Problem Statement

The problem that we will solve is the same one we encountered in the decision tree course(link) related to the banking sector. Just to refresh, the problem statement is as follows: 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. To get a deeper understanding of the problem, you can read more about the problem here.

We will be using the dataset with all the 17 features for the purpose of understanding ensemble methods.

- `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)

We will again try to fit the decision tree model we learned on the data

---

Accuracy score using Decision Tree(Note the overfitting of train data)

```
#Training the model
Decision_Tree.fit(data_train, label_train)

#Accuracy of the train data
Decision_Tree_Score=Decision_Tree.score(data_train,label_train)
print("Training Score: %.2f "%Decision_Tree_Score)

#Accuracy of the test data
Decision_Tree_Score=Decision_Tree.score(data_test,label_test)
print("Training Score: %.2f "%Decision_Tree_Score)
```

Output:

```
Training Score: 1.00

Test Score: 0.76
```

Can we do better than that?

We know the decision tree is powerful, what if we combine multiple decision tree models together?

Let's call this combination model as an ensemble model

Accuracy score using the Ensemble model

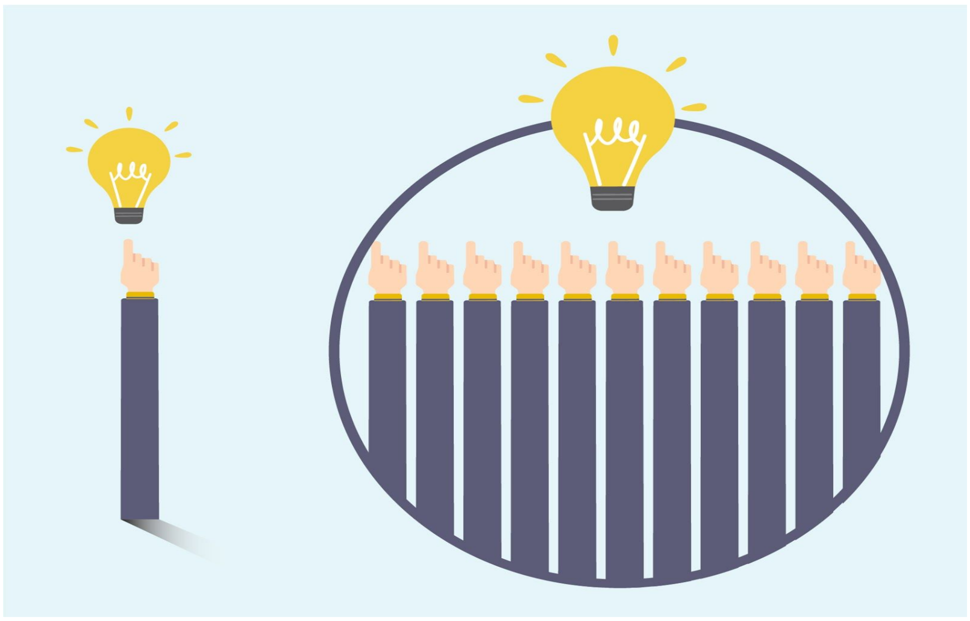
```
Ensemble_Model.fit(data_train, label_train)

#Accuracy of the train data
Ensemble_Model_Score=Ensemble_Model.score(data_train,label_train)
print("Training Score: %.2f "%Ensemble_Model_Score)

#Accuracy of the test data
Ensemble_Model_Score=Ensemble_Model.score(data_test,label_test)
print("Test Score: %.2f "%Ensemble_Model_Score)
```

Output:

```
Training Score: 0.83

Test Score: 0.82
```

Wisdom of Crowd

It is the idea that when it comes to problem-solving and decision making, the collective intelligence of many often surpasses the intelligence of a single expert.



For eg: Suppose you decide you want to go to 'Rome' for your vacation. However, you are not sure if it is a good place to visit during Summer. So you ask a bunch of people

1. A travel guide, whose opinions about travel destination are 70% times similar to yours.
2. A YouTube trip vlogger, who is 80% times similar to your opinions about a destination.
3. A close friend of yours, who is 60% of times similar to your opinions.

Though individually each one would may have some sort of bias(For eg: Your friend said no cause of his aversion towards forts in Rome)but when taken together, the probabilty of them being wrong simultaneously is equal to-

- P = (1 - 0.7) x (1 - 0.8) x (1 - 0.6)
- P = 0.024

Which means that there is 97.6% chance that their opinion will be good (given their opinions are independent from each other).

Ensemble works on similar principles.

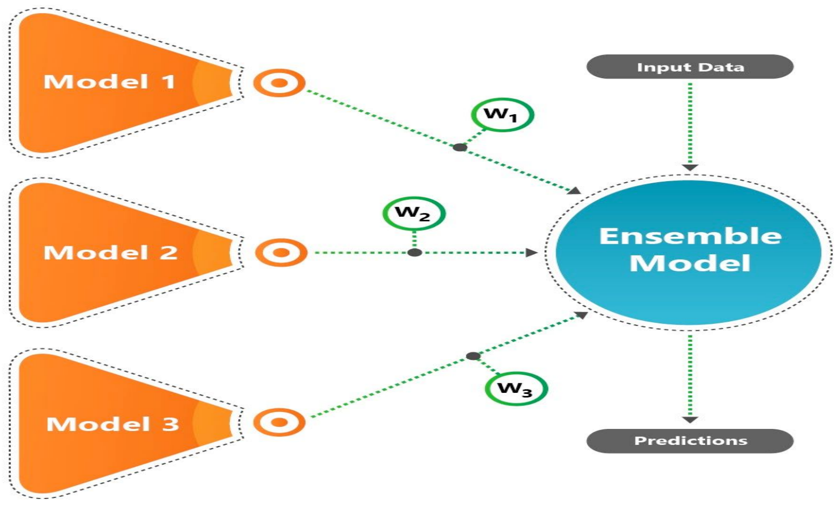
Definition

Ensemble modeling is a machine learning technique of combining multiple machine learning models to produce one optimal model . Though there exists many different techniques of it, at their core they all employ the same two methods:

- Produce a cohort of predictions using simple ML algorithms.
- Combine the predictions into one "aggregated" model.

Why ensemble modeling?

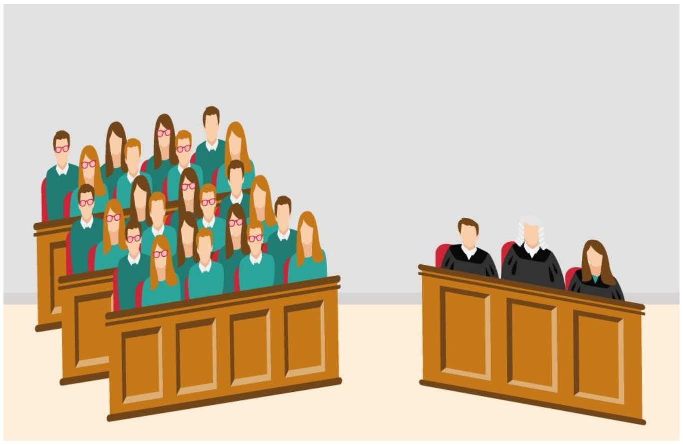
In real world scenarios, generalizing on a dataset by a single model can be challenging. Some models will be able to capture one aspect of the data well while others will do well in capturing something else. Ensemble modeling provides us a with family of techniques that help reduce errors and make predictions where the final evaluation metrics(For eg: Accuracy) are better than they are for each of the individual models.



Let's further explore this technique of Ensembling using a mathematical thought process.

Strong vs Weak Learner

Condorcet’s Jury Theorem



Let's say a jury of voters are needed to make a decision regarding a binary outcome (for example to convict a defendant or not).

If each voter has a probability p of being correct and the probability of a majority of voters being correct is L, then L > p if p > 0.5 if the voters as independent from each other. Interestingly, L approaches 1 as the number of voters approaches infinity.

In human language, p > 0.5 means that the individual judgments (votes) are at least a little better than random chance.

Now, let's take this analogy to the world of ML:

- Verdict ~> classification prediction
- Jury members ~> ML models
- votes ~> individual predictions

This means that employing multiple ML models should improve the performance according to the Condorcet's theorem(and it does!)

We only need a large number of learners, whose predictive power is just slightly better than random chance (tossing a coin in case of binary classification problem!) for ensembling to work.Such learners have a special name --"weak learners".

Formally, they are defined as:

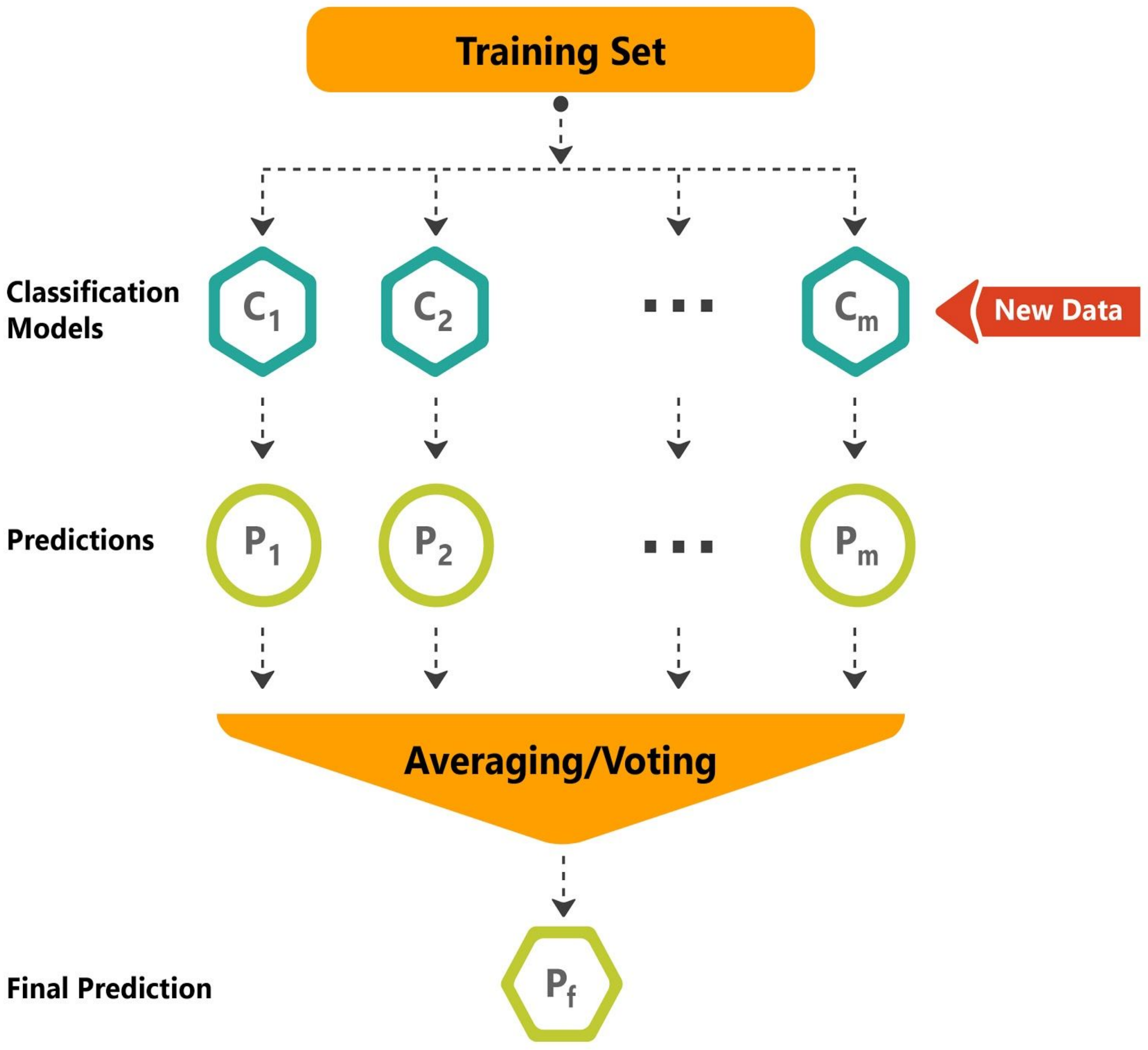
- Weak Learner: Given a labeled dataset, a Weak Learner produces a classifier which is at least a little more accurate than random classification.
- Strong Learner: We call a machine learning model a Strong Learner which, given a labeled dataset, can produce results arbitrarily well-correlated with the true classification.

