**A PROJECT**

**ON**

**“CLASSIFICATION LABORATORY”**

**INDEX**

|  |  |  |
| --- | --- | --- |
| **Sr.**  **No.** | **Topics** | **Page No.** |
| **1** | **INTRODUCTION OF CLASSIFICATION LABORATORY** |  |
| **2** | **OBJECTIVE** |  |
| **3** | **ASSUMPTION** |  |
| **4** | **FLOWCHART** |  |
| **5** | **DATA COLLECTION**   * **ABOUT THE DATA** * **DEALING WITH THE MISSING VALUES** • **EDA** |  |
| **6** | **METHODOLOGY**   * **K-NEAREST NEIGHBOR** * **LOGISTIC REGRESSION** * **DECISION TREE** * **RANDOM FOREST** * **ADA BOOST** * **GRADIENT BOOSTING** * **HYPER PARAMETER TUNING** * **PCA** |  |
| **7** | **RESULTS**   * **BIAS VARIANCE TRADEOFF** * **CONFUSION METRIX** * **CLASSIFICATION REPORT** * **AUC AND ROC CURVE** * **MC NEMAR TEST** |  |
| **8** | **INTERPRETATION** |  |
| **9** | **ENVIRONMENT** |  |
| **10** | **LIMITATION** |  |
| **11** | **SUGGESTION** |  |
| **12** | **RESEARCH GAP** |  |
| **13** | **SIGNIFICANCE OF STUDY** |  |
| **14** | **REQUIRED LIBRARIES AND TOOLS** |  |
| **15** | **BIBLIOGRAPHY** |  |

# INTRODUCTION OF CLASSIFICATION LABORATORY

The Classification Laboratory is a dynamic model that accepts csv data from users for classification issues with a binary outcome as the dependent variable.

This laboratory assists users in gaining insight into data, and users will receive an EDA report📑 based on pandas profiling of the file they provide.

A classifier manager aids in the management of various machine learning algorithms and the selection of the most appropriate parameters for all models. Deploy a classification laboratory using Streamlit.

**What is Machine Learning?**

"Machine learning" is not two different words. It is a single word that describes a machine that learns continuously on its own and evolves on its own. However, the questions are how and who? How do machines learn and who will teach them to become good learners? The answers to both questions are the same. Let's see who will teach the machine in such a way that it becomes a process of learning. We have the Python programming language as a medium of learning. that a computer can understand. From that understanding, it can collect and store data, so it can describe data in numeric terms. Statisticians serveas a bridge between a computer and a language. A statistician is a qualified professor who can teach machines. Because statisticians knowhow to deal with data. What will be the standard procedure for collecting data and describing it in a meaningful way so that it can be analysed and interpreted in a graphical view so it can be easily understood and also make inferences about the data so it can be used for predictive analysis*,* So the question for both answers is the same: statisticians who can teach a machine in an efficient way so it can perform well and accurately.

In a nutshell, machine learning is the process of describing data in such a way that it can detect patterns in it, understand the patterns, perform predictive analytics using patterns, and produce results that can be used as a criterion for choosing a learning method. This is the stander procedure of learning for machine.

**Why we need machine learning?**

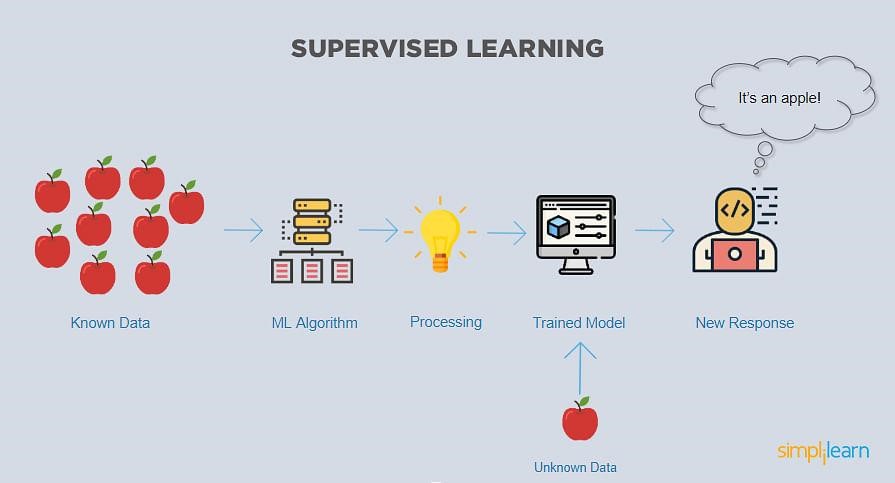
The human mind has a particular learning capacity. It is divided into two parts: the conscious mind and the subconscious mind. Only a few things are remembered by the conscious mind. It only has enough pieces to help you survive. Who told the conscious mind to remember this or that, by the way? It's all down to your subconscious mind. The conscious mind is similar to machine learning; the real population, like our subconscious mind, is unknown. Our onetenth component is conscious and well-known, but the remaining nine-tenths remains a mystery to us. The conscious mind is well represented by machine learning. It has the ability to collect and process large amounts of data all at once. The most fascinating aspect about machines is that they never get bored of repeating the same tasks. As a result, we require machines capable of processing large volumes of data, which is what a machine entails. If it can learn to understand whatever data is stored in that machine in order to provide a result, it will be able to build a pattern and consistently deliver an effective and accurate result.

**Types of machine learning**

Machine learning is divided into various categories , each one has a specific purpose and action, yielding results and utilizing various forms of data.

1. **Supervised Learning**

In supervised learning, we use known or labelled data for the training data. Since the [data](https://www.simplilearn.com/what-is-data-article) is known, the learning is, therefore, supervised, i.e., directed into successful execution. The input data goes through the Machine Learning algorithm and is used to train the model. Once the model is trained based on the known data, you can use unknown data into the model and get a new response.



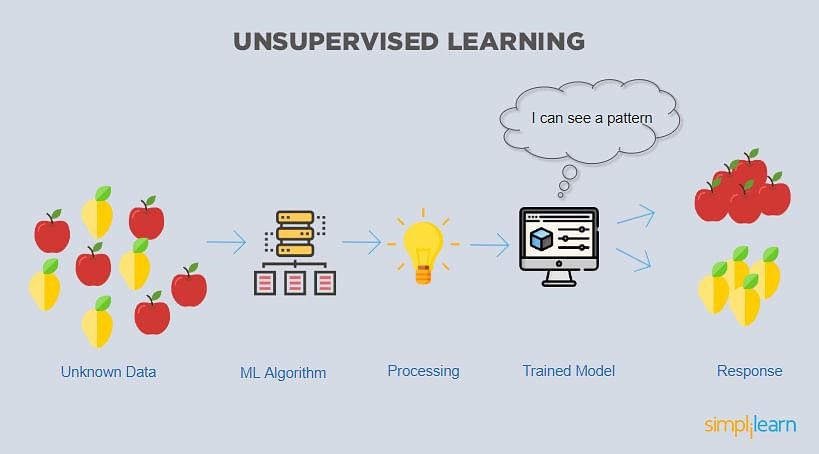
In this case, the model tries to figure out whether the data is an apple or another fruit. Once the model has been trained well, it will identify that the data is an apple and give the desired response.

Here is the list of top algorithms currently being used for supervised learning are:

* + - [Polynomial regression](https://www.analyticsvidhya.com/blog/2021/07/all-you-need-to-know-about-polynomial-regression/)
    - [Random forest](https://www.simplilearn.com/tutorials/machine-learning-tutorial/random-forest-algorithm)
    - [Linear regression](https://www.simplilearn.com/tutorials/machine-learning-tutorial/linear-regression-in-python)
    - [Logistic regression](https://www.simplilearn.com/tutorials/machine-learning-tutorial/logistic-regression-in-python)
    - [Decision trees](https://towardsdatascience.com/decision-tree-classifier-explained-in-real-life-picking-a-vacation-destination-6226b2b60575)
    - [K-nearest neighbors](https://www.simplilearn.com/tutorials/machine-learning-tutorial/knn-in-python)
    - [Naive Bayes](https://www.simplilearn.com/tutorials/machine-learning-tutorial/naive-bayes-classifier)

1. **Unsupervised Learning**

In unsupervised learning, the training data is unknown and unlabeled - meaning that no one has looked at the data before. Without the aspect of known data, the input cannot be guided to the algorithm, which is where the unsupervised term originates from. This data is fed to the Machine Learning algorithm and is used to train the model. The trained model tries to search for a pattern and give the desired response. In this case, it is often like the algorithm is trying to [break code like the Enigma machine](https://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code) but without the human mind directly involved but rather a machine.



In this case, the unknown data consists of apples and pears which look similar to each other. The trained model tries to put them all together so that you get the same things in similar groups.

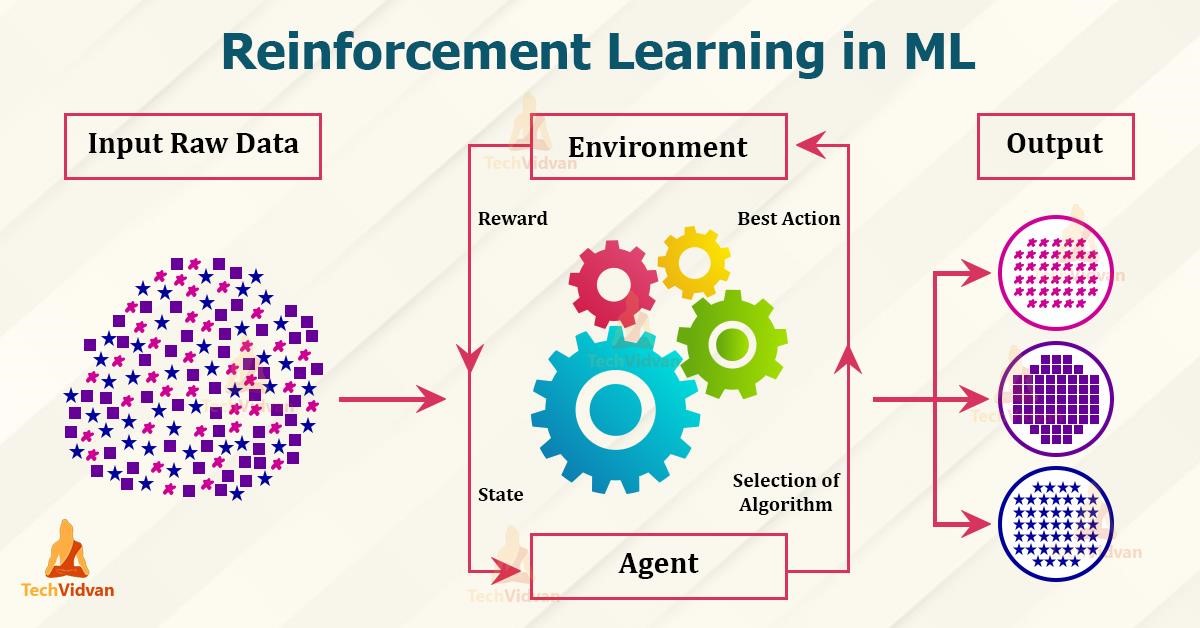
The top 7 algorithms currently being used for unsupervised learning are:

* + - Partial least squares
    - Fuzzy means
    - Singular value decomposition
    - [K-means clustering](https://www.simplilearn.com/tutorials/machine-learning-tutorial/k-means-clustering-algorithm)
    - Apriori
    - Hierarchical clustering
    - Principal component analysis

1. **Reinforcement Learning**

Like traditional types of data analysis, here, the algorithm discovers data through a process of trial and error and then decides what action results in higher rewards. Three major components make up reinforcement learning: the agent, the environment, and the actions. The agent is the learner or decisionmaker, the environment includes everything that the agent interacts with, and the actions are what the agent does.

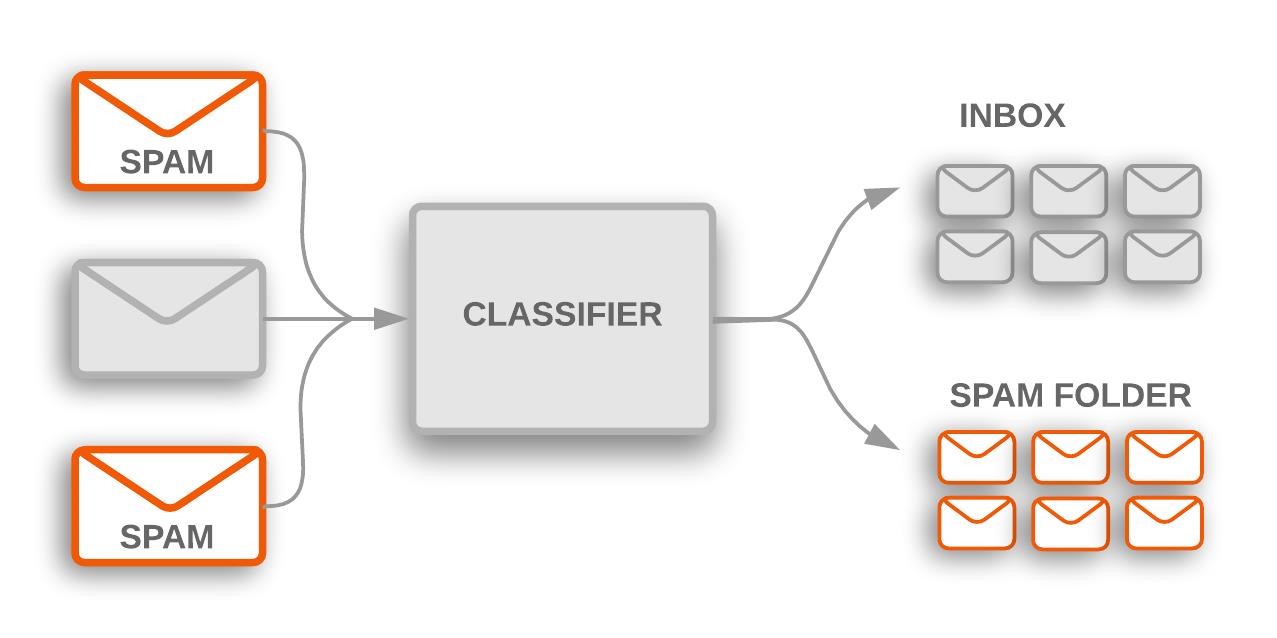
[Reinforcement learning](https://www.simplilearn.com/tutorials/machine-learning-tutorial/reinforcement-learning) happens when the agent chooses actions that maximize the expected reward over a given time. This is easiest to achieve when the agent is working within a sound policy framework.



In our case we have utilised supervised learning. Supervised learning is again classified under two categories:

* + **Classification** :

Classification uses an algorithm to accurately assign test data into specific categories. It recognizes specific entities within the dataset and attempts to draw some conclusions on how those entities should be labelled or defined. Common classification algorithms are linear classifiers, support vector machines (SVM), decision trees, k-nearest neighbor, and random forest, which are described in more detail below.

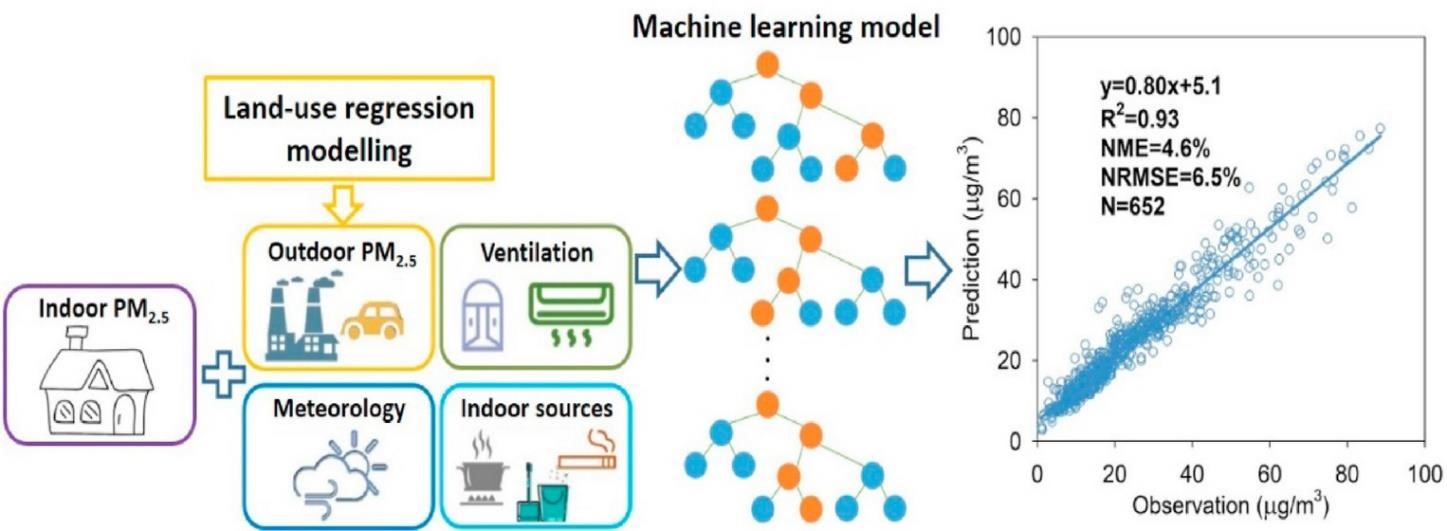


**In this we have utilised the following classification algorithms:**

* + - 1. K-Nearest neighbor
      2. Logistic Regression
      3. Decision Tree
      4. Random Forest
      5. Ada Boost
      6. Gradient Boosting

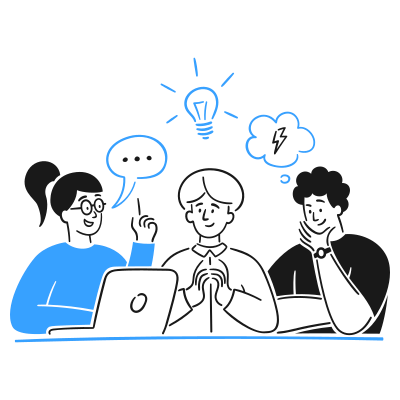
* + **Regression:**

Regression is used to understand the relationship between dependent and independent variables. It is commonly used to make projections, such as for sales revenue for a given business. [Linear regression,](https://www.ibm.com/analytics/learn/linear-regression) [logistical regression,](https://www.ibm.com/analytics/learn/logistic-regression) and polynomial regression are popular regression algorithms.



**In this, we are mainly dealing with the classification techniques.**

# OBJECTIVE



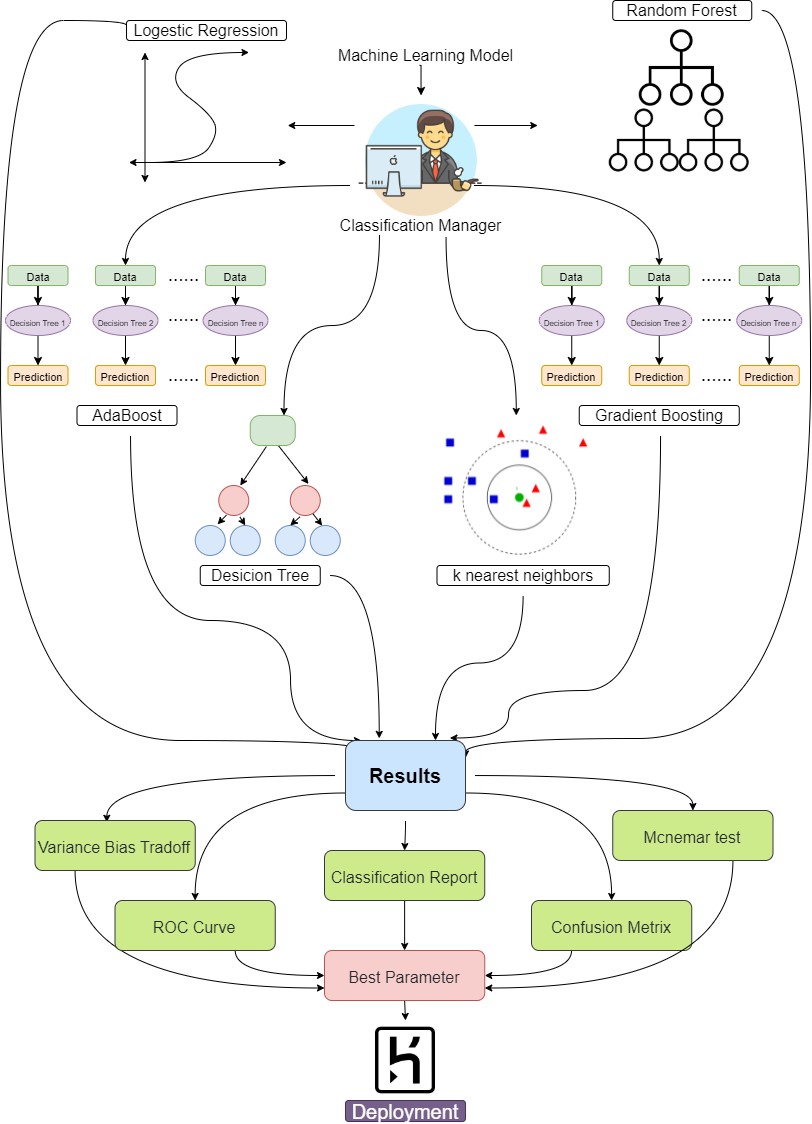
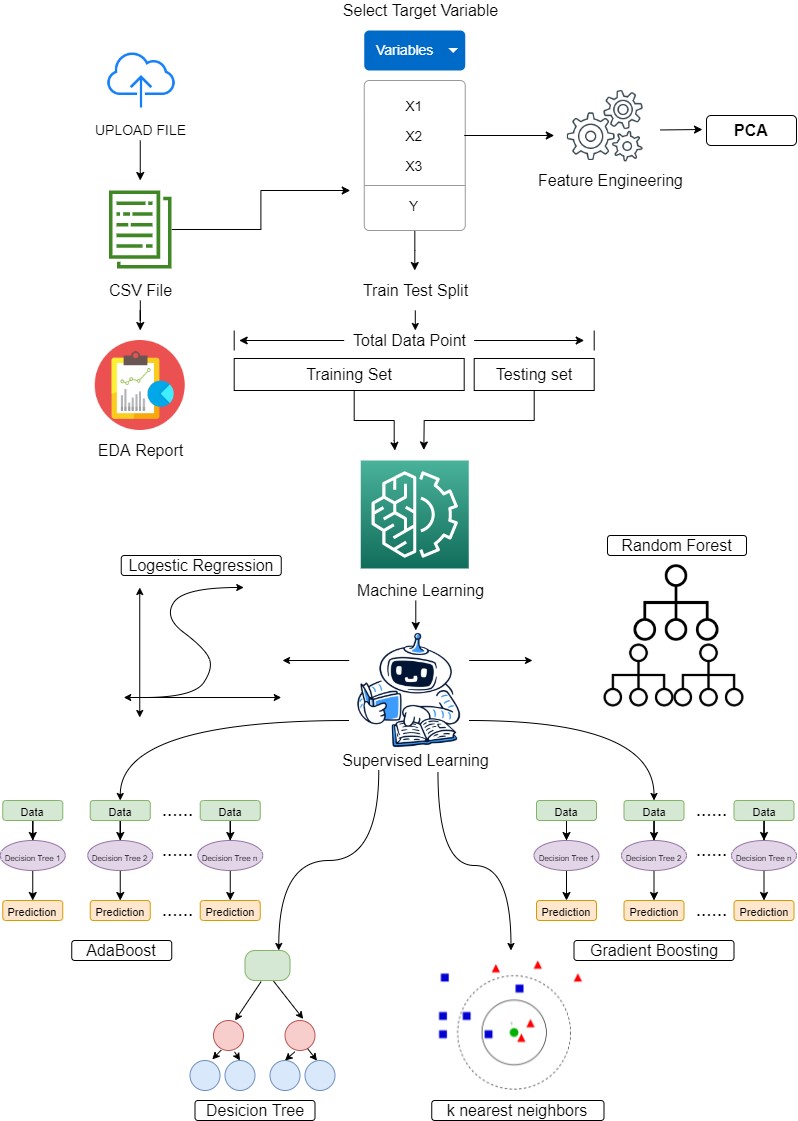
1. To Solve the classification problem using various algorithms.
2. To Find the best fitted algorithm using various statistical techniques
   * To Find the best parameters for the optimum performance of the algorithm
   * To Examine whether the model underfitted or overfitted
   * To Compare the classification report of various models
   * To minimize Type 2 error

# ASSUMPTIONS

* The target variable should contain binary values.
* There should not be any missing values in the data



# FLOWCHART



# DATA COLLECTION

The data is of secondary nature. It has been taken from Kaggle website.

Link address:

https://www.kaggle.com/uciml/pima-indians-diabetes-database

# ABOUT THE DATA

The data contains 7 independent variables and one dependent variable which is of binary nature. The following are the variables in the data:

**Pregnancies**

The data shows the number of times the person under study has got pregnant.

**Glucose**

Blood sugar, or glucose, is the main sugar found in your blood. It comes from the food you eat, and is your body's main source of energy. Your blood carries

glucose to all of your body's cells to use for energy. When this blood sugar level is high it causes diabetes.

The glucose tolerance test is a medical test in which glucose is given and blood samples taken afterward to determine how quickly it is cleared from the blood.

It can tell whether you are at risk for diabetes or if you already have it.

|  |  |
| --- | --- |
| Range | Type |
| [Below 140 mg/dL](https://www.webmd.com/diabetes/type-1-diabetes-guide/normal-blood-sugar-levels-chart-young-kids-infants-toddlers) | Normal blood sugar |
| [Between 140 and 199](https://www.webmd.com/diabetes/type-2-diabetes-guide/what-is-prediabetes) | Impaired glucose tolerance, or prediabetes |
| 200 or higher | Diabetes |

Low blood sugar is called hypoglycemia. A blood sugar level below 70 mg/dL (3.9 mmol/L) is low and can harm you. A blood sugar level below 54 mg/dL (3.0 mmol/L) is a cause for immediate action.

**Blood Pressure**

Blood pressure is the pressure of circulating blood against the walls of blood vessels. There are two types of blood pressure, the systolic and diastolic.

When your heart beats, it squeezes and pushes blood through your arteries to

the rest of your body. This force creates pressure on those blood vessels, and that's your systolic blood pressure.

The diastolic reading, or the bottom number, is the pressure in the arteries when the heart rests between beats. This is the time when the heart fills with blood and gets oxygen.

The data in our study is of diastolic blood pressure. This is what your diastolic blood pressure number means:

|  |  |
| --- | --- |
| Type | Range |
| Normal | Lower than 80 |
| Stage 1 hypertension | 80-89 |
| Stage 2 hypertension | 90 or more |
| Hypertensive crisis | 120 or more. |
| Hypotension | 60 or less |

**Skin Thickness**

The triceps skinfold thickness gives information about the fat reserves of the body. Normal thickness in males is 12 mm; in females, 23 mm.

**Insulin**

Insulin is a hormone created by your pancreas that controls the amount of glucose in your bloodstream at any given moment. It also helps store glucose in your liver, fat, and muscles. Finally, it regulates your body’s metabolism of carbohydrates, fats, and proteins.

|  |  |  |
| --- | --- | --- |
|  | Insulin Level | Insulin Level (SI Units\*) |
| Fasting | < 25 mIU/L | < 174 pmol/L |
| 30 minutes after glucose administration | 30-230 mIU/L | 208-1597 pmol/L |
| 1 hour after glucose administration | 18-276 mIU/L | 125-1917 pmol/L |
| 2 hour after glucose administration | 16-166 mIU/L | 111-1153 pmol/L |
| ≥3 hours after glucose administration | < 25 mIU/L | < 174 pmol/L |
| \*SI unit: conversional units x 6.945 | |  |

**BMI**

Body mass index, or BMI, is a measure of body size. It combines a person’s weight with their height. The results of a BMI measurement can give an idea about whether a person has the correct weight for their height.

|  |  |
| --- | --- |
| BMI | Weight status |
| Below 18.5 | Underweight |
| 18.5–24.9 | Healthy |
| 25.0–29.9 | Overweight |
| 30.0 and above | Obese |

**Diabetes pedigree function**

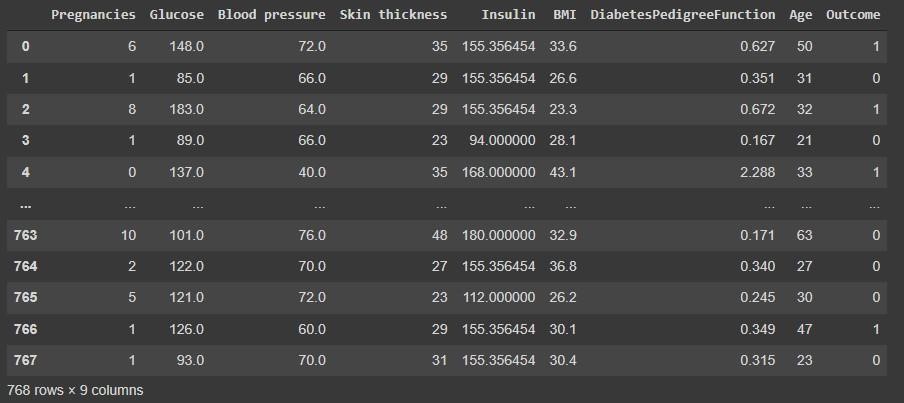
This is the function which shows the likelihood of a person suffering diabetes based on the family history.

**Age**

This shows the age in years of the person under study.

**Outcome**

This shows the final outcome, whether the person is suffering from diabetes or not.



# DEALING WITH THE MISSING VALUES

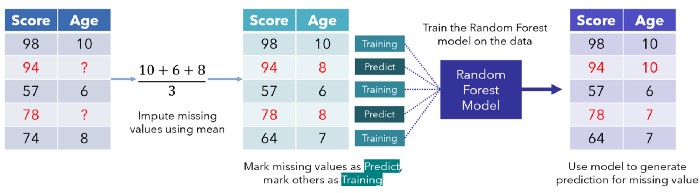
The dataset contains missing values. So before moving further to any operations, we first need to fill all the missing data with the appropriate values. For this we have use MissForest Algorithm.

MissForest is a machine learning-based imputation technique. It uses a Random Forest algorithm to do the task. It is based on an iterative approach, and at each iteration the generated predictions are better.

Stekhoven and Buhlmann, creators of the algorithm, conducted a study in 2011 in which imputation methods were compared on datasets with randomly introduced missing values. MissForest outperformed all other algorithms in all metrics, including KNN-Impute, in some cases by over 50%.

First, the missing values are filled in using median/mode imputation. Then, we mark the missing values as ‘Predict’ and the others as training rows, which are

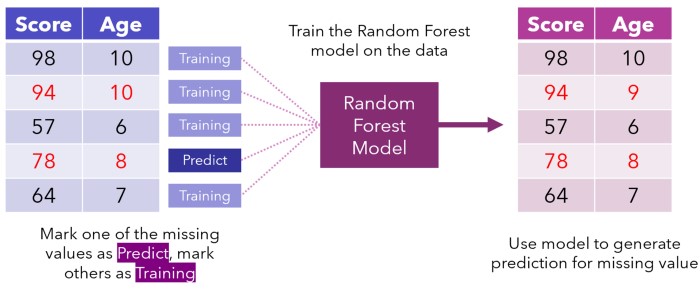
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| fed into a Random Forest model trained to predict, in this case, | | | Age | based |
| on | Score | . The generated prediction for that row is then filled in to produce a transformed dataset. | | |
|  |



Assume that the dataset is truncated. Image created by author.

This process of looping through missing data points repeats several times, each iteration improving on better and better data. It’s like standing on a pile of rocks while continually adding more to raise yourself: the model uses its current position to elevate itself further.

The model may decide in the following iterations to adjust predictions or to keep them the same.



Iterations continue until some stopping criteria is met or after a certain number of iterations has elapsed. As a general rule, datasets become well imputed after four to five iterations, but it depends on the size and amount of missing data.

**Advantages:**

* Can be applied to **mixed data types** (missing in numeric & categorical variables)
* **No pre-processing** required (no dummy-coding, standardization, data splitting, etc.)
* **No assumptions** required (aside from the normal assumption of being MAR/MCAR)
* **Robust to noisy data**, as random forests effectively have build-in feature selection. Methods like KNN imputation will have poor predictions in datasets with weak & non-informative predictors,

whereas missForest() will make little to no use of these features

* **Non-parametric**: makes no assumptions about the relationship between the features, unlike MICE which assumes linearity
* Excellent **predictive power**
* Can leverage **non-linear** and **interaction effects** between features to improve imputation accuracy
* Gives an **OOB error estimate** for its predictions (Numeric: NRMSE/MSE, Categorical: PFC)
* Works with **high dimensionality data** (pnp n)

**Disadvantages:**

* **Imputation time**, which increases with the number of observations, predictors and number of predictors containing missing values
* It inherits the same **lack of interpretability** of random forests
* **It is an algorithm**, not a model object you can store somewhere. This means it has to run each time missing data has to be imputed, which could be problematic in some production environments

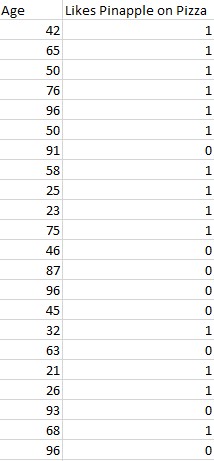
# METHODOLOGY



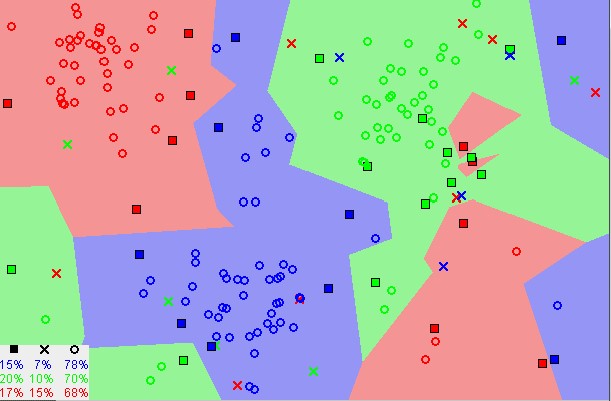
**1.K-Nearest Neighbors Algorithm:**

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems.

A **classification problem** has a discrete value as its output. For example, “likes pineapple on pizza” and “does not like pineapple on pizza” are discrete. There is no middle ground.



The KNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other.



Notice in the image above that most of the time, similar data points are close to each other. The KNN algorithm hinges on this assumption being true enough for the algorithm to be useful. KNN captures the idea of similarity (sometimes called distance, proximity, or closeness) with some mathematics we might have learned in our childhood— calculating the distance between points on a graph.

There are other ways of calculating distance, and one way might be preferable depending on the problem we are solving. However, the straight-line distance (also called the Euclidean distance) is a popular and familiar choice.

## The KNN Algorithm

1. Load the data
2. Initialize K to your chosen number of neighbors
3. For each example in the data
   1. Calculate the distance between the query example and the current example from the data.
   2. Add the distance and the index of the example to an ordered collection
4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. If regression, return the mean of the K labels
8. If classification, return the mode of the K labels

## Choosing the right value for K

To select the K that’s right for your data, we run the KNN algorithm several times with different values of K and choose the K that reduces the number of errors we encounter while maintaining the algorithm’s ability to accurately make predictions when it’s given data it hasn’t seen before. Here are some things to keep in mind:

1. As we decrease the value of K to 1, our predictions become less stable. Just think for a minute, imagine K=1 and we have a query point surrounded by several reds and one green (I’m thinking about the top left corner of the coloured plot above), but the green is the single nearest neighbor. Reasonably, we would think the query point is most likely red, but because K=1, KNN incorrectly predicts that the query point is green.
2. Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point). Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.
3. In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

## Advantages

1. The algorithm is simple and easy to implement.
2. There’s no need to build a model, tune several parameters, or make additional assumptions.
3. The algorithm is versatile. It can be used for classification, regression, and search (as we will see in the next section).

## Disadvantages

1. The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase.

## KNN in practice

KNN’s main disadvantage of becoming significantly slower as the volume of data increases makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.

However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, KNN can still be useful in solving problems that have solutions that depend on identifying similar objects. An example of this is using the KNN algorithm in recommender systems, an application of KNN-search.

|  |
| --- |
| [sklearn.neighbors**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors)**KNeighborsClassifier** |
| *class* sklearn.neighbors.KNeighborsClassifier(*n\_neighbors=5*, *\**, *weights='unifor m'*, *algorithm='auto'*, *leaf\_size=30*, *p=2*, *metric='minkowski'*, *metric\_params=N one*, *n\_jobs=None*[)[source]](https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/neighbors/_classification.py#L22) |

Classifier implementing the k-nearest neighbors vote.

**Parameters**

**n\_neighbors**

***int, default=5***

Number of neighbors to use by default for [**kneighbors**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.kneighbors) queries.

