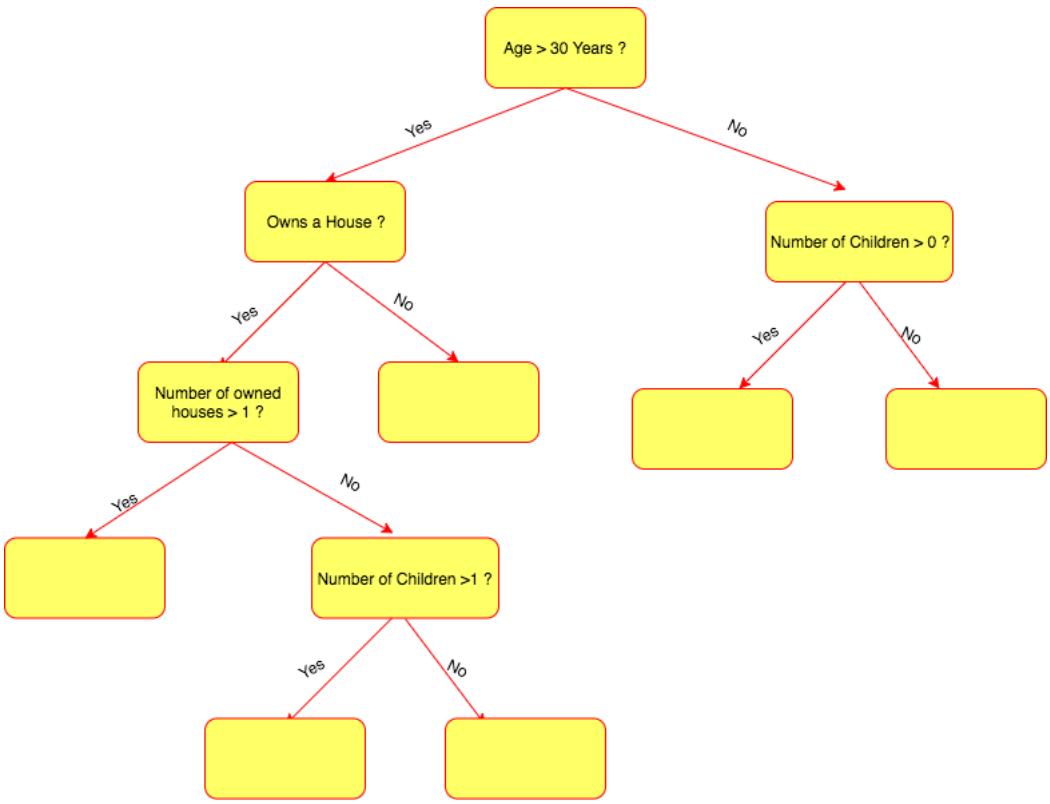
Decision Trees and Random Forests

We will start our discussion with Decision Trees. Decision trees are a hierarchical way of partitioning the data starting with the entire data and recursively partitioning it into smaller parts.

We will cover the following points in our discussion:

- Basic structure of a decision tree
- Building a classification tree (Regression tree is similar with some changes)
- Interpretation in the absence of coefficients
- Decision tree implementation
- Overfitting with decision trees
- Random Forests
- Random Forest implementation

Lets start with a classification example of predicting whether someone will buy an insurance or not. We have been given a set of rules which can be shown as the diagram below:



What we see here is an example of a decision tree. A decision tree is drawn upside down with the root at the top.

Starting from the top, the first question asked is whether the a persons age is greater than 30 years or not. Depending on the answer, a second question is asked, either a person owns a house or not or does the person have 1 or more children and so on. Lets say that the person we consider is 45 years old, then the answer to the first question is Yes and then the next question for this person would be whether this person owns a house or not. Lets say he/she does not own a house. Now we end up in a node where no further questions are asked. This is the **terminal node**. In the terminal node, no further questions are asked. The nodes where we ask questions are the **decision nodes** with the top one being considered as the **parent node**. A thing to note here is that all the questions have binary answers - yes or no.

Now, we know that this person who is 45 years old and does not own a house ends up in one of the terminal nodes - we can say that this person belongs to this bucket. Now, our main question is to predict whether the person with these characteristics will buy the insurance or not.

Before we can answer this question, we need to understand a few more things:

1. **How do we make predictions if some observation ends up in the terminal node?**
2. **Where do these rules come from?**
3. **How do we pick rules to split each node?**
4. **When do we stop splitting a node and consider it as a terminal node?**

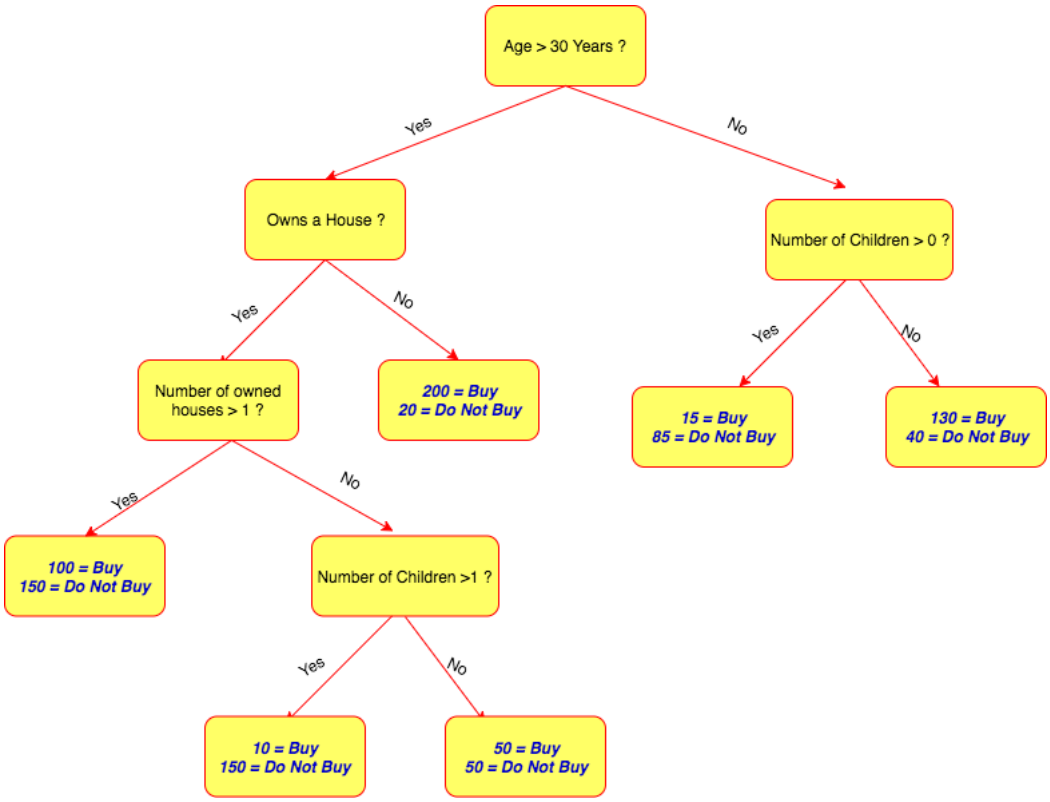
Once we come up with the answers for the questions above, we will have a better idea about decision trees.

