# BUSINESS IMPACT



**JUN 18, 2019**

# PREDICTING SALES WITH THE AID OF PANDAS

POSTED BY MEGAN QUINN

Pandas is an open-source Python package that provides users with high-performing and flexible data structures. These structures are designed to make analyzing relational or labeled data both easy and intuitive. Pandas is one of the most popular and quintessential tools leveraged by data scientists when developing a machine learning model. The most crucial step in the machine learning process is not simply fitting a model to a given data set. Most of the model development process takes place in the pre-processing and data exploration phase. An accurate model requires good predictors and, in order to acquire them, the user must understand the raw data. Through Pandas' numerous data wrangling and analysis tools, this important step can easily be achieved. The goal of this blog is to highlight some of the central and most commonly used tools in Pandas while illustrating their significance in model development. The data set used for this demo consists of a supermarket chain's sales across multiple stores in a variety of cities. The sales data is broken down by items within the stores. The goal is to predict a certain item's sale.

## Reading the Data

When starting a new Python script, modules required for the analysis, including Pandas, must be imported into the environment:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

## Latest Posts

- Predicting Sales with the Aid of Pandas
- Consider 4 Factors to Amplify Health Analytics Efforts
- Meet Our Team: Leo Furlong
- Knowledge Mining Showcase: Azure Search
- BlueGranite Joins Developers and Industry Experts at Microsoft Build 2019
- BlueGranite is a Great Place to Work in 2019 – Here's Why!
- Create a Dynamic Title in Power BI (Updated)

in the current working directory folder for the Python environment.

```
StoreSales_df=pd.read_csv('StoreSales.csv')
```

Once the data frame is created, there are a variety of viewing and inspecting tools available in order to achieve a better understanding of the raw data. The df.head(n) and df.tail(n) functions allow users to examine the first and last rows respectively:

```
StoreSales_df.head(5)
StoreSales_df.tail(5)
```

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Outlet_Location |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | FDA15 | 9.30 | Low Fat | 0.016047 | Dairy | 249.8092 | OUT049 | 1999 | Medium | |
| **1** | DRC01 | 5.92 | Regular | 0.019278 | Soft Drinks | 48.2692 | OUT018 | 2009 | Medium | |
| **2** | FDN15 | 17.50 | Low Fat | 0.016760 | Meat | 141.6180 | OUT049 | 1999 | Medium | |
| **3** | FDX07 | 19.20 | Regular | 0.000000 | Fruits and Vegetables | 182.0950 | OUT010 | 1998 | NaN | |
| **4** | NCD19 | 8.93 | Low Fat | 0.000000 | Household | 53.8614 | OUT013 | 1987 | High | |

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Outlet_Locat |
|---|---|---|---|---|---|---|---|---|---|---|
| **8518** | FDF22 | 6.865 | Low Fat | 0.056783 | Snack Foods | 214.5218 | OUT013 | 1987 | High | |
| **8519** | FDS36 | 8.380 | Regular | 0.046982 | Baking Goods | 108.1570 | OUT045 | 2002 | NaN | |
| **8520** | NCJ29 | 10.600 | Low Fat | 0.035186 | Health and Hygiene | 85.1224 | OUT035 | 2004 | Small | |
| **8521** | FDN46 | 7.210 | Regular | 0.145221 | Snack Foods | 103.1332 | OUT018 | 2009 | Medium | |
| **8522** | DRG01 | 14.800 | Low Fat | 0.044878 | Soft Drinks | 75.4670 | OUT046 | 1997 | Small | |

The df.shape and df.info provide information about the number of rows and columns in a data frame, the data types, and missing data:

```
StoreSales_df.shape
```

```
(8523, 12)
```

```
StoreSales_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8523 entries, 0 to 8522
Data columns (total 12 columns):
Item_Identifier              8523 non-null object
Item_Weight                  7060 non-null float64
Item_Fat_Content             8523 non-null object
Item_Visibility              8523 non-null float64
Item_Type                    8523 non-null object
Item_MRP                     8523 non-null float64
Outlet_Identifier            8523 non-null object
Outlet_Establishment_Year    8523 non-null int64
Outlet_Size                  6113 non-null object
Outlet_Location_Type         8523 non-null object
Outlet_Type                  8523 non-null object
Item_Outlet_Sales            8523 non-null float64
dtypes: float64(4), int64(1), object(7)
memory usage: 799.1+ KB
```

```
min(StoreSales_df.Outlet_Establishment_Year)
max(StoreSales_df.Outlet_Establishment_Year)
```

# Data Cleaning and Feature Engineering

After getting a general overview and understanding of the data, the next step toward successful model development is cleaning the data and creating new, possibly more influential variables from the existing raw data.

