In this module, we will talk about decision tree model. We will discuss how we can solve a classification problem with a Classification tree and a regression problem with a regression tree. We will also discuss how we can compute variable importance for both classification trees and regression trees.

Let's get started. Decision trees can be used to solve both regression and classification problems. The underlying theory behind how decision trees work is heavily influenced by a branch of computer science known as Information Theory. The classic use case of decision trees is analysis of segments in business data.

Let's start our discussion with classification tree. Let's assume, we have data of some bank customers and we would like to build a predictive model to classify customers into profitable and unprofitable ones. We can build a logistic regression classifier as well to solve this problem, but, let's see besides a logistic classifier, what else can we do.

If we look at this data, we can see that the proportion of profitable and unprofitable customers is 50% each. Let's try to subset our total data and see what happens. Let's subset our data by Age variable, with one subset coming from all the rows where age is greater than 35. The other coming from where the age

is less than or equal to 35. When we subset the data, these are the class proportions we obtain. In the subset where Age is more than 35, the proportion of profitable customers is 66%. Compare that to the proportion of profitable customers in the population. It seems that the segment of data, with Age more than 35, has a higher chance of seeing a profitable customer.

We can continue to subset our data, further, for example we can subset this portion of data by the marital status. If we do so, we can see that this is how, the proportion of profitable customers look like in each of these new segments.

We can interpret these results as a set of rules that can help us classify people into profitable/unprofitable customers. Here it is very evident that people who are more than 35 years and are Married have a good chance of being profitable. This is the essence behind a decision tree classifier. By recursively sub setting our data we can unearth interesting patterns. The idea is to split the data in such a way so that the subsets of data end up being dominated by one class of the target variable. For example, in this split, you can see all the observations have a class of “Profitable” customer. While in this subset all the observations belong to the other class, i.e. group of people who are unprofitable. You

would have also noticed that the decision tree splits into two parts at each node. Such trees are called binary trees. Most implementations of a decision trees produce binary splits.

Now that we understand what a decision tree classifier is, let us understand how decision tree algorithm works. One of the obvious questions that we would need to answer would be, 'How do we decide which variable should be used to create splits?'. We will try to understand the intuition behind how this is done and then we will formalise our intuition by introducing purity metrics.

For the sake of discussion let's go back to our original example. These are proportions of profitable and unprofitable customers that we obtain, when we split on age.If on the other hand we split on gender, these are the proportions of profitable and unprofitable customers that we obtain. Let's compare both splits to understand which split is better.

Here we have both splits shown side by side for comparison. Can you say which variable produces better splits? Age or gender? To answer that we should qualitatively think about what a good split means in the context of a classification problem. We can justifiably say that the splits produced by the variable age are better than splits produced by the variable gender. Why would that be? If you

look closely, you can see that there is greater class imbalance produced by variable age versus the split caused by the variable gender. The greater the class imbalance is the better a split is.

Now that we understand qualitatively how to think about a good split, lets formalise this notion of class imbalance. We can measure class imbalance by computing gini or entropy. This is the formula for both. Let's look at a numerical example to understand how gini and entropy computations work.

Let's look at how gini measure is computed. This is what the formula looks like, we compute gini measure for each of the two nodes. For this node, we have two classes, profitable and unprofitable, we compute their proportions and subtract the squared sum form one to obtain a value of 0.44. For the next node again we will do this computation and obtain a value of 0.375.

Out of these two terminal nodes, this node has a higher-class imbalance. And you can notice that a higher-class imbalance leads to lower gini value. We can compute the gini for first split of variable gender like this obtaining a value of 0.48. And for the second split again we will do the computation like this, to obtain a value of 0.48

Now if you look closely what has happened now is we have two gini values per variable. Ideally, we would want a single value per variable. We can do so by taking a weighted average.

We will weigh each part of the split with the number of observations in the split to total number of observations. You can see that the total number of observations in the data were 10 but only 6 observations belong to this node, so the gini value for this node gets a weight of 6/10. We weigh the other part of the split as well and on computing the weighted average we get a value of 0.41.

A same computation on the other node yields a gini value for gender variable equal to 0.48.

Now if we compare the gini value for age and gender, we can see that value for Age is low indicating a better split. So now we have a measure that can talk about how good a split is.

Let's look at how entropy measure is computed by taking a numerical example. Using the definition of entropy listed above we can compute entropy for each node of the split. For the first node, it turns out to be 0.91. For the second one, it turns out to be 0.81. Again, you can notice that the more the class imbalance is in a node, the lower the value of entropy is.

We can do the same computations for the splits produced by variable Gender. To get a consolidated measure of entropy at variable level, we can compute the weighted average. Again, you can see that entropy for age variable is lower than gender variable indicating that age is better at discriminating between the classes than gender.

Here is how decision tree algorithm can be summarised. For each split we compute the purity metric, Gini or Entropy, for each variable. Then we choose the variable which results in lowest value of purity metrics. We continue to do this sub-task until some sort of stopping criteria is met.

The stopping criteria is usually in the form of a user specified value of how deep a tree should be, for example if the user specifies that the tree should not be 3 levels deep, the algorithm terminates. Another way of controlling the growth of tree is specifying the minimum change in purity metric from one split to another. For example, a user can specify that a split should only occur if the change in gini from preceding split to succeeding is more than some value. Another way of stopping tree growth is by specifying that no splits should occur if there are only a few values in a node. For example, a user can specify the splitting should stop once the

number of observations in a node are less than 100.

