The Random Forest Algorithm

towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

Niklas Donges

February 22, 2018

Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time. It is also one of the most used algorithms, because it's simplicity and the fact that it can be used for both classification and regression tasks. In this post, you are going to learn, how the random forest algorithm works and several other important things about it.

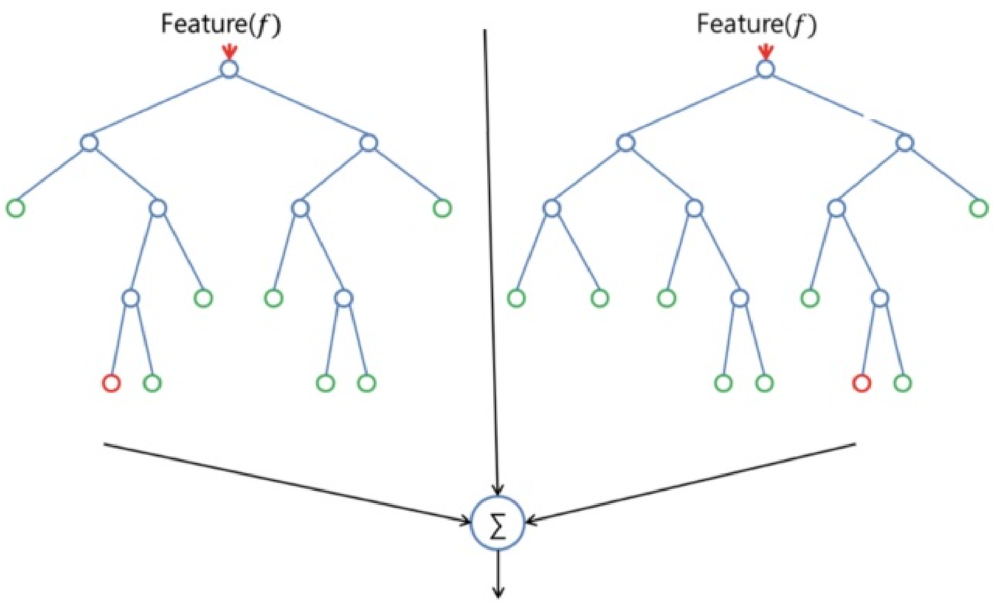
Table of Contents:

How it works:

Random Forest is a supervised learning algorithm. Like you can already see from it's name, it creates a forest and makes it somehow random. The „forest" it builds, is an ensemble of Decision Trees, most of the time trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result.

> To say it in simple words: Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

One big advantage of random forest is, that it can be used for both classification and regression problems, which form the majority of current machine learning systems. I will talk about random forest in classification, since classification is sometimes considered the building block of machine learning. Below you can see how a random forest would look like with two trees:



Random Forest has nearly the same hyperparameters as a decision tree or a bagging classifier. Fortunately, you don't have to combine a decision tree with a bagging classifier and can just easily use the classifier-class of Random Forest. Like I already said, with Random Forest, you can also deal with Regression tasks by using the Random Forest regressor.

Random Forest adds additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model.

Therefore, in Random Forest, only a random subset of the features is taken into consideration by the algorithm for splitting a node. You can even make trees more random, by additionally using random thresholds for each feature rather than searching for the best possible thresholds (like a normal decision tree does).

Real Life Analogy:

Imagine a guy named Andrew, that want’s to decide, to which places he should travel during a one-year vacation trip. He asks people who know him for advice. First, he goes to a friend, tha asks Andrew where he traveled to in the past and if he liked it or not. Based on the answers, he will give Andrew some advice.

This is a typical decision tree algorithm approach. Andrews friend created rules to guide his decision about what he should recommend, by using the answers of Andrew.

Afterwards, Andrew starts asking more and more of his friends to advise him and they again ask him different questions, where they can derive some recommendations from. Then he chooses the places that where recommend the most to him, which is the typical Random Forest algorithm approach.

Feature Importance:

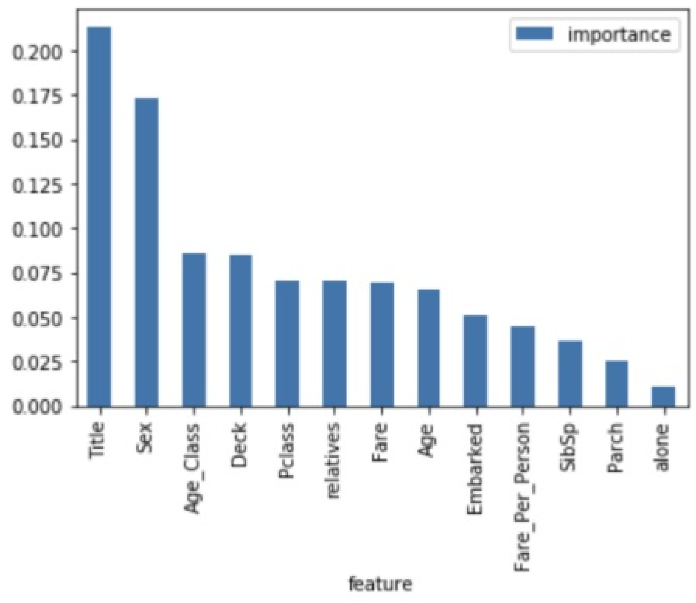
Another great quality of the random forest algorithm is that it is very easy to measure the relative importance of each feature on the prediction. Sklearn provides a great tool for this, that measures a features importance by looking at how much the tree nodes, which use that feature, reduce impurity across all trees in the forest. It computes this score automatically for each feature after training and scales the results, so that the sum of all importance is equal to 1.

If you don’t know how a decision tree works and if you don’t know what a leaf or node is, here is a good description from Wikipedia: In a decision tree each internal node represents a “test” on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes).A node that has no children is a leaf.

Through looking at the feature importance, you can decide which features you may want to drop, because they don’t contribute enough or nothing to the prediction process. This is important, because a general rule in machine learning is that the more features you have, the more likely your model will suffer from overfitting and vice versa.

Below you can see a table and a visualization that show the importance of 13 features, which I used during a supervised classification project with the famous Titanic dataset on kaggle. You can find the whole project here.

feature	importance
Title	0.213
Sex	0.173
Age_Class	0.086
Deck	0.085
Pclass	0.071
relatives	0.070
Fare	0.069
Age	0.065
Embarked	0.051
Fare_Per_Person	0.045
SibSp	0.037
Parch	0.025
alone	0.011



Difference between Decision Trees and Random Forests:

Like I already mentioned, Random Forest is a collection of Decision Trees, but there are some differences.

If you input a training dataset with features and labels into a decision tree, it will formulate some set of rules, which will be used to make the predictions.

For example, if you want to predict whether a person will click on an online advertisement, you could collect the ad's the person clicked in the past and some features that describe his decision. If you put the features and labels into a decision tree, it will generate some rules. Then you can predict whether the advertisement will be clicked or not. In comparison, the Random Forest algorithm randomly selects observations and features to build several decision trees and then averages the results.

Another difference is that „deep" decision trees might suffer from overfitting. Random Forest prevents overfitting most of the time, by creating random subsets of the features and building smaller trees using these subsets. Afterwards, it combines the subtrees. Note that this doesn't work every time and that it also makes the computation slower, depending on how many trees your random forest builds.

Important Hyperparameters:

The Hyperparameters in random forest are either used to increase the predictive power of the model or to make the model faster. I will here talk about the hyperparameters of sklearns built-in random forest function.

1. Increasing the Predictive Power

Firstly, there is the **„n_estimators“** hyperparameter, which is just the number of trees the algorithm builds before taking the maximum voting or taking averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation.

Another important hyperparameter is **„max_features“**, which is the maximum number of features Random Forest is allowed to try in an individual tree. Sklearn provides several options, described in their documentation.

The last important hyper-parameter we will talk about in terms of speed, is **„min_sample_leaf “**. This determines, like its name already says, the minimum number of leafs that are required to split an internal node.

2. Increasing the Models Speed

The **„n_jobs“** hyperparameter tells the engine how many processors it is allowed to use. If it has a value of 1, it can only use one processor. A value of “-1” means that there is no limit.

„random_state“ makes the model’s output replicable. The model will always produce the same results when it has a definite value of random_state and if it has been given the same hyperparameters and the same training data.

Lastly, there is the **„oob_score“** (also called oob sampling), which is a random forest cross validation method. In this sampling, about one-third of the data is not used to train the model and can be used to evaluate its performance. These samples are called the out of bag samples. It is very similar to the leave-one-out cross-validation method, but almost no additional computational burden goes along with it.

Advantages and Disadvantages:

Like I already mentioned, an advantage of random forest is that it can be used for both regression and classification tasks and that it’s easy to view the relative importance it assigns to the input features.

Random Forest is also considered as a very handy and easy to use algorithm, because it’s default hyperparameters often produce a good prediction result. The number of hyperparameters is also not that high and they are straightforward to understand.

One of the big problems in machine learning is overfitting, but most of the time this won’t happen that easy to a random forest classifier. That’s because if there are enough trees in the forest, the classifier won’t overfit the model.

The main limitation of Random Forest is that a large number of trees can make the algorithm to slow and ineffective for real-time predictions. In general, these algorithms are fast to train, but quite slow to create predictions once they are trained. A more accurate prediction requires more trees, which results in a slower model. In most real-world applications the random forest algorithm is fast enough, but there can certainly be situations where run-time performance is important and other approaches would be preferred.

And of course Random Forest is a predictive modeling tool and not a descriptive tool. That means, if you are looking for a description of the relationships in your data, other approaches would be preferred.

Use Cases:

The random forest algorithm is used in a lot of different fields, like Banking, Stock Market, Medicine and E-Commerce. In Banking it is used for example to detect customers who will use the bank's services more frequently than others and repay their debt in time. In this domain it is also used to detect fraud customers who want to scam the bank. In finance, it is used to determine a stock's behaviour in the future. In the healthcare domain it is used to identify the correct combination of components in medicine and to analyze a patient's medical history to identify diseases. And lastly, in E-commerce random forest is used to determine whether a customer will actually like the product or not.

Summary:

Random Forest is a great algorithm to train early in the model development process, to see how it performs and it's hard to build a "bad" Random Forest, because of its simplicity. This algorithm is also a great choice, if you need to develop a model in a short period of time. On top of that, it provides a pretty good indicator of the importance it assigns to your features.

Random Forests are also very hard to beat in terms of performance. Of course you can probably always find a model that can perform better, like a neural network, but these usually take much more time in the development. And on top of that, they can handle a lot of different feature types, like binary, categorical and numerical.

Overall, Random Forest is a (mostly) fast, simple and flexible tool, although it has its limitations.

This post was initially published at my blog (https://machinelearning-blog.com).

An Implementation and Explanation of the Random Forest in Python

towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76

William Koehrsen August 30, 2018

(Source)

A guide for using and understanding the random forest by building up from a single decision tree.

Fortunately, with libraries such as Scikit-Learn, it's now easy to implement hundreds of machine learning algorithms in Python. It's so easy that we often don't need any underlying knowledge of how the model works in order to use it. While knowing *all* the details is not necessary, it's still helpful to have an idea of how a machine learning model works under the hood. This lets us diagnose the model when it's underperforming or explain how it makes decisions, which is crucial if we want to convince others to trust our models.

In this article, we'll look at how to build and use the Random Forest in Python. In addition to seeing the code, we'll try to get an understanding of how this model works. Because a random forest in made of many decision trees, we'll start by understanding how a single decision tree makes classifications on a simple problem. Then, we'll work our way to using a random forest on a real-world data science problem. The complete code for this article is available as a Jupyter Notebook on GitHub.

Note: this article originally appeared on enlight, a community-driven, open-source platform with tutorials for those looking to study machine learning.

Understanding a Decision Tree

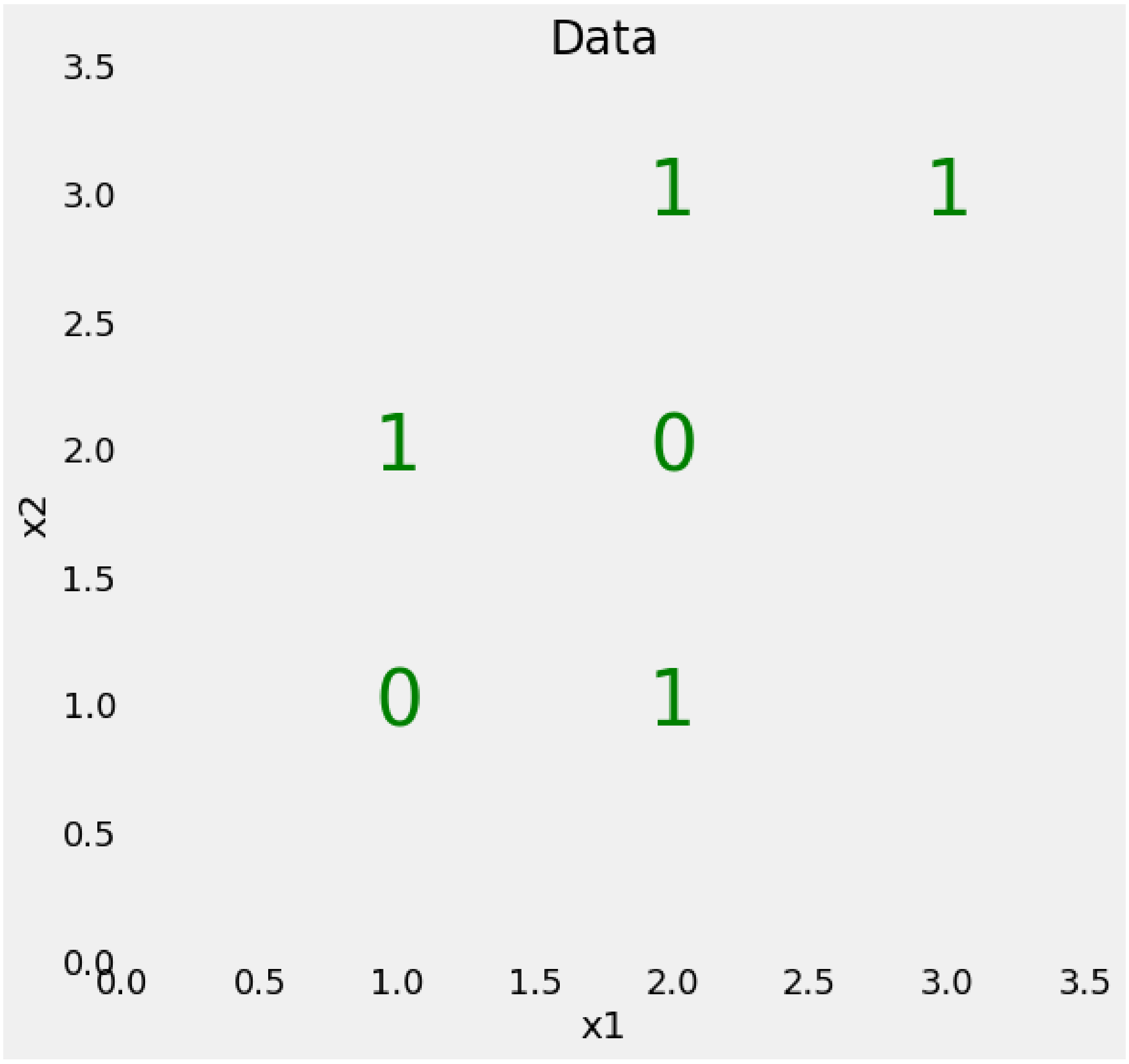
A decision tree is the building block of a random forest and is an intuitive model. We can think of a decision tree as a series of yes/no questions asked about our data eventually leading to a predicted class (or continuous value in the case of regression). This is an interpretable model because it makes classifications much like we do: we ask a sequence of queries about the available data we have until we arrive at a decision (in an ideal world).

The technical details of a decision tree are in how the questions about the data are formed. In the CART algorithm, a decision tree is built by determining the questions (called splits of nodes) that, when answered, lead to the greatest reduction in Gini Impurity. What this means is the decision tree tries to form nodes containing a high proportion of samples (data points) from a single class by finding values in the features that cleanly divide the data into classes.

We'll talk in low-level detail about Gini Impurity later, but first, let's build a Decision Tree so we can understand it on a high level.

Decision Tree on Simple Problem

We'll start with a very simple binary classification problem as shown below:



The goal is to divide the data points into their respective classes.

Our data only has two features (predictor variables), `x1` and `x2` with 6 data points — samples — divided into 2 different labels. Although this problem is simple, it's not linearly separable, which means that we can't draw a single straight line through the data to classify the points.

We can however draw a series of straight lines that divide the data points into boxes, which we'll call nodes. In fact, this is what a decision tree does during training. Effectively, a decision tree is a non-linear model built by constructing many linear boundaries.

To create a decision tree and train (`fit`) it on the data, we use Scikit-Learn.

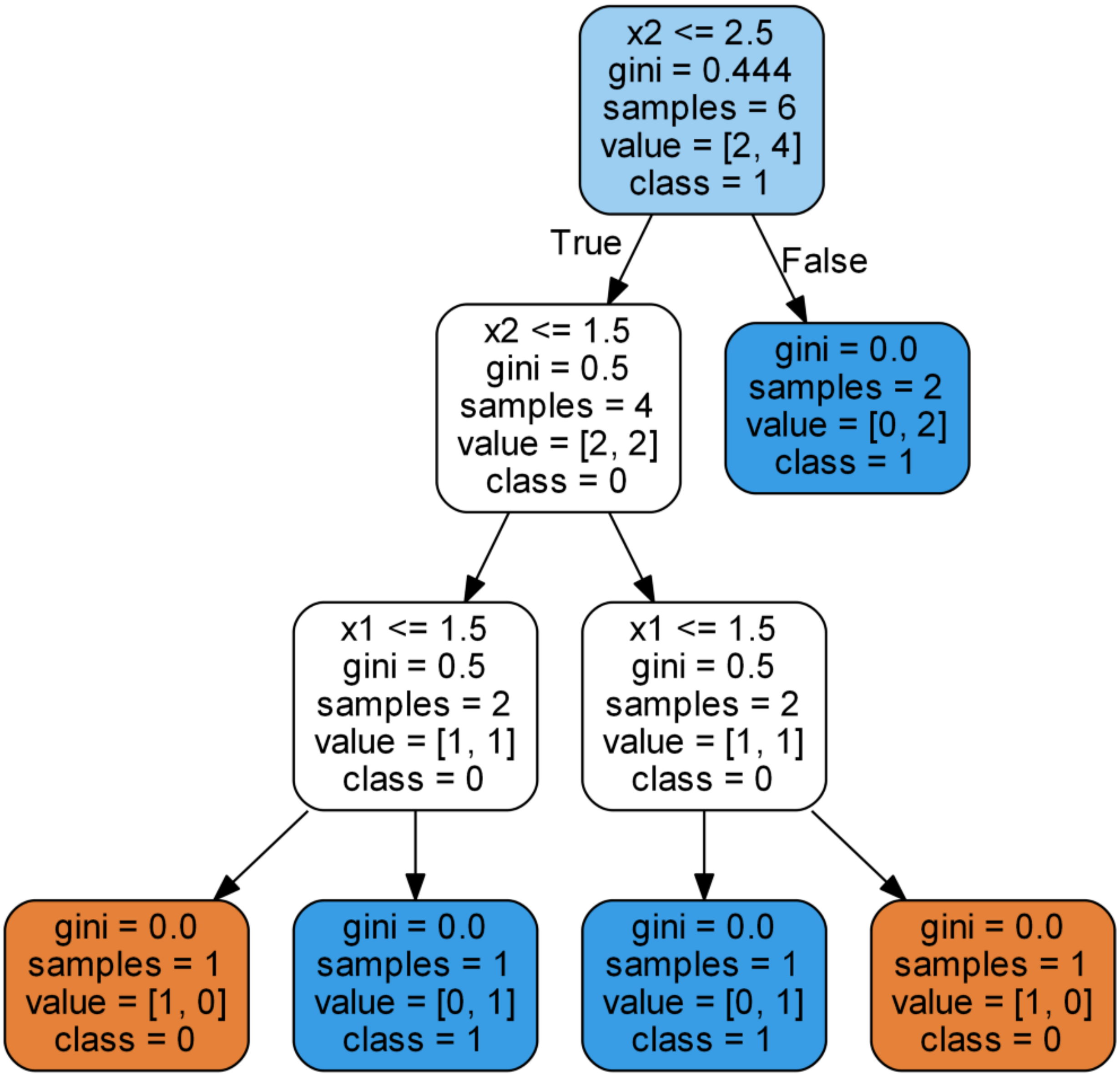
During training we give the model both the features and the labels so it can learn to classify points based on the features. (We don't have a testing set for this simple problem, but when testing, we only give the model the features and have it make predictions about the labels.)

We can test the accuracy of our model on the training data:

We see that it gets 100% accuracy, which is what we expect because we gave it the answers (`y`) for training and did not limit the depth of the tree. It turns out this ability to completely learn the training data can be a downside of a decision tree because it may lead to *overfitting* as we'll discuss later.

Visualizing a Decision Tree

So what's actually going on when we train a decision tree? I find a helpful way to understand the decision tree is by visualizing it, which we can do using a Scikit-Learn function (for details check out the notebook or this article).