The variable *Item_Identifier* follows a labeling pattern of letters per each product (i.e., 'FD' for food, 'DR' for drinks and 'NC' for n) followed by a three-digit code. It may be more useful to have a group type variable with just these two letters rather than the entire code. To achieve this, the .map() function applies a selection of only the first two values in the item identifier and returns as a new column labeled *Item_Group_Type*.

In [12]:
```
StoreSales_df['Item_Group_Type']=StoreSales_df.Item_Identifier.map(lambda x: x[:2])
StoreSales_df[["Item_Identifier","Item_Group_Type"]].head(5)
```

Out[12]:

| | Item_Identifier | Item_Group_Type |
|---|---|---|
| 0 | FDA15 | FD |
| 1 | DRC01 | DR |
| 2 | FDN15 | FD |
| 3 | FDX07 | FD |
| 4 | NCD19 | NC |

In order to analyze the outlet establishment year as a numerical variable, a new column entitled *Outlet_Age* can be calculated by subtracting the outlet's year by the the max year value of the dataset plus one (assuming this data's collection ended the prior year). The *Outlet_Establishment_Year* variable can then be dropped using the Pandas df.drop() function.

```
StoreSales_df["Outlet_Age"]=(max(StoreSales_df.Outlet_Establishment_Year)+1)-StoreSales_df["Outlet_Establishment_Year"]
StoreSales_df=StoreSales_df.drop(columns="Outlet_Establishment_Year")
StoreSales_df.head()
```

Out[13]:

| Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Size | Outlet_Location_Type | Outlet_Type | Item_Outlet_Sales | Item_Group_Type | Outlet_Age |
|---|---|---|---|---|---|---|---|---|---|---|
| Low Fat | 0.016047 | Dairy | 249.8092 | OUT049 | Medium | Tier 1 | Supermarket Type1 | 3735.1380 | FD | 11 |
| Regular | 0.019278 | Soft Drinks | 48.2692 | OUT018 | Medium | Tier 3 | Supermarket Type2 | 443.4228 | DR | 1 |
| Low Fat | 0.016760 | Meat | 141.6180 | OUT049 | Medium | Tier 1 | Supermarket Type1 | 2097.2700 | FD | 11 |
| Regular | 0.000000 | Fruits and Vegetables | 182.0950 | OUT010 | NaN | Tier 3 | Grocery Store | 732.3800 | FD | 12 |
| Low Fat | 0.000000 | Household | 53.8614 | OUT013 | High | Tier 3 | Supermarket Type1 | 994.7052 | NC | 23 |

Since most machine learning models in Python will return errors if null values exist within the data, identifying their existence and rectifying the issue is a crucial step. The code below counts the number of null values for each column in the store sales data frame:

```
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Size               2410
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
Item_Group_Type              0
Outlet_Age                   0
dtype: int64
```

There are a variety of methods to address null values within a data set. Some common approaches include simply removing the rows containing the null values, forward or backward filling of values for timeseries data, or replacing the null value with a calculated value. This data set contains a significant number of null values. Removing the entire subject could lead to a lack of complete data, therefore filling in the missing values is a more appropriate method.

Since the *Item_Weight* variable is numeric, replacing a null value with the item's average weight is a logical approach. The first line of the code uses Pandas df.groupby() function combined with the .agg function to find the mean weight of each unique item and store the results in another Pandas data frame. In lines two and three, setting the index of the original data frame to match the index of the new *Item_Identifier_Mean* data frame allows the null values to be easily imputed with their matching mean values. It is then necessary to check to see if this method resolved all the null values. Line 5 of the code reveals that four rows still contain null values.

```
In [8]: Item_Identifier_Mean=StoreSales_df.groupby("Item_Identifier").agg({'Item_Weight': 'mean'})

        StoreSales_df2=StoreSales_df.set_index("Item_Identifier")
        StoreSales_df2.Item_Weight.fillna(Item_Identifier_Mean.Item_Weight, inplace= True)
        StoreSales_df2=StoreSales_df2.reset_index()

        StoreSales_df2[pd.isnull(StoreSales_df2.Item_Weight)]
```

