Predicting Housing Prices Using Scikit-Learn's Random Forest Model

Santosh Yadaw Jun 8, 2020 · 13 min read ★

Photo via Getty Images

seller and the buyer as it can aid them in making well informed decision. For sellers, it may help them to determine the average price at which they should put their house for sale while for buyers, it may help them find out the right average price to purchase the house.

Objective

To build a random forest regression model, which is able to predict the median value of houses. We will also briefly walk through some Exploratory Data Analysis, Feature Engineering and Hyperparameter tuning to improve the performance of our Random Forest model.

Our Machine Learning Pipeline

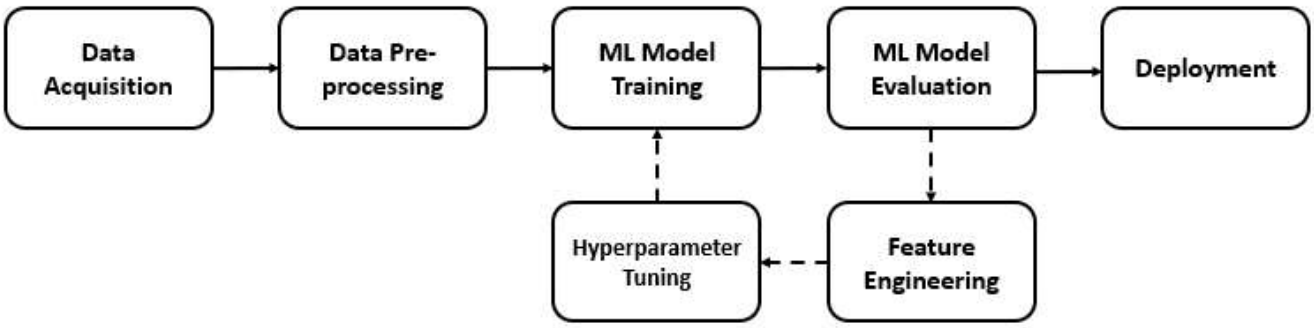


Image by Author: Simple Machine Learning Pipeline

Our Machine Learning Pipeline can be broadly summarized into the following task:

1. Data Acquisition
2. Data Pre-Processing and Exploratory Data Analysis
3. Creating a Base Model
4. Feature Engineering
5. Hyperparameter Tuning
6. Final Model Training and Evaluation

Step 1: Data Acquisition

```
#Importing the necessary libraries we will be using

%load_ext autoreload
%autoreload 2
%matplotlib inline

from fastai.imports import *
from fastai.structured import *

from pandas_summary import DataFrameSummary
from sklearn.ensemble import RandomForestRegressor
from IPython.display import display

from sklearn import metrics
from sklearn.model_selection import RandomizedSearchCV

#Loading the Dataset

PATH = 'data/Boston Housing Dataset/'
df_raw_train = pd.read_csv(f'{PATH}train.csv',low_memory = False)
df_raw_test = pd.read_csv(f'{PATH}test.csv',low_memory = False)
```

Step 2: Data Pre-Processing and Exploratory Data Analysis (EDA)

2.1 Checking and handling if there are any missing data and outliers.

```
df_raw_train.info
```

```
In [7]: 1 df_raw_train.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 406 entries, 0 to 405
Data columns (total 15 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   ID       406 non-null    int64
 1   CRIM     406 non-null    float64
 2   ZN       406 non-null    float64
 3   INDUS    406 non-null    float64
 4   CHAS     406 non-null    int64
 5   NOX      406 non-null    float64
 6   RM       406 non-null    float64
 7   AGE      406 non-null    float64
 8   DIS      406 non-null    float64
 9   RAD      406 non-null    int64
 10  TAX      406 non-null    int64
 11  PTRATIO  406 non-null    float64
```

```
memory usage: 47.7 KB
```

Image by Author

Understanding the raw data:

From the raw training dataset above:

(a) There are **14 variables (13 independent variables — Features and 1 dependent variable — Target Variable)**.
(b) The **data types** are either **integers** or **floats**.
(c) **No categorical data** is present.
(d) There are **no missing values** in our dataset.

2.2 As part of EDA, we will first try to determine the distribution of the dependent variable (MDEV).

```
#Plot the distribution of MEDV

plt.figure(figsize=(10, 6))
sns.distplot(df_raw_train['MEDV'],bins=30)
```

