Introduction to Boosting

Suppose in your town, there was an annual gathering of blind men.

They were taken to see a life-like replication of a dinosaur but since there were so many of them, this had to be done in groups (sequentially).

They were divided into six groups.

The first group of blind men was led to randomly selected spots near the dinosaur model. Each of their (partial) descriptions was tested out to see how well it matched the full description of the dinosaur. One of the blind men who happened to be near the horns gave the most accurate description and he was selected as the best description (first weak classifier) of the dinosaur. But since he described only the horns, some of his descriptions were not complete.

The incomplete sections were noted down and when the second group of blind men was led, they were gently steered to these incomplete parts. Of the second group, one touched the tail and when he described it, it matched the dinosaur better than the descriptions others gave and so this was selected as the second weak classifier.

For each of the remaining four groups of blind men, the best classifier was obtained in a similar way. In the end, the best descriptions from each of the six groups were combined (additively). While combining, since the horns and belly were larger than the other parts, those descriptions were given more weight by the challenge organizers. This combination of descriptions ended up describing the dinosaur model satisfactorily.

Each of the blind men had a partial description (a weak classifier) of the dinosaur and by combining these they were able to get the complete picture. In other words, a strong classifier was created from an ensemble of weak classifiers.

The above example explains the value of collaboration and demonstrates the basic principle of Boosting.

In the last module, we learned about the different methods of ensembling in machine learning. We touched a little about 'Boosting' in that.

Let's understand in detail what it is and why it has gone on to become one of the most used ML methods in recent times.

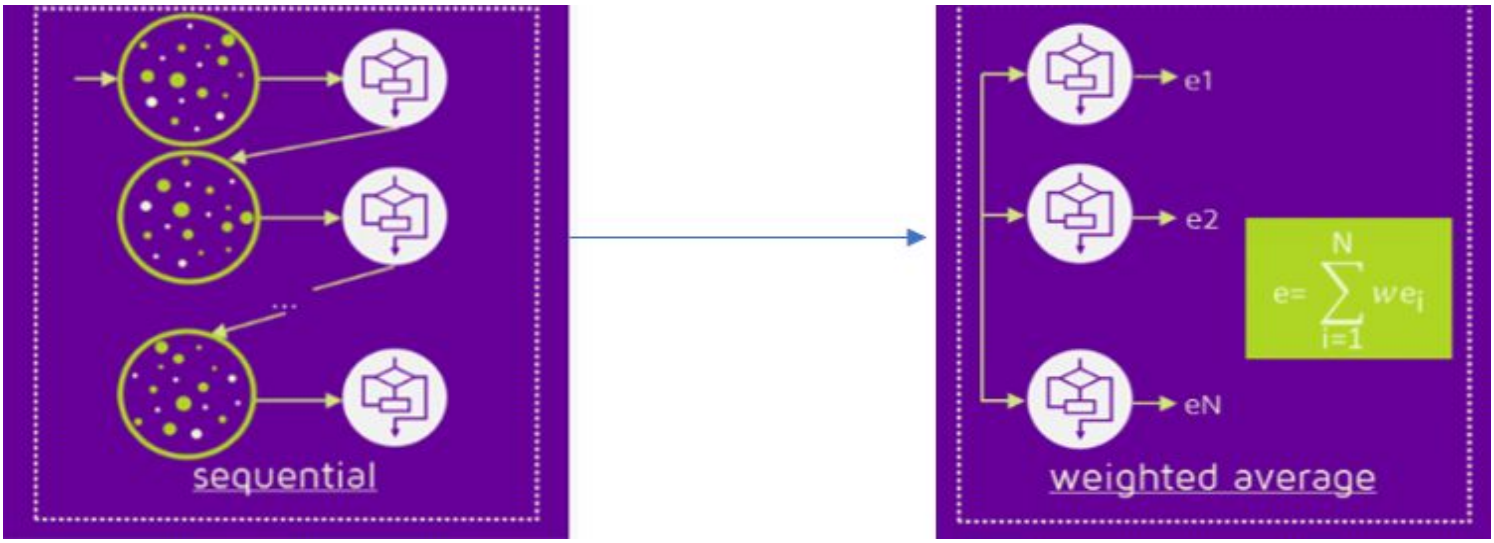
Definition

The idea of boosting came out of the idea of whether a weak learner can be sequentially modified to become better.

A weak learner is defined as a model whose performance is slightly better than random chance.

Boosting (aka hypothesis boosting) refers to any Ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods involves a sequential process, where each subsequent model attempts to correct the errors of the previous model. More importance is given to examples that were misclassified by earlier rounds and then the final prediction is produced by ensuring that the subsequent models avoid the same mistakes.



The idea is to use the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified.In essence, Boosting improves by reducing bias of every subsequent weak learner.

Types of boosting

The two most popular ones are :

- AdaBoost (Adaptive Boosting)
- Gradient Boosting

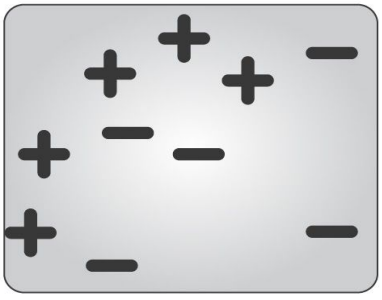
AdaBoost

AdaBoost is one of the first boosting algorithms to be adapted in solving practices.

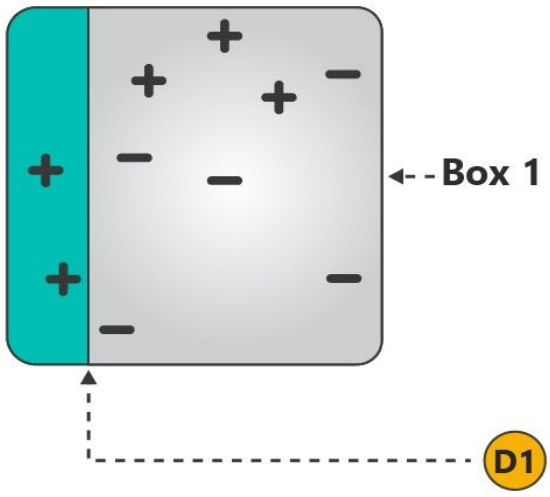
The weak learners in AdaBoost are decision trees with a single split, called decision stumps. AdaBoost works by putting more weight on difficult to classify instances and less on those already handled well.

Let's try to understand how Ada boosting works(intuition) by taking a simple example.

Consider a target class whose distribution when plotted in `2-D` looks similar to this:

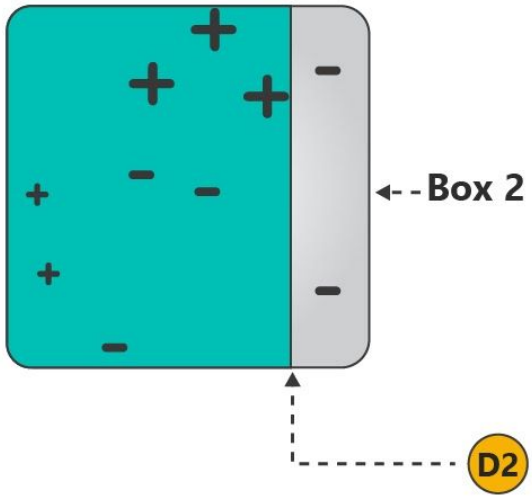


Step 1: We start with box 1, we employ a weak learner which creates a boundary D1, which predicts the dark region as the positive and light region as negative.



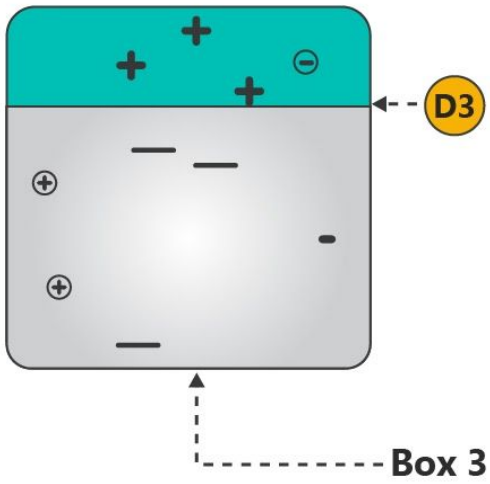
Hence, we have identified 2 observations on the left-hand side correctly positive, but we marked three positive observations on the right side incorrectly.

