HW4 - Part B.

Class: ISOM-672-4102: Intro to Business Analytics - Fall 2023
Member: Chenli Zhou, Jiatong Song, Pamela Cheng, Shaonan Wang, Vishal Visha

---

(145 points) Use numeric prediction techniques to build a predictive model for the HW4.xlsx dataset. This dataset is provided on Canvas and contains data about whether or not different consumers made a purchase in response to a test mailing of a certain catalog and, in case of a purchase, how much money each consumer spent. The data file has a brief description of all the attributes in a separate worksheet. We would like to build predictive models to predict how much will the customers spend; Spending is the target variable (numeric value: amount spent).
Use Python for this exercise.
<u>Whenever applicable use random state 42 (10 points).</u>

(a) **(50 points) After exploring the data, build numeric prediction models that predict Spending. Use linear regression, k-NN, and regression tree techniques. Briefly discuss the models you have built. Use cross-validation with 10 folds to estimate the generalization performance. Present the results for each of the three techniques and discuss which one yields the best performance.**

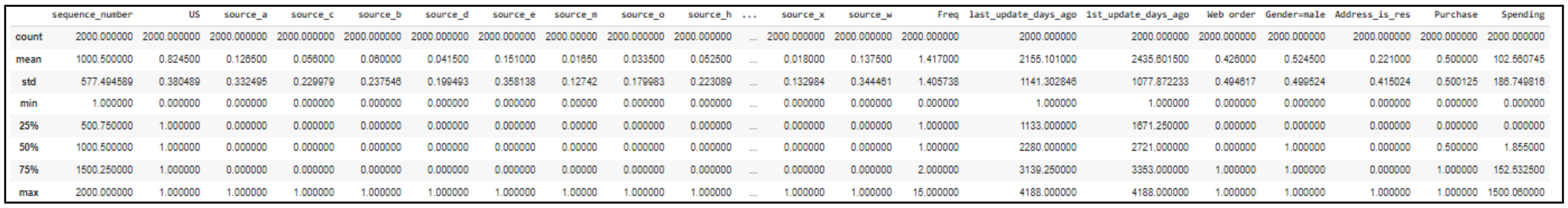
Exploring Our Data:

After importing our data, we have to understand the meaning of our data. To get a clear understanding, we followed a few steps to explore our dataset.

1. **Getting Descriptive Statistics of The Data:**

   First, we used the describe() function in Python to generate descriptive statistics of the data, giving a high-level insight into the data distribution. It returns the following statistics table for all the numeric columns as follows:

```
df.describe()
```

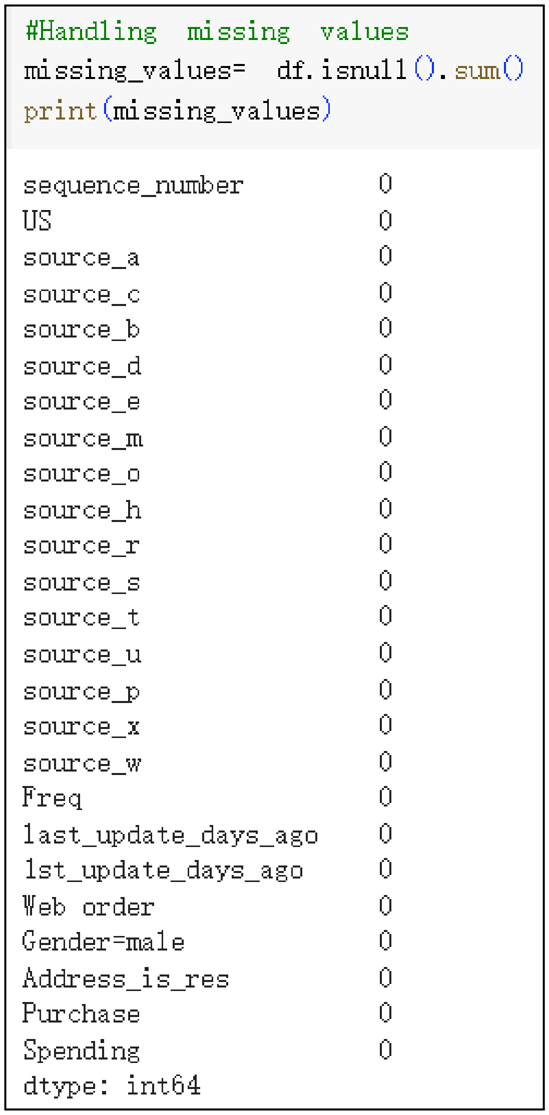


	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	...
mean	1000.500000	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	...
std	577.494589	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	...
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...
25%	500.750000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...
50%	1000.500000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...
75%	1500.250000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...
max	2000.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	...

	source_x	source_w	Freq	last_update_days_ago	1st_update_days_ago	Web order	Gender=male	Address_is_res	Purchase	Spending
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	0.018000	0.137500	1.417000	2155.101000	2435.601500	0.426000	0.524500	0.221000	0.500000	102.560745
std	0.132984	0.344461	1.405738	1141.302846	1077.872233	0.494617	0.499524	0.415024	0.500125	186.749816
min	0.000000	0.000000	0.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	1.000000	1133.000000	1671.250000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	1.000000	2280.000000	2721.000000	0.000000	1.000000	0.000000	0.500000	1.855000
75%	0.000000	0.000000	2.000000	3139.250000	3353.000000	1.000000	1.000000	0.000000	1.000000	152.532500
max	1.000000	1.000000	15.000000	4188.000000	4188.000000	1.000000	1.000000	1.000000	1.000000	1500.060000

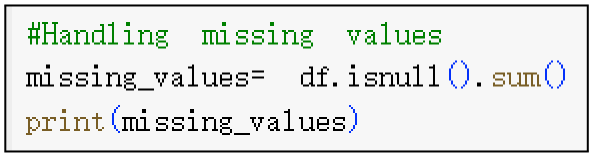
2. **Detect Missing Values:**

   During the process, we dived into the dataset, figuring out any potential missing values. This step guarantees the completeness of our dataset, allowing us to avoid mistakes in our analysis and modeling using the dataset.
   After performing a thorough evaluation of our datasets, there is no missing value included in our dataset, which means our data is in a good position to be used in our models.



```
#Handling missing values
missing_values= df.isnull().sum()
print(missing_values)
```

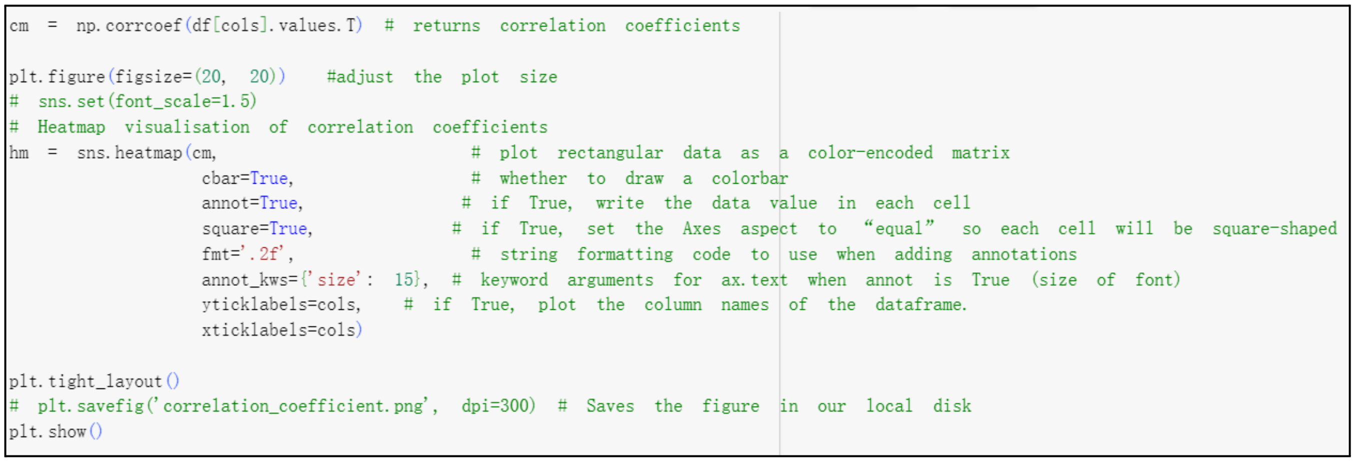
```
sequence_number         0
US                      0
source_a                0
source_c                0
source_b                0
source_d                0
source_e                0
source_m                0
source_o                0
source_h                0
source_r                0
source_s                0
source_t                0
source_u                0
source_p                0
source_x                0
source_w                0
Freq                    0
last_update_days_ago    0
1st_update_days_ago     0
Web order               0
Gender=male             0
Address_is_res          0
Purchase                0
Spending                0
dtype: int64
```



```
#Handling missing values
missing_values= df.isnull().sum()
print(missing_values)
```

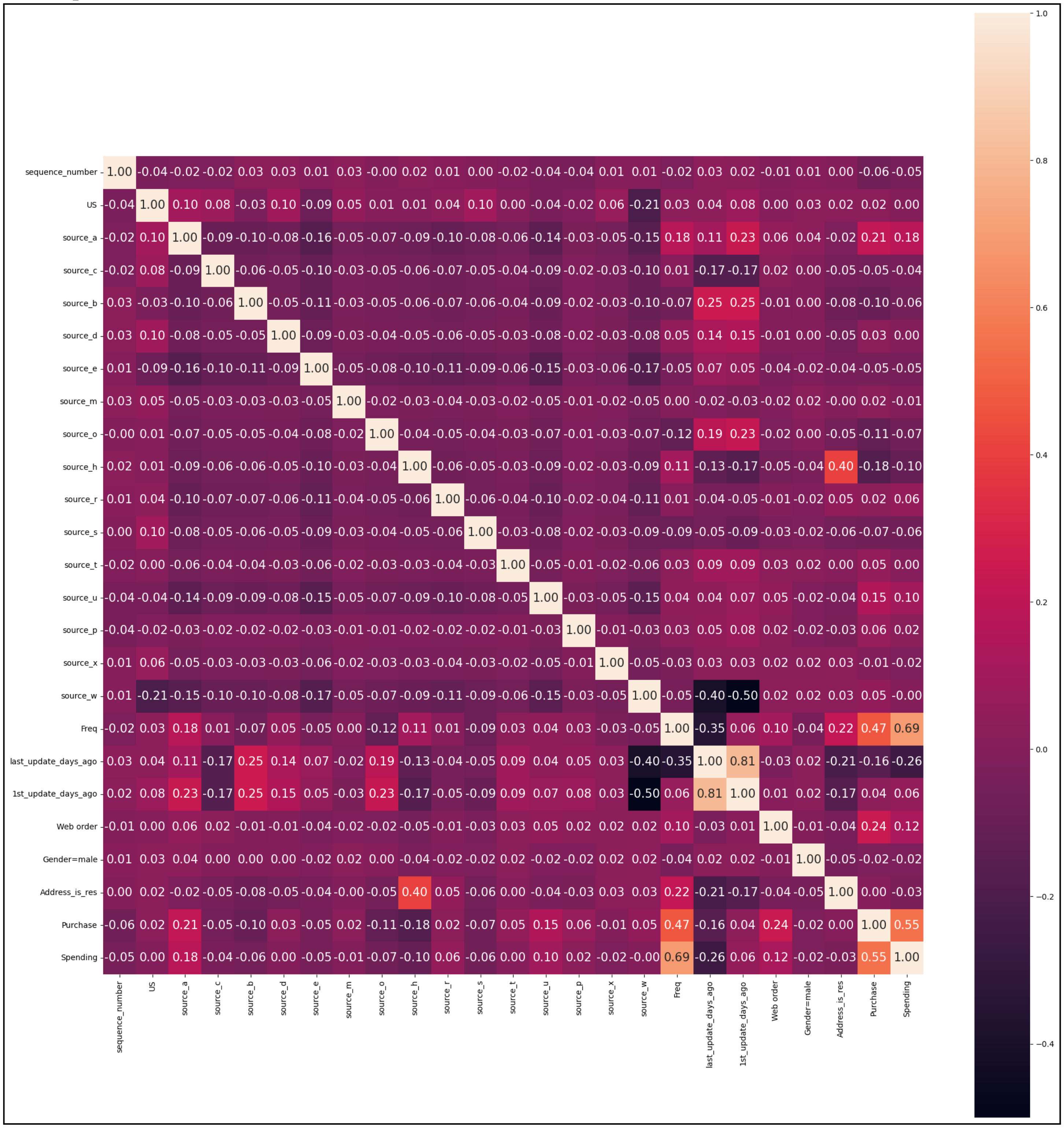
3. **<u>Correlation Matrix of The Data:</u>**

We created the heat maps to provide us with a clearer understanding of the correlation matrix. The corresponding codes and outputs are as follows:



```
cm = np.corrcoef(df[cols].values.T)  # returns correlation coefficients

plt.figure(figsize=(20, 20))   #adjust the plot size
# sns.set(font_scale=1.5)
# Heatmap visualisation of correlation coefficients
hm = sns.heatmap(cm,                        # plot rectangular data as a color-encoded matrix
                cbar=True,                  # whether to draw a colorbar
                annot=True,                 # if True, write the data value in each cell
                square=True,                # if True, set the Axes aspect to "equal" so each cell will be square-shaped
                fmt='.2f',                  # string formatting code to use when adding annotations
                annot_kws={'size': 15},     # keyword arguments for ax.text when annot is True (size of font)
                yticklabels=cols,     # if True, plot the column names of the dataframe.
                xticklabels=cols)

plt.tight_layout()
# plt.savefig('correlation_coefficient.png', dpi=300)  # Saves the figure in our local disk
plt.show()
```