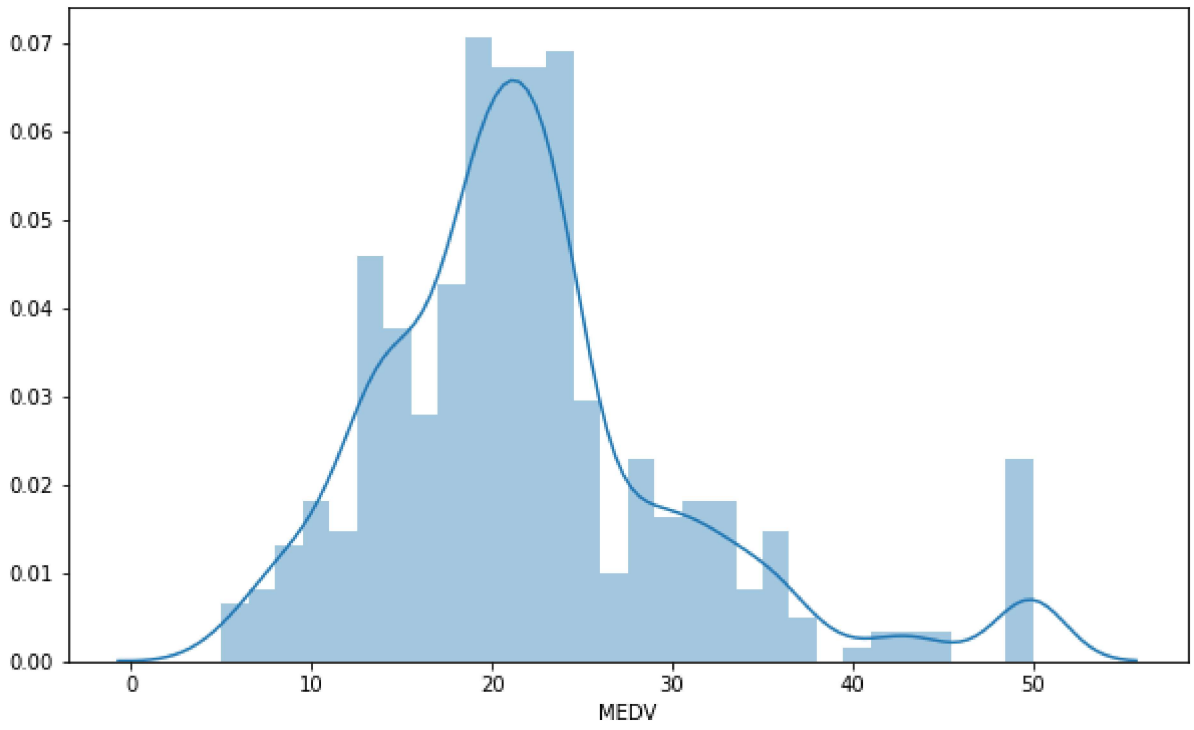


Image by Author: Distribution of MEDV

2.3 Next, try to determine if there are any correlations between:
(i) the independent variables themselves
(ii) the independent variables and dependent variable

To do this, let's do a correlation heatmap.

```
# Plot the correlation heatmap

plt.figure(figsize=(14, 8))
corr_matrix = df_raw_train.corr().round(2)
sns.heatmap(data=corr_matrix,cmap='coolwarm',annot=True)
```

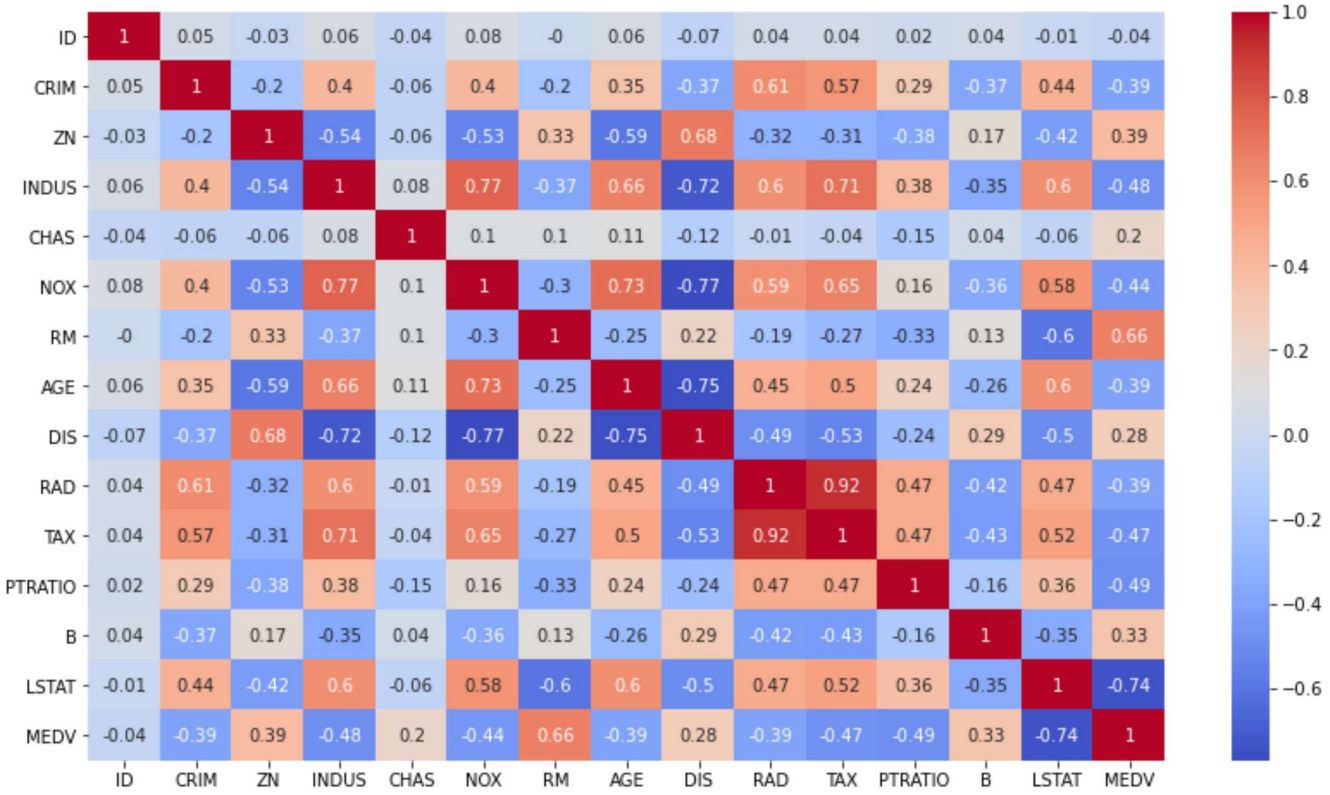


Image by Author: Correlation Heatmap

(i) Correlation between independent variables:

We would need to look out for features of multi-collinearity (i.e. features that are correlated with each other)as this will affect our relationship with the independent variable.

another with a correlation score of around 0.70 (INDUS and TAX, NOX and INDUS, AGE and DIS, AGE and INDUS).

(ii) Correlation between independent variable and dependent variable:
In order for our regression model to perform well, we ideally need to select those features that are highly correlated with our dependent variable (MEDV).