Out[8]:

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Size | Outlet_Location_Type | Outlet_Type | Ite |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 927 | FDN52 | NaN | Regular | 0.130933 | Frozen Foods | 86.9198 | OUT027 | Medium | Tier 3 | Supermarket Type3 | |
| 1922 | FDK57 | NaN | Low Fat | 0.079904 | Snack Foods | 120.0440 | OUT027 | Medium | Tier 3 | Supermarket Type3 | |
| 4187 | FDE52 | NaN | Regular | 0.029742 | Dairy | 88.9514 | OUT027 | Medium | Tier 3 | Supermarket Type3 | |
| 5022 | FDQ60 | NaN | Regular | 0.191501 | Baking Goods | 121.2098 | OUT019 | Small | Tier 1 | Grocery Store | |

To investigate the cause of this persisting issue, the new mean imputed data frame is merged on *Item_Idenitifier* with the original data frame using Pandas df.merge(). The merge reveals that the items with null values in the new data frame only appeared once in the original data and had no weight information, therefore a mean could not be calculated.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | FDE52 | NaN | Regular | 0.029742 | Dairy | 88.9514 | OUT027 | Medium | Tier 3 | |
| 3 | FDQ60 | NaN | Regular | 0.191501 | Baking Goods | 121.2098 | OUT019 | Small | Tier 1 | C |

4 rows × 25 columns

This same method is then applied to the *Outlet_Size* variable, except the mode of the outlet type is used as the imputed value since *Outlet_Size* is categorical. To calculate the mode, the outlet types are converted into a Pandas structure called a series, then the mode is applied to each of these series using the .apply function. From there, a similar merge as before can take place. However, since the merge is on a column name instead of the index, both *Outlet_Size* columns are retained in the new dataset with "_x" and "_y" appended to the names. Only one of these columns is needed, therefore renaming to remove the "_x" and dropping the *Outlet_Size_y* column is conducted in line 4.

In [10]:
```
Outlet_Identifier_Mode=pd.DataFrame(StoreSales_df2.groupby("Outlet_Type")['Outlet_Size'].apply(pd.Series.mode))

StoreSales_df3=StoreSales_df2.merge(Outlet_Identifier_Mode,on='Outlet_Type')
StoreSales_df3['Outlet_Size_x'].fillna(StoreSales_df3.Outlet_Size_y,inplace=True)
StoreSales_df3=StoreSales_df3.rename(columns={'Outlet_Size_x': 'Outlet_Size'}).drop(columns=['Outlet_Size_y'])
StoreSales_df3
```

Out[10]:

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Size | Outlet_Location_Type | Outlet_Type | I |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | FDA15 | 9.300 | Low Fat | 0.016047 | Dairy | 249.8092 | OUT049 | Medium | Tier 1 | Supermarket Type1 | |
| 1 | FDN15 | 17.500 | Low Fat | 0.016760 | Meat | 141.6180 | OUT049 | Medium | Tier 1 | Supermarket Type1 | |
| 2 | NCD19 | 8.930 | Low Fat | 0.000000 | Household | 53.8614 | OUT013 | High | Tier 3 | Supermarket Type1 | |
| 3 | FDO10 | 13.650 | Regular | 0.012741 | Snack Foods | 57.6588 | OUT013 | High | Tier 3 | Supermarket Type1 | |
| 4 | FDH17 | 16.200 | Regular | 0.016687 | Frozen Foods | 96.9726 | OUT045 | Small | Tier 2 | Supermarket Type1 | |
| 5 | FDU28 | 19.200 | Regular | 0.094450 | Frozen Foods | 187.8214 | OUT017 | Small | Tier 2 | Supermarket Type1 | |
| 6 | FDY07 | 11.800 | Low Fat | 0.000000 | Fruits and Vegetables | 45.5402 | OUT049 | Medium | Tier 1 | Supermarket Type1 | |

Examining the new cleaned data shows that Outlet_Type no longer contains null values.

