Homework #4 Part B

Neil Jiang, Hayley Huang, Foster Mosden, Icy Wang

Note: This is a team homework assignment. Discussing this homework with your classmates outside your MSBA team is a **violation** of the Honor Code. If you **borrow code** from somewhere else, please add a comment in your code to **make it clear** what the source of the code is (e.g., a URL would sufficient). If you borrow code and you don't provide the source, it is a violation of the Honor Code.

Total grade: _____ out of ___145___ points

(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.
Whenever applicable use random state 42 (10 points).

(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.

a. 1.

The dataset includes 23 independent variables on the customer general information and one target variable being the amount spent by customer in test mailing in USD. The detailed descriptive statistics result is as follows:

	sequence_number	US	source_a	source_x	source_w	Freq	last_update_days_ago	1st_update_days_ago	Web order	Gender=male	Address_is_res	Purchase	Spending
count	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000
mean	1000.5	0.8245	0.1265	0.018	0.1375	1.417	2155.1	2435.6	0.426	0.5245	0.221	0.5	102.561
std	577.495	0.38049	0.3325	0.13298	0.34446	1.40574	1141.3	1077.87	0.49462	0.49952	0.41502	0.50013	186.75
min	1	0	0	0	0	0	1	1	0	0	0	0	0
25%	500.75	1	0	0	0	1	1133	1671.25	0	0	0	0	0
50%	1000.5	1	0	0	0	1	2280	2721	0	1	0	0.5	1.855
75%	1500.25	1	0	0	0	2	3139.25	3353	1	1	0	1	152.533
max	2000	1	1	1	1	15	4188	4188	1	1	1	1	1500.06

