# Data Pre-processing and Model Building

November 2, 2019

## 1 Data Gathering

#### 1.0.1 Loading Modules

```
In [1]: import pandas as pd
        from pandas import DataFrame as df
        import numpy as np
        import warnings # Ignores any warning
        warnings.filterwarnings("ignore")
```

#### 1.0.2 Loading Dataset

```
In [2]: train = pd.read_csv("Train.csv")
        test = pd.read_csv("Test.csv")
```

```
In [3]: train['Item_Outlet_Sales']=train['Item_Outlet_Sales'].transform(func='sqrt')
```

```
In [4]: data = pd.concat([train, test],ignore_index=True)
```

## 2 Data Preprocessing

#### 2.0.1 Handling Missing Data

```
In [5]: #Checking count of null values
        data.isnull().sum()
```

```
Out[5]: Item_Fat_Content                0
        Item_Identifier                 0
        Item_MRP                        0
        Item_Outlet_Sales            5681
        Item_Type                       0
        Item_Visibility                 0
        Item_Weight                  2439
        Outlet_Establishment_Year       0
        Outlet_Identifier               0
        Outlet_Location_Type            0
        Outlet_Size                  4016
        Outlet_Type                     0
        dtype: int64
```

Mode Imputation technique is applied for Categorical attrributes whereas Mean Imputation technique is applied for Numeric attributes

```
In [6]: #Item_Weight
        print ('Orignal #missing: %d'%sum(data['Item_Weight'].isnull()))
        data['Item_Weight'].fillna(value=data['Item_Weight'].mean(),inplace=True)
        print ('Final #missing: %d'%sum(data['Item_Weight'].isnull()))
```

```
Orignal #missing: 2439
Final #missing: 0
```

```
In [7]: data.groupby('Outlet_Identifier').Outlet_Size.value_counts(dropna=False)
```

```
Out[7]: Outlet_Identifier  Outlet_Size
        OUT010             NaN             925
        OUT013             High           1553
        OUT017             NaN            1543
        OUT018             Medium         1546
        OUT019             Small           880
        OUT027             Medium         1559
        OUT035             Small          1550
        OUT045             NaN            1548
        OUT046             Small          1550
        OUT049             Medium         1550
        Name: Outlet_Size, dtype: int64
```

```
In [8]: data.groupby('Outlet_Type').Outlet_Size.value_counts(dropna=False)
```

```
Out[8]: Outlet_Type        Outlet_Size
        Grocery Store      NaN             925
                           Small           880
        Supermarket Type1  Small          3100
                           NaN            3091
                           High           1553
                           Medium         1550
        Supermarket Type2  Medium         1546
        Supermarket Type3  Medium         1559
        Name: Outlet_Size, dtype: int64
```

We see that only OUT010, OUT017, OUT045 HAS NA values and grocery store and supermarket Type 1 has only Small as the outlet size, so we can replace the nan with small

```
In [9]: data.loc[data.Outlet_Identifier.isin(['OUT010','OUT017','OUT045']),
                  'Outlet_Size'] = 'Small'
```

```
In [10]: data.Outlet_Size.value_counts()
```

```
Out[10]: Small     7996
         Medium    4655
         High      1553
         Name: Outlet_Size, dtype: int64
```

### 2.0.2 Handling Outliers

Item Visibility has 0 in most cases, this is an outlier that can be replaced by the mean of Item visibility of each category of Item Identifier.

In [11]:
```
#Count of '0's
count=0
for i in range(len(data)):
    if(data['Item_Visibility'][i]==0):
        count= count+1
print(count)
```

```
879
```

In [12]:
```
data.loc[data.Item_Visibility == 0, 'Item_Visibility'] = np.nan

#aggregate by Item_Identifier
IV_mean = data.groupby('Item_Identifier').Item_Visibility.mean()
IV_mean.head()
```

Out[12]:
```
Item_Identifier
DRA12    0.044920
DRA24    0.045646
DRA59    0.148204
DRB01    0.091127
DRB13    0.007648
Name: Item_Visibility, dtype: float64
```

In [13]:
```
data.Item_Visibility.fillna(0, inplace=True)

for index, row in data.iterrows():
    if(row.Item_Visibility == 0):
        data.loc[index, 'Item_Visibility'] = IV_mean[row.Item_Identifier]

data.Item_Visibility.describe()
#See that min value is not zero anymore
```