Let's see now how a decision tree classifier can be used for prediction. Suppose we get a data point where we know that a person is 20 years old. Using our model, we can predict that there is 25% chance of him being a profitable customer.

Since a decision tree classifier can output probabilities, so we can use Confusion Matrix, ROC curves, Area Under ROC curves to measure the performance of the decision tree classifier. For multiclass problems, usually people use accuracy as a performance measure.

Let's talk a little about what could be the hyperparameters or parameters of a decision tree. As a user, besides data, you may need to specify parameters such as which purity metric to be used, Gini or Entropy. How deep should a tree grow, etc. These parameters are estimated using cross validation. As far as a decision tree is concerned, at a model level we end up learning rules, that help us in predicting probabilities or classes.

So far we have talked about what Classification Trees are. We have also looked into how classification trees work by talking about purity metrics such as Gini and Entropy. We have also briefly discussed how you can

measure the performance of a Classification tree. Since the classification tree gives you an output very similar to output given by Logistic Regression ie. a classification tree can give us probability estimates as well as the labels so we can measure matrixes such as ROC Curve, Area Under the Curve and Confusion Matrix.

In this video, we will talk about how we can build a decision tree classifier using scikitlearn's API.

So let's get started. The first thing I will do is, I will do some standard imports, for example, I am importing the OS module as well as the PANDAS module. After I have done this, I will make sure that python understands that I am referring to a particular directory on my system. This directory contains all my data sets that I will use for the demo. Now to make sure that python understands that I am referring to this particular directory, I will make use of the change directory method in the OS module.

So at this point in time, python understands that I am referring to this particular directory on my system to do any read or write tasks. Within this directory there is a file called credit_history, so I will read this file using read_csv method in the PANDAS module and after I have created a PANDAS data frame, I will use the head method to take a look at the first few observations of this data set that I just recommended.

This is how the data set looks like. We will try to predict this column called default. Now think of this data set as a data set belonging to a credit card company where the company

might want to understand what is causing the default in their customers.

Before we build a model, let's do a bit of data audit, let's try and understand if we have any missing values in any of our columns, it turns out that the column years has 279 missing values. Now at this point, we might want to impute these missing values. Now for us to be able to impute the missing values in this particular column, one starting point could be to look at the distribution of this variable in our dataset. This is what the distribution of the years column looks like and a naive way to do an imputation for these 279 missing values could be to replace these missing values by the median of our data. Now median of my data at this point is 4, so I will use the fill NA method to replace the missing values with the value of 4 in the column years of my data set. After this I will create my predictor matrix, this is how the predictor matrix looks like. Now you can notice if there are some variables in my predictor matrix that are non numeric.

So the next thing I will do is I will create a numeric representation for these non numeric variables. The most easiest way of creating a numeric representation of these non numeric variables is to do one hot encoding. I can do one hot encoding using the get_dumies method of a PANDAS data frame. This is how

my predictor matrix now looks like, after I have one hot encoded my categorical or non numerical variables.

Next I will create a target vector. Since I am predicting the default column so I am going to store all the values in a default column in an object Y. At this point in time, I will split my data set into test and training components. I will use the model selection module. Within the model selection module, there is a trained test split method that helps us in splitting a data set into testing and training components.

After I have done this, I am now ready to build my decision tree model. To build a decision tree model, I will have to import the tree module. Within the tree module, I have a class called decision tree classifier. This class accepts many parameters when you instantiate an object of this class. Let's look at some of the parameters that this particular class accepts.

These are some of the parameters that this class instantiation can accept. I am referring to the official documentation of this particular class. For the purposes of discussion, I am only instantiating one parameter which is the max depths. After I instantiate an object of decision tree classifier class, I will use the fit method and pass my training data set. Till

here, my decision tree classifier would have been trained.

Now the next thing I will do is, I will use my decision tree classifier that I just trained to obtain the accuracy score on my test data set. So let's execute these sequences of commands and as we can see, on my data set, the accuracy is around 62%. I may also want to compute some different performance matrix, for example, I might want to compute the area under the curve, so I will import the matrix module which has this method called roc_auc_score and it accepts two parameters, the actual labels and the predicted probabilities from the decision tree model. Let's run this.

The area under the curve turns out to be .67, which is more than. 50 so definitely, my decision tree model is better than a naive classifier, now after we have built this decision tree model, the next thing we might want to do is we might want to visualize a decision tree that we have just created. Now in order to visualize the decision tree, we will need to make use of a couple of packages.

One of the packages we will need to make use of is called pydotplus. This package is not a part of standard anaconda installations, so you will have to install it separately. You can use

you conda install manager or your pip install manager to install pydotplus.

The second piece of software that we need to visualize the decision trees is called graphviz. If you are working on a windows machine, you will have to explicitly install graphviz, you can just google search about graphviz and you will be able to reach to a page where you can download graphviz. Now after you download graphviz, unlike other pieces of software, the binaries of the graphviz package are not appended to the path variable. So either you can manually append the binaries of the graphviz installation to your path variable or you can also programmatically do this. This is what I am doing here.

