-

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA HỆ THỐNG THÔNG TIN

# Enterprise Resource Planning

## Sale Forecasting using Machine – Deep Learning

ACCT5123.N21.CTTT

**Lecturer: Đỗ Duy Thanh**

**Lecturer: Huỳnh Đức Huy**


**Authors: Phạm Thùy Dung - 20521214**

**Đậu Đình Quang Anh – 20521059**

**Võ Trọng Huân - 20520524**

# Table of Contents



# I. Introduction

The objective of this report is to effectively manage and keep track of the sales data of individual items at Big Mart, a prominent retail chain. By analyzing this data, we can forecast future client demand and make necessary adjustments to our inventory management strategies. This will allow us to stay ahead of the competition and maximize profitability. To achieve this, we aim to execute a sophisticated model that leverages machine learning and deep learning techniques to predict future sales patterns accurately. By utilizing Big Mart's history dataset, this model will provide valuable insights into consumer behavior and market trends. The comprehensive report resulting from this analysis will serve as a retail-chain resource, providing detailed and extensive information on the utilization of data analytics in the retail sector. It will delve into the process of data collection, the implementation of various machine learning algorithms, and the interpretation of results. The report will emphasize the importance of leveraging data to optimize inventory management, anticipate customer demands, and ultimately enhance business performance.

# II. Sale Forecasting

## About Dataset

### Overview

In this paper, we present our analysis of the 2013 Big Mart sales data, which consists of 12 features includes: Item_Identifie, Item_Weight , Item_Fat_Content, Item_Visibility, Item_Type, Item_MRP, Outlet_Identifier, Outlet_Establishment_Year, Outlet_Size, Outlet_Location_Type, Outlet_Type, Item_Outlet_Sales. We aim to predict the Item Outlet Sales feature using the other features as independent variables. Our dataset contains 8523 products from different regions and cities. The dataset also reflects product-level and store-level factors that may affect sales. Product-level factors include product characteristics, advertising, etc., while store-level factors include city, population density, store capacity, location, etc. We preprocess the dataset and split it into two parts: training and testing.

### Description

| Variable | Description |
|---|---|
| Item_Identifier | Unique product ID |
| Item_Weight | Weight of product |
| Item_Fat_Content | Whether the product is low fat or not |
| Item_Visibility | The % of total display area of all products in a store allocated to the particular product |
| Item_Type | The category to which the product belongs |
| Item_MRP | Maximum Retail Price (list price) of the product |
| Outlet_Identifier | Unique store ID |
| Outlet_Establishment_Year | The year in which store was established |
| Outlet_Size | The size of the store in terms of ground area covered |
| Outlet_Location_Type | The type of city in which the store is located |
| Outlet_Type | Whether the outlet is just a grocery store or some sort of supermarket |
| Item_Outlet_Sales | Sales of the product in the particulat store. This is the outcome variable to be predicted. |

Table: Description of each feature in Dataset

Figure: Working procedure of proposed model in Machine Learning

## XGBoost model

XGBoost is one of the most popular machine learning frameworks among data scientists. According to the Kaggle State of Data Science Survey 2021, almost 50% of respondents said they used XGBoost, ranking below only TensorFlow and Sklearn.

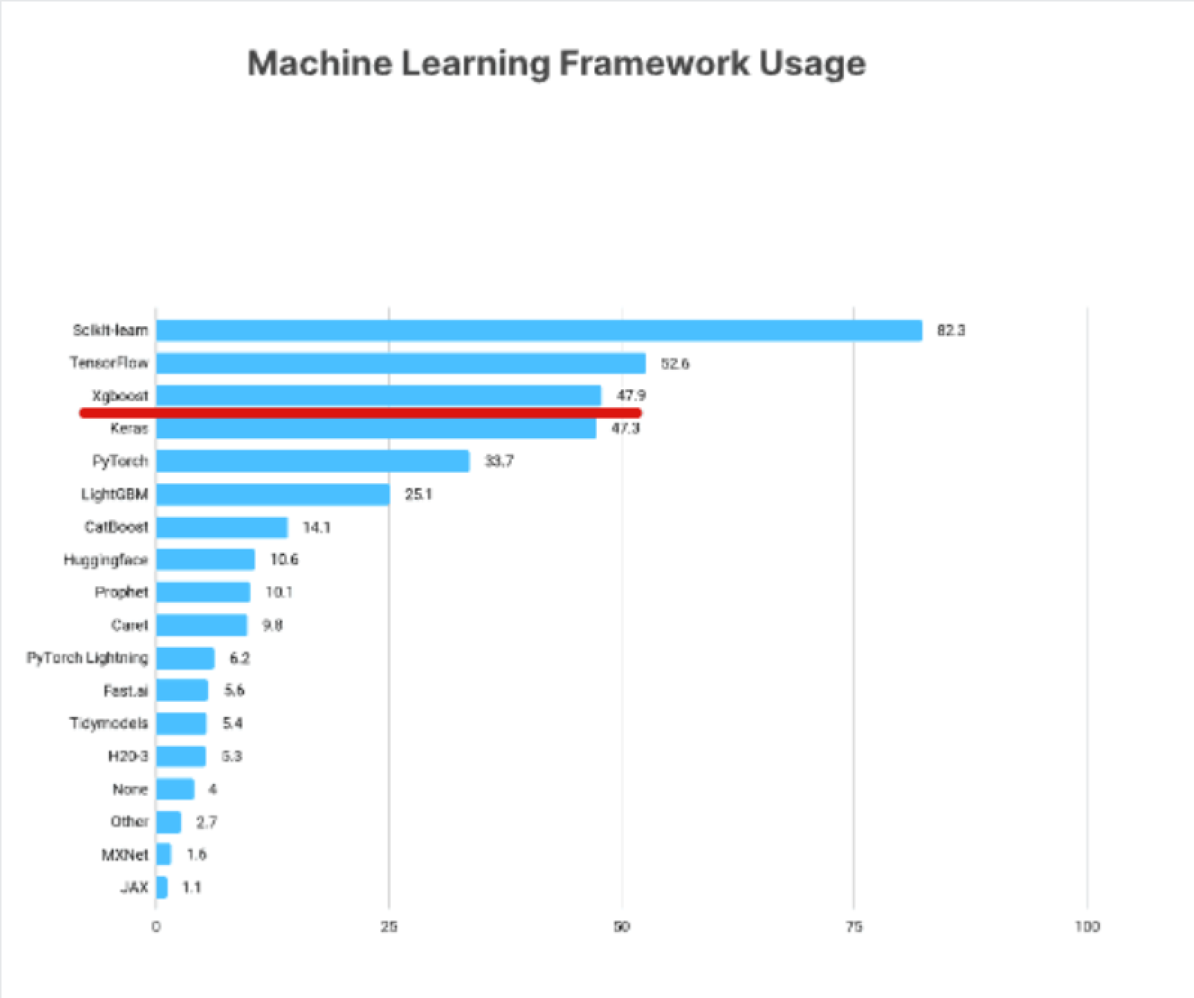



Figure: Comparing Machine Learning Frame Work Usage [1]

## Definition

XGBoost, short for Extreme Gradient Boosting, is a machine learning library that is widely used for regression, classification, and ranking problems. It is an optimized distributed gradient boosting library designed for efficient and scalable training of machine learning models. XGBoost is based on the concept of decision tree ensembles, where multiple weak models are combined to create a stronger prediction.

## How XGBoost work?

Gradient boosting is typically used with decision trees (especially CARTs) of a fixed size as base learners. For this special case, Friedman proposes a modification to gradient boosting method which improves the quality of fit of each base learner.

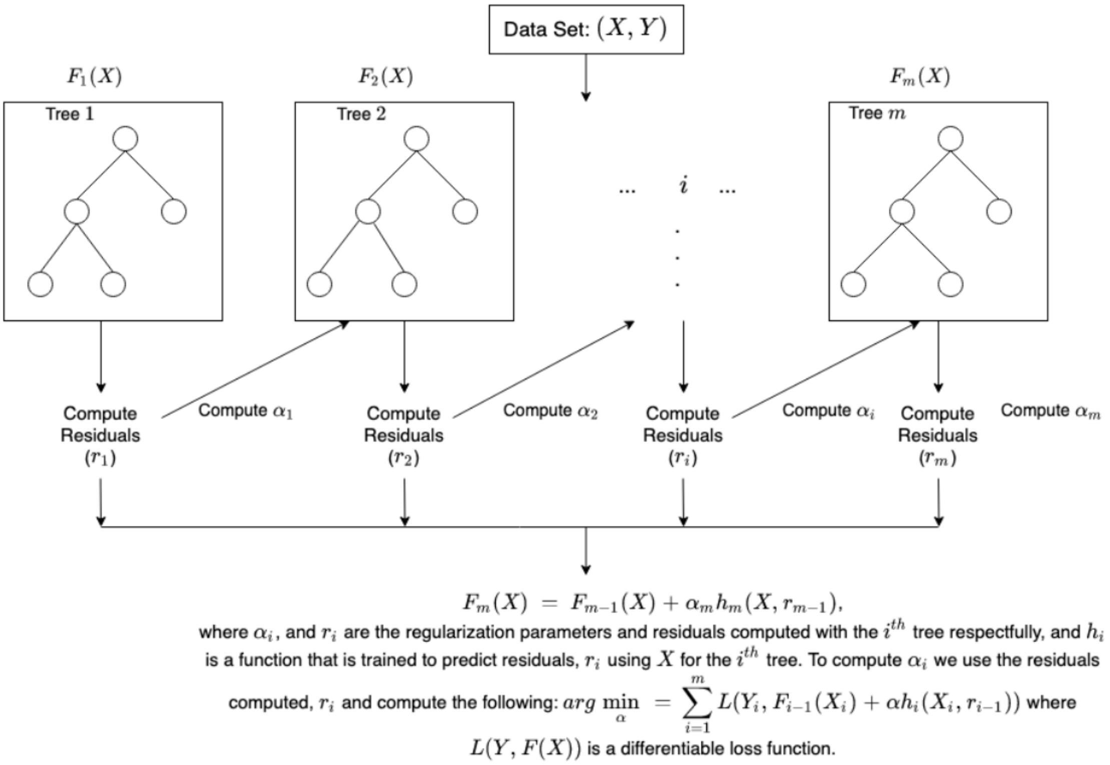



Figure: brief illustration on how gradient tree boosting works[2]

When using XGBoost for regression, the weak learners are regression trees, and each regression tree maps an input data point to one of its leaves that contains a continuous score. XGBoost minimizes a regularized (L1 and L2) objective function that combines a convex loss function (based on the difference between the predicted and target outputs) and a penalty term for model complexity (in other words, the regression tree functions). The training proceeds iteratively, adding new trees that predict the residuals or errors of prior trees that are then combined with previous trees to make the final prediction. It's called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models

### *1.2 Explaining the code*

Step 1: Import necessary dataset

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder
from xgboost import XGBRegressor
from sklearn import metrics
```

Step 2: Import dataset

```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```python
train = pd.read_csv("/content/Train.csv")
train.head()
```

Step 4: Using train info to provide information about the DataFrame train, including the column names, data types, and number of non-null values in each column

```python
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8523 entries, 0 to 8522
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype  
---  ------                     --------------  -----  
 0   Item_Identifier            8523 non-null   object 
 1   Item_Weight                7060 non-null   float64
 2   Item_Fat_Content           8523 non-null   object 
 3   Item_Visibility            8523 non-null   float64
 4   Item_Type                  8523 non-null   object 
 5   Item_MRP                   8523 non-null   float64
 6   Outlet_Identifier          8523 non-null   object 
 7   Outlet_Establishment_Year  8523 non-null   int64  
 8   Outlet_Size                6113 non-null   object 
 9   Outlet_Location_Type       8523 non-null   object 
 10  Outlet_Type                8523 non-null   object 
 11  Item_Outlet_Sales          8523 non-null   float64
dtypes: float64(4), int64(1), object(7)
memory usage: 799.2+ KB
```

Step 5: he train.isnull().sum() function calculates the number of missing values (NaN or null values) in each column of the train DataFrame.

```
train.isnull().sum()
```

```
Item_Identifier                 0
Item_Weight                  1463
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
dtype: int64
```

In here, you can see Item_Weight having missing value is 1463, and Outlet_Size had missing values is 2410. So, two columns here are useful information for cleaning and handling missing values.

Step 6: fill the missing values in the 'Item_Weight' and 'Outlet_Size' columns of the train DataFrame, and then checks for any remaining missing values

```
train['Item_Weight'] = train['Item_Weight'].fillna(train['Item_Weight'].mean())
train['Outlet_Size'] = train['Outlet_Size'].fillna(train['Outlet_Size'].mode()[0])
train.isnull().sum()
```

```
Item_Identifier              0
Item_Weight                  0
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Establishment_Year    0
Outlet_Size                  0
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
dtype: int64
```

Step 7: counts the occurrences of each unique value in the 'Item_Fat_Content' column of the train DataFrame.

