Model Validation in Python

How good is your machine learning model?

Black Raven (James Ng)
14 Aug 2020 · 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

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.

Photo by Craige McGonigle on Unsplash

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?

- ○ `.predict()`
- ○ `.fit()`
- ○ `.validate()`

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.

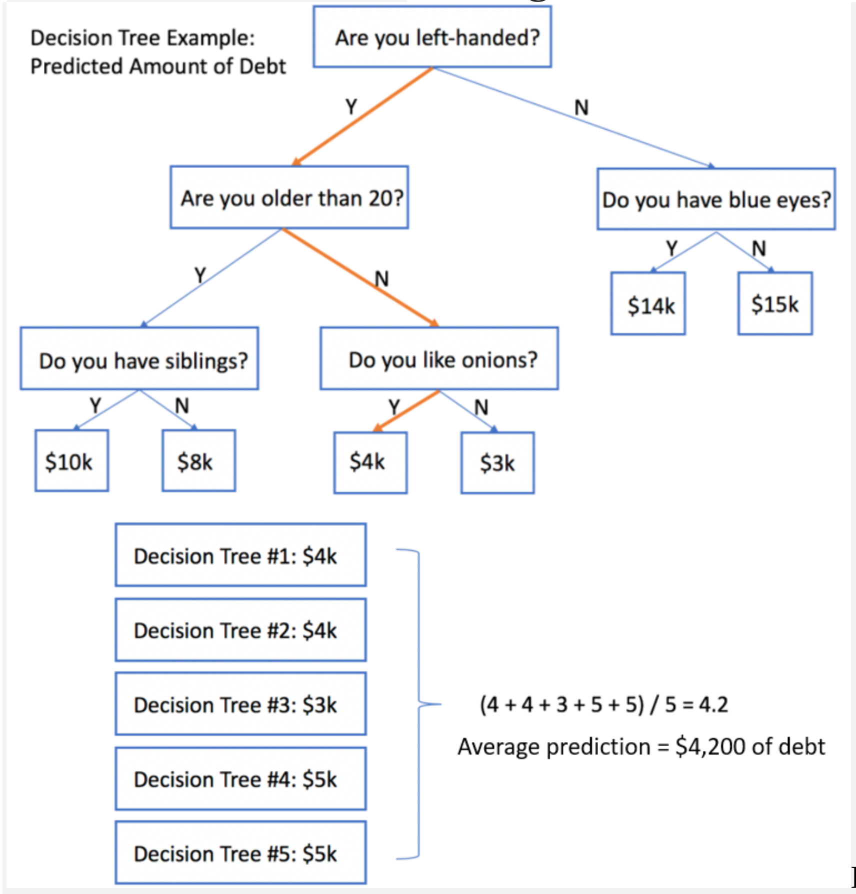
```
# The model is fit using X_train and y_train
model.fit(X_train, y_train)
# Create vectors of predictions
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)
# Train/Test Errors
train_error = mae(y_true=y_train, y_pred=train_predictions)
test_error = mae(y_true=y_test, 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))
```

```
<script.py> output:
    Model error on seen data: 3.28.
    Model error on unseen data: 11.07.
```

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

```
for i, item in enumerate(rfr.feature_importances_):
    print("{0:s}: {1:.2f}".format(X.columns[i], item))
```

```
weight: 0.50
height: 0.39
left_handed: 0.72
union_preference: 0.05
eye_color: 0.03
```

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)
```

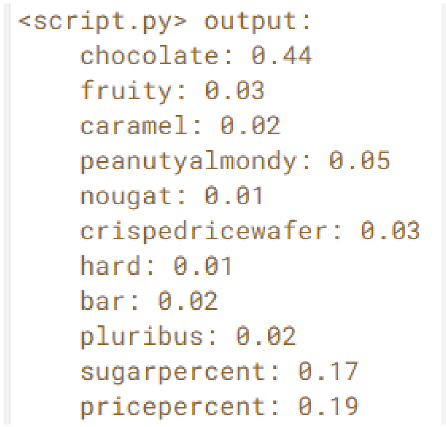
This approach is helpful when you need to update parameters after the model was initialised.

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))
```

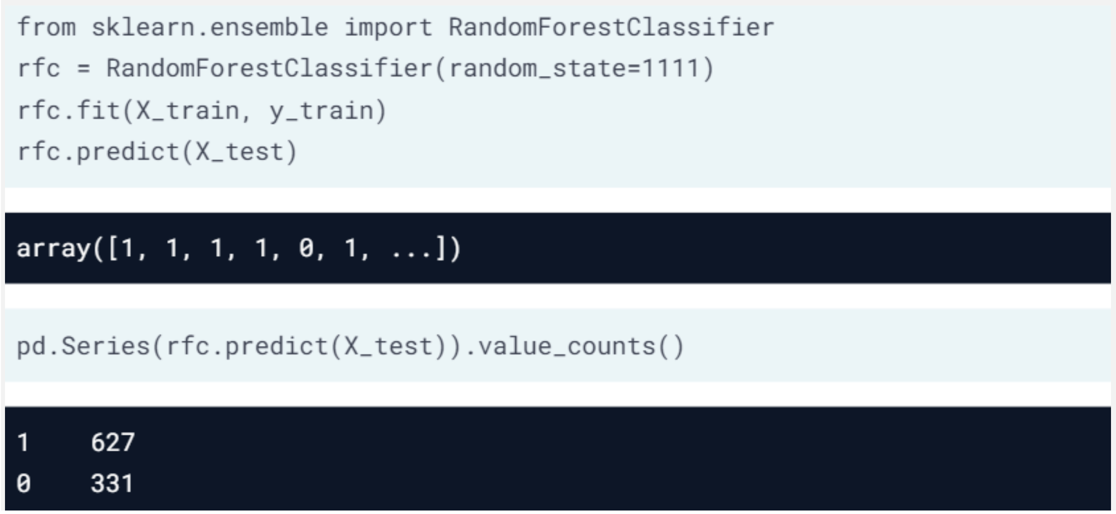


```
<script.py> output:
    chocolate: 0.44
    fruity: 0.03
    caramel: 0.02
    peanutyalmondy: 0.05
    nougat: 0.01
    crispedricewafer: 0.03
    hard: 0.01
    bar: 0.02
    pluribus: 0.02
    sugarpercent: 0.17
    pricepercent: 0.19
```

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.



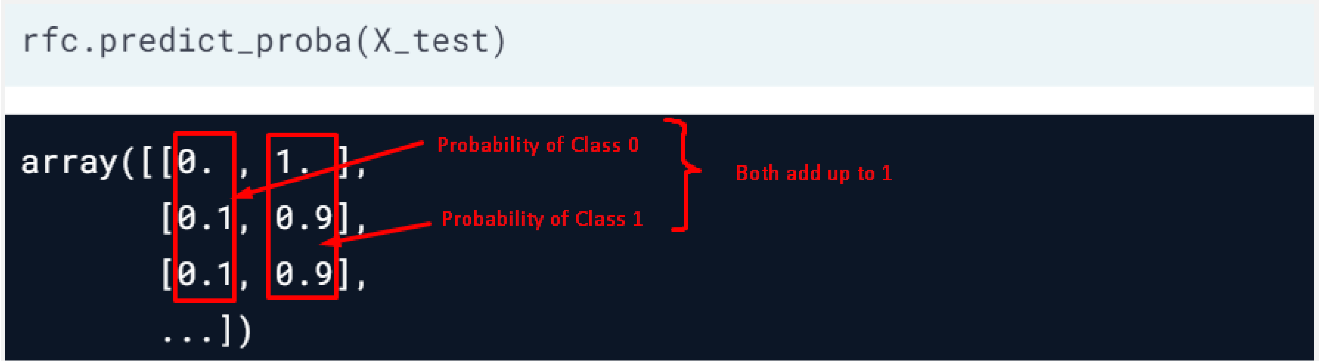
```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(random_state=1111)
rfc.fit(X_train, y_train)
rfc.predict(X_test)
```

```
array([1, 1, 1, 1, 0, 1, ...])
```

```
pd.Series(rfc.predict(X_test)).value_counts()
```

```
1    627
0    331
```

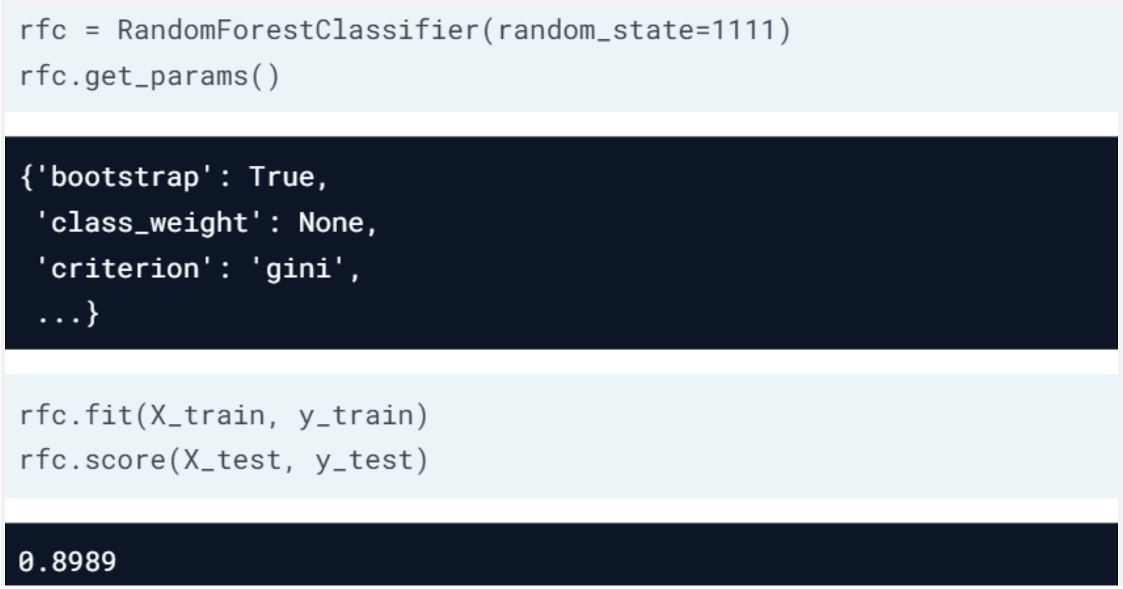
Using .predict() for classification



```
rfc.predict_proba(X_test)
```

```
array([[0. , 1. ],
       [0.1, 0.9],
       [0.1, 0.9],
       ...])
```

Using .predict_proba() to get probabilities



```
rfc = RandomForestClassifier(random_state=1111)
rfc.get_params()
```

```
{'bootstrap': True,
 'class_weight': None,
 'criterion': 'gini',
 ...}
```

```
rfc.fit(X_train, y_train)
rfc.score(X_test, y_test)
```

```
0.8989
```

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.

