Importing Libraries

```python
import numpy as np
import pandas as pd

# Set random seed to ensure reproducible runs
RSEED = 50
```

We create the features X and the labels y. There are only two features, which will allow us to visualize the data and which makes this a very easy problem.

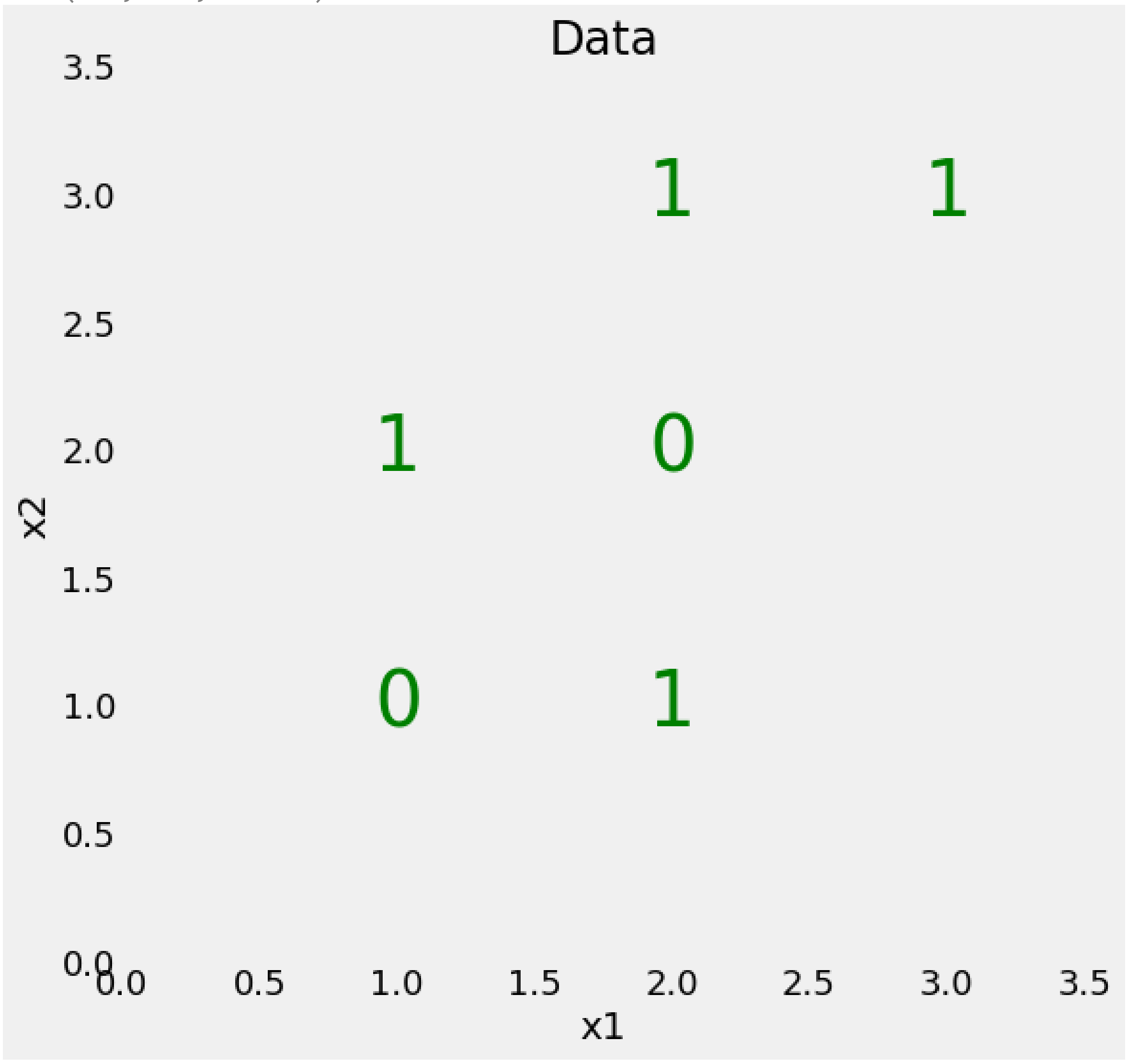
```python

X = np.array([[2, 2],
              [2, 1],
              [2, 3],
              [1, 2],
              [1, 1],
              [3, 3]])

y = np.array([0, 1, 1, 1, 0, 1])
```

