WEEK 7

Classification Algorithm in Machine Learning

As we know, the Supervised Machine Learning algorithm can be broadly classified into Regression and Classification Algorithms. In Regression algorithms, we have predicted the output for continuous values, but to predict the categorical values, we need Classification algorithms.

The Classification algorithm is a Supervised Learning technique that is used to identify the category of new observations on the basis of training data. In Classification, a program learns from the given dataset or observations and then classifies new observation into a number of classes or groups. Such as, **Yes or No, 0 or 1, Spam or Not Spam, cat or dog,** etc. Classes can be called as targets/labels or categories.

Unlike regression, the output variable of Classification is a category, not a value, such as "Green or Blue", "fruit or animal", etc. Since the Classification algorithm is a Supervised learning technique, hence it takes labeled input data, which means it contains input with the corresponding output.

Types of classification

1. **Binary Classification:**
   Binary is a type of problem in classification in machine learning that has only two possible outcomes. For example, yes or no, true or false, spam or not spam, etc. Some common binary classification algorithms are logistic regression, decision trees, simple bayes, and support vector machines**.**

2. **Multi-Class Classification:**
   Multi-class is a type of classification problem with more than two outcomes and does not have the concept of normal and abnormal outcomes. Here each outcome is assigned to only one label. For example, classifying images, classifying species, and categorizing faces, among others. Some common multi-class algorithms are choice trees, progressive boosting, nearest k neighbors, and rough forest.

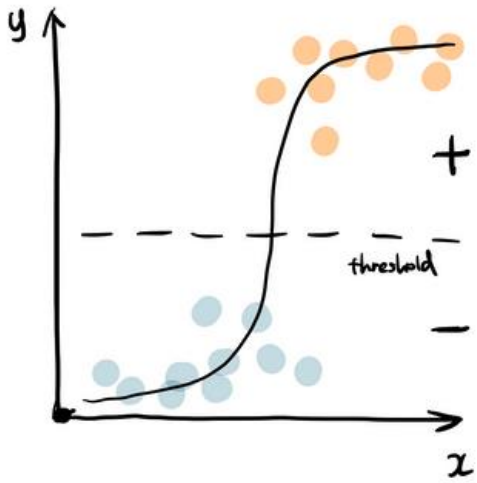
3. **Multi-Label Classification:**
   Multi-label is a type of classification problem that may have more than one class label assigned to the data. Here the model will have multiple outcomes. For example, a book or a movie can be categorized into multiple genres, or an image can have multiple objects. Some common multi-label algorithms are multi-label decision trees, multi-label gradient boosting, and multi-label random forests.

4. Imbalanced Classification:

Most machine learning algorithms assume equal data distribution. When the data distribution is not equal, it leads to imbalance. An imbalanced classification problem is a classification problem where the distribution of the dataset is skewed or biased. This method employs specialized techniques to change the composition of data samples. Some examples of imbalanced classification are spam filtering, disease screening, and fraud detection.

Classification Models

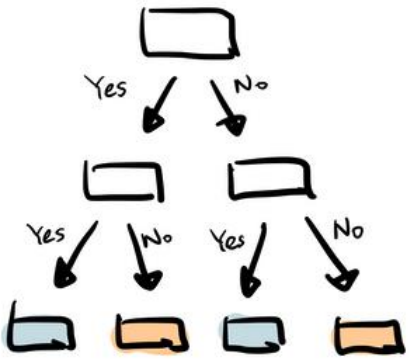
1. Logistic Regression



Logistics regression uses sigmoid function above to return the probability of a label. It is widely used when the classification problem is binary — true or false, win or lose, positive or negative The sigmoid function generates a probability output. By comparing the probability with a pre-defined threshold, the object is assigned to a label accordingly.

```
from sklearn.linear_model import LogisticRegression
reg = LogisticRegression()
reg.fit(X_train, y_train)
y_pred = reg.predict(X_test)
```

