Model Validation in Python

How good is your machine learning model?

[](https://medium.com/@jnyh?source=post_page-----ad23c1d215b----------------------)

[Black Raven (James Ng)](https://medium.com/@jnyh?source=post_page-----ad23c1d215b----------------------)

[Aug 14](https://medium.com/@jnyh/model-validation-in-python-ad23c1d215b?source=post_page-----ad23c1d215b----------------------) · 27 min read

This is a memo to share what I have learnt in Model Validation (using Python), capturing the learning objectives as well as my personal notes. The course is taught by Kasey Jones from DataCamp, and it includes 4 chapters:

Chapter 1. Basic Modeling in scikit-learn  
Chapter 2. Validation Basics  
Chapter 3. Cross Validation  
Chapter 4. Selecting the best model with Hyperparameter tuning



Photo by [Craige McGonigle](https://unsplash.com/@craigewmcg123?utm_source=medium&utm_medium=referral" \t "_blank) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

Machine learning models are easier to implement now more than ever before. Without proper validation, the results of running new data through a model might not be as accurate as expected.

Model validation allows analysts to confidently answer the question, how good is your model? We will answer this question for classification models using the complete set of tic-tac-toe endgame scenarios, and for regression models using fivethirtyeight’s ultimate Halloween candy power ranking dataset.

In this course, we will cover the basics of model validation, discuss various validation techniques, and begin to develop tools for creating validated and high performing models.

**Chapter 1. Basic Modeling in scikit-learn**

Before we can validate models, we need an understanding of how to create and work with them. This chapter provides an introduction to running regression and classification models in scikit-learn. We will use this model building foundation throughout the remaining chapters.

**Introduction to model validation**

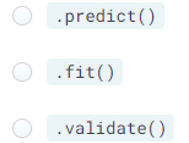
Model validation is done to choose the right model, the best parameters, and even the best performance metric. The goal is to ensure model performs as expected on new (unseen or holdout) data.

Modeling steps in scikit-learn:  
1. instantiate a model, specifying its type and parameters  
2. fit the model, ie, train the model on training data X\_train and y\_train  
3. use the trained model to generate predictions on test data  
4. measure model performance and accuracy by comparing true values vs predicted values

**Modeling steps**

The process of using scikit-learn to create and test models has four steps, and you will use these four steps throughout this course:

Which of the following is **NOT** a valid method in the four-step scikit-learn model validation framework?



Answer: Validation is a technique all in its own and is not done with .validate(). You need to learn a few tools and techniques before you can validate a model.

**Seen vs. unseen data**

Model’s tend to have higher accuracy on observations they have seen before. In the candy dataset, predicting the popularity of Skittles will likely have higher accuracy than predicting the popularity of Andes Mints; Skittles is in the dataset, and Andes Mints is not.

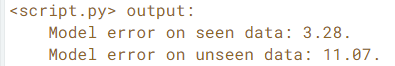
You’ve built a model based on 50 candies using the dataset X\_train and need to report how accurate the model is at predicting the popularity of the 50 candies the model was built on, and the 35 candies (X\_test) it has never seen. You will use the mean absolute error, mae(), as the accuracy metric.

# The model is fit using X\_train and y\_train  
model.fit(X\_train, y\_train)

# Create vectors of predictions  
train\_predictions = model.predict(X\_train)  
test\_predictions = model.predict(X\_test)

# Train/Test Errors  
train\_error = mae(y\_true=y\_train, y\_pred=train\_predictions)  
test\_error = mae(y\_true=y\_test, y\_pred=test\_predictions)

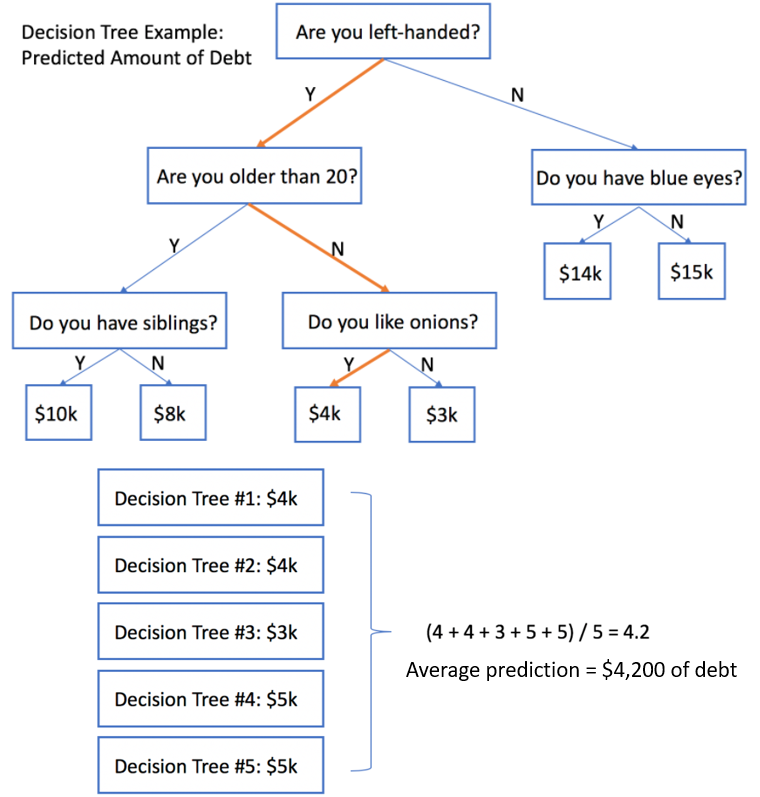
# Print the accuracy for seen and unseen data  
print("Model error on seen data: {0:.2f}.".format(train\_error))  
print("Model error on unseen data: {0:.2f}.".format(test\_error))



When models perform differently on training and testing data, you should look to model validation to ensure you have the best performing model.

**Regression models**

RandomForestRegressor is a regression model built for continuous variables.

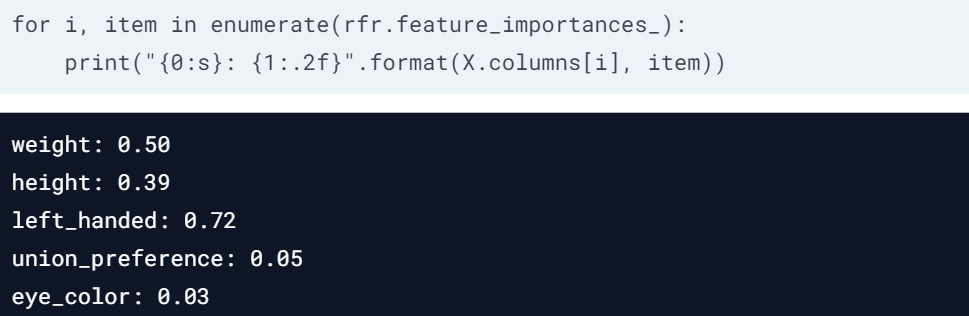


Decision Tree and Random Forest example

Random forest regression models generate a bunch of different decision trees and use the mean prediction of the decision trees as the final value for a new observation.

Focus on 3 parameters of random forest:  
· n\_estimators = number of trees in the random forest  
· max\_depth = how many layers to split the data, to end nodes  
· random\_state = random seed to create reproducible models

Feature importance = how much influence each feature has on the model



left\_handed is an important feature; eye\_color is not useful

**Set parameters and fit a model**

Predictive tasks fall into one of two categories: regression or classification. In the candy dataset, the outcome is a *continuous* variable describing how often the candy was chosen over another candy in a series of 1-on-1 match-ups. To predict this value (the win-percentage), you will use a **regression** model.

In this exercise, you will specify a few parameters using a random forest regression model rfr.

from sklearn.ensemble import RandomForestRegressor  
rfr = RandomForestRegressor()

# Set the number of trees  
rfr.n\_estimators = 100

# Add a maximum depth  
rfr.max\_depth = 6

# Set the random state  
rfr.random\_state = 1111

# Fit the model  
rfr.fit(X\_train, y\_train)

This approach is helpful when you need to update parameters after the model was initialised.

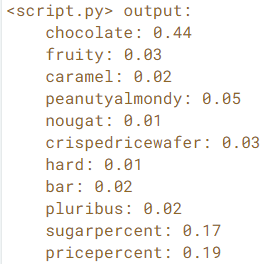
**Feature importances**

Although some candy attributes, such as chocolate, may be extremely popular, it doesn’t mean they will be *important* to model prediction. After a random forest model has been fit, you can review the model’s attribute, .feature\_importances\_, to see which variables had the biggest impact. You can check how important each variable was in the model by looping over the feature importance array using enumerate().

If you are unfamiliar with Python’s enumerate() function, it can loop over a list while also creating an automatic counter.

