Chapter 10: Prediction Models

The materials in this chapter are also from the LinkedIn Online Course *Python: Working with Predictive Analytics*. You are required to complete this course. It's 1 hour 22 minutes long. The link to the course is here

https://www.linkedin.com/learning/python-working-with-predictive-analytics

You'll receive a certificate upon completion.

In this chapter, you'll learn different types of predictive models.

Introduction to predictive models

With our prepared data, now we can start training the prediction models. Data prep part is where most engineers and data scientists spend the majority of their time and effort. So be sure to celebrate once you've finished preparing your data.

Let's take a step back and talk about what prediction is, what the common use cases and examples are, and look at some of the most used models. First, let's start from the beginning. Machine learning models divide into three categories. First, supervised which are models with label data. Second, there's unsupervised which are the models without the labeled data. And finally, the reinforcement learning, which is a method based on trial and error using feedback.

So what are some examples of prediction? For supervised learning, where we used labeled data, the most commonly used learning methods are regression and classification. What's the difference? The output variable in regression is numerical and the output variable for classification is categorical. Determining whether a credit card expense is fraud is considered as a classification model. Housing price estimates is a regression model. In this chapter I focus on the regression models.

Now, what if we were working with unlabeled data? We will be working with unsupervised learning. Grouping or clustering are some of the methods used. It's a lot less controllable compared to supervised learning. The machine learns through observation and finds structures in data like clustering and association examples. As you can see here in the customer behavior one, grouping customers by their purchasing behavior without having prior knowledge is considered an example of clustering. And in the customer behavior two, people who buy product A also tend to buy product B is considered an example of association.

Now that we've prepped the data in the prior sessions, we will create predictive models using this data, followed by measuring each model's performance. If performance is not satisfying, we will look into ways to improve. There are quite a few models to work with. We will only cover a few commonly used ones in this course. They each have advantages and disadvantages based on the problem and the data we have. Some of the commonly used models we will be looking at includes linear regression, polynomial regression, support vector regression, decision tree, and random forest regression. If the predicted output is a category, classification is the method to use. If the predicted output is a numeric, regression is the method.

Linear regression

We are now at the modeling section.

In this chapter, we cover regression models to make predictions. Let's start with simple linear regression. Simple linear regression is nothing but explaining the dependent variable Y with independent variable X.

Imagine you are recovering from an injury and had a doctor's appointment. They ask you to walk every day and increase the duration of the walk gradually. You started to record your step counts on the Post-it notes every day. However, no one would have expected a thunderstorm and you left the windows open. Only half of your notes survived, 10 out of 20. Guess what, linear regression is here to save your back.

Let's get to work. One way to fill in the missing data points would be taking the mean. So, looking at this graph, the distance between the data points and the mean is the error. Error on the negative side is as important as the error on the positive side. Thus, we will square and sum the error. Another way is to fit a linear regression line and measure the error as the distance between this line and the actual data points. Again, we will square and sum the error in order to prevent a negative error to cancel out the positive, and thus reduce the total error. This is simple linear regression.

There is also multiple linear regression, where we have more than one independent variable X. Here are some ways we can predict the lost data before your next doctor's appointment. One way, we can take the mean.

Now, let's get to the code for multiple linear regression on our data in Python.

In the subfolder */ExerciseFiles/02_02*, open the Python program *02_02_Finish.py* in Spyder. The program is as follows:

```
--snip–
lr = LinearRegression().fit(X_train,y_train)
y_train_pred = lr.predict(X_train)
y_test_pred = lr.predict(X_test)

#print score
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
print('lr train score %.3f, lr test score: %.3f' % (
lr.score(X_train,y_train),
lr.score(X_test, y_test)))
```

We will first fit the function with `lr = LinearRegression().fit(X_train,y_train)`. This command runs a linear regression of Y on X using the training data. Based on the regression results, we can calculate the predicted Y value in the training dataset, using the command `y_train_pred = lr.predict(X_train)`. Similarly, we can calculate the predicted Y value in the testing dataset, using the command `y_test_pred = lr.predict(X_test)`.

Finally, we print out the coefficients on the X variables, as well as the intercept in the regression. We also print out the fitness scores of the model on the train data as well as the test data. The score is about 78%. This is how we make a linear regression model in Python.