Now I will use my trained model which is stored in the object clf, to create a representation of my model, that can be further used to create a visual output of my trained tree model. Now we will use this method called export_ graphviz, which is part of the tree module. This method accepts a couple of parameters. A trained model is one of the parameters. The other two important parameters are called feature_names , to this I am passing the column names in my predictor matrix and I have another parameter, if in case you are building a classification tree, you will need to talk about what are the classes in

your data set, so if you remember in my target variable, I had only two classes, 0 & 1 which are what I am specifying here. After I am done with this, I will create a graph representation from this object that I just created and then I will use the image module to visualize the tree that I have just built.

This is the visual representation of my tree model. Now the first split is occurring on the variable called grade A. Now if you remember, grade A was a one hot encoded variable. So grade A less than equal to 0.5 indicates that the person does not have a grade A, if a person does not have a grade A, then we move in this direction and we reach here, this means all the values where people do not have a grade A as well as a grade B, and I reach to this node and this node specifies if a person has an income of less than this number, then I reach to these two decision nodes.

Now you can see that in each of my decision nodes, I have the class proportions and it is also specified which is a majority class, for example here, the majority class is 1 while here, the majority class is 0. You can see some numbers here, for example, 15.0%, this talks about the percentage of total data that resides in this particular node. I have also the gini number, or the gini measure for this particular node.

Another thing that we have not touched upon is our decision tree classifier can take many parameters which we might need to tune. We can do this tuning process using grid search cross validation. Lets take an example where we are just doing grid search on the max depth parameter. So again I will use the gridsearchcv..I repeat again, so again I will use the gridsearchcv method in the model selection module. I will specify the model object and I will specify the parameter grid. Lets do the grid search on max depth. Lets take a look at the best estimator, so my best estimator is for a max depth of 2, lets look at its score, its score turns out to be .63

So in this particular code demo, we have seen how we can build a simple decision tree classifier. We have also seen how we can visualize a decision tree classifier and we have also seen how we can do a grid search if we want to tune some parameters of a decision tree classifier.

Decision tree can be used to do regression as well. When the target variable is continuous, one can use a decision tree regressor. The prediction in the case of a decision tree is the mean value of the target variable. Let's take an example to understand how regression tree works in bit more detail.

Suppose we have this data set about price of various cars along with other attributes such as: where the car was manufactured; what is the size of wheel rim; size of tire; and type of car. We can try to build a decision tree model to predict price. Note that price is a continuous variable, so we will try to build a regression tree.

To build a regression tree we will recursively subset the data. One example of subsetting is this, here we are sub setting the data on the variable rim. Now in a regression framework since the target variable is continuous, so we look at the average value of the target variable in each subset. As you can see due to this split we have two nodes and in each node the average price is 17.50 for this subset and 26.97 for this subset.

Just like a classification tree, in a regression tree, we recursively sub set the data and as a prediction we obtain the average value of continuous target variable. Just like a

classification tree, a regression tree can also be summarised as a series of rules.

Once we understand what regression trees are, the next question that needs to be answered is how does a regression tree algorithm pick up which variable to split on? While doing regression, we want our predictions to be accurate. In a regression tree, the prediction is the average of target variable in a decision node. One way to think about how good a split is in terms of how accurately the split helps in making predictions. For regression trees one usually either computes Mean Squared Error (MSE) or Residual Sum of Square (RSS) as a proxy of accuracy in each node.

Let's take an example to understand how MSE or RSS helps in deciding which variable to choose for a split. This the average value of price in each node, when the split is done on variable rim

This is what the average value when the split is done on variable Germany. Now, you can see our subsets have changed due to the choice of different variables to split.

Here is a summary of what we did in last two slides. Now, how can we decide if rim is better variable to split compared to country? Or, country is a better variable to split compared to rim.

As discussed earlier we will need to see which variable helps in creating more accurate predictions. We can use MSE or we can use another quantity called RSS to measure that. MSE as you can see here is just the average of RSS. Also, if you look at this term closely, you will realise that this is nothing but variance in the values of target variable in a node.

Here is the subset of data caused due to this split and this is another subset. Now, what do you think will be the predictions for this node and this node according to this decision tree? As we have discussed earlier the predictions are the average value for each node computed by taking the average of the subset of the target variable created. For this node again, the prediction will be the average of price in this particular node. Same process will be repeated for this tree as well. Now the MSE tried to find out how accurate a prediction is in each node. One example of computing the MSE is to compute this number for this node using this formulation as well. Now if you compute these numbers these are the MSE for both the trees for each of its nodes. Now we will want to obtain a single number talking about the MSE of the whole split

So, what we will do is we will compute the weighted average of MSE. Since 6 observations passed through this node out of 10 so then this node will get a weight of 6/10. Similarly, this

node will get a weight of 4/10. This is our computed estimate of MSE for this whole split. And, this is our estimate of MSE for this whole split.

Now since the weighted estimate of MSE for this split is lower than this split, hence it means that rim is a better variable to split on compared to country.

Let's talk a little bit about the hyperparameters of a Regression Tree. Just like a classification tree a user has to decide on: depth of tree, number of observations in terminal nodes etc. Now one can use a grid search to figure out the appropriate values of these hyperparameters

Both classification tree and regression tree give us the ability to arrive at an idea of how important the predictors are. This is done by computing feature importance. Feature importance is computed as the total reduction of purity measure brought out by a feature.

