblearn.over_sampling import SMOTE
```

SMOTE (**Synthetic Minority Oversampling Technique**) is a technique that increases the number of instances in the minority class using oversampling techniques.

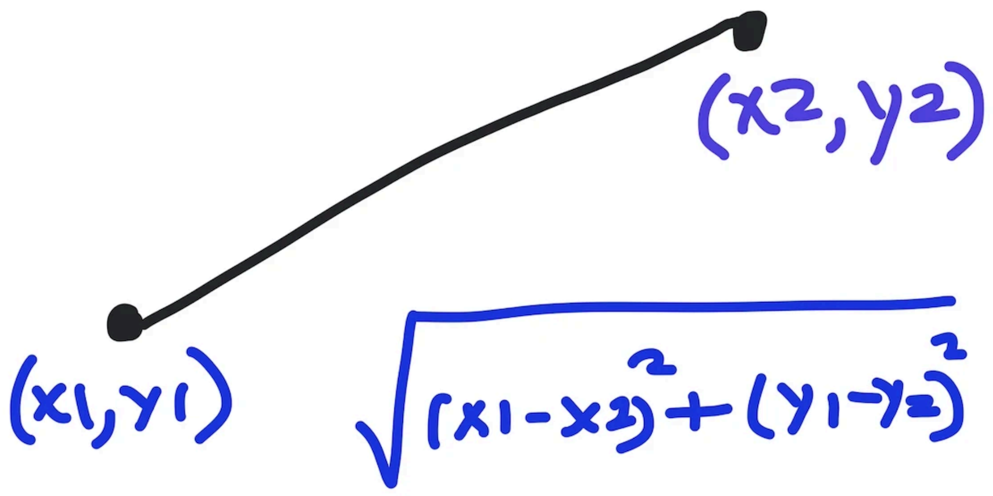
▼ How SMOTE works:

SMOTE generates new (synthetic) data instances instead of duplicating existing ones. It does this by creating new data points that lie between existing minority class instances.



The steps:

1. For each minority class instance, SMOTE identifies its k-nearest neighbours (typically k = 5) using a distance metric like Euclidean distance



2. A random neighbour is selected, and a new data point is generated along the line that connects the original instance and the selected neighbour.
3. These steps are repeated until the desired level of over-sampling is achieved.

Applying SMOTE:

```
sm = SMOTE(random_state = 42)
x_train_res, y_train_res = sm.fit_resample(x_train, y_train)
sm2 = SMOTE(random_state= 40)
x_test_res , y_test_res = sm2.fit_resample(x_test , y_test)
```

we took and instance of SMOTE and applied it on the training data then did the same for the test data separately to avoid data leakage.

The new size of x_train is 3538 rows and still 20 columns, while x_test is 1518 rows and 20 columns.

---

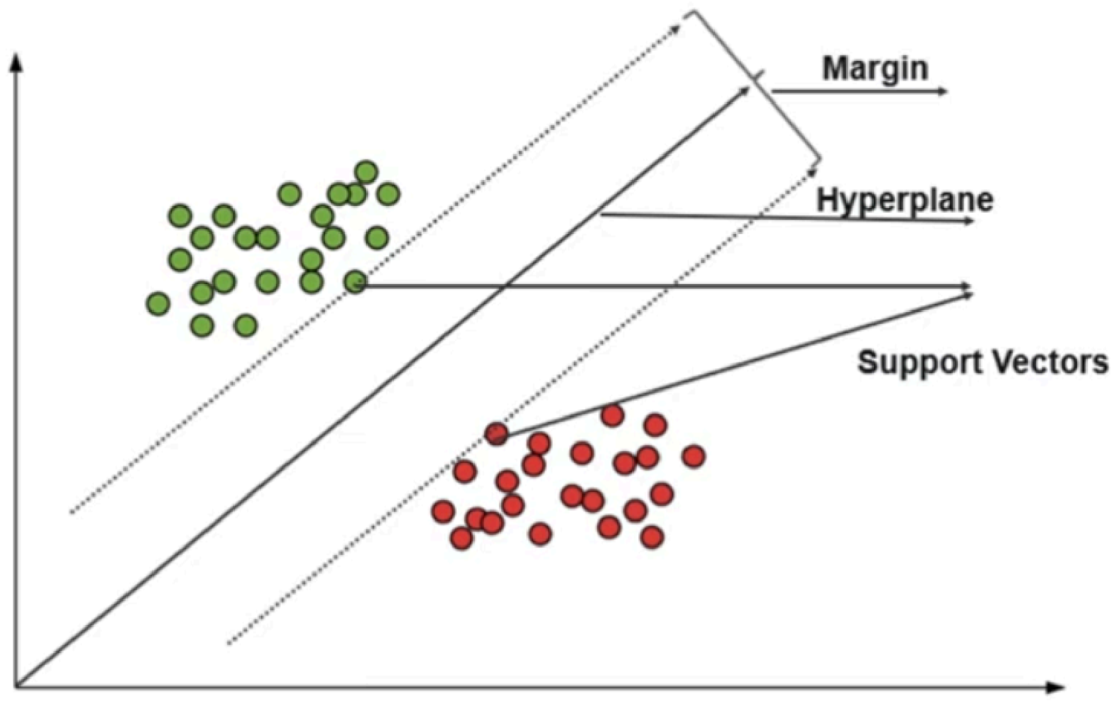
6. Importing the required models to build the ensemble learning model

```
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold
```

The models we used are:

▼ SVM (Support Vector Machine)

Support vector machine is a supervised learning model that is typically used with classification problems.



