Andhra Pradesh State Skill Development Corporation

eSkill AP
Learn Anytime Anywhere
Andhra Pradesh State Skill Development Corporation

Machine Learning

Ensemble Learning and Random Forests

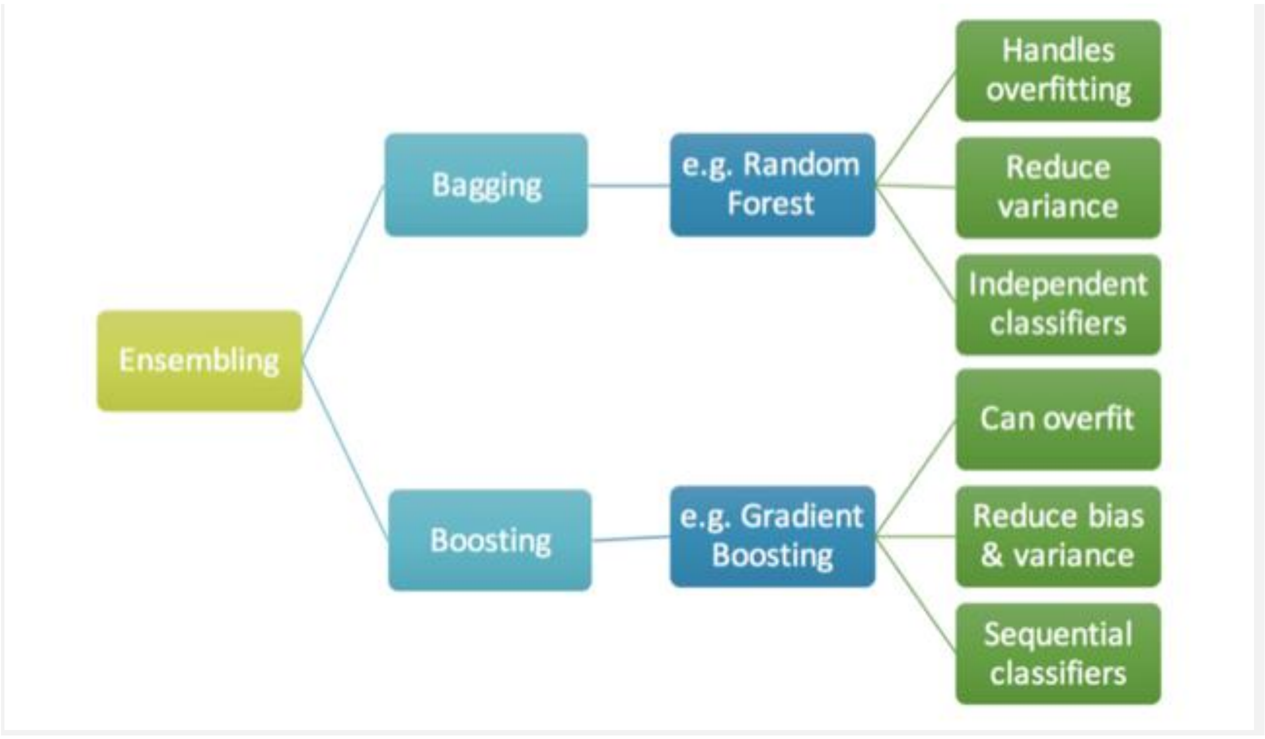
CHAPTER 9

Ensemble Learning and Random Forests

Introduction to Ensemble Learning

Ensemble Learning combines the Decisions from Multiple Models to improve the Overall Performance As Ensemble methods are meta-algorithms that combine several machine learning techniques into one predictive model in order to Decrease variance (Bagging), Bias (Boosting) along with majority votes (Random Forest ,Voting Classifier)

The goal of **ensemble algorithms** is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator.



There are two families of ensemble methods which are usually distinguished:

1. **Averaging methods.** The driving principle is to build several estimators independently and then to average their predictions. On average, the combined estimator is usually better than any of the single base estimators because its variance is reduced. **Examples:** Bagging methods, Forests of randomized trees.
2. **Boosting methods.** Base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. **Examples:** AdaBoost, Gradient Tree Boosting.

The three most popular methods for combining the predictions from different models are:

1. **Bagging.** Building multiple models (typically of the same type) from different subsamples of the training dataset.
2. **Boosting.** Building multiple models (typically of the same type) each of which learns to fix the prediction errors of a prior model in the sequence of models.
3. **Voting.** Building multiple models (typically of differing types) and simple statistics (like calculating the mean) are used to combine predictions.

Understanding Random Forest

Random forest is a **Supervised Learning algorithm** which uses ensemble learning methods for **classification and regression**.

Random forest is a **bagging** technique and **not a boosting** technique. The trees in **random forests** are run in parallel. There is no interaction between these trees while building the trees.

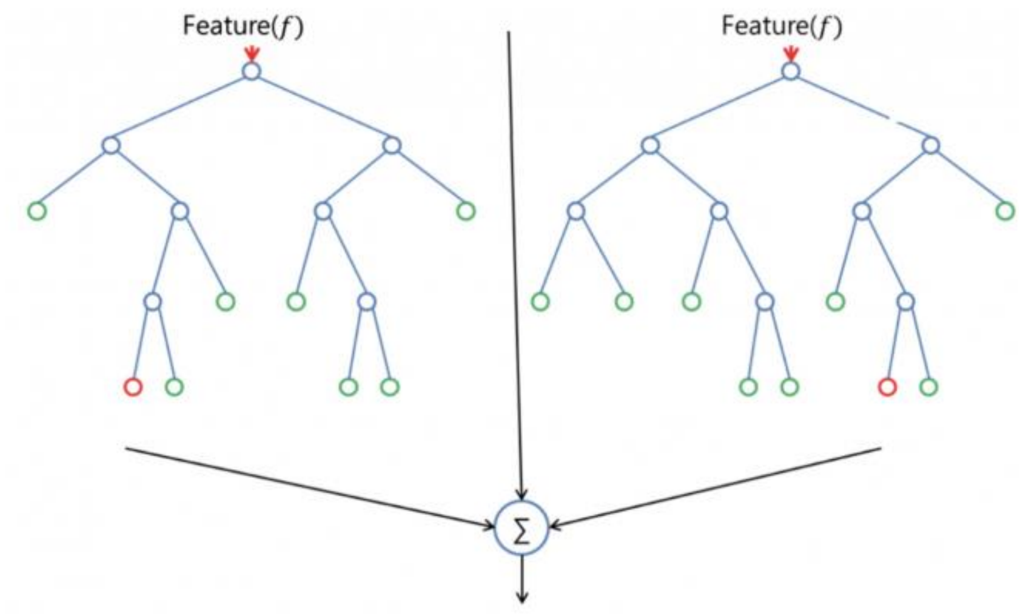
It operates by constructing a multitude of decision trees at training time and outputting the class that is the **mode** of the **classes (classification)** or **mean prediction (regression)** of the individual trees.