```
train['Item_Fat_Content'].value_counts()
```

```
Low Fat    5089
Regular    2889
LF          316
reg         117
low fat     112
Name: Item_Fat_Content, dtype: int64
```

Step 8: replaces specific values in the 'Item_Fat_Content' column of the train DataFrame and then counts the occurrences of each unique value

```
train.replace({'Item_Fat_Content': {'low fat': 'Low Fat', 'LF':'Low Fat', 'reg': 'Regular'}}, inplace = True)
train['Item_Fat_Content'].value_counts()
```

```
Low Fat    5517
Regular    3006
Name: Item_Fat_Content, dtype: int64
```

In here, Low Fat, low fat, and LF is the same meaning, so we collect in 1 cluster, and reg and Regular is 1 cluster

Step 9: using the LabelEncoder from scikit-learn to encode categorical variables in the train DataFrame

```
encoder = LabelEncoder()
train['Item_Identifier'] = encoder.fit_transform(train['Item_Identifier'])
train['Item_Fat_Content'] = encoder.fit_transform(train['Item_Fat_Content'])
train['Item_Type'] = encoder.fit_transform(train['Item_Type'])
train['Outlet_Identifier'] = encoder.fit_transform(train['Outlet_Identifier'])
train['Outlet_Size'] = encoder.fit_transform(train['Outlet_Size'])
train['Outlet_Location_Type'] = encoder.fit_transform(train['Outlet_Location_Type'])
train['Outlet_Type'] = encoder.fit_transform(train['Outlet_Type'])
train.head()
```

```
Data columns (total 12 columns):
 #   Column                     Non-Null Count  Dtype  
---  ------                     --------------  -----  
 0   Item_Identifier            8523 non-null   object 
 1   Item_Weight                7060 non-null   float64
 2   Item_Fat_Content           8523 non-null   object 
 3   Item_Visibility            8523 non-null   float64
 4   Item_Type                  8523 non-null   object 
 5   Item_MRP                   8523 non-null   float64
 6   Outlet_Identifier          8523 non-null   object 
 7   Outlet_Establishment_Year  8523 non-null   int64  
 8   Outlet_Size                6113 non-null   object 
 9   Outlet_Location_Type       8523 non-null   object 
 10  Outlet_Type                8523 non-null   object 
 11  Item_Outlet_Sales          8523 non-null   float64
```

Look at this table, we can see Object being datatype includes: Item_Identifier, Item_Fat_Content, Item_Visibility, Item_Type, Outlet_Identifier, Outlet_Size, Outlet_Location_Type, Outlet_Type. So we encode to transforms the column values to their encoded form.

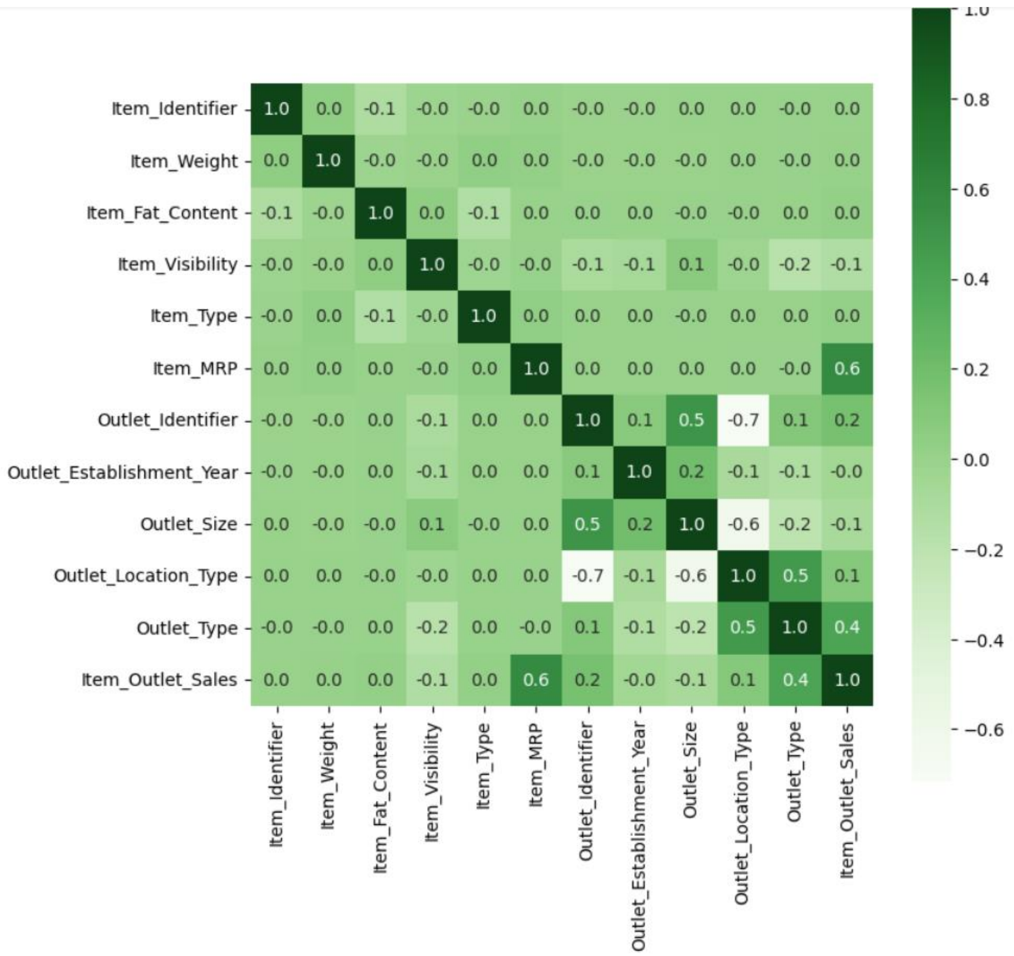
Here is result:

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Out: |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 156 | 9.30 | 0 | 0.016047 | 4 | 249.8092 | 9 | 1999 | 1 | |
| 1 | 8 | 5.92 | 1 | 0.019278 | 14 | 48.2692 | 3 | 2009 | 1 | |
| 2 | 662 | 17.50 | 0 | 0.016760 | 10 | 141.6180 | 9 | 1999 | 1 | |
| 3 | 1121 | 19.20 | 1 | 0.000000 | 6 | 182.0950 | 0 | 1998 | 1 | |
| 4 | 1297 | 8.93 | 0 | 0.000000 | 9 | 53.8614 | 1 | 1987 | 0 | |

Step 10: Calculates the correlation matrix of the train DataFrame and creates a heatmap visualization using the Seaborn library