# Fit the model using X and y  
rfr.fit(X\_train, y\_train)

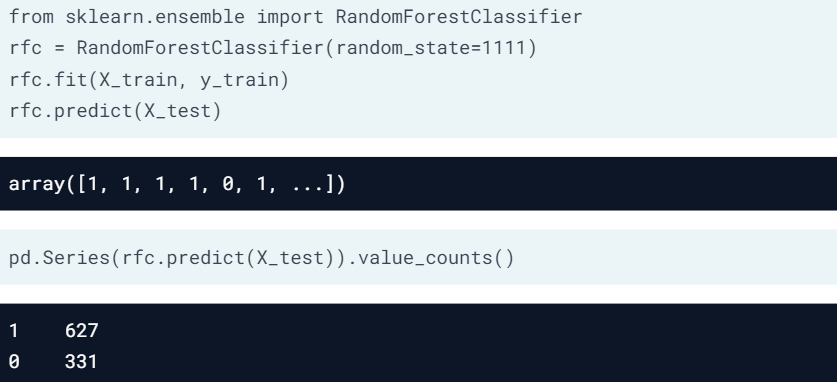
# Print how important each column is to the model  
# Use i & item to print out feature importance of each column  
for i, item in enumerate(rfr.feature\_importances\_):  
 print("{0:s}: {1:.2f}".format(X\_train.columns[i], item))



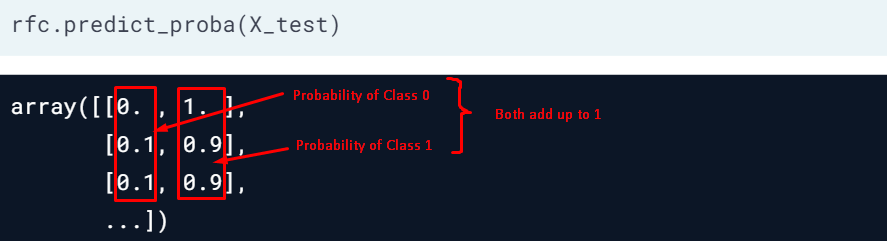
No surprise here — chocolate *is* the most important variable. .feature\_importances\_ is a great way to see which variables were important to your random forest model.

**Classification models**

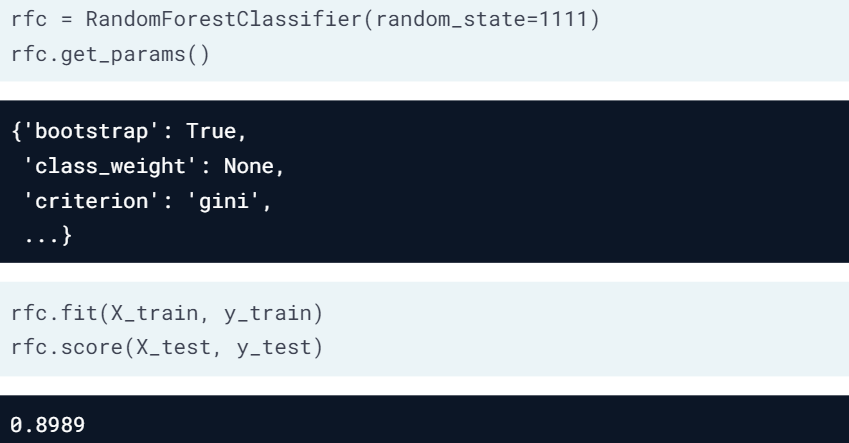
RandomForestClassifier is a classification model built for categorical variables.



Using .predict() for classification

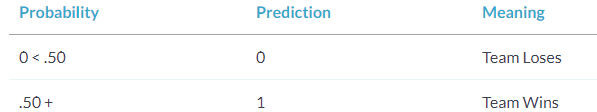


Using .predict\_proba() to get probabilities



**Classification predictions**

In model validation, it is often important to know more about the predictions than just the final classification. When predicting who will win a game, most people are also interested in *how likely* it is a team will win.



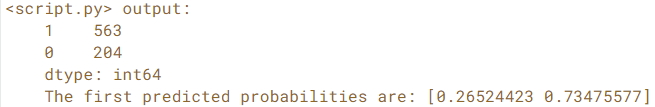
In this exercise, you look at the methods, .predict() and .predict\_proba() using the tic\_tac\_toe dataset. The first method will give a prediction of whether Player One will win the game, and the second method will provide the probability of Player One winning. Use rfc as the random forest classification model.

# Fit the rfc model.   
rfc.fit(X\_train, y\_train)

# Create arrays of predictions  
classification\_predictions = rfc.predict(X\_test)  
probability\_predictions = rfc.predict\_proba(X\_test)

# Print out count of binary predictions  
print(pd.Series(classification\_predictions).value\_counts())

# Print the first value from probability\_predictions  
print('The first predicted probabilities are:   
 {}'.format(probability\_predictions[0]))



You can see there were 563 observations where Player One was predicted to win the Tic-Tac-Toe game. Also, note that the predicted\_probabilities array contains lists with only two values because you only have two possible responses (win or lose).

**Reusing model parameters**

Replicating model performance is vital in model validation. Replication is also important when sharing models with co-workers, reusing models on new data or asking questions on a website such as [**Stack Overflow**](https://stackoverflow.com/). You might use such a site to ask other coders about model errors, output, or performance. The best way to do this is to replicate your work by reusing model parameters.

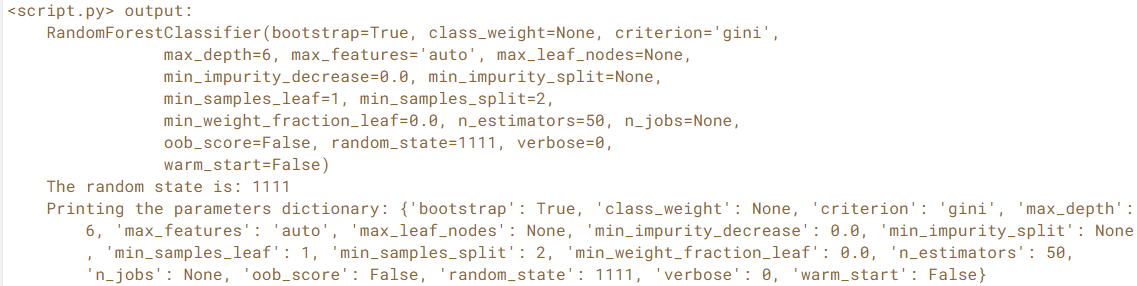
In this exercise, you use various methods to recall which parameters were used in a model.

rfc = RandomForestClassifier(n\_estimators=50, max\_depth=6, random\_state=1111)

# Print the classification model  
print(rfc)

# Print the classification model's random state parameter  
print('The random state is: {}'.format(rfc.random\_state))

# Print all parameters  
print('Printing the parameters dictionary:   
 {}'.format(rfc.get\_params()))



Recalling which parameters were used will be helpful going forward. Model validation and performance rely heavily on which parameters were used, and there is no way to replicate a model without keeping track of the parameters used!

**Random forest classifier**

This exercise reviews the four modeling steps discussed throughout this chapter using a random forest classification model. You will:

1. Create a random forest classification model.
2. Fit the model using the tic\_tac\_toe dataset.
3. Make predictions on whether Player One will win (1) or lose (0) the current game.
4. Finally, you will evaluate the overall accuracy of the model.

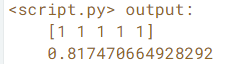
from sklearn.ensemble import RandomForestClassifier

# Create a random forest classifier  
rfc = RandomForestClassifier(n\_estimators=50,   
 max\_depth=6, random\_state=1111)

# Fit rfc using X\_train and y\_train  
rfc.fit(X\_train, y\_train)

# Create predictions on X\_test  
predictions = rfc.predict(X\_test)  
print(predictions[0:5])

# Print model accuracy using score() and the testing data  
print(rfc.score(X\_test, y\_test))

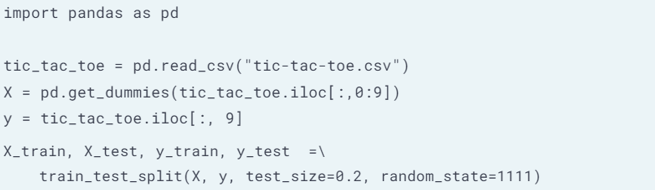


Notice the first five predictions were all 1, indicating that Player One is predicted to win all five of those games. You also see the model accuracy was only 82%.

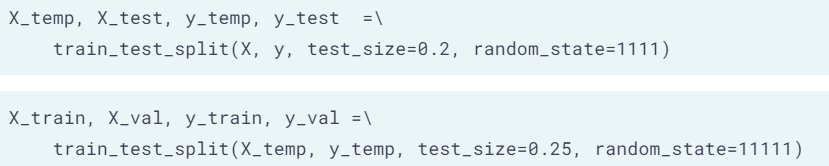
**Chapter 2. Validation Basics**

This chapter focuses on the basics of model validation. From splitting data into training, validation, and testing datasets, to creating an understanding of the bias-variance tradeoff, we build the foundation for the techniques of K-Fold and Leave-One-Out validation practiced in chapter three.