Step 2: We come to box 2. We employ another weak learner. But this time we pay more attention to classify the three positive observations (enlarged +) that we predicted incorrectly.



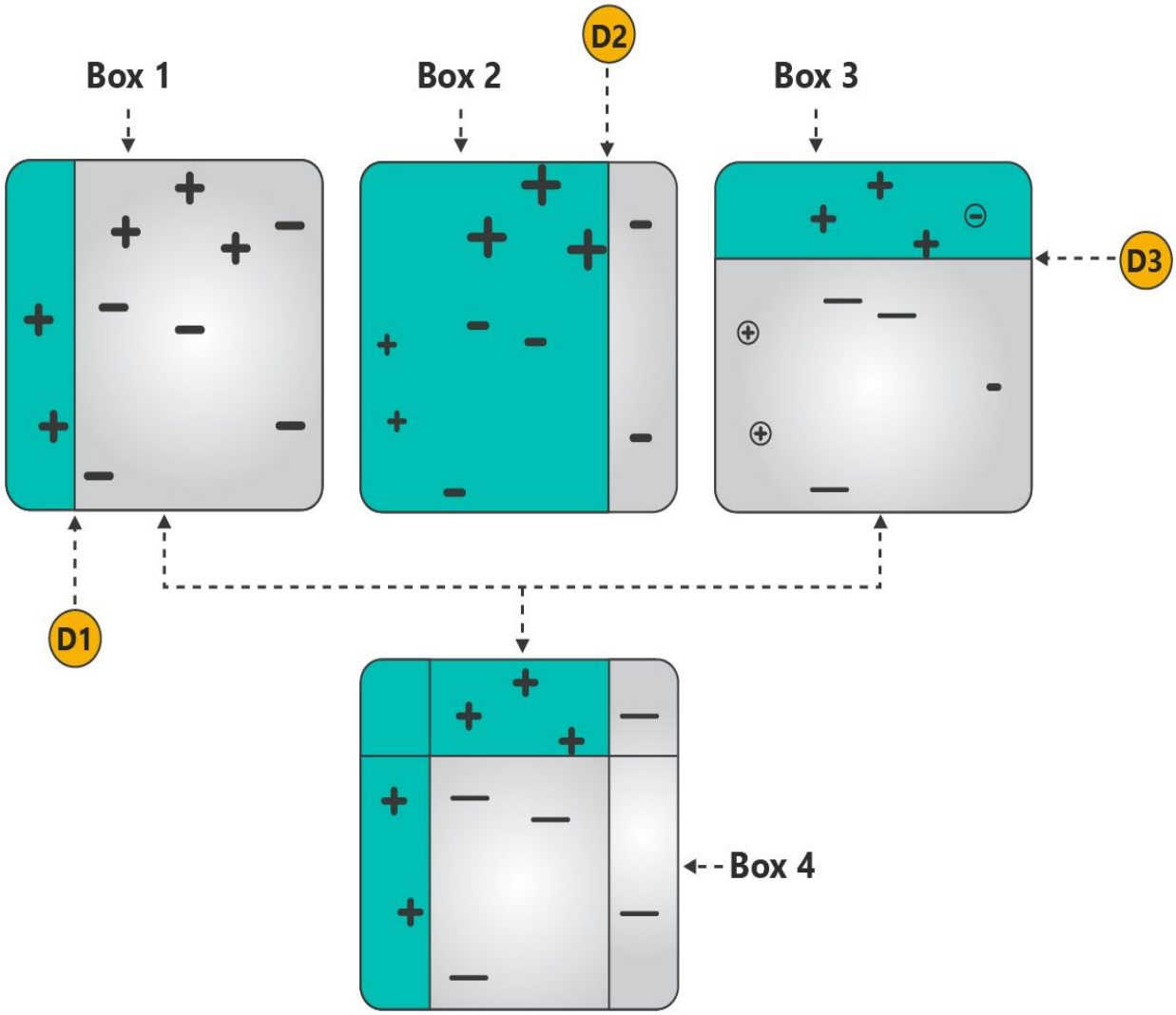
The learner comes up with the boundary D2, which correctly identifies previously wrongly predicted observations. However, this time it wrongly predicts 3 negative observations as positives (- in dark region)

Step 3: We move on to box 3. This time, the objective is to identify the 3 negative observations (enlarged --) correctly.



The weak learner comes up with boundary D3, which identifies the three "-" correctly as negatives.

Step 4: We combine all the boundaries to come with the boosting model which is a strong learner, made of weak rules.



Working

Since the intuition is clear, let's get to the mathematical part of the algorithm

Following are the steps:

1- Preparation of weak classifier

A weak classifier (decision stumps) is prepared on the training data using the weighted samples.

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 respectively).

2- Calculation of misclassification rate

The initial misclassification rate is calculated for the trained model as:

$$m_{rate} = (total - correct)/total$$

Where m_rate is the misclassification rate, correct is the number of correctly predicted training instances by the model and total is the total number of training instances.

For e.g., if the model correctly predicted 94 of 100 training instances misclassification rate would be (100-94)/100 or 0.06.

The rate is then modified to use the weight of the training instances:

$$m_{rate} = sum(w(i) * t_{error}(i))/sum(w)$$

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

Note: The initial weight before applying AdaBoost is set to w(i) = 1/n (Where n is the total number of instances)

For e.g., if we had 4 training instances with the weights 0.02, 0.5, 0.1, and 0.3. The predicted values were -1, -1, 1, and 1, and the actual output variables in the instances were -1, 1,-1, and 1, then the t_errors would be 0, 1,1, and 0.

The misclassification rate would be then calculated as:

$$m_{rate} = (0.02*0+0.5*1+0.1*1+0.3*0)/(0.02+0.5+0.1+0.3)$$

or

$$m_{rate} = 0.65$$

3- Calculation of 'stage value'

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:

$$stage = ln(\ (1-m_{rate})\ /\ m_{rate})$$

Where the stage is the stage value used to weight predictions from the model, ln() is the natural logarithm and m_rate 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.`

4- Updation of training weights

Finally, 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:

$$w = w * exp(stage * t_{error})$$

Where w is the weight for a specific training instance, exp() is the numerical constant e, the stage is the misclassification rate for the weak classifier and t_error is the error the weak classifier made predicting the output variable for the training instance

This has the effect of not changing the weight if the training instance was classified correctly(because t_error will be 0, making $w = w * \exp 0 = w * 1 = w$) and making the weight slightly larger if the weak learner misclassified the instance.

AdaBoost algorithms can be used for both classification and regression problems.

Adaboost Implementation

Let's continue solving the same problem statement we encountered in the Ensemble methods module.

To refresh,

Problem Statement

A bank has put out a marketing campaign and wants to know how the campaign is working. Given the features of the client and the marketing campaign, we have to predict whether the customer will subscribe to a term deposit or not.

About the Dataset

The dataset has 11162 rows with the following 17 features:

- `age (numeric)` - age of the bank customer
- `job(categorical)`- job of the bank customer
- `marital(categorical)`- marital status of the bank customer
- `education(categorical)`- Education status of the customer
- `default(categorical)` - Whether the customer has credit in default?
- `balance (numeric)` - average yearly balance in euros
- `housing (categorical)` - Whether the customer has a housing loan?

- `loan(categorical)` - Whether the customer has a personal loan?
- `contact(categorical)` - contact communication type
- `day(numeric)` - last contact date(of the month) of the year
- `month(categorical)` - last contact month of year
- `day(categorical)` - last contact day of the week (: 'mon','tue','wed','thu','fri')
- `duration (numeric)` - last contact duration, in seconds
- `campaign (numeric)` - number of contacts performed during this campaign and for this client
- `pdays (numeric)` - number of days that passed by after the client was last contacted from a previous campaign
- `previous (numeric)` - number of contacts performed before this campaign and for this client (numeric)

- `Target`: deposit - has the client subscribed a term deposit? (binary- 0: no, 1:yes)

For modeling purposes, the data has already been preprocessed.

Adaboost Sklearn Implementation

Let's now try to see it's python implementation using sklearn.

For this we will use a subset of our original dataset containing only 3000 datapoints.