```
corr = train.corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr,cbar=True,square=True,fmt='.1f',annot=True,cmap='Greens')
```

Here is result:

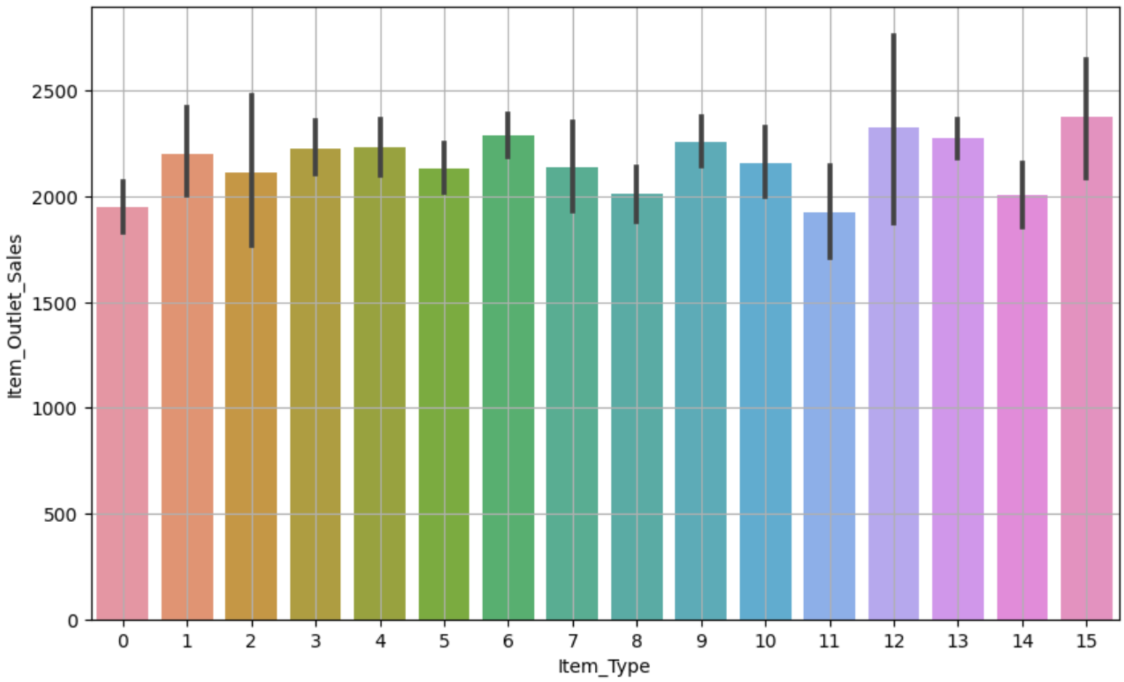


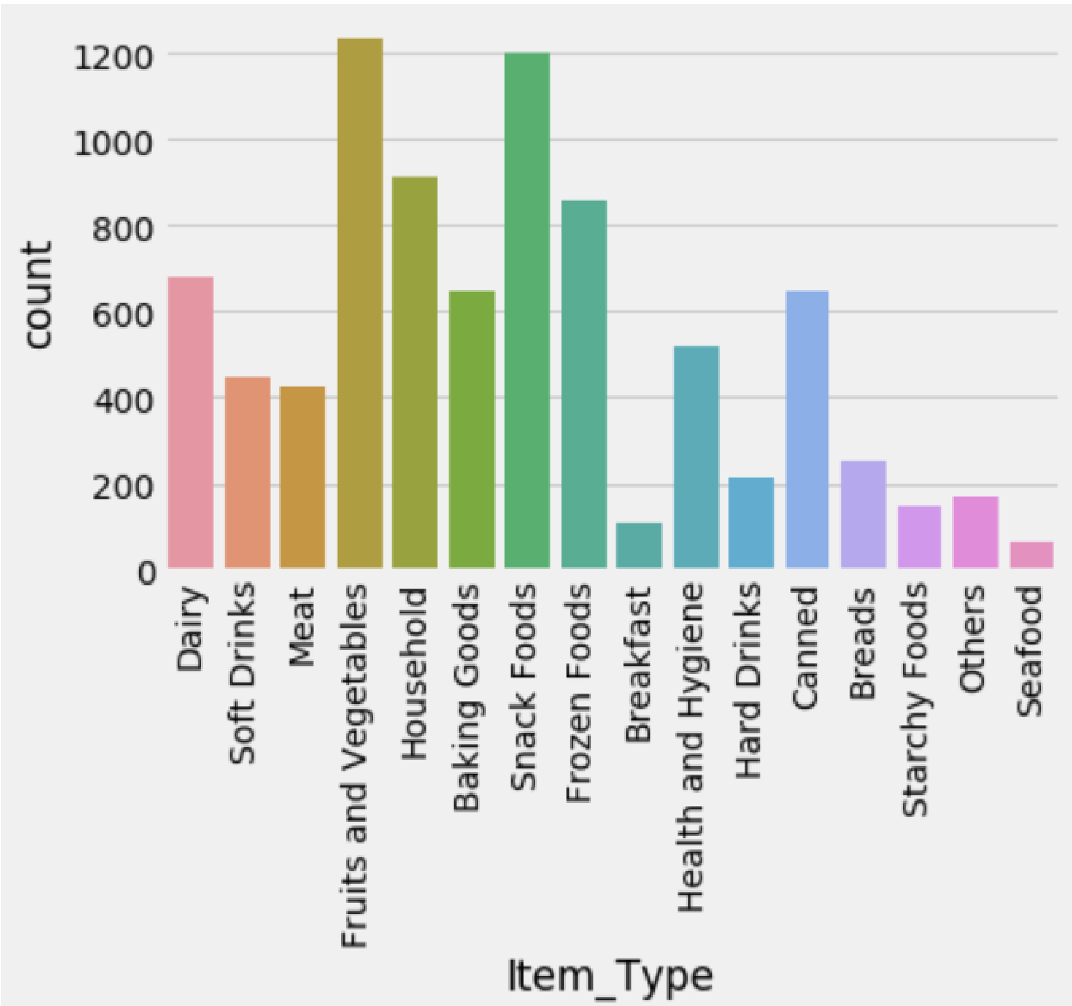

Look at this heatmap, you can see, There isn't much correlation between the variables, except that if Item_MRP increases, Item_Outlet_Sales increases.

Step 11: visualize the relationship between the 'Item_Type' and 'Item_Outlet_Sales' columns

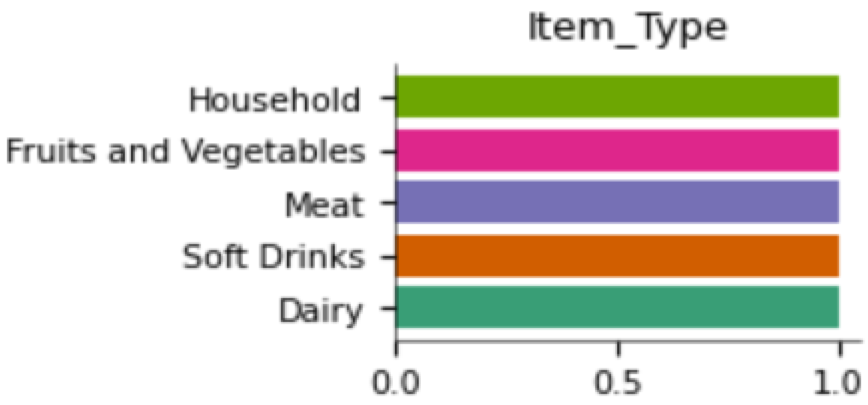
```
plt.figure(figsize=(10,6))
sns.barplot(x='Item_Type', y='Item_Outlet_Sales', data = train)
plt.grid()
```

Here is result

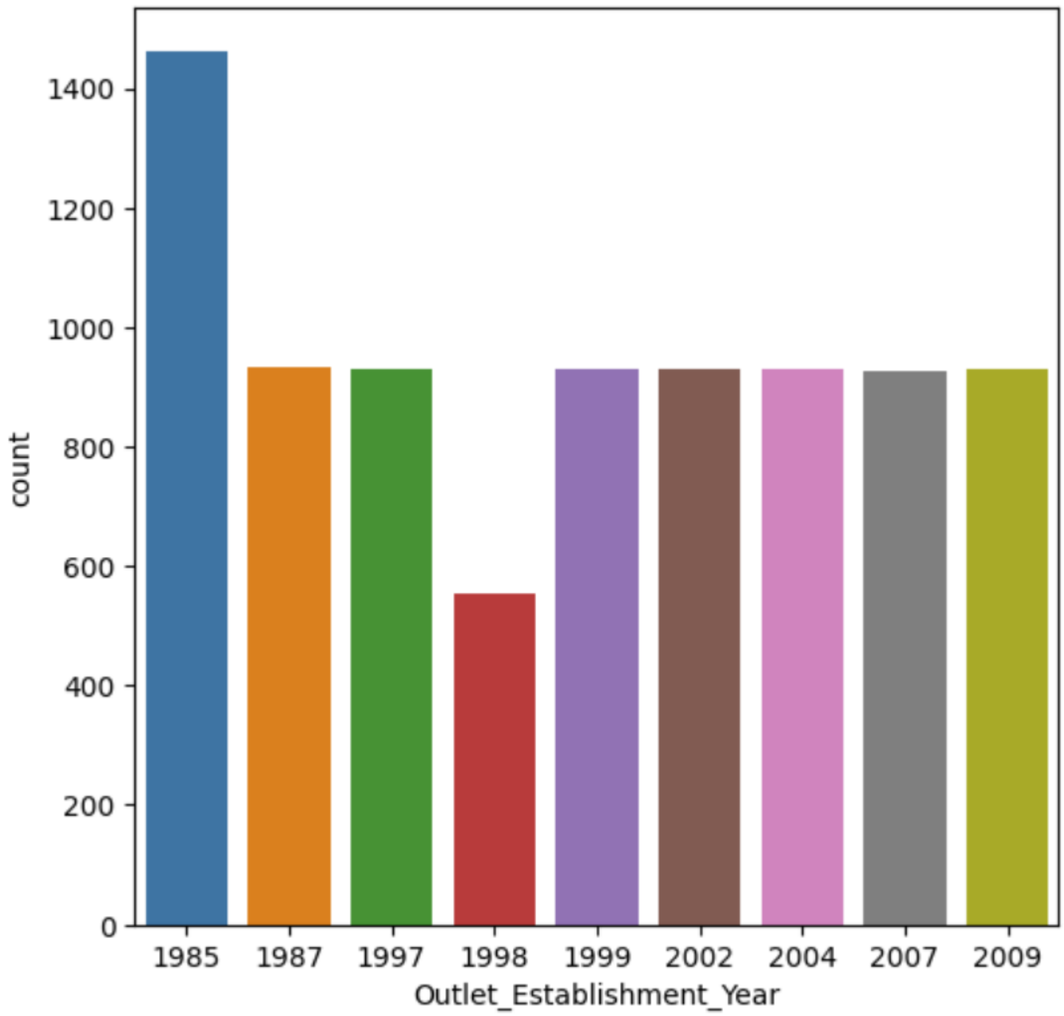



From the illustration above, we can tell items like fruits and vegetables, household goods, snacks, starchy foods and seafood (From are more sold than the other items so the mall should consider keeping more of these items in inventory, give special offers and discounts to these goods so volume of sales increases more.

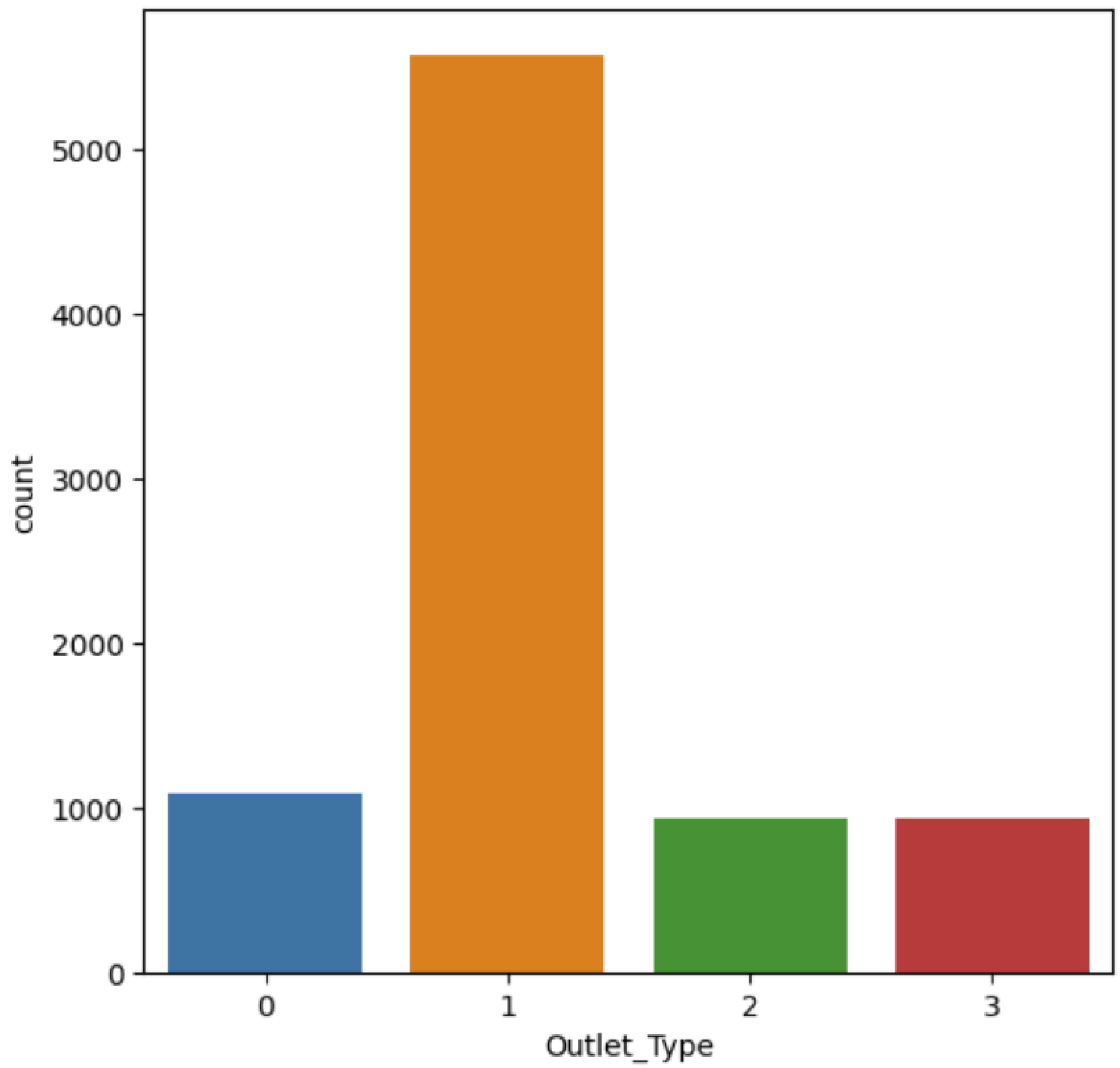



Step 12: creates a count plot using Seaborn's countplot() function to visualize the distribution of the 'Outlet_Establishment_Year' column in the train DataFrame




In 1985, the debut year of the mall had the highest sales but from then on, sales volume was pretty much constant.

Step 13: creates a count plot using Seaborn's countplot() function to visualize the distribution of the 'Outlet_Type' column in the train DataFrame




Supermarket Type1 have the most sales, much higher than other types so the mall owners should consider building more of these types in other locations.

Step 14: drop column Item_Outlet_Sales because it is target to predict, and then split train, test dataset into 8:2

```
X = train.drop(columns='Item_Outlet_Sales', axis = 1)
y = train['Item_Outlet_Sales']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 2)
print(X.shape,X_train.shape)
```

```
(8523, 11) (6818, 11)
```

As we can see, 6818 rows are used for testing out of 8523 which is about 80% of the data.

Step 15: Build the Model XGBoost

```
model1 = XGBRegressor()

# Now we need to train the model
model1.fit(X_train,y_train) # fitting means training
```

```
XGBRegressor
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, early_stopping_rounds=None,
             enable_categorical=False, eval_metric=None, feature_types=None,
             gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=None, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=None, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             n_estimators=100, n_jobs=None, num_parallel_tree=None,
             predictor=None, random_state=None, ...)
```

Step 16: predicting the target variable (Item_Outlet_Sales) for the training set using a trained model and storing the predictions in the variable train_pred.

```
train_pred = model.predict(X_train)
train_pred
```

```
array([2172.693 , 2844.0671, 3308.6353, ..., 3363.3127, 1717.4066,
       2013.252 ], dtype=float32)
```

Step 17: prints the evaluation scores for an XGBoost model with Train dataset

```
print('The evalution scores XGBoost: ')
r2 = metrics.r2_score(y_train, train_pred)
mse = metrics.mean_squared_error(y_train, train_pred)
rmse = mse ** 0.5
mae = metrics.mean_absolute_error(y_train, train_pred)
mdae = metrics.median_absolute_error(y_train, train_pred)
print('R2: ', r2)
print('MSE :', mse)
print('RMSE: ', rmse)
print('MAE: ', mae)
print('MDAE: ', mdae)
print('\n')
```

```
The evalution scores XGBoost:
R2:  0.854983167058186
MSE : 415766.97370557557
RMSE:  644.7999485930311
MAE:  464.38981288805894
MDAE:  317.6078312499999
```

Step 18: generate predictions for the test set (X_test) using a trained model

```
test_pred = model.predict(X_test)
test_pred
```

```
array([2098.7969, 4360.376 , 1454.3608, ..., 2883.5608, 1158.3351,
       3164.4902], dtype=float32)
```

Step 20: calculates and prints the evaluation scores for an XGBoost model

```
print('The evalution scores XGBoost: ')
r2 = metrics.r2_score(y_test, test_pred)
mse = metrics.mean_squared_error(y_test, test_pred)
rmse = mse ** 0.5
mae = metrics.mean_absolute_error(y_test, test_pred)
mdae = metrics.median_absolute_error(y_test, test_pred)
print('R2: ', r2)
print('MSE :', mse)
print('RMSE: ', rmse)
print('MAE: ', mae)
print('MDAE: ', mdae)
print('\n')
```