Polynomial regression

In most cases, data does not contain a linear relationship, and we may need a more complex relationship to work with. We will look into polynomial regression in this session.

You can use a linear model to fit nonlinear data. One way to do it is to add powers to each variable as if they were new variables, in other words, new features. Then, we will train a model on these variables. This model will be linear and is called polynomial regression. Suppose you want to calculate the bonus of the employees based on how many years of experience they have on the job. You can fit a more complex, a cubical equation, with degree equals to three, which will bring the bonus levels up with seniority.

Now, let's get to the code for polynomial regression on our data in Python.

In the subfolder */ExerciseFiles/02_03*, open the Python program *02_03_Finish.py* in Spyder. The program is as follows:

```
--snip—
poly = PolynomialFeatures (degree = 2) #❶
X_poly = poly.fit_transform(X_final) #❷

X_train,X_test,y_train,y_test = train_test_split(X_poly,y_final, test_size =
0.33, random_state = 0) #❸

#standard scaler (fit transform on train, fit only on test)
sc = StandardScaler() #❹
X_train = sc.fit_transform(X_train.astype(np.float))
X_test= sc.transform(X_test.astype(np.float))

#fit model
poly_lr = LinearRegression().fit(X_train,y_train) #❺

y_train_pred = poly_lr.predict(X_train)
y_test_pred = poly_lr.predict(X_test)

#print score
print('poly train score %.3f, poly test score: %.3f' % (
poly_lr.score(X_train,y_train),
poly_lr.score(X_test, y_test))) #❻
```

You will see a polynomial model with polynomial degree equal to two is defined ❶. Then it's fitted and transformed to `X_poly` from `X_final` ❷. Then, we do a test train split ❸. Then we apply scaling to the X data ❹. Remember, we fit and transform the X train data, but we only transform the X test data. Then we will now fit the linear regression on the data. Notice that we are still fitting a linear model, but the features are polynomial here ❺. Finally we print out the test scores ❻.

The output is as follows:

```
poly train score 0.828, poly test score: 0.871
```

As we can see, the poly test score is around 87% with the quadratic fit. You may change the degree of the polynomial fit and apply it again to explore different scenarios. Let's keep in mind, in most examples we may end up with a lot of features, which will then bring the necessity of feature engineering, transformation, and selection. For instance, we may need to use feature A multiplied by feature B as a new feature, and so on. There are algorithms to automatically choose the important features. We will not cover them in this course but that's something you should keep in mind to explore the future step. We just added one more model to our analysis, polynomial regression. Degree is the key. It also affects under or over-fitting. Remember those bonus values.

Support Vector Regression (SVR)

And now, let's discuss the Support Vector Regression. This is the third model out of five models I'll show you in this course.

Imagine a bowling area. Left lane is allowed to use only blue and right lane is only allowed to use green balls. You and your friends just arrived to start your game in the middle lane. That middle lane is acting like a separator between the left and the right lanes. There is a linear separation here. Let's draw that bowling lane-like separation on the graph here. Its main goal is to create an optimal margin which can separate the maximum amount of data points.

This method was first used for the classification problems. Then later it was also applied to regression problems to predict numerical data. The plane which separates two classes is called a hyperplane. The data points which are sitting closest to the hyperplane are called the support vectors. The dashed lines are called margins. If we do not allow any of the points to be in the margin area, it's called hard margin. If we do have a tolerance, then it's called soft margin. Sometimes data is not linearly separable and it's not as simple as drawing a line between them. In these cases, we use what's called kernel tricks to make them linearly separable. When this is the case, we project the data into a higher dimension and find a hyperplane to separate them that way.

In Support Vector Regression, logic is very similar. Output is a continuous number. Goal is to minimize the error and obtain a minimum margin interval with maximum number of data points. To sum up, a few of the commonly used kernel options in Support Vector Regression are linear, RBF, poly, and exponential. They are called kernel trick or kernel functions.