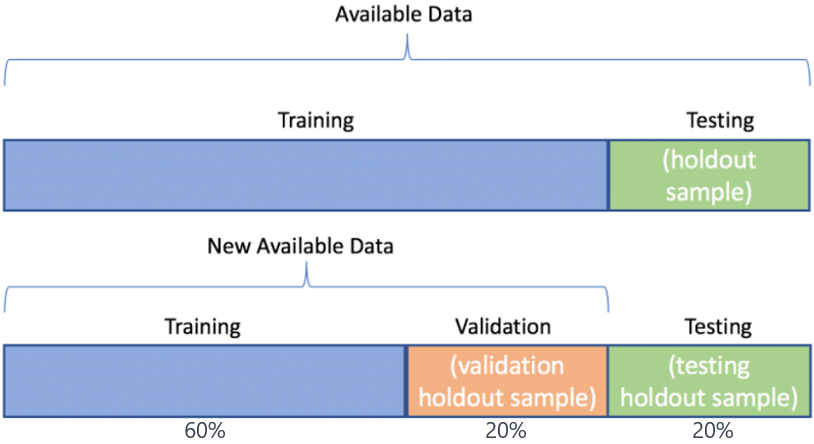
**Creating train, test, and validation datasets**



Train-Test split ratio examples:  
· 80:20 is most commonly used  
· 90:10 used when we have little data  
· 70:30 used when model is computationally expensive



Applying train\_test\_split twice to get training, validation, and testing datasets



Training dataset is used to fit the model. Validation holdout sample (orange) is used to validate model when tuning hyperparameters. Testing holdout sample (green) is the unseen data used to assess the performance of the final trained model.

**Create one holdout set**

Your boss has asked you to create a simple random forest model on the tic\_tac\_toe dataset. She doesn't want you to spend much time selecting parameters; rather she wants to know how well the model will perform on future data. For future Tic-Tac-Toe games, it would be nice to know if your model can predict which player will win.

The dataset tic\_tac\_toe has been loaded for your use.

Note that in Python, =\ indicates the code was too long for one line and has been split across two lines.

# Create dummy variables using pandas  
X = pd.get\_dummies(tic\_tac\_toe.iloc[:,0:9])  
y = tic\_tac\_toe.iloc[:, 9]

# Create training and testing datasets. Use 10% for the test set  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,   
 test\_size=0.1, random\_state=1111)

Note: Without the holdout set, you cannot truly validate a model.

**Create two holdout sets**

You recently created a simple random forest model to predict Tic-Tac-Toe game wins for your boss, and at her request, you did not do any parameter tuning. Unfortunately, the overall model accuracy was too low for her standards. This time around, she has asked you to focus on model performance.

Before you start testing different models and parameter sets, you will need to split the data into training, validation, and testing datasets. Remember that after splitting the data into training and testing datasets, the validation dataset is created by splitting the training dataset.

The datasets X and y have been loaded for your use.

# Create temporary training and final testing datasets  
X\_temp, X\_test, y\_temp, y\_test =\  
 train\_test\_split(X, y, test\_size=0.2, random\_state=1111)

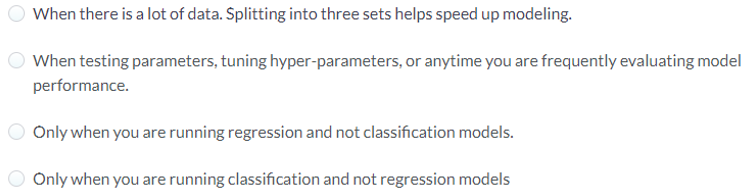
# Create the final training and validation datasets  
X\_train, X\_val, y\_train, y\_val =\  
 train\_test\_split(X\_temp, y\_temp,   
 test\_size=0.25, random\_state=1111)

After applying train\_test\_split twice, you now have training, validation, and testing datasets.

**Why use holdout sets**

It is important to understand when you would use three datasets (training, validation, and testing) instead of two (training and testing). There is no point in creating an additional dataset split if you are not going to use it.

When should you consider using training, validation, *and* testing datasets?

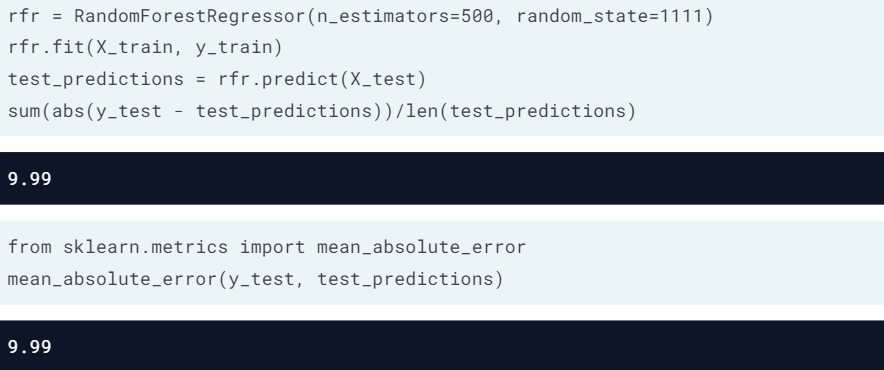


Answer: When testing parameters, tuning hyper-parameters, or anytime you are frequently evaluating model performance. Anytime we are evaluating model performance repeatedly we need to create training, validation, and testing datasets.

**Accuracy metrics: regression models**

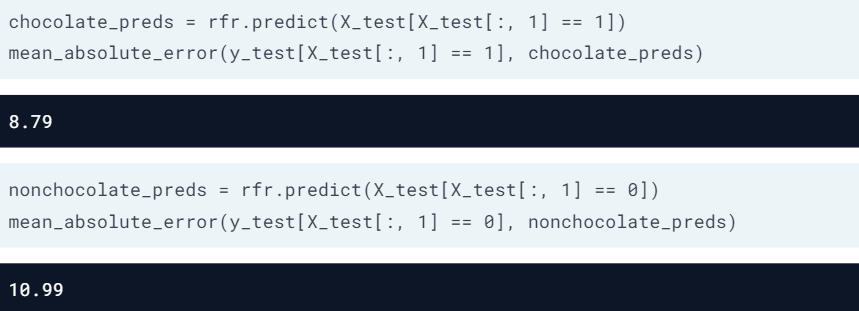
Regression models are built for continuous variables.

**Mean Absolute Error**(MAE) = average absolute difference between predictions and actual values, treats all points equally, not sensitive to outliers.



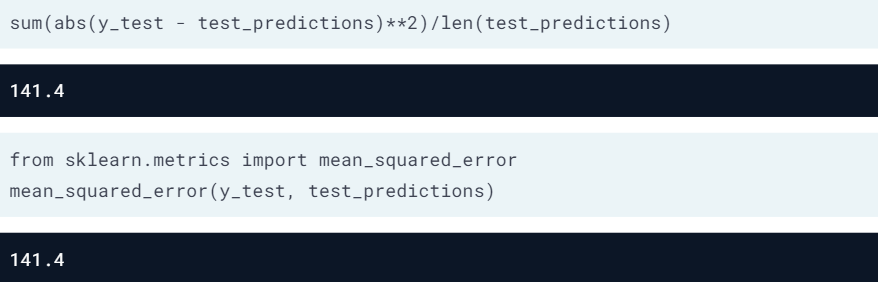
This error means that we are about 10% off on average when predicting the win-percentage.

Model performance (accuracy) for a subset of data



The model performed better on on chocolate candy.

**Mean Squared Error**(MSE) = average squared difference between predictions and actual values, allows larger errors (eg. outliers) to have a larger impact on the model



Outliers have more impact on the model’s performance.

**Mean absolute error**

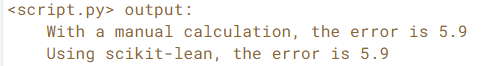
Communicating modeling results can be difficult. However, most clients understand that on average, a predictive model was off by some number. This makes explaining the mean absolute error easy. For example, when predicting the number of wins for a basketball team, if you predict 42, and they end up with 40, you can easily explain that the error was two wins.

In this exercise, you are interviewing for a new position and are provided with two arrays. y\_test, the true number of wins for all 30 NBA teams in 2017 and predictions, which contains a prediction for each team. To test your understanding, you are asked to both manually calculate the MAE and use sklearn.

from sklearn.metrics import mean\_absolute\_error

# Manually calculate the MAE  
n = len(predictions)  
mae\_one = sum(abs(y\_test - predictions)) / n  
print('With a manual calculation, the error is {}'.format(mae\_one))

# Use scikit-learn to calculate the MAE  
mae\_two = mean\_absolute\_error(y\_test, predictions)  
print('Using scikit-lean, the error is {}'.format(mae\_two))



These predictions were about six wins off on average. This isn’t too bad considering NBA teams play 82 games a year.

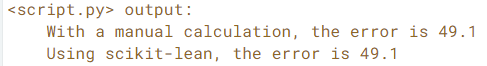
**Mean squared error**

Let’s focus on the 2017 NBA predictions again. Every year, there are at least a couple of NBA teams that win *way* more games than expected. If you use the MAE, this accuracy metric does not reflect the bad predictions as much as if you use the MSE. Squaring the large errors from bad predictions will make the accuracy look worse.

In this example, NBA executives want to better predict team wins. You will use the mean squared error to calculate the prediction error. The actual wins are loaded as y\_test and the predictions as predictions.

from sklearn.metrics import mean\_squared\_errorn = len(predictions)  
# Finish the manual calculation of the MSE  
mse\_one = sum((y\_test - predictions)\*\*2) / n  
print('With a manual calculation, the error is {}'.format(mse\_one))

# Use the scikit-learn function to calculate MSE  
mse\_two = mean\_squared\_error(y\_test, predictions)  
print('Using scikit-lean, the error is {}'.format(mse\_two))



If you run any additional models, you will try to beat an MSE of 49.1, which is the average squared error of using your model. Although the MSE is not as interpretable as the MAE, it will help us select a model that has fewer ‘large’ errors.