Let's take an example to understand this better. Let's take an example of a classification tree to explain how feature importance is computed. Let's assume that we have this decision tree model. First split happens on Feature A and second split happens on Feature B. You can also see the class proportions and the number of observations for each split in each node.

Out of Feature A and Feature B which feature do you think is more important? Do you think that since Feature A causes the first split so it should be more important than Feature B?

Do you think that Feature B is more important as it is able to create greater class imbalance?

Now if the look at the nodes created by the split by Feature B you can see the proportion of classes is more disproportionate compared to let's say if we observed the split caused by the Feature A.

This is the Gini measure for each node of our decision tree.

While computing Variable importance both the sequence of the split and the purity of a node should be considered. As you can see, Feature A precedes Feature B but Feature B seems to create a greater class imbalance or creates greater node purity.

Also notice that the ability to create class imbalance can be measured by the difference in Gini from one split to the other.

This is how feature importance for A will be computed, the decrease in gini captures the ability of a variable to create class imbalance compared to preceding split. While the proportion of data, tries to capture the sequence in which this variable causes the split. It's easy to see if a split occurs early,

then more observations will pass through that node.

This is how feature importance for B will be computed. As you can see that decrease in gini in this case carries more weight but since only 400 observations are contained in this node, so the overall impact of this term is low.

For a regression tree, one would look at the decrease in MSE or RSS by each feature and weigh this decrease appropriately to obtain feature importance.

In this code demo, we will talk about how we can build a decision tree regressor using scikit-learn's API. let's get started.

Here I am doing some standard imports. I am importing the OS module as well as the PANDAS module. Here I am setting a working directory. Within this directory in my computer there is a file called dm.csv which I am reading and there I am displaying the column names of this particular file that I have just read.

So, these are the column names, let's take a look at a snapshot of our data set. You can think of this data set as coming from an e-retailer. This e-retailer has recorded the demographic information of each of his customer and he is also capturing the quarterly mean expenditure by each of his customers. For the purposes of this demo let us assume that this e-retailer wants to predict the amount spent column using the other information that he has about his customers. Now since the amount spent column is a continuous column so we will be using a decision tree regressor. The next thing we will do is we will create a predictor matrix. Since amount spent is a column that we want to predict so, amount spent variable cannot be a part of our predictor matrix. Also, customer Id column should not be made a predictor. So,

hence we are dropping the customer Id column as well.

Let's take a look at our predictor matrix. This is how our predictor matrix looks like. As you can notice there are some variables in our data set that are non-numeric. So, we will need to create a numeric representation of these variables which are non-numeric. We will use the get_dummies method to do that. Let's take a look at our predictor matrix after we have converted the non-numeric variables into their numeric representation.

Now, I will create the target vector. Before building our decision tree regressor model, I will split my data set into test and training components. I will use the model_selection module. Within this module there is a method called test train split, I will use this method to split my data into test and training components. At this point in time, we are ready to create a decision tree regressor. I will import the tree module. Within the tree module I have a class called decision tree regressor. I will be using this to instantiate an object of decision tree regressor class and once I instantiate this object, I will use the fit method on this object to fit my training data on a decision tree regressor model. Let's do that.

At this point in time we fit in a regression tree. Let's take a look at the accuracy score of this regression tree on our test data set. I can use the score method to do this. This prints out the mean squared error on the test data set of this model. I can also printout which predictors are important by printing out their feature importances. This gives me an array of feature importances of all the predictors that I have used.

A better way to represent feature importance is to create a series object out of this numpy array and also provide a name corresponding to the magnitude of feature importances. The name corresponds to the column names in my predictor matrix. Let's create the series object. Here I am creating the series object. After the series object is created, I will be sorting the values of the series object and arranging them in descending order and then printing out the top five predictors by their feature importance. So, it turns out that in this model, salary is the most important feature followed by catalogues and the others so on.

Let's visualise the decision tree regressor model that we have just created. This is how the visual representation of our decision tree regressor looks like. As you can see, the first split occurs on the salary variable. And these

are my decision nodes. We can see some other numbers as well in these decision nodes. For example, we can see a number called values in each of these decision nodes. This number corresponds to the average of our target variable which in this case is the amount spent. You can also see some other numbers here for example, you can see that samples= 114 in this node, 97 in this node. This talks about the total number of observations corresponding to this subset of our data. Also, you can notice that mean squared errors are also reported for each of our splits.

Now, decision tree models are regularly used in the business setting to create segments in the data. Let's see how we can use this decision tree to figure out segments in our data set.

What I have done here is, I have noted down the rules corresponding to all of my decision nodes. I have also noticed down the average value of amount spent in each of my decision nodes, the sample size and I also know what is the global average of amount spent column in my total data set. Now, I can always compare the total average of the amount spent column with the average in each of the segments or each of the nodes created by my decision tree. Now, this output can be interpreted as segments. For example, we can say this, this

and this are three different segments in our data set in which the average of amount spent is more than the population average of amount spent, or in other words we can say that these are the three segments in our data in which people tend to spend more.

With this we conclude our discussion on decision tree regressor models.