In the subfolder */ExerciseFiles/02_04*, open the Python program *02_04_Finish.py* in Spyder. The program is as follows:

```
--snip—
svr = SVR(kernel='linear', C = 300) #❶

#test train split
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )  #❷

#standard scaler (fit transform on train, fit only on test)
sc = StandardScaler()  #❸
X_train = sc.fit_transform(X_train.astype(np.float))
X_test= sc.transform(X_test.astype(np.float))

#fit model
svr = svr.fit(X_train,y_train.values.ravel())  #❹
y_train_pred = svr.predict(X_train)
y_test_pred = svr.predict(X_test)

#print score
print('svr train score %.3f, svr test score: %.3f' % (
svr.score(X_train,y_train),
svr.score(X_test, y_test)))
```

The Support Vector Regressor is defined as SVR in line ❶. Then, test train data is split, as you can see, in line ❷. Since Support Vector Regression is sensitive to outliers, it's essential to apply scaling before using this method. Recall that the reason behind scaling is to prevent the features with the largest range dominating the model like our ants versus dinosaurs metaphor. So first in line ❸, we are defining a StandardScaler. Then, we fit and transform that StandardScaler to X_train, then we only transform that StandardScaler to X-test.

We fit the SVR model ❹ and do the predictions. Finally we print the score values. The output is:

```
svr train score 0.598, svr test score: 0.628
```

The SVR test score is about 63%. This is how we view the Support Vector Regressor in Python. Note that we also use the graphics for the Support Vector Classifier for better visualization. Logic is similar in the regressor as well as the commonly used kernel functions and the application of the modeling in Python.

Decision tree regression

In this video, I'll explain the decision tree algorithm. This is also called Classification and Regression Trees or CART for short.

Decisions, decisions, decisions. It's in our lives every day. Suppose we are in the middle of a decision to whether or not to purchase a car, how would we approach it? We would start with defining the most important factors or features for us to help our decision. Decision trees were first used in classification algorithms or predicting categorical variables. A decision tree is a tree where each node represents a feature or attribute, each link or branch represents the decision also called a role and each leaf represents an outcome.

Here is a decision tree example about a car purchase. Looking at the price range, if the price of a car is less than $30,000 we continue to the automatic transmission question. The car has an automatic transmission. Then, we move to the next question. Is the mileage less than 50,000? If the answer is no, then we decide not to purchase the car. If yes, the car does have less than 50,000 mileage, then we move on to the color. If we want a red car, so if the color is red we decide to purchase the car and if not, we do not purchase it.

Decision tree algorithms are similar to how we make decisions as humans. This course's focus is regression. So, in this example we will look at a decision regressor. Say we are trying to predict the number of days a car stays on the market. Here is a chart for the data with the mileage on the x-axis and the price on the y-axis. If we did not know the mileage what would be a guess of the average price? Let's say $25,000. So, we mark our chart with our line number one. This is called the root of the tree. At this point, we have either lower or higher than $25,000. The lower part is now a leaf and since we want to focus about 25,000 section, we will continue this branch.

The second split will be if the mileage is less than 6,000 or not. So, we add a line number two in the top section. Then, we will do one more split for the price in that region being larger and smaller than $34,000 and we have a line number three. The nodes we do branch out from are called the decision nodes. After the split notice that we have four areas on the chart and four corresponding leaves.

After reviewing one classifier and one regressor tree example, let's get to the code and train a decision tree regressor.

In the subfolder */ExerciseFiles/02_05*, open the Python program *02_05_Finish.py* in Spyder. The program is as follows:

```
--snip—
dt = DecisionTreeRegressor(random_state=0) #❶

#test train split
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )

#standard scaler (fit transform on train, fit only on test)
sc = StandardScaler()
X_train = sc.fit_transform(X_train.astype(np.float))
X_test= sc.transform(X_test.astype(np.float))

#fit model
dt = dt.fit(X_train,y_train.values.ravel())  #❷
y_train_pred = dt.predict(X_train)
y_test_pred = dt.predict(X_test)

#print score
print('dt train score %.3f, dt test score: %.3f' % (
dt.score(X_train,y_train),
dt.score(X_test, y_test)))  #❸
```

You will see that we are calling the decision tree regressor with random state equals to zero ❶. Then as usual we have a test train split and a scaling steps here. We do not have to repeat these steps each time if we have not changed the x-train, x-tests, y-train, and y-test, but we do that repetition to have a stand alone code which will be completed for each model for future use.