In [11]:
```
pd.isnull(StoreSales_df3.Outlet_Type).sum()
```

Out[11]: 0

Returning to the null *Item_Weight* values, since there were four items with only one record each, these values can simply be dropped. Checking the final cleaned data set reveals that all null values have been corrected:

```
Item_MRP                0
Outlet_Identifier       0
Outlet_Size             0
Outlet_Location_Type    0
Outlet_Type             0
Item_Outlet_Sales       0
Item_Group_Type         0
Outlet_Age              0
dtype: int64
```

Dividing the features into categorical and numerical sets allows for further examination of any other possible incongruities in the data. Using the df.select_dtypes() function, two data frames are created containing only the specified data types:

```
In [13]: categorical_features = StoreSales_Cleaned.select_dtypes(include=[np.object])
         numerical_features = StoreSales_Cleaned.select_dtypes(include=[np.number])

         categorical_features.dtypes
         numerical_features.dtypes

Out[13]: Item_Identifier         object
         Item_Fat_Content        object
         Item_Type               object
         Outlet_Identifier       object
         Outlet_Size             object
         Outlet_Location_Type    object
         Outlet_Type             object
         Item_Group_Type         object
         dtype: object

Out[13]: Item_Weight          float64
         Item_Visibility      float64
         Item_MRP             float64
         Item_Outlet_Sales    float64
         Outlet_Age             int64
         dtype: object
```

Through the use of a for loop and the .unique() function, the number of unique values for each categorical feature and the label can be displayed. As is fairly common with categorical variables recorded from different sources, the labeling technique of *Item_Fat_Content* is inconsistent. “Low Fat” is represented as both “low fat” and “LF” while “Regular” is also recorded as “reg”. This type of discrepancy will cause issues when creating dummy variables.

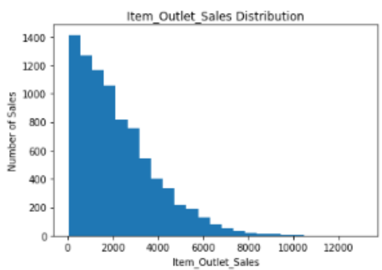
```
In [14]: for name in list(categorical_features):
             print (name + ":")
             print ("    Count of unique values:" + str(len(categorical_features[name].unique())))
             print (categorical_features[name].unique())

         Item_Identifier:
             Count of unique values:1555
         ['FDA15' 'FDN15' 'NCD19' ... 'FDC23' 'FDR07' 'FDP15']
         Item_Fat_Content:
             Count of unique values:5
         ['Low Fat' 'Regular' 'low fat' 'reg' 'LF']
         Item_Type:
             Count of unique values:16
         ['Dairy' 'Meat' 'Household' 'Snack Foods' 'Frozen Foods'
          'Fruits and Vegetables' 'Breakfast' 'Hard Drinks' 'Breads' 'Soft Drinks'
          'Health and Hygiene' 'Canned' 'Baking Goods' 'Starchy Foods' 'Others'
          'Seafood']
         Outlet_Identifier:
             Count of unique values:10
         ['OUT049' 'OUT013' 'OUT045' 'OUT017' 'OUT046' 'OUT035' 'OUT018' 'OUT010'
          'OUT019' 'OUT027']
         Outlet_Size:
             Count of unique values:3
         ['Medium' 'High' 'Small']
         Outlet_Location_Type:
             Count of unique values:3
         ['Tier 1' 'Tier 3' 'Tier 2']
         Outlet_Type:
             Count of unique values:4
         ['Supermarket Type1' 'Supermarket Type2' 'Grocery Store'
          'Supermarket Type3']
         Item_Group_Type:
             Count of unique values:3
         ['FD' 'NC' 'DR']
```

```
Out[15]: array(['Low Fat', 'Regular'], dtype=object)
```

# Data Exploration

With the data cleaning and feature engineering completed, an even closer examination of the data and its relations can be conducted using Pandas in conjunction with the Matplotlib library. The histogram below shows that the target variable, *Item_Outlet_Sales*, is right skewed:

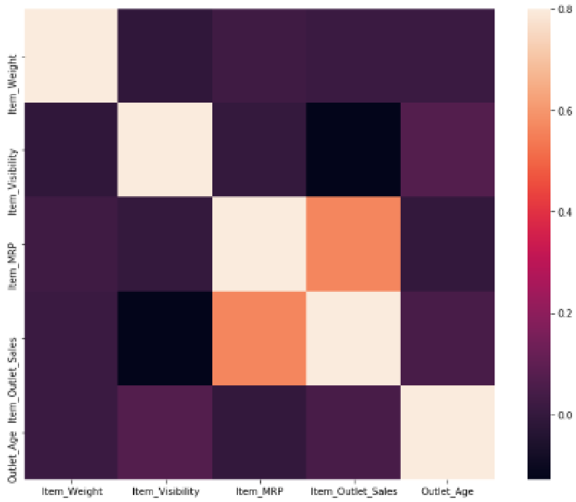
```
In [16]: plt.hist(StoreSales_Cleaned.Item_Outlet_Sales, bins=25)
         plt.xlabel("Item_Outlet_Sales")
         plt.ylabel("Number of Sales")
         plt.title("Item_Outlet_Sales Distribution")

Out[16]: (array([1.414e+03, 1.267e+03, 1.166e+03, 1.058e+03, 8.210e+02, 7.550e+02,
                 5.440e+02, 4.050e+02, 3.350e+02, 2.150e+02, 1.880e+02, 1.280e+02,
                 8.000e+01, 5.600e+01, 3.100e+01, 1.900e+01, 1.100e+01, 1.200e+01,
                 6.000e+00, 4.000e+00, 1.000e+00, 1.000e+00, 0.000e+00, 1.000e+00,
                 1.000e+00]),
          array([   33.29    ,   555.436992,  1077.583984,  1599.730976,
                  2121.877968,  2644.02496 ,  3166.171952,  3688.318944,
                  4210.465936,  4732.612928,  5254.75992 ,  5776.906912,
                  6299.053904,  6821.200896,  7343.347888,  7865.49488 ,
                  8387.641872,  8909.788864,  9431.935856,  9954.082848,
                 10476.22984 , 10998.376832, 11520.523824, 12042.670816,
                 12564.817808, 13086.9648  ]),
          <a list of 25 Patch objects>)

Out[16]: Text(0.5,0,'Item_Outlet_Sales')

Out[16]: Text(0,0.5,'Number of Sales')

Out[16]: Text(0.5,1,'Item_Outlet_Sales Distribution')
```




Applying the .corr() function to the numerical features data frame created earlier, provides a symmetrical data frame of the variables' correlations to each other. Plotting a heatmap of this correlation data frame highlights that Item_MRP has the strongest, positive correlation with the target variable while Item_Visibility has a negative correlation.

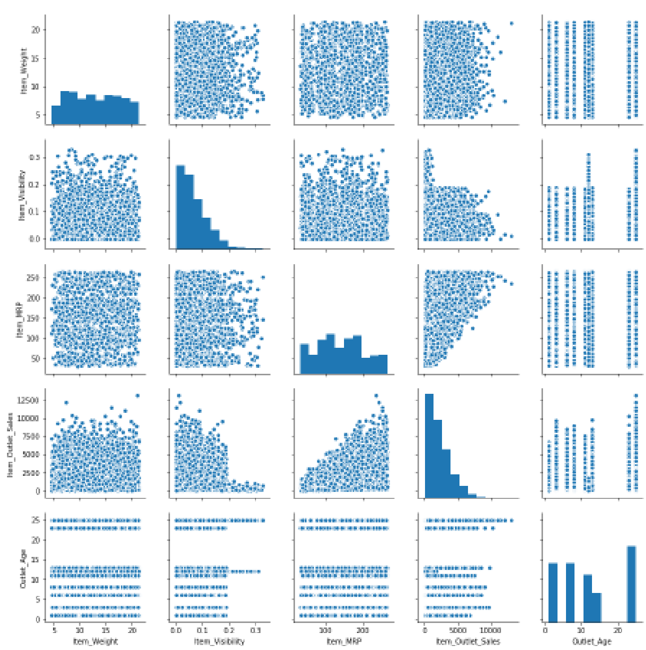
| | | | | | |
|---|---|---|---|---|---|
| Item_MRP | 0.025975 | -0.001155 | 1.000000 | 0.567803 | -0.004599 |
| Item_Outlet_Sales | 0.013168 | -0.128297 | 0.567803 | 1.000000 | 0.049083 |
| Outlet_Age | 0.013426 | 0.074325 | -0.004599 | 0.049083 | 1.000000 |

```
In [18]: f, ax = plt.subplots(figsize=(12, 9))
         sns.heatmap(corr, vmax=.8, square=True)

Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x1fa501d2ba8>
```