In this module, we will talk about another category of machine learning models, called ensemble models. We will confine our discussion to Tree based Ensembles. We will introduce bagged trees, Random Forests and Boosted Trees in detail.

Till now we've been talking about single models. There was either one good single regression model or one good single classification model. When we say that we are building ensemble models, what we do is combine several simple models together and this combination of several models is treated as one single meta model called an ensemble.

Using an ensemble strategy, one can obtain very powerful models. We can build regression ensembles as well as classification ensembles.

A regression ensemble works by combining several simple regressors and the final prediction of an ensemble model is a simple average of predictions from all the simple regression models that make up this ensemble. Similarly, for an ensemble classifier, the final prediction is a majority vote out of all the predictions made by the constituent classifiers that make up this ensemble.

Here is a schematic of how ensembles work. Here we have three models that make up this ensemble. We get a final prediction from this ensemble in the form of either an average of

predictions if the constituent models were regressors or a majority vote if the constituent models were classifiers.

Theoretically, an ensemble model can be built by combining any set of simple models. But tree models are a more popular choice. Another characteristic of tree based ensemble models is that while training these models, only a subset of total data is used by each base learner, the way this data set is fed into each of the base learners is based on a data sampling scheme. Different sampling schemes give rise to different types of tree based ensembles. We will discuss about these resampling schemes in more detail later.

There are three popular tree based ensemble models - (i) Bagged Trees (ii) Random Forests (iii) Boosted Trees

These three models differ primarily with respect to the data sampling scheme they use. We will talk about these data sampling schemes in detail when we undertake detailed discussions on each of these ensemble techniques.

Let's talk about how a bagged ensemble works. The working of a bagged ensemble can be understood if we understand formally how we define the total error due to any machine learning algorithm.

The total error due to any ML algorithm can be thought of as sum of In-sample error and Out-of-sample error. In sample error is caused by the inability of an ML algorithm to fit the training data well. Out-of-sample error is caused by the inability of a model to generalize well. It can so happen that our model is able to do good predictions on our training data but is unable to do predictions on unseen data.

There is also a trade-off between the model complexity and the kind of error that our model makes.

More complicated models have very low in-sample error but have a high out-of-sample error. While simpler models have low out-of-sample error but high in-sample error. This would mean that there is a limit to minimum error, that one can theoretically achieve.

The only way to reduce error further is if we can find a way to simultaneously decrease the in-sample error and out-of-sample error. This is precisely what ensemble models end up doing.

The way bagged ensemble models reduce in-sample error is by using tree based models as base learners. While training a tree based ensemble the constituent tree models are allowed to grow many levels deep. What this does is, it makes sure that intricacies of

training data are captured intimately, thereby reducing the in-sample error.

To reduce the out-of-sample error, each of the tree models is fed a resampled data. In the case of “Bagged trees”, each of the unpruned trees are fed bootstrapped samples of original data set. Before we discuss how bootstrapped samples can reduce out of sample errors. Let’s discuss what bootstrapped sampling is.

Bootstrapped sampling simply refers to sampling by replacement. For example, in the exhibit here you can see we are drawing samples with replacement as many observations are getting repeated in the sample more often than they are present in the data. You can see that blue and red dots are repeated more often in the samples than they are present in the original data.

At a data level, this is how bootstrapped sampling manifests itself. When we take bootstrapped samples from data, you can see that some rows get repeated more often in the samples.

Now, let’s see how bootstrap sampling helps in reducing out of sample error. Imagine if we fit an unpruned decision tree to any data set, then it’s easy to see that such a model will have very high out of sample error. What is the reason?

The reason is that the test data can be very different from training data and since we are overfitting our tree model on training data, the out of sample error will be high.

Now, hypothetically speaking, if we fit our unpruned tree model on total population data then, since the model has seen all the possible data that exists, the error will be very low. But such a scenario, where we have access to total population data, is not realistic. The reason we will see low error is because all the variation and variety in data has been already seen by the model as we have used the population data to train our model.

The purpose that bootstrap sampling serves is the simulation of the variety and variation in data. If you think about it, the bootstrap sampling creates a more realistic view of the data generation process. When we take bootstrap samples we are able to synthetically generate variation and variety in the training data itself.

So, one of the first ensembles i.e. a Bagged Model is characterised by (i) The use of unpruned decision trees as base learners and (ii) Use of bootstrapped sampling to create samples that are fed to each of the base learners.

After understanding what a bagged tree ensemble is, let's look at some of the

peculiarities of this model. One of the peculiarities of a bagged tree ensemble is that, since the model is comprised of 100s of decision trees, such a model is not as interpretable as a, linear model or a simple decision tree. Having said that, there are some qualitative statements that one can still make even if they are using an ensemble model like a bagged tree. For example, one can always figure out which predictors are more important by looking at a metric called variable importance.