8 rows × 25 columns

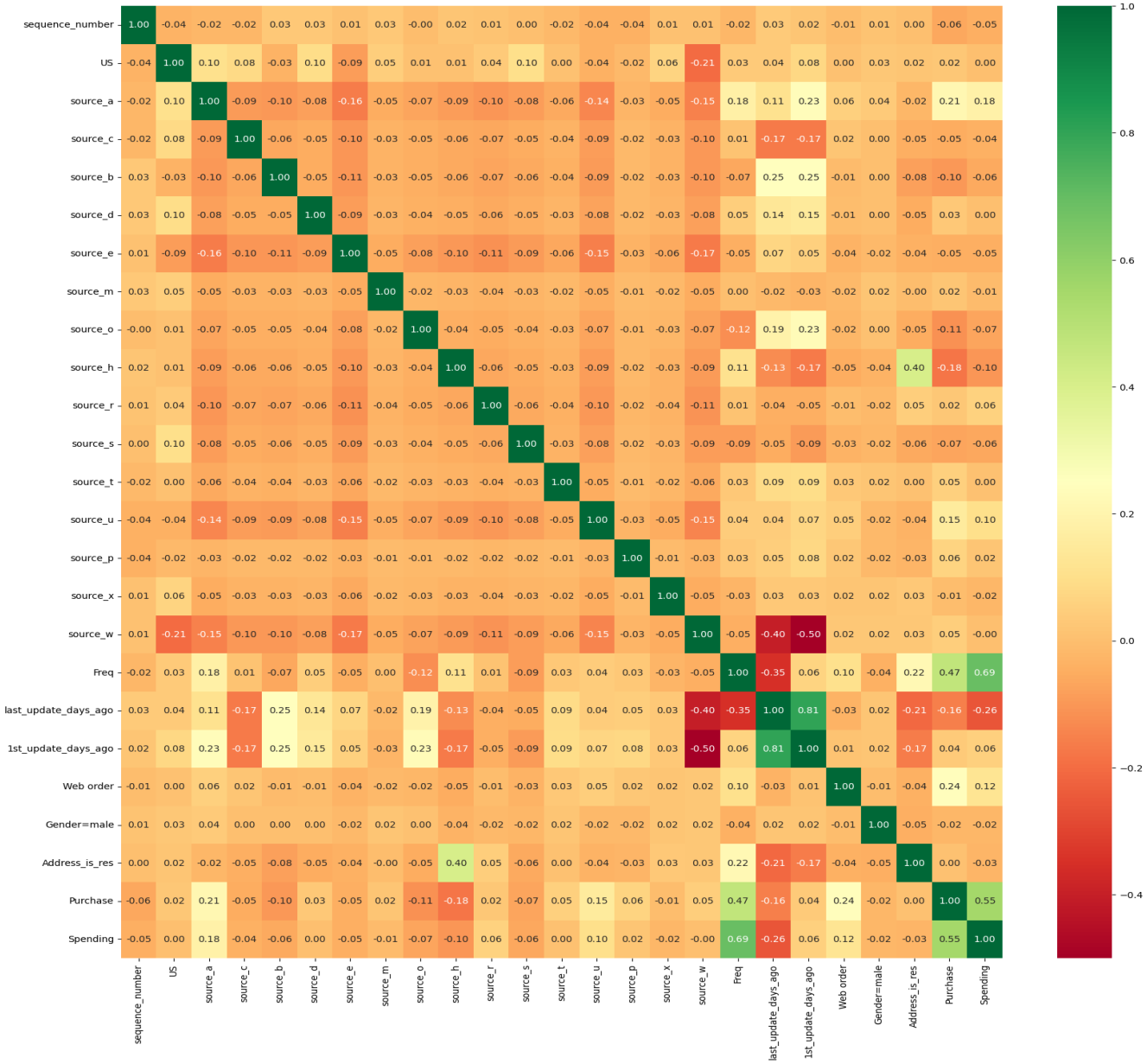
For simplicity, we made the columns between source_a and source_x hidden.
The corresponding column datatype are as follows:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 25 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   sequence_number       2000 non-null   int64
 1   US                    2000 non-null   int64
 2   source_a              2000 non-null   int64
 3   source_c              2000 non-null   int64
 4   source_b              2000 non-null   int64
 5   source_d              2000 non-null   int64
 6   source_e              2000 non-null   int64
 7   source_m              2000 non-null   int64
 8   source_o              2000 non-null   int64
 9   source_h              2000 non-null   int64
 10  source_r              2000 non-null   int64
 11  source_s              2000 non-null   int64
 12  source_t              2000 non-null   int64
 13  source_u              2000 non-null   int64
 14  source_p              2000 non-null   int64
 15  source_x              2000 non-null   int64
 16  source_w              2000 non-null   int64
 17  Freq                  2000 non-null   int64
 18  last_update_days_ago  2000 non-null   int64
 19  1st_update_days_ago   2000 non-null   int64
 20  Web order             2000 non-null   int64
 21  Gender=male           2000 non-null   int64
 22  Address_is_res        2000 non-null   int64
 23  Purchase              2000 non-null   int64
 24  Spending              2000 non-null   float64
dtypes: float64(1), int64(24)
memory usage: 390.8 KB
```

Then, we conducted pairwise correlation analysis to check if there are highly correlated features:

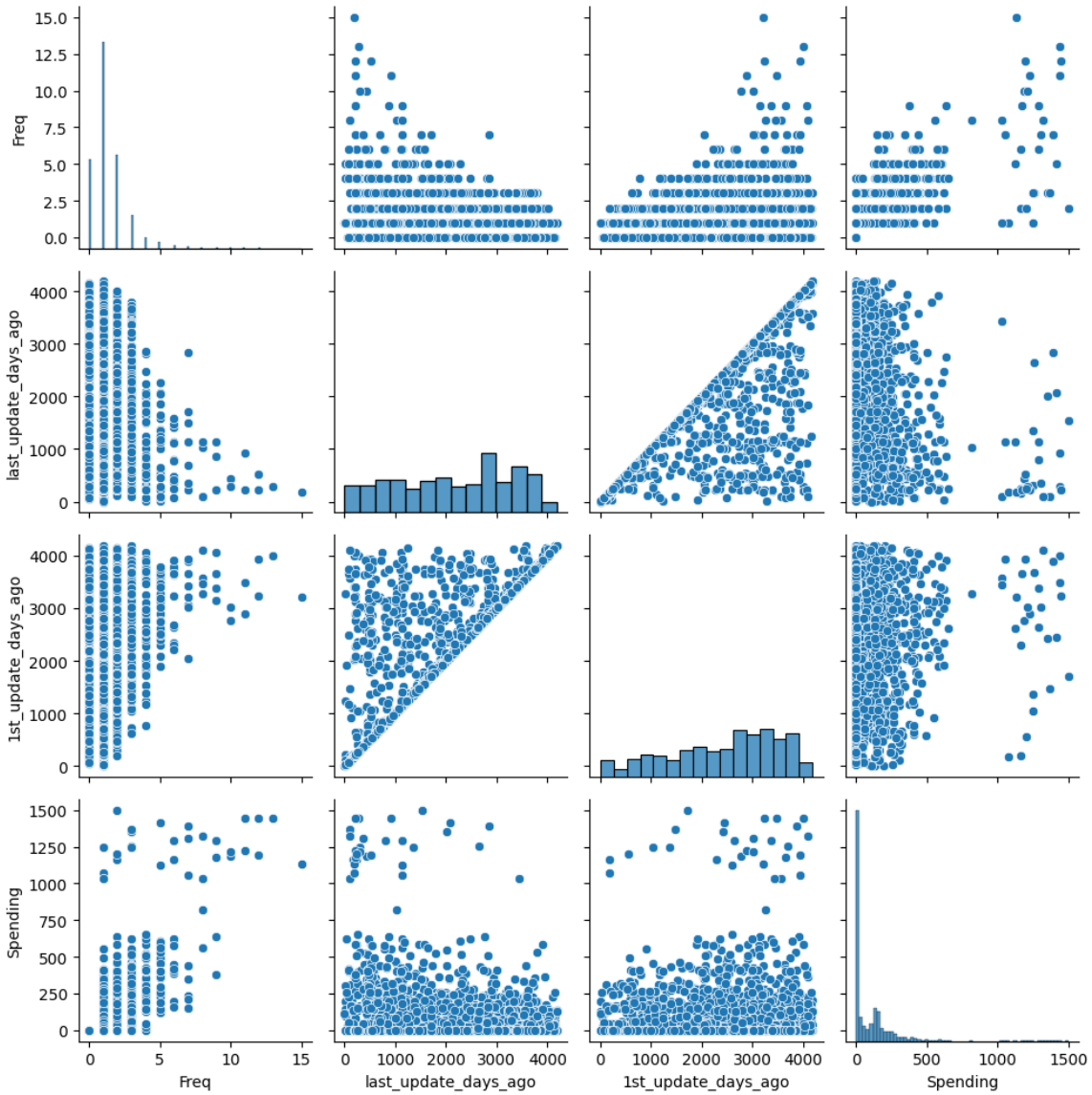
```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(20,20))
sns.heatmap(df.corr(), annot=True, fmt=".2f",cmap='RdYlGn')
plt.show()
#no highly correlated features
```



There are no apparent highly correlated features that need to be dropped in this dataset. One of the higher correlations is between last_update_days_ago and 1st_update_days_ago, which is a positive 0.81 correlation relationship. This is a good indication that we could conduct some feature engineering in later part of the assignment.

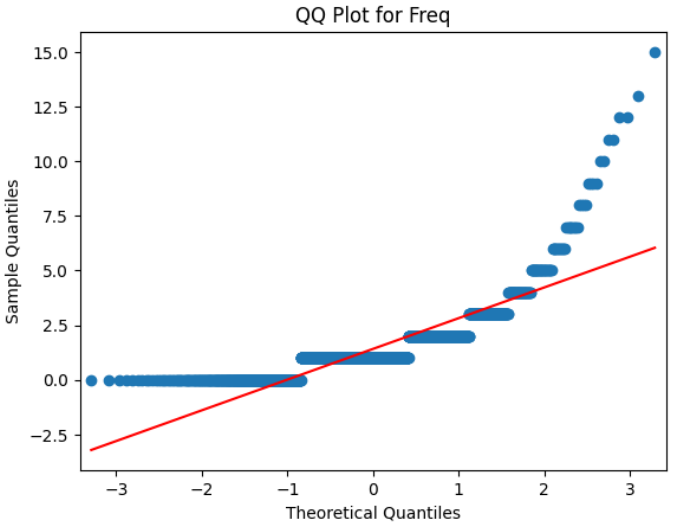
Next, we created histograms for the continuous variables, including the target variable to help visualize the distribution of these variables:

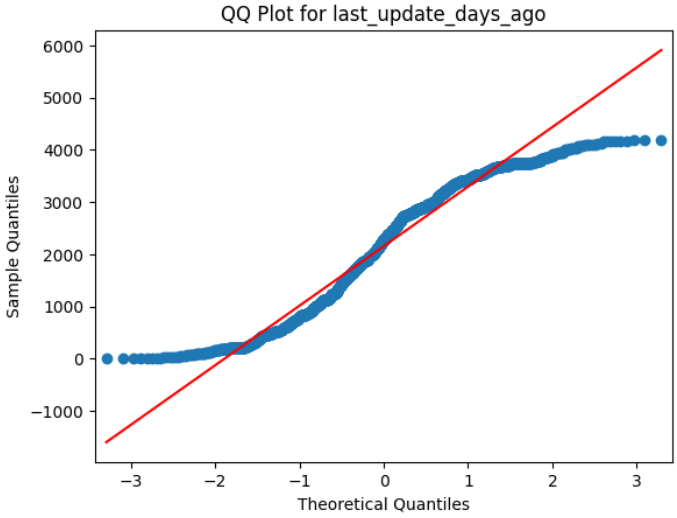
```
cols = ['Freq','last_update_days_ago','1st_update_days_ago','Spending'] # select