1. How do we make predictions if an observation ends up in a terminal node?

The way predictions are made for a classification problem are different than the way they are made for a regression problem. The details are described below.

Classification:

Someone had given us the rules using which we made the decision tree shown above. Instead, we ask this person to give us the information using which he built the tree i.e. share the data using which he/she could come up with the rules. This data can also be referred to as the training data. We took this training data and passed each observation from this data through the decision tree, resulting in the following tree:



Note the terminal nodes now. We can see that among people with age greater than 30 years and who do not own a house, 200 of them bought the insurance and 20 did not buy. Among people with age less than 30 years and who have children, 15 people bought the insurance and 85 did not buy. Using this information present in the the terminal nodes, we can make prediction for new people for whom we do not know the outcome. For example, lets consider a new person comes whose age is greater than 30 years and does not own a house. Using our training data we saw that most of the people who end up in that node buy the insurance. So our prediction will be that this new person will buy the insurance using a simple majority vote. Instead of using the majority vote, we can also consider the probability of this person buying the insurance i.e. 200/220 = 0.9, which is quite high. We can say that if someone ends up in this terminal node, there is a high chance or probability of this person to buy the insurance.

Now lets consider a person who owns 1 house and has no children, then this person ends in a node where 50% of the people buy the insurance and 50% don't. This probability of a person buying the insurance is as random as a coin flip and hence not desirable.

In short, predictions can be made using either of the following:

1. Probability:

   Given by proportions in the terminal nodes

2. Hard classes:
   - Simple majority vote
   - Cutoff on the probability score

Regression:

When the response variable is continuous, in order to make predictions, we take an average of the response values present in the terminal nodes.

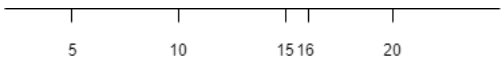
Whether we have a classification decision tree or a regression decision tree, we now know how to make predictions.

But we still don't know how the decision tree was built in the first place. In order to figure this out, we need to answer the next question:

2. Where do these rules come from?

The rules are primarily binary questions. How do we come up with binary questions from numerical and categorical variables?

Numeric variables: For continuous variables, we simply discretize the range and ask questions on the intervals. Lets say we have values 5, 10, 15, 16 and 20 in a variable say 'age' as follows:



Questions like 'is age greater than 10' have an answer either 'yes' or 'no' which covers the entire data. Similarly, the rule 'is age greater than 15' will cover the entire range as well. Binary questions like these can cover the entire data range. Also, it is not necessary that the intervals in which we break the range should be equidistant. For example, in the line above, there is no value present between 10 and 15. The question 'is age greater than 11' and the question 'is age greater than 14' will result in the same partitioning of the data. This is because there are no observations for the variable 'age' in the range 10 to 15. So whatever question we ask between the range 10 to 15 will result in the same partition of the data.

The way a continuous variable is discretized and how the interval questions are decided depends on the kinds of values the continuous variable has.

Categorical variables:

We are already aware how to make dummy variables from categorical predictors. Lets consider the following dummy variables created for the 'City' predictor:

City	var_delhi	var_new_york	var_beijing
delhi	1	0	0
new york	0	1	0
beijing	0	0	1
new york	0	1	0
...	...	...	...
...	...	...	...
delhi	1	0	0

Using the categorical column 'City', we create three dummy variables. In decision trees, an example of a rule is 'Is var_delhi greater than 0.5' which is the same as asking whether it is 0 or 1 i.e if the variables value is greater than 0.5 then the value is delhi else it is either of the other cities.

This is how we get rules from numerical and categorical variables.

3. How to pick rules for splitting at each node?

We have understood how the rules are made for categorical and numerical variables. Given some features, the number of rules can be quite big. How do we choose the best rule amongst these to split a node. The way to pick rules would differ for classification and regression.

Classification:

Lets, for a moment, consider what an ideal decision tree would be like. Which decisions would we be most happy with? The decisions which gives a clear majority or in more specific words, a decision which results in a more homogeneous child node. Lets consider the example of a person who is 45 years old and does not own a home - whether this person will buy the insurance or not. 200 out of the 220 people present in the terminal node end up buying the insurance. Hence, if we get another person with similar characteristics, we can reasonably predict that that person would buy the insurance. Similarly, if a person ends up in a terminal node in which 10 out of 170 people usually end up buying the insurance, then we can be reasonably certain that this person will not buy the insurance. Both these cases are preferred since they result in a more homogeneous terminal node.

However, we would not be certain whether a person will end up buying the insurance if 50 out of 100 people ending up in the terminal node buy the insurance i.e. 50% of the people can either buy or not buy the insurance. The decision is as good as a coin flip.

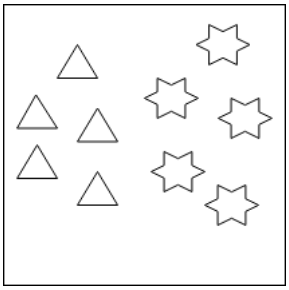
We would want our nodes to have as clear majority as possible i.e. the terminal nodes should be as homogeneous as possible.

There are different mathematical measures that quantify this homogeneity:

- Gini Index
- Entropy
- Deviance

Lets go through each of these measures of homogeneity. Lower the values of each of these measures, higher is the homogeneity of the node.

Consider the following diagram:



Gini Index: To calculate the Gini Index we use the following:

$$Gini\ Index = 1 - \sum_{i=1}^{k} p_i^2$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$

Referring to the diagram above, the Gini Index will be computed as follows:

$$Gini\ Index = 1 - \Big[(\frac{5}{10})^2\ +\ (\frac{5}{10})^2\Big]\ =\ 0.5$$

Entropy: We calculate entropy as follows:

$$Entropy = -\sum_{i=1}^{k} p_i * log(p_i)$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$

Referring to the diagram above, the Entropy can be computed as follows:

$$Entropy = -\Big[(\frac{5}{10}) * log(\frac{5}{10})\ +\ (\frac{5}{10}) * log(\frac{5}{10})\Big]\ =\ 0.6931472$$

Deviance: We calculate deviance as follows:

$$Deviance = -\sum_{i=1}^{k} n_i * log(p_i)$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$
$$and\ n\ = no.\ of\ obs.\ in\ a\ class$$

Referring to the diagram above, we compute the deviance of that node as follows:

$$Deviance = -\Big[5 * log(\frac{5}{10})\ +\ 5 * log(\frac{5}{10})\Big]\ =\ 6.931472$$

How do these measures quantify homogeneity?

An important property of all these measures is that their value will be lowest when any of the classes have a clear majority.