Probability	Prediction	Meaning
0 < .50	0	Team Loses
.50 +	1	Team Wins

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]))
```

```
<script.py> output:
    1    563
    0    204
    dtype: int64
    The first predicted probabilities are: [0.26524423 0.73475577]
```

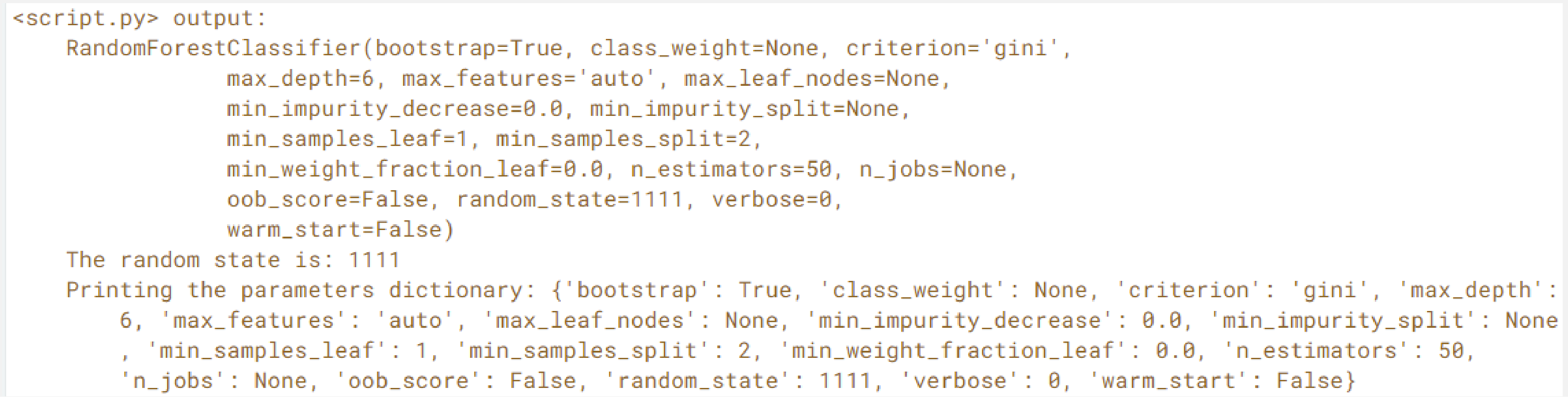
You can see there were 563 observations where Player One was predicted to win the Tic-Tac-Toe game. Also, note that the `predicted_probabilities` array contains lists with only two values because you only have two possible responses (win or lose).

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**. 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()))
```



```
<script.py> output:
    RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                max_depth=6, max_features='auto', max_leaf_nodes=None,
                min_impurity_decrease=0.0, min_impurity_split=None,
                min_samples_leaf=1, min_samples_split=2,
                min_weight_fraction_leaf=0.0, n_estimators=50, n_jobs=None,
                oob_score=False, random_state=1111, verbose=0,
                warm_start=False)
    The random state is: 1111
    Printing the parameters dictionary: {'bootstrap': True, 'class_weight': None, 'criterion': 'gini', 'max_depth':
        6, 'max_features': 'auto', 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None
        , 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 50,
        'n_jobs': None, 'oob_score': False, 'random_state': 1111, 'verbose': 0, 'warm_start': False}
```

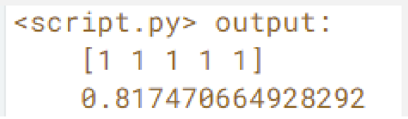
Recalling which parameters were used will be helpful going forward. Model validation and performance rely heavily on which parameters were used, and there is no way to replicate a model without keeping track of the parameters used!

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))
```



```
<script.py> output:
    [1 1 1 1 1]
    0.817470664928292
```

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

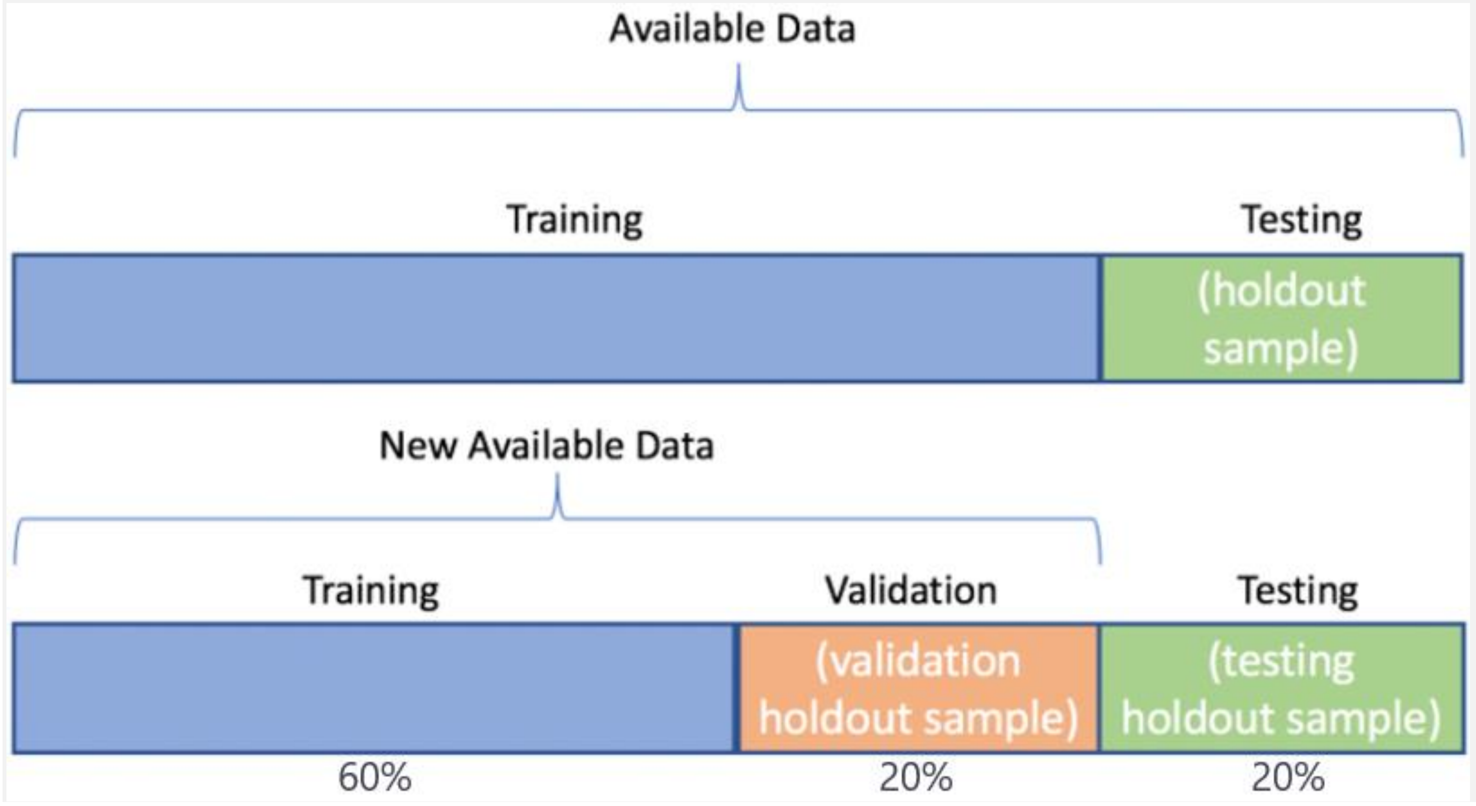
```
import pandas as pd

tic_tac_toe = pd.read_csv("tic-tac-toe.csv")
X = pd.get_dummies(tic_tac_toe.iloc[:,0:9])
y = tic_tac_toe.iloc[:, 9]
X_train, X_test, y_train, y_test  =\
    train_test_split(X, y, test_size=0.2, random_state=1111)
```

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

```
X_temp, X_test, y_temp, y_test  =\
    train_test_split(X, y, test_size=0.2, random_state=1111)
```

```
X_train, X_val, y_train, y_val =\
    train_test_split(X_temp, y_temp, test_size=0.25, random_state=11111)
```

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.

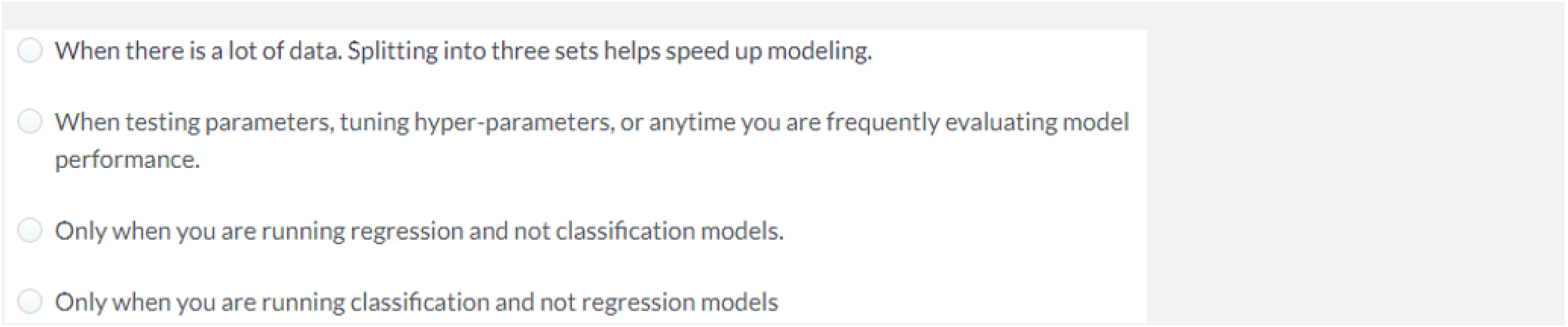
```
# Create temporary training and final testing datasets
X_temp, X_test, y_temp, y_test  =\
    train_test_split(X, y, test_size=0.2, random_state=1111)
# Create the final training and validation datasets
X_train, X_val, y_train, y_val =\
    train_test_split(X_temp, y_temp,
    test_size=0.25, random_state=1111)
```

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?



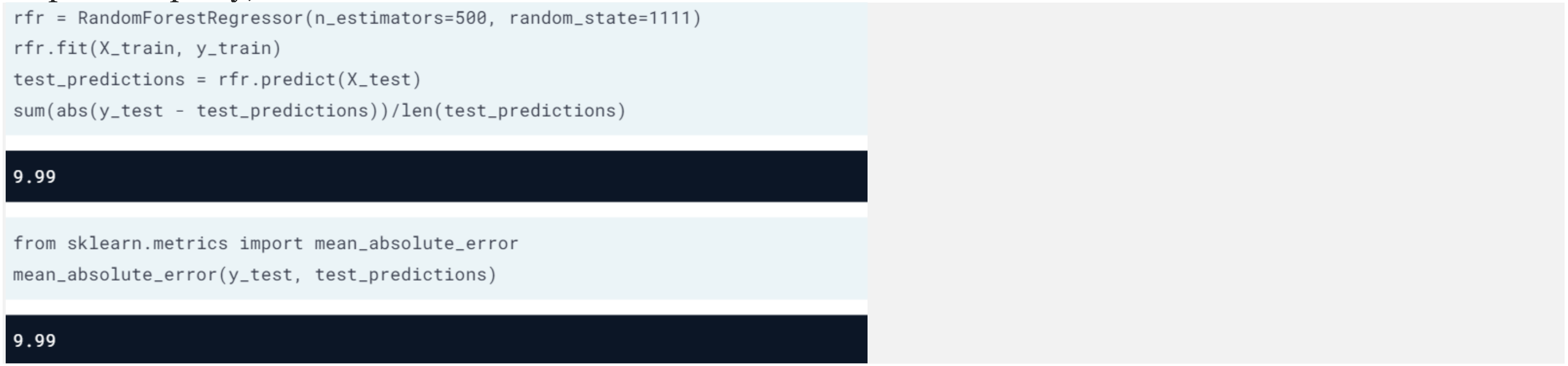
- When there is a lot of data. Splitting into three sets helps speed up modeling.
- When testing parameters, tuning hyper-parameters, or anytime you are frequently evaluating model performance.
- Only when you are running regression and not classification models.
- Only when you are running classification and not regression models

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.