We fit the model and make predictions in the fit model section starting at ❷, then print the results in the next section ❸. The result is as follows:

```
dt train score 0.999, dt test score: 0.701
```

Okay, we see that our decision tree test score is about 70%, whereas our train score is almost 99%. Training result is turned out better than the test. So, the week side of decision trees are they're tendency to what's called over fit the model. So, the model cannot be generalized well enough for new data. For this reason, it's not uncommon to achieve higher scores in training data and lower scores in test data. Especially when we do not have enough data. The closest modeling algorithm to the human brain is decision tree. It contains conditions or nodes, branches or edges, and decisions or leaves. It's easy for us to interpret but this algorithm is prone to over fit. In other words, memorize the data which hinders the generalization of this model to new data.

Random forest regression

This is the last algorithm of the modeling section, random forest regressor.

We are moving from the one deep tree to a forest of relatively shallow trees. There comes the wisdom of the crowd with the collective opinion of the trees as opposed to a single tree. Random forest consists of multiple decision trees. It's based on ensemble learning, which means multiple learning methods are working together as a team. In other words, there's more than one decision tree in the model. Thus, all these individual trees get to cast their own vote. The main difference between regression and classification trees are, in regression trees, we take the mean. And in the classification trees, we take the mode, which is the most occurring value when making a prediction. In other words, the majority voting.

This is one of the strongest algorithms among all, so what makes this strong? I'd like to introduce you to the term bagging. It's subdividing the data into smaller components, as if your grocery bag in the store was heavy and now you are dividing the groceries into multiple bags. This method divides the data into smaller samples, and creates multiple trees. And they, as a result of the smaller sample size, are now shallower. Let's apply the bagging logic to this tree. We take smaller samples from the original dataset, and apply learning models to each tree. That way, these smaller trees are shallower than one big tree. We apply learning models to each one and finally apply aggregation, which means we take the mean, in this case, of regression. Another commonly used concept is boosting, that I would like to mention here. It's another machine learning ensemble method, primarily to reduce bias and variance.

Let's apply this to the Python code.

In the subfolder */ExerciseFiles/02_06*, open the Python program *02_06_Finish.py* in Spyder. The program is as follows:

```
--snip—
forest = RandomForestRegressor(n_estimators = 100,
                              criterion = 'mse',
                              random_state = 1,
                              n_jobs = -1)   #❶
#test train split
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )

#standard scaler (fit transform on train, fit only on test)
sc = StandardScaler()
X_train = sc.fit_transform(X_train.astype(np.float))
X_test= sc.transform(X_test.astype(np.float))

#fit model
forest.fit(X_train,y_train.values.ravel())   #❷
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)

#print score
print('forest train score %.3f, forest test score: %.3f' % (
forest.score(X_train,y_train),
forest.score(X_test, y_test)))   #❸
```

At ❶, we call a random forest regressor, and assign that to a variable *forest*. There are a few different parameters in this regressor, such as `n_estimators`, `criterion`, et cetera. Although they require a lot more time to digest, in summary, `n_estimators` is the number of trees and `criterion` is the feature selection criteria here. We are using the default one, so we can also not specify it and Python will still take it as MSE, which is the mean squared error. `n_jobs` is the number of jobs to run in parallel for both fit and predict, and `-1` means use all processors.

After splitting and scaling, we fit and predict the forest starting at ❷. Finally we print out the result ❸ as follows:

```
forest train score 0.973, forest test score: 0.859
```

Here's an important take-away. When we worked on the simple decision tree, our train score was great at 99%, but our test score was way lower, around 70%. But if you look at our forest test score, the models working together gave us a much higher test score of around 85%. This is a big improvement. There are many more parameters in the random forest regressor, such as the max leaf nodes and more. Please see the official *SKLearn* website to learn more about this. I encourage you to read the regression section, but there are also many other resources for you to learn more. With ensemble learning and using methods like bagging and boosting, random forest generalized better than a single decision tree and resulted in reduced over-fitting.

Evaluation of predictive models

Now that you've seen how to build a few regression models, we are moving on to the Evaluation section of our roadmap.