Data Visualization

To get a sense of the data, we can graph the data points with the number showing the label.

```python
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

# Plot formatting
plt.style.use('fivethirtyeight')
plt.rcParams['font.size'] = 18
plt.figure(figsize = (8, 8))

# Plot each point as the label
for x1, x2, label in zip(X[:, 0], X[:, 1], y):
    plt.text(x1, x2, str(label), fontsize = 40, color = 'g',
             ha='center', va='center')

# Plot formatting
plt.grid(None);
plt.xlim((0, 3.5));
plt.ylim((0, 3.5));
plt.xlabel('x1', size = 20); plt.ylabel('x2', size = 20); plt.title('Data', size = 24)
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarni
  import pandas.util.testing as tm
Text(0.5, 1.0, 'Data')
```



Decision Tree

Here we quickly build and train a single decision tree on the data using Scikit-Learn. The tree will learn how to separate the points, building a flowchart of questions based on the feature values and the labels. At each stage, the decision tree makes splits by maximizing the reduction in Gini impurity.

We'll use the default hyperparameters for the decision tree which means it can grow as deep as necessary in order to completely separate the classes. This will lead to overfitting because the model memorizes the training data, and in practice, we usually want to limit the depth of the tree so it can generalize to testing data.

```python
from sklearn.tree import DecisionTreeClassifier

# Make a decision tree and train
tree = DecisionTreeClassifier(random_state=RSEED)
tree.fit(X, y)
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=50, splitter='best')
```

```
print(f'Decision tree has {tree.tree_.node_count} nodes with maximum depth {tree.tree_.
```

```
Decision tree has 9 nodes with maximum depth 3.
```

```
print(f'Model Accuracy: {tree.score(X, y)}')
```

```
Model Accuracy: 1.0
```

Visualize Decision Tree

To get a sense of how the decision tree "thinks", it's helpful to visualize the entire structure. This will show each node in the tree which we can use to make new predictions. Because the tree is relatively small, we can understand the entire image.

Gini Impurity

The Gini Impurity represents the probability that a randomly selected sample from the node will be incorrectly classified according to the distribution of samples in the node. At the top, there is a 44.4% chance that a randomly selected point would be incorrectly classified. The Gini Impurity is how the decision tree makes splits. It splits the samples based on the value of a feature that reduces the Gini Impurity by the largest amount. If we do the math, the average (weighted by number of samples) Gini Impurity decreases as we move down the tree.

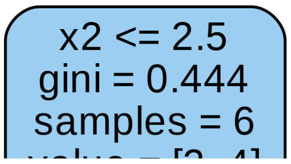
Eventually, the average Gini Impurity goes to 0.0 as we correctly classify each point. However, correctly classifying every single training point is usually not a good indicator because that means the model will not be able to generalize to the testing data! This model correclty classifies every single training point because we did not limit the maximum depth and during training, we give the model the answers as well as the features.

First we export the tree as a dot file making sure to label the features and the classes.

```
from sklearn.tree import export_graphviz

# Export as dot
export_graphviz(tree, 'tree.dot', rounded = True,
                feature_names = ['x1', 'x2'],
                class_names = ['0', '1'], filled = True)
```

```
from subprocess import call
# Convert to png
call(['dot', '-Tpng', 'tree.dot', '-o', 'tree.png', '-Gdpi=400']);
```