```
rfr = RandomForestRegressor(n_estimators=500, random_state=1111)
rfr.fit(X_train, y_train)
test_predictions = rfr.predict(X_test)
sum(abs(y_test - test_predictions))/len(test_predictions)
```

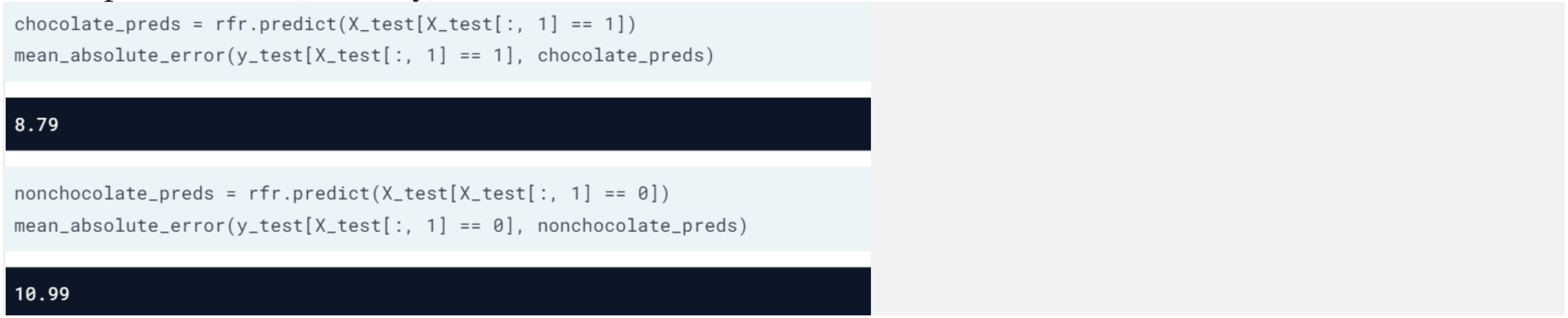
```
9.99
```

```
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, test_predictions)
```

```
9.99
```

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

Model performance (accuracy) for a subset of data



```
chocolate_preds = rfr.predict(X_test[X_test[:, 1] == 1])
mean_absolute_error(y_test[X_test[:, 1] == 1], chocolate_preds)
```

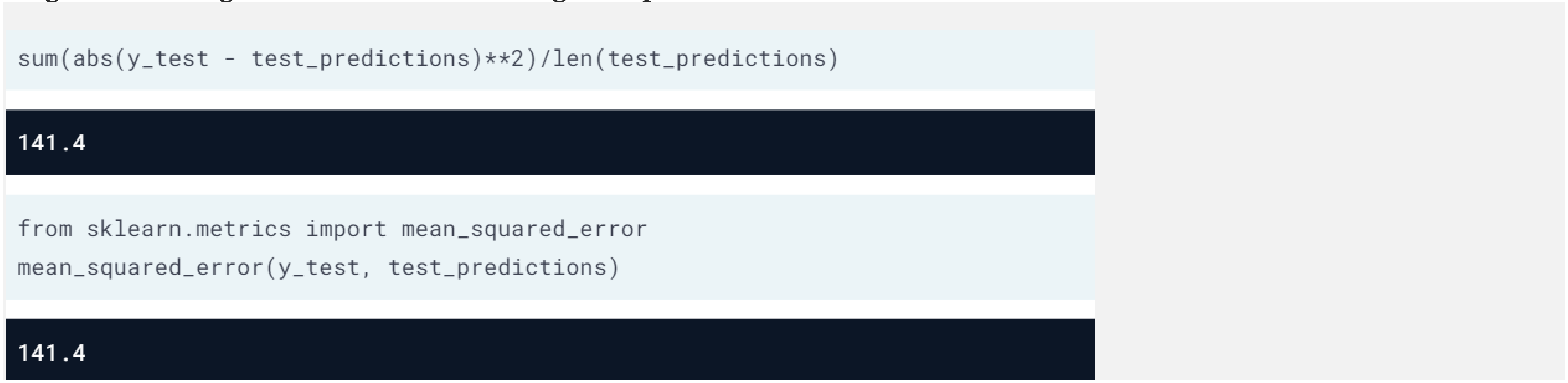
```
8.79
```

```
nonchocolate_preds = rfr.predict(X_test[X_test[:, 1] == 0])
mean_absolute_error(y_test[X_test[:, 1] == 0], nonchocolate_preds)
```

```
10.99
```

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



```
sum(abs(y_test - test_predictions)**2)/len(test_predictions)
```

```
141.4
```

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, test_predictions)
```

```
141.4
```

Outliers have more impact on the model’s performance.

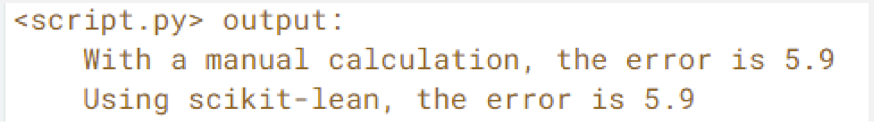
Mean absolute error

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))
```



```
<script.py> output:
    With a manual calculation, the error is 5.9
    Using scikit-lean, the error is 5.9
```

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))
```

```
<script.py> output:
    With a manual calculation, the error is 49.1
    Using scikit-lean, the error is 49.1
```

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))
```

```
<script.py> output:
    The MAE for East teams is 6.733333333333333
    The MAE for West conference is 5.01
```

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.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, test_predictions)
print(cm)
```

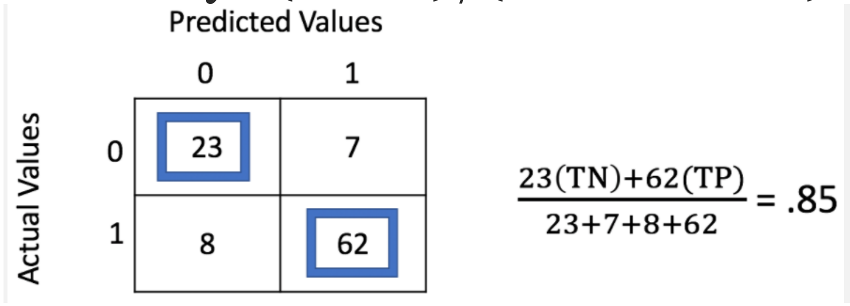
```
array([[ 23,  7],
       [  8, 62]])
```

```
cm[<true_category_index>, <predicted_category_index>]
cm[1, 0]
```

```
8
```

Classification metrics: Accuracy, Precision, Recall

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

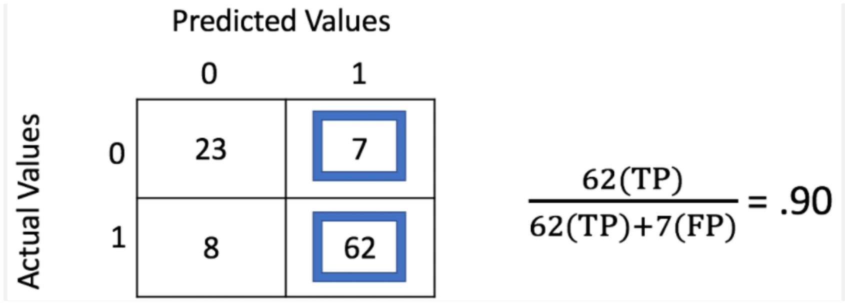


$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & \boxed{23} & 7 \\ 1 & 8 & \boxed{62} \end{array}$$

$$\frac{23(TN)+62(TP)}{23+7+8+62} = .85$$

· **Precision** = TP / (TP+FP)

Precision is used when we do not want to overpredict positive values.