I'm going to summarize the strengths and weaknesses of each model. So far, we have used R squared as a way of measuring the success scores of the regression models. Please keep in mind that this score by itself is not enough to make decisions. It's recommended to further visualize, combine it with domain knowledge, and do further tests before making a final judgment. Now, let's take a look at each model. Linear regression has an advantage when there's a linear relationship between the independent variables and dependent variable. However, we need to keep in mind that this may become a disadvantage when we do not have a linear relationship between the independent variables and the dependent variable.

Polynomial regression can be a strong model when there's a nonlinear relationship between the independent variables and dependent variable. The degree of the polynomial regression is the key to define success here. If working with a nonoptimal degree, this may become a disadvantage.

Support vector regressions shines when we build different models with different kernels in a relatively memory efficient way. A disadvantage of this model presents itself in the occurrence of outliers in the data, especially if data scaling is not done properly.

Decision tree regression does not need data scaling, because it is resilient to outliers, and it's very intuitive to the human brain. It's simplistic nature makes this a great model to visualize, however it's very prone to overfitting, especially with smaller datasets and yields lower scores in new data. In other words, in test data.

Random forest regression is based on decision trees. This also does not need scaling and is resilient to outliers. Compared to the simple decision tree regression, overfitting is reduced due to aggregations, such as taking the mean. Since we may end up with quite a few trees compared to decision tree, this hinders the simplistic visualization.

It's important to keep in mind that in order to build a good model, we need to understand, prep, visualize, and analyze the data statistically, as well as study the relationship between variables to each other and to the target variable. Understanding the advantages and disadvantages of each model will better equip you with the knowledge of when to use which model in your predictive analytics journey. Finally, it's also important to note that this evaluation stage may not give you the scores you want from your model. When this happens, you can go back to the modeling stage to improve the model.

Hyperparameter optimization

We came a long way from preparing our data, to working with the predictive models. In most cases this is an iterative approach of evaluation and modeling until we reach a satisfactory result. We can find the best parameters by trial and error, but it will take a very long time.

Imagine trying and evaluating a model on a very large data set by changing one parameter at a time. It would work, only if we had unlimited time. This is not the most time efficient option. Instead I want to introduce you to a new term, hyperparameter optimization. This is the technique of identifying an ideal set of parameters for a prediction algorithm, which provides the optimum performance. The algorithm learns which parameter provides us with better performance by iteratively working on a pre-defined set of parameters. For example, one method traditionally used is grid search. Let's say we have parameter A and parameter B. Sometimes we may have more or less depending on the learning model. We make a grid of the search space, evaluate each hyperparameter setting we've chosen, then, repeat in as many dimensions as needed.

Some methods in addition to grid search include random search, Bayesian optimization, gradient-based optimization, evolutionary optimization, population based, and there are others as well.

In the subfolder */ExerciseFiles/02_08*, open the Python program *02_08_Finish.py* in Spyder. The program is as follows:

```
--snip—
#Function to print best hyperparamaters:
def print_best_params(gd_model):
    param_dict = gd_model.best_estimator_.get_params()
    model_str = str(gd_model.estimator).split('(')[0]
    print("\n*** {} Best Parameters ***".format(model_str))
    for k in param_dict:
        print("{}: {}".format(k, param_dict[k]))
    print()

#test train split
X_train, X_test, y_train, y_test = train_test_split(X_final, y_final,
test_size = 0.33, random_state = 0 )

#standard scaler (fit transform on train, fit only on test)
sc = StandardScaler()
X_train = sc.fit_transform(X_train.astype(np.float))
X_test= sc.transform(X_test.astype(np.float))

###Challenge 1: SVR parameter grid###
param_grid_svr = dict(kernel=[ 'linear', 'poly'],
                     degree=[2],
                     C=[600, 700, 800, 900],
                     epsilon=[0.0001, 0.00001, 0.000001])  #❶
svr = GridSearchCV(SVR(), param_grid=param_grid_svr, cv=5, verbose=3) #❷
#fit model
svr = svr.fit(X_train,y_train.values.ravel()) #❸
#print score
print('\n\nsvr train score %.3f, svr test score: %.3f' % (
svr.score(X_train,y_train),
svr.score(X_test, y_test)))
#print(svr.best_estimator_.get_params())

print_best_params(svr)
```

We first define a function `print_best_params()` for making a pretty print for the optimum parameters. After the print function, test strain, split, and scaler is already provided. So now let's define a search space for svr in the dictionary ❶. This search space includes parameters such as kernel function, degree C, and epsilon. We will give it two kernels. One is linear and the other one is poly. And we will give it degree equals to two, and we will define C values as 600, 700, 800, and 900. And let's define epsilon, which will include 0.0001, 0.00001, and 0.000001. We do the grid search using the GridSearchCV() function .❷.