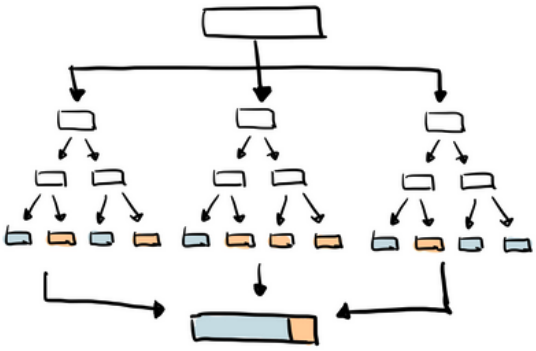
2. Decision Tree



Decision tree builds tree branches in a hierarchy approach and each branch can be considered as an if-else statement. The branches develop by partitioning the dataset into subsets based on most important features. Final classification happens at the leaves of the decision tree.

```
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier()
dtc.fit(X_train, y_train)
y_pred = dtc.predict(X_test)
```

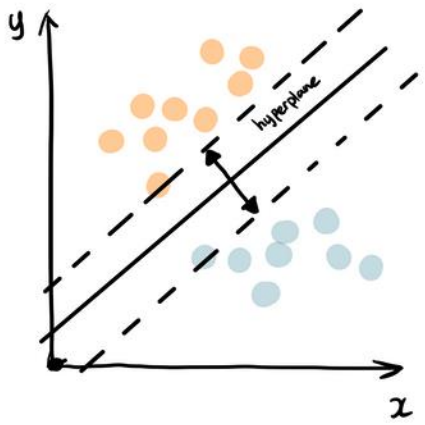
3. Random Forest



As the name suggest, random forest is a collection of decision trees. It is a common type of ensemble methods which aggregate results from multiple predictors. Random forest additionally utilizes bagging technique that allows each tree trained on a random sampling of original dataset and takes the majority vote from trees. Compared to decision tree, it has better generalization but less interpretable, because of more layers added to the model.

```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
rfc.fit(X_train, y_train)
y_pred = rfc.predict(X_test)
```

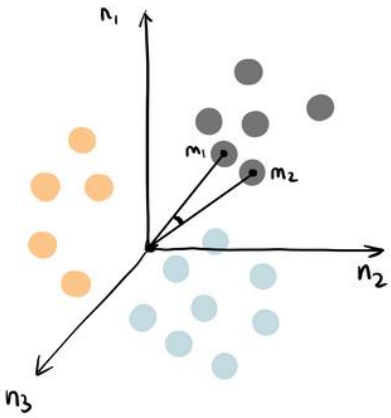
4. Support Vector Machine (SVM)



Support vector machine finds the best way to classify the data based on the position in relation to a border between positive class and negative class. This border is known as the hyperplane which maximize the distance between data points from different classes. Similar to decision tree and random forest, support vector machine can be used in both classification and regression, SVC (support vector classifier) is for classification problem.

```
from sklearn.svm import SVC
svc = SVC()
svc.fit(X_train, y_train)
y_pred = svc.predict(X_test)
```

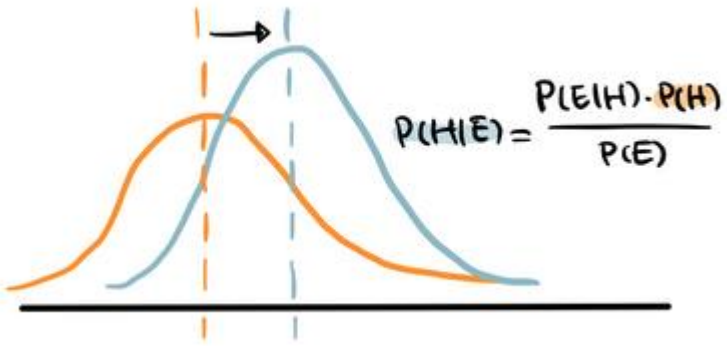
5. K-Nearest Neighbour (KNN)



You can think of k nearest neighbour algorithm as representing each data point in a n dimensional space — which is defined by n features. And it calculates the distance between one point to another, then assign the label of unobserved data based on the labels of nearest observed data points. KNN can also be used for building recommendation systems.

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

6. Naive Bayes



Naive Bayes is based on Bayes' Theorem — an approach to calculate conditional probability based on prior knowledge, and the naive assumption that each feature is independent to each other. The biggest advantage of Naive Bayes is that, while most machine learning algorithms rely on large amount of training data, it performs relatively well even when the training data size is small. Gaussian Naive Bayes is a type of Naive Bayes classifier that follows the normal distribution.