```
The evalution scores XGBoost:
R2:  0.5191234777241828
MSE : 1484501.750774029
RMSE:  1218.4013094108316
MAE:  856.8716943000217
MDAE:  576.9436505859376
```

R squared error is closer to 0, meaning that the better predicting, so I will check the other model to compare all of them to find the best model to predicting the sale price.

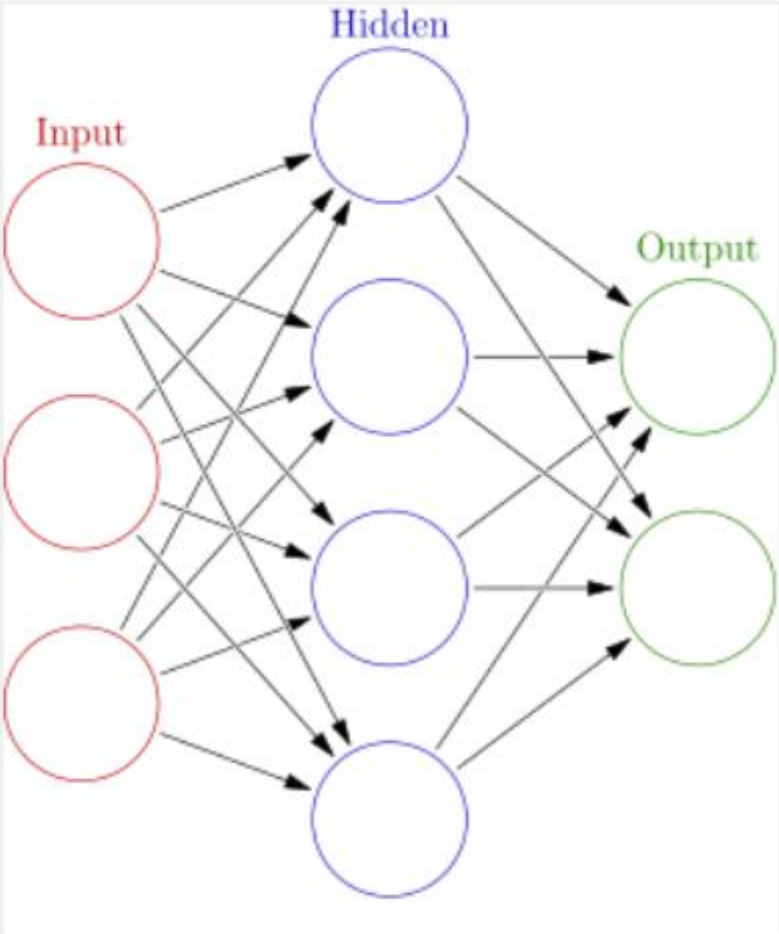
## ANN model

### Defintion

An Artificial Neural Network (ANN) is a computational model inspired by the biological neural networks in animal brains. It consists of interconnected artificial neurons organized in layers, including an input layer, one or more hidden layers, and an output layer. ANNs can learn from data through learning algorithms and adjust their parameters to make predictions or classifications.

## How it work

ANNs utilize the hidden layer as a place to store and evaluate how significant one of the inputs is to the output. The hidden layer stores information regarding the input's importance, and it also makes associations between the importance of combinations of inputs.




*One hidden layer is considered an Artificial Neural Network (ANN)*

Figure: Artificial Neural NetWork architecture[3]

## Explaining code

### Step 1: Import necessary libraries

```
import numpy as np
import pandas as pd
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
%matplotlib inline
```

### Step 2: Import dataset

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
train = pd.read_csv('/content/Train.csv')
train.head()
```

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

### Step 3: calculates the percentage of missing values in each column

```
train.isnull().sum()/len(train)*100
```

```
Item_Identifier               0.000000
Item_Weight                  17.165317
Item_Fat_Content              0.000000
Item_Visibility               0.000000
Item_Type                     0.000000
Item_MRP                      0.000000
Outlet_Identifier             0.000000
Outlet_Establishment_Year     0.000000
Outlet_Size                  28.276428
Outlet_Location_Type          0.000000
Outlet_Type                   0.000000
Item_Outlet_Sales             0.000000
dtype: float64
```

Step 4: fill the missing value of two columns: Outlet_Size and Item_Weight

```
train['Outlet_Size'].fillna(train['Outlet_Size'].mode()[0], inplace=True)
train['Item_Weight'].fillna(train['Item_Weight'].mean(), inplace=True)
```

```
train.isnull().sum()
```

```
Item_Identifier              0
Item_Weight                  0
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Establishment_Year    0
Outlet_Size                  0
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
dtype: int64
```

Step 5: perform one-hot encoding on the specified categorical columns of the DataFrame train.

```
train = pd.get_dummies(train, columns = ['Item_Fat_Content', 'Item_Type', 'Outlet_Size', 'Outlet_Location_Type', 'Outlet_Type'],prefix = '',prefix_sep = '')
```

```
train = pd.get_dummies(train, columns = ['Outlet_Identifier'],prefix = '',prefix_sep = '')
```

```
train.head()
```

| | Item_Identifier | Item_Weight | Item_Visibility | Item_MRP | Outlet_Establishment_Year | Item_Outlet_Sales | LF | Low Fat | Regular | low fat | ... | OUT010 | OUT013 | OUT017 | OUT018 | OUT019 | OUT027 | OUT035 | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | FDA15 | 9.30 | 0.016047 | 249.8092 | 1999 | 3735.1380 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **1** | DRC01 | 5.92 | 0.019278 | 48.2692 | 2009 | 443.4228 | 0 | 0 | 1 | 0 | ... | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| **2** | FDN15 | 17.50 | 0.016760 | 141.6180 | 1999 | 2097.2700 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **3** | FDX07 | 19.20 | 0.000000 | 182.0950 | 1998 | 732.3800 | 0 | 0 | 1 | 0 | ... | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **4** | NCD19 | 8.93 | 0.000000 | 53.8614 | 1987 | 994.7052 | 0 | 1 | 0 | 0 | ... | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

5 rows × 47 columns

Step 6: perform min-max scaling on the numerical columns of the DataFrame train and define x, y to training model

```
for i in train.columns[1:]:
    train[i] = (train[i] - train[i].min()) / (train[i].max() - train[i].min())
```

```
train = train.drop('Item_Identifier', axis=1)
```

```
X = train.drop('Item_Outlet_Sales', axis=1)

y = train['Item_Outlet_Sales']
```

Step 7: Split two train, test dataset and see the result

```
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=10,test_size=0.2)
```

```
(X_train.shape, y_train.shape), (X_test.shape, y_test.shape)
```

```
(((6818, 45), (6818,)), ((1705, 45), (1705,)))
```

Step 8: Import necessaries libraries

```python
import keras
print(keras.__version__)
```

```
2.12.0
```

```python
import tensorflow as tf
print(tf.__version__)
```

```
2.12.0
```

```python
from keras.models import Sequential
```

```python
from keras.layers import InputLayer, Dense,Flatten,Dropout
```

Step 9: define the model and model summary

```
model = Sequential()
```

```
model.add(Dense(128,kernel_initializer='normal',input_shape=(X_train.shape[1],),activation='relu'))
```

```
model.add(Dense(256,kernel_initializer='normal',activation='relu'))
model.add(Dense(256,kernel_initializer='normal',activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(256,kernel_initializer='normal',activation='relu'))
```

```
model.add(Dense(1,kernel_initializer='normal',activation='linear'))
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               5888

 dense_1 (Dense)             (None, 256)               33024

 dense_2 (Dense)             (None, 256)               65792

 dropout (Dropout)           (None, 256)               0

 dense_3 (Dense)             (None, 256)               65792

 dense_4 (Dense)             (None, 1)                 257

=================================================================
Total params: 170,753
Trainable params: 170,753
Non-trainable params: 0
_________________________________________________________________
```

Step 10: Compiling the model and train it

```
[ ] model.compile(optimizer='adam',loss='mean_absolute_error',metrics=['mean_absolute_error'])
```

```
model_history = model.fit(X_train,y_train,epochs=300,batch_size=60,validation_split=0.2)
```

```
Epoch 8/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0581 - mean_absolute_error: 0.0581 - val_loss: 0.0617 - val_mean_absolute_error: 0.0617
Epoch 9/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0588 - mean_absolute_error: 0.0588 - val_loss: 0.0616 - val_mean_absolute_error: 0.0616
Epoch 10/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0582 - mean_absolute_error: 0.0582 - val_loss: 0.0584 - val_mean_absolute_error: 0.0584
Epoch 11/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0576 - mean_absolute_error: 0.0576 - val_loss: 0.0590 - val_mean_absolute_error: 0.0590
Epoch 12/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0574 - mean_absolute_error: 0.0574 - val_loss: 0.0630 - val_mean_absolute_error: 0.0630
Epoch 13/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0581 - mean_absolute_error: 0.0581 - val_loss: 0.0583 - val_mean_absolute_error: 0.0583
Epoch 14/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0569 - mean_absolute_error: 0.0569 - val_loss: 0.0585 - val_mean_absolute_error: 0.0585
Epoch 15/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0567 - mean_absolute_error: 0.0567 - val_loss: 0.0581 - val_mean_absolute_error: 0.0581
Epoch 16/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0566 - mean_absolute_error: 0.0566 - val_loss: 0.0581 - val_mean_absolute_error: 0.0581
Epoch 17/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0562 - mean_absolute_error: 0.0562 - val_loss: 0.0614 - val_mean_absolute_error: 0.0614
Epoch 18/300
91/91 [==============================] - 1s 6ms/step - loss: 0.0568 - mean_absolute_error: 0.0568 - val_loss: 0.0587 - val_mean_absolute_error: 0.0587
Epoch 19/300
91/91 [==============================] - 1s 7ms/step - loss: 0.0564 - mean_absolute_error: 0.0564 - val_loss: 0.0579 - val_mean_absolute_error: 0.0579
Epoch 20/300
91/91 [==============================] - 1s 8ms/step - loss: 0.0559 - mean_absolute_error: 0.0559 - val_loss: 0.0583 - val_mean_absolute_error: 0.0583
Epoch 21/300
91/91 [==============================] - 1s 12ms/step - loss: 0.0554 - mean_absolute_error: 0.0554 - val_loss: 0.0589 - val_mean_absolute_error: 0.0589
Epoch 22/300
91/91 [==============================] - 1s 12ms/step - loss: 0.0555 - mean_absolute_error: 0.0555 - val_loss: 0.0592 - val_mean_absolute_error: 0.0592
Epoch 23/300
91/91 [==============================] - 2s 22ms/step - loss: 0.0560 - mean_absolute_error: 0.0560 - val_loss: 0.0606 - val_mean_absolute_error: 0.0606
Epoch 24/300
91/91 [==============================] - 1s 13ms/step - loss: 0.0545 - mean_absolute_error: 0.0545 - val_loss: 0.0617 - val_mean_absolute_error: 0.0617
Epoch 25/300
91/91 [==============================] - 1s 10ms/step - loss: 0.0555 - mean_absolute_error: 0.0555 - val_loss: 0.0588 - val_mean_absolute_error: 0.0588
Epoch 26/300
91/91 [==============================] - 1s 12ms/step - loss: 0.0545 - mean_absolute_error: 0.0545 - val_loss: 0.0592 - val_mean_absolute_error: 0.0592
```

Step 11: get prediction for validation test

```
prediction = model.predict(X_test)
```

```
54/54 [==============================] - 0s 2ms/step
```

```
prediction
```

```
array([[0.2750923 ],
       [0.11984784],
       [0.1351529 ],
       ...,
       [0.0535614 ],
       [0.26889634],
       [0.07705434]], dtype=float32)
```

```
from sklearn.metrics import mean_squared_error, mean_absolute_error, median_absolute_error, r2_score

mse = mean_squared_error(y_test, prediction)
rmse = mean_squared_error(y_test, prediction, squared=False)
mae = mean_absolute_error(y_test, prediction)
mdae = median_absolute_error(y_test, prediction)
r2 = r2_score(y_test, prediction)

print('The evaluation scores for the ANN model are:')
print('R2:', r2)
print('MSE:', mse)
print('RMSE:', rmse)
print('MAE:', mae)
print('MDAE:', mdae)
```

```
The evaluation scores for the ANN model are:
R2: 0.3381651964239938
MSE: 0.01152943890524462
RMSE: 0.10737522482046136
MAE: 0.07606321682762442
MDAE: 0.05088797999646821
```

## DNN model

### Definition

Deep Neural Networks (DNNs) are a specific type of ANN with multiple hidden layers between the input and output layers. The term "deep" refers to the depth of the network, indicating the presence of multiple layers. DNNs can model complex non-linear relationships and extract high-level features from data. They are capable of learning hierarchical representations, allowing them to automatically learn key features and patterns from the input data.