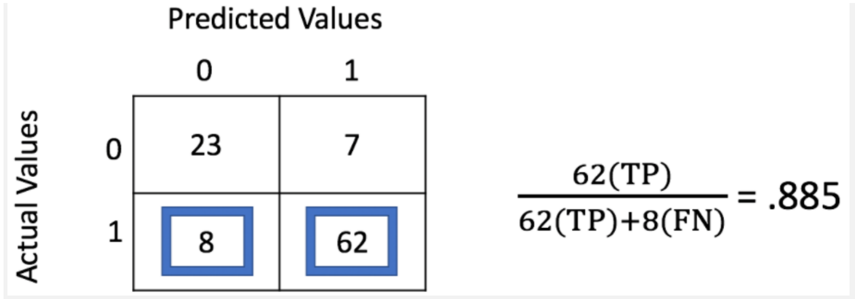


$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 23 & \boxed{7} \\ 1 & 8 & \boxed{62} \end{array}$$

$$\frac{62(TP)}{62(TP)+7(FP)} = .90$$

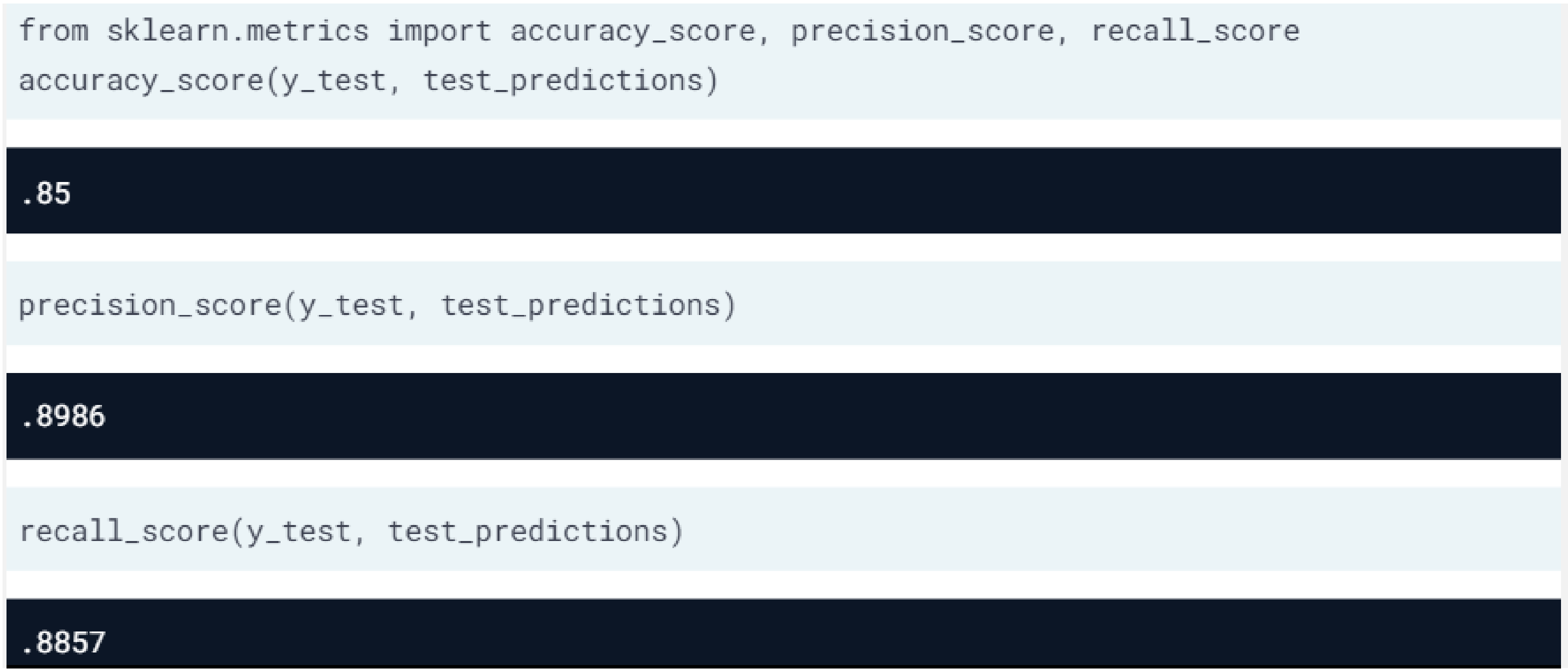
· **Recall** (also called sensitivity)

Recall is used when we cannot afford to miss out any positive values.



$$\begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 23 & 7 \\ 1 & \boxed{8} & \boxed{62} \end{array}$$

$$\frac{62(TP)}{62(TP)+8(FN)} = .885$$



```
from sklearn.metrics import accuracy_score, precision_score, recall_score
accuracy_score(y_test, test_predictions)
```

```
.85
```

```
precision_score(y_test, test_predictions)
```

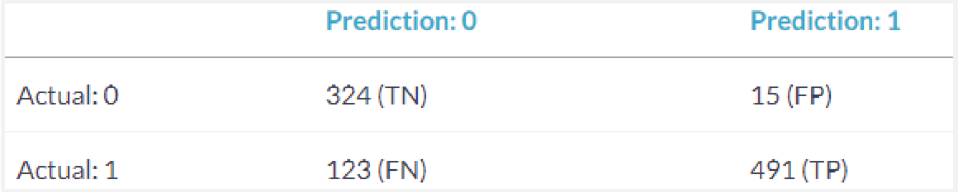
```
.8986
```

```
recall_score(y_test, test_predictions)
```

```
.8857
```

Confusion matrices

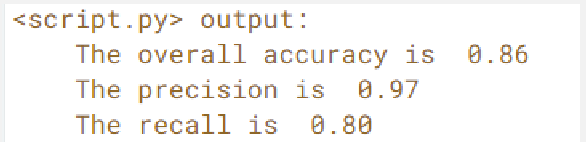
Confusion matrices are a great way to start exploring your model's accuracy. They provide the values needed to calculate a wide range of metrics, including sensitivity, specificity, and the F1-score.
You have built a classification model to predict if a person has a broken arm based on an X-ray image. On the testing set, you have the following confusion matrix:



	Prediction: 0	Prediction: 1
Actual: 0	324 (TN)	15 (FP)
Actual: 1	123 (FN)	491 (TP)

```
# 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))
```



```
<script.py> output:
    The overall accuracy is  0.86
    The precision is  0.97
    The recall is  0.80
```

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.

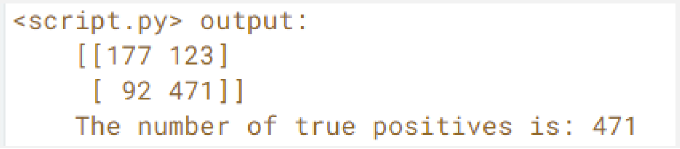
Confusion matrices, again

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]))
```



```
<script.py> output:
    [[177 123]
     [ 92 471]]
    The number of true positives is: 471
```

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!

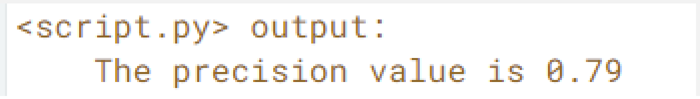
Precision vs. recall

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))
```

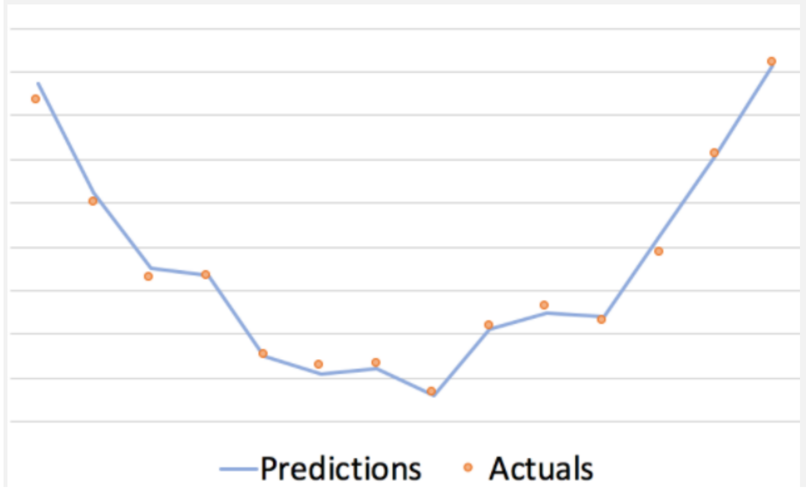


```
<script.py> output:
    The precision value is 0.79
```

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.

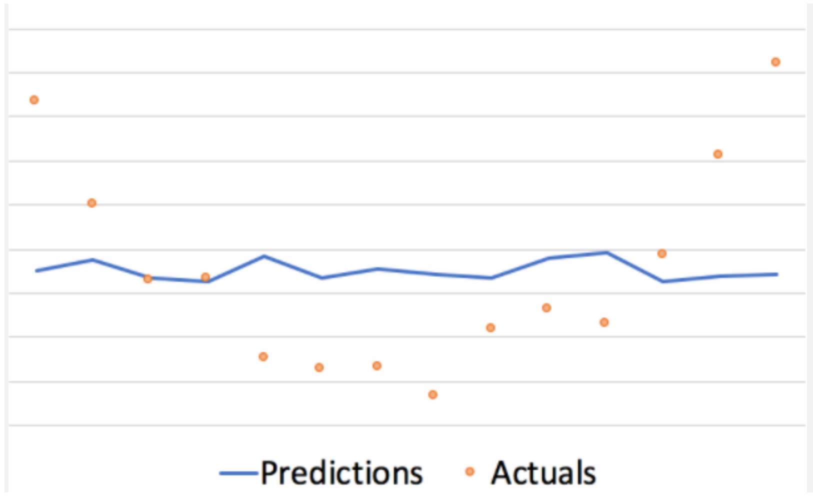
The bias–variance tradeoff

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.