Different techniques of Ensembling

Following are the different techniques, ensemble modeling is broadly divided into:

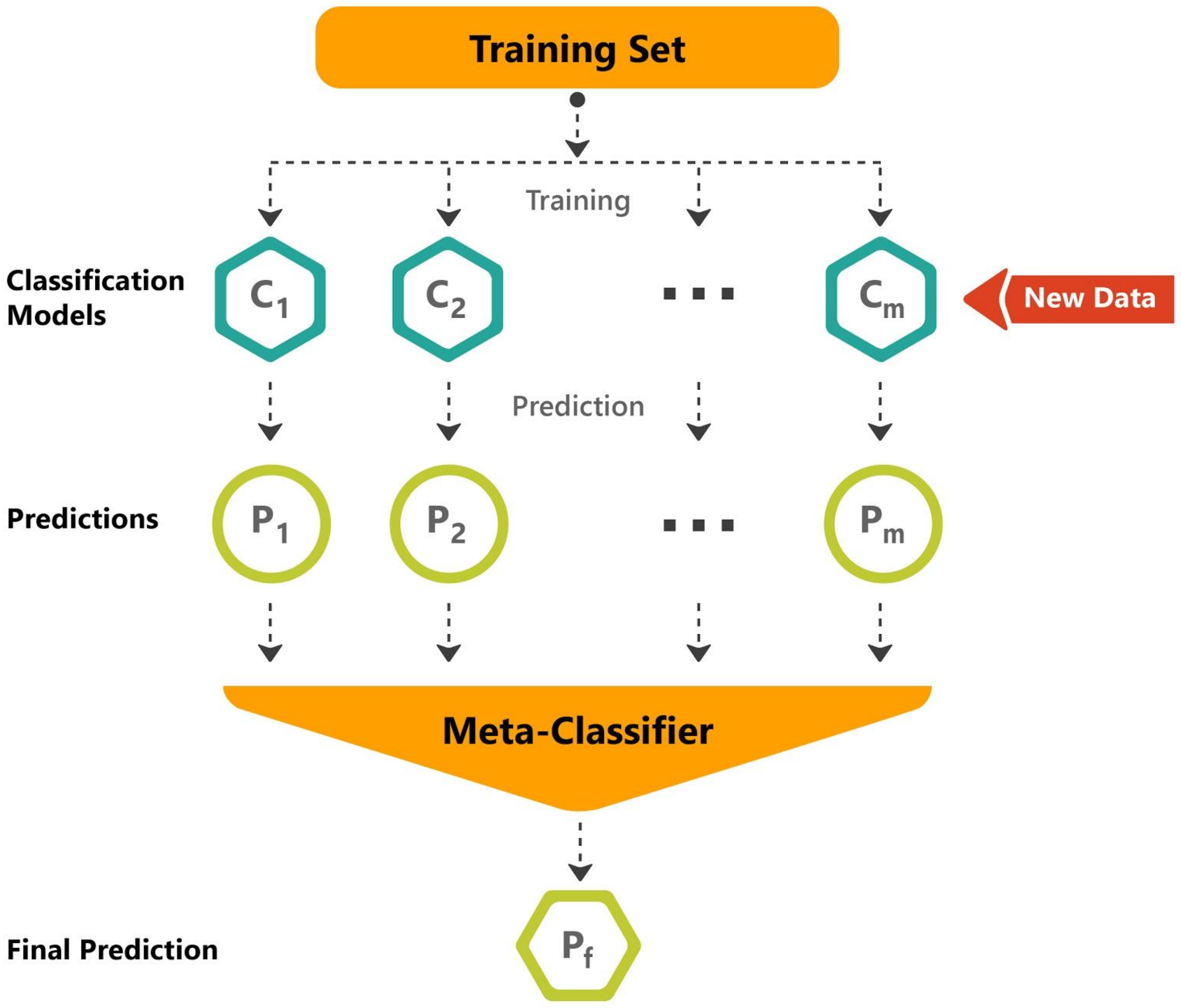
1. Voting/Aggregating

This technique involves building multiple models(usually of differing types) and the predictions which we get after averaging(regression) or voting(classification) the results of the models are used as the final prediction.



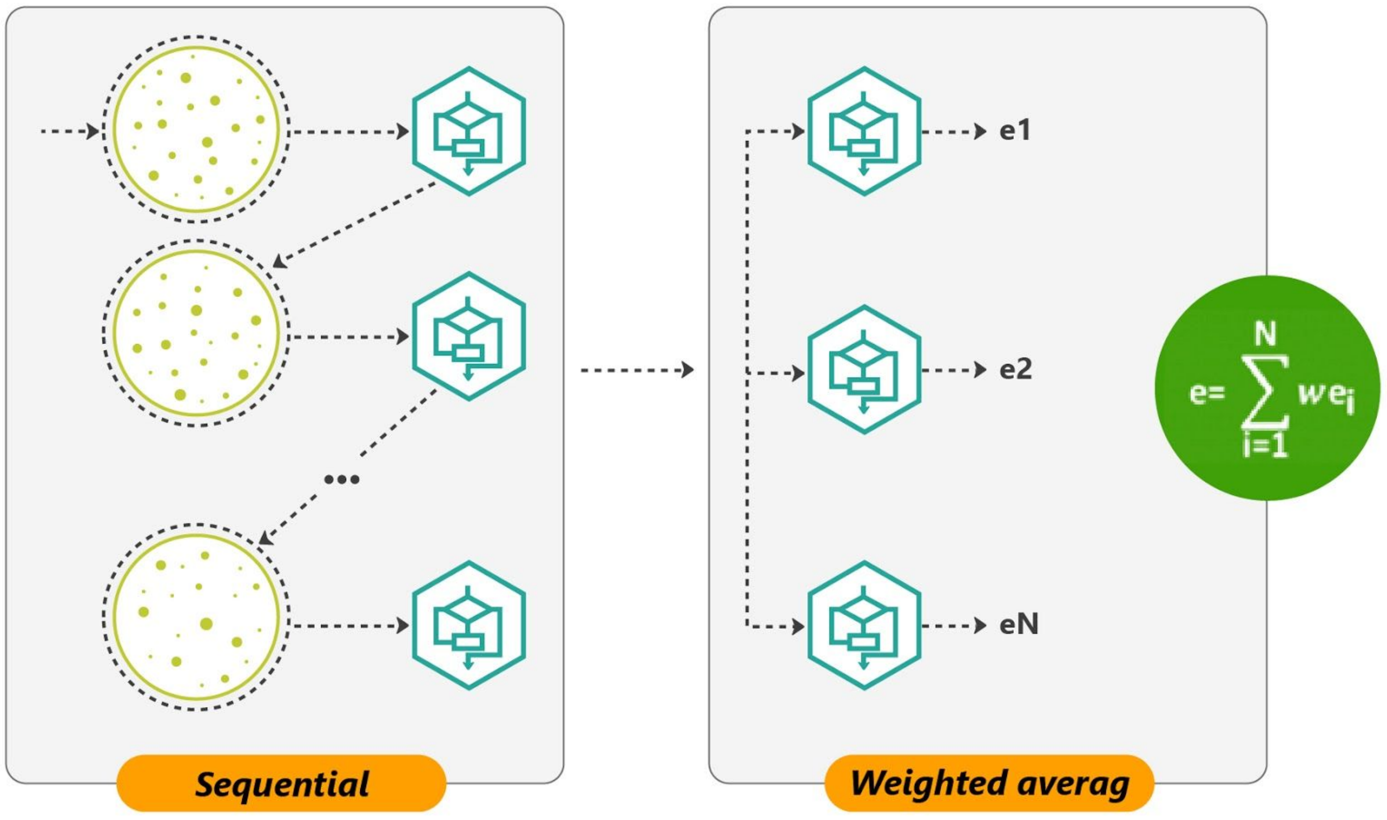
2. Stacking

This technique involves combining multiple classification models via a meta-classifier i.e. instead of using trivial functions to aggregate the predictions , a model is trained to perform this aggregation.



3. Boosting

This technique involves a sequential process, where each subsequent model attempts to correct the errors of the previous model. More weight is given to examples that were misclassified by earlier rounds and then the final prediction is produced by combining the results using a weighted average approach.



Naive Aggregation

Suppose you want to watch a movie of your favourite actor this weekend. His last two movies had been disappointing, so you decide to watch the movie based upon the ratings given by your three friends. The most obvious and intuitive way then would be to average out all the three ratings and make your decision.

Similarly the most intuitive way to combine models is averaging out their indvidual predictions.

Naive aggregation works by aggregating the final output through averaging (regression) or voting (classification). It works best with algorithms which learn very differently from each other, thereby complementing each others' decisions.

Soft Voting vs Hard Voting

Since, every classification algorithm first calculates the probabilities of each outcome, and then produces the prediction, the aggregation could be done either on calculated probabilities, or final predictions.

- In hard voting, the voting classifier takes majority of its base learners' predictions
- In soft voting, the voting classifier takes into account the probability values by its base learners

In general, soft voting has been observed to perform better than hard voting.

Python Implementation of Voting

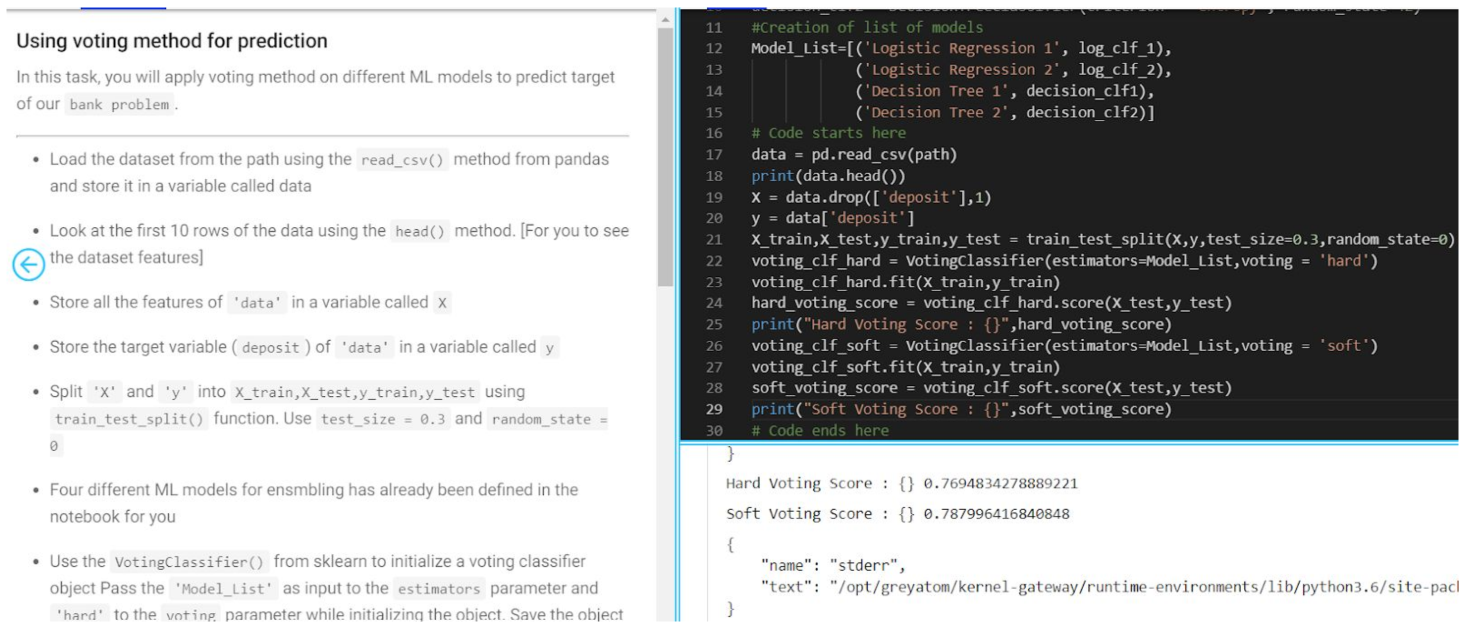
For this we will use a subset of our original dataset containing only 3000 datapoints.

```
from sklearn.ensemble import VotingClassifier
```

```
#Four random models intitialised
log_clf_1 = LogisticRegression(random_state=0)
log_clf_2 = LogisticRegression(random_state=42)
decision_clf1 = DecisionTreeClassifier(criterion = 'entropy',random_state=0)
decision_clf2 = DecisionTreeClassifier(criterion = 'entropy', random_state=42)


#Creating a list of models
Model_List=[('Logistic Regression 1', log_clf_1),
            ('Logistic Regression 2', log_clf_2),
            ('Decision Tree 1', decision_clf1),
            ('Decision Tree 2', decision_clf2)]


#Features
X= bank_sample.drop(['deposit'],1)

#Target variable
y=bank_sample['deposit'].copy()


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


#Initialising hard voting model
voting_clf_hard = VotingClassifier(estimators = Model_List,
                                   voting = 'hard')

#Fitting the data
voting_clf_hard.fit(Data_train, label_train)

#Scoring the model for train
hard_voting_score=voting_clf_hard.score(Data_train,label_train)
print("Hard Voting Train Accuracy:%.2f"%hard_voting_score)


#Scoring the model for test
hard_voting_score=voting_clf_hard.score(Data_test,label_test)
print("Hard Voting Test Accuracy:%.2f"%hard_voting_score)

#Initialising soft voting model
voting_clf_soft = VotingClassifier(estimators = Model_List,voting = 'soft')


#Fitting the data
voting_clf_soft.fit(Data_train, label_train)

#Scoring the model for train
soft_voting_score= voting_clf_soft.score(Data_train,label_train)
print("Soft Voting Train Accuracy: %.2f"%soft_voting_score)

#Scoring the model for test
soft_voting_score= voting_clf_soft.score(Data_test,label_test)
print("Soft Voting Test Accuracy: %.2f"%soft_voting_score)

#Solution ends
```

Output:

```
Hard Voting Train Accuracy:0.88

Hard Voting Test Accuracy:0.75

Soft Voting Train Accuracy: 1.00

Soft Voting Test Accuracy: 0.75
```



Using voting method for prediction

In this task, you will apply voting method on different ML models to predict target of our `bank problem`.

