EMORY UNIVERSITY

Introduction to Business Data Analytics

Homework #4B

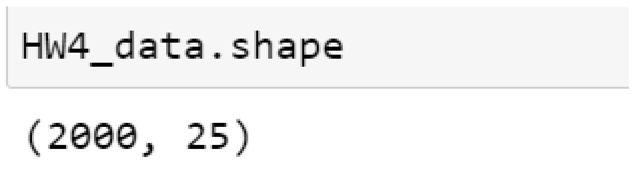
Team 9

Xiao Han, Jent Lapalm, Aoran Wang, Yushan Yang

(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>

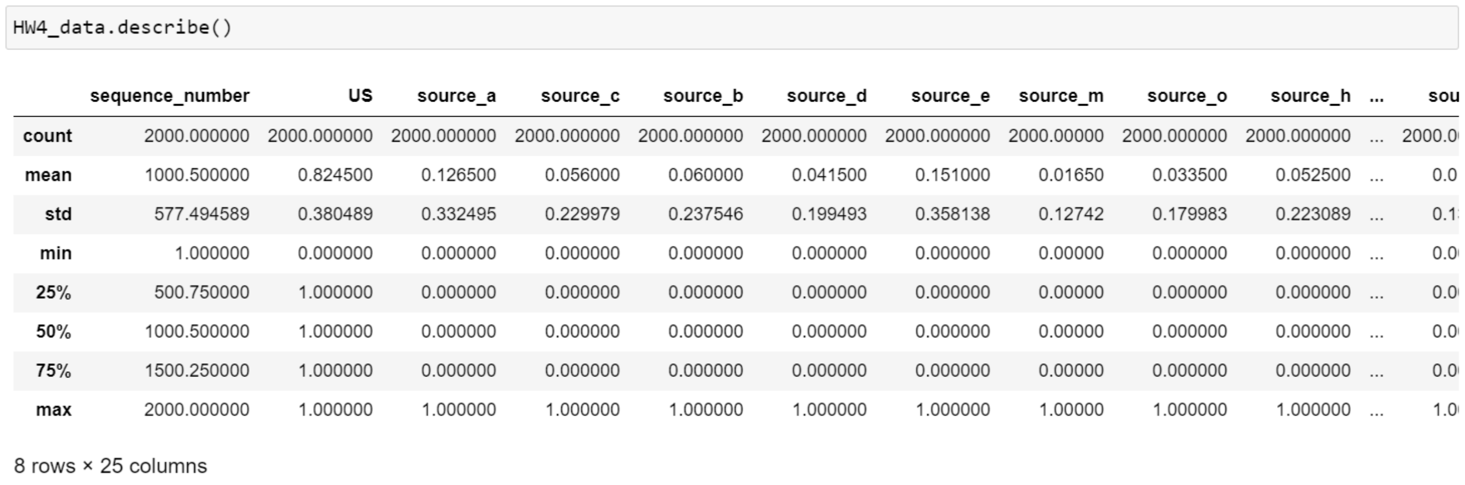
(a) (50 points) After exploring the data, build numeric prediction models that predict Spending. Use linear regression, k-NN, and regression tree techniques. Briefly discuss the models you have built. Use cross-validation with 10 folds to estimate the generalization performance. Present the results for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

[part a is worth 50 points in total:
15 points for exploring the data (i.e., descriptive statistics including min max mean and stdv, visualizations, target variable distribution)
10 points for correctly building linear regression model - provide screenshots and explain what you are doing and the corresponding results
10 points for correctly building k-NN model - provide screenshots and explain what you are doing and the corresponding results
10 points for correctly building regression tree model - 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]



```
HW4_data.shape
```

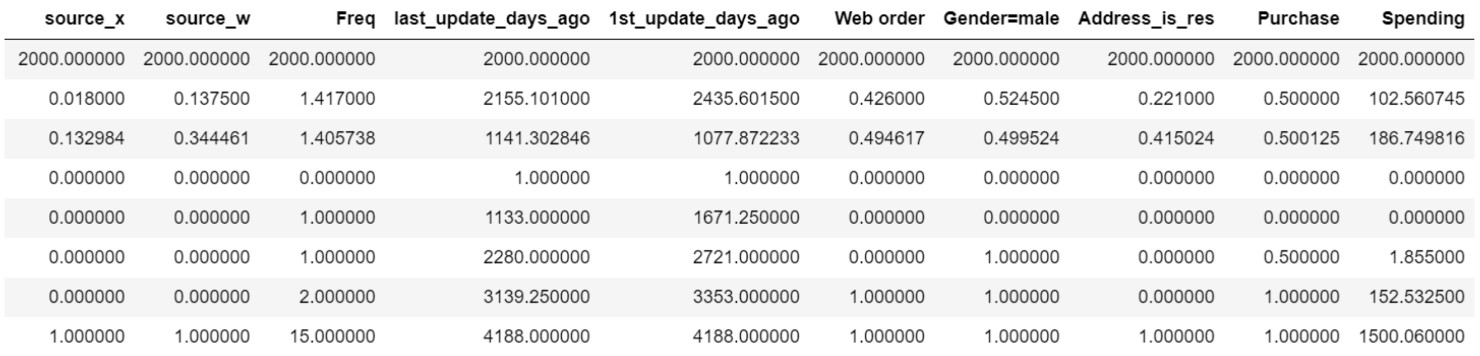
```
(2000, 25)
```



```
HW4_data.describe()
```

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...	sou
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	...	2000.0
mean	1000.500000	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	...	0.0
std	577.494589	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	...	0.1
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0
25%	500.750000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0
50%	1000.500000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0
75%	1500.250000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0
max	2000.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	...	1.0