### How Deep Neural Network works?

DNNs are a subset of ANNs that have multiple hidden layers, enabling them to learn complex patterns and representations

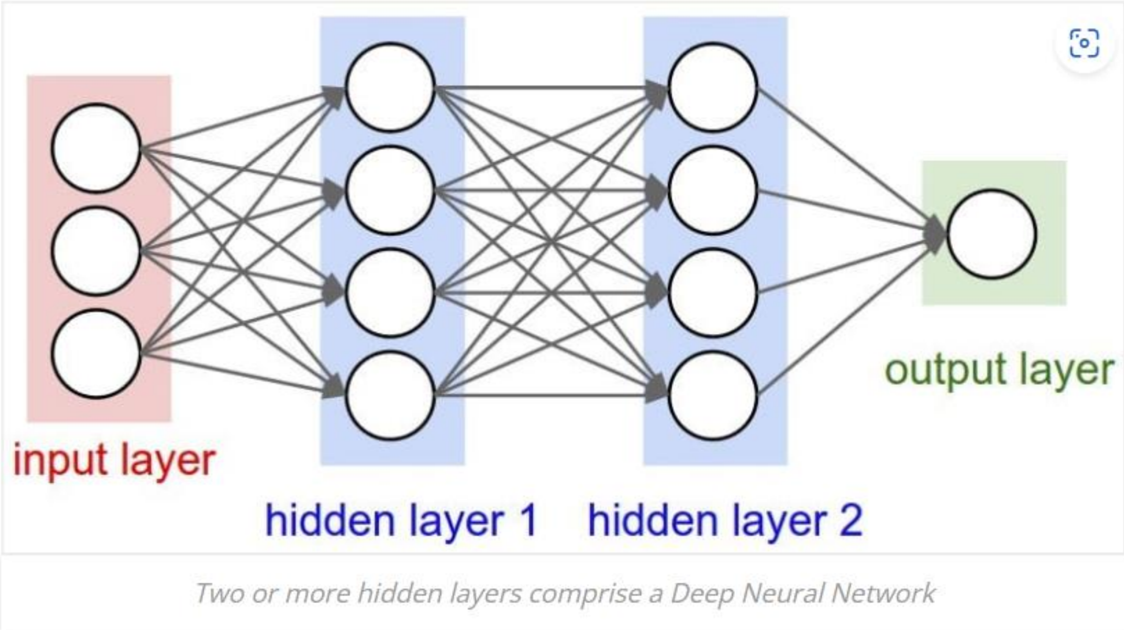



Figure: Deep Neural Network Architecture[3]

Explaining code

Step 1: Import necessary libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.model_selection import train_test_split
plt.style.use('ggplot')
pd.set_option('display.max_columns', 100)
%matplotlib inline
```

‘plt.style.use('ggplot')’: This line sets the style of the matplotlib plots to use the 'ggplot' style, which is a popular style known for its aesthetic appeal.

‘pd.set_option('display.max_columns', 100)’: This line sets the maximum number of columns to be displayed when printing pandas DataFrames. By setting it to 100, it ensures that all columns are shown when printing DataFrames.

Step 2: Import the dataset

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
train = pd.read_csv('/content/Train.csv')
train.head()
```

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Out: |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 | |

Step 3: count the occurrences of each unique value in the Item_Fat_Content column

```
train.Item_Fat_Content.value_counts()
```

```
Low Fat     5089
Regular     2889
LF           316
reg          117
low fat      112
Name: Item_Fat_Content, dtype: int64
```

Step 4: replace specific values in the train DataFrame with new values

```
train.replace({'LF':'Low Fat', 'reg' : 'Regular', 'low fat':'Low Fat'}, inplace = True)
```

Step 5: replace spaces with underscores in the categorical columns of the train DataFrame

```
cat_columns = train.select_dtypes('object').columns
train[cat_columns] = train[cat_columns].apply(lambda x: x.str.replace(' ', '_'))
train.head()
```

And here is result

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | FDA15 | 9.30 | Low_Fat | 0.016047 | Dairy | 249.8092 | OUT049 | 1999 | Me |
| 1 | DRC01 | 5.92 | Regular | 0.019278 | Soft_Drinks | 48.2692 | OUT018 | 2009 | Me |
| 2 | FDN15 | 17.50 | Low_Fat | 0.016760 | Meat | 141.6180 | OUT049 | 1999 | Me |
| 3 | FDX07 | 19.20 | Regular | 0.000000 | Fruits_and_Vegetables | 182.0950 | OUT010 | 1998 | |
| 4 | NCD19 | 8.93 | Low_Fat | 0.000000 | Household | 53.8614 | OUT013 | 1987 | |

Step 6: used to count the number of missing values (NaN) in each column of the train DataFrame.

```
train.isna().sum()
```

```
Item_Identifier                 0
Item_Weight                  1463
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
dtype: int64
```

Step 7: group the train DataFrame by the columns 'Outlet_Type' and 'Outlet_Size', and then count the occurrences of each unique combination of these two columns.

```
train.groupby(['Outlet_Type', 'Outlet_Size'], dropna = False).aggregate({'Outlet_Size':'size'}).unstack()
```

| | Outlet_Size | | | |
|---|---|---|---|---|
| **Outlet_Size** | **High** | **Medium** | **Small** | **NaN** |
| **Outlet_Type** | | | | |
| **Grocery_Store** | NaN | NaN | 528.0 | 555.0 |
| **Supermarket_Type1** | 932.0 | 930.0 | 1860.0 | 1855.0 |
| **Supermarket_Type2** | NaN | 928.0 | NaN | NaN |
| **Supermarket_Type3** | NaN | 935.0 | NaN | NaN |

All Grocery Store are small stores

Step 8: group the train DataFrame by the columns 'Outlet_Location_Type' and 'Outlet_Size', and then count the occurrences of each unique combination of these two columns.

```
train.groupby(['Outlet_Location_Type', 'Outlet_Size'], dropna = False).aggregate({'Outlet_Size':'size'}).unstack()
```

| | Outlet_Size | | | |
|---|---|---|---|---|
| **Outlet_Size** | **High** | **Medium** | **Small** | **NaN** |
| **Outlet_Location_Type** | | | | |
| **Tier_1** | NaN | 930.0 | 1458.0 | NaN |
| **Tier_2** | NaN | NaN | 930.0 | 1855.0 |
| **Tier_3** | 932.0 | 1863.0 | NaN | 555.0 |

All Tier 2 stores are small stores

Step 9: group the train DataFrame by the columns 'Outlet_Location_Type', 'Outlet_Type', and 'Outlet_Size', and then count the occurrences of each unique combination of these three columns.

```
train.groupby(['Outlet_Location_Type', 'Outlet_Type', 'Outlet_Size'], dropna = False).aggregate({'Outlet_Size':'size'}).unstack()
```

| | | Outlet_Size | | | |
|---|---|---|---|---|---|
| | **Outlet_Size** | **High** | **Medium** | **Small** | **NaN** |
| **Outlet_Location_Type** | **Outlet_Type** | | | | |
| **Tier_1** | **Grocery_Store** | NaN | NaN | 528.0 | NaN |
| | **Supermarket_Type1** | NaN | 930.0 | 930.0 | NaN |
| **Tier_2** | **Supermarket_Type1** | NaN | NaN | 930.0 | 1855.0 |
| **Tier_3** | **Grocery_Store** | NaN | NaN | NaN | 555.0 |
| | **Supermarket_Type1** | 932.0 | NaN | NaN | NaN |
| | **Supermarket_Type2** | NaN | 928.0 | NaN | NaN |
| | **Supermarket_Type3** | NaN | 935.0 | NaN | NaN |

Final Wordict: Impute all missing Outlet_Size as 'Small'

Step 10: fill the missing values in the 'Outlet_Size' column of the train DataFrame with the value 'Small'

```
train.Outlet_Size.fillna('Small', inplace = True)
```

Step 11: creates a new DataFrame that contains only the specified columns from the original train DataFrame because Item_Weight is depend on 'Item_Fat_Content', 'Item_Type'.

```
train[['Item_Weight', 'Item_Fat_Content', 'Item_Type']]
```

| | Item_Weight | Item_Fat_Content | Item_Type |
|---|---|---|---|
| **0** | 9.300 | Low_Fat | Dairy |
| **1** | 5.920 | Regular | Soft_Drinks |
| **2** | 17.500 | Low_Fat | Meat |
| **3** | 19.200 | Regular | Fruits_and_Vegetables |
| **4** | 8.930 | Low_Fat | Household |
| **...** | ... | ... | ... |
| **8518** | 6.865 | Low_Fat | Snack_Foods |
| **8519** | 8.380 | Regular | Baking_Goods |
| **8520** | 10.600 | Low_Fat | Health_and_Hygiene |
| **8521** | 7.210 | Regular | Snack_Foods |
| **8522** | 14.800 | Low_Fat | Soft_Drinks |

8523 rows × 3 columns

Step 12: grouping and aggregation operations on the train DataFrame based on the columns 'Item_Fat_Content' and 'Item_Type', calculating the mean of the 'Item_Weight' column for each unique combination of these two columns

```
weight_mask = train.groupby(['Item_Fat_Content', 'Item_Type']).aggregate({'Item_Weight':'mean'})
weight_mask.columns = ['Mean_Item_Weight']
weight_mask.reset_index(inplace=True)
weight_mask.head()
```

| | Item_Fat_Content | Item_Type | Mean_Item_Weight |
|---|---|---|---|
| 0 | Low_Fat | Baking_Goods | 12.552996 |
| 1 | Low_Fat | Breads | 12.429912 |
| 2 | Low_Fat | Breakfast | 11.849412 |
| 3 | Low_Fat | Canned | 11.864650 |
| 4 | Low_Fat | Dairy | 13.391497 |

Step 13: impute missing values in the 'Item_Weight' column of the train DataFrame based on the mean values calculated from the 'Item_Fat_Content' and 'Item_Type' groups

```
impute_weights = train[['Item_Weight', 'Item_Fat_Content', 'Item_Type']]
impute_weights = impute_weights[pd.isnull(impute_weights.Item_Weight)].\
                 merge(weight_mask, how = 'left', left_on = ['Item_Fat_Content', 'Item_Type'], right_on = ['Item_Fat_Content', 'Item_Type'])
impute_weights = impute_weights.Mean_Item_Weight
impute_weights
```

```
0       13.707177
1       11.400328
2       12.013303
3       12.552996
4       12.804289
          ...
1458    11.963444
1459    11.963444
1460    13.853285
1461    13.708363
1462    13.384736
Name: Mean_Item_Weight, Length: 1463, dtype: float64
```

Step 14: replace the missing values in the 'Item_Weight' column of the train DataFrame with the imputed values from the impute_weights variable.

```
train.loc[pd.isnull(train.Item_Weight), 'Item_Weight'] = impute_weights.values
```

Step 15: new column 'Item_Type_Combined' in the train DataFrame, which combines the first two characters of the 'Item_Identifier' column. It then maps specific values to the 'Item_Type_Combined' column based on the mapped dictionary. Finally, it counts the occurrences of each unique value in the 'Item_Type_Combined' column.

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

train['Item_Type_Combined'].value_counts()
```

```
Food              6125
Non-Consumable    1599
Drinks             799
Name: Item_Type_Combined, dtype: int64
```

Step 16: create a new DataFrame train_new by dropping the columns 'Item_Identifier', 'Outlet_Identifier', and 'Outlet_Establishment_Year' from the original train DataFrame because all of this is unnecessary features, that means it doesn’t affect the result of prediction.

```
train_new = train.drop(['Item_Identifier', 'Outlet_Identifier', 'Outlet_Establishment_Year'], axis=1)
train_new.head()
```

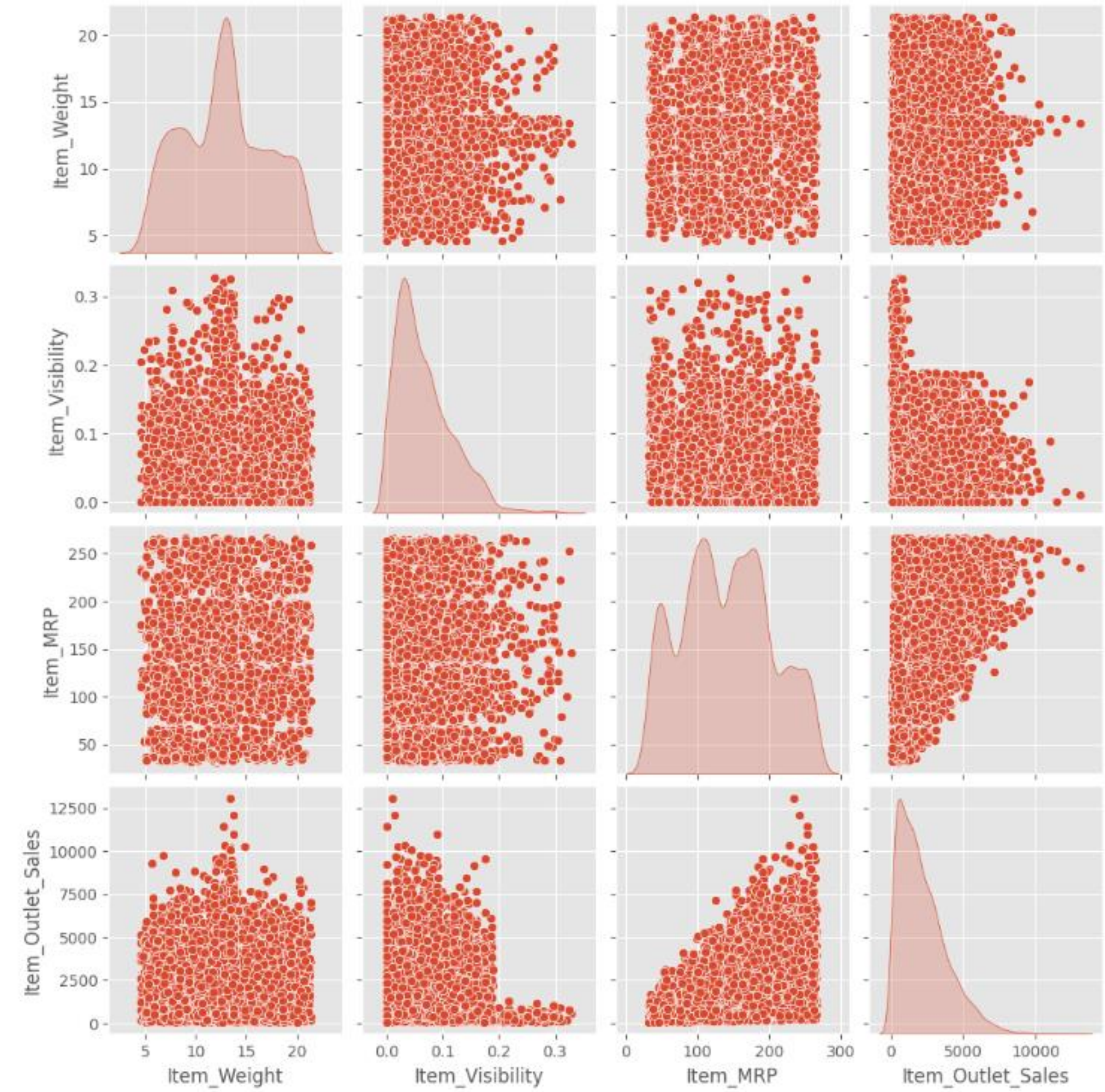
Here is a new table:

| | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Size | Outlet_Location_Type | Outlet_Type | Item_Outlet_Sale |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 9.30 | Low_Fat | 0.016047 | Dairy | 249.8092 | Medium | Tier_1 | Supermarket_Type1 | 3735.138 |
| 1 | 5.92 | Regular | 0.019278 | Soft_Drinks | 48.2692 | Medium | Tier_3 | Supermarket_Type2 | 443.422 |
| 2 | 17.50 | Low_Fat | 0.016760 | Meat | 141.6180 | Medium | Tier_1 | Supermarket_Type1 | 2097.270 |
| 3 | 19.20 | Regular | 0.000000 | Fruits_and_Vegetables | 182.0950 | Small | Tier_3 | Grocery_Store | 732.380 |
| 4 | 8.93 | Low_Fat | 0.000000 | Household | 53.8614 | High | Tier_3 | Supermarket_Type1 | 994.705 |

Step 17: create a pairplot visualization of the train_new DataFrame using seaborn (sns)