- Load the dataset from the path using the `read_csv()` method from pandas and store it in a variable called data
- Look at the first 10 rows of the data using the `head()` method. [For you to see the dataset features]
- Store all the features of `'data'` in a variable called `X`
- Store the target variable (`deposit`) of `'data'` 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`
- Four different ML models for ensmbling has already been defined in the notebook for you
- Use the `VotingClassifier()` from sklearn to initialize a voting classifier object Pass the `'Model_List'` as input to the `estimators` parameter and `'hard'` to the `voting` parameter while initializing the object. Save the object

```
#Creation of list of models
Model_List=[('Logistic Regression 1', log_clf_1),
            ('Logistic Regression 2', log_clf_2),
            ('Decision Tree 1', decision_clf1),
            ('Decision Tree 2', decision_clf2)]
# Code starts here
data = pd.read_csv(path)
print(data.head())
X = data.drop(['deposit'],1)
y = data['deposit']
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=0)
voting_clf_hard = VotingClassifier(estimators=Model_List,voting = 'hard')
voting_clf_hard.fit(X_train,y_train)
hard_voting_score = voting_clf_hard.score(X_test,y_test)
print("Hard Voting Score : {}",hard_voting_score)
voting_clf_soft = VotingClassifier(estimators=Model_List,voting = 'soft')
voting_clf_soft.fit(X_train,y_train)
soft_voting_score = voting_clf_soft.score(X_test,y_test)
print("Soft Voting Score : {}",soft_voting_score)
# Code ends here
```

```
}
Hard Voting Score : {} 0.7694834278889221

Soft Voting Score : {} 0.787996416840848

{
    "name": "stderr",
    "text": "/opt/greyatom/kernel-gateway/runtime-environments/lib/python3.6/site-pac
}
```

Code:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import VotingClassifier
import warnings
warnings.filterwarnings('ignore')


#Different models initialised
log_clf_1 = LogisticRegression(random_state=0)
log_clf_2 = LogisticRegression(random_state=42)
decision_clf1 = DecisionTreeClassifier(criterion = 'entropy',random_state=0)
decision_clf2 = DecisionTreeClassifier(criterion = 'entropy', random_state=42)


#Creation of list of models
Model_List=[('Logistic Regression 1', log_clf_1),
           ('Logistic Regression 2', log_clf_2),
           ('Decision Tree 1', decision_clf1),
           ('Decision Tree 2', decision_clf2)]
```

```
#Solution begins


data=pd.read_csv(path)

#Features
X= data.drop(['deposit'],1)

#Target variable
y=data['deposit'].copy()


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


#Initialising hard voting model
voting_clf_hard = VotingClassifier(estimators = Model_List,
                                  voting = 'hard')

#Fitting the data
voting_clf_hard.fit(X_train, y_train)

#Scoring the model for test
hard_voting_score=voting_clf_hard.score(X_test,y_test)
print("Hard Voting Test Accuracy:%.2f"%hard_voting_score)

#Initialising soft voting model
voting_clf_soft = VotingClassifier(estimators = Model_List,voting = 'soft')


#Fitting the data
voting_clf_soft.fit(X_train, y_train)

#Scoring the model for test
soft_voting_score= voting_clf_soft.score(X_test,y_test)
print("Soft Voting Test Accuracy: %.2f"%soft_voting_score)

#Solution ends
```

Bootstrap Aggregation(Bagging)

Continuing with the movie dilemma of your favourite actor. After getting your friends' opinions, you are still not satisfied and think to yourself,

What could better than the wisdom of crowds?

Ans. Wisdom of diverse experts!.

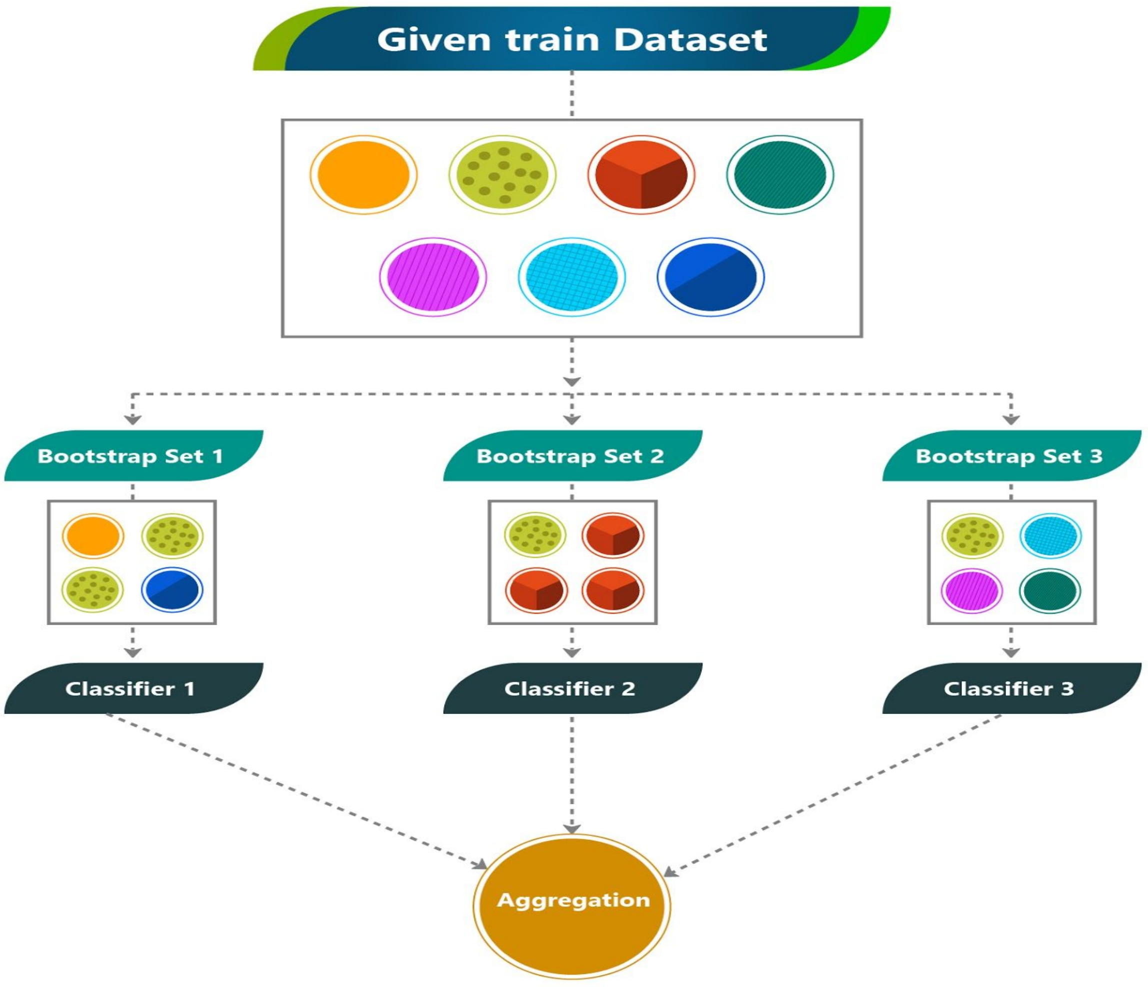
A classic example of it, is the minister cabinet of the kings in the older times, where each minister used to be an expert of a particular area and the king would ask for opinions from them before taking any major decisions.

So instead of going with the opinion of your friends, you decide to see the review of some respected critics. Now instead of getting a general opinion, you have taken the review of movie critics who have unrelated and independent views about the movie.

A similar approach is used in bagging. Each base learner is trained on different sample of data making each learner, a `specialist base learner`.

Definition
Bagging which Bootstrap AGGregatING is usually called, is an approach to ensemble learning where given a training set, multiple different training sets (called bootstrap samples) are created, by sampling with replacement from the original dataset. Then, for each bootstrap sample, a model is built.The individual predictions are then aggregated to form a final prediction.

Unlike naive aggregator, bagging uses a single type of base learner



Bias-Variance trade off

To better understand model predictions, it’s important to understand the prediction errors:

1. Bias
2. Variance

Consider the following:

Assume a dataset with features 'X' and target variable 'y' and you create a model 'F(X)' for predicting 'y'.

The expected squared error for a point x, will be then defined as :

$$Error(x) = E[(y - F(x))^2]$$ `#E(x)=avg(x)`

$Error(X)$can be further broken down into the following:

$$Error(x) = (E[F(x)] - y)^2 + E[(F(x) - E[F(x)])^2] + \alpha$$

Which can also be written as:

$$E(X) = Bias^2 + Variance + Noise$$

Error due to Bias:

The error due to bias is taken as the difference between the expected (or average) prediction of our model and the correct value which we are trying to predict.

Error due to Variance:

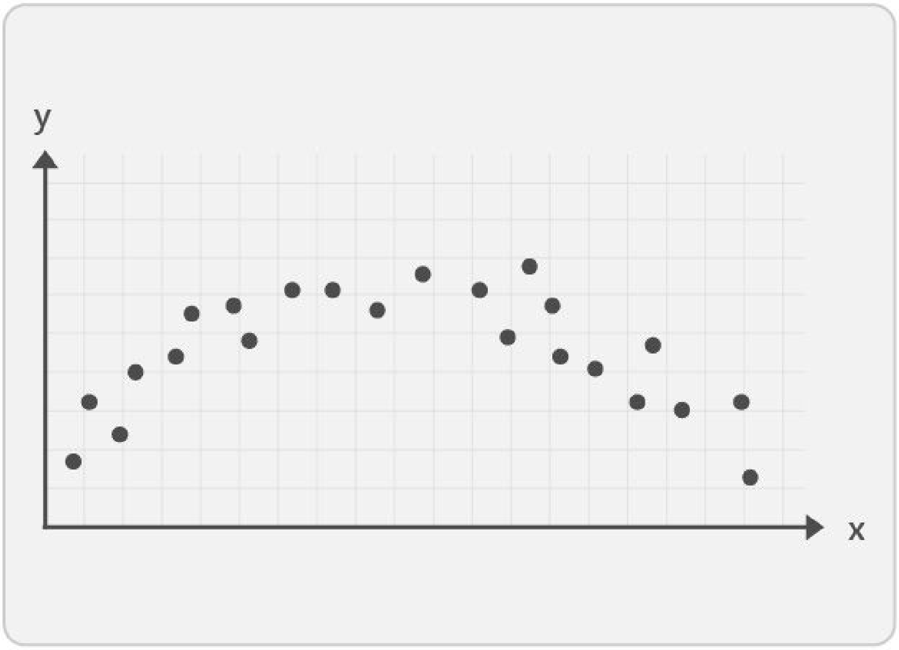
The error due to variance is taken as the variability of a model prediction for a given data point.

Since both are errors, you would ideally want a model having both low bias and low variance

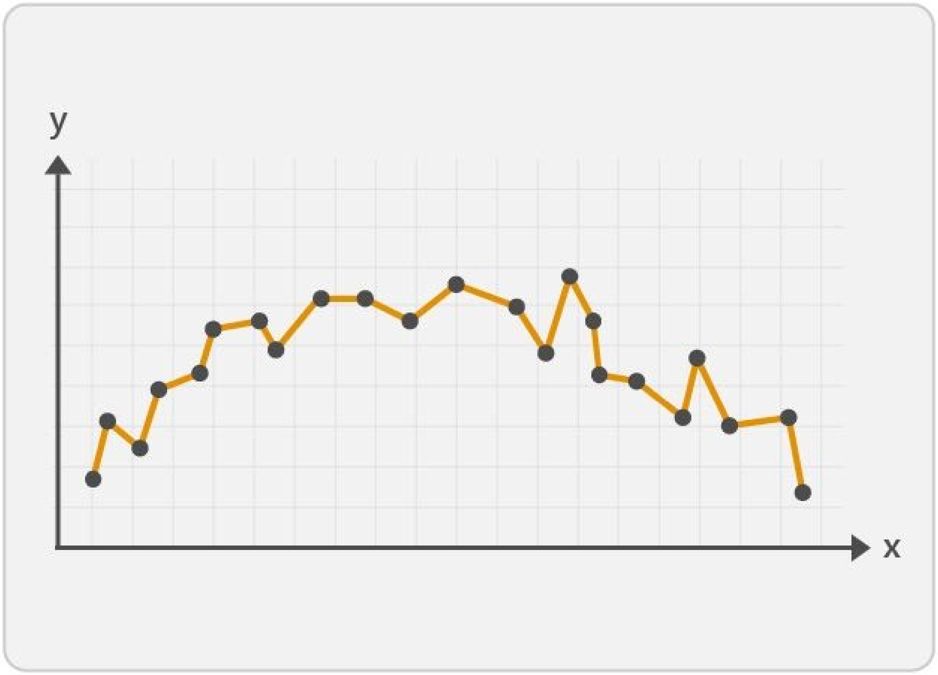
However to achieve that is difficult and there is always a tradeoff between a model’s ability to minimize bias and variance.

Let's understand that with an example:

Consider the following data distibution



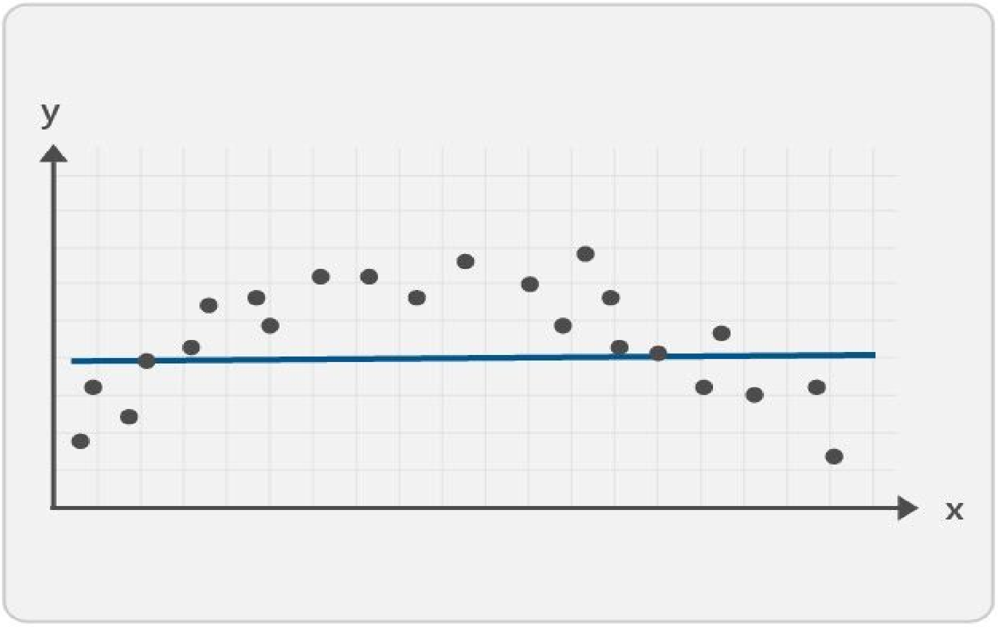
Following will be the model if we try to `reduce bias error`:



Two things you can observe:

- The model has overfit the data(The model has become super complex and has fitted even noise)
- While reducing bias, the variance has increased

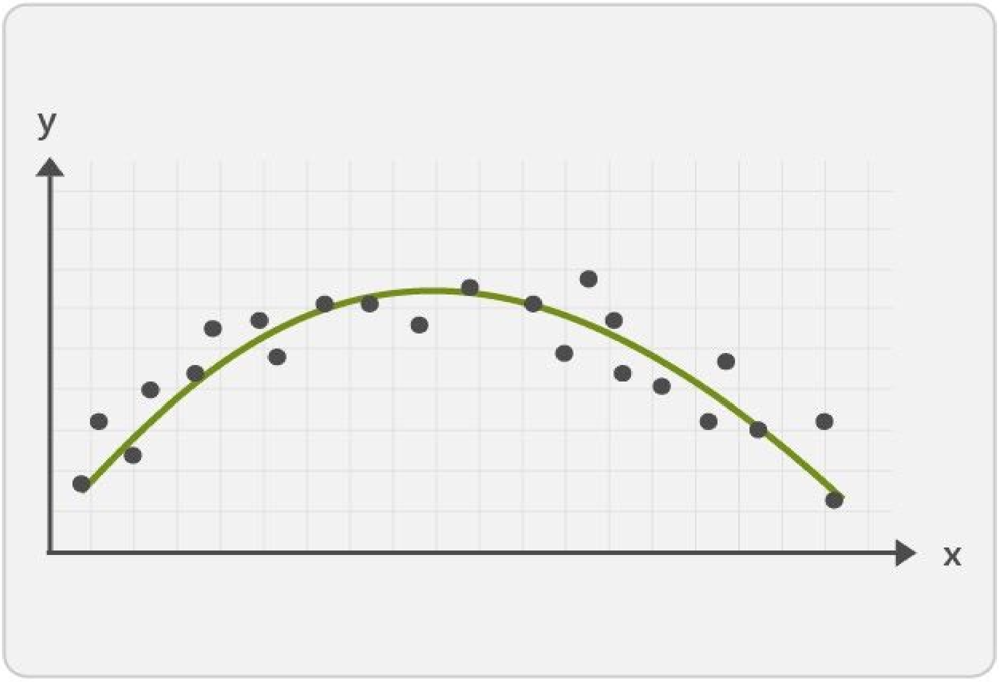
Similarly, following will be the model if we try to `reduce variance error`:



Two things you can observe:

- The model has underfit the data(The model is too simple and has fitted the data too poorly)
- While reducing variance, the bias has increased

Your ideal model therefore becomes one having the perfect-bias variance tradeoff

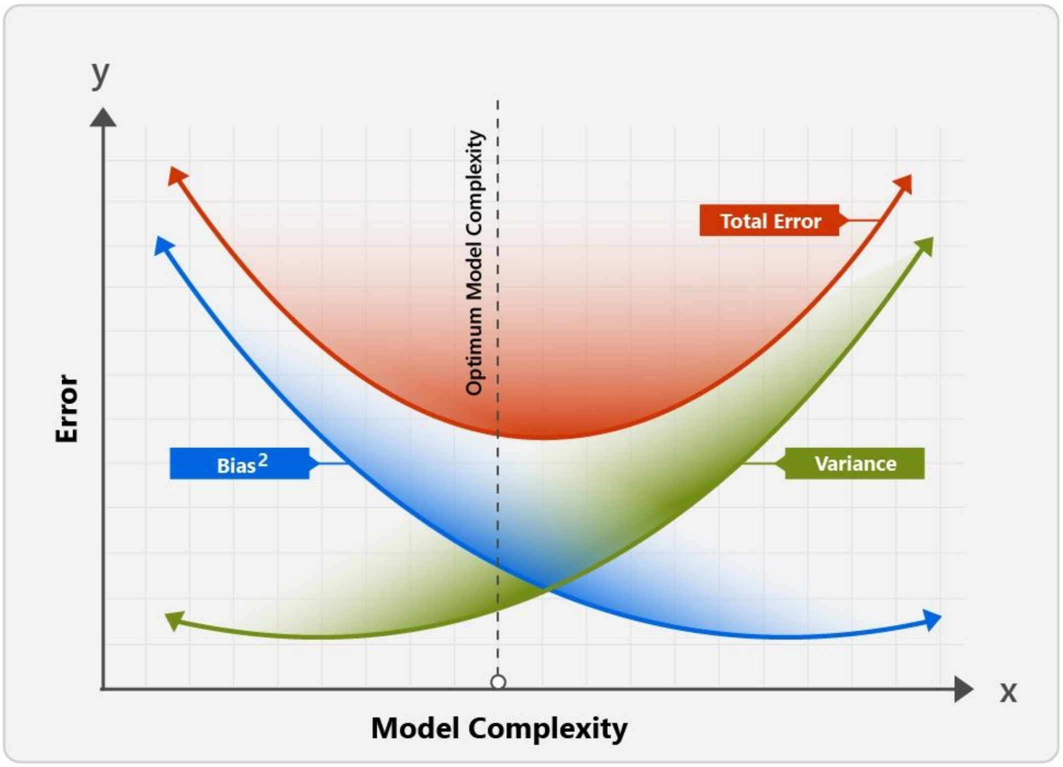


As clear from the example, since a model can't be both less complex and more complex at the same time, there is no escaping the relationship between bias and variance in machine learning.

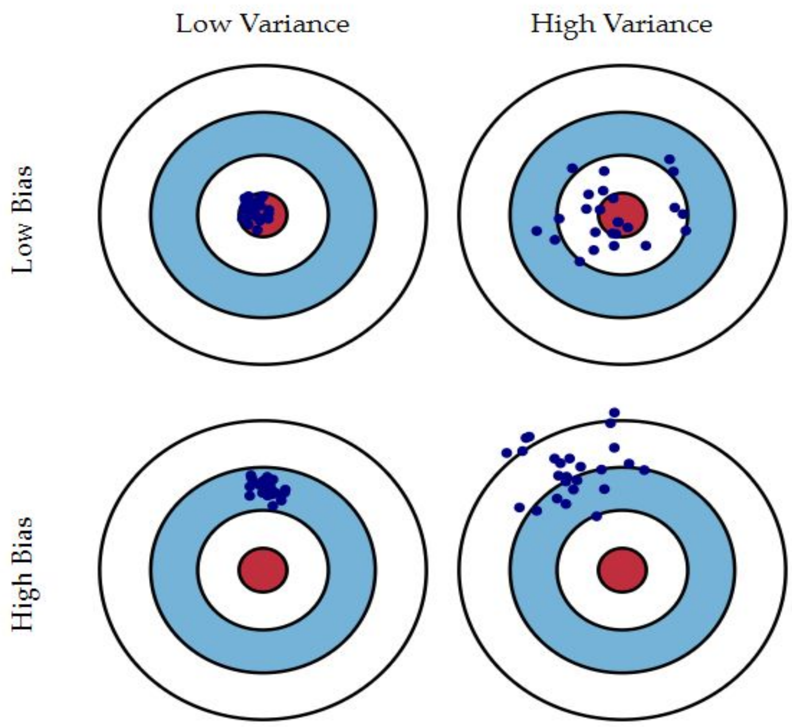
- Increasing in bias = decrease in variance.

- Increasing in variance = decrease in bias.

Following is a graph explaining the error contributed by the bias and variance



Another way to look at it is using the following diagram:



Imagine that the center of the target(bulls eye) is a model that perfectly predicts the correct values. As we move away from the target, our predictions get worse and worse.

The optimum model of course is the one with low bias and low variance.

High bias(low variance) algorithm train models that are consistent but inaccurate on average.

High variance(low bias) algorithm train models that are accurate but inconsistent on average.

Having both high variance and high bias results in the least optimum model which is both inconsistent and inaccurate.

Bias-Variance Tradeoff in Bagging

In bagging, because of bootstrapping, each individual predictor has a higher bias than if it were trained on the original training set. However, a large number of such biases will get reduced when aggregated, hence the bias of the resulting bagging is only slightly higher than a comparable single predictor strong learner.

At the same time, because bagging provides a way to reduce overfitting owing to less dependence on one particular subset of training data, the variance of resulting strong learner reduces significantly.

Generally, the net result is that the ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.

Python Implementation of Bagging

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

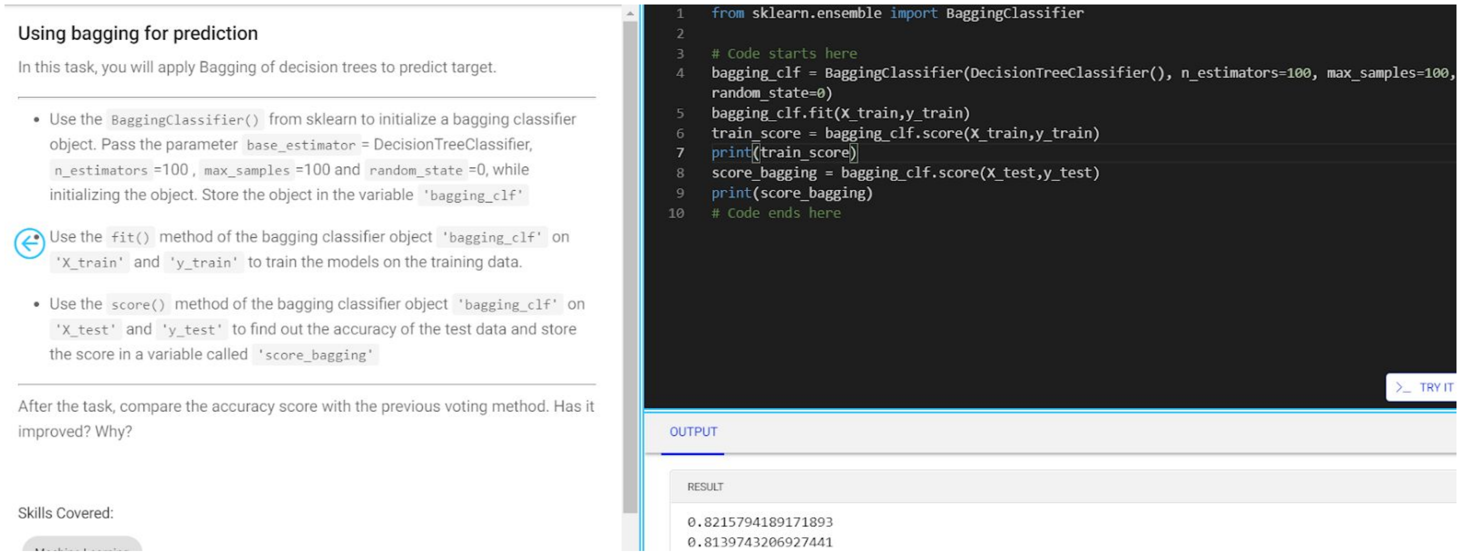
```
#Features
X= bank_sample.drop(['deposit'],1)