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)
```

Iris dataset from sci-kit learn : Perform data exploration, preprocessing and splitting
https://www.geeksforgeeks.org/exploratory-data-analysis-on-iris-dataset/

Decision Tree

- Decision tree algorithm falls under the category of supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem.
- A decision tree is a classification and prediction tool having a tree-like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.

Below are some assumptions that we made while using decision tree:

- At the beginning, we consider the whole training set as the root.
- Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
- On the basis of attribute values records are distributed recursively.
- We use statistical methods for ordering attributes as root or the internal node.

In Decision Tree the major challenge is to identification of the attribute for the root node in each level. This process is known as attribute selection. We have two popular attribute selection measures:

1. Information Gain
2. Gini Index

Attribute selection using Information Gain

Entropy:

Entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information.

$$\mathrm{H}(X) = -\sum_{i=1}^{n} \mathrm{P}(x_i) \log_b \mathrm{P}(x_i)$$

Information Gain:

Information gain can be defined as the amount of information gained about a random variable or signal from observing another random variable. It can be considered as the difference between the entropy of parent node and weighted average entropy of child nodes.

$$IG(S, A) = H(S) - H(S, A)$$

Alternatively,

$$IG(S, A) = H(S) - \sum_{i=0}^{n} P(x) * H(x)$$

Building Decision Tree using Information Gain

- Start with all training instances associated with the root node
- Use information gain to choose which attribute to label each node with
- Note: No root-to-leaf path should contain the same discrete attribute twice
- Recursively construct each subtree on the subset of training instances that would be classified down that path in the tree.

Example:

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Find the entropy of the class variable.

E(S) = -[(9/14)log(9/14) + (5/14)log(5/14)] = 0.94

where, S = Playing Football

From the above data for outlook, we can arrive at the following table

		play		
		yes	no	total
	sunny	3	2	5
Outlook	overcast	4	0	4
	rainy	2	3	5
				14

Now we have to calculate average weighted entropy. ie, we have found the total of weights of each feature multiplied by probabilities.

E(S, outlook)
= P(Sunny)* Entropy(Sunny) + P(Overcast)*Entropy(Overcast)+ P(Rain)*Entropy(Rain)
= (5/14)*E(2,3) + (4/14)*E(4,0) + (5/14)*E(3,2)
= (5/14)[-(2/5)log(2/5)-(3/5)log(3/5)] + (4/14)(0) +(5/14)[-(3/5)log(3/5)-(2/5)log(2/5)]
= 0.693

The next step is to find the information gain. It is the difference between parent entropy and average weighted entropy we found above.

IG(S, outlook) = 0.94 - 0.693 = 0.247

Similarly find Information gain for Temperature, Humidity, and Windy.
IG(S, Temperature) = 0.940 - 0.911 = 0.029
IG(S, Humidity) = 0.940 - 0.788 = 0.152
IG(S, Windy) = 0.940 - 0.8932 = 0.048

Now select the feature having the largest entropy gain. Here it is Outlook. So it forms the first node (root node) of our decision tree.