Out[13]:
```
count    14204.000000
mean         0.070458
std          0.050086
min          0.003575
25%          0.031381
50%          0.058064
75%          0.098042
max          0.328391
Name: Item_Visibility, dtype: float64
```

Creating a broad category of Type of Item into Food, Non-Consumable and Drinks

```
In [14]: data['Item_Type_Combined'] = data['Item_Identifier'].apply(lambda x: x[0:2])
         data['Item_Type_Combined'] = data['Item_Type_Combined'].map({'FD':'Food',
                                                                      'NC':'Non-Consumable',
                                                                      'DR':'Drinks'})
         data['Item_Type_Combined'].value_counts()
```

```
Out[14]: Food              10201
         Non-Consumable     2686
         Drinks             1317
         Name: Item_Type_Combined, dtype: int64
```

Making data manipulation for Fat Content to 'Low Fat' and 'Regular'. Even there are items that are Non Consumable, so 'Non Edible' is also added.

```
In [15]: #Change categories of low fat:
         print ('Original Categories:')
         print (data['Item_Fat_Content'].value_counts())

         print ('\nModified Categories:')
         data['Item_Fat_Content'] = data['Item_Fat_Content'].replace({'LF':'Low Fat',
                                                                      'reg':'Regular',
                                                                      'low fat':'Low Fat'})
         print (data['Item_Fat_Content'].value_counts())
```

```
Original Categories:
Low Fat    8485
Regular    4824
LF          522
reg         195
low fat     178
Name: Item_Fat_Content, dtype: int64

Modified Categories:
Low Fat    9185
Regular    5019
Name: Item_Fat_Content, dtype: int64
```

```
In [16]: #Mark non-consumables as separate category in low_fat:
         data.loc[data['Item_Type_Combined']=="Non-Consumable",
                  'Item_Fat_Content'] = "Non-Edible"
         data['Item_Fat_Content'].value_counts()
```

```
Out[16]: Low Fat       6499
         Regular       5019
         Non-Edible    2686
         Name: Item_Fat_Content, dtype: int64
```

```
In [17]: #data.head()
```

### 2.0.3 Label Encoding

Label encoding of categorical variables are done after which One hot encoding is applied on these transformed variables.

```
In [18]: from sklearn.preprocessing import LabelEncoder
         le = LabelEncoder()

         data['Outlet'] = le.fit_transform(data['Outlet_Identifier'])
         var_mod = ['Item_Fat_Content','Outlet_Location_Type','Outlet_Size',
                    'Item_Type_Combined','Outlet_Type','Outlet']

         le = LabelEncoder()
         for i in var_mod:
             data[i] = le.fit_transform(data[i])
```

```
In [19]: data = pd.get_dummies(data, columns=['Item_Fat_Content',
                                'Outlet_Location_Type','Outlet_Size','Outlet_Type',
                                              'Item_Type_Combined','Outlet'])
```

# 3 Feature Engineering

Remember the data is from 2013. So to take outlet's year in to account, it is subtracted from 2013

```
In [20]: data['Outlet_Years'] = 2013 - data['Outlet_Establishment_Year']
         data['Outlet_Years'].describe()
```

```
Out[20]: count    14204.000000
         mean        15.169319
         std          8.371664
         min          4.000000
         25%          9.000000
         50%         14.000000
         75%         26.000000
         max         28.000000
         Name: Outlet_Years, dtype: float64
```

Attributes such as Item Identifier, Outlet Identifier and Item Type are dropped from the data since new attributes have been created from these.

```
In [21]: data.drop('Item_Identifier', axis=1,inplace=True)
         data.drop('Outlet_Identifier', axis=1,inplace=True)
         data.drop('Item_Type', axis=1,inplace=True)
```

```
In [22]: #data.head()
```

# 4 Model Building

### 4.0.1 Pre Model Building Tasks

Training and testing sets will be created with 75%-25% ratio split.