In case where all the observations in a node belong to the same class, then each of the measures described above would have the value of 0.

For implementation, we can use any of the measures described above; there is no theoretical favorite.

For a detailed example of rule selection for classification, you may refer to the class presentation.

Regression:

We know that Gini Index, Entropy and Deviance are used to select rules for classification. But if the response variable is continuous, how do we select the rules?

In case of regression, the prediction is an average of the node. In order to measure how good our predictions are, we compute the error sum of squares (SSE).

$$SSE = \sum_{i=1}^{n_k} (y_i - y_k)^2$$
$$where\ n_k = \ no.\ of\ obs.\ in\ each\ node$$
$$and\ y_i = \ value\ of\ each\ obs.$$
$$and\ y_k = \ mean\ of\ the\ node$$

You may refer to the class presentation for a detailed example.

We choose the rules based on lower SSE; lower the SSE, more homogeneous is the node.

4. When do we stop splitting the nodes?

There are two natural criteria to stop splitting the nodes:

- Node is completely homogeneous - in case of classification, all observations belong to a single class and in case of regression, all the observations have the same value
- The terminal node is left with a single observation

If we let our tree grow too big, there are higher chances of overfitting the data. It might be that the model is too perfect for the training data but does not generalize well.

In case we wish to stop growing a tree before they reach their natural stopping criterion, we can use hyperparameters to control the size of the decision tree.

Following are some of the hyperparameters to tune:

- max_depth: This fixes the maximum depth of the tree. If it is not set then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_sample_split: The minimum number of samples required to split an internal node; default value - 2. It is a good idea to keep it slightly higher in order to reduce overfitting of the data.
- min_sample_leaf: The minimum number of samples required to be in each leaf node on splitting their parent node. This defaults to 1. If this number is higher and a split results in a leaf node having lesser number of samples than specified then that split is cancelled.
- max_leaf_nodes: It defines the maximum number of possible leaf nodes. If it is not set then it takes an unlimited number of leaf nodes. This parameter controls the size of the tree.

Decision Tree Implementation:

We will start with building our decision tree now.

```
import pandas as pd
import numpy as np
from sklearn import tree
import numpy as np
from sklearn.metrics import roc_auc_score
import matplotlib as plt
%matplotlib inline
```

We will consider the demographic data 'census_income.csv' for this module. This is typical census data. This data has been labeled with annual income less than 50K dollars or not. We want to build a model such that given these census characteristics we can figure out if someone will fall in a category in which their income is higher than 50K dollars or not. Such models are mainly used when formulating government policies.

```
# data prep from previous module
file=r'/Users/anjal/Dropbox/0.0 Data/census_income.csv'
ci_train=pd.read_csv(file)
```

```
ci_train.head()
```

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	race	sex	capital.g
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0

Lets start with the data preparation steps.

We know that there should not be any redundancy in the data. e.g. consider the 'education' and 'education_num' variables.

```
pd.crosstab(ci_train['education'],ci_train['education.num'])
```

education.num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
education																
10th	0	0	0	0	0	933	0	0	0	0	0	0	0	0	0	0
11th	0	0	0	0	0	0	1175	0	0	0	0	0	0	0	0	0
12th	0	0	0	0	0	0	0	433	0	0	0	0	0	0	0	0
1st-4th	0	168	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5th-6th	0	0	333	0	0	0	0	0	0	0	0	0	0	0	0	0
7th-8th	0	0	0	646	0	0	0	0	0	0	0	0	0	0	0	0
9th	0	0	0	0	514	0	0	0	0	0	0	0	0	0	0	0
Assoc-acdm	0	0	0	0	0	0	0	0	0	0	0	1067	0	0	0	0
Assoc-voc	0	0	0	0	0	0	0	0	0	0	1382	0	0	0	0	0
Bachelors	0	0	0	0	0	0	0	0	0	0	0	0	5355	0	0	0
Doctorate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	413
HS-grad	0	0	0	0	0	0	0	0	10501	0	0	0	0	0	0	0
Masters	0	0	0	0	0	0	0	0	0	0	0	0	0	1723	0	0
Preschool	51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Prof-school	0	0	0	0	0	0	0	0	0	0	0	0	0	0	576	0
Some-college	0	0	0	0	0	0	0	0	0	7291	0	0	0	0	0	0

We observe that there is one to one correspondence. e.g. for 'education' 10th has been labelled as 'education.num' 6, 11th has been labelled as 'education.num' 7 etc. Hence, instead of using the variable 'education' we can use 'education.num' only. We will go ahead and drop 'education' here.

```
ci_train.drop(['education'],axis=1,inplace=True)
```

Now lets have a look at our outcome variable 'Y'.

```
ci_train['Y'].value_counts().index
```

```
Index([' <=50K', ' >50K'], dtype='object')
```

Notice that it has whitespace values which should be removed. During data preparation we need to be careful with this else when comparing these values we may get unexpected results if the whitespace is not considered.

Now we convert the response column 'Y' to 1's and 0's.

```
ci_train['Y']=(ci_train['Y']==' >50K').astype(int)
```

For the remaining categorical columns we will need to make dummies.

```
cat_cols=ci_train.select_dtypes(['object']).columns
```

```
cat_cols
```

```
Index(['workclass', 'marital.status', 'occupation', 'relationship', 'race',
       'sex', 'native.country'],
      dtype='object')
```

When making dummies, we will ignore those values which have a frequency less than 500. You can always reduce this number and check if more dummies result in a better model.

```
for col in cat_cols:
    freqs=ci_train[col].value_counts()
    k=freqs.index[freqs>500][:-1]
    for cat in k:
        name=col+'_'+cat
        ci_train[name]=(ci_train[col]==cat).astype(int)
    del ci_train[col]
    print(col)
```

```
workclass
marital.status
occupation
relationship
race
sex
native.country
```

The above steps result in my data having 32561 rows and 39 columns including the response variable.

```
ci_train.shape
```

```
(32561, 39)
```

Next we will need to check for missing values.

```
ci_train.isnull().sum()
```

```
age                                0
fnlwgt                             0
education.num                      0
capital.gain                       0
capital.loss                       0
hours.per.week                     0
Y                                  0
workclass_ Private                 0
workclass_ Self-emp-not-inc        0
workclass_ Local-gov               0
workclass_ ?                       0
workclass_ State-gov               0
workclass_ Self-emp-inc            0
marital.status_ Married-civ-spouse 0
marital.status_ Never-married      0
marital.status_ Divorced           0
marital.status_ Separated          0
occupation_ Prof-specialty         0
occupation_ Craft-repair           0
occupation_ Exec-managerial        0
occupation_ Adm-clerical           0
occupation_ Sales                  0
occupation_ Other-service          0
occupation_ Machine-op-inspct      0
occupation_ ?                      0
occupation_ Transport-moving       0
occupation_ Handlers-cleaners      0
occupation_ Farming-fishing        0
occupation_ Tech-support           0
relationship_ Husband              0
relationship_ Not-in-family        0
relationship_ Own-child            0
relationship_ Unmarried            0
relationship_ Wife                 0
race_ White                        0
race_ Black                        0
sex_ Male                          0
native.country_ United-States      0
native.country_ Mexico             0
dtype: int64
```

As we can observe, there are no missing values in the data.