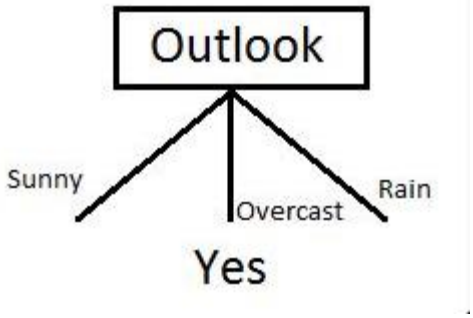
Now our data look as follows

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Overcast	Hot	High	Weak	Yes
Overcast	Cool	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes

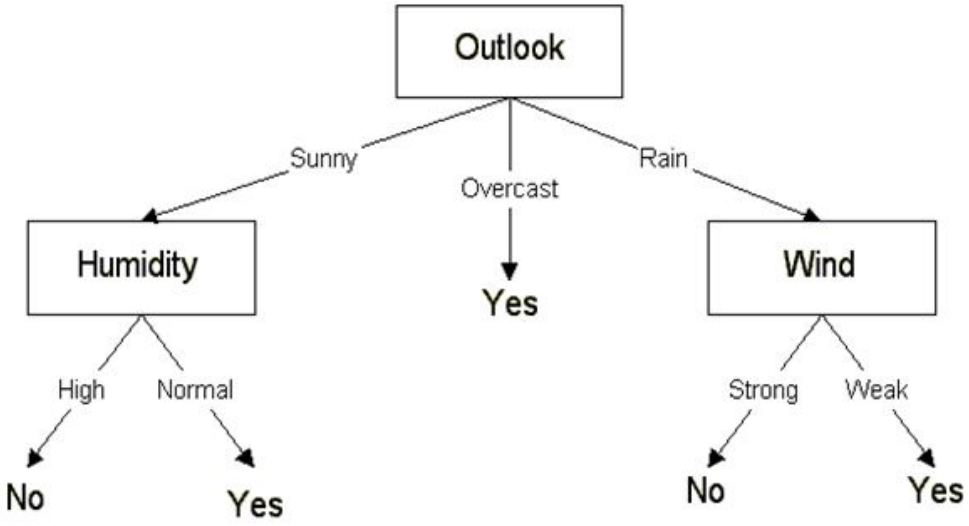
Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Rain	Mild	Normal	Weak	Yes
Rain	Mild	High	Strong	No

Since overcast contains only examples of class 'Yes' we can set it as yes. That means If outlook is overcast football will be played. Now our decision tree looks as follows.



Repeat the above process for each type under Outlook, i.e., Sunny and Rain to find the attribute with highest Information gain.

At the end, our decision tree will look as below:



Overfitting and Decision Trees

Decision Trees are prone to over-fitting. A decision tree will always overfit the training data if we allow it to grow to its max depth. A decision tree is overfit when the tree is trained to fit all samples in the training data set perfectly.

You can tweak some parameters such as min_samples_leaf to minimize default overfitting. The deeper you allow your tree to grow, the more complex the sequence of decision rules becomes. Assigning a **maximum depth** to a tree can simplify it and combat overfitting.

Pruning

Pruning is a technique that is used to reduce overfitting. Pruning also simplifies a decision tree by removing the weakest rules. Pruning is often distinguished into:

- **Pre-pruning** (early stopping) stops the tree before it has completed classifying the training set,
- **Post-pruning** allows the tree to classify the training set perfectly and then prunes the tree.

Pre-pruning

- The pre-pruning technique of Decision Trees is tuning the hyperparameters prior to the training pipeline.
- It involves the heuristic known as 'early stopping' which stops the growth of the decision tree - preventing it from reaching its full depth.
- It stops the tree-building process to avoid producing leaves with small samples.
- During each stage of the splitting of the tree, the cross-validation error will be monitored.
- If the value of the error does not decrease anymore - then we stop the growth of the decision tree.
- The hyperparameters that can be tuned for early stopping and preventing overfitting are: max_depth, min_samples_leaf and min_samples_split

Post-pruning

- Post-pruning does the opposite of pre-pruning and allows the Decision Tree model to grow to its full depth.
- Once the model grows to its full depth, tree branches are removed to prevent the model from overfitting.
- The algorithm will continue to partition data into smaller subsets until the final subsets produced are similar in terms of the outcome variable.
- The final subset of the tree will consist of only a few data points allowing the tree to have learned the data to the T.
- However, when a new data point is introduced that differs from the learned data - it may not get predicted well.

- The hyperparameter that can be tuned for post-pruning and preventing overfitting is ccp_alpha
- ccp stands for Cost Complexity Pruning and can be used as another option to control the size of a tree. A higher value of ccp_alpha will lead to an increase in the number of nodes pruned.

sklearn.tree.DecisionTreeClassifier