Heatmap Results:



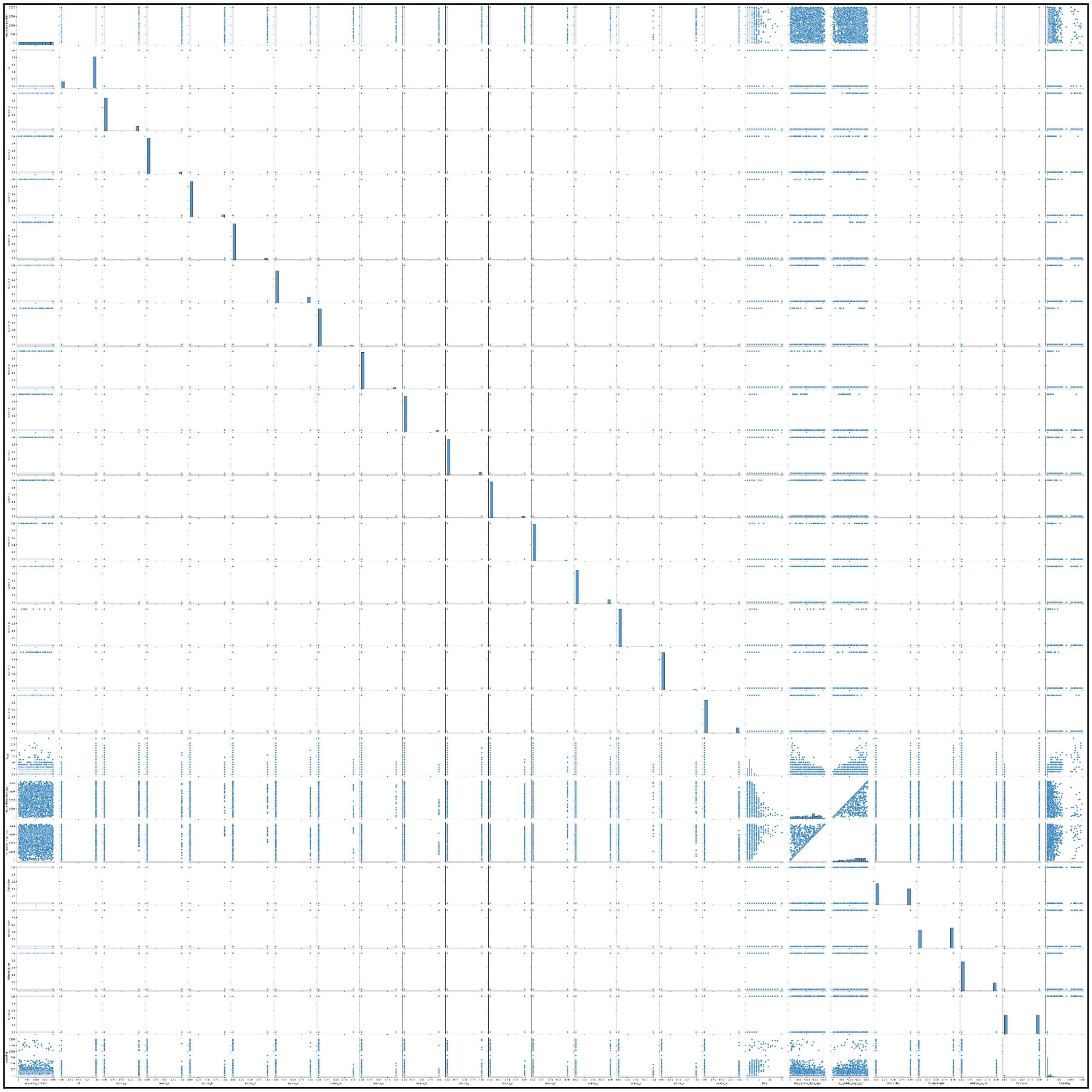
	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	source_s	source_t	source_u	source_p	source_x	source_w	Freq	last_update_days_ago	1st_update_days_ago	Web order	Gender=male	Address_is_res	Purchase	Spending
sequence_number	1.00	-0.04	-0.02	-0.02	0.03	0.03	0.01	0.03	-0.00	0.02	0.01	0.00	-0.02	-0.04	-0.04	0.01	0.01	-0.02	0.03	0.02	-0.01	0.01	0.00	-0.06	-0.05
US	-0.04	1.00	0.10	0.08	-0.03	0.10	-0.09	0.05	0.01	0.01	0.04	0.10	0.00	-0.04	-0.02	0.06	-0.21	0.03	0.04	0.08	0.00	0.03	0.02	0.02	0.00
source_a	-0.02	0.10	1.00	-0.09	-0.10	-0.08	-0.16	-0.05	-0.07	-0.09	-0.10	-0.08	-0.06	-0.14	-0.03	-0.05	-0.15	0.18	0.11	0.23	0.06	0.04	-0.02	0.21	0.18
source_c	-0.02	0.08	-0.09	1.00	-0.06	-0.05	-0.10	-0.03	-0.05	-0.06	-0.07	-0.05	-0.04	-0.09	-0.02	-0.03	-0.10	0.01	-0.17	-0.17	0.02	0.00	-0.05	-0.05	-0.04
source_b	0.03	-0.03	-0.10	-0.06	1.00	-0.05	-0.11	-0.03	-0.05	-0.06	-0.07	-0.06	-0.04	-0.09	-0.02	-0.03	-0.10	-0.07	0.25	0.25	-0.01	0.00	-0.08	-0.10	-0.06
source_d	0.03	0.10	-0.08	-0.05	-0.05	1.00	-0.09	-0.03	-0.04	-0.05	-0.06	-0.05	-0.03	-0.08	-0.02	-0.03	-0.08	0.05	0.14	0.15	-0.01	0.00	-0.05	0.03	0.00
source_e	0.01	-0.09	-0.16	-0.10	-0.11	-0.09	1.00	-0.05	-0.08	-0.10	-0.11	-0.09	-0.06	-0.15	-0.03	-0.06	-0.17	-0.05	0.07	0.05	-0.04	-0.02	-0.04	-0.05	-0.05
source_m	0.03	0.05	-0.05	-0.03	-0.03	-0.03	-0.05	1.00	-0.02	-0.03	-0.04	-0.03	-0.02	-0.05	-0.01	-0.02	-0.05	0.00	-0.02	-0.03	-0.02	0.02	-0.00	0.02	-0.01
source_o	-0.00	0.01	-0.07	-0.05	-0.05	-0.04	-0.08	-0.02	1.00	-0.04	-0.05	-0.04	-0.03	-0.07	-0.01	-0.03	-0.07	-0.12	0.19	0.23	-0.02	0.00	-0.05	-0.11	-0.07
source_h	0.02	0.01	-0.09	-0.06	-0.06	-0.05	-0.10	-0.03	-0.04	1.00	-0.06	-0.05	-0.03	-0.09	-0.02	-0.03	-0.09	0.11	-0.13	-0.17	-0.05	-0.04	0.40	-0.18	-0.10
source_r	0.01	0.04	-0.10	-0.07	-0.07	-0.06	-0.11	-0.04	-0.05	-0.06	1.00	-0.06	-0.04	-0.10	-0.02	-0.04	-0.11	0.01	-0.04	-0.05	-0.01	-0.02	0.05	0.02	0.06
source_s	0.00	0.10	-0.08	-0.05	-0.06	-0.05	-0.09	-0.03	-0.04	-0.05	-0.06	1.00	-0.03	-0.08	-0.02	-0.03	-0.09	-0.09	-0.05	-0.09	-0.03	-0.02	-0.06	-0.07	-0.06
source_t	-0.02	0.00	-0.06	-0.04	-0.04	-0.03	-0.06	-0.02	-0.03	-0.03	-0.04	-0.03	1.00	-0.05	-0.01	-0.02	-0.06	0.03	0.09	0.09	0.03	0.02	0.00	0.05	0.00
source_u	-0.04	-0.04	-0.14	-0.09	-0.09	-0.08	-0.15	-0.05	-0.07	-0.09	-0.10	-0.08	-0.05	1.00	-0.03	-0.05	-0.15	0.04	0.04	0.07	0.05	-0.02	-0.04	0.15	0.10
source_p	-0.04	-0.02	-0.03	-0.02	-0.02	-0.02	-0.03	-0.01	-0.01	-0.02	-0.02	-0.02	-0.01	-0.03	1.00	-0.01	-0.03	0.03	0.05	0.08	0.02	-0.02	-0.03	0.06	0.02
source_x	0.01	0.06	-0.05	-0.03	-0.03	-0.03	-0.06	-0.02	-0.03	-0.03	-0.04	-0.03	-0.02	-0.05	-0.01	1.00	-0.05	-0.03	0.03	0.03	0.02	0.02	0.03	-0.01	-0.02
source_w	0.01	-0.21	-0.15	-0.10	-0.10	-0.08	-0.17	-0.05	-0.07	-0.09	-0.11	-0.09	-0.06	-0.15	-0.03	-0.05	1.00	-0.05	-0.40	-0.50	0.02	0.02	0.03	0.05	-0.00
Freq	-0.02	0.03	0.18	0.01	-0.07	0.05	-0.05	0.00	-0.12	0.11	0.01	-0.09	0.03	0.04	0.03	-0.03	-0.05	1.00	-0.35	0.06	0.10	-0.04	0.22	0.47	0.69
last_update_days_ago	0.03	0.04	0.11	-0.17	0.25	0.14	0.07	-0.02	0.19	-0.13	-0.04	-0.05	0.09	0.04	0.05	0.03	-0.40	-0.35	1.00	0.81	-0.03	0.02	-0.21	-0.16	-0.26
1st_update_days_ago	0.02	0.08	0.23	-0.17	0.25	0.15	0.05	-0.03	0.23	-0.17	-0.05	-0.09	0.09	0.07	0.08	0.03	-0.50	0.06	0.81	1.00	0.01	0.02	-0.17	0.04	0.06
Web order	-0.01	0.00	0.06	0.02	-0.01	-0.01	-0.04	-0.02	-0.02	-0.05	-0.01	-0.03	0.03	0.05	0.02	0.02	0.02	0.10	-0.03	0.01	1.00	-0.01	-0.04	0.24	0.12
Gender=male	0.01	0.03	0.04	0.00	0.00	0.00	-0.02	0.02	0.00	-0.04	-0.02	-0.02	0.02	-0.02	-0.02	0.02	0.02	-0.04	0.02	0.02	-0.01	1.00	-0.05	-0.02	-0.02
Address_is_res	0.00	0.02	-0.02	-0.05	-0.08	-0.05	-0.04	-0.00	-0.05	0.40	0.05	-0.06	0.00	-0.04	-0.03	0.03	0.03	0.22	-0.21	-0.17	-0.04	-0.05	1.00	0.00	-0.03
Purchase	-0.06	0.02	0.21	-0.05	-0.10	0.03	-0.05	0.02	-0.11	-0.18	0.02	-0.07	0.05	0.15	0.06	-0.01	0.05	0.47	-0.16	0.04	0.24	-0.02	0.00	1.00	0.55
Spending	-0.05	0.00	0.18	-0.04	-0.06	0.00	-0.05	-0.01	-0.07	-0.10	0.06	-0.06	0.00	0.10	0.02	-0.02	-0.00	0.69	-0.26	0.06	0.12	-0.02	-0.03	0.55	1.00

When two features in a dataset are highly correlated, it can lead to multicollinearity, which can cause overfitting, unreliable Coefficient estimation, and so on. To avoid this, we set 0.8 as the threshold.
As the plot shows, the correlation of “last_update_days_ago” and “1st_update_days_ago” is the only pair that has over 0.8. Therefore, after careful consideration, we decided to drop one of them in the modeling step to avoid multicollinearity.