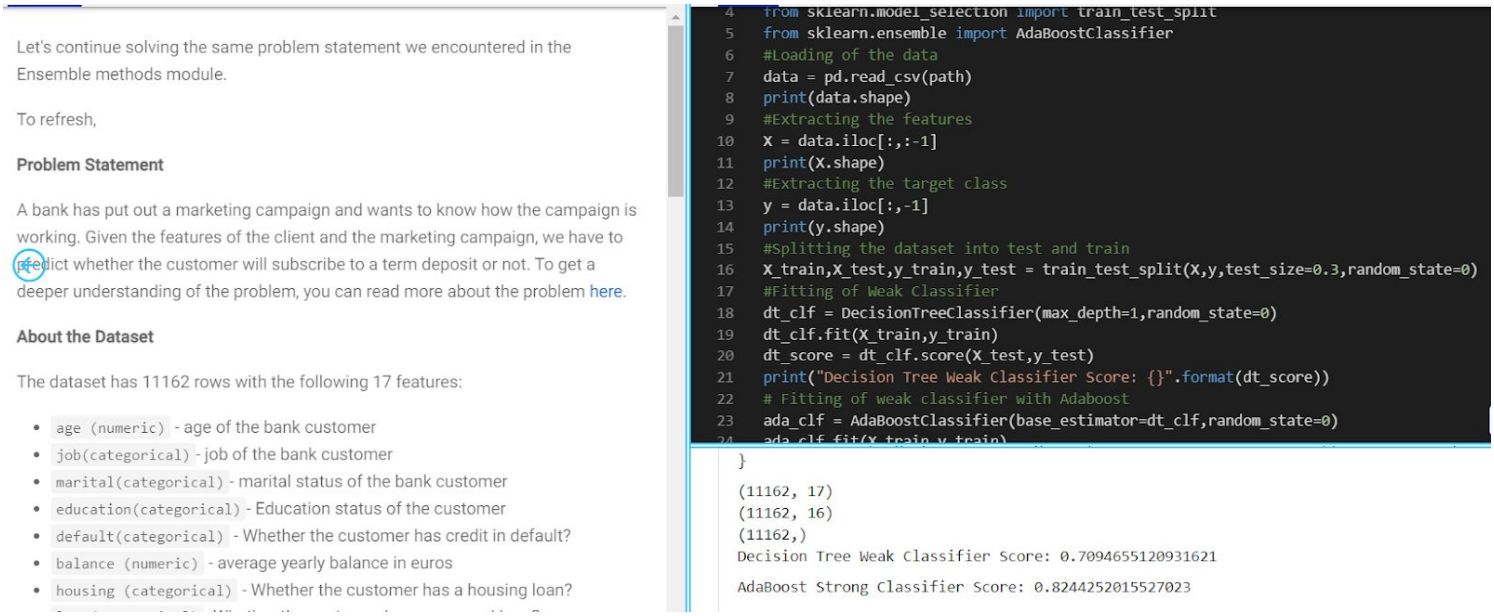
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

#Loading of data
data=pd.read_csv('../data/bank_data.csv')

#Sampling of data
sub_data=data.sample(frac=0.3, random_state=28)

#Extracting features
X=sub_data.drop(['deposit'],1)

#Extracting target class
y=sub_data['deposit'].copy()

#Splitting features and target class
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

#Weak Classifier

dt_clf=DecisionTreeClassifier(max_depth=1,random_state=0)
dt_clf.fit(X_train,y_train)
dt_score=dt_clf.score(X_test,y_test)
print("Score of Weak classifier:",dt_score)


# AdaBoost with the weak classifier

ada_clf = AdaBoostClassifier(base_estimator=dt_clf,random_state=0)
ada_clf.fit(X_train, y_train)
ada_score=ada_clf.score(X_test,y_test)
```

```
print("\nScore of AdaBoost:",ada_score)
```

Output:

```
Score of Weak classifier: 0.70

Score of AdaBoost: 0.79
```

Note: Though by default, AdaBoost takes DecisionTreeClassifer(max_depth=1) as its base model, you can still change it to any of the sklearn models.



Code:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier

#Loading of the data
data=pd.read_csv(path)

#Extracting the features
X=data.drop(['deposit'],1)

#Extracting the target class
y=data['deposit'].copy()

#Splitting the dataset into test and train
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.3, random_state=0)

#Fitting of Weak Classifier

dt_clf=DecisionTreeClassifier(max_depth=1,random_state=0)
```

```
dt_clf.fit(X_train,y_train)
dt_score=dt_clf.score(X_test,y_test)
print("Score of Weak classifier:",dt_score)


# Fitting of weak classifier with Adaboost
ada_clf = AdaBoostClassifier(base_estimator=dt_clf,random_state=0)
ada_clf.fit(X_train, y_train)
ada_score=ada_clf.score(X_test,y_test)
print("\nScore of AdaBoost:",ada_score)
```

About Gradient Boosting

One of the shortcomings of Adaboost is that it can only be possible if `exponential loss` function is used as the loss function.

What if we wanted to use an arbitrary loss function? Let's say MSE or f1 score?

This is where the Gradient Boosting Machine comes in.

GBM is a generalized version of AdaBoosting.

Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor.

However, instead of tweaking the instance weights at every iteration as AdaBoost does, this method tries to calculate the negative gradients("shortcomings" of the model) made by the previous predictor and fit the new predictor on it.

For eg: If the loss function is MSE, then the residuals(predicted value- true value) can be interpreted as negative gradients.

In such a case, the GBM algorithm can be given by the following steps.

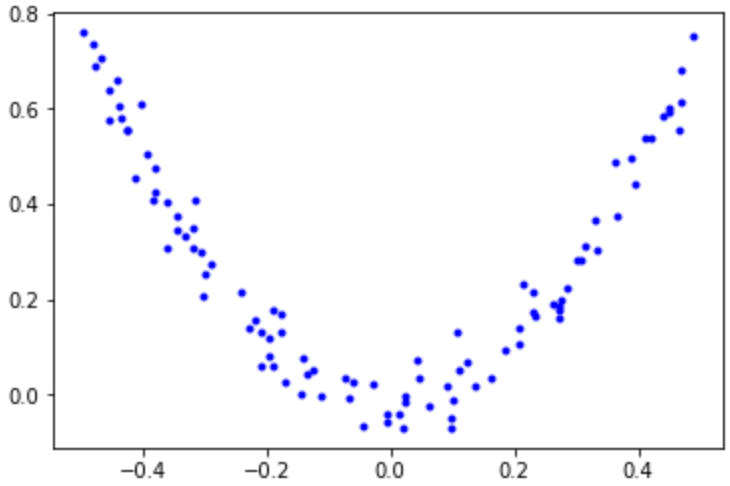
- Fit a model to the data, $F1(x) = y$
- Fit a model to the residuals, $h1(x) = y - F1(x)$
- Create a new model, $F2(x) = F1(x) + h1(x)$

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework so that any differentiable loss function can be used.

Let's understand this using an example.

Working of Gradient Boost

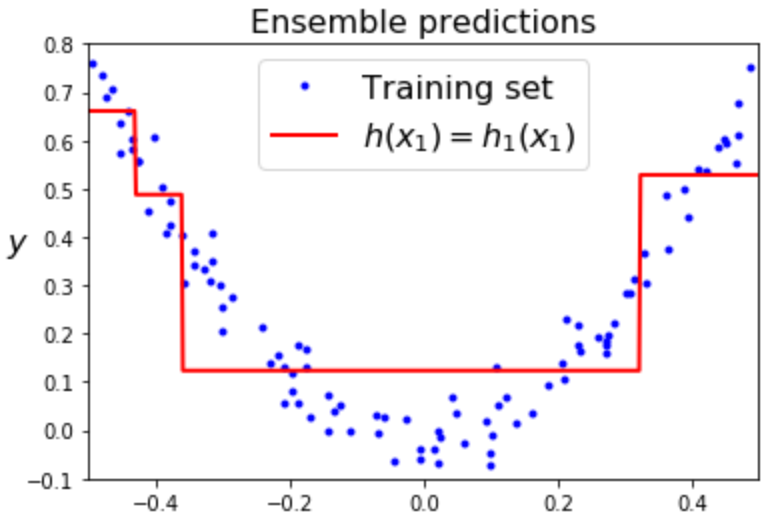
Let's take an example data roughly having the distribution of (y = 3x^2)



Now let's fit the first decision tree as usual on the data, find out the residuals for each point and save them in a variable called y2.