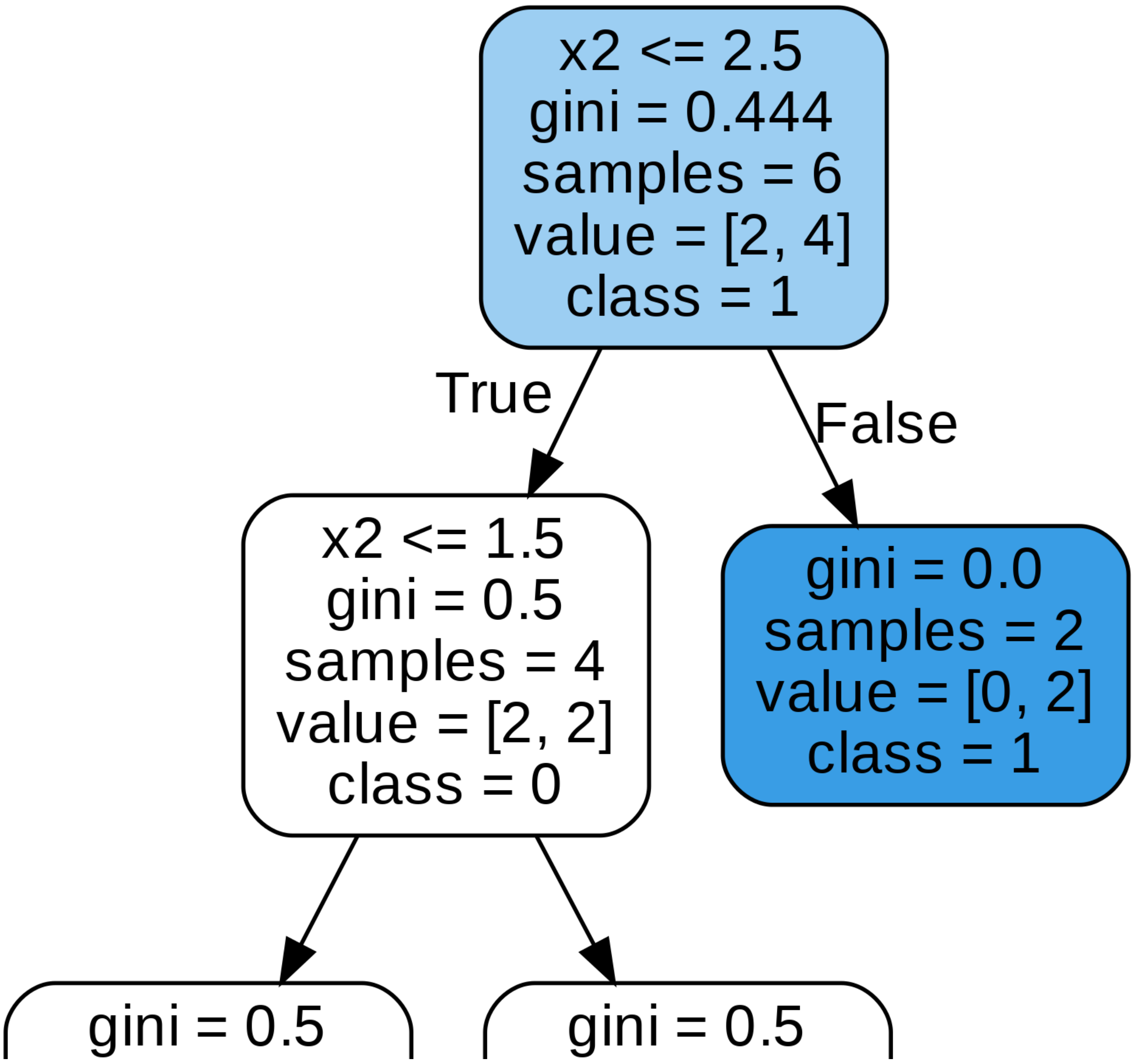
```

from IPython.display import Image
Image('tree.png')
```



```
# Limit maximum depth and train
short_tree = DecisionTreeClassifier(max_depth = 2, random_state=RSEED)
short_tree.fit(X, y)

print(f'Model Accuracy: {short_tree.score(X, y)}')
```

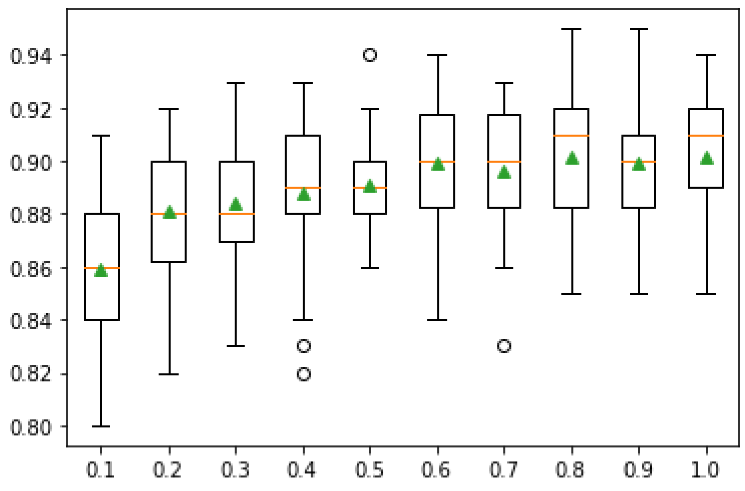
```
Model Accuracy: 0.6666666666666666
```

```

# Export as dot
export_graphviz(short_tree, 'shorttree.dot', rounded = True,
                feature_names = ['x1', 'x2'],
                class_names = ['0', '1'], filled = True)

call(['dot', '-Tpng', 'shorttree.dot', '-o', 'shorttree.png', '-Gdpi=400']);
Image('shorttree.png')
```



```
# explore random forest bootstrap sample size on performance
from numpy import mean
from numpy import std
from numpy import arange
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redunda
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # explore ratios from 10% to 100% in 10% increments
  for i in arange(0.1, 1.1, 0.1):
    key = '%.1f' % i
    # set max_samples=None to use 100%
    if i == 1.0:
      i = None
    models[key] = RandomForestClassifier(max_samples=i)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

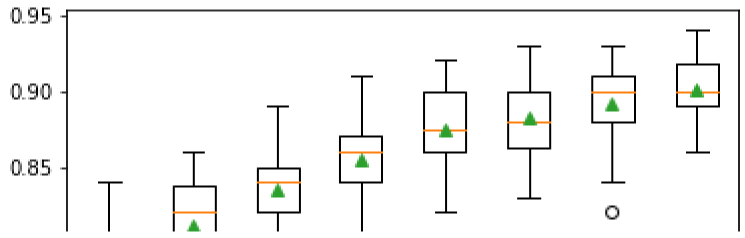
```
>0.1 0.859 (0.028)
>0.2 0.881 (0.025)
>0.3 0.884 (0.021)
>0.4 0.888 (0.028)
>0.5 0.891 (0.019)
>0.6 0.900 (0.023)
>0.7 0.897 (0.024)
>0.8 0.901 (0.024)
>0.9 0.900 (0.026)
>1.0 0.901 (0.026)
```