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

#2 update_days_ago correlated because they have strict before and after relations
```

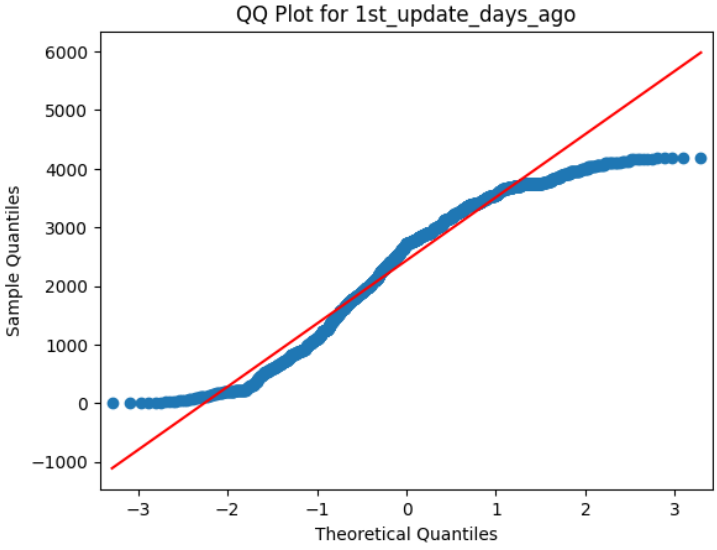


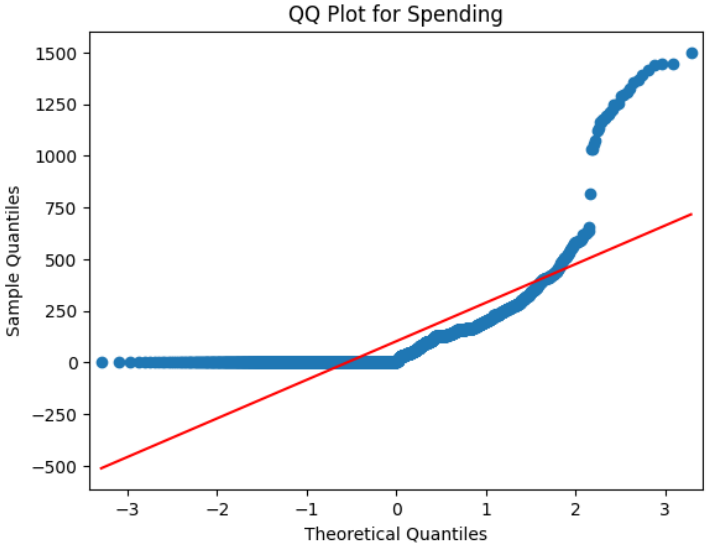
To better assess the skewness, we further created QQ plot for the above variables:

```python
# QQ Plots

import statsmodels.api as sm
import matplotlib.pyplot as plt

cols = ['Freq', 'last_update_days_ago', '1st_update_days_ago', 'Spending']  # Se

# Create QQ plots for each column
for col in cols:
    sm.qqplot(df[col], line='s')
    plt.title(f'QQ Plot for {col}')
    plt.show()
```









```
# Calculate skewness for each column
skewness_scores = df.skew(numeric_only=True)

# Print or return the skewness scores
print(skewness_scores)

#only binary features are skewed
```

The corresponding skewness scores are as follows:

```
sequence_number          0.000000
US                      -1.707406
source_a                 2.248898
source_c                 3.865083
source_b                 3.708250
source_d                 4.601243
source_e                 1.950918
source_m                 7.596669
source_o                 5.189007
source_h                 4.015870
source_r                 3.419007
source_s                 4.284091
source_t                 6.602953
source_u                 2.355155
source_p                12.803068
source_x                 7.256228
source_w                 2.106847
Freq                     2.981068
last_update_days_ago    -0.187871
1st_update_days_ago     -0.489562
Web order                0.299521
Gender=male             -0.098192
Address_is_res           1.345846
Purchase                 0.000000
Spending                 3.928441
dtype: float64
```

which shows that some features are highly skewed including y (Spending) that's better transformed in later modeling.

a.2 Linear Regression Model

Firstly, we built a simple linear regression model.

This is the steps for creating the train test split:

```
from sklearn.model_selection import train_test_split    # split validation class

X = df.iloc[:, :-1].values                              # use all features as att
y = df['Spending'].values                                   # set last column as

X_train, X_test, y_train, y_test = train_test_split(X,  # split validation
                                                    y,
                                                    test_size=0.3,
                                                    random_state=42)
```

Having completed the train-test split, we then fitted the model and tested the performance on the test data using different metrics

```
[ ] ################################### Fit a Linear Regression Model  ##############
    # X_linear = df.drop('source_a',axis=1).iloc[:, :-1].values

    from sklearn.linear_model import LinearRegression

    slr2 = LinearRegression()                              # linear regression class
    slr2.fit(X_train, y_train)                             # fit model to train data
    y_train_pred = slr2.predict(X_train)                   # apply model to train dat
    y_test_pred = slr2.predict(X_test)                     # apply model to test data

    print('Slope: %.3f', slr2.coef_)                       # estimated coefficients f

    ######################### Linear Regression Model - Evaluation Metrics  ##########

    from math import sqrt
    from sklearn.metrics import mean_absolute_error # mean absolute error regression
    from sklearn.metrics import mean_squared_error  # mean squared error regression l

    # See all regression metrics here http://scikit-learn.org/stable/modules/model_ev
    print('MSE train: %.3f, test: %.3f' % ( # mean_absolute_error
            mean_squared_error(y_train, y_train_pred),
            mean_squared_error(y_test, y_test_pred)))  # y_test: Ground truth (correc
                                                       # y_test_pred: Estimated targe

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

    print('MAE train: %.3f, test: %.3f' % ( # mean_squared_error
            mean_absolute_error(y_train, y_train_pred),
            mean_absolute_error(y_test, y_test_pred))) # y_test: Ground truth (correc
                                                       # y_test_pred: Estimated targe
```

The model we have generates the following performance values:
MSE train: 14520.069, test: 14829.884
RMSE train: 120.499, test: 121.778
MAE train: 69.945, test: 67.281

RMSE stands for Root Mean Square Error. It is a widely used metric in statistics and machine learning for evaluating the accuracy of a predictive model, typically a regression model. RMSE measures the average magnitude of errors between predicted values and actual (observed) values in a dataset. It is particularly useful when we want to quantify how well our model's predictions align with the real data.

Mathematically, MSE and RMSE are calculated as follows:

$$\text{MSE} = \frac{\Sigma(predicted - actual)^2}{n}$$

$$\text{RMSE} = \sqrt{\frac{\Sigma(predicted - actual)^2}{n}}$$

MAE stands for Mean Absolute Error calculated by $\text{MAE} = \frac{\Sigma|predicted - actual|}{n}$. Unlike the Root Mean Square Error (RMSE), which squares the differences, MAE treats all errors equally and does not penalize large errors more heavily. We want to penalize big errors so we'll focus on comparing RMSE.

To have a clear view of the performance of the model. We also included a cross validation assessment.