**Performance on data subsets**

In professional basketball, there are two conferences, the East and the West. Coaches and fans often only care about how teams in their own conference will do this year.

You have been working on an NBA prediction model and would like to determine if the predictions were better for the East or West conference. You added a third array to your data called labels, which contains an "E" for the East teams, and a "W" for the West.

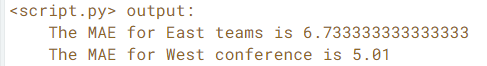
y\_test and predictions have again been loaded for your use.  
The mean\_absolute\_error function has been loaded as mae.  
The variable west\_error contains the MAE for the West teams.

# Find the East conference teams  
east\_teams = labels == "E"

# Create arrays for the true and predicted values  
true\_east = y\_test[east\_teams]  
preds\_east = predictions[east\_teams]

# Print the accuracy metrics  
print('The MAE for East teams is {}'.format(  
 mae(true\_east, preds\_east)))

# Print the West accuracy  
west\_error = mae(y\_test[labels=="W"], predictions[labels=="W"])  
print('The MAE for West conference is {}'.format(west\_error))



It looks like the Western conference predictions were about two games better on average. Over the past few seasons, the Western teams have generally won the same number of games as the experts have predicted. Teams in the East are just not as predictable as those in the West.

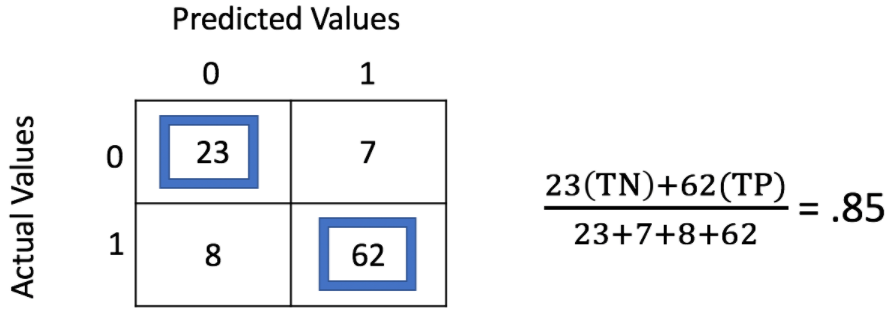
**Classification metrics**

Classification models predict what category an observation falls into.  
Confusion Matrix displays true (correct) and false (wrong) predictions.

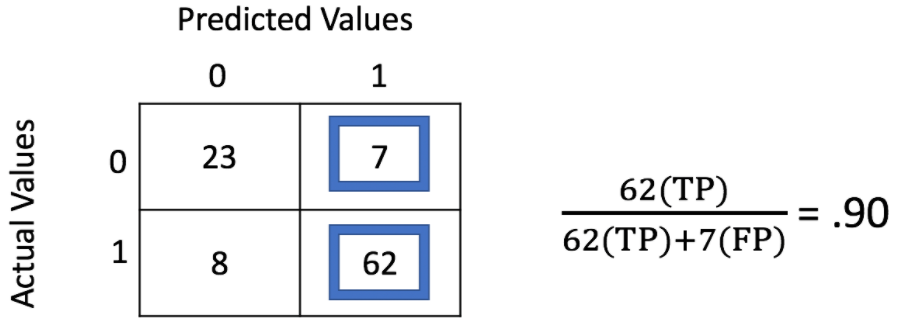


Classification metrics: Accuracy, Precision, Recall

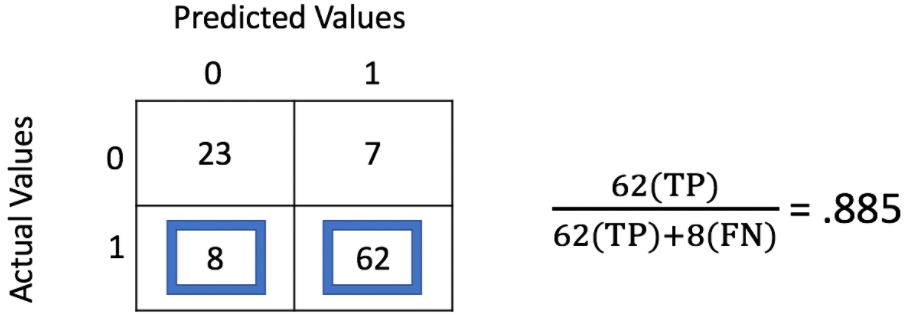
· **Accuracy**= (TN+TP) / (TN+TP+FN+FP)

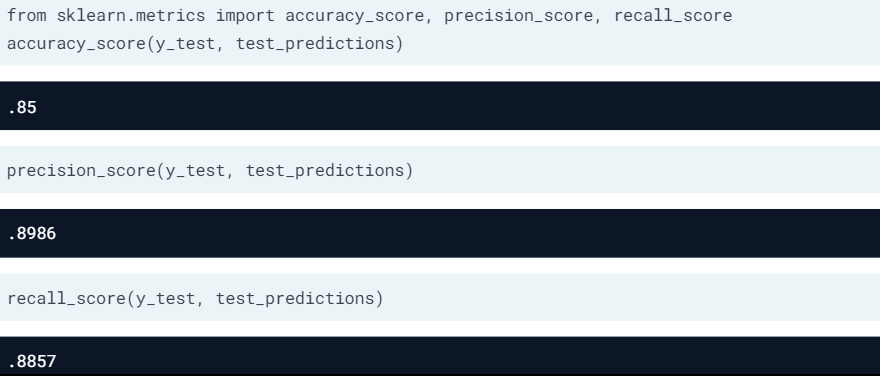


· **Precision** = TP / (TP+FP)  
Precision is used when we do not want to overpredict positive values.



· **Recall**(also called sensitivity)  
Recall is used when we cannot afford to miss out any positive values.





**Confusion matrices**

Confusion matrices are a great way to start exploring your model’s accuracy. They provide the values needed to calculate a wide range of metrics, including sensitivity, specificity, and the F1-score.

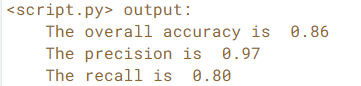
You have built a classification model to predict if a person has a broken arm based on an X-ray image. On the testing set, you have the following confusion matrix:



# Calculate and print the accuracy  
accuracy = (324 + 491) / (953)  
print("The overall accuracy is {0: 0.2f}".format(accuracy))

# Calculate and print the precision  
precision = (491) / (15 + 491)  
print("The precision is {0: 0.2f}".format(precision))

# Calculate and print the recall  
recall = (491) / (123 + 491)  
print("The recall is {0: 0.2f}".format(recall))



In this case, a true positive is a picture of an actual broken arm that was also predicted to be broken. Doctors are okay with a few additional false positives (predicted broken, not actually broken), as long as you don’t miss anyone who needs immediate medical attention.

**Confusion matrices, again**

Creating a confusion matrix in Python is simple. The biggest challenge will be making sure you understand the orientation of the matrix. This exercise makes sure you understand the sklearn implementation of confusion matrices. Here, you have created a random forest model using the tic\_tac\_toe dataset rfc to predict outcomes of 0 (loss) or 1 (a win) for Player One.

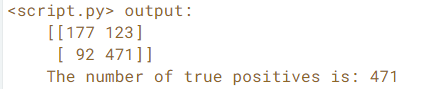
*Note:* If you read about confusion matrices on another website or for another programming language, the values might be reversed.

from sklearn.metrics import confusion\_matrix

# Create predictions  
test\_predictions = rfc.predict(X\_test)

# Create and print the confusion matrix  
cm = confusion\_matrix(y\_test, test\_predictions)  
print(cm)

# Print the true positives (actual 1s that were predicted 1s)  
print("The number of true positives is: {}".format(cm[1, 1]))



Row 1, column 1 represents the number of actual 1s that were predicted 1s (the true positives). Always make sure you understand the orientation of the confusion matrix before you start using it!

**Precision vs. recall**

The accuracy metrics you use to evaluate your model should *always* be based on the specific application. For this example, let’s assume you are a really sore loser when it comes to playing Tic-Tac-Toe, but only when you are certain that you are going to win.

Choose the most appropriate accuracy metric, either precision or recall, to complete this example. But remember, *if you think you are going to win, you better win!*

Use rfc, which is a random forest classification model built on the tic\_tac\_toe dataset.

from sklearn.metrics import precision\_scoretest\_predictions = rfc.predict(X\_test)

# Create precision or recall score based on the metric you imported  
score = precision\_score(y\_test, test\_predictions)

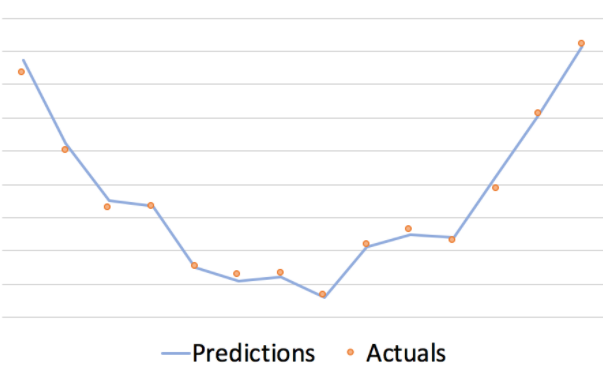
# Print the final result  
print("The precision value is {0:.2f}".format(score))

Image for post

Precision is the correct metric here. Sore-losers can’t stand losing when they are certain they will win! For that reason, our model needs to be as precise as possible. With a precision of only 79%, you may need to try some other modeling techniques to improve this score.

