Big Mart Sales Prediction Datasets

Alžbeta Valachová

August 4, 2024

1 Introduction

This paper is my solution of prediction sales data. The assignment stands for:

The data scientists at BigMart have collected 2013 sales data for 1559 products across 10 stores in different cities. Also, certain attributes of each product and store have been defined. The aim is to build a predictive model and predict the sales of each product at a particular outlet.

Using this model, BigMart will try to understand the properties of products and outlets which play a key role in increasing sales.

We have a train (8523) and test (5681) data set, the train data set has both input and output variable(s). You need to predict the sales for the test data set.

Train file: CSV containing the item outlet information with a sales value

Test file: CSV containing item outlet combinations for which sales need to be forecasted. [1]

2 Analysis of the data

Generally, the solution of the problem is to prepare a model for predicting sales. Here are the columns of both sets:

Column Name	**Data Type**
Item_Identifier	object
Item_Weight	float64
Item_Fat_Content	object
Item_Visibility	float64
Item_Type	object
Item_MRP	float64
Outlet_Identifier	object
Outlet_Establishment_Year	int64
Outlet_Size	object
Outlet_Location_Type	object
Outlet_Type	object
Item_Outlet_Sales	float64

Table 1: Dataset Attributes and Their Data Types

Item_Outlet_Sales data could be predicted from input values - all the other columns. Every item has identifier and some features that could be associated with how well are they selling - some of them more than others.

In order to use any machine learning model, it's usefull to check if there are some missing values in the datasets:

Column Name	**Number of NaN values**
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

Table 2: Number of NaN values in train set

Column Name	**Number of NaN values**
Item_Identifier	0
Item_Weight	976
Item_Fat_Content	0
Item_Visibility	0
Item_Type	0
Item_MRP	0
Outlet_Identifier	0
Outlet_Establishment_Year	0
Outlet_Size	1606
Outlet_Location_Type	0
Outlet_Type	0

Table 3: Number of NaN values in test set

By looking at the actual data, there are two problems: the column Item_Weight contains some NaN values. It's important to say that there are multiple rows with the same Item_Identifier, so it's possible, that some of the values can be in different rows. By filtering the data and using XLOOKUP function in excel, the lost weight values are found in different rows and the problem is solved. However, this logic cannot be used, if the data is actually missing. In that case we can use some kind of approximation, for example sorting items by their Item_Type and using mean value of the Item_Weight from the same type.

Second problem is missing Outlet_Size of three out of ten outlets - Outlet10, Outlet17, Outlet45. This information cannot be found in either test or train dataset. The intuitive approach is to calculate turnover of all the Outlets and then defining the size with the most similar Outlet in the meaning of the turnover.

After those two fixes, the dataset is complete.

The deeper analysis could contain prioritizing some features from the columns. Interesting thought is to remove some data - for example the ones that have 0 Item_Visibility in the stores, because the products that cannot be seen by customers maybe wouldn't have to have the same conditions for the predictions.

We could also divide the products by their Item_Type and train models separately. There are a lot of extra steps that could be made in order to make the solution more effective.

2.1 Prediction

The first approach is to use a python package tensorflow for deep learning model built with Keras, a high-level neural networks API.

In order to use this model, first it's important to prepare the data. The data is split into input features (X) and the target variable (y). The target variable y is the Item_Outlet_Sales column, while the input features X are all the other columns. The training data is further split into a training set (X_train, y_train) and a validation set (X_test, y_test) using an 80-20 split. This helps evaluate the model's performance on unseen data. The numerical and categorical features are identified in both the training and test datasets. This helps in applying the appropriate transformations to these features.

A pipeline is created for numerical features using StandardScaler, which standardizes the features by removing the mean and scaling to unit variance. Encoding Categorical Features: A pipeline is created for categorical features using OneHotEncoder, which converts categorical values into binary (0/1) encoding, handling any unknown categories during transformation.