```
#Use cross-validation with 10 folds to estimate the generalization performance
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=10,          # number of re-shuffling & splitting ite
                  test_size=0.3
                  ,random_state=42)

scores = cross_val_score(estimator=slr2,              # 10-fold cross validation
                         X=X,
                         y=y,
                         cv=cv,
                         scoring = 'neg_mean_squared_error',
                         n_jobs=1)
print('Nested MSE score:', scores.mean(), " +/- ", scores.std())
scores = cross_val_score(estimator=slr2,              # 10-fold cross validation
                         X=X,
                         y=y,
                         cv=cv,
                         scoring = 'neg_root_mean_squared_error',
                         n_jobs=1)
print('Nested RMSE score:', scores.mean(), " +/- ", scores.std())
```

In scoring='neg_mean_squared_error' orscoring='neg_root_mean_squared_error' , scikit-learn negates the MSE and RMSE, meaning it multiplies it by -1. The reason for negating the MSE is that scikit-learn's scoring convention assumes that higher values are better for scoring. Thus, we'll compare the negative value of them with directly calculated RMSE.

The 10 fold cross validation generates an average nested MSE score of 14482.35 and an average nested RMSE score of 119.62, both being slightly lower than the values in the first step.

a.3. Knn

For the KNN model, we first normalized the data with min max scaling, after which we went to a knn regression with k = 3. The RMSE value resulting from the model is 154.12

```
################################### kNN Regressor Example    ###################################
from sklearn import neighbors
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from math import sqrt

#normalize data with min max scaling
sc = MinMaxScaler(feature_range=(0, 1))
sc.fit(X_train)                                # scaling parameters to be learned for later scaling

x_train_scaled = sc.transform(X_train)         # scaling of features in train data
x_train_sc = pd.DataFrame(x_train_scaled)      # constructing dataFrame

x_test_scaled = sc.transform(X_test)           # scaling of features in test data
x_test_sc = pd.DataFrame(x_test_scaled)        # constructing dataFrame

x_scaled = sc.transform(X)           # scaling of features in test data
x_sc = pd.DataFrame(x_scaled)

#3NN regressor
knn_regressor = neighbors.KNeighborsRegressor(n_neighbors = 3)

#Fit and Evaluate Model
knn_regressor.fit(x_train_sc, y_train)       # fit the model
pred=knn_regressor.predict(x_test_sc)        # make prediction on test set
error = sqrt(mean_squared_error(y_test,pred)) # calculate rmse on test set
print('RMSE value is:', error)
```

With 10-fold cross validation, we are able to get an better understanding of the performance of the model: mean Nested MSE score: 24259.61, mean Nested RMSE score: 155.32

```
#Use cross-validation with 10 folds to estimate the generalization performance
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=knn_regressor,              # 10-fold cross validation
                         X=x_sc,
                         y=y,
                         cv=cv,
                         scoring = 'neg_mean_squared_error',
                         n_jobs=1)
print('Nested MSE score:', scores.mean(), " +/- ", scores.std())
scores = cross_val_score(estimator=knn_regressor,              # 10-fold cross validation
                         X=x_sc,
                         y=y,
                         cv=cv,
                         scoring = 'neg_root_mean_squared_error',
                         n_jobs=1)
print('Nested RMSE score:', scores.mean(), " +/- ", scores.std())
```

a.4. Decision Tree

Finally, we conducted a decision tree model for the target variable as well.

```
####################################       Regressor Tree - Numeric Prediction

from sklearn.tree import DecisionTreeRegressor


# Decision Tree Regressor
# Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.tree.De
tree = DecisionTreeRegressor(criterion = 'squared_error',                    # the
                                                                             # supp
                                                                             # whic
                                                                             # "abs
                             max_depth=3,
                             random_state=42)                                    #

scores = cross_val_score(tree,                                               # cros
                         X,
                         y,
                         scoring = 'neg_root_mean_squared_error',
                         cv=cv)                                              # wher
                                                                             # In s
                                                                             # perc

print("RMSE score: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2)) # estim
```

```
RMSE score: -119.76 (+/- 34.57)
```

This yields an RMSE score of 119.76

The cross validation gives us a general performance result of average Nested MSE score: 24259.61 and average Nested RMSE score: 119.76

```
#Use cross-validation with 10 folds to estimate the generalization performance
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=knn_regressor,              # 10-fold cross validation
                         X=x_sc,
                         y=y,
                         cv=cv,
                         scoring = 'neg_mean_squared_error',
                         n_jobs=1)
print('Nested MSE score:', scores.mean(), " +/- ", scores.std())
scores = cross_val_score(estimator=tree,               # 10-fold cross validation
                         X=x_sc,
                         y=y,
                         cv=cv,
                         scoring = 'neg_root_mean_squared_error',
                         n_jobs=1)
print('Nested RMSE score:', scores.mean(), " +/- ", scores.std())
```

Here is the overview of the RMSE scores we obtained from three models:
Linear Regression : 119.62
KNN: 155.32
Decision Tree: 119.76

Based on the preliminary analysis without hyperparameter tuning, we think Linear Regression yields the best performance because it has the lowest RMSE score from cross validation

(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.

[part a is worth 50 points in total:
10 points for correctly building the new 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 building the new 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 building the new regression tree model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results
20 points for discussing if the generalization performance was improved or not for each of the techniques (linear regression, kNN, and regression tree) and justifying why it was improved or alternatively why it was not improved]

B - Linear Regression

```
median_freq = df_linreg['Freq'].median()
# Create a new column 'Above_Median_Freq' with values 1 if Freq is above median,
else 0
df_linreg['Above_Median_Freq'] = (df_linreg['Freq'] > median_freq).astype(int)

# 2. Extract the month and year from the last_update_days_ago and
1st_update_days_ago features
df_linreg['Last_Update_Month'] = pd.to_datetime(df_linreg['last_update_days_ago'],
unit='D').dt.month
df_linreg['Last_Update_Year'] = pd.to_datetime(df_linreg['last_update_days_ago'],
unit='D').dt.year
df_linreg['First_Update_Month'] = pd.to_datetime(df_linreg['1st_update_days_ago'],
unit='D').dt.month
df_linreg['First_Update_Year'] = pd.to_datetime(df_linreg['1st_update_days_ago'],
unit='D').dt.year
# Create a new column that captures the time between the first and last update
df_linreg['Update_Time_Difference'] = abs(df_linreg['last_update_days_ago'] -
df_linreg['1st_update_days_ago'])
```

After some research, trial, and error, we decided to perform the above feature engineering in preparation for our linear regression model. When analyzing the quartile values of `Freq`, we found that most values fell between 0 and 1. Thus, we decided to engineer a new column titled “Above_Median_Freq” which assigns a 1 to rows that have a Freq value above the median, and a 0 for rows where Freq is at or below the median. We also stripped down the “..._update_days_ago” columns into their month and year, and created a new column to store the time difference between the first and last update. In all, we broke down our data into more basic features, and added a

feature to store the time difference between updates. We found this helped improve the linearity of the features contributing to our linear regression.