4. **Pair Plot:**

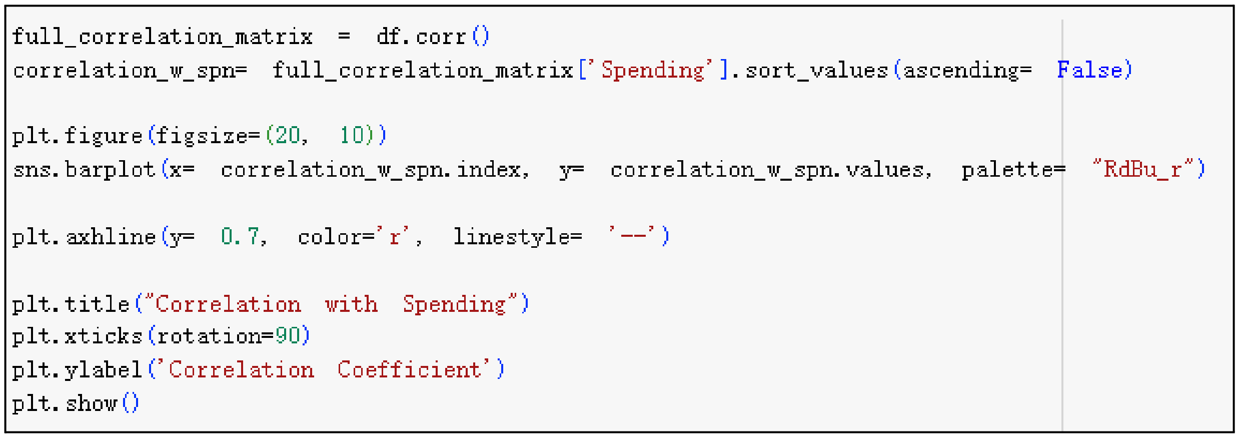
Also, we created and displayed a pair plot of columns for identifying patterns, relationships, and potential outliers in our dataset.

```
sns.pairplot(df[cols],
                   height=2.5)                        # plot pairwise relationships in a dataset
plt.tight_layout()                               # tight_layout automatically adjusts subplot params
                                                      # so that the subplot(s) fits in to the figure area.
# plt.savefig('housing_dataset.png', dpi=300)  # saves the figure in our local disk
plt.show()                                             # display figure
```

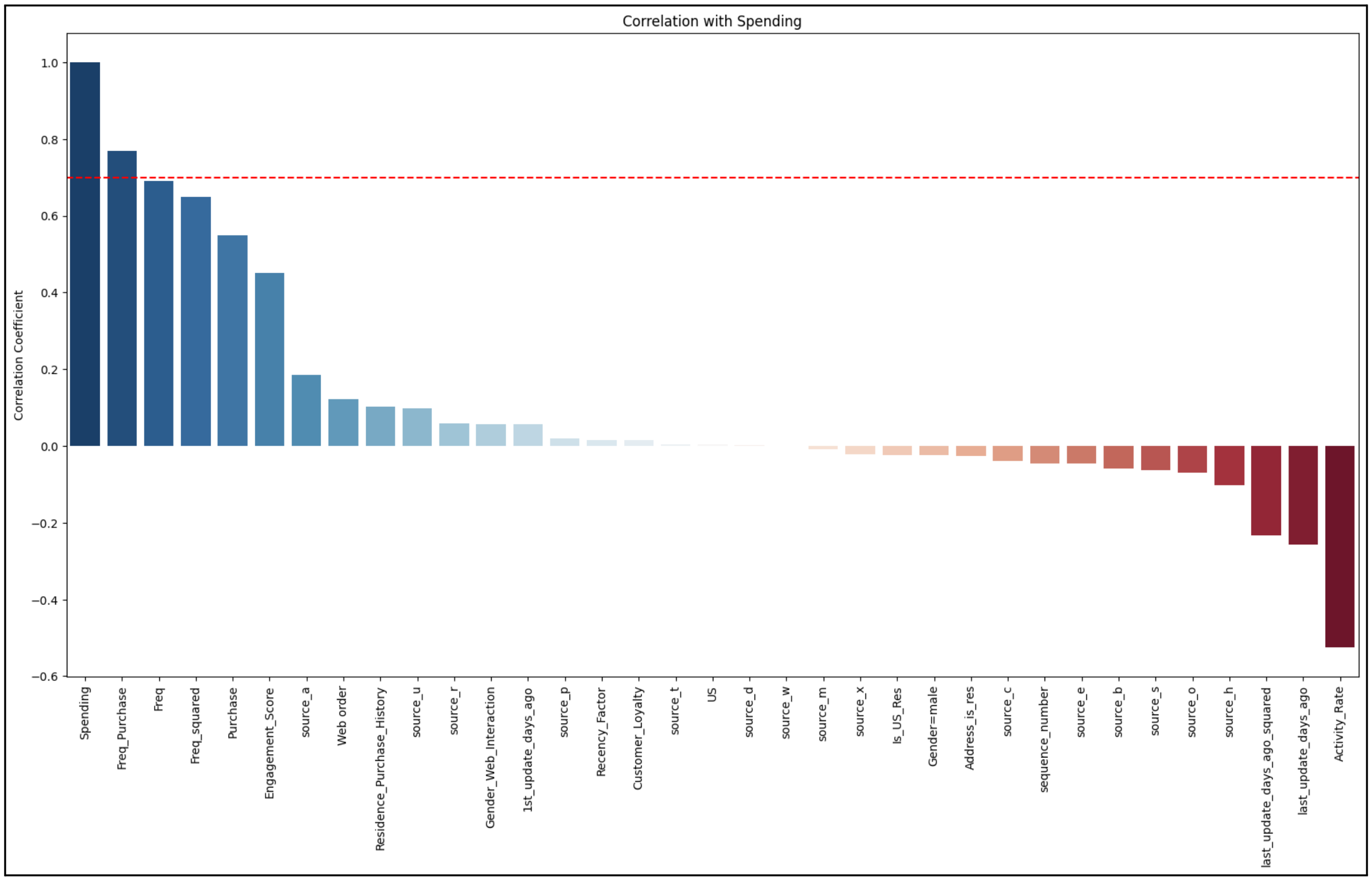


5. **<u>Check Correlation with Target Variable:</u>**

In the last step, we created a bar plot for checking the correlation of every feature with the target variable "Spending". Visualizing the correlation allows for a quick visual inspection of which features are most and least correlated. As the following shows, "Freq_Purchase" is the attribute most correlated to the target variance, while "Activity_Rate" is the least one.



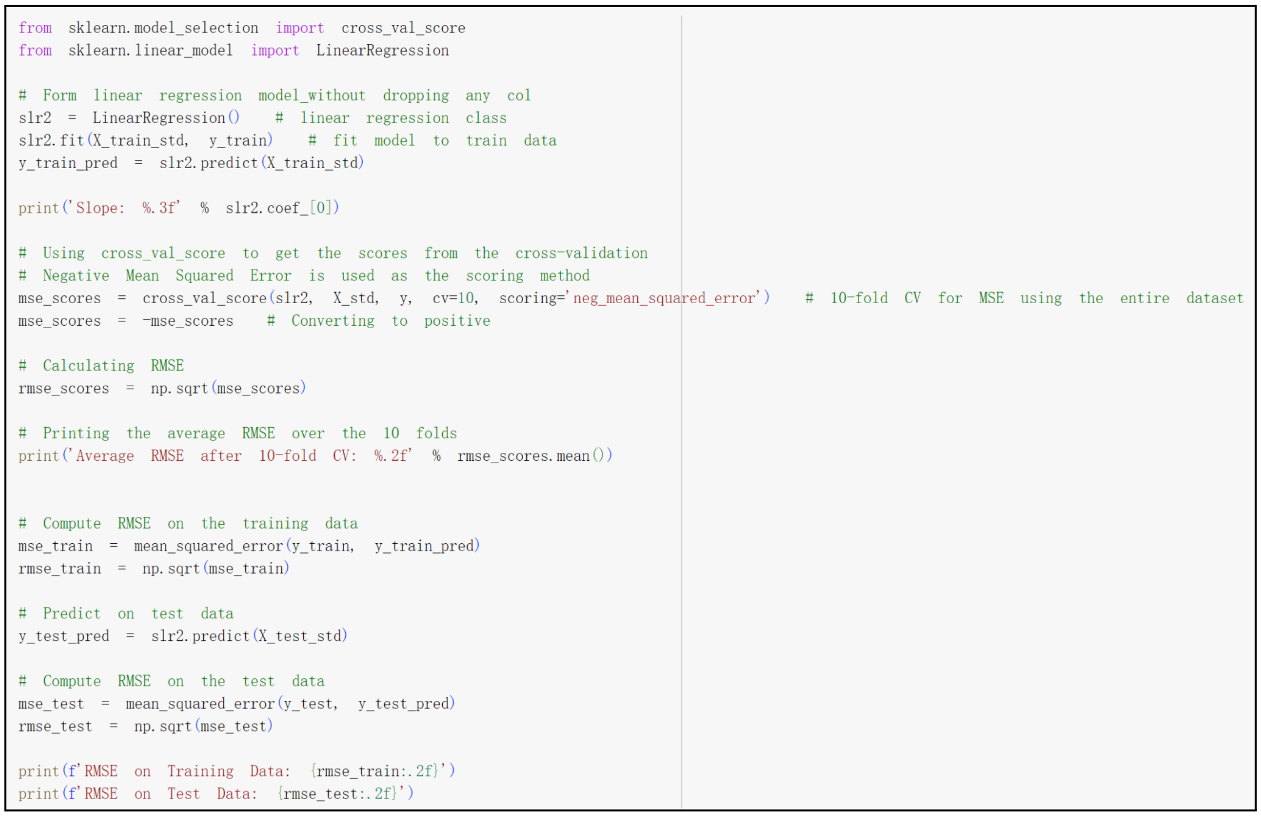
```
full_correlation_matrix  =  df.corr()
correlation_w_spn=  full_correlation_matrix['Spending'].sort_values(ascending=  False)

plt.figure(figsize=(20,  10))
sns.barplot(x=  correlation_w_spn.index,  y=  correlation_w_spn.values,  palette=  "RdBu_r")

plt.axhline(y=  0.7,  color='r',  linestyle=  '--')

plt.title("Correlation  with  Spending")
plt.xticks(rotation=90)
plt.ylabel('Correlation  Coefficient')
plt.show()
```



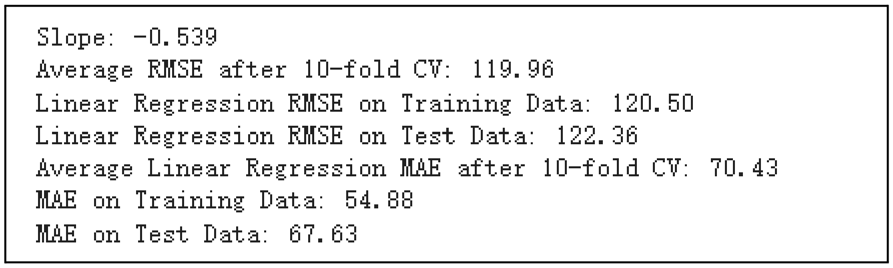
<u>Building Model Pharse:</u>

- **LinearRegression**

As we mentioned in the previous part, the correlation between "last_update_days_ago" and "1st_update_days_ago" surpasses our threshold; it is crucial to eliminate one of these columns when constructing a linear regression model. Upon evaluating their absolute correlation value with the target value, we have determined that "1st_update_days_ago" has a weaker association with "Spending" (Referring to the Correlation plot above). Therefore, we have chosen to discard this column. We will take two steps to assess the impact of dropping the column on the training of the linear regression model.
First, we form the linear regression model without dropping any columns and test the model with cross-validation.

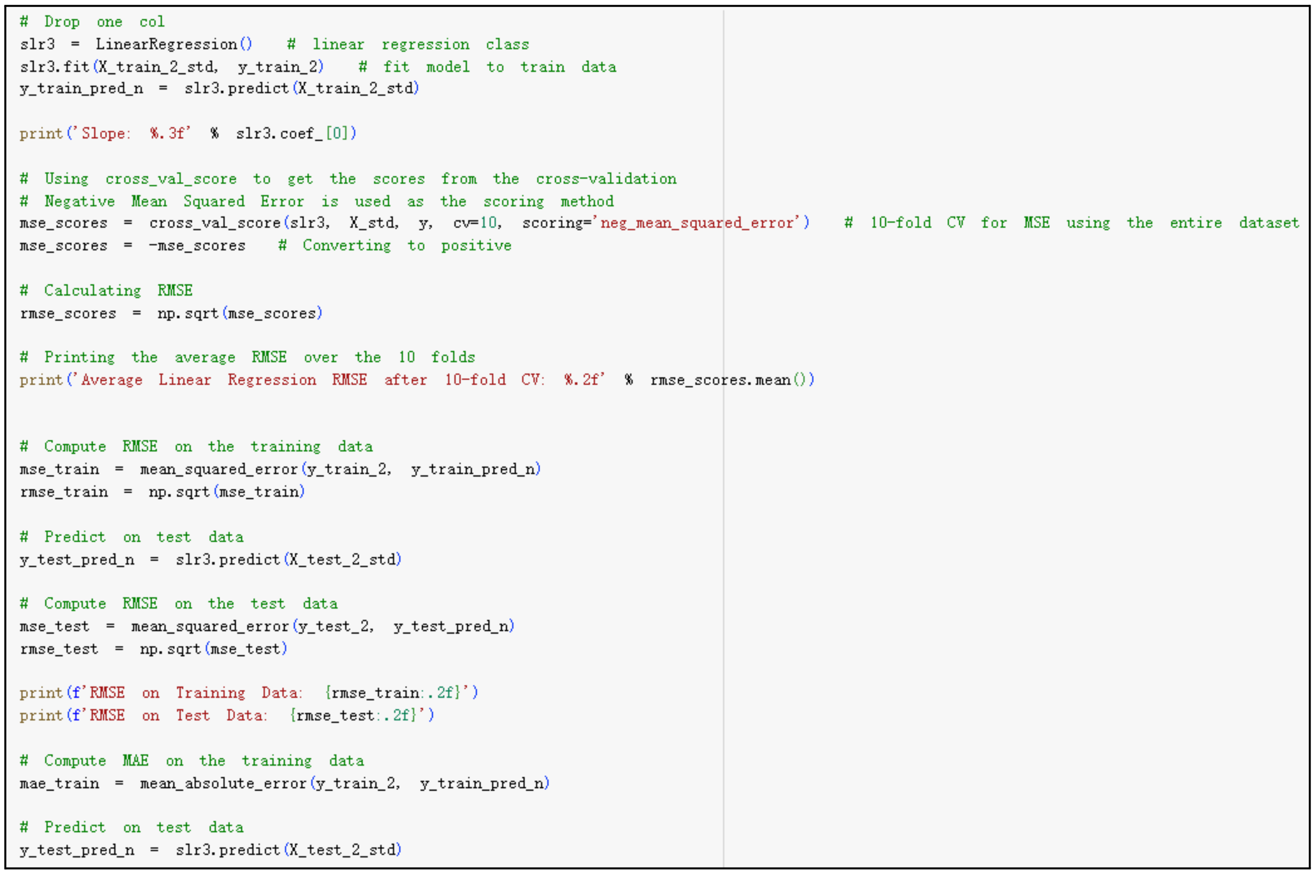


```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

# Form linear regression model_without dropping any col
slr2 = LinearRegression()   # linear regression class
slr2.fit(X_train_std, y_train)   # fit model to train data
y_train_pred = slr2.predict(X_train_std)

print('Slope: %.3f' % slr2.coef_[0])

# Using cross_val_score to get the scores from the cross-validation
# Negative Mean Squared Error is used as the scoring method
mse_scores = cross_val_score(slr2, X_std, y, cv=10, scoring='neg_mean_squared_error')   # 10-fold CV for MSE using the entire dataset
mse_scores = -mse_scores   # Converting to positive

# Calculating RMSE
rmse_scores = np.sqrt(mse_scores)

# Printing the average RMSE over the 10 folds
print('Average RMSE after 10-fold CV: %.2f' % rmse_scores.mean())


# Compute RMSE on the training data
mse_train = mean_squared_error(y_train, y_train_pred)
rmse_train = np.sqrt(mse_train)

# Predict on test data
y_test_pred = slr2.predict(X_test_std)

# Compute RMSE on the test data
mse_test = mean_squared_error(y_test, y_test_pred)
rmse_test = np.sqrt(mse_test)

print(f'RMSE on Training Data: {rmse_train:.2f}')
print(f'RMSE on Test Data: {rmse_test:.2f}')
```