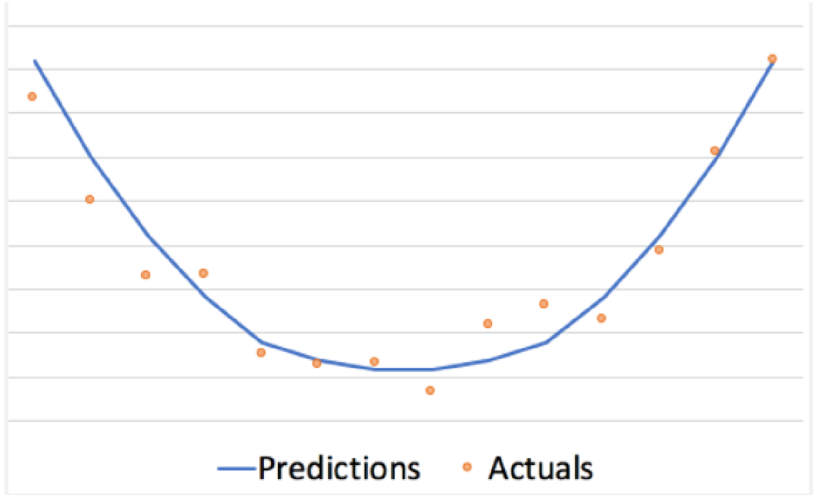


Overfitting: training error < testing error

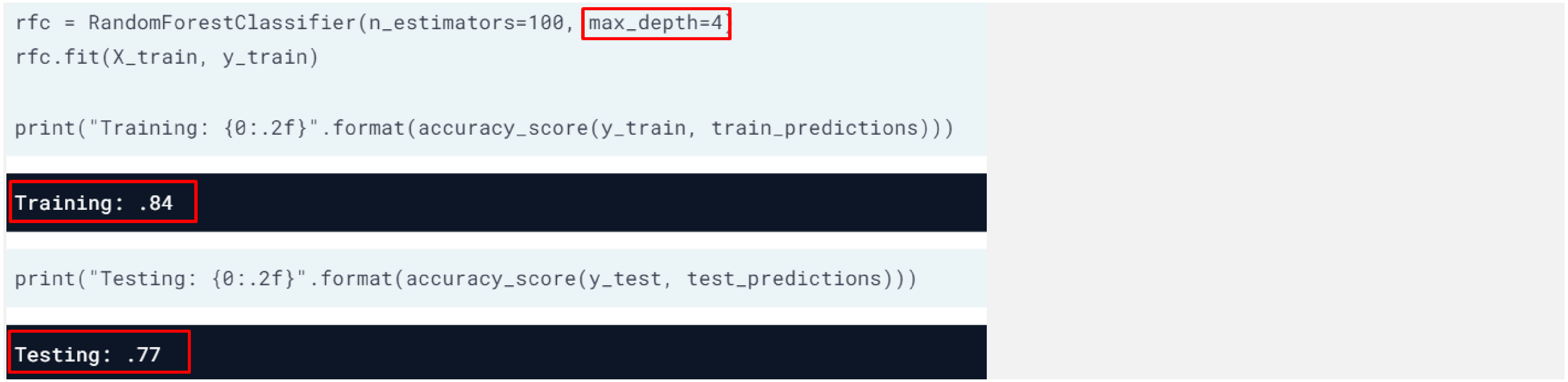
Bias occurs when a model fails to find the relationships between the data and the response value. Bias leads to high errors on both the training and testing datasets, ie, **underfitting** = model could not find the underlying patterns available in the data.



Underfitting: both training error and testing error are high



Optimal performance



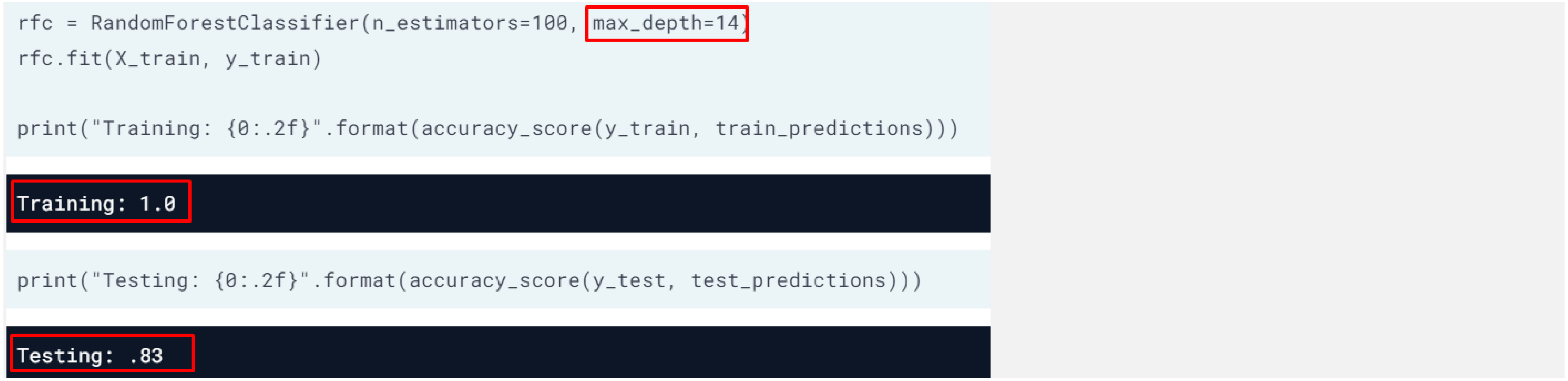
```
rfc = RandomForestClassifier(n_estimators=100, max_depth=4)
rfc.fit(X_train, y_train)

print("Training: {0:.2f}".format(accuracy_score(y_train, train_predictions)))
```

```
Training: .84
```

```
print("Testing: {0:.2f}".format(accuracy_score(y_test, test_predictions)))
```

```
Testing: .77
```

max_depth = 4 is probably not deep enough



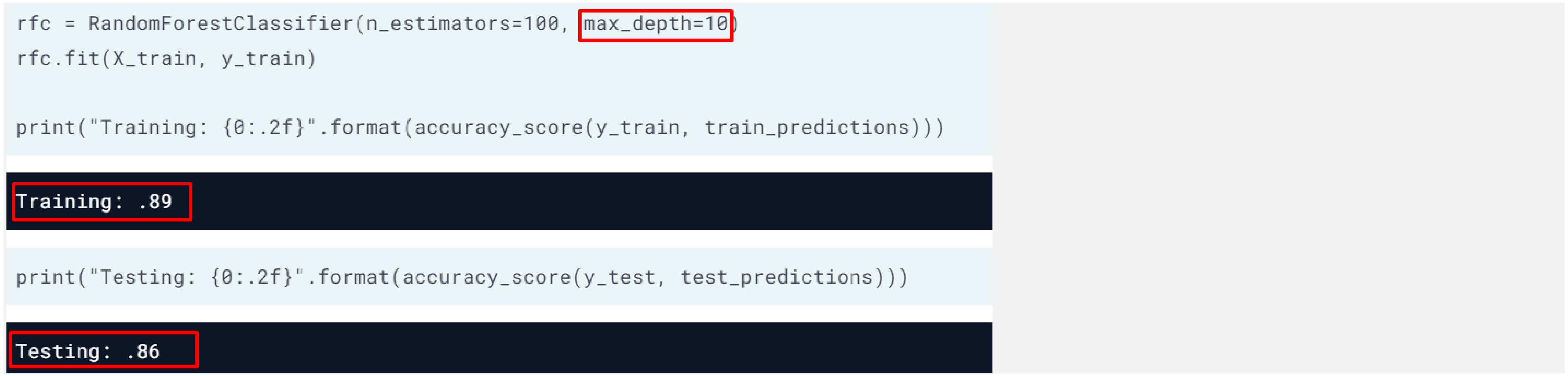
```
rfc = RandomForestClassifier(n_estimators=100, max_depth=14)
rfc.fit(X_train, y_train)

print("Training: {0:.2f}".format(accuracy_score(y_train, train_predictions)))
```

```
Training: 1.0
```

```
print("Testing: {0:.2f}".format(accuracy_score(y_test, test_predictions)))
```

```
Testing: .83
```

max_depth too large and overfitting



```
rfc = RandomForestClassifier(n_estimators=100, max_depth=10)
rfc.fit(X_train, y_train)

print("Training: {0:.2f}".format(accuracy_score(y_train, train_predictions)))
```

```
Training: .89
```

```
print("Testing: {0:.2f}".format(accuracy_score(y_test, test_predictions)))
```

```
Testing: .86
```

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))))
```

```
<script.py> output:
    The training error is 3.88
    The testing error is 9.15
```

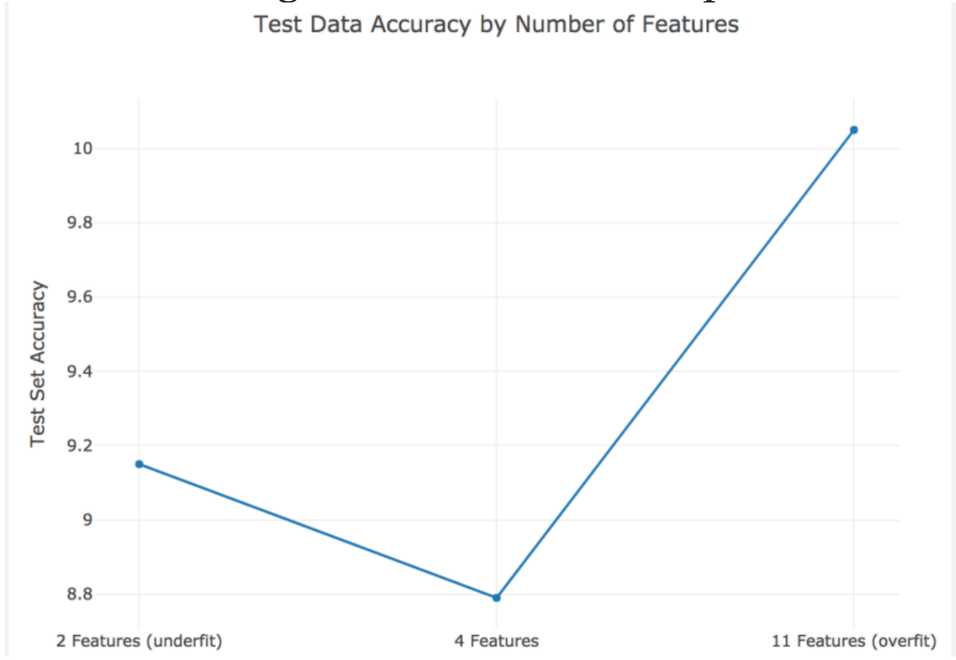
```
# Update the rfr model
rfr = RandomForestRegressor(n_estimators=25,
                            random_state=1111,
                            max_features=11)
rfr.fit(X_train, y_train)
# Print the training and testing accuracies
print('The training error is {0:.2f}'.format(
  mae(y_train, rfr.predict(X_train))))
print('The testing error is {0:.2f}'.format(
  mae(y_test, rfr.predict(X_test))))
```

```
<script.py> output:
    The training error is 3.57
    The testing error is 10.05
```

```
# 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))))
```

```
<script.py> output:
    The training error is 3.60
    The testing error is 8.79
```

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

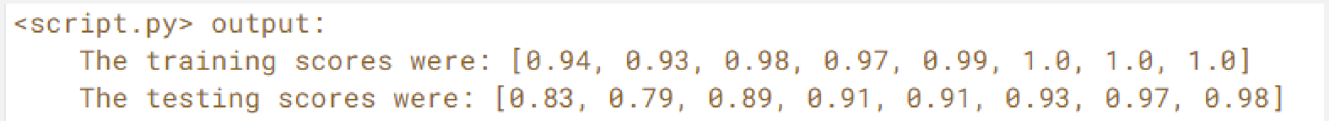


Am I underfitting?

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))
```



```
<script.py> output:
    The training scores were: [0.94, 0.93, 0.98, 0.97, 0.99, 1.0, 1.0, 1.0]
    The testing scores were: [0.83, 0.79, 0.89, 0.91, 0.91, 0.93, 0.97, 0.98]
```

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 the`tic_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())
```