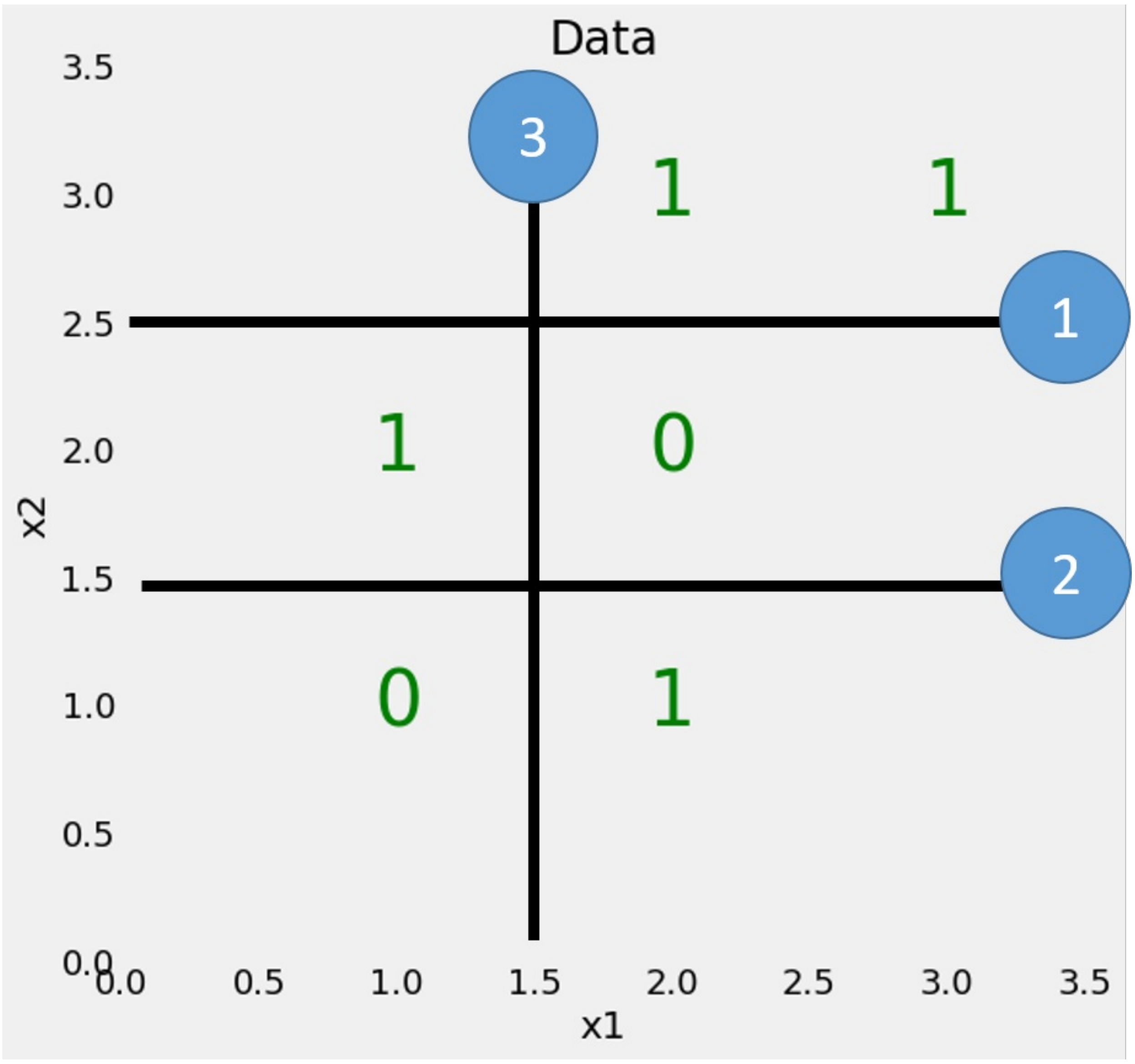
Simple decision tree

All the nodes, except the leaf nodes (colored terminal nodes), have 5 parts:

1. Question asked about the data based on a value of a feature. Each question has either a True or False answer that splits the node. Based on the answer to the question, a data point moves down the tree.
2. `gini` : The Gini Impurity of the node. The average weighted Gini Impurity decreases as we move down the tree.
3. `samples` : The number of observations in the node.
4. `value` : The number of samples in each class. For example, the top node has 2 samples in class 0 and 4 samples in class 1.
5. `class` : The majority classification for points in the node. In the case of leaf nodes, this is the prediction for all samples in the node.

The leaf nodes do not have a question because these are where the final predictions are made. To classify a new point, simply move down the tree, using the features of the point to answer the questions until you arrive at a leaf node where the `class` is the prediction.

To make see the tree in a different way, we can draw the splits built by the decision tree on the original data.



Splits made by the decision tree.

Each split is a single line that divides data points into nodes based on feature values. For this simple problem and with no limit on the maximum depth, the divisions place each point in a node with only points of the same class. (Again, later we'll see that this perfect division of the *training* data might not be what we want because it can lead to *overfitting*.)

Gini Impurity

At this point it'll be helpful to dive into the concept of Gini Impurity (the math is not intimidating!) The Gini Impurity of a node is the probability that a randomly chosen sample in a node would be incorrectly labeled if it was labeled by the distribution of samples in the node. For example, in the top (root) node, there is a 44.4% chance of incorrectly classifying a data point chosen at random based on the sample labels in the node. We arrive at this value using the following equation:

$$I_G(n) = 1 - \sum_{i=1}^{J} (p_i)^2$$

Gini impurity of a node n.

The Gini Impurity of a node `n` is 1 minus the sum over all the classes `J` (for a binary classification task this is 2) of the fraction of examples in each class `p_i` squared. That might be a little confusing in words, so let's work out the Gini impurity of the root node.

$$I_{root} = 1 - ((\frac{2}{6})^2 + (\frac{4}{6})^2) = 1 - \frac{5}{9} = 0.444$$

Gini Impurity of the root node

At each node, the decision tree searches through the features for the value to split on that results in the *greatest reduction* in Gini Impurity. (An alternative for splitting nodes is using the information gain, a related concept).

It then repeats this splitting process in a greedy, recursive procedure until it reaches a maximum depth, or each node contains only samples from one class. The weighted total Gini Impurity at each level of tree must decrease. At the second level of the tree, the total weighted Gini Impurity is 0.333:

$$I_{\text{second layer}} = \frac{n_{\text{left}}}{n_{\text{parent}}} * I_{\text{left node}} + \frac{n_{\text{right}}}{n_{\text{parent}}} * I_{\text{right node}} = \frac{4}{6} * 0.5 + \frac{2}{6} * 0.0 = 0.333$$

(The Gini Impurity of each node is weighted by the fraction of points from the parent node in that node.) You can continue to work out the Gini Impurity for each node (check the visual for the answers). Out of some basic math, a powerful model emerges!

Eventually, the weighted total Gini Impurity of the last layer goes to 0 meaning each node is completely pure and there is no chance that a point randomly selected from that node would be misclassified. While this may seem like a positive, it means that the model may potentially be overfitting because the nodes are constructed only using *training data*.

Overfitting: Or Why a Forest is better than One Tree

You might be tempted to ask why not just use one decision tree? It seems like the perfect classifier since it did not make any mistakes! A critical point to remember is that the tree made no mistakes **on the training data**. We expect this to be the case since we gave the tree the answers and didn't limit the max depth (number of levels). The objective of a machine learning model is to generalize well to **new data** *it has never seen before.*

Overfitting occurs when we have a very flexible model (the model has a high capacity) which essentially **memorizes** the training data by fitting it closely. The problem is that the model learns not only the actual relationships in the training data, but also any noise that is present. A flexible model is said to have high ***variance*** because the learned parameters (such as the structure of the decision tree) will vary considerably with the training data.

On the other hand, an inflexible model is said to have high ***bias*** because it makes **assumptions** about the training data (it's biased towards pre-conceived ideas of the data.) For example, a linear classifier makes the assumption that the data is linear and does not have the flexibility to fit non-linear relationships. An inflexible model may not have the capacity to fit even the training data and in both cases—high variance and high bias—the model is not able to generalize well to new data.

> The balance between creating a model that is so flexible it memorizes the training data versus an inflexible model that can't learn the training data is known as the bias-variance tradeoff and is a foundational concept in machine learning.

The reason the decision tree is prone to overfitting when we don't limit the maximum depth is because it has unlimited flexibility, meaning that it can keep growing until it has exactly one leaf node for every single observation, perfectly classifying all of them. If you go back to the image of the decision tree and limit the maximum depth to 2 (making only a single split), the classifications are no longer 100% correct. We have reduced the variance of the decision tree but at the cost of increasing the bias.

As an alternative to limiting the depth of the tree, which reduces variance (good) and increases bias (bad), we can combine many decision trees into a single ensemble model known as the random forest.

Random Forest

The random forest is a model made up of many decision trees. Rather than just simply averaging the prediction of trees (which we could call a "forest"), this model uses two key concepts that gives it the name *random*:

1. Random sampling of training data points when building trees
2. Random subsets of features considered when splitting nodes

Random sampling of training observations

When training, each tree in a random forest learns from a **random** sample of the data points. The samples are drawn with replacement, known as *bootstrapping,* which means that some samples will be used multiple times in a single tree. The idea is that by training each tree on different samples, although each tree might have high variance with respect to a particular set of the training data, overall, the entire forest will have lower variance but not at the cost of increasing the bias.

At test time, predictions are made by averaging the predictions of each decision tree. This procedure of training each individual learner on different bootstrapped subsets of the data and then averaging the predictions is known as *bagging*, short for *bootstrap aggregating*.

Random Subsets of features for splitting nodes

The other main concept in the random forest is that only a subset of all the features are considered for splitting each node in each decision tree. Generally this is set to `sqrt(n_features)` for classification meaning that if there are 16 features, at each node in each tree, only 4 random features will be considered for splitting the node. (The random forest can also be trained considering all the features at every node as is common in regression. These options can be controlled in the Scikit-Learn Random Forest implementation).

If you can comprehend a single decision tree, the idea of *bagging,* and random subsets of features, then you have a pretty good understanding of how a random forest works:

> The random forest combines hundreds or thousands of decision trees, trains each one on a slightly different set of the observations, splitting nodes in each tree considering a limited number of the features. The final predictions of the random forest are made by averaging the predictions of each individual tree.

To understand why a random forest is better than a single decision tree imagine the following scenario: you have to decide whether Tesla stock will go up and you have access to a dozen analysts who have no prior knowledge about the company. Each analyst has low bias because they don't come in with any assumptions, and is allowed to learn from a dataset of news reports.

This might seem like an ideal situation, but the problem is that the reports are likely to contain noise in addition to real signals. Because the analysts are basing their predictions entirely on the data — they have high flexibility — they can be swayed by irrelevant information. The analysts might come up with differing predictions from the same dataset. Moreover, each individual analyst has high variance and would come up with drastically different predictions if given a *different* training set of reports.

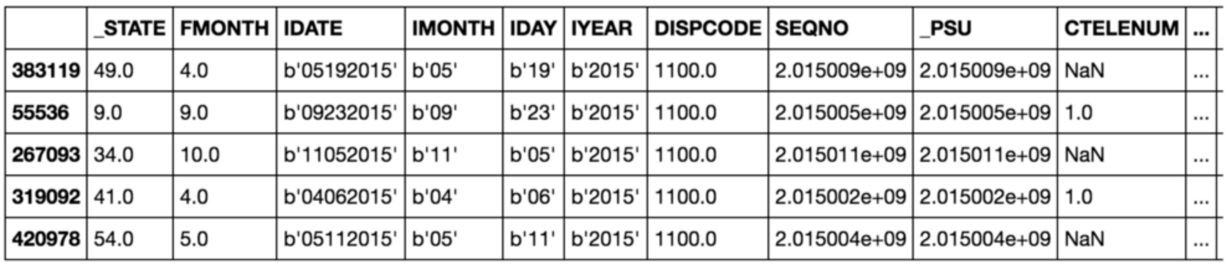
The solution is to not rely on any one individual, but pool the votes of each analyst. Furthermore, like in a random forest, allow each analyst access to only a section of the reports and hope the effects of the noisy information will be cancelled out by the sampling. In real life, we rely on multiple sources (never trust a solitary Amazon review), and therefore, not only is a decision tree intuitive, but so is the idea of combining them in a random forest.

Random Forest in Practice

Next, we'll build a random forest in Python using Scikit-Learn. Instead of learning a simple problem, we'll use a real-world dataset split into a training and testing set. We use a *test set* as an estimate of how the model will perform on new data which also lets us determine how much the model is overfitting.

Dataset

The problem we'll solve is a binary classification task with the goal of predicting an individual's health. The features are socioeconomic and lifestyle characteristics of individuals and the label is `0` for poor health and `1` for good health. This dataset was collected by the Centers for Disease Control and Prevention and is available here.



	_STATE	FMONTH	IDATE	IMONTH	IDAY	IYEAR	DISPCODE	SEQNO	_PSU	CTELENUM	...
383119	49.0	4.0	b'05192015'	b'05'	b'19'	b'2015'	1100.0	2.015009e+09	2.015009e+09	NaN	...
55536	9.0	9.0	b'09232015'	b'09'	b'23'	b'2015'	1100.0	2.015005e+09	2.015005e+09	1.0	...
267093	34.0	10.0	b'11052015'	b'11'	b'05'	b'2015'	1100.0	2.015011e+09	2.015011e+09	NaN	...
319092	41.0	4.0	b'04062015'	b'04'	b'06'	b'2015'	1100.0	2.015002e+09	2.015002e+09	1.0	...
420978	54.0	5.0	b'05112015'	b'05'	b'11'	b'2015'	1100.0	2.015004e+09	2.015004e+09	NaN	...

Sample of Data

Generally, 80% of a data science project is spent cleaning, exploring, and making features out of the data. However, for this article, we'll stick to the modeling. (For details of the other steps, look at this article).

This is an imbalanced classification problem, so accuracy is not an appropriate metric. Instead we'll measure the Receiver Operating Characteristic Area Under the Curve (ROC AUC), a measure from 0 (worst) to 1 (best) with a random guess scoring 0.5. We can also plot the ROC curve to assess a model.

The notebook contains the implementation for both the decision tree and the random forest, but here we'll just focus on the random forest. After reading in the data, we can instantiate and train a random forest as follows:

After a few minutes to train, the model is ready to make predictions on the testing data as follows:

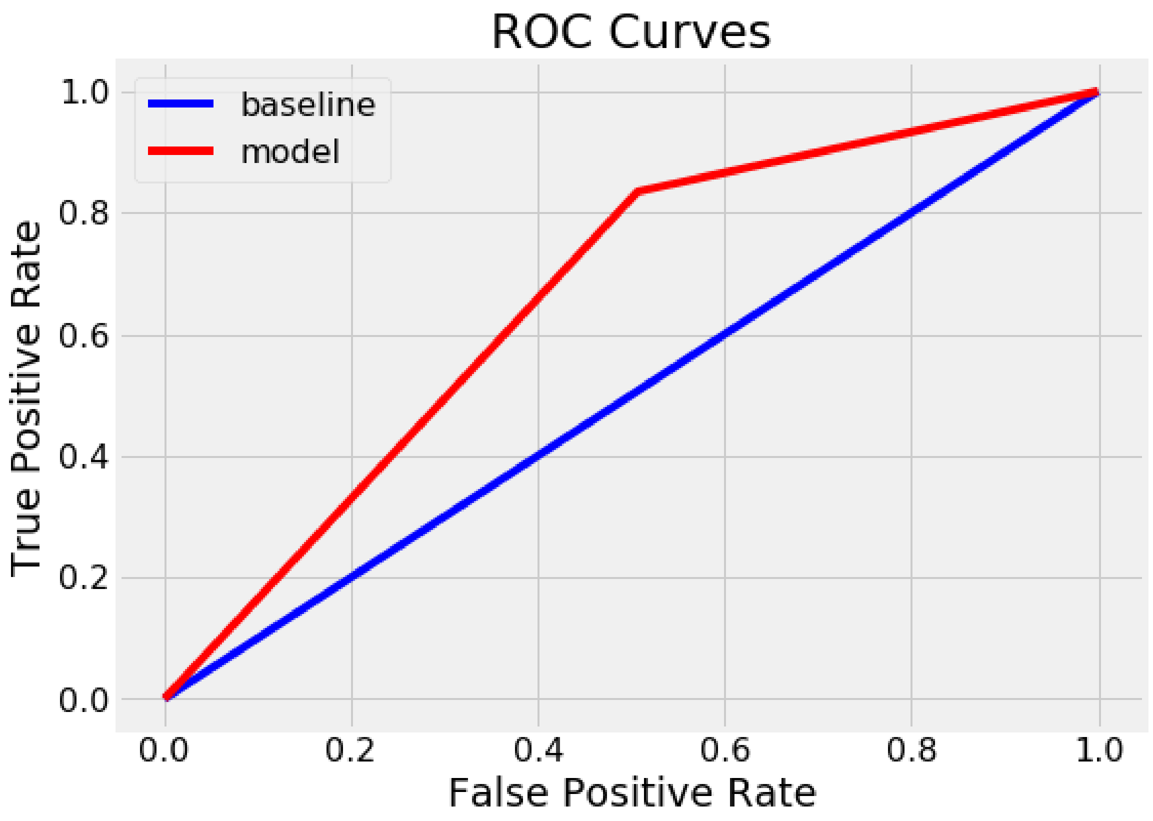
We make class predictions (`predict`) as well as predicted probabilities (`predict_proba`) to calculate the ROC AUC. Once we have the testing predictions, we can calculate the ROC AUC.

Results

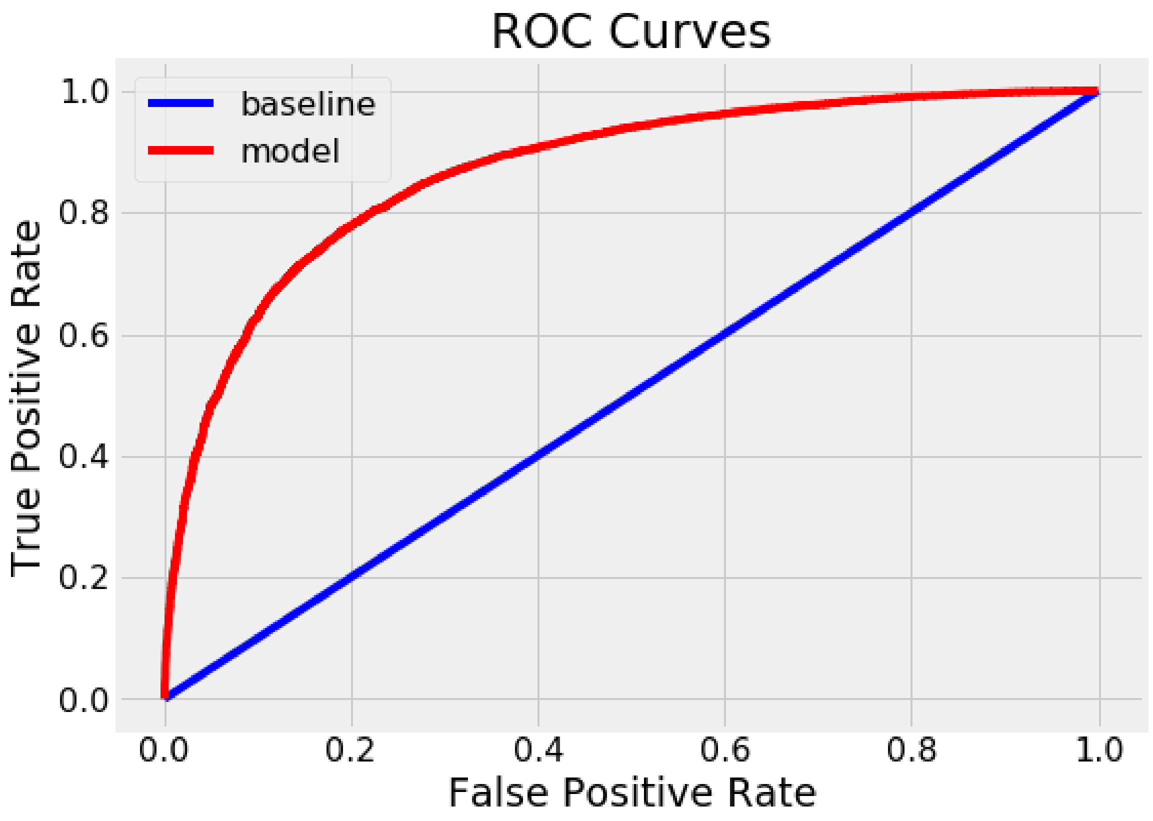
The final testing ROC AUC for the random forest was **0.87** compared to **0.67** for the single decision tree with an unlimited max depth. If we look at the training scores, both models achieved 1.0 ROC AUC, which again is as expected because we gave these models the training answers and did not limit the maximum depth of each tree.

Although the random forest overfits (doing better on the training data than on the testing data), it is able to generalize much better to the testing data than the single decision tree. The random forest has lower variance (good) while maintaining the same low bias (also good) of a decision tree.