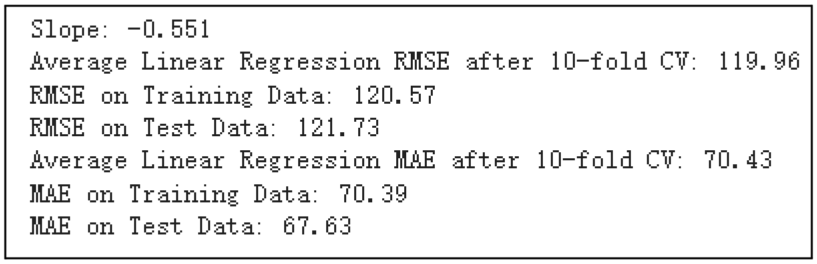


```
Slope: -0.539
Average RMSE after 10-fold CV: 119.96
Linear Regression RMSE on Training Data: 120.50
Linear Regression RMSE on Test Data: 122.36
Average Linear Regression MAE after 10-fold CV: 70.43
MAE on Training Data: 54.88
MAE on Test Data: 67.63
```

Then, we developed a new linear regression model using the training data by excluding the "1st_update_days_ago" column. The average RMSE and MAE after 10-fold cross-validation seem to be the same as the value without dropping the attribute. Nonetheless, while we further compared the gap between training and testing data, the gap of the new model became smaller than the original one (without dropping any columns). Furthermore, the RMSE on testing data was lower, suggesting that removing the attribute helps enhance the model's overall performance.



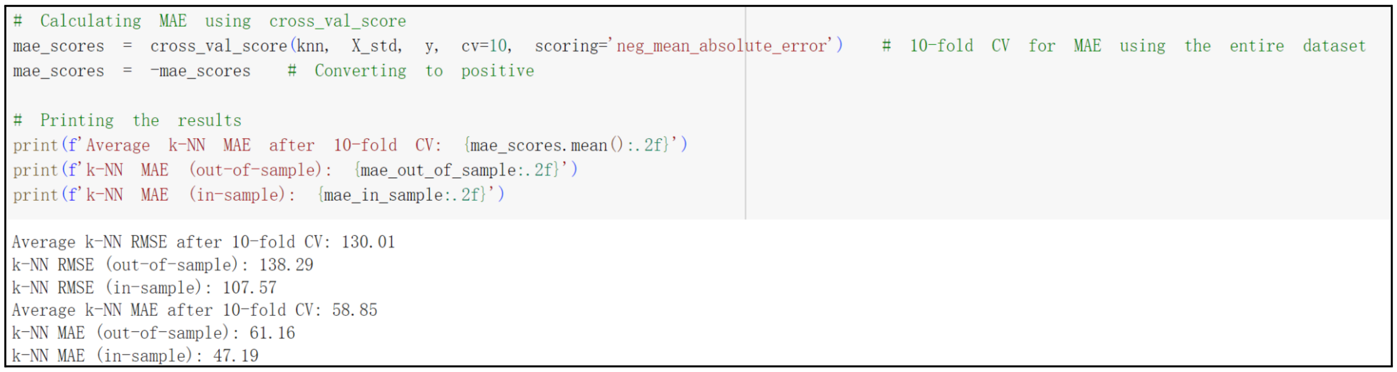
```
# Drop one col
slr3 = LinearRegression()   # linear regression class
slr3.fit(X_train_2_std, y_train_2)   # fit model to train data
y_train_pred_n = slr3.predict(X_train_2_std)

print('Slope: %.3f' % slr3.coef_[0])

# Using cross_val_score to get the scores from the cross-validation
# Negative Mean Squared Error is used as the scoring method
mse_scores = cross_val_score(slr3, X_std, y, cv=10, scoring='neg_mean_squared_error')   # 10-fold CV for MSE using the entire dataset
mse_scores = -mse_scores   # Converting to positive

# Calculating RMSE
rmse_scores = np.sqrt(mse_scores)

# Printing the average RMSE over the 10 folds
print('Average Linear Regression RMSE after 10-fold CV: %.2f' % rmse_scores.mean())


# Compute RMSE on the training data
mse_train = mean_squared_error(y_train_2, y_train_pred_n)
rmse_train = np.sqrt(mse_train)

# Predict on test data
y_test_pred_n = slr3.predict(X_test_2_std)

# Compute RMSE on the test data
mse_test = mean_squared_error(y_test_2, y_test_pred_n)
rmse_test = np.sqrt(mse_test)

print(f'RMSE on Training Data: {rmse_train:.2f}')
print(f'RMSE on Test Data: {rmse_test:.2f}')

# Compute MAE on the training data
mae_train = mean_absolute_error(y_train_2, y_train_pred_n)

# Predict on test data
y_test_pred_n = slr3.predict(X_test_2_std)
```



```
Slope: -0.551
Average Linear Regression RMSE after 10-fold CV: 119.96
RMSE on Training Data: 120.57
RMSE on Test Data: 121.73
Average Linear Regression MAE after 10-fold CV: 70.43
MAE on Training Data: 70.39
MAE on Test Data: 67.63
```

- **KNN**

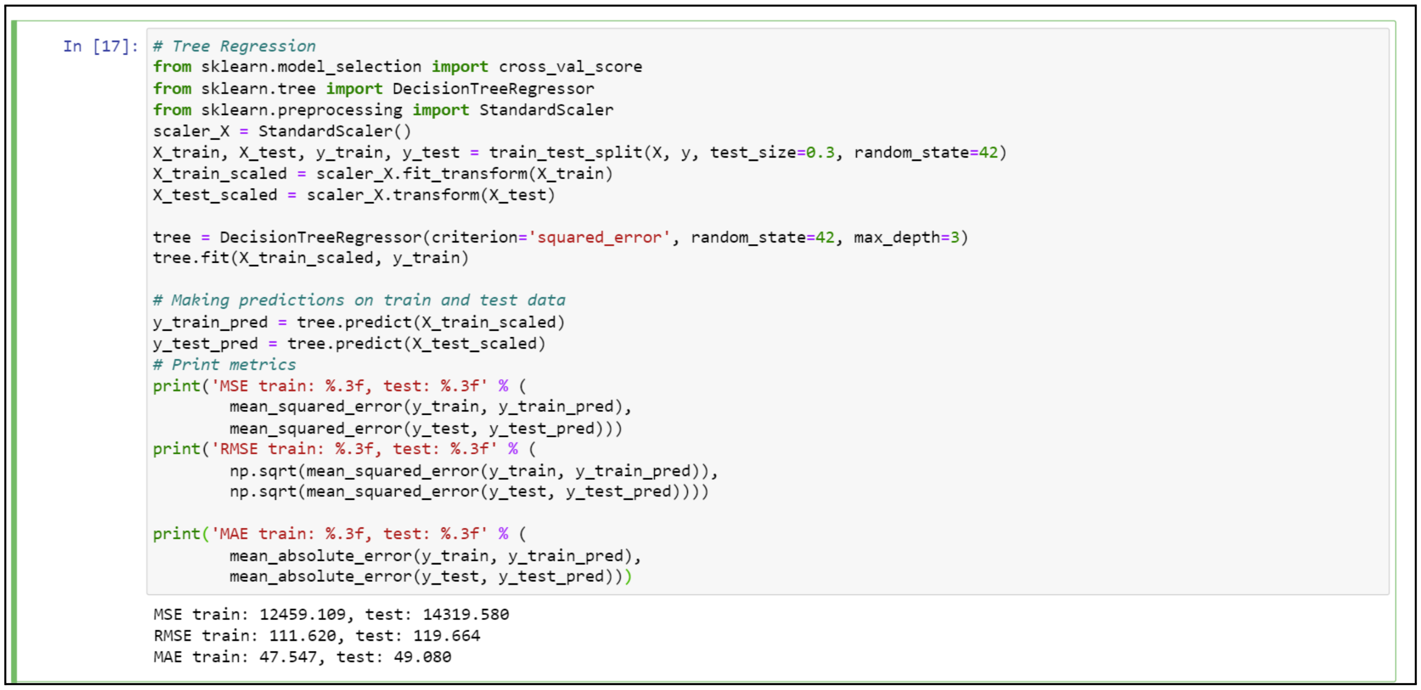
kNN model building code, and the outcome of using 10-fold cross-validation as follows:



```
# Calculating MAE using cross_val_score
mae_scores = cross_val_score(knn, X_std, y, cv=10, scoring='neg_mean_absolute_error')   # 10-fold CV for MAE using the entire dataset
mae_scores = -mae_scores   # Converting to positive

# Printing the results
print(f'Average k-NN MAE after 10-fold CV: {mae_scores.mean():.2f}')
print(f'k-NN MAE (out-of-sample): {mae_out_of_sample:.2f}')
print(f'k-NN MAE (in-sample): {mae_in_sample:.2f}')
```

```
Average k-NN RMSE after 10-fold CV: 130.01
k-NN RMSE (out-of-sample): 138.29
k-NN RMSE (in-sample): 107.57
Average k-NN MAE after 10-fold CV: 58.85
k-NN MAE (out-of-sample): 61.16
k-NN MAE (in-sample): 47.19
```

- **DecisionTreeRegressor**



```
In [17]: # Tree Regression
         from sklearn.model_selection import cross_val_score
         from sklearn.tree import DecisionTreeRegressor
         from sklearn.preprocessing import StandardScaler
         scaler_X = StandardScaler()
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
         X_train_scaled = scaler_X.fit_transform(X_train)
         X_test_scaled = scaler_X.transform(X_test)

         tree = DecisionTreeRegressor(criterion='squared_error', random_state=42, max_depth=3)
         tree.fit(X_train_scaled, y_train)

         # Making predictions on train and test data
         y_train_pred = tree.predict(X_train_scaled)
         y_test_pred = tree.predict(X_test_scaled)
         # Print metrics
         print('MSE train: %.3f, test: %.3f' % (
                 mean_squared_error(y_train, y_train_pred),
                 mean_squared_error(y_test, y_test_pred)))
         print('RMSE train: %.3f, test: %.3f' % (
                 np.sqrt(mean_squared_error(y_train, y_train_pred)),
                 np.sqrt(mean_squared_error(y_test, y_test_pred))))

         print('MAE train: %.3f, test: %.3f' % (
                 mean_absolute_error(y_train, y_train_pred),
                 mean_absolute_error(y_test, y_test_pred)))

         MSE train: 12459.109, test: 14319.580
         RMSE train: 111.620, test: 119.664
         MAE train: 47.547, test: 49.080
```

To choose the best model, we use RMSE (Root Mean Square Error) as our standard, which is relatively common and effective. Comparing the value of each model's RMSE after cross-validation with 10 folds, the RMSE for Linear Regression is 121.73, for KNN is 138.29, and for Decision Tree Regressor is 147.76.
In conclusion, linear Regression has the best performance under the context.

(b) (50 points) Engage in feature engineering (i.e., create new features based on existing features) to optimize the performance of linear regression, k-NN, and regression tree techniques. Present the results for each of the three techniques (choose the best-performing model for each technique in case you try multiple models) and discuss which of the three yields the best performance. Use cross-validation with 10 folds to estimate the generalization performance. Discuss whether and why the generalization performance was improved or not.

We created the following new features after conducting the feature engineering process:

<u>Numeric Features</u>

(c) **Customer_Loyalty:** We divided the number of transactions by the number of days of the first update on customer activity. The longer customers have not been active or the fewer purchases they made, the less loyal that is for that customer. It's based on the notion that a customer who has made many purchases since their first interaction is likely more loyal. Loyal customers are more likely to respond positively to new marketing campaigns like the test mailing. Tracking this can help the business focus its marketing efforts.