**weights*{‘uniform’, ‘distance’} or callable, default=’uniform’*** Weight function used in prediction. Possible values:

* ‘uniform’ : uniform weights. All points in each neighborhood are weighted equally.
* ‘distance’ : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
* [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights. **algorithm*{‘auto’, ‘ball\_tree’, ‘kd\_tree’, ‘brute’}, default=’auto’*** Algorithm used to compute the nearest neighbors:
* ‘ball\_tree’ will use [**BallTree**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.BallTree.html#sklearn.neighbors.BallTree)
* ‘kd\_tree’ will use [**KDTree**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KDTree.html#sklearn.neighbors.KDTree)
* ‘brute’ will use a brute-force search.
* ‘auto’ will attempt to decide the most appropriate algorithm based on the values passed to [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.fit) method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

### leaf\_size*int, default=30*

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

### p*int, default=2*

Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan\_distance (l1), and euclidean\_distance (l2) for p = 2. For arbitrary p, minkowski\_distance (l\_p) is used.

### metric*str or callable, default=’minkowski’*

The distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. For a list of available metrics, see the documentation of [**DistanceMetric**.](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html#sklearn.metrics.DistanceMetric) If metric is “precomputed”, X is assumed to be a distance matrix and must be square during fit. X may be a [sparse graph,](https://scikit-learn.org/stable/glossary.html#term-sparse-graph) in which case only “nonzero” elements may be considered neighbors.

### metric\_params*dict, default=None*

Additional keyword arguments for the metric function. **n\_jobs*int, default=None***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The number of parallel jobs to run for neighbors search. | | | None | means 1 unless |
| in a [**joblib.parallel\_backend**](https://joblib.readthedocs.io/en/latest/parallel.html#joblib.parallel_backend) context. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-n_jobs) for more details. Doesn’t affect | -1 | means using all processors.  [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.fit) method. | | |
|  |
| **Attributes** | | | | |
| **classes\_*array of shape (n\_classes,)*** | | | | |

Class labels known to the classifier

## effective\_metric\_*str or callble*

The distance metric used. It will be same as the metric parameter or a

|  |  |  |
| --- | --- | --- |
| synonym of it, e.g. ‘euclidean’ if the | metric | parameter set to ‘minkowski’ |
| and p parameter set to 2. | | |

## effective\_metric\_params\_*dict*

Additional keyword arguments for the metric function. For most metrics will be same with metric\_params parameter, but may also contain the p parameter value if the effective\_metric\_ attribute is set to ‘minkowski’.

## n\_features\_in\_*int*

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

*New in version 0.24.*

### feature\_names\_in\_*ndarray of shape (n\_features\_in\_,)*

Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when X has feature names that are all strings.

*New in version 1.0.* **n\_samples\_fit\_*int***

Number of samples in the fitted data.

## outputs\_2d\_*bool*

|  |  |
| --- | --- |
| False when | y’s shape is (n\_samples, ) or (n\_samples, 1) during fit otherwise |
| True. | |

## Examples

|  |
| --- |
| >>>  **>>>** X = [[0], [1], [2], [3]]  **>>>** y = [0, 0, 1, 1]  **>>> from** **sklearn.neighbors** **import** KNeighborsClassifier  **>>>** neigh = KNeighborsClassifier(n\_neighbors=3)  **>>>** neigh.fit(X, y)  KNeighborsClassifier(...)  **>>>** print(neigh.predict([[1.1]]))  [0]  **>>>** print(neigh.predict\_proba([[0.9]]))  [[0.666... 0.333...]] |

## Methods

|  |  |
| --- | --- |
| [**fit**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.fit)X, y) | Fit the k-nearest neighbors classifier from the training dataset. |

[**get\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.get_params)[deep]) Get parameters for this estimator.

|  |
| --- |
| [**kneighbors**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.kneighbors)[X, n\_neighbors, retu  Find the K-neighbors of a point. rn\_distance]) |

[**kneighbors\_graph**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.kneighbors_graph)[X, n\_neighbo Compute the (weighted) graph of krs, mode]) Neighbors for points in X.

|  |  |
| --- | --- |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.predict)X) | Predict the class labels for the provided data. |

Return probability estimates for the test [**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.predict_proba)X)

data X.

|  |  |
| --- | --- |
| [**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.score)X, y[, sample\_weight]) | Return the mean accuracy on the given test data and labels. |

[**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier.set_params)\*\*params) Set the parameters of this estimator.

### Summary

The k-nearest neighbors (KNN) algorithm is a simple, supervised machine learning algorithm that can be used to solve both classification and regression problems. It’s easy to implement and understand, but has a major drawback of becoming significantly slows as the size of that data in use grows.

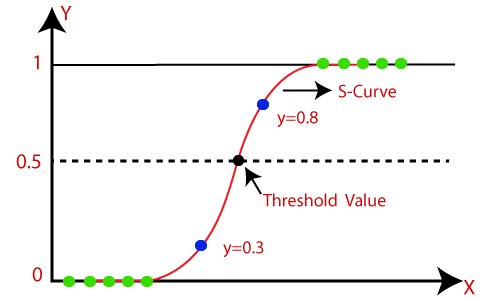
KNN works by finding the distances between a query and all the examples in the data, selecting the specified number examples (K) closest to the query, then votes for the most frequent label (in the case of classification) or averages the labels (in the case of regression).

In the case of classification and regression, we saw that choosing the right K for our data is done by trying several Ks and picking the one that works best.

Finally, we looked at an example of how the KNN algorithm could be used in recommender systems, an application of KNN-search.

**2.Logistic Regression**

* Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
* Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, true or False, etc. but instead of giving the exact value as 0 and 1, **it gives the probabilistic values which lie between 0 and 1**.
* Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas **Logistic regression is used for solving the classification problems**.
* In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
* The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
* Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
* Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification. The below image is showing the logistic function:



|  |
| --- |
| *Note:* ***Logistic regression uses the concept of predictive modeling as regression; therefore, it is called logistic regression, but is used to classify samples; Therefore, it falls under the classification algorithm.*** |

**Logistic Function (Sigmoid Function):**

* The sigmoid function is a mathematical function used to map the predicted values to probabilities.
* It maps any real value into another value within a range of 0 and 1.
* The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The Sform curve is called the Sigmoid function or the logistic function.
* In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

**Assumptions for Logistic Regression:**

* The dependent variable must be categorical in nature.
* The independent variable should not have multi-collinearity.

**Logistic Regression Equation:**

The Logistic regression equation can be obtained from the Linear Regression equation. The mathematical steps to get Logistic Regression equations are given below:

* We know the equation of the straight line can be written as:



* In Logistic Regression y can be between 0 and 1 only, so for this let's divide the above equation by (1-y):



* But we need range between -[infinity] to +[infinity], then take logarithm of the equation it will become:



The above equation is the final equation for Logistic Regression.

**Type of Logistic Regression:**

On the basis of the categories, Logistic Regression can be classified into three types:

* **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
* **Multinomial:** In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
* **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

**Advantages and Disadvantages**

Many of the pros and cons of the [linear regression model](https://christophm.github.io/interpretable-ml-book/limo.html#limo) also apply to the logistic regression model. Logistic regression has been widely used by many different people, but it struggles with its restrictive expressiveness (e.g. interactions must be added manually) and other models may have better predictive performance.

Another disadvantage of the logistic regression model is that the interpretation is more difficult because the interpretation of the weights is multiplicative and not additive.

Logistic regression can suffer from **complete separation**. If there is a feature that would perfectly separate the two classes, the logistic regression model can no longer be trained. This is because the weight for that feature would not converge, because the optimal weight would be infinite. This is really a bit unfortunate, because such a feature is really useful. But you do not need machine learning if you have a simple rule that separates both classes. The problem of complete separation can be solved by introducing penalization of the weights or defining a prior probability distribution of weights.

On the good side, the logistic regression model is not only a classification model, but also gives you probabilities. This is a big advantage over models that can only provide the final classification. Knowing that an instance has a 99% probability for a class compared to 51% makes a big difference.

Logistic regression can also be extended from binary classification to multiclass classification. Then it is called Multinomial Regression.

|  |
| --- |
| [sklearn.linear\_model**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model)**LogisticRegression** |
| *class* sklearn.linear\_model.LogisticRegression(*penalty='l2'*, *\**, *dual=False*, *tol=0.0001*, *C=1.0*, *fit \_intercept=True*, *intercept\_scaling=1*, *class\_weight=None*, *random\_state=None*, *solver='lbfgs'*, *max\_iter=100*, *multi\_class='auto'*, *verbose=0*, *warm\_start=False*, *n\_jobs=None*, *l1\_ratio=None*[)[ source]](https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/linear_model/_logistic.py#L1120) |

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the ‘multi\_class’ option is set to ‘ovr’, and uses the cross-entropy loss if the ‘multi\_class’ option is set to ‘multinomial’. (Currently the ‘multinomial’ option is supported only by the ‘lbfgs’, ‘sag’, ‘saga’ and ‘newton-cg’ solvers.)

This class implements regularized logistic regression using the ‘liblinear’ library, ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

The ‘newton-cg’, ‘sag’, and ‘lbfgs’ solvers support only L2 regularization with primal formulation, or no regularization. The ‘liblinear’ solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the ‘saga’ solver.

**Parameters**

**penalty**

***{***

***‘l1’, ‘l2’, ‘elasticnet’, ‘none’}, default=’l2’***

Specify the norm of the penalty:

* 'none': no penalty is added;
* 'l2': add a L2 penalty term and it is the default choice;
* 'l1': add a L1 penalty term;
* 'elasticnet': both L1 and L2 penalty terms are added.

**Warning**

Some penalties may not work with some solvers. See the parameter solver below, to know the compatibility between the penalty and solver.

*New in version 0.19:* l1 penalty with SAGA solver (allowing ‘multinomial’ + L1) **dual*bool, default=False***

Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n\_samples > n\_features. **tol*float, default=1e-4***

Tolerance for stopping criteria.

#### Cfloat, default=1.0

Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

#### fit\_interceptbool, default=True

Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

**intercept\_scaling*float, default=1***

Useful only when the solver ‘liblinear’ is used and self.fit\_intercept is set to True. In this case, x becomes [x, self.intercept\_scaling], i.e. a “synthetic” feature with constant value equal to intercept\_scaling is appended to the instance vector. The intercept becomes intercept\_scaling \* synthetic\_feature\_weight.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept\_scaling has to be increased.

**class\_weight*dict or ‘balanced’, default=None***

Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one.

The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y)).

Note that these weights will be multiplied with sample\_weight (passed through the fit method) if sample\_weight is specified.

#### New in version 0.17: class\_weight=’balanced’ random\_stateint, RandomState instance, default=None

|  |  |  |
| --- | --- | --- |
| Used when | solver | == ‘sag’, ‘saga’ or ‘liblinear’ to shuffle the data. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-random_state) for |
| details. | | |

**solver*{‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’}, default=’lbfgs’***

Algorithm to use in the optimization problem. Default is ‘lbfgs’. To choose a solver, you might want to consider the following aspects:

* For small datasets, ‘liblinear’ is a good choice, whereas ‘sag’ and ‘saga’ are faster for large ones;
* For multiclass problems, only ‘newton-cg’, ‘sag’, ‘saga’ and ‘lbfgs’ handle multinomial loss;
* ‘liblinear’ is limited to one-versus-rest schemes.

|  |  |
| --- | --- |
| **Warning**  The choice of the algorithm depends on the penalty chosen: Supported penalties by solver: | |
|  | * ‘newton-cg’ - [‘l2’, ‘none’] * ‘lbfgs’ - [‘l2’, ‘none’] * ‘liblinear’ - [‘l1’, ‘l2’] * ‘sag’ - [‘l2’, ‘none’] * ‘saga’ - [‘elasticnet’, ‘l1’, ‘l2’, ‘none’] |
| **Note**  ‘sag’ and ‘saga’ fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from [**sklearn.preprocessing**.](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing) | |
|  | |

#### max\_iterint, default=100

Maximum number of iterations taken for the solvers to converge.

**multi\_class*{‘auto’, ‘ovr’, ‘multinomial’}, default=’auto’***

If the option chosen is ‘ovr’, then a binary problem is fit for each label. For ‘multinomial’ the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. ‘multinomial’ is unavailable when solver=’liblinear’. ‘auto’ selects ‘ovr’ if the data is binary, or if solver=’liblinear’, and otherwise selects ‘multinomial’.

*New in version 0.18:* Stochastic Average Gradient descent solver for ‘multinomial’ case.

|  |
| --- |
| *Changed in version 0.22:* Default changed from ‘ovr’ to ‘auto’ in 0.22. |

**verbose*int, default=0***

For the liblinear and lbfgs solvers set verbose to any positive number for verbosity.

#### warm\_startbool, default=False

When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Useless for liblinear solver. See [the Glossary.](https://scikit-learn.org/stable/glossary.html#term-warm_start)

*New in version 0.17: warm\_start* to support *lbfgs*, *newton-cg*, *sag*, *saga* solvers.

#### n\_jobsint, default=None

|  |  |  |  |
| --- | --- | --- | --- |
| parameter is ignored when the | solver | | is set to ‘liblinear’ regardless of whether |
| ‘multi\_class’ is specified or not.  a[**joblib.parallel\_backend**](https://joblib.readthedocs.io/en/latest/parallel.html#joblib.parallel_backend) more details. | |  | | --- | | None | | means 1 unless in | |
| context. | |  | | --- | | -1 | | means using all processors. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-n_jobs) for |
|  |

Number of CPU cores used when parallelizing over classes if multi\_class=’ovr’”. This **l1\_ratio*float, default=None***

The Elastic-Net mixing parameter, with 0 <= l1\_ratio <= 1. Only used if penalty='elasticnet'. Setting l1\_ratio=0 is equivalent to using penalty='l2', while setting l1\_ratio=1 is equivalent to using penalty='l1'. For 0 < l1\_ratio <1, the penalty is a combination of L1 and L2.

#### Attributes classes\_ndarray of shape (n\_classes, )

A list of class labels known to the classifier.

#### coef\_ndarray of shape (1, n\_features) or (n\_classes, n\_features)

Coefficient of the features in the decision function.

coef\_ is of shape (1, n\_features) when the given problem is binary. In particular, when multi\_class='multinomial', coef\_ corresponds to outcome 1 (True) and coef\_ corresponds to outcome 0 (False).

#### intercept\_ndarray of shape (1,) or (n\_classes,)

Intercept (a.k.a. bias) added to the decision function.

If fit\_intercept is set to False, the intercept is set to zero. intercept\_ is of shape (1,) when the given problem is binary. In particular, when multi\_class='multinomial', intercept\_ corresponds to outcome 1 (True) and intercept\_ corresponds to outcome 0 (False).

**n\_features\_in\_*int***

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

#### New in version 0.24. feature\_names\_in\_ndarray of shape (n\_features\_in\_,)

|  |  |
| --- | --- |
| Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when | X has feature names that are all |
| strings. | |

*New in version 1.0.*

#### n\_iter\_ndarray of shape (n\_classes,) or (1, )

Actual number of iterations for all classes. If binary or multinomial, it returns only 1 element. For liblinear solver, only the maximum number of iteration across all classes is given.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Changed in version 0.20:* In SciPy <= 1.0.0 the number of lbfgs iterations may | | | | | | |
| exceed | max\_iter | . | n\_iter\_ | will now report at most | max\_iter | . |

**Examples**

>>>

**>>>**

**from**

**sklearn.datasets**

**import**

load\_iris

**>>>**

**from**

**sklearn.linear\_model**

**import**

LogisticRegression

**>>>**

X, y

=

load\_iris(return\_X\_y

=

**True**

)

**>>>**

clf

=

LogisticRegression(random\_state

=

0

)

.

fit(X, y)

**>>>**

clf

.

predict(X[:

2

, :])

array([0, 0])

**>>>**

clf

.

predict\_proba(X[:

2

, :])

array([[9.8...e

-

, 1.8...e

01

-

02

, 1.4...e

-

,

08]

[9.7

...e

-

01

, 2.8...e

-

02

, ...e

-

08]])

**>>>**

clf

.

score(X, y)

0.97...

**Methods**

|  |  |
| --- | --- |
| [**decision\_function**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.decision_function)X) | Predict confidence scores for samples. |
| [**densify**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.densify)) | Convert coefficient matrix to dense array format. |
| [**fit**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.fit)X, y[, sample\_weight]) | Fit the model according to the given training data. |
| [**get\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.get_params)[deep]) | Get parameters for this estimator. |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.predict)X) | Predict class labels for samples in X. |
| [**predict\_log\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.predict_log_proba)X) | Predict logarithm of probability estimates. |
| [**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.predict_proba)X) | Probability estimates. |

[**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.score)X, y[, sample\_weight Return the mean accuracy on the given test data and

]) labels.

|  |  |
| --- | --- |
| [**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.set_params)\*\*params) | Set the parameters of this estimator. |

[**sparsify**(](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html?highlight=logistic%20regression#sklearn.linear_model.LogisticRegression.sparsify)) Convert coefficient matrix to sparse format.

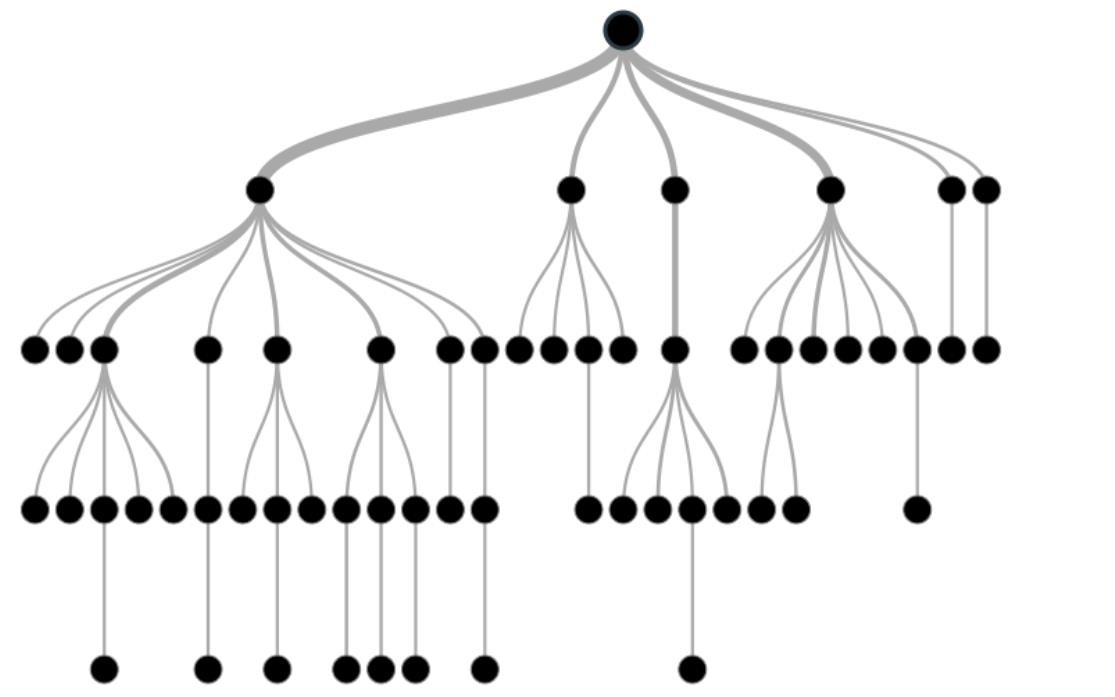
## 3. Decision Tree

### Introduction

Decision Tree is probably the easiest algorithm in Machine Learning. There’s not much mathematics involved here. Since it is very easy to use and interpret it is one of the most widely used and practical methods used in Machine Learning.

**What is a Decision Tree?**

It is a tool that has applications spanning several different areas. Decision trees can be used for classification as well as regression problems. The name itself suggests that it uses a flowchart like a tree structure to show the predictions that result from a series of feature-based splits. It starts with a root node and ends with a decision made by leaves.



Before learning more about decision trees let’s get familiar with some of the terminologies.

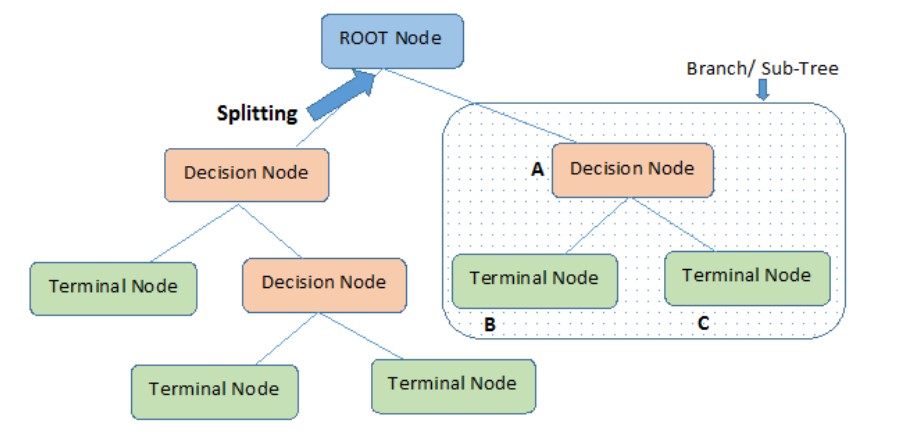
**Root Nodes** – It is the node present at the beginning of a decision tree from this node the population starts dividing according to various features.

**Decision Nodes** – the nodes we get after splitting the root nodes are called Decision Node

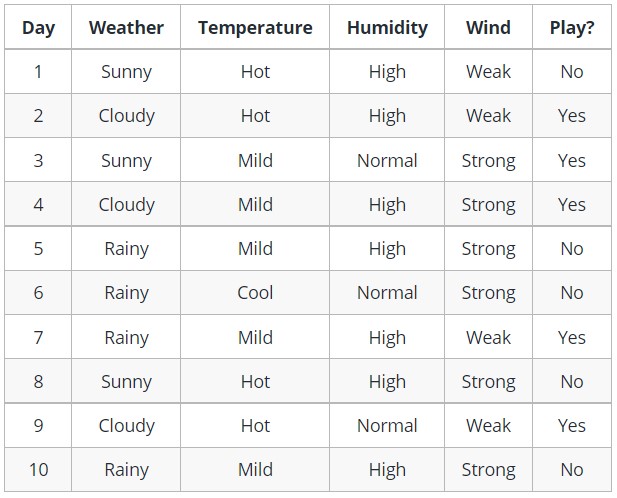
**Leaf Nodes** – the nodes where further splitting is not possible are called leaf nodes or terminal nodes

**Sub-tree** – just like a small portion of a graph is called sub-graph similarly a sub-section of this decision tree is called sub-tree.

**Pruning** – is nothing but cutting down some nodes to stop overfitting.

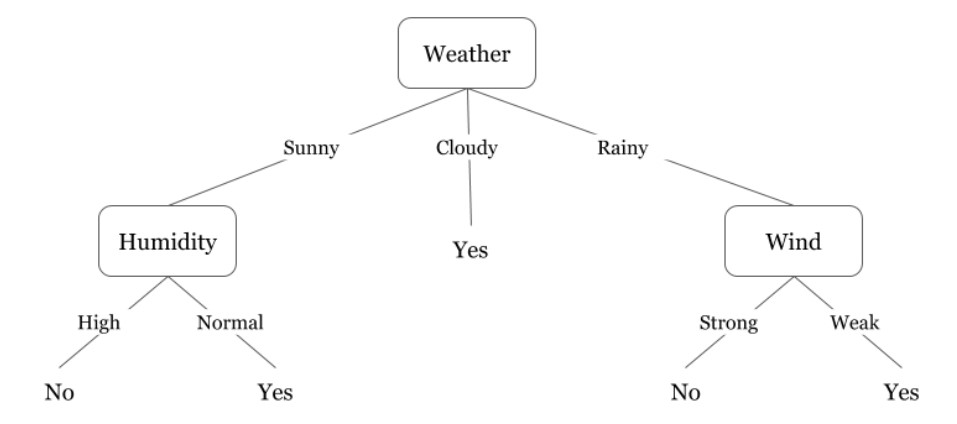


### Example of a decision tree



Decision trees are upside down which means the root is at the top and then this root is split into various several nodes. Decision trees are nothing but a bunch of if-else statements in layman terms. It checks if the condition is true and if it is then it goes to the next node attached to that decision.

In the below diagram the tree will first ask what is the weather? Is it sunny, cloudy, or rainy? If yes then it will go to the next feature which is humidity and wind. It will again check if there is a strong wind or weak, if it’s a weak wind and it’s rainy then the person may go and play.



Did you notice anything in the above flowchart? We see that if the *weather is cloudy* then we must go to play. Why didn’t it split more? Why did it stop there?

To answer this question, we need to know about few more concepts like entropy, information gain, and Gini index. But in simple terms, I can say here that the output for the training dataset is always yes for cloudy weather, since there is no disorderliness here we don’t need to split the node further.

The goal of machine learning is to decrease uncertainty or disorders from the dataset and for this, we use decision trees.

But the question is how do I know what should be the root node? what should be the decision node? when should I stop splitting? To decide this, there is a metric called “Entropy” which is the amount of uncertainty in the dataset.

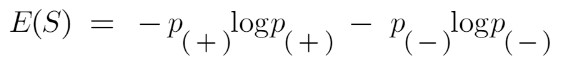
### Entropy

Entropy is nothing but the uncertainty in our dataset or measure of disorder. It can be explained using the following example:

Suppose you have a group of friends who decides which movie they can watch together on Sunday. There are 2 choices for movies, one is ***“Lucy”*** and the second is ***“Titanic”*** and now everyone has to tell their choice. After everyone gives their answer we see that *“Lucy” gets 4 votes* and *“Titanic” gets 5 votes*. Which movie do we watch now? Isn’t it hard to choose 1 movie now because the votes for both the movies are somewhat equal.

This is exactly what we call disorderness, there is an equal number of votes for both the movies, and we can’t really decide which movie we should watch. It would have been much easier if the votes for “Lucy” were 8 and for “Titanic” it was 2. Here we could easily say that the majority of votes are for “Lucy” hence everyone will be watching this movie.

In a decision tree, the output is mostly “yes” or “no” The formula for Entropy is shown below:



Here p+ is the probability of positive class p– is the probability of negative class

S is the subset of the training example

**How do Decision Trees use Entropy?**

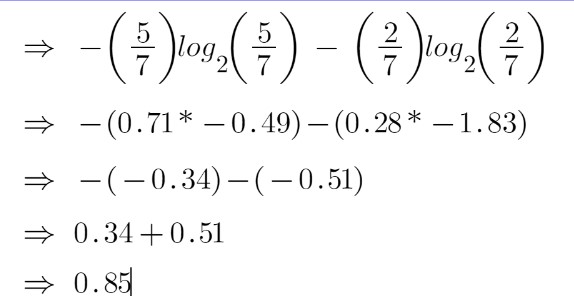
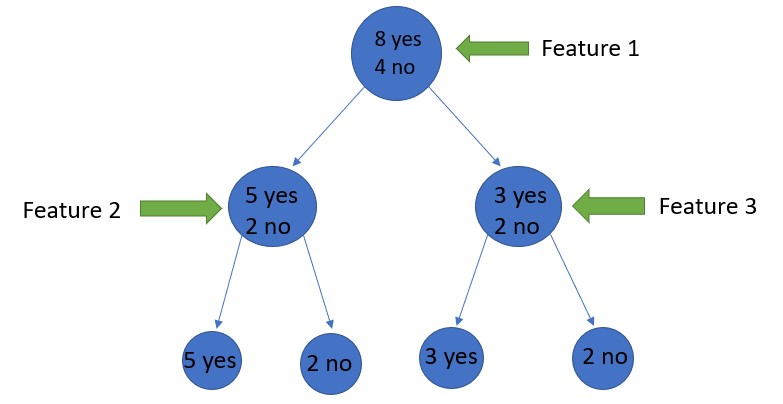
Next, we need to know that how exactly does entropy work in this algorithm.

Entropy basically measures the impurity of a node. Impurity is the degree of randomness; it tells how random our data is. A **pure sub-split** means that either you should be getting “yes”, or you should be getting “no”.

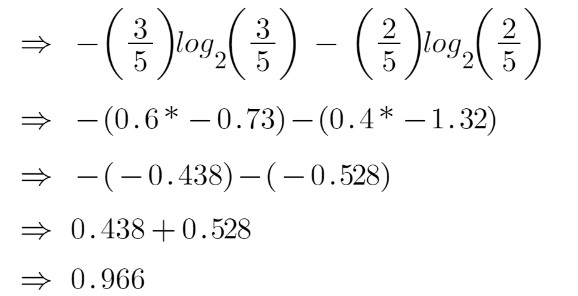
Suppose a *feature* has 8 “yes” and 4 “no” initially, after the first split the left node *gets 5 ‘yes’ and 2 ‘no’* whereas right node *gets 3 ‘yes’ and 2 ‘no’.*

We see here the split is not pure, why? Because we can still see some negative classes in both the nodes. In order to make a decision tree, we need to calculate the impurity of each split, and when the purity is 100%, we make it as a leaf node.

To check the impurity of feature 2 and feature 3 we will take the help for Entropy formula.



For feature 3,



We can clearly see from the tree itself that left node has low entropy or more purity than right node since left node has a greater number of “yes” and it is easy to decide here.

Always remember that the higher the Entropy, the lower will be the purity and the higher will be the impurity.

As mentioned earlier the goal of machine learning is to decrease the uncertainty or impurity in the dataset, here by using the entropy we are getting the impurity of a particular node, we don’t know if the parent entropy or the entropy of a particular node has decreased or not.

For this, we bring a new metric called “Information gain” which tells us how much the parent entropy has decreased after splitting it with some feature. **Information Gain**

Information gain measures the reduction of uncertainty given some feature and it is also a deciding factor for which attribute should be selected as a decision node or root node.



It is just entropy of the full dataset – entropy of the dataset given some feature.

To understand this better let’s consider an example:

Suppose our entire population has a total of 30 instances. The dataset is to predict whether the person will go to the gym or not. Let’s say 16 people go to the gym and 14 people don’t

Now we have two features to predict whether he/she will go to the gym or not.

Feature 1 is **“Energy”** which takes two values *“high” and “low”*

Feature 2 is **“Motivation”** which takes 3 values *“No motivation”, “Neutral” and “Highly motivated”.*

Let’s see how our decision tree will be made using these 2 features. We’ll use information gain to decide which feature should be the root node and which feature should be placed after the split.

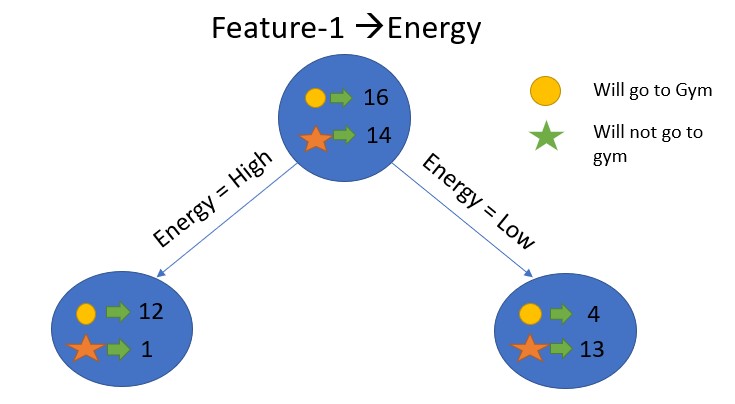
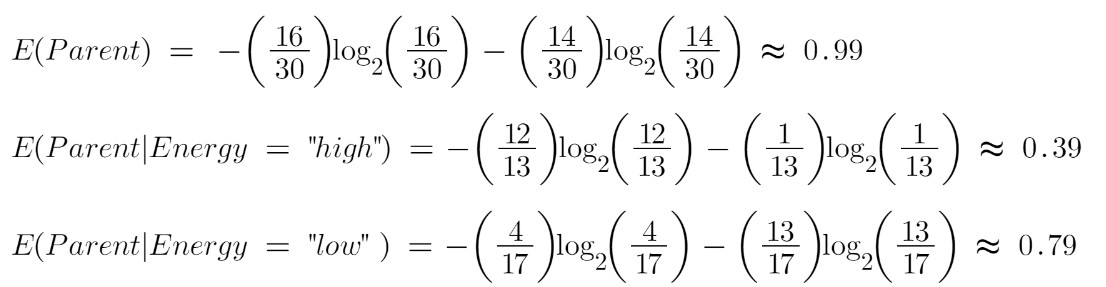
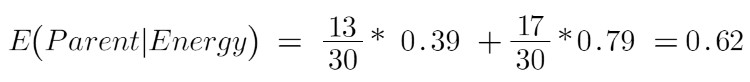


Image Source: Author

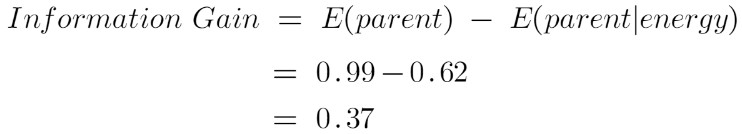
Let’s calculate the entropy:



To see the weighted average of entropy of each node we will do as follows:



Now we have the value of E(Parent) and E(Parent|Energy), information gain will be:



Our parent entropy was near 0.99 and after looking at this value of information gain, we can say that the entropy of the dataset will decrease by 0.37 if we make “Energy” as our root node.

Similarly, we will do this with the other feature “Motivation” and calculate its information gain.

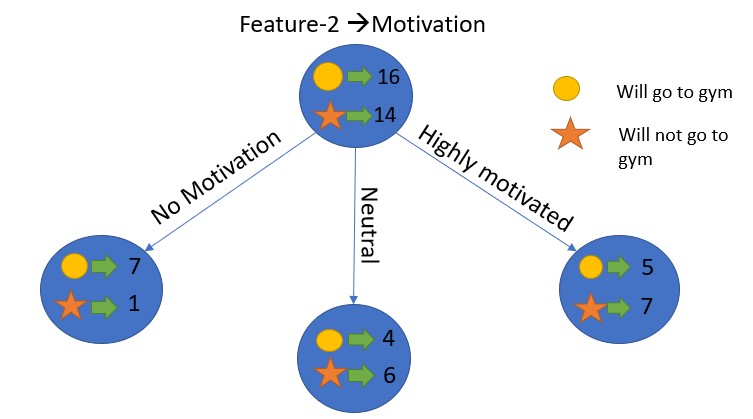
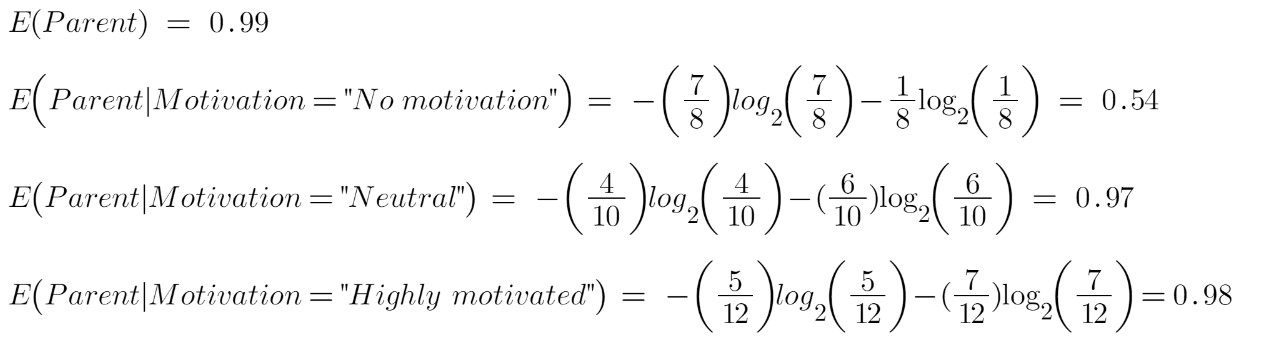
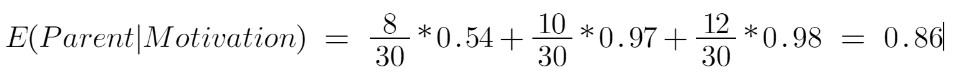


Image Source: Author

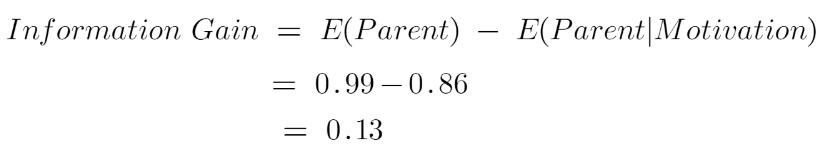
Let’s calculate the entropy here:



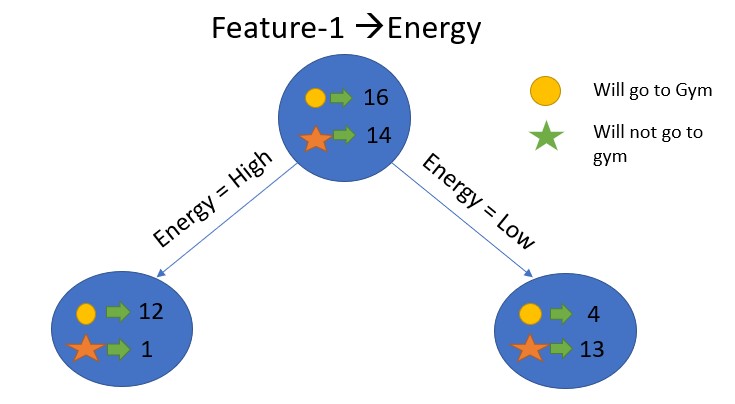
To see the weighted average of entropy of each node we will do as follows:



Now we have the value of E(Parent) and E(Parent|Motivation), information gain will be:



We now see that the “Energy” feature gives more reduction which is 0.37 than the “Motivation” feature. Hence we will select the feature which has the highest information gain and then split the node based on that feature.