Next, we separate the predictors and the response variable.

```
x_train=ci_train.drop(['Y'],1)
y_train=ci_train['Y']
```

The data preparation steps are similar for all models. Here we are given only the training dataset. However, if we are given a test dataset also, we should do the data preparation steps for both training and test datasets.

Hyper Parameters For Decision Trees

- criterion: Used to set which homogeneity measure to use. Two options available: "entropy" and "gini" (default).
- max_depth: This fixes the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if 'max_leaf_nodes' is not None.
- min_sample_split: The minimum number of samples required to split an internal node; default value - 2. It is a good idea to keep it slightly higher in order to reduce overfitting of the data. Recommended values are between 5 to 10.
- min_sample_leaf: The minimum number of samples required to be in each leaf node on splitting their parent node. This defaults to 1. If this number is higher and a split results in a leaf node having lesser number of samples than specified then that split is cancelled.
- max_leaf_nodes: It defines the maximum number of possible leaf nodes. If None then it takes an unlimited number of leaf nodes. By default, it takes “None” value. This parameter controls the size of the tree. We will be finding optimal value of this through cross validation.
- class_weight: Default is None, in which case each class is given equal weightage. If the goal of the problem is good classification instead of accuracy (especially in the case of imbalanced datasets) then you should set this to "balanced", in which case class weights assigned are inversely proportional to class frequencies in the input data.
- random_state: Used to reproduce random result.

For selecting the best parameters we will use RandomizedSearchCV instead of GridSearchCV.

Why should we consider using Randomized Search instead of Grid Search?

The variable 'params' below consists of 5 different parameters we intend to tune. Each of these parameters contain some values. The possible combinations will be 960 as shown below. If we use grid search and use a 10 fold CV, we will build around 9600 individual trees. It will result in the best possible combination but will also take a lot of time. In order to handle this, instead of trying out all 960 combinations, we can try only 10% of these combinations i.e 96 combinations. Randomized Search will randomly select 96 of the combinations and will result in good enough results - though we do not have a guarantee of the best combination. It will give us a good enough combination at a fraction of the runtime. The tradeoff is between the run time and how good our result will be.

We can make the Randomized Search better by running it multiple times (each time we get a different combination) and check whether the combinations are consistent across different runs. We can expand in the neighbourhood values as well e.g we get max_depth as 70 using Randomized Search; in the next run we would want to add 80 and 90 and check again. Another example would be if we get max_depth as 30, but we did not consider any other value between 30 and 50. We may want to add a max_depth of 40 and try again.

Once we get out best max_depth value as 30, we can also try values around 30, like 25, 26, 31, 32 etc which may result in better performance of the model.

```
2*2*8*6*5
```

```
960
```

```
# RandomSearchCV/GridSearchCV accept parameters values as dictionaries.
# In example given below we have constructed dictionary for different parameter values that we want to
# try for decision tree model
params={ 'class_weight':[None,'balanced'],
         'criterion':['entropy','gini'],
         'max_depth':[None,5,10,15,20,30,50,70],
         'min_samples_leaf':[1,2,5,10,15,20],
         'min_samples_split':[2,5,10,15,20]
       }
```

Now lets see how RandomizedSearchCV works. It works just like GridSearchCV except that we need to tell RandomizedSearchCV that out of all these combinations how many do we want to try out. e.g. We may mention that out of 960 combinations only try out 10. We use the argument n_iter to set this. Ideally, we should try about 10 to 20% of all the combinations.

```
from sklearn.model_selection import RandomizedSearchCV
```

The classifier we want to try to do this classification is the Decision Tree Classifier.

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

```
clf=DecisionTreeClassifier()
```

```
# We try the decision tree classifier here supplying different parameter ranges to our randomSearchCV which
in turn will pass it on to this classifier
# n_iter parameter - this number should be 10 to 20% of the total number of combinations
# n_iter parameter of RandomizedSeacrhCV controls how many parameter combinations of all given combinations
will be tried

random_search=RandomizedSearchCV(clf, cv=10, param_distributions=params,
             scoring='roc_auc',n_iter=10, n_jobs=-1, verbose=20)
```