We can also plot the ROC curve for the single decision tree (top) and the random forest (bottom). A curve to the top and left is a better model:



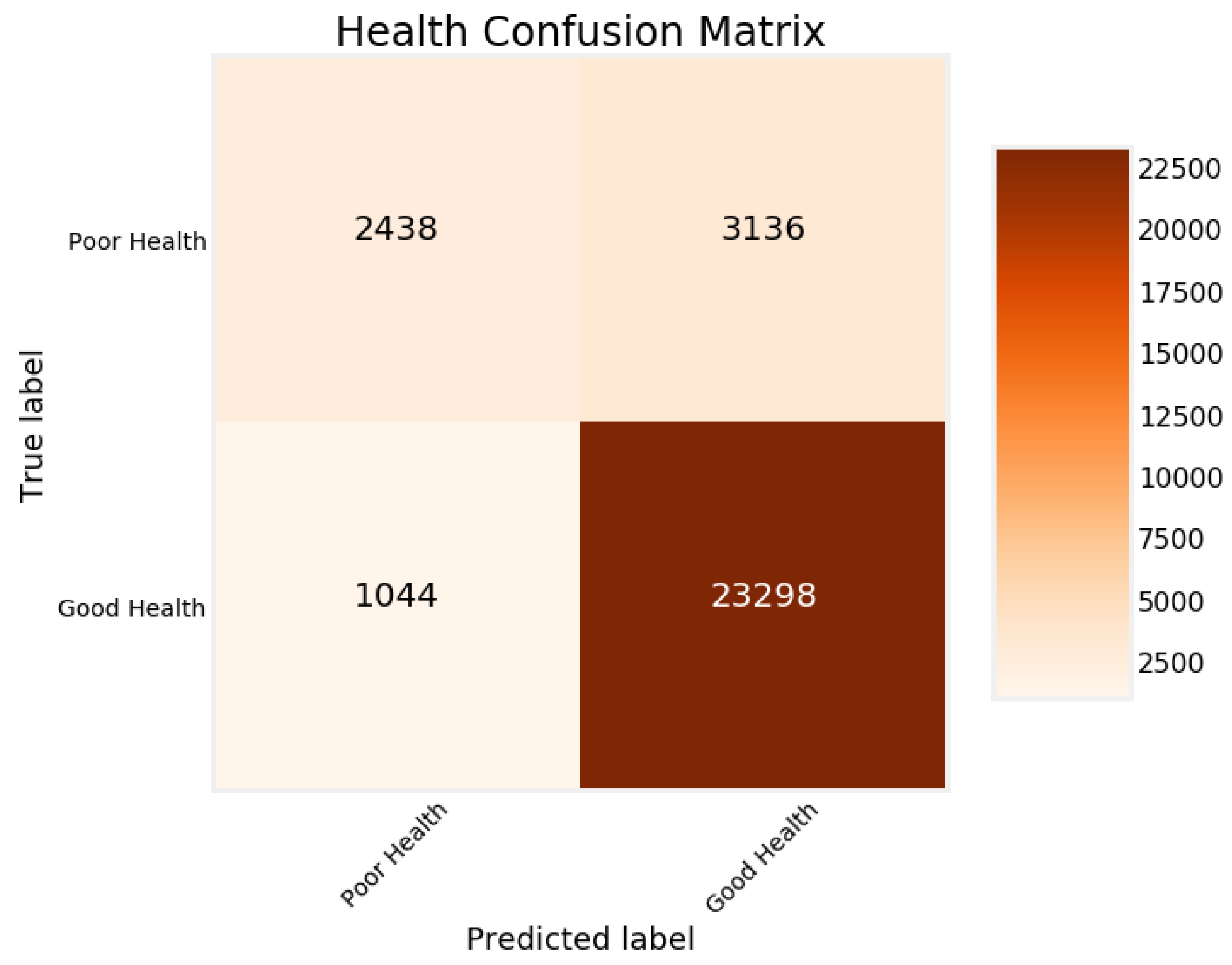
Decision Tree ROC Curve



Random Forest ROC Curve

The random forest significantly outperforms the single decision tree.

Another diagnostic measure of the model we can take is to plot the confusion matrix for the testing predictions (see the notebook for details):



This shows the predictions the model got correct in the top left and bottom right corners and the predictions missed by the model in the lower left and upper right. We can use plots such as these to diagnose our model and decide whether it's doing well enough to put into production.

Feature Importances

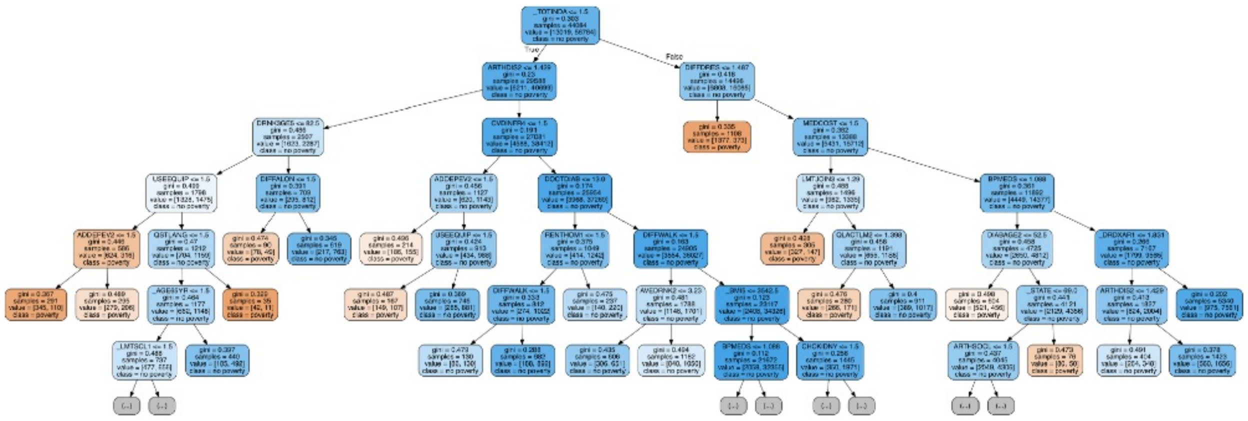
The feature importances in a random forest indicate the sum of the reduction in Gini Impurity over all the nodes that are split on that feature. We can use these to try and figure out what predictor variables the random forest considers most important. The feature importances can be extracted from a trained random forest and put into a Pandas dataframe as follows:

Feature importances can give us insight into a problem by telling us what variables are the most discerning between classes. For example, here `DIFFWALK,` indicating whether the patient has difficulty walking, is the most important feature which makes sense in the problem context.

Feature importances can be used for feature engineering by building additional features from the most important. We can also use feature importances for feature selection by removing low importance features.

Visualize Tree in Forest

Finally, we can visualize a single decision tree in the forest. This time, we have to limit the depth of the tree otherwise it will be too large to be converted into an image. To make the figure below, I limited the maximum depth to 6. This still results in a large tree that we can't completely parse! However, given our deep dive into the decision tree, we grasp how our model is working.



Single decision tree in random forest.

Next Steps

A further step is to optimize the random forest which we can do through random search using the `RandomizedSearchCV` in Scikit-Learn. Optimization refers to finding the best hyperparameters for a model on a given dataset. The best hyperparameters will vary between datasets, so we have to perform optimization (also called model tuning) separately on each datasets.

I like to think of model tuning as finding the best settings for a machine learning algorithm. Examples of what we might optimize in a random forest are the number of decision trees, the maximum depth of each decision tree, the maximum number of features considered for splitting each node, and the maximum number of data points required in a leaf node.

For an implementation of random search for model optimization of the random forest, refer to the Jupyter Notebook.

Complete Running Example

The below code is created with repl.it and presents a complete interactive running example of the random forest in Python. Feel free to run and change the code (loading the packages might take a few moments).

Complete Python example of random forest.

Conclusions

While we can build powerful machine learning models in Python without understanding anything about them, I find it's more effective to have knowledge about what is occurring behind the scenes. In this article, we not only built and used a random forest in Python, but we also developed an understanding of the model by starting with the basics.

We first looked at an individual decision tree, the building block of a random forest, and then saw how we can overcome the high variance of a single decision tree by combining hundreds of them in an ensemble model known as a random forest. The random forest uses the concepts of random sampling of observations, random sampling of features, and averaging predictions.

The key concepts to understand from this article are:

1. **Decision tree**: an intuitive model that makes decisions based on a sequence of questions asked about feature values. Has low bias and high variance leading to overfitting the training data.
2. **Gini Impurity**: a measure that the decision tree tries to minimize when splitting each node. Represents the probability that a randomly selected sample from a node will be incorrectly classified according to the distribution of samples in the node.
3. **Bootstrapping**: sampling random sets of observations with replacement.
4. **Random subsets of features**: selecting a random set of the features when considering splits for each node in a decision tree.
5. **Random Forest**: ensemble model made of many decision trees using bootstrapping, random subsets of features, and average voting to make predictions. This is an example of a bagging ensemble.
6. **Bias-variance tradeoff**: a core issue in machine learning describing the balance between a model with high flexibility (high variance) that learns the training data very well at the cost of not being able to generalize to new data , and an inflexible model (high bias) that cannot learn the training data. A random forest reduces the variance of a single decision tree leading to better predictions on new data.

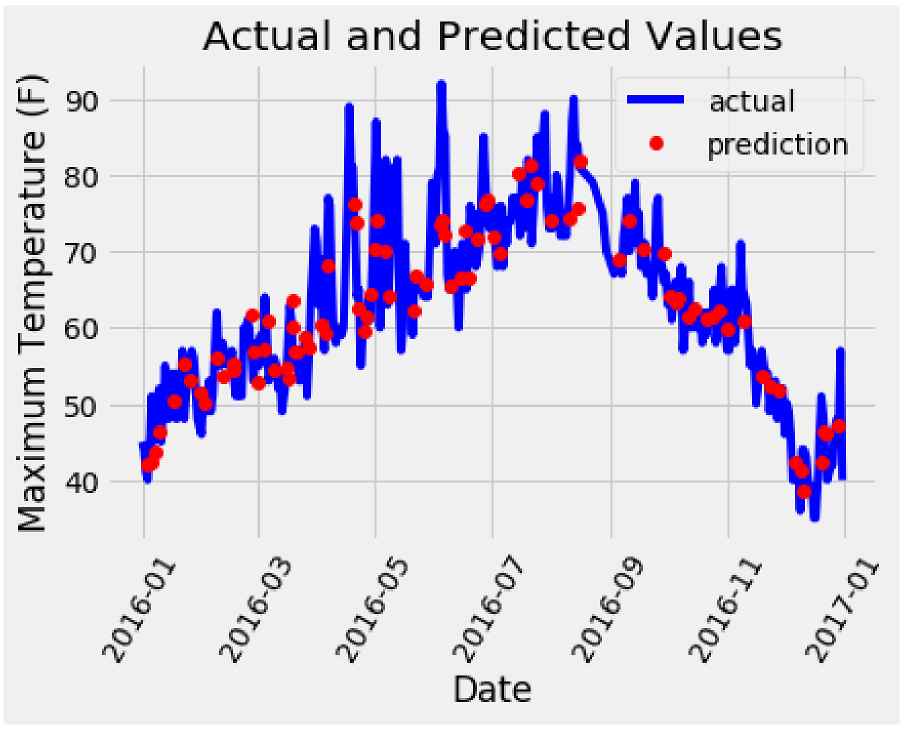
Hopefully this article has given you the confidence and understanding needed to start using the random forest on your projects. The random forest is a powerful machine learning model, but that should not prevent us from knowing how it works. The more we know about a model, the better equipped we will be to use it effectively and explain how it makes predictions.

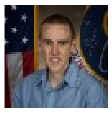
As always, I welcome comments, feedback, and constructive criticism. I can be reached on Twitter @koehrsen_will. This article was originally published on enlight, an open-source community for studying machine learning. I would like to thank enlight and also repl.it for hosting the code in the article.

Random Forest in Python

towardsdatascience.com/random-forest-in-python-24d0893d51c0

William Koehrsen December 27, 2017





William Koehrsen
Dec 27, 2017

A Practical End-to-End Machine Learning Example

There has never been a better time to get into machine learning. With the learning resources available online, free open-source tools with implementations of any algorithm imaginable, and the cheap availability of computing power through cloud services such as AWS, machine learning is truly a field that has been democratized by the internet. Anyone with access to a laptop and a willingness to learn can try out state-of-the-art algorithms in minutes. With a little more time, you can develop practical models to help in your daily life or at work (or even switch into the machine learning field and reap the economic benefits). This post will walk you through an end-to-end implementation of the powerful random forest machine learning model. It is meant to serve as a complement to my conceptual explanation of the random forest, but can be read entirely on its own as long as you have the basic idea of a decision tree and a random forest. A follow-up post details how we can improve upon the model built here.

There will of course be Python code here, however, it is not meant to intimate anyone, but rather to show how accessible machine learning is with the resources available today! The complete project with data is available on GitHub, and the data file and Jupyter Notebook can also be downloaded from Google Drive. All you need is a laptop with Python installed and the ability to start a Jupyter Notebook and you can follow along. (For installing Python and running a Jupyter notebook check out this guide). There will be a few necessary machine learning topics touched on here, but I will try to make them clear and provide resources for learning more for those interested.

Problem Introduction

The problem we will tackle is predicting the max temperature for tomorrow in our city using one year of past weather data. I am using Seattle, WA but feel free to find data for your own city using the NOAA Climate Data Online tool. We are going to act as if we don't have access to any weather forecasts (and besides, it's more fun to make our own predictions rather than rely on others). What we do have access to is one year of historical max temperatures, the temperatures for the previous two days, and an estimate from a friend who is always claiming to know everything about the weather. This is a supervised, regression machine learning problem. It's supervised because we have both the features (data for the city) and the targets (temperature) that we want to predict. During training, we give the random forest both the features and targets and it must learn how to map the data to a prediction. Moreover, this is a regression task because the target value is continuous (as opposed to discrete classes in classification). That's pretty much all the background we need, so let's start!

Roadmap

Before we jump right into programming, we should lay out a brief guide to keep us on track. The following steps form the basis for any machine learning workflow once we have a problem and model in mind:

1. State the question and determine required data
2. Acquire the data in an accessible format
3. Identify and correct missing data points/anomalies as required

4. Prepare the data for the machine learning model
5. Establish a baseline model that you aim to exceed
6. Train the model on the training data
7. Make predictions on the test data
8. Compare predictions to the known test set targets and calculate performance metrics
9. If performance is not satisfactory, adjust the model, acquire more data, or try a different modeling technique
10. Interpret model and report results visually and numerically

Step 1 is already checked off! We have our question: “can we predict the max temperature tomorrow for our city?” and we know we have access to historical max temperatures for the past year in Seattle, WA.

Data Acquisition

First, we need some data. To use a realistic example, I retrieved weather data for Seattle, WA from 2016 using the NOAA Climate Data Online tool. Generally, about 80% of the time spent in data analysis is cleaning and retrieving data, but this workload can be reduced by finding high-quality data sources. The NOAA tool is surprisingly easy to use and temperature data can be downloaded as clean csv files which can be parsed in languages such as Python or R. The complete data file is available for download for those wanting to follow along.

The following Python code loads in the csv data and displays the structure of the data:

```
# Pandas is used for data manipulation
import pandas as pd

# Read in data and display first 5 rows
features = pd.read_csv('temps.csv')
features.head(5)
```

	year	month	day	week	temp_2	temp_1	average	actual	friend
0	2016	1	1	Fri	45	45	45.6	45	29
1	2016	1	2	Sat	44	45	45.7	44	61
2	2016	1	3	Sun	45	44	45.8	41	56
3	2016	1	4	Mon	44	41	45.9	40	53
4	2016	1	5	Tues	41	40	46.0	44	41

The information is in the tidy data format with each row forming one observation, with the variable values in the columns.

Following are explanations of the columns:

year: 2016 for all data points

month: number for month of the year

day: number for day of the year

week: day of the week as a character string

temp_2: max temperature 2 days prior

temp_1: max temperature 1 day prior

average: historical average max temperature

actual: max temperature measurement

friend: your friend's prediction, a random number between 20 below the average and 20 above the average

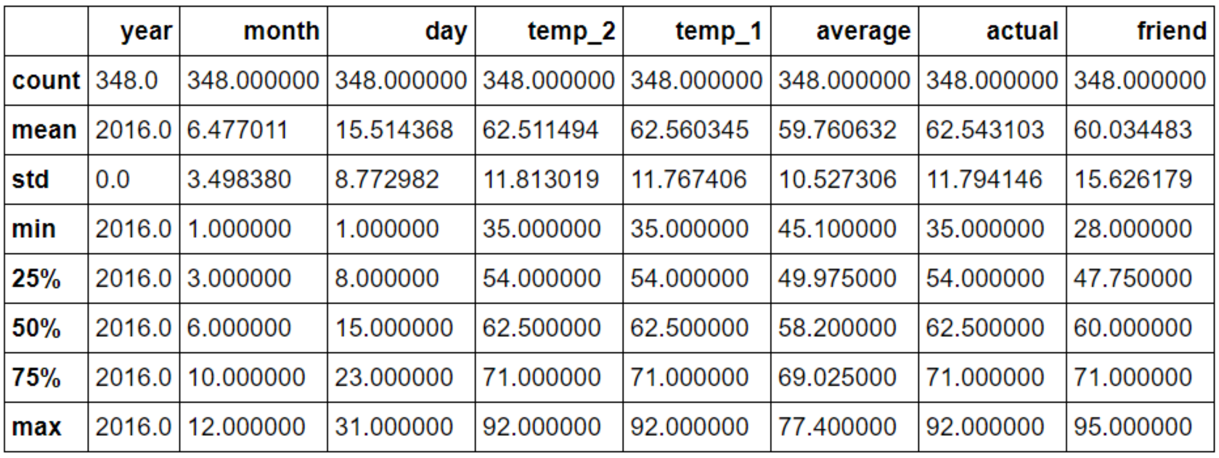
Identify Anomalies/ Missing Data

If we look at the dimensions of the data, we notice only there are only 348 rows, which doesn't quite agree with the 366 days we know there were in 2016. Looking through the data from the NOAA, I noticed several missing days, which is a great reminder that data collected in the real-world will never be perfect. Missing data can impact an analysis as can incorrect data or outliers. In this case, the missing data will not have a large effect, and the data quality is good because of the source. We also can see there are nine columns which represent eight features and the one target ('actual').

```
print('The shape of our features is:', features.shape)
The shape of our features is: (348, 9)
```

To identify anomalies, we can quickly compute summary statistics.

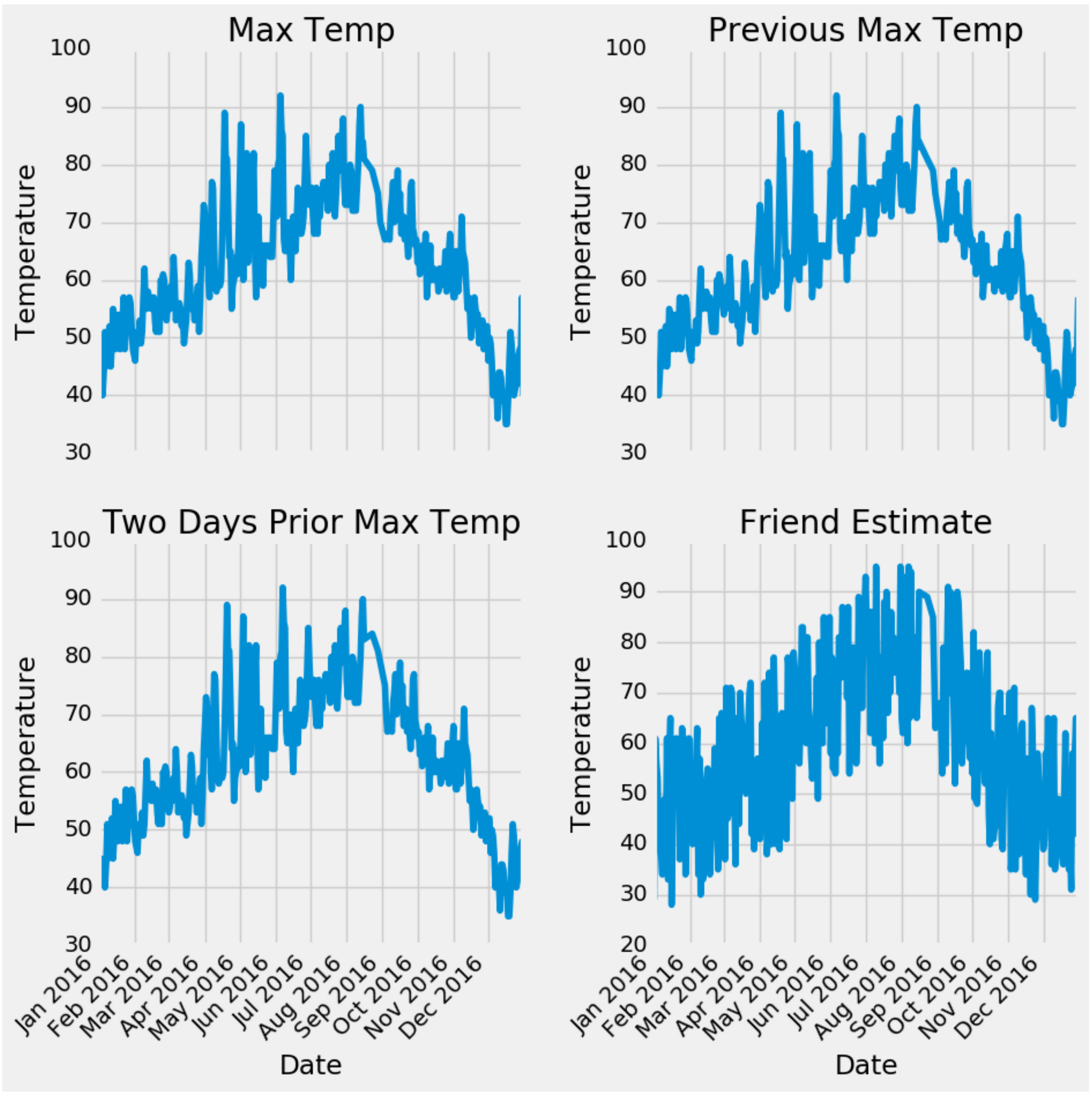
```
# Descriptive statistics for each column
features.describe()
```



	year	month	day	temp_2	temp_1	average	actual	friend
count	348.0	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000	348.000000
mean	2016.0	6.477011	15.514368	62.511494	62.560345	59.760632	62.543103	60.034483
std	0.0	3.498380	8.772982	11.813019	11.767406	10.527306	11.794146	15.626179
min	2016.0	1.000000	1.000000	35.000000	35.000000	45.100000	35.000000	28.000000
25%	2016.0	3.000000	8.000000	54.000000	54.000000	49.975000	54.000000	47.750000
50%	2016.0	6.000000	15.000000	62.500000	62.500000	58.200000	62.500000	60.000000
75%	2016.0	10.000000	23.000000	71.000000	71.000000	69.025000	71.000000	71.000000
max	2016.0	12.000000	31.000000	92.000000	92.000000	77.400000	92.000000	95.000000

Data Summary

There are not any data points that immediately appear as anomalous and no zeros in any of the measurement columns. Another method to verify the quality of the data is make basic plots. Often it is easier to spot anomalies in a graph than in numbers. I have left out the actual code here, because plotting is Python is non-intuitive but feel free to refer to the notebook for the complete implementation (like any good data scientist, I pretty much copy and pasted the plotting code from Stack Overflow).