In this example “Energy” will be our root node and we’ll do the same for subnodes. Here we can see that when the energy is “high” the entropy is low and hence we can say a person will definitely go to the gym if he has high energy, but what if the energy is low? We will again split the node based on the new feature which is “Motivation”.

**When to stop splitting?**

You must be asking this question to yourself that when do we stop growing our tree? Usually, real-world datasets have a large number of features, which will result in a large number of splits, which in turn gives a huge tree. Such trees take time to build and can lead to overfitting. That means the tree will give very good accuracy on the training dataset but will give bad accuracy in test data.

There are many ways to tackle this problem through hyperparameter tuning. We can set the maximum depth of our decision tree using the ***max\_depth*** parameter. The more the value of ***max\_depth***, the more complex your tree will be. The training error will off-course decrease if we increase the ***max\_depth*** value but when our test data comes into the picture, we will get a very bad accuracy. Hence you need a value that will not overfit as well as underfit our data and for this, you can use GridSearchCV.

Another way is to set the minimum number of samples for each spilt. It is denoted by ***min\_samples\_split***. Here we specify the minimum number of samples required to do a spilt. For example, we can use a minimum of 10 samples to reach a decision. That means if a node has less than 10 samples then using this parameter, we can stop the further splitting of this node and make it a leaf node.

There are more hyperparameters such as :

***min\_samples\_leaf*** – represents the minimum number of samples required to be in the leaf node. The more you increase the number, the more is the possibility of overfitting.

***max\_features*** – it helps us decide what number of features to consider when looking for the best split.

To read more about these hyperparameters you can read it [here.](https://www.youtube.com/watch?v=XABw4Y3GBR4)

### Pruning

It is another method that can help us avoid overfitting. It helps in improving the performance of the tree by cutting the nodes or sub-nodes which are not significant. It removes the branches which have very low importance.

There are mainly 2 ways for pruning:

1. **Pre-pruning** – we can stop growing the tree earlier, which means we can prune/remove/cut a node if it has low importance **while growing** the tree.
2. **Post-pruning** – once our **tree is built to its depth**, we can start pruning the nodes based on their significance.

**Advantages of the Decision Tree**

* + It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
  + It can be very useful for solving decision-related problems.
  + It helps to think about all the possible outcomes for a problem.
  + There is less requirement of data cleaning compared to other algorithms.

**Disadvantages of the Decision Tree**

* + The decision tree contains lots of layers, which makes it complex.
  + It may have an overfitting issue, which can be resolved using the **Random Forest algorithm.**
  + For more class labels, the computational complexity of the decision tree may increase.

|  |
| --- |
| [sklearn.tree**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.tree)**DecisionTreeClassifier** |
| *class* sklearn.tree.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*) |

A decision tree classifier.

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

**Parameters**

The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the 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\_depthint, 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\_splitint or float, default=2

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

* If int, then consider min\_samples\_split as the minimum number.
* If float, then min\_samples\_split is a fraction and ceil(min\_samples\_split \* n\_samples) are the minimum number of samples for each split.

*Changed in version 0.18:* Added float values for fractions. **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.

* If int, then consider min\_samples\_leaf as the minimum number.
* If float, then min\_samples\_leaf is a fraction and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node.

*Changed in version 0.18:* Added float values for fractions. **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 int, then consider | | | | | max\_features | | features at each split. | | | | |
| • | If float, then | | max\_features | | | | is a fraction | | | | | |
| and | int(max\_features \* n\_features) | | | | | | | features are considered at | | | |
| each split. | | | | | | |
| • | If “auto”, then | | | | max\_features=sqrt(n\_features) | | | | | | | . |
| •  •  • | If “sqrt”, then If “log2”, then | | | | max\_features=sqrt(n\_features) | | | | | | . | |
| max\_features=log2(n\_features) | | | | | | | . |
| If None, then | | | max\_features=n\_features | | | | | | . | |

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features.

#### random\_stateint, 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".

When max\_features < n\_features, the algorithm will select max\_features at random at each split before finding the best split among them. But the best found split may vary across different runs, even if max\_features=n\_features. That is the case, if the improvement of the criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, random\_state has to be fixed to an integer. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-random_state) for details.

#### max\_leaf\_nodesint, 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.

### min\_impurity\_decrease*float, default=0.0*

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

N\_t / N \* (impurity - N\_t\_R / N\_t \* right\_impurity

- N\_t\_L / N\_t \* left\_impurity)

where N is the total number of samples, N\_t is the number of samples at the current node, N\_t\_L is the number of samples in the left child, and N\_t\_R is the number of samples in the right child.

N, N\_t, N\_t\_R and N\_t\_L all refer to the weighted sum, if sample\_weight is passed.

*New in version 0.19.*

#### class\_weightdict, list of dict or “balanced”, default=None

Weights associated with classes in the form {class\_label: weight}. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y))

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample\_weight (passed through the fit method) if sample\_weight is specified.

#### ccp\_alphanon-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. See [Minimal Cost-Complexity Pruning](https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning) for details.

*New in version 0.22.*

|  |
| --- |
| **Attributes** |

#### classes\_ndarray of shape (n\_classes,) or list of ndarray

The classes labels (single output problem), or a list of arrays of class labels (multi-output problem).

[**feature\_importances\_*n***](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.feature_importances_)***darray of shape (n\_features,)*** Return the feature importances.

### max\_features\_*int*

The inferred value of max\_features.

#### n\_classes\_int or list of int

The number of classes (for single output problems), or a list containing the number of classes for each output (for multi-output problems).

[**n\_features\_*i***](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.n_features_)***nt***

DEPRECATED: The attribute n\_features\_ is deprecated in 1.0 and will be removed in 1.2.

### n\_features\_in\_*int*

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

*New in version 0.24.*

#### feature\_names\_in\_ndarray of shape (n\_features\_in\_,)

Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when X has feature names that are all strings.

*New in version 1.0.* **n\_outputs\_*int***

The number of outputs when fit is performed.

#### tree\_Tree instance

The underlying Tree object. Please refer to help(sklearn.tree.\_tree.Tree) for attributes of Tree object and [Understanding the decision tree structure](https://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.html#sphx-glr-auto-examples-tree-plot-unveil-tree-structure-py) for basic usage of these attributes.

### Notes

The default values for the parameters controlling the size of the trees (e.g. max\_depth, min\_samples\_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The [**predict**](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.predict) method operates using the [**numpy.argmax**](https://numpy.org/doc/stable/reference/generated/numpy.argmax.html#numpy.argmax) function on the outputs of [**predict\_proba**.](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.predict_proba) This means that in case the highest predicted probabilities are tied, the classifier will predict the tied class with the lowest index in [classes\_.](https://scikit-learn.org/stable/glossary.html#term-classes_)

### Examples

**>>> from** **sklearn.datasets** **import** load\_iris

**>>> from** **sklearn.model\_selection** **import** cross\_val\_score

**>>> from** **sklearn.tree** **import** DecisionTreeClassifier

**>>>** clf = DecisionTreeClassifier(random\_state=0)

**>>>** iris = load\_iris()

**>>>** cross\_val\_score(clf, iris.data, iris.target, cv=10)

**...**

**...**

array([ 1. , 0.93..., 0.86..., 0.93..., 0.93...,

0.93..., 0.93..., 1. , 0.93..., 1. ])

### Methods

|  |  |
| --- | --- |
| [**apply**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.apply)X[, check\_input]) | Return the index of the leaf that each sample is predicted as. |

[**cost\_complexity\_pruning\_pa**](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.cost_complexity_pruning_path)[C](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.cost_complexity_pruning_path)ompute the pruning path during Minimal Cost[**th**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.cost_complexity_pruning_path)X, y[, ...]) Complexity Pruning.

|  |  |
| --- | --- |
| [**decision\_path**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.decision_path)X[, check\_inpu  Return the decision path in the tree. t]) | |
| [**fit**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.fit)X, y[, sample\_weight, chec k\_input, ...]) | Build a decision tree classifier from the training set (X, y). |
| [**get\_depth**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.get_depth)) | Return the depth of the decision tree. |
| [**get\_n\_leaves**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.get_n_leaves)) | Return the number of leaves of the decision tree. |
| [**get\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.get_params)[deep]) | Get parameters for this estimator. |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.predict)X[, check\_input]) | Predict class or regression value for X. |
| [**predict\_log\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.predict_log_proba)X) | Predict class log-probabilities of the input samples X. |

[**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.predict_proba)X[, check\_inpu

Predict class probabilities of the input samples X. t])

|  |
| --- |
| Return the mean accuracy on the given test data [**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.score)X, y[, sample\_weight]) and labels. |

[**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html?highlight=decision%20tree#sklearn.tree.DecisionTreeClassifier.set_params)\*\*params) Set the parameters of this estimator.

## 4.RandomForestClassifier

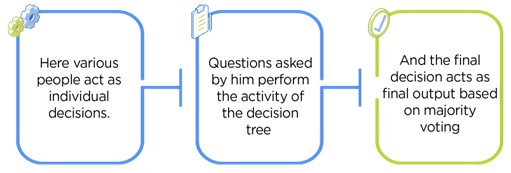
### Introduction

Random forest is a **Supervised Machine Learning Algorithm** that is **used widely in Classification and Regression problems**. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression.

One of the most important features of the Random Forest Algorithm is that it can handle the data set containing **continuous variables** as in the case of regression and **categorical variables** as in the case of classification. It performs better results for classification problems.

### Real Life Analogy

Let’s dive into a real-life analogy to understand this concept further. A student named X wants to choose a course after his 10+2, and he is confused about the choice of course based on his skill set. So he decides to consult various people like his cousins, teachers, parents, degree students, and working people. He asks them varied questions like why he should choose, job opportunities with that course, course fee, etc. Finally, after consulting various people about the course he decides to take the course suggested by most of the people.

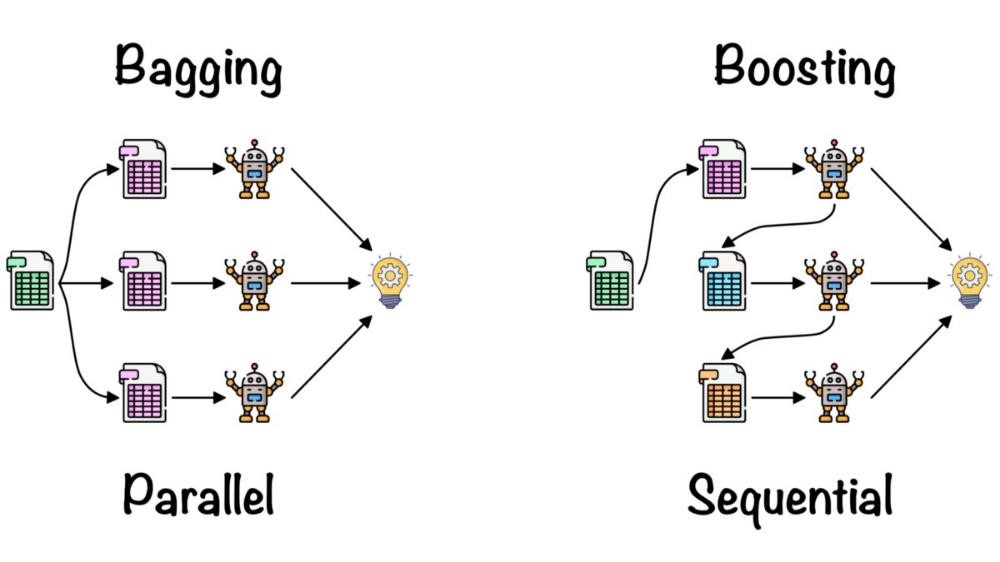


### Working of Random Forest Algorithm

Before understanding the working of the random forest we must look into the ensemble technique. **Ensemble** simplymeans combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.

Ensemble uses two types of methods:

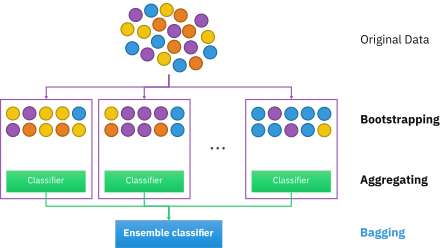
1. **Bagging**– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.
2. **Boosting**– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XG BOOST



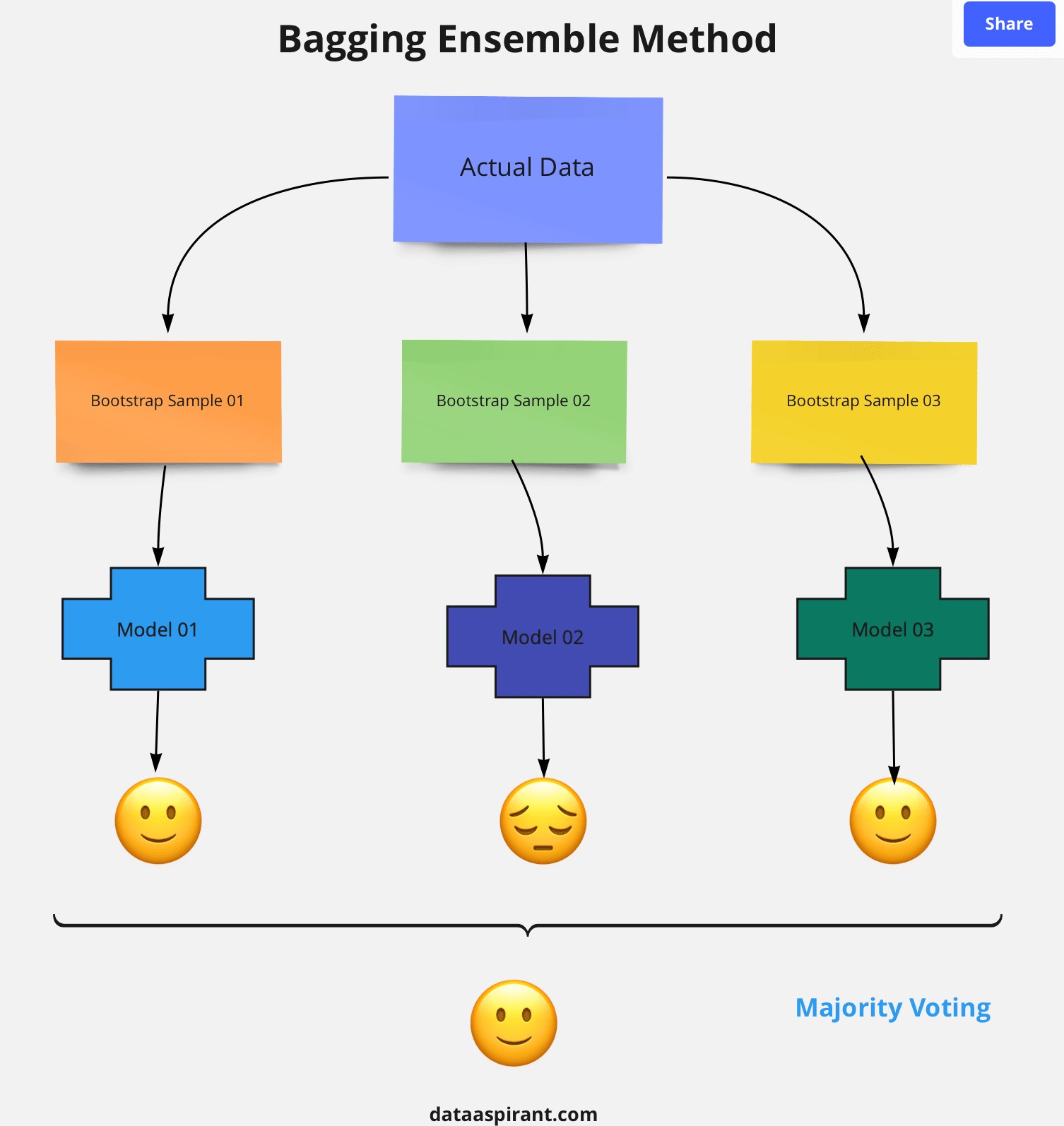
As mentioned earlier, Random forest works on the Bagging principle.

### Bagging

Bagging, also known as **Bootstrap Aggregation** is the ensemble technique used by random forest.Bagging chooses a random sample from the data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as **row sampling**. This step of row sampling with replacement is called **bootstrap**. Now each model is trained independently which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting is known as **aggregation**.



Now let’s look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won’t contain unique data. Now the model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now Happy emoji is having a majority when compared to sad emoji. Thus based on majority voting final output is obtained as Happy emoji.



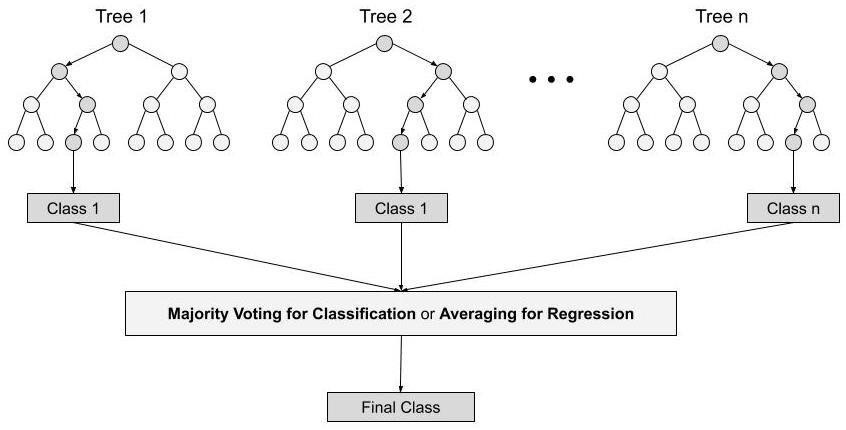
**Steps involved in random forest algorithm:**

Step 1: In Random forest n number of random records are taken from the data set having k number of records.

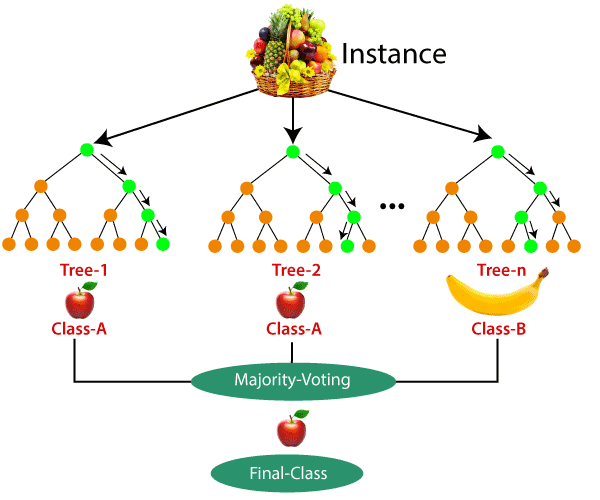
Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on **Majority Voting or Averaging** for Classification and regression respectively.



For example: consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket and an individual decision tree is constructed for each sample. Each decision tree will generate an output as shown in the figure. The final output is considered based on majority voting. In the below figure you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.



Important Features of Random Forest

1. **Diversity-** Not all attributes/variables/features are considered while making an individual tree, each tree is different.
2. **Immune to the curse of dimensionality-** Since each tree does not consider all the features, the feature space is reduced.
3. **Parallelization-**Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.
4. **Train-Test split-** In a random forest we don’t have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.
5. **Stability-** Stability arises because the result is based on majority voting/ averaging.

**Difference Between Decision Tree & Random Forest**

Random forest is a collection of decision trees; still, there are a lot of differences in their behavior.

|  |  |
| --- | --- |
| **Decision trees** | **Random Forest** |
| 1. Decision trees normally suffer from the problem of overfitting if it’s allowed to grow without any control. | 1. Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of. |
| 2. A single decision tree is faster in computation. | 2. It is comparatively slower. |
| 3. When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction. | 3. Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn’t use any set of formulas. |
| Thus random forests are much more successful than decision trees only if the | |

trees are diverse and acceptable.

### Important Hyperparameters

Hyperparameters are used in random forests to either enhance the performance and predictive power of models or to make the model faster.

Following hyperparameters increases the predictive power:

1. **n\_estimators**– number of trees the algorithm builds before averaging the predictions.
2. **max\_features**– maximum number of features random forest considers splitting a node.
3. **mini\_sample\_leaf**– determines the minimum number of leaves required to split an internal node.

Following hyperparameters increases the speed:

1. **n\_jobs**– it tells the engine how many processors it is allowed to use. If the value is 1, it can use only one processor but if the value is -1 there is no limit.
2. **random\_state**– controls randomness of the sample. The model will always produce the same results if it has a definite value of random state and if it has been given the same hyperparameters and the same training data.
3. **oob\_score** – OOB means out of the bag. It is a random forest crossvalidation method. In this one-third of the sample is not used to train the data instead used to evaluate its performance. These samples are called out of bag samples.

|  |
| --- |
| [sklearn.ensemble**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble)**RandomForestClassifier** |
| *class* sklearn.ensemble.RandomForestClassifier(*n\_estimators=100*, *\**, *criterion='gini'*, *max\_dept h=None*, *min\_samples\_split=2*, *min\_samples\_leaf=1*, *min\_weight\_fraction\_leaf=0.0*, *max\_featur es='auto'*, *max\_leaf\_nodes=None*, *min\_impurity\_decrease=0.0*, *bootstrap=True*, *oob\_score=Fals e*, *n\_jobs=None*, *random\_state=None*, *verbose=0*, *warm\_start=False*, *class\_weight=None*, *ccp\_a lpha=0.0*, *max\_samples=None*[)[source]](https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/ensemble/_forest.py#L1075) |

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max\_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree.

**Parameters**

**n\_estimators**

***int, default=100***

The number of trees in the forest.

|  |  |  |
| --- | --- | --- |
| *Changed in version 0.22:* The default value of in 0.22. | n\_estimators | changed from 10 to 100 |
|  |
| **criterion*{“gini”, “entropy”}, default=”gini”*** | | |

The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. Note: this parameter is treespecific.

#### max\_depthint, 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\_splitint or float, default=2

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

* If int, then consider min\_samples\_split as the minimum number.
* If float, then min\_samples\_split is a fraction and ceil(min\_samples\_split \* n\_samples) are the minimum number of samples for each split.

*Changed in version 0.18:* Added float values for fractions.

#### min\_samples\_leafint 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.

* If int, then consider min\_samples\_leaf as the minimum number.
* If float, then min\_samples\_leaf is a fraction and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node.

*Changed in version 0.18:* Added float values for fractions.

**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*{“auto”, “sqrt”, “log2”}, int or float, default=”auto”*** The number of features to consider when looking for the best split:

* If int, then consider max\_features features at each split.
* If float, then max\_features is a fraction and round(max\_features \* n\_features) features are considered at each split.
* If “auto”, then max\_features=sqrt(n\_features).
* If “sqrt”, then max\_features=sqrt(n\_features) (same as “auto”).
* If “log2”, then max\_features=log2(n\_features).
* If None, then max\_features=n\_features.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features.

#### max\_leaf\_nodesint, default=None

Grow trees 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.

**min\_impurity\_decrease*float, default=0.0***

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

N\_t / N \* (impurity - N\_t\_R / N\_t \* right\_impurity

- N\_t\_L / N\_t \* left\_impurity)

where N is the total number of samples, N\_t is the number of samples at the current node, N\_t\_L is the number of samples in the left child, and N\_t\_R is the number of samples in the right child.

N, N\_t, N\_t\_R and N\_t\_L all refer to the weighted sum, if sample\_weight is passed.

*New in version 0.19.*

#### bootstrapbool, default=True

Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree. **oob\_score*bool, default=False***

Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True. **n\_jobs*int, default=None***

The number of jobs to run in parallel. [**fit**,](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.fit) [**predict**,](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.predict) [**decision\_path**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.decision_path) and [**apply**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.apply) are all

|  |  |  |
| --- | --- | --- |
| parallelized over the trees. | None | means 1 unless in |
| a[**joblib.parallel\_backend**](https://joblib.readthedocs.io/en/latest/parallel.html#joblib.parallel_backend) context. -1 means using all processors. See [Glossary](https://scikit-learn.org/stable/glossary.html#term-n_jobs) for | | |

more details.

#### random\_stateint, RandomState instance or None, default=None

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| building trees (if | bootstrap=True | ) and the sampling of the features to consider when | | |
| looking for the best split at each node (if See [Glossary](https://scikit-learn.org/stable/glossary.html#term-random_state) for details. | | | max\_features < n\_features | ). |
|  |

Controls both the randomness of the bootstrapping of the samples used when **verbose*int, default=0***

Controls the verbosity when fitting and predicting.

#### warm\_startbool, default=False

|  |  |  |
| --- | --- | --- |
| When set to | True | , reuse the solution of the previous call to fit and add more |
| estimators to the ensemble, otherwise, just fit a whole new forest. See [the Glossary.](https://scikit-learn.org/stable/glossary.html#term-warm_start) | | |

**class\_weight*{“balanced”, “balanced\_subsample”}, dict or list of dicts, default=None***

Weights associated with classes in the form {class\_label: weight}. If not given, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

Note that for multioutput (including multilabel) weights should be defined for each class of every column in its own dict. For example, for four-class multilabel classification weights should be [{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}] instead of [{1:1}, {2:5}, {3:1}, {4:1}].

The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as n\_samples / (n\_classes \* np.bincount(y))

The “balanced\_subsample” mode is the same as “balanced” except that weights are computed based on the bootstrap sample for every tree grown.

For multi-output, the weights of each column of y will be multiplied.

Note that these weights will be multiplied with sample\_weight (passed through the fit method) if sample\_weight is specified. **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. See [Minimal Cost-Complexity Pruning](https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning) for details. | | |

*New in version 0.22.*

#### max\_samplesint or float, default=None

If bootstrap is True, the number of samples to draw from X to train each base estimator.

* If None (default), then draw X.shape[0] samples.
* If int, then draw max\_samples samples.
* If float, then draw max\_samples \* X.shape[0] samples.

Thus, max\_samples should be in the interval (0.0, 1.0].

*New in version 0.22.*

|  |
| --- |
| **Attributes** |

#### base\_estimator\_DecisionTreeClassifier

The child estimator template used to create the collection of fitted sub-estimators.

**estimators\_*list of DecisionTreeClassifier*** The collection of fitted sub-estimators.

#### classes\_ndarray of shape (n\_classes,) or a list of such arrays

The classes labels (single output problem), or a list of arrays of class labels (multioutput problem). **n\_classes\_*int or list***

The number of classes (single output problem), or a list containing the number of classes for each output (multi-output problem).

[**n\_features\_*i***](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.n_features_)***nt***

|  |  |  |
| --- | --- | --- |
| DEPRECATED: Attribute | n\_features\_ | was deprecated in version 1.0 and will be |
| removed in 1.2. | | |

**n\_features\_in\_*int***

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

*New in version 0.24.*

#### feature\_names\_in\_ndarray of shape (n\_features\_in\_,)

|  |  |
| --- | --- |
| Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when | X has feature names that are all |
| strings. | |

*New in version 1.0.*

**n\_outputs\_*int***

The number of outputs when fit is performed. [**feature\_importances\_*n***](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.feature_importances_)***darray of shape (n\_features,)*** The impurity-based feature importances.

**oob\_score\_*float***

Score of the training dataset obtained using an out-of-bag estimate. This attribute exists only when oob\_score is True.

#### oob\_decision\_function\_ndarray of shape (n\_samples, n\_classes) or (n\_samples, n\_classes, n\_outputs)

Decision function computed with out-of-bag estimate on the training set. If n\_estimators is small it might be possible that a data point was never left out during the bootstrap. In this case, oob\_decision\_function\_ might contain NaN. This attribute exists only when oob\_score is True.

**Notes**

The default values for the parameters controlling the size of the trees

(e.g. max\_depth, min\_samples\_leaf, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, max\_features=n\_features and bootstrap=False, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random\_state has to be fixed.

**Examples**

>>>

**>>> from** **sklearn.ensemble** **import** RandomForestClassifier

**>>> from** **sklearn.datasets** **import** make\_classification

**>>>** X, y = make\_classification(n\_samples=1000, n\_features=4,

**...**  n\_informative=2, n\_redundant=0,

**...**  random\_state=0, shuffle=**False**)

**>>>** clf = RandomForestClassifier(max\_depth=2, random\_state=0)

**>>>** clf.fit(X, y)

RandomForestClassifier(...)

**>>>** print(clf.predict([[0, 0, 0, 0]])) [1] **Methods**

|  |  |
| --- | --- |
| [**apply**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.apply)X) | Apply trees in the forest to X, return leaf indices. |
| [**decision\_path**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.decision_path)X) | Return the decision path in the forest. |
| [**fit**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.fit)X, y[, sample\_weight]) | Build a forest of trees from the training set (X, y). |
| [**get\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.get_params)[deep]) | Get parameters for this estimator. |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.predict)X) | Predict class for X. |
| [**predict\_log\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.predict_log_proba)X) | Predict class log-probabilities for X. |
| [**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.predict_proba)X) | Predict class probabilities for X. |

[**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.score)X, y[, sample\_weight]) Return the mean accuracy on the given test data and labels.

|  |  |
| --- | --- |
| [**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.set_params)\*\*params) | Set the parameters of this estimator. |

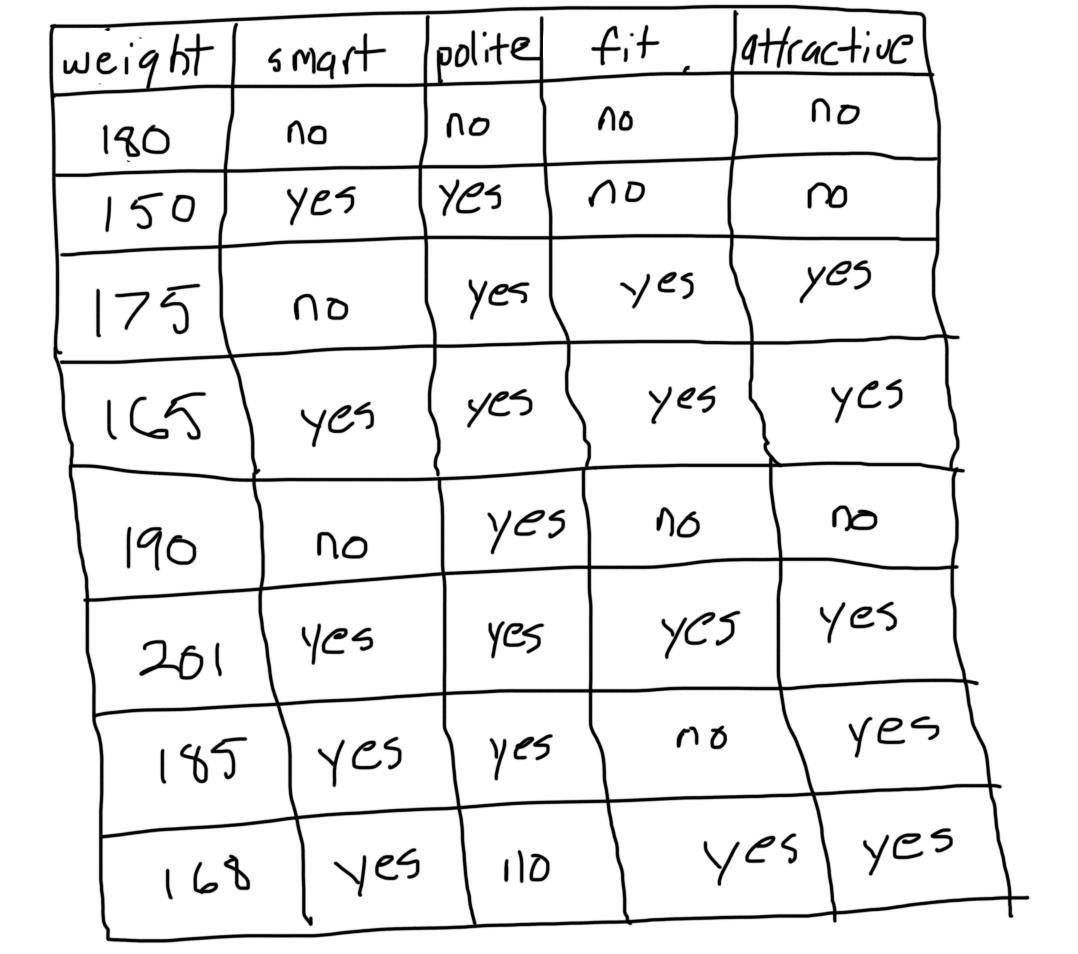
## 5.AdaBoost Classifier

AdaBoost Classifier Example In Python

The general idea behind boosting methods is to train predictors sequentially, each trying to correct its predecessor. The two most commonly used boosting algorithms are AdaBoost and Gradient Boosting. At a high level, AdaBoost is similar to Random Forest in that they both tally up the predictions made by each decision trees within the forest to decide on the final classification. There are however, some subtle differences. For instance, in AdaBoost, the decision trees have a depth of 1 (i.e. 2 leaves). In addition, the predictions made by each decision tree have varying impact on the final prediction made by the model.

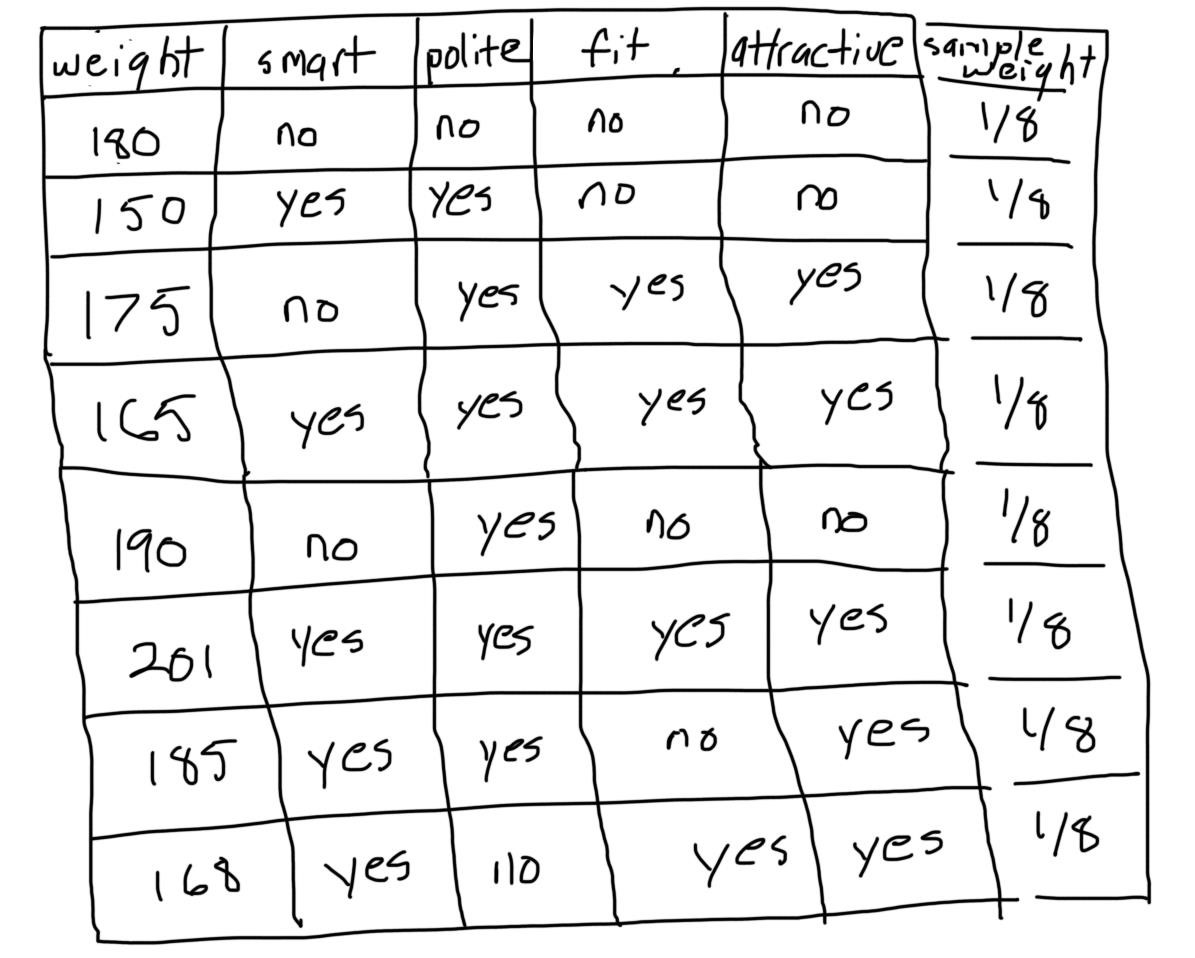
### Algorithm

In the proceeding example, we’ll be using a dataset that categories people as attractive or not based on certain features.