(d) **Recency_Facto**r: This measures how recently the customer has interacted with the catalog. More recent, indicating customers' interest to purchase. Recently engaged customers are more likely to purchase when prompted by a new campaign.

(e) **Engagement_Score:** Multiplying Freq with Web order creates an engagement score based on frequency and web-based interactions. It's a way of quantifying the digital engagement of a customer.

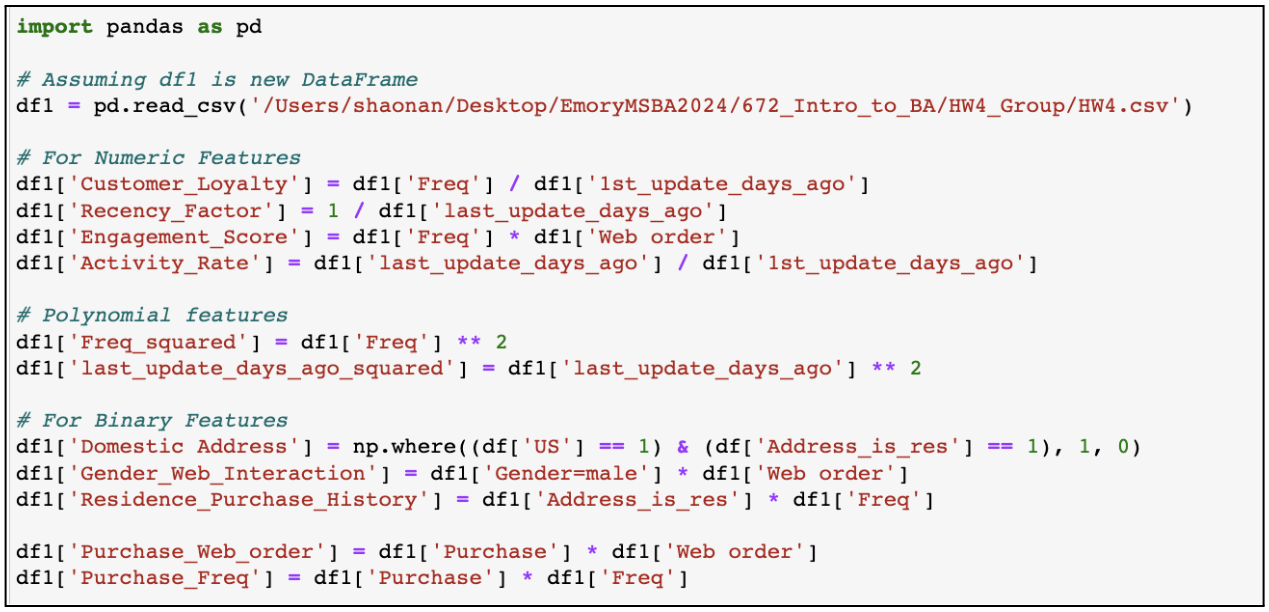
(f) **Activity_Rat**e: The higher activity rate indicates higher interest in the catalog, making those customers good targets for new marketing campaigns.

Polynomial Features

(g) **Freq_Purchase:** This feature multiplies frequency (Freq) with purchase, indicating the total purchases made for each frequency count. It amplifies the significance of high-frequency customers.
(h) **last_update_days_ago_squared**: Squaring the last update days can help capture nonlinear patterns regarding how long it's been since the customer last updated. It's another way to give more weight to more recent or more distant interactions.

Binary Features

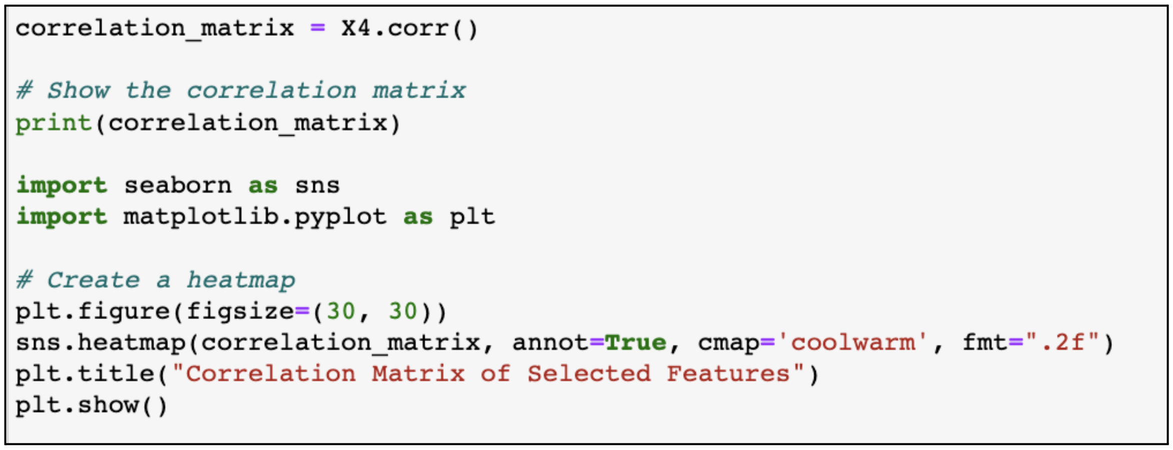
1. **Gender_Web_Interaction:** By multiplying Gender=male (presumably a binary flag where 1 means male and 0 means not male) with Web order, you capture the interaction between gender (specifically males) and online ordering behavior. It will be useful to differentiate patterns between male online ordering versus others.
2. **Is_US_Res**: Multiplying US with Address_is_res determines if a US-based customer has provided a residential address. It can help target specific demographics like US residents.
3. **Residence_Purchase_History**: By multiplying Address_is_res with Freq, you're gauging how often residential address holders purchase. It can indicate the buying behavior of customers providing residential addresses.
   Polynomial Features:
4. **Freq_squared:** This is the square of the frequency, which can help capture non-linearities in the data related to frequency. High-frequency customers will have a disproportionately large value here, highlighting them more.



```python
import pandas as pd

# Assuming df1 is new DataFrame
df1 = pd.read_csv('/Users/shaonan/Desktop/EmoryMSBA2024/672_Intro_to_BA/HW4_Group/HW4.csv')

# For Numeric Features
df1['Customer_Loyalty'] = df1['Freq'] / df1['1st_update_days_ago']
df1['Recency_Factor'] = 1 / df1['last_update_days_ago']
df1['Engagement_Score'] = df1['Freq'] * df1['Web order']
df1['Activity_Rate'] = df1['last_update_days_ago'] / df1['1st_update_days_ago']

# Polynomial features
df1['Freq_squared'] = df1['Freq'] ** 2
df1['last_update_days_ago_squared'] = df1['last_update_days_ago'] ** 2

# For Binary Features
df1['Domestic Address'] = np.where((df['US'] == 1) & (df['Address_is_res'] == 1), 1, 0)
df1['Gender_Web_Interaction'] = df1['Gender=male'] * df1['Web order']
df1['Residence_Purchase_History'] = df1['Address_is_res'] * df1['Freq']

df1['Purchase_Web_order'] = df1['Purchase'] * df1['Web order']
df1['Purchase_Freq'] = df1['Purchase'] * df1['Freq']
```

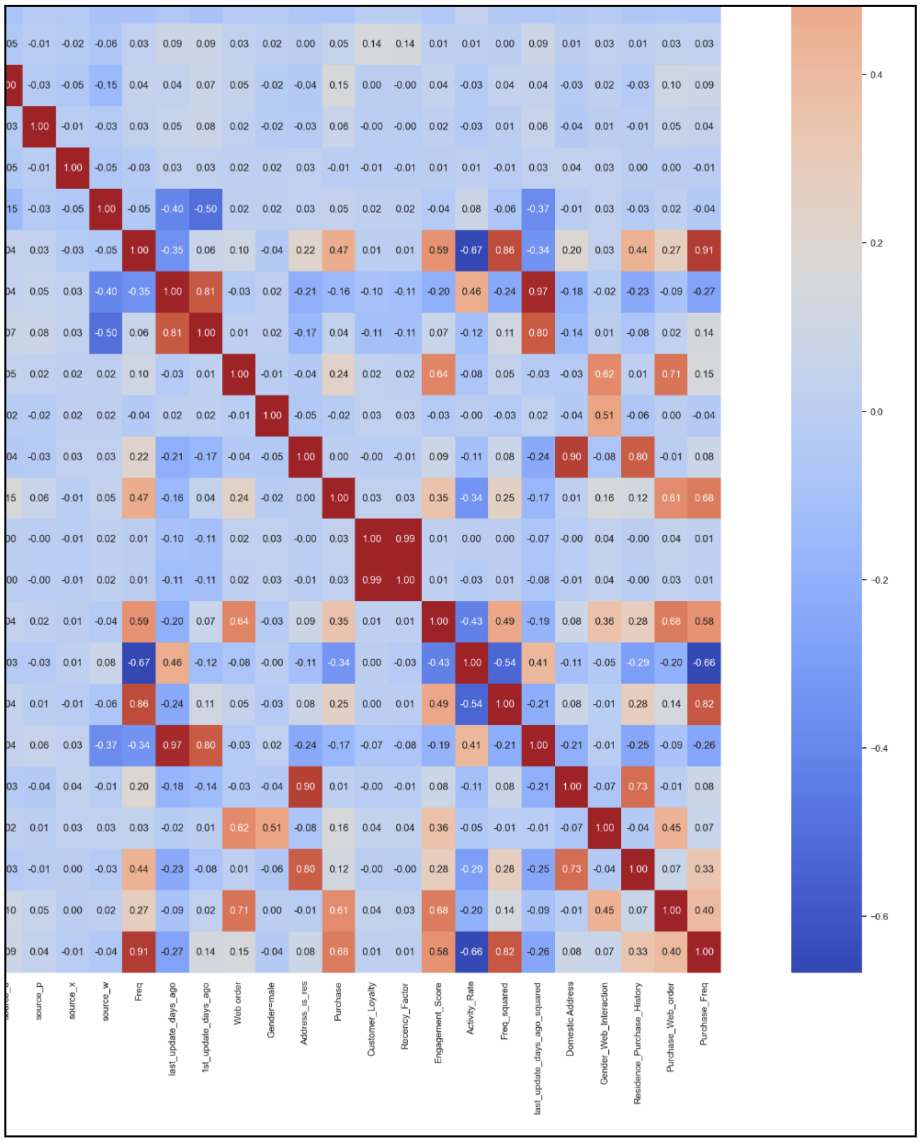
<u>Re-Assessment of Feature Correlations Post-Feature Engineering</u>

After the feature engineering phase, it's crucial to re-evaluate the dataset for multicollinearity. Multicollinearity can result in overfitting and unstable coefficient estimates in linear regression models, reducing the model's interpretability and generalization capability.
In our updated dataset, labeled as the X4 dataframe, we conducted a correlation analysis with a threshold set at 0.8. This analysis revealed several original and newly engineered features that exhibited high correlation. Specifically, the features '1st_update_days_ago,' 'Recency_Factor,' 'Freq_squared,' 'last_update_days_ago_squared,' 'Domestic Address,' and 'Purchase_Freq' were identified as highly correlated.
We considered removing these correlated features before proceeding with the linear regression model building to mitigate the risks associated with multicollinearity.
The following code generates the heatmap among all attributes:



```python
correlation_matrix = X4.corr()

# Show the correlation matrix
print(correlation_matrix)

import seaborn as sns
import matplotlib.pyplot as plt

# Create a heatmap
plt.figure(figsize=(30, 30))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Correlation Matrix of Selected Features")
plt.show()
```

Heatmap:



Filter out the highly-correlated attributes:

```python
import pandas as pd
import numpy as np

# Assuming `df` is your DataFrame containing the features
correlation_matrix = X4.corr().abs()

# Select the upper triangle of the correlation matrix
upper_triangle = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).astype(np.bool))

# Identify columns to drop based on a threshold (e.g., 0.95)
cols_to_drop = [col for col in upper_triangle.columns if any(upper_triangle[col] >= 0.8)]

# Drop correlated columns
df_dropped = X4.drop(X4[cols_to_drop], axis=1, errors='ignore')

print(f"Columns to drop: {cols_to_drop}")
print(f"Remaining columns: {df_dropped.columns.tolist()}")
```