```
# explore random forest tree depth effect on performance
```

```
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from matplotlib import pyplot

# get the dataset
def get_dataset():
  X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_redunda
  return X, y

# get a list of models to evaluate
def get_models():
  models = dict()
  # consider tree depths from 1 to 7 and None=full
  depths = [i for i in range(1,8)] + [None]
  for n in depths:
    models[str(n)] = RandomForestClassifier(max_depth=n)
  return models

# evaluate a given model using cross-validation
def evaluate_model(model, X, y):
  # define the evaluation procedure
  cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
  # evaluate the model and collect the results
  scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)
  return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
results, names = list(), list()
for name, model in models.items():
  # evaluate the model
  scores = evaluate_model(model, X, y)
  # store the results
  results.append(scores)
  names.append(name)
  # summarize the performance along the way
  print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.show()
```

```
>1 0.766 (0.045)
>2 0.811 (0.039)
>3 0.836 (0.032)
>4 0.854 (0.027)
>5 0.874 (0.028)
>6 0.882 (0.026)
>7 0.891 (0.027)
>None 0.901 (0.022)
```



Random Forest Classifer using a dataset

```
#Start by connecting gdrive into the google colab

from google.colab import drive

drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
# Pandas is used for data manipulation
import pandas as pd
# Read in data and display first 5 rows
features = pd.read_csv('/content/gdrive/My Drive/Colab Notebooks/clustering/temps.csv')
features.head(5)
```

	year	month	day	week	temp_2	temp_1	average	actual	forecast_noaa	forecast_
0	2016	1	1	Fri	45	45	45.6	45	43	
1	2016	1	2	Sat	44	45	45.7	44	41	
2	2016	1	3	Sun	45	44	45.8	41	43	
3	2016	1	4	Mon	44	41	45.9	40	44	
4	2016	1	5	Tues	41	40	46.0	44	46	

```
print('The shape of our features is:', features.shape)

```

```
The shape of our features is: (348, 12)
```

```
# Descriptive statistics for each column
features.describe()
```

	year	month	day	temp_2	temp_1	average	actual
count	348.0	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000
mean	2016.0	6.477011	15.514368	62.652299	62.701149	59.760632	62.543103
std	0.0	3.498380	8.772982	12.165398	12.120542	10.527306	11.794146
min	2016.0	1.000000	1.000000	35.000000	35.000000	45.100000	35.000000
25%	2016.0	3.000000	8.000000	54.000000	54.000000	49.975000	54.000000
50%	2016.0	6.000000	15.000000	62.500000	62.500000	58.200000	62.500000

```
# One-hot encode the data using pandas get_dummies
features = pd.get_dummies(features)
# Display the first 5 rows of the last 12 columns
features.iloc[:,5:].head(5)
```

	average	actual	forecast_noaa	forecast_acc	forecast_under	friend	week_Fri
0	45.6	45	43	50	44	29	1
1	45.7	44	41	50	44	61	0
2	45.8	41	43	46	47	56	0
3	45.9	40	44	48	46	53	0
4	46.0	44	46	46	46	41	0

```
# Use numpy to convert to arrays
import numpy as np
# Labels are the values we want to predict
labels = np.array(features['actual'])
# Remove the labels from the features
# axis 1 refers to the columns
features= features.drop('actual', axis = 1)
# Saving feature names for later use
feature_list = list(features.columns)
# Convert to numpy array
features = np.array(features)
```