It's worthwhile to do extra reading on them if you are interested but in a nutshell C is the penalty parameter of the error term and epsilon defines a margin of tolerance where no penalty is given to errors. cv is cross validation and verbose means the higher it is, the more messages we will see on the spider console.

At ❸, we fit the function using `svr.fit()` and then print the best parameter score. Remember, grid search may take a while to finish depending on the computer so reduce the number of parameters if needed.

The result is as follows:

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[CV] C=600, degree=2, epsilon=0.0001, kernel=linear .................
[CV]  C=600, degree=2, epsilon=0.0001, kernel=linear, score=0.671, total=
0.0s
[CV] C=600, degree=2, epsilon=0.0001, kernel=linear .................
[CV]  C=600, degree=2, epsilon=0.0001, kernel=linear, score=0.663, total=
0.0s
--snip--
[CV] C=900, degree=2, epsilon=1e-06, kernel=poly ....................
[CV]  C=900, degree=2, epsilon=1e-06, kernel=poly, score=0.295, total=   0.0s


svr train score 0.683, svr test score: 0.734

*** SVR Best Parameters ***
C: 700
cache_size: 200
coef0: 0.0
degree: 2
```

```
epsilon: 0.0001
gamma: scale
kernel: linear
max_iter: -1
shrinking: True
tol: 0.001
verbose: False

[Parallel(n_jobs=1)]: Done 120 out of 120 | elapsed:    2.5s finished
```

And when we look at the results we see svr train score is about 68% and test score is about 73%. We see a noticeable improvement compared to the prior result.

Challenge: Hyperparameter optimization

Now that you know about hyperparameter optimization, it's your turn, I have a challenge for you.

In this challenge, I want you to first create a parameter grid for decision tree and random forest. Then, you will create the GridSearchCV with the grid created in step one. Finally, you will run the fit model and print out the best parameters and the test scores. To do this, remember to use the scikit-learn resource to look up your parameters. Finally, let me show you the begin file for this challenge. When you open the file *02_10_Solution_Begin.py*, you will see that I have included the commented out parameter grid with parameters. Let's scroll down to see that. So, here it is, starting on line 328. This isn't complete, so you will need to replace the asterisks with what you think should go there. Also, the GridSearch is provided on line 333.

Good luck, and I'll meet you back at the solution video.

Solution: Hyperparameter optimization

Here is how I solved the hyperparameter optimization challenge.

In the subfolder */ExerciseFiles/02_10*, open the Python program *02_10_Solution_Finish.py* in Spyder. The program is as follows:

```
--snip—
###Challenge 2:Decision Tree parameter grid###
param_grid_dt = dict(min_samples_leaf=np.arange(9, 13, 1, int),
                    max_depth = np.arange(4,7,1, int),
                    min_impurity_decrease = [0, 1, 2],)   #1
```

```
dt = GridSearchCV(DecisionTreeRegressor(random_state=0),
param_grid=param_grid_dt, cv=5,  verbose=3)  #❷

#fit model
dt = dt.fit(X_train,y_train.values.ravel())  #❸

#print score
print('\n\ndt train score %.3f, dt test score: %.3f' % (
dt.score(X_train,y_train),
dt.score(X_test, y_test)))
print_best_params(dt)  #❹

###Challenge 3:Random Forest parameter grid###
param_grid_rf = dict(n_estimators=[20],
                     max_depth=np.arange(1, 13, 2),
                     min_samples_split=[2],
                     min_samples_leaf= np.arange(1, 15, 2, int),
                     bootstrap=[True, False],
                     oob_score=[False, ])  #❺

forest = GridSearchCV(RandomForestRegressor(random_state=0),
param_grid=param_grid_rf, cv=5, verbose=3)