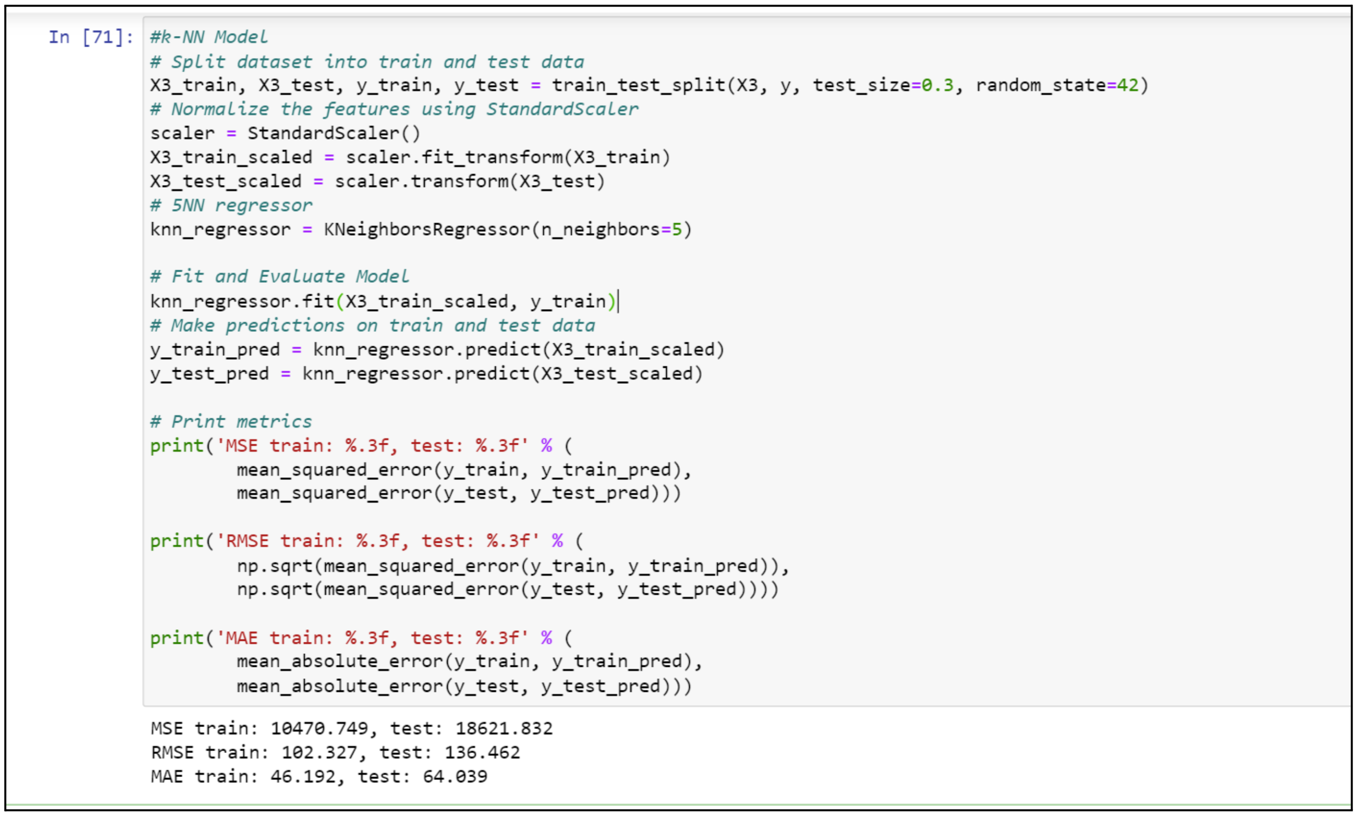
```
Columns to drop: ['1st_update_days_ago', 'Recency_Factor', 'Freq_squared', 'last_update_days_ago_squared', 'Domestic
Address', 'Purchase_Freq']
Remaining columns: ['US', 'source_a', 'source_c', 'source_b', 'source_d', 'source_e', 'source_m', 'source_o', 'source
_h', 'source_r', 'source_s', 'source_t', 'source_u', 'source_p', 'source_x', 'source_w', 'Freq', 'last_update_days_ag
o', 'Web order', 'Gender=male', 'Address_is_res', 'Purchase', 'Customer_Loyalty', 'Engagement_Score', 'Activity_Rat
e', 'Gender_Web_Interaction', 'Residence_Purchase_History', 'Purchase_Web_order']
```

Linear Regression Model After Feature Engineering

```
feature_names_3 = ['US', 'source_a', 'source_c', 'source_b',
                   'source_d', 'source_e', 'source_m', 'source_o',
                   'source_h', 'source_r', 'source_s', 'source_t',
                   'source_u', 'source_p', 'source_x', 'source_w',
                   'Freq', 'last_update_days_ago', 'Web order',
                   'Gender=male', 'Address_is_res', 'Purchase',
                   'Customer_Loyalty', 'Engagement_Score', 'Activity_Rate',
                   'Gender_Web_Interaction', 'Residence_Purchase_History', 'Purchase_Web_order']

X_lr_fe_features = df1[feature_names_3] # drop the varb that is high correlated
y = df1.Spending

# Assuming X_lr_fe_features and y are your feature matrix and target vector
# Linear Regression with X_lr_fe_features
X_train, X_test, y_train, y_test = train_test_split(X_lr_fe_features, y, test_size=0.3, random_state=42)
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

linear_reg = LinearRegression()
linear_reg.fit(X_train_std, y_train)
y_test_pred_lr = linear_reg.predict(X_test_std)
y_train_pred_lr = linear_reg.predict(X_train_std)

print('\nLinear Regression Model with X_lr_fe_features')
print('RMSE train: %.3f, test: %.3f' % (
    sqrt(mean_squared_error(y_train, y_train_pred_lr)),
    sqrt(mean_squared_error(y_test, y_test_pred_lr))))
print('MAE train: %.3f, test: %.3f' % (
    mean_absolute_error(y_train, y_train_pred_lr),
    mean_absolute_error(y_test, y_test_pred_lr)))

# 10-fold-cross-validation Linear Regression
# Initialize the Linear Regression Model
linear_regressor = LinearRegression()

# Perform 10-fold cross-validation
cv_scores_MSE = cross_val_score(linear_regressor, X_lr_fe_features, y, cv=10, scoring='neg_mean_squared_error')
cv_scores_MAE = cross_val_score(linear_regressor, X_lr_fe_features, y, cv=10, scoring='neg_mean_absolute_error')

# Convert negative MSE scores to positive
cv_scores_MSE = -cv_scores_MSE
cv_scores_MAE = -cv_scores_MAE

# Calculate the RMSE for each fold
rmse_scores = np.sqrt(cv_scores_MSE)

# Calculate the average RMSE and MAE across all folds
avg_rmse = np.mean(rmse_scores)
avg_MAE = np.mean(cv_scores_MAE)

print(f'Cross-Validation (10-Fold)')
print(f'Average RMSE: {avg_rmse}')
print(f'Average MAE: {avg_MAE}')
```

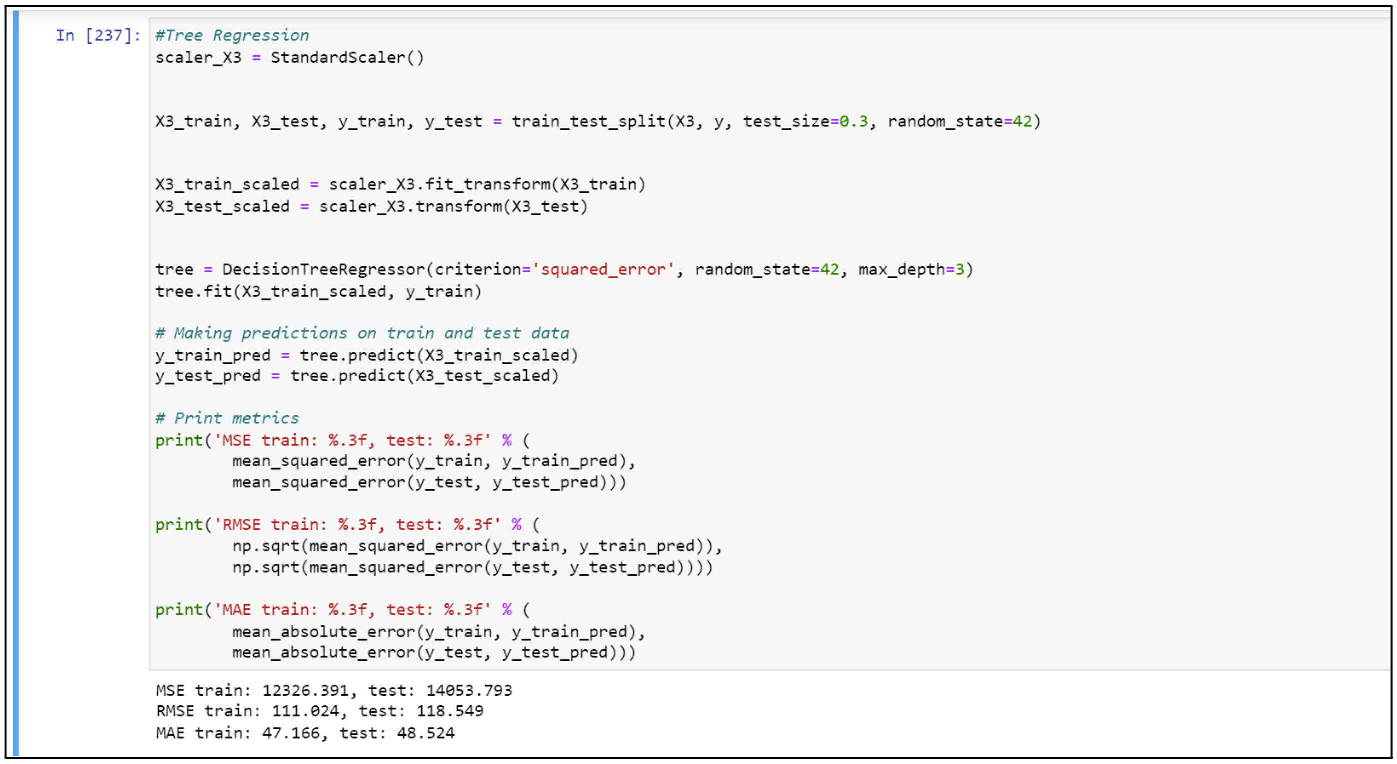
```
Linear Regression Model with X_lr_fe_features
RMSE train: 117.136, test: 119.288
MAE train: 68.457, test: 66.288
Cross-Validation (10-Fold)
Average RMSE: 116.97607446919798
Average MAE: 68.78461273220121
```

K-NN after Feature Engineering



```
In [71]: #k-NN Model
         # Split dataset into train and test data
         X3_train, X3_test, y_train, y_test = train_test_split(X3, y, test_size=0.3, random_state=42)
         # Normalize the features using StandardScaler
         scaler = StandardScaler()
         X3_train_scaled = scaler.fit_transform(X3_train)
         X3_test_scaled = scaler.transform(X3_test)
         # 5NN regressor
         knn_regressor = KNeighborsRegressor(n_neighbors=5)

         # Fit and Evaluate Model
         knn_regressor.fit(X3_train_scaled, y_train)
         # Make predictions on train and test data
         y_train_pred = knn_regressor.predict(X3_train_scaled)
         y_test_pred = knn_regressor.predict(X3_test_scaled)

         # Print metrics
         print('MSE train: %.3f, test: %.3f' % (
                 mean_squared_error(y_train, y_train_pred),
                 mean_squared_error(y_test, y_test_pred)))

         print('RMSE train: %.3f, test: %.3f' % (
                 np.sqrt(mean_squared_error(y_train, y_train_pred)),
                 np.sqrt(mean_squared_error(y_test, y_test_pred))))

         print('MAE train: %.3f, test: %.3f' % (
                 mean_absolute_error(y_train, y_train_pred),
                 mean_absolute_error(y_test, y_test_pred)))

         MSE train: 10470.749, test: 18621.832
         RMSE train: 102.327, test: 136.462
         MAE train: 46.192, test: 64.039
```

Tree Regression after Feature Engineering



```
In [237]: #Tree Regression
          scaler_X3 = StandardScaler()


          X3_train, X3_test, y_train, y_test = train_test_split(X3, y, test_size=0.3, random_state=42)


          X3_train_scaled = scaler_X3.fit_transform(X3_train)
          X3_test_scaled = scaler_X3.transform(X3_test)


          tree = DecisionTreeRegressor(criterion='squared_error', random_state=42, max_depth=3)
          tree.fit(X3_train_scaled, y_train)

          # Making predictions on train and test data
          y_train_pred = tree.predict(X3_train_scaled)
          y_test_pred = tree.predict(X3_test_scaled)

          # Print metrics
          print('MSE train: %.3f, test: %.3f' % (
                  mean_squared_error(y_train, y_train_pred),
                  mean_squared_error(y_test, y_test_pred)))

          print('RMSE train: %.3f, test: %.3f' % (
                  np.sqrt(mean_squared_error(y_train, y_train_pred)),
                  np.sqrt(mean_squared_error(y_test, y_test_pred))))

          print('MAE train: %.3f, test: %.3f' % (
                  mean_absolute_error(y_train, y_train_pred),
                  mean_absolute_error(y_test, y_test_pred)))

          MSE train: 12326.391, test: 14053.793
          RMSE train: 111.024, test: 118.549
          MAE train: 47.166, test: 48.524
```

Conclusion: The generalization of feature engineering is improved for all three models.

Observations:

<u>Linear Regression Model</u>

Before the adjustments, the linear regression model reported a training RMSE of 120.5 and a testing RMSE of 122.36, with a difference of 1.86. After implementing feature engineering and removing highly correlated columns, the training RMSE decreased to 119.288 while the testing RMSE dropped to 117.136. Although the difference between them grew to 2.152, the overall reduction in RMSE for both datasets suggests an improved model. The lowered RMSE values confirm the beneficial impact of the recent feature modifications on the model's precision.

<u>K-NN Model</u>

Before feature engineering, the K-NN model's training RMSE was 107.57, and the testing RMSE was 138.29, a gap of 30.72. Post feature engineering, the training RMSE was reduced to 102.327 and the testing RMSE to 136.462, though the gap widened slightly to 34.135. Despite this increase in the gap, the model has improved, as evidenced by the reduced RMSE values for training and testing. The decline in RMSE indicates that the newly engineered features have enhanced the model's accuracy.

<u>Regression Tree Model</u>

Before feature engineering, the K-NN model's training RMSE was 119.664 and testing RMSE was 111.620, a difference of 8.044. After feature engineering, the training RMSE slightly rose to 111.024, but the testing RMSE dropped significantly to 118.549, narrowing the gap to 7.525. This reduced gap indicates better model generalization to unseen data. After feature engineering, the overall decrease in RMSE suggests the added features enhanced the model's performance.

