Random Forests and the Bias-Variance Tradeoff

Prratek Ramchandani Follow
Oct 10, 2018 · 10 min read

The Random Forest is an extremely popular machine learning algorithm. Often, with not too much pre-processing, one can throw together a quick and dirty model with no hyperparameter tuning and acheive results that aren't awful. As an example, I put together a `RandomForestRegressor` in Python using `scikit-learn` for the New York City Taxi Fare Prediction playground competition on Kaggle recently, passing in no arguments to the model constructor and using 1/100 for the training data (554238 of ~55M rows), for a validation R^2 of ~0.8. Try it yourself!

```
1  import pandas as pd
```

```
from sklearn.ensemble import RandomForestRegressor

n = 100   # use every 100th row
df = pd.read_csv('{PATH_TO_DATA}train.csv', skiprows=lambda i: i % n != 0)

m = RandomForestRegressor()   # instantiate the RandomForestRegressor objects
m.fit(X_train, y_train)   # train the model
m.score(X_valid, y_valid)   # score it on your validation set
```

nycTaxi_rf_baseline.py hosted with ❤️ by GitHub view raw

NOTE: This snippet assumes you split the data into training and validation sets with your features and target variable separated. You can see the full code on my GitHub profile.

Part of what makes this algorithm so clever is how it handles something called **the bias-variance tradeoff**. I explore this aspect of Random Forests in the following 5 steps:

1. **Bias and Variance**
2. **Decision Trees**
3. **Bagging, Bootstrapping, and Random Forests**
4. **Hyperparameter Tuning**
5. **Random Forests and the Bias-Variance Tradeoff**

Bias and Variance

The Mean Squared Error (MSE) of a statistical model can be expressed as the sum of the **squared bias of its predictions**, the **variance of those predictions**, and the **variance of some error term ϵ**. Since both squared bias and variance are non-negative, and ϵ, which captures randomness in the data, is beyond our control, we minimize MSE by **minimizing the variance and bias** of our model. I have found the image in *Fig. 1* to be particularly good at illustrating what the two terms mean.