#Target variable
y=bank_sample['deposit'].copy()


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

```
#Initialising bagging with appropriate parameters
bagging_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100, max_samples=100,
random_state=0)

#Fitting the data
bagging_clf.fit(X_train, y_train)

#Scoring the model for train data
score_bc_dt = bagging_clf.score(X_train, y_train)
print("Training score: %.2f " % score_bc_dt)


#Scoring the model for test data
score_bc_dt = bagging_clf.score(X_test, y_test)
```

```
print("Testing score: %.2f " % score_bc_dt)
```

Output:

```
Training score: 0.82

Testing score: 0.80
```



Using bagging for prediction

In this task, you will apply Bagging of decision trees to predict target.

- Use the `BaggingClassifier()` from sklearn to initialize a bagging classifier object. Pass the parameter `base_estimator` = DecisionTreeClassifier, `n_estimators` =100 , `max_samples` =100 and `random_state` =0, while initializing the object. Store the object in the variable `'bagging_clf'`
- Use the `fit()` method of the bagging classifier object `'bagging_clf'` on `'X_train'` and `'y_train'` to train the models on the training data.
- Use the `score()` method of the bagging classifier object `'bagging_clf'` on `'X_test'` and `'y_test'` to find out the accuracy of the test data and store the score in a variable called `'score_bagging'`

After the task, compare the accuracy score with the previous voting method. Has it improved? Why?

Skills Covered:

```
from sklearn.ensemble import BaggingClassifier

# Code starts here
bagging_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=100, max_samples=100,
random_state=0)
bagging_clf.fit(X_train,y_train)
train_score = bagging_clf.score(X_train,y_train)
print(train_score)
score_bagging = bagging_clf.score(X_test,y_test)
print(score_bagging)
# Code ends here
```

```
0.8215794189171893
0.8139743206927441
```

Pasting

In bagging we tried to create samples through resampling with replacement, in the same way, we can create samples resampling without replacement for each base learner. Ensemble on such samples is known as Pasting.

Replacement introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting.