Recommendation:

Here is what we have found:

- Lowest In-sample RMSE: K-NN model with 102.327
- Lowest Out-of-sample RMSE: Linear Regression model with 117.136
- Smallest RMSE Gap (In-Sample vs. Out-of-Sample): Linear Regression model with 2.152

The K-NN model has the lowest in-sample RMSE but performs poorly on the out-of-sample data, indicating potential overfitting.

The Linear Regression model has the smallest gap between in-sample and out-of-sample RMSE and the lowest out-of-sample RMSE, indicating good generalization.

The Regression Tree model shows an increase in out-of-sample RMSE after feature engineering, which may not be desirable.

Based on the RMSE for both in-sample and out-of-sample, and the difference between these values, the <u>Linear Regression model</u> would be the recommended choice. It offers the best generalization to unseen data, as evidenced by the smallest RMSE gap and the lowest out-of-sample RMSE.

5. **(35 points) Engage in parameter tuning to optimize the performance of linear regression, k-NN, and regression tree techniques. Use cross-validations with 10 folds to estimate the generalization performance. Present the results for each of the three techniques and discuss which one yields the best performance.**

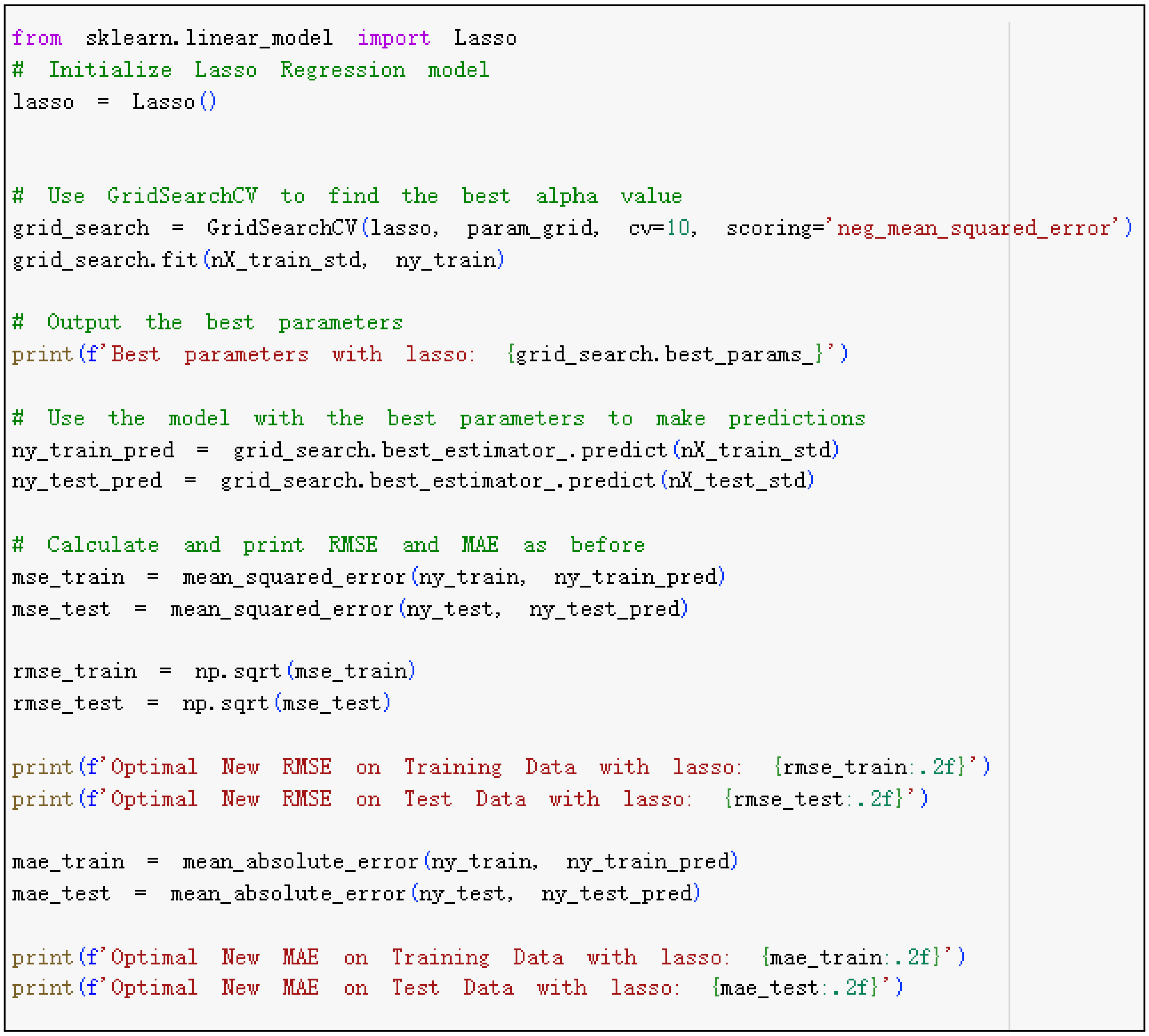
   [part a is worth 35 points in total:
   10 points for correctly optimizing at least two parameters for the linear regression model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results
   10 points for correctly optimizing at least two parameters for the linear k-NN model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results
   10 points for correctly optimizing at least two parameters for the linear regression tree model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results
   5 points for discussing which of the three models yields the best performance.]

Observations:

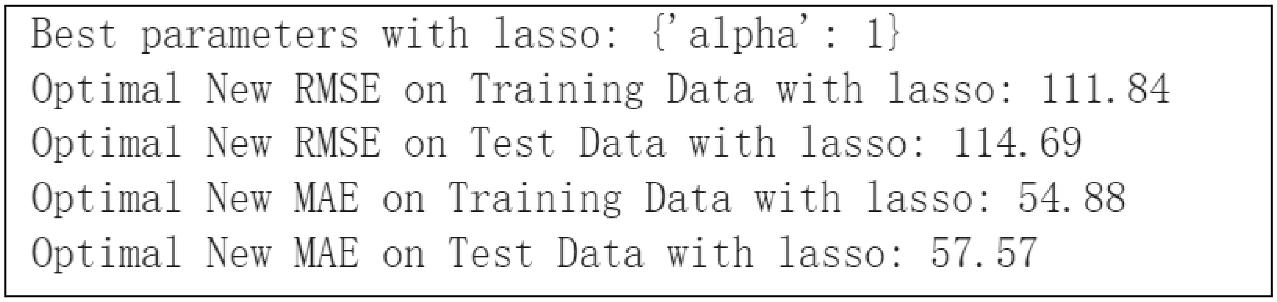
We used Grid Search cross-validation to find the best alpha for the lasso regression model and ridge regression model, respectively. Then, we calculated RMSE and MAE for both in-sample and out-of-sample datasets. The RMSE for lasso regression on training data is 111.84, and on test data is 114.69. The RMSE for ridge regression on training data is 112.22, and on test data is 114.88. After feature engineering, the overall decrease in RMSE suggests the added features enhanced the model's performance.
We used GridSearch cross-validation to find the best number of neighbors and weights. Then we calculated RMSE and MAE for both in-sample and out-of-sample datasets. The RMSE on training data is 109.27, and on test data is 123.33. After feature engineering, the overall decrease in RMSE suggests the added features enhanced the model's performance.
We used GridSearch cross-validation to find max_depth ranging from 1 to 31 and minimum sample split. Then, we calculated RMSE and MAE for both in-sample and out-of-sample datasets. The RMSE on training data is 122.27, and on test data is 126.14. After feature engineering, the overall decrease in RMSE suggests the added features enhanced the model's performance.

Linear Regression after parameter Tuning

We applied two ways of regularization- Lasso and Ridge to optimize the model and prevent it from overfitting. With the following codes and process, we compared the results of them and found the best Parameters for the Linear Regression Model: **Lasso Alpha =1 and Ridge Alpha = 100.**



```python
from sklearn.linear_model import Lasso
# Initialize Lasso Regression model
lasso = Lasso()


# Use GridSearchCV to find the best alpha value
grid_search = GridSearchCV(lasso, param_grid, cv=10, scoring='neg_mean_squared_error')
grid_search.fit(nX_train_std, ny_train)

# Output the best parameters
print(f'Best parameters with lasso: {grid_search.best_params_}')

# Use the model with the best parameters to make predictions
ny_train_pred = grid_search.best_estimator_.predict(nX_train_std)
ny_test_pred = grid_search.best_estimator_.predict(nX_test_std)

# Calculate and print RMSE and MAE as before
mse_train = mean_squared_error(ny_train, ny_train_pred)
mse_test = mean_squared_error(ny_test, ny_test_pred)

rmse_train = np.sqrt(mse_train)
rmse_test = np.sqrt(mse_test)

print(f'Optimal New RMSE on Training Data with lasso: {rmse_train:.2f}')
print(f'Optimal New RMSE on Test Data with lasso: {rmse_test:.2f}')

mae_train = mean_absolute_error(ny_train, ny_train_pred)
mae_test = mean_absolute_error(ny_test, ny_test_pred)

print(f'Optimal New MAE on Training Data with lasso: {mae_train:.2f}')
print(f'Optimal New MAE on Test Data with lasso: {mae_test:.2f}')
```



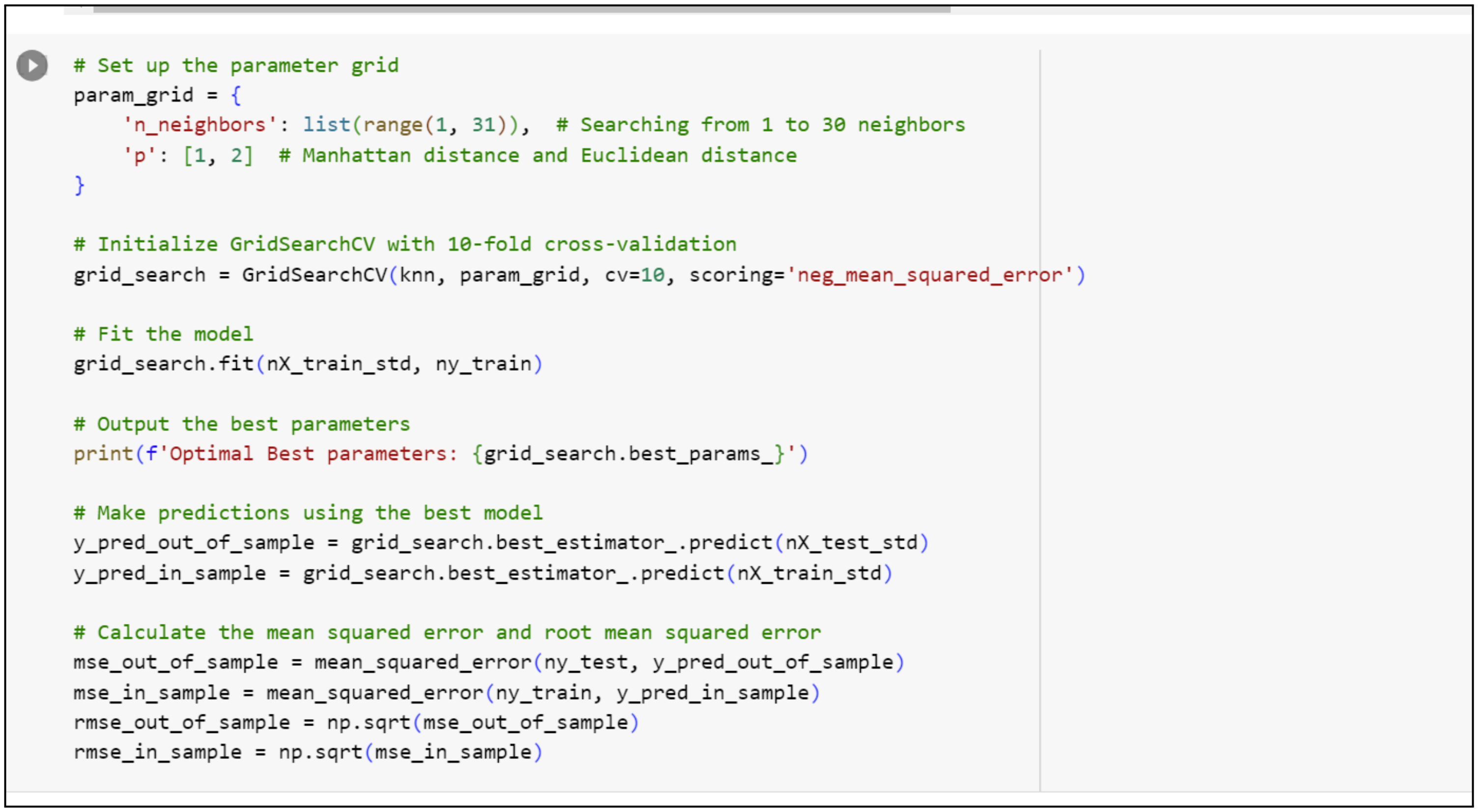
```
Best parameters with lasso: {'alpha': 1}
Optimal New RMSE on Training Data with lasso: 111.84
Optimal New RMSE on Test Data with lasso: 114.69
Optimal New MAE on Training Data with lasso: 54.88
Optimal New MAE on Test Data with lasso: 57.57
```

```python
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_absolute_error

# Initialize Ridge Regression model
ridge = Ridge()

# Set up the grid of alpha values to search over
param_grid = {'alpha': [0.001, 0.01, 0.1, 1, 10, 100]}

# Use GridSearchCV to find the best alpha value
grid_search = GridSearchCV(ridge, param_grid, cv=10, scoring='neg_mean_squared_error')
grid_search.fit(nX_train_std, ny_train)

# Output the best parameters
print(f'Best parameters: {grid_search.best_params_}')

# Use the model with the best parameters to make predictions
ny_train_pred = grid_search.best_estimator_.predict(nX_train_std)
ny_test_pred = grid_search.best_estimator_.predict(nX_test_std)

# Calculate and print RMSE and MAE as before
mse_train = mean_squared_error(ny_train, ny_train_pred)
mse_test = mean_squared_error(ny_test, ny_test_pred)

rmse_train = np.sqrt(mse_train)
rmse_test = np.sqrt(mse_test)

print(f'Optimal New RMSE on Training Data with ridge: {rmse_train:.2f}')
print(f'Optimal New RMSE on Test Data with ridge: {rmse_test:.2f}')

mae_train = mean_absolute_error(ny_train, ny_train_pred)
mae_test = mean_absolute_error(ny_test, ny_test_pred)

print(f'Optimal New MAE on Training Data with ridge: {mae_train:.2f}')
print(f'Optimal New MAE on Test Data with ridge: {mae_test:.2f}')
```