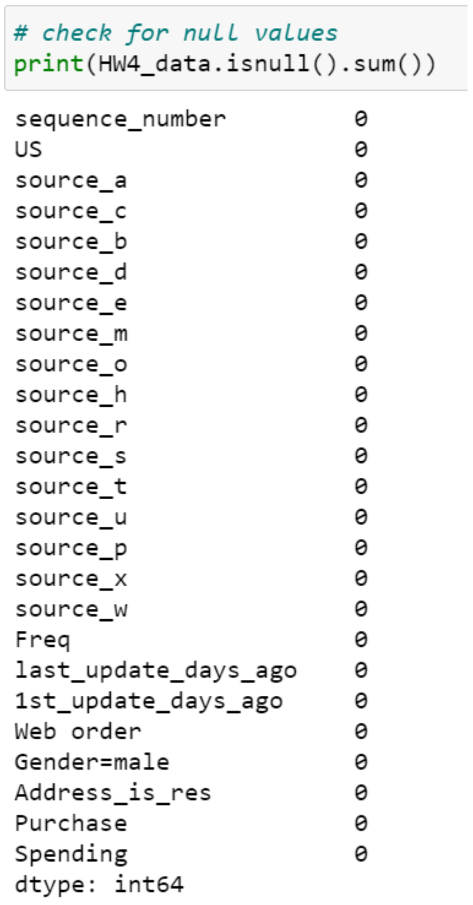
8 rows × 25 columns



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

We first checked the data shape, and we found out that the data has 2000 rows and 25 columns. Later on we checked the descriptive statistic for the dataset. As the graph indicates, most attributes are binary, meaning that they only contain 0 and 1. Based on the Min, Max, and Std, we found out that the target variable, “Spending” has a wide range of instances, and relatively small standard deviation. The discrepancy between Mean and max is also significant, which implies that the distribution of the target variable is highly right skewed, and so does the variable “Frequency”.

We also checked whether there are any null values in the dataset and fortunately there is none.



```
# check for null values
print(HW4_data.isnull().sum())
```

```
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
```

```
HW4_data.head()
```

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...	source_x	source_w	Freq	last_update_
0	1	1	0	0	1	0	0	0	0	0	...	0	0	2	
1	2	1	0	0	0	0	1	0	0	0	...	0	0	0	
2	3	1	0	0	0	0	0	0	0	0	...	0	0	2	
3	4	1	0	1	0	0	0	0	0	0	...	0	0	1	
4	5	1	0	1	0	0	0	0	0	0	...	0	0	1	

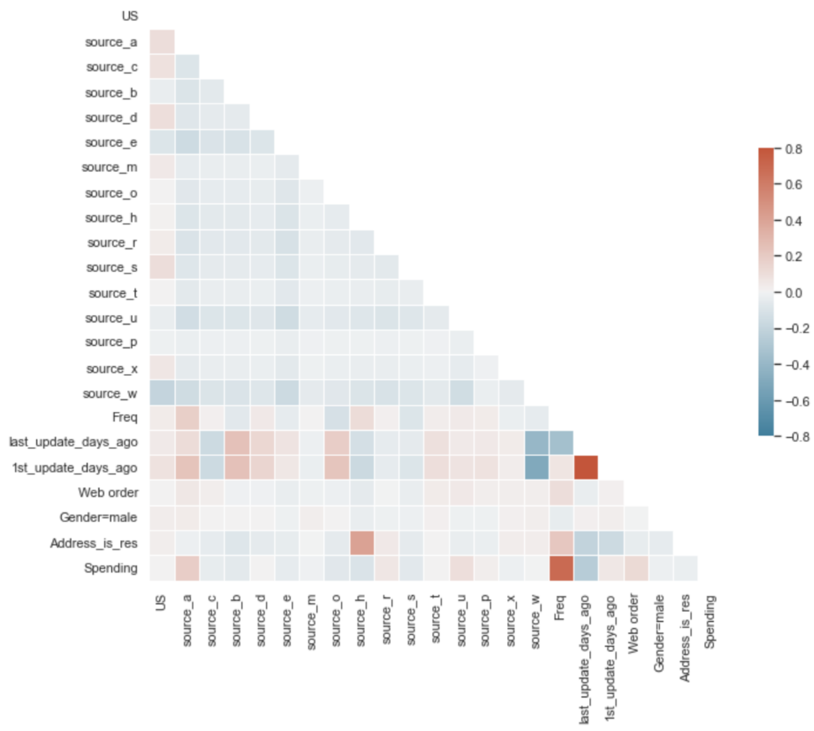
5 rows × 25 columns

```python
del X['Purchase']
del HW4_data['Purchase']
del HW4_data['sequence_number']
```

To proceed with the data exploration, we displayed the first five rows of the dataset, and found out that the sequence numbers are almost identical with the dataset index, so we decided to drop it from our dataset. What is more, the attribute "purchase" is not the information that will be available at the time of the prediction, the we decided to remove it to avoid data leakage.

```python
sns.set_theme(style="white")
corr = HW4_data.iloc[:, 0:].corr()
mask = np.triu(np.ones_like(corr))
f, ax = plt.subplots(figsize=(11, 9))
cmap = sns.diverging_palette(230, 20, as_cmap=True)
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.8, vmin=-.8,center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