Examining the quantitative statistics and the graphs, we can feel confident in the high quality of our data. There are no clear outliers, and although there are a few missing points, they will not detract from the analysis.

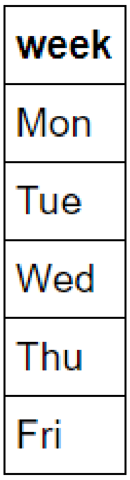
Data Preparation

Unfortunately, we aren't quite at the point where you can just feed raw data into a model and have it return an answer (although people are working on this)! We will need to do some minor modification to put our data into machine-understandable terms. We will use the Python library Pandas for our data manipulation relying, on the structure known as a dataframe, which is basically an excel spreadsheet with rows and columns.

The exact steps for preparation of the data will depend on the model used and the data gathered, but some amount of data manipulation will be required for any machine learning application.

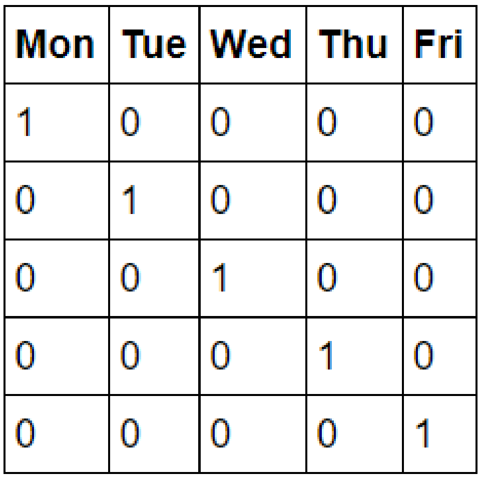
One-Hot Encoding

The first step for us is known as one-hot encoding of the data. This process takes categorical variables, such as days of the week and converts it to a numerical representation without an arbitrary ordering. Days of the week are intuitive to us because we use them all the time. You will (hopefully) never find anyone who doesn't know that 'Mon' refers to the first day of the workweek, but machines do not have any intuitive knowledge. What computers know is numbers and for machine learning we must accommodate them. We could simply map days of the week to numbers 1–7, but this might lead to the algorithm placing more importance on Sunday because it has a higher numerical value. Instead, we change the single column of weekdays into seven columns of binary data. This is best illustrated pictorially. One hot encoding takes this:



week
Mon
Tue
Wed
Thu
Fri

and turns it into

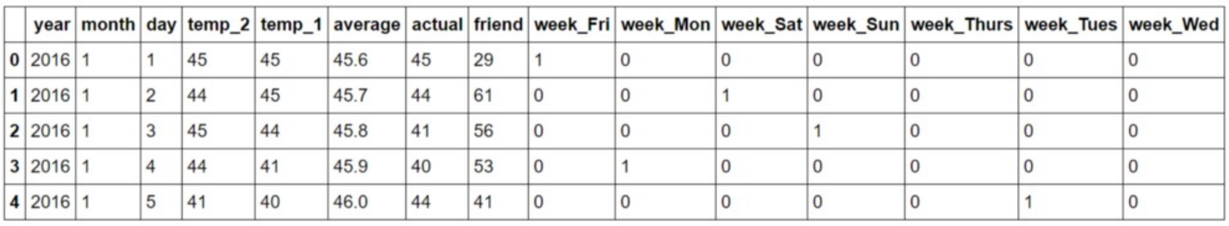


Mon	Tue	Wed	Thu	Fri
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

So, if a data point is a Wednesday, it will have a 1 in the Wednesday column and a 0 in all other columns. This process can be done in pandas in a single line!

```
# 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)
```

Snapshot of data after one-hot encoding:



	year	month	day	temp_2	temp_1	average	actual	friend	week_Fri	week_Mon	week_Sat	week_Sun	week_Thurs	week_Tues	week_Wed
0	2016	1	1	45	45	45.6	45	29	1	0	0	0	0	0	0
1	2016	1	2	44	45	45.7	44	61	0	0	1	0	0	0	0
2	2016	1	3	45	44	45.8	41	56	0	0	0	1	0	0	0
3	2016	1	4	44	41	45.9	40	53	0	1	0	0	0	0	0
4	2016	1	5	41	40	46.0	44	41	0	0	0	0	0	1	0

Data after One-Hot Encoding

The shape of our data is now 349 x 15 and all of the column are numbers, just how the algorithm likes it!

Features and Targets and Convert Data to Arrays

Now, we need to separate the data into the features and targets. The target, also known as the label, is the value we want to predict, in this case the actual max temperature and the features are all the columns the model uses to make a prediction. We will also convert the Pandas dataframes to Numpy arrays because that is the way the algorithm works. (I save the column headers, which are the names of the features, to a list to use for later visualization).

```
# 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)
```

Training and Testing Sets

There is one final step of data preparation: splitting data into training and testing sets. During training, we let the model 'see' the answers, in this case the actual temperature, so it can learn how to predict the temperature from the features. We expect there to be some relationship between all the features and the target value, and the model's job is to learn this relationship during training. Then, when it comes time to evaluate the model, we ask it to make predictions on a testing set where it only has access to the features (not the answers)! Because we do have the actual answers for the test set, we can compare these predictions to the true value to judge how accurate the model is. Generally, when training a model, we randomly split the data into training and testing sets to get a representation of all data points (if we trained on the first nine months of the year and then used the final three months for prediction, our algorithm would not perform well because it has not seen any data from those last three months.) I am setting the random state to 42 which means the results will be the same each time I run the split for reproducible results.

The following code splits the data sets with another single line:

```
# Using Skicit-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, labels, test_size = 0.25, random_state = 42)
```

We can look at the shape of all the data to make sure we did everything correctly. We expect the training features number of columns to match the testing feature number of columns and the number of rows to match for the respective training and testing features and the labels :

```
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, 14)
Training Labels Shape: (261,)
Testing Features Shape: (87, 14)
Testing Labels Shape: (87,)
```

It looks as if everything is in order! Just to recap, to get the data into a form acceptable for machine learning we:

1. One-hot encoded categorical variables
2. Split data into features and labels
3. Converted to arrays
4. Split data into training and testing sets

Depending on the initial data set, there may be extra work involved such as removing outliers, imputing missing values, or converting temporal variables into cyclical representations. These steps may seem arbitrary at first, but once you get the basic workflow, it will be generally the same for any machine learning problem. It's all about taking human-readable data and putting it into a form that can be understood by a machine learning model.

Establish Baseline

Before we can make and evaluate predictions, we need to establish a baseline, a sensible measure that we hope to beat with our model. If our model cannot improve upon the baseline, then it will be a failure and we should try a different model or admit that machine learning is not right for our problem. The baseline prediction for our case can be the historical max temperature averages. In other words, our baseline is the error we would get if we simply predicted the average max temperature for all days.

```
# 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 degrees.
```

We now have our goal! If we can't beat an average error of 5 degrees, then we need to rethink our approach.

Train Model

After all the work of data preparation, creating and training the model is pretty simple using Scikit-learn. We import the random forest regression model from skicit-learn, instantiate the model, and fit (scikit-learn's name for training) the model on the training data. (Again setting the random state for reproducible results). This entire process is only 3 lines in scikit-learn!

```
# 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);
```

Make Predictions on the Test Set

Our model has now been trained to learn the relationships between the features and the targets. The next step is figuring out how good the model is! To do this we make predictions on the test features (the model is never allowed to see the test answers). We then compare the predictions to the known answers. When performing regression, we need to make sure to use the absolute error because we expect some of our answers to be low and some to be high. We are interested in how far away our average prediction is from the actual value so we take the absolute value (as we also did when establishing the baseline).

Making predictions with out model is another 1-line command in Skicit-learn.

```
# 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: 3.83 degrees.
```

Our average estimate is off by 3.83 degrees. That is more than a 1 degree average improvement over the baseline. Although this might not seem significant, it is nearly 25% better than the baseline, which, depending on the field and the problem, could represent millions of dollars to a company.

Determine Performance Metrics

To put our predictions in perspective, we can calculate an accuracy using the mean average percentage error subtracted from 100 %.

```
# 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: 93.99 %.
```

That looks pretty good! Our model has learned how to predict the maximum temperature for the next day in Seattle with 94% accuracy.

Improve Model if Necessary

In the usual machine learning workflow, this would be when start hyperparameter tuning. This is a complicated phrase that means "adjust the settings to improve performance" (The settings are known as hyperparameters to distinguish them from model parameters learned during training). The most common way to do this is simply make a bunch of models with different settings, evaluate them all on the same validation set, and see which one does best. Of course, this would be a tedious process to do by hand, and there are automated methods to do this process in Skicit-learn. Hyperparameter tuning is often more engineering than theory-based, and I would encourage anyone interested to check out the documentation and start playing around! An accuracy of 94% is satisfactory for this problem, but keep in mind that the first model built will almost never be the model that makes it to production.

Interpret Model and Report Results

At this point, we know our model is good, but it's pretty much a black box. We feed in some Numpy arrays for training, ask it to make a prediction, evaluate the predictions, and see that they are reasonable. The question is: how does this model arrive at the values? There are two approaches to get under the hood of the random forest: first, we can look at a single tree in the forest, and second, we can look at the feature importances of our explanatory variables.

Visualizing a Single Decision Tree

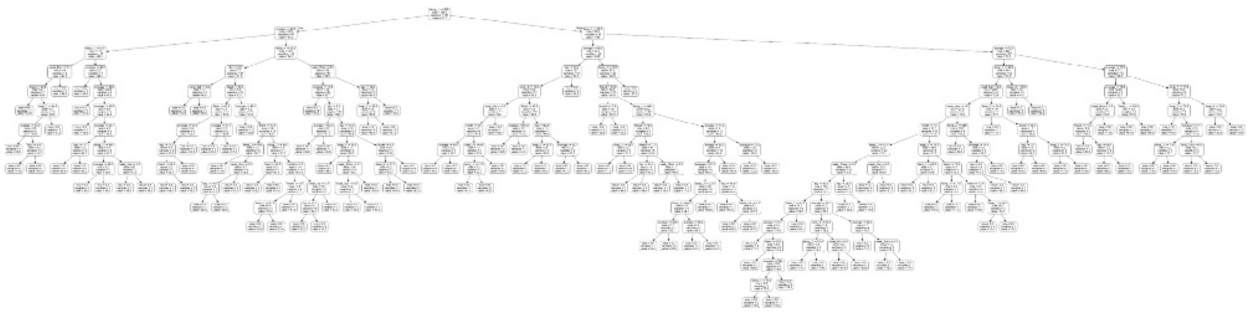
One of the coolest parts of the Random Forest implementation in Skicit-learn is we can actually examine any of the trees in the forest. We will select one tree, and save the whole tree as an image.

The following code takes one tree from the forest and saves it as an image.

```
# 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 = 'tree.dot', feature_names = feature_list, rounded =
True, precision = 1)
```

```
# Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('tree.dot')

# Write graph to a png file
graph.write_png('tree.png')
```

Let’s take a look:

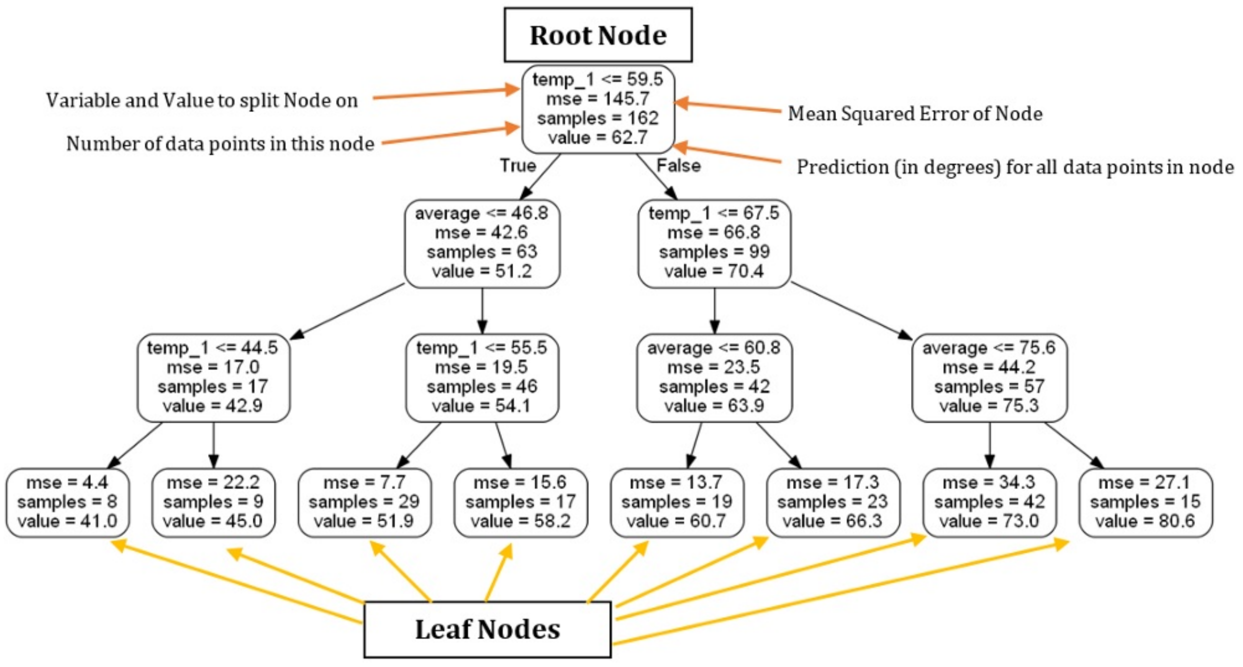


Single Full Decision Tree in Forest

Wow! That looks like quite an expansive tree with 15 layers (in reality this is quite a small tree compared to some I’ve seen). You can download this image yourself and examine it in greater detail, but to make things easier, I will limit the depth of trees in the forest to produce an understandable image.

```
# 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, rounded = True, precision = 1)

(graph, ) = pydot.graph_from_dot_file('small_tree.dot')

graph.write_png('small_tree.png');
```

Here is the reduced size tree annotated with labels



Based solely on this tree, we can make a prediction for any new data point. Let's take an example of making a prediction for Wednesday, December 27, 2017. The (actual) variables are: temp_2 = 39, temp_1 = 35, average = 44, and friend = 30. We start at the root node and the first answer is True because temp_1 ≤ 59.5. We move to the left and encounter the second question, which is also True as average ≤ 46.8. Move down to the left and on to the third and final question which is True as well because temp_1 ≤ 44.5. Therefore, we conclude that our estimate for the maximum temperature is 41.0 degrees as indicated by the value in the leaf node. An interesting observation is that in the root node, there are only 162 samples despite there being 261 training data points. This is because each tree in the forest is trained on a random subset of the data points with replacement (called bagging, short for bootstrap aggregating). (We can turn off the sampling with replacement and use all the data points by setting bootstrap = False when making the forest). Random sampling of data points, combined with random sampling of a subset of the features at each node of the tree, is why the model is called a 'random' forest.

Furthermore, notice that in our tree, there are only 2 variables we actually used to make a prediction! According to this particular decision tree, the rest of the features are not important for making a prediction. Month of the year, day of the month, and our friend's prediction are utterly useless for predicting the maximum temperature tomorrow! The only important information according to our simple tree is the temperature 1 day prior and the historical average. Visualizing the tree has increased our domain knowledge of the problem, and we now know what data to look for if we are asked to make a prediction!

Variable Importances

In order to quantify the usefulness of all the variables in the entire random forest, we can look at the relative importances of the variables. The importances returned in Skicit-learn represent how much including a particular variable improves the prediction. The actual calculation of the importance is beyond the scope of this post, but we can use the numbers to make relative comparisons between variables.

The code here takes advantage of a number of tricks in the Python language, namely list comprehensive, zip, sorting, and argument unpacking. It's not that important to understand these at the moment, but if you want to become skilled at Python, these are tools you should have in your arsenal!

```
# 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(feature_list, importances)]

# 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: 0.7
Variable: average              Importance: 0.19
Variable: day                  Importance: 0.03
Variable: temp_2               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
```

At the top of the list is temp_1, the max temperature of the day before. This tells us the best predictor of the max temperature for a day is the max temperature of the day before, a rather intuitive finding. The second most important factor is the historical average max temperature, also not that surprising. Your friend turns out to not be very helpful, along with the day of the week, the year, the month, and the temperature 2 days prior. These importances all make sense as we would not expect the day of the week to be a predictor of maximum temperature as it has nothing to do with weather. Moreover, the year is the same for all data points and hence provides us with no information for predicting the max temperature.

In future implementations of the model, we can remove those variables that have no importance and the performance will not suffer. Additionally, if we are using a different model, say a support vector machine, we could use the random forest feature importances as a kind of feature selection method. Let's quickly make a random forest with only the two most important variables, the max temperature 1 day prior and the historical average and see how the performance compares.

```
# 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: 3.9 degrees.
Accuracy: 93.8 %.
```

This tells us that we actually do not need all the data we collected to make accurate predictions! If we were to continue using this model, we could only collect the two variables and achieve nearly the same performance. In a production setting, we would need to weigh the decrease in accuracy versus the extra time required to obtain more information. Knowing how to find the right balance between performance and cost is an essential skill for a machine learning engineer and will ultimately depend on the problem!

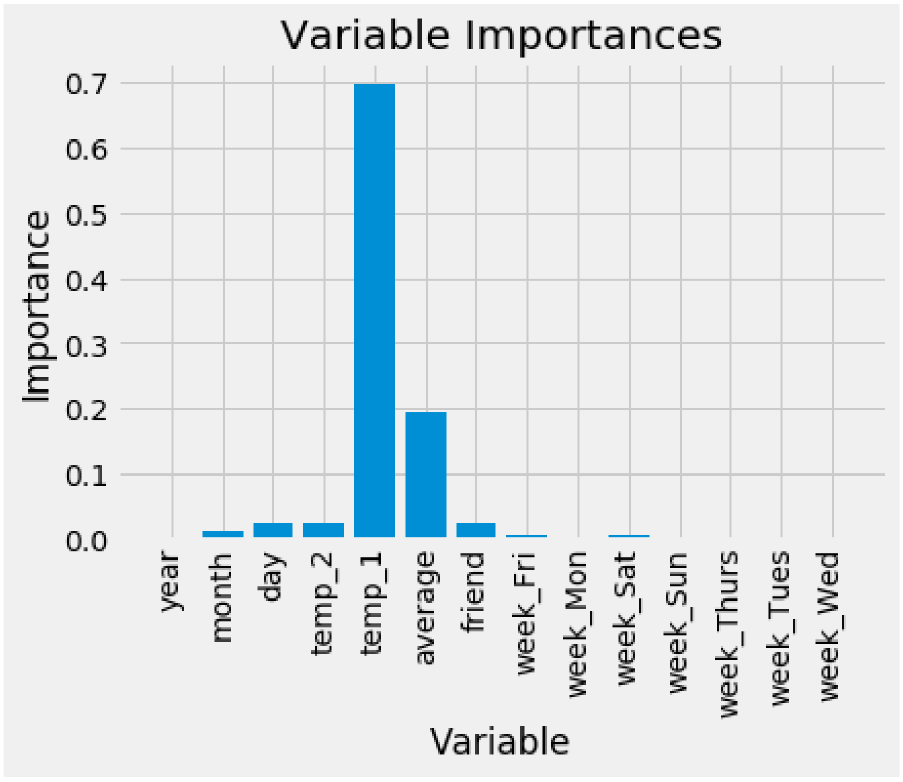
At this point we have covered pretty much everything there is to know for a basic implementation of the random forest for a supervised regression problem. We can feel confident that our model can predict the maximum temperature tomorrow with 94% accuracy from one year of historical data. From here, feel free to play around with this example, or use the model on a data set of your choice. I will wrap up this post by making a few visualizations. My two favorite parts of data science are graphing and modeling, so naturally I have to make some charts! In addition to being enjoyable to look at, charts can help us diagnose our model because they compress a lot of numbers into an image that we can quickly examine.