The way it works is by trying to find the best hyperplane that maximizes the distance between the instances belonging to each class. Meaning that it aims to get the maximum separation between two classes.

This model relies on a more advanced version of the cost function used with logistic regression:

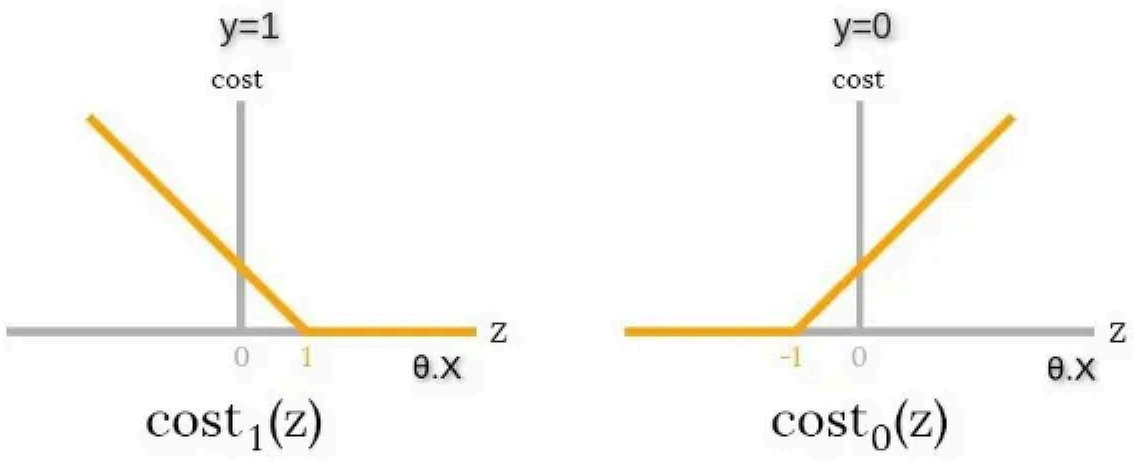
$$Cost(h_\theta(x), y) = \begin{cases} max(0, 1 - \theta^T x) & \text{if } y = 1 \\ max(0, 1 + \theta^T x) & \text{if } y = 0 \end{cases}$$

$$J(\theta) = \sum_{i=1}^{m} y^{(i)} Cost_1(\theta^T(x^{(i)}) + (1 - y^{(i)})Cost_0(\theta^T(x^{(i)})$$