```
<AxesSubplot:>
```



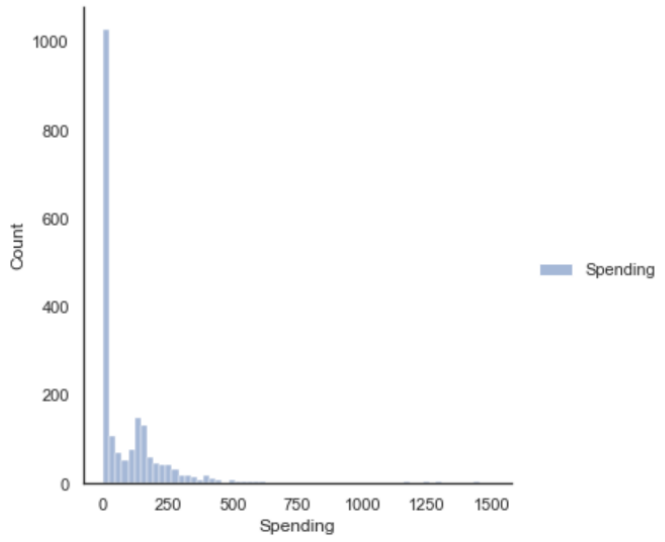
```python
del X['1st_update_days_ago']
del HW4_data['1st_update_days_ago']
```

We also checked the correlation matrix of the target variable and each independent variable, according to the correlation graph, variable "1st_update_days_ago" and "last_update_days_ago" are highly correlated. represent

“How many days ago was the 1st update to cust. record” and “How many days ago was the last update to cust. record”. We think that the latter one should be a better indicator for predicting customer spending so we decided to remove the former one to avoid multicollinearity problems.

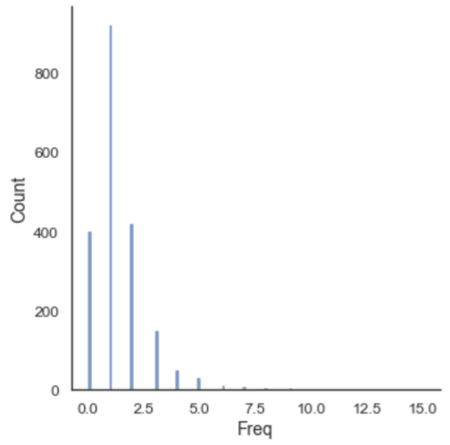
```
y = HW4_data.iloc[:,-1:]
X = HW4_data.iloc[:, 1:-1]
del X['Purchase']
del HW4_data['Purchase']
del HW4_data['sequence_number']
plt.figure(3,figsize=(8, 6))
sns.displot(y)
plt.xlabel("Spending")
print("Spending", "statistical results:", sp.stats.describe(y))
```

```
Spending statistical results: DescribeResult(nobs=2000, minmax=(array([0.]), array([1500.06])), mean=array([102.560745]), va
riance=array([34875.49372826]), skewness=array([3.92549439]), kurtosis=array([20.53912844]))

<Figure size 576x432 with 0 Axes>
```



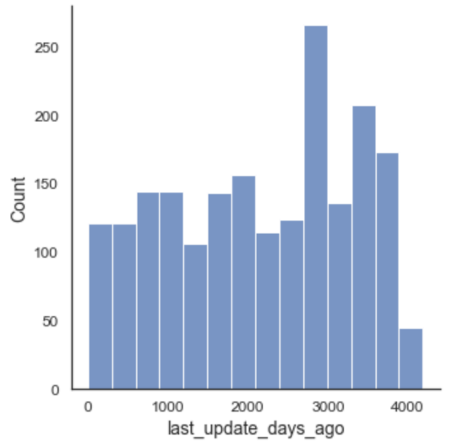
```
plt.figure(3,figsize=(8, 6))
sns.displot(X.loc[:,'Freq'])
plt.xlabel("Freq")
print("Freq", "statistical results:", sp.stats.describe(X.loc[:,'Freq']))
```

```
Freq statistical results: DescribeResult(nobs=2000, minmax=(0, 15), mean=1.417, variance=1.9760990495247626, skewness=2.9788
31957861886, kurtosis=15.997895658160584)

<Figure size 576x432 with 0 Axes>
```



```
plt.figure(3,figsize=(8, 6))
sns.displot(X.loc[:,'last_update_days_ago'])
plt.xlabel("last_update_days_ago")
print("last_update_days_ago", "statistical results:", sp.stats.describe(X.loc[:,'last_update_days_ago']))
```

```
last_update_days_ago statistical results: DescribeResult(nobs=2000, minmax=(1, 4188), mean=2155.101, variance=1302572.186892
4464, skewness=-0.1877302916019205, kurtosis=-1.191994205413851)

<Figure size 576x432 with 0 Axes>
```

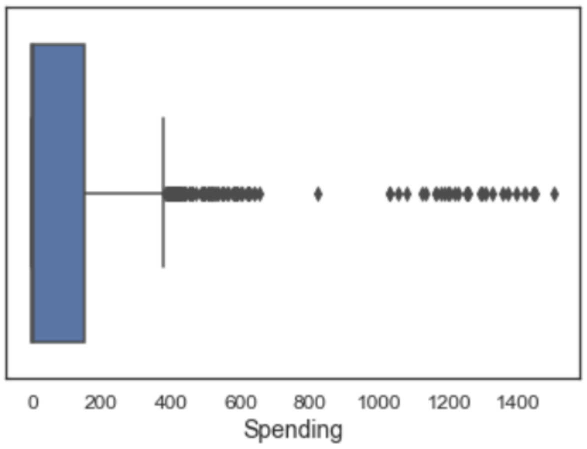


Moving on, we checked the distribution of the target variable. As the graph and statistical results indicate, the target variable is highly right skewed. We also checked the distribution of other numeric variables as well. As a result, variable “frequency” is highly right skewed and “last_update_days_ago” seems fine.