```
<script.py> output:
    40
    positive    134
    negative     66
    Name: Class, dtype: int64
    positive    123
    negative     77
    Name: Class, dtype: int64
```

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.

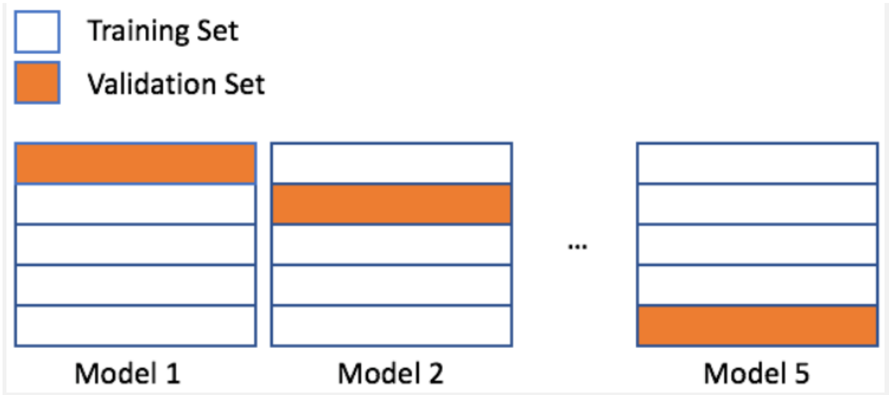
Potential problems

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



```
from sklearn.model_selection import KFold

X = np.array(range(40))
y = np.array([0] * 20 + [1] * 20)

kf = KFold(n_splits=5)
splits = kf.split(X)
```

```
rfr = RandomForestRegressor(n_estimators=25, random_state=1111)
errors = []
for train_index, val_index in splits:
    X_train, y_train = X[train_index], y[train_index]
    X_val, y_val = X[val_index], y[val_index]

    rfr.fit(X_train, y_train)
    predictions = rfr.predict(X_test)
    errors.append(<some_accuracy_metric>)
print(np.mean(errors))
```

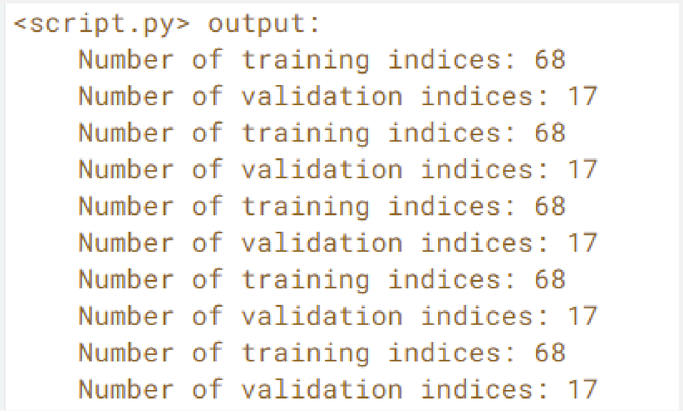
```
4.25
```

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))
```



```
<script.py> output:
    Number of training indices: 68
    Number of validation indices: 17
    Number of training indices: 68
    Number of validation indices: 17
    Number of training indices: 68
    Number of validation indices: 17
    Number of training indices: 68
    Number of validation indices: 17
    Number of training indices: 68
    Number of validation indices: 17
```

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

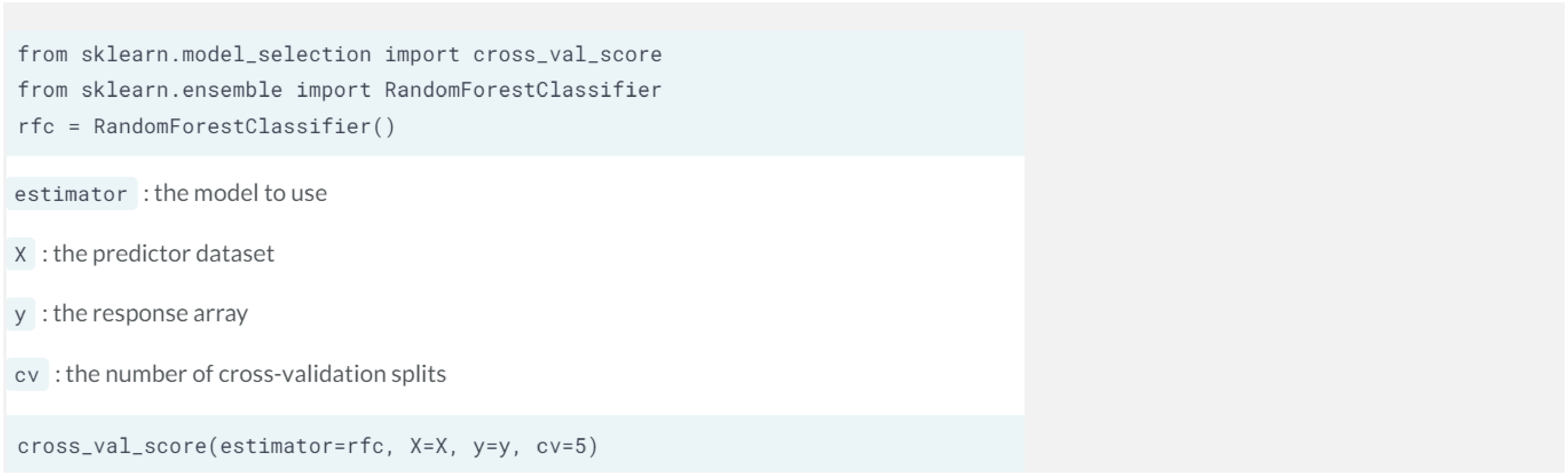
You have already created `splits`, which contains indices for the candy-data dataset to complete 5-fold cross-validation. To get a better estimate for how well a colleague's random forest model will perform on a new data, you want to run this model on the five different training and validation indices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



```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier()
```

`estimator` : the model to use

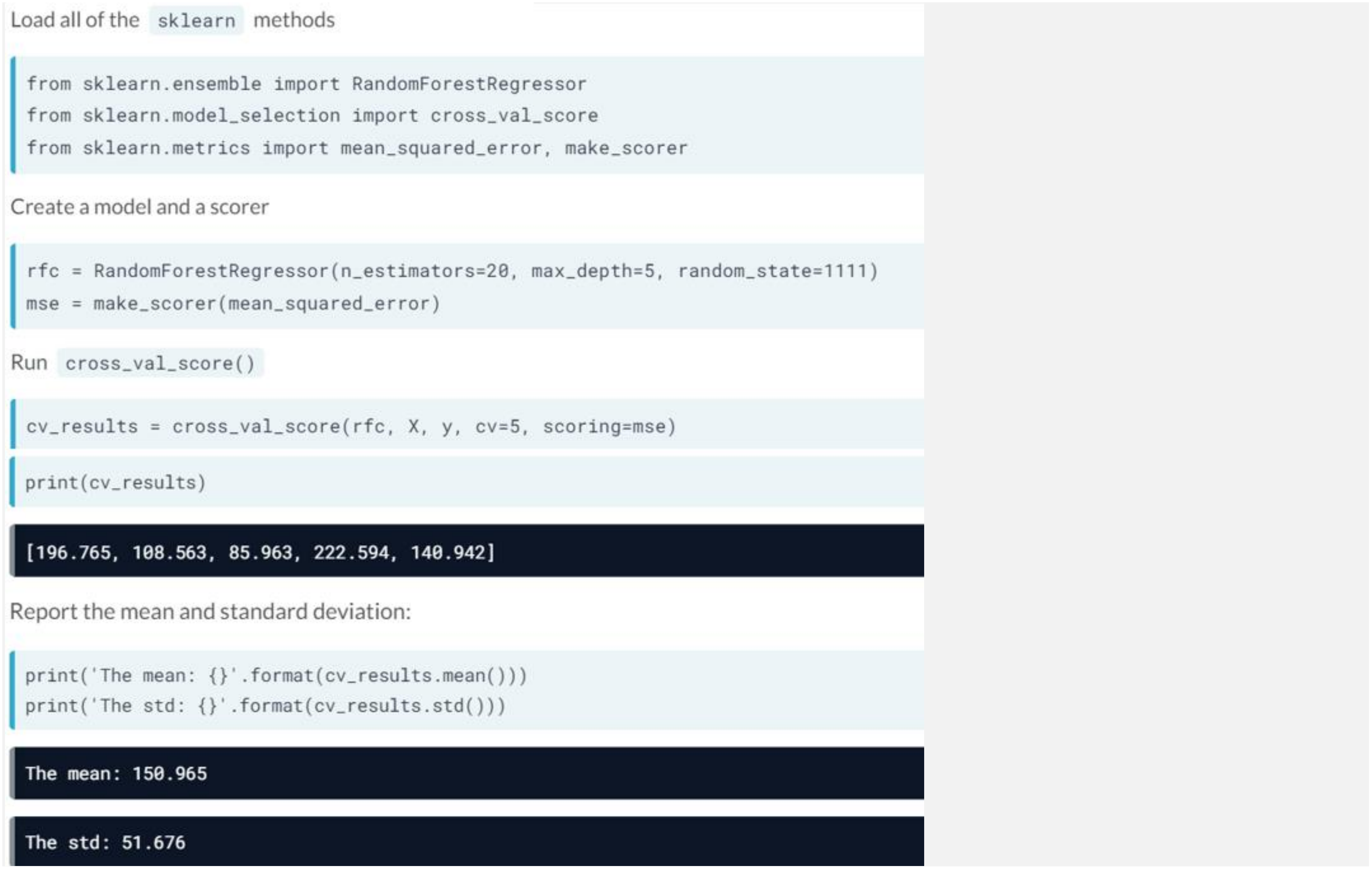
`X` : the predictor dataset

`y` : the response array

`cv` : the number of cross-validation splits

```
cross_val_score(estimator=rfc, X=X, y=y, cv=5)
```

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



Load all of the `sklearn` methods

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_squared_error, make_scorer
```

Create a model and a scorer

```
rfc = RandomForestRegressor(n_estimators=20, max_depth=5, random_state=1111)
mse = make_scorer(mean_squared_error)
```

Run `cross_val_score()`

```
cv_results = cross_val_score(rfc, X, y, cv=5, scoring=mse)
```

```
print(cv_results)
```

```
[196.765, 108.563, 85.963, 222.594, 140.942]
```

Report the mean and standard deviation:

```
print('The mean: {}'.format(cv_results.mean()))
print('The std: {}'.format(cv_results.std()))
```

```
The mean: 150.965
```

```
The std: 51.676
```

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.

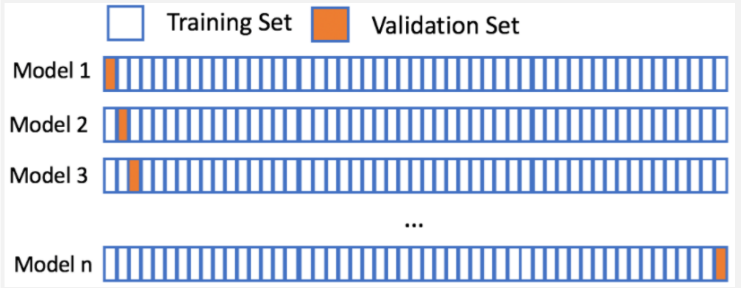
Before you begin trying different regression models, you have decided to run cross-validation on a simple random forest model to get a baseline error to compare with any future results.