A random forest is a meta-estimator (i.e. it combines the result of multiple predictions) which **aggregates many decision trees**, with some helpful modifications:

1. The number of features that can be split on at each node is limited to some percentage of the total (which is known as the **hyperparameter**). This ensures that the ensemble model **does not rely too heavily on any individual feature**, and makes **fair use of all potentially predictive features**.
2. Each tree draws a random sample from the original data set when generating its splits, adding a further element of randomness that prevents **overfitting**.

Put simply: random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. Let's look at random forest in classification, since classification is sometimes considered the building block of machine learning. Below you can see how a random forest would look like with two trees:



Random forest has nearly the same hyper parameters as a decision tree or a bagging classifier. Fortunately, there's no need to combine a decision tree with a bagging classifier because you can easily use the classifier-class of random forest. With random forest, you can also deal with regression tasks by using the algorithm's regressor.

Random forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

Therefore, in random forest, only a random subset of the features is taken into consideration by the algorithm for splitting a node. You can even make trees more random by additionally using random thresholds for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

Feature and Advantages of Random Forest :

1. It is one of the most accurate learning algorithms available. For many data sets, it produces a **highly accurate classifier**.
2. It runs efficiently on large databases.
3. It can **handle thousands of input variables** without variable deletion.
4. It gives estimates of what variables that are important in the classification.
5. It generates an internal **unbiased estimate of the generalization error** as the forest building progresses.
6. It has an **effective method for estimating missing data** and maintains accuracy when a large proportion of the data are missing.

Disadvantages of Random Forest :

1. Random forests have been observed to **overfit for some datasets** with noisy classification/regression tasks.
2. For data including categorical variables with different number of levels, **random forests are biased in favor of those attributes with more levels**. Therefore, the variable importance scores from random forest are not reliable for this type of data.

Data Collection and Preprocessing

Data collection is the process of gathering and measuring information from countless different sources. In order to use the data we collect to develop practical artificial intelligence (AI) and machine learning solutions, it must be collected and stored in a way that makes sense for the business problem at hand.

Data Preprocessing

Steps involved in data preprocessing :

1. Importing the required Libraries
2. Importing the data set
3. Handling the Missing Data.
4. Encoding Categorical Data.
5. Splitting the data set into a test set and training set.
6. Feature Scaling

Step 1: Importing the required Libraries

Every time we make a new model, we will import Numpy and Pandas. Numpy is a Library which contains Mathematical functions and is used for scientific computing while Pandas is used to import and manage the data sets.

Step 2: Importing the Dataset

Data sets are available in .csv format. A CSV file stores tabular data in plain text. Each line of the file is a data record. We use the read_csv method of the pandas library to read a local CSV file as a **dataframe**.

Step 3: Handling the Missing Data

The data we get is rarely homogenous. Sometimes data can be missing and it needs to be handled so that it does not reduce the performance of our machine learning model.
To do this we need to replace the missing data by the Mean or Median of the entire column. For this we will be using the sklearn.preprocessing Library which contains a class called Imputer which will help us in taking care of our missing data.

1. **missing values** : It is the placeholder for the missing values. All occurrences of missing values will be imputed. We can give it an integer or "NaN" for it to find missing values.
2. **strategy**: It is the imputation strategy — If "mean", then replace missing values using the mean along the axis (Column). Other strategies include "median" and "most frequent".
3. **axis**: It can be assigned 0 or 1, 0 to impute along columns and 1 to impute along rows.

Step 4: Encoding categorical data

Any variable that is not quantitative is categorical. Examples include Hair color, gender, field of study, college attended, political affiliation, status of disease infection.

But why encoding ?

We cannot use values like "Male" and "Female" in mathematical equations of the model so we need to encode these variables into numbers.
To do this we import "LabelEncoder" class from "sklearn.preprocessing" library and create an object labelencoder_X of the LabelEncoder class. After that we use the fit_transform method on the categorical features.

After Encoding it is necessary to distinguish between the variables in the same column, for this we will use OneHotEncoder class from sklearn.preprocessing library.

Step 5: Splitting the Data set into Training set and Test Set

Now we divide our data into two sets, one for training our model called the **training set** and the other for testing the performance of our model called the **test set**. The split is generally 80/20. To do this we import the "train_test_split" method of "sklearn.model_selection" library.
Now to build our training and test sets, we will create 4 sets

1. **X_train** (training part of the matrix of features),
2. **X_test** (test part of the matrix of features),
3. **Y_train** (training part of the dependent variables associated with the X train sets, and therefore also the same indices) ,
4. **Y_test** (test part of the dependent variables associated with the X test sets, and therefore also the same indices).

We will assign to them the test_train_split, which takes the parameters — arrays (X and Y), test_size (Specifies the ratio in which to split the data set).