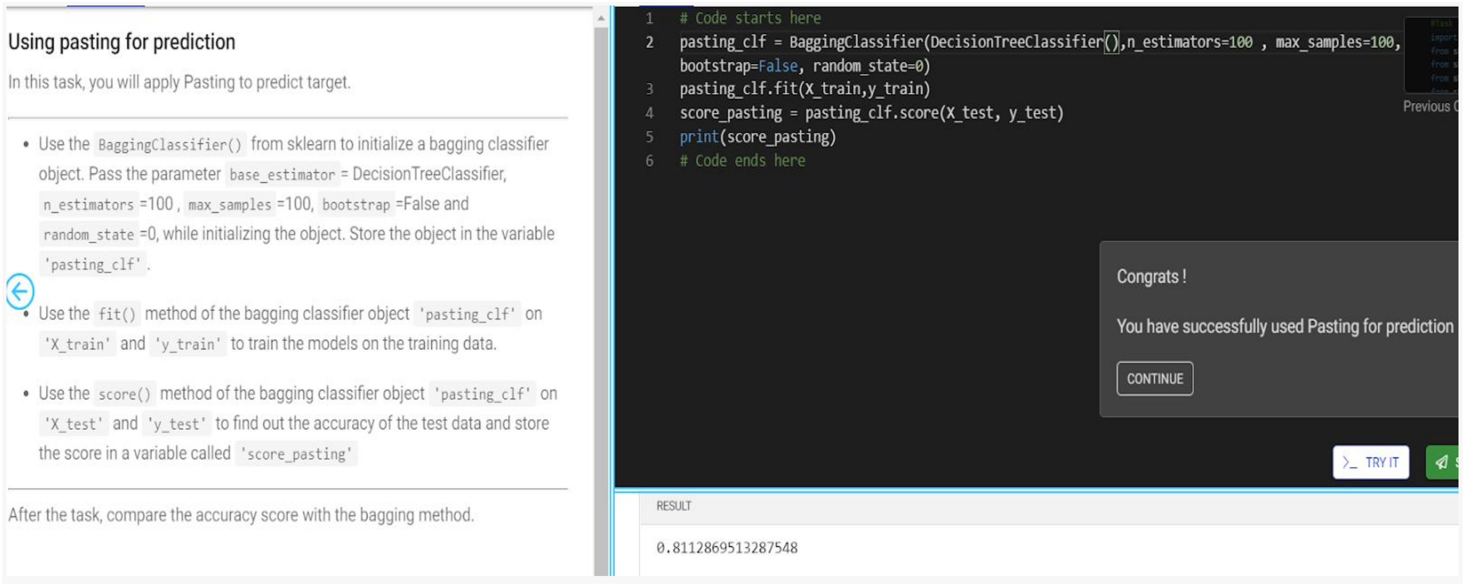
But in pasting the predictors end up being more correlated so the ensemble's variance is increased.

Overall, bagging often results in better models than pasting

However, given spare time and CPU power it is worth using cross- validation to evaluate both bagging and pasting and select the one that works best.

Python implementation of `pasting` is same as `bagging` with an added parameter of changing `bootstrap=False`

```
BaggingClassifier(DecisionTreeClassifier(), n_estimators=100,
max_samples=100,bootstrap=False, random_state=0)
```


Using pasting for prediction

In this task, you will apply Pasting to predict target.

- Use the `BaggingClassifier()` from sklearn to initialize a bagging classifier object. Pass the parameter `base_estimator` = DecisionTreeClassifier, `n_estimators` =100 , `max_samples` =100, `bootstrap` =False and `random_state` =0, while initializing the object. Store the object in the variable `'pasting_clf'`.
- Use the `fit()` method of the bagging classifier object `'pasting_clf'` on `'X_train'` and `'y_train'` to train the models on the training data.
- Use the `score()` method of the bagging classifier object `'pasting_clf'` on `'X_test'` and `'y_test'` to find out the accuracy of the test data and store the score in a variable called `'score_pasting'`

After the task, compare the accuracy score with the bagging method.

```python
# Code starts here
pasting_clf = BaggingClassifier(DecisionTreeClassifier(),n_estimators=100 , max_samples=100,
bootstrap=False, random_state=0)
pasting_clf.fit(X_train,y_train)
score_pasting = pasting_clf.score(X_test, y_test)
print(score_pasting)
# Code ends here
```

Congrats !

You have successfully used Pasting for prediction

```
0.8112869513287548
```

Random Forest

Just when you were about to decide whether you want to watch a movie, one of your friends asked you to go to a party with him. Now you realise in order to make your decision you need to be absolutely sure if the movie is good, otherwise you will regret not going to party. You start thinking,

How can you improve upon the wisdom of diverse experts?

Ans. Experts whose area of expertise is more inclined towards and suitable for solving that particular problem

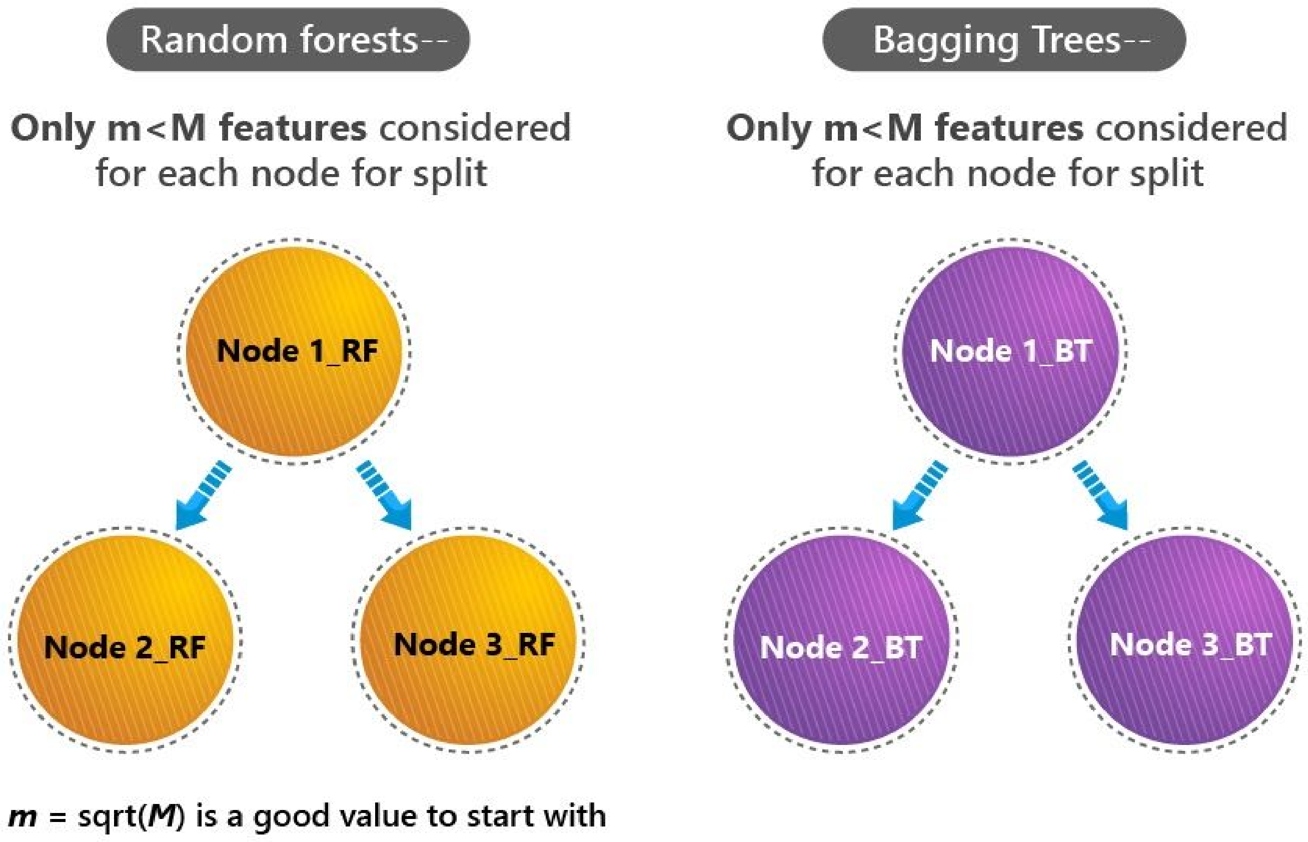
So keeping that in mind, from the many movie critics, you select the following three:

- Critic 1: A respected critic whose opinion about movies usually resonates with your taste in movies.
- Critic 2: A youtube critic who is a huge fan of the actor's movies
- Critic 3: A filmmaker turned critic who specialises in reviewing the particular genre the film is of.

Random forests work similarly by taking only those features that work best to find the optimum solution.

Definition

Random forest is an ensemble method of bagging multiple decision trees. The fundamental difference is that in Random Forests, along with bootstrap sampling, only a subset of features are selected at random out of the total features


Random forests--
Only m<M features considered for each node for split
Node 1_RF
Node 2_RF
Node 3_RF
m = sqrt(M) is a good value to start with
Bagging Trees--
Only m<M features considered for each node for split
Node 1_BT
Node 2_BT
Node 3_BT

Random forest is one of the most popular ensemble algorithms(or for that matter, one of the most popular ML algorithms) owing to it's

- inherent feature selection
- simplicity to train
- and versatilite problem solving(it can be used for classification, regression, cluster analysis..).

Although there exists models that can beat Random Forests for a given dataset (usually boosting or neural network), it’s never by a big margin. Additionaly it also takes much longer to build those models than it takes to build the Random Forest, making them excellent benchmark models.

Bias variance trade off in Random Forests

Random forests results in a greater tree diversity, which trades a higher bias(Owing to features getting subsetted) for a lower variance, generally yielding an overall better model.

Python Implementation of Random Forests

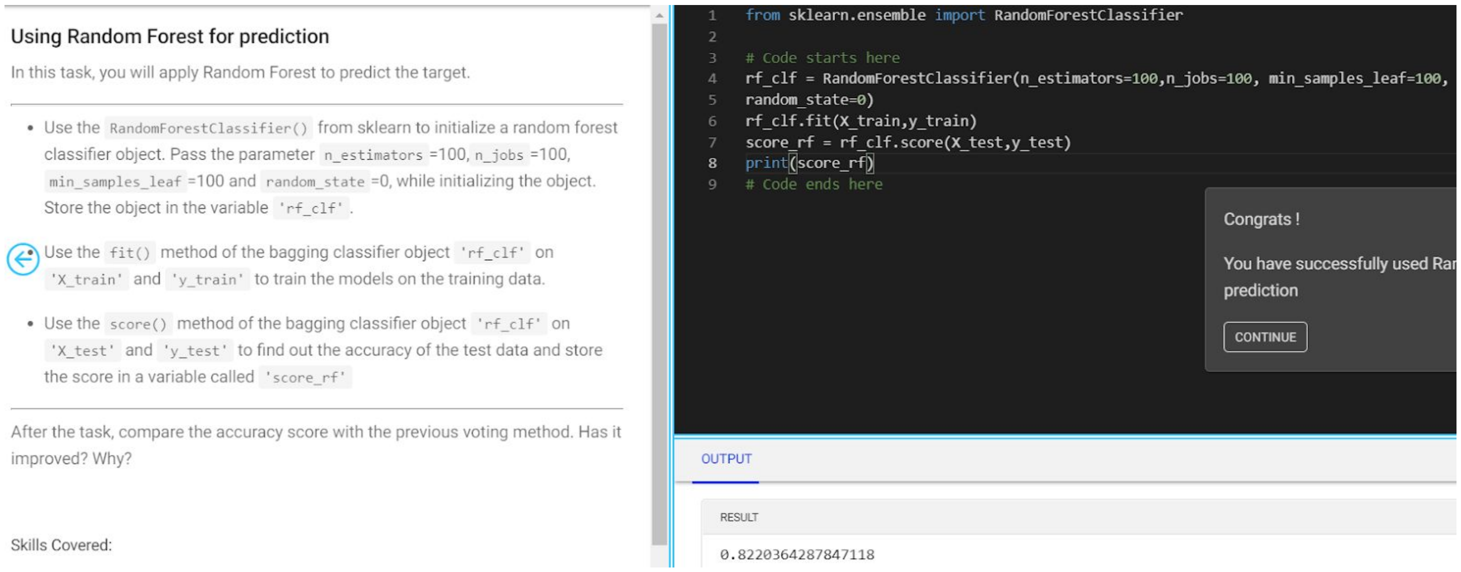
For this we will use the same subset of our original dataset containing only 3000 datapoints.

```
#Features
X= bank_sample.drop(['deposit'],1)

#Target variable
y=bank_sample['deposit'].copy()


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

```
#Initialising Random Forest model
rf_clf=RandomForestClassifier(n_estimators=100,n_jobs=100,random_state=0,
min_samples_leaf=100)

#Fitting on data
rf_clf.fit(X_train, y_train)

#Scoring the model on train data
score_rf=rf_clf.score(X_train, y_train)
print("Training score: %.2f " % score_rf)

#Scoring the model on test_data
score_rf=rf_clf.score(X_test, y_test)
print("Testing score: %.2f " % score_rf)
```

Output:

```
Training score: 0.80

Testing score: 0.79
```



Using Random Forest for prediction

In this task, you will apply Random Forest to predict the target.

- Use the `RandomForestClassifier()` from sklearn to initialize a random forest classifier object. Pass the parameter `n_estimators` =100, `n_jobs` =100, `min_samples_leaf` =100 and `random_state` =0, while initializing the object. Store the object in the variable `'rf_clf'`.
- Use the `fit()` method of the bagging classifier object `'rf_clf'` on `'X_train'` and `'y_train'` to train the models on the training data.
- Use the `score()` method of the bagging classifier object `'rf_clf'` on `'X_test'` and `'y_test'` to find out the accuracy of the test data and store the score in a variable called `'score_rf'`

After the task, compare the accuracy score with the previous voting method. Has it improved? Why?

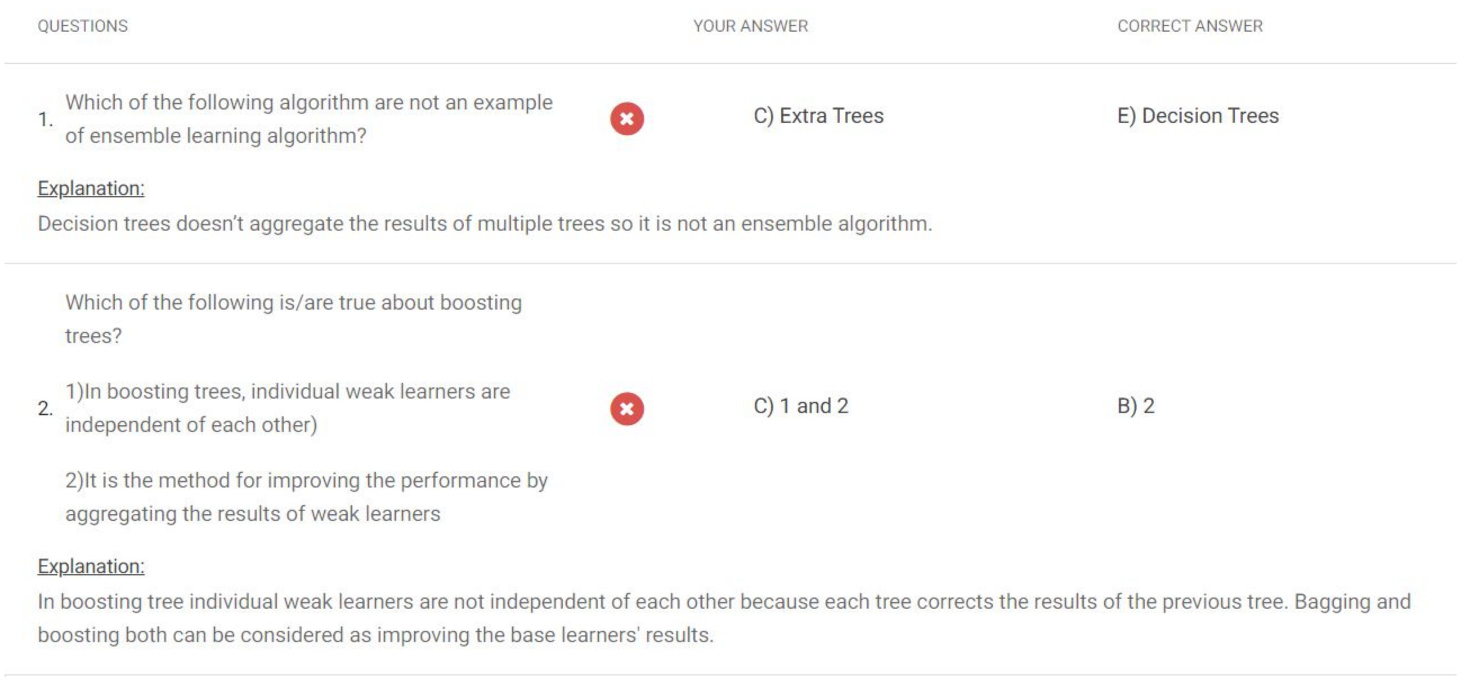
Skills Covered:

```
from sklearn.ensemble import RandomForestClassifier

# Code starts here
rf_clf = RandomForestClassifier(n_estimators=100,n_jobs=100, min_samples_leaf=100,
random_state=0)
rf_clf.fit(X_train,y_train)
score_rf = rf_clf.score(X_test,y_test)
print(score_rf)
# Code ends here
```

Congrats !

You have successfully used Ra
prediction

```
0.8220364287847118
```



QUESTIONS		YOUR ANSWER	CORRECT ANSWER
1. Which of the following algorithm are not an example of ensemble learning algorithm?	✖	C) Extra Trees	E) Decision Trees
Explanation: Decision trees doesn't aggregate the results of multiple trees so it is not an ensemble algorithm.			
2. Which of the following is/are true about boosting trees? 1)In boosting trees, individual weak learners are independent of each other) 2)It is the method for improving the performance by aggregating the results of weak learners	✖	C) 1 and 2	B) 2
Explanation: In boosting tree individual weak learners are not independent of each other because each tree corrects the results of the previous tree. Bagging and boosting both can be considered as improving the base learners' results.			

Definition

Random Forest is indeed impressive. We were able to increase the score of the model by almost 6%(in comparision to the score of our decision tree).

Note: Random forests was the ensemble method used as the example while introducing the ensmble method concept in the first chapter.

The Random Forest object that we used in the previous task was the default one, more specifically the one below:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini', max_depth=None,
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=100, min_samples_split=2, min_weight_fraction_leaf=0.0, n_estimators=100,
n_jobs=100, oob_score=False, random_state=0, verbose=0, warm_start=False)
```

You can learn what each parameter in the model means from the beautifully maintained documentation of Random Forest by sklearn.

Now the question that we are interested in is this,

If the above given parameters are the reason for model to behave a particular way, coudn't we change them to a desired combination to achieve the optimum model?

Enter Hyperparameter Tuning

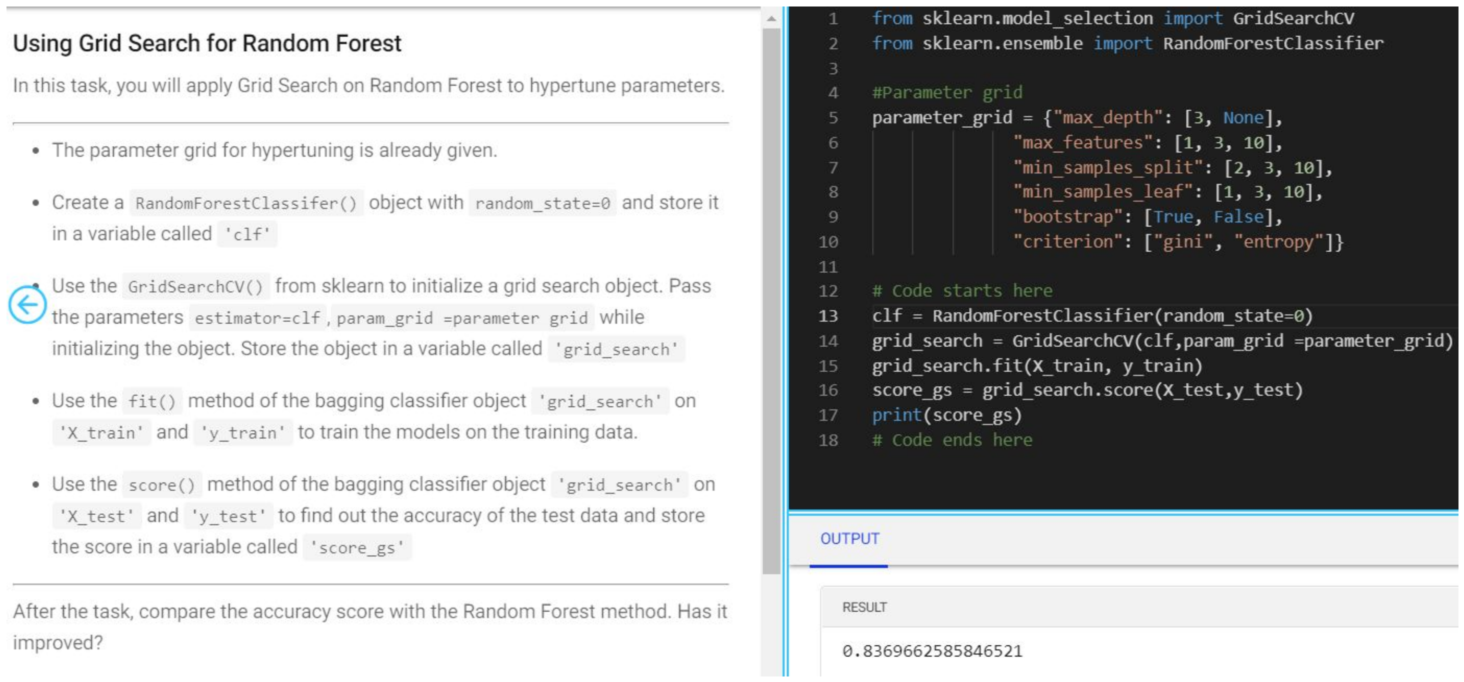
You already learned about hyperparameter tuning before

Just to refresh, hyperparameters are the parameters which define the architecture of model and the process of searching for the ideal hyperparameters for model optimization is referred to as hyperparameter tuning.

For this task we will be using two kinds of hyperparameter tuning method:

1. Grid Search : In Grid search, exhaustive search over specified parameter values for an estimator is done.
2. Randomised Search : In Grid search, exhaustive search over specified parameter values for an estimator is done.

Random Forest and Extra Trees don't have learning rate as a hyperparameter.



Using Grid Search for Random Forest

In this task, you will apply Grid Search on Random Forest to hypertune parameters.

- The parameter grid for hypertuning is already given.
- Create a `RandomForestClassifer()` object with `random_state=0` and store it in a variable called `'clf'`
- Use the `GridSearchCV()` from sklearn to initialize a grid search object. Pass the parameters `estimator=clf` , `param_grid =parameter grid` while initializing the object. Store the object in a variable called `'grid_search'`
- Use the `fit()` method of the bagging classifier object `'grid_search'` on `'X_train'` and `'y_train'` to train the models on the training data.
- Use the `score()` method of the bagging classifier object `'grid_search'` on `'X_test'` and `'y_test'` to find out the accuracy of the test data and store the score in a variable called `'score_gs'`

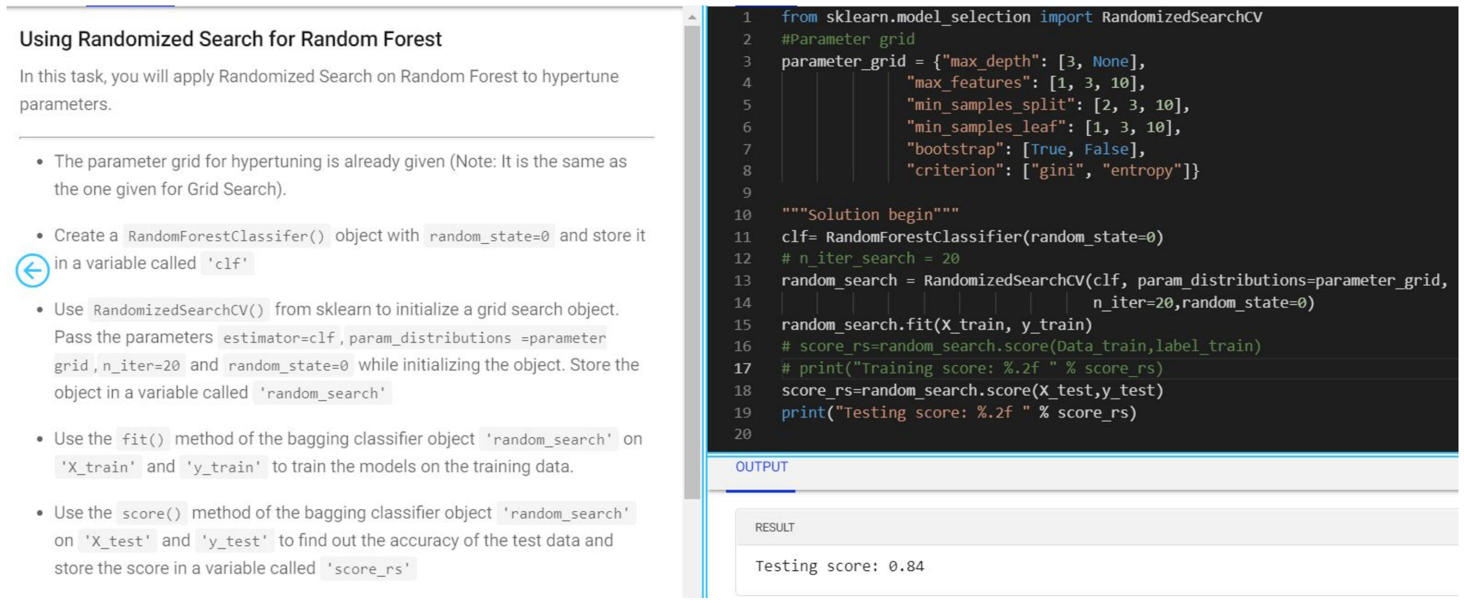
After the task, compare the accuracy score with the Random Forest method. Has it improved?