We observe that both **RM** and **LSTAT** are **correlated** with **MEDV** with a correlation score of 0.66 and 0.74 respective. This can also be illustrated via the scatter plot .

```
#Scatter plot to observe the correlations between the features that
are highly correlated with MEDV

target_var = df_raw_train['MEDV']

plot1 = plt.figure(1)
plt.scatter(df_raw_train['RM'],target_var)
plt.xlabel('RM')
plt.ylabel('MEDV')

plot2 = plt.figure(2)
plt.scatter(df_raw_train['LSTAT'],target_var)
plt.xlabel('LSTAT')
plt.ylabel('MEDV')
```

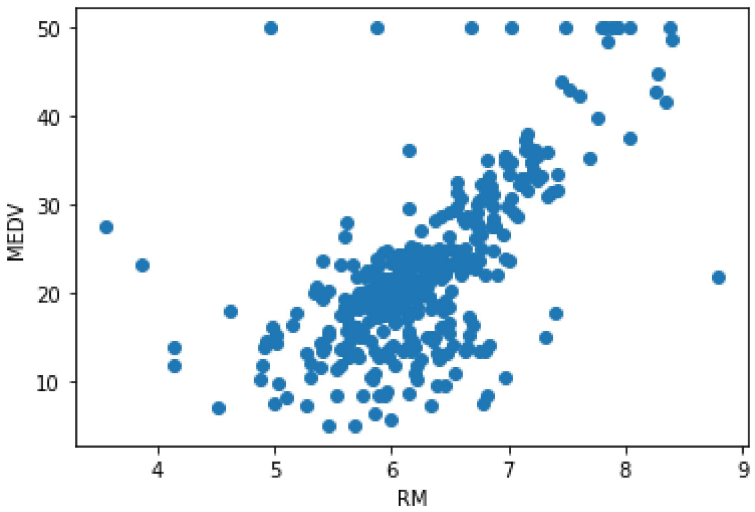


Image by Author: Scatter Plot of RM with MEDV

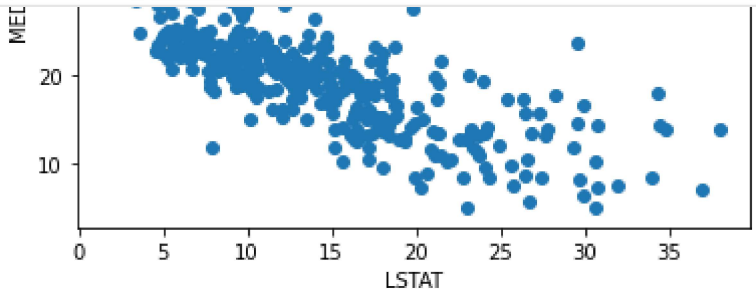


Image by Author: Scatter Plot of RM with LSTAT

From the scatter plot above:
i) **MEDV increases linearly with the RM.** This makes sense as we would expect the median price of a house would be **more generally more expensive** as the **number of rooms increases**.
ii) **MEDV decreases linearly with LSTAT**. This also makes sense since we can expect the median price of a house would generally be **less expensive at places of lower status**.

Step 3: Creating a base model

Hold up! Before we create our base Random Forest Model, it very important to pick a suitable evaluation metric.

Image taken from imgflip.com

3.1 Picking the right evaluation metric

Picking the right evaluation metric will help us to evaluate whether our model's performance is good. For regression problems, the go-to evaluation metric is either **Root Mean Square Error(RMSE)** or **Root Mean Square Logistic Error (RMLSE)**.

RMSE: It is a measure of the squared difference between the prediction from our model and the actual value.

Root Mean Square Error formula

where y': predicted value, y: actual value

RMSLE: It is a measure of the squared difference between the log of the prediction from our model and the log of the actual value.

$$RMLSE(y) = \sqrt{\sum_{i=1}^{n} (\log(y' + 1) - \log(y + 1))^2}$$

Root Mean Square Logistic Error

where y': predicted value, y: actual value

The RMSLE might be a better evaluation metric as (1) it is robust enough to deal with outliers, which we saw is present in our dataset (2) RMSLE incurs a larger penalty for underestimation of the actual value. If we put ourselves in the sellers perspective, we do not want to underestimate the price as it would result in losses. However, for this project we shall not take anyone's side and shall choose the **RMSE** as the **evaluation metric** as we would be using the Random Forest model which is immune to outliers.

3.2 Creating our base Random Forest Model

The next step would be to create a base model first without any feature engineering and hyperparameter tuning. We would use this model's performance as a benchmark for comparison later on after we have done feature engineering and hyperparameter tuning later.

The machine learning model we have chosen is the Random Forest Regression Model and here is why:

1. Random Forests models **require minimal data preparation**. It is able to easily hand categorical, numerical and binary features without scaling or normalization required.
2. Random Forests models can help us in performing implicit feature selections as they provide **good indicators of the important features**.

models which perform much better after being normalized.