```
# Using Scikit-learn to split data into training and testing sets
from sklearn.model_selection import train_test_split
# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels = train_test_split(features, 

```

```
print('Training Features Shape:', train_features.shape)
print('Training Labels Shape:', train_labels.shape)
print('Testing Features Shape:', test_features.shape)
print('Testing Labels Shape:', test_labels.shape)
```

```
Training Features Shape: (261, 17)
Training Labels Shape: (261,)
Testing Features Shape: (87, 17)
Testing Labels Shape: (87,)
```

```
# The baseline predictions are the historical averages
baseline_preds = test_features[:, feature_list.index('average')]
# Baseline errors, and display average baseline error
baseline_errors = abs(baseline_preds - test_labels)
print('Average baseline error: ', round(np.mean(baseline_errors), 2))
```

```
Average baseline error:  5.06
```

```
# Import the model we are using
from sklearn.ensemble import RandomForestRegressor
# Instantiate model with 1000 decision trees
rf = RandomForestRegressor(n_estimators = 1000, random_state = 42)
# Train the model on training data
rf.fit(train_features, train_labels);
```

```
# Use the forest's predict method on the test data
predictions = rf.predict(test_features)
# Calculate the absolute errors
errors = abs(predictions - test_labels)
# Print out the mean absolute error (mae)
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
```

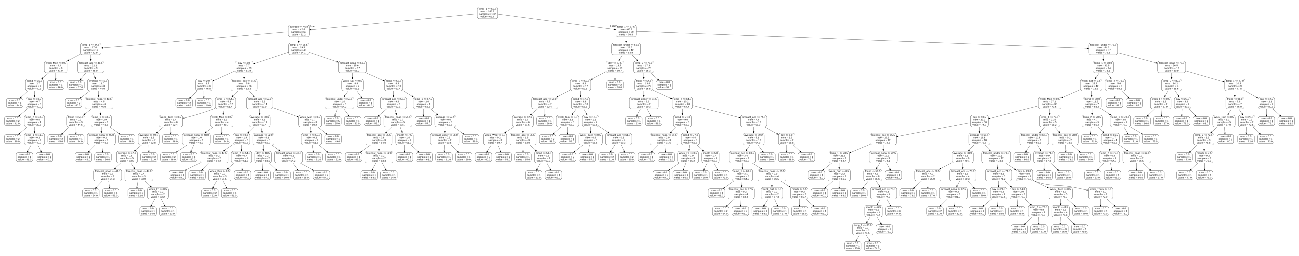
```
Mean Absolute Error: 3.87 degrees.
```

```
# Calculate mean absolute percentage error (MAPE)
mape = 100 * (errors / test_labels)
# Calculate and display accuracy
accuracy = 100 - np.mean(mape)
print('Accuracy:', round(accuracy, 2), '%.')

```

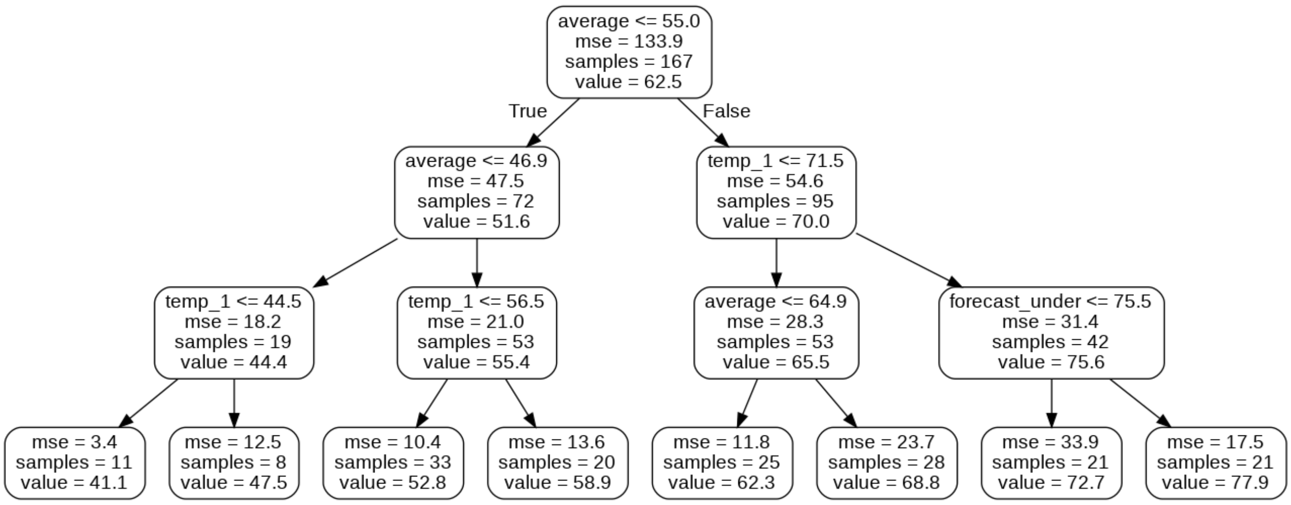
```
Accuracy: 93.93 %.
```