$$J(\theta) = \sum_{i=1}^{m} y^{(i)} max(0, 1 - \theta^T x) + (1 - y^{(i)})max(0, 1 + \theta^T x)$$

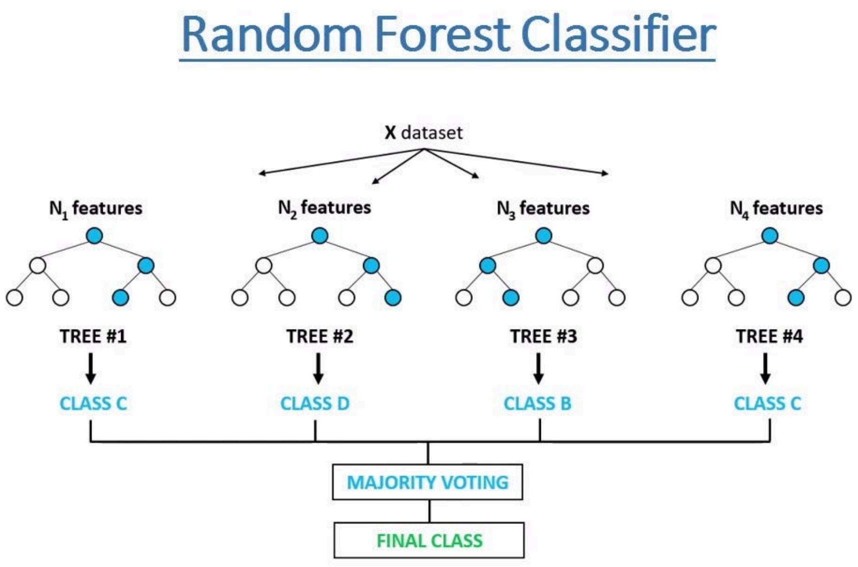
$$m = number\ of\ samples$$

$$\text{cost} = \sum_{i=0}^{m} \{y_i(\text{cost}_1(\Theta^T x_i)) + (1 - y_i)(\text{cost}_0(\Theta^T x_i))$$



▼ Random Forest (classifier)

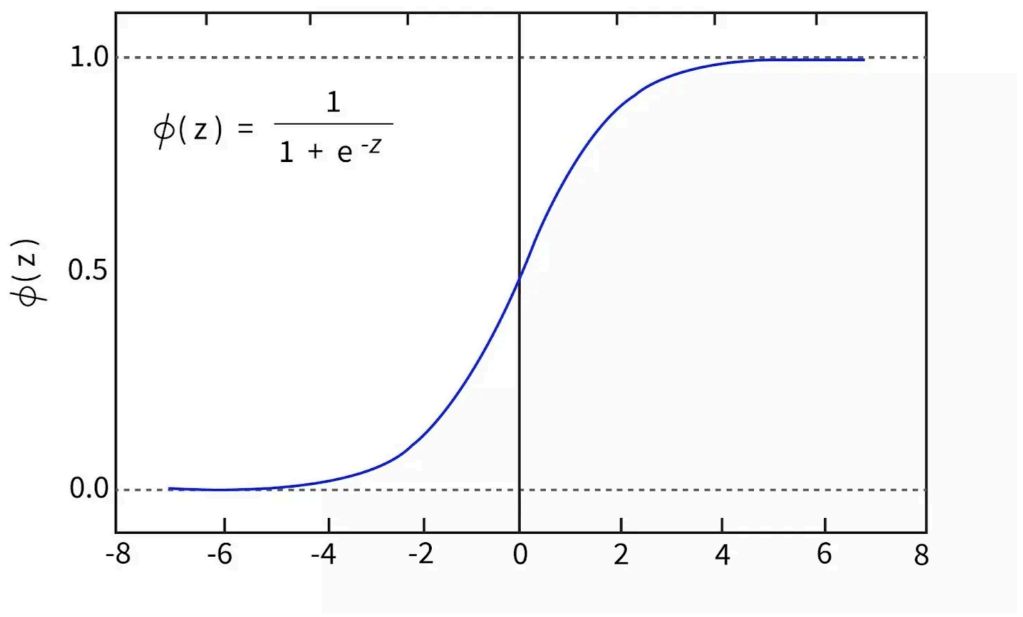
Random Forest is a supervised learning model that relies on the output of multiple Decision Trees, so Random forest itself is an example of an ensemble learning algorithm.



The Decision Tree models uses “majority voting” to decided the final class of the input variable.