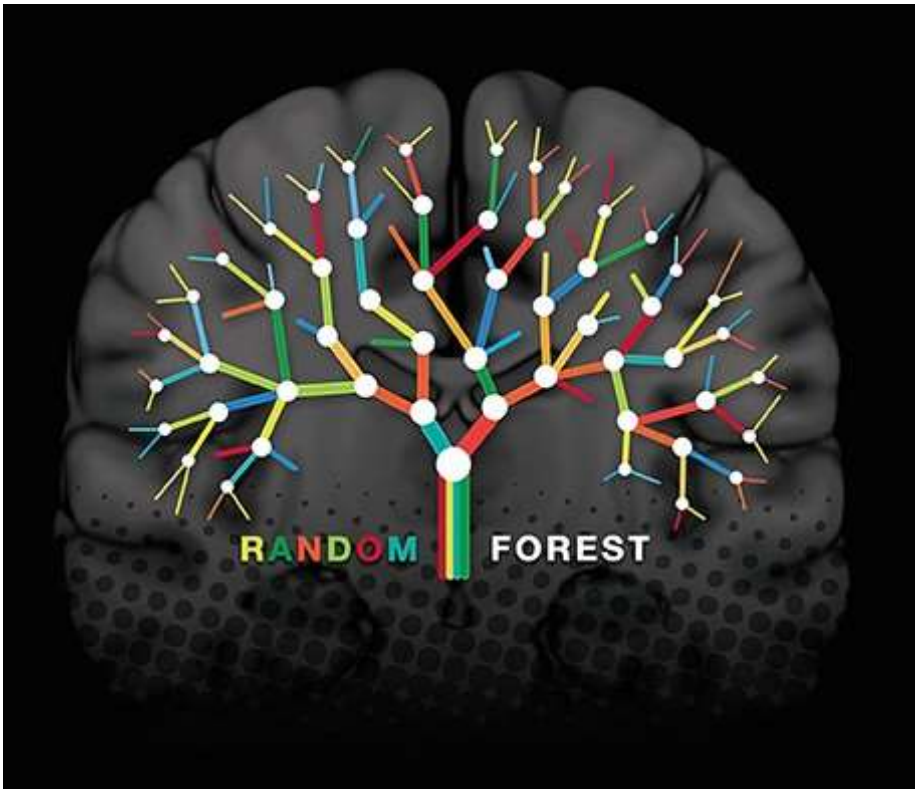


Photo taken from frontiersin.org

We would like to create some helpful functions :

```
# A function to split our training data into a training set to train
our  model and a validations set, which will be used to validate our
model.

def split_vals(a,n):
    return a[:n],a[n:]

# Functions that will help us calculate the RMSE and print the
score.

def rmse(x,y):
    return math.sqrt(((x-y)**2).mean())

def print_score(m):
    res =
[rmse(m.predict(X_train),y_train),rmse(m.predict(X_valid),y_valid),m
.score(X_train,y_train),m.score(X_valid,y_valid)]
    if hasattr(m,'oob_score_'):res.append(m.oob_score_)
    print(res)
```

```
n_valid = 100
n_train = len(df_raw_train)-n_valid
X_train,X_valid =
split_vals(df_raw_train.drop('MEDV',axis=1),n_train)
y_train,y_valid = split_vals(df_raw_train['MEDV'],n_train)
X_test = df_raw_test
```

Creating and Fitting our Random Forest Model w**ithout feature selection and hyperparameter tuning**.

```
In [14]:
1 m = RandomForestRegressor(n_jobs=-1,oob_score=True)
2 m.fit(X_train,y_train)
3 print_score(m)

[1.3946434360529185, 3.0218688588355374, 0.9765849452413979, 0.8547978744183
689, 0.8342340188939908]
```

From our base Random Forest model, we already get a very decent result with the **training RMSE** to be **1.394** while the **validation RMSE** is **3.021**. However, notice that our model seems to be **ovefitting** since the validation RMSE is around 3 times higher than the training RMSE.

Hence, the **baseline score to beat** is **validation RMSE 3.021**!

Step 4: Feature Engineering

Feature Engineering and Hyperparameter tuning are **essential steps** within a Machine Learning Pipeline.

4.1 Determining the important features

The features in our data are directly influencing our Random Forest model and the result it achieves (i.e. the better the features we prepare and choose, the better final result we will achieve!). Hence we are going to explore and fine tune our Random Forest model by determining which of the features it had deemed to be important in our base model earlier.

```
def feat_importance(m,df_train):
    importance = m.feature_importances_
    importance =
```

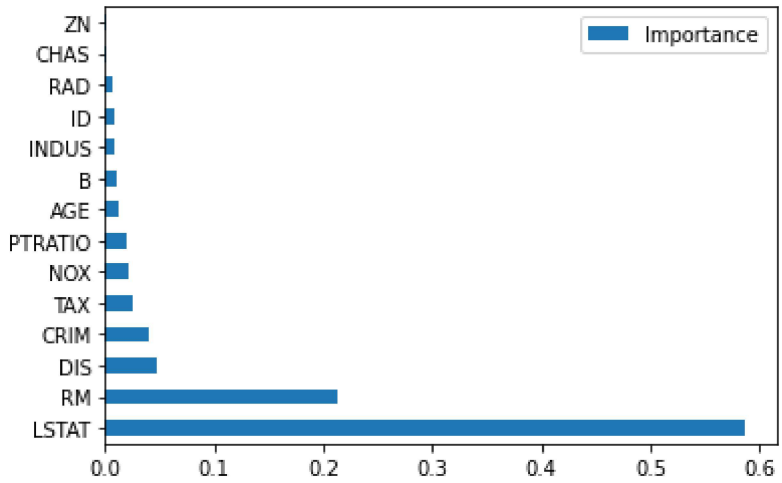
```
importance = feat_importance(m,X_train)
importance[:]
```

Out[16]:

	Importance
LSTAT	0.587072
RM	0.212774
DIS	0.047266
CRIM	0.039221
TAX	0.025601
NOX	0.020579
PTRATIO	0.019630
AGE	0.012799
B	0.009339
INDUS	0.008916
ID	0.007798
RAD	0.006564
CHAS	0.001484
ZN	0.000955

Ranked Feature Importance Coefficients

```
importance.plot(kind='barh')
```



Bar Plot of Ranked Feature Importance

correlated with the MDEV.

4.2 Discarding non-important features

The next step will be to try and discard features with importance coefficient less than 0.01 and use it to model our Random Forest again to see if there is an improvement in our prediction results.

```
#Discarding features with feature coefficients less than 0.01

to_keep = importance[importance['Importance'] > 0.01].index
df_raw_train_keep = df_raw_train[to_keep].copy()
df_raw_test_keep = df_raw_test[to_keep].copy()

#Splitting data into training and validation set

X_train,X_valid = split_vals(df_raw_train_keep,n_train)

# Fitting our Random Forest Model after discarding the less
important features.
```

```
In [21]:
1 m = RandomForestRegressor(n_jobs=-1,oob_score=True)
2 m.fit(X_train,y_train)
3 print_score(m)

[1.3908989299490224, 2.9445053591392885, 0.9767105115463636, 0.8621374060185
167, 0.8401838253352027]
```

Our Random Forest Model performs **slightly better** after removing some of the redundant features (i.e. 6 features!). We have obtain the top 9 most important features in determining the dependent variable MEDV (LSTAT, RM, DIS,CRIM, TAX, NOX, PTRATIO,NOX and AGE).