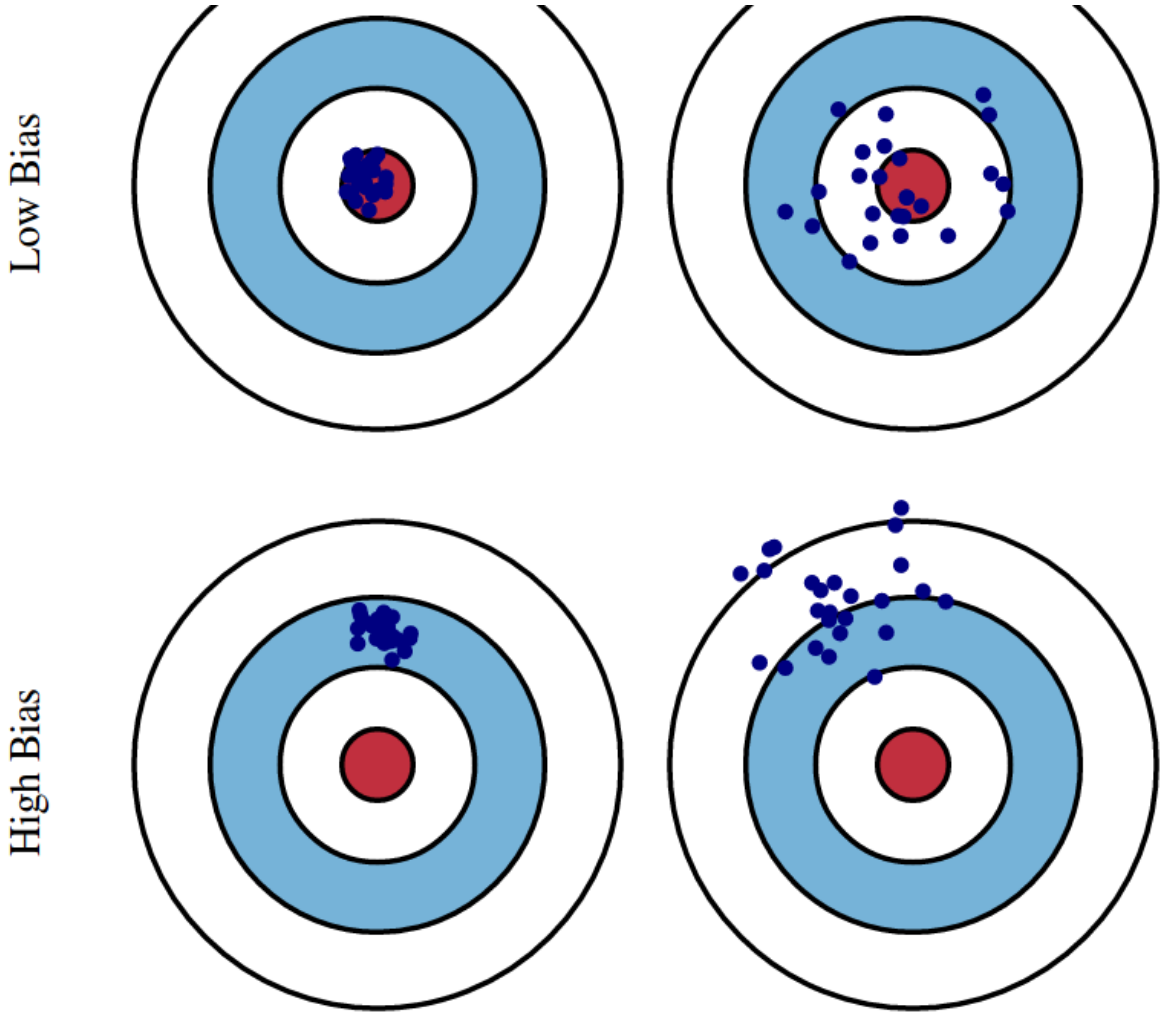


Fig. 1: A visual representation of the terms bias and variance.

We say our model is biased if it systematically under or over predicts the target variable. In machine learning, this is often the result either of the statistical assumptions made by our model of choice or of bias in the training data. Take a look at this article for an example of bias where Google's Cloud Natural Language API learned through text on the internet that the word "homosexual" carries an inherent negative connotation.

Variance, on the other hand, in some sense captures the generalizability of the model. Put more precisely, it is a measure of how much our prediction would change if we trained it on different data. High variance typicaly means that we are overfitting to our training data, finding patterns and complexity that are a product of randomness as opposed to some real trend. Generally, a more complex or flexible model will tend to have high variance due to overfitting but lower bias because, averaged over several predictions, our model more accurately predicts the target variable. On the other hand,

an underfit or oversimplified model, while having lower variance, will likely be more biased since it lacks the tools to fully capture trends in the data.

What we would like, ideally, is **low bias-low variance.** To see how to acheive this, let's first look at a typical bias squared-variance curve.