▼ Logistic Regression

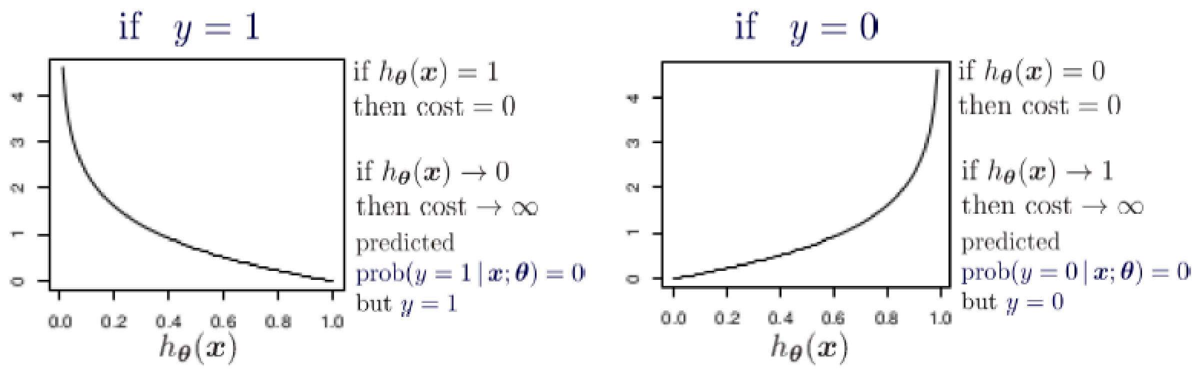
Logistic regression is a supervised learning model used with classification problems.



It works by specifying a decision boundary (typically 0.5)where if the output value is ≥ 0.5 , then it belongs to class 1, and if it is <0.5, then it belongs to class 0.

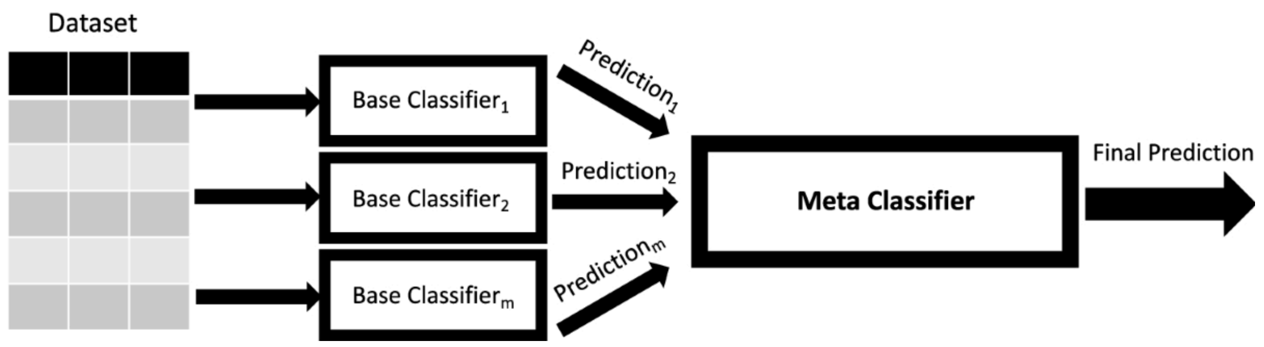
The model depends on a sigmoid function to squash any output value to a number between 0 and 1.

$$\text{cost}\,(h_{\boldsymbol{\theta}}(\boldsymbol{x}),\ y) = \begin{cases} -\log(h_{\boldsymbol{\theta}}(\boldsymbol{x})) & \text{if } \ y = 1 \\ -\log(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x})) & \text{if } \ y = 0 \end{cases}$$



It calculates the cost using - log , so that if the prediction was the same as the actual class, the cost is equal to 0, else, It is equal to infinity.

We also made use of an ensemble learning method called “Stacking”;



Where the dataset is entered into the base models first (in our case: SVM, Random Forest, and Logistic Regression. then, the predictions of the base models is entered to the meta model (in our case logistic regression) which makes the final decision.

```
estimators=[("rf",RandomForestClassifier(n_estimators=100)),
        ("svm",SVC(kernel="rbf",probability=True)),
        ("lr",LogisticRegression())]
```

```
meta_model=LogisticRegression()
stacker = StackingClassifier(estimators=estimators,final_estimator=meta_model,
stacker.fit(x_train_res, y_train_res)
```

We then use this stacker object to predict the labels for the test data, which we will then compare to the actual labels to evaluate the performance.