```
# Check if any column contains 0 values
zero_columns = (df_linreg == 0).any()

# Add 1/3 to all values of columns containing 0
for column in zero_columns[zero_columns].index:
     df_linreg[column] = df_linreg[column] + (1/3)

# Logarithm of the column "Freq"
df_linreg['Freq_log'] = np.log(df_linreg['Freq'])
df_linreg = df_linreg.drop(['Freq'], axis=1)

# Logarithm of the column "Spending"
df_linreg['Spending_log'] = np.log(df_linreg['Spending'])
df_linreg = df_linreg.drop(['Spending'], axis=1)

# Logarithm of the column "Spending"
df_linreg['Update_Time_Difference_log'] =
np.log(df_linreg['Update_Time_Difference'])
df_linreg = df_linreg.drop(['Update_Time_Difference'], axis=1)

# Cube root of the column "sequence_number"
df_linreg['sequence_number_log'] = np.log(df_linreg['sequence_number'])
df_linreg = df_linreg.drop(['sequence_number'], axis=1)

# Cube root of the column "last_update_days_ago"
df_linreg['last_update_days_ago_log'] = np.log(df_linreg['last_update_days_ago'])
df_linreg = df_linreg.drop(['last_update_days_ago'], axis=1)

# Cube root of the column "1st_update_days_ago"
df_linreg['1st_update_days_ago_log'] = np.log(df_linreg['1st_update_days_ago'])
df_linreg = df_linreg.drop(['1st_update_days_ago'], axis=1)
```

After feature engineering, we performed feature transformation. Again, this required a good deal of trial and error. Ultimately, we decided to log all of our continuous variables after adding ⅓ to all values of the column if any zeroes were present. We found that the log function was the most effective out of our known options to improve the linearity of our variables, which should theoretically improve the modeling fit of our linear regression

```
X = df_linreg.drop(['Spending_log'], axis=1)
y = df_linreg['Spending_log']

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
# Scale features:
from sklearn.preprocessing import StandardScaler
# Scale data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)          # computes the mean and std to be
used for scaling and performs scaling
X_test = scaler.transform(X_test)                # performs standardization of
test set X attributes by centering and scaling
```

Finally, we create our train test split and scale our x variables based on the training dataset. We can't scale our test dataset based on its' own values, because in a real world use of our model, our test set should be considered "unseen".

```
# Fit a Linear Regression Model
slr2 = LinearRegression()
slr2.fit(X_train, y_train)
y_train_pred = slr2.predict(X_train)
y_test_pred = slr2.predict(X_test)
print('Slope: %.3f', slr2.coef_)                        # estimated coefficients
for the linear regression model
```

```
Slope: %.3f [-2.59313770e-02  3.10440182e-02 -3.65330562e-02  8.82298908e-03
 -3.29725400e-03 -2.14523072e-02  2.59777811e-03 -1.46037094e-02
 -7.03194229e-02  1.22103599e-02 -4.47345006e-02  2.62376645e-03
 -2.40862677e-02 -1.88199218e-02 -9.09399023e-03 -4.39071562e-03
 -9.79290032e-03 -7.05021479e-03 -5.96542692e-02  2.76265069e+00
  5.63332931e-02  2.45020903e-02 -1.07799959e-01 -8.54372282e-03
  1.17001570e-01  1.31604940e-01  1.06335667e-01 -2.28476586e-02
 -1.70730047e-01  9.45740665e-02]
```

Now, we ran our basic linear regression model and returned our coefficients.

```
# See all regression metrics here
http://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics
print('MSE train: %.3f, test: %.3f' % ( # mean_absolute_error
     mean_squared_error(y_train, y_train_pred),
     mean_squared_error(y_test, y_test_pred)))

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

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

```
MSE train: 0.331, test: 0.326
RMSE train: 0.576, test: 0.571
MAE train: 0.360, test: 0.363
```

We ran some basic evaluation metrics on our linear regression with feature engineering. Because this version of our model has an engineered and transformed y variable (Spending_log), our error metrics are not comparable to our simple linear regression model we built in part A (more on comparing our models at the end of this section). Regardless, having the train and test error metrics do give us a picture of model fit. Given that our test error values seem to be lower or close to our training error, it may be the case that our model is overfitting slightly.

```
cv = ShuffleSplit(n_splits=10,        # number of re-shuffling & splitting
iterations
                  test_size=0.3
                  ,random_state=42)

scores = cross_val_score(estimator=slr2,              # 10-fold cross validation
                           X=X,
                           y=y,
                           cv=cv,
                           scoring = 'neg_mean_squared_error',
                           n_jobs=1)
print('Nested MSE score:', -scores.mean(), " +/- ", scores.std())
scores = cross_val_score(estimator=slr2,              # 10-fold cross validation
                           X=X,
                           y=y,
                           cv=cv,
                           scoring = 'neg_root_mean_squared_error',
                           n_jobs=1)
print('Nested RMSE score:', -scores.mean(), " +/- ", scores.std())
```

```
Nested MSE score: 0.33177511547911787  +/-  0.01795578684383654
Nested RMSE score: 0.575793549000504  +/-  0.015391699338348086
```

10 folds cross validation provides us with a better indication of generalization performance, although in this instance the error values we calculate are quite similar to our simple evaluation.

```
lasso_model = LassoCV(cv=cv, max_iter=10000)

# Fit the Lasso model to the training data
lasso_model.fit(X_train, y_train)

# Get selected feature indices (non-zero coefficients)
selected_feature_indices = np.where(lasso_model.coef_ != 0)[0]

# Get the names of the selected features
selected_feature_names = df_linreg.columns[selected_feature_indices]

# Extract the subset of features based on selected indices
X_subset_train = X_train[:, selected_feature_indices]

# Perform 10-fold cross-validation and calculate the RMSE
scores = np.sqrt(-cross_val_score(lasso_model, X_subset_train, y_train, cv=cv,
scoring='neg_mean_squared_error', n_jobs=-1))
mean_score = scores.mean()

# Print the selected features and their names
print("Selected Features (Names):", selected_feature_names)
print("Number of Selected Features:", len(selected_feature_names))
print("Cross-Validation RMSE:", mean_score)
```

```
Selected Features (Names): Index(['US', 'source_a', 'source_c', 'source_b', 'source_h', 'source_r',
       'source_s', 'source_p', 'Address_is_res', 'Purchase',
       'Above_Median_Freq', 'Last_Update_Month', 'Last_Update_Year',
       'First_Update_Year', 'Freq_log', 'Spending_log',
       'Update_Time_Difference_log', 'sequence_number_log',
       'last_update_days_ago_log'],
      dtype='object')
Number of Selected Features: 19
Cross-Validation RMSE: 0.5832054255209445
```

Finally, we run lasso regression. Based on RMSE, the lasso regression did not run much better than our other models. But, the selected features by the lasso regression do indicate to us that our feature engineering did produce valuable features, as nearly every single feature we engineered for the model appears on the selected features list by lasso regression.

```
Cross-Validation RMSE: 141.65537430133057
```

In our feature engineering stage, we got the log of Spending to help linearize the variable. However, this makes it difficult to compare to our non-engineered Linear Regression. Thus, we also ran the engineered linear regression with a non-log Spending variable, and got an RMSE of ~142. When compared to our non-engineered Linear regression RMSE of ~120, our engineered model still did worse. However, we just performed this step to make direct comparison easier and more fair.