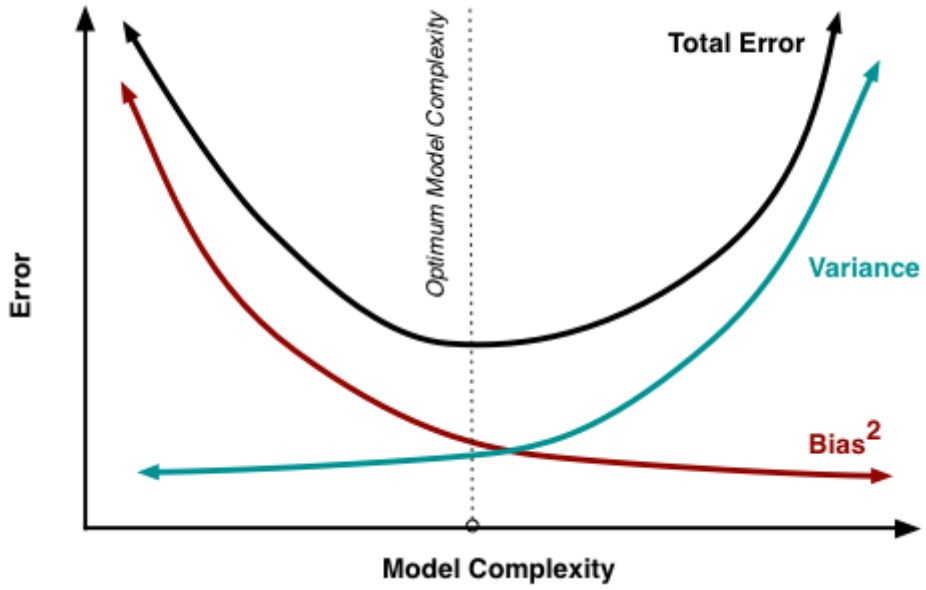
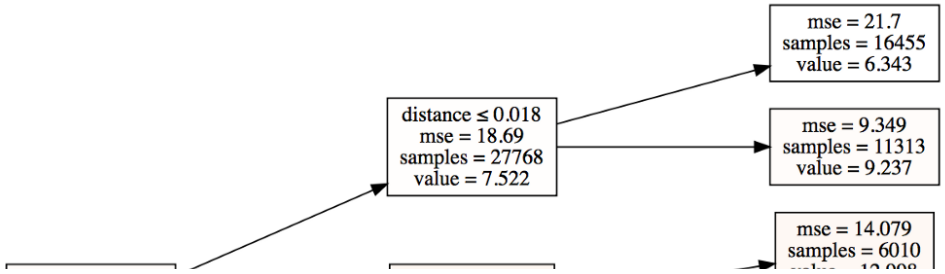


Fig. 2: A curve of squared bias vs variance showing the inverse correlation that is typical of the relation between the two as the model gets more complex. It is not uncommon for the resulting Total Error to follow some variant of the U-shape shown in the figure.

Fig. 2 illustrates the general trend I described above of decreasing bias and increasing variance as our model gets more complex. Our goal is to **choose a model that minimizes the sum of the two** as illustrated by the dotted line. To see how a Random Forest does this particularly well, let's start with a simple decision tree.

. . .