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
y2 = y - tree_reg1.predict(X)
```

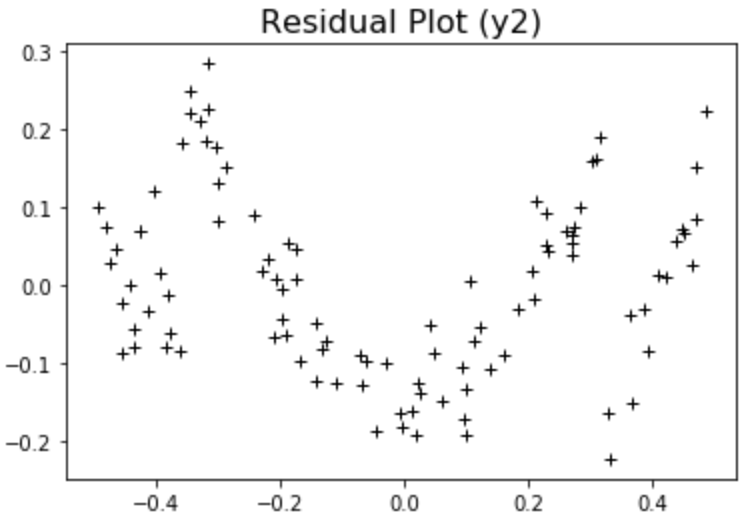
Plotting the decision boundary of the decision tree model(h1) looks like the following:



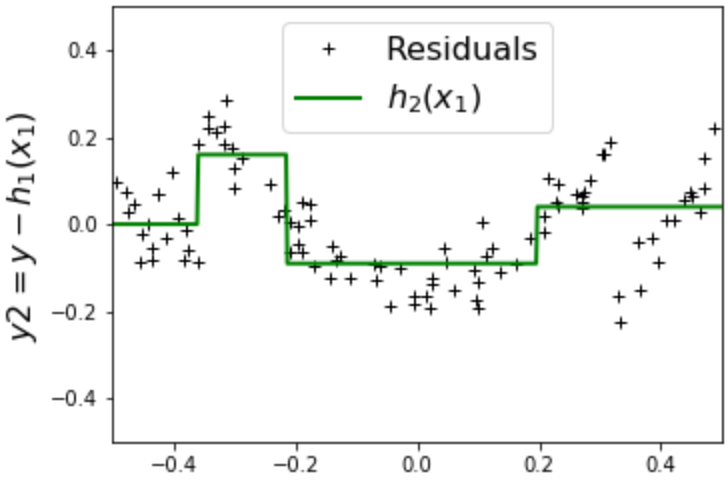
The red line is the decision tree boundary.

Now, as mentioned before, GBM fits models sequentially to correct the errors by the previously fit models.

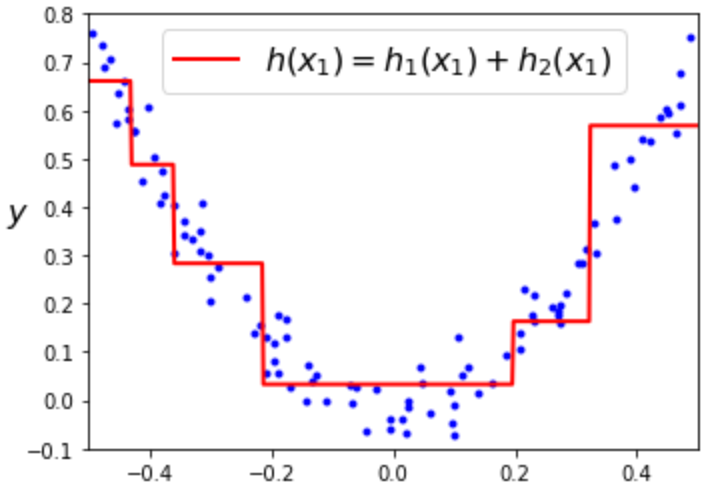
Following is the error residual from the previous model:



So, let's fit the decision tree on the residuals (y2) and call the model h2. The following graph shows how the second model fits the model on the residuals.



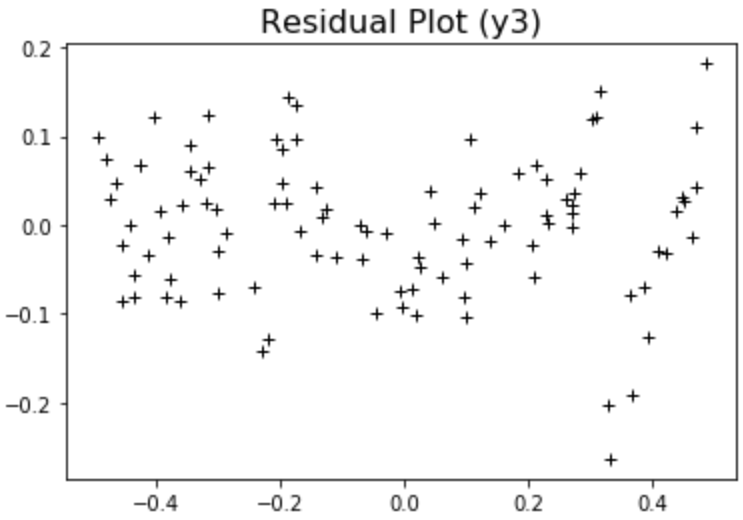
The following graph shows the collective predictions made by the 1st and 2nd models (h1 and h2) on the original data. As can we can notice, the fit is better.



Now, we will calculate the errors made by the second model, and save it in y3.