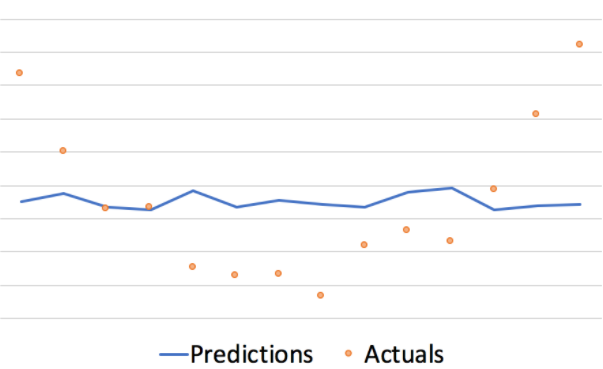
**The bias-variance tradeoff**

**Variance**occurs when a model pays too close attention to the training data (performs well), and fails to generalise to the testing data (poor test score), ie, **overfitting**= model attaches meaning to the noise in the training data.

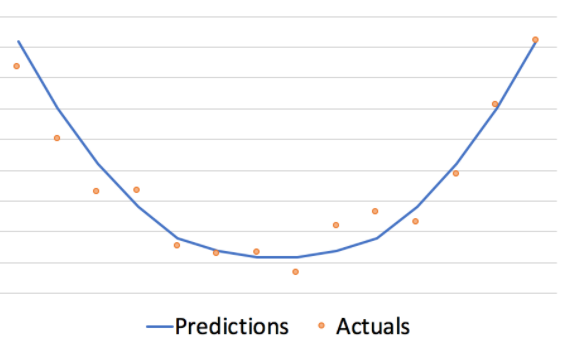


Overfitting: training error < testing error

**Bias**occurs when a model fails to find the relationships between the data and the response value. Bias leads to high errors on both the training and testing datasets, ie, **underfitting**= model could not find the underlying patterns available in the data.



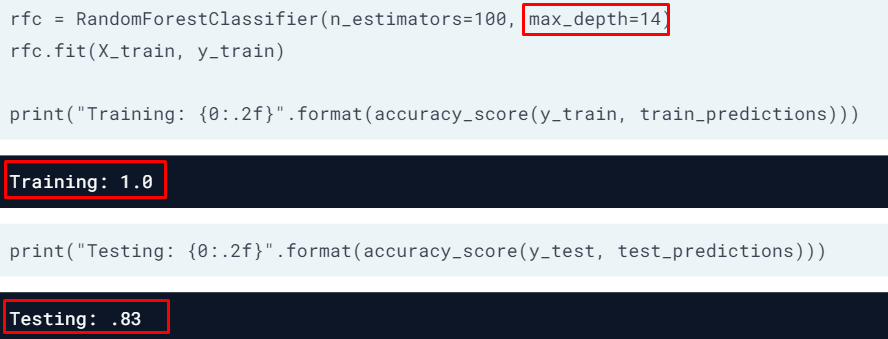
Underfitting: both training error and testing error are high



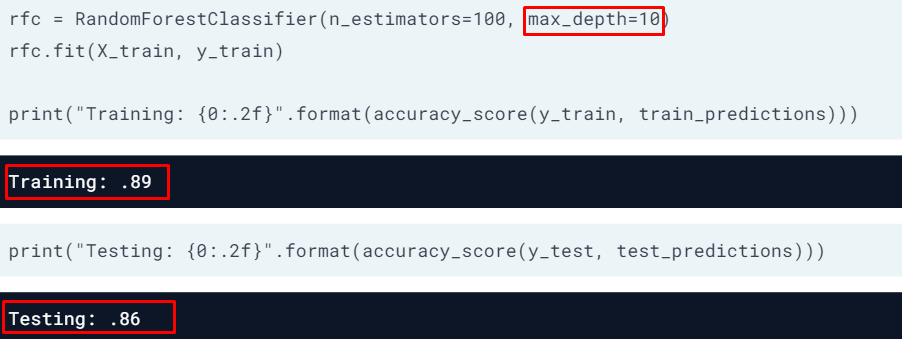
Optimal performance



max\_depth = 4 is probably not deep enough



max\_depth too large and overfitting



model has high accuracy on train data, and is generalising well on test data

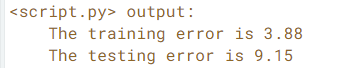
**Error due to under/over-fitting**

The candy dataset is prime for overfitting. With only 85 observations, if you use 20% for the testing dataset, you are losing a lot of vital data that could be used for modeling. Imagine the scenario where most of the chocolate candies ended up in the training data and very few in the holdout sample. Our model might *only* see that chocolate is a vital factor, but fail to find that other attributes are also important. In this exercise, you’ll explore how using too many features (columns) in a random forest model can lead to overfitting.

A *feature* represents which columns of the data are used in a decision tree. The parameter max\_features limits the number of features available.

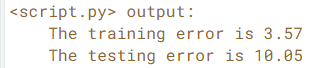
# Update the rfr model  
rfr = RandomForestRegressor(n\_estimators=25,  
 random\_state=1111,  
 max\_features=2)  
rfr.fit(X\_train, y\_train)

# Print the training and testing accuracies   
print('The training error is {0:.2f}'.format(  
 mae(y\_train, rfr.predict(X\_train))))  
print('The testing error is {0:.2f}'.format(  
 mae(y\_test, rfr.predict(X\_test))))



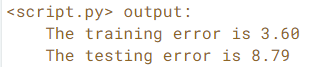
# Update the rfr model  
rfr = RandomForestRegressor(n\_estimators=25,  
 random\_state=1111,  
 max\_features=11)  
rfr.fit(X\_train, y\_train)

# Print the training and testing accuracies   
print('The training error is {0:.2f}'.format(  
 mae(y\_train, rfr.predict(X\_train))))  
print('The testing error is {0:.2f}'.format(  
 mae(y\_test, rfr.predict(X\_test))))

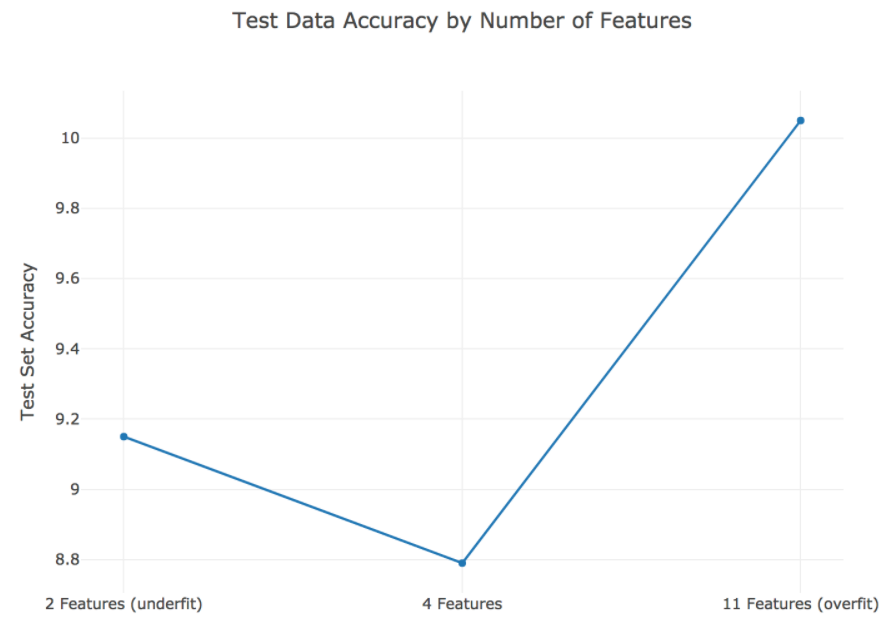


# Update the rfr model  
rfr = RandomForestRegressor(n\_estimators=25,  
 random\_state=1111,  
 max\_features=4)  
rfr.fit(X\_train, y\_train)

# Print the training and testing accuracies   
print('The training error is {0:.2f}'.format(  
 mae(y\_train, rfr.predict(X\_train))))  
print('The testing error is {0:.2f}'.format(  
 mae(y\_test, rfr.predict(X\_test))))



The chart below shows the performance at various max feature values. Sometimes, setting parameter values can make a huge difference in model performance.

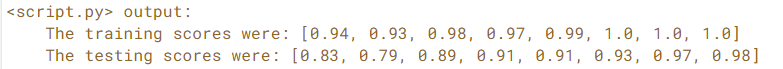


**Am I underfitting?**

You are creating a random forest model to predict if you will win a future game of Tic-Tac-Toe. Using the tic\_tac\_toe dataset, you have created training and testing datasets, X\_train, X\_test, y\_train, and y\_test.