B - k-NN Regression

```
# 1. Calculate the median of the 'Freq' column
median_freq = df_knn['Freq'].median()
# Create a new column 'Above_Median_Freq' with values 1 if Freq is above median, else 0
df_knn['Above_Median_Freq'] = (df_knn['Freq'] > median_freq).astype(int)
df_knn = df_knn.drop(['Freq'], axis=1)

# Perform binning, and time series feature engineering steps:

# 1. Calculate the median of the 'update_days_ago' columns
median_freq = df_knn['last_update_days_ago'].median()
# Create a new column 'Above_Median_last_update' with values 1 if last_update_days_ago is above median, else 0
df_knn['Above_Median_last_update'] = (df_knn['last_update_days_ago'] > median_freq).astype(int)
df_knn = df_knn.drop(['last_update_days_ago'], axis=1)

median_freq = df_knn['1st_update_days_ago'].median()
# Create a new column 'Above_Median_1st_update' with values 1 if '1st_update_days_ago' is above median, else 0
df_knn['Above_Median_1st_update'] = (df_knn['1st_update_days_ago'] > median_freq).astype(int)
df_knn = df_knn.drop(['1st_update_days_ago'], axis=1)
```

We started our knn regression by performing feature engineering. We performed research and some trial and error to figure out which kind of feature engineering, and what kind of features, benefit a k-NN regression the most. We once again created a binary variable column for whether or not a row was above or below the median Freq value. We used this idea to perform a similar binning column for the last_ and first_update_days_ago_columns.

We also performed the train-test split and feature scaling, the same as we did for our linear regression setup.

k-NN Regression Modeling

```
#3NN regressor
knn_regressor = neighbors.KNeighborsRegressor(n_neighbors = 3)

#Fit and Evaluate Model
knn_regressor.fit(X_train, y_train)        # fit the model
pred=knn_regressor.predict(X_test)         # make prediction on test set
error = sqrt(mean_squared_error(y_test,pred)) # calculate rmse on test set
print('RMSE value is:', error)
```

```
RMSE value is: 172.23444899048377
```

```
[228] #Use cross-validation with 10 folds to estimate the generalization performance
      from sklearn.model_selection import cross_val_score
      scores = cross_val_score(estimator=knn_regressor,             # 10-fold cross validation
                               X=X,
                               y=y,
                               cv=cv,
                               scoring = 'neg_mean_squared_error',
                               n_jobs=1)
      print('Nested MSE score:', scores.mean(), " +/- ", scores.std())
      scores = cross_val_score(estimator=knn_regressor,             # 10-fold cross validation
                               X=X,
                               y=y,
                               cv=cv,
                               scoring = 'neg_root_mean_squared_error',
                               n_jobs=1)
      print('Nested RMSE score:', scores.mean(), " +/- ", scores.std())
```

```
Nested MSE score: -43693.825559994446  +/-  5029.76451678513
Nested RMSE score: -208.67913860972345  +/-  12.117865698969885
```

Finally, we ran our modeling. our basic 3-NN regression resulted in an RMSE of ~172, while our 10 fold cross validation provided us with an RME of ~209. Unfortunately, this is actually a worse RMSE when

compared to the nested RMSE of our simple, unaltered KNN regression model. Ultimately, we would choose the simple, unaltered KNN regression model over our feature-engineered model.

B - Regression Tree

Our feature engineering, scaling, and train-test split steps are the same for our regression tree as they were for our k-NN regression (aside from changing the variable names, of course). Again, our research and trial-and-error approach indicated to us that our k-NN regression and Regression Tree benefitted from the same feature engineering steps, and we ultimately used the same steps for both models.

```
tree = DecisionTreeRegressor(criterion = 'squared_error',



                             max_depth=3,
                             random_state=42)

scores = cross_val_score(tree,
                         X,
                         y,
                         scoring = 'neg_root_mean_squared_error',
                         cv=cv)



print("RMSE score: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
RMSE score: -143.31 (+/- 27.15)
```

Here you can see that our 10 fold cross evaluation resulted in an RMSE of ~143, which is once again actually slightly worse than our simple non-engineered regressor tree, which had an RMSE of ~120.

Overall, despite our hard work and trial and error, it would appear that our feature engineering did not result in improved RMSE numbers over our non-engineered models. Thus, we would likely choose our initial models over these newly engineered one's. However, it’s possible that with the right hyper-parameter tuning, our feature engineering could improve performance.

(c) (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.

In part 1, we used cross validation to assess the generalization performance of the models, in this part, we used nest cross validation to perform model selection and hyperparameter tuning effectively.

For parameter tuning, we imported the following libraries using code below:

```
[16] from sklearn.model_selection import GridSearchCV, KFold, cross_val_score
     from sklearn.linear_model import LinearRegression
     from sklearn.neighbors import KNeighborsRegressor
     from sklearn.tree import DecisionTreeRegressor
     from sklearn import neighbors, datasets
     # Standardize features by removing the mean and scaling to unit variance
     from sklearn.preprocessing import StandardScaler
     # Pipeline of transforms with a final estimator
     from sklearn.pipeline import Pipeline

     inner_cv = KFold(n_splits=5, shuffle=True, random_state=42) # inner cross-validation folds
     outer_cv = KFold(n_splits=5, shuffle=True, random_state=42) # outer cross-validation folds
```

[part a is worth 35 points in total:
c.1 10 points for correctly optimizing at least two parameters for linear regression model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results

1. The first parameter we chose is fit_intercept hyperparameter (True or False)

   We created a GridSearchCV to set up a grid search for hyperparameter tuning, the grid search includes the search for the fit_intercept hyperparameter with values(True, False). The scoring metric used for evaluation is the negative root mean squared error (neg_root_mean_squared_error), and the grid search is performed using inner cross-validation (inner_cv) on the training data (X_train and y_train).

   Then, we fit the grid search to the training data using gs_lr2.fit(X_train, y_train), the best cross-validated negative RMSE score= -121, the optimal hyperparameters is fit_intercept = False, and the best estimator is Linear Regression model, obtained from the grid search.

   Finally, we performed nested cross-validation using cross_val_score function to perform nested cross-validation on the test data (X_test and y_test). It calculates the negative RMSE scores for each fold of the outer cross-validation (outer_cv) using the best estimator obtained from the grid search. We got a mean of negative RMSE of -125.47.

```
[17]  #To ignore the convergence warnings
      from  warnings import simplefilter
      from sklearn.exceptions import ConvergenceWarning
      simplefilter("ignore", category=ConvergenceWarning)


      gs_lr2 = GridSearchCV(estimator=LinearRegression(),
                  param_grid=[{'fit_intercept': [ True, False]}],
                  scoring='neg_root_mean_squared_error',
                  cv=inner_cv)

      gs_lr2 = gs_lr2.fit(X_train,y_train)

      print("Non-nested CV score: ", gs_lr2.best_score_)
      print("Optimal Parameter: ", gs_lr2.best_params_)
      print("Optimal Estimator: ", gs_lr2.best_estimator_)

      nested_score_gs_lr2_f1 = cross_val_score(gs_lr2, X=X_test, y=y_test, cv=outer_cv, scoring='neg_root_mean_squared_error')
      print('Nested RMSE score:', nested_score_gs_lr2_f1.mean(), " +/- ", nested_score_gs_lr2_f1.std())
```

```
Non-nested CV score:  -121.26191569603311
Optimal Parameter:  {'fit_intercept': False}
Optimal Estimator:  LinearRegression(fit_intercept=False)
Nested RMSE score: -125.47071019752335  +/-  17.698532315607302
```

2. The second parameter we chose is regularization, and we used two different regularization methods which are lasso regression and ridge regression

   2.1 lasso regression (feature selection)

   We created a Lasso regression model (lasso), in the model, we set the alpha parameter=0.1 to control the strength of regularization. A larger alpha leads to stronger regularization, fit_intercept is set to True, which means that an intercept term will be included in the model. Next, we used StandardScaler to standardize the training and test data. Standardization helps in improving the performance of Lasso regression.