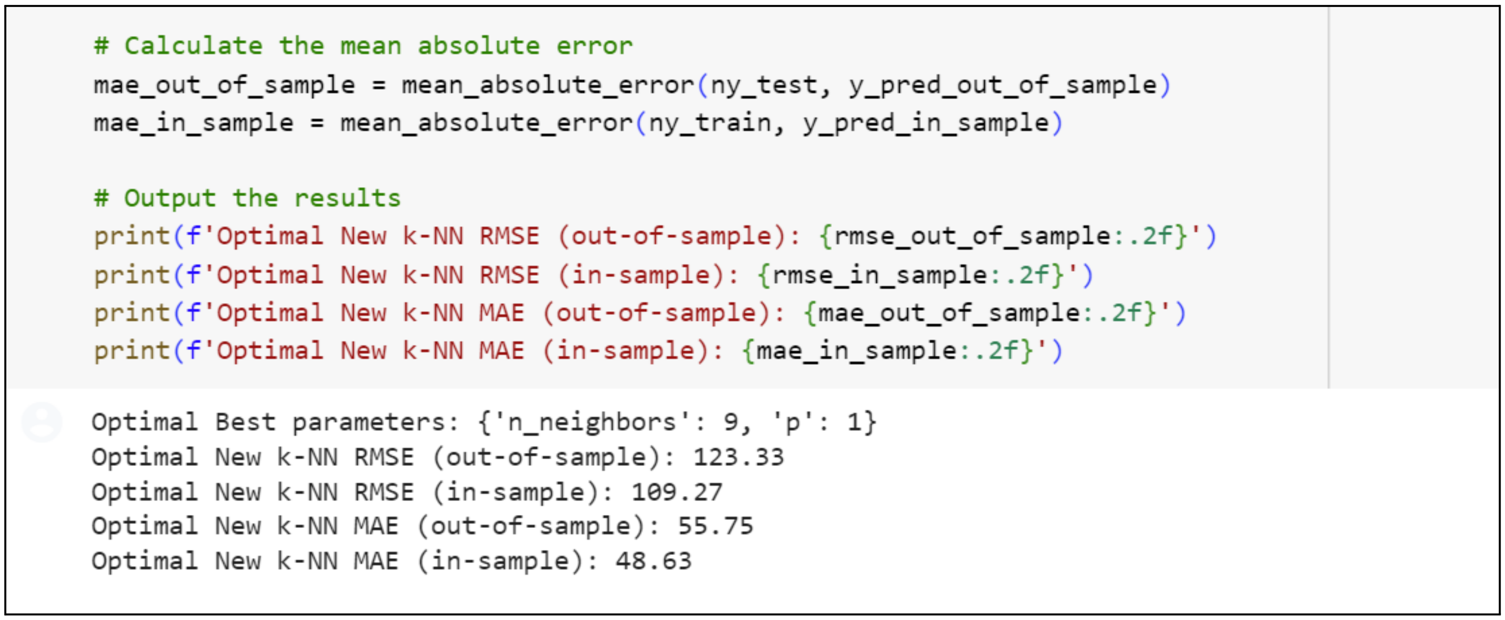
```
Best parameters: {'alpha': 100}
Optimal New RMSE on Training Data with ridge: 112.22
Optimal New RMSE on Test Data with ridge: 114.88
Optimal New MAE on Training Data with ridge: 56.12
Optimal New MAE on Test Data with ridge: 58.23
```

KNN after parameter tuning

We executed a grid search to find the best parameters for our k-NN regression model, using mean squared error as the metric to minimize. The grid search is conducted over a range of n_neighbors (from 1 to 30) and two distance metrics (p = 1 for Manhattan distance, p = 2 for Euclidean distance). As the result, **the best Parameters for the kNN model: k=9 and P=1**



```
# Set up the parameter grid
param_grid = {
    'n_neighbors': list(range(1, 31)),  # Searching from 1 to 30 neighbors
    'p': [1, 2]  # Manhattan distance and Euclidean distance
}

# Initialize GridSearchCV with 10-fold cross-validation
grid_search = GridSearchCV(knn, param_grid, cv=10, scoring='neg_mean_squared_error')

# Fit the model
grid_search.fit(nX_train_std, ny_train)

# Output the best parameters
print(f'Optimal Best parameters: {grid_search.best_params_}')

# Make predictions using the best model
y_pred_out_of_sample = grid_search.best_estimator_.predict(nX_test_std)
y_pred_in_sample = grid_search.best_estimator_.predict(nX_train_std)

# Calculate the mean squared error and root mean squared error
mse_out_of_sample = mean_squared_error(ny_test, y_pred_out_of_sample)
mse_in_sample = mean_squared_error(ny_train, y_pred_in_sample)
rmse_out_of_sample = np.sqrt(mse_out_of_sample)
rmse_in_sample = np.sqrt(mse_in_sample)
```

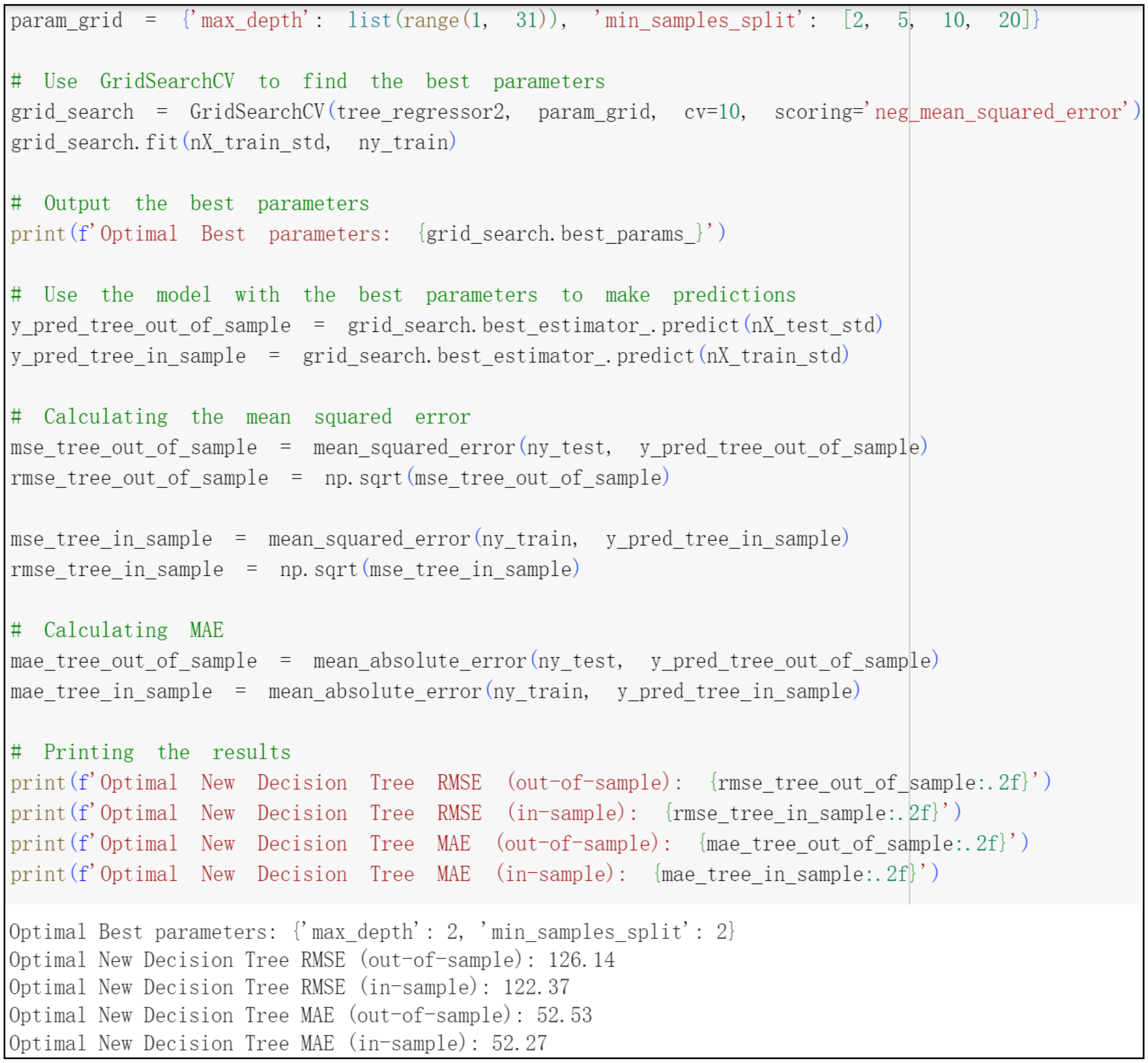


```
# Calculate the mean absolute error
mae_out_of_sample = mean_absolute_error(ny_test, y_pred_out_of_sample)
mae_in_sample = mean_absolute_error(ny_train, y_pred_in_sample)

# Output the results
print(f'Optimal New k-NN RMSE (out-of-sample): {rmse_out_of_sample:.2f}')
print(f'Optimal New k-NN RMSE (in-sample): {rmse_in_sample:.2f}')
print(f'Optimal New k-NN MAE (out-of-sample): {mae_out_of_sample:.2f}')
print(f'Optimal New k-NN MAE (in-sample): {mae_in_sample:.2f}')
```

```
Optimal Best parameters: {'n_neighbors': 9, 'p': 1}
Optimal New k-NN RMSE (out-of-sample): 123.33
Optimal New k-NN RMSE (in-sample): 109.27
Optimal New k-NN MAE (out-of-sample): 55.75
Optimal New k-NN MAE (in-sample): 48.63
```

DecisionTree Regressor after parameter tuning

As the previous step, we utilized a grid search to find out that the **best Parameters: max_depth=4 and min_sample_splits=2**



```
param_grid = {'max_depth': list(range(1, 31)), 'min_samples_split': [2, 5, 10, 20]}

# Use GridSearchCV to find the best parameters
grid_search = GridSearchCV(tree_regressor2, param_grid, cv=10, scoring='neg_mean_squared_error')
grid_search.fit(nX_train_std, ny_train)

# Output the best parameters
print(f'Optimal Best parameters: {grid_search.best_params_}')

# Use the model with the best parameters to make predictions
y_pred_tree_out_of_sample = grid_search.best_estimator_.predict(nX_test_std)
y_pred_tree_in_sample = grid_search.best_estimator_.predict(nX_train_std)

# Calculating the mean squared error
mse_tree_out_of_sample = mean_squared_error(ny_test, y_pred_tree_out_of_sample)
rmse_tree_out_of_sample = np.sqrt(mse_tree_out_of_sample)

mse_tree_in_sample = mean_squared_error(ny_train, y_pred_tree_in_sample)
rmse_tree_in_sample = np.sqrt(mse_tree_in_sample)

# Calculating MAE
mae_tree_out_of_sample = mean_absolute_error(ny_test, y_pred_tree_out_of_sample)
mae_tree_in_sample = mean_absolute_error(ny_train, y_pred_tree_in_sample)

# Printing the results
print(f'Optimal New Decision Tree RMSE (out-of-sample): {rmse_tree_out_of_sample:.2f}')
print(f'Optimal New Decision Tree RMSE (in-sample): {rmse_tree_in_sample:.2f}')
print(f'Optimal New Decision Tree MAE (out-of-sample): {mae_tree_out_of_sample:.2f}')
print(f'Optimal New Decision Tree MAE (in-sample): {mae_tree_in_sample:.2f}')
```

```
Optimal Best parameters: {'max_depth': 2, 'min_samples_split': 2}
Optimal New Decision Tree RMSE (out-of-sample): 126.14
Optimal New Decision Tree RMSE (in-sample): 122.37
Optimal New Decision Tree MAE (out-of-sample): 52.53
Optimal New Decision Tree MAE (in-sample): 52.27
```

Based on the above result, we can determine that linear Regression is the best model. Linear Regression is the most effective among the three models evaluated based on its RMSE values. It has the lowest out-of-sample RMSE of 114.88, indicating high accuracy. Furthermore, the small difference of 2.66 between its in-sample and out-of-sample RMSE suggests that it generalizes well to unseen data, reducing the risk of overfitting.