Decision Trees



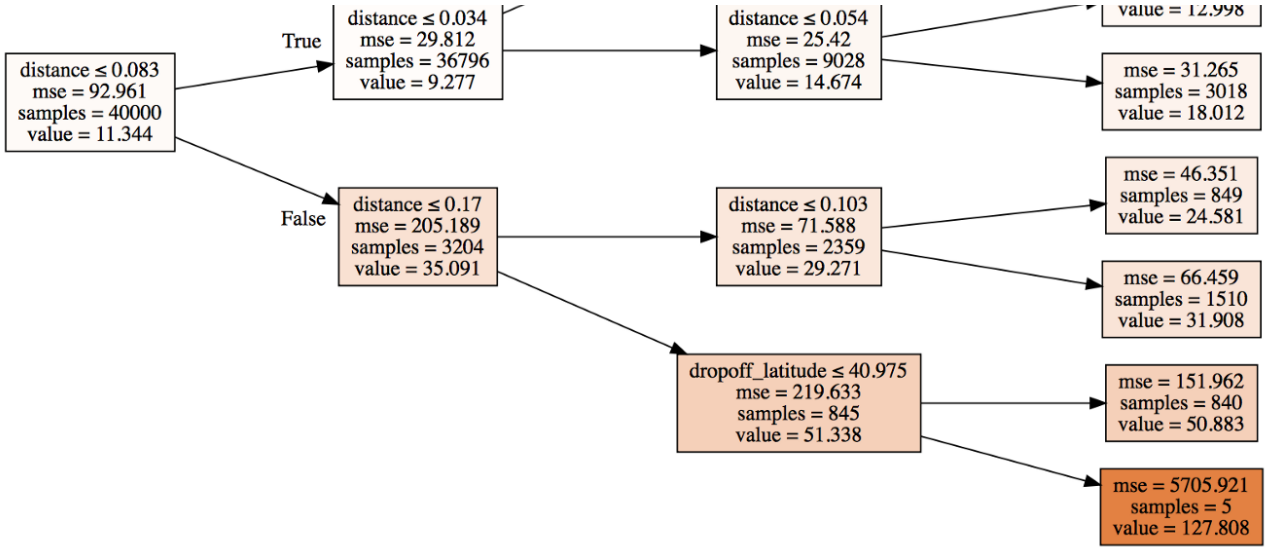


Fig. 3: Representation of a single decision tree with no bootstrapping and max_depth of 3 that I created for the New York City Taxi Fare Prediction competition on Kaggle.

Let's go back to the taxi fare prediction task. Given certain features of a particular taxi ride, a decision tree starts off by simply predicting the average taxi fare in the training dataset ($11.33) as shown in the leftmost box in *Fig. 3*. It then goes through the list of all features and their values to find a **binary split that gives us the maximum improvement in MSE**. This is typically calculated by predicting the mean of each of the two new subsets and computing their respective MSEs weighted by the number of observations in each.

Fig. 4: A New York City taxi meter.

In this case the best split happens to be whether the ride distance was less than or equal to 0.083 (units of degrees latitude/longitude = 5.727 miles). When it is less than 5.727 miles, the tree predicts $9.28 — the mean fare of those rides with distance less than this — and through the same procedure reaches a prediction of $35.09 for those longer than our threshold. This makes sense. Long rides are more expensive.

As illustrated by the figure, it then splits each *branch* into new branches on the same criterion of maximal improvement in MSE, continuing recursively until each *leaf* (group of samples) has only one training observation in it. Decision Trees have extremely low bias because they maximally overfit to the training data. **Each "prediction" it makes on the validation set would in essence be the fare of some taxi ride in our training data that ended up in the same final leaf node as the ride whose fare we are predicting.** This overfitting, however, also results in unacceptably high variance and consequently poor predictions on unseen data.

. . .

Bagging, Bootstrapping, and Random Forests

While an individual tree is overfit to the training data and is likely to have large error, *bagging* (**B**ootstrap **Agg**regat**ing**) uses the insight that a suitably **large number of uncorrelated errors average out to zero** to solve this problem. Bagging chooses multiple random samples of observations from the training data, with replacement, constructing a tree from each one. Since each tree learns from different data, they are fairly uncorrelated from one another. Plotting the R^2 of our model as we increase the number of "bagged" trees (`scikit-learn` calls these trees `estimators`) illustrates the power of this technique.

```
1  import numpy as np
```

```
import matplotlib.pyplot as plt
from sklearn import metrics

preds = np.stack([t.predict(X_valid) for t in m.estimators_])
preds[:,0], np.mean(preds[:,0])
plt.plot([metrics.r2_score(y_valid, np.mean(preds[:i+1], axis=0)) for i in range(10)]);
```

bagged_trees_r2.py hosted with ❤️ by GitHub view raw

```
(array([ 9. ,  8.1,  8.1, 11.5,  6.1,  6.9,  6.5,  8.1, 10.5,
6.5]), 8.13)
```

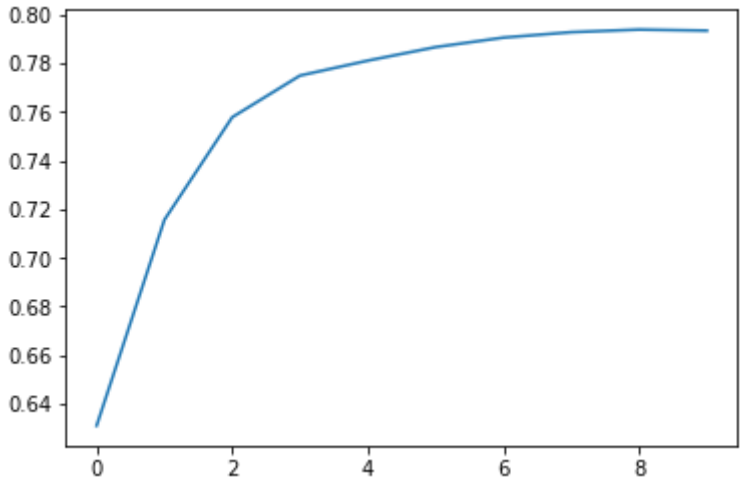


Fig. 5: A plot showing incresing R^2 as n_estimators is increased from 1 to 9.