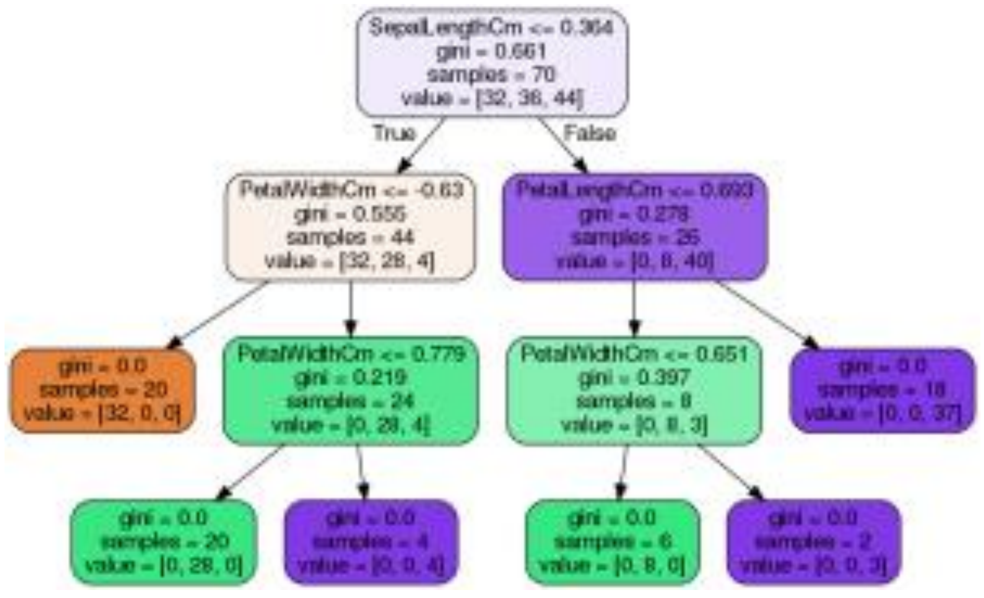
Step 6: Feature Scaling

Most of the machine learning algorithms use the **Euclidean distance** between two data points in their computations . Because of this, **high magnitudes features will weigh more** in the distance calculations **than features with low magnitudes**. To avoid this Feature standardization or Z-score normalization is used. This is done by using the "StandardScaler" class of "sklearn.preprocessing".

Modelling and Visualizing Tree

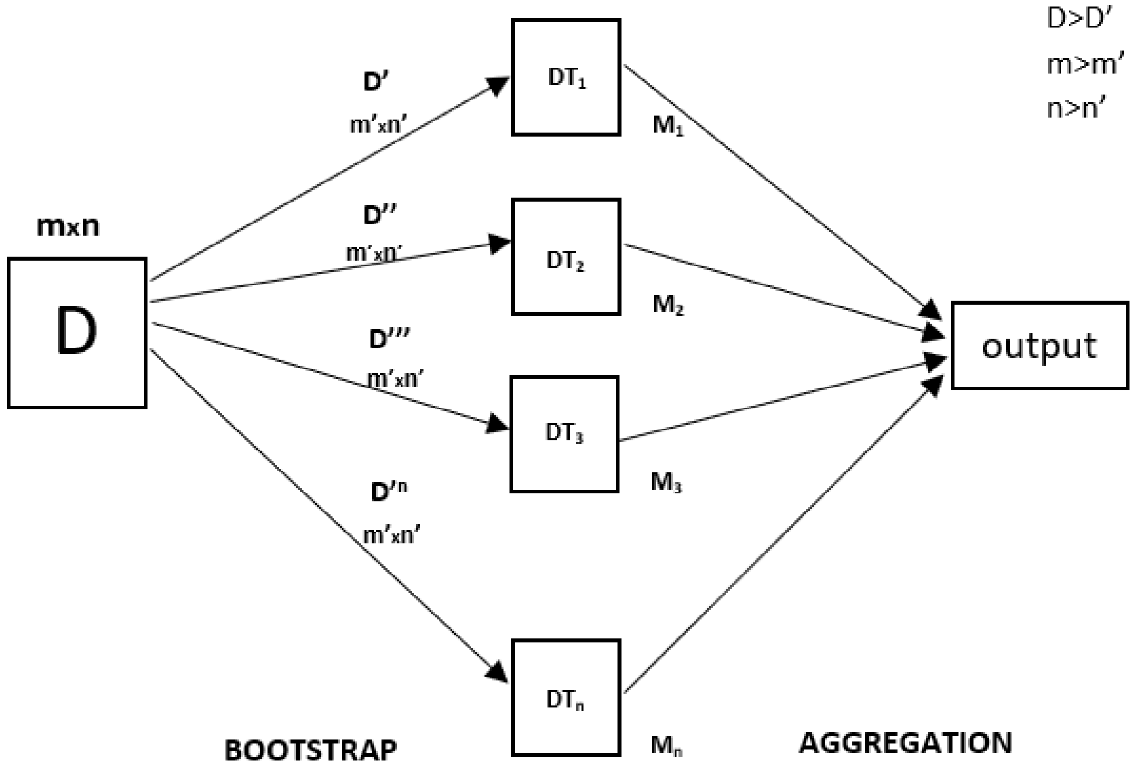
Visualizing the model

Now that we have our decision tree model and let's visualize it by utilizing the 'plot_tree' function provided by the scikit-learn package in python. Follow the code to produce a beautiful tree diagram out of your decision tree model in python.



Random Forest Regressor:

Every decision tree has high variance, but when we combine all of them together in parallel then the resultant variance is low as each decision tree gets perfectly trained on that particular sample data and hence the output doesn't depend on one decision tree but multiple decision trees. In the case of a classification problem, the final output is taken by using the majority voting classifier. In the case of a regression problem, the final output is the mean of all the outputs. This part is Aggregation.



A Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called Bootstrap and Aggregation, commonly known as bagging. The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.

Random Forest has multiple decision trees as base learning models. We randomly perform row sampling and feature sampling from the dataset forming sample datasets for every model. This part is called Bootstrap.

We need to approach the Random Forest regression technique like any other machine learning technique

- Design a specific question or data and get the source to determine the required data.
- Make sure the data is in an accessible format else convert it to the required format.
- Specify all noticeable anomalies and missing data points that may be required to achieve the required data.
- Create a machine learning model
- Set the baseline model that you want to achieve
- Train the data machine learning model.
- Provide an insight into the model with test data
- Now compare the performance metrics of both the test data and the predicted data from the model.
- If it doesn't satisfy your expectations, you can try improving your model accordingly or dating your data or use another data modeling technique.
- At this stage you interpret the data you have gained and report accordingly.

Understanding Voting Classifier:

Voting Ensembles

A voting ensemble (or a “*majority voting ensemble*“) is an ensemble machine learning model that combines the predictions from multiple other models.