Visualizations

The first chart I'll make is a simple bar plot of the feature importances to illustrate the disparities in the relative significance of the variables. Plotting in Python is kind of non-intuitive, and I end up looking up almost everything on Stack Overflow when I make graphs. Don't worry if the code here doesn't quite make sense, sometimes fully understanding the code isn't necessary to get the end result you want!

```
# 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');
```

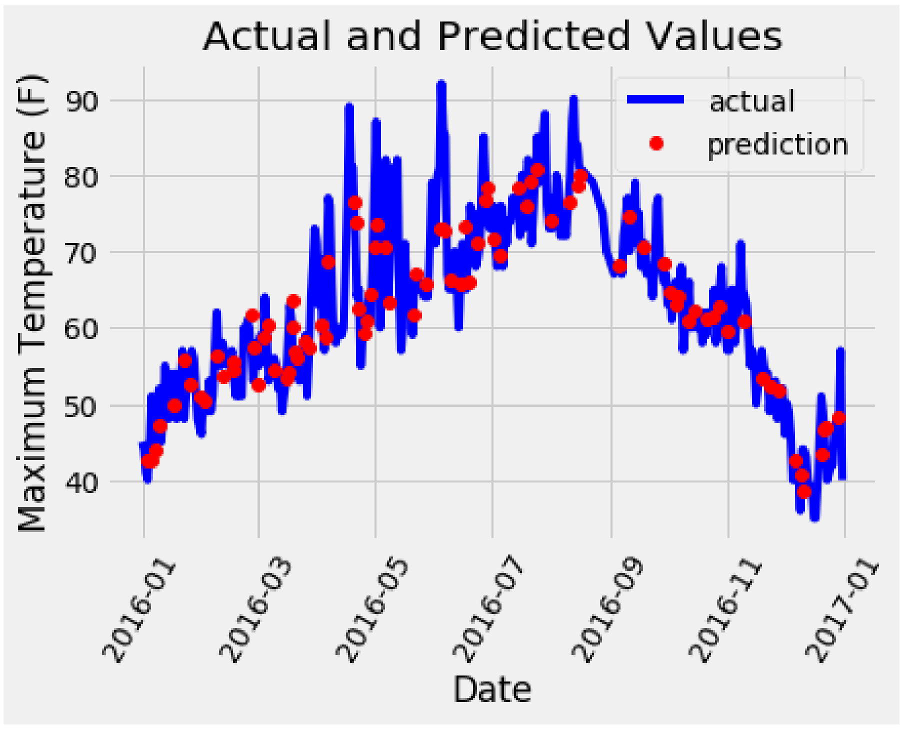


Variable Importances

Next, we can plot the entire dataset with predictions highlighted. This requires a little data manipulation, but its not too difficult. We can use this plot to determine if there are any outliers in either the data or our predictions.

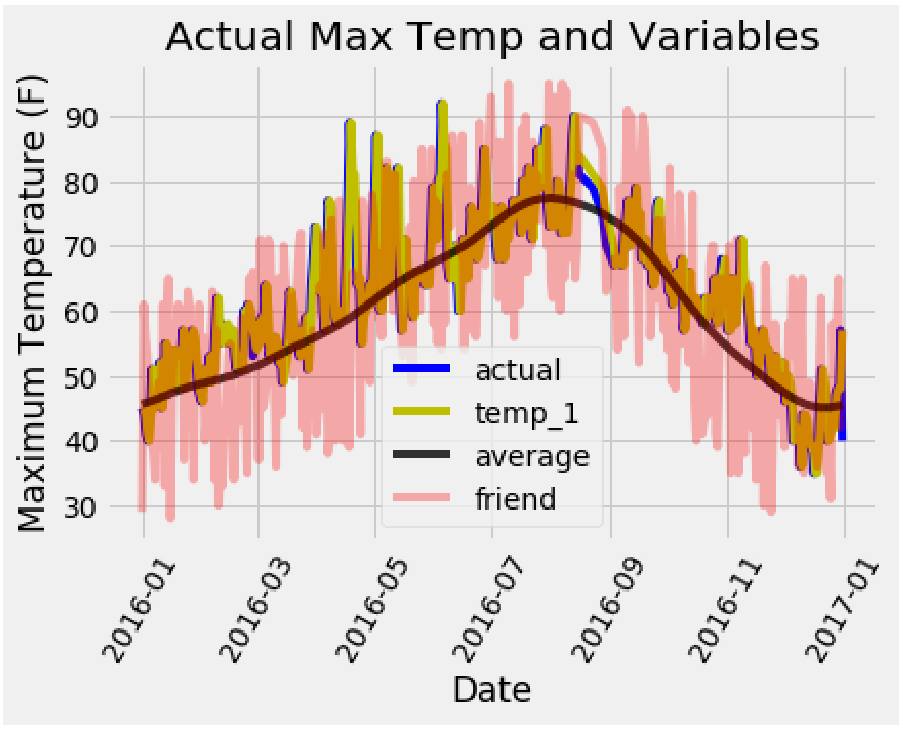
```
# Use datetime for creating date objects for plotting
import datetime

# 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, day in zip(years, months, days)]
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,
month, day in zip(years, months, days)]

# 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 =
'prediction')
plt.xticks(rotation = '60');
plt.legend()

# Graph labels
plt.xlabel('Date'); plt.ylabel('Maximum Temperature (F)'); plt.title('Actual and
Predicted Values');
```



A little bit of work for a nice looking graph! It doesn't look as if we have any noticeable outliers that need to be corrected. To further diagnose the model, we can plot residuals (the errors) to see if our model has a tendency to over-predict or under-predict, and we can also see if the residuals are normally distributed. However, I will just make one final chart showing the actual values, the temperature one day previous, the historical average, and our friend's prediction. This will allow us to see the difference between useful variables and those that aren't so helpful.

```
# 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 and Variables');
```



Actual Values and Variables

It is a little hard to make out all the lines, but we can see why the max temperature one day prior and the historical max temperature are useful for predicting max temperature while our friend is not (don't give up on the friend yet, but maybe also don't place so much weight on their estimate!). Graphs such as this are often helpful to make ahead of time so we can choose the variables to include, but they also can be used for diagnosis. Much as in the case of Anscombe's quartet, graphs are often more revealing than quantitative numbers and should be a part of any machine learning workflow.

Conclusions

With those graphs, we have completed an entire end-to-end machine learning example! At this point, if we want to improve our model, we could try different hyperparameters (settings) try a different algorithm, or the best approach of all, gather more data! The performance of any model is directly proportional to the amount of valid data it can learn from, and we were using a very limited amount of information for training. I would encourage anyone to try and improve this model and share the results. From here you can dig more into the random forest theory and application using numerous online (free) resources. For those looking for a single book to cover both theory and Python implementations of machine learning models, I highly recommend Hands-On Machine Learning with Scikit-Learn and Tensorflow. Moreover, I hope everyone who made it through has seen how accessible machine learning has become and is ready to join the welcoming and helpful machine learning community.

As always, I welcome feedback and constructive criticism! My email is wjk68@case.edu.

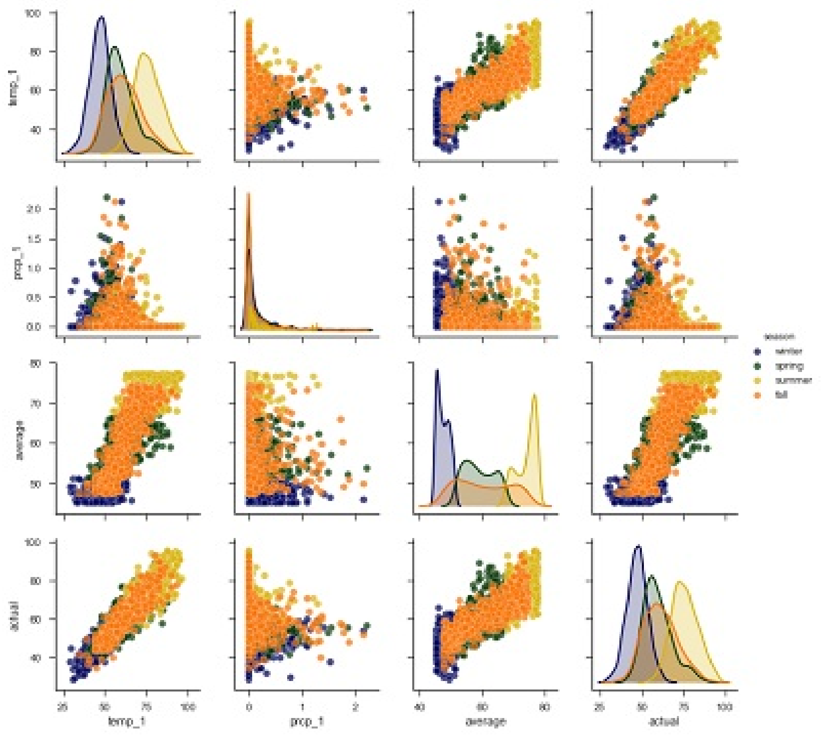
Improving the Random Forest in Python Part 1

towardsdatascience.com/improving-random-forest-in-python-part-1-893916666cd

William Koehrsen January 7, 2018

Gathering More Data and Feature Engineering

In a previous post we went through an end-to-end implementation of a simple random forest in Python for a supervised regression problem. Although we covered every step of the machine learning process, we only briefly touched on one of the most critical parts: improving our initial machine learning model. The model we finished with achieved decent performance and beat the baseline, but we should be able to better the model with a couple different approaches. This article is the first of two that will explore how to improve our random forest machine learning model using Python and the Scikit-Learn library. I would recommend checking out the introductory post before continuing, but the concepts covered here can also stand on their own.



How to Improve a Machine Learning Model

There are three general approaches for improving an existing machine learning model:

1. Use more (high-quality) data and feature engineering
2. Tune the hyperparameters of the algorithm
3. Try different algorithms

These are presented in the order in which I usually try them. Often, the immediate solution proposed to improve a poor model is to use a more complex model, often a deep neural network. However, I have found that approach inevitably leads to frustration. A complex model is built over many hours, which then also fails to deliver, leading to another model and so on. Instead, my first question is always: "Can we get more data relevant to the problem?". As Geoff Hinton (the father of deep neural networks) has pointed out in an article titled 'The Unreasonable Effectiveness of Data", the amount of useful data is more important to the problem than the complexity of the model. Others have echoed the idea that a simple model and plenty of data will beat a complex model with limited data. If there is more information that can help with our problem that we are not using, the best payback in terms of time invested versus performance gained is to get that data.

This post will cover the first method for improving ML models, and the second approach will appear in a subsequent article. I will also write end-to-end implementations of several algorithms which may or may not beat the random forest (If there are any requests for specific algorithms, let me know in the comments)! All of the code and data for this example can be found on the project GitHub page. I have included plenty of code in this post, not to discourage anyone unfamiliar with Python, but to show how accessible machine learning has become and to encourage anyone to start implementing these useful models!

Problem Recap

As a brief reminder, we are dealing with a temperature prediction problem: given historical data, we want to predict the maximum temperature tomorrow in our city. I am using Seattle, WA, but feel free to use the NOAA Climate Data Online Tool to get info for your city. This task is a supervised, regression machine learning problem because we have the labels (targets) we want to predict, and those labels are continuous values (in contrast to unsupervised learning where we do not have labels, or classification, where we are predicting discrete classes). Our original data used in the simple model was a single year of max temperature measurements from 2016 as well as the historical average max temperature. This was supplemented by the predictions of our “meteorologically-inclined” friend, calculated by randomly adding or subtracting 20 degrees from the historical average.

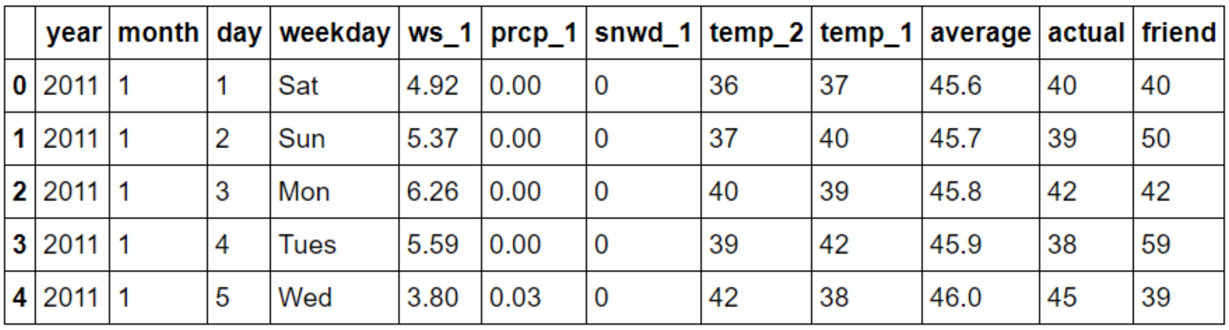
Our final performance using the original data was an average error of 3.83 degrees compared to a baseline error of 5.03 degrees. This represented a final accuracy of 93.99%.

Getting More Data

In the first article, we used one year of historical data from 2016. Thanks to the NOAA (National Atmospheric and Oceanic Administration), we can get data going back to 1891. For now, let’s restrict ourselves to six years (2011–2016), but feel free to use additional data to see if it helps. In addition to simply getting more years of data, we can also include more features. This means we can use additional weather variables that we believe will be useful for predicting the max temperature. We can use our domain knowledge (or advice from the experts), along with correlations between the variable and our target to determine which features will be helpful. From the plethora of options offered by the NOAA (seriously, I have to applaud the work of this organization and their open data policy), I added average wind speed, precipitation, and snow depth on the ground to our list of variables. Remember, because we are predicting the maximum temperature for tomorrow, we can’t actually use the measurement on that day. We have to shift it one day into the past, meaning we are using today’s precipitation total to predict the max temperature tomorrow. This prevents us from ‘cheating’ by having information from the future today.

The additional data was in relatively good shape straight from the source, but I did have to do some slight modifications before reading it into Python. I have left out the “data munging” details to focus on the Random Forest implementation, but I will release a separate post showing how to clean the data. I use the R statistical language for munging because I like how it makes data manipulation interactive, but that’s a discussion for another article. For now, we can load in the data and examine the first few rows.

```
# Pandas is used for data manipulation
import pandas as pd

# Read in data as a dataframe
features = pd.read_csv('data/temps_extended.csv')
features.head(5)
```



	year	month	day	weekday	ws_1	prcp_1	snwd_1	temp_2	temp_1	average	actual	friend
0	2011	1	1	Sat	4.92	0.00	0	36	37	45.6	40	40
1	2011	1	2	Sun	5.37	0.00	0	37	40	45.7	39	50
2	2011	1	3	Mon	6.26	0.00	0	40	39	45.8	42	42
3	2011	1	4	Tues	5.59	0.00	0	39	42	45.9	38	59
4	2011	1	5	Wed	3.80	0.03	0	42	38	46.0	45	39

Expanded Data Subset

The new variables are:

ws_1: average wind speed from the day before (mph)

prcp_1: precipitation from the day before (in)

snwd_1: snow depth on the ground from the day before (in)

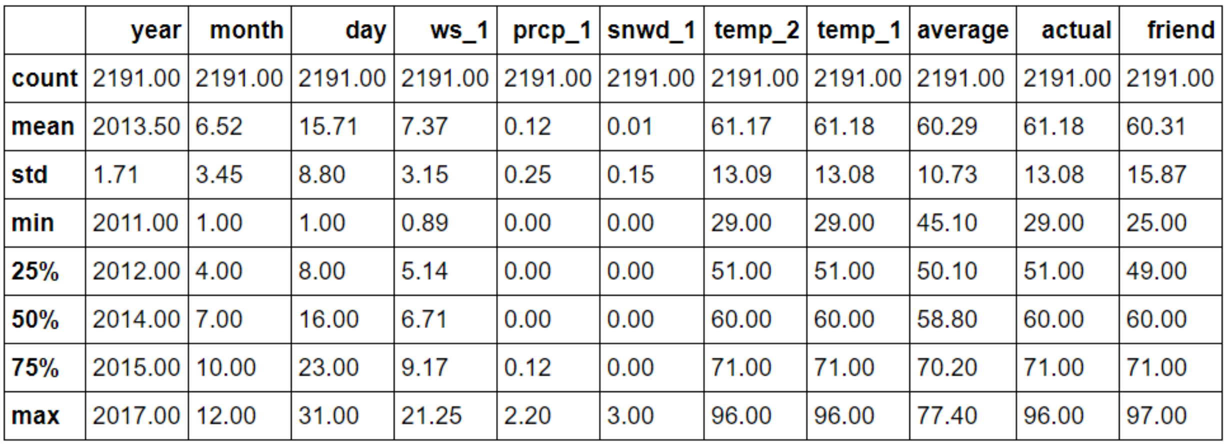
Before we had 348 days of data. Let's look at the size now.

```
print('We have {} days of data with {} variables'.format(*features.shape))
```

```
We have 2191 days of data with 12 variables.
```

There are now over 2000 days of historical temperature data (about 6 years). We should summarize the data to make sure there are no anomalies that jump out in the numbers.

```
round(features.describe, 2)
```

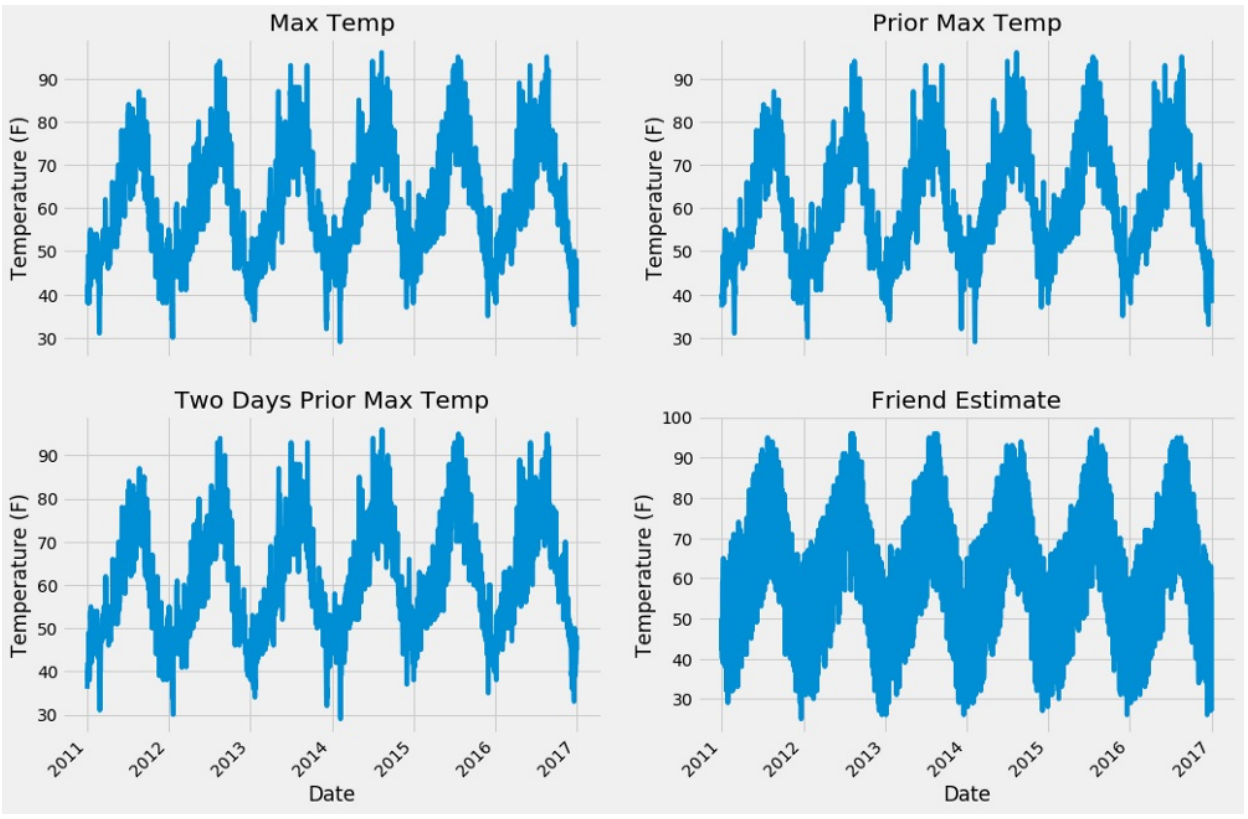


	year	month	day	ws_1	prcp_1	snwd_1	temp_2	temp_1	average	actual	friend
count	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00	2191.00
mean	2013.50	6.52	15.71	7.37	0.12	0.01	61.17	61.18	60.29	61.18	60.31
std	1.71	3.45	8.80	3.15	0.25	0.15	13.09	13.08	10.73	13.08	15.87
min	2011.00	1.00	1.00	0.89	0.00	0.00	29.00	29.00	45.10	29.00	25.00
25%	2012.00	4.00	8.00	5.14	0.00	0.00	51.00	51.00	50.10	51.00	49.00
50%	2014.00	7.00	16.00	6.71	0.00	0.00	60.00	60.00	58.80	60.00	60.00
75%	2015.00	10.00	23.00	9.17	0.12	0.00	71.00	71.00	70.20	71.00	71.00
max	2017.00	12.00	31.00	21.25	2.20	3.00	96.00	96.00	77.40	96.00	97.00