Variable importance is computed by averaging or summing the improvement in a purity metric such as Gini or Entropy for a classification model and RSS for a regression model for all the variables. Let's take an example

Since a bagged tree ensemble model is composed of many tree models, we can find out the feature importance of each variable in each of the constituent trees. This variable importance can be measured by tracking the decrease in gini metric and weighing this decrease appropriately. For example, here we are showing two trees constituting a bagged ensemble. For the first tree we are computing variable importance for all the variables used in the split. We do the same process for the second tree as well

If our ensemble has N trees, then we can easily tabulate the variable importance of each variable across all trees. To get a consolidated number we can find the average values of these importance measures per variable. Most machine learning frameworks are able to compute variable importance

Bagged tree models also are parametrized in some way. Can you think of what could be the user specified parameters while building a bagged tree model? Number of trees to be used to build an ensemble, the depth of tree, the number of observations per node of a tree can all be specified by the user.

These user specified parameters have an implication, i.e. based on what parameters are specified, one can end up with a different ensemble model. For example, we can have a model with 100 trees and depth of 4, or a model with 150 trees and depth of 3 or a model with 500 trees and depth of 4.

Now if you had to choose between these three models, one of the things that you can do is K-Fold Cross Validation and get an estimate of out of sample error. Now doing K-Fold CV is computationally very expensive, in most tree based models the way to estimate out of sample model performance is to compute something known as an Out of Bag Error.

Let's try to understand what out of bag error means? Now when, we do bootstrap sampling, within each sample it is possible to leave out some observations from the original data. For example, if we look at the sample supplied to the first tree, second row of the original data is left out. Now this will happen with all the trees in the model. These observations then become out of bag and we use each tree to make predictions on these out of bag observations to arrive at an estimate of out of sample model performance. On an average when bootstrapped sampling is done around 33% of observations become out of bag. Now the beauty of OOB error metric is that, we don't need to do any extra work to compute OOB error as out of bag observations are created the moment bootstrapped samples are taken.

Besides bagged trees, another very popular tree based ensemble is random forests. Random forests are like bagged trees, the only difference being how data is resampled in the case of Random Forests. For Random Forests, the sampling scheme is a little different. The difference is that, while taking random bootstrapped samples for each tree, only a subset of features is used at each split of the tree.Just like bagged trees we still use random bootstrap samples.

Let's look in more detail at how random forests works. Assume we have a dataset with n variables and to build a random forest model we take one bootstrapped sample and use it to build a decision tree model.

For the first split, instead of choosing across all N features, we randomly sample only 3 features and out of these three features we decide which one is most suitable for split. Again, for choosing the second split of this tree, we again randomly choose between only 3 features instead of all N features. For the purposes of demonstration, we are taking random samples of 3 features but in reality, this number can be different from 3 and is usually a hyperparameter that the user of the algorithm specifies.

Yet another bootstrapped sample is drawn from the data to create another tree model. For this tree again, for each split we only look at a random subset of features to decide which feature will be used to split the data.

We take yet another bootstrapped sample to create a new tree model. For this tree again, for each split we only look at a random subset of features to decide which feature will be used to split the data.

Since, Random Forests also use tree models as base learner, so just like a bagged tree model one can extract variable importance as well as

compute OOB error to get an estimate of out of sample model performance for parameter tuning.

Random Forests also have some user specified parameters such as the number of trees used in a model, depth of trees, number of observation in root node, number of features to be considered for each split etc. And as discussed earlier these parameters can be tuned using Out of Bag error.

In this code demo, we'll talk about how we can build a boosted tree ensemble. Let's get started.

Let's do our usual imports such as Pandas and numpy. Let's set up our working directory. This is the directory on my system where I have stored a data set which I'll use for this demo. Let's use the change directory method in the OS module to make sure that Python understands that this is the directory I will be referring to by default. Let's read in a data set called hr.csv

Let's look at a snapshot of our data set. As we have discussed earlier, we will be using the data set collected by the HR department of a company wherein using these features, we would try to predict if someone is likely to leave our organisation or not.

Let's do a bit of data audit and see if we have any missing values. Let's also check the data types and also check this column called sales and see what unique values we have in this column. It seems like this column needs to be renamed from sales to department, this is what we are doing here. Also, let's look at the salary column in our data set and as you can see this is not a numeric column, but a categorical column, so we'll need to do something about columns with non-numeric

data later on. Let's create a predictor matrix and the target vector.

Let's look at our predictor matrix. As you can see there are some variables which are non-numeric, let's one hot encode them or create dummy variables out of them. This is what I am doing here. Let's again look at the snapshot of our data once we have one hot encoded our non-numeric variables.
Now let's split our data into test and training components. Once we do this, let's import the gradient boosting classifier class from the ensemble module and lets instantiate the gradient boosting classifier object. This object while instantiating accepts many parameters, some of them are hyperparameters of the gradient boosting classifier. One of the hyperparameters is how many trees I should have in my gradient boosting classifier. Here I am picking a value of 18, later on I will do a grid search to figure out what could be a good value of number of trees in my gradient boosting classifier.

Let's call the fit method on my training data. This will fit the gradient boosting classifier. Let's see what is the accuracy score on the test data. The accuracy on my test data is around 97%. Now to tune the hyperparameters of a gradient boosted classifier, we will make use of GridSearchCV again. Since there is no

bootstrapped sample taken when we build a gradient boosted or adaboost kind of ensemble so, we'll not be able to use out of bag errors. Hence, we'll again resort to GridSearchCV.