It is a technique that may be used to improve model performance, ideally achieving better performance than any single model used in the ensemble.

A voting ensemble works by combining the predictions from multiple models. It can be used for classification or regression. In the case of regression, this involves calculating the average of the predictions from the models. In the case of classification, the predictions for each label are summed and the label with the majority vote is predicted.

- Regression Voting Ensemble: Predictions are the average of contributing models.
- Classification Voting Ensemble: Predictions are the majority vote of contributing models.

There are two approaches to the majority vote prediction for classification; they are hard voting and soft voting.

Hard voting involves summing the predictions for each class label and predicting the class label with the most votes. Soft voting involves summing the predicted probabilities (or probability-like scores) for each class label and predicting the class label with the largest probability.

- Hard Voting. Predict the class with the largest sum of votes from models
- Soft Voting. Predict the class with the largest summed probability from models.

A voting ensemble may be considered a meta-model, a model of models.

As a meta-model, it could be used with any collection of existing trained machine learning models and the existing models do not need to be aware that they are being used in the ensemble. This means you could explore using a voting ensemble on any set or subset of fit models for your predictive modeling task.

A voting ensemble is appropriate when you have two or more models that perform well on a predictive modeling task. The models used in the ensemble must mostly agree with their predictions.

Use voting ensembles when:

- All models in the ensemble have generally the same good performance.
- All models in the ensemble mostly already agree.

Hard voting is appropriate when the models used in the voting ensemble predict crisp class labels. Soft voting is appropriate when the models used in the voting ensemble predict the probability of class membership. Soft voting can be used for models that do not natively predict a class membership probability, although may require calibration of their probability-like scores prior to being used in the ensemble (e.g. support vector machine, k-nearest neighbors, and decision trees).

- Hard voting is for models that predict class labels.
- Soft voting is for models that predict class membership probabilities.

The voting ensemble is not guaranteed to provide better performance than any single model used in the ensemble. If any given model used in the ensemble performs better than the voting ensemble, that model should probably be used instead of the voting ensemble.

This is not always the case. A voting ensemble can offer lower variance in the predictions made over individual models. This can be seen in a lower variance in prediction error for regression tasks. This can also be seen in a lower variance in accuracy for classification tasks. This lower variance may result in a lower mean performance of the ensemble, which might be desirable given the higher stability or confidence of the model.

Use a voting ensemble if:

- It results in better performance than any model used in the ensemble.
- It results in a lower variance than any model used in the ensemble.

A voting ensemble is particularly useful for machine learning models that use a stochastic learning algorithm and result in a different final model each time it is trained on the same dataset. One example is neural networks that are fit using stochastic gradient descent.

Voting ensembles are most effective when:

- Combining multiple fits of a model trained using stochastic learning algorithms.
- Combining multiple fits of a model with different hyperparameters.

A limitation of the voting ensemble is that it treats all models the same, meaning all models contribute equally to the prediction. This is a problem if some models are good in some situations and poor in others.

An extension to the voting ensemble to address this problem is to use a weighted average or weighted voting of the contributing models. This is sometimes called blending. A further extension is to use a machine learning model to learn when and how much to trust each model when making predictions. This is referred to as stacked generalization, or stacking for short.

Extensions to voting ensembles:

- Weighted Average Ensemble (blending).
- Stacked Generalization (stacking)

Simple example for Voting Classifier :-

As we can see, the ensemble takes advantage of the different algorithms and yields better performance than any single one. Make sure to include a diverse classifier so that models which fall prey to similar types of errors do not aggregate the errors. Similarly we can repeat it with soft voting.
A voting classifier can be a good choice whenever a single strategy is not able to reach the desired accuracy threshold. In short voting classifier instead allows the mixing of different classifiers adopting a majority vote to decide which class must be considered as the winning one during a prediction.

Understanding Hard Voting Classifier:

Hard Voting: Hard voting is the simplest case of majority voting. In this case, the class that received the highest number of votes Nc(yt) will be chosen. Here we predict the class label y^ via majority voting of each classifier.

Implementation of Hard Voting Classifier
The k-fold cross-validation procedure is a standard method for estimating the performance of a machine learning algorithm or configuration on a dataset.

A single run of the k-fold cross-validation procedure may result in a noisy estimate of model performance. Different splits of the data may result in very different results.

Repeated k-fold cross-validation provides a way to improve the estimated performance of a machine learning model. This involves simply repeating the cross-validation procedure multiple times and reporting the mean result across all folds from all runs. This mean result is expected to be a more accurate estimate of the true unknown underlying mean performance of the model on the dataset, as calculated using the standard error.

In this you will discover repeated k-fold cross-validation for model evaluation.

After completing you will know:

- The mean performance reported from a single run of k-fold cross-validation may be noisy.
- Repeated k-fold cross-validation provides a way to reduce the error in the estimate of mean model performance.
- How to evaluate machine learning models using repeated k-fold cross-validation in Python.

A **box** and **whisker plot**—also called a **box plot**—displays the five-number summary of a set of data. The five-number summary is the minimum, first quartile, median, third quartile, and maximum. In a **box plot**, we draw a **box** from the first quartile to the third quartile. A vertical line goes through the **box** at the median.
Implementation of Soft Voting Classifier

We can demonstrate soft voting with the support vector machine (SVM) algorithm.
The SVM algorithm does not natively predict probabilities, although it can be configured to predict probability-like scores by setting the "*probability*" argument to "*True*" in the SVC class.
We can fit five different versions of the SVM algorithm with a polynomial kernel, each with a different polynomial degree, set via the "*degree*" argument. We will use degrees 1-5.