```
res_y_pred = stacker.predict(x_test_res)
```

7.Evaluating model performance

To use the required evaluation metrics, we import them from the metrics module of sklearn

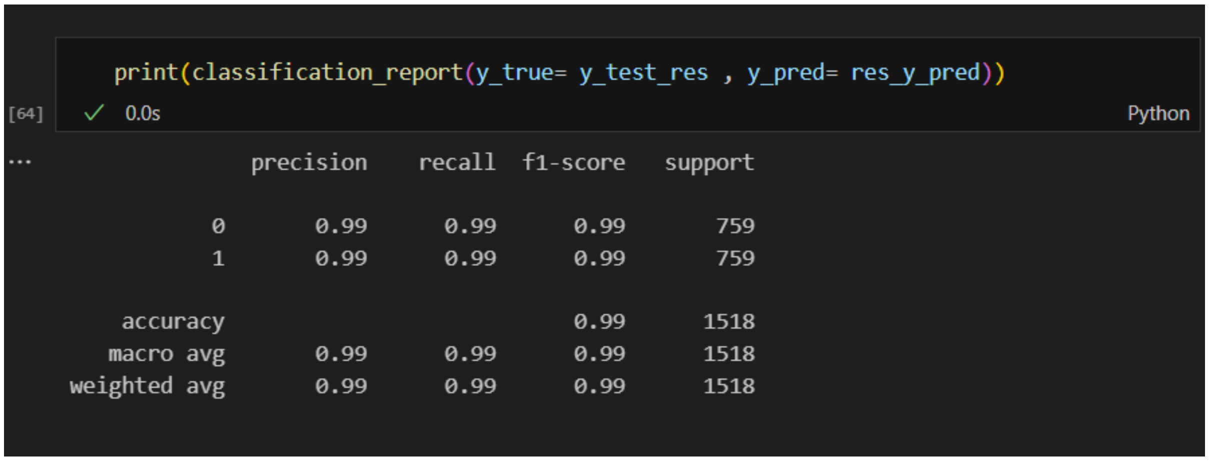
```
from sklearn.metrics import classification_report , recall_score , precision_score
```

The evaluation metrics:

▼ classification_report

The classification report is a combination of other evaluation metrics like precision, recall, and f1-score.

The results:



```
print(classification_report(y_true= y_test_res , y_pred= res_y_pred))
[64]  ✓ 0.0s                                                     Python
...
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       759
           1       0.99      0.99      0.99       759

    accuracy                           0.99      1518
   macro avg       0.99      0.99      0.99      1518
weighted avg       0.99      0.99      0.99      1518
```

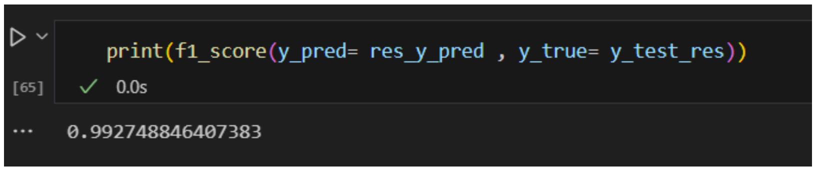
▼ f1_score

f1-score is the balanced measure between precision and recall.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

f1-score is used when we want to reduce both false positives and false negatives.

Our model’s f1_score:



```
print(f1_score(y_pred= res_y_pred , y_true= y_test_res))
[65] ✓ 0.0s
... 0.992748846407383
```

▼ precision_score

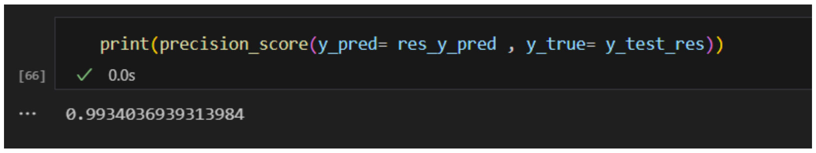
precision is the ratio of true positives to the sum of true positives and false positives.

$$Precision = \frac{TP}{TP + FP}$$

Precision is used when the false positive is more costly.

(when the objective is to reduce the false positives)

Our model’s precision:



```
print(precision_score(y_pred= res_y_pred , y_true= y_test_res))
[66] ✓ 0.0s
... 0.9934036939313984
```

▼ recall_score

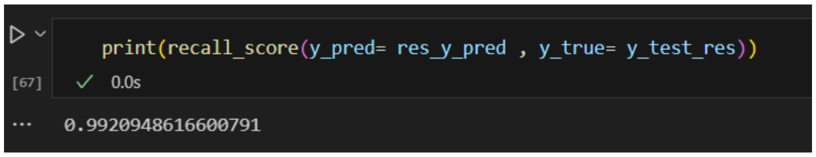
Recall is the ratio of true positives to the sum of true positives and false negatives.

$$Recall = \frac{TP}{TP + FN}$$

Recall is used when the false negative is more costly.