Expanded Data Summary

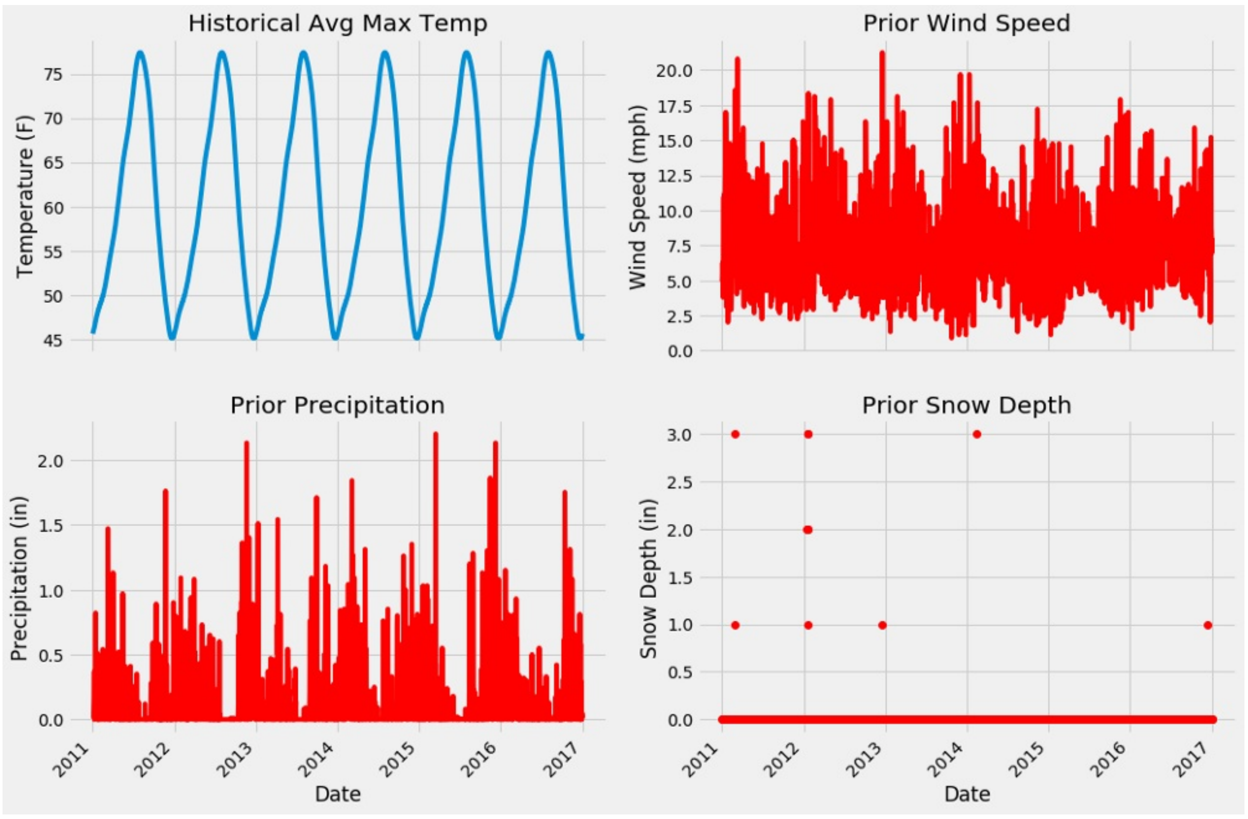
Nothing appears immediately anomalous from the descriptive statistics. We can quickly graph all of the variables to confirm this. I have left out the plotting code because while the matplotlib library is very useful, the code is non-intuitive and it can be easy to get lost in the details of the plots. (All the code is available for inspection and modification on GitHub).

First up are the four temperature plots.



Expanded Data Temperature Plots

Next we can look at the historical max average temperature and the three new variables.

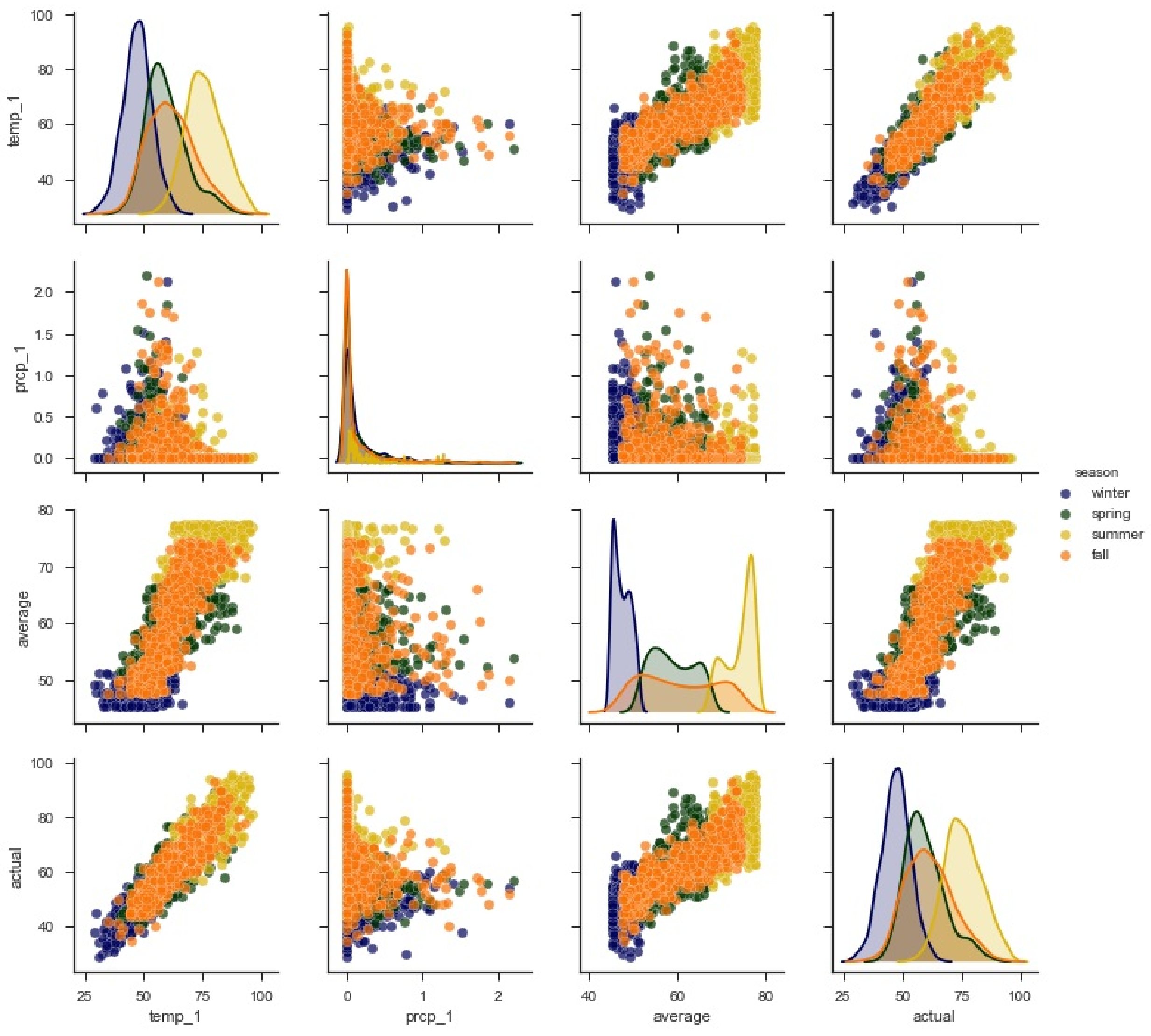


Expanded Data Additional Variables

From the numerical and graphical inspection, there are no apparent outliers n our data. Moreover, we can examine the plots to see which features will likely be useful. I think the snow depth will be the least helpful because it is zero for the majority of days, likewise, the wind speed looks to be too noisy to be of much assistance. From prior experience, the historical average max temperature and prior max temperature will probably be most important, but we will just have to see!

We can make one more exploratory plot, the pairplot, to visualize the relationships between variables. This plots all the variables against each other in scatterplots allowing us to inspect correlations between features. The code for this impressive-looking plot is rather simple compared to the above graphs!

```
# Create columns of seasons for pair plotting colors
seasons = []

for month in features['month']:
    if month in [1, 2, 12]:
        seasons.append('winter')
    elif month in [3, 4, 5]:
        seasons.append('spring')
    elif month in [6, 7, 8]:
        seasons.append('summer')
    elif month in [9, 10, 11]:
        seasons.append('fall')

# Will only use six variables for plotting pairs
reduced_features = features[['temp_1', 'prcp_1', 'ws_1', 'average', 'friend',
'actual']]
reduced_features['season'] = seasons

# Use seaborn for pair plots
import seaborn as sns
sns.set(style="ticks", color_codes=True);

# Create a custom color palete
palette = sns.xkcd_palette(['dark blue', 'dark green', 'gold', 'orange'])

# Make the pair plot with a some aesthetic changes
sns.pairplot(reduced_features, hue = 'season', diag_kind = 'kde', palette= palette,
plot_kws=dict(alpha = 0.7),
                   diag_kws=dict(shade=True))
```



Pairplots

The diagonal plots show the distribution of each variable because graphing each variable against itself would just be a straight line! The colors represent the four seasons as shown in the legend on the right. What we want to concentrate on are the trends between the actual max temperature and the other variables. These plots are in the bottom row, and to see a specific relationship with the actual max, move to the row containing the variable. For example, the plot in the bottom left shows the relationship between the actual max temperature and the max temperature from the previous day (temp_1). This is a strong positive correlation, indicating that as the max temperature one day prior increases, the max temperature the next day also increases

Data Preparation

The data has been validation both numerically and graphically, and now we need to put it in a format understandable by the machine learning algorithm. We will perform exactly the same data formatting procedure as in the simple implementation:

1. One-hot encode categorical variables (day of the week)
2. Separate data into features (independent varibles) and labels (targets)
3. Convert dataframes to Numpy arrays
4. Create random training and testing sets of features and labels

We can do all of those steps in a few lines of Python.

```
# One Hot Encoding
features = pd.get_dummies(features)

# Extract features and labels
labels = features['actual']
features = features.drop('actual', axis = 1)

# List of features for later use
feature_list = list(features.columns)

# Convert to numpy arrays
import numpy as np

features = np.array(features)
labels = np.array(labels)

# Training and Testing Sets
from sklearn.model_selection import train_test_split

train_features, test_features, train_labels, test_labels =
train_test_split(features, labels,                                          test_size
= 0.25, random_state = 42)
```

We set a random seed (of course it has to be 42) to ensure consistent results across runs. Let’s quickly do a check of the sizes of each array to confirm everything is in order.

```
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: (1643, 17)
Training Labels Shape: (1643,)
Testing Features Shape: (548, 17)
Testing Labels Shape: (548,)
```

Good to go! We have about 4.5 years of training data and 1.5 years of testing data. However, before we can get to the fun part of modeling, there is one additional step.

Establish a New Baseline

In the previous post, we used the historical average maximum temperature as our target to beat. That is, we evaluated the accuracy of predicting the max temperature tomorrow as the historical average max temperature on that day. We already know even the model trained on a single year of data can beat that baseline so we need to raise our expectations. For a new baseline, we will use the model trained on the original data. To make a fair comparison, we need to test it against the new, expanded test set. However, the new test set has 17 features, whereas the original model was only trained on 14 features. We first have to remove the 3 new features from the test set and then evaluate the original model. The original random forest has already been trained on the original data and code below shows preparing the testing features and evaluating the performance (refer to the notebook for the model training).

```
# Find the original feature indices
original_feature_indices = [feature_list.index(feature) for feature in feature_list
if feature not in ['ws_1', 'prcp_1', 'snwd_1']]

# Create a test set of the original features
original_test_features = test_features[:, original_feature_indices]

# Make predictions on test data using the model trained on original data
predictions = rf.predict(original_test_features)

# Performance metrics
errors = abs(predictions - test_labels)

print('Metrics for Random Forest Trained on Original Data')
print('Average absolute error:', round(np.mean(errors), 2), '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), '%.')
```

```
Metrics for Random Forest Trained on Original Data
Average absolute error: 4.3 degrees.
Accuracy: 92.49 %.
```

The random forest trained on the single year of data was able to achieve an average absolute error of 4.3 degrees representing an accuracy of 92.49% on the expanded test set. If our model trained with the expanded training set cannot beat these metrics, then we need to rethink our method.

Training and Evaluating on Expanded Data

The great part about Scikit-Learn is that many state-of-the-art models can be created and trained in a few lines of code. The random forest is one example:

```
# Instantiate random forest and train on new features
from sklearn.ensemble import RandomForestRegressor

rf_exp = RandomForestRegressor(n_estimators= 1000, random_state=100)
rf_exp.fit(train_features, train_labels)
```

Now, we can make predictions and compare to the known test set targets to confirm or deny that our expanded training dataset was a good investment:

```
# Make predictions on test data
predictions = rf_exp.predict(test_features)

# Performance metrics
errors = abs(predictions - test_labels)

print('Metrics for Random Forest Trained on Expanded Data')
print('Average absolute error:', round(np.mean(errors), 2), 'degrees.')

# Calculate mean absolute percentage error (MAPE)
mape = np.mean(100 * (errors / test_labels))
```

```
# Compare to baseline
improvement_baseline = 100 * abs(mape - baseline_mape) / baseline_mape
print('Improvement over baseline:', round(improvement_baseline, 2), '%.')

# Calculate and display accuracy
accuracy = 100 - mape
print('Accuracy:', round(accuracy, 2), '%.')
```

```
Metrics for Random Forest Trained on Expanded Data
Average absolute error: 3.7039 degrees.
Improvement over baseline: 16.67 %.
Accuracy: 93.74 %.
```

Well, we didn't waste our time getting more data! Training on six years worth of historical measurements and using three additional features has netted us a 16.41% improvement over the baseline model. The exact metrics will change depending on the random seed, but we can be confident that the new model outperforms the old model.

Why does a model improve with more data? The best way to answer this is to think in terms of how humans learn. We increase our knowledge of the world through experiences, and the more times we practice a skill, the better we get. A machine learning model also "learns from experience" in the sense that each time it looks at another training data point, it learns a little more about the relationships between the features and labels. Assuming that there are relationships in the data giving the model more data will allow it to better understand how to map a set of features to a label. For our case, as the model sees more days of weather measurements, it better understands how to take those measurements and predict the maximum temperature on the next day. Practice improves human abilities and machine learning model performance alike.

Feature Reduction

In some situations, we can go too far and actually use too much data or add too many features. One applicable example is a machine learning prediction problem involving building energy which I am currently working on. The problem is to predict building energy consumption in 15-minute intervals from weather data. For each building, I have 1–3 years of historical weather and electricity use data. Surprisingly, I found as I included more data for some buildings, the prediction accuracy decreased. Asking around, I determined some buildings had undergone retrofits to improve energy efficiency in the middle of data collection, and therefore, recent electricity consumption differed significantly from before the retrofit. When predicting current consumption, using data from before the modification actually decreased the performance of my models. More recent data from after the change was more relevant than the older data, and for several buildings, I ended up decreasing the amount of historical data to improve performance!

For our problem, the length of the data is not an issue because there have been no major changes affecting max temperatures in the six years of data (climate change is increasing temperatures but on a longer timescale). However, it may be possible we have too many features. We saw earlier that some of the features, especially our friend's prediction, looked more like noise than an accurate predictor of the maximum temperature. Extra features can decrease performance because they may "confuse" the model by giving it irrelevant data that prevents it from learning the actual relationships. The random forest performs implicit feature selection because it splits nodes on the most important variables, but other machine learning models do not. One approach to improve other models is therefore to use the random forest feature importances to reduce the number of variables in the problem. In our case, we will use the feature importances to decrease the number of features for our random forest model, because, in addition to potentially increasing performance, reducing the number of features will shorten the run time of the model. This post does not touch on more complex dimensionality reductions such as PCA (principal components analysis) or ICA (independent component analysis). These do a good job of decreasing the number of features while not decreasing information, but they transform the features such that they no longer represent our measured variables. I like machine learning models to have a blend of interpretability and accuracy, and I generally therefore stick to methods that allow me to understand how the model is making predictions.

Feature Importances

Finding the feature importances of a random forest is simple in Scikit-Learn. The actual calculation of the importances is beyond this blog post, but this occurs in the background and we can use the relative percentages returned by the model to rank the features.

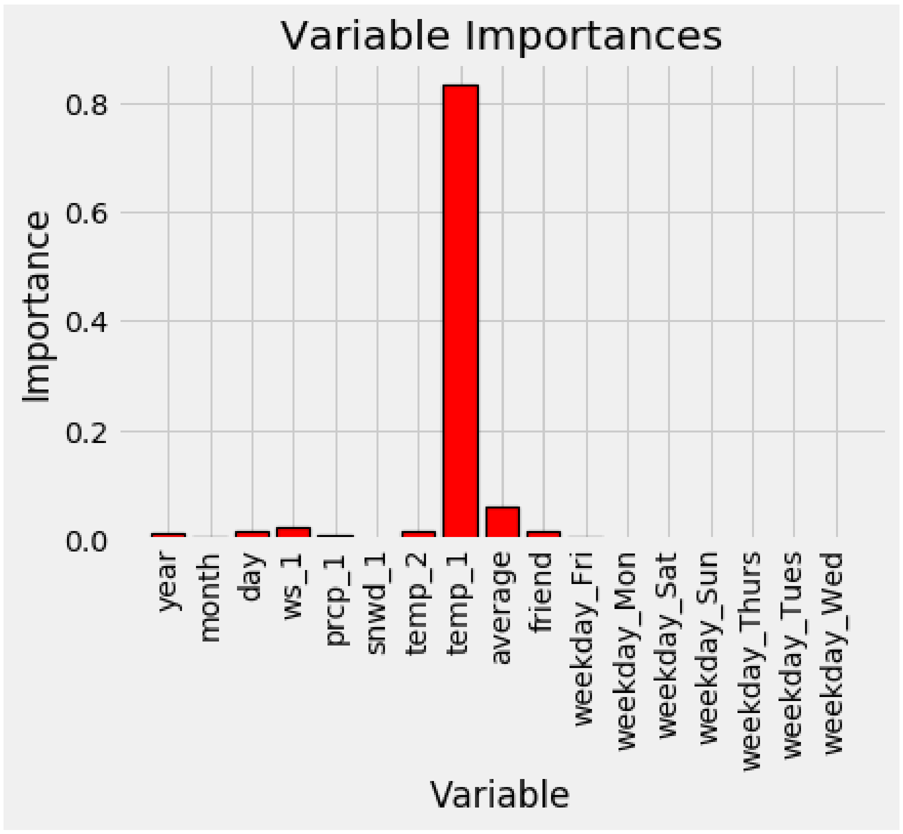
The following Python code creates a list of tuples where each tuple is a pair, (feature name, importance). The code here takes advantage of some neat tricks in the Python language, namely list comprehensive, zip, sorting, and argument unpacking. Don't worry if you do not understand these entirely, but if you want to become skilled at Python, these are tools you should have in your arsenal!

```
# Get numerical feature importances
importances = list(rf_exp.feature_importances_)

# List of tuples with variable and importance
feature_importances = [(feature, round(importance, 2)) for feature, importance in
zip(feature_list, importances)]

# 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: 0.83
Variable: average              Importance: 0.06
Variable: ws_1                 Importance: 0.02
Variable: temp_2               Importance: 0.02
Variable: friend               Importance: 0.02
Variable: year                 Importance: 0.01
Variable: month                Importance: 0.01
Variable: day                  Importance: 0.01
Variable: prcp_1               Importance: 0.01
Variable: snwd_1               Importance: 0.0
Variable: weekday_Fri          Importance: 0.0
Variable: weekday_Mon          Importance: 0.0
Variable: weekday_Sat          Importance: 0.0
Variable: weekday_Sun          Importance: 0.0
Variable: weekday_Thurs        Importance: 0.0
Variable: weekday_Tues         Importance: 0.0
Variable: weekday_Wed          Importance: 0.0
```

These stats definitely prove that some variables are much more important to our problem than others! Given that there are so many variables with zero importance (or near-zero due to rounding), it seems like we should be able to get rid of some of them without impacting performance. First, let's make a quick graph to represent the relative differences in feature importances. I left this plotting code in because it's a little easier to understand.