```
sns.pairplot(train_new, diag_kind='kde')
plt.show()
```

Here is result of pairplot



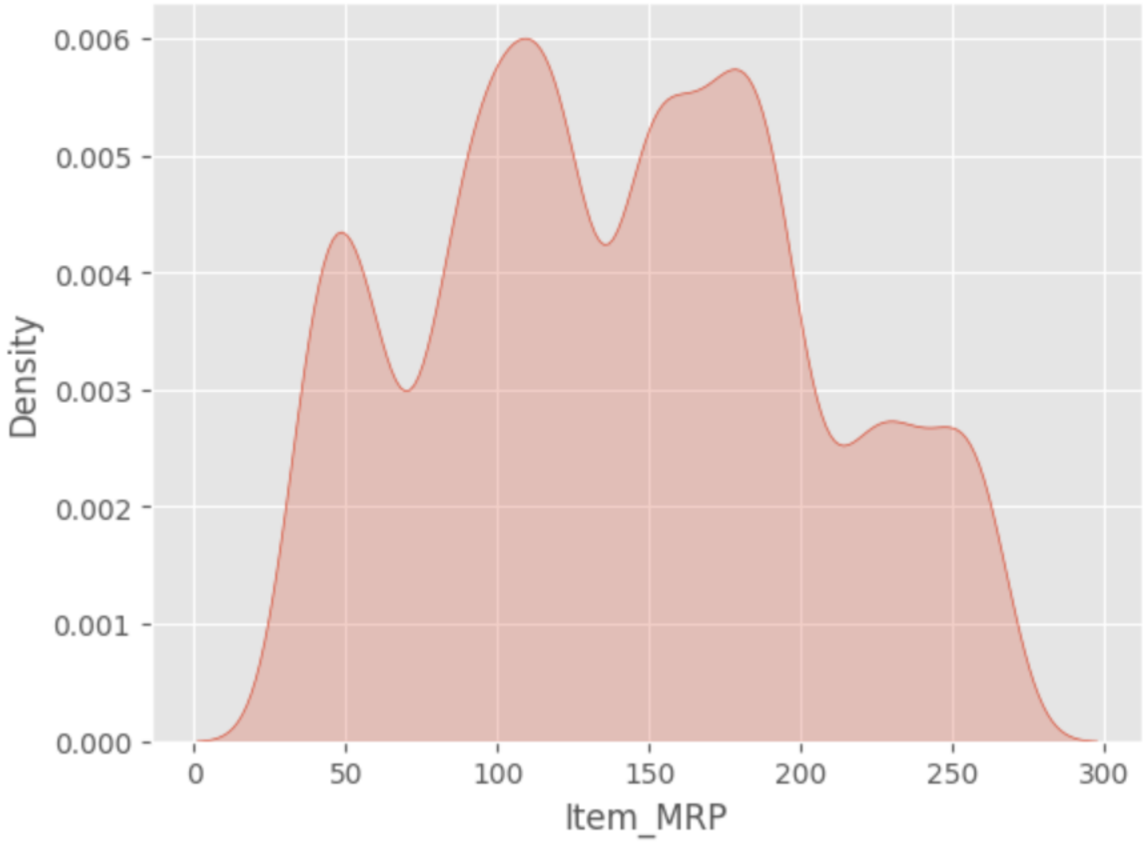

Mostly all features have uniform distribution, seems no relation in particular

Sales of items having higher Item_Visibility > 0.2, Sales tend to be lower

Item_MRP and Item_Visibility might be good candidate for predicting Item_Outlet_Sales.

Step 18: plot a kernel density estimate (KDE) plot for the column 'Item_MRP' in the DataFrame 'train_new' using the seaborn library (sns)

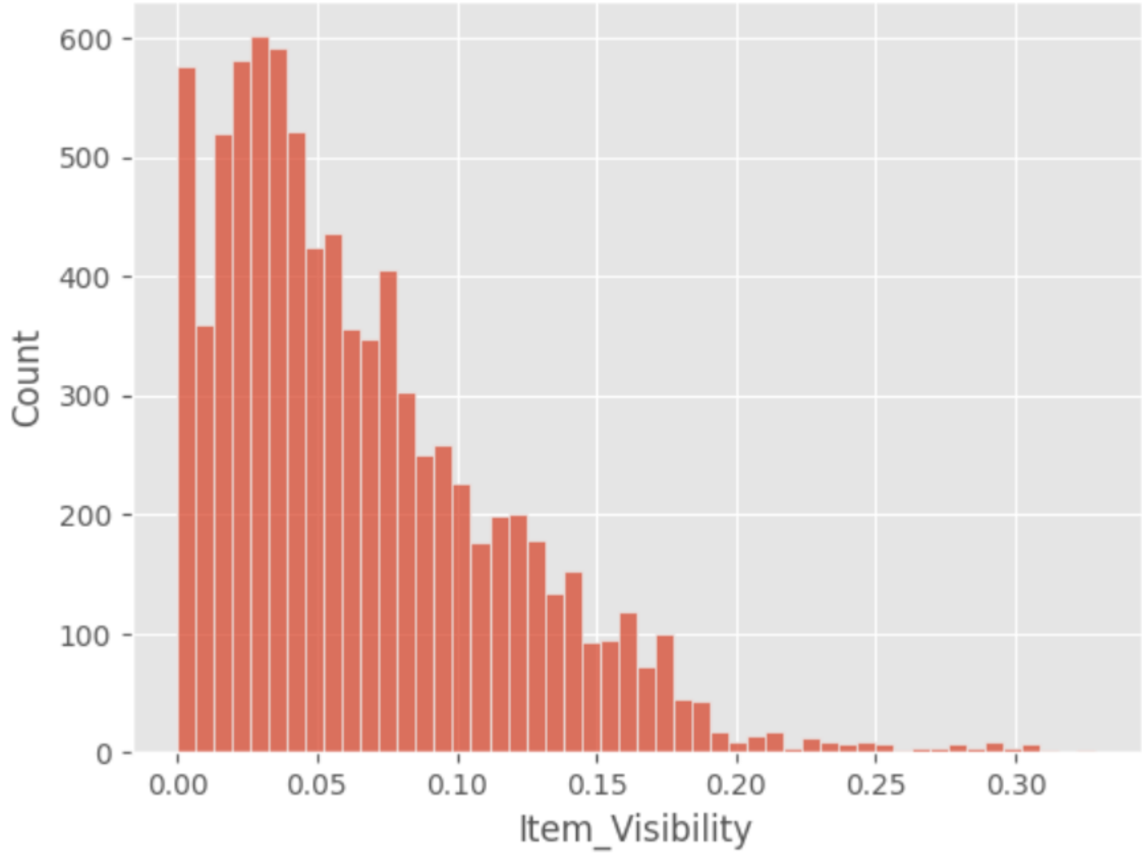
```
sns.kdeplot(train_new.Item_MRP, fill=True)
plt.show()
```




Multi-modal plot - Can be grouped

Step 19: plot a histogram for the column 'Item_Visibility' in the DataFrame 'train_new' using the seaborn library (sns).

```
sns.histplot(train_new.Item_Visibility)
plt.show()
```




Mostly Items have visibility < 0.1

Step 20: split the DataFrame train_new into the feature set X and the target variable y

```
X = train_new.drop('Item_Outlet_Sales', axis = 1)
y = train_new['Item_Outlet_Sales']
```

```
X.dtypes
```

```
Item_Weight                float64
Item_Fat_Content            object
Item_Visibility            float64
Item_Type                   object
Item_MRP                   float64
Outlet_Size                 object
Outlet_Location_Type        object
Outlet_Type                 object
Item_Type_Combined          object
dtype: object
```

---

After drop the column Item_Outlet_Sales, before, we also drop 3 column: 'Item_Identifier', 'Outlet_Identifier', 'Outlet_Establishment_Year' because all of them are unnecessary features. Therefore, after dropping 4 column, combing Item_Type_Combine created, we have total 9 column in new train dataset.

Step 21: separate the numeric (continuous) and categorical columns from the feature set X.

```
num_columns = X.select_dtypes(['int', 'float']).columns
cat_columns = X.select_dtypes(['object']).columns
num_columns, cat_columns
```

```
(Index(['Item_Weight', 'Item_Visibility', 'Item_MRP'], dtype='object'),
 Index(['Item_Fat_Content', 'Item_Type', 'Outlet_Size', 'Outlet_Location_Type',
        'Outlet_Type', 'Item_Type_Combined'],
       dtype='object'))
```

Step 22: split the feature set X and the target variable y into training and testing sets with ratio is 80:20

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 23: Import tensorflow

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/compat/v2_compat.py:107: disable_resource_variables (from tensorflow.
Instructions for updating:
non-resource variables are not supported in the long term
```

Step 24: create feature columns for a TensorFlow model based on categorical and numerical columns in the dataset

```
feature_columns = []

for feature_name in cat_columns:
    vocabulary = X[feature_name].unique()
    feature_col = tf.feature_column.categorical_column_with_vocabulary_list(feature_name, vocabulary)
    feature_col_embed = tf.feature_column.embedding_column(feature_col, dimension = 5)
    feature_columns.append(feature_col_embed)

for feature_name in num_columns:
    feature_columns.append(tf.feature_column.numeric_column(feature_name, dtype = tf.float32))
```

```
WARNING:tensorflow:From <ipython-input-26-8bb82c0775da>:5: categorical_column_with_vocabulary_list (from tensorflow.python.feature_column.feature_colu
Instructions for updating:
Use Keras preprocessing layers instead, either directly or via the `tf.keras.utils.FeatureSpace` utility. Each of `tf.feature_column.*` has a function
WARNING:tensorflow:From <ipython-input-26-8bb82c0775da>:6: embedding_column (from tensorflow.python.feature_column.feature_column_v2) is deprecated an
Instructions for updating:
Use Keras preprocessing layers instead, either directly or via the `tf.keras.utils.FeatureSpace` utility. Each of `tf.feature_column.*` has a function
WARNING:tensorflow:From <ipython-input-26-8bb82c0775da>:10: numeric_column (from tensorflow.python.feature_column.feature_column_v2) is deprecated and
Instructions for updating:
Use Keras preprocessing layers instead, either directly or via the `tf.keras.utils.FeatureSpace` utility. Each of `tf.feature_column.*` has a function
```

Step 25: create input functions for TensorFlow models based on the provided data and labels

```
def make_input_fn(data_df, label_df, num_echos = 10, batch_size = 32, shuffle = True):
    def input_function():
        ds = tf.data.Dataset.from_tensor_slices((dict(data_df), label_df))
        if shuffle:
            ds = ds.shuffle(1000)
        ds = ds.batch(batch_size).repeat(num_echos)
        return ds
    return input_function
train_input_fn = make_input_fn(X_train, y_train)
test_input_fn = make_input_fn(X_test, y_test, num_echos = 1, shuffle = False)
```