Think of each tree in our forest as learning some unique insights about what contributes to the fare of a New York City taxi from the subset of data it models. Averaging together each of their predictions then gives us a stronger, more stable model able to predict the fare of a taxi ride it hasn't previously been exposed to with better accuracy.

. . .

Hyperparameter Tuning

Random Forests, however, are more than just bagged trees and use a number of interesting techniques to further decrease correlation between trees and reduce

overfitting. A quick look at the documentation for scikit-learn's implementation of the `RandomForestRegressor` shows us the *hyperparameters* we can pass in:

```
class sklearn.ensemble.RandomForestRegressor(n_estimators=10,
criterion='mse', max_depth=None, min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0,
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1,
random_state=None, verbose=0, warm_start=False
```

There's over a dozen of them but here I take a closer look what `n_estimators`, `max_depth`, `min_samples_leaf`, and `max_features` do and why each of them could potentially decrease the error of your model.

n_estimators

`n_estimators` is simply the number of trees. **The more uncorrelated trees in our forest, the closer their individual errors get to averaging out.** However, more is not always better and here are some considerations to keep in mind:

1. **More trees = more computation.** Beyond a certain point, the tradeoff may not be worth it.
2. On a related note, increasing `n_estimators` gives **diminishing returns** as evidenced by Fig. 5. Increasing from 370 trees to 400, for example, isn't even guaranteed to decrease your validation MSE.
3. No number of uncorrelated trees will get rid of error caused by the assumptions made by your model or bias resulting from unrepresentative training data.

max_depth

`max_depth` is the how many splits deep you want each tree to go. `max_depth = 50`, for example, would limit trees to at most 50 splits down any given branch. This has the consequence that our Random Forest can no more fit the training data as closely, and is consequently more stable. It has lower variance, giving our model lower error. Remember that even though severely constraining `max_depth` could increase the bias of each tree given that they may not be able to capture certain patterns in the data before hitting their limit, we need not worry about this. A suitable choice of `n_estimators`,

coupled with bagging, ensures that the bias of the forest as a whole doesn't increase in the process.

`max_depth` is a hyperparameter that I typically leave untouched simply because what I really care about is how many observations are at the end of a branch before I forbid the tree from splitting further. This is a better predictor of how overfit the Random Forest is.

min_samples_leaf

`min_samples_leaf` allows us to do exactly what I described above. `min_samples_leaf = 10`, for instance, tells each tree to stop splitting if doing so would result in the end node of any resulting branch having less than 10 leaves. To better understand why this is useful, think about how a Decision Tree makes predictions, which I outline again within the context of our taxi fare prediction problem. Once done training, it predicts the fare of a taxi ride by passing features of that ride through the tree and finding the end node to which this ride is closest in tree space.

If this is a leaf node, which would be the case if `min_samples_leaf = 1` (the default), the forest is predicting the actual fare of the particular ride in the training set to which this ride happened to be closest. It is almost a certainty that splits toward the ends of the branches aren't capturing actual patterns about the fare of taxi rides in New York City, but just what happened to correspond to a higher or lower fare in the training data. Passing in some larger `min_samples_leaf` means that we now predict the average of some group of samples to which the ride in question is closest in tree space. This technique generalizes noticeably better.