```
In [37]: del X['1st_update_days_ago']
         del HW4_data['1st_update_days_ago']
```

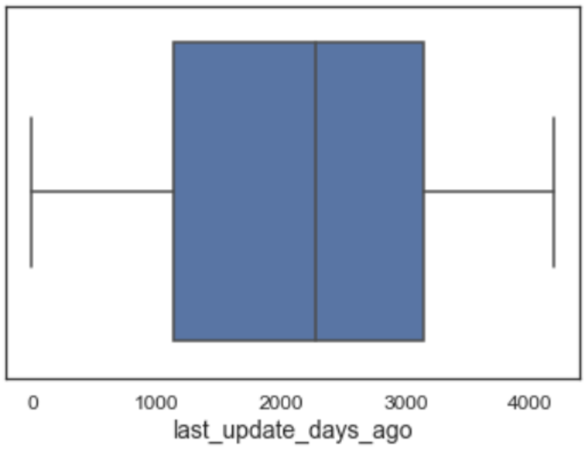
```
In [61]: sns.boxplot(x=HW4_data.loc[:,'Spending'])
```

```
Out[61]: <AxesSubplot:xlabel='Spending'>
```



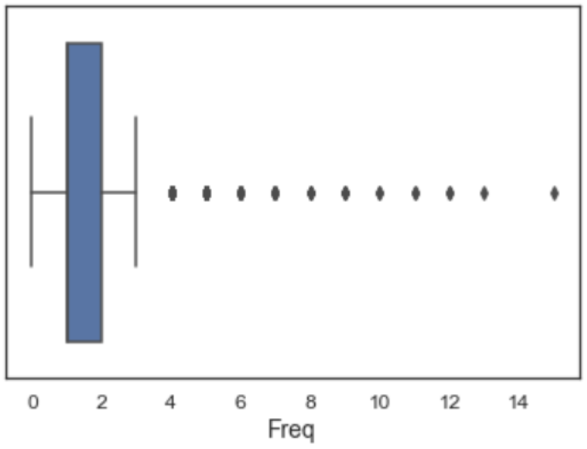
```
In [65]: sns.boxplot(x=HW4_data.loc[:,'last_update_days_ago'])
```

```
Out[65]: <AxesSubplot:xlabel='last_update_days_ago'>
```

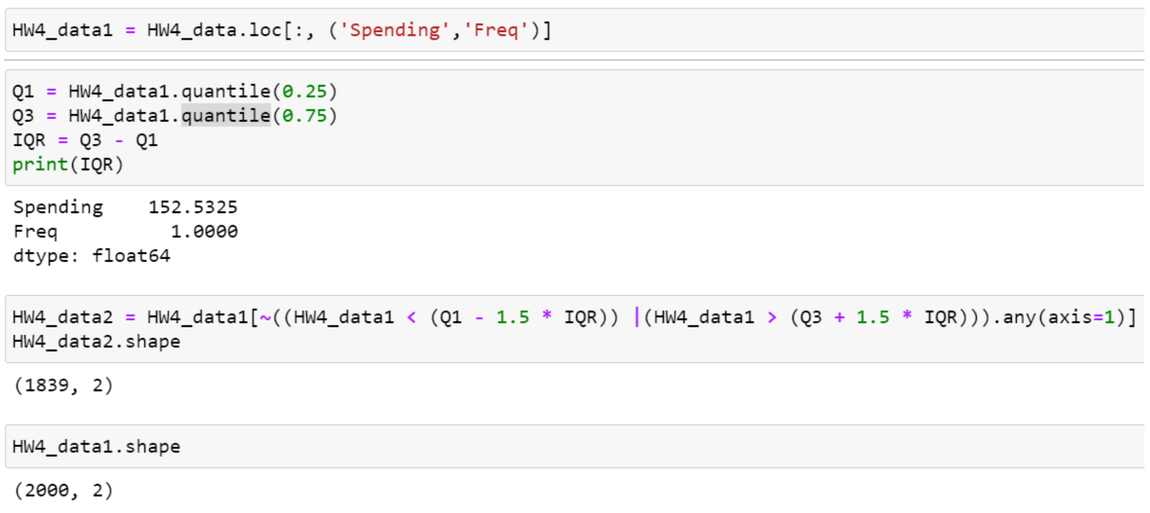


```
In [66]: sns.boxplot(x=HW4_data.loc[:,'Freq'])
```

```
Out[66]: <AxesSubplot:xlabel='Freq'>
```



Considering two of the tree numeric variables are not normally distributed, we decided to check for outliers before doing feature engineering. We first checked the box plot of three numeric variables, and found out that “spending” and “frequency” have a number of outliers exceeding the 75% quantile.



```
HW4_data1 = HW4_data.loc[:, ('Spending','Freq')]
```

```
Q1 = HW4_data1.quantile(0.25)
Q3 = HW4_data1.quantile(0.75)
IQR = Q3 - Q1
print(IQR)
```

```
Spending    152.5325
Freq          1.0000
dtype: float64
```

```
HW4_data2 = HW4_data1[~((HW4_data1 < (Q1 - 1.5 * IQR)) |(HW4_data1 > (Q3 + 1.5 * IQR))).any(axis=1)]
HW4_data2.shape
```

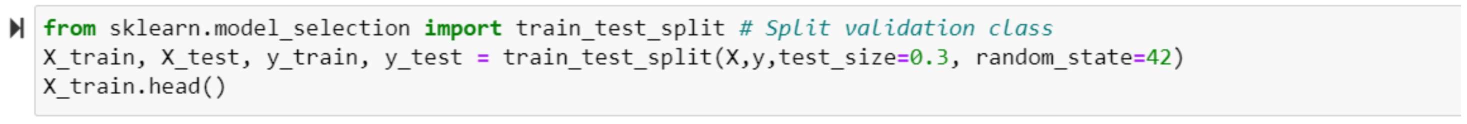
```
(1839, 2)
```