```
DecisionTreeClassifier(*, criterion='gini',
                       splitter='best',
                       max_depth = None,
                       min_samples_split=2,
                       min_samples_leaf=1,
                       min_weight_fraction_leaf=0.0,
                       max_features=None,
                       random_state=None,
                       max_leaf_nodes=None,
                       min_impurity_decrease=0.0,
                       class_weight=None,
                       ccp_alpha=0.0)
```

Important Parameters:

criterion{“gini”, “entropy”, “log_loss”}, default=”gini”

The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “log_loss” and “entropy” both for the Shannon information gain.

splitter{“best”, “random”}, default=”best”

The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.

max_depth:int, default=None

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split:int or float, default=2

The minimum number of samples required to split an internal node:

min_samples_leaf:int or float, default=1

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

min_weight_fraction_leaf:float, default=0.0
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features:int, float or {“auto”, “sqrt”, “log2”}, default=None
The number of features to consider when looking for the best split:
If “auto”, then max_features=sqrt(n_features).
If “sqrt”, then max_features=sqrt(n_features).
If “log2”, then max_features=log2(n_features).
If None, then max_features=n_features.

random_state:int, RandomState instance or None, default=None
Controls the randomness of the estimator. The features are always randomly permuted at each split, even if splitter is set to "best".

max_leaf_nodes:int, default=None
Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

ccp_alpha: non-negative float, default=0.0
Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than ccp_alpha will be chosen. By default, no pruning is performed.

Build decision tree-based model in python for Breast Cancer Wisconsin (diagnostic) dataset from sci-kit learn

Evaluation Metrics for Classification

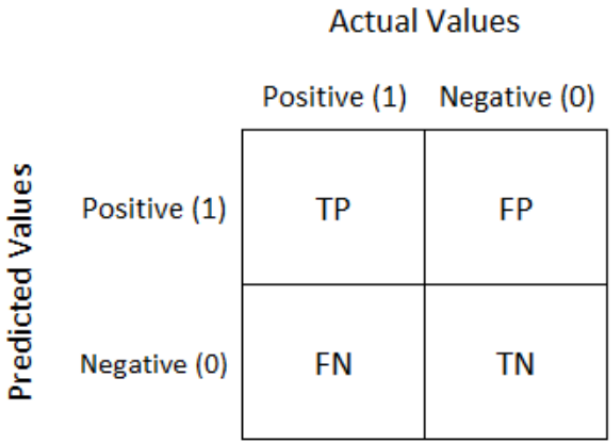
- The most important task in building any machine learning model is to evaluate its performance.
- Evaluation metrics are tied to machine learning tasks.

- Using different metrics for performance evaluation, we should be able to improve our model's overall predictive power before we roll it out for production on unseen data.
- Without doing a proper evaluation of the Machine Learning model by using different evaluation metrics, and only depending on accuracy, can lead to a problem when the respective model is deployed on unseen data and may end in poor predictions.

Confusion Matrix

Confusion Matrix is a performance measurement for the machine learning classification problems where the output can be two or more classes. It is a table with combinations of predicted and actual values.

A confusion matrix is defined as the table that is often used to describe the performance of a classification model on a set of the test data for which the true values are known.



It is extremely useful for measuring the Recall, Precision, Accuracy, and AUC-ROC curves.

True Positive: We predicted positive and it's true. In the image, we predicted that a woman is pregnant and she actually is.

True Negative: We predicted negative and it's true. In the image, we predicted that a man is not pregnant and he actually is not.

False Positive (Type 1 Error)- We predicted positive and it's false. In the image, we predicted that a man is pregnant but he actually is not.

False Negative (Type 2 Error)- We predicted negative and it's false. In the image, we predicted that a woman is not pregnant but she actually is.

Accuracy

Accuracy simply measures how often the classifier correctly predicts. We can define accuracy as the ratio of the number of correct predictions and the total number of predictions.

$$\boldsymbol{Accuracy = \frac{TP+TN}{TP+TN+FP+FN} = \frac{No\ of\ Correct\ Predictions}{Total\ no\ of\ predictions}}$$

- When any model gives an accuracy rate of 99%, you might think that model is performing very good but this is not always true and can be misleading in some situations.
- Accuracy is useful when the target class is ***well balanced*** but is not a good choice for the unbalanced classes.
- Imagine the scenario where we had 99 images of the dog and only 1 image of a cat present in our training data. Then our model would always predict the dog, and therefore we got 99% accuracy.
- In reality, Data is always imbalanced for example Spam email, credit card fraud, and medical diagnosis.
- Hence, if we want to do a better model evaluation and have a full picture of the model evaluation, other metrics such as recall and precision should also be considered

Precision

- Precision explains how many of the correctly predicted cases actually turned out to be positive.
- Precision is useful in the cases where False Positive is a higher concern than False Negatives.
- The importance of Precision is in music or video recommendation systems, e-commerce websites, etc. where wrong results could lead to customer churn and this could be harmful to the business.
- **Precision for a label is defined as the number of true positives divided by the number of predicted positives.**

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