You have decided to create a bunch of random forest models with varying amounts of trees (1, 2, 3, 4, 5, 10, 20, and 50). The more trees you use, the longer your random forest model will take to run. However, if you don’t use enough trees, you risk underfitting. You have created a for loop to test your model at the different number of trees.

from sklearn.metrics import accuracy\_scoretest\_scores, train\_scores = [], []  
for i in [1, 2, 3, 4, 5, 10, 20, 50]:  
 rfc = RandomForestClassifier(n\_estimators=i, random\_state=1)  
 rfc.fit(X\_train, y\_train)  
 # Create predictions for the X\_train and X\_test datasets  
 train\_predictions = rfc.predict(X\_train)  
 test\_predictions = rfc.predict(X\_test)  
 # Append the accuracy score for the test and train predictions  
 train\_scores.append(round(accuracy\_score(y\_train,   
 train\_predictions), 2))  
 test\_scores.append(round(accuracy\_score(y\_test,   
 test\_predictions), 2))  
# Print the train and test scores.  
print("The training scores were: {}".format(train\_scores))  
print("The testing scores were: {}".format(test\_scores))



Notice that with only one tree, both the train and test scores are low. As you add more trees, both errors improve. Even at 50 trees, this still might not be enough. Every time you use more trees, you achieve higher accuracy. At some point though, more trees increase training time, but do not decrease testing error.

**Chapter 3. Cross Validation**

Holdout sets are a great start to model validation. However, using a single train and test set is often not enough. Cross-validation is considered the gold standard when it comes to validating model performance and is almost always used when tuning model hyper-parameters. This chapter focuses on performing cross-validation to validate model performance.

**Two samples**

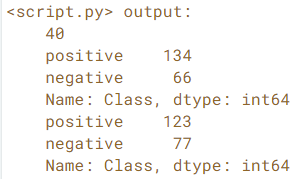
After building several classification models based on thetic\_tac\_toe dataset, you realize that some models do not generalize as well as others. You have created training and testing splits just as you have been taught, so you are curious why your validation process is not working.

After trying a different training, test split, you noticed differing accuracies for your machine learning model. Before getting too frustrated with the varying results, you have decided to see what else could be going on.

# Create two different samples of 200 observations   
sample1 = tic\_tac\_toe.sample(200, random\_state=1111)  
sample2 = tic\_tac\_toe.sample(200, random\_state=1171)

# Print the number of common observations   
print(len([index for index in sample1.index if index in sample2.index]))

# Print the number of observations in the Class column for both samples   
print(sample1['Class'].value\_counts())  
print(sample2['Class'].value\_counts())



Notice that there are a varying number of positive observations for both sample test sets. Sometimes creating a single test holdout sample is not enough to achieve the high levels of model validation you want. You need to use something more robust.

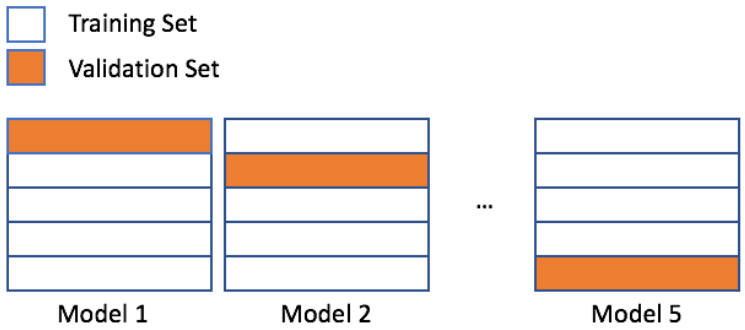
**Potential problems**

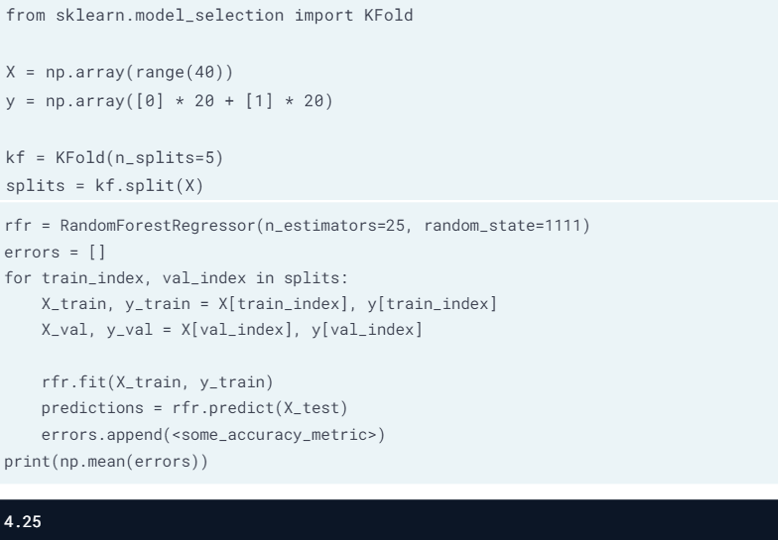
Which of the following statements (more than 1 answer) **are** **TRUE** regarding potential problems with holdout samples:

* **A**: Using different data splitting methods may lead to varying data in the final holdout samples.
* **B**: If you have limited data, your holdout accuracy may be misleading.
* **C**: There are no problems. Creating a single train and test sample is the only way to validate models.
* **D**: You shouldn’t use holdout samples with limited data because you are limiting the potential training data.

Answer: A & B. If our models are not generalizing well or if we have limited data, we should be careful using a single training/validation split.

**Cross-validation**





**scikit-learn’s KFold()**

You just finished running a colleagues code that creates a random forest model and calculates an out-of-sample accuracy. You noticed that your colleague’s code did not have a random state, and the errors you found were completely different than the errors your colleague reported.

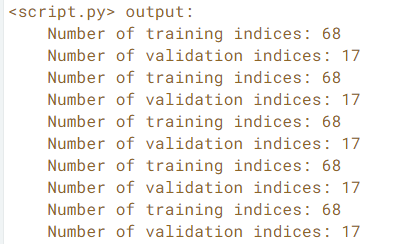
To get a better estimate for how accurate this random forest model will be on new data, you have decided to generate some indices to use for KFold cross-validation.

from sklearn.model\_selection import KFold

# Use KFold  
kf = KFold(n\_splits=5, shuffle=True, random\_state=1111)

# Create splits  
splits = kf.split(X)

# Print the number of indices  
for train\_index, val\_index in splits:  
 print("Number of training indices: %s" % len(train\_index))  
 print("Number of validation indices: %s" % len(val\_index))



This dataset has 85 rows. You have created five splits — each containing 68 training and 17 validation indices. You can use these indices to complete 5-fold cross-validation.

**Using KFold indices**

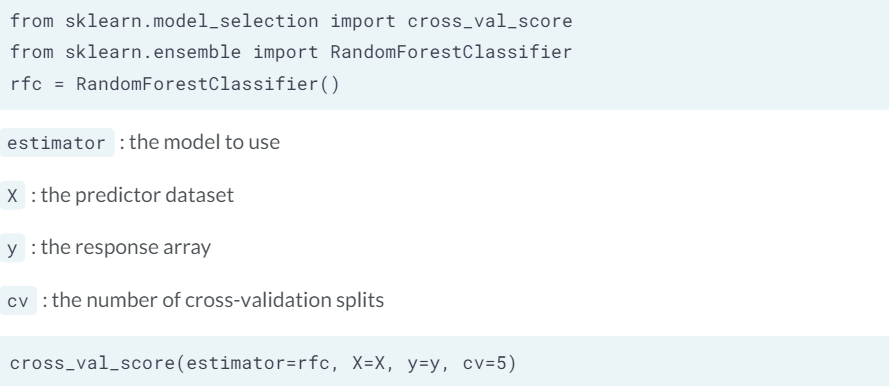
You have already created splits, which contains indices for the candy-data dataset to complete 5-fold cross-validation. To get a better estimate for how well a colleague's random forest model will perform on a new data, you want to run this model on the five different training and validation indices you just created.

In this exercise, you will use these indices to check the accuracy of this model using the five different splits. A for loop has been provided to assist with this process.

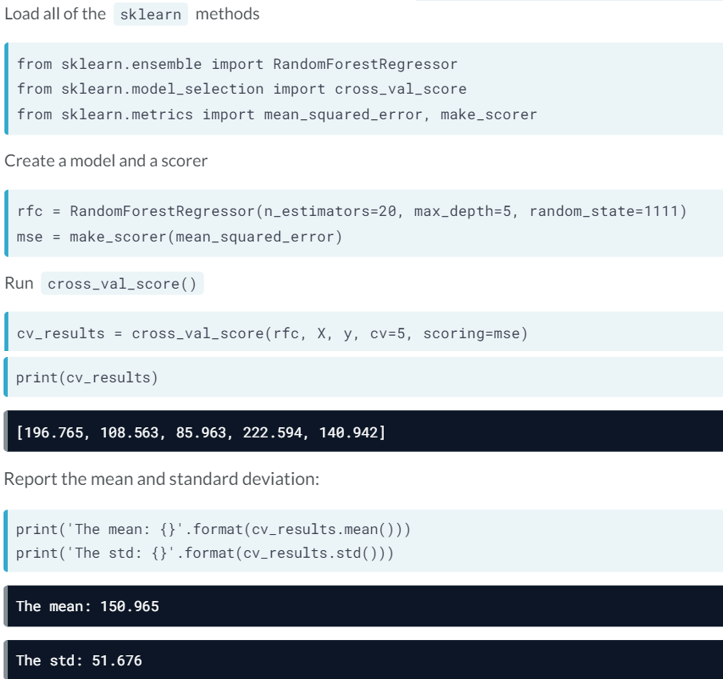
from sklearn.ensemble import RandomForestRegressor  
from sklearn.metrics import mean\_squared\_errorrfc = RandomForestRegressor(n\_estimators=25, random\_state=1111)# Access the training and validation indices of splits  
for train\_index, val\_index in splits:  
 # Setup the training and validation data  
 X\_train, y\_train = X[train\_index], y[train\_index]  
 X\_val, y\_val = X[val\_index], y[val\_index]  
 # Fit the random forest model  
 rfc.fit(X\_train, y\_train)  
 # Make predictions, and print the accuracy  
 predictions = rfc.predict(X\_val)  
 print("Split accuracy: " + str(mean\_squared\_error(y\_val, predictions)))

KFold() is a great method for accessing individual indices when completing cross-validation. One drawback is needing a for loop to work through the indices though. In the next lesson, you will look at an automated method for cross-validation using sklearn.