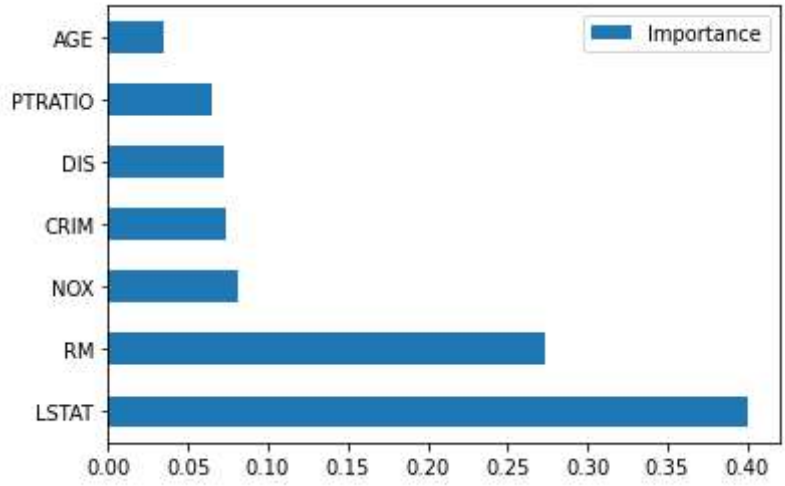
Next, let's see if there have been any changes to the rank of the features which are found to be important to our Random Forest Model.

```
def feat_importance(m,df_raw_train_keep):
    importance = m.feature_importances_
    importance =
pd.DataFrame(importance,index=df_train.columns,columns=
["Importance"])
    return importance.sort_values(by=['Importance'],ascending=False)
```

Out[48]:

	Importance
LSTAT	0.400693
RM	0.272831
NOX	0.080857
CRIM	0.073526
DIS	0.072469
PTRATIO	0.064878
AGE	0.034745

Ranked Feature Importance Coefficients after removing of redundant features



Bar Plot of Ranked Feature Importance after removing redundant features

We observe that the most important features after removing the redundant features previously are still LSTAT and RM. We would like to explore how dropping each of the remaining features one by one would affect our overall score.

In [52]:
```
feats = ['LSTAT','RM','DIS','NOX','PTRATIO','CRIM','AGE']
m = RandomForestRegressor(n_jobs=-1,oob_score=True)
m.fit(X_train,y_train)
print_score(m)
```

```
[1.4185045471646536, 3.1188934287660417, 0.9757768695453068, 0.8453240406556
793, 0.8285318036503034]
```

In [53]:
```
for f in feats:
    df_subs = df_raw_train_keep.drop(f,axis=1)
    X_train,X_valid =split_vals(df_subs,n_train)
    m = RandomForestRegressor(n_jobs=-1,oob_score=True)
    m.fit(X_train,y_train)
    print(f)
```

```
[1.805623910420536?, 3.32787512536152?3, 0.960751469798295?, 0.8239014456718
035, 0.6908574729986297]
RM
[1.5675250954407727, 3.952312881086211, 0.9704200246092192, 0.75161556164623
69, 0.7875551279800503]
DIS
[1.4621917216011568, 3.2395115943610993, 0.9742618430421569, 0.8331290167664
831, 0.8269028689415385]
NOX
[1.5008497092224788, 3.2441961731683233, 0.9728829017705362, 0.8326460516884
883, 0.8219839866244096]
PTRATIO
[1.4379716151625086, 3.0952967192177225, 0.9751074475876751, 0.8476556604442
624, 0.8161589049421294]
CRIM
[1.4274427520771256, 3.1864395679190265, 0.9754706407936581, 0.8385518323912
685, 0.8288521212609287]
AGE
[1.3739555938443608, 3.0659043445613237, 0.9772744607851648, 0.8505351917480
173, 0.8261463201965449]
```

Notice removing RM, LSTAT, DIS and CRIM results in a worse validation RMSE while removing AGE and PTRATIO gives us a slightly better score. Hence, we will **further remove AGE** and **PTRATIO** from the dataset before doing running the final model.

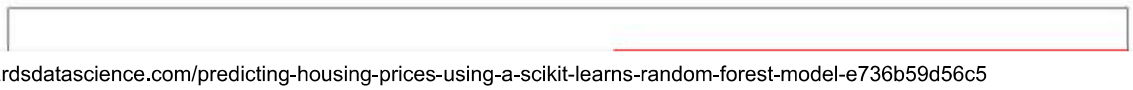
4.3 Removing Multi collinearity between independent features

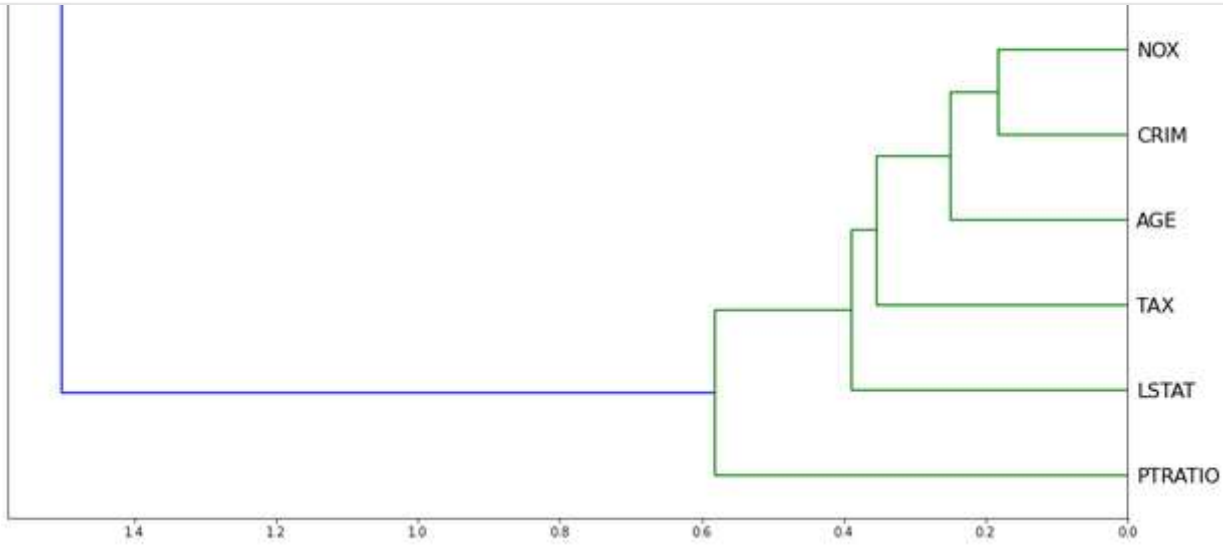
In 2.3(i), there are some features which are correlated to each other. To improve the model's performance, we would ideally like to **remove any multi collinearity** between the features.