In [23]:
```
old = data[:8522]
new = data[8523:]
```

In [24]:
```
column = data.columns
x = old[column]
x.drop('Item_Outlet_Sales', axis=1,inplace=True)
y = old['Item_Outlet_Sales']
#df(y.head())
```

In [25]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
                                                    test_size=0.25, random_state=42)
```

In [26]:
```
x_train.head()
```

Out[26]:
```
       Item_MRP  Item_Visibility  Item_Weight  Outlet_Establishment_Year  \
6864  214.7192         0.030155    14.000000                       2002
7006  262.1278         0.048738     9.895000                       1999
5325  222.0114         0.030144    12.792854                       1985
2867  178.5318         0.044230    12.792854                       1985
4376   97.2726         0.063851     6.905000                       1998

      Item_Fat_Content_0  Item_Fat_Content_1  Item_Fat_Content_2  \
6864                   1                   0                   0
7006                   0                   0                   1
5325                   0                   0                   1
2867                   0                   0                   1
4376                   0                   0                   1

      Outlet_Location_Type_0  Outlet_Location_Type_1  Outlet_Location_Type_2  \
6864                       0                       1                       0
7006                       1                       0                       0
5325                       0                       0                       1
2867                       1                       0                       0
4376                       0                       0                       1

      ...  Outlet_1  Outlet_2  Outlet_3  Outlet_4  Outlet_5  Outlet_6  \
6864  ...         0         0         0         0         0         0
7006  ...         0         0         0         0         0         0
5325  ...         0         0         0         0         1         0
2867  ...         0         0         0         1         0         0
4376  ...         0         0         0         0         0         0
```

```
      Outlet_7  Outlet_8  Outlet_9  Outlet_Years
6864         1         0         0            11
7006         0         0         1            14
5325         0         0         0            28
2867         0         0         0            28
4376         0         0         0            15

[5 rows x 31 columns]
```

Before choosing a particular machine learning algorithm it is trained using different algorithms like Linear Regression, Lasso Regression, Decision Tree Regression and Random Forest Regression.

### 4.0.2 Linear Regression

```
In [27]: from sklearn.linear_model import LinearRegression, Ridge

In [28]: #Training the model
         lr = LinearRegression(normalize=True)
         lr.fit(x_train, y_train)

Out[28]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=True)

In [29]: # Predict training set
         y_pred = lr.predict(x_test)

         #Evaluating the model
         from sklearn.metrics import mean_absolute_error, r2_score
         print('MSE : ',mean_absolute_error(y_test, y_pred))
         print('R2 : ',r2_score(y_test, y_pred))

MSE :  8.41228844392911
R2 :  0.6508406847623198
```

Mean Square Error is 8.41 for Linear Regression and Co-efficient of determination is 56.2%

### 4.0.3 Ridge Regression

```
In [30]: rr = Ridge(alpha=0.05, normalize=True, random_state=42)
         rr.fit(x_train, y_train)

Out[30]: Ridge(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=None,
            normalize=True, random_state=42, solver='auto', tol=0.001)

In [31]: y_pred_r = rr.predict(x_test)
         print('MSE : ',mean_absolute_error(y_test, y_pred_r))
         print('R2 : ',r2_score(y_test, y_pred_r))

MSE :  8.40949871102736
R2 :  0.6508941272616803
```

#### 4.0.4 Decision Tree

```
In [32]: from sklearn.tree import DecisionTreeRegressor

In [33]: dr = DecisionTreeRegressor(random_state=42,
                                    max_depth=5, min_samples_leaf=73)
         dr.fit(x_train, y_train)

Out[33]: DecisionTreeRegressor(criterion='mse', max_depth=5, max_features=None,
                    max_leaf_nodes=None, min_impurity_decrease=0.0,
                    min_impurity_split=None, min_samples_leaf=73,
                    min_samples_split=2, min_weight_fraction_leaf=0.0,
                    presort=False, random_state=42, splitter='best')

In [34]: y_pred_d = dr.predict(x_test)
         print('MSE : ',mean_absolute_error(y_test, y_pred_d))
         print('R2 : ',r2_score(y_test, y_pred_d))

MSE :  8.059848505988153
R2 :  0.6658139225765828
```

#### 4.0.5 Random Forest

```
In [35]: from sklearn.ensemble import RandomForestRegressor

In [36]: rf = RandomForestRegressor(n_estimators=8,
                                    max_depth=5, random_state=42)
         rf.fit(x_train, y_train)

Out[36]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=5,
                    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=8, n_jobs=None,
                    oob_score=False, random_state=42, verbose=0, warm_start=False)

In [37]: y_pred_f = rf.predict(x_test)
         print('MSE : ',mean_absolute_error(y_test, y_pred_f))
         print('R2 : ',r2_score(y_test, y_pred_f))