Step 26: build a regression model using a multi-layer neural network (DNN)

```
DNNmodel = tf.estimator.DNNRegressor(
    feature_columns = feature_columns,
    hidden_units = [30, 15, 10, 15, 30],
    optimizer = 'Adam',
    activation_fn = tf.nn.relu
)
```

```
WARNING:tensorflow:From <ipython-input-28-ee5c900dcbdb>:1: DNNRegressor.__init__ (from tensorflow_estimator.python.estimator.canned.dnn) is deprecated
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/canned/dnn.py:1221: Estimator.__init__ (from ten:
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/estimator.py:1842: RunConfig.__init__ (from tens
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:Using temporary folder as model directory: /tmp/tmpn74jtr3t
```

Step 27: train a model

```
DNNmodel.train(input_fn = train_input_fn, steps = 3000)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/estimator.py:385: StopAtStepHook.__init__ (from
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/training_util.py:396: Variable.initialized_value (from ten
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun) contexts.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/canned/dnn.py:446: dnn_logit_fn_builder (from t
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/model_fn.py:250: EstimatorSpec.__new__ (from te
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/estimator.py:1414: NanTensorHook.__init__ (from
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/estimator.py:1417: LoggingTensorHook.__init__ (
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/basic_session_run_hooks.py:232: SecondOrStepTimer.__init__
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/estimator.py:1454: CheckpointSaverHook.__init__
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/monitored_session.py:579: StepCounterHook.__init__ (from t
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/monitored_session.py:586: SummarySaverHook.__init__ (from
Instructions for updating:
```

Step 28: evaluate a trained model on a specified dataset

```
result = DNNmodel.evaluate(test_input_fn)
result
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow/python/training/evaluation.py:260: FinalOpsHook.__init__ (from tensorflow.py
Instructions for updating:
Use tf.keras instead.
{'average_loss': 1108194.0,
 'label/mean': 2097.008,
 'loss': 34990200.0,
 'prediction/mean': 2289.9246,
 'global_step': 2140}
```

Step 29: calculates several evaluation metrics for the DNN model predictions using the sklearn.metrics module

```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error, median_absolute_error

y_pred = DNNmodel.predict(test_input_fn)
y_predict = [elem['predictions'][0] for elem in y_pred]

r2 = r2_score(y_test, y_predict)
mse = mean_squared_error(y_test, y_predict)
rmse = mean_squared_error(y_test, y_predict, squared=False)
mae = mean_absolute_error(y_test, y_predict)
mdae = median_absolute_error(y_test, y_predict)

print('The evaluation scores for the DNN model:')
print('R2:', r2)
print('MSE:', mse)
print('RMSE:', rmse)
print('MAE:', mae)
print('MDAE:', mdae)
print('\n')
```

```
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/canned/head.py:1583: RegressionOutput.__init__ (
Instructions for updating:
Use tf.keras instead.
WARNING:tensorflow:From /usr/local/lib/python3.10/dist-packages/tensorflow_estimator/python/estimator/canned/head.py:1591: PredictOutput.__init__ (fro
Instructions for updating:
Use tf.keras instead.
The evaluation scores for the DNN model:
R2: 0.5922714589302618
MSE: 1108193.9699328553
RMSE: 1052.7079224233355
MAE: 750.2956788281107
MDAE: 529.0412925781252
```

## NAM Model

### Defintion

Neural Additive Models (NAMs) combine some of the expressivity of DNNs with the inherent intelligibility of generalized additive models. NAMs learn a linear combination of neural networks that each attend to a single input feature1.

### How NAM model work?

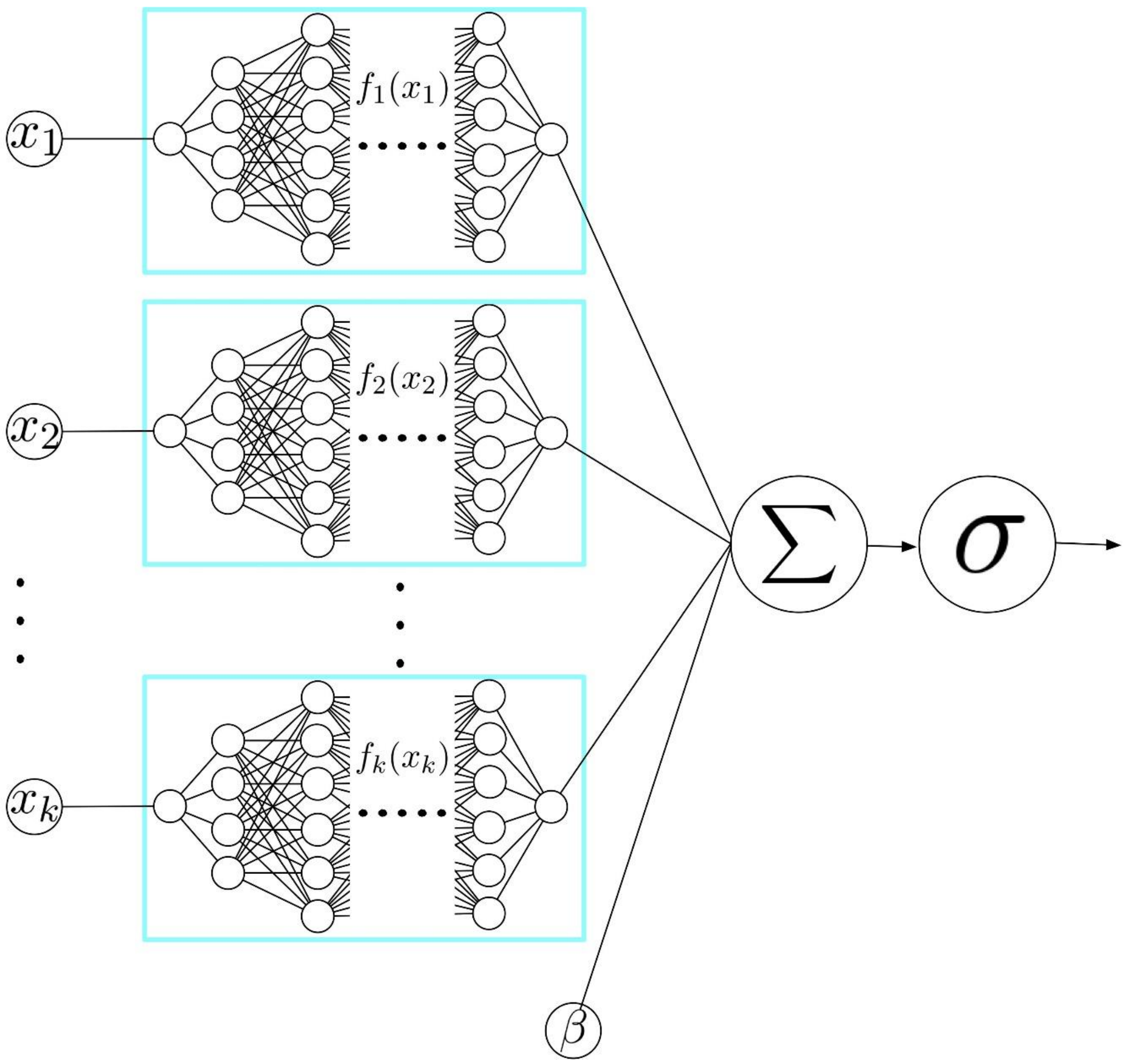



Figure: How NAM work?[4]

Neural Additive Models (NAMs) which combine some of the expressivity of DNNs with the inherent intelligibility of generalized additive models. NAMs learn a linear combination of neural networks that each attend to a single input feature. These networks are trained jointly and can learn arbitrarily complex relationships between their input feature and the output.

## Explaing the code

1. Import required library:

```
In [28]:
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import seaborn as sns
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
```

2. Load dataset:

In [2]:
```
# Load the dataset
data = pd.read_csv('train_data.csv')
data.head()
```

Out[2]:

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Outlet_Loc |
|---|---|---|---|---|---|---|---|---|---|---|
| 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 | |

3. Check for missing value in dataset:

In [3]:
```
# Check for missing values in the dataset
print(data.isnull().sum())
```

```
Item_Identifier                 0
Item_Weight                  1463
Item_Fat_Content                0
Item_Visibility                 0
Item_Type                       0
Item_MRP                        0
Outlet_Identifier               0
Outlet_Establishment_Year       0
Outlet_Size                  2410
Outlet_Location_Type            0
Outlet_Type                     0
Item_Outlet_Sales               0
dtype: int64
```

Item_Idenfitier and Outlet_Size have 1463 and 2410 null value.

4. Filling the missing values in "Item_weight column" with "Mean" value:

In [4]:
```
# filling the missing values in "Item_weight column" with "Mean" value
data['Item_Weight'].fillna(data['Item_Weight'].mean(), inplace=True)
```

5. Filling the missing values in "Outlet_Size" column with Mode:

In [5]:
```
# filling the missing values in "Outlet_Size" column with Mode
#Here we take Outlet_Size column & Outlet_Type column since they are correlated
mode_of_Outlet_size = data.pivot_table(values='Outlet_Size', columns='Outlet_Type', aggfunc=(lambda x: x.mode()[0]))
```

In [6]:
```
miss_values = data['Outlet_Size'].isnull()
```

In [7]:
```
data.loc[miss_values, 'Outlet_Size'] = data.loc[miss_values,'Outlet_Type'].apply(lambda x: mode_of_Outlet_size[x])
```

6. Check for missing value again:

In [8]:
```
# checking for missing valua
data.isnull().sum()
```

Out[8]:
```
Item_Identifier              0
Item_Weight                  0
Item_Fat_Content             0
Item_Visibility              0
Item_Type                    0
Item_MRP                     0
Outlet_Identifier            0
Outlet_Establishment_Year    0
Outlet_Size                  0
Outlet_Location_Type         0
Outlet_Type                  0
Item_Outlet_Sales            0
dtype: int64
```

## 7. Checking for similar words:

In [9]:
```
# check for similar words
data['Item_Fat_Content'].value_counts()
```

Out[9]:
```
Low Fat     5089
Regular     2889
LF           316
reg          117
low fat      112
Name: Item_Fat_Content, dtype: int64
```

## 8. Replace similar words with Low Fat and Regular only:

In [10]:
```
# replace similar words
data.replace({'Item_Fat_Content': {'low fat':'Low Fat','LF':'Low Fat', 'reg':'Regular'}}, inplace=True)
```

## 9. Checking for similar word again:

In [11]:
```
data['Item_Fat_Content'].value_counts()
```

Out[11]:
```
Low Fat     5517
Regular     3006
Name: Item_Fat_Content, dtype: int64
```

## 10.Convertion of the labels into a numeric form so as to convert them into the machine-readable form:

In [12]:
```
encoder = LabelEncoder()
```

In [13]:
```
data['Item_Identifier'] = encoder.fit_transform(data['Item_Identifier'])

data['Item_Fat_Content'] = encoder.fit_transform(data['Item_Fat_Content'])

data['Item_Type'] = encoder.fit_transform(data['Item_Type'])

data['Outlet_Identifier'] = encoder.fit_transform(data['Outlet_Identifier'])

data['Outlet_Size'] = encoder.fit_transform(data['Outlet_Size'])

data['Outlet_Location_Type'] = encoder.fit_transform(data['Outlet_Location_Type'])

data['Outlet_Type'] = encoder.fit_transform(data['Outlet_Type'])
```