```
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)
y3 = y2 - tree_reg2.predict(X)
```

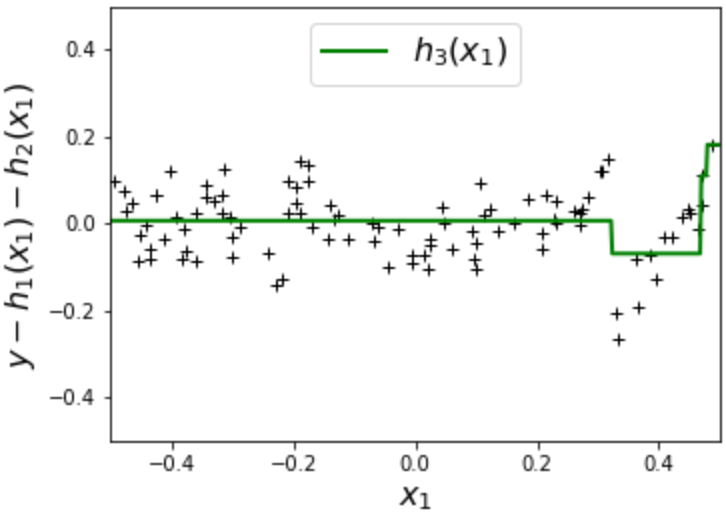
Now, let's repeat the same process for y3. Following is the graph of y3 residuals:



We will be fitting just 3 sequential models, so we will not calculate the errors by the third model.

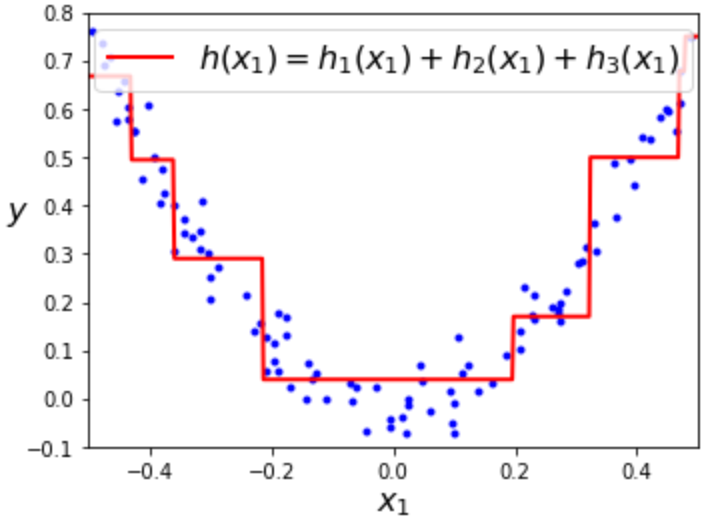
```
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)
```

The following graph shows how the third model fits the errors from the second model.

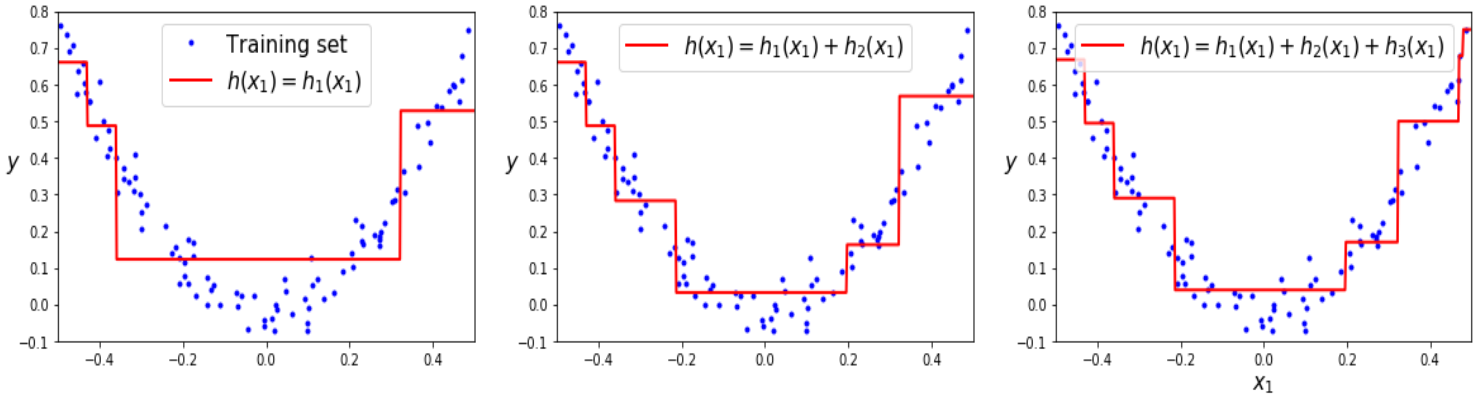


Also, notice how the errors are gradually converging to 0 with each iteration.

Following is the graph showing the collective predictions made by the 1st, 2nd, and 3rd model on the original data. As can we can notice, the fit is better than the previous iteration.



Following is the graph showing the three models side by side for better comparison:



With each iteration, the model is fitting closer to the data. It can also be understood that the increase in estimators will eventually lead to overfitting.

Iterating a previously mentioned fact: `Since the model is working on negative gradients, there is no limitation on the loss function and we are free to choose anyone`

Gradient Boosting Implementation

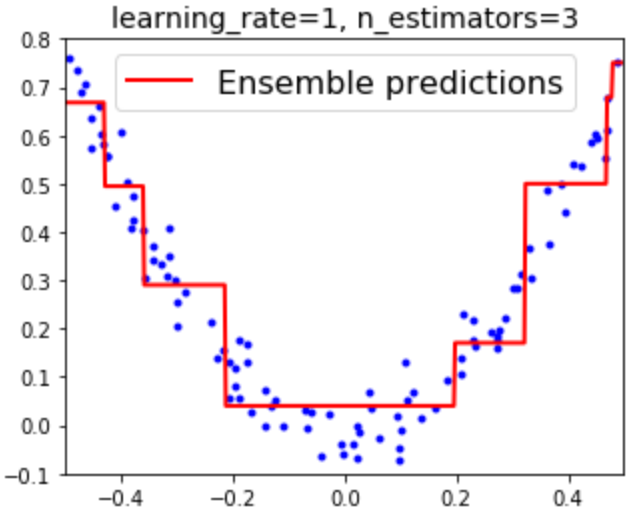
We will try implementing what we just did using Gradient Boosting. The data is the same data having the

(y = 3x^2) distribution.

Here is a sklearn implementation of GBM with n_estimators = 3

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3, learning_rate=1.0,
random_state=42)
gbrt.fit(X, y)
```

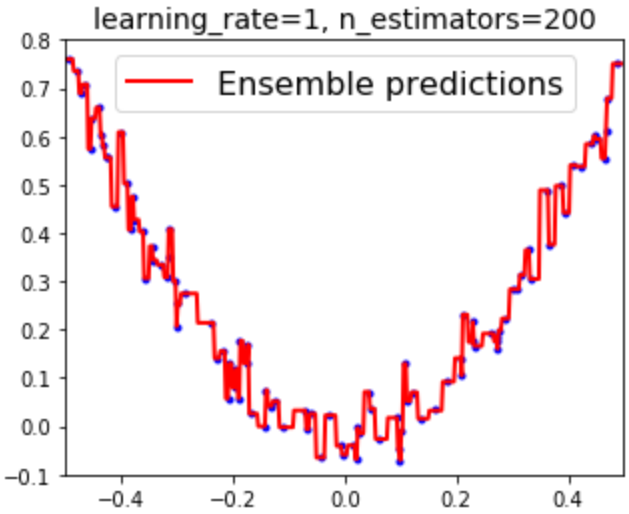
Following is the decison boundary:



Here is a sklearn implementation of GBM with n_estimators = 200

```
gbrt_slow = GradientBoostingRegressor(max_depth=2, n_estimators=200, learning_rate=0.1,
random_state=42)
gbrt_slow.fit(X, y)
```

Following is the decision boundary:



We can clearly see how both models vary in terms of decision boundaries.

Gradient Boosting

In this task, we will try to apply Gradient Boosting to our 'bank deposit' problem.

Instructions

- Initialise a Gradient Boost model with `GradientBoostingClassifier()` having `random_state=0` and save it to a variable called `'gb_clf'`.
- Fit the model on the training data `'X_train'` and `'y_train'` using the `'fit()'` method.
- Find out the accuracy score between `X_test` and `'y_test'` using the `'score()'` method and save it in a variable called `'gb_score'`

Things to ponder:

- Does Gradient Boost model have better performance than Ada Boost model? Why?

Skills Covered:

```
1  from sklearn.ensemble import GradientBoostingClassifier
2  gb_clf = GradientBoostingClassifier(random_state=0)
3  gb_clf.fit(X_train,y_train)
4  gb_score = gb_clf.score(X_test, y_test)
5  print("Gradient Boosting Score: {}".format(gb_score))
```

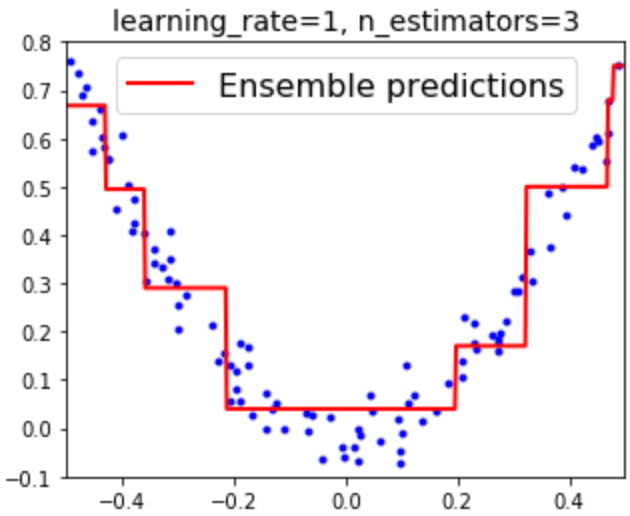
OUTPUT

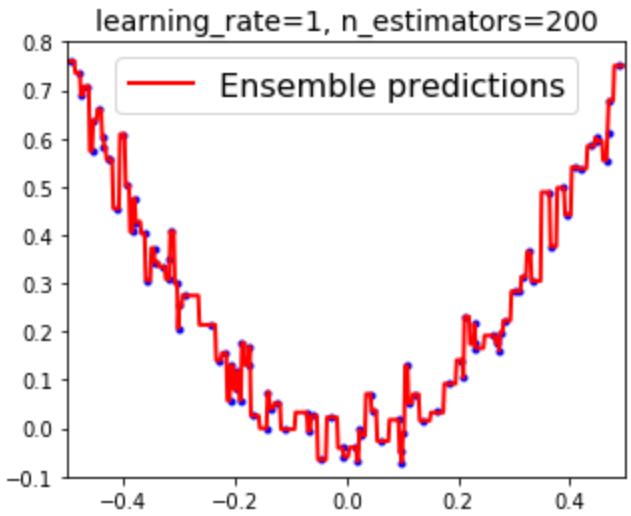
RESULT

```
Gradient Boosting Score: 0.8462227530606151
```

Gradient Boost Improvement

From our Gradient Boosting implementation, we got the following result





As we can see, the model with n_estimators = 200 led to much more overfitting than the one with 3 estimators. This brings us to the most crucial part of using GBM: Hyperparameter Tuning.

