There are primarily 3 features which can be tuned to improve the predictive power of the model :

1.a. max_features:

These are the maximum number of features Random Forest is allowed to try in individual tree. There are multiple options available in Python to assign maximum features. Here are a few of them :

1. *Auto/None* : This will simply take all the features which make sense in every tree.Here we simply do not put any restrictions on the individual tree.
2. *sqrt* : This option will take square root of the total number of features in individual run. For instance, if the total number of variables are 100, we can only take 10 of them in individual tree."log2" is another similar type of option for max_features.
3. *0.2* : This option allows the random forest to take 20% of variables in individual run. We can assign and value in a format "0.x" where we want x% of features to be considered.

How does "max_features" impact performance and speed?

Increasing max_features generally improves the performance of the model as at each node now we have a higher number of options to be considered. However, this is not necessarily true as this decreases the diversity of individual tree which is the USP of random forest. But, for sure, you decrease the speed of algorithm by increasing the max_features. Hence, you need to strike the right balance and choose the optimal max_features.

1.b. n_estimators :

This is the number of trees you want to build before taking the maximum voting or averages of predictions. Higher number of trees give you better performance but makes your code slower. You should choose as high value as your processor can handle because this makes your predictions stronger and more stable.

1.c. min_sample_leaf :

If you have built a decision tree before, you can appreciate the importance of minimum sample leaf size. Leaf is the end node of a decision tree. A smaller leaf makes the model more prone to capturing noise in train data. Generally I prefer a minimum leaf size of more than 50. However, you should try multiple leaf sizes to find the most optimum for your use case.

2. Features which will make the model training easier

There are a few attributes which have a direct impact on model training speed. Following are the key parameters which you can tune for model speed :

2.a. n_jobs :

This parameter tells the engine how many processors is it allowed to use. A value of "-1" means there is no restriction whereas a value of "1" means it can only use one processor. Here is a simple experiment you can do with Python to check this metric :

```
%timeit

model = RandomForestRegressor(n_estimator = 100, oob_score =

TRUE,n_jobs = 1,random_state =1)

model.fit(X,y)
```

Output ———- 1 loop best of 3 : 1.7 sec per loop

```
%timeit
```

```
model = RandomForestRegressor(n_estimator = 100,oob_score =

TRUE,n_jobs = -1,random_state =1)


model.fit(X,y)
```

Output ———- 1 loop best of 3 : 1.1 sec per loop

“%timeit” is an awsum function which runs a function multiple times and gives the fastest loop run time. This comes out very handy while scalling up a particular function from prototype to final dataset.

2.b. random_state :

This parameter makes a solution easy to replicate. A definite value of random_state will always produce same results if given with same parameters and training data. I have personally found an ensemble with multiple models of different random states and all optimum parameters sometime performs better than individual random state.

2.c. oob_score :

This is a random forest cross validation method. It is very similar to leave one out validation technique, however, this is so much faster. This method simply tags every observation used in different tress. And then it finds out a maximum vote score for every observation based on only trees which did not use this particular observation to train itself.

Here is a single example of using all these parameters in a single function :

```
model = RandomForestRegressor(n_estimator = 100, oob_score =

TRUE, n_jobs = -1,random_state =50,

max_features = "auto", min_samples_leaf = 50)


model.fit(X,y)
```

Learning through a case study

We have referred to Titanic case study in many of our previous articles. Let's try the same problem again. The objective of this case here will be to get a feel of random forest parameter tuning and not getting the right features. Try following code to build a basic model :

```
from sklearn.ensemble import RandomForestRegressor


from sklearn.metrics import roc_auc_score


import pandas as pd


x = pd.read_csv("train.csv")


y = x.pop("Survived")


model =  RandomForestRegressor(n_estimator = 100 , oob_score =

TRUE, random_state = 42)


model.fit(x(numeric_variable,y)
```

```
print "AUC - ROC : ", roc_auc_score(y,model.oob_prediction)
```

AUC – ROC : 0.7386

This is a very simplistic model with no parameter tuning. Now let's do some parameter tuning. As we have discussed before, we have 6 key parameters to tune. We have some grid search algorithm built in Python, which can tune all parameters automatically.But here let's get our hands dirty to understand the mechanism better. Following code will help you tune the model for different leaf size.

Exercise : Try runing the following code and find the optimal leaf size in the comment box.

```
sample_leaf_options = [1,5,10,50,100,200,500]

for leaf_size in sample_leaf_options :

    model = RandomForestRegressor(n_estimator = 200, oob_score =
TRUE, n_jobs = -1,random_state =50,
max_features = "auto", min_samples_leaf = leaf_size)

     model.fit(x(numeric_variable,y)

     print "AUC - ROC : ", roc_auc_score(y,model.oob_prediction)
```