- Precision is very useful when you have a model that starts some kind of business workflow (e.g. marketing campaigns) when it predicts 1.
- So, you want your model to be as correct as possible when it says 1 and don't care too much when it predicts 0.
- Precision is very much used in marketing campaigns, because a marketing automation campaign is supposed to start an activity on a user when it predicts that they will respond successfully. That's why we need high precision, which is the probability that our model is correct when it predicts 1.
- Low values for precision will make our business lose money, because we are contacting customers that are not interested in our commercial offer.

Recall (Sensitivity)

- Recall explains how many of the actual positive cases we were able to predict correctly with our model.
- It is a useful metric in cases where False Negative is of higher concern than False Positive. It is important in medical cases where it doesn't matter whether we raise a false alarm but the actual positive cases should not go undetected.
- **Recall for a label is defined as the number of true positives divided by the total number of actual positives.**

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

- Recall is used when you have to correctly classify some event that has already occurred.
- For example, fraud detection models must have a high recall in order to detect frauds properly.
- In such situations, we don't care about the real 0s, because we are interested only in spotting the real 1s as often as possible.
- So, we're working with the second row of the confusion matrix.
- Common uses of recall are, as said, fraud detection models or even disease detection on a patient. If somebody is ill, we need to spot their illness avoiding the false negatives. A

false negative patient may become contagious and it's not safe. That's why, when we have to spot an event that already occurred, we need to work with recall.

Specificity (True negative rate)

- Specificity (SP) is calculated as the number of correct negative predictions divided by the total number of negatives. It is also called true negative rate (TNR).

$$\boldsymbol{Specificity = \frac{TN}{TN + FP}}$$

- This metric is of interest if you are concerned about the accuracy of your negative rate and there is a high cost to a positive outcome.
- Let's say we wanted to send a handwritten note to the family of each passenger who died as identified by our model for titanic dataset.
- Since the Titanic sunk in 1912, we feel the families have had time to heal from their loss and so would not be distraught by receiving a note.
- However, we feel it would be incredibly insensitive to send a note to a family of a survivor, as their death would have been more recent (the last Titanic survivor died in 2009 at age 97) and the family would still be grieving.

F1 Score

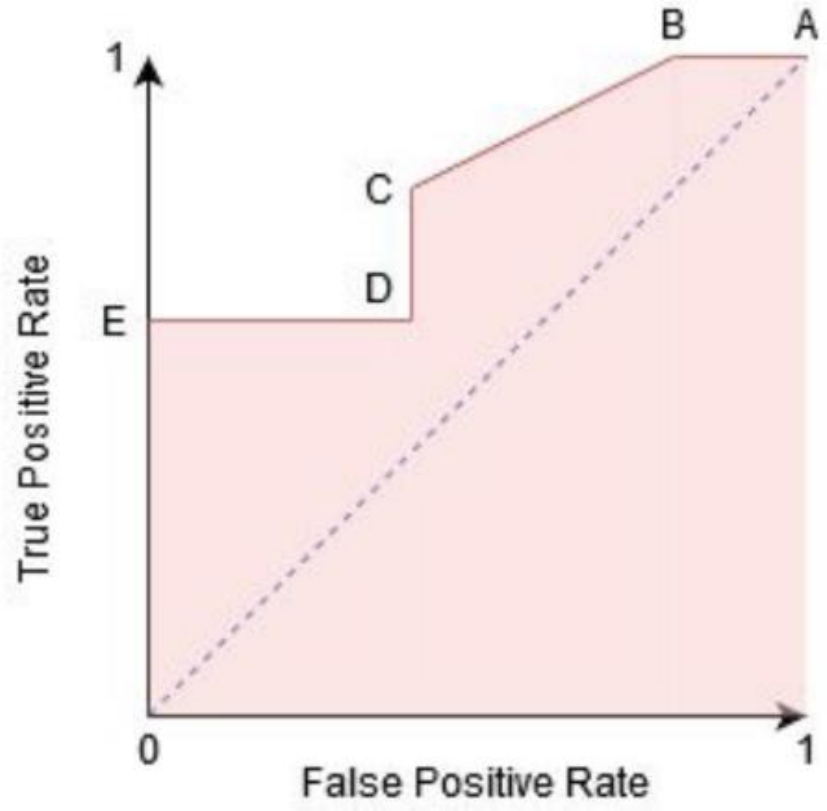
- It gives a combined idea about Precision and Recall metrics. It is maximum when Precision is equal to Recall.
- **F1 Score is the harmonic mean of precision and recall.**

$$\boldsymbol{F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall}}$$

- The F1 score punishes extreme values more. F1 Score could be an effective evaluation metric in the following cases:
  - When FP and FN are equally costly.
  - Adding more data doesn't effectively change the outcome
  - True Negative is high

AUC-ROC

- The Receiver Operator Characteristic (ROC) is a probability curve that plots the TPR(True Positive Rate) against the FPR(False Positive Rate) at various threshold values and separates the 'signal' from the 'noise'.
- The **Area Under the Curve (AUC)** is the measure of the ability of a classifier to distinguish between classes. From the graph, we simply say the area of the curve ABDE and the X and Y-axis.
- From the graph shown below, the greater the AUC, the better is the performance of the model at different threshold points between positive and negative classes.
- This simply means that When AUC is equal to 1, the classifier is able to perfectly distinguish between all Positive and Negative class points.
- When AUC is equal to 0, the classifier would be predicting all Negatives as Positives and vice versa.
- When AUC is 0.5, the classifier is not able to distinguish between the Positive and Negative classes.



*** Evaluation of Breast cancer decision tree model with different metrics**

Hyper parameter tuning for DecisionTreeClassifier

- The models can have many hyperparameters.
- Parameters like in decision criterion, max_depth, min_sample_split, etc. are called hyperparameters.
- We can find the best combination of the parameter using grid search methods.
- Grid search is a technique for tuning hyperparameter that may facilitate to build a model and evaluate a model for every combination of algorithms parameters per grid.
- In the Grid Search, all the mixtures of hyperparameters combinations will pass through one by one into the model and check the score on each model.
- It gives us the set of hyperparameters which gives the best score.
- Grid Search takes the model or objects you'd prefer to train and different values of the hyperparameters.
- It then calculates the error for various hyperparameter values, permitting you to choose the best values.

Implementation of Hyper parameter tuning using GridSearchCV