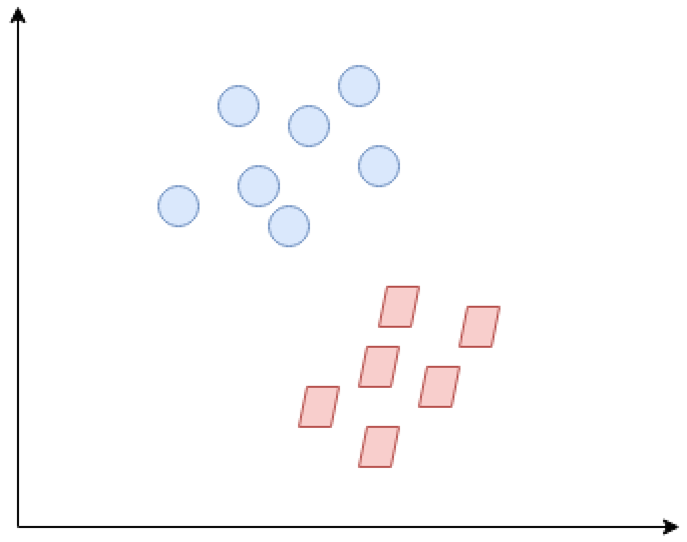
Our expectation is that by combining the predicted class membership probability scores predicted by each different SVM model that the soft voting ensemble will achieve a better predictive performance than any standalone model used in the ensemble, on average.

First, we can create a function named *get_voting()* that creates the SVM models and combines them into a soft voting ensemble.

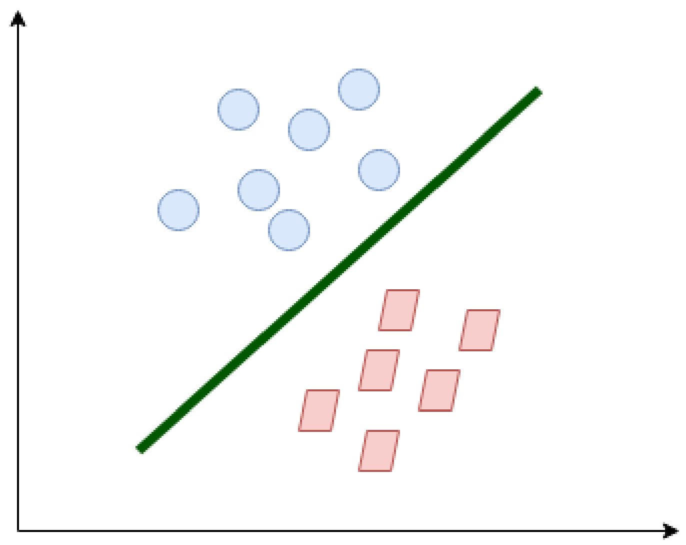
Lets understand about Support Vector Machine in detail as of now :

What is a Support Vector Machine (SVM)?

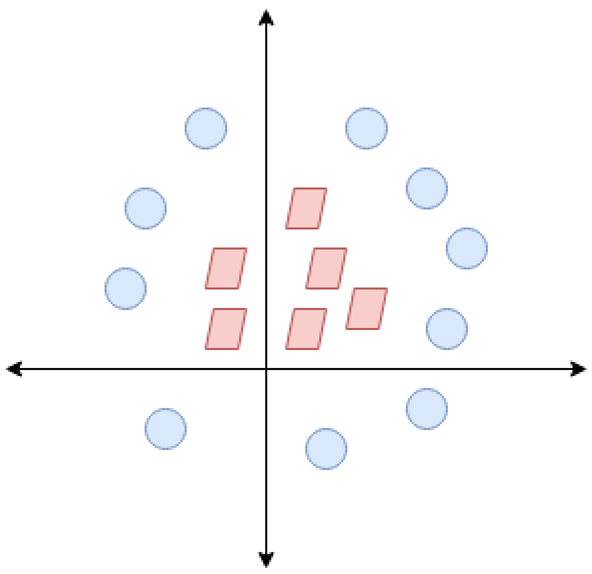
So what exactly is Support Vector Machine (SVM)? We'll start by understanding SVM in simple terms. Let's say we have a plot of two label classes as shown in the figure below:



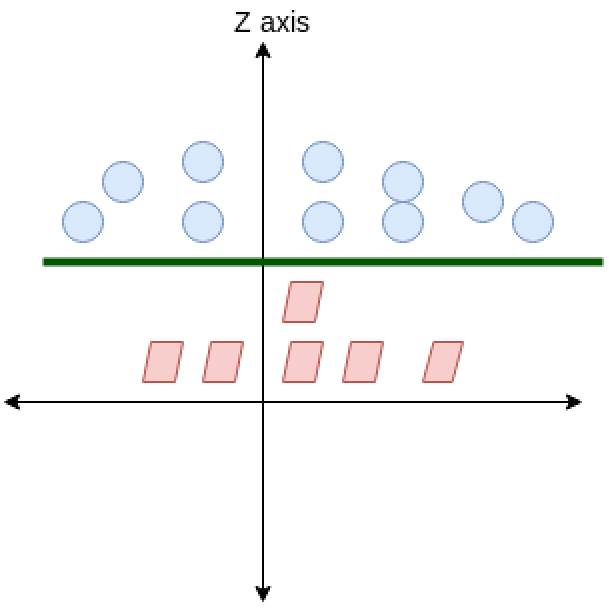
Can you decide what the separating line will be? You might have come up with this:



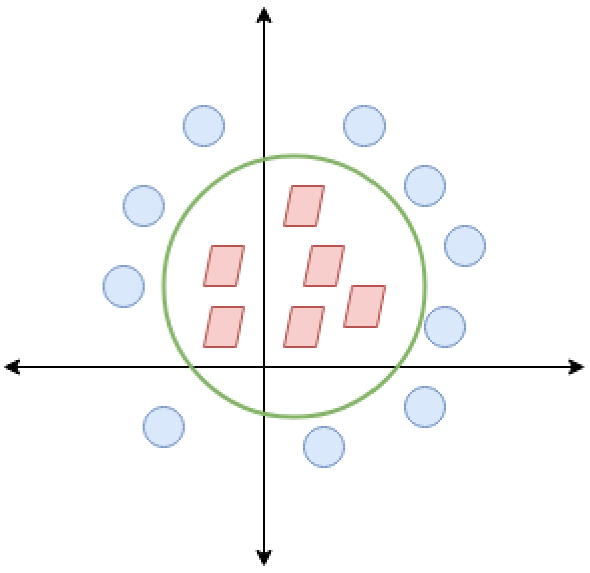
The line fairly separates the classes. This is what SVM essentially does – **simple class separation.** Now, what is the data was like this:



Here, we don't have a simple line separating these two classes. So we'll extend our dimension and introduce a new dimension along the z-axis. We can now separate these two classes:



When we transform this line back to the original plane, it maps to the circular boundary as I've shown here:

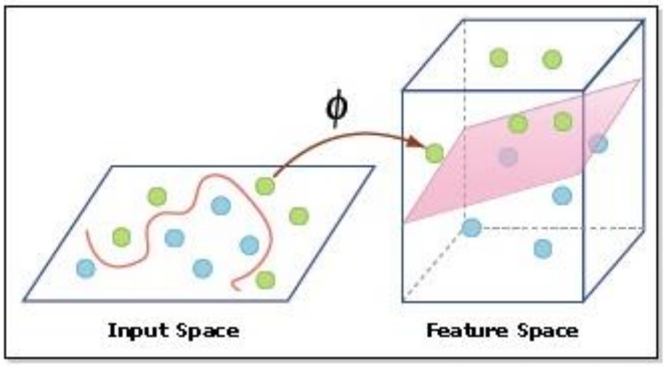


This is exactly what SVM does! It tries to find a line/hyperplane (in multidimensional space) that separates these two classes. Then it classifies the new point depending on whether it lies on the positive or negative side of the hyperplane depending on the classes to predict.

Hyperparameters of the Support Vector Machine (SVM) Algorithm

There are a few important parameters of SVM that you should be aware of before proceeding further:

- **Kernel:** A kernel helps us find a hyperplane in the higher dimensional space without increasing the computational cost. Usually, the computational cost will increase if the dimension of the data increases. This increase in dimension is required when we are unable to find a separating hyperplane in a given dimension and are required to move in a higher dimension:



- **Hyperplane:** This is basically a separating line between two data classes in SVM. But in Support Vector Regression, this is the line that will be used to predict the continuous output
- **Decision Boundary**: A decision boundary can be thought of as a demarcation line (for simplification) on one side of which lie positive examples and on the other side lie the negative examples. On this very line, the examples may be classified as either positive or negative. This same concept of SVM will be applied in Support Vector Regression as well

We can then create a list of models to evaluate, including each standalone version of the SVM model configurations and the soft voting ensemble.

This will help us directly compare each standalone configuration of the SVM model with the ensemble in terms of the distribution of classification accuracy scores. The *get_models()* function below creates the list of models for us to evaluate.We can evaluate and report model performance using repeated k-fold cross-validation as we did in the previous videos.
A box-and-whisker plot is then created comparing the distribution accuracy scores for each model, allowing us to clearly see that soft voting ensemble performing better than all standalone models on average.

If we choose a soft voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model.

First, the soft voting ensemble is fit on all available data, then the *predict()* function can be called to make predictions on new data.If we choose a soft voting ensemble as our final model, we can fit and use it to make predictions on new data just like any other model.

Understanding Bagging and Pasting in Scikit-Learn

Bagging means bootstrap+aggregating and it is an ensemble method in which we first bootstrap our data and for each bootstrap sample we train one model. After that, we aggregate them with equal weights. When it's not used as a replacement, the method is called pasting.

Bagging Classifier Implementation

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it.

Each base classifier is trained in parallel with a training set which is generated by randomly drawing, with replacement, N examples(or data) from the original training dataset – *where N is the size of the original training set*. Training set for each of the base classifiers is independent of each other. Many of the original data may be repeated in the resulting training set while others may be left out.

Bagging reduces overfitting (variance) by averaging or voting, however, this leads to an increase in bias, which is compensated by the reduction in variance though.

How Bagging work on a training dataset ?

Since Bagging resamples the original training dataset with replacement, some instances(or data) may be present multiple times while others are left out.So we will understand it clearly by implementing it on a dataset.

Pasting Classifier Implementation

In Machine Learning, one way to use the same training algorithm for more prediction models and to train them on different sets of the data is known as Bagging and Pasting.

Bagging and pasting are techniques that are used in order to create varied subsets of the training data. The subsets produced by these techniques are then used to train the predictors of an ensemble.

Bagging, short for bootstrap aggregating, creates a dataset by sampling the training set with replacement. Pasting creates a dataset by sampling the training set without replacement.

Ensemble models perform best when the predictors they are made up of are very different from one another. When predictors differ greatly from one another it follows that the instances in which they make errors will also be different. This leads to higher accuracy for the ensemble, as it is less likely that the majority of the predictors in the ensemble will make the same error.

In order to attain a varied set of predictors one method is to use different training algorithms for each of the predictors. Bagging provides an alternate method for achieving diversity among the predictors in an ensemble. Each of the predictors is created using the same training algorithm, but is trained on a random subset of the training data that has been sampled with replacement.

Bagging means to perform sampling with replacement and when the process of bagging is done without replacement then this is known as Pasting.Generally speaking, both bagging and pasting allow the training samples for sampling a lot of time across multiple prediction models, but only bagging can allow the training samples for sampling a lot of time on the same prediction model.

Both bagging and pasting are very simple to implement. For pasting each dataset is created by randomly sampling the chosen number of samples from the training set without replacement. For bagging each dataset is created by randomly sampling the chosen number of samples from the dataset with replacement. A separate predictor is then trained on each of the created datasets. These predictors are then used to form an ensemble.

We will work on **scikit-learn builtin dataset make_moons** as it generates isotropic Gaussian blobs for clustering where we understand the importance of bootstrap parameter.

Boosting Classifier

What is Boosting?

Definition: The term ‘Boosting’ refers to a family of algorithms which converts weak learners to strong learners.

Let’s understand this definition in detail by solving a problem of spam email identification:

How would you classify an email as SPAM or not? Like everyone else, our initial approach would be to identify ‘spam’ and ‘not spam’ emails using following criteria. If:

1. Email has only one image file (promotional image), It’s a SPAM
2. Email has only link(s), It’s a SPAM
3. Email body consist of sentence like “You won a prize money of $ xxxxxx”, It’s a SPAM
4. Email from our official domain “xxxxxxx.com” , Not a SPAM
5. Email from known source, Not a SPAM

Above, we’ve defined multiple rules to classify an email into ‘spam’ or ‘not spam’. But, do you think these rules individually are strong enough to successfully classify an email? No.