I typically try `min_samples_leaf = [1, 3, 5, 10, 20, 50]`, stopping once increasing `min_samples_leaf` doesn't improve the metric I am interested in.

max_features

`max_features` tells each tree how many features to check when looking for the best split to make. A subtlety here is that passing in `max_features = 15` doesn't mean that each tree picks some subset of 15 features to model. Rather, an individual tree chooses a different random sample of 15 features for each split. Like `min_samples_leaf`, this doesn't allow a tree to fit too closely to the data. More importantly, the trees in the Random Forest are now even less correlated with one another since they weren't even trained on the same data. There has been some practical machine learning research

showing that forests of less accurate, less correlated trees perform better than forests of more accurate, more correlated trees.

A practical example

A common approach to hyperparameter tuning in machine learning is to use a technique implemented in `scikit-learn` called `RandomizedSearchCV`. This takes as required arguments the model itself and the parameter space we are interested in.

```
class sklearn.model_selection.RandomizedSearchCV(estimator,
param_distributions, n_iter=10, scoring=None, fit_params=None,
n_jobs=1, iid=True, refit=True, cv=None, verbose=0,
pre_dispatch='2*n_jobs', random_state=None, error_score='raise',
return_train_score='warn')
```

It randomly tries `n_iter` combinations of these parameters and returns the best hyperparameters of those sampled and the corresponding score. To illustrate the effect of changing individual parameters, however, I show here how my `RandomForestRegressor` performed for some manually chosen combinations. First let's see how well a Random Forest with `n_estimators = 100` with no other arguments passed in performs:

```
m = RandomForestRegressor(n_jobs=-1)
m.fit(X_train, y_train)
print_score(m)
```

rf_tuning_benchmark.py hosted with ❤ by GitHub view raw

```
[4.262844864358172, 4.098490006842908, 0.8090730776697076,
0.8239641822866088]
```

`n_jobs = -1` just tells `scikit-learn` to use all available cores on the computer. Also, `print_score()` is a function from the fast.ai library that returns training error, validation error, training R^2, and validation R^2. Our baseline then is a validation R^2 of ~0.824. Here's what playing with `max_features` does:

```
m = RandomForestRegressor(max_features='log2', n_jobs=-1)
m.fit(X_train, y_train)
```

```
print_score(m)
```

rf_tuning_max_features.py hosted with ❤ by GitHub view raw

```
[4.289944262409593, 3.9900355584905385, 0.8066378726809564,
0.8331574522424692]
```

```
m = RandomForestRegressor(max_features='sqrt', n_jobs=-1)
m.fit(X_train, y_train)
print_score(m)
```

rf_tuning_max_features2.py hosted with ❤ by GitHub view raw

```
[4.322218978148121, 4.136388496420818, 0.8037174696184352,
0.820693545036434]
```

While using `max_features = 'log2'` times the number of features improved performance somewhat, `max_features = 'sqrt'` did the opposite. What works best is likely to vary from case to case, making some form of trial and error the simplest and most popular option. Feel free to play around with some other hyperparameters by yourself. For the sake of brevity, I won't do that here. Finally, after choosing my hyperparameters, I trained one `RandomForestRegressor` using a larger number of estimators to further improve R^2:

```
m = RandomForestRegressor(n_estimators=100, max_features='log2', n_jobs=-1)
m.fit(X_train, y_train)
print_score(m)
```

rf_tuned.py hosted with ❤ by GitHub view raw

```
[4.013807675575337, 3.8493455368979235, 0.830729516598271,
0.8447158691316341]
```

Conclusion

To sum up, the Random Forest employs a number of techniques to reduce variance in predictions while maintaining (to some extent) the low variance that was characteristic of the lone Decision Tree. It does this primarily by averaging together a number of very weakly correlated (if not completely uncorrelated) trees. Hyperparameters like `max_features` and `min_samples_leaf` are among the techniques useful in reducing this correlation between trees, but they often come at the cost of some increase in bias, since each tree now has less data to work with.

Our goal, then, is to choose a set of hyperparameters that nagivates this tradeoff between bias and variance so as to minimize error (or maximize goodness-of-fit) on some new set of data (the validation set) that we believe is representative of what the model might encounter when solving the real world problem it was designed for (the test set).