To see how the features are correlated, we can plot a dendrogram diagram.

```
#Dendogram plot

from scipy.cluster import hierarchy as hc
corr =
np.round(scipy.stats.spearmanr(df_raw_train_keep).correlation,4)
corr_condensed = hc.distance.squareform(1-corr)
z = hc.linkage(corr_condensed,method='average')
fig = plt.figure(figsize=(16,10))
dendogram =
hc.dendrogram(z,labels=df_raw_train_keep.columns,orientation='left',
leaf_font_size=16)
plt.show()
```





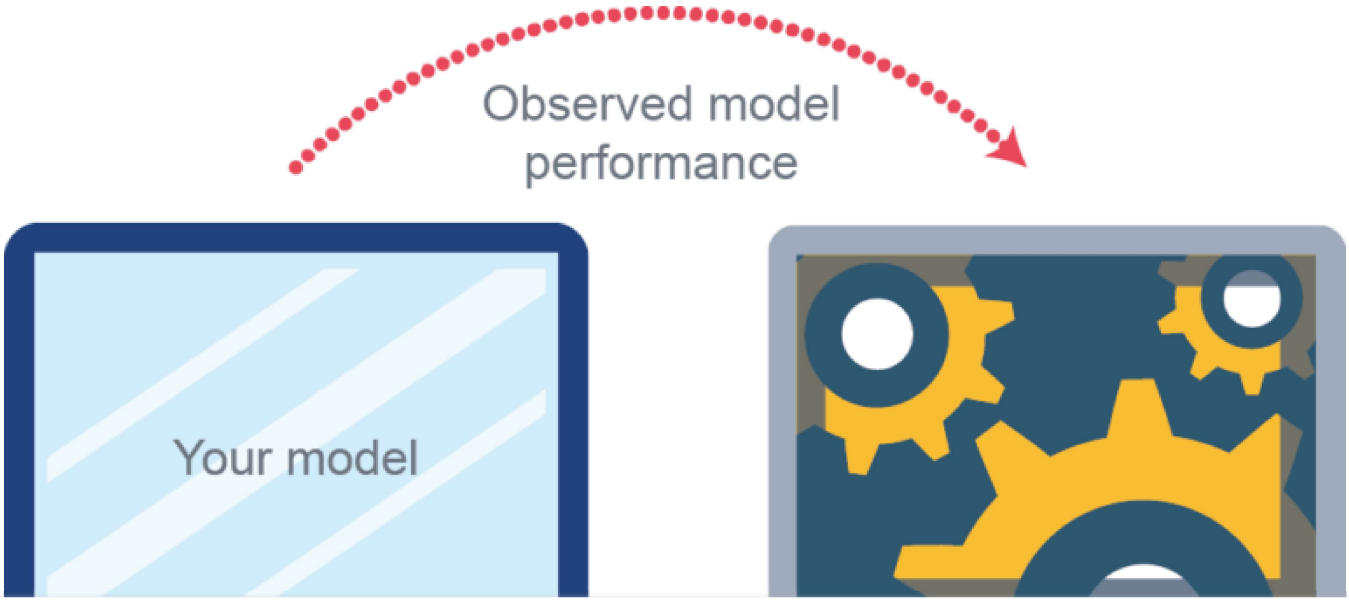
Dendrogram Plot between the important features

From the dendrogram plot, **in terms of ranking** we can see that none of the features are measuring the same thing. The closest one we have in terms of ranking is NOX and CRIM.

If there were features which were very close to each other in terms of ranking, the next step would have been to remove these features, one at a time, and see if our model could be simplified further without impacting our validation RMSE score. However for this case, we don't need to.

Step 5: Hyperparameter Tuning

We have come to the final step (hooray!) before we can build our final Random Forest Model.



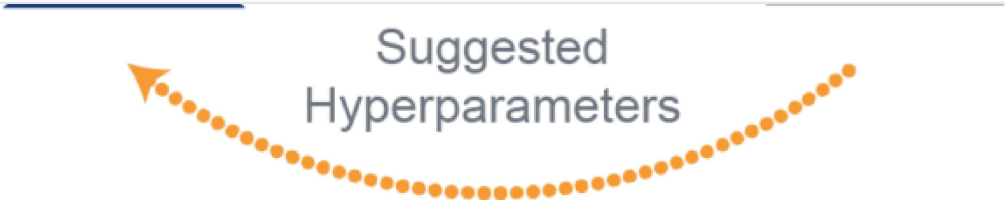


Image taken from tech-quantum.com

Hyperparameter tuning is an **iterative process** whereby we **select** the **best configurations** of the hyperparameters that give us the **best model performance output**. For Random Forest model, we will focus on tuning 3 hyperparameters:

(1) n_estimators : Number of trees to create and generalize over our Random Forest. Likely, the more trees we create, the better as it would enable us to sample more of the dataset and help us to generalize better. However, there would come a point whereby increasing the number of trees will only result in very small changes to our validation RMSE at the cost of computational power.