```
HW4_data1.shape
```

```
(2000, 2)
```

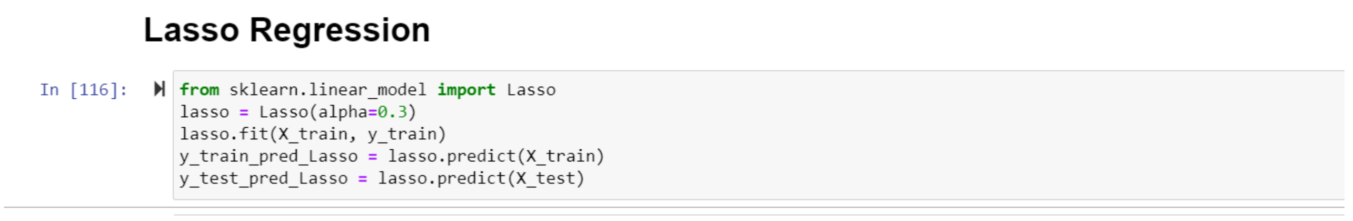
Then we checked the IQR score (IQR = Q3 − Q1) and experimented removing all data points beyond 1.5 times of IQR score. Through the comparison between the shapes before and after, we found out that we lost 161 rows by removing the outliers, which is significant considering we only have 2000 rows of data. And since they are all continuous variables, we decided not to remove the outliers and instead do the feature engineering later.

After data cleaning, we built four basic models and evaluated them using RMSE and MAE. We wanted to estimate the standard deviation of an observed value models’ prediction and the average of absolute difference between the predicted values and observed value. Since RMSE penalized outliers more and we chose not to remove outliers(mentioned above),we would take RMSE as main evaluation and MAE as a supportive evaluation, Besides, we also applied cross-validation with 10 folds to estimate the generalization performance.
First, we split the train and test data as 3:7, which is a common split and set the random state as 42.



```
from sklearn.model_selection import train_test_split # Split validation class
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3, random_state=42)
X_train.head()
```

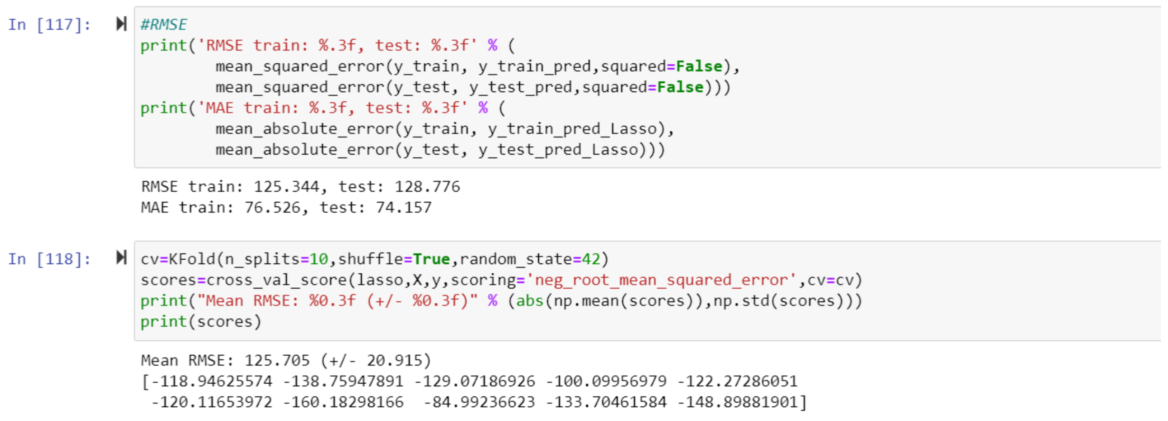
The first model was Lasso Regression. It was used over regression methods for a more accurate prediction.We built a very basic model by setting alpha=0.3, and we would do grid search in part c.



Lasso Regression

```
In [116]:  from sklearn.linear_model import Lasso
           lasso = Lasso(alpha=0.3)
           lasso.fit(X_train, y_train)
           y_train_pred_Lasso = lasso.predict(X_train)
           y_test_pred_Lasso = lasso.predict(X_test)
```

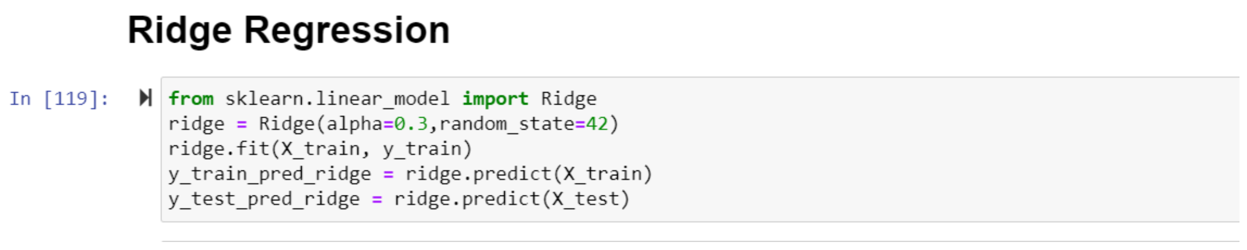
The test RMSE and test MAE for this model were 125.344 and 128.776. The cross validation with 10 folds gave us a mean RMSE 125.705.