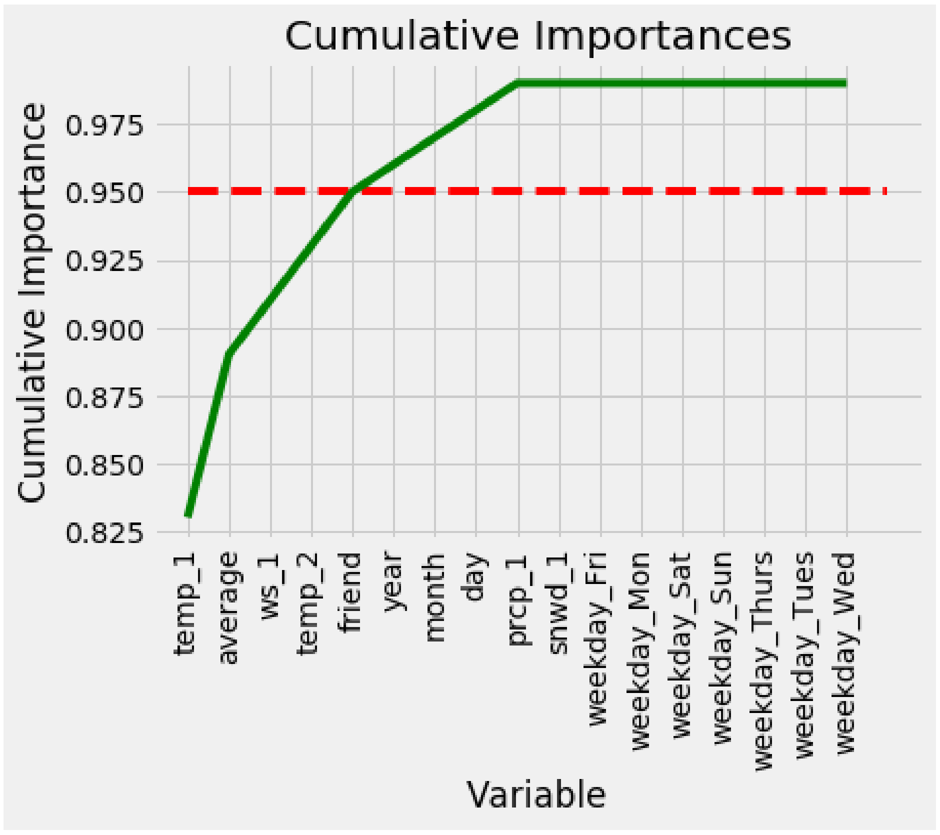
```
# list of x locations for plotting
x_values = list(range(len(importances)))

# Make a bar chart
plt.bar(x_values, importances, orientation = 'vertical', color = 'r', edgecolor =
'k', linewidth = 1.2)

# 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');
```



Expanded Model Variable Importances

We can also make a cumulative importance graph that shows the contribution to the overall importance of each additional variable. The dashed line is drawn at 95% of total importance accounted for.

```
# List of features sorted from most to least important
sorted_importances = [importance[1] for importance in feature_importances]
sorted_features = [importance[0] for importance in feature_importances]

# Cumulative importances
cumulative_importances = np.cumsum(sorted_importances)

# Make a line graph
plt.plot(x_values, cumulative_importances, 'g-')

# Draw line at 95% of importance retained
plt.hlines(y = 0.95, xmin=0, xmax=len(sorted_importances), color = 'r', linestyles =
'dashed')

# Format x ticks and labels
plt.xticks(x_values, sorted_features, rotation = 'vertical')

# Axis labels and title
plt.xlabel('Variable'); plt.ylabel('Cumulative Importance'); plt.title('Cumulative
Importances');
```



Cumulative Feature Importances

We can now use this to remove unimportant features. 95% is an arbitrary threshold, but if it leads to noticeably poor performance we can adjust the value. First, we need to find the exact number of features to exceed 95% importance:

```
# Find number of features for cumulative importance of 95%
# Add 1 because Python is zero-indexed
print('Number of features for 95% importance:', np.where(cumulative_importances >
0.95)[0][0] + 1)
```

```
Number of features for 95% importance: 6
```

We can then create a new training and testing set retaining only the 6 most important features.

```
# Extract the names of the most important features
important_feature_names = [feature[0] for feature in feature_importances[0:5]]
# Find the columns of the most important features
important_indices = [feature_list.index(feature) for feature in
important_feature_names]

# Create training and testing sets with only the important features
important_train_features = train_features[:, important_indices]
important_test_features = test_features[:, important_indices]

# Sanity check on operations
print('Important train features shape:', important_train_features.shape)
print('Important test features shape:', important_test_features.shape)
```

```
Important train features shape: (1643, 6)
Important test features shape: (548, 6)
```

We decreased the number of features from 17 to 6 (although to be fair, 7 of those features were created from the one-hot encoding of day of the week so we really only had 11 pieces of unique information). Hopefully this will not significantly decrease model accuracy and will considerably decrease the training time.

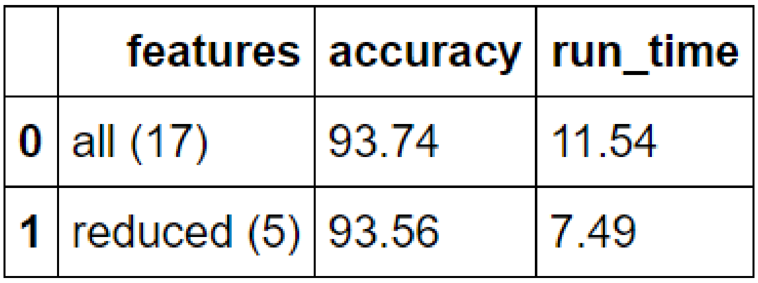
Training and Evaluating on Important Features

Now we go through the same train and test procedure as we did with all the features and evaluate the accuracy.

```
# Train the expanded model on only the important features
rf_exp.fit(important_train_features, train_labels);

# Make predictions on test data
predictions = rf_exp.predict(important_test_features)

# Performance metrics
errors = abs(predictions - test_labels)

print('Average absolute error:', round(np.mean(errors), 2), '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), '%.')
```

```
Average absolute error: 3.821 degrees.
Accuracy: 93.56 %.
```

The performance suffers a minor increase of 0.12 degrees average error using only 6 features. Often with feature reduction, there will be a minor decrease in performance that must be weighed against the decrease in run-time. Machine learning is a game of making trade-offs, and run-time versus performance is usually one of the critical decisions. I will quickly do some bench-marking to compare the relative run-times of the two models (see Jupyter Notebook for code).



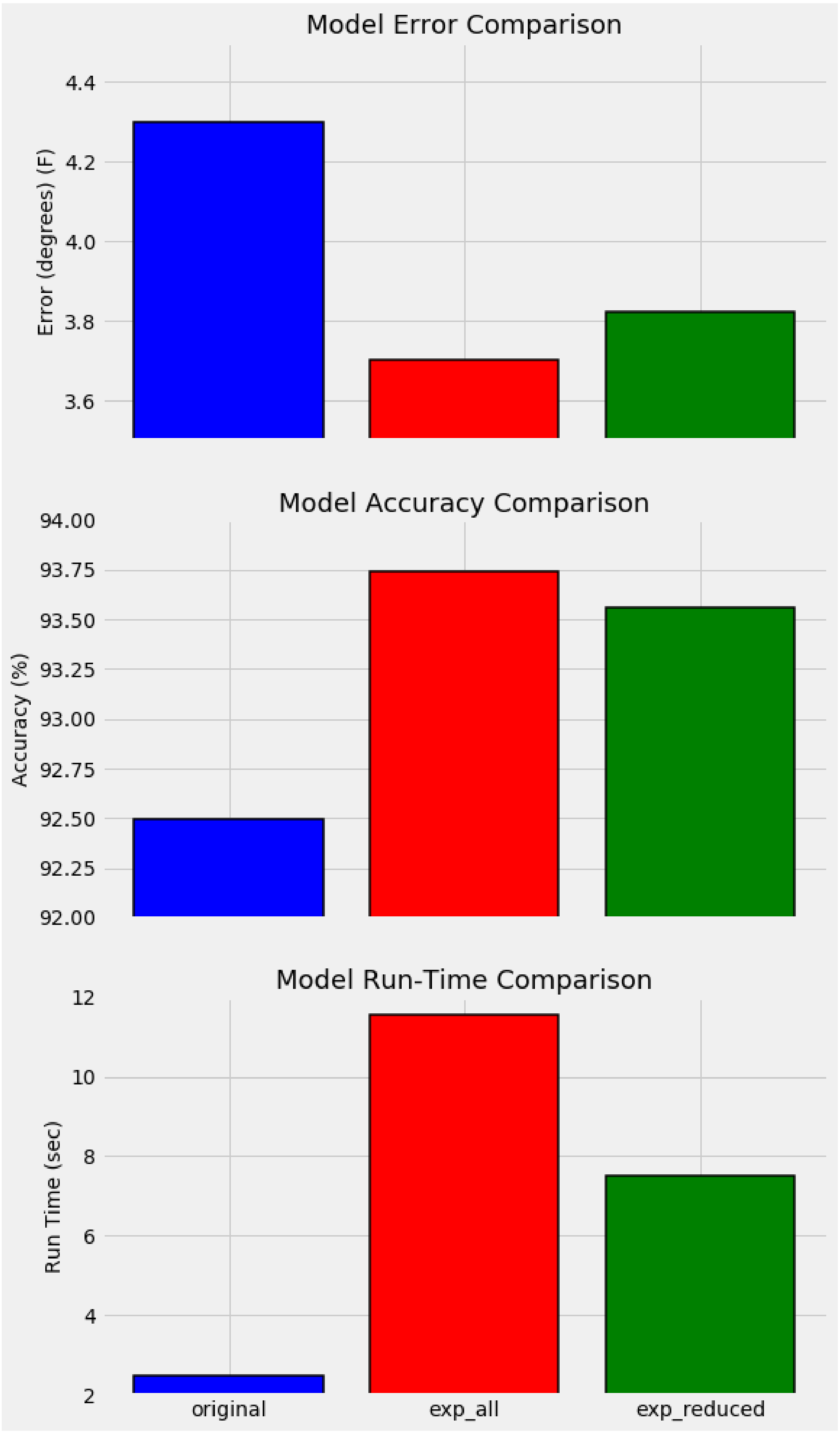
	features	accuracy	run_time
0	all (17)	93.74	11.54
1	reduced (5)	93.56	7.49

Model Tradeoffs

Overall, the reduced features model has a relative accuracy decrease of **0.131%** with a relative run-time decrease of **35.1%**. In our case, run-time is inconsequential because of the small size of the data set, but in a production setting, this trade-off likely would be worth it.

Conclusions

Instead of developing a more complex model to improve our random forest, we took the sensible step of collecting more data points and additional features. This approach was validated as we were able to decrease the error of compared to the model trained on limited data by 16.7%. Furthermore, by reducing the number of features from 17 to 6, we decreased our run-time by 35% while suffering only a minor decrease in accuracy. The best way to summarize these improvements is with another graph. The model trained on one year of training data is on the left, the model using six years of data and all features is in the middle, and the model on the right used six years of data but only a subset of the most important features.



Model Comparisons

This example has demonstrated the effectiveness of increasing the amount of data. While most people make the mistake of immediately moving to a more powerful model, we have learned most problems can be improving by collecting more relevant data points. In further parts of this series, we will take a look at the other ways to improve our model, namely, hyperparameter tuning and using different algorithms. However, getting more data is likely to have the largest payoff in terms of time invested versus increase in performance in this situation. The next time you see someone rushing to implement a complex deep learning model after failing on the first model, nicely ask them if they have exhausted all sources of data. Chances are, if there is still data out there relevant to their problem, they can get better performance and save time in the process!

As always, I appreciate any comments and constructive feedback. I can be reached at wjk68@case.edu

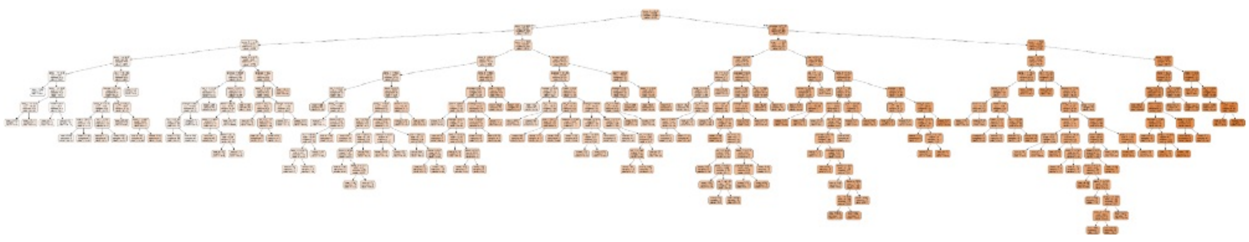
Hyperparameter Tuning the Random Forest in Python

towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

William Koehrsen January 10, 2018

Improving the Random Forest Part Two

So we've built a random forest model to solve our machine learning problem (perhaps by following this end-to-end guide) but we're not too impressed by the results. What are our options? As we saw in the first part of this series , our first step should be to gather more data and perform feature engineering. Gathering more data and feature engineering usually has the greatest payoff in terms of time invested versus improved performance, but when we have exhausted all data sources, it's time to move on to model hyperparameter tuning. This post will focus on optimizing the random forest model in Python using Scikit-Learn tools. Although this article builds on part one, it fully stands on its own, and we will cover many widely-applicable machine learning concepts.

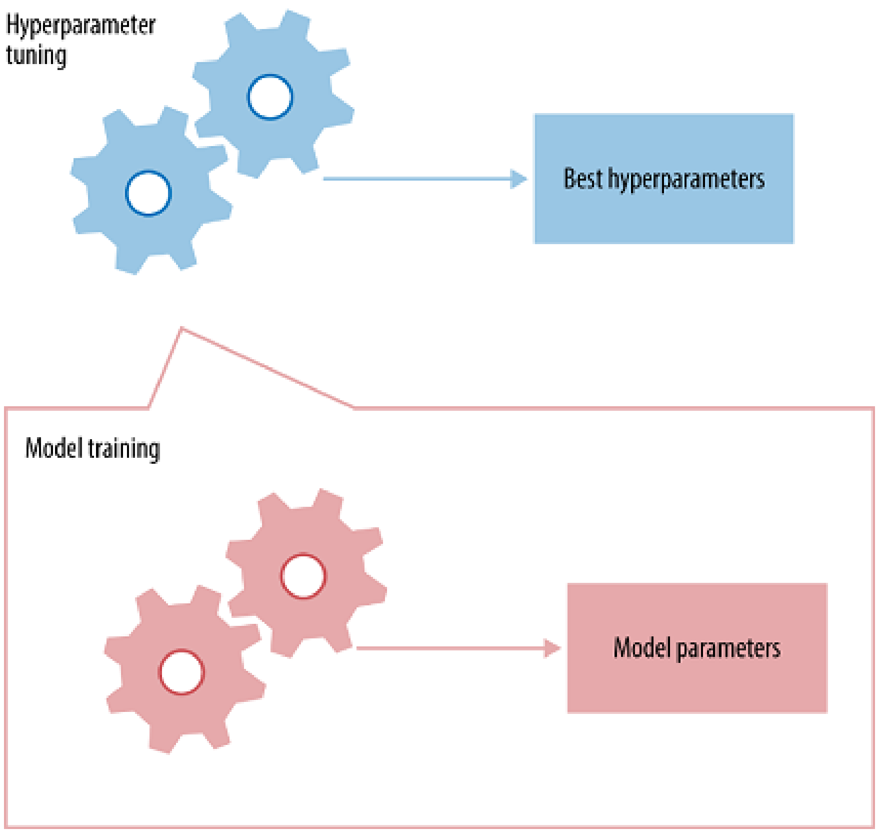


One Tree in a Random Forest

I have included Python code in this article where it is most instructive. Full code and data to follow along can be found on the project Github page.

A Brief Explanation of Hyperparameter Tuning

The best way to think about hyperparameters is like the settings of an algorithm that can be adjusted to optimize performance, just as we might turn the knobs of an AM radio to get a clear signal (or your parents might have!). While model *parameters* are learned during training — such as the slope and intercept in a linear regression — *hyperparameters* must be set by the data scientist beforetraining. In the case of a random forest, hyperparameters include the number of decision trees in the forest and the number of features considered by each tree when splitting a node. (The parameters of a random forest are the variables and thresholds used to split each node learned during training). Scikit-Learn implements a set of sensible default hyperparameters for all models, but these are not guaranteed to be optimal for a problem. The best hyperparameters are usually impossible to determine ahead of time, and tuning a model is where machine learning turns from a science into trial-and-error based engineering.



Hyperparameters and Parameters

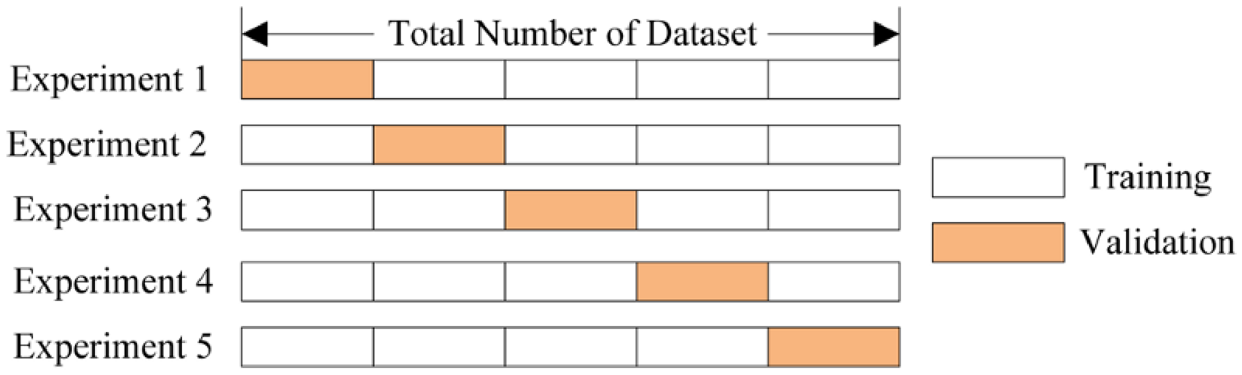
Hyperparameter tuning relies more on experimental results than theory, and thus the best method to determine the optimal settings is to try many different combinations evaluate the performance of each model. However, evaluating each model only on the training set can lead to one of the most fundamental problems in machine learning: overfitting.

If we optimize the model for the training data, then our model will score very well on the training set, but will not be able to generalize to new data, such as in a test set. When a model performs highly on the training set but poorly on the test set, this is known as overfitting, or essentially creating a model that knows the training set very well but cannot be applied to new problems. It’s like a student who has memorized the simple problems in the textbook but has no idea how to apply concepts in the messy real world.

An overfit model may look impressive on the training set, but will be useless in a real application. Therefore, the standard procedure for hyperparameter optimization accounts for overfitting through cross validation.

Cross Validation

The technique of cross validation (CV) is best explained by example using the most common method, K-Fold CV. When we approach a machine learning problem, we make sure to split our data into a training and a testing set. In K-Fold CV, we further split our training set into K number of subsets, called folds. We then iteratively fit the model K times, each time training the data on K-1 of the folds and evaluating on the Kth fold (called the validation data). As an example, consider fitting a model with K = 5. The first iteration we train on the first four folds and evaluate on the fifth. The second time we train on the first, second, third, and fifth fold and evaluate on the fourth. We repeat this procedure 3 more times, each time evaluating on a different fold. At the very end of training, we average the performance on each of the folds to come up with final validation metrics for the model.



5 Fold Cross Validation (Source)

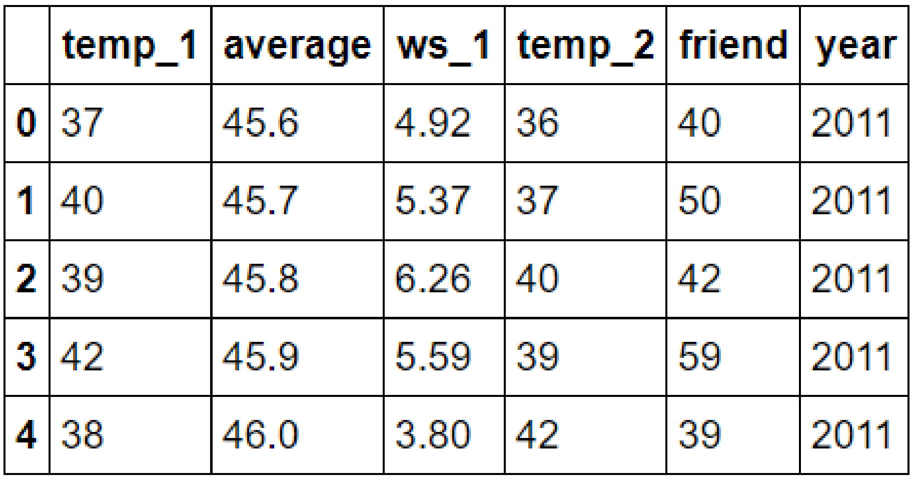
For hyperparameter tuning, we perform many iterations of the entire K-Fold CV process, each time using different model settings. We then compare all of the models, select the best one, train it on the full training set, and then evaluate on the testing set. This sounds like an awfully tedious process! Each time we want to assess a different set of hyperparameters, we have to split our training data into K fold and train and evaluate K times. If we have 10 sets of hyperparameters and are using 5-Fold CV, that represents 50 training loops. Fortunately, as with most problems in machine learning, someone has solved our problem and model tuning with K-Fold CV can be automatically implemented in Scikit-Learn.

Random Search Cross Validation in Scikit-Learn

Usually, we only have a vague idea of the best hyperparameters and thus the best approach to narrow our search is to evaluate a wide range of values for each hyperparameter. Using Scikit-Learn’s RandomizedSearchCV method, we can define a grid of hyperparameter ranges, and randomly sample from the grid, performing K-Fold CV with each combination of values.