Let's try and understand a bit more about it.

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. Regularization methods penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

The overall parameters can be divided into 3 categories:

- Algorithm-Specific Parameters: These affect each individual tree in the model

- Training Parameters: These affect the boosting operation in the model
- Miscellaneous Parameters: Other parameters for overall functioning

Note: All the below mentioned points are explained considering the default model in GBM i.e. Decision Tree

Algorithm specific parameters()

---

min_samples_split :

- minimum number of samples required at a node to be considered for further splitting.
- Controls over-fitting.
- Higher values prevent a model from learning relations which might be highly specific to the particular sample selected for a tree.
- Too high values can lead to under-fitting hence, it should be tuned using cross validation.

min_samples_leaf :

- minimum samples required in a terminal node or leaf.
- Controls over-fitting.
- Generally lower values should be chosen for imbalanced class problems because the regions in which the minority class will be in majority will be very small.

max_depth :

- The maximum depth of a tree.
- Controls over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- Should be tuned using CV.

max_features:

- The number of features to consider while searching for a best split.
- These will be randomly selected.
- As a thumb-rule, square root of the total number of features works great but we should check upto 30-40% of the total number of features.
- Higher values may lead to over-fitting

Training parameters

---

learning_rate :

- Determines the impact of each tree on the final outcome. GBM works by starting with an initial estimate which is updated using the output of each tree.
- The learning parameter controls the magnitude of this change in the estimates.
- Lower values are generally preferred as they make the model robust to the specific characteristics of tree and thus allowing it to generalize well.

- Lower values would require higher number of trees to model all the relations and will be computationally expensive.

n_estimators :

- The number of sequential trees to be modeled
- More robust at higher number of trees but it can still overfit at a point.
- Hence, this should be tuned using CV for a particular learning rate.

subsample

- The fraction of observations to be selected for each tree. Selection is done by random.
- Values slightly less than 1 make the model robust by reducing the variance.
- Typical values ~0.8 generally work fine but can be fine-tuned further.

Misc Parameters

---

loss

- It refers to the loss function to be minimized in each split.

Limitations of Boosting

Just like any other ensemble method, Boosting also does suffer limitations

- Computationly expensive:

With so much of iteration of weak models happening in the implementation, this algorithm is expensive both resource and time wise.

This is also because Boosting can easily lead to overfitting of the data and therefore a good fit requires careful tuning of the learning rate and other parameters. Therefore the training process can require a lot of computation.

- Slow learning

With each iteration the complexity of the classification increases. Large datasets or iterations on multiple datasets won't give good results economically. That also makes it hard to implement in real time platform.

Also, when the featured space has thousands of features with sparse values,it is usually not a good choice for accuracy and computational cost reasons.

- Lack of interpretability

Owing to the gradual increase in complexity of the classification, boosting model can be very difficult for people to interpret.

Q: Till now you might have understood Boosting is a powerful algorithm with limitations. But why is it popular though?

A: There has been a recent implementation of GBM which has suddenly made it one the most popular ML algorithm out there - XGBoost

1. Which of the following is true about "max_depth" hyperparameter in Gradient Boosting?

- Increase the value of max_depth may overfit the data
- Lower is better parameter in case of same validation accuracy
- Lower is better parameter in case of same validation accuracy
- Increase the value of max_depth may overfit the data

Explanation:

Increase the depth from the certain value of depth may overfit the data and for 2 depth values validation accuracies are same we always prefer the small depth in final model building.

XGBoost

XGBoost (standing for eXtreme Gradient Boosting) has recently been dominating applied machine learning and Kaggle competitions for structured or tabular data.

Need for XGBoost:

Gradient boosting machines are generally very slow in implementation because of sequential model training. Hence, they are not very scalable

However, it all changed with XGboost Library. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

Performance comparision

- `Model Performance`

XGBoost is the dominant algorithm with respect to classification and regression problems on structured datasets .

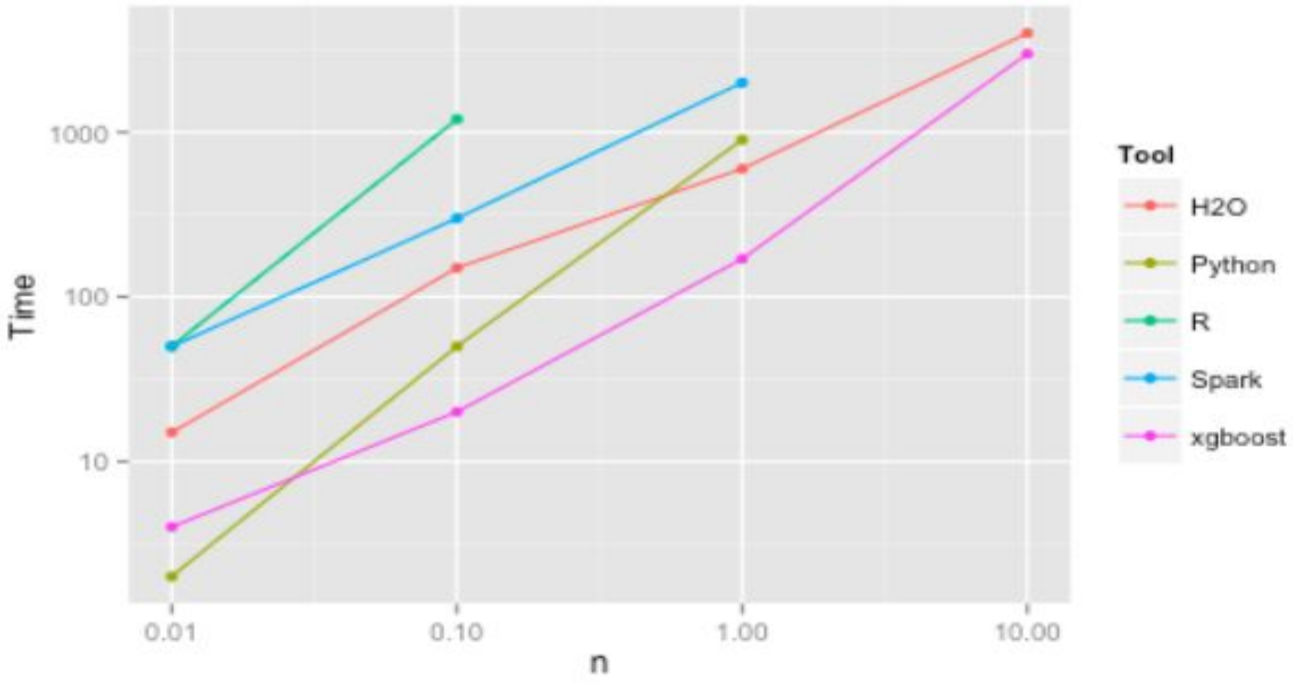
The evidence is that it is the go-to algorithm for Kaggle competition winners

For e.g: There is a list of first, second and third place competition winners that used XGBoost titled: XGBoost: Machine Learning Challenge Winning Solutions.

- `Speed`

XGBoost is fast when compared to other implementations of gradient boosting.

According to this speed benchmarking study, XGBoost was almost always faster than the other benchmarked implementations from R, Python Spark and H2O.



Let's now discuss some interesting features of XGBoost that make it powerful:

Handling sparse data: Missing values or data processing steps like one-hot encoding make data sparse. XGBoost incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data.

Regularization: XGBoost has an option to penalize complex models using both L1 and L2 regularization. Regularization helps prevent overfitting.

Weighted quantile sketch: XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data.

Out-of-core computing: This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

Block structure for parallel learning: For faster computing, XGBoost can make use of multiple cores on the CPU.

XGBoost Python Implementation

XGBoost has the same api as sklearn that we have learnt so far.