```
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())
```

```
<script.py> output:
    155.55845080026586
```

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)

```
n = X.shape[0]
mse = make_scorer(mean_squared_error)
cv_results = cross_val_score(estimator, X, y, scoring=mse, cv=n)
```

```
print(cv_results)
```

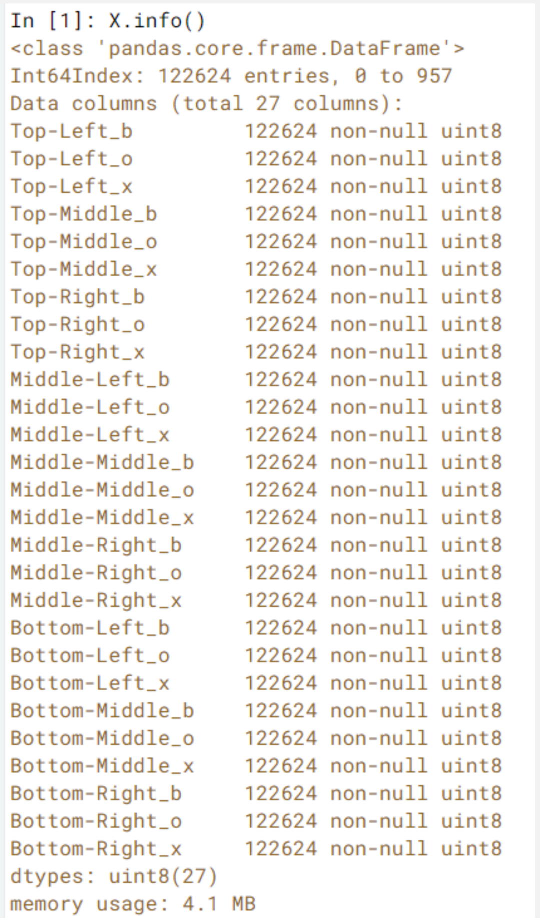
```
[5.45, 10.52, 6.23, 1.98, 11.27, 9.21, 4.65, ... ]
```

```
print(cv_results.mean())
```

```
6.32
```

When to use LOOCV

The `X` data has been loaded.



```
In [1]: X.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 122624 entries, 0 to 957
Data columns (total 27 columns):
Top-Left_b          122624 non-null uint8
Top-Left_o          122624 non-null uint8
Top-Left_x          122624 non-null uint8
Top-Middle_b        122624 non-null uint8
Top-Middle_o        122624 non-null uint8
Top-Middle_x        122624 non-null uint8
Top-Right_b         122624 non-null uint8
Top-Right_o         122624 non-null uint8
Top-Right_x         122624 non-null uint8
Middle-Left_b       122624 non-null uint8
Middle-Left_o       122624 non-null uint8
Middle-Left_x       122624 non-null uint8
Middle-Middle_b     122624 non-null uint8
Middle-Middle_o     122624 non-null uint8
Middle-Middle_x     122624 non-null uint8
Middle-Right_b      122624 non-null uint8
Middle-Right_o      122624 non-null uint8
Middle-Right_x      122624 non-null uint8
Bottom-Left_b       122624 non-null uint8
Bottom-Left_o       122624 non-null uint8
Bottom-Left_x       122624 non-null uint8
Bottom-Middle_b     122624 non-null uint8
Bottom-Middle_o     122624 non-null uint8
Bottom-Middle_x     122624 non-null uint8
Bottom-Right_b      122624 non-null uint8
Bottom-Right_o      122624 non-null uint8
Bottom-Right_x      122624 non-null uint8
dtypes: uint8(27)
memory usage: 4.1 MB
```

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

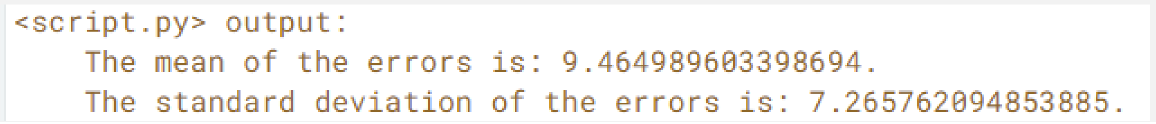
Answer: A & C. This many observations will definitely slow things down and could be computationally expensive. If you don't have time to wait while your computer runs through 1,000 models, you might want to use 5 or 10-fold cross-validation.

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))
```



```
<script.py> output:
    The mean of the errors is: 9.464989603398694.
    The standard deviation of the errors is: 7.265762094853885.
```

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

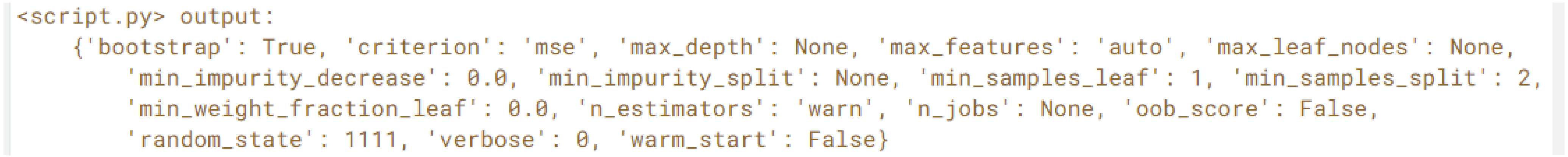
Creating Hyperparameters

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]
```



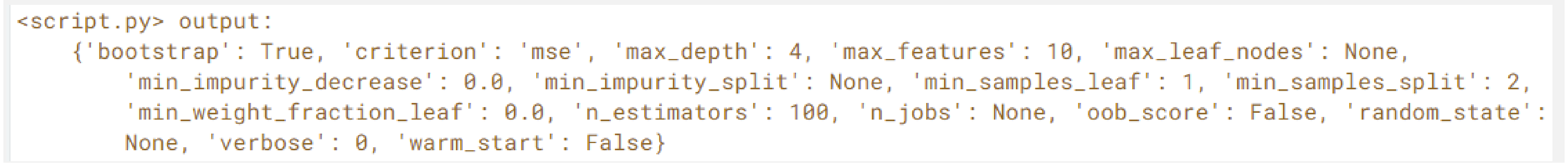
```
<script.py> output:
    {'bootstrap': True, 'criterion': 'mse', 'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None,
        'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2,
        'min_weight_fraction_leaf': 0.0, 'n_estimators': 'warn', 'n_jobs': None, 'oob_score': False,
        'random_state': 1111, 'verbose': 0, 'warm_start': False}
```

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())
```

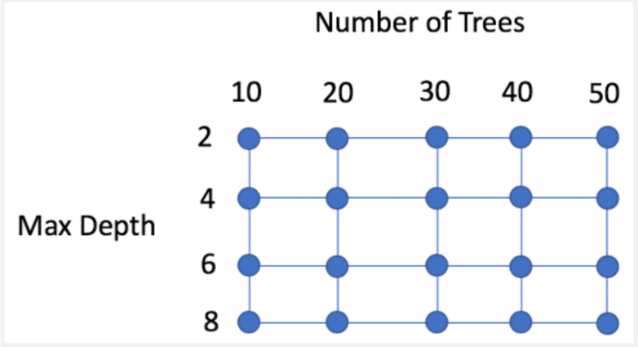


```
<script.py> output:
    {'bootstrap': True, 'criterion': 'mse', 'max_depth': 4, 'max_features': 10, 'max_leaf_nodes': None,
        'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2,
        'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state':
        None, 'verbose': 0, 'warm_start': False}
```

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.

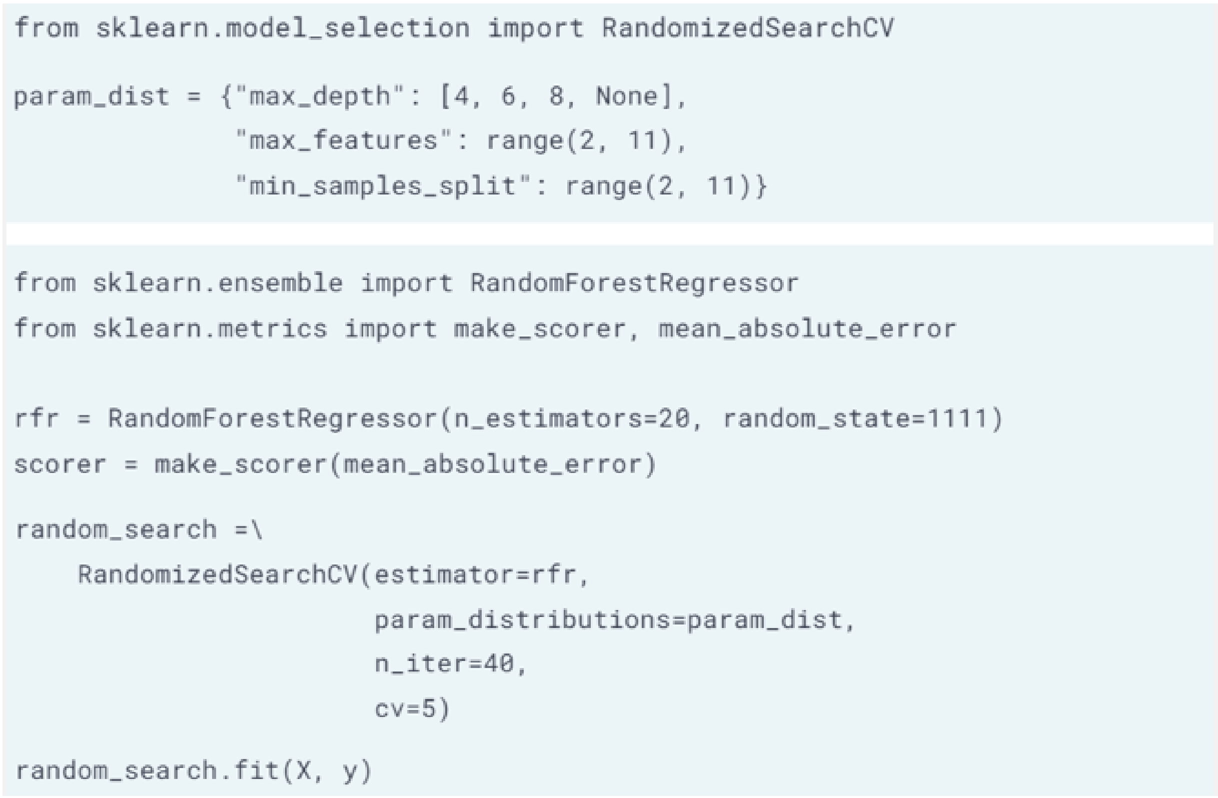


```
from sklearn.model_selection import RandomizedSearchCV

random_search = RandomizedSearchCV()
```