```
# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
# Pull out one tree from the forest
tree = rf.estimators_[5]
# Import tools needed for visualization
from sklearn.tree import export_graphviz
import pydot
# Pull out one tree from the forest
tree = rf.estimators_[5]
# Export the image to a dot file
export_graphviz(tree, out_file = 'tree1.dot', feature_names = feature_list, rounded = 1
# Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('tree1.dot')
# Write graph to a png file
```

```
graph.write_png('tree1.png')
from IPython.display import Image
Image('tree1.png')
```



```
# Limit depth of tree to 3 levels
rf_small = RandomForestRegressor(n_estimators=10, max_depth = 3)
rf_small.fit(train_features, train_labels)
# Extract the small tree
tree_small = rf_small.estimators_[5]
# Save the tree as a png image
export_graphviz(tree_small, out_file = 'small_tree.dot', feature_names = feature_list,
(graph, ) = pydot.graph_from_dot_file('small_tree.dot')
graph.write_png('small_tree.png');
from IPython.display import Image
Image('small_tree.png')


```



```
# Get numerical feature importances
```

```
importances = list(rf.feature_importances_)
# List of tuples with variable and importance
feature_importances = [(feature, round(importance, 2)) for feature, importance in zip(f
# Sort the feature importances by most important first
feature_importances = sorted(feature_importances, key = lambda x: x[1], reverse = True)
# Print out the feature and importances
[print('Variable: {:20} Importance: {}'.format(*pair)) for pair in feature_importances]
```

```
Variable: temp_1               Importance: 0.66
Variable: average              Importance: 0.15
Variable: forecast_noaa        Importance: 0.05
Variable: forecast_acc         Importance: 0.03
Variable: day                  Importance: 0.02
Variable: temp_2               Importance: 0.02
Variable: forecast_under       Importance: 0.02
Variable: friend               Importance: 0.02
Variable: month                Importance: 0.01
Variable: year                 Importance: 0.0
Variable: week_Fri             Importance: 0.0
Variable: week_Mon             Importance: 0.0
Variable: week_Sat             Importance: 0.0
Variable: week_Sun             Importance: 0.0
Variable: week_Thurs           Importance: 0.0
Variable: week_Tues            Importance: 0.0
Variable: week_Wed             Importance: 0.0
```

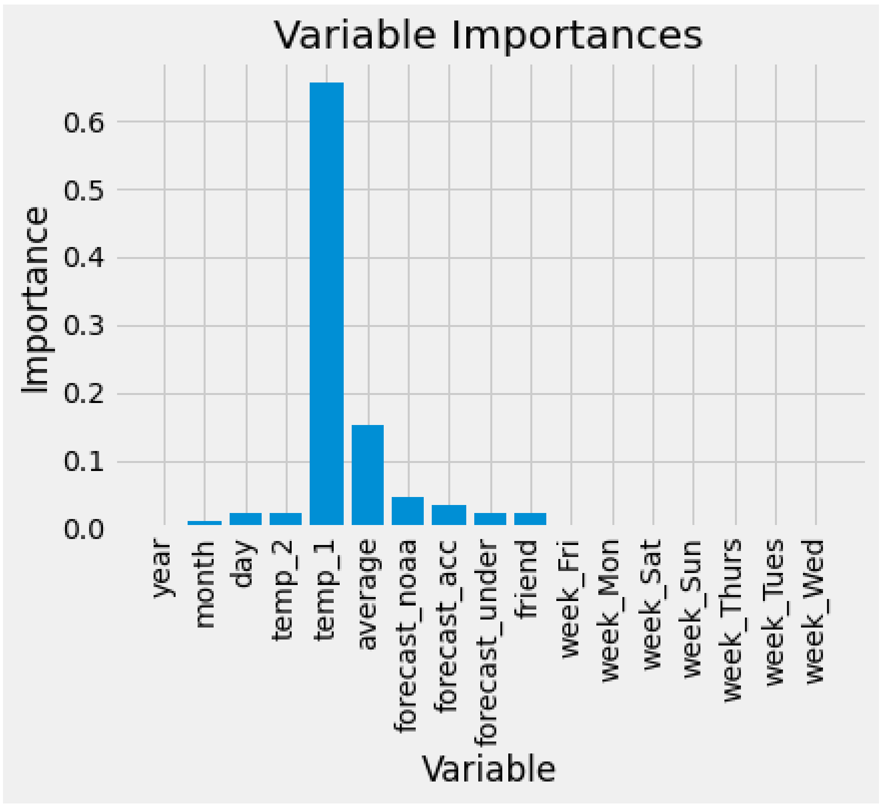
Double-click (or enter) to edit