(2) min_samples_leaf : This indicates the number of samples that will be in our leaf node. Each time we double the min_sample_leaf, we are removing one layer from the tree and halving the number of leaf nodes. Hence, the result of increasing min_samples_leaf is that each of our leaf nodes will have more than one sample inside them so when we calculate the average on that leaf node, it would be more stable although we will get a little less depth and have smaller number of nodes.Hence, though each tree will not be less predictive and less correlated, our model should be able to generalize better and prevent overfitting.

(3) max_features : This indicates how many of the features to consider at each split.

To optimize and search for the best hyperparameters, we will be using the **Randomized Grid Search method**!

```
n_estimators = [int(x) for x in np.arange(start = 10, stop = 2000,
step = 10)]
max_features = [0.5,'auto', 'sqrt','log2']
min_samples_leaf = [1, 2, 4]
bootstrap = [True, False]
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
```

```
m_random = RandomizedSearchCV(estimator = m, param_distributions =
random_grid, n_iter = 100, cv = 3, verbose=2, random_state=42,
n_jobs = -1)
m_random.fit(X_train, y_train)
m_random.best_params_
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 6 concurrent workers.
[Parallel(n_jobs=-1)]: Done  29 tasks      | elapsed:    7.8s
[Parallel(n_jobs=-1)]: Done 150 tasks      | elapsed:   27.6s
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed:   53.5s finished
```

Out[27]:
```
{'n_estimators': 640,
 'min_samples_leaf': 1,
 'max_features': 0.5,
 'bootstrap': False}
```

From our Randomized Grid Search, we found that the most optimized hyperparameters for our Random Forest Model to be those above.

Step 6: Final Model

The last step — building our final mode. To do so, we are going to drop the AGE and PTRATIO feature as discussed earlier.

Image taken from gurutzeunzalu.blogspot.com

In [40]:
```
df_sub = df_raw_train_keep.drop(['AGE','PTRATIO'],axis=1)
X_train,X_valid = split_vals(df_sub,n_train)
m = RandomForestRegressor(n_estimators=640,min_samples_leaf=1,max_features=
```

```
[1.2802023276444934, 2.8471628275790706, 0.9802700480104705, 0.8711019469962
523, 0.8593855257402043]
```

We have obtained a **validation RMSE score of 2.847**, which is better than our original base model **validation RMSE score of 3.021**!

In addition, the **final validation R^2 score** is **0.87** and this is better than the **validation R^2 score** of the **base model** of **0.85**. The R^2 score tells us how much our model is able to explain the variation in the dataset. Having a score of 0.87 indicates that our model can explain 87% of the variation in the dataset.

Hence, we see the simple final model even with **less input features performs much better!**

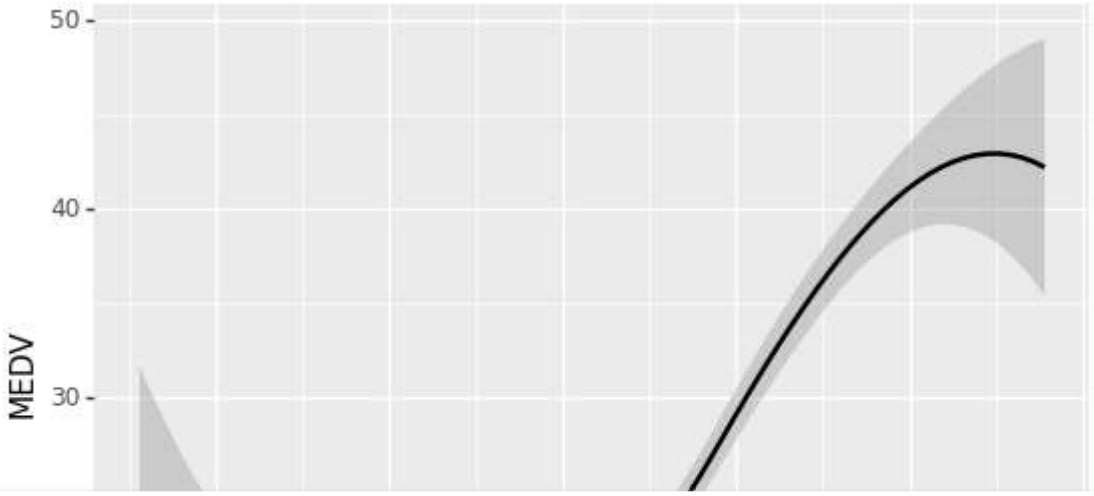
BONUS Step: Partial Independence

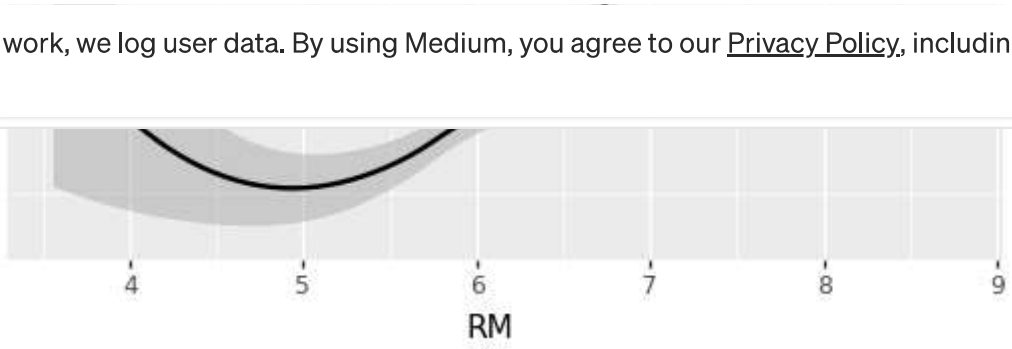
While we have created a much better model already, let's take a step back and explore our results further.

Previously, we observed there is a **linear relationship between the median price of houses increases (MDEV) and number of rooms (RM)**.