Using the .pairplot() function allows for visualization of relationships among the all the numeric variables at once:

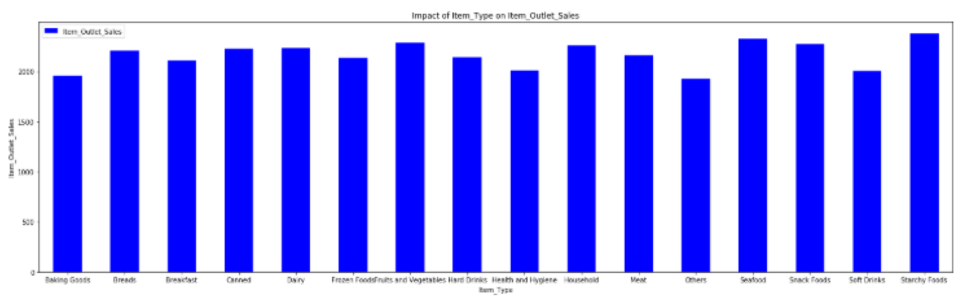
```
In [62]: sns.pairplot(StoreSales_Cleaned)

Out[62]: <seaborn.axisgrid.PairGrid at 0x1fa527c5c88>
```



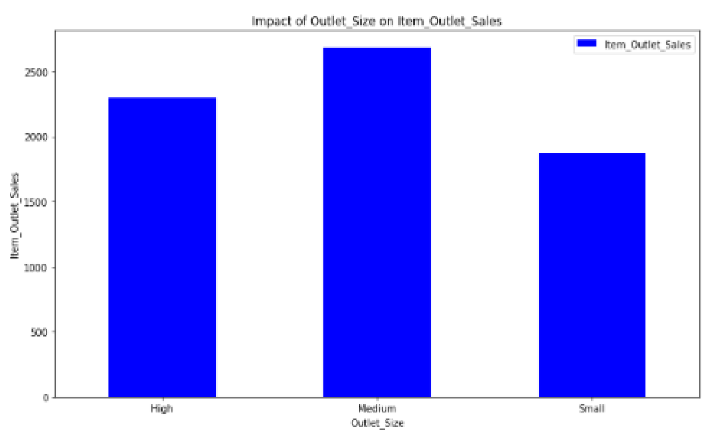

The categorical variables' relationships with the target variable can also be examined through the use of df.pivot_table(). This function operates like pivot tables in Excel by creating an index and applying an aggregation function over a specified value. Plotting these sales.

.

```
In [19]: Item_Type_pivot = StoreSales_Cleaned.pivot_table(index='Item_Type', values="Item_Outlet_Sales", aggfunc=np.mean)

         Item_Type_pivot.plot(kind='bar', color='blue',figsize=(25,7))
         plt.xlabel("Item_Type")
         plt.ylabel("Item_Outlet_Sales")
         plt.title("Impact of Item_Type on Item_Outlet_Sales")
         plt.xticks(rotation=0)
         plt.show()

Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1fa50296550>
Out[19]: Text(0.5,0,'Item_Type')
Out[19]: Text(0,0.5,'Item_Outlet_Sales')
Out[19]: Text(0.5,1,'Impact of Item_Type on Item_Outlet_Sales')
Out[19]: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15]),
          <a list of 16 Text xticklabel objects>)
```




```
In [20]: Outlet_Size_pivot = StoreSales_Cleaned.pivot_table(index='Outlet_Size', values="Item_Outlet_Sales", aggfunc=np.mean)

         Outlet_Size_pivot.plot(kind='bar', color='blue',figsize=(12,7))
         plt.xlabel("Outlet_Size")
         plt.ylabel("Item_Outlet_Sales")
         plt.title("Impact of Outlet_Size on Item_Outlet_Sales")
         plt.xticks(rotation=0)
         plt.show()

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1fa501bf048>
Out[20]: Text(0.5,0,'Outlet_Size')
Out[20]: Text(0,0.5,'Item_Outlet_Sales')
Out[20]: Text(0.5,1,'Impact of Outlet_Size on Item_Outlet_Sales')
Out[20]: (array([0, 1, 2]), <a list of 3 Text xticklabel objects>)
```




This data analysis and exploration will aid in guiding feature, model, and parameter selection during the model development phase.