Parameter Distribution:

```
param_dist = {"max_depth": [4, 6, 8, None],
              "max_features": range(2, 11),
              "min_samples_split": range(2, 11)}
```

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.


```
from sklearn.model_selection import RandomizedSearchCV

param_dist = {"max_depth": [4, 6, 8, None],
              "max_features": range(2, 11),
              "min_samples_split": range(2, 11)}
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import make_scorer, mean_absolute_error

rfr = RandomForestRegressor(n_estimators=20, random_state=1111)
scorer = make_scorer(mean_absolute_error)

random_search =\
    RandomizedSearchCV(estimator=rfr,
                       param_distributions=param_dist,
                       n_iter=40,
                       cv=5)

random_search.fit(X, y)
```

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:

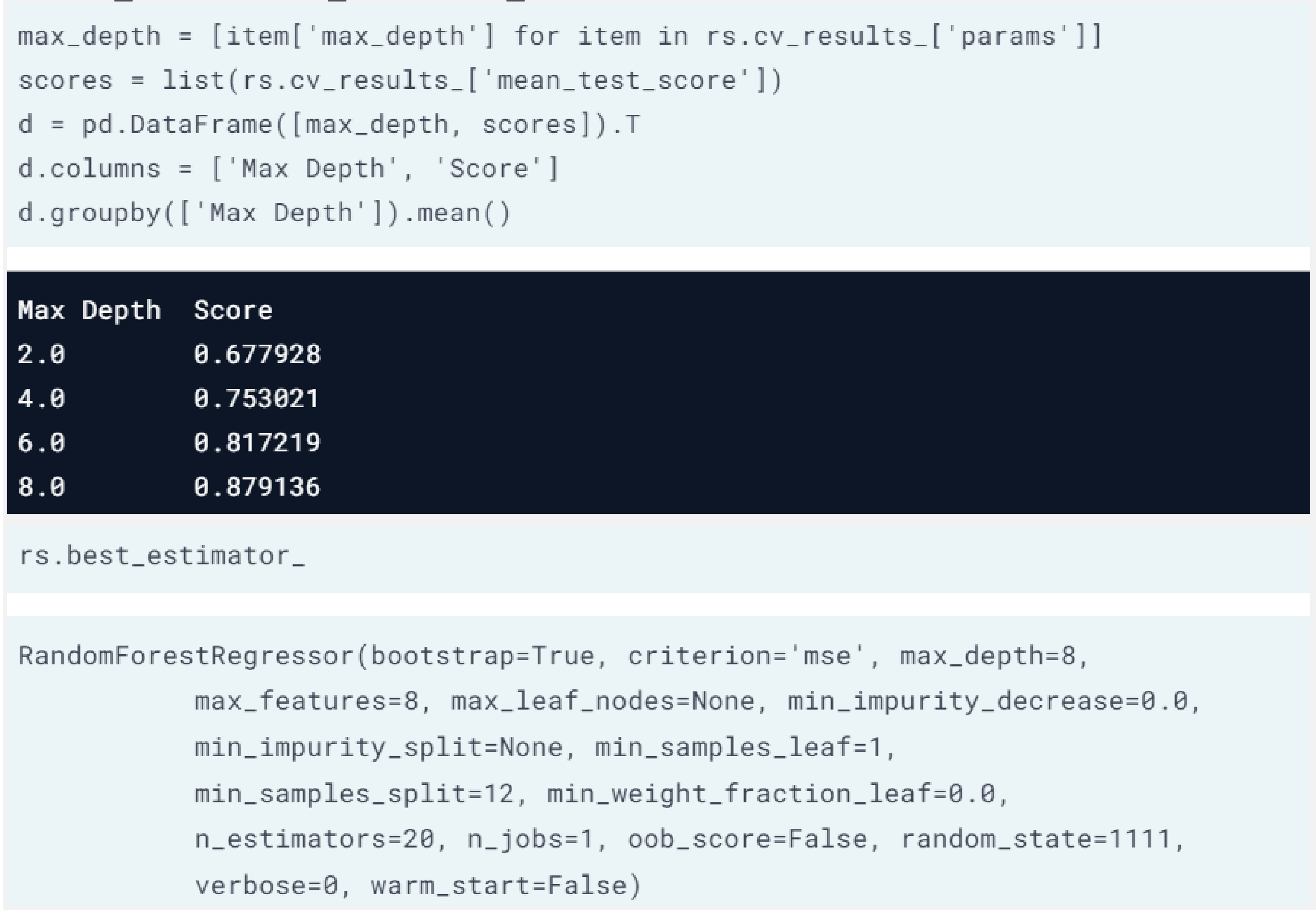
- `param_dist`: the hyperparameter distributions
- `rfr`: a random forest regression model
- `scorer`: a scoring method to use

```
# 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_
```



```
max_depth = [item['max_depth'] for item in rs.cv_results_['params']]
scores = list(rs.cv_results_['mean_test_score'])
d = pd.DataFrame([max_depth, scores]).T
d.columns = ['Max Depth', 'Score']
d.groupby(['Max Depth']).mean()
```

```
Max Depth  Score
2.0        0.677928
4.0        0.753021
6.0        0.817219
8.0        0.879136
```

```
rs.best_estimator_
```

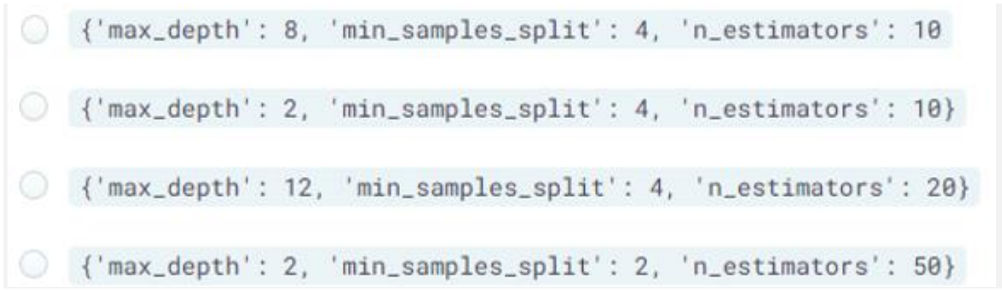
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=8,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=12, min_weight_fraction_leaf=0.0,
           n_estimators=20, n_jobs=1, oob_score=False, random_state=1111,
           verbose=0, warm_start=False)
```

This shows the model that performed the best during cross validation

Best classification accuracy

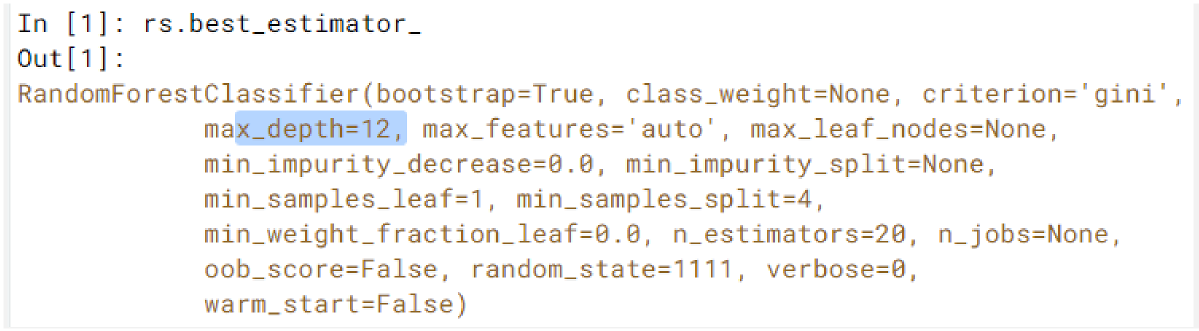
You are in a competition at work to build the best model for predicting the winner of a Tic-Tac-Toe game. You already ran a random search and saved the results of the most accurate model to `rs`.

Which parameter set produces the best classification accuracy?



- ○ `{'max_depth': 8, 'min_samples_split': 4, 'n_estimators': 10`
- ○ `{'max_depth': 2, 'min_samples_split': 4, 'n_estimators': 10}`
- ○ `{'max_depth': 12, 'min_samples_split': 4, 'n_estimators': 20}`
- ○ `{'max_depth': 2, 'min_samples_split': 2, 'n_estimators': 50}`

Answer: {'max_depth': 12, 'min_samples_split': 4, 'n_estimators': 20}

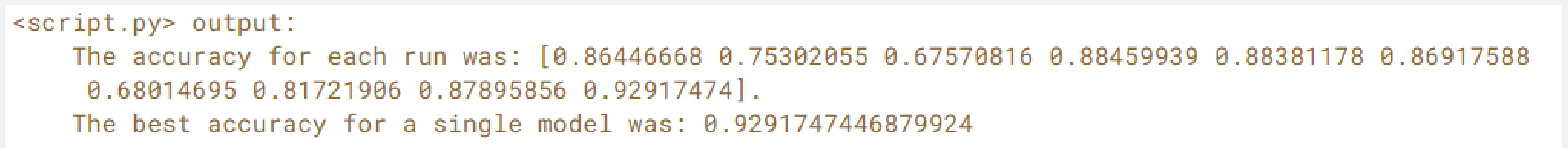


```
In [1]: rs.best_estimator_
Out[1]:
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=12, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=4,
            min_weight_fraction_leaf=0.0, n_estimators=20, n_jobs=None,
            oob_score=False, random_state=1111, verbose=0,
            warm_start=False)
```

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_))
```



```
<script.py> output:
    The accuracy for each run was: [0.86446668 0.75302055 0.67570816 0.88459939 0.88381178 0.86917588
     0.68014695 0.81721906 0.87895856 0.92917474].
    The best accuracy for a single model was: 0.9291747446879924
```

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:

- Basics of model validation
- Accuracy and evaluation metrics
- Splitting data into train, validation, and test sets
- Validation techniques
  Cross-validation and LOOCV
- tools for creating validated and high performing models
- Hyperparameter tuning

Happy learning!