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

#Parameter grid
parameter_grid = {"max_depth": [3, None],
                  "max_features": [1, 3, 10],
                  "min_samples_split": [2, 3, 10],
                  "min_samples_leaf": [1, 3, 10],
                  "bootstrap": [True, False],
                  "criterion": ["gini", "entropy"]}

# Code starts here
clf = RandomForestClassifier(random_state=0)
grid_search = GridSearchCV(clf,param_grid =parameter_grid)
grid_search.fit(X_train, y_train)
score_gs = grid_search.score(X_test,y_test)
print(score_gs)
# Code ends here
```

OUTPUT

RESULT

```
0.8369662585846521
```



Using Randomized Search for Random Forest

In this task, you will apply Randomized Search on Random Forest to hypertune parameters.

- The parameter grid for hypertuning is already given (Note: It is the same as the one given for Grid Search).
- Create a `RandomForestClassifer()` object with `random_state=0` and store it in a variable called `'clf'`
- Use `RandomizedSearchCV()` from sklearn to initialize a grid search object. Pass the parameters `estimator=clf` , `param_distributions =parameter grid` , `n_iter=20` and `random_state=0` while initializing the object. Store the object in a variable called `'random_search'`
- Use the `fit()` method of the bagging classifier object `'random_search'` on `'X_train'` and `'y_train'` to train the models on the training data.
- Use the `score()` method of the bagging classifier object `'random_search'` on `'X_test'` and `'y_test'` to find out the accuracy of the test data and store the score in a variable called `'score_rs'`

```
from sklearn.model_selection import RandomizedSearchCV
#Parameter grid
parameter_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [2, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

"""Solution begin"""
clf= RandomForestClassifier(random_state=0)
# n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=parameter_grid,
                                   n_iter=20,random_state=0)
random_search.fit(X_train, y_train)
# score_rs=random_search.score(Data_train,label_train)
# print("Training score: %.2f " % score_rs)
score_rs=random_search.score(X_test,y_test)
print("Testing score: %.2f " % score_rs)

```

OUTPUT

RESULT

```
Testing score: 0.84
```

Stacking

So far we have been using naive methods of averaging and voting to combine the predictions, thereby implicitly assigning same weights to the predictions made by all the base learners

However, it might be possible that some base learners might be better at predicting than the others. So, a better aggregation scheme could be to assign some kind of weights to the predictions made by base learners.

We could do it manually but since we have been learning machine learning to predict things anyway,why not use a machine learning model to do it.

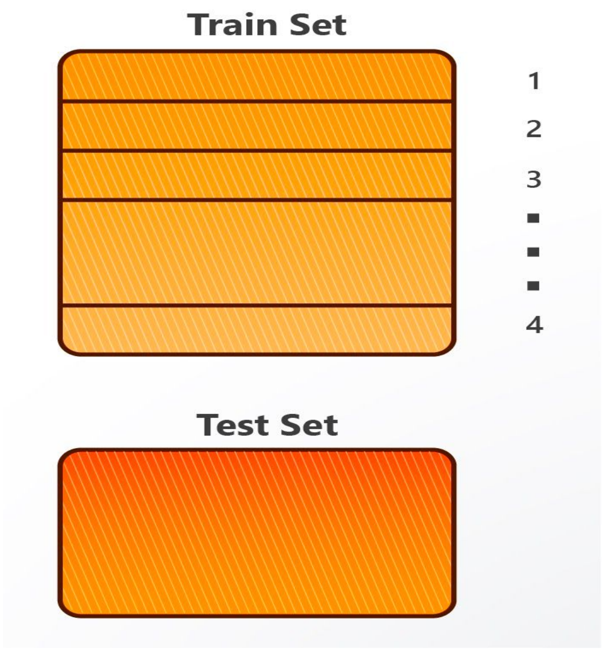
That's exactly what stacking helps us achieve

Definition

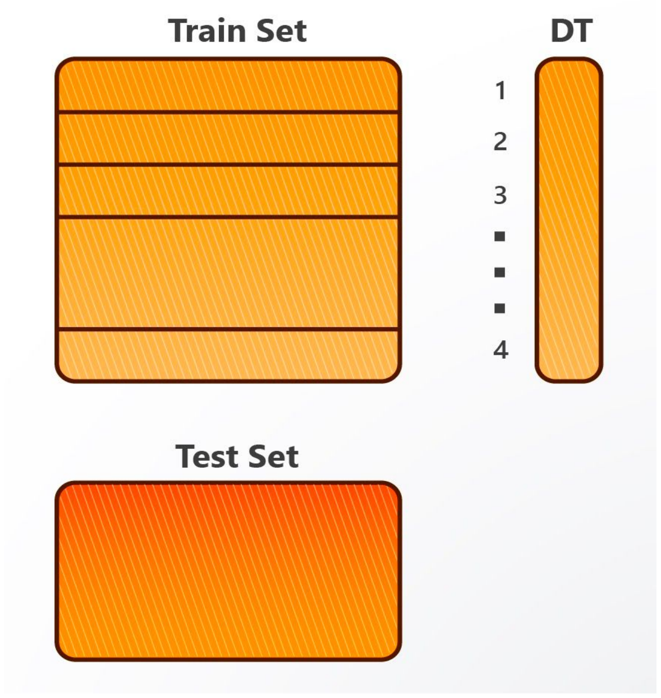
Stacking is an ensemble learning technique to combine multiple classification models via a meta-classifier(fancy name for a 'classifier of classifiers'). It is based on a simple idea: instead of using trivial functions to aggregate the predictions of all predictors in an ensemble, we train a model to perform this aggregation.

Steps of Stacking

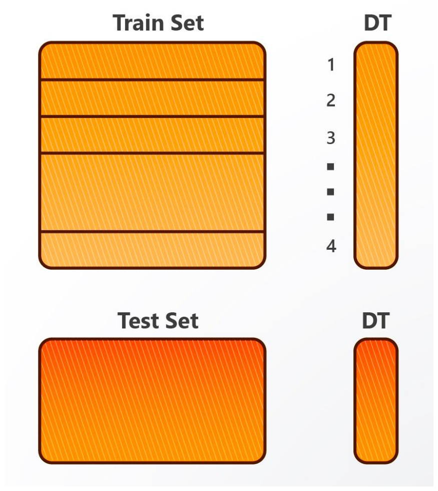
- First, the training set is split in two subsets.



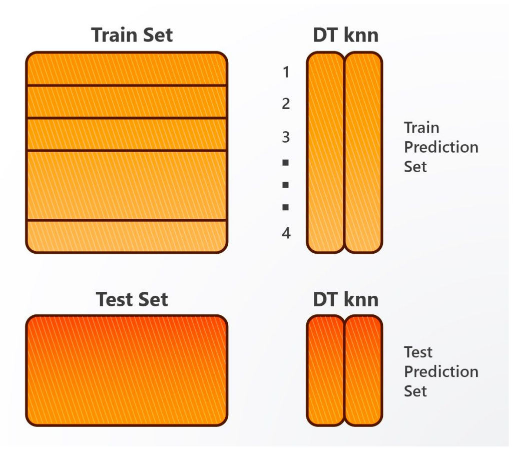
- The first subset is used to train the 'n' models in the first layer



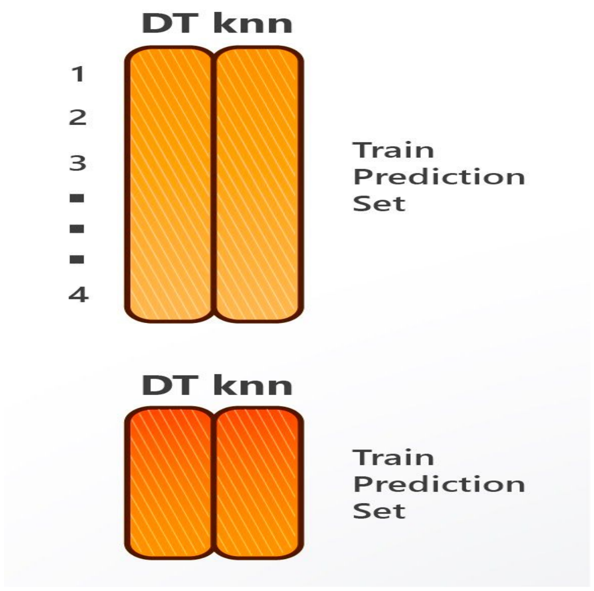
- Next, the first layer models are used to make predictions on the second (held-out) subset

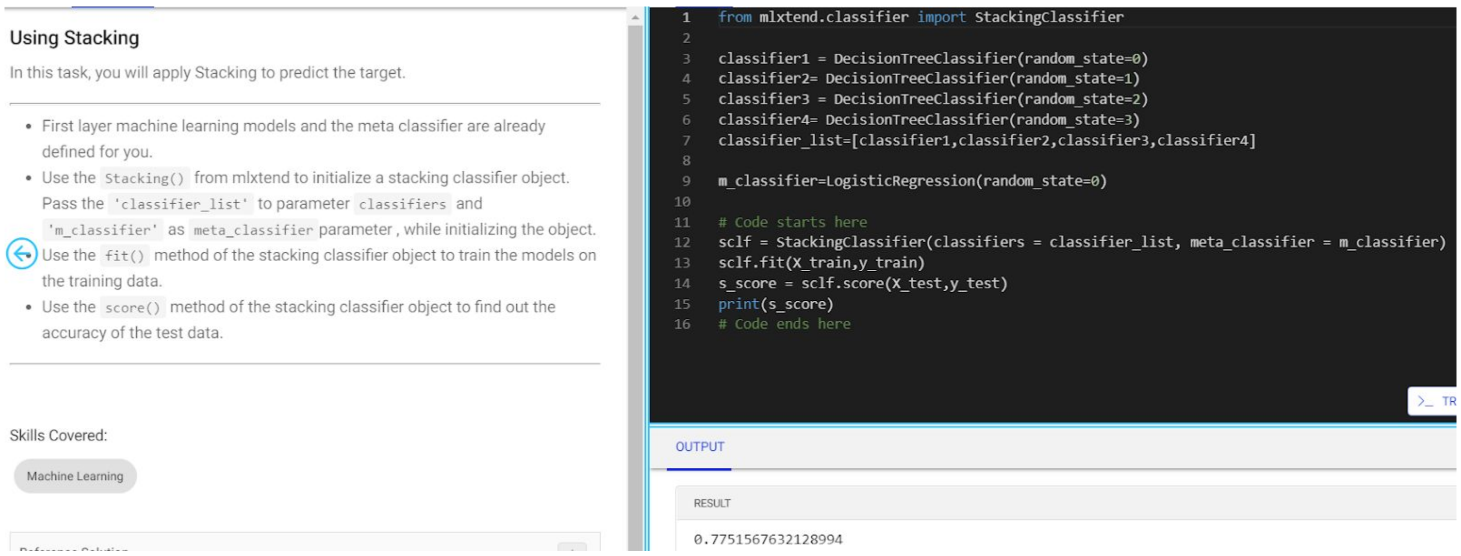


- The predictions of the models are then stored along with the actual predictions as a new training set.



- The meta-classifier is then trained on this new training set, so it learns to predict the target value given the first layer's models.





Using Stacking

In this task, you will apply Stacking to predict the target.

- First layer machine learning models and the meta classifier are already defined for you.
- Use the `Stacking()` from mlxtend to initialize a stacking classifier object. Pass the `'classifier_list'` to parameter `classifiers` and `'m_classifier'` as `meta_classifier` parameter , while initializing the object.
- Use the `fit()` method of the stacking classifier object to train the models on the training data.
- Use the `score()` method of the stacking classifier object to find out the accuracy of the test data.

Skills Covered:

Machine Learning

```python
from mlxtend.classifier import StackingClassifier

classifier1 = DecisionTreeClassifier(random_state=0)
classifier2= DecisionTreeClassifier(random_state=1)
classifier3 = DecisionTreeClassifier(random_state=2)
classifier4= DecisionTreeClassifier(random_state=3)
classifier_list=[classifier1,classifier2,classifier3,classifier4]

m_classifier=LogisticRegression(random_state=0)

# Code starts here
sclf = StackingClassifier(classifiers = classifier_list, meta_classifier = m_classifier)
sclf.fit(X_train,y_train)
s_score = sclf.score(X_test,y_test)
print(s_score)
# Code ends here
```

OUTPUT

RESULT