Here I am using the GridSearchCV API , I am only doing grid search on the number of estimators, this is the grid of estimators on which I am doing the grid search. Let's execute this. Let's see what's the best classifier or best estimator out of the grid of different models that we have done a grid search on. So, the best model seems to be the one where the number of estimators are 160. Let's use this suggestion by our grid search to now create a classifier with 160 estimators or 160 trees and retrain our model.

Let's look at the score, so this is the model where the number of trees are 160. Let's look at the feature importance. For gradient boosted classifier, the scikit-learn API gives us a nice way to extract feature importances. It gives us an attribute called feature_importances_....this lists down the array of all the features.

Let's print it and convert this array into a series and assign names to each of these numbers and see this variable importance corresponds to which variables in my data. So, I'll convert this feature importance array into

a series object, assign the names and then arrange it in the descending order to see what are the top variables by their feature importance.

So it turns out that satisfaction level is a top variable by a feature importance, followed by number of projects, followed by average monthly hours and so on and so forth. I can also visualise all of this. This is what the visual for feature importance looks like.

Now, another thing that we have talked about was that we can always build partial dependence plots with our ensemble models. Within scikit-learn, scikit-learn allows us to build partial dependence plots for gradient boosted ensembles, it doesn't allow us to build partial dependence plots for let's say random forest or backed trees. So, let's look at the scikit-learn API to build a partial dependence plot on gradient boosted classifier.

So, first I will import the plot_partial dependence method from the partial dependence class. Now, I will call the plot_partial dependence method, I will pass on the model that I have just trained, the predictor matrix, and I will also tell with respect to which variable do I want a partial dependence plot. Here I will specify which

predictor I want to look at when I am creating a partial dependence plot. Remember in a partial dependence plot, the Y-axis is always the target variable, the X-axis is always the predictor so here I want to know how the target variable changes with respect to the first predictor in my data set. Let's build it.

So, the first predictor in my data set is satisfaction level and it talks about that less people will go out of my organisation as their satisfaction level increases.

So, in this way I can create different partial dependence plots to understand the relationship between my target variable and my predictors.

Boosting is yet another ensemble technique that can use decision trees as base learners. Boosted trees work differently as compared to bagged trees and random forests. Instead of taking bootstrapped samples, boosted trees use a data re-weighing strategy to build an ensemble. Adaboost is a very popular boosting technique. Also unlike bagged trees and random forests, the tree models in a boosted ensemble aren't grown very large, mostly the trees are grown to a depth of 2 or 3 levels.

Let's understand what data re-weighing means in the context of adaboost ensembles. Assume we have a dataset and we are doing a classification task. In an adaboost ensemble, this is what the sequence of events looks like: This data set is passed to a decision tree learner. Usually this is a tree which is shallow hardly 2 to 3 levels deep. This model is then used to score the data set, as you can see, the model makes mistakes for some rows. This is where re-weighing comes into picture. Where ever the model makes a mistake, that row is given more importance. As you can see here row 1 and 2 have been mislabelled by the model, these rows are given more weight. This re-weighted dataset is then passed to another decision tree model,which again scores the data set and wherever it makes mistake. That row is given more importance.

This process is repeated many times in succession.

As mentioned earlier, the tree models will be shallow tree models, could be stumps as well. The final model is a combination of these trees.

The rationale of this re-weighing strategy is that we are making sure that, each successive tree pays more attention to the parts of the data that preceding trees have failed to correctly predict. In this way, successive trees try to improve the error rate.

Adaboost is not the only, boosting algorithm, there is another popular boosting technique known as gradient boosting.

Let's see how gradient boosting works. We will explain this by taking an example in the context of regression, but the mechanics of our discussion are still valid even in a classification setting. Just like adaboost, gradient boosting is an iterative algorithm

In step 1 suppose we have this data set with one predictor and one target variable. We fit a simple tree model to this data. And obtain predictions. Also notice that there is some error in our prediction, this error is being captured in this column of residuals. We fit

another tree model now, on the residuals obtained.

In the next step we fit the combination of the two tree models on our training data set again. We again get predictions, this time our predictions seem to have improved as can be seen in the residuals column. We again fit a tree model to the residuals obtained via the combination of T1 and T2. We keep on repeating this process quite a few times and eventually end up with an ensemble of trees.

One thing to keep in mind here is that gradient boosting is a general ensemble framework. We have discussed boosting taking decision trees as base learners, it's entirely possible to use other base learners as well.

The ensemble models, we've discussed so far, are not as interpretable as simple models such as decision trees or linear models. We have seen that ensembles do give us an ability to list out important predictors by computing variable importance measures. But even if we find out which variables are important predictors, we still don't have a way of finding out direction of impact a given predictor has on the dependent variable. For example, one might want to know, if a given predictor positively or negatively impacts the dependent variable.

One way to understand relationships between a dependent variable and an independent variable is to create a partial dependence plots. These plots help in establishing the direction of impact of a predictor on target variable. One drawback of using partial dependence plots is that only bivariate relationships can be understood but unearthing interaction effects might be challenging. Also creating partial dependence plots is computationally very expensive. Depending on which machine learning framework you are using, partial dependence plots for the ensembles we have discussed so far, may or may not be supported.

