Random Forest:

**Class Imbalance:**

* We should use up sampling or down sampling If there is a class imbalance, impacts the entropy and mse calculations

**Metrics:**

**Hyper Parameters:**

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 :

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.

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.

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

here 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)

**Categorical Features with many values:**

* We should handle by applying some techniques

**Interpretability:**

* No interpreatability

**Train and Run time complexities:**

**After Training Runtime and space complexities:**

**Low latency:**

* Yes. Because of Runtime complexities

**Large Data:**

* Good for large data

**Less Data:**

**Large Dimensions:**

* If number of dimension increases, time taken to train DT will increase
* Dimensionality should be less

**One hot encoding:**

* Should avoid one hot encoding if we have a very large levels (Zip Code)

**Column Standardization/Normalization:**

* Not required (As this is a distance based problem)

**Null Values:**

* It treat as a new level. Hence we have to handle the null values

**Co linearity:**

* Decision trees are by nature immune to multi-collinearity. For example, if you have 2 features which are 99% correlated, when deciding upon a split the tree will choose only one of them. Other models such as Logistic regression would use both the features.

**Multi Class classification:**

* It will be used for multi class classification

**Outliers:**

* Outliers will not impact, tree will become unstable

**Advantages:**

* It works fairly well when we have less data
* Can be easily paralized In Python/R
* It will prevent overfitting, It does this by creating random subsets of the features and building smaller (shallow) trees using the subsets and then it combines the subtrees.

**Disadvantages:**

* It might fail when there are rare predictors as this is a boot strap sampling
* It is a black box model

**Assumptions:**

**Classification and Regression:**