MSE :  7.992545238240991
R2 :  0.6693265181610324
```

Random Forest Regression performs better than others with R2 score of 66.9%. Parameter Tuning is done to check if any improvement cwn be done in the score.

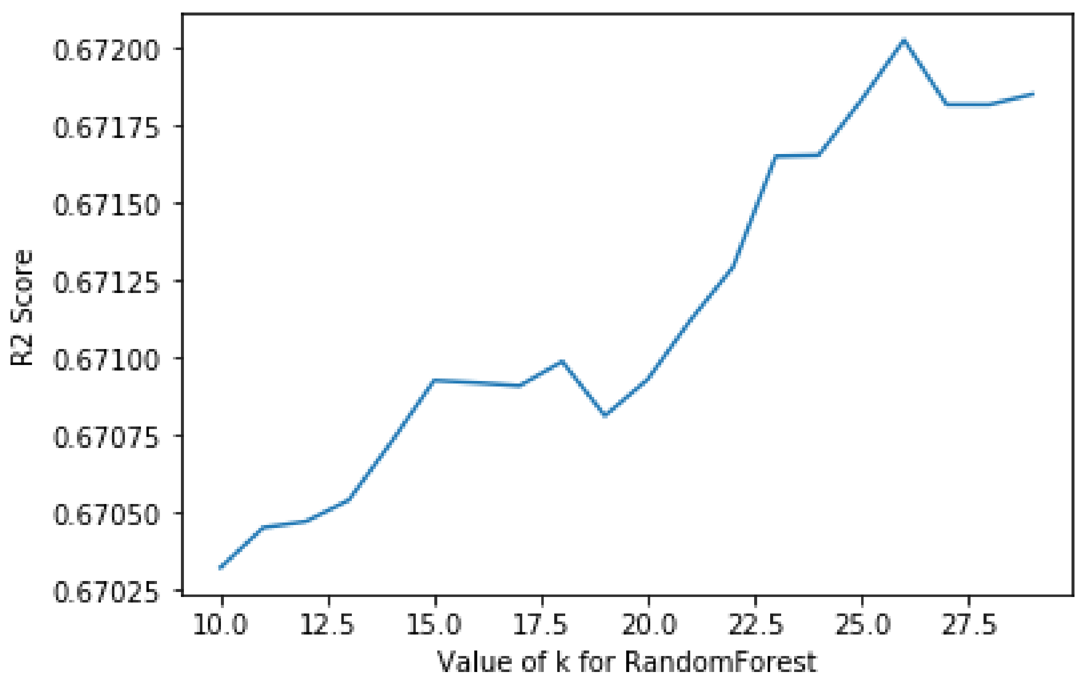
#### 4.0.6 Parameter Tuning

In [38]:
```
d_range = range(10, 30)
dscores = []

for k in d_range:
    dt = RandomForestRegressor(n_estimators=k, max_depth=5, random_state=42)
    dt.fit(x_train, y_train)
    y_pred = dt.predict(x_test)
    dscores.append(r2_score(y_test, y_pred))
print(np.array(dscores).max())
```

```
0.6720251381201396
```

In [39]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.plot(d_range, dscores)
plt.xlabel('Value of k for RandomForest')
plt.ylabel('R2 Score')
```

Out[39]:
```
Text(0, 0.5, 'R2 Score')
```




There's an improvement in the R2 score of 1.3%.

### 4.0.7 Prediction for new data

The Random Forest Regression is used with the optimal parameters to predict Output Sales for new data.

In [40]:
```
newt = new
newt.drop('Item_Outlet_Sales', axis=1, inplace=True)
```

In [41]:
```
rf = RandomForestRegressor(n_estimators=26,
                           max_depth=5, random_state=42)
rf.fit(x_train, y_train)
```

Out[41]:
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=5,
           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=26, n_jobs=None,
           oob_score=False, random_state=42, verbose=0, warm_start=False)
```

In [42]:
```
new['Item_Outlet_Sales'] = (rf.predict(newt))**2
new['Item_Outlet_Sales'].head()
```

Out[42]:
```
8523    1549.518385
8524    1340.570837
8525     603.002299
8526    2374.357841
8527    5896.627593
Name: Item_Outlet_Sales, dtype: float64
```