```
0.7751567632128994
```

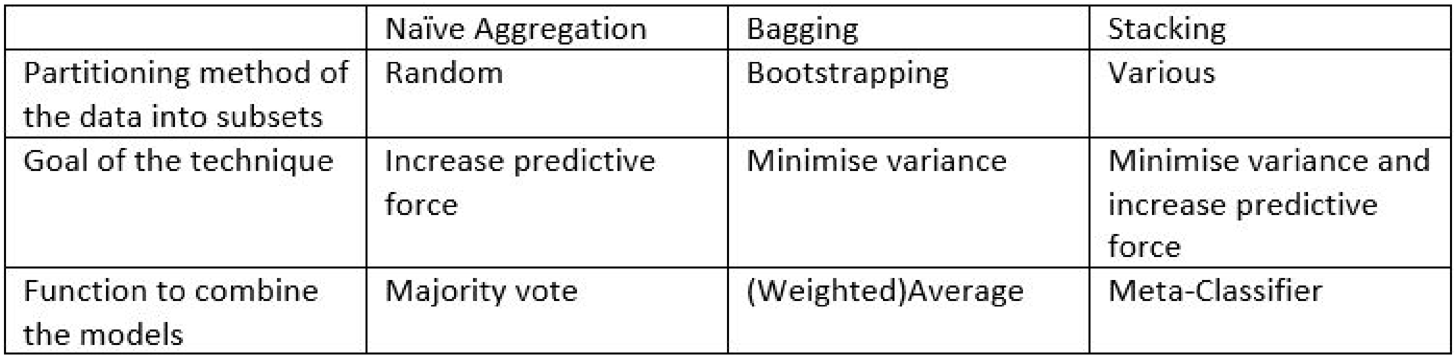
QUESTIONS		YOUR ANSWER	CORRECT ANSWER
1. Which of the following are correct statement(s) about stacking? 1. A machine learning model is trained on predictions of multiple machine learning models 2. A Logistic regression will definitely work better in the second stage as compared to other classification methods 3. First stage models are trained on full / partial feature space of training data	✓	C. 1 and 3	C. 1 and 3

Explanation:

In stacking, you train a machine learning model on predictions of multiple base models. It is not necessary – we can use different algorithms for aggregating the results. First stage models are trained on all the original features.

Phew! That's a lot of ensembling.

Following table is a quick summary of the different ensemble methods we just learned:



	Naïve Aggregation	Bagging	Stacking
Partitioning method of the data into subsets	Random	Bootstrapping	Various
Goal of the technique	Increase predictive force	Minimise variance	Minimise variance and increase predictive force
Function to combine the models	Majority vote	(Weighted)Average	Meta-Classifier

Real Life Use Case

Business Objective:

An educational institution offering certification programs on courses like business analytics, blockchain, software development, digital marketing wanted to upgrade its procedure of selecting candidates for their programs using machine learning. The institution had a dedicated admissions team that would get in touch with candidates who enquired about the courses and then counsel the candidates on the course offerings. The counselling session would then be followed by further back and forth engagement with the candidates until the candidate finally enrolled for the course or chose not to enrol.

Off late the institution was receiving several applications from candidates and it wanted a solution using machine learning capable of automatically screening the candidates and output a probability whether the candidate would enrol for the program or not. In this way, the management could have a prior sense of the likelihood whether a candidate would enrol for the course or not. If the likelihood was high, then the admissions team could get the candidate enrolled within a short span and dedicate more efforts on the candidates having a lower likelihood of getting enrolled.

Solution Methodology:

The institution had the details of candidates whom the admissions team had counselled previously. It consisted of the qualitative and quantitative attributes for every candidate including a binary feature indicating whether the candidate had enrolled or not. This data was to be used for training and validation during machine learning.

A bagging model consisting of several decision trees (i.e. a Random Forest model) with each tree tuned to a different set of hyperparameters was implemented. Every tree model worked on a different subset of features and sample points and the overall model prediction was arrived at by aggregating the predictions of all the individual trees. Since Random forest model could internally rank the features as per their importance to the data, a few of the below insights given by the model were : ·

- Majority of the candidates who enrolled for the business analytics, blockchain and software development courses were engineers whereas a significant number of candidates opting for the digital marketing specialization held an MBA. Hence the stream of graduation of the candidates was important in deciding the enrolment of the candidate.
- · Location was another important feature in deciding the enrolment. The institution offered only classroom-based mode of learning and hence was not able to attract candidates residing overseas & other locations.
- · Work experience of candidates was instrumental in them enrolling for the courses. Majority of the candidates who had enrolled had a work-experience of greater than 2 years.
- · Mode of payment for the courses offered by the institution was another important feature influencing the enrolment.

Business Impact:

The trained model was first tested on a new dataset of 100 candidates captured by the admissions team and the future counselling of these candidates was to be conducted based on the predictions of this model. These predictions helped the admissions team to gather an insight in advance about the enrolment status of the candidates and enabled them to counsel the candidates accordingly and help them to enrol.

The management also decided to work upon the insights given by the model. Since location was a major concern for the candidates, it was decided that an online mode of learning should be made available to the candidates. The management was also of the opinion that freshers should be encouraged more to enrol for the courses. Also on learning that most of the candidates wanted some flexibility in the method of payment, the management wanted to rethink on the fee structure for its programs.

Mars Crater

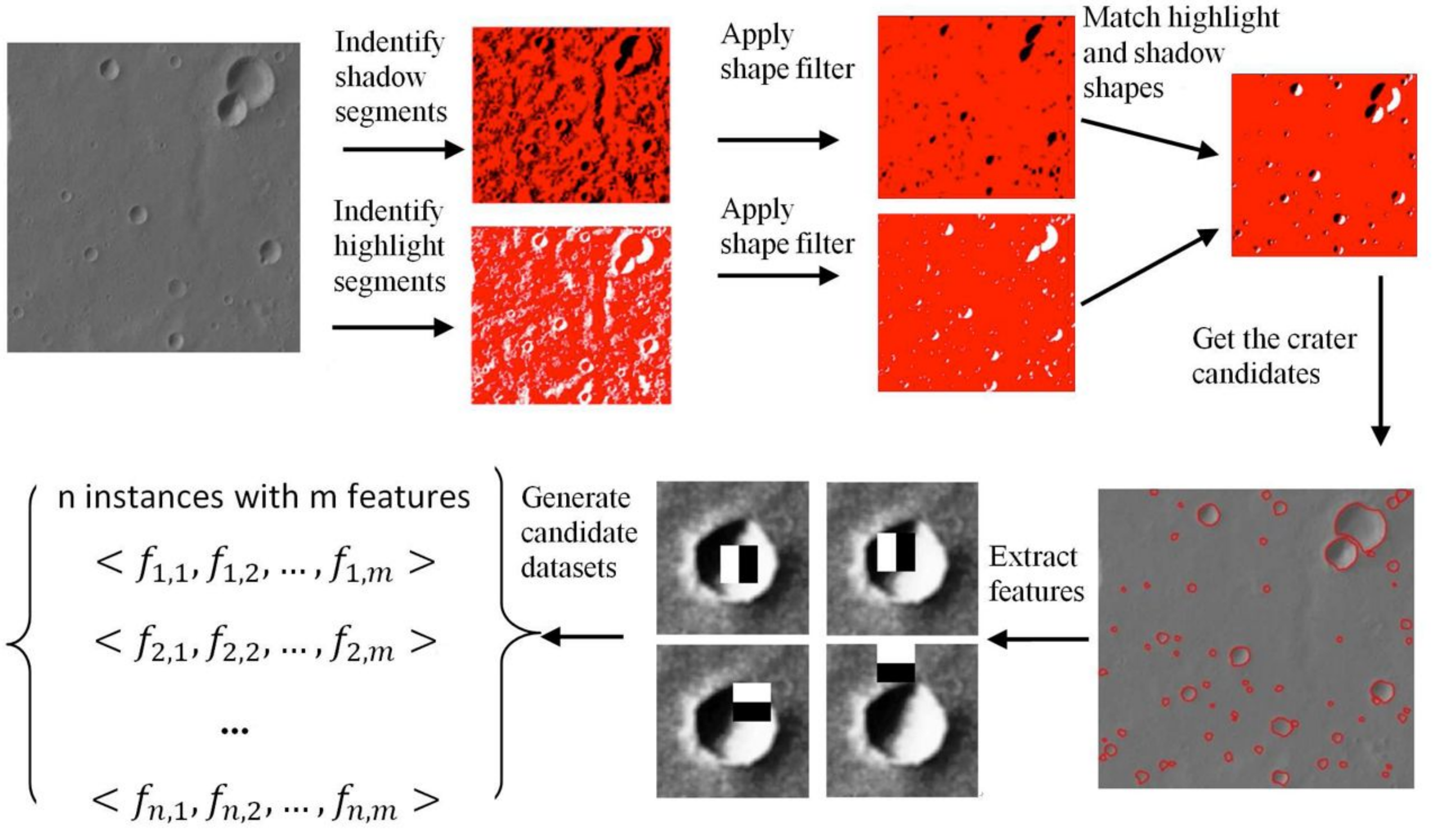
This dataset was generated using HRSC nadir panchromatic image h0905_0000 taken by the Mars Express spacecraft. The images is located in the Xanthe Terra, centered on Nanedi Vallis and covers mostly Noachian terrain on Mars. The image had a resolution of 12.5 meters/pixel.

Problem statement

Determine if the instance is a crater or not a crater. 1=Crater, 0=Not Crater

About the dataset

Using the technique described by L. Bandeira (Bandeira, Ding, Stepinski. 2010.Automatic Detection of Sub-km Craters Using Shape and Texture Information) we identify crater candidates in the image using the pipeline depicted in the figure below. Each crater candidate image block is normalized to a standard scale of 48 pixels. Each of the nine kinds of image masks probes the normalized image block in four different scales of 12 pixels, 24 pixels, 36 pixels, and 48 pixels, with a step of a third of the mask size (meaning 2/3 overlap). We totally extract 1,090 Haar-like attributes using nine types of masks as the attribute vectors to represent each crater candidate. The dataset was converted to the Weka ARFF format by Joseph Paul Cohen in 2012.


Indentify shadow segments
Indentify highlight segments
Apply shape filter
Apply shape filter
Match highlight and shadow shapes
Get the crater candidates
Extract features
Generate candidate datasets
n instances with m features
$< f_{1,1}, f_{1,2}, \dots, f_{1,m} >$
$< f_{2,1}, f_{2,2}, \dots, f_{2,m} >$
...
$< f_{n,1}, f_{n,2}, \dots, f_{n,m} >$

Attribute Information:

We construct a attribute vector for each crater candidate using Haar-like attributes described by Papageorgiou 1998. These attributes are simple texture attributes which are calculated using Haar-like image masks that were used by Viola in 2004 for face detection consisting only black and white sectors. The value of an attribute is the difference between the sum of gray pixel values located within the black sector and the white sector of an image mask. The figure below shows nine image masks used in our case study. The first five masks focus on capturing diagonal texture gradient changes while the remaining four masks on horizontal or vertical textures.

How to read an image ?

Python supports very powerful tools when comes to image processing.Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002. We will use Matplotlib library to convert the image to numpy as array.

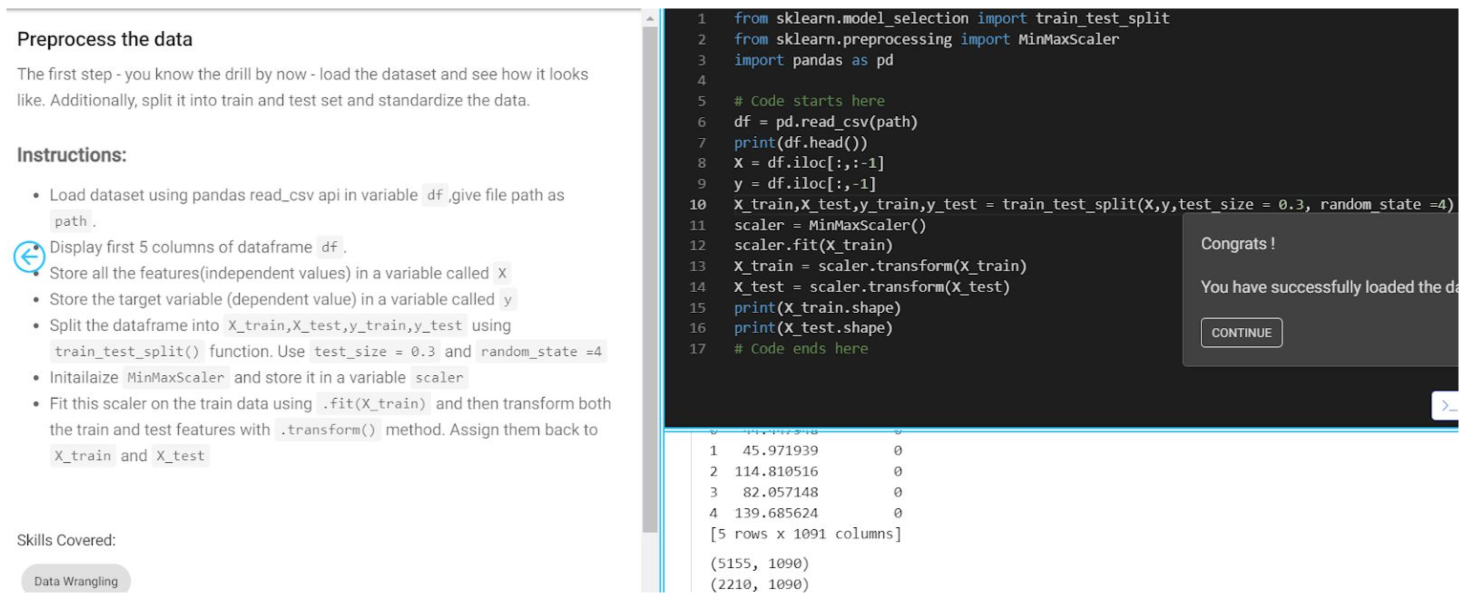
- We import `image` from the `Matplotlib` library as `mpimg`.
- Use `mpimg.imread` to read the image as numpy as array.

** INPUT **

```
import matplotlib.image as mpimg
<div class="w-percent-100 flex-hbox flex-cross-center flex-main-center">
          <div style="width:100%" class="flex-auto">
            <div style="width:100%; max-width:100%; overflow: hidden "><p><img
src="https://storage.googleapis.com/ga-commit-live-prod-live-data/account/b92/11111111-1111-1
111-1111-000000000000/b-43/9301164e-92b3-4f64-b699-737433839cd8/file.png" alt="tile"
/></p></div>
          </div>
        </div>

image = mpimg.imread('crater1.png')
```

** OUTPUT **

```
array([[0.40784314, 0.40784314, 0.40784314, ..., 0.42745098, 0.42745098,
        0.42745098],
       [0.4117647 , 0.4117647 , 0.4117647 , ..., 0.42745098, 0.42745098,
        0.42745098],
       [0.41960785, 0.41568628, 0.41568628, ..., 0.43137255, 0.43137255,
        0.43137255],
       ...,
       [0.4392157 , 0.43529412, 0.43137255, ..., 0.45490196, 0.4509804 ,
        0.4509804 ],
       [0.44313726, 0.44313726, 0.4392157 , ..., 0.4509804 , 0.44705883,
        0.44705883],
       [0.44313726, 0.4509804 , 0.4509804 , ..., 0.44705883, 0.44705883,
        0.44313726]], dtype=float32)
```



Preprocess the 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 and standardize the data.

Instructions:

- Load dataset using pandas read_csv api in variable `df` ,give file path as `path`.
- Display first 5 columns of dataframe `df`.
- Store all the features(independent values) in a variable called `X`
- Store the target variable (dependent value) in a variable called `y`
- Split the dataframe into `X_train,X_test,y_train,y_test` using `train_test_split()` function. Use `test_size = 0.3` and `random_state =4`
- Initailaize `MinMaxScaler` and store it in a variable `scaler`
- Fit this scaler on the train data using `.fit(X_train)` and then transform both the train and test features with `.transform()` method. Assign them back to `X_train` and `X_test`

Skills Covered:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import pandas as pd

# Code starts here
df = pd.read_csv(path)
print(df.head())
X = df.iloc[:,:-1]
y = df.iloc[:,-1]
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size = 0.3, random_state =4)
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
print(X_train.shape)
print(X_test.shape)
# Code ends here
```

Congrats !

You have successfully loaded the da

```
1   45.971939        0
2  114.810516        0
3   82.057148        0
4  139.685624        0
[5 rows x 1091 columns]

(5155, 1090)
(2210, 1090)
```