**sklearn’s cross\_val\_score()**



default scoring function is the mean overall accuracy or R-squared value



**scikit-learn’s methods**

You have decided to build a regression model to predict the number of new employees your company will successfully hire next month. You open up a new Python script to get started, but you quickly realize that sklearn has *a lot* of different modules. Let's make sure you understand the names of the modules, the methods, and which module contains which method.

Follow the instructions below to load in all of the necessary methods for completing cross-validation using sklearn. You will use modules:

* metrics
* model\_selection
* ensemble

# Instruction 1: Load the cross-validation method  
from sklearn.model\_selection import cross\_val\_score

# Instruction 2: Load the random forest regression model  
from sklearn.ensemble import RandomForestRegressor

# Instruction 3: Load the mean squared error method  
# Instruction 4: Load the function for creating a scorer  
from sklearn.metrics import mean\_squared\_error, make\_scorer

**Implement cross\_val\_score()**

Your company has created several new candies to sell, but they are not sure if they should release all five of them. To predict the popularity of these new candies, you have been asked to build a regression model using the candy dataset. Remember that the response value is a head-to-head win-percentage against other candies.

Before you begin trying different regression models, you have decided to run cross-validation on a simple random forest model to get a baseline error to compare with any future results.

rfc = RandomForestRegressor(n\_estimators=25, random\_state=1111)  
mse = make\_scorer(mean\_squared\_error)

# Set up cross\_val\_score  
cv = cross\_val\_score(estimator=rfc,  
 X=X\_train,  
 y=y\_train,  
 cv=10,  
 scoring=mse)

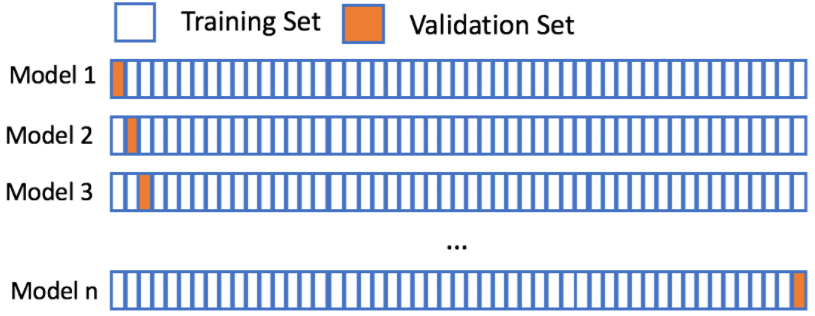
# Print the mean error  
print(cv.mean())

Image for post

Image for post

You now have a baseline score to build on. If you decide to build additional models or try new techniques, you should try to get an error lower than 155.56. Lower errors indicate that your predictions on popularity are improving.

**Leave-one-out-cross-validation (LOOCV)**



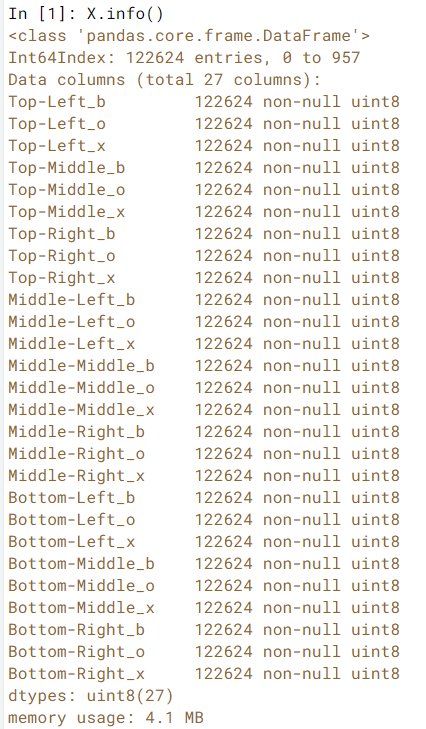
n = number of observation

When to use LOOCV?  
· the amount of training data is limited  
· low number of parameters  
· provides the best error estimate possible for a single new point  
· not constrained by computational resources (expensive)



**When to use LOOCV**

The X data has been loaded.



Which of the following are reasons you might **NOT** run LOOCV on the provided X dataset?

* **A**: The X dataset has 122,624 data points, which might be computationally expensive and slow.
* **B**: You cannot run LOOCV on classification problems.
* **C**: You want to test different values for 15 different parameters

Answer: A & C. This many observations will definitely slow things down and could be computationally expensive. If you don’t have time to wait while your computer runs through 1,000 models, you might want to use 5 or 10-fold cross-validation.

**Leave-one-out-cross-validation**

Let’s assume your favorite candy is not in the candy dataset, and that you are interested in the popularity of this candy. Using 5-fold cross-validation will train on only 80% of the data at a time. The candy dataset *only* has 85 rows though, and leaving out 20% of the data could hinder our model. However, using leave-one-out-cross-validation allows us to make the most out of our limited dataset and will give you the best estimate for your favorite candy’s popularity!

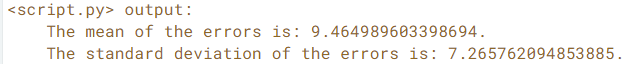
In this exercise, you will use cross\_val\_score() to perform LOOCV.

from sklearn.metrics import mean\_absolute\_error, make\_scorer

# Create scorer  
mae\_scorer = make\_scorer(mean\_absolute\_error)rfr = RandomForestRegressor(n\_estimators=15, random\_state=1111)

# Implement LOOCV  
scores = cross\_val\_score(rfr, X=X, y=y, cv=85, scoring=mae\_scorer)

# Print the mean and standard deviation  
print("The mean of the errors is: %s." % np.mean(scores))  
print("The standard deviation of the errors is: %s." % np.std(scores))



**Chapter 4. Selecting the best model with Hyperparameter tuning**

The first three chapters focused on model validation techniques. In chapter 4 we apply these techniques, specifically cross-validation, while learning about hyperparameter tuning. After all, model validation makes tuning possible and helps us select the overall best model.

**Introduction to hyperparameter tuning**

Model **parameters**are:  
· learned or estimated from the data  
· the result of fitting a model  
· used when making future predictions  
· not manually set  
· example: coefficient and intercept in a simple linear model

Model **hyperparameters**are:  
· manually set before the training occurs  
· specify how the training is supposed to happen  
· example: n\_estimators, max\_depth, max\_features, min\_samples\_split

Hyperparameter tuning is to find the most optimal set of hyperparameters in a model that would achieve the best model performance:  
Step 1 — select hyperparameters and possible range of values  
Step 2 — specify a single performance metric eg. accuracy  
Step 3 —run a single model with different value sets, record scores  
Step 4 — find the highest scoring set of hyperparameters

**Creating Hyperparameters**

For a school assignment, your professor has asked your class to create a random forest model to predict the average test score for the final exam.

After developing an initial random forest model, you are unsatisfied with the overall accuracy. You realize that there are too many hyperparameters to choose from, and each one has *a lot* of possible values. You have decided to make a list of possible ranges for the hyperparameters you might use in your next model.

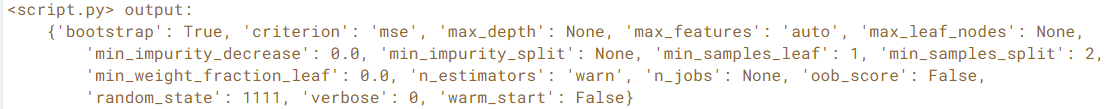
Your professor has provided de-identified data for the last ten quizzes to act as the training data. There are 30 students in your class.

# Review the parameters of rfr  
print(rfr.get\_params())

# Maximum Depth  
max\_depth = [4, 8, 12]

# Minimum samples for a split  
min\_samples\_split = [2, 5, 10]

# Max features   
max\_features = [4, 6, 8, 10]



Hyperparameter tuning requires selecting parameters to tune, as well the possible values these parameters can be set to.