Let's try to do ggplot.

```
from pdpbox import pdp
from plotnine import *
ggplot(df_raw_train, aes('RM', 'MEDV'))+stat_smooth(se=True,
method='loess')
```





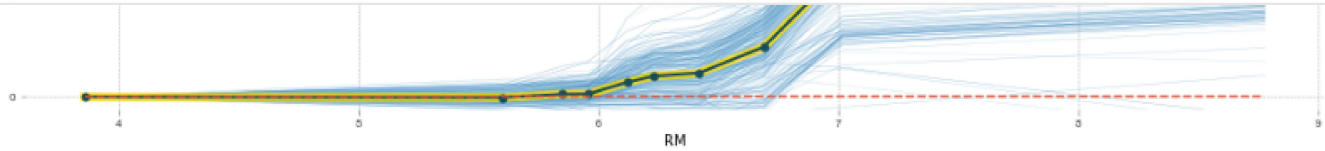
ggplot

Based on the ggplot above, we observe that the relationship between the RM and MEDV is **not** what we expected. For instance, one would generally expect the price of the house to increase with the number of rooms. However, we see in a drop in price between 4 rooms and 5 rooms, likewise a price drop between 8 rooms to 9 rooms.

The issue here is we are looking at **univariate relationships** and there are **a lot of interactions** between the features that are **being lost** in univariate plots. For instance, why did the price drop for houses that are 5 rooms as compared to 4 rooms and the price of a 6 room house is almost similar to a 4 room house?

Hence, to find the true relationship between RM and MEDV, we would need to do a **partial independence plot** (i.e. assuming that all other features being equal, how does the price of the house vary with the number of rooms) to see the true relationship.

```
def plot_pdp(feat,clusters=None,feat_name=None):
    feat_name = feat_name or feat
    p = pdp.pdp_isolate(m,X_train,feat)
    return pdp.pdp_plot(p,feat_name,plot_lines=True,cluster=clusters
is not None,n_cluster_centers = clusters)
plot_pdp('RM')
```

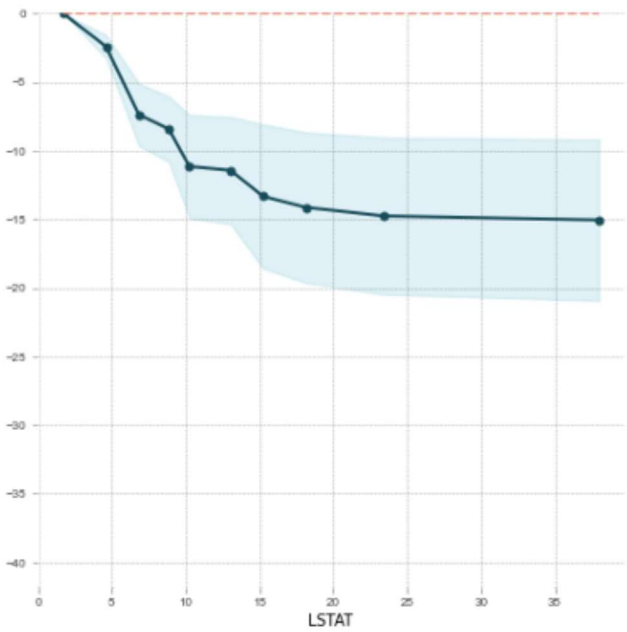


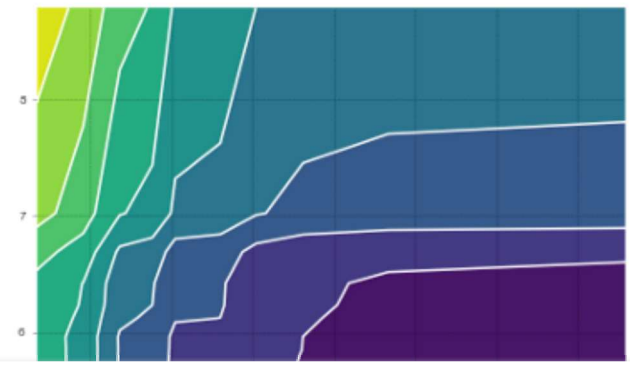
PDP plot

Notice from the Partial Dependence Plot (PDP) above, after removing all other externalities we observe that the relationship between the RM and MEDV is almost a straight line (i.e. roughly linear), which is what we would expect. The yellow line represents the average MEDV of all the transactions.

Now let's explore how LSTAT and RM influences the median housing prices.

```
feats = ['LSTAT','RM']
p = pdp.pdp_interact(m,X_train,feats)
pdp.pdp_interact_plot(p,feats)
```





Hence, looking at the PDP plots, we can safely conclude that the **number of rooms (RM) influences the median housing prices (MEDV) linearly**. The s**tatus of the population (LSTAT) influences MEDV inversely**.

Final Thoughts and Next Steps

We have built a Random Forest model which has **validation RMSE score of 3.021** and **validation R^2 score of 0.87**.

We have also determined from our Random Forest model the **key features** that **affects** the **median housing prices (MEDV)** in Boston are **(1) LSAT** : Percentage of the lower population status **(2) RM**: The average number of rooms per dwelling **(3) NOX**: Concentration of Nitrogen Oxide **(4) CRIM**: The crime rate per capita by town.

The final model we have built is far from perfect. For instance, observe that the **training R^2 score** in our final model is **much higher 0.98** than our **validation R^2 score** of **0.87**. This indicates the final model is able to explain 98% of the variations in the training data while only explain 87% of the variation in the validation data (i.e. the final model is still **overfitting** the training data and **does not generalize as well** to the validation data).

One simple way to address overfitting could be to try to **increase the size of our dataset to train our model on.** Currently, we only have 406 entries in our dataset which is **simply insufficient**.

Congratulations and thank you for making it to the end. I sincerely hope you have learnt something new! Happy learning!😀

Random Forest Boston Housing Price Prediction Sklearn Towards Data Science