```
In [117]:  #RMSE
            print('RMSE train: %.3f, test: %.3f' % (
                    mean_squared_error(y_train, y_train_pred,squared=False),
                    mean_squared_error(y_test, y_test_pred,squared=False)))
            print('MAE train: %.3f, test: %.3f' % (
                    mean_absolute_error(y_train, y_train_pred_Lasso),
                    mean_absolute_error(y_test, y_test_pred_Lasso)))

            RMSE train: 125.344, test: 128.776
            MAE train: 76.526, test: 74.157
```

```
In [118]:  cv=KFold(n_splits=10,shuffle=True,random_state=42)
            scores=cross_val_score(lasso,X,y,scoring='neg_root_mean_squared_error',cv=cv)
            print("Mean RMSE: %0.3f (+/- %0.3f)" % (abs(np.mean(scores)),np.std(scores)))
            print(scores)

            Mean RMSE: 125.705 (+/- 20.915)
            [-118.94625574 -138.75947891 -129.07186926 -100.09956979 -122.27286051
             -120.11653972 -160.18298166  -84.99236623 -133.70461584 -148.89881901]
```

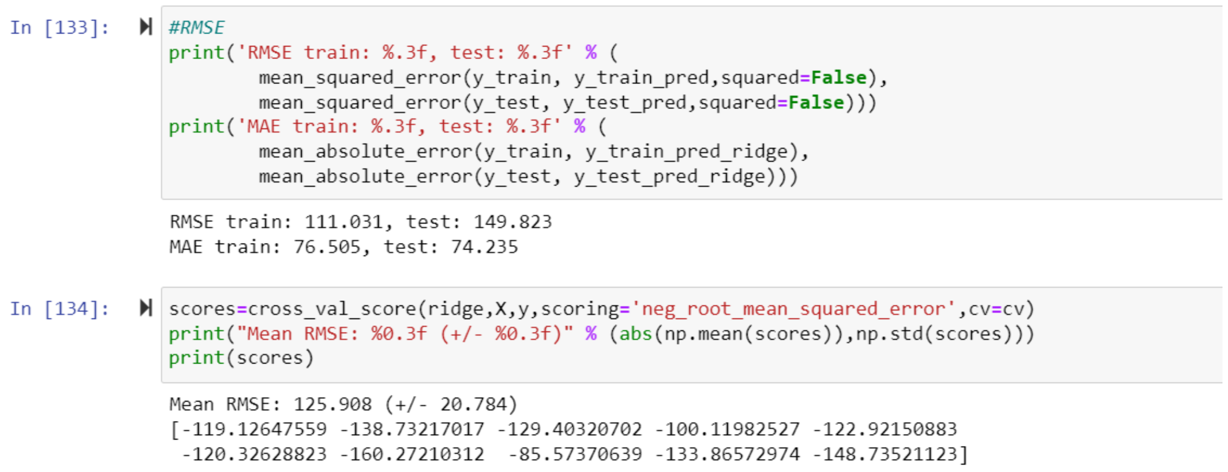
The second model was Ridge Regression. We built a very basic model by setting alpha=0.3, and we would do grid search in part c.



Ridge Regression

```
In [119]:  from sklearn.linear_model import Ridge
            ridge = Ridge(alpha=0.3,random_state=42)
            ridge.fit(X_train, y_train)
            y_train_pred_ridge = ridge.predict(X_train)
            y_test_pred_ridge = ridge.predict(X_test)
```

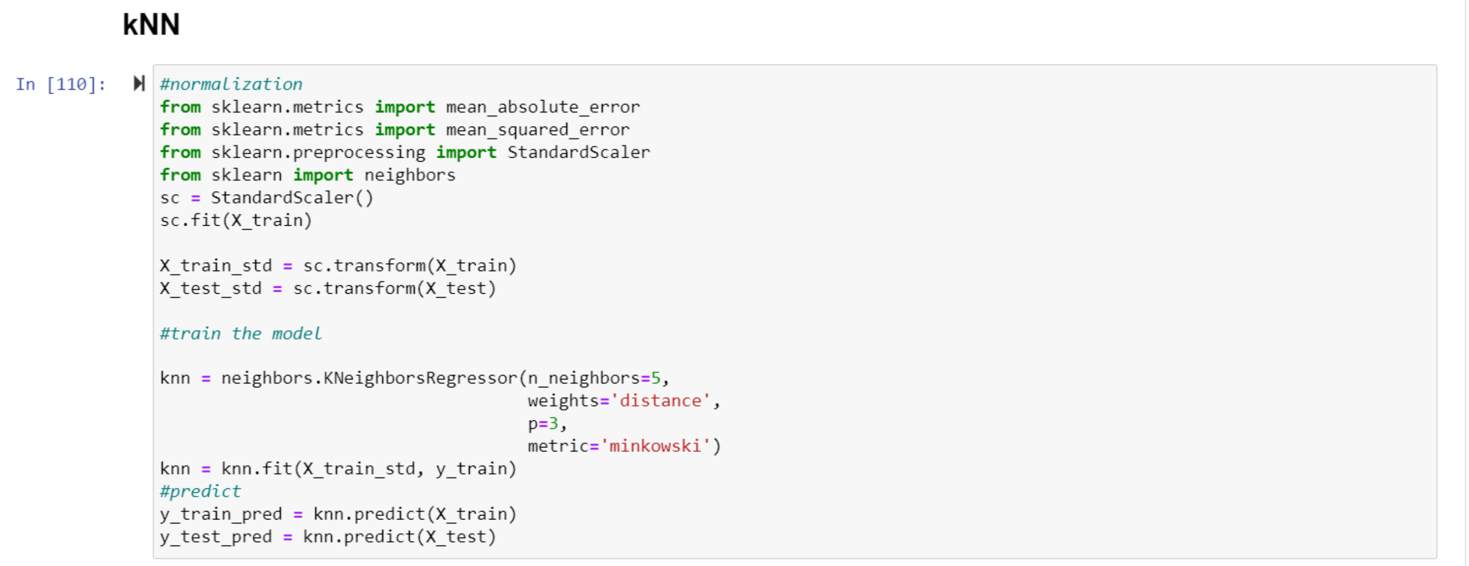
The test RMSE and test MAE for this model were 149.823 and 74.235. The cross validation with 10 folds gave us a mean RMSE 125.908.