```
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
from sklearn.datasets import load_breast_cancer
cancer_data = load_breast_cancer()
```

```
cancer_df = pd.DataFrame(cancer_data.data, columns=cancer_data.feature_names)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(cancer_df,
                                                    cancer_data.target,
                                                    test_size = 0.33,
                                                    random_state = 42)
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
```

```
# Create Decision Tree classifer object
dtc = DecisionTreeClassifier()

# Train Decision Tree Classifer
dtc.fit(X_train,y_train)
```

```
DecisionTreeClassifier()
```

```
param_dict = {
"criterion":['gini','entropy'],
    "max_depth":range(1,10),
    "min_samples_split":range(1,10),
    "min_samples_leaf":range(1,5)
}
```

```
from sklearn.model_selection import GridSearchCV
grid = GridSearchCV(dtc,
                    param_grid=param_dict,
                    cv=10,
                    verbose=1,
                    n_jobs=-1)
grid.fit(X_train,y_train)
```

```
Fitting 10 folds for each of 648 candidates, totalling 6480 fits

GridSearchCV(cv=10, estimator=DecisionTreeClassifier(), n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': range(1, 10),
                         'min_samples_leaf': range(1, 5),
                         'min_samples_split': range(1, 10)},
             verbose=1)
```

```
grid.best_params_
```

```
{'criterion': 'entropy',
 'max_depth': 3,
 'min_samples_leaf': 4,
 'min_samples_split': 4}
```

```
grid.best_estimator_
```

```
DecisionTreeClassifier(criterion='entropy', max_depth=3, min_samples_leaf=4,
                       min_samples_split=4)
```

```
grid.best_score_
```

```
0.9474358974358974
```