   Then, we fit the Lasso model to the training data and made predictions using lasso.predict(X_train) and lasso.predict(X_test) make predictions on the training and test data, respectively. Print(lasso.coef_) prints the estimated coefficients of the Lasso regression model. These coefficients indicate the importance of each feature in the model, and Lasso can set some coefficients to exactly zero, effectively performing feature selection.

   We got MSE, RMSE and MAE evaluation scores of lasso regression, in which the mean of RMSE is 120.504.

```
[18]  ########################## Decreasing Linear Regression Model Complexity ##########################


      ######################### Regularized Linear Regression Model - Lasso  ########################

      from sklearn.linear_model import Lasso # lasso Regression class
      from sklearn.preprocessing import StandardScaler # standardize features by removing the mean and scaling to unit variance
                                  # the standard score of a sample x is calculated as: z = (x - u) / s
                                  # where u is the mean of the training sample , and s is the standard deviation of the training sample.
                                  # centering and scaling happens independently on each feature.

      # Lasso Regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html#sklearn.linear_model.Lasso
      lasso = Lasso(alpha=0.1,fit_intercept=True)  # alpha : constant that multiplies the L1 term, controlling regularization strength
                        # the larger the value of alpha, the more aggressive the penalization is.
                        # alpha defaults to 1.0
                        # alpha = 0 is equivalent to an ordinary least square Linear Regression
      # Scale features
      scaler = StandardScaler()
      X_train = scaler.fit_transform(X_train)       # computes the mean and std to be used for scaling and performs scaling
      X_test = scaler.transform(X_test)           # performs standardization of test set X attributes by centering and scaling

      lasso.fit(X_train, y_train)               # fit model to data
      y_train_pred = lasso.predict(X_train)         # apply model to train data
      y_test_pred = lasso.predict(X_test)          # apply model to test data
      print(lasso.coef_)                     # estimated coefficients for the lasso regression model
```

```
      ################################  Lasso - Evaluation Metrics  ################################

      print('MSE train: %.3f, test: %.3f' % ( # MSE
            mean_squared_error(y_train, y_train_pred),
            mean_squared_error(y_test, y_test_pred))) # MSE Test Set
                                    # y_test: Ground truth (correct) target values
                                    # y_test_pred: Estimated target values

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

      print('MAE train: %.3f, test: %.3f' % ( # MAE
            mean_absolute_error(y_train, y_train_pred),
            mean_absolute_error(y_test, y_test_pred))) # MAE Test Set
                                     # y_test: Ground truth (correct) target values
                                     # y_test_pred: Estimated target values
```

```
[ -0.89484597  -0.44508092   3.03455413 -11.62378214  -3.47057071
  -9.57587983  -7.26702118  -4.79590267   2.08502779 -16.77364204
   3.95504655  -5.15652541  -3.68933399  -1.10482287  -5.04853221
  -2.28113288  -2.22456465 112.54161894 -18.78915195   9.45527528
  -0.70295307  -0.62673847 -27.08700364  42.72810721]
MSE train: 14521.188, test: 14828.720
RMSE train: 120.504, test: 121.773
MAE train: 70.008, test: 67.284
```

2.2 ridge regression

We created a Ridge regression model (ridge), in the model, we set the alpha parameter (regularization strength) to 1.0, fit_intercept is set to True, meaning an intercept term will be included in the model.

Then, we fit the Ridge model to the training data and made predictions using ridfe.predict(X_train) and ridge.predict(X_test) to make predictions on the training and test data, respectively. Print(ridge.coef_) prints the estimated coefficients of the Ridge regression model. Ridge does not set coefficients exactly to zero but shrinks them towards zero.

We got MSE, RMSE and MAE evaluation scores of lasso regression, in which the mean of RMSE is 120.499.

```
######################### Decreasing Linear Regression Model Complexity   ########################

######################### Regularized Linear Regression Model - Ridge      ########################

from math import sqrt
from sklearn.linear_model import Ridge # ridge Regression class

ridge = Ridge(alpha=1.0,fit_intercept=True)           # regularization strength; must be a positive float.
                          # larger values specify stronger regularization.
                          # alpha corresponds to C^-1 in other linear models such as LogisticRegression

ridge.fit(X_train, y_train)         # fit model (features have been scaled)
y_train_pred = ridge.predict(X_train)  # apply model to train data
y_test_pred = ridge.predict(X_test)    # apply model to test data
print(ridge.coef_)                  # estimated coefficients for the ridge regression model

print('MSE train: %.3f, test: %.3f' % ( #MSE
      mean_squared_error(y_train, y_train_pred),
      mean_squared_error(y_test, y_test_pred))) # MSE Test Set
                                  # y_test: Ground truth (correct) target values
                                  # y_test_pred: Estimated target values

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

print('MAE train: %.3f, test: %.3f' % ( #MAE
      mean_absolute_error(y_train, y_train_pred),
      mean_absolute_error(y_test, y_test_pred))) # MAE Test Set
                                   # y_test: Ground truth (correct) target values
                                   # y_test_pred: Estimated target values
```

```
[ -0.98340954  -0.56250538   1.7757125  -12.60968689  -4.51491946
 -10.45141941  -8.80172559  -5.36976827   1.47052529 -17.62914966
   3.04971962  -6.05940314  -4.35589191  -2.50824197  -5.47613168
  -2.90198042  -3.64922211 112.29873024 -19.50635308  10.08473075
  -0.82576252  -0.73720623 -27.11536425  42.95853892]
MSE train: 14520.084, test: 14830.320
RMSE train: 120.499, test: 121.780
MAE train: 69.925, test: 67.259
```

c.2 10 points for correctly optimizing at least two parameters for linear k-NN model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results

For kNN model's parameter tuning, we chose

1. 'knn__n_neighbors': The number of nearest neighbors to consider (values from 1 to 21)
2. 'knn__weights': The weight function used in prediction ('uniform' or 'distance')

We first defined a data preprocessing pipeline, StandardScaler ('sc') standardizes the input features by subtracting the mean and scaling to unit variance, the second step KNeighborsRegressor ('knn') represents the KNN regression model, which will then use the standardized data.

Then, we defined hyperparameters to optimize by specifying a grid of hyperparameters to search over using a dictionary named params.