(when the objective is to reduce the false negatives)

Our model’s recall:



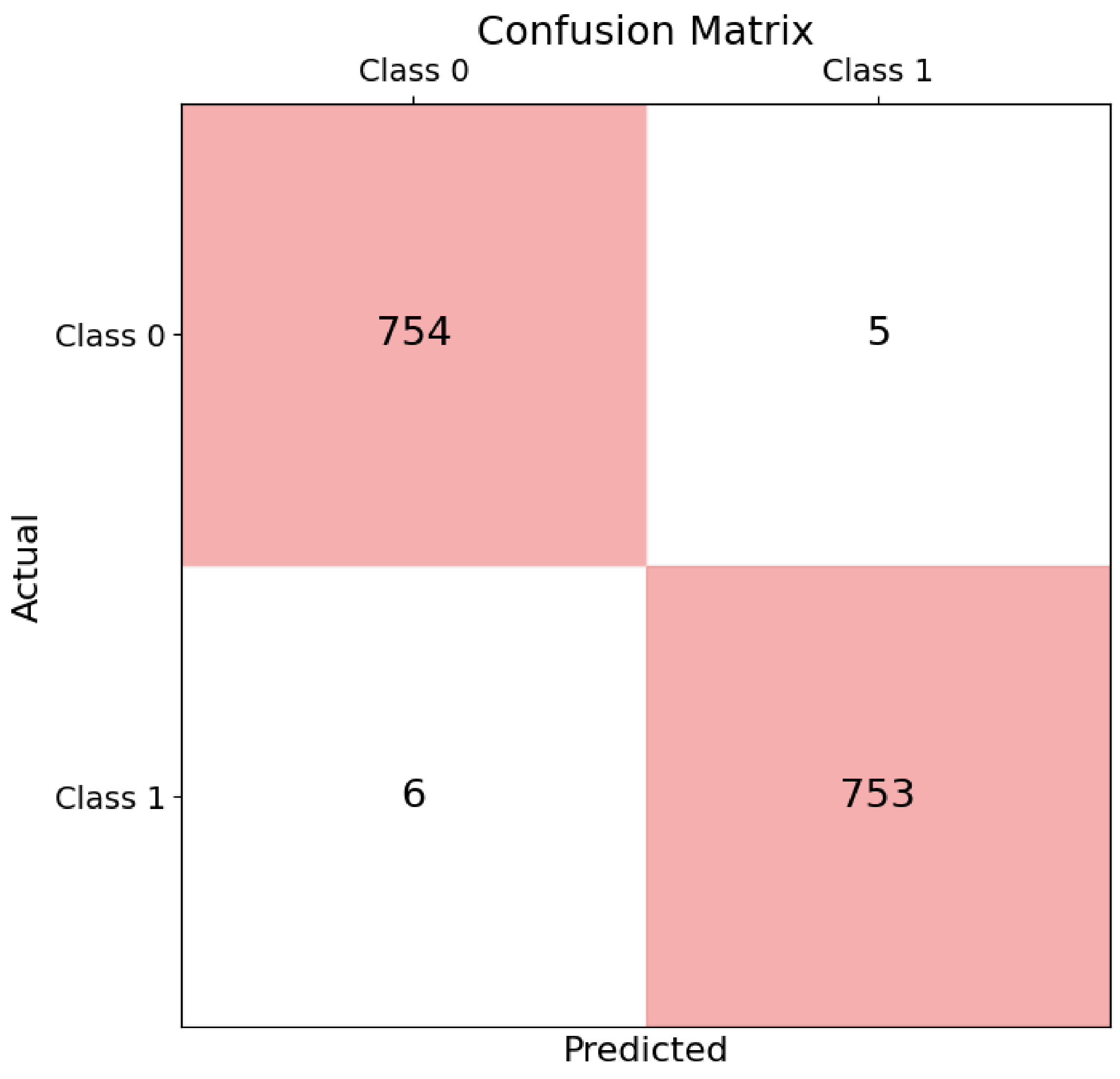
```
print(recall_score(y_pred= res_y_pred , y_true= y_test_res))
```

```
0.9920948616600791
```

▼ Confusion_matrix

Confusion matrix is the visual representation of the model’s performance as a table/matrix.

Our model’s confusion matrix:


Confusion Matrix
Class 0
Class 1
Class 0
Class 1
Actual
Predicted
754
5
6
753