### Step 1: Initialize the sample weights

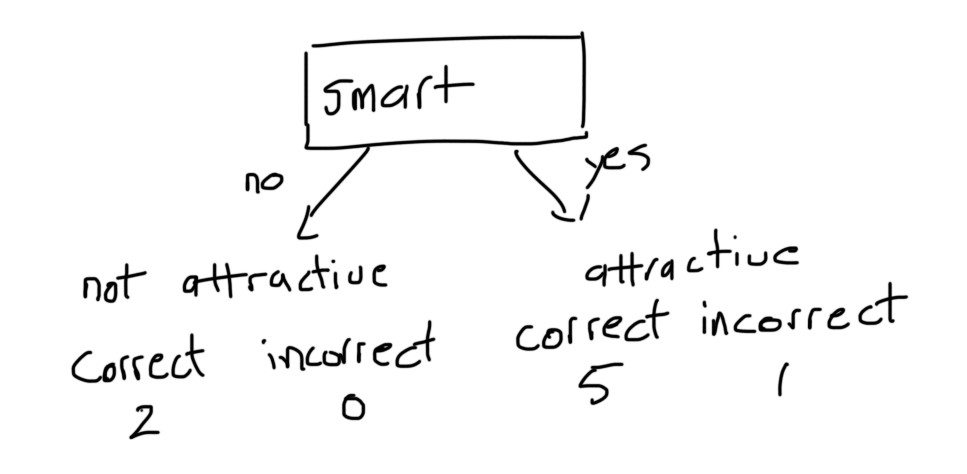
In first step of AdaBoost each sample is associated with a weight that indicates how important it is with regards to the classification. Initially, all the samples have identical weights (1 divided by the total number of samples).



### Step 2: Build a decision tree with each feature, classify the data and evaluate the result

Next, for each feature, we build a decision tree with a depth of 1. Then, we use every decision tree to classify the data. Afterwards, we compare the predictions made by each tree with the actual labels in the training set. The feature and corresponding tree that did the best job of classifying the training samples becomes the next tree in the forest.

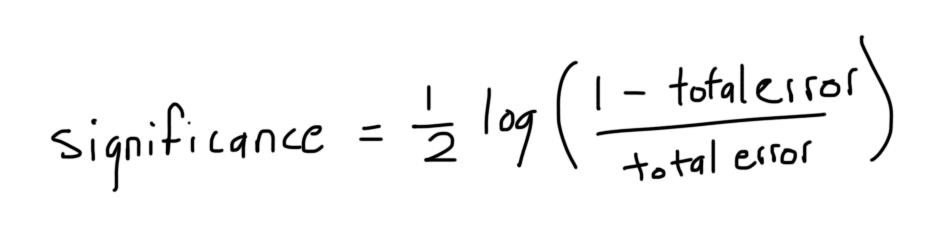
For example, assume that we built a tree that classifies people as attractive if they’re smart and unattractive if they’re not.



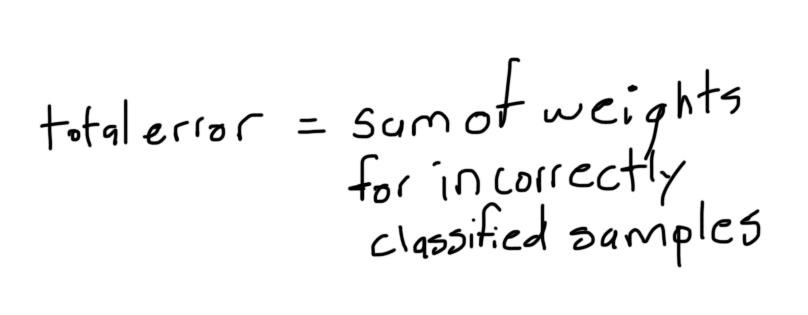
The decision tree incorrectly classified 1 person as being attractive based off the fact that they were smart. We repeat the process for all trees and select the one with the smallest number of incorrect predictions.

### Step 3: Calculate the significance of the tree in the final classification

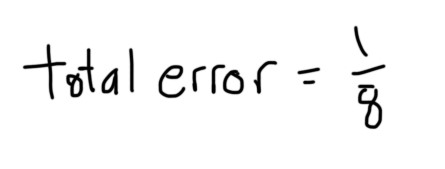
Once we have decided on a decision tree. We use the proceeding formula to calculate the amount of say the it has in the final classification.



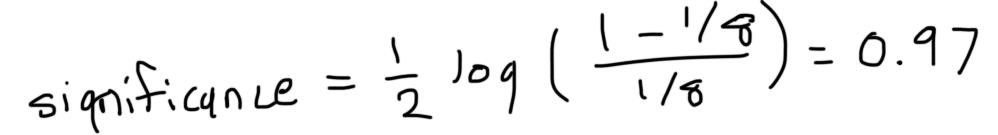
Where the total error is the sum of the weights of the incorrectly classified samples.



Going back to our example, total error would be equal to the following.



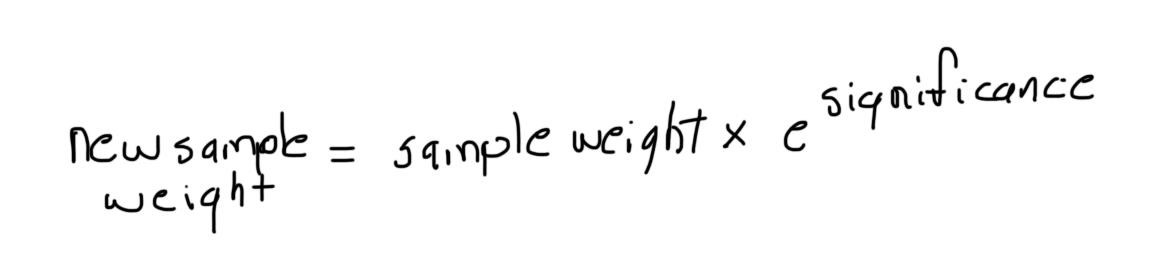
By plugging the error into our formula, we get:



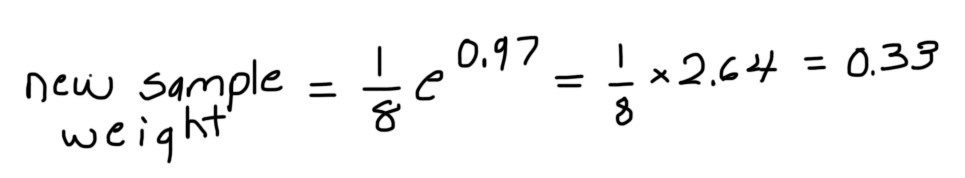
As we’ll see later, this number is used to determine the prediction made by the forest as a whole.

### Step 4: Update the sample weights so that the next decision tree will take the errors made by the preceding decision tree into account

We look at the samples that the current tree classified incorrectly and increase their associated weights using the following formula.



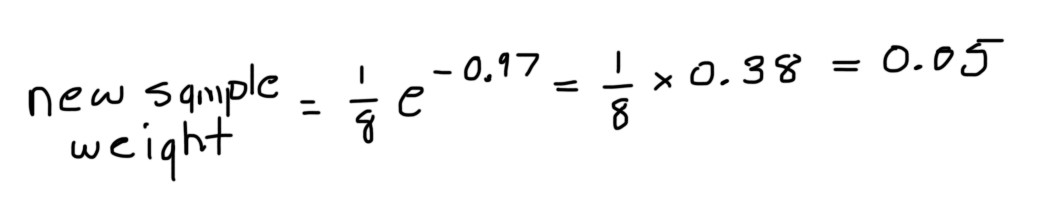
There’s nothing fancy going on here. We raise **e** to the power of the significance computed in the previous step because we want the new sample weight to grow exponentially.



Then, we look at the samples that the tree classified correctly and decrease their associated weights using the following formula.

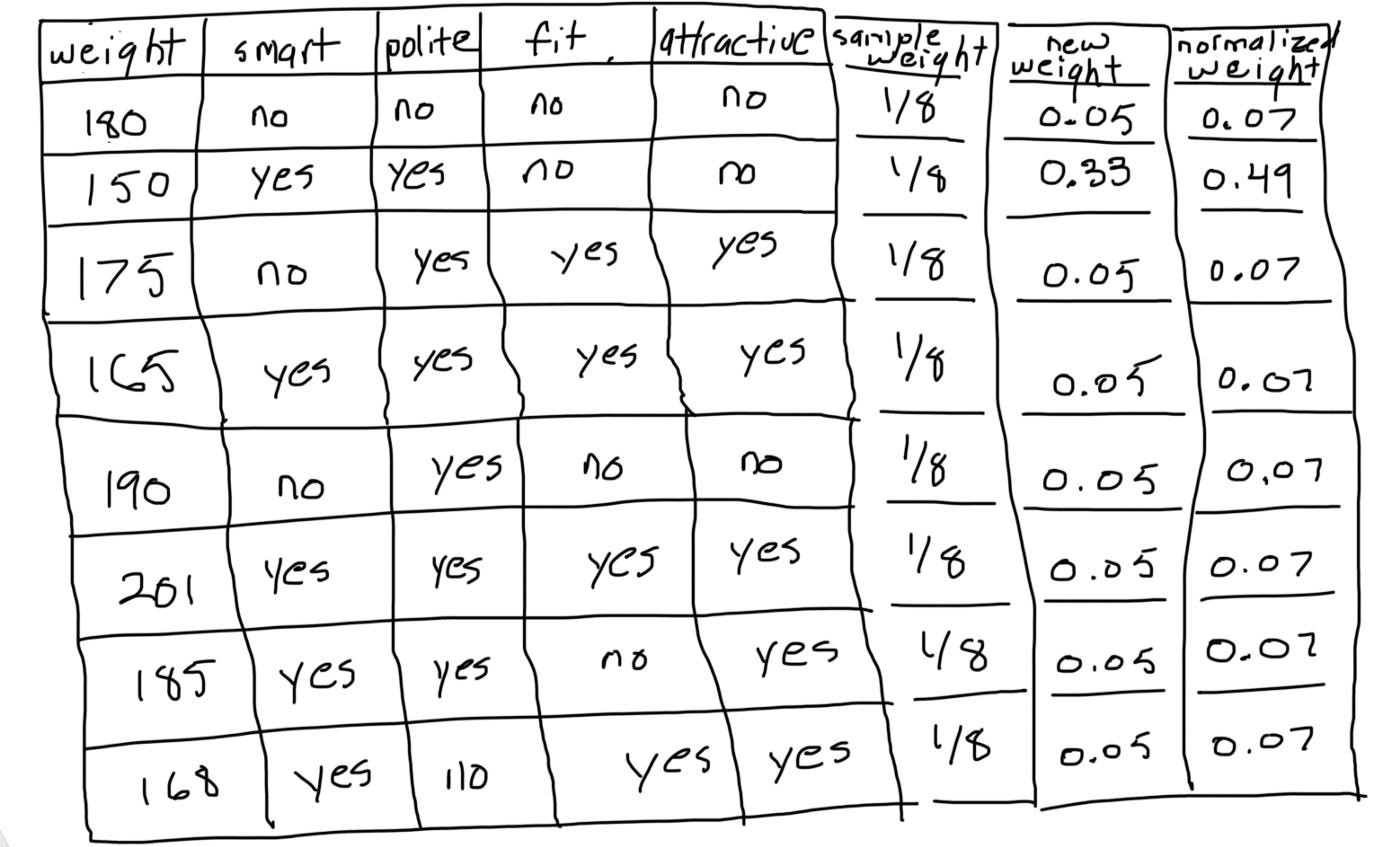


The equation is the same as before only this time, we raise **e** to the power of a negative exponent.



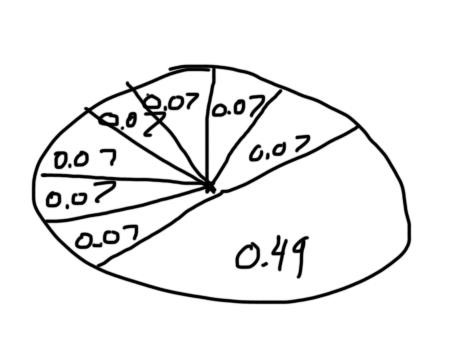
The main take away here is that the samples which the previous stump incorrectly classified should be associated with larger sample weights and the ones it classified correctly should be associated with smaller sample weights.

Notice how if we summed all the sample weights, we’d get a number that is smaller than 1. Thus, we normalize the new sample weights so that they add up to 1.

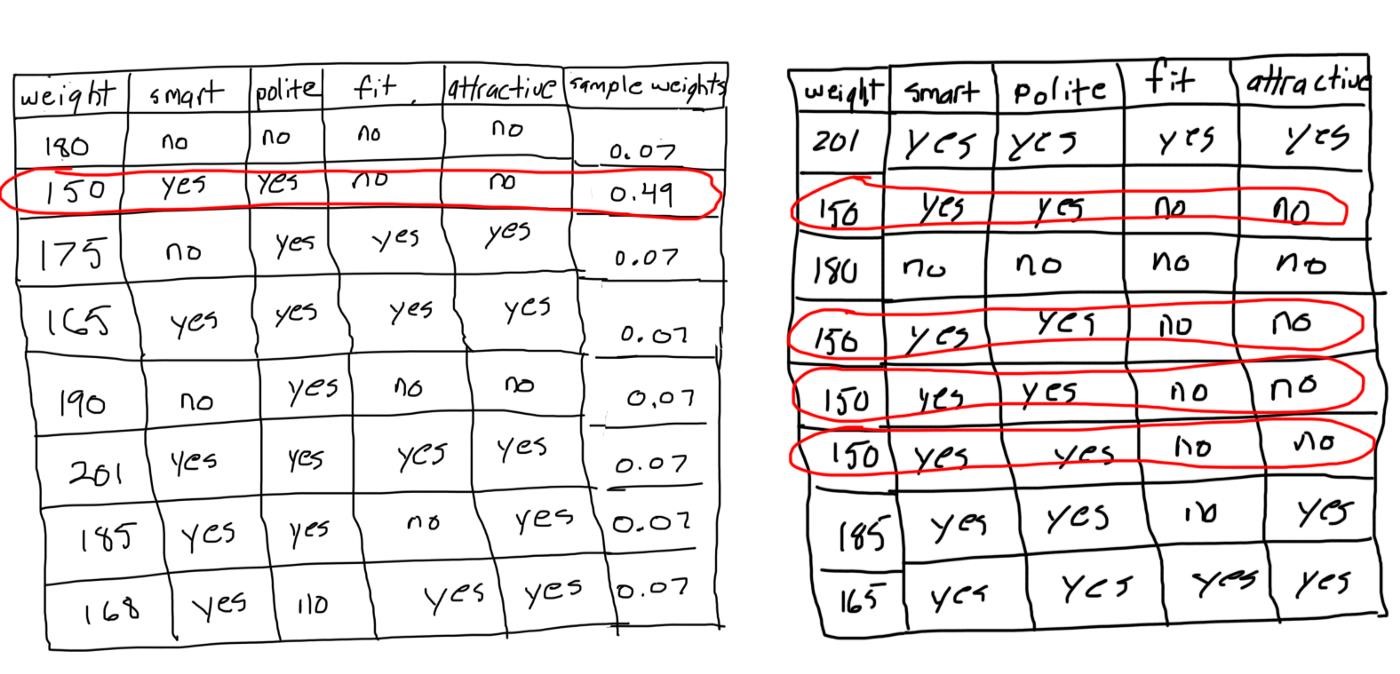


### Step 5: Form a new dataset

We start by making a new and empty dataset that is the same size as the original. Then, imagine a roulette table where each pocket corresponds to a sample weight. We select numbers between 0 and 1 at random. The location where each number falls determines which sample we place in the new dataset.



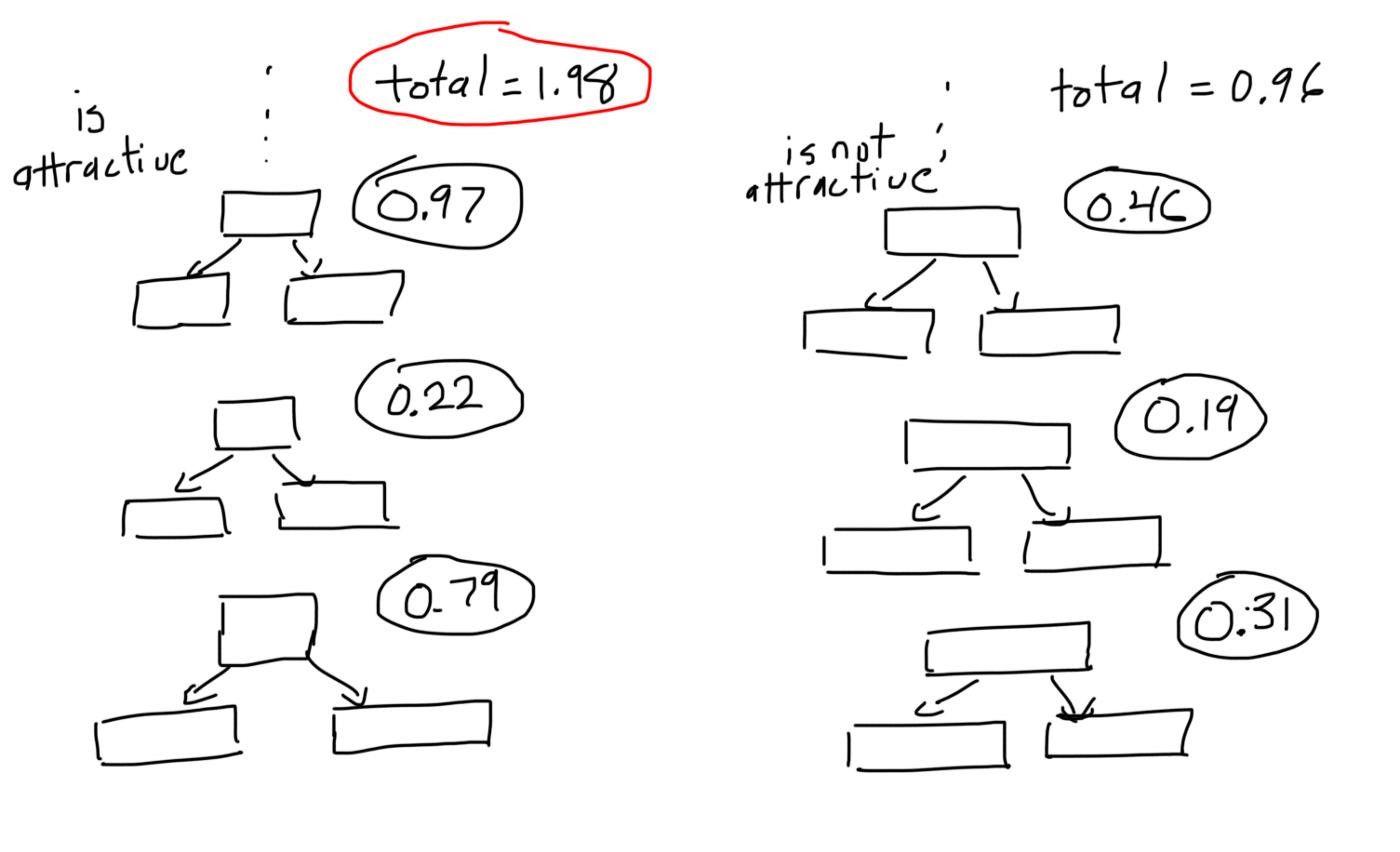
Since the samples that were incorrectly classified have higher weights in relation to the others, the likelihood that the random number falls under their slice of the distribution is greater. Therefore, the new dataset will have a tendency to contain multiple copies of the samples that were misclassified by the previous tree. As a result, when we go back to the step where we evaluate the predictions made by each decision tree, the one with the highest score will have correctly classified the samples the previous tree misclassified.



**Step 6: Repeat steps 2 through 5 until the number of iterations equals the number specified by the hyperparameter (i.e. number of estimators)**

### Step 7: Use the forest of decision trees to make predictions on data outside of the training set

The AdaBoost model makes predictions by having each tree in the forest classify the sample. Then, we split the trees into groups according to their decisions. For each group, we add up the significance of every tree inside the group. The final classification made by the forest as a whole is determined by the group with the largest sum.



|  |
| --- |
| [sklearn.ensemble**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble)**AdaBoostClassifier** |
| *class* sklearn.ensemble.AdaBoostClassifier(*base\_estimator=None*, *\**, *n\_estimators=50*, *learning*  *\_rate=1.0*, *algorithm='SAMME.R'*, *random\_state=None*[)[source]](https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/ensemble/_weight_boosting.py#L313) |

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

**Parameters base\_estimator*object, default=None***

The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper classes\_ and n\_classes\_ attributes. If None, then the base estimator is [**DecisionTreeClassifier**](https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier) initialized with max\_depth=1. **n\_estimators*int, default=50***

The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

#### learning\_ratefloat, default=1.0

Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learning\_rate and n\_estimators parameters. **algorithm*{‘SAMME’, ‘SAMME.R’}, default=’SAMME.R’***

If ‘SAMME.R’ then use the SAMME.R real boosting algorithm. base\_estimator must support calculation of class probabilities. If ‘SAMME’ then use the SAMME discrete boosting algorithm. The SAMME.R algorithm typically converges faster than SAMME, achieving a lower test error with fewer boosting iterations. **random\_state*int, RandomState instance or None, default=None***

Controls the random seed given at each base\_estimator at each boosting iteration. Thus, it is only used when base\_estimator exposes a random\_state. Pass an int for reproducible output across multiple function calls. See [Glossary.](https://scikit-learn.org/stable/glossary.html#term-random_state)

**Attributes**

**base\_estimator\_**

***estimator***

The base estimator from which the ensemble is grown.

#### estimators\_list of classifiers

The collection of fitted sub-estimators. **classes\_*ndarray of shape (n\_classes,)*** The classes labels.

**n\_classes\_*int***

The number of classes.

**estimator\_weights\_*ndarray of floats***

Weights for each estimator in the boosted ensemble.

**estimator\_errors\_*ndarray of floats***

Classification error for each estimator in the boosted ensemble.

[**feature\_importances\_*n***](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.feature_importances_)***darray of shape (n\_features,)*** The impurity-based feature importances.

**n\_features\_in\_*int***

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

*New in version 0.24.*

#### feature\_names\_in\_ndarray of shape (n\_features\_in\_,)

Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when X has feature names that are all strings.

**Examples**

>>>

**>>> from** **sklearn.ensemble** **import** AdaBoostClassifier

**>>> from** **sklearn.datasets** **import** make\_classification

**>>>** X, y = make\_classification(n\_samples=1000, n\_features=4,

**...**  n\_informative=2, n\_redundant=0,

**...**  random\_state=0, shuffle=**False**)

**>>>** clf = AdaBoostClassifier(n\_estimators=100, random\_state=0)

**>>>** clf.fit(X, y)

AdaBoostClassifier(n\_estimators=100, random\_state=0)

**>>>** clf.predict([[0, 0, 0, 0]]) array([1]) **>>>** clf.score(X, y) 0.983... **Methods**

|  |  |  |
| --- | --- | --- |
| [**decision\_function**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.decision_function)X) | Compute the decision function of | X. |
| [**fit**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.fit)X, y[, sample\_weight]) | Build a boosted classifier from the training set (X, y). | |
| [**get\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.get_params)[deep]) | Get parameters for this estimator. | |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.predict)X) | Predict classes for X. | |
| [**predict\_log\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.predict_log_proba)X) | Predict class log-probabilities for X. | |
| [**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.predict_proba)X) | Predict class probabilities for X. | |
| [**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.score)X, y[, sample\_weight]) | Return the mean accuracy on the given test data and labels. | |
| [**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.set_params)\*\*params) | Set the parameters of this estimator. | |
| [**staged\_decision\_function**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.staged_decision_function)X) | Compute decision function of X for each boosting iteration. | |
| [**staged\_predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.staged_predict)X) | Return staged predictions for X. | |
| [**staged\_predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.staged_predict_proba)X) | Predict class probabilities for X. | |

[**staged\_score**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html#sklearn.ensemble.AdaBoostClassifier.staged_score)X, y[, sample\_weigh

Return staged scores for X, y. t])

### 6.Gradient boosting classifiers

**What is Gradient Boosting?**

Let's start by briefly reviewing **ensemble learning**. Like the name suggests, ensemble learning involves building a strong model by using a collection (or "ensemble") of "weaker" models. Gradient boosting falls under the category of boosting methods, which iteratively learn from each of the weak learners to build a strong model. It can optimize:

* Regression
* Classification
* Ranking

The scope here will be limited to classification in particular.

The idea behind boosting comes from the intuition that weak learners could be modified in order to become better. [AdaBoost](https://blog.paperspace.com/adaboost-optimizer/) was the first boosting algorithm. AdaBoost and related algorithms were first cast in a statistical framework by [Leo Breiman (1997),](https://statistics.berkeley.edu/sites/default/files/tech-reports/486.pdf) which laid the foundation for other researchers such as [Jerome H. Friedman](https://statweb.stanford.edu/~jhf/ftp/trebst.pdf) to modify this work into the development of the gradient boosting algorithm for regression. Subsequently, many researchers developed this boosting algorithm for many more fields of machine learning and statistics, far beyond the initial applications in regression and classification.

The term "Gradient" in Gradient Boosting refers to the fact that you have two or more derivatives of the same function (we'll cover this in more detail later on). Gradient Boosting is an *iterative functional gradient algorithm*, i.e an algorithm which minimizes a loss function by iteratively choosing a function that points towards the negative gradient; a weak hypothesis.

**Gradient Boosting in Classification**

Over the years, gradient boosting has found applications across various technical fields. The algorithm can look complicated at first, but in most cases we use only one predefined configuration for classification and one for regression, which can of course be modified based on your requirements. In this article we'll focus on Gradient Boosting for classification problems. Firstly, we shall have a look at how the algorithm works behind-the-scenes, intuitively and mathematically.

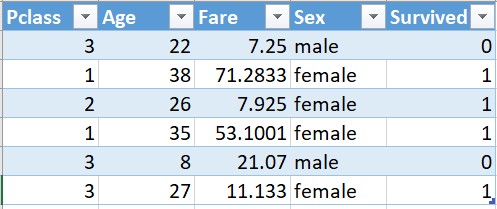
Gradient Boosting has three main components:

* **Loss Function** - The role of the loss function is to estimate how good the model is at making predictions with the given data. This could vary depending on the problem at hand. For example, if we're trying to predict the weight of a person depending on some input variables (a regression problem), then the loss function would be something that helps us find the difference between the predicted weights and the observed weights. On the other hand, if we're trying to categorize if a person will like a certain movie based on their personality, we'll require a loss function that helps us understand how accurate our model is at classifying people who did or didn't like certain movies.
* **Weak Learner** - A weak learner is one that classifies our data but does so poorly, perhaps no better than random guessing. In other words, it has a high error rate. These are typically [decision trees](https://blog.paperspace.com/decision-trees/) (also called decision stumps, because they are less complicated than typical decision trees).
* **Additive Model** - This is the iterative and sequential approach of adding the trees (weak learners) one step at a time. After each iteration, we need to be closer to our final model. In other words, each iteration should reduce the value of our loss function.

**An Intuitive Understanding: Visualizing Gradient Boost**

Let's start with looking at one of the most common binary classification machine learning problems. It aims at predicting the fate of the passengers on Titanic based on a few features: their age, gender, etc. We will take only a subset of the dataset and choose certain columns, for convenience. Our dataset looks something like this:

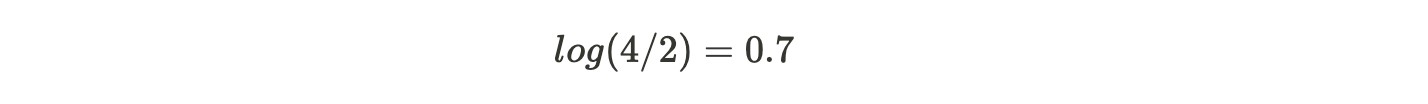
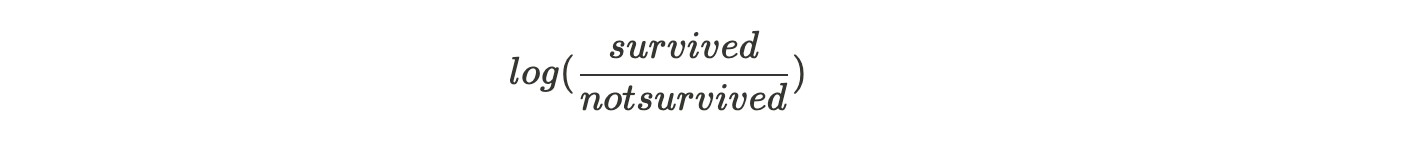
Titanic Passenger Data



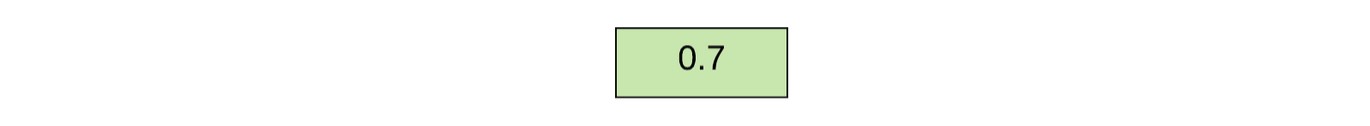
* *Pclass*, or Passenger Class, is categorical: 1, 2, or 3.
* *Age* is the age of the passenger when they were on the Titanic.
* *Fare* is the Passenger Fare.
* *Sex* is the gender of the person.
* *Survived* refers to whether or not the person survived the crash; 0 if they did not, 1 if they did.

Now let's look at how the Gradient Boosting algorithm solves this problem.

We start with one leaf node that predicts the initial value for every individual passenger. For a classification problem, it will be the log(odds) of the target value. log(odds) is the equivalent of average in a classification problem. Since four passengers in our case survived, and two did not survive, log(odds) that a passenger survived would be:

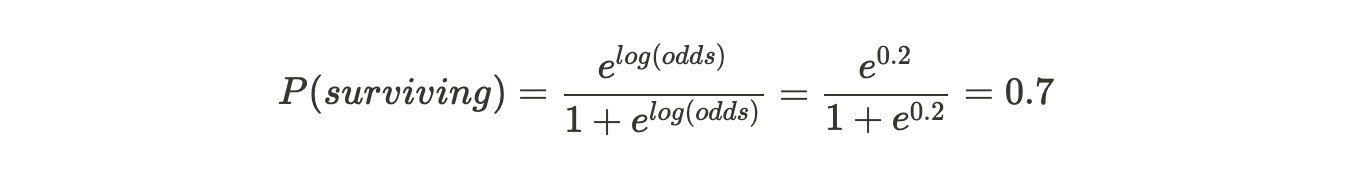


This becomes our initial leaf.



Initial Leaf Node

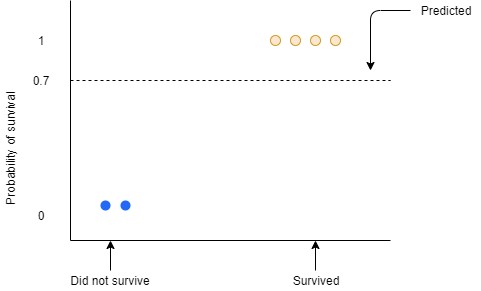
The easiest way to use the log(odds) for classification is to convert it to a probability. To do so, we'll use this formula:



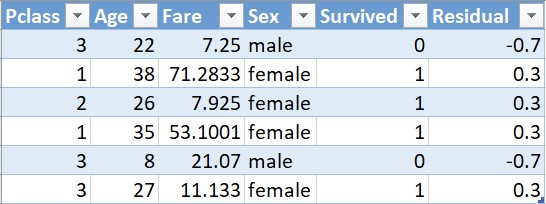
**Note: Please bear in mind that we have rounded off everything to one decimal place here, and hence the log(odds) and probability are the same, which may not be the case always.**

If the probability of surviving is greater than 0.5, then we first classify everyone in the training dataset as survivors. (0.5 is a common threshold used for classification decisions made based on probability; note that the threshold can easily be taken as something else.)

Now we need to calculate the **Pseudo Residual**, i.e, the difference between the observed value and the predicted value. Let us draw the residuals on a graph.

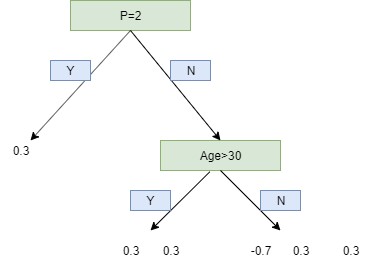


The blue and the yellow dots are the observed values. The blue dots are the passengers who did not survive with the probability of 0 and the yellow dots are the passengers who survived with a probability of 1. The dotted line here represents the predicted probability which is 0.7 We need to find the residual which would be :



Here, 1 denotes Yes and 0 denotes No.

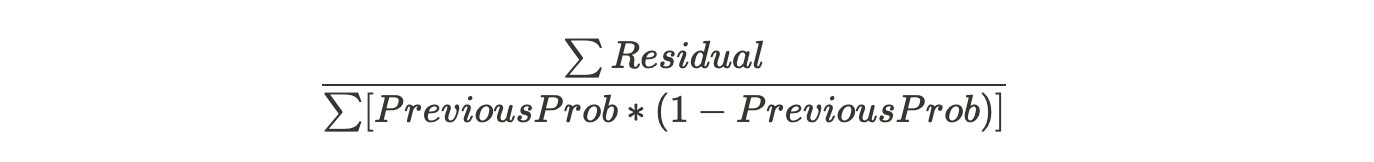
We will use this residual to get the next tree. It may seem absurd that we are considering the residual instead of the actual value, but we shall throw more light ahead.



Branching out data points using the residual values

We use a limit of two leaves here to simplify our example, but in reality, Gradient Boost has a range between **8 leaves to 32 leaves**.

Because of the limit on leaves, one leaf can have multiple values. Predictions are in terms of log(odds) but these leaves are derived from probability which cause disparity. So, we can't just add the single leaf we got earlier and this tree to get new predictions because they're derived from different sources. We have to use some kind of transformation. The most common form of transformation used in Gradient Boost for Classification is :



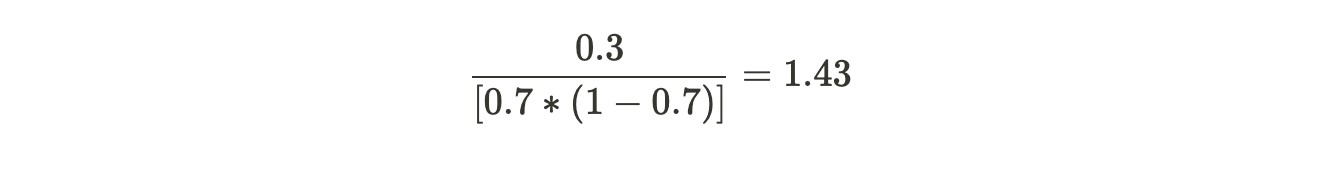
The numerator in this equation is sum of residuals in that particular leaf.

The denominator is sum of (previous prediction probability for each residual ) \* (1 - same previous prediction probability).

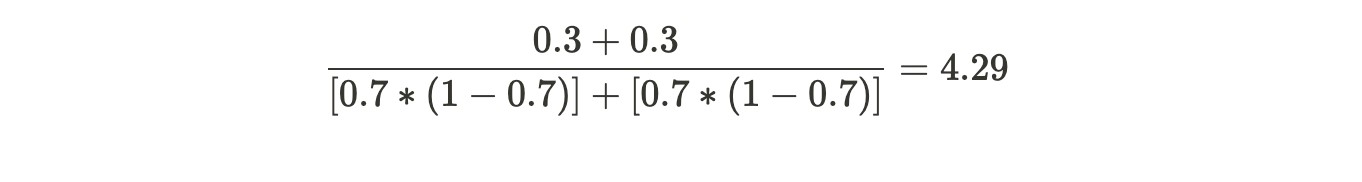
The derivation of this formula shall be explained in the Mathematical section of this article.