- In the code above, we use the object 'clf' of the DecisionTreeClassifier.
- cv=10 refers to using a 10 fold cross validation.
- We use 'scoring='roc_auc'' since its a binary classification problem. In case of multi-class classification, we would use 'accuracy' instead.
- n_iter=10 states that we will be considering 10 parameter combinations.
- n_jobs=-1 indicates that the code can use all the available CPU cores present in the system, helping us take advantage of multicore processing.
- verbose=20 is used to display detailed logging information.

```
random_search
```

```
RandomizedSearchCV(cv=10, error_score='raise-deprecating',
          estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          fit_params=None, iid='warn', n_iter=10, n_jobs=-1,
          param_distributions={'class_weight': [None, 'balanced'], 'criterion': ['entropy', 'gini'],
'max_depth': [None, 5, 10, 15, 20, 30, 50, 70], 'min_samples_leaf': [1, 2, 5, 10, 15, 20],
'min_samples_split': [2, 5, 10, 15, 20]},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', scoring='roc_auc', verbose=20)
```

```
random_search.fit(x_train,y_train)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:    8.4s
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:    8.4s
[Parallel(n_jobs=-1)]: Done   3 tasks      | elapsed:    8.4s
[Parallel(n_jobs=-1)]: Done   4 tasks      | elapsed:    8.4s
[Parallel(n_jobs=-1)]: Done   5 tasks      | elapsed:    9.1s
[Parallel(n_jobs=-1)]: Done   6 tasks      | elapsed:    9.1s
[Parallel(n_jobs=-1)]: Done   7 tasks      | elapsed:    9.1s
[Parallel(n_jobs=-1)]: Done   8 tasks      | elapsed:    9.1s
[Parallel(n_jobs=-1)]: Done   9 tasks      | elapsed:    9.7s
[Parallel(n_jobs=-1)]: Done  10 tasks      | elapsed:    9.8s
[Parallel(n_jobs=-1)]: Done  11 tasks      | elapsed:   10.0s
[Parallel(n_jobs=-1)]: Done  12 tasks      | elapsed:   10.0s
[Parallel(n_jobs=-1)]: Done  13 tasks      | elapsed:   10.8s
[Parallel(n_jobs=-1)]: Done  14 tasks      | elapsed:   10.9s
[Parallel(n_jobs=-1)]: Done  15 tasks      | elapsed:   11.1s
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:   11.1s
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:   12.2s
[Parallel(n_jobs=-1)]: Done  18 tasks      | elapsed:   12.2s
[Parallel(n_jobs=-1)]: Done  19 tasks      | elapsed:   12.4s
[Parallel(n_jobs=-1)]: Done  20 tasks      | elapsed:   12.4s
[Parallel(n_jobs=-1)]: Done  21 tasks      | elapsed:   13.1s
[Parallel(n_jobs=-1)]: Done  22 tasks      | elapsed:   13.1s
[Parallel(n_jobs=-1)]: Done  23 tasks      | elapsed:   13.3s
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:   13.4s
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:   13.8s
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:   13.9s
[Parallel(n_jobs=-1)]: Done  27 tasks      | elapsed:   14.0s
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:   14.1s
[Parallel(n_jobs=-1)]: Done  29 tasks      | elapsed:   14.6s
[Parallel(n_jobs=-1)]: Done  30 tasks      | elapsed:   14.6s
[Parallel(n_jobs=-1)]: Done  31 tasks      | elapsed:   14.9s
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed:   14.9s
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed:   15.4s
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:   15.5s
[Parallel(n_jobs=-1)]: Done  35 tasks      | elapsed:   15.7s
[Parallel(n_jobs=-1)]: Done  36 tasks      | elapsed:   15.7s
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:   16.3s
```

```
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed:   16.3s
[Parallel(n_jobs=-1)]: Done  39 tasks      | elapsed:   16.6s
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed:   16.6s
[Parallel(n_jobs=-1)]: Done  41 tasks      | elapsed:   17.0s
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   17.0s
[Parallel(n_jobs=-1)]: Done  43 tasks      | elapsed:   17.3s
[Parallel(n_jobs=-1)]: Done  44 tasks      | elapsed:   17.3s
[Parallel(n_jobs=-1)]: Done  45 tasks      | elapsed:   17.7s
[Parallel(n_jobs=-1)]: Done  46 tasks      | elapsed:   17.7s
[Parallel(n_jobs=-1)]: Done  47 tasks      | elapsed:   18.0s
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:   18.0s
[Parallel(n_jobs=-1)]: Done  49 tasks      | elapsed:   18.4s
[Parallel(n_jobs=-1)]: Done  50 tasks      | elapsed:   18.4s
[Parallel(n_jobs=-1)]: Done  51 tasks      | elapsed:   18.6s
[Parallel(n_jobs=-1)]: Done  52 tasks      | elapsed:   18.7s
[Parallel(n_jobs=-1)]: Done  53 tasks      | elapsed:   19.1s
[Parallel(n_jobs=-1)]: Done  54 tasks      | elapsed:   19.1s
[Parallel(n_jobs=-1)]: Done  55 tasks      | elapsed:   19.3s
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed:   19.3s
[Parallel(n_jobs=-1)]: Done  57 tasks      | elapsed:   19.7s
[Parallel(n_jobs=-1)]: Done  58 tasks      | elapsed:   19.8s
[Parallel(n_jobs=-1)]: Done  59 tasks      | elapsed:   19.9s
[Parallel(n_jobs=-1)]: Done  60 tasks      | elapsed:   19.9s
[Parallel(n_jobs=-1)]: Done  61 tasks      | elapsed:   20.5s
[Parallel(n_jobs=-1)]: Done  62 tasks      | elapsed:   20.6s
[Parallel(n_jobs=-1)]: Done  63 tasks      | elapsed:   20.8s
[Parallel(n_jobs=-1)]: Done  64 tasks      | elapsed:   20.8s
[Parallel(n_jobs=-1)]: Done  65 tasks      | elapsed:   21.3s
[Parallel(n_jobs=-1)]: Done  66 tasks      | elapsed:   21.4s
[Parallel(n_jobs=-1)]: Done  67 tasks      | elapsed:   21.5s
[Parallel(n_jobs=-1)]: Done  68 tasks      | elapsed:   21.6s
[Parallel(n_jobs=-1)]: Done  69 tasks      | elapsed:   21.8s
[Parallel(n_jobs=-1)]: Done  70 tasks      | elapsed:   21.9s
[Parallel(n_jobs=-1)]: Done  71 tasks      | elapsed:   22.1s
[Parallel(n_jobs=-1)]: Done  72 tasks      | elapsed:   22.1s
[Parallel(n_jobs=-1)]: Done  73 tasks      | elapsed:   22.2s
[Parallel(n_jobs=-1)]: Done  74 tasks      | elapsed:   22.2s
[Parallel(n_jobs=-1)]: Done  75 tasks      | elapsed:   22.4s
[Parallel(n_jobs=-1)]: Done  76 tasks      | elapsed:   22.4s
[Parallel(n_jobs=-1)]: Done  77 tasks      | elapsed:   22.5s
[Parallel(n_jobs=-1)]: Done  78 tasks      | elapsed:   22.6s
[Parallel(n_jobs=-1)]: Done  79 tasks      | elapsed:   22.7s
[Parallel(n_jobs=-1)]: Done  80 tasks      | elapsed:   22.7s
[Parallel(n_jobs=-1)]: Done  81 tasks      | elapsed:   23.2s
[Parallel(n_jobs=-1)]: Done  82 tasks      | elapsed:   23.2s
[Parallel(n_jobs=-1)]: Done  83 tasks      | elapsed:   23.4s
[Parallel(n_jobs=-1)]: Done  84 tasks      | elapsed:   23.4s
[Parallel(n_jobs=-1)]: Done  85 tasks      | elapsed:   23.9s
[Parallel(n_jobs=-1)]: Done  86 tasks      | elapsed:   24.0s
[Parallel(n_jobs=-1)]: Done  87 tasks      | elapsed:   24.1s
[Parallel(n_jobs=-1)]: Done  88 tasks      | elapsed:   24.1s
[Parallel(n_jobs=-1)]: Done  89 tasks      | elapsed:   24.6s
[Parallel(n_jobs=-1)]: Done  90 tasks      | elapsed:   24.6s
[Parallel(n_jobs=-1)]: Done  91 tasks      | elapsed:   24.8s
[Parallel(n_jobs=-1)]: Done  92 tasks      | elapsed:   24.8s
[Parallel(n_jobs=-1)]: Done  93 tasks      | elapsed:   25.2s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:   26.3s finished
```

```
RandomizedSearchCV(cv=10, error_score='raise-deprecating',
          estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          fit_params=None, iid='warn', n_iter=10, n_jobs=-1,
          param_distributions={'class_weight': [None, 'balanced'], 'criterion': ['entropy', 'gini'],
'max_depth': [None, 5, 10, 15, 20, 30, 50, 70], 'min_samples_leaf': [1, 2, 5, 10, 15, 20],
'min_samples_split': [2, 5, 10, 15, 20]},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', scoring='roc_auc', verbose=20)
```

In the output above, we can see all the parameters that were tried out.

We get the best estimator as follows:

```
random_search.best_estimator_
```

```
DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
            max_depth=10, max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=10, min_samples_split=15,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

The report function below gives the rank-wise details of each model.