Individually, these rules are not powerful enough to classify an email into ‘spam’ or ‘not spam’. Therefore, these rules are called weak **learners**.

To convert weak learner to strong learner, we’ll combine the prediction of each weak learner using methods like:

- Using average/ weighted average
- Considering prediction has higher vote

For example: Above, we have defined 5 weak learners. Out of these 5, 3 are voted as ‘SPAM’ and 2 are voted as ‘Not a SPAM’. In this case, by default, we’ll consider an email as SPAM because we have a higher(3) vote for ‘SPAM’.

Understanding Boosting Classifier

Now we know that, boosting combines weak learner a.k.a. base learner to form a strong rule. An immediate question which should pop in your mind is, ‘*How boosting identify weak rules?*‘

To find weak rule, we apply base learning (ML) algorithms with a different distribution. Each time base learning algorithm is applied, it generates a new weak prediction rule. This is an iterative process. After many iterations, the boosting algorithm combines these weak rules into a single strong prediction rule.

Here's another question which might haunt you, 'How do we choose different distribution for each round?'
For choosing the right distribution, here are the following steps:

Step 1: The base learner takes all the distributions and assign equal weight or attention to each observation.

Step 2: If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Then, we apply the next base learning algorithm.

Step 3: Iterate Step 2 till the limit of base learning algorithm is reached or higher accuracy is achieved.

Finally, it combines the outputs from weak learner and creates a strong learner which eventually improves the prediction power of the model. Boosting pays higher focus on examples which are mis-classified or have higher errors by preceding weak rules.

Types of Boosting Algorithms

Underlying engine used for boosting algorithms can be anything. It can be a decision stamp, margin-maximizing classification algorithm etc. There are many boosting algorithms which use other types of engine such as:

1. AdaBoost (**Ada**ptive **Boost**ing)
2. Gradient Boosting

Working on Adaboost Classifier

AdaBoost is best used to boost the performance of decision trees on binary classification problems.

AdaBoost was originally called AdaBoost.M1 by the authors of the technique Freund and Schapire. More recently it may be referred to as discrete AdaBoost because it is used for classification rather than regression.

AdaBoost can be used to boost the performance of any machine learning algorithm. It is best used with weak learners. These are models that achieve accuracy just above random chance on a classification problem.

The most suited and therefore most common algorithm used with AdaBoost are decision trees with one level. Because these trees are so short and only contain one decision for classification, they are often called decision stumps.

Each instance in the training dataset is weighted. The initial weight is set to:

$$\text{weight}(xi) = 1/n$$

Where xi is the i'th training instance and n is the number of training instances.

How To Train One Model

A weak classifier (decision stump) is prepared on the training data using the weighted samples. Only binary (two-class) classification problems are supported, so each decision stump makes one decision on one input variable and outputs a +1.0 or -1.0 value for the first or second class value.

The misclassification rate is calculated for the trained model. Traditionally, this is calculated as:

error = (correct – N) / N

Where error is the misclassification rate, correct are the number of training instance predicted correctly by the model and N is the total number of training instances. For example, if the model predicted 78 of 100 training instances correctly the error or misclassification rate would be (78-100)/100 or 0.22.

This is modified to use the weighting of the training instances:

error = sum(w(i) * terror(i)) / sum(w)

Which is the weighted sum of the misclassification rate, where w is the weight for training instance i and terror is the prediction error for training instance i which is 1 if misclassified and 0 if correctly classified.

For example, if we had 3 training instances with the weights 0.01, 0.5 and 0.2. The predicted values were -1, -1 and -1, and the actual output variables in the instances were -1, 1 and -1, then the terrors would be 0, 1, and 0. The misclassification rate would be calculated as:

error = (0.01*0 + 0.5*1 + 0.2*0) / (0.01 + 0.5 + 0.2)

or

error = 0.704

A stage value is calculated for the trained model which provides a weighting for any predictions that the model makes. The stage value for a trained model is calculated as follows:

stage = ln((1-error) / error)

Where stage is the stage value used to weight predictions from the model, ln() is the natural logarithm and error is the misclassification error for the model. The effect of the stage weight is that more accurate models have more weight or contribution to the final prediction.

The training weights are updated giving more weight to incorrectly predicted instances, and less weight to correctly predicted instances.

For example, the weight of one training instance (w) is updated using:

w = w * exp(stage * terror)

Where w is the weight for a specific training instance, exp() is the numerical constant e or Euler's number raised to a power, stage is the misclassification rate for the weak classifier and terror is the error the weak classifier made predicting the output variable for the training instance, evaluated as:

terror = 0 if(y == p), otherwise 1

Where y is the output variable for the training instance and p is the prediction from the weak learner.

This has the effect of not changing the weight if the training instance was classified correctly and making the weight slightly larger if the weak learner misclassified the instance.

AdaBoost Ensemble

Weak models are added sequentially, trained using the weighted training data.

The process continues until a pre-set number of weak learners have been created (a user parameter) or no further improvement can be made on the training dataset.

Once completed, you are left with a pool of weak learners each with a stage value.

Making Predictions with AdaBoost

Predictions are made by calculating the weighted average of the weak classifiers.
For a new input instance, each weak learner calculates a predicted value as either +1.0 or -1.0.
The predicted values are weighted by each weak learners stage value. The prediction for the ensemble model is taken as a the sum of the weighted predictions. If the sum is positive, then the first class is predicted, if negative the second class is predicted.

For example, 5 weak classifiers may predict the values 1.0, 1.0, -1.0, 1.0, -1.0. From a majority vote, it looks like the model will predict a value of 1.0 or the first class. These same 5 weak classifiers may have the stage values 0.2, 0.5, 0.8, 0.2 and 0.9 respectively. Calculating the weighted sum of these predictions results in an output of -0.8, which would be an ensemble prediction of -1.0 or the second class.

Data Preparation for AdaBoost

This section lists some heuristics for best preparing your data for AdaBoost.

- **Quality Data**: Because the ensemble method continues to attempt to correct misclassifications in the training data, you need to be careful that the training data is of a high-quality.
- **Outliers**: Outliers will force the ensemble down the rabbit hole of working hard to correct for cases that are unrealistic. These could be removed from the training dataset.
- **Noisy Data**: Noisy data, specifically noise in the output variable can be problematic. If possible, attempt to isolate and clean these from your training dataset.