```
# New random forest with only the two most important variables
rf_most_important = RandomForestRegressor(n_estimators= 1000, random_state=42)
# Extract the two most important features
important_indices = [feature_list.index('temp_1'), feature_list.index('average')]
train_important = train_features[:, important_indices]
test_important = test_features[:, important_indices]
# Train the random forest
rf_most_important.fit(train_important, train_labels)
# Make predictions and determine the error
predictions = rf_most_important.predict(test_important)
errors = abs(predictions - test_labels)
# Display the performance metrics
print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')
mape = np.mean(100 * (errors / test_labels))
accuracy = 100 - mape
print('Accuracy:', round(accuracy, 2), '%.')
```

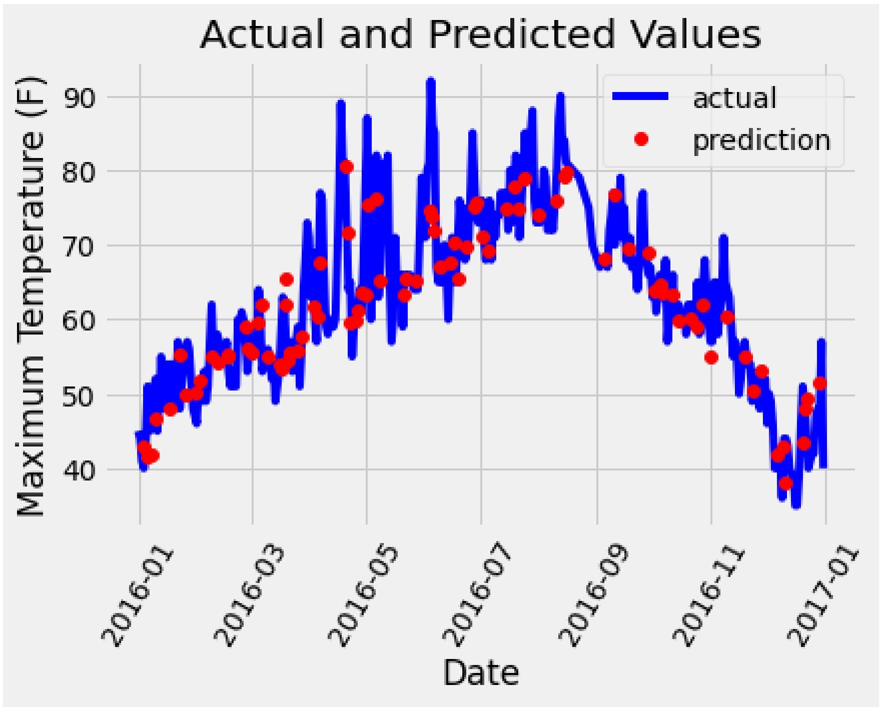
```
Mean Absolute Error: 3.92 degrees.
Accuracy: 93.76 %.
```