```
# Utility function to report best scores. This simply accepts grid scores from our
randomSearchCV/GridSearchCV and picks
# and gives top few combination according to their scores.

def report(results, n_top=3):
    for i in range(1, n_top + 1):
        # np.flatnonzero extracts index of `True` in a boolean array
        candidate = np.flatnonzero(results['rank_test_score'] == i)[0]
        # print rank of the model
        # values passed to function format here are put in the curly brackets when printing
        # 0 , 1 etc refer to placeholder for position of values passed to format function
        # .3f means upto 2 decimal digits
        print("Model with rank: {0}".format(i))
        # this prints cross validated performance and its standard deviation
        print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
              results['mean_test_score'][candidate],
              results['std_test_score'][candidate]))
        # prints the paramter combination for which this performance was obtained
        print("Parameters: {0}".format(results['params'][candidate]))
        print("")
```

Below we can see the details of the top 5 models. The best model has the auc_roc score of 0.894 and we can check the parameters that resulted in this.

```
report(random_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: 0.89473 (std: 0.00684)
Parameters: {'min_samples_split': 10, 'min_samples_leaf': 15, 'max_depth': 15, 'criterion': 'gini',
'class_weight': 'balanced'}

Model with rank: 2
Mean validation score: 0.89388 (std: 0.00668)
Parameters: {'min_samples_split': 20, 'min_samples_leaf': 20, 'max_depth': 30, 'criterion': 'gini',
'class_weight': None}

Model with rank: 3
Mean validation score: 0.89069 (std: 0.00572)
Parameters: {'min_samples_split': 20, 'min_samples_leaf': 10, 'max_depth': 5, 'criterion': 'gini',
'class_weight': 'balanced'}

Model with rank: 4
Mean validation score: 0.88994 (std: 0.00735)
Parameters: {'min_samples_split': 15, 'min_samples_leaf': 15, 'max_depth': 20, 'criterion': 'gini',
'class_weight': 'balanced'}

Model with rank: 5
Mean validation score: 0.88210 (std: 0.00643)
Parameters: {'min_samples_split': 20, 'min_samples_leaf': 1, 'max_depth': 15, 'criterion': 'gini',
'class_weight': None}
```

We will now fit the best estimator separately.

```
dtree=random_search.best_estimator_
dtree
```

```
DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
            max_depth=10, max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=10, min_samples_split=15,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

```
dtree.fit(x_train,y_train)
```

```
DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
            max_depth=10, max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=10, min_samples_split=15,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

The tentative performance of the model is 0.894. We can now use this model to make predictions on the test data using either the predict() or predict_proba() functions.

In order to visualize the decision tree we need to do the following:

```
dotfile = open("mytree.dot", 'w')
tree.export_graphviz(dtree,out_file=dotfile, feature_names=x_train.columns, class_names=["0","1"],
proportion=True)
dotfile.close()
```

We first need to output the decision tree to a dot file using the export_graphviz() function. 'dtree' is the model we built, 'out_file' is where we will write this decision tree, 'feature_names' are names of the features on which the rules are based, 'class_names' stores the classes and 'proportions' set to True means that we want to see the proportions in the nodes too.

Open mytree.dot file in a simple text editor and copy and paste the code at http://webgraphviz.com to visualise the tree.

We discussed only about Classification using Decision Trees.

As far as Decision Trees for Regression are concerned there will be the following differences: we use a DecisionTreeRegressor instead of DecisionTreeClassifier to create the Decision Tree object. The arguments: "class_weight" and "criterion" will not make sense in case of Regression. Also, when using Regression, the evaluation criterion will be 'neg_mean_absolute_error' instead of 'roc_auc' score.

Rest of the process remains identical for Decision Tree Regressor and Decision Tree Classifier.

Next we will discuss the issues with Decision Trees and how is the issue handled.

Random Forests

Issues with decision trees: Decision trees help capture niche non-linear patterns in data. The model may fit the training data very well but may not do well with newer data. Whenever we build a model, we want it to do well with future data too and hence needs to be as generalizable as possible. How do we handle this? On one hand we want to capture decision tree's amazing ability to capture non-linear data and at the same time we do not want it to be susceptible to noise or niche patterns from the training data.

A very simple and powerful idea resulted in the popular algorithm called Random Forest.

Random Forests introduce two levels of randomness in the tree building process which help in handling the noise in the data.

- Random Forest takes a random subset of all the observations present in training dataset
- It also takes a random subset of all the variables from the training dataset

In order to understand this idea, lets assume that we have 10000 observations and 200 variables in the dataset. In Random Forest, many decision trees are built instead of a single tree; for our example lets consider 500 trees are built. Now, if we build these trees using the same 10000 observations 500 times we will end up with the same 500 trees. But this is not what we want since we will get the same outcome from each of these 500 trees. Hence, each tree in the Random Forest does not use the entire data, but instead the individual trees are built on random subset of the observations (we can consider 10000 sample observations with replacement or fewer sample observations without replacement). In short, in the first level of randomness, instead of using the entire data, we use a random sample of the data. How does this help in cancelling out the effect of noise/niche patterns specific to the training data? By definition, noise will be a small chuck of the data. Hence, when we sample observations from the training dataset repeatedly for our 500 trees, we can safely assume that a majority of the trees built will not be affected by noise.

Now, how does taking a random subset of all the variables from the dataset help? When building a single decision tree (not in Random Forest), in order to choose a rule to split a node, the decision tree algorithm considers all the variables i.e. in our example all 200 variables will be considered. In Random Forest algorithm, on the other hand, in order to choose a rule to split a node, instead of considering all possible variables, the algorithm considers only a random subset of variables. Lets say only 20 variables are randomly picked up from the 200 variables present and only the rules generated from these 20 variables are considered when choosing the best rule to split the node. At every node a fresh random subset of variables is selected from which the best rule is picked up. How does this help in cancelling out the effect of noisy variables? The noisy variables will be a small chuck of all the variables present and when the variables are randomly selected, the chances of the noisy variables being selected are low. This in turn will not always let the noisy variables affect all the 500 individual trees made. The noisy variables may affect some of the trees a lot, but since we will take a majority over 500 trees, their effect will be minimized.

In short, the first randomness removes the effect of noisy observations and the second randomness removes the effect of noisy variables.

The final predictions made by Random Forest model will be a majority vote from all the trees in the forest in case of classification. For Regression, the predictions will be the average of predictions made by the 500 trees.

Implementation of Random Forest

Now we will build a Random Forest model. We will use the same hyperparameters as for decision trees amongst others since Random Forest is ultimately a collection of decision trees.

Additional hyperparameters for Random Forests

- n_estimators: Number of trees to be built in the forest - default: 10; good starting point can be 100.
- max_features: Number of features being considered for selecting the best rule at each split. Note: the value of this parameter should not exceed the total number of features available.
- bootstrap : Allows for sampling with replacement or without replacement. Takes a boolean value; if True, sampling with replacement and if False, sampling without replacement i.e sampled only once.

We will import the RandomForestClassifier as we used the DecisionTreeClassifier earlier.