Finally, the creation of dummy variables is required in order to use the categorical variables in the modeling process. Pandas has an easy to use function, pd.get_dummies(), that

```
In [22]: StoreSales_Dummy = pd.get_dummies(StoreSales_Cleaned, columns =['Item_Fat_Content','Outlet_Identifier','Outlet_Location_Type','Out
         StoreSales_Dummy
```

| n_Fat_Content_Low Fat | Item_Fat_Content_Regular | Outlet_Identifier_OUT010 | ... | Outlet_Size_High | Outlet_Size_Medium | Outlet_Size_Small | Outlet_Type_Grocery Store | Outlet_Type_S |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | ... | 0 | 1 | 0 | 0 | |
| 1 | 0 | 0 | ... | 0 | 1 | 0 | 0 | |
| 1 | 0 | 0 | ... | 1 | 0 | 0 | 0 | |
| 0 | 1 | 0 | ... | 1 | 0 | 0 | 0 | |
| 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | |
| 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | |
| 1 | 0 | 0 | ... | 0 | 1 | 0 | 0 | |
| 0 | 1 | 0 | ... | 0 | 0 | 1 | 0 | |

# Model Development

With pre-processing, cleaning, and data exploration complete, the final phase of modeling can now take place. Sklearn is a commonly used machine learning library in Python that contains multiple modeling and evaluation tools. The first step is to enable the train_test_split() function of this package to divide the cleaned data frame into two separate data frames. The larger data frame, which will represent 85% of the entire data, will be used to train the model, while the remaining 15% will be used to evaluate and determine whether the model is appropriate.

```
In [26]: from sklearn.model_selection import train_test_split

         train , test = train_test_split(StoreSales_Dummy,test_size=.15)

         len(train)
         len(test)

Out[26]: 7241

Out[26]: 1278
```

Next, the train and test data frames are each divided into two separate data frames, one containing the desired predictors and the other containing the target variable:

Out[45]:

| | Item_Weight | Item_Visibility | Item_MRP | Outlet_Age | Item_Fat_Content_Low Fat | Item_Fat_Content_Regular | Outlet_Identifier_OUT010 | Outlet_Identifier_OUT013 |
|---|---|---|---|---|---|---|---|---|
| 7566 | 19.00 | 0.031024 | 210.5244 | 25 | 1 | 0 | 0 | 0 |
| 1236 | 9.80 | 0.047454 | 101.7016 | 3 | 1 | 0 | 0 | 0 |
| 7104 | 17.60 | 0.175546 | 163.6868 | 12 | 1 | 0 | 1 | 0 |
| 227 | 15.85 | 0.107765 | 59.5904 | 11 | 1 | 0 | 0 | 0 |
| 4200 | 15.30 | 0.022959 | 101.6332 | 23 | 1 | 0 | 0 | 1 |

5 rows × 29 columns

Out[45]:

| | Item_Outlet_Sales |
|---|---|
| 7566 | 1482.0708 |
| 1236 | 1518.0240 |
| 7104 | 163.7868 |
| 227 | 703.0848 |
| 4200 | 1845.5976 |

Since the target variable is continuous, a simple, yet standard approach is to test a linear regression model. Once imported from the sklearn package, the function is applied to the train data using the model.fit() function. The predictions are then stored in an array using model.predict(). Model evaluation is conducted by using a variety of the metric functions from sklearn, along with plotting the actual vs. predicted values. From these results, it appears linear regression may not be the best model for this data.

In [59]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression(normalize=True)
lr.fit(train_predictors,train_target)

train_predictions=lr.predict(train_predictors)

train_predictions
```

Out[59]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=True)
```

Out[59]:
```
array([[1421.81051499],
       [1772.80318599],
       [ 663.53051985],
       ...,
       [2385.76875632],
       [2952.31715161],
       [2596.73167759]])
```

In [60]:
```
from sklearn import metrics

np.sqrt(metrics.mean_squared_error(train_target, train_predictions))
metrics.r2_score(train_target, train_predictions)
metrics.mean_absolute_error(train_target, train_predictions)
```