```
# Import matplotlib for plotting and use magic command for Jupyter Notebooks
import matplotlib.pyplot as plt
%matplotlib inline
# Set the style
plt.style.use('fivethirtyeight')
# list of x locations for plotting
x_values = list(range(len(importances)))
```

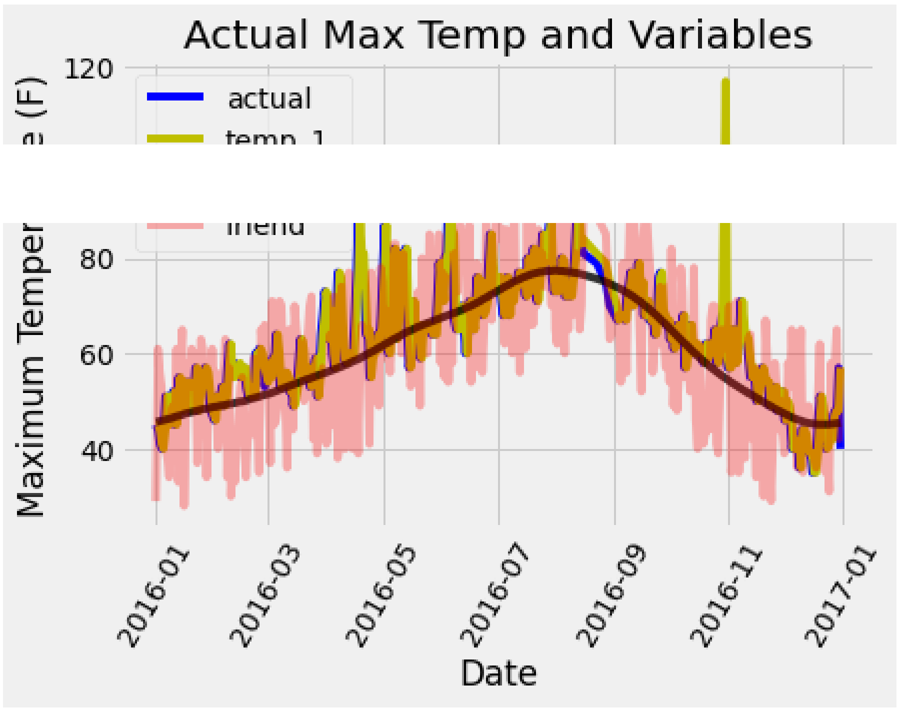
```
# Make a bar chart
plt.bar(x_values, importances, orientation = 'vertical')
# Tick labels for x axis
plt.xticks(x_values, feature_list, rotation='vertical')
# Axis labels and title
plt.ylabel('Importance'); plt.xlabel('Variable'); plt.title('Variable Importances');
```



```
# Use datetime for creating date objects for plotting
import datetime
# Dates of training values
months = features[:, feature_list.index('month')]
days = features[:, feature_list.index('day')]
years = features[:, feature_list.index('year')]
# List and then convert to datetime object
dates = [str(int(year)) + '-' + str(int(month)) + '-' + str(int(day)) for year, month,
dates = [datetime.datetime.strptime(date, '%Y-%m-%d') for date in dates]
# Dataframe with true values and dates
true_data = pd.DataFrame(data = {'date': dates, 'actual': labels})
# Dates of predictions
months = test_features[:, feature_list.index('month')]
days = test_features[:, feature_list.index('day')]
years = test_features[:, feature_list.index('year')]
# Column of dates
test_dates = [str(int(year)) + '-' + str(int(month)) + '-' + str(int(day)) for year, mo
# Convert to datetime objects
test_dates = [datetime.datetime.strptime(date, '%Y-%m-%d') for date in test_dates]
# Dataframe with predictions and dates
predictions_data = pd.DataFrame(data = {'date': test_dates, 'prediction': predictions})
# Plot the actual values
plt.plot(true_data['date'], true_data['actual'], 'b-', label = 'actual')
# Plot the predicted values
plt.plot(predictions_data['date'], predictions_data['prediction'], 'ro', label = 'predi
plt.xticks(rotation = '60');
plt.xticks(rotation = '60');
plt.legend()
# Graph labels
plt.xlabel('Date'); plt.ylabel('Maximum Temperature (F)'); plt.title('Actual and Predic
```



```
# Make the data accessible for plotting
true_data['temp_1'] = features[:, feature_list.index('temp_1')]
true_data['average'] = features[:, feature_list.index('average')]
true_data['friend'] = features[:, feature_list.index('friend')]
# Plot all the data as lines
plt.plot(true_data['date'], true_data['actual'], 'b-', label  = 'actual', alpha = 1.0)
plt.plot(true_data['date'], true_data['temp_1'], 'y-', label  = 'temp_1', alpha = 1.0)
plt.plot(true_data['date'], true_data['average'], 'k-', label = 'average', alpha = 0.8)
plt.plot(true_data['date'], true_data['friend'], 'r-', label = 'friend', alpha = 0.3)
# Formatting plot
plt.legend(); plt.xticks(rotation = '60');
# Lables and title
plt.xlabel('Date'); plt.ylabel('Maximum Temperature (F)'); plt.title('Actual Max Temp a

```


Actual Max Temp and Variables
actual
temp_1
friend
Maximum Temperature (F)
120
80
60
40
2016-01
2016-03
2016-05
2016-07
2016-09
2016-11
2017-01
Date

1