```
from sklearn.ensemble import RandomForestClassifier
```

```
clf = RandomForestClassifier()
```

The dictionary below has different values of parameters that will be tried to figure out the model giving the best performance. Apart from 'n_estimators', 'max_features' and 'bootstrap' parameters which are specific to Random Forest, the rest of the parameters are the same as the ones used for decision trees.

```
# RandomSearchCV/GridSearchCV accept parameters values as dictionaries.
# In example given below we have constructed dictionary for different parameter values that we want to
# try for Random Forest model
param_dist = {"n_estimators":[100,200,300,500,700,1000],
              "max_features": [5,10,20,25,30,35],
              "bootstrap": [True, False],
              "class_weight":[None,'balanced'],
              "criterion":['entropy','gini'],
              "max_depth":[None,5,10,15,20,30,50,70],
              "min_samples_leaf":[1,2,5,10,15,20],
              "min_samples_split":[2,5,10,15,20]}
```

The number of possible combinations are huge now as shown below.

```
960*6*6*2
```

```
69120
```

We are looking at 69120 combinations. Having said this, each Random Forest can have around 100 to 1000 trees; on an average around 300 trees. If we try all these combinations, we are looking at building about 20 million decision trees. The time required to execute this will be huge and the performance improvement we get may not be worth the investment.

Hence, instead of looking at all the possible combinations, we will consider a random subset which will give us a good enough solution, maybe not the best; the trade off being how good the model is to the execution time.

```
960*6*6*2*300
```

```
20736000
```

A good number of start with would be about 10 to 20 percent of the total number of combinations.

Note: the value of max_features cannot exceed the total number of features in the data. As seen below, max_features cannot exceed 38.

```
x_train.shape
```

```
(32561, 38)
```

```
n_iter_search = 10 # this number should be 10 to 20% of the total number of combinations
#n_iter parameter of RandomizedSeacrhCV controls, how many parameter combination will be tried; out of all
possible given values

# We try the random forest classifier here supplying different parameter ranges to our RandomizedSearchCV
which in turn
# will pass it on to this classifier
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search,
                                   scoring='roc_auc',
                                   cv=5, n_jobs=-1,verbose=20)
random_search.fit(x_train, y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done   1 tasks      | elapsed:  4.4min
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:  4.4min
[Parallel(n_jobs=-1)]: Done   3 tasks      | elapsed:  4.4min
[Parallel(n_jobs=-1)]: Done   4 tasks      | elapsed:  4.4min
[Parallel(n_jobs=-1)]: Done   5 tasks      | elapsed:  4.7min
[Parallel(n_jobs=-1)]: Done   6 tasks      | elapsed:  4.7min
[Parallel(n_jobs=-1)]: Done   7 tasks      | elapsed:  4.7min
[Parallel(n_jobs=-1)]: Done   8 tasks      | elapsed:  4.9min
[Parallel(n_jobs=-1)]: Done   9 tasks      | elapsed:  5.0min
[Parallel(n_jobs=-1)]: Done  10 tasks      | elapsed:  5.0min
[Parallel(n_jobs=-1)]: Done  11 tasks      | elapsed:  5.0min
[Parallel(n_jobs=-1)]: Done  12 tasks      | elapsed:  5.2min
[Parallel(n_jobs=-1)]: Done  13 tasks      | elapsed:  5.2min
[Parallel(n_jobs=-1)]: Done  14 tasks      | elapsed:  5.2min
[Parallel(n_jobs=-1)]: Done  15 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done  16 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done  18 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done  19 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done  20 tasks      | elapsed:  7.1min
[Parallel(n_jobs=-1)]: Done  21 tasks      | elapsed:  7.2min
[Parallel(n_jobs=-1)]: Done  22 tasks      | elapsed:  7.2min
[Parallel(n_jobs=-1)]: Done  23 tasks      | elapsed:  8.2min
[Parallel(n_jobs=-1)]: Done  24 tasks      | elapsed:  8.9min
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:  9.0min
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:  9.8min
[Parallel(n_jobs=-1)]: Done  27 tasks      | elapsed: 10.9min
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed: 11.2min
[Parallel(n_jobs=-1)]: Done  29 tasks      | elapsed: 11.5min
[Parallel(n_jobs=-1)]: Done  30 tasks      | elapsed: 11.6min
[Parallel(n_jobs=-1)]: Done  31 tasks      | elapsed: 11.6min
[Parallel(n_jobs=-1)]: Done  32 tasks      | elapsed: 11.8min
[Parallel(n_jobs=-1)]: Done  33 tasks      | elapsed: 11.9min
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed: 11.9min
[Parallel(n_jobs=-1)]: Done  35 tasks      | elapsed: 12.3min
[Parallel(n_jobs=-1)]: Done  36 tasks      | elapsed: 12.3min
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed: 12.3min
[Parallel(n_jobs=-1)]: Done  38 tasks      | elapsed: 12.4min
[Parallel(n_jobs=-1)]: Done  39 tasks      | elapsed: 12.7min
[Parallel(n_jobs=-1)]: Done  40 tasks      | elapsed: 12.8min
[Parallel(n_jobs=-1)]: Done  41 tasks      | elapsed: 12.8min
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed: 12.8min
[Parallel(n_jobs=-1)]: Done  43 tasks      | elapsed: 13.2min
[Parallel(n_jobs=-1)]: Done  46 out of  50 | elapsed: 13.3min remaining:  1.2min
[Parallel(n_jobs=-1)]: Done  50 out of  50 | elapsed: 13.7min finished
```

```
RandomizedSearchCV(cv=5, error_score='raise-deprecating',
          estimator=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=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators='warn', n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
          fit_params=None, iid='warn', n_iter=10, n_jobs=-1,
          param_distributions={'n_estimators': [100, 200, 300, 500, 700, 1000], 'max_features': [5, 10, 20,
25, 30, 35], 'bootstrap': [True, False], 'class_weight': [None, 'balanced'], 'criterion': ['entropy',
'gini'], 'max_depth': [None, 5, 10, 15, 20, 30, 50, 70], 'min_samples_leaf': [1, 2, 5, 10, 15, 20],
'min_samples_split': [2, 5, 10, 15, 20]},
          pre_dispatch='2*n_jobs', random_state=None, refit=True,
          return_train_score='warn', scoring='roc_auc', verbose=20)
```

Amongst all the models built, we get the best estimator using the 'best_estimator_' argument of the random_search object.

```
random_search.best_estimator_
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=10, max_features=20, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Note: This is a result from one of the runs, you can very well get different results from a different run. Your results need not match with this.