We will use the same sampled dataset having 3000 points we used for our `AdaBoost` implementation and compare

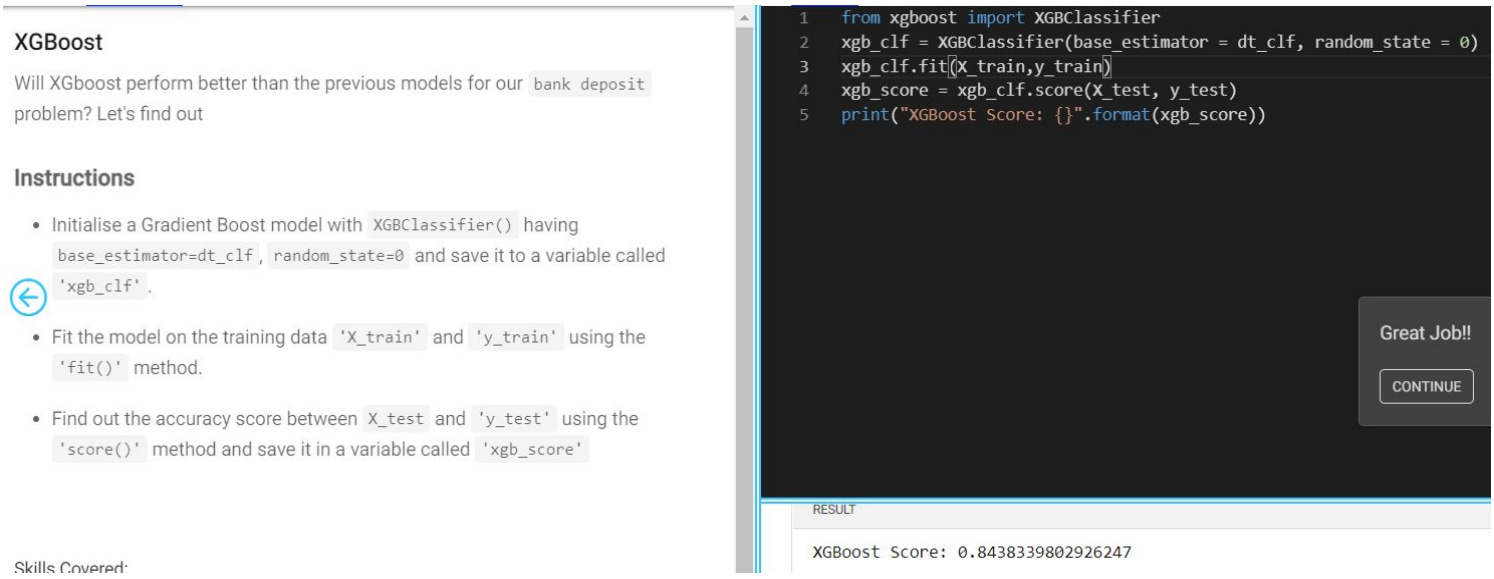
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier  #Not a sklearn implementation
```

```
#Loading of data
data=pd.read_csv('../data/bank_data.csv')

#Sampling of data
sub_data=data.sample(frac=0.3, random_state=28)

#Extracting features
X=sub_data.drop(['deposit'],1)

#Extracting target class
y=sub_data['deposit'].copy()

#Splitting features and target class
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

#Fitting of Weak Classifier

dt_clf=DecisionTreeClassifier(max_depth=1,random_state=0)
dt_clf.fit(X_train,y_train)
dt_score=dt_clf.score(X_test,y_test)
print("Score of Weak classifier:",dt_score)

# Fitting of weak classifier with Adaboost
ada_clf = AdaBoostClassifier(base_estimator=dt_clf,random_state=0)
ada_clf.fit(X_train, y_train)
ada_score=ada_clf.score(X_test,y_test)
print("\nScore of AdaBoost:",ada_score)

# Fitting of weak classifier with XGBoost
xgb_clf = XGBClassifier(base_estimator=dt_clf,random_state=0)
xgb_clf.fit(X_train, y_train)
xgb_score= xgb_clf.score(X_test,y_test)
print("\nScore of XGBoost:",xgb_score)
```

Output:

```
Score of Weak classifier: 0.70
Score of AdaBoost: 0.79
Score of XGBoost: 0.81
```

Note: Though obviously XGBoost has led to a better score in our example, but the true power of XGBoost can only be realised when dealing with large, complex datasets.



1. Which of the following statement is correct about XGBOOST parameters:

Sub Sampling / Row Sampling percentage should lie between 0 to 1
Number of trees / estimators can be 1

Sub Sampling / Row Sampling percentage should lie between 0 to 1
Number of trees / estimators can be 1

Explanation:
1 and 4 are wrong statements, whereas 2 and 3 are correct. Therefore D is true.

Telecom Churn Prediction with Boosting

Customer churn, also known as customer attrition, customer turnover, or customer defection, is the loss of clients or customers.Telephone service companies, Internet service providers, pay TV companies, insurance firms, and alarm monitoring services, often use customer attrition analysis and customer attrition rates as one of their key business metrics because the cost of retaining an existing customer is far less than acquiring a new one.

Predictive analytics use churn prediction models that predict customer churn by assessing their propensity of risk to churn. Since these models generate a small prioritized list of potential defectors, they are effective at focusing customer retention marketing programs on the subset of the customer base who are most vulnerable to churn.

For this project we will be exploring the dataset of a telecom company and try to predict the customer churn

Problem Statement

Using the method of Boosting, classify whether or not the customer will churn

About the Dataset

The snapshot of the dataset you will be working on

customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	OnlineBackup	DeviceProtection	TechSupport	StreamingTV	StreamingMovies	Contract	PaperlessBilling	PaymentMethod	MonthlyCharges	TotalCharges	Churn
7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	Yes	No	No	No	No	Month-to-month	Yes	Electronic check	29.85	29.85	No
5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	No	Yes	No	No	No	One year	No	Mailed check	56.95	1889.5	No
3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	Yes	No	No	No	No	Month-to-month	Yes	Mailed check	53.85	108.15	Yes
7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	No	Yes	Yes	No	No	One year	No	Bank transfer (automatic)	42.3	1840.75	No
9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	No	No	No	No	No	Month-to-month	Yes	Electronic check	70.7	151.65	Yes
9305-CDSKC	Female	0	No	No	8	Yes	Yes	Fiber optic	No	No	Yes	No	Yes	Yes	Month-to-month	Yes	Electronic check	99.65	820.5	Yes
1452-KIOVK	Male	0	No	Yes	22	Yes	Yes	Fiber optic	No	Yes	No	No	Yes	No	Month-to-month	Yes	Credit card (automatic)	89.1	1949.4	No
6713-OKOMC	Female	0	No	No	10	No	No phone service	DSL	Yes	No	No	No	No	No	Month-to-month	No	Mailed check	29.75	301.9	No
7892-POOKP	Female	0	Yes	No	28	Yes	Yes	Fiber optic	No	No	Yes	Yes	Yes	Yes	Month-to-month	Yes	Electronic check	104.8	3046.05	Yes
6388-TABGU	Male	0	No	Yes	62	Yes	No	DSL	Yes	Yes	No	No	No	No	One year	No	Bank transfer (automatic)	56.15	3487.95	No
9763-GRSKD	Male	0	Yes	Yes	13	Yes	No	DSL	Yes	No	No	No	No	No	Month-to-month	Yes	Mailed check	49.95	587.45	No

Why solve this project ?

After completing this project, you will have a better understanding of how to build a boosting model. In this project, you will apply the following concepts.

- Handling missing values in data
- Applying AdaBoost
- Applying XGBoost
- Interpreting evaluation metrics

Load data

The first step - you know the drill by now - load the dataset and see how it looks like. Additionally, split it into train and test set.

Instructions:

- Load the dataset from path using the `"read_csv()"` method from pandas and store it in a variable called `'df'`
- Store all the features(All columns except `'customerID'`, `'churn'`) of `'df'` in a variable called `X`
- Store the target variable (`Churn`) of `'df'` in a variable called `y`
- Split `'X'` and `'y'` into `X_train,X_test,y_train,y_test` using `train_test_split()` function. Use `test_size = 0.3` and `random_state = 0`

Skills Covered:

Data Wrangling | Python

```
import pandas as pd
from sklearn.model_selection import train_test_split
#path - Path of file

# Code starts here
df = pd.read_csv(path)
print(df.shape)
X = df.drop(['customerID','Churn'],axis=1)
print(X.shape)
y = df['Churn']
print(y.shape)
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=0)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

OUTPUT

RESULT

```
(7043, 21)
(7043, 19)
(7043,)
(4930, 19)
(2113, 19)
(4930,)
(2113,)
```

Clean Data

In this task, we will try to replace the missing values and modify some column values.