As a brief recap before we get into model tuning, we are dealing with a supervised regression machine learning problem. We are trying to predict the temperature tomorrow in our city (Seattle, WA) using past historical weather data. We have 4.5 years of training data, 1.5 years of test data, and are using 6 different features (variables) to make our predictions. (To see the full code for data preparation, see the notebook).

Let’s examine the features quickly.



	temp_1	average	ws_1	temp_2	friend	year
0	37	45.6	4.92	36	40	2011
1	40	45.7	5.37	37	50	2011
2	39	45.8	6.26	40	42	2011
3	42	45.9	5.59	39	59	2011
4	38	46.0	3.80	42	39	2011

Features for Temperature Prediction

- temp_1 = max temperature (in F) one day prior
- average = historical average max temperature
- ws_1 = average wind speed one day prior
- temp_2 = max temperature two days prior
- friend = prediction from our “trusty” friend
- year = calendar year

In previous posts, we checked the data to check for anomalies and we know our data is clean. Therefore, we can skip the data cleaning and jump straight into hyperparameter tuning.

To look at the available hyperparameters, we can create a random forest and examine the default values.

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(random_state = 42)

from pprint import pprint

# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(rf.get_params())
```

```
Parameters currently in use:

{'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': 10,
 'n_jobs': 1,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

Wow, that is quite an overwhelming list! How do we know where to start? A good place is the documentation on the random forest in Scikit-Learn . This tells us the most important settings are the number of trees in the forest (n_estimators) and the number of features considered for splitting at each leaf node (max_features). We could go read the research papers on the random forest and try to theorize the best hyperparameters, but a more efficient use of our time is just to try out a wide range of values and see what works! We will try adjusting the following set of hyperparameters:

- n_estimators = number of trees in the foreset
- max_features = max number of features considered for splitting a node
- max_depth = max number of levels in each decision tree
- min_samples_split = min number of data points placed in a node before the node is split
- min_samples_leaf = min number of data points allowed in a leaf node
- bootstrap = method for sampling data points (with or without replacement)

Random Hyperparameter Grid

To use RandomizedSearchCV, we first need to create a parameter grid to sample from during fitting:

```
from sklearn.model_selection import RandomizedSearchCV
```

```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Method of selecting samples for training each tree
bootstrap = [True, False]

# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

pprint(random_grid)
```

```
{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}
```

On each iteration, the algorithm will choose a difference combination of the features. Altogether, there are 2 * 12 * 2 * 3 * 3 * 10 = 4320 settings! However, the benefit of a random search is that we are not trying every combination, but selecting at random to sample a wide range of values.

Random Search Training

Now, we instantiate the random search and fit it like any Scikit-Learn model:

```
# Use the random grid to search for best hyperparameters
# First create the base model to tune
rf = RandomForestRegressor()
# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid,
n_iter = 100, cv = 3, verbose=2, random_state=42, n_jobs = -1)

# Fit the random search model
rf_random.fit(train_features, train_labels)
```

The most important arguments in RandomizedSearchCV are n_iter, which controls the number of different combinations to try, and cv which is the number of folds to use for cross validation (we use 100 and 3 respectively). More iterations will cover a wider search space and more cv folds reduces the chances of overfitting, but raising each will increase the run time. Machine learning is a field of trade-offs, and performance vs time is one of the most fundamental.

We can view the best parameters from fitting the random search:

```
rf_random.best_params_
```

```
{'bootstrap': True,
 'max_depth': 70,
 'max_features': 'auto',
 'min_samples_leaf': 4,
 'min_samples_split': 10,
 'n_estimators': 400}
```

From these results, we should be able to narrow the range of values for each hyperparameter.

Evaluate Random Search

To determine if random search yielded a better model, we compare the base model with the best random search model.

```
def evaluate(model, test_features, test_labels):
    predictions = model.predict(test_features)
    errors = abs(predictions - test_labels)
    mape = 100 * np.mean(errors / test_labels)
    accuracy = 100 - mape
    print('Model Performance')
    print('Average Error: {:0.4f} degrees.'.format(np.mean(errors)))
    print('Accuracy = {:0.2f}%.'.format(accuracy))

return accuracy

base_model = RandomForestRegressor(n_estimators = 10, random_state = 42)
base_model.fit(train_features, train_labels)
base_accuracy = evaluate(base_model, test_features, test_labels)
```

```
Model Performance
Average Error: 3.9199 degrees.
Accuracy = 93.36%.
```

```
best_random = rf_random.best_estimator_
random_accuracy = evaluate(best_random, test_features, test_labels)
```

```
Model Performance
Average Error: 3.7152 degrees.
Accuracy = 93.73%.
```

```
print('Improvement of {:0.2f}%.'.format( 100 * (random_accuracy - base_accuracy) / base_accuracy))
```

```
Improvement of 0.40%.
```

We achieved an unspectacular improvement in accuracy of 0.4%. Depending on the application though, this could be a significant benefit. We can further improve our results by using grid search to focus on the most promising hyperparameters ranges found in the random search.

Grid Search with Cross Validation

Random search allowed us to narrow down the range for each hyperparameter. Now that we know where to concentrate our search, we can explicitly specify every combination of settings to try. We do this with GridSearchCV, a method that, instead of sampling randomly from a distribution, evaluates all combinations we define. To use Grid Search, we make another grid based on the best values provided by random search:

```
from sklearn.model_selection import GridSearchCV

# Create the parameter grid based on the results of random search
param_grid = {
    'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': [3, 4, 5],
    'min_samples_split': [8, 10, 12],
    'n_estimators': [100, 200, 300, 1000]
}

# Create a based model
rf = RandomForestRegressor()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                          cv = 3, n_jobs = -1, verbose = 2)
```

This will try out 1 * 4 * 2 * 3 * 3 * 4 = 288 combinations of settings. We can fit the model, display the best hyperparameters, and evaluate performance:

```
# Fit the grid search to the data
grid_search.fit(train_features, train_labels)

grid_search.best_params_
```

```
{'bootstrap': True,
 'max_depth': 80,
 'max_features': 3,
 'min_samples_leaf': 5,
 'min_samples_split': 12,
 'n_estimators': 100}
```

```
best_grid = grid_search.best_estimator_
grid_accuracy = evaluate(best_grid, test_features, test_labels)
```

```
Model Performance
Average Error: 3.6561 degrees.
Accuracy = 93.83%.
```

```
print('Improvement of {:0.2f}%.'.format( 100 * (grid_accuracy - base_accuracy) /
base_accuracy))
```

```
Improvement of 0.50%.
```

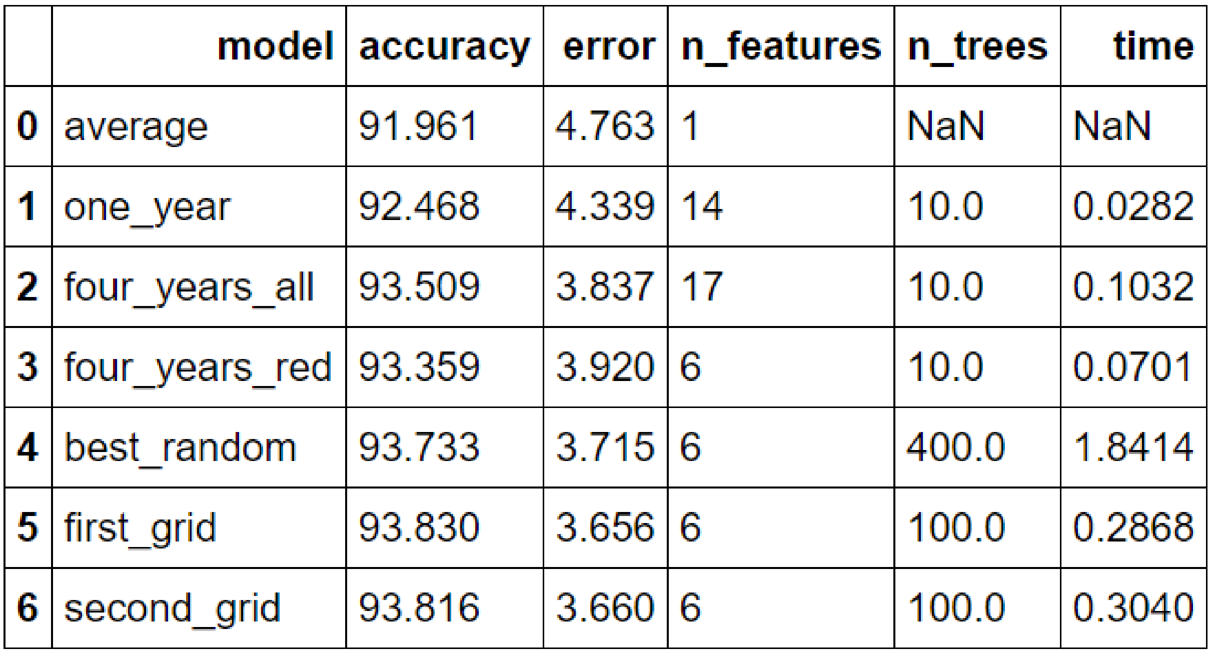
It seems we have about maxed out performance, but we can give it one more try with a grid further refined from our previous results. The code is the same as before just with a different grid so I only present the results:

```
Model Performance
Average Error: 3.6602 degrees.
Accuracy = 93.82%.

Improvement of 0.49%.
```

A small decrease in performance indicates we have reached diminishing returns for hyperparameter tuning. We could continue, but the returns would be minimal at best.

Comparisons

We can make some quick comparisons between the different approaches used to improve performance showing the returns on each. The following table shows the final results from all the improvements we made (including those from the first part):



	model	accuracy	error	n_features	n_trees	time
0	average	91.961	4.763	1	NaN	NaN
1	one_year	92.468	4.339	14	10.0	0.0282
2	four_years_all	93.509	3.837	17	10.0	0.1032
3	four_years_red	93.359	3.920	6	10.0	0.0701
4	best_random	93.733	3.715	6	400.0	1.8414
5	first_grid	93.830	3.656	6	100.0	0.2868
6	second_grid	93.816	3.660	6	100.0	0.3040

Comparison of All Models

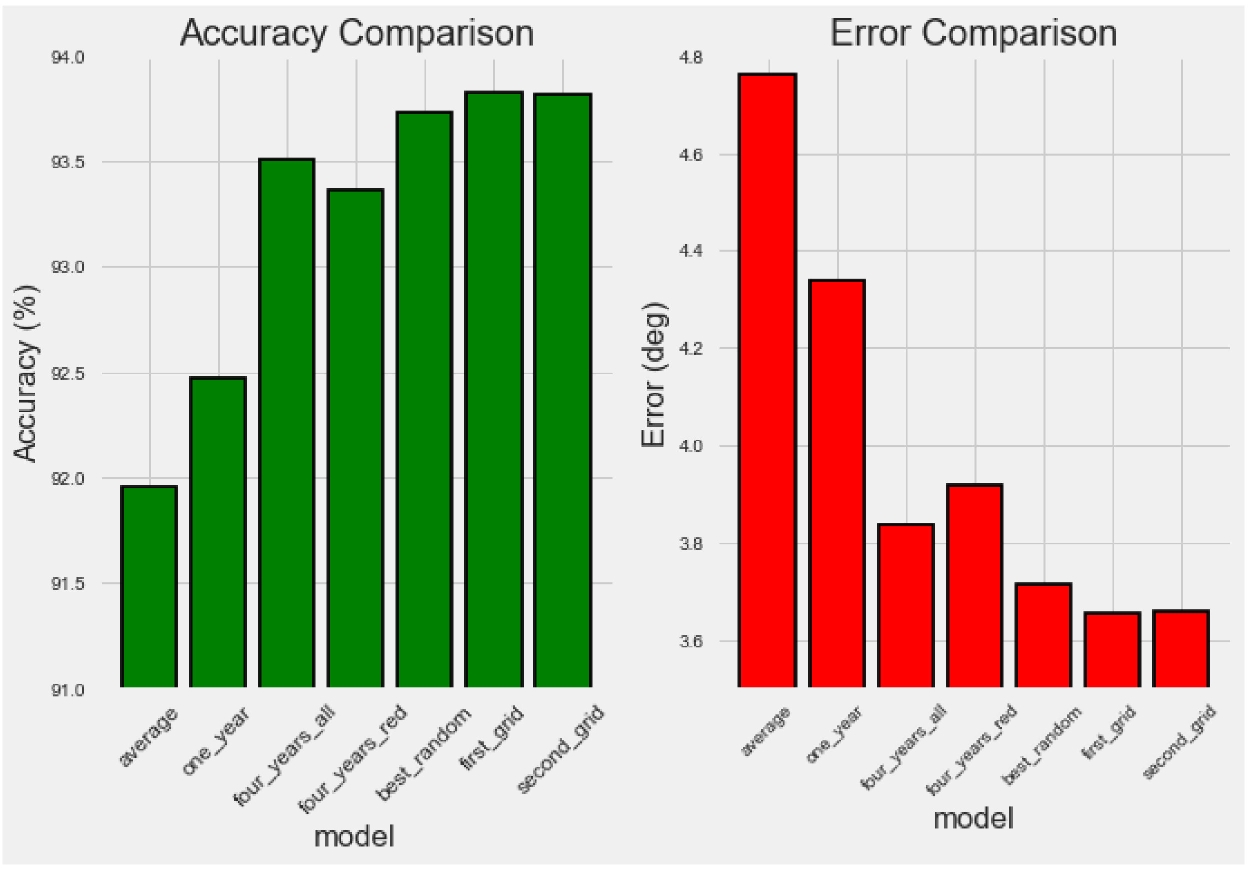
Model is the (very unimaginative) names for the models, accuracy is the percentage accuracy, error is the average absolute error in degrees, n_features is the number of features in the dataset, n_trees is the number of decision trees in the forest, and time is the training and predicting time in seconds.

The models are as follows:

- average: original baseline computed by predicting historical average max temperature for each day in test set
- one_year: model trained using a single year of data
- four_years_all: model trained using 4.5 years of data and expanded features (see Part One for details)
- four_years_red: model trained using 4.5 years of data and subset of most important

features
- best_random: best model from random search with cross validation
- first_grid: best model from first grid search with cross validation (selected as the final model)
- second_grid: best model from second grid search

Overall, gathering more data and feature selection reduced the error by 17.69% while hyperparameter further reduced the error by 6.73%.



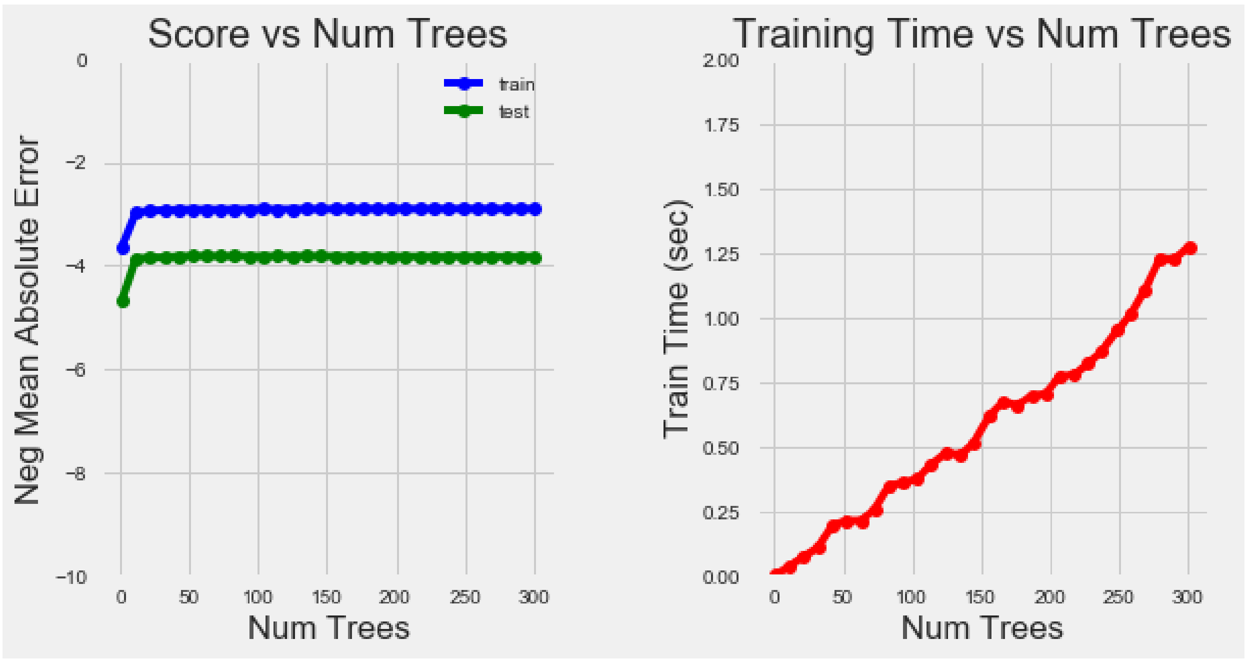
Model Comparison (see Notebook for code)

In terms of programmer-hours, gathering data took about 6 hours while hyperparameter tuning took about 3 hours. As with any pursuit in life, there is a point at which pursuing further optimization is not worth the effort and knowing when to stop can be just as important as being able to keep going (sorry for getting all philosophical). Moreover, in any data problem, there is what is called the Bayes error rate, which is the absolute minimum possible error in a problem. Bayes error, also called reproducible error, is a combination of latent variables, the factors affecting a problem which we cannot measure, and inherent noise in any physical process. Creating a perfect model is therefore not possible. Nonetheless, in this example, we were able to significantly improve our model with hyperparameter tuning and we covered numerous machine learning topics which are broadly applicable.

Training Visualizations

To further analyze the process of hyperparameter optimization, we can change one setting at a time and see the effect on the model performance (essentially conducting a controlled experiment). For example, we can create a grid with a range of number of trees, perform grid search CV, and then plot the results. Plotting the training and testing error and the training time will allow us to inspect how changing one hyperparameter impacts the model.

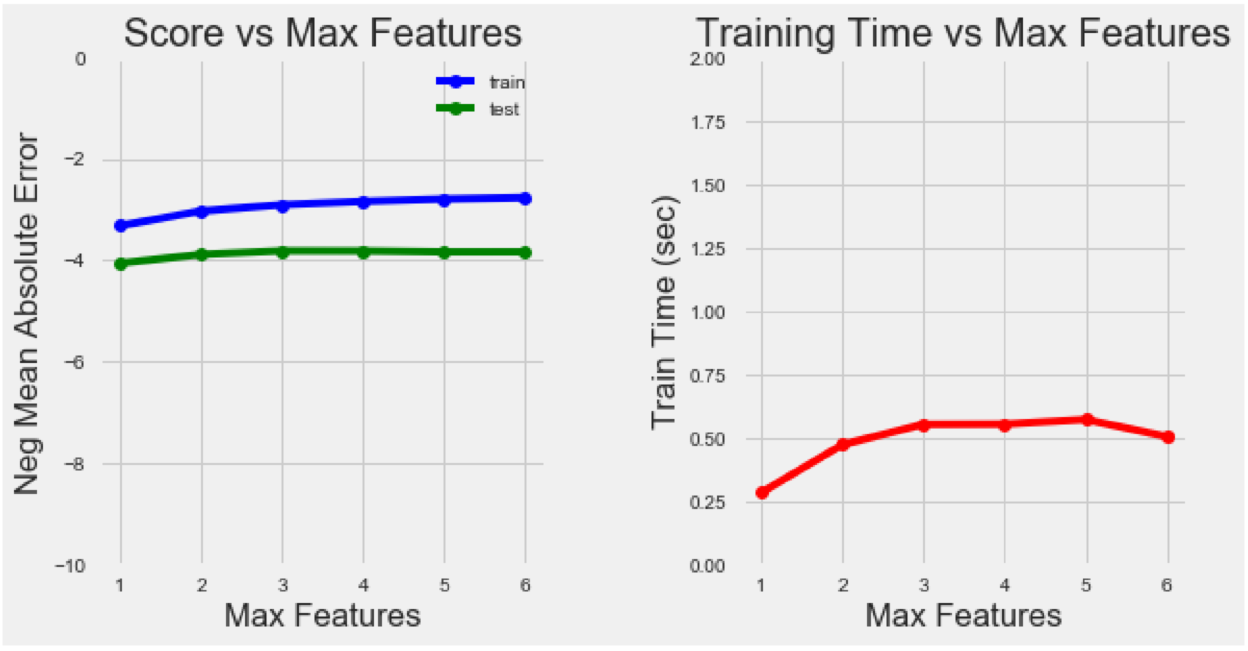
First we can look at the effect of changing the number of trees in the forest. (see notebook for training and plotting code)



Number of Trees Training Curves

As the number of trees increases, our error decreases up to a point. There is not much benefit in accuracy to increasing the number of trees beyond 20 (our final model had 100) and the training time rises consistently.

We can also examine curves for the number of features to split a node:



Number of Features Training Curves

As we increase the number of features retained, the model accuracy increases as expected. The training time also increases although not significantly.

Together with the quantitative stats, these visuals can give us a good idea of the trade-offs we make with different combinations of hyperparameters. Although there is usually no way to know ahead of time what settings will work the best, this example has demonstrated the simple tools in Python that allow us to optimize our machine learning model.

As always, I welcome feedback and constructive criticism. I can be reached at wjk68@case.edu