#fit model
forest.fit(X_train,y_train.values.ravel())

#print score
print('\n\nforest train score %.3f, forest test score: %.3f' % (
forest.score(X_train,y_train),
forest.score(X_test, y_test)))

print_best_params(forest)
```

First, let's look at decision tree ❶. Minimum samples leaf is the minimum number of samples required to be at the leaf node. So I will add values between nine through 13 with step size of one. Maximum depth is the maximum depth of the tree. We will give the values between four and seven with step size of one. And minimum impurity decrease is a node will be split if this split induces a decrease of the impurity greater than or equal to this value. Let's assign it values of zero, one, and two. Let's put these in the grid search with random state equals zero ❷. We fit the model ❸ and print the scores ❹.

Please keep in mind that it may take a while depending on the computer you are using. Let's look at the scores. So the scores of train and test are printed right here. And the best parameters are as shown below:

```
--snip—

dt train score 0.856, dt test score: 0.880


*** DecisionTreeRegressor Best Parameters ***
ccp_alpha: 0.0
criterion: mse
max_depth: 5
max_features: None
max_leaf_nodes: None
min_impurity_decrease: 0
min_impurity_split: None
min_samples_leaf: 12
min_samples_split: 2
min_weight_fraction_leaf: 0.0
presort: deprecated
random_state: 0
splitter: best

[Parallel(n_jobs=1)]: Done 180 out of 180 | elapsed:    0.2s finished
```

So let's repeat these steps for Random Forest ❺. Let's select the parameter grid section. So the n_estimators is the number of trees in the forest. So let's give this one 20. We can always give a larger interval here, but also be mindful of the computational cost as it may get expensive. Maximum depth is the depth of each tree. We will give that values between one through 13 with the interval of two. Minimum samples split is the minimum number of samples required to split an internal node. We will give that two, which is the default value, because we need minimum two observations to consider a split, and increasing the number will increase the computational time. Minimum samples leaf is the minimum number of samples required to be at a leaf node. We will give values between one through 15 with an interval of two. Next, we have bootstrap. Bootstrap is whether bootstrap samples are used when building trees. If false, the whole dataset is used to build each tree. This is the bagging concept we discussed. Remember the heavy grocery bag divided into multiple bags example? We will give true and false to investigate both options. And finally, oob_score is whether to use out of bag samples to estimate the r squared on unseen data. We will assign this as false. We will then pass this parameter grid we just created to the grid search and create the forest.

The result is as follows:

```
--snip--
forest train score 0.864, forest test score: 0.892

*** RandomForestRegressor Best Parameters ***
bootstrap: True
ccp_alpha: 0.0
criterion: mse
max_depth: 5
max_features: auto
max_leaf_nodes: None
max_samples: None
min_impurity_decrease: 0.0
min_impurity_split: None
min_samples_leaf: 7
min_samples_split: 2
min_weight_fraction_leaf: 0.0
n_estimators: 20
n_jobs: None
oob_score: False
random_state: 0
verbose: 0
warm_start: False

[Parallel(n_jobs=1)]: Done 420 out of 420 | elapsed:   11.4s finished
```

So we've got 86% for train and 89% for the test score. And these are our best parameters. So this is how we do a basic hyperparameter optimization through a grid search in Python.

Next steps

Congrats. You have completed the course. Thank you for the privilege of being your instructor and for surviving my metaphors. But this is just the beginning of a long and fun journey to predictive analytics. We scratched the surface using Python with regression problems and I encourage you to find a dataset at Kaggle.com and practice the steps you have learned during this class.

Practice, practice, practice. Get your hands wet with as much data as you can. I highly recommend watching the data analytics and Python classes we have here at LinkedIn Learning. I also strongly recommend the book *Hands On Machine Learning with Scikit-Learn and TensorFlow*. Once you have a few projects completed on regression, a good next step would be to move on to classification algorithms. Read more on our helpful scikit-learn website we referenced throughout this course. Feel free to reach out to me on LinkedIn.com or from my tech consulting website at ibuniversal.com. See you next time and remember to have fun while exploring your data.