Partial dependence plots tell us how the value of target variable changes as the value of a predictor variable is changed, after considering the effect of all the other variables. For example, this exhibit shows three partial dependence plots. The X axis represents values of a predictor while the Y axis represents the values of target variable. As can be seen in the first two plots, the value of target variable doesn't change, while changing the value of predictor variable. In the third plot you can see that there is positive relationship between the target variable and the predictor variable.
Let us assume that this is how the predictor matrix looks like for our data set. We have

predictor 1, predictor 2 and predictor 3 and we have their corresponding values.

Now to create a partial dependence plot, talking about the relationship between predictor 1 and the target variable, this is what the sequence of events looks like. We first enumerate for each unique value of the predictor 1, all the possible combination of values for predictor 2 and 3. Then, we pass each row so obtained to our model to obtain the prediction of our target variable. Now, we do this process for each unique value of our first predictor variable and then we obtain the average of the predictions that we got from our model by passing each of these rows.

Now once we do that, we obtain a table like this which tells us how the prediction varies as the values of the first predictor is changed taking into account all the possible values for the other predictors. Now, this can be easily plotted in a bivariate scatterplot like this to unearth the relationship between first predictor and the target variable.

Here you can notice one thing here, to create a partial dependence plot, we'll have to look at all the possible combinations of the other predictor variables as well. This makes the whole process of finding out the partial

dependence plots computationally very expensive.

In this code demo, we'll talk about how we can build a boosted tree ensemble. Let's get started.

Let's do our usual imports such as Pandas and numpy. Let's set up our working directory. This is the directory on my system where I have stored a data set which I'll use for this demo. Let's use the change directory method in the OS module to make sure that Python understands that this is the directory I will be referring to by default. Let's read in a data set called hr.csv

Let's look at a snapshot of our data set. As we have discussed earlier, we will be using the data set collected by the HR department of a company wherein using these features, we would try to predict if someone is likely to leave our organisation or not.

Let's do a bit of data audit and see if we have any missing values. Let's also check the data types and also check this column called sales and see what unique values we have in this column. It seems like this column needs to be renamed from sales to department, this is what we are doing here. Also, let's look at the salary column in our data set and as you can see this is not a numeric column, but a categorical column, so we'll need to do something about columns with non-numeric

data later on. Let's create a predictor matrix and the target vector.

Let's look at our predictor matrix. As you can see there are some variables which are non-numeric, let's one hot encode them or create dummy variables out of them. This is what I am doing here. Let's again look at the snapshot of our data once we have one hot encoded our non-numeric variables.
Now let's split our data into test and training components. Once we do this, let's import the gradient boosting classifier class from the ensemble module and lets instantiate the gradient boosting classifier object. This object while instantiating accepts many parameters, some of them are hyperparameters of the gradient boosting classifier. One of the hyperparameters is how many trees I should have in my gradient boosting classifier. Here I am picking a value of 18, later on I will do a grid search to figure out what could be a good value of number of trees in my gradient boosting classifier.

Let's call the fit method on my training data. This will fit the gradient boosting classifier. Let's see what is the accuracy score on the test data. The accuracy on my test data is around 97%. Now to tune the hyperparameters of a gradient boosted classifier, we will make use of GridSearchCV again. Since there is no

bootstrapped sample taken when we build a gradient boosted or adaboost kind of ensemble so, we'll not be able to use out of bag errors. Hence, we'll again resort to GridSearchCV.

Here I am using the GridSearchCV API , I am only doing grid search on the number of estimators, this is the grid of estimators on which I am doing the grid search. Let's execute this. Let's see what's the best classifier or best estimator out of the grid of different models that we have done a grid search on. So, the best model seems to be the one where the number of estimators are 160. Let's use this suggestion by our grid search to now create a classifier with 160 estimators or 160 trees and retrain our model.

Let's look at the score, so this is the model where the number of trees are 160. Let's look at the feature importance. For gradient boosted classifier, the scikit-learn API gives us a nice way to extract feature importances. It gives us an attribute called feature_importances_....this lists down the array of all the features.

Let's print it and convert this array into a series and assign names to each of these numbers and see this variable importance corresponds to which variables in my data. So, I'll convert this feature importance array into

a series object, assign the names and then arrange it in the descending order to see what are the top variables by their feature importance.

So it turns out that satisfaction level is a top variable by a feature importance, followed by number of projects, followed by average monthly hours and so on and so forth. I can also visualise all of this. This is what the visual for feature importance looks like.

Now, another thing that we have talked about was that we can always build partial dependence plots with our ensemble models. Within scikit-learn, scikit-learn allows us to build partial dependence plots for gradient boosted ensembles, it doesn't allow us to build partial dependence plots for let's say random forest or backed trees. So, let's look at the scikit-learn API to build a partial dependence plot on gradient boosted classifier.

So, first I will import the plot_partial dependence method from the partial dependence class. Now, I will call the plot_partial dependence method, I will pass on the model that I have just trained, the predictor matrix, and I will also tell with respect to which variable do I want a partial dependence plot. Here I will specify which

predictor I want to look at when I am creating a partial dependence plot. Remember in a partial dependence plot, the Y-axis is always the target variable, the X-axis is always the predictor so here I want to know how the target variable changes with respect to the first predictor in my data set. Let's build it.

So, the first predictor in my data set is satisfaction level and it talks about that less people will go out of my organisation as their satisfaction level increases.

So, in this way I can create different partial dependence plots to understand the relationship between my target variable and my predictors.