Instructions

- Replace the spaces(' ') in `TotalCharges` column of `'X_train'` with `np.NaN` and assign it back to the same column. Do the same for `'X_test'`
- Change the `TotalCharges` column of `'X_train'` to `float` data type. Do the same for `'X_test'`
- Fill the missing values(NaN) of `TotalCharges` column of `'X_train'` with its mean using `"fillna()"` assign it back to the same column. Do the same for `'X_test'`
- Check whether any other NaN value exists in train data using `"isnull().sum()"`
- Label encode all categorical columns of `'X_train'` using `"LabelEncoder()"`. Do the same for `'X_test'`
- Using `"replace()"` function, replace the values of `'y_train'` in the following way : `{'No':0, 'Yes':1}` (i.e. Replace No with 0 and Yes with 1). Do the same with `y_test`.

```
import numpy as np
from sklearn.preprocessing import LabelEncoder

# Code starts here
X_train['TotalCharges'] = X_train['TotalCharges'].replace(r'^\s*$', np.nan, regex=True).astype(float)
X_test['TotalCharges'] = X_test['TotalCharges'].replace(r'^\s*$', np.nan, regex=True).astype(float)
X_train['TotalCharges'] = X_train['TotalCharges'].fillna(X_train['TotalCharges'].mean())
X_test['TotalCharges'] = X_test['TotalCharges'].fillna(X_test['TotalCharges'].mean())
print(X_train.isnull().sum())
print(X_test.isnull().sum())

cat_cols = X_train.select_dtypes(include='O').columns.tolist()
#Label encoding train data
for x in cat_cols:
    le = LabelEncoder()
    X_train[x] = le.fit_transform(X_train[x])
    X_test[x] = le.fit_transform(X_test[x])
y_train = y_train.replace({'No':0, 'Yes':1})
y_test = y_test.replace({'No':0, 'Yes':1})
```

TRY IT

OUTPUT

RESULT

```
gender            0
SeniorCitizen     0
Partner           0
Dependents        0
```

In this task, we will try to predict the churning of customers using AdaBoost

Instructions

- Print values of `X_train`, `X_test`, `y_value`, `y_test` to take a look at their transformed versions
- Initialise a AdaBoost model with `AdaBoostClassifier()` having `random_state=0` and save it to a variable called `'ada_model'`.
- Fit the model on the training data `'X_train'` and `'y_train'` using the `'fit()'` method.
- Store the prediction of `'X_test'` by `'ada_model'` in a variable called `'y_pred'`
- Find out the accuracy score between `y_test` and `y_pred` using `"accuracy_score()"` and save it in a variable called `'ada_score'`
- Since it's a slightly imbalanced dataset, find out the `confusion matrix` between `'y_test'` and `'y_pred'` using `"confusion_matrix()"` and save it in a variable called `'ada_cm'`
- Also find out the `classification_report` between `'y_test'` and `'y_pred'` using `"classification_report()"` and save it in a variable called `'ada_cr'`

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score,classification_report,confusion_matrix

# Code starts here
print(X_train.head())
print(X_test.head())
print(y_train.head())
print(y_test.head())
ada_model = AdaBoostClassifier(random_state=0)
ada_model.fit(X_train,y_train)
y_pred = ada_model.predict(X_test)
ada_score = accuracy_score(y_test,y_pred)
print("AdaBoost Score : {}".format(ada_score))
ada_cm = confusion_matrix(y_test,y_pred)
print("Confusion Matrix For AdaBoost Classifier: {}".format(ada_cm))
ada_cr = classification_report(y_test,y_pred)
print("Classification Report For AdaBoost Classifier: {}".format(ada_cr))
```

TRY IT

```
Name: Churn, dtype: int64

AdaBoost Score : 0.795551348793185
Confusion Matrix For AdaBoost Classifier: [[1371  189]
 [ 243  310]]
Classification Report For AdaBoost Classifier:              precision    recall  f1-score   support
          0       0.85      0.88      0.86      1560
          1       0.62      0.56      0.59       553
avg / total       0.79      0.80      0.79      2113
```

XGBoost Implementation

Let's also try and implement XGBoost for our customer churn problem and see how it performs in comparision to AdaBoost

Instructions

- Initialise a XGBoost Classifier model with `XGBClassifier()` having `random_state=0` and save it to a variable called `'xgb_model'`.
- Fit the model on the training data `'X_train'` and `'y_train'` using the `'fit()'` method.
- Store the prediction of `'X_test'` by `'xgb_model'` in a variable called `'y_pred'`
- Find out the accuracy score between `y_test` and `y_pred` using `"accuracy_score()"` and save it in a variable called `'xgb_score'`
- Since it's a slightly imbalanced dataset, find out the `confusion matrix` between `'y_test'` and `'y_pred'` using `"confusion_matrix()"` and save it in a variable called `'xgb_cm'`
- Also find out the `classification_report` between `'y_test'` and `'y_pred'` using `"classification_report()"` and save it in a variable called `'xgb_cr'`

Observation

The score of XGBoost is very close to the AdaBoost score(~0.001).

Let's try to see if we can make it perform better

Instructions

- Initialise a grid search object with `GridSearch()` having `estimator=xgb_clf` & `param_grid=parameter` and save it to a variable called `'clf_model'`.
- Fit the model on the training data `'X_train'` and `'y_train'` using the `'fit()'` method.
- Store the prediction of `'X_test'` by `'clf_model'` in a variable called `'y_pred'`
- Find out the accuracy score between `y_test` and `y_pred` using `"accuracy_score()"` and save it in a variable called `'clf_score'`
- Find out the `confusion matrix` between `'y_test'` and `'y_pred'` using `"confusion_matrix()"` and save it in a variable called `'clf_cm'`.
- Also find out the `classification_report` between `'y_test'` and `'y_pred'` using `"classification_report()"` and save it in a variable called `'clf_cr'`.
- Print and compare the accuracy score, confusion matrix, classification report between `'xgb_model'` and `'clf_model'`.

Code:

```
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV

#Parameter list
parameters={'learning_rate':[0.1,0.15,0.2,0.25,0.3],
            'max_depth':range(1,3)}
```

```
# Code starts here

#Initializing the model
xgb_model = XGBClassifier(random_state=0)

#Fitting the model on train data
xgb_model.fit(X_train,y_train)

#Making prediction on test data
y_pred = xgb_model.predict(X_test)

#Finding the accuracy score
xgb_score = accuracy_score(y_test,y_pred)
print("Accuracy: ",xgb_score)

#Finding the confusion matrix
xgb_cm=confusion_matrix(y_test,y_pred)
print('Confusion matrix: \n', xgb_cm)

#Finding the classification report
xgb_cr=classification_report(y_test,y_pred)
print('Classification report: \n', xgb_cr)



### GridSearch CV

#Initialsing Grid Search
clf = GridSearchCV(xgb_model, parameters)

#Fitting the model on train data
clf.fit(X_train,y_train)

#Making prediction on test data
y_pred = clf.predict(X_test)

#Finding the accuracy score
clf_score = accuracy_score(y_test,y_pred)
print("Accuracy: ",clf_score)

#Finding the confusion matrix
clf_cm=confusion_matrix(y_test,y_pred)
print('Confusion matrix: \n', clf_cm)

#Finding the classification report
clf_cr=classification_report(y_test,y_pred)
```

```
print('Classification report: \n', clf_cr)


#Code ends here
```

```
XGB Score: 0.79649787032655
Confusion Matrix For XGB Classifier: [[1388  172]
 [ 258  295]]
Classification Report For XGB Classifier:              precision    recall  f1-score   support
          0       0.84      0.89      0.87      1560
          1       0.63      0.53      0.58       553
avg / total       0.79      0.80      0.79      2113
```

```
GridSearchCV Score: 0.8017037387600567
Confusion Matrix For GridSearchCV Classifier: [[1394  166]
 [ 253  300]]
Classification Report For GridSearchCV Classifier:              precision    recall  f1-score   support
          0       0.85      0.89      0.87      1560
          1       0.64      0.54      0.59       553
avg / total       0.79      0.80      0.80      2113
```