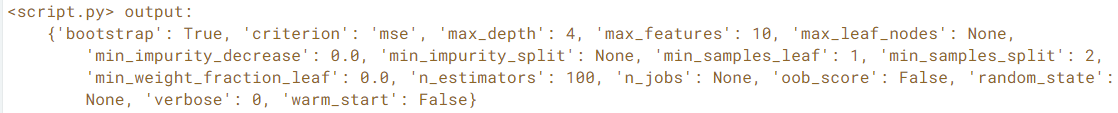
**Running a model using ranges**

You have just finished creating a list of hyperparameters and ranges to use when tuning a predictive model for an assignment. You have used max\_depth, min\_samples\_split, and max\_features as your range variable names.

from sklearn.ensemble import RandomForestRegressor

# Fill in rfr using your variables  
rfr = RandomForestRegressor(  
 n\_estimators=100,  
 max\_depth=random.choice(max\_depth),  
 min\_samples\_split=random.choice(min\_samples\_split),  
 max\_features=random.choice(max\_features))

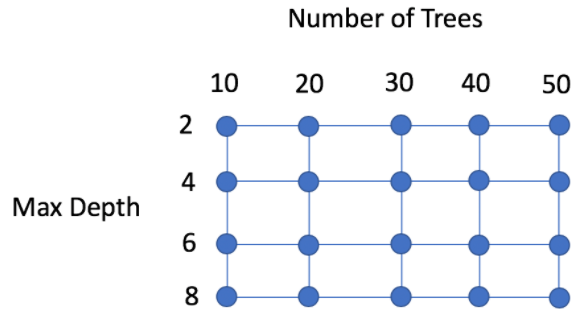
# Print out the parameters  
print(rfr.get\_params())

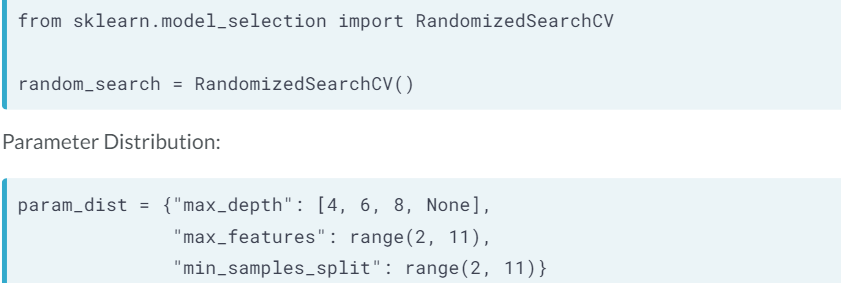


Notice that min\_samples\_split was randomly set to 2. Since you specified a random state, min\_samples\_split will always be set to 2 if you only run this model one time.

**RandomizedSearchCV**

Grid searching for hyperparameters, to test every possible combination, but computationally expensive.





Using a grid search will take 4x9x9=324 total model runs; Using a **random search**, we can get similar results only using 30 or 40 runs.



**Preparing for RandomizedSearch**

Last semester your professor challenged your class to build a predictive model to predict final exam test scores. You tried running a few different models by randomly selecting hyperparameters. However, running each model required you to code it individually.

After learning about RandomizedSearchCV(), you're revisiting your professors challenge to build the best model. In this exercise, you will prepare the three necessary inputs for completing a random search.

from sklearn.ensemble import RandomForestRegressor  
from sklearn.metrics import make\_scorer, mean\_squared\_error

# Finish the dictionary by adding the max\_depth parameter  
param\_dist = {"max\_depth": [2, 4, 6, 8],  
 "max\_features": [2, 4, 6, 8, 10],  
 "min\_samples\_split": [2, 4, 8, 16]}

# Create a random forest regression model  
rfr = RandomForestRegressor(n\_estimators=10, random\_state=1111)

# Create a scorer to use (use the mean squared error)  
scorer = make\_scorer(mean\_squared\_error)

To use RandomizedSearchCV(), you need a distribution dictionary, an estimator, and a scorer—once you've got these, you can run a random search to find the best parameters for your model.

**Implementing RandomizedSearchCV**

You are hoping that using a random search algorithm will help you improve predictions for a class assignment. You professor has challenged your class to predict the overall final exam average score.

In preparation for completing a random search, you have created:

* param\_dist: the hyperparameter distributions
* rfr: a random forest regression model
* scorer: a scoring method to use

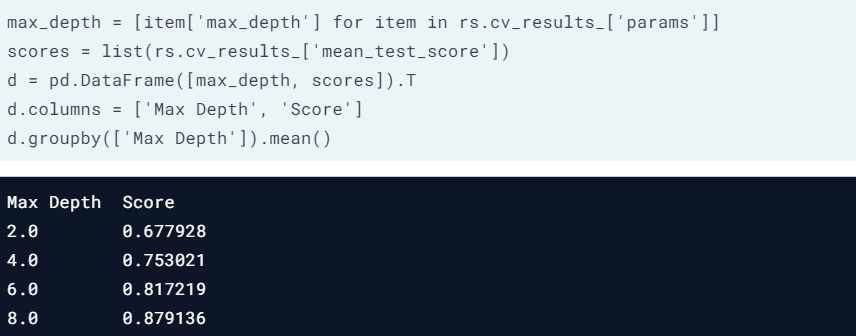
# Import the method for random search  
from sklearn.model\_selection import RandomizedSearchCV

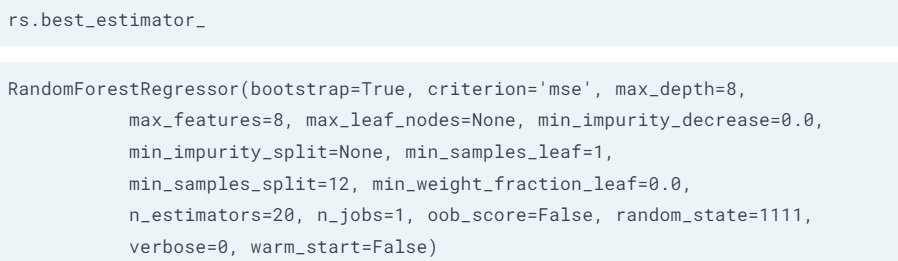
# Build a random search using param\_dist, rfr, and scorer  
random\_search =\  
 RandomizedSearchCV(  
 estimator=rfr,  
 param\_distributions=param\_dist,  
 n\_iter=10,  
 cv=5,  
 scoring=scorer)

Although it takes a lot of steps, hyperparameter tuning with random search is well worth it and can improve the accuracy of your models. Plus, you are already using cross-validation to validate your best model.

**Selecting your final model**

random\_search.best\_score\_  
random\_search.best\_params\_  
random\_search.best\_estimator\_  
random\_search.cv\_results\_  
random\_search.cv\_results\_['mean\_test\_score']  
random\_search.cv\_results\_['params']  
random\_search.best\_estimator\_





This shows the model that performed the best during cross validation

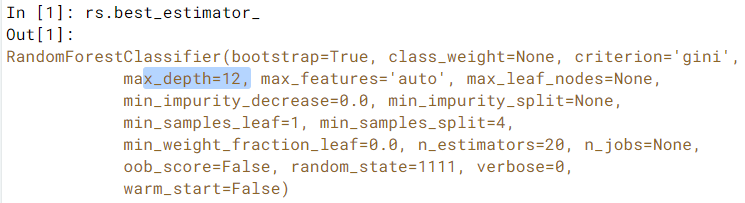
**Best classification accuracy**

You are in a competition at work to build the best model for predicting the winner of a Tic-Tac-Toe game. You already ran a random search and saved the results of the most accurate model to rs.

Which parameter set produces the best classification accuracy?



Answer: {‘max\_depth’: 12, ‘min\_samples\_split’: 4, ‘n\_estimators’: 20}



**Selecting the best precision model**

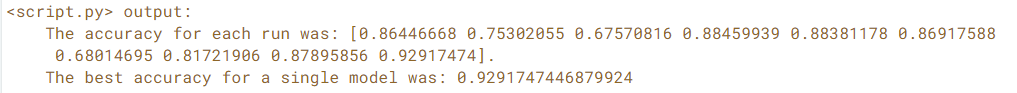
Your boss has offered to pay for you to see three sports games this year. Of the 41 home games your favorite team plays, you want to ensure you go to three home games that they will *definitely* win. You build a model to decide which games your team will win.

To do this, you will build a random search algorithm and focus on model precision (to ensure your team wins). You also want to keep track of your best model and best parameters, so that you can use them again next year (if the model does well, of course). You have already decided on using the random forest classification model rfc and generated a parameter distribution param\_dist.

from sklearn.metrics import precision\_score, make\_scorer

# Create a precision scorer  
precision = make\_scorer(precision\_score)  
# Finalize the random search  
rs = RandomizedSearchCV(  
 estimator=rfc, param\_distributions=param\_dist,  
 scoring = precision,  
 cv=5, n\_iter=10, random\_state=1111)  
rs.fit(X, y)

# print the mean test scores:  
print('The accuracy for each run was: {}.'.format(rs.cv\_results\_['mean\_test\_score']))  
# print the best model score:  
print('The best accuracy for a single model was: {}'.format(rs.best\_score\_))



Your model’s precision was 93%! The best model accurately predicts a winning game 93% of the time. If you look at the mean test scores, you can tell some of the other parameter sets did really poorly. Also, since you used cross-validation, you can be confident in your predictions.

**Course completed!**

Recap topics covered:

* Accuracy and evaluation metrics
* Splitting data into train, validation, and test sets
* Cross-validation and LOOCV
* Hyperparameter tuning

Happy learning!