The ColumnTransformer applies the numerical and categorical pipelines to the respective features in both the training and test datasets, ensuring consistent preprocessing.

```
original_test = test[['Item_Identifier', 'Outlet_Identifier']].copy()

# Identifying predictors and target variable for train
X = train.drop(columns=['Item_Outlet_Sales']) #input
y = train['Item_Outlet_Sales'] #output

# Splitting data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

# Identifying numerical and categorical features in the training set
numerical_features_train = X_train.select_dtypes(include=['int64',
    'float64']).columns
categorical_features_train =
    X_train.select_dtypes(include=['object']).columns

```

```
# Identifying numerical and categorical features in the test set
numerical_features_test = test.select_dtypes(include=['int64',
    'float64']).columns
categorical_features_test =
    test.select_dtypes(include=['object']).columns

# Union of numerical and categorical features from both sets
numerical_features =
    numerical_features_train.union(numerical_features_test)
categorical_features =
    categorical_features_train.union(categorical_features_test)


# Scaling numerical values
numerical_transformer = Pipeline(steps=[
('scaler', StandardScaler())
])

# Encoding categorical values
categorical_transformer = Pipeline(steps=[
('onehot', OneHotEncoder(handle_unknown='ignore'))
])

# Creating ColumnTransformer
preprocessor = ColumnTransformer(
transformers=[
('num', numerical_transformer, numerical_features),
('cat', categorical_transformer, categorical_features)
])
```

Listing 1: Python code for sales prediction model

After the preparation is done, the model is defined: A sequential model with several dense (fully connected) layers:

- Input Layer: The first layer has 256 neurons and uses the ReLU activation function.
- Hidden Layers: There are two additional hidden layers with 128 and 64 neurons, both using ReLU.
- Output Layer: The output layer has a single neuron with ReLU activation, which outputs the predicted sales value.

The model is compiled with the Adam optimizer, which is an adaptive learning rate optimization algorithm. The loss function used is Mean Squared Error (MSE), which is minimized during training. The model also tracks Mean Absolute Error (MAE) as a metric for evaluation.

```
# Applying preprocessor to data
X_train = preprocessor.fit_transform(X_train)
X_test = preprocessor.transform(X_test)
test = preprocessor.transform(test)

# Creating the model
model = keras.Sequential([
layers.Dense(256, activation='relu', input_shape=[X_train.shape[1]]),
layers.Dense(128, activation='relu'),
layers.Dense(64, activation='relu'),
layers.Dense(1, activation='relu')
])

# Compiling the model
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Training the model
history = model.fit(X_train, y_train, epochs=100, validation_split=0.2,
    verbose=1)

# Evaluating the model
test_loss, test_mae = model.evaluate(X_test, y_test, verbose=1)
print(f'Test MAE: {test_mae}')

# Making predictions
y_pred = model.predict(X_test)

# Calculating RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print(f'Test RMSE: {rmse}')

print(f'Minimum sales value: {y.min()}')
print(f'Maximum sales value: {y.max()}')

# Making predictions on submission data
SUB_predictions = model.predict(test)
```

Listing 2: Python code for sales prediction model

3 Results

Here's a part of the output - we can see that during the training of the model is the loss on training and validation set getting smaller. The final RMSE (Root Mean Square Error) of the trained model is 1339.6, which is quite a big number considering the minimal and maximal sales value. However, we were predicting the data of various types of items and we had limited number of features. To make this error smaller, we could use more predictive models or skipping some data.

```
    171/171 --------------------- 1s 7ms/step - loss: 852563.0000 -
        mae: 625.8137 - val_loss: 1421992.1250 - val_mae: 859.2656
    Epoch 10/100
    171/171 --------------------- 1s 7ms/step - loss: 844244.6250 -
        mae: 626.9360 - val_loss: 1359064.8750 - val_mae: 829.7415
    Epoch 11/100
    171/171 --------------------- 1s 5ms/step - loss: 804796.8750 -
        mae: 614.2317 - val_loss: 1429558.1250 - val_mae: 856.2762
    Epoch 12/100
    171/171 --------------------- 1s 5ms/step - loss: 757427.1250 -
        mae: 593.7103 - val_loss: 1447694.1250 - val_mae: 858.6331
    Epoch 13/100
    ...
    Test RMSE: 1339.5981088534643
    Min sales value: 33.29
    Max sales value: 13086.9648
```

Listing 3: Python code for sales prediction model

Comparing the real and predicted sales values, we get following graph:

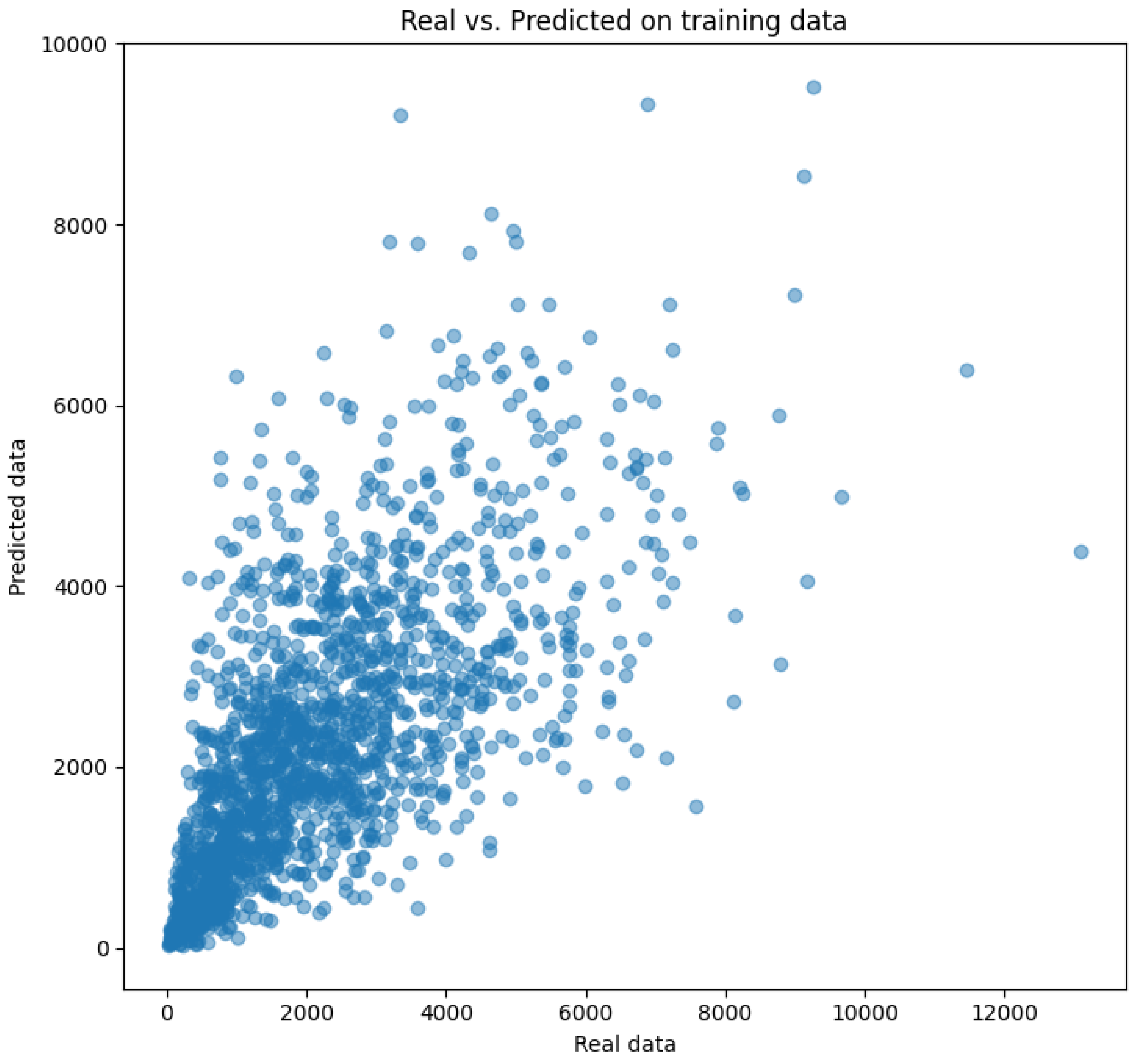

Figure 1: Output

References

[1] Shivan: *Big Mart Sales Prediction Datasets.* URL: https://www.kaggle.com/datasets/shivan118/big-mart-sales-prediction-datasets/data. [Accessed: 2024]