For now, let us put the formula into practice:

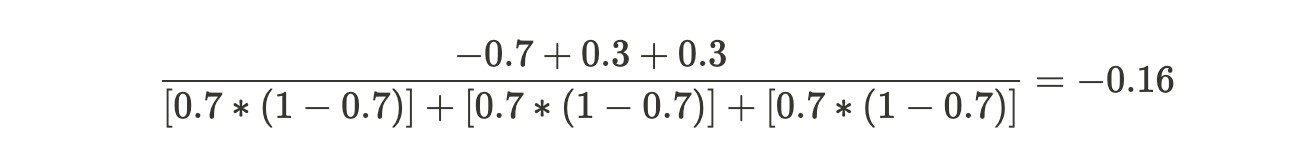
The first leaf has only one residual value that is 0.3, and since this is the first tree, the previous probability will be the value from the initial leaf, thus, same for all residuals. Hence,



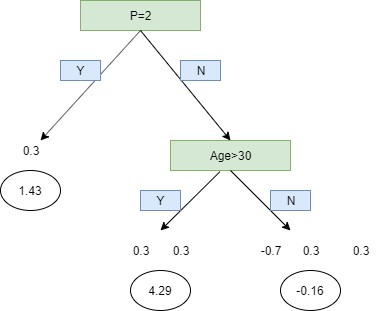
For the second leaf,



Similarly, for the last leaf:



Now the transformed tree looks like:



Transformed tree

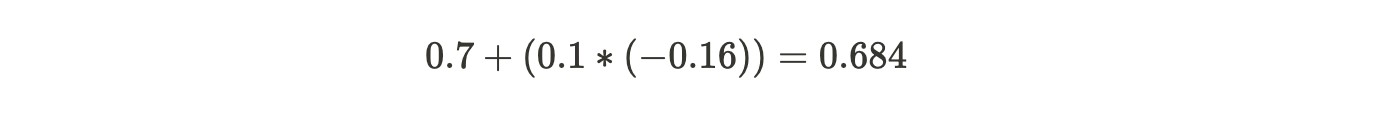
Now that we have transformed it, we can add our initial lead with our new tree with a learning rate.



**Learning Rate** is used to scale the contribution from the new tree. This results in a small step in the right direction of prediction. Empirical evidence has proven that taking lots of small steps in the right direction results in better prediction with a testing dataset i.e the dataset that the model has never seen as compared to the perfect prediction in 1st step. Learning Rate is usually a small number like 0.1

We can now calculate new log(odds) prediction and hence a new probability.

For example, for the first passenger, Old Tree = 0.7. Learning Rate which remains the same for all records is equal to 0.1 and by scaling the new tree, we find its value to be -0.16. Hence, substituting in the formula we get:



Similarly, we substitute and find the new log(odds) for each passenger and hence find the probability. Using the new probability, we will calculate the new residuals.

This process repeats until we have made the maximum number of trees specified or the residuals get super small.

**A Mathematical Understanding**

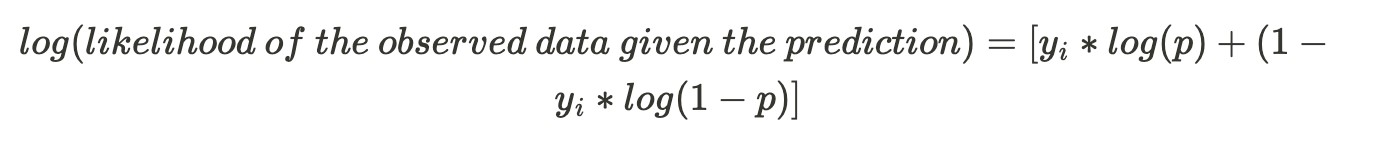
Now that we have understood how a Gradient Boosting Algorithm works on a classification problem, intuitively, it would be important to fill a lot of blanks that we had left in the previous section which can be done by understanding the process mathematically.

We shall go through each step, one at a time and try to understand them.



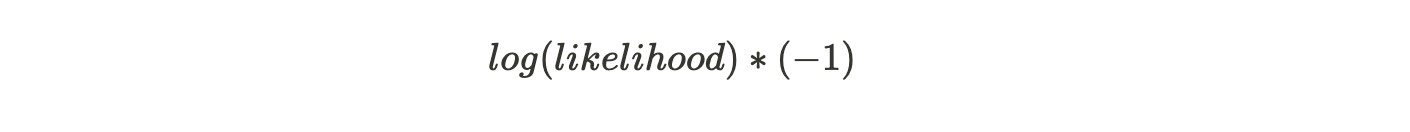
xi - This is the input variables that we feed into our model. yi- This is the target variable that we are trying to predict.

We can predict the log likelihood of the data given the predicted probability

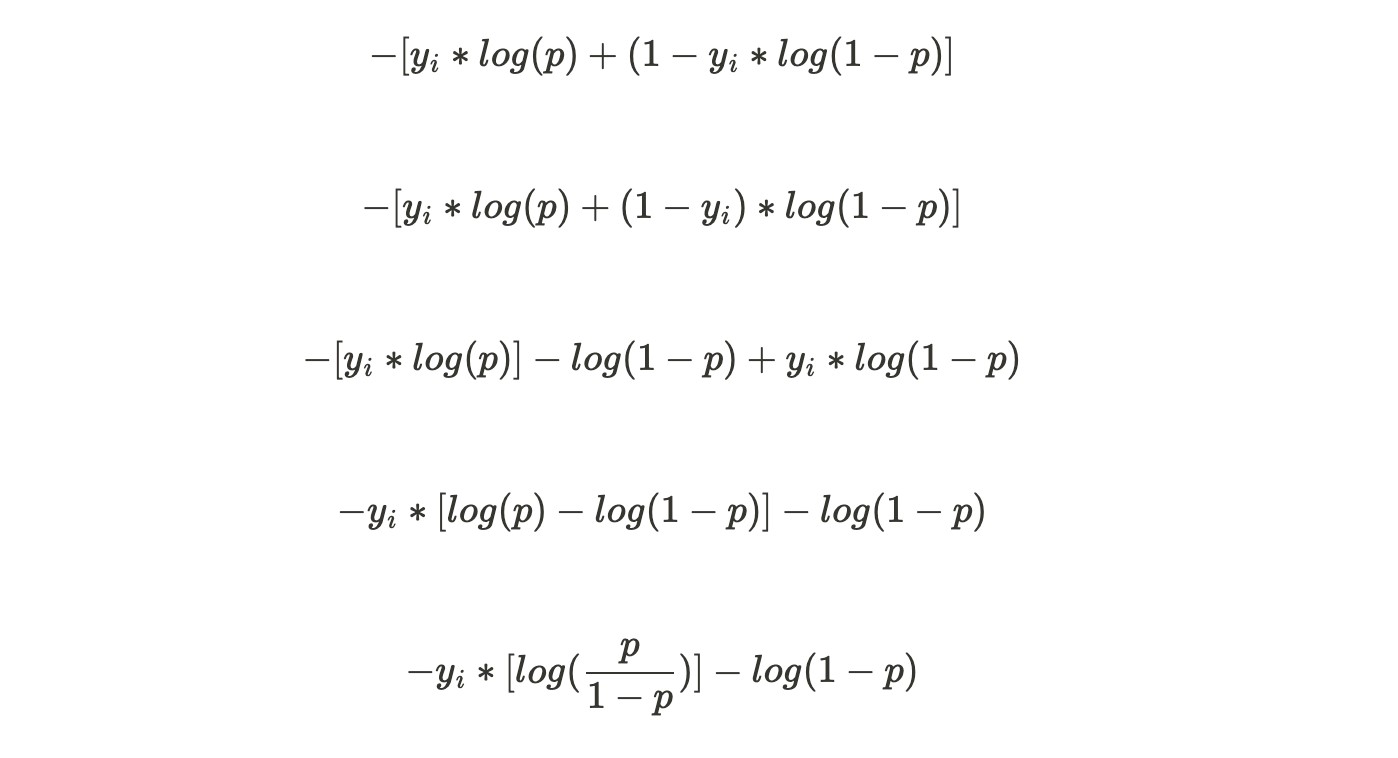


yi is observed value ( 0 or 1 ). p is the predicted probability.

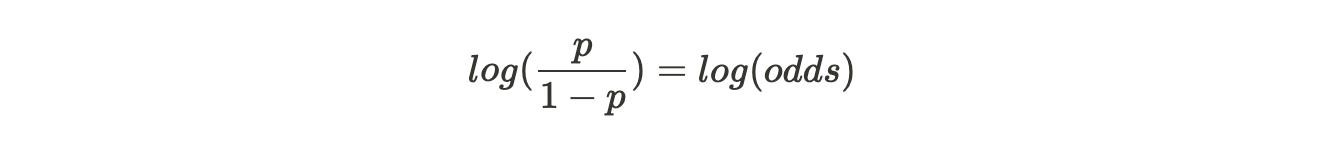
The goal would be to maximize the log likelihood function. Hence, if we use the **log(likelihood)** as our loss function where smaller values represent better fitting models then:



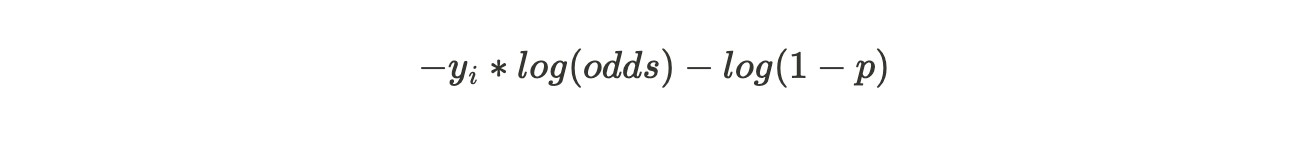
Now the log(likelihood) is a function of predicted probability p but we need it to be a function of predictive log(odds). So, let us try and convert the formula :



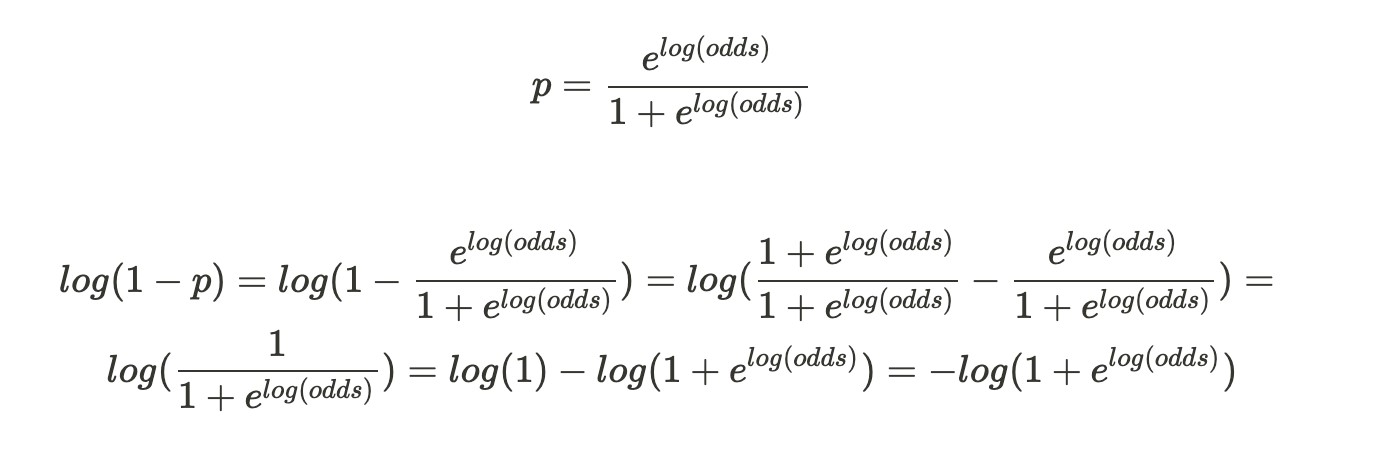
We know that :



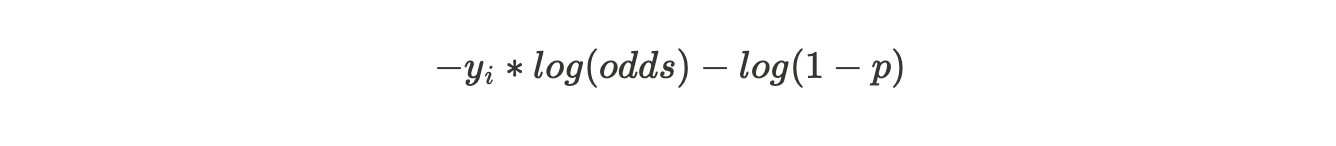
Substituting,



Now,

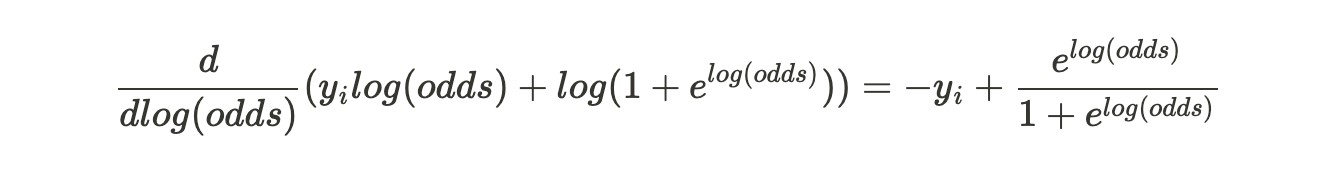


Hence,



Now that we have converted the p to log(odds), this becomes our **Loss Function**.

We have to show that this is differentiable.

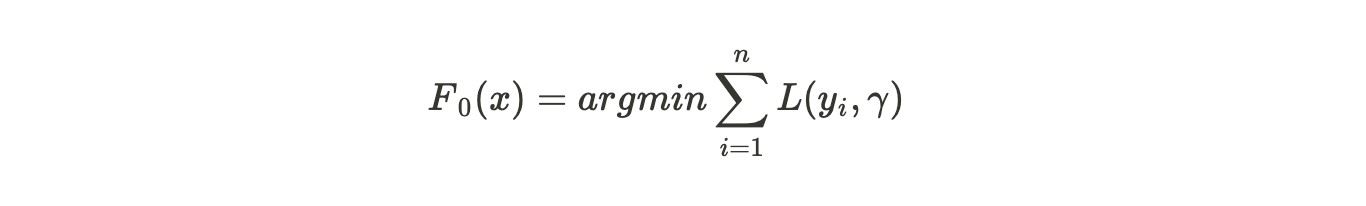


This can also be written as :



Now we can proceed to the actual steps of the model building.

**Step 1: Initialize model with a constant value**

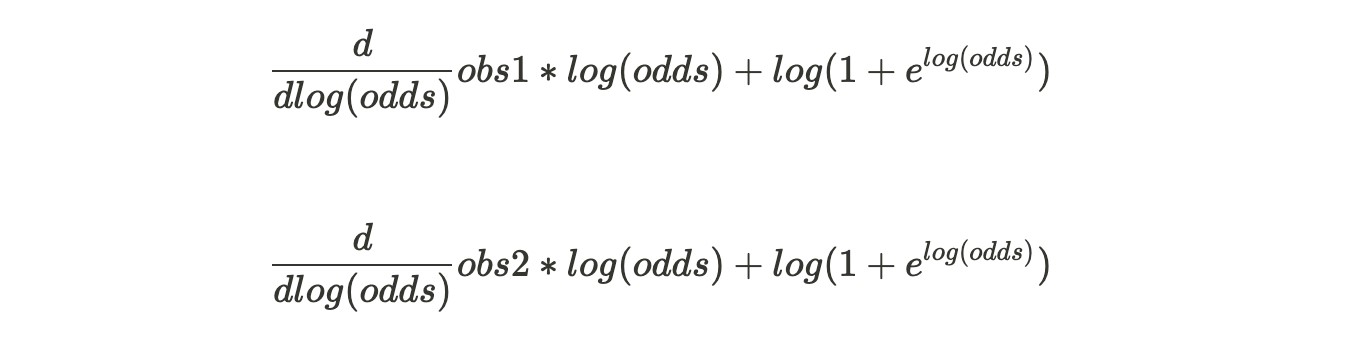


Here, yi is the observed values, L is the loss function, and gamma is the value for log(odds).

We are summating the loss function i.e. we add up the Loss Function for each observed value.

argmin over gamma means that we need to find a log(odds) value that minimizes this sum.

Then, we take the derivative of each loss function :



... and so on.

**Step 2: for m = 1 to M:**

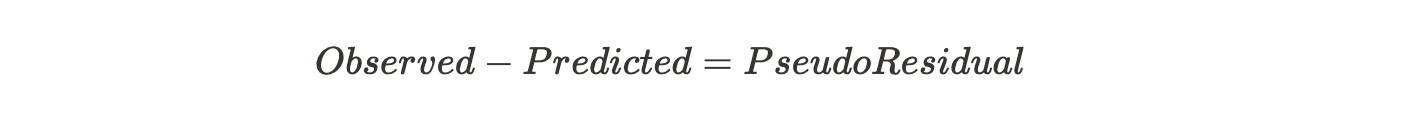
(A)



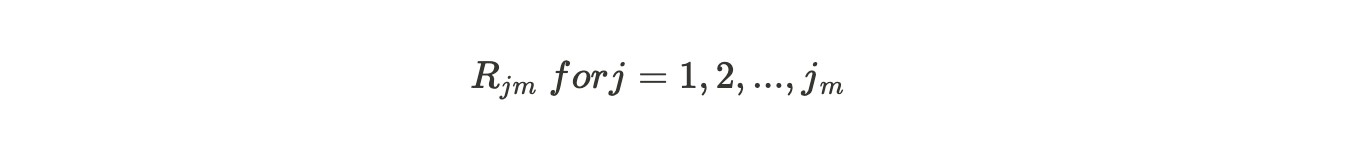
This step needs you to calculate the residual using the given formula. We have already found the Loss Function to be as :



Hence,



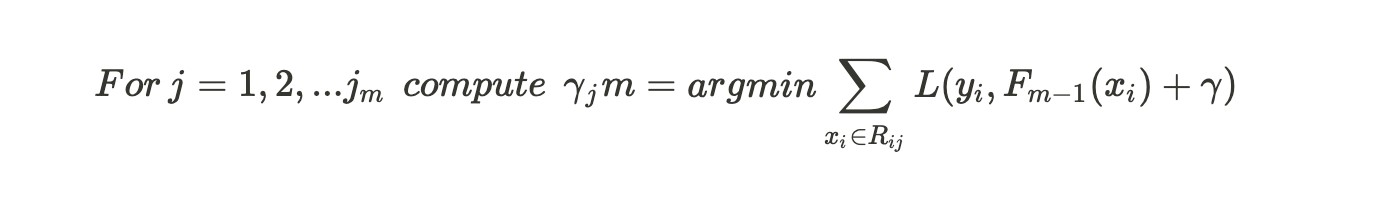
(B) Fit a regression tree to the residual values and create terminal regions



Because the leaves are limited for one branch hence, we might have more than one value in a particular terminal region.

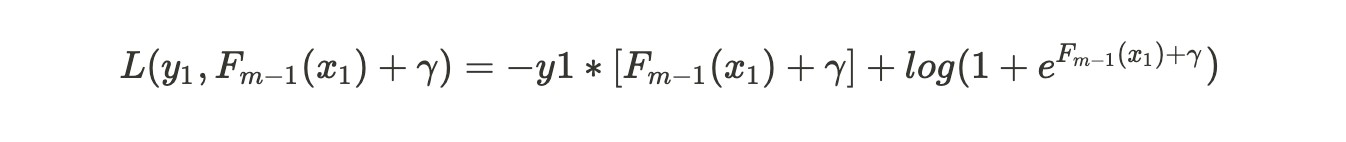
In our first tree, m=1 and j will be the unique number for each terminal node. So R11, R21 and so on.

(C)

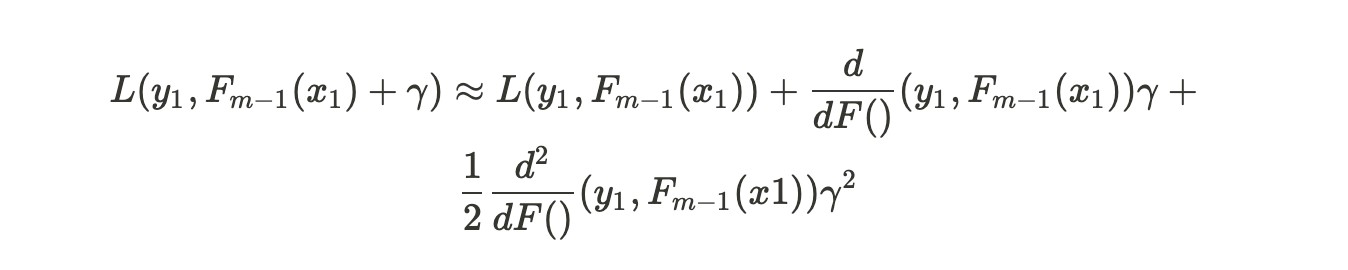


For each leaf in the new tree, we calculate gamma which is the output value. The summation should be only for those records which goes into making that leaf. In theory, we could find the derivative with respect to gamma to obtain the value of gamma but that could be extremely wearisome due to the hefty variables included in our loss function.

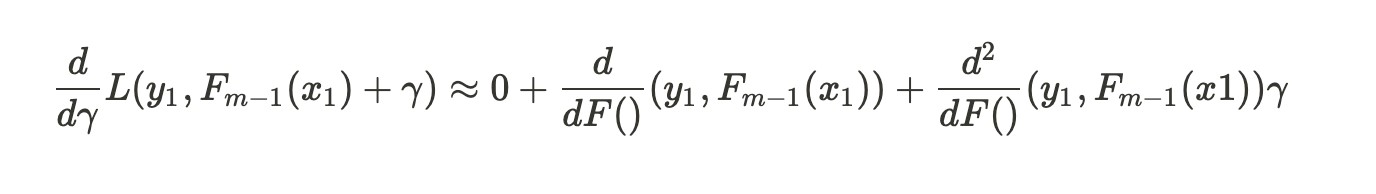
Substituting the loss function and i=1 in the equation above, we get:



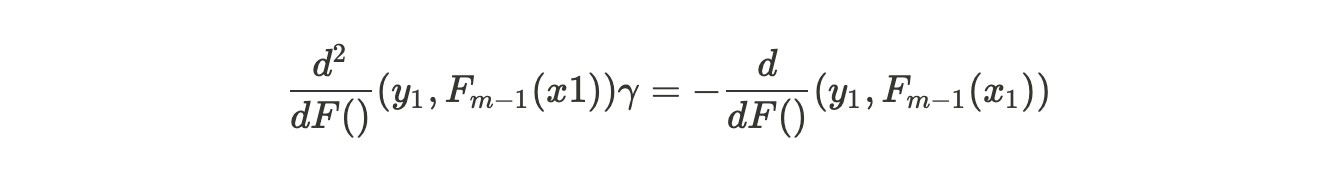
We use second order Taylor Polynomial to approximate this Loss Function :



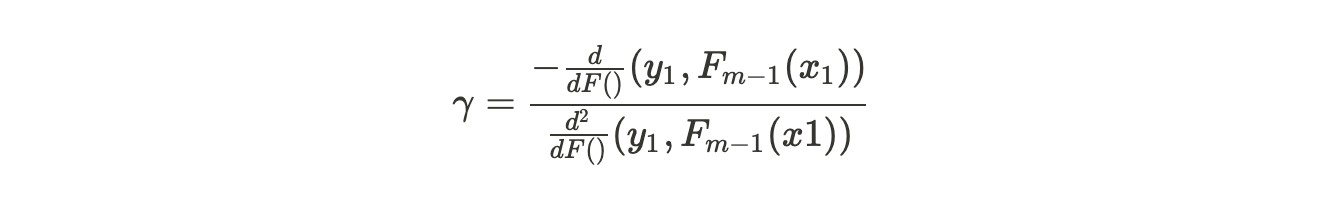
There are three terms in our approximation. Taking derivative with respect to gamma gives us:



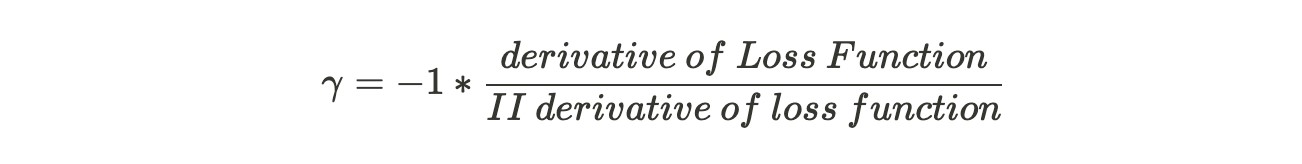
Equating this to 0 and subtracting the single derivative term from both the sides.



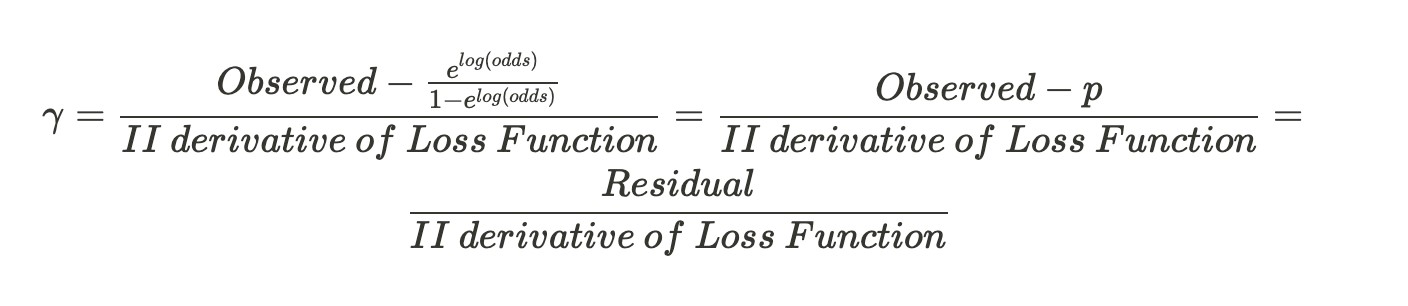
Then, gamma will be equal to :



The gamma equation may look humongous but in simple terms, it is :



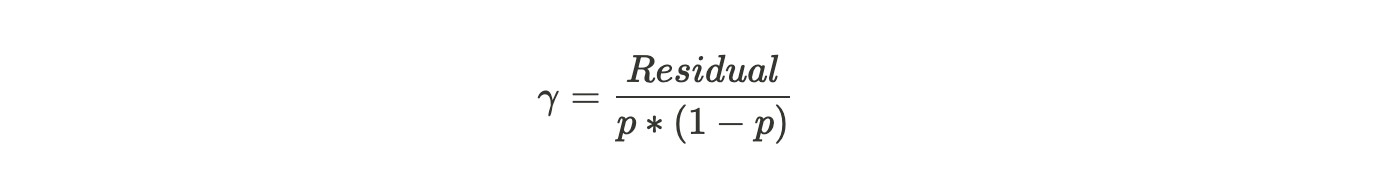
We will just substitute the value of derivative of Loss Function



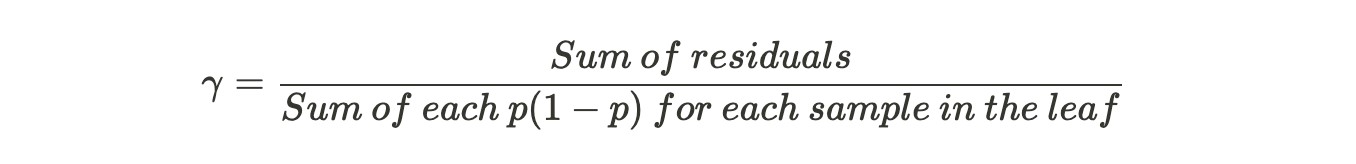
Now we shall solve for the second derivative of the Loss Function. After some heavy computations, we get :



We have simplified the numerator as well as the denominator. The final gamma solution looks like :



We were trying to find the value of gamma that when added to the most recent predicted log(odds) minimizes our Loss Function. This gamma works when our terminal region has only one residual value and hence one predicted probability. But, do recall from our example above that because of the restricted leaves in Gradient Boosting, it is possible that one terminal region has many values. Then the generalized formula would be:



Hence, we have calculated the output values for each leaf in the tree. (D)



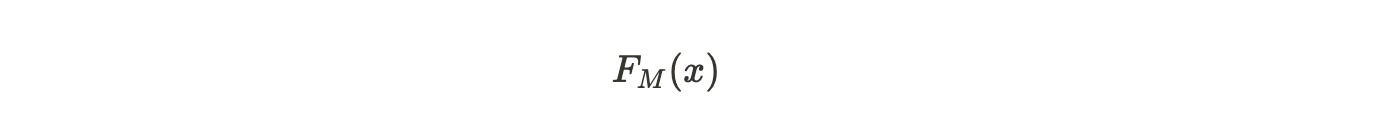
This formula is asking us to update our predictions now. In the first pass, m =1 and we will substitute F0(x), the common prediction for all samples i.e. the initial leaf value plus nu, which is the learning rate into the output value from the tree we built, previously. The summation is for the cases where a single sample ends up in multiple leaves.

Now we will use this new F1(x) value to get new predictions for each sample.

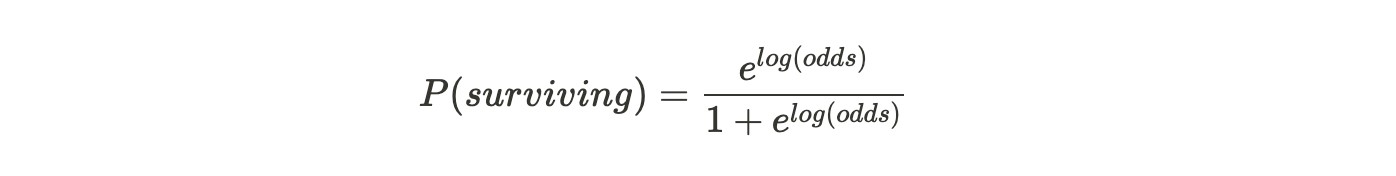
The new predicted value should get us a little closer to actual value. It is to be noted that in contrary to one tree in our consideration, gradient boosting builds a lot of trees and M could be as large as 100 or more.

This completes our for loop in Step 2 and we are ready for the final step of Gradient Boosting.

**Step 3: Output**



If we get a new data, then we shall use this value to predict if the passenger survived or not. This would give us the log(odds) that the person survived. Plugging it into 'p' formula:



|  |
| --- |
| [sklearn.ensemble**.**](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble)**GradientBoostingClassifier** |
| *class* sklearn.ensemble.GradientBoostingClassifier(*\**, *loss='deviance'*, *learning\_rate=0.1*, *n\_esti mators=100*, *subsample=1.0*, *criterion='friedman\_mse'*, *min\_samples\_split=2*, *min\_samples\_lea f=1*, *min\_weight\_fraction\_leaf=0.0*, *max\_depth=3*, *min\_impurity\_decrease=0.0*, *init=None*, *rand om\_state=None*, *max\_features=None*, *verbose=0*, *max\_leaf\_nodes=None*, *warm\_start=False*, *v alidation\_fraction=0.1*, *n\_iter\_no\_change=None*, *tol=0.0001*, *ccp\_alpha=0.0*[)[source]](https://github.com/scikit-learn/scikit-learn/blob/baf828ca1/sklearn/ensemble/_gb.py#L900) |

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n\_classes\_ regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced.

Read more in the [User Guide.](https://scikit-learn.org/stable/modules/ensemble.html#gradient-boosting)

|  |
| --- |
| **Parameters** |

**loss*{‘deviance’, ‘exponential’}, default=’deviance’***

The loss function to be optimized. ‘deviance’ refers to deviance (= logistic regression) for classification with probabilistic outputs. For loss ‘exponential’ gradient boosting recovers the AdaBoost algorithm. **learning\_rate*float, default=0.1***

Learning rate shrinks the contribution of each tree by learning\_rate. There is a trade-off between learning\_rate and n\_estimators.

#### n\_estimatorsint, default=100

The number of boosting stages to perform. Gradient boosting is fairly robust to overfitting so a large number usually results in better performance.

#### subsamplefloat, default=1.0

The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting. subsample interacts with the parameter n\_estimators. Choosing subsample < 1.0 leads to a reduction of variance and an increase in bias.

**criterion*{‘friedman\_mse’, ‘squared\_error’, ‘mse’, ‘mae’}, default=’friedman\_mse’***

The function to measure the quality of a split. Supported criteria are ‘friedman\_mse’ for the mean squared error with improvement score by Friedman, ‘squared\_error’ for mean squared error, and ‘mae’ for the mean absolute error. The default value of ‘friedman\_mse’ is generally the best as it can provide a better approximation in some cases.

#### min\_samples\_splitint or float, default=2

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

•

If int, then c

onsider

min\_samples\_split

as the minimum number.

•

If fl

oat, then

min\_samples\_spli

t

is a fractio

n

and

ceil(min\_samples\_split

\*

n\_samples)

are the mini

mum number of

samples for each split.

*Changed in version 0.18:*

Added float values for fractions.

**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.

•

If int, then c

onsider

min\_samples\_leaf

as the minimum number.

•

oat, th

If fl

en

min\_samples\_lea

f

is a fractio

n

and

ceil(min\_samples\_leaf

\*

n\_samples)

are the minimum number of

samples for each node.

*Changed in version 0.18:*

Added float values for fractions.

**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\_depthint, default=3

The maximum depth of the individual regression estimators. The maximum depth limits the number of nodes in the tree. Tune this parameter for best performance; the best value depends on the interaction of the input variables.

**min\_impurity\_decrease*float, default=0.0***

A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

The weighted impurity decrease equation is the following:

N\_t / N \* (impurity - N\_t\_R / N\_t \* right\_impurity

- N\_t\_L / N\_t \* left\_impurity)

where N is the total number of samples, N\_t is the number of samples at the current node, N\_t\_L is the number of samples in the left child, and N\_t\_R is the number of samples in the right child.

N, N\_t, N\_t\_R and N\_t\_L all refer to the weighted sum, if sample\_weight is passed.

*New in version 0.19.*

**init*estimator or ‘zero’, default=None***

An estimator object that is used to compute the initial predictions. init has to provide [**fit**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.fit) and [**predict\_proba**.](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.predict_proba) If ‘zero’, the initial raw predictions are set to zero. By default, a DummyEstimator predicting the classes priors is used. **random\_state*int, RandomState instance or None, default=None***

Controls the random seed given to each Tree estimator at each boosting iteration. In addition, it controls the random permutation of the features at each split (see Notes for more details). It also controls the random splitting of the training data to obtain a validation set if n\_iter\_no\_change is not None. Pass an int for reproducible output across multiple function calls. See [Glossary.](https://scikit-learn.org/stable/glossary.html#term-random_state)

**max\_features*{‘auto’, ‘sqrt’, ‘log2’}, int or float, default=None*** The number of features to consider when looking for the best split:

* If int, then consider max\_features features at each split.
* If float, then max\_features is a fraction and int(max\_features \* n\_features) features are considered at each 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.

Choosing max\_features < n\_features leads to a reduction of variance and an increase in bias.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features.

#### verboseint, default=0

Enable verbose output. If 1 then it prints progress and performance once in a while (the more trees the lower the frequency). If greater than 1 then it prints progress and performance for every tree.

#### max\_leaf\_nodesint, default=None

Grow trees 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.

#### warm\_startbool, default=False

|  |  |  |
| --- | --- | --- |
| When set to | True | , reuse the solution of the previous call to fit and add more |
| estimators to the ensemble, otherwise, just erase the previous solution. See [the](https://scikit-learn.org/stable/glossary.html#term-warm_start) | | |

[Glossary.](https://scikit-learn.org/stable/glossary.html#term-warm_start)

**validation\_fraction*float, default=0.1***

The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if n\_iter\_no\_change is set to an integer.

*New in version 0.20.*

#### n\_iter\_no\_changeint, default=None

n\_iter\_no\_change is used to decide if early stopping will be used to terminate training when validation score is not improving. By default it is set to None to disable early stopping. If set to a number, it will set aside validation\_fraction size of the

training data as v

alidation and terminate training when validation score is not

improving in all of the previous

n\_iter\_no\_chang

e

numbers of iterations. The split is

stratified.

*New in version 0.20.*

**tol**

***float, default=1e***

***-***

***4***

Tolerance for the early stopping. When the loss is not improving by at least tol for n\_iter\_no\_change iterations (if set to a number), the training stops.

*New in version 0.20.*

#### ccp\_alphanon-negative float, default=0.0

|  |  |  |
| --- | --- | --- |
| the largest cost complexity that is smaller than | ccp\_alpha | will be chosen. By default, |
| no pruning is performed. See [Minimal Cost-Complexity Pruning](https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning) for details. | | |

Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with *New in version 0.22.*

|  |  |  |
| --- | --- | --- |
| **Attributes** | | |
| **n\_estimators\_*int*** | | |
| The number of estimators as selected by early stopping (if | n\_iter\_no\_change | is |
| specified). Otherwise it is set to n\_estimators. | | |

*New in version 0.20.*

[**feature\_importances\_*n***](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.feature_importances_)***darray of shape (n\_features,)*** The impurity-based feature importances.

#### oob\_improvement\_ndarray of shape (n\_estimators,)

The improvement in loss (= deviance) on the out-of-bag samples relative to the previous iteration. oob\_improvement\_[0] is the improvement in loss of the first stage over the init estimator. Only available if subsample < 1.0 **train\_score\_*ndarray of shape (n\_estimators,)***

The i-th score train\_score\_[i] is the deviance (= loss) of the model at iteration i on the in-bag sample. If subsample == 1 this is the deviance on the training data.