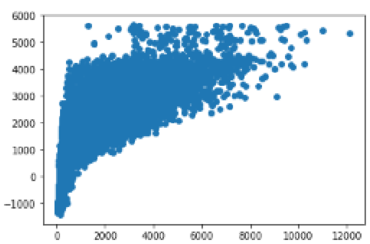
Out[60]: 1120.252575629138

Out[60]: 0.5630719324461717

Out[60]: 831.0963740409376

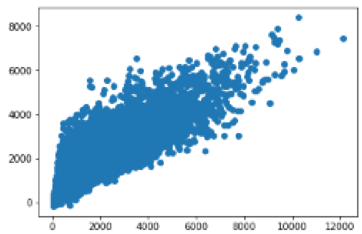
In [61]:
```
plt.scatter(train_target,train_predictions)
```

Out[61]:
```
<matplotlib.collections.PathCollection at 0x1fa521644e0>
```



XGBRegressor tool is imported, it is applied to the train data and predictions are stored to evaluate the model. The metrics and plot both reveal higher performance than the linear regression model:

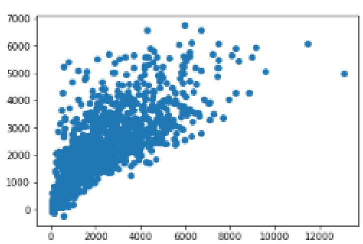
```
In [66]: from xgboost import XGBRegressor

         xg_model = XGBRegressor(n_estimators=1000, learning_rate=0.05)
         xg_model.fit(train_predictors, train_target)

         xg_train_predictions = xg_model.predict(train_predictors)

Out[66]: XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                colsample_bytree=1, gamma=0, importance_type='gain',
                learning_rate=0.05, max_delta_step=0, max_depth=3,
                min_child_weight=1, missing=None, n_estimators=1000, n_jobs=1,
                nthread=None, objective='reg:linear', random_state=0, reg_alpha=0,
                reg_lambda=1, scale_pos_weight=1, seed=None, silent=True,
                subsample=1)

In [71]: plt.scatter(train_target,xg_train_predictions)

Out[71]: <matplotlib.collections.PathCollection at 0x1fa68020198>
```




```
In [72]: np.sqrt(metrics.mean_squared_error(train_target, xg_train_predictions))
         metrics.r2_score(train_target, xg_train_predictions)
         metrics.mean_absolute_error(train_target, xg_train_predictions)

Out[72]: 922.0719026834847

Out[72]: 0.7039891577388857

Out[72]: 661.121754605931
```

This model then needs to be evaluated against the test data to determine if it is in fact a good model. The test data reveals that there is still some relevance to the model but further parameter tuning, and possibly other model selections may lead to better results.

```
In [73]: xg_test_predictions = xg_model.predict(test_predictors)

         plt.scatter(test_target,xg_test_predictions)
         np.sqrt(metrics.mean_squared_error(test_target, xg_test_predictions))
         metrics.r2_score(test_target, xg_test_predictions)
         metrics.mean_absolute_error(test_target, xg_test_predictions)

Out[73]: <matplotlib.collections.PathCollection at 0x1fa68075828>

Out[73]: 1141.6160734892937

Out[73]: 0.5843002719742592

Out[73]: 788.4657297851826
```




Overall, as evident by this demo, the actual fitting and tuning of a model is a small step compared to the entire machine learning process. In order to even obtain a data set for also display the results of their analysis and modeling to their audience, making this library imperative for successful machine learning in Python.

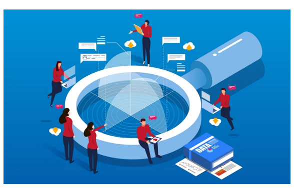


**ABOUT THE AUTHOR**

## MEGAN QUINN

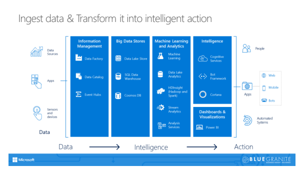
Megan has expertise in statistical analysis and machine learning as well as statistical theory. Her recent focus has been centered on predictive maintenance for military fleets with a background in education research as well. She is knowledgeable in a variety of analytical tools including Python, R, SQL, and most recently Spark & Databricks.

# RELATED ARTICLES



Knowledge Mining Showcase: Azure Search

Azure Data Lake Analytics Holds a Unique Spot in the Modern Data Architecture




Cortana Intelligence Suite Tour

ALL POSTS PREVIOUS POST

Enterprise BI &
Reporting
Modern Data Platform

Events
Resources

2750 Old Centre Rd Ste
150
Portage, MI 49024
T. 877-817-0736