Next, we created GridSearchCV: It sets up a grid search for hyperparameter tuning. The grid search is performed on the pipeline defined earlier. It searches for the combination of hyperparameters that minimizes the negative root mean squared error (RMSE) as the scoring metric. The grid search is performed using inner cross-validation (inner_cv) on the training data.

After setting up the grid search, we fit it to the training data, and print the results of the grid search, including the best cross-validated RMSE score, the optimal hyperparameters (knn__n_neighbors and knn__weights), and the best estimator (KNN regressor with data preprocessing).

Finally, we performed nested cross-validation. We used the cross_val_score function to perform nested cross-validation on the test data. The function calculates the negative RMSE scores for each fold of the outer cross-validation (outer_cv) using the best estimator obtained from the grid search. We got a mean of -145 and standard deviation of 20 of the negative RMSE scores obtained from the nested cross-validation.

```
#Normalize Data
pipe = Pipeline([
      ('sc', StandardScaler()),
      ('knn', KNeighborsRegressor(metric='minkowski'))
     ])

#Parameters to optimize:  k for number of nearest neighbors AND type of distance

params = {
      'knn__n_neighbors': [1,3,5,7,9,11,13,15,17,19,21],
      'knn__weights': ['uniform', 'distance']
    }

gs_knn2 = GridSearchCV(estimator=pipe,
              param_grid=params,
              scoring='neg_root_mean_squared_error',
              cv=inner_cv,
              n_jobs=1)

gs_knn2 = gs_knn2.fit(X_train,y_train)
print("Non-nested CV score: ", gs_knn2.best_score_)
print("Optimal Parameter: ", gs_knn2.best_params_)
print("Optimal Estimator: ", gs_knn2.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest score

print('Nested RMSE score:', cross_val_score(gs_knn2, X=X_test, y=y_test, cv=outer_cv, scoring='neg_root_mean_squared_error').mean(), " +/- ", cross_val_score(gs_knn2, X=X_test, y=y_test, cv=outer_cv, scoring='neg_root_mean_s
```

```
Non-nested CV score:  -129.1628499665049
Optimal Parameter:  {'knn__n_neighbors': 9, 'knn__weights': 'distance'}
Optimal Estimator:  Pipeline(steps=[('sc', StandardScaler()),
                ('knn',
                 KNeighborsRegressor(n_neighbors=9, weights='distance'))])
Nested RMSE score: -145.4827040999749  +/-  20.315238251509694
```

10 points for correctly optimizing at least two parameters for linear regression tree model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results

For linear regression tree model's parameter tuning, we chose:

1. 'max_depth': The maximum depth of the decision tree, ranging from 2 to 9
2. 'criterion': The criterion used for splitting ('squared_error' or 'absolute_error')

First, we set up a grid search for hyperparameter tuning using the Decision Tree Regressor as the estimator and the specified hyperparameter grid. The grid search aims to minimize the negative root mean squared error (RMSE) as the scoring metric. The grid search is performed using inner cross-validation (inner_cv) on the entire dataset.

From the grid search, we got the best cross-validated RMSE score, the optimal hyperparameters (max_depth=5, and criterion='squared error'), and the best estimator (Decision Tree Regressor).

Then, we perform nested cross-validation using the cross_val_score function, it calculates the negative RMSE scores for each fold of the outer cross-validation (outer_cv) using the best estimator obtained from the grid search.

Eventually, we got a mean of -126 and standard deviation of 11 of the negative RMSE scores.

```
[21]
    # Nested cross-validation
    # Choosing optimal depth of the tree AND optimal splitting criterion
    gs_dt = GridSearchCV(estimator=DecisionTreeRegressor(random_state=42),
                param_grid=[{'max_depth': range(2,10),
                        'criterion':['squared_error','absolute_error']}],
                scoring='neg_root_mean_squared_error',
                cv=inner_cv
               )

    gs_dt = gs_dt.fit(X,y)
    print("Non-nested CV score: ", gs_dt.best_score_)
    print("Optimal Parameter: ", gs_dt.best_params_)
    print("Optimal Estimator: ", gs_dt.best_estimator_)

    nested_score_gs_dt_f1 = cross_val_score(gs_dt, X=X, y=y, cv=outer_cv, scoring='neg_root_mean_squared_error')
    print('Nested RMSE score:', nested_score_gs_dt_f1.mean(), " +/- ", nested_score_gs_dt_f1.std())
```

```
Non-nested CV score:  -123.78662976123917
Optimal Parameter:  {'criterion': 'squared_error', 'max_depth': 5}
Optimal Estimator:  DecisionTreeRegressor(max_depth=5, random_state=42)
Nested RMSE score: -126.30443088874179  +/-  11.802322810029178
```

5 points for discussing which of the three models yields the best performance]

Comparing the RMSE of the three models, the ridge regression(linear regression) performed the best, with a RMSE=120.499.

The characteristics of the dataset:

Size:

We have 2000 observations in this dataset, which is relatively big, so Ridge Regression and Decision Trees are likely to fit better compared to KNN, which generally out performs in smaller datasets.

Feature:

1)Binary Features (e.g., "US," "Web_order," "Gender=mal," "Address_is_res," "Purchase"):

This dataset contains several binary features that indicate yes or no conditions. Ridge Regression can handle binary features effectively as it treats them as numeric variables. Decision Trees can also work well with binary features, but they may overfit if there are many features and few samples. KNN relies on distance metrics and may not be sensitive to the binary nature of these features.

2)Numeric Features (e.g., "Freq.," "last_update_days_ago," "1st_update_days_ago," "Spending"):

Ridge Regression is well-suited for numeric features, as it can capture linear relationships effectively. It is also robust to outliers, which is beneficial if "Spending" has extreme values. Decision Trees can handle numeric features, but they might require careful tuning to avoid overfitting, especially if there are many features. KNN can work with numeric features, but its performance can be sensitive to the dimensionality and choice of the k value.

The characteristics of each algorithm:

Linearity of the Data:

Ridge Regression: Ridge Regression is a linear model, and it assumes a linear relationship between the input features and the target variable. If the dataset exhibits a predominantly linear relationship between the features and the target, Ridge Regression is well-suited to capture this relationship effectively.

Decision Tree Regression: Decision Trees are non-linear models that can capture complex relationships within the data. However, if the data primarily follows a linear pattern, Decision Trees may struggle to capture it as efficiently as Ridge Regression.

KNN: KNN is a non-parametric, instance-based method that doesn't make strong assumptions about the data's linearity. However, it can perform poorly in high-dimensional spaces or when the data doesn't have clear clusters, which can be the case if the dataset is primarily linear.

Overfitting and Regularization:

Ridge Regression: Ridge Regression includes L2 regularization, which helps prevent overfitting by penalizing large coefficients. This can be beneficial when dealing with noisy or high-dimensional data.

Decision Tree Regression: Decision Trees are prone to overfitting, especially if they are deep and not pruned. They can fit the training data very closely, which may lead to poor generalization to unseen data.

KNN: KNN is also prone to overfitting, especially if the value of k is too low. A small k value can make KNN sensitive to noise in the training data.