**loss\_*LossFunction***

The concrete LossFunction object.

#### init\_estimator

The estimator that provides the initial predictions. Set via the init argument or loss.init\_estimator.

#### estimators\_ndarray of DecisionTreeRegressor of shape (n\_estimators, loss\_.K)

|  |  |  |
| --- | --- | --- |
| The collection of fitted sub-estimators. | loss\_.K | is 1 for binary classification, otherwise |
| n\_classes. | | |

#### classes\_ndarray of shape (n\_classes,) The classes labels. [n\_features\_i](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.n_features_)nt

DEPRECATED: Attribute n\_features\_ was deprecated in version 1.0 and will be removed in 1.2. **n\_features\_in\_*int***

Number of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit)

*New in version 0.24.*

#### feature\_names\_in\_ndarray of shape (n\_features\_in\_,)

|  |  |
| --- | --- |
| Names of features seen during [fit.](https://scikit-learn.org/stable/glossary.html#term-fit) Defined only when | X has feature names that are all |
| strings. | |

*New in version 1.0.*

**n\_classes\_*int***

The number of classes.

**max\_features\_*int***

The inferred value of max\_features.

**Notes**

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data and max\_features=n\_features, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, random\_state has to be fixed.

**Examples**

The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners.

>>>

**>>>**

**from**

**sklearn.datasets**

**import**

make\_hastie\_10\_2

**>>>**

**from**

**sklearn.ensemble**

**import**

GradientBoostingClassifier

>>>

**>>>**

X

,

y

=

make\_hastie\_10\_2

(

random\_state

=

0

)

**>>>**

X\_train

,

X\_test

=

X

[:

2000

]

,

X

[

2000

:]

**>>>**

y\_train

,

y\_test

=

y

[:

2000

,

]

y

[

2000

:]

>>>

**>>>**

clf

=

GradientBoostingClassifier

(

n\_estimators

=

100

,

learning\_rate

=

1.0

,

**...**

max\_depth

=

1

,

random\_state

=

0

)

.

fit

(

X\_train

,

y\_train

)

**>>>**

clf

.

score

(

X\_test

,

y\_test

)

0.913...

**Methods**

[**appl**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.apply)

[**y**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.apply)

[(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.apply)

X)

Apply trees in the ensem

ble to X, return leaf indices.

[**decision**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.decision_function)

[**\_**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.decision_function)

[**functio**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.decision_function)

[**n**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.decision_function)

[(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.decision_function)

X)

Compute the decision function of

X

.

[**t**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.fit)

[**fi**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.fit)

X,

[(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.fit)

y[,

sample\_weight,

monitor

])

Fit the gradient boosting model.

[**get\_param**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.get_params)

[**s**](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.get_params)

[(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.get_params)

[

deep

])

Get parameters for this estimator.

|  |  |
| --- | --- |
| [**predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.predict)X) | Predict class for X. |
| [**predict\_log\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.predict_log_proba)X) | Predict class log-probabilities for X. |
| [**predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.predict_proba)X) | Predict class probabilities for X. |

Return the mean accuracy on the given test data and [**score**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.score)X, y[, sample\_weight])

labels.

|  |  |
| --- | --- |
| [**set\_params**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.set_params)\*\*params) | Set the parameters of this estimator. |
| [**staged\_decision\_function**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.staged_decision_function)X) | Compute decision function of X for each iteration. |
| [**staged\_predict**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.staged_predict)X) | Predict class at each stage for X. |

[**staged\_predict\_proba**(](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier.staged_predict_proba)X) Predict class probabilities at each stage for X.

#### 7. Hyperparameter Tuning with Grid Search and Random Search

**Hyperparameters** are parameters that are **defined** **before training** to specify how we want model training to happen.We have full control over hyperparameter settings and by doing that we control the learning process.

For example in the random forest model n estimators (number of decision trees we want to have) is a hyperparameter. It can be set to any integer value but of course, setting it to 10 or 1000 changes the learning process significantly.

**Parameters**,on the other hand,are **found during the training**. We have no control over parameter values as they are the result of model training. For example, in linear regression *coefficients* and *intercept* are parameters and found at the end of model training.

To learn model hyperparameters and their values we can simply call get\_params in Python. 🔍

from sklearn.ensemble import RandomForestClassifier# Instantiate the model rf\_model = RandomForestClassifier()# Print hyperparameters rf\_model.get\_params

*RandomForestClassifier(****bootstrap****=True,* ***ccp\_alpha****=0.0,* ***class\_weight****=None,* ***cr iterion****=’gini’,* ***max\_depth****=None,* ***max\_features****=’auto’,* ***max\_leaf\_nodes****=None,* ***max\_samples****=None,* ***min\_impurity\_decrease****=0.0,* ***min\_impurity\_split****=None,* ***min\_samples\_leaf****=1,* ***min\_samples\_split****=2,* ***min\_weight\_fraction\_leaf****=0.0,* ***n\_ estimators****=100,* ***n\_jobs****=None,* ***oob\_score****=False,* ***random\_state****=None,* ***verbose*** *=0,* ***warm\_start****=False)*

As you can see random forest classifier has many hyperparameters and if the model is instantiated without defining hyperparameters, then it will have default values. In the random forest by default n estimators=100 which results in a moderately small forest. (only 100 trees 🌳)

You need to know that in any model (in this case random forest) some hyperparameters are more important than others such as:

|  |
| --- |
| **n\_estimators**: Number of decision trees  **max\_features**: Maximum number of features considered while splitting **max\_depth**: Max depth of the tree  **min\_samples\_leaf**: Minimum number of data points in a leaf node **bootstrap**: Sampling with or without replacement |

And some hyperparameters do not affect model performance such as:

|  |
| --- |
| **n\_jobs**: Number of jobs to run in parallel  **random\_state**: Seed  **verbose**: Printing information while training continues **oob\_score**: Whether or not to use out-of-bag samples |

Finally, **hyperparameter tuning** is finding the best combination of hyperparameters that gives the best performance according to the defined scoring metric.

Data and Initial Model

In this article, we will be using [Glass Identification data set from UCI,](https://www.kaggle.com/uciml/glass) where we have 9 attributes to predict the type of glass (out of 7 discrete values).



5 random rows from data

Next, I will separate X and y, and generate train and test sets. # Seperate X and yX = df.drop(columns=['Type'], axis=1)

y = df['Type']# Generate training and test sets for X and yX\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=0.8, random\_state=1)

After that, we simply run a random forest classifier with default values and get the predictions for the test set.

# Instantiate and fit random forest classifierrf\_model = RandomForestClassifier() rf\_model.fit(X\_train, y\_train)# Predict on the test set and call accuracyy\_pred = rf\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)print(accuracy)

0.81

As you can see accuracy with the default model is 81%. Now we will see how to use grid search to tune selected hyperparameters.

##### Grid Search

Grid Search starts with defining a search space **grid**. The grid consists of selected hyperparameter names and values, and **grid search** exhaustively searches the best combination of these given values. 🚀

Let’s say we decided to define the following parameter grid to optimize some hyperparameters for our random forest classifier. **param\_grid:**

n\_estimators = [50, 100, 200, 300] max\_depth = [2, 4, 6, 8, 10] min\_samples\_leaf = [1, 5, 10] max\_features = ['auto', 'sqrt'] bootstrap = [True, False]

We will use GridSearchCV class from Scikit-Learn library for this optimization. The first thing to mention is grid search will have to run and compare 240 models (=4\*5\*3\*2\*2, multiplication of the selected number of values).

Moreover, GridSearchCV class has the option to perform [cross validation](https://scikit-learn.org/stable/modules/cross_validation.html) to resample the training and test data into multiple folds. By applying cross validation we use every record in data for training and testing instead of splitting the dataset at one time as training and testing. If we decide to use cross validation (let’s say with 5 folds) this means grid search will have to evaluate 1200 (=240\*5) model performances.

Let’s have a look at all the input parameters of GridSearchCV class: *class* sklearn.model\_selection.**GridSearchCV**(*estimator*, *param\_grid*, *scoring=None*, *n\_jobs=None*, *refit=True*, *cv=None*, *return\_train\_score=False*)

We start with defining a dictionary for the grid which we will be an input for GridSeachCv.

|  |
| --- |
|  |
| # Define the gridparam\_grid = {  'n\_estimators': [50, 100, 200, 300],  'min\_samples\_leaf': [1, 5, 10],  'max\_depth': [2, 4, 6, 8, 10],  'max\_features': ['auto', 'sqrt'],  'bootstrap': [True, False]}# Instantiate GridSearchCVmodel\_gridsearch = GridSearchCV( estimator=rf\_model, param\_grid=param\_grid, scoring='accuracy', n\_jobs=4, cv=5, refit=True,  return\_train\_score=True) |

As decided previouslyestimator is RandomForestClassifier (rf\_model) and param\_grid is the parameter grid we defined above. scoring is the desired evaluation metric such as accuracy for a classification task and n\_jobs executes model evaluation in parallel, but be careful if you set n\_jobs=-1 it uses all the processors! Since grid search is an uninformed tuning, we can take advantage of running models in parallel as their outcome does not affect other models’ runs.

Settingrefit=True refits the estimator with the best-found hyperparameter values in the end so we do not need to code them in an additional step. cv defines the cross validation strategy and setting return\_train\_score=True we can print the logs of model runs to make further analysis. # Record the current time start = time()# Fit the selected model

model\_gridsearch.fit(X\_train, y\_train)# Print the time spend and number of models ran

print("GridSearchCV took %.2f seconds for %d candidate parameter settings." % ((time() - start), len(model\_gridsearch.cv\_results\_['params'])))

*GridSearchCV took 247.79 seconds for 240 candidate parameter settings.*

# Predict on the test set and call accuracyy\_pred\_grid = model\_gridsearch.predict(X\_test)

accuracy\_grid = accuracy\_score(y\_test, y\_pred\_grid)

0.88

As you can see, simply tuning some hyperparameters increased the initial accuracy from 81% to 88% spending 247 seconds to hyperparameter tuning.

Grid search always finds the best-performing model with hyperparameter values mentioned in the grid. It is also easy to implement and explain. However, with the increasing number of hyperparameters and values to test it can easily become computationally expensive because it models all of the combinations of hyperparameters. The drawback of not learning from already ran models makes grid search inefficient and time-consuming. In addition, parameter grid plays an extremely important role: even though grid search will always find the best combination if the parameter grid is selected poorly the best combination will not be performing well.

After running the GridSeachCV, we can return the following attributes for further investigation:

##### • cv\_results\_ • best\_estimator\_ • best\_score\_ • best\_params\_

Let’s look at some of them:

print(model\_gridsearch.best\_params\_)

*{‘bootstrap’: True, ‘max\_depth’: 10, ‘max\_features’: ‘sqrt’, ‘min\_samples\_leaf’:*

*1*

*, ‘n\_estimators’*

*: 300}*

print(model\_gridsearch.best\_estimator\_)

*RandomForestClassifier(bootstrap=True, ccp\_alpha=0.0, class\_weight=None, criterion=’gini’, max\_depth=10, max\_features=’sqrt’, max\_leaf\_nodes=None, max\_samples=None, min\_impurity\_decrease=0.0, min\_impurity\_split=None, min\_samples\_leaf=1, min\_samples\_split=2, min\_weight\_fraction\_leaf=0.0, n\_estimators=300, n\_jobs=None, oob\_score=False, random\_state=None, verbose=0, warm\_start=False)*

##### Random Search

In random search, we define **distributions** for each hyperparameter which can be defined *uniformly* or with a *sampling method*. The key difference from grid search is in random search, not all the values are tested and values tested are selected at random.

For example, if there are 500 values in the distribution and if we input n\_iter=50 then random search will randomly sample 50 values to test. By doing that random search optimizes time spend and not defining an absolute grid allows it to explore other values in the given distribution.

Since random search does not try every hyperparameter combination, it does not necessarily return the best performing values, but it returns a relatively

good performing model in a *significantly* shorter time. ⏰ **param\_distributions** n\_estimators = list(range(100, 300, 10)) min\_samples\_leaf = list(range(1, 50)) max\_depth = list(range(2, 20) max\_features = ['auto', 'sqrt'] bootstrap = [True, False]

RandomizedSearchCV from Scikit-Learn has the following input parameters: *class* sklearn.model\_selection.**RandomizedSearchCV**(*estimator*, *param\_distributions*, *n\_iter=10*, *scoring=None*, *n\_jobs=None*, *refit=True*, *cv=None*, *verbose=0*, *pre\_dispatch='2\*n\_jobs'*, *random\_state=None*, *error\_score=nan*, *return\_train\_score=False*)

We start with defining a dictionary for parameter distributions which will be an input for RandomizedSearchCV.

|  |
| --- |
| # specify distributions to sample fromparam\_dist = {  'n\_estimators': list(range(50, 300, 10)),  'min\_samples\_leaf': list(range(1, 50)),  'max\_depth': list(range(2, 20)),  'max\_features': ['auto', 'sqrt'],  'bootstrap': [True, False]}# specify number of search iterationsn\_iter = 50#  Instantiate RandomSearchCVmodel\_random\_search = RandomizedSearchCV( estimator=rf\_model,  param\_distributions=param\_dist, n\_iter=n\_iter) |

Most parameters are similar to GridSearchCV, however, in

RandomizedSearchCV we have param\_distributions for defining the search distributions. n\_iter is used to limit the total number of model runs (in other words parameter combinations that are sampled from the grid). Be careful with the trade-off here, since setting ***high n\_iter*** increases search runtime, and setting ***low n\_iter*** decreases model quality.

# Record the current time start = time()# Fit the selected model

model\_random\_search.fit(X\_train, y\_train)# Print the time spend and number of models ran print("RandomizedSearchCV took %.2f seconds for %d candidate parameter settings." % ((time() - start), len(model\_random\_search.cv\_results\_['params'])))

*RandomizedSearchCV took 64.17 seconds for 50 candidate parameter settings.*

# Predict on the test set and call accuracyy\_pred\_random = model\_random\_search.predict(X\_test)

accuracy\_random = accuracy\_score(y\_test, y\_pred\_random)

0.86

As you see only in 64 seconds we were able to increase accuracy of the initial model from 81% to 86%. Random search did not reach 88% accuracy of grid search, however, this is the tradeoff between two tuning methods.

Finally, let’s have a look at the best parameters found with random search. print(model\_random\_search.best\_params\_)

*{‘n\_estimators’: 230, ‘min\_samples\_leaf’: 4, ‘max\_features’: ‘auto’,*

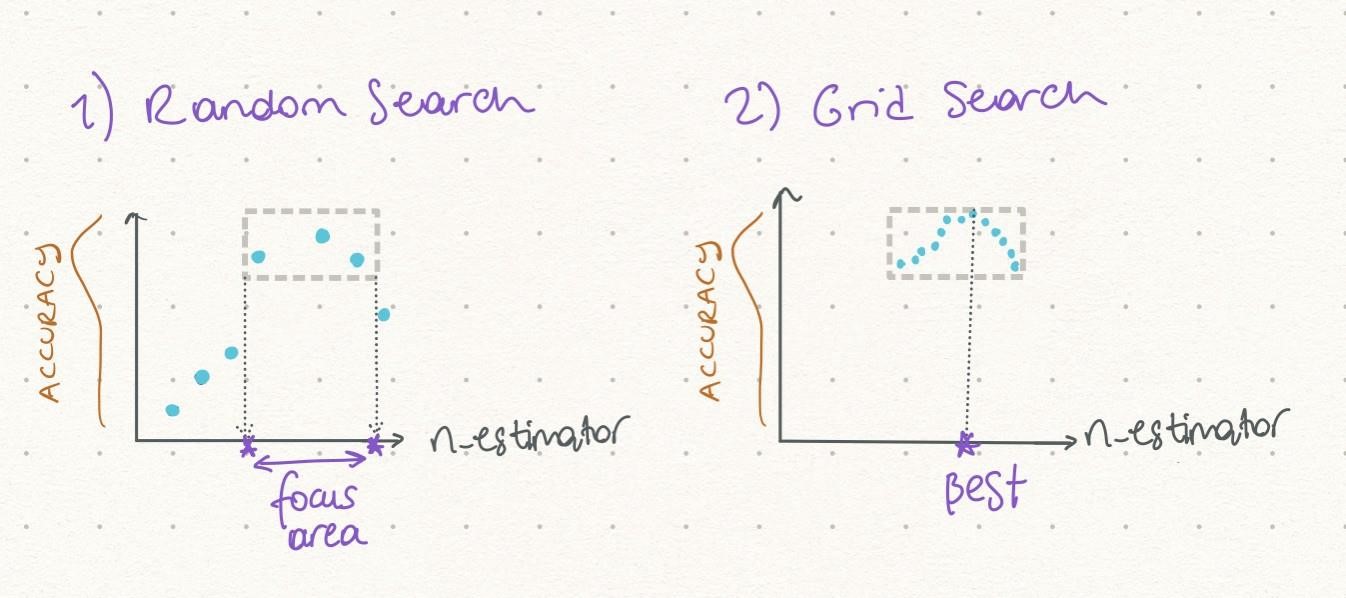
*‘max\_depth’: 13, ‘bootstrap’: False}*

##### Coarse-to-Fine Tuning

From the beginning of the article, we have seen how to apply grid search and random search for hyperparameter optimization.

Using grid search, we were able to test all of the hyperparameter values given in the grid to find the best among them. However, an increased number of hyperparameters easily becomes a bottleneck. Ideally, we can combine grid search with random search to prevent this inefficiency.

In coarse-to-fine tuning, we start with a random search to find the promising value ranges for each hyperparameter. For example, if the random search returns high performance for n\_estimators between 150 and 200, this is the range we want grid search to focus on. After getting focus area for each hyperparameter using random search, we can define the grid accordingly for grid search to find **the** best values amongst them.



In the illustration above first, we are testing values at random to find a focus area with random search. Second, we are testing all values in this focus area with grid search and eventually find the optimal value.

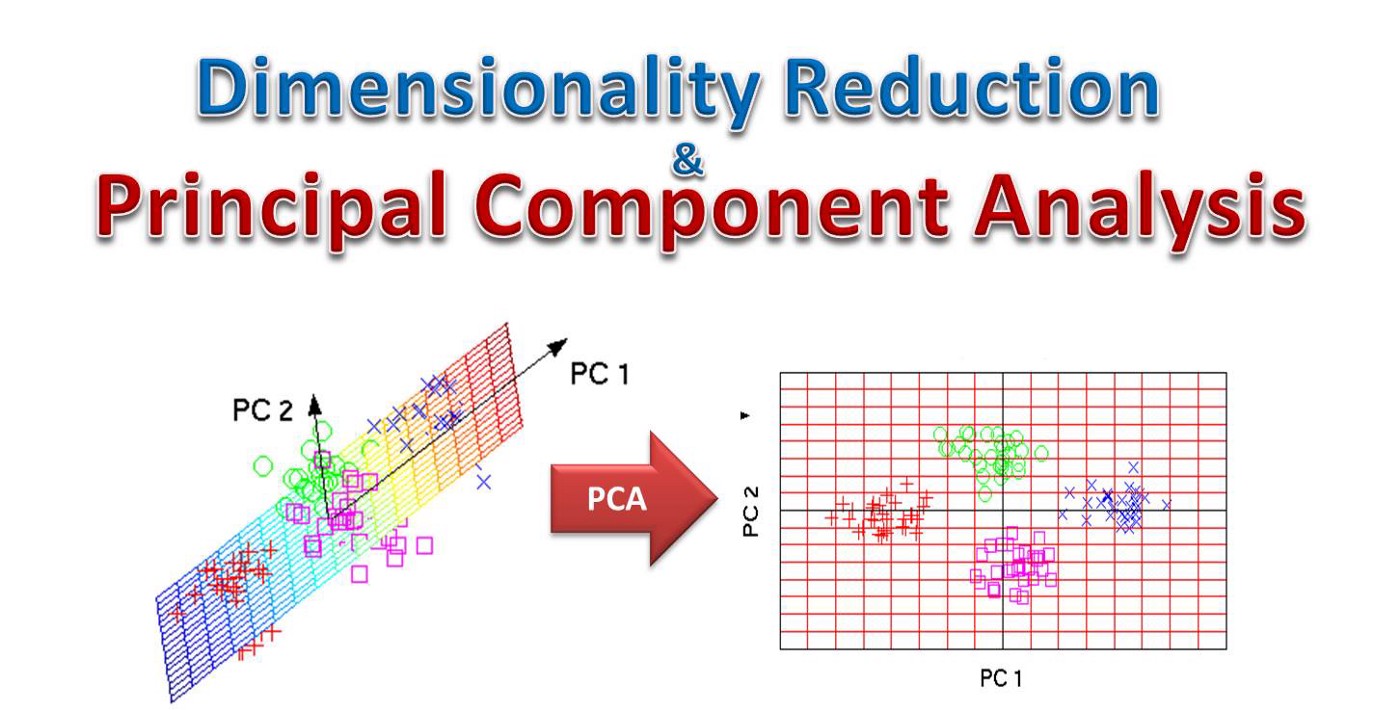
Another useful strategy to fine-tune machine learning models is using [**ensemble learning**](https://12ft.io/proxy?ref=&q=https://towardsdatascience.com/practical-guide-to-ensemble-learning-d34c74e022a0) techniques. Different from hyperparameter tuning, ensemble learning aims to improve model performance by combining multiple models into a group model. This group model aims to perform better than each model alone. The most used ensemble learning methods are **voting, bagging, boosting** and **stacking**

##### Conclusion

In this article, we used a random forest classifier to predict “type of glass” using

9 different attributes. Initial random forest classifier with default hyperparameter values reached 81% accuracy on the test. Using grid search we were able to tune selected hyperparameters in 247 seconds and increased accuracy to 88%. Next, we did the same job using random search and in 64 seconds we increased accuracy to 86%. Last but not least, we discussed coarseto-fine tuning to combine these two methods and get advantages from both!

**8.Feature Engineering PCA**



# Introduction

Understanding the math behind Principal Component Analysis (PCA) without a solid linear algebra foundation is challenging. When I taught Data Science at General Assembly in San Francisco, I found that helping students visualize the transformation between features and principal components greatly enhanced their understanding.**PCA is a dimensionality reduction technique that has four main parts: feature covariance, eigendecomposition, principal component transformation, and choosing components in terms of explained variance.** The purpose of this blog is to share a visual demo that **helped the students understand the final two steps**.

## PCA

PCA is a dimensionality reduction that **identifies important relationships** in our data, **transforms the existing data** based on these relationships, and then **quantifies the importance** of these relationships so we can keep the most important relationships and drop the others. To remember this definition, we can break it down into four steps:

1. We identify the relationship among features through a [**Covariance Matrix**](https://en.wikipedia.org/wiki/Covariance_matrix)**.**
2. Through the linear transformation or [**eigendecomposition**](https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix) of the Covariance Matrix, we get [**eigenvectors**and **eigenvalues**](https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors).
3. Then we transform our data using Eigenvectors into principal components.
4. Lastly, we quantify the importance of these relationships using Eigenvalues and keep the important principal components**.**

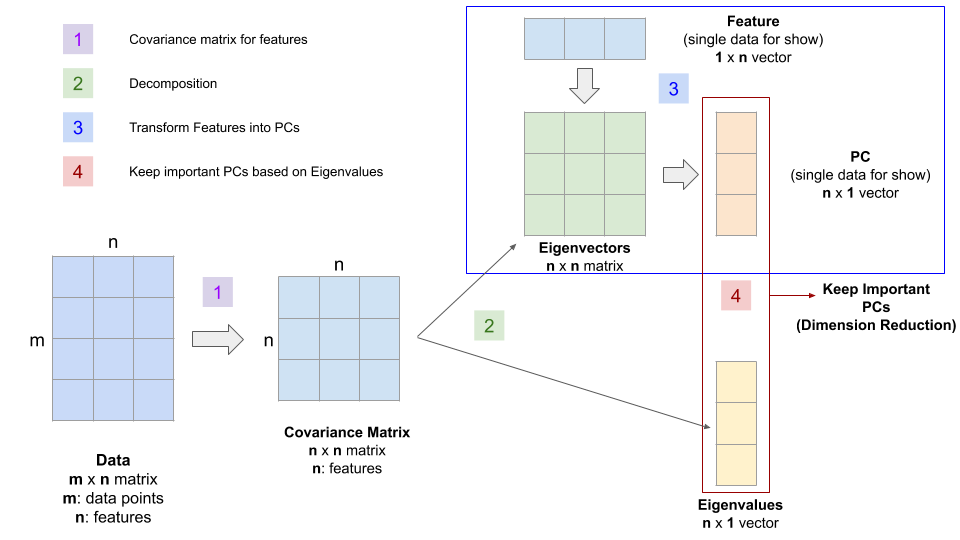


Figure 1

The following demo presents the linear transformation between features and principal components using eigenvectors for a single data point from the Iris database. I describe the calculations without using any linear algebra terms. However, it would be helpful if you understand the dot product between two vectors (since we demonstrate the transformation for a single data point) and matrix multiplication (when we transform all data points).

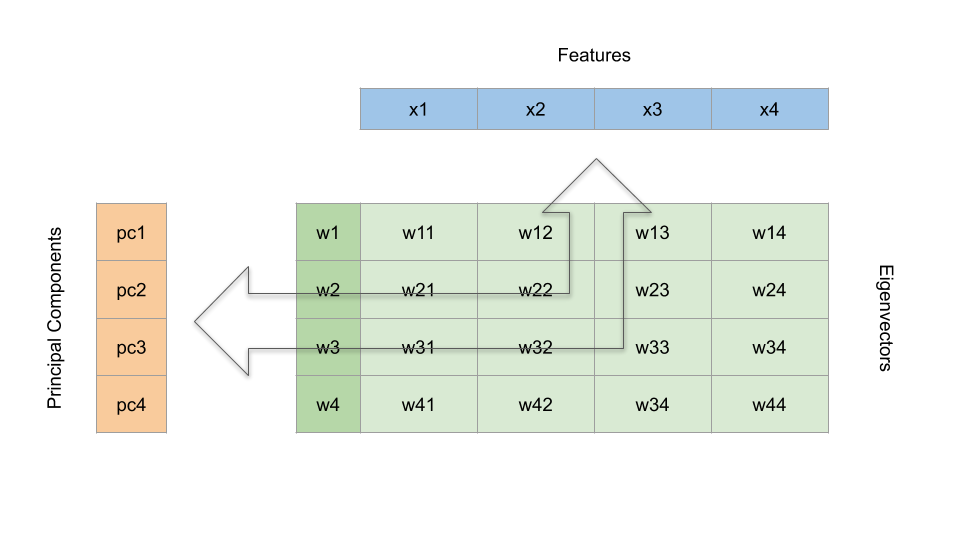


Figure 2

1. **Features**: represented by the blue horizontal on the top. Note that x1, x2, x3, and x4 represents the four features of a single iris (i.e., sepal length, sepal width, petal length, and petal width), not four different irises.
2. **Eigenvectors**: represented by the green matrix
3. **Principal components**: represented by the orange vertical bar to the left

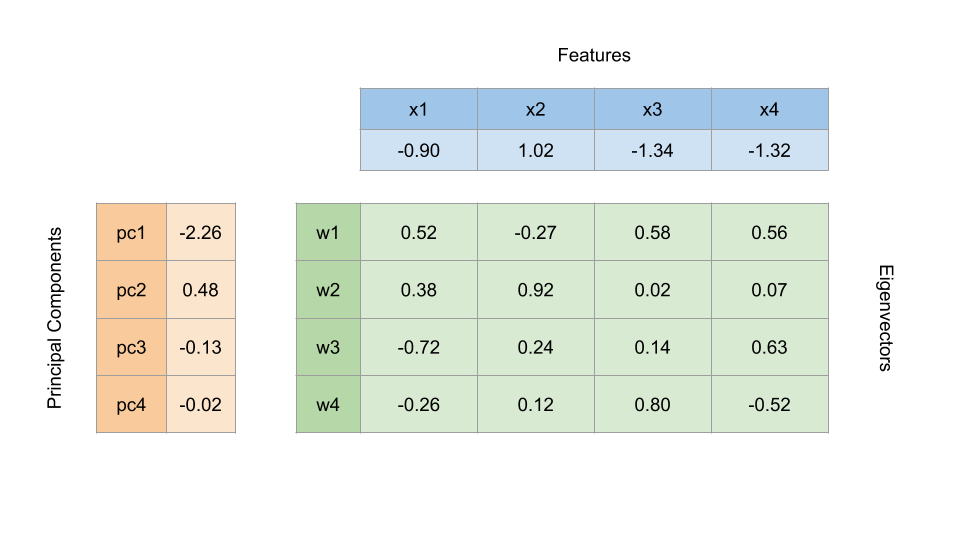


Figure 3

Figure 3 includes the values for each part. One of the best ways to learn and understand a mathematical method is to validate that the numbers add up. From here, we are going to do some simple calculations.

## Visual Demo for Step 3

**Transform Features into Principal Components.**

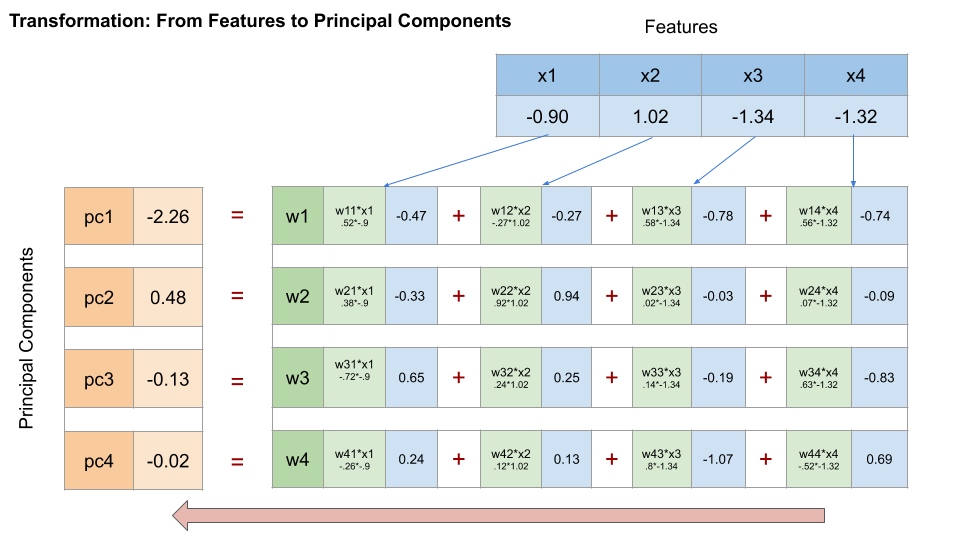
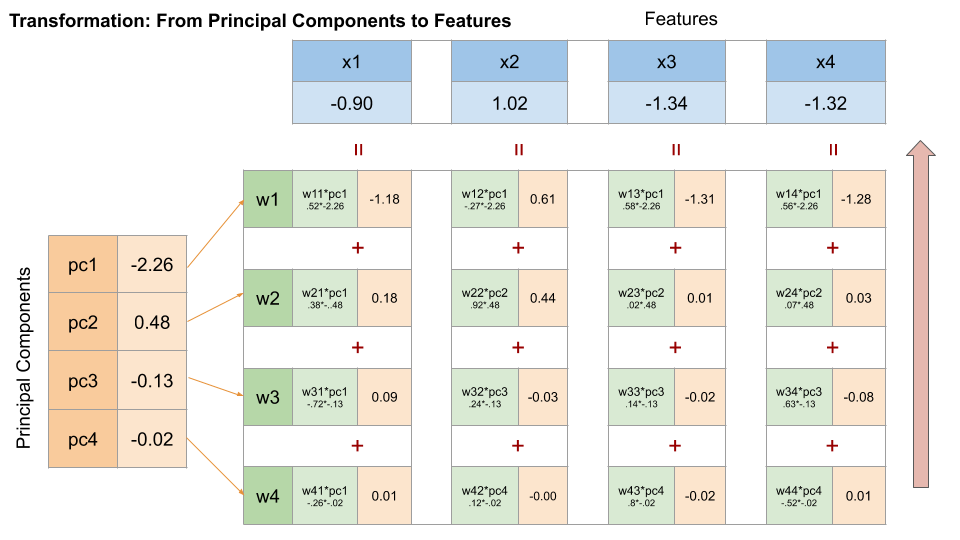


Figure 4

To transform the features into principal components, we multiply the features with the eigenvectors element by element, as shown in Figure 4 above, and add them in the horizontal direction.

**2. Transform Principal Components back into Features.**



Well, you can certainly transform the principal components back into the original features by performing the calculation shown in Figure 5 above.

## Visual Demo for Step 4

After visualizing the math behind the transformation, let’s demonstrate the difference between feature elimination and feature extraction.

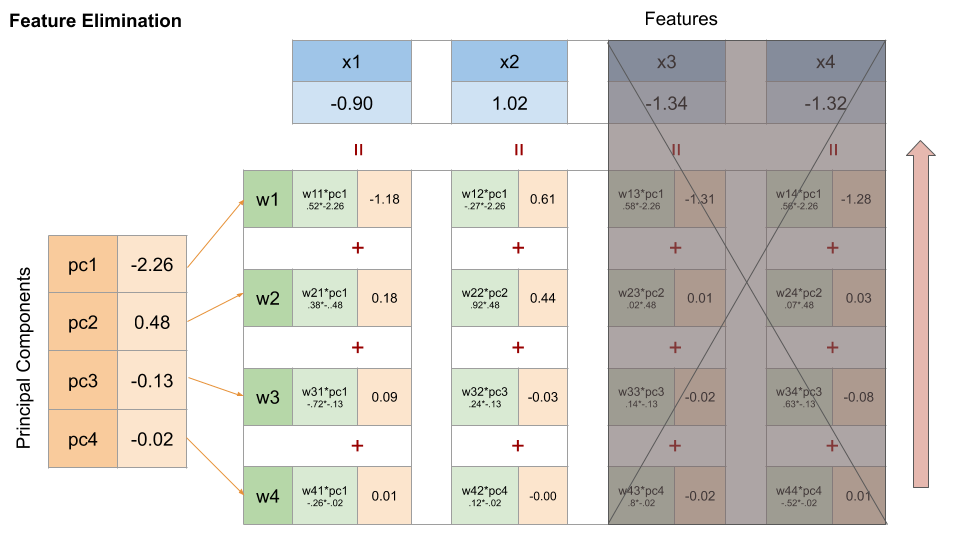


Figure 6

As shown in Figure 6 above, if we choose to reduce dimensionality through feature elimination, we remove some features entirely, in this case, x3 and x4. All information provided by petal length and petal width is lost when we reduce the dimension from 4 to 2.

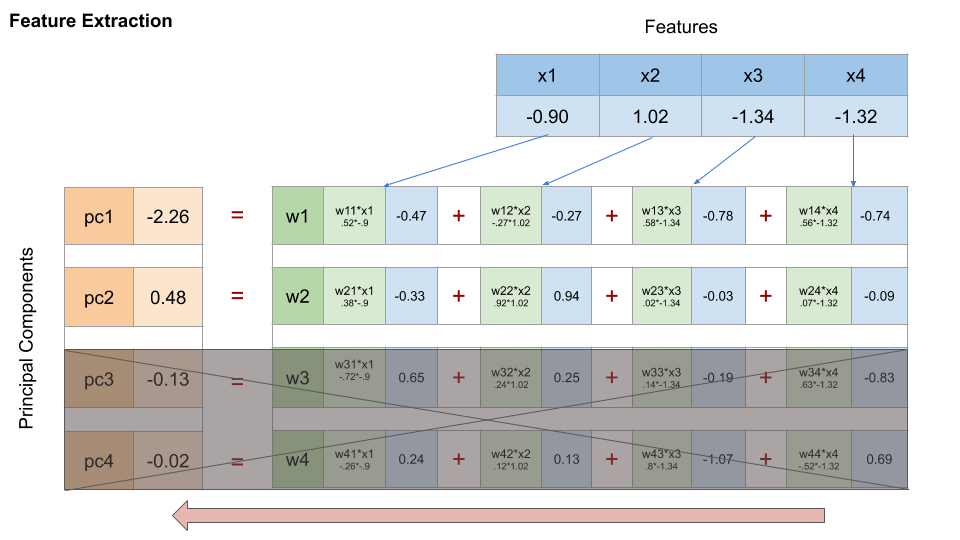
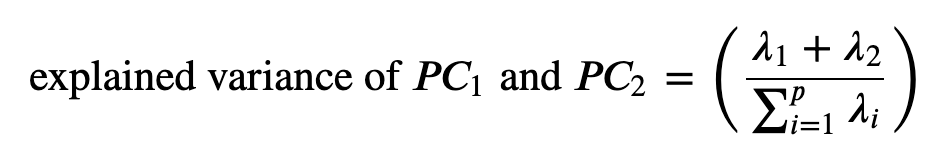


Figure 7

In contrast, when we reduce dimensionality through feature extraction methods such as PCA, we keep the most important information by selecting the principal components that explain most of the relationships among the features. Here is where the eigenvalues kick in and help us learn how much information each principal component contains.