Understanding Gradient Boosting Classifier

Gradient boosting is a type of machine learning boosting. It relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error. How are the targets calculated? The target outcome for each case in the data depends on how much changing that case's prediction impacts the overall prediction error:

- If a small change in the prediction for a case causes a large drop in error, then the next target outcome of the case is a high value. Predictions from the new model that are close to its targets will reduce the error.
- If a small change in the prediction for a case causes no change in error, then the next target outcome of the case is zero. Changing this prediction does not decrease the error.

The name *gradient boosting* arises because target outcomes for each case are set based on the gradient of the error with respect to the prediction. Each new model takes a step in the direction that minimizes prediction error, in the space of possible predictions for each training case.

Implementation of Gradient Boosting :

"Boosting" in machine learning is a way of combining multiple simple models into a single composite model. This is also why boosting is known as an additive model, since simple models (also known as weak learners) are added one at a time, while keeping existing trees in the model unchanged. As we combine more and more simple models, the complete final model becomes a stronger predictor. The term "gradient" in "gradient boosting" comes from the fact that the algorithm uses gradient descent to minimize the loss.

When gradient boost is used to predict a continuous value – like age, weight, or cost – we're using gradient boost for regression. This is not the same as using linear regression. This is slightly different than the configuration used for classification, so we'll stick to regression in this article.

Decision trees are used as the weak learners in gradient boosting. Decision Tree solves the problem of machine learning by transforming the data into tree representation. Each internal node of the tree representation denotes an attribute and each leaf node denotes a class label. The loss function is generally the squared error (particularly for regression problems). The loss function needs to be differentiable.

Also like linear regression we have concepts of **residuals** in Gradient Boosting Regression as well. Gradient boosting Regression calculates the difference between the current prediction and the known correct target value.

This difference is called residual. After that Gradient boosting Regression trains a weak model that maps features to that residual. This residual predicted by a weak model is added to the existing model input and thus this process nudges the model towards the correct target. Repeating this step again and again improves the overall model prediction.

Also it should be noted that Gradient boosting regression is used to predict continuous values like house price , while Gradient Boosting Classification is used for predicting classes like whether a patient has a particular disease or not.

The high level steps that we follow to implement Gradient Boosting Regression is as below:

1. Select a weak learner
2. Use an additive model
3. Define a loss function
4. Minimize the loss function

Comparison of Gradient Boost with Ada Boost

Both Gradient boost and Ada boost work with decision trees however, Trees in Gradient Boost are larger than trees in Ada Boost.
Both Gradient boost and Ada boost scales decision trees however, Gradient boost scales all trees by same amount unlike Ada boost.

Advantages of Gradient Boosting

Better accuracy: Gradient Boosting Regression generally provides better accuracy. When we compare the accuracy of GBR with other regression techniques like Linear Regression, GBR is mostly winner all the time. This is why GBR is being used in most of the online hackathon and competitions.

Less pre-processing: As we know that data pre processing is one of the vital steps in machine learning workflow, and if we do not do it properly then it affects our model accuracy. However, Gradient Boosting Regression requires minimal data preprocessing, which helps us in implementing this model faster with lesser complexity. Though pre-processing is not mandatory here we should note that we can improve model performance by spending time in preprocessing the data.

Higher flexibility: Gradient Boosting Regression provides can be used with many hyper-parameter and loss functions. This makes the model highly flexible and it can be used to solve a wide variety of problems.

Missing data: Missing data is one of the issue while training a model. Gradient Boosting Regression handles the missing data on its own and does not require us to handle it explicitly. This is clearly a great win over other similar algorithms. In this algorithm the missing values are treated as containing information. Thus during tree building, splitting decisions for node are decided by minimizing the loss function and treating missing values as a separate category that can go either left or right.

Gradient Boosting parameters

Let us discuss few important parameters used in Gradient Boosting Regression. These are the parameters we may like to tune for getting the best output from our algorithm implementation.
Number of Estimators: It is denoted as n_estimators.
The default value of this parameter is 100.

Number of estimators is basically the number of boosting stages to be performed by the model. In other words number of estimators denotes the number of trees in the forest. More number of trees helps in learning the data better. On the other hand, more number of trees can result in higher training time. Hence we need to find the right and balanced value of n_estimators for optimal performance.

Maximum Depth: It is denoted as max_depth.
The default value of max_depth is 3 and it is an optional parameter.
The maximum depth is the depth of the decision tree estimator in the gradient boosting regressor. We need to find the optimum value of this hyperparameter for best performance. As an example the best value of this parameter may depend on the input variables.

Learning Rate: It is denoted as learning_rate.
The default value of learning_rate is 0.1 and it is an optional parameter.
The learning rate is a hyper-parameter in gradient boosting regressor algorithm that determines the step size at each iteration while moving toward a minimum of a loss function.

Criterion: It is denoted as criterion.

The default value of criterion is friedman_mse and it is an optional parameter.
criterion is used to measure the quality of a split for decision tree.
mse stands for mean squared error.

Loss: It is denoted as loss.
The default value of loss is ls and it is an optional parameter.
This parameter indicates loss function to be optimized. There are various loss functions like ls which stands for least squares regression. Least absolute deviation abbreviated as lad is another loss function. Huber a third loss function is a combination of least squares regression and least absolute deviation.

Subsample: It is denoted as subsample
.
The default value of subsample is 1.0 and it is an optional parameter.
Subsample is fraction of samples used for fitting the individual tree learners.If subsample is smaller than 1.0 this leads to a reduction of variance and an increase in bias.

Number of Iteration no change: It is denoted by n_iter_no_change.
The default value of subsample is None and it is an optional parameter.

This parameter is used to decide whether early stopping is used to terminate training when validation score is not improving with further iteration.

If this parameter is enabled, it will set aside validation_fraction size of the training data as validation and terminate training when validation score is not improving.

Getting the data

Before we start implementing the model, we need to get the data. I have uploaded a sample data here. You can download the data on your local if you want to try on your own machine.

Implementation of Gradient Boosting Classifier

We implement it by checking the performance metrics like classification report,confusion matrix along with learning rate.