Looking at the outcome of the report function, we observe that the model with rank 1 has the mean validation score of 0.918 which is a slight improvement over decision trees. Trying higher values of 'cv' and 'n_iter' arguments should give better models.

```
report(random_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: 0.91790 (std: 0.00296)
Parameters: {'n_estimators': 100, 'min_samples_split': 15, 'min_samples_leaf': 2, 'max_features': 5,
'max_depth': 30, 'criterion': 'entropy', 'class_weight': 'balanced', 'bootstrap': True}

Model with rank: 2
Mean validation score: 0.91643 (std: 0.00334)
Parameters: {'n_estimators': 500, 'min_samples_split': 10, 'min_samples_leaf': 10, 'max_features': 20,
'max_depth': 10, 'criterion': 'gini', 'class_weight': 'balanced', 'bootstrap': False}

Model with rank: 3
Mean validation score: 0.91434 (std: 0.00365)
Parameters: {'n_estimators': 300, 'min_samples_split': 10, 'min_samples_leaf': 5, 'max_features': 25,
'max_depth': 30, 'criterion': 'gini', 'class_weight': None, 'bootstrap': True}

Model with rank: 4
Mean validation score: 0.91338 (std: 0.00333)
Parameters: {'n_estimators': 700, 'min_samples_split': 20, 'min_samples_leaf': 20, 'max_features': 25,
'max_depth': 30, 'criterion': 'entropy', 'class_weight': 'balanced', 'bootstrap': False}

Model with rank: 5
Mean validation score: 0.91172 (std: 0.00273)
Parameters: {'n_estimators': 200, 'min_samples_split': 15, 'min_samples_leaf': 5, 'max_features': 30,
'max_depth': 15, 'criterion': 'gini', 'class_weight': 'balanced', 'bootstrap': False}
```

Now, lets build the model giving the best performance.

```
rf = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=10, max_features=20, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

```
rf.fit(x_train,y_train)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=10, max_features=20, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

Once the model is fit, we can use it to make predictions using hard classes or probability.

Feature Importance

In scikit-learn, one of the ways in which feature importance is described is by something called as "mean decrease impurity". This "mean decrease impurity" is the total decrease in node impurity averaged over all trees built in the random forest.

If there is a big decrease in impurity, then the feature is important else it is not important.

In the code below, feature importance can be obtained using the 'feature_importances_' attribute of the model. The use of this attribute makes sense only after the model is fit. We store the feature importance along with the corresponding feature names in a dataframe and then sort them according to importance.

```
feat_imp_df=pd.DataFrame({'features':x_train.columns,'importance':rf.feature_importances_})

print(feat_imp_df.sort_values('importance',ascending=False).head())

print(feat_imp_df.sort_values('importance',ascending=False).tail())
```

```
                             features  importance
12  marital.status_ Married-civ-spouse    0.240887
3                         capital.gain    0.189728
2                        education.num    0.162019
28               relationship_ Husband    0.085335
0                                  age    0.075243
                         features  importance
15      marital.status_ Separated    0.000524
37         native.country_ Mexico    0.000482
23                  occupation_ ?    0.000426
25  occupation_ Handlers-cleaners    0.000389
31        relationship_ Unmarried    0.000365
```

We can observe that 'marital.status_ Married-civ-spouse' is identified as the most important variable and 'relationship_ Unmarried' as the least important.

Random Forests can also be used as a dimensionality reduction technique. In case of high dimensional data, we can run a random forest model and get features sorted by importance. We can then go ahead and choose the top, say 100, features for further processing. Using this technique we would not be losing relevant information and at the same time we can drastically reduce the number of features considered. In other words, even if we do not use the Random Forest model as the final model, we can use it for reducing the features.

Partial Dependence Plot

Random Forest model is an example of a blackbox model - where we do not get any coefficients for any variable and hence are unable to make an interpretation of different variables on the response variable.

In order to make an interpretation of a variable used in Random Forest, we need to make prediction on the entire data and average it for the variable we are interested in.

Lets take an example of the 'education.num' variable.

```
var_name='education.num'

preds=rf.predict_proba(x_train)[:,1] # making a prediction in the entire data
```

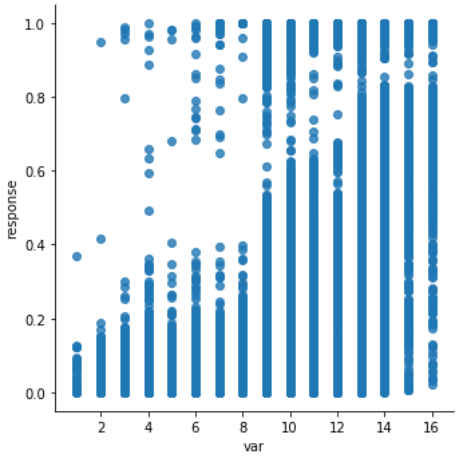
```
var_data=pd.DataFrame({'var':x_train[var_name],'response':preds})
var_data.head() # Create a dataframe of the 'education.num' variable and the corresponding predictions
```

	var	response
0	13	0.079281
1	13	0.285775
2	9	0.027492
3	7	0.114180
4	13	0.506204

We plot the two columns 'education.num' against the 'response' as shown below but it is not very informative i.e. there is a lot of variation since the response contains the effects of other variables also.

```
import seaborn as sns
sns.lmplot(x='var',y='response',data=var_data,fit_reg=False)
```

```
<seaborn.axisgrid.FacetGrid at 0x26a1276fb00>
```



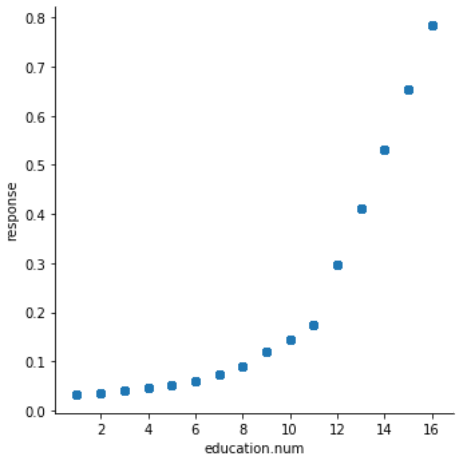
We will plot a smoothing curve through this which will give an approximate effect of the variable 'education.num' on the response. We basically will average the response at each of the 'education.num' values.

```
import statsmodels.api as sm
smooth_data=sm.nonparametric.lowess(var_data['response'],var_data['var'])
```

```
df=pd.DataFrame({'response':smooth_data[:,1],var_name:smooth_data[:,0]})

sns.lmplot(x=var_name,y='response',data=df,fit_reg=False)
```

```
<seaborn.axisgrid.FacetGrid at 0x26a1378dfd0>
```



In the plot above we notice that as the 'education.num' value goes up the chances of having income greater than 50000 dollars go up. The 'education.num' variable does not have much impact on the response till its value is around 10 or 12, but then it rises steeply indicating that with higher education the probability of earning more than 50000 dollars increases.

This exercise can be done for any of the other variables we wish to assess. However, it does not make sense for dummy variables since it has only two values. But it works fine for continuous variables.