In our case, the first and second principal components (i.e., pc1 and pc2) explained more than 95% of the variation from the features based on the normalized eigenvalue associated with each eigenvector, as shown in Figure 8 below. Therefore, we only keep pc1 and pc2.



**Lambda is the eigenvalue**

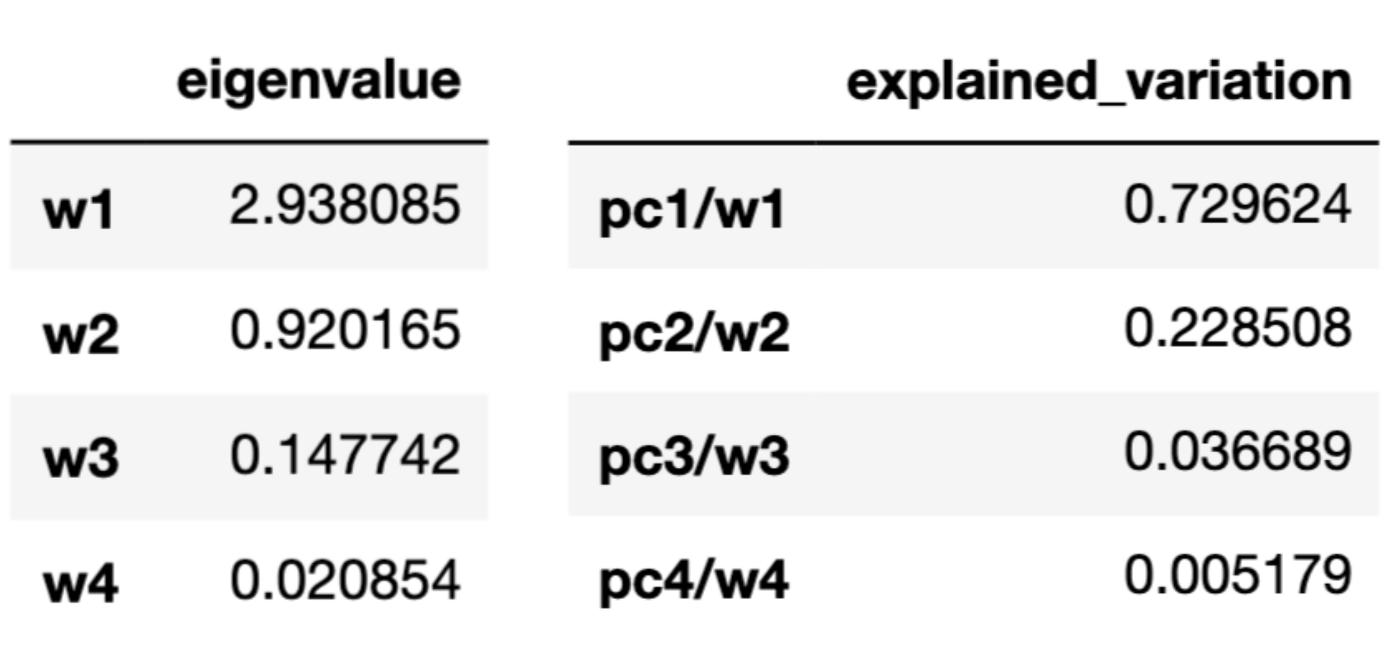


Figure 8

Note that each of the principal components contains information from all four features. By keeping only pc1 and pc2, we extract the most important information from all four features and reduce the dimension from 4 to 2.

**Remark**

I want to conclude my blog post with several remarks, including the use of PCA, the pros and cons of PCA, as well as when not to use PCA.

**Data Cleaning is Important**

* PCA is sensitive to outliers and missing values.

**Standardize Data**

* PCA uses Euclidean distance as its feature vector similarity metric, so make sure we scale the features before applying PCA.
* If we use GridSearchCV in the model, make sure we put PCA in Pipeline. Fitting and transforming the training data before running GridSearch lead to data leakage during cross-validation within GridSearch. See detail in my [last blog post](https://towardsdatascience.com/pre-process-data-with-pipeline-to-prevent-data-leakage-during-cross-validation-e3442cca7fdc).

**Pros**

* PCA reduces the dimensionality without losing information from any features.
* Reduce storage space needed to store data
* Speed up the learning algorithm (with lower dimension).
* Address the multicollinearity issue (all principal components are orthogonal to each other).
* Help visualize data with high dimensionality (after reducing the dimension to 2 or 3).

**Cons**

* Using PCA prevents interpretation of the original features, as well as their impact because eigenvectors are not meaningful.

**Potential Use Cases for PCA (not an exhaustive list)**

* We have many features with high multicollinearity.
* We have too many features that cause the algorithm to run very slowly.

# RESULTS

## 1.Variance Bias Trade off

**What is bias?**

Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

**What is variance?**

Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn’t seen before. As a result, such models perform very well on training data but has high error rates on test data.

### Mathematically

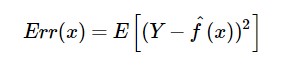
Let the variable we are trying to predict as Y and other covariates as X. We assume there is a relationship between the two such that

Y=f(X) + e

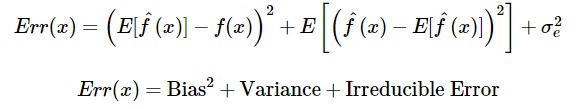
Where e is the error term and it’s normally distributed with a mean of 0.

We will make a model f^(X) of f(X) using linear regression or any other modeling technique.

So the expected squared error at a point x is



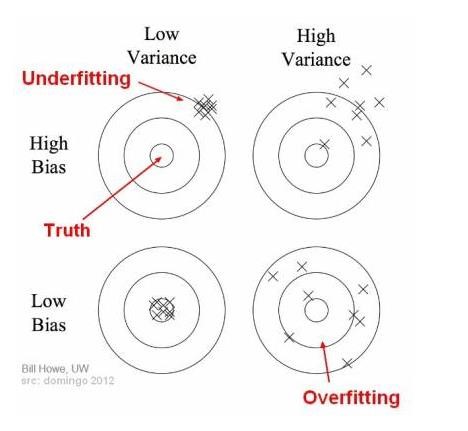
The Err(x) can be further decomposed as



Err(x) is the sum of Bias², variance and the irreducible error.

Irreducible error is the error that can’t be reduced by creating good models. It is a measure of the amount of noise in our data. Here it is important to understand that no matter how good we make our model, our data will have certain amount of noise or irreducible error that can not be removed.

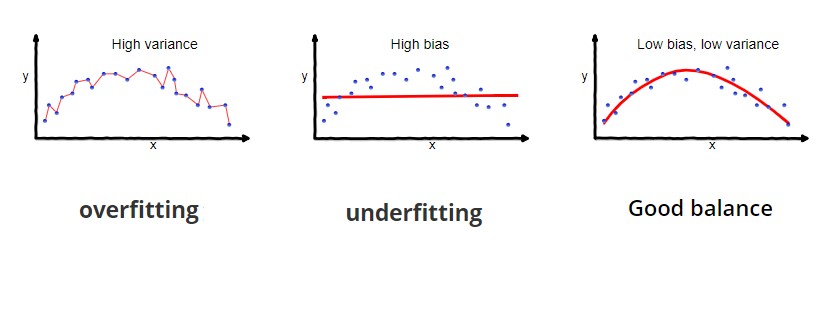
### Bias and variance using bulls-eye diagram



In the above diagram, center of the target is a model that perfectly predicts correct values. As we move away from the bulls-eye our predictions become get worse and worse. We can repeat our process of model building to get separate hits on the target.

In supervised learning, **underfitting** happens when a model unable to capture the underlying pattern of the data. These models usually have high bias and low variance. It happens when we have very less amount of data to build an accurate model or when we try to build a linear model with a nonlinear data. Also, these kind of models are very simple to capture the complex patterns in data like Linear and logistic regression.

In supervised learning, **overfitting** happens when our model captures the noise along with the underlying pattern in data. It happens when we train our model a lot over noisy dataset. These models have low bias and high variance. These models are very complex like Decision trees which are prone to overfitting.

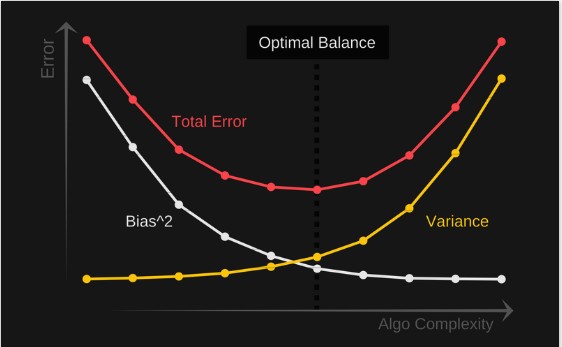
**Why is Bias Variance Tradeoff?**

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has large number of parameters then it’s going to have high variance and low bias. So we need to find the right/good balance without overfitting and underfitting the data.

This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can’t be more complex and less complex at the same time.

### Total Error

To build a good model, we need to find a good balance between bias and variance such that it minimizes the total error.



#### Bias-Variance Decomposition of the 0-1 Loss

Note that decomposing the 0-1 loss into bias and variance components is not as straight-forward as for the squared error loss. To quote Pedro Domingos, a well-known machine learning researcher and professor at University of Washington:

"several authors have proposed bias-variance decompositions related to zeroone loss (Kong & Dietterich, 1995; Breiman, 1996b; Kohavi & Wolpert, 1996; Tibshirani, 1996; Friedman, 1997). However, each of these decompositions has significant shortcomings.". [1]

In fact, the paper this quote was taken from may offer the most intuitive and general formulation at this point. However, we will first, for simplicity, go over Kong & Dietterich formulation [2] of the 0-1 loss decomposition, which is the same as Domingos's but excluding the noise term (for simplicity).

The table below summarizes the relevant terms we used for the squared loss in relation to the 0-1 loss. Recall that the 0-1 loss, , is 0 if a class label is predicted correctly, and one otherwise. The main prediction for the squared error loss is simply the average over the predictions (the expectation is over training sets), for the 0-1 loss Kong & Dietterich and Domingos defined it as the mode. I.e., if a model predicts the label one more than 50% of the time (considering all possible training sets), then the main prediction is 1, and 0 otherwise.

|  |  |  |
| --- | --- | --- |
|  | **Squared Loss** | **0-1 Loss** |
| Single loss |  |  |
| Expected loss |  |  |
| Main prediction | mean (average) | mode |
| Bias |  |  |
| Variance |  |  |

Hence, as result from using the mode to define the main prediction of the 0-1 loss, the bias is 1 if the main prediction does not agree with the true label , and 0 otherwise:

The variance of the 0-1 loss is defined as the probability that the predicted label does not match the main prediction:

Next, let us take a look at what happens to the loss if the bias is 0. Given the general definition of the loss, loss = bias + variance, if the bias is 0, then we define the loss as the variance:

In other words, if a model has zero bias, it's loss is entirely defined by the variance, which is intuitive if we think of variance in the context of being proportional overfitting.

The more surprising scenario is if the bias is equal to 1. If the bias is equal to 1, as explained by Pedro Domingos, the increasing the variance can decrease the loss, which is an interesting observation. This can be seen by first rewriting the 0-1 loss function as

(Note that we have not done anything new, yet.) Now, if we look at the previous equation of the bias, if the bias is 1, we have . If is not equal to the main prediction, but is also is equal to , then must be equal to the main prediction. Using the "inverse" ("1 minus"), we can then write the loss as

Since the bias is 1, the loss is hence defined as "loss = bias - variance" if the bias is 1 (or "loss = 1 - variance"). This might be quite unintuitive at first, but the explanations Kong, Dietterich, and Domingos offer was that if a model has a very high bias such that it main prediction is always wrong, increasing the variance can be beneficial, since increasing the variance would push the decision boundary, which might lead to some correct predictions just by chance then. In other words, for scenarios with high bias, increasing the variance can improve (decrease) the loss!

## Bias Variance Decomposition of a Decision Tree Classifier

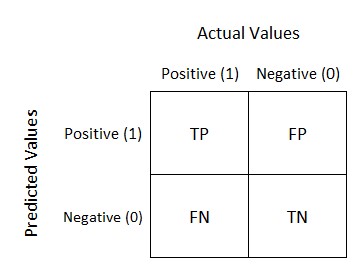
|  |
| --- |
| **from** mlxtend.evaluate **import** bias\_variance\_decomp  **from** sklearn.tree **import** DecisionTreeClassifier  **from** mlxtend.data **import** iris\_data  **from** sklearn.model\_selection **import** train\_test\_split |

|  |  |  |
| --- | --- | --- |
| X, y = iris\_data()  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,  test\_size=0.3,  random\_state=123,  shuffle=**True**,  stratify=y)  tree = DecisionTreeClassifier(random\_state=123)  avg\_expected\_loss, avg\_bias, avg\_var = bias\_variance\_decomp(  tree, X\_train, y\_train, X\_test, y\_test,  loss='0-1\_loss',  random\_seed=123)  print('Average expected loss: %.3f' % avg\_expected\_loss)  print('Average bias: %.3f' % avg\_bias)  print('Average variance: %.3f' % avg\_var)  Average expected loss: 0.062  Average bias: 0.022  Average variance: 0.040  For comparison, the bias-variance decomposition of a bagging classifier, which should intuitively have a lower variance compared than a single decision tree:  **from** sklearn.ensemble **import** BaggingClassifier  tree = DecisionTreeClassifier(random\_state=123)  bag = BaggingClassifier(base\_estimator=tree,  n\_estimators=100,  random\_state=123)  102 | | |
| avg\_expected\_loss, avg\_bias, avg\_var = bias\_variance\_decomp(  bag, X\_train, y\_train, X\_test, y\_test,  loss='0-1\_loss',  random\_seed=123)  print('Average expected loss: %.3f' % avg\_expected\_loss)  print('Average bias: %.3f' % avg\_bias)  print('Average variance: %.3f' % avg\_var)  Average expected loss: 0.048  Average bias: 0.022  Average variance: 0.026 |

### 2.Confusion Matrix

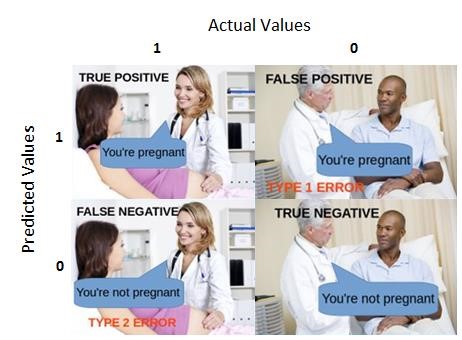
**What is Confusion Matrix and why you need it?**

Well, it is a performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values.



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

Let’s understand TP, FP, FN, TN in terms of pregnancy analogy.



**True Positive:**

Interpretation: You predicted positive and it’s true.

You predicted that a woman is pregnant and she actually is.

**True Negative:**

Interpretation: You predicted negative and it’s true.

You predicted that a man is not pregnant and he actually is not.

#### False Positive: (Type 1 Error)

Interpretation: You predicted positive and it’s false.

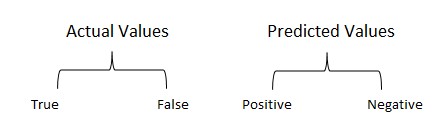
You predicted that a man is pregnant but he actually is not.

#### False Negative: (Type 2 Error)

Interpretation: You predicted negative and it’s false.

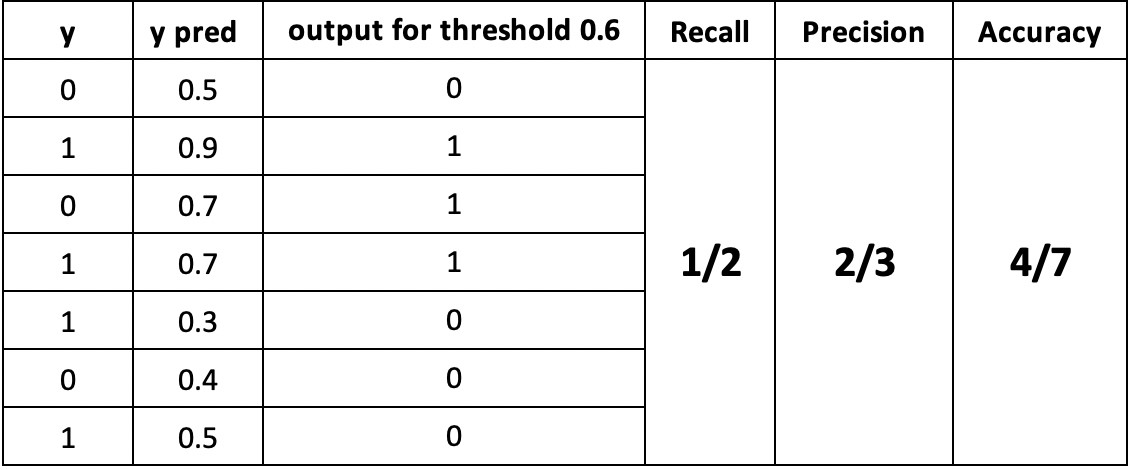
You predicted that a woman is not pregnant but she actually is.

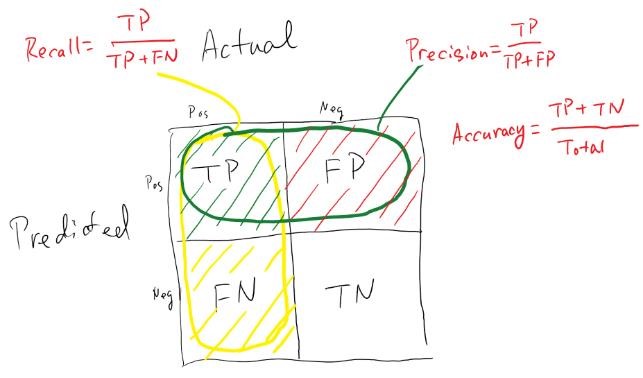
Just Remember, We describe predicted values as Positive and Negative and actual values as True and False.

 Actual vs Predicted

**How to Calculate Confusion Matrix for a 2-class classification problem?**

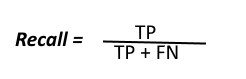
Let’s understand the confusion matrix through math.





Confusion Matrix

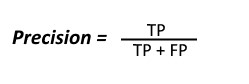
#### Recall



The above equation can be explained by saying, from all the positive classes, how many we predicted correctly.

Recall should be high as possible.

#### Precision



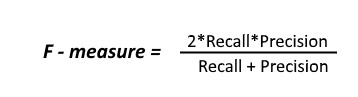
The above equation can be explained by saying, from all the classes we have predicted as positive, how many are actually positive.

Precision should be high as possible.

#### Accuracy

From all the classes (positive and negative), how many of them we have predicted correctly. In this case, it will be 4/7. Accuracy should be high as possible.

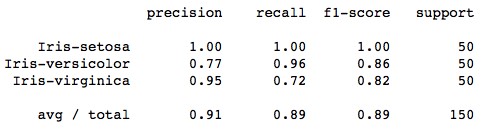
#### F-measure



It is difficult to compare two models with low precision and high recall or vice versa. So to make them comparable, we use F-Score. F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

### 3.Classification Report

A Classification report is used to measure the quality of predictions from a classification algorithm. How many predictions are True and how many are False. More specifically, True Positives, False Positives, True negatives and False Negatives are used to predict the metrics of a classification report as shown below.



The code to generate a report similar to the one above is:

from sklearn.metrics import classification\_report target\_names = ['Iris-setosa',

'Iris-versicolor', 'Iris-virginica']

print(classification\_report(irisdata['Class'],kmeans.labels\_,target\_names=targe t\_names))

The report shows the main classification metrics precision, recall and f1-score on a per-class basis. The metrics are calculated by using true and false positives, true and false negatives. Positive and negative in this case are generic names for the predicted classes. There are four ways to check if the predictions are right or wrong:

1. **TN / True Negative:** when a case was negative and predicted negative
2. **TP / True Positive:** when a case was positive and predicted positive
3. **FN / False Negative:** when a case was positive but predicted negative
4. **FP / False Positive:** when a case was negative but predicted positive

### 4.McNemar Test

**What is the McNemar Test?**

The McNemar test is a [non-parametric test f](https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/parametric-and-non-parametric-data/)or paired [nominal data.](https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/nominal-ordinal-interval-ratio/) It’s used when you are interested in finding a change in proportion for the [paired data.](https://www.statisticshowto.com/paired-data/) For example, you could use this test to analyze retrospective [case-control studies,](https://www.statisticshowto.com/case-control-study/) where each treatment is paired with a control. It could also be used to analyze an experiment where two treatments are given to [matched pairs.](https://www.statisticshowto.com/experimental-design/#MatchedPairsD) This test is sometimes referred to as **McNemar’s Chi-Square test** because the test statistic has a [chi-square distribution.](https://www.statisticshowto.com/probability-and-statistics/chi-square/#chisquaredist)

**Assumptions for the McNemar Test**

The three main assumptions for the test are:

1. You must have one nominal variable with two categories (i.e. [dichotomous variables)](https://www.statisticshowto.com/dichotomous-variable/) and one [independent variable](https://www.statisticshowto.com/independent-variable-definition/) with two connected groups.
2. The two groups in your the [dependent variable](https://www.statisticshowto.com/dependent-variable-definition/) must be [mutually exclusive.](https://www.statisticshowto.com/mutually-exclusive-event/)

In other words, participants cannot appear in more than one group.

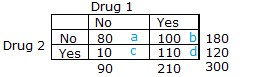
1. Your sample must be a [random sample.](https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/simple-random-sample/)

If your data does not meet these three assumptions, considering running another test for your data like a regular [chi-square test.](https://www.statisticshowto.com/probability-and-statistics/chi-square/)

**Calculating the Test**

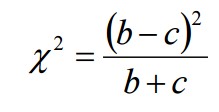
In order to run a McNemar test, your data should be placed into a

2×2 [contingency table,](https://www.statisticshowto.com/what-is-a-contingency-table/) with the cell frequencies equaling the number of pairs. For example, a researcher is testing a new medication and records if the drug worked (“yes”) or did not (“no”). A table is set up with the count of individuals before and after being given the medication. The cell labels a-d are in blue:



Cells b and c are used to calculate the [test statistic;](https://www.statisticshowto.com/test-statistic/) these cells are called “discordant.”

The McNemar test formula is: For the set of data above, we have:

= (100-10)2 / (100 + 10)

= 902 / 110

= 73.63

**5.AUC & ROC Curve**

### Introduction

Some of the accuracy scores you've encountered thus far probably seem pretty impressive; an 80% accuracy seems pretty darn good on the first try! What you have to keep in mind is that for binary classification, you are bound to be right sometimes, even just by random guessing. For example, a person should be roughly 50% accurate in guessing whether or not a coin lands on heads. This also can lead to issues when tuning models down the road. If you have a skewed dataset with rare events (such as a disease or winning the lottery) where there are only 2 positive cases in 1000, then even a trivial algorithm that classifies everything as 'not a member' will achieve an accuracy of 99.8% (998 out of 1000 times it was correct). So remember that an 80% accuracy must be taken into account in a larger context. AUC is an alternative comprehensive metric to confusion matrices, and ROC graphs allow us to determine optimal precision-recall tradeoff balances specific to the problem you are looking to solve.

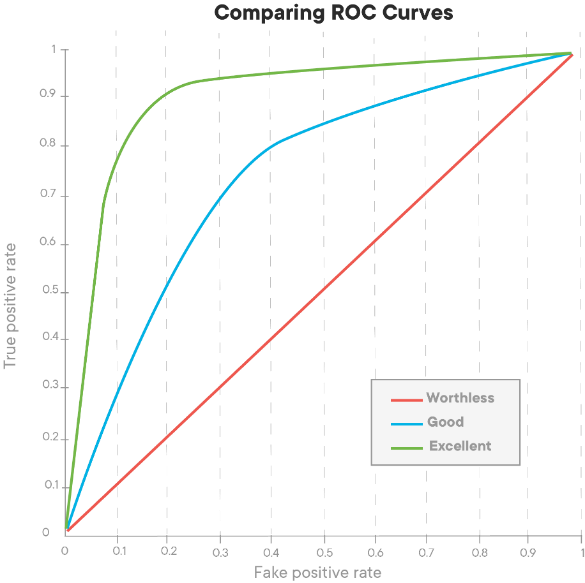
### The ROC curve

The Receiver Operator Characteristic curve (ROC curve) illustrates the true positive rate against the false positive rate of our classifier.

You've already seen the True Positive Rate before, it's another name for **recall**! As a reminder, it's the ratio of the true positive predictions compared to all values that are actually positive.

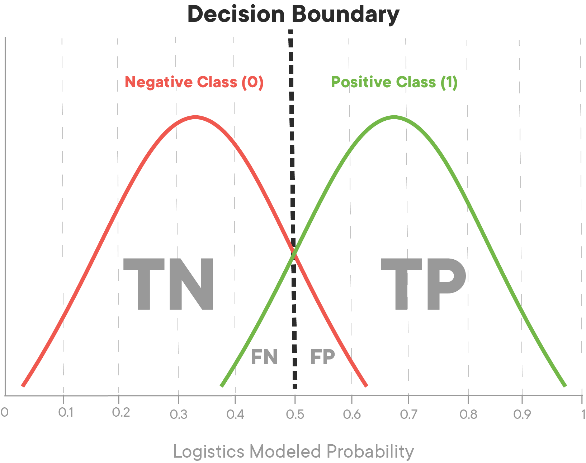
False positive rate is the ratio of the false positive predictions compared to all values that are actually negative.

When training a classifier, the best performing models will have an ROC curve that hugs the upper left corner of the graph. A classifier with 50-50 accuracy is deemed 'worthless'; this is no better than random guessing, as in the case of a coin flip.

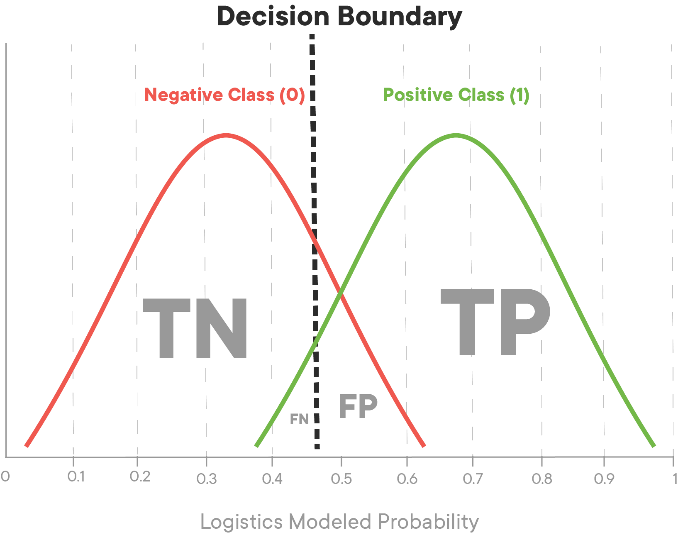


The ROC curve gives us a graph of the tradeoff between this false positive and true positive rate. The AUC, or area under the curve, gives us a singular metric to compare these. An AUC of 1 being a perfect classifier, and an AUC of 0.5 being that which has a precision of 50%.

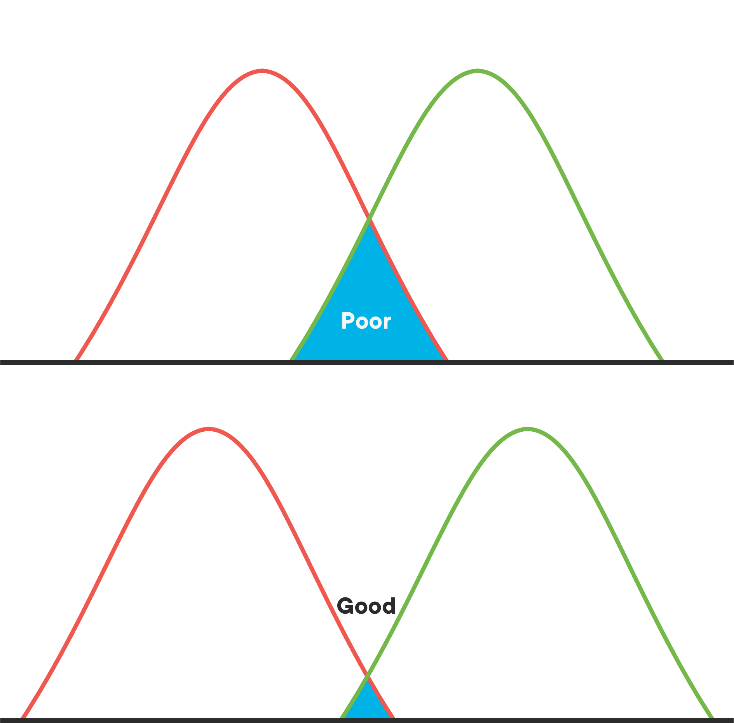
Another perspective to help understand the ROC curve is to think about the underlying model fueling our classification algorithm. Remember that the logistic model produces probabilities that each observation is of a specific class. Imagine that the values produced from the logistic model look something like this:



Here you see the majority of the two classes probabilities land at around 0.25 or 0.75. If we alter the cutoff point, it could sacrifice precision, increasing the false positive rate in order to also increase the true positive rate or vice versa. Imagine in this instance that green is the positive case 1 (in this case heart disease) and red the negative case 0. Shifting the decision boundary to the left from 0.5 will result in capturing more of the positive (1) cases. At the same time, it will pick up some false positives, those red cases at the far right of the negative (0) case distribution that will be incorrectly identified as being part of the positive case distribution.



Models with poor ROC might have large overlaps in the probability estimates for the two classes. This would indicate that the algorithm performed poorly and had difficulty separating the two classes from each other.



With that, let's take a look at drawing the ROC curve in practice.

#### Train a classifier

from sklearn.linear\_model import LogisticRegression from sklearn.model\_selection import train\_test\_split import pandas as pd

# Load the data

df = pd.read\_csv('heart.csv')

# Define appropriate X and y y = df['target']

X = df.drop(columns='target', axis=1)

# Normalize the Data

X = X.apply(lambda x : (x - x.min()) /(x.max() - x.min()),axis=0)

# Split the data into train and test sets.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

# Fit a model

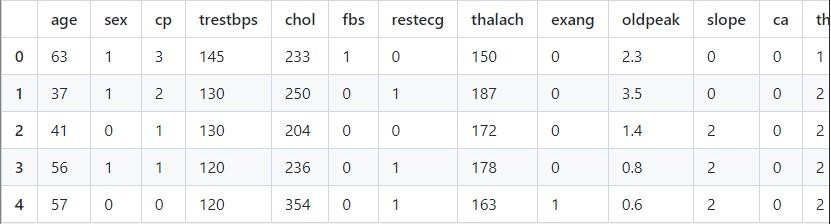
logreg = LogisticRegression(fit\_intercept=False, C=1e12, solver='liblinear') logreg.fit(X\_train, y\_train) print(logreg) # Preview model params

# Predict

y\_hat\_test = logreg.predict(X\_test)

# Data preview print("") df.head()

LogisticRegression(C=1000000000000.0, class\_weight=None, dual=False, fit\_intercept=False, intercept\_scaling=1, l1\_ratio=None, max\_iter=100, multi\_class='warn', n\_jobs=None, penalty='l2', random\_state=None, solver='liblinear', tol=0.0001, verbose=0, warm\_start=False)



#### Draw the ROC curve

In practice, a good way to implement AUC and ROC is via sklearn's built-in functions:

from sklearn.metrics import roc\_curve, auc

# Scikit-learn's built in roc\_curve method returns the fpr, tpr, and thresholds # for various decision boundaries given the case member probabilites

# First calculate the probability scores of each of the datapoints: y\_score = logreg.fit(X\_train, y\_train).decision\_function(X\_test)

fpr, tpr, thresholds = roc\_curve(y\_test, y\_score) From there it's easy to calculate the AUC:

print('AUC: {}'.format(auc(fpr, tpr))) AUC: 0.8823114869626498

#### Putting it all together as a cohesive visual

import matplotlib.pyplot as plt import seaborn as sns %matplotlib inline

# Seaborn's beautiful styling

sns.set\_style('darkgrid', {'axes.facecolor': '0.9'})

print('AUC: {}'.format(auc(fpr, tpr))) plt.figure(figsize=(10, 8)) lw = 2

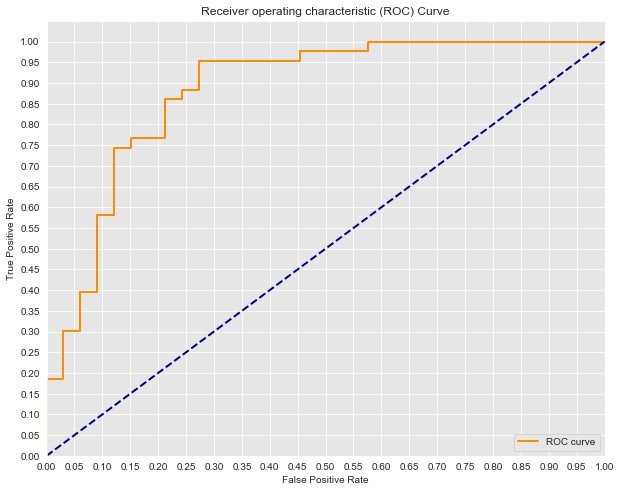
plt.plot(fpr, tpr, color='darkorange', lw=lw, label='ROC curve')

plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--') plt.xlim([0.0, 1.0]) plt.ylim([0.0, 1.05])

plt.yticks([i/20.0 for i in range(21)]) plt.xticks([i/20.0 for i in range(21)]) plt.xlabel('False Positive Rate') plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic (ROC) Curve') plt.legend(loc='lower right') plt.show()

AUC: 0.8823114869626498



### Summary

The **Receiver Operating Characteristic** curve (ROC) which graphs the *False Positive Rate* against the *True Positive Rate*. The overall accuracy of a classifier can thus be quantified by the AUC, the **Area Under the Curve**. Perfect classifiers would have an AUC score of 1.0 while an AUC of 0.5 is deemed trivial or worthless.

# ENVIRONMENT AND DEPLOYMENT

## 1.Streamlit

Streamlit is an open-source python framework for building web apps for Machine Learning and Data Science. We can instantly develop web apps and deploy them easily using Streamlit. Streamlit allows you to write an app the same way you write a python code. Streamlit makes it seamless to work on the interactive loop of coding and viewing results in the web app.

### Data flow

Streamlit allows you to write an app the same way you write a python code. The streamlit has a distinctive data flow, any time something changes in your code or anything needs to be updated on the screen, streamlit reruns your python script entirely from the top to the bottom. This happens when the user interacts with the widgets like a select box or drop-down box or when the source code is changed.

If you have some costly operations while rerunning your web app, like loading data from databases, you can use streamlit’s st.cache method to cache those datasets, so that it loads faster.

**What we use in classification Laboratory is given below.**

### Displaying the data

Streamlit provides you with many methods to display various types of data like arrays, tables, and data frames.

* To write a string simply use, st.write(“Your string”)
* To display a data frame use, st.dataframe method
* To display object or string as a pretty-printed JSON string use, st.json

### Displaying the charts

Streamlit provides you with many methods to display various types of Charts like line chart, bar plot ,map ,matplotlib, pyDeck,plotly etc.

* To display a matplotlib.pyplot figure use ,st.pyplot
* To draw a chart using the PyDeck library, This supports 3D maps, point clouds : use , st.pydeck\_chart

### Displaying the input widget

Streamlit provides you with many methods to display various types of

Input widget like, button, checkbox, radio button etc

* To display a checkbox widget use , st.cheakbox
* To display a select widget use , st.selectbox
* To display sider widget use , st.slider
* To display number input widget use , st.number\_input
* To display file uploader widget use , st.file\_uploader **2. Git and GitHub**

**What is Git?**

[Git](https://git-scm.com/) is a widely used **Version Control System (VCS)** that lets you keep track of all the modifications you make to your code. This means that if a new feature is causing any errors, you can easily roll back to a previous version.

But Git isn’t just any VCS, it’s a Distributed VCS. This means that every collaborator of the project will have a history of the changes made on their local machine. So people can work on different features of the project without having to communicate with the server hosting the remote version of the project. This is super efficient and you can easily merge any changes made to the project with the remote copy.

Since it is written in the C language, speed and performance are ingrained in Git right from its inception. Besides this, Git also provides a lot of buffers before actually saving any changes to the project.