## 11.Dataset after processing:

In [14]:
```
data.head()
```

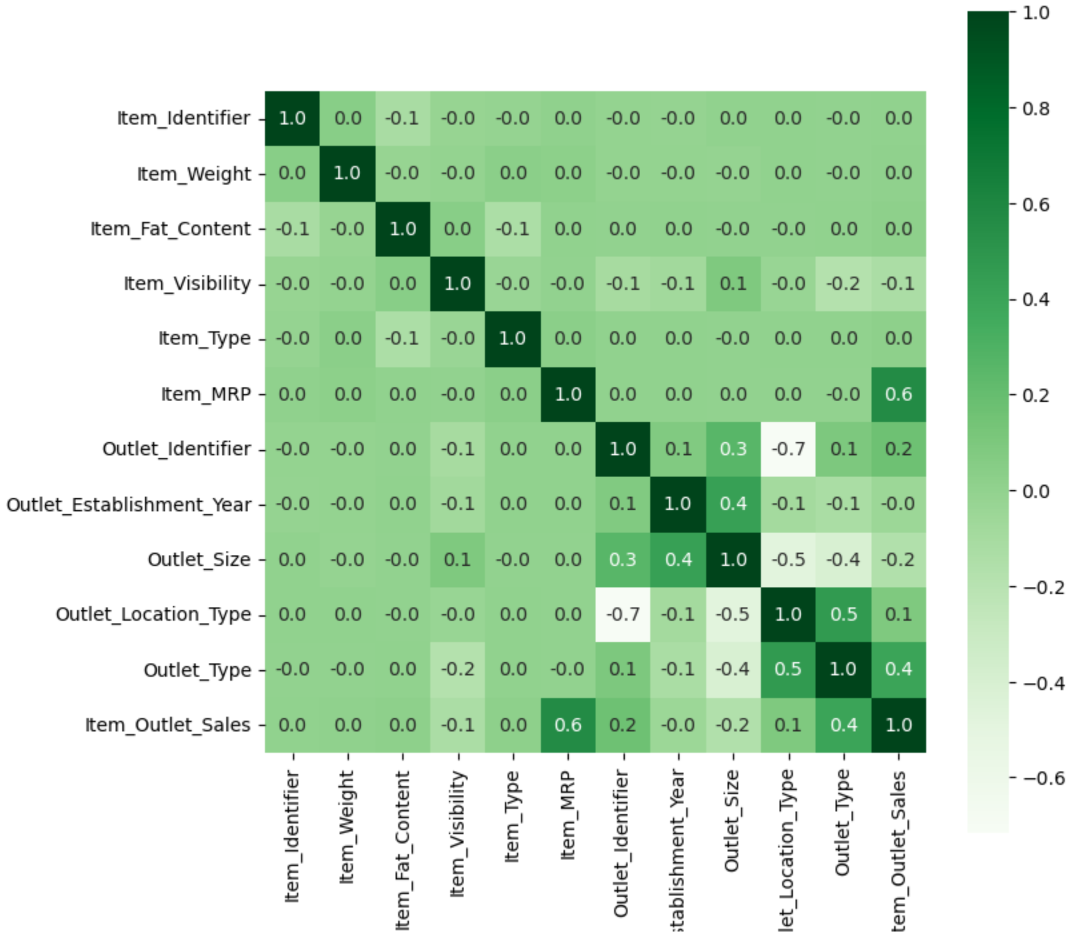
Out[14]:

| | Item_Identifier | Item_Weight | Item_Fat_Content | Item_Visibility | Item_Type | Item_MRP | Outlet_Identifier | Outlet_Establishment_Year | Outlet_Size | Outlet_Loca |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 156 | 9.30 | 0 | 0.016047 | 4 | 249.8092 | 9 | 1999 | 1 | |
| **1** | 8 | 5.92 | 1 | 0.019278 | 14 | 48.2692 | 3 | 2009 | 1 | |
| **2** | 662 | 17.50 | 0 | 0.016760 | 10 | 141.6180 | 9 | 1999 | 1 | |
| **3** | 1121 | 19.20 | 1 | 0.000000 | 6 | 182.0950 | 0 | 1998 | 2 | |
| **4** | 1297 | 8.93 | 0 | 0.000000 | 9 | 53.8614 | 1 | 1987 | 0 | |

## 12.Heat map of the dataset:

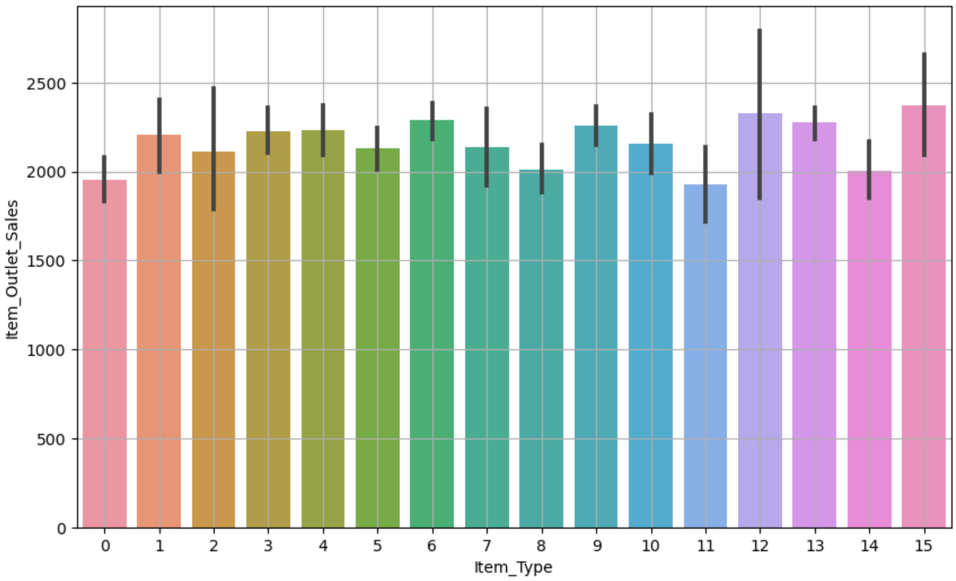
In [29]:
```
corr = data.corr()
plt.figure(figsize=(8,8))
sns.heatmap(corr,cbar=True,square=True,fmt='.1f',annot=True,cmap='Greens')
```

Out[29]: <Axes: >




# 13.Bar plot for the dataset:

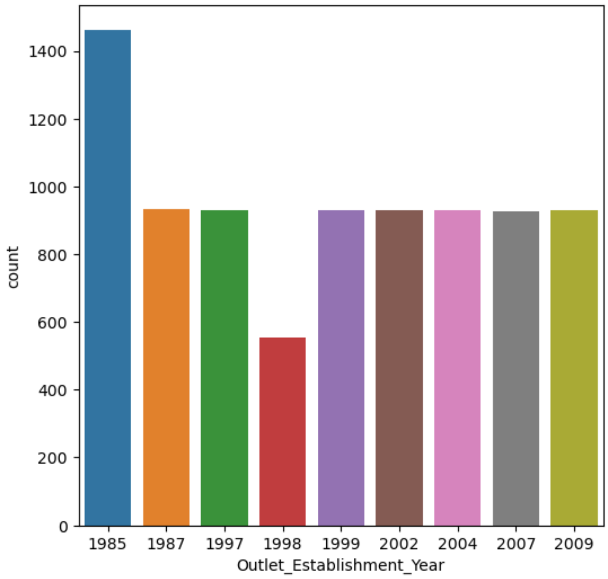
In [31]:
```
plt.figure(figsize=(10,6))
sns.barplot(x='Item_Type', y='Item_Outlet_Sales', data = data)
plt.grid()
```




# 14.Count plot for Outlet_Establishment_Year column of the dataset:

```
In [33]: plt.figure(figsize=(6,6))
         sns.countplot(x='Outlet_Establishment_Year', data = data)
```

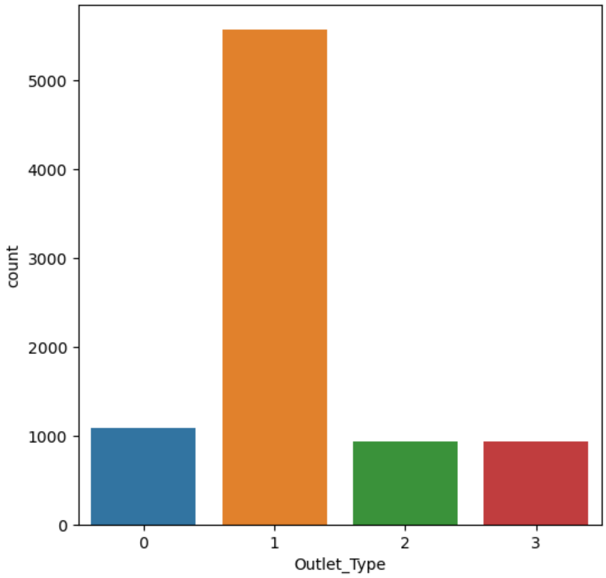
```
Out[33]: <Axes: xlabel='Outlet_Establishment_Year', ylabel='count'>
```




15.Count plot for Outlet_type column of the dataset:

```
In [35]: plt.figure(figsize=(6,6))
         sns.countplot(x="Outlet_Type", data = data)
```

```
Out[35]: <Axes: xlabel='Outlet_Type', ylabel='count'>
```




16.Preprocess the data:

```
In [15]: # Preprocess the data
         X = data.drop('Item_Outlet_Sales', axis=1)
         y = data['Item_Outlet_Sales']
         scaler = StandardScaler()
         X = scaler.fit_transform(X)
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Separate the independent variables (features) by dropping the Item_Outlet_Sales (target).

- Assign the dependent variable (target) to the variable y_train.
- Create an instance of StandardScaler from scikit-learn to scale the numerical features.
- Scales the numerical features in the training data using the fitted scaler.
- Split data to train.

17.Convert the data to PyTorch tensors:

```
In [16]:  # Convert the data to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train.values.reshape(-1, 1), dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
```

In machine learning frameworks like PyTorch, data is typically represented as tensors. Tensors are multi-dimensional arrays that can store numerical data, and they are the fundamental data structure used for computations in PyTorch.

18.Define the Neural Additive Model (NAM) architecture:

```
In [17]:  # Define the Neural Additive Model (NAM) architecture
class NAM(nn.Module):
    def __init__(self, input_dim):
        super(NAM, self).__init__()
        self.linear = nn.Linear(input_dim, 1)
        self.nonlinear = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1)
        )

    def forward(self, x):
        linear_part = self.linear(x)
        nonlinear_part = self.nonlinear(x)
        output = linear_part + nonlinear_part
        return output
```

19.Create an instance of the NAM model:

```
In [18]:  # Create an instance of the NAM model
input_dim = X_train.shape[1]
model = NAM(input_dim)
```

Create an instance of the NAM model to initialize the model, configure its architecture and parameters, and use it for training or prediction tasks.

20.Define the loss function and optimizer:

```
In [19]:  # Define the loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- The loss function measures the inconsistency between the predicted output and the true target values.
- The optimizer is responsible for updating the model's parameters based on the computed loss.

21.Train the NAM model:

```
In [20]: # Train the NAM model
num_epochs = 100
batch_size = 32
for epoch in range(num_epochs):
    model.train()
    permutation = torch.randperm(X_train.size()[0])
    for i in range(0, X_train.size()[0], batch_size):
        indices = permutation[i:i+batch_size]
        batch_x, batch_y = X_train[indices], y_train[indices]

        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
```

## 22.Evaluate NAM model:

```
In [21]: # Evaluate the NAM model
model.eval()
with torch.no_grad():
    y_pred = model(X_test).numpy()
    r2 = r2_score(y_test, y_pred)
    print('R-squared:', r2)

R-squared: 0.6111445468539286
```

```
In [22]: print('The evalution scores XGBoost: ')
r2 = metrics.r2_score(y_test, y_pred)
mse = metrics.mean_squared_error(y_test, y_pred)
rmse = mse ** 0.5
mae = metrics.mean_absolute_error(y_test, y_pred)
mdae = metrics.median_absolute_error(y_test, y_pred)
print('R2: ', r2)
print('MSE :', mse)
print('RMSE: ', rmse)
print('MAE: ', mae)
print('MDAE: ', mdae)
print('\n')

The evalution scores XGBoost:
R2:  0.6111445468539286
MSE : 1056897.4818917033
RMSE:  1028.0551939909176
MAE:  728.6469746496747
MDAE:  502.82359101562497
```

Evaluate NAM model using R2, MSE, RMSE, MAE, MDAE

## Result

| | R2 | MSE | RMSE | MAE | MDAE |
|---|---|---|---|---|---|
| XGBoost | 0.5191234777241828 | 1484501.750774029 | 1484501.750774029 | 856.8716943000217 | 576.9436505859376 |
| ANN | 0.3381651964239938 | 0.01152943890524462 | 0.10737522482046136 | 0.07606321682762442 | 0.05088797999646821 |
| DNN | 0.5922714589302618 | 1108193.9699328553 | 1108193.9699328553 | 750.2956788281107 | 529.0412925781252 |
| NAM | 0.6111445468539286 | 1056897.4818917033 | 1028.0551939909176 | 728.6469746496747 | 502.82359101562497 |

## Conclusion

Look at the results of all prediction error, ANN has the best model with the smallest error of all 5 forecasting error.

## Link of the dataset and paper

Dataset: https://www.kaggle.com/datasets/brijbhushannanda1979/bigmart-sales-data/code

Paper: http://ir.juit.ac.in:8080/jspui/bitstream/123456789/3600/1/Big%20Mart%20Sales%20Prediction%20Using%20Machine%20Learning.pdf

## Reference


[1] https://www.kaggle.com/kaggle-survey-2021

[2] https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost-HowItWorks.html

[3] https://www.bmc.com/blogs/deep-neural-network/

[4] https://neural-additive-models.github.io/


## Link of Video and Model

### Video

ANN: https://www.youtube.com/watch?v=MHcFyaQ5ZzI&feature=youtu.be

DNN: https://www.youtube.com/watch?v=EoCiMarbsW8

NAM: https://www.youtube.com/watch?v=wnBkBEl4-BM

XGBoost:
https://www.youtube.com/watch?v=mNWL6FMjVrc&feature=youtu.be

Slide: https://uithcm.sharepoint.com/:v:/s/Sakura/EZatZ-dIhjJBmPPSXdGYxZoBFeRRbzn7im8xLf_LO-NRww?e=3NJ3Pm

Model

Drive: https://drive.google.com/drive/folders/1RwRakVUq-0GcutIjPfZ8CbPe6SK6Sv0C?usp=sharing