```
In [133]:  #RMSE
            print('RMSE train: %.3f, test: %.3f' % (
                    mean_squared_error(y_train, y_train_pred,squared=False),
                    mean_squared_error(y_test, y_test_pred,squared=False)))
            print('MAE train: %.3f, test: %.3f' % (
                    mean_absolute_error(y_train, y_train_pred_ridge),
                    mean_absolute_error(y_test, y_test_pred_ridge)))

            RMSE train: 111.031, test: 149.823
            MAE train: 76.505, test: 74.235
```

```
In [134]:  scores=cross_val_score(ridge,X,y,scoring='neg_root_mean_squared_error',cv=cv)
            print("Mean RMSE: %0.3f (+/- %0.3f)" % (abs(np.mean(scores)),np.std(scores)))
            print(scores)

            Mean RMSE: 125.908 (+/- 20.784)
            [-119.12647559 -138.73217017 -129.40320702 -100.11982527 -122.92150883
             -120.32628823 -160.27210312  -85.57370639 -133.86572974 -148.73521123]
```

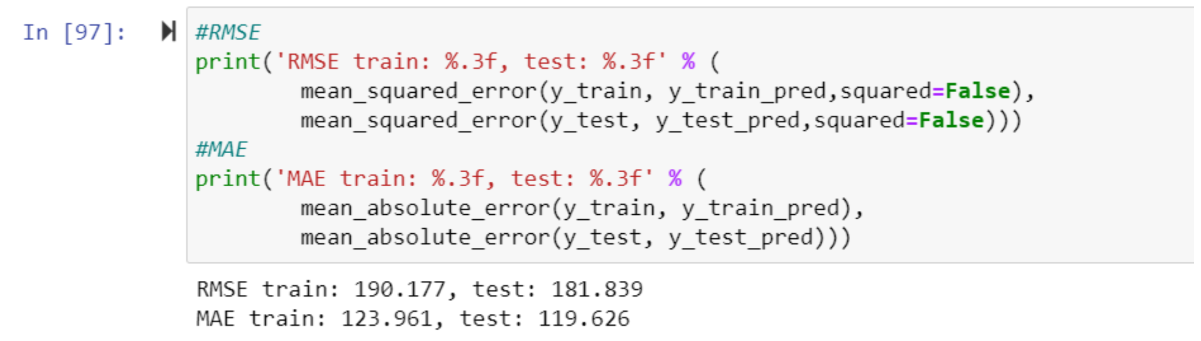
The third model was kNN Regression. It was used over regression methods for a more accurate prediction.We built a very basic model by setting neighbours=5,weights=distance,p=3 and metric ='minkowski'.



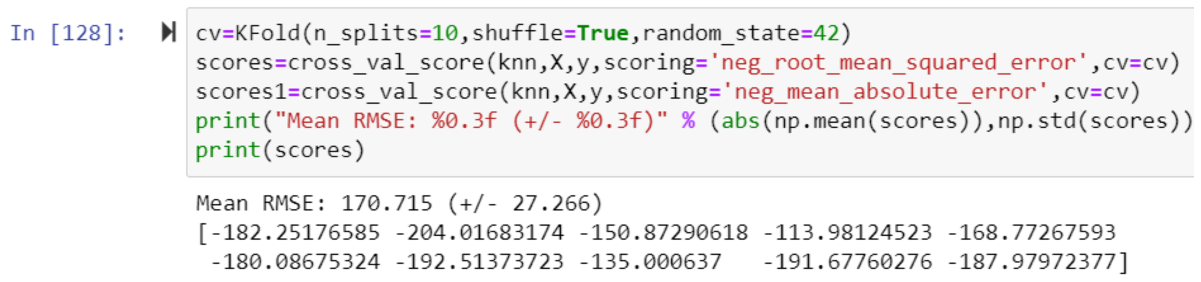
kNN

```
In [110]:  #normalization
           from sklearn.metrics import mean_absolute_error
           from sklearn.metrics import mean_squared_error
           from sklearn.preprocessing import StandardScaler
           from sklearn import neighbors
           sc = StandardScaler()
           sc.fit(X_train)

           X_train_std = sc.transform(X_train)
           X_test_std = sc.transform(X_test)

           #train the model

           knn = neighbors.KNeighborsRegressor(n_neighbors=5,
                                               weights='distance',
                                               p=3,
                                               metric='minkowski')
           knn = knn.fit(X_train_std, y_train)
           #predict
           y_train_pred = knn.predict(X_train)
           y_test_pred = knn.predict(X_test)
```

The test RMSE and test MAE for this model were 181.839 and 119.626. The cross validation with 10 folds gave us a mean RMSE 170.715.

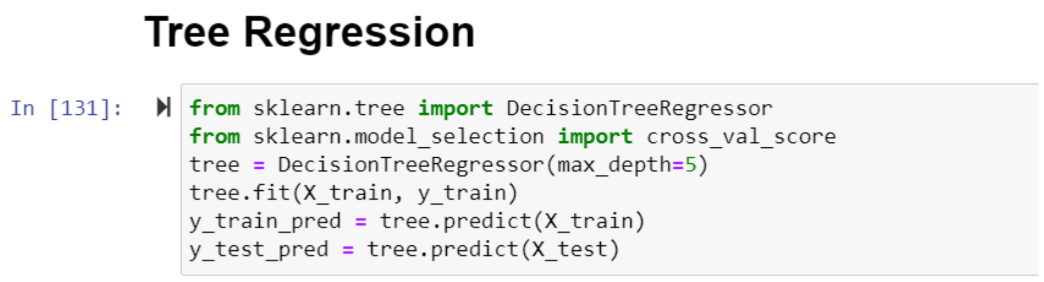