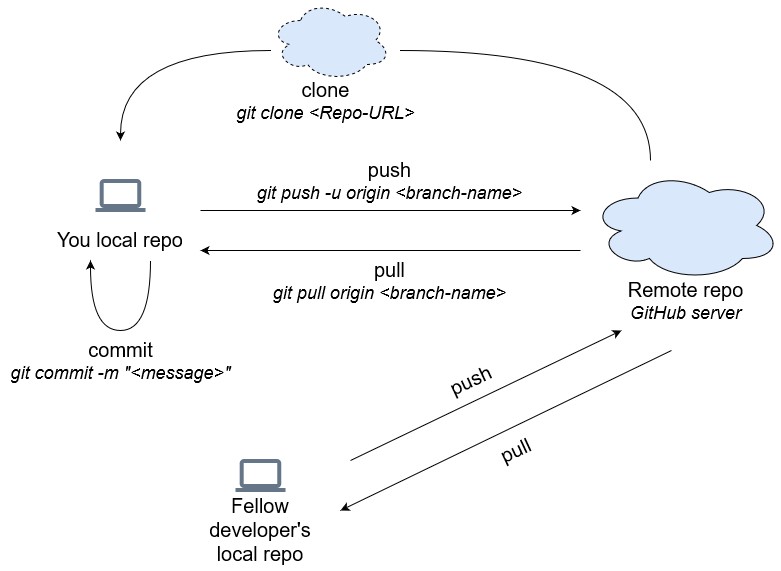
**What is GitHub?**

[GitHub](https://github.com/) is a widely used platform for version control that uses Git at its core. It lets you host the remote version of your project from where all the collaborators can have access to it. Not just your own team members, but any member of GitHub can contribute to your code (that is of course if you choose to accept the changes made). We will discuss all of this in detail in this article.

GitHub is like a social platform where you can find a plethora of open-source projects with their codes. All the new and emerging technologies can be found on this platform. You can collaborate on amazing projects and have discussions on your contributions! This is the best open-source platform you’ll find and is a data scientist’s dream!

**Getting Familiar with the Git Terminology**

### Repository



**Repository** or **Repo** is a folder that contains all the project files and the history of the revisions made to each file. There are two repositories of your project that you will work with throughout the lifetime of your project – Remote repo and Local repo:

* **Remote repo** contains your project that can be accessed from anywhere and by anyone. Your remote repository lives on the GitHub server and anybody can access them
* **Local repo** is a copy of the remote repo that resides on your local machine. All the changes you make will be saved in your local repo. Your fellow developers will not be able to see them yet until you push them to the remote repo

### Cloning git clone <Repo-URL>

**Cloning** means creating a copy of the remote repo on your local machine. Now you can make changes to the project on your local machine.

**Commit** **git commit -m “<commit message>”**

When you **commit** a change, you save the changes you made to your files in the repo. When working with Git from your local machine, using the commit command will save your files in the local repo. To make those changes in the remote repo, you will use the push command.

### Push git push origin <branch>

**Push** command allows you to transfer all the changes on your local repo to the remote repo. Now all the fellow developers will have access to the changes you made and they can update their local repositories.

### Pull git pull <remote-repo>

If push meant transferring code to the remote repo, the **Pull** command allows you to transfer all the changes from the remote repo to your local repo. So any changes that your fellow developer pushed to the remote repo, you can transfer them to your local repo using the pull command.

There are a few more terms that you will need to know but they are not required right now. We will cover them in detail in the latter part of this article. For now, let’s create our very first GitHub repository!

### Getting Started with Git

Git programs are designed to work with a Unix style command-line environment. Linux and macOS already have an interface for this in their native command-line terminals. So all the git commands that I will be using in this article should work fine with their terminals.

Windows, however, has a completely different command-line interface called Command Prompt which is not a Unix style command-line environment. So what do we do? Well, don’t worry, you already installed **Git Bash** when you installed Git.

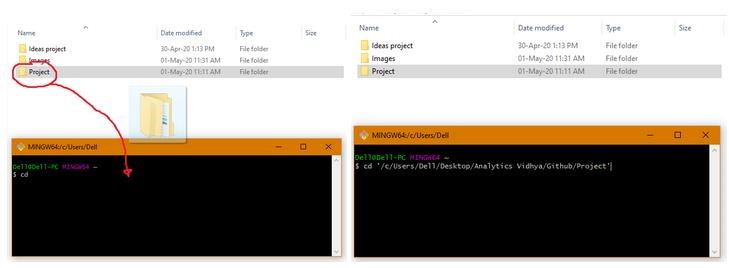
Git Bash is a command-line interface for Windows that emulates the Git command-line experience. So, as long as you are implementing Git commands inside Git Bash, you should be fine.

The next thing I want you to do is to create a project folder where you will save your local repository. Then follow these steps to open your terminal inside that project folder:

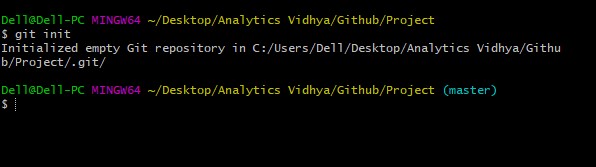
1. Open up terminal or Git Bash
2. Type the command **cd** which means change directory (or a folder)
3. Navigate to where your project folder is located but don’t open it up
4. Now just drag and drop the folder inside the terminal. This will navigate the terminal to your project directory/working directory

You are ready to start working with Git!

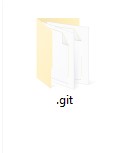
Initializing a Repository



Repository or Repo is a folder that contains all the project files and the revision made to each file. The project directory you made above isn’t a repository. A repo needs to be initialized using the [git init](https://git-scm.com/docs/git-init) command.



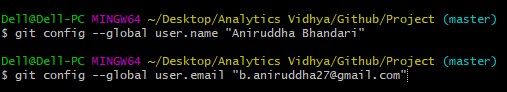
Once you do that, a hidden **.git** folder will be created inside your project/working repository. This is your local Git repo. *If you don’t see it, it’s probably because it is hidden and you need to change some properties in the settings to make it visible. But don’t worry, it is still there even if you don’t see it.* Git will store all the changes you make to your project files inside this folder.



### Configuring Git

But before we make any changes to our repo, we want Git to know who we are. We can do that using the [git config](https://git-scm.com/docs/git-config) command. Using this we can set the user name and user email address. Now every time we make a commit, this information is saved by Git so that you know who made that change.

### git config — global user.name <user-name> git config — global user.email <user-email>



If you use the **–global** option, then Git will save this information for all the repositories in the system. So, you can leave it out if you only want the information to be saved for this particular repository.

### Learn How to Commit Files in Git

Now, you can start building your project and adding files to your project directory. My project is about writing *“Hello world”* in different programming languages. So, I am going to add a few files to my project directory for that purpose. You should do the same for your project directory too as it is empty right now.

Once you have created and added files to your project directory, you can add them to your local Git repository using **git add <file-name>**



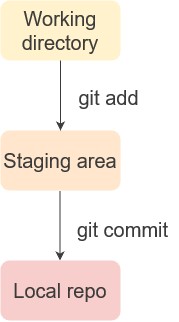
If you want to add more than one file at a time, use **git add**



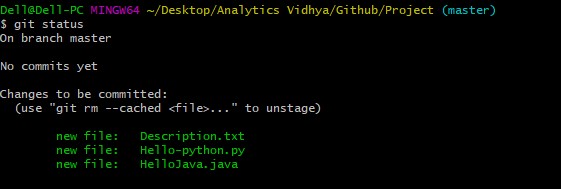
Are we done? Wasn’t the command for committing changes something else?

You are absolutely right! We haven’t added the files to the local repo yet. We have just told Git that some changes were made and we want to save these changes in the next commit/save. As of now, these “added” files are in a place called the **Staging area**.

*Staging area is an intermediate place between your working directory and local Git repo where any changes that you made can be reviewed before you actually commit them to the repo.*



You can check the state of the staging area using **git status**



You will see a message similar to the one shown above which tells you that a change was made that needs to be committed. All the pending commits will be shown here.

Now you can take a snapshot of all the changes you made, which are reflected in the staging area, and save them in the Git repo using **git commit -m “<commit message>”**

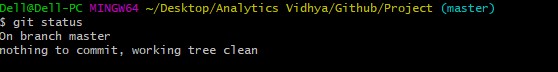
Your commit message should be terse but lucid so that fellow developers can easily determine why you made that change.



Once you do that, you will get the following message:

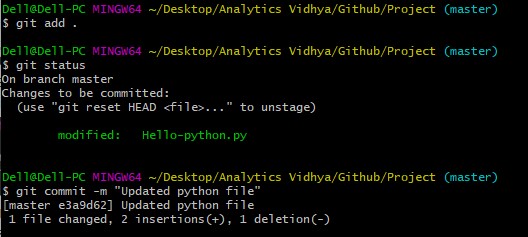
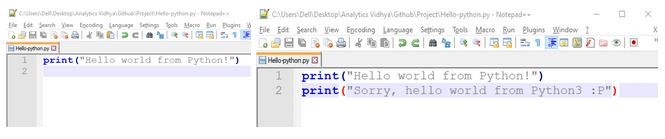


Now all your files have been committed to the Git repo. You can check the status of the staging area and this time it will reflect that there is nothing to commit:



**Henceforth, whenever you make a change to any of the files, like a bug fix or anything, and add it to the staging area, Git will know which files the changes were made to and will record the entire content of the file in the commit.**

I am going to make a few changes to one of my code files and commit it after the changes are made:



As you can see, when I added my files again, Git was smart enough to know that I only made changes to a single file which is reflected in its output. The commit that I made after this only updated that specific file and not the others.

### Viewing Logs

The whole point of version control is to keep a record of the changes that were made. You can do this using the [git log](https://git-scm.com/docs/git-log) command. It gives you a complete view of all the commits that were made in reverse chronological order:



As you can see, my name, email address, timestamp, and the commit message are all reflected in the logs. This makes it fairly easy to track who made what changes and determine when the bug was first introduced in the project.

### Uploading To Remote Repo on GitHub

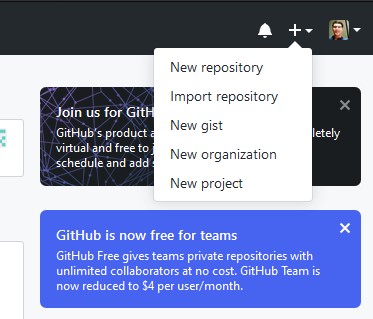
So far, we were working on a local repository, meaning all the changes that were made were tracked in your local machine and our fellow developers were not able to see them yet. To make that happen, you need to create a remote repository – a repository that can be accessed from anywhere and by anyone. This is where GitHub comes in!

Your remote repository lives on the GitHub server and anybody can access it. So let’s create a remote repository!

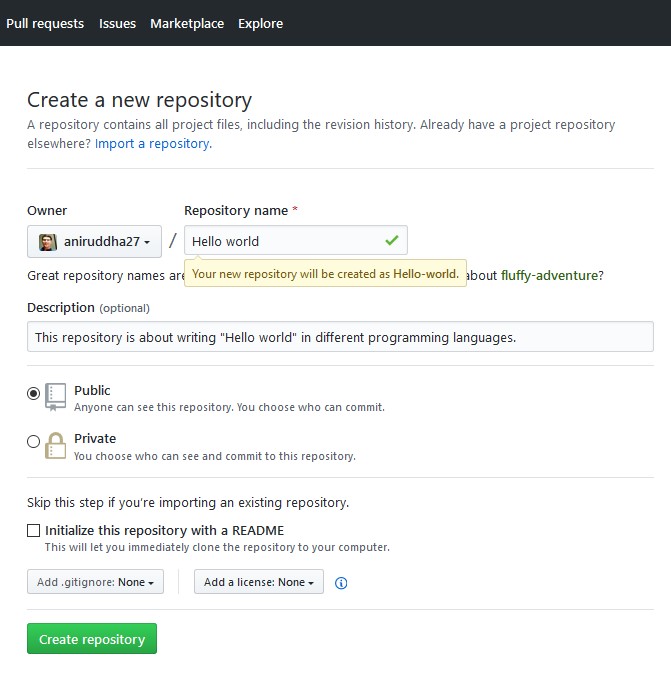
*Note: You would need to create a GitHub account for this.*

***Steps to create a remote repository***

1. In the upper right-hand corner, click the **+** sign and click **New Repository**:

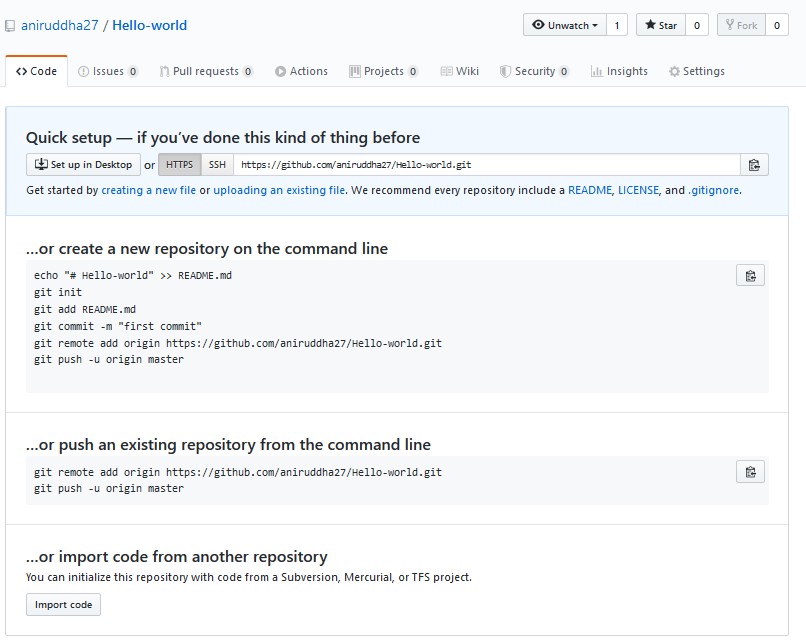


1. On the next page, give a name to your repository and a short description. Once you are done, click **Create repository**:



### Adding Git Remote to Your Repository

Once you have created your GitHub repository, GitHub will prompt you to upload your files to the remote repository:



As we have already created our local repository, we first need to sync our local and remote repos. We can do this using **git remote add origin <URL>**

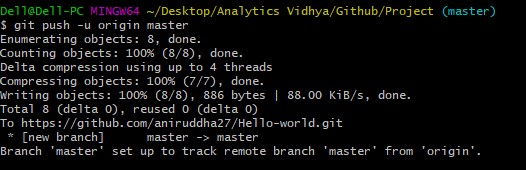


The command creates a connection between the local and remote repos. Once we do that, we no longer have to refer to the remote repo by the URL every time. We can just use the name **origin** to refer to the remote repo.

### Push using Git

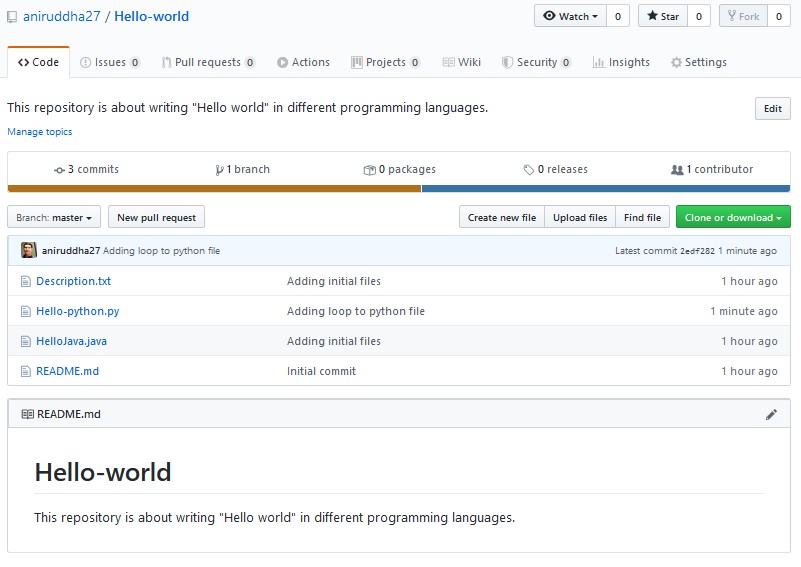
Now that the remote repo has been added, all you have to do is **push** your commits from local repo to the remote repo so that all your fellow developers can view the changes.

You transfer the local repo to the remote repo on GitHub server using **git push -u origin <branch-name>**



*Origin is the name of our remote repository.*

Now all our fellow collaborators have access to this newly updated repository.

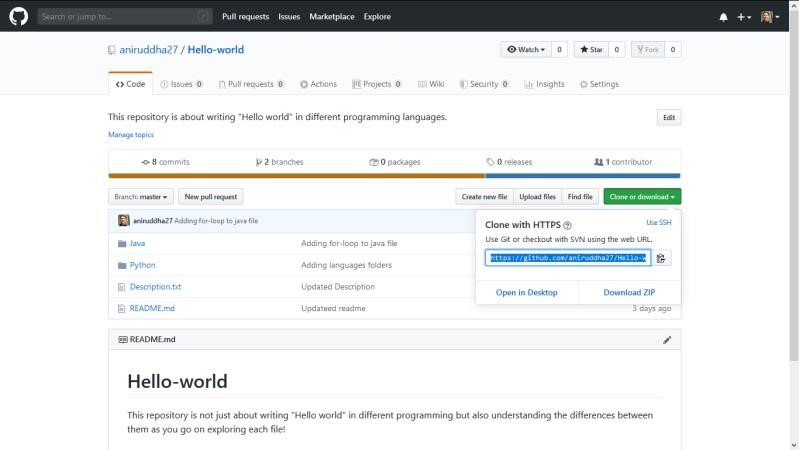


### Cloning a GitHub Repository

We saw how we can create our own local repository and push it to GitHub. Now, if you weren’t the one who created this repo, you would have to make a copy of this on your local machine.

Once a project has been uploaded onto the remote repository, developers can get their own copy of this repo using the [git clone](https://git-scm.com/docs/git-clone) command. Developers can then work on their local copy of the repo, make changes to it, and upload it to the remote repo.

It is very easy to clone remote repos from GitHub. Just head over to whichever repository you want to clone and click the **Clone or download** button to copy the URL:

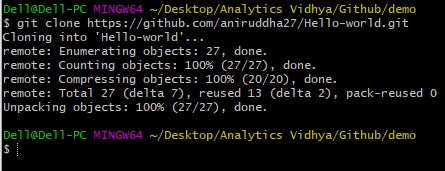


*You could also download the repo directly from here as a Zip file, but we will use Git to download it from our terminal.*

To clone it to your local machine, you need to head over to your terminal and provide the URL in the following command”

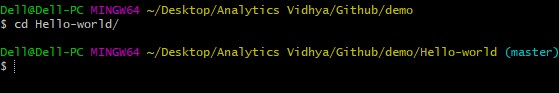
**git clone <URL>**

This will make a local copy of the repo on your local machine.



You will notice that a folder with the same name as the remote repository is created inside your current directory. This folder is your project/working directory containing the local Git repo. You need to navigate inside this directory to make changes to your local repo.

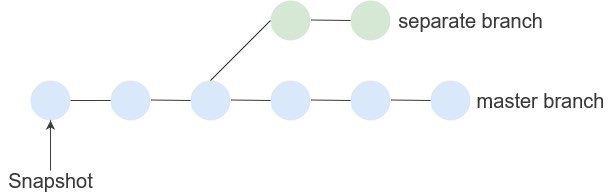
Use the **cd <repository-name>/** to navigate inside your working directory:



Now that you are inside your working directory, you can make any changes you like. If you like those changes and think it will solve a bug or add a really cool feature to the project, just commit to your local repo first, and then push it to the remote repo on GitHub so that your fellow developers are up to date on the new change.

### Branching and Merging

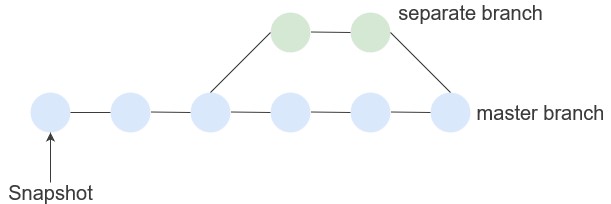
**Branching** is one of the most fundamental features of Git. Branches let you work on new features or bug fixes of the main project code that is present on the **master** branch.



Branches are like a reference to a commit. Any changes that you now make to this branch will ensue from this point onwards. Even if you mess up with this one, rest assured as it will do no harm to your actual working code.

Branches let you experiment with new features or ideas and even let you create multiple branches in parallel to experiment with different features. Any number of people can work on a particular branch and you can have as many branches as you want.

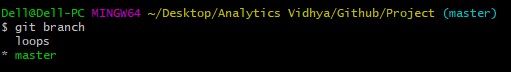
You can create a new branch using [git branch.](https://git-scm.com/docs/git-clone) This will contain all the files that are present in the master branch. You can make changes to files on this branch. Once you are confident that your code is working well, you can integrate it with the master branch using the [git merge](https://git-scm.com/docs/git-merge) command.



So far we have been working on the **master** branch. Now we will see how to create a new branch using **git branch <branch-name>**

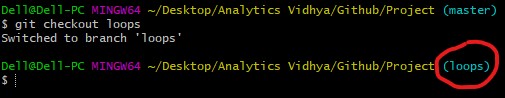


This will create a new branch and you can check that out using the git branch command:

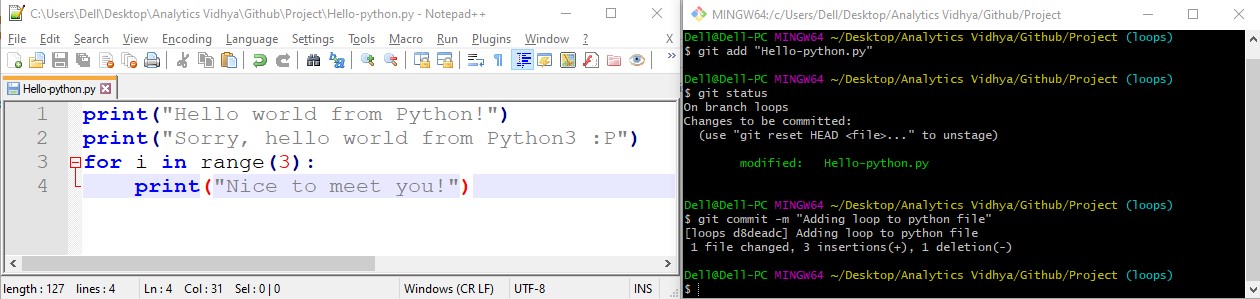


*Git will always keep you up to date on which branch you are working on by mentioning it within parenthesis at every command. This will ensure you are working on the right branch.*

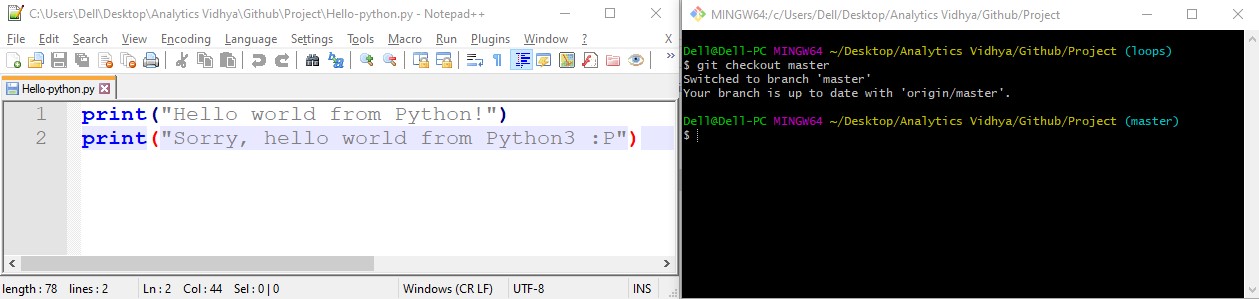
Let us now work in a different branch using **git checkout <branch-name>**



Right now I am working in the *loops* branch. I want to make changes to one of the files in my repo by adding a *for-loop* to my code. I will do that and then commit the changes:

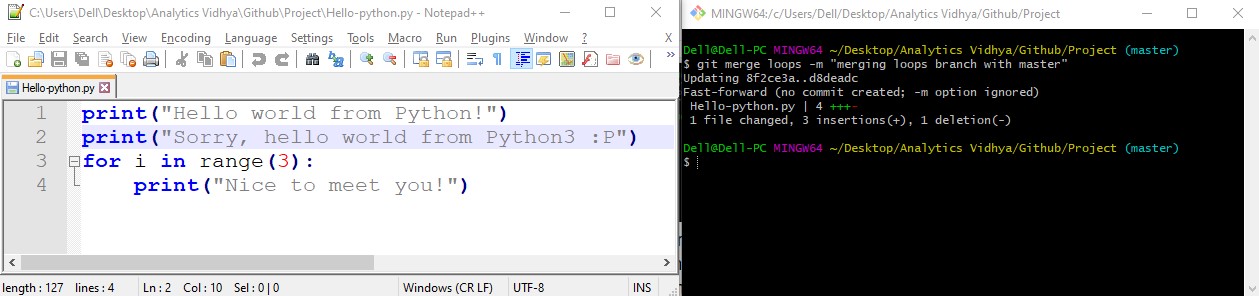


Once I have done that, let me just move back to the **master** branch and chec k the change that I had made:



The change I made has not reflected in the master branch! And that is exactly what should have happened. When I made the commit, I was inside the *loop* branch so my changes were saved in that specific branch. Hence, the change did not get committed to the master branch. My main code is safe.

Next, if I wanted to commit those changes to my master branch as it is free from any bugs and I love the new “loops” feature, I could do that using the **git merge <branch name> -m “<message>”** command from the branch you want to merge it to (the master branch in this case). This will merge the *loop* branch with the master branch:

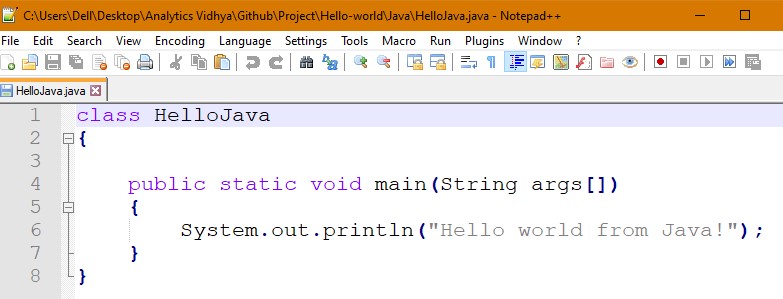


We have added a fully functional new feature to our main code. Now all that is left to do is to commit these changes to the remote repository!

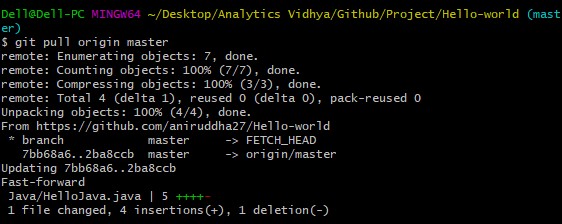
### Pull using Git

The last Git command you need to know is **pull**. This lets you fetch any update from the remote repo and merge it with the local repo. For instance, after you have cloned your remote repo, someone has made an update to the remote repo on some branch. Now you need to fetch these changes and merge with your local repo so that you are up to date on this new change.

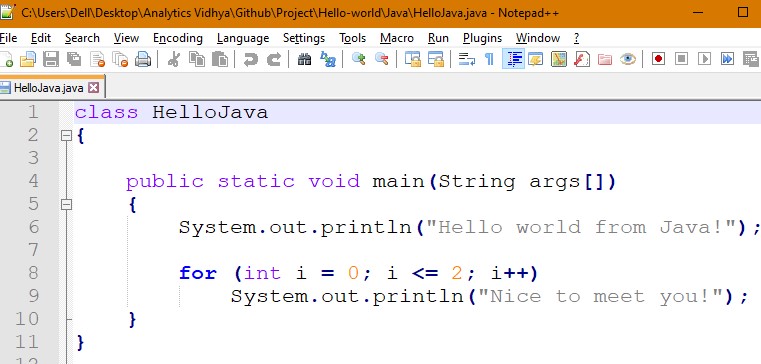
Let’s say after I added a *for-loop* to the Python code file, someone added a *forloop* to the Java code file as mine is without a *for-loop*:



I can update my local repo using the pull command. All you have to do is type **git pull origin <branch-name>**



This will update my local repo. Now even I have a *for-loop* in my Java file!



### Forking and Contributing to the World

With GitHub, you can explore anybody’s repository. That’s the beauty of opensource, right? There will be times when you will genuinely like someone’s project and would be tempted to contribute to it. Or if you want to suggest fixes to someone’s project, it is better to make those fixes and then send a pull request so that you can contribute to their project.

This can be achieved by a process called **forking.**

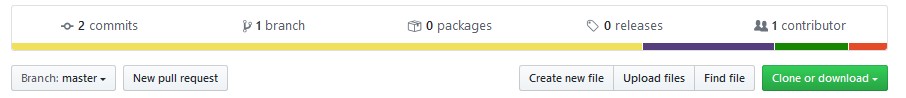


To fork someone’s repository, head to the repository page and on the top, click **Fork**. This will create a copy of the repository in your account.

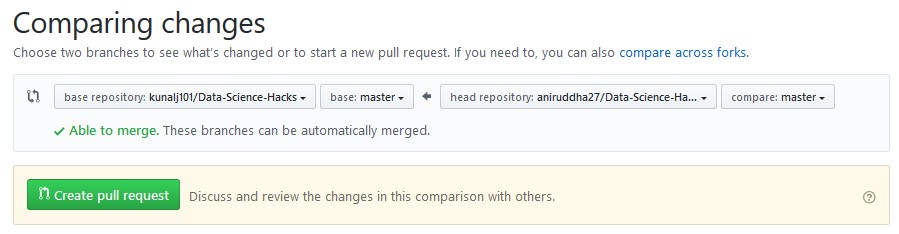
Now you can clone this repository onto your local machine, make the changes you want to make or the features you want to add, and push it to your remote copy.

Finally, you can request the creator of the project to accept the fixes that you have made or the new features you added. This is called a **pull request**.

When you navigate to the repository in your profile, you will see an option called ‘**New pull request**‘:



Click that button and GitHub will take you to the next page where it will show you the branches you want the original creator to merge:



Once you create a pull request, you will be prompted to define the changes that you made to the original creator’s repository. You can provide as much description as you want about the changes that you made. And finally, send the pull request.

Now the ball is in the original creator’s court. Either they will accept your changes and merge it with their branch in which case congratulations as you just made an open-source contribution and you should be proud of yourself! But even if they don’t, don’t be heartbroken There are a plethora of other open-source projects waiting for your attention. So move on and keep making those contributions!

## 3.Heroku

Heroku is a cloud service platform whose popularity has grown in recent years. Heroku is so easy to use that it’s a top choice for many development projects.

With a special focus on supporting customer-focused apps, it enables simple [application development](http://mentormate.com/blog/rethink-know-offshore-software-development/) and deployment. Since the Heroku platform manages hardware and servers, businesses that use Heroku are able to focus on perfecting their apps. And not the infrastructure that supports them.

**Model Deployment using Heroku:**

Heroku is a container-based cloud Platform as a Service (PaaS). Developers use Heroku to deploy, manage, and scale modern apps. Our platform is elegant, flexible, and easy to use, offering developers the simplest path to getting their apps to market.

Heroku is fully maintained, allowing developers to focus on their core product rather than having to worry about servers, hardware, or infrastructure. It provides tools, services, workflows, and support for polyglot—all intended to improve developer productivity.

We’re ready to start our Heroku deployment now that our model has been trained, the machine learning pipeline has been set up, and the application has been tested locally. There are a few ways to upload your application source code onto Heroku. The easiest way is to link a GitHub repository to your Heroku account.

### Heroku requires three files for model deployment

* The first file is the Procfile that contains three pieces of information: application type, server information, and file name from where the application should start.
* The second file is the requirement.txt file that contains all the libraries used during model building.
* The third file is clf.py file. (classification laboratory’s python file) In the end, the project directory should have the following structure:



### Model deployment

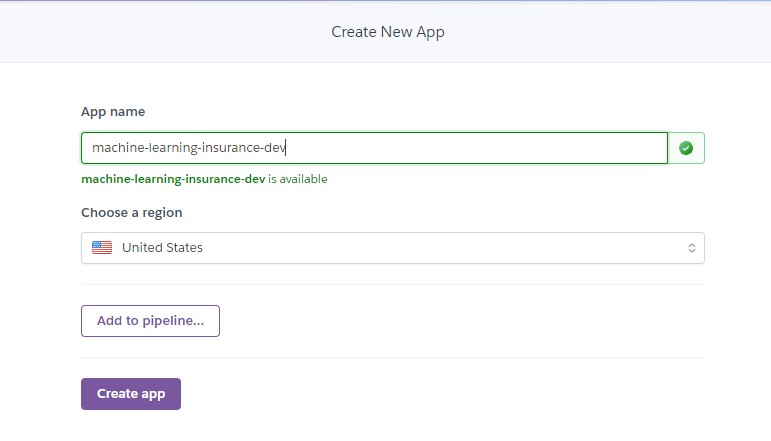
Heroku offers three methods of model deployment: Heroku git, GitHub, and container registry.



This article shows model deployment using the GitHub method.

1. Upload the project folder to a git repository. Please make sure that the repository is **public**.

1. Enter **App name and region.**



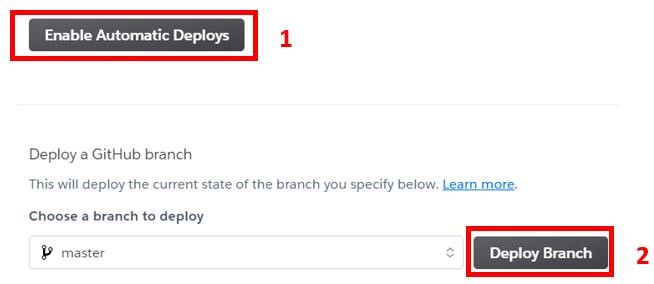
1. Connect to your GitHub repository where code is uploaded.

In the

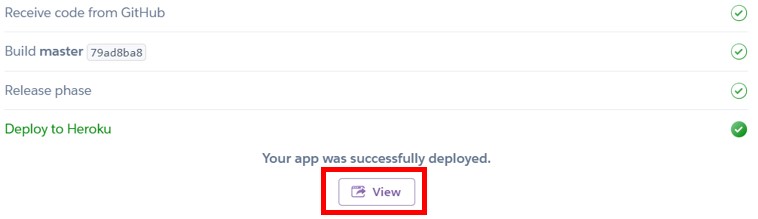
search, choose the repository where you had uploaded the project code.



**4.**After connecting the repository to Heroku, it is time to deploy a repository branch. Heroku also offers automatic deploys, which means Heroku deploys the new version of the app automatically whenever the branch is updated.



**5.** Start the deployment by clicking on the Deploy branch button. Upon the successful completion of deployment of the app, click on the View button.



**The app is published at URL: https://classification-laboratoryapp.herokuapp.com**

# LIMITATIONS

* It can not be used when the output variable is not of binary nature.
* It can give misleading results if the data contains missing values.
* The procedure of hyper parameter tuning in the case of random forest and Ada boosting algorithms, might be time consuming.
* Decision tree, K-Nearest neighbor, and Ada boosting algorithm estimates the parameter in such a way that it focuses on minimizing only the Type-2 error.
* The algorithms may take too long to generate the report if the data is too huge ( say 5-10 MB ).

# SUGGESTIONS

* The results would be more efficient and effective if sampling techniques could be used in proper way.
* If the hyper parameter tuning had been implemented optimally in the cases of Logistic regression, gradient boosting, and Random forest, the results may have been much better. We utilised randomised grid search in our case, although GridSearchCV is a superior approach that is more time consuming.

# BIBLIOGRAPHY

**Language:**

* Python

**Cloud:**

* Google colaboratory
* Git

**Software:**

* VS Code
* Ms Office

**Platform:**

* Github

**Framework:**

* Streamlit

**Libraries:**

* streamlit
* pandas==1.4.2
* numpy==1.22.3
* sklearn
* seaborn
* matplotlib
* mlxtend
* plotly==5.7.0
* Pandas-Profiling==3.1.0
* streamlit-pandas-profiling==0.1.3

**Reference website:**

* [https://www.analyticsvidhya.com](https://www.analyticsvidhya.com/)
* [https://towardsdatascience.com](https://towardsdatascience.com/)
* [https://medium.com](https://medium.com/)
* <http://rasbt.github.io/mlxtend>
* <https://scikit-learn.org/stable>
* [https://www.statisticshowto.com](https://www.statisticshowto.com/)
* <https://github.com/krishnaik06>
* <https://github.com/dataprofessor>
* [https://www.javatpoint.com](https://www.javatpoint.com/)
* [https://streamlit.io](https://streamlit.io/)
* [https://www.heroku.com](https://www.heroku.com/)
* [https://app.diagrams.net](https://app.diagrams.net/)
* [https://www.streamlinehq.com](https://www.streamlinehq.com/)
* [https://github.com](https://github.com/)
* [https://www.kaggle.com](https://www.kaggle.com/)
* [https://www.slidescarnival.com](https://www.slidescarnival.com/)

**Book Reference**

* Applied Multivariate Statistical Analysis by RICHARD A. JOHNSON University of Wisconsin-Madison & DEAN W. WICHERN Texas A&M University