```
In [97]:  #RMSE
          print('RMSE train: %.3f, test: %.3f' % (
                  mean_squared_error(y_train, y_train_pred,squared=False),
                  mean_squared_error(y_test, y_test_pred,squared=False)))
          #MAE
          print('MAE train: %.3f, test: %.3f' % (
                  mean_absolute_error(y_train, y_train_pred),
                  mean_absolute_error(y_test, y_test_pred)))

          RMSE train: 190.177, test: 181.839
          MAE train: 123.961, test: 119.626
```



```
In [128]:  cv=KFold(n_splits=10,shuffle=True,random_state=42)
           scores=cross_val_score(knn,X,y,scoring='neg_root_mean_squared_error',cv=cv)
           scores1=cross_val_score(knn,X,y,scoring='neg_mean_absolute_error',cv=cv)
           print("Mean RMSE: %0.3f (+/- %0.3f)" % (abs(np.mean(scores)),np.std(scores))
           print(scores)

           Mean RMSE: 170.715 (+/- 27.266)
           [-182.25176585 -204.01683174 -150.87290618 -113.98124523 -168.77267593
            -180.08675324 -192.51373723 -135.000637   -191.67760276 -187.97972377]
```

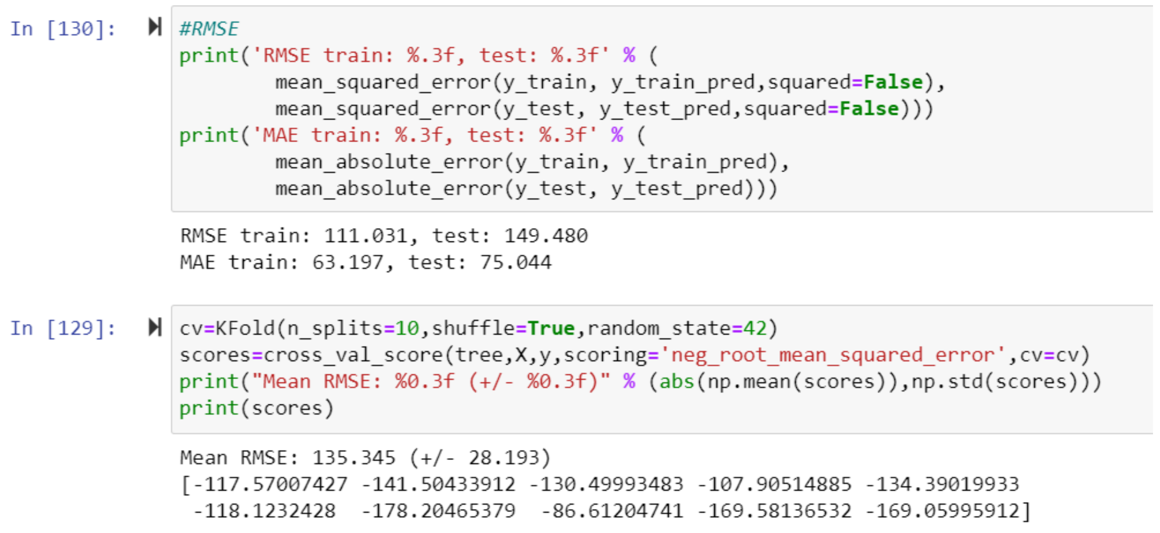
The fourth model was Tree Regression. We built a very basic model by setting max_depth=5, and we would do grid search in part c.



Tree Regression

In [131]:
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
tree = DecisionTreeRegressor(max_depth=5)
tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)
```

The test RMSE and test MAE for this model were 149.480 and 75.044. The cross validation with 10 folds gave us a mean RMSE 135.345.



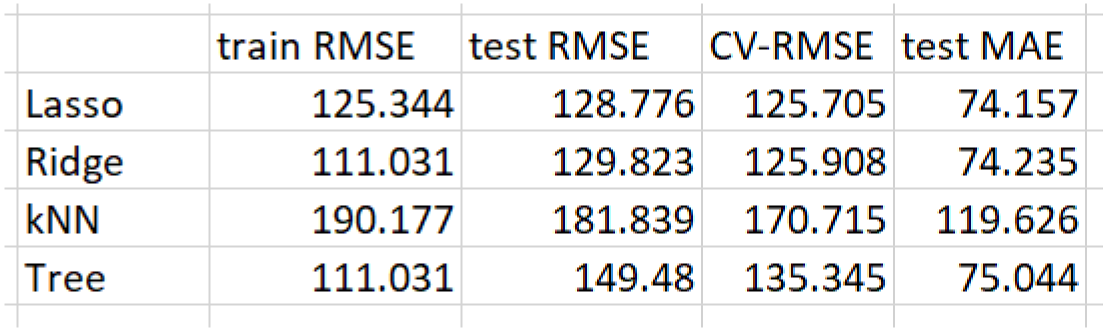
In [130]:
```
#RMSE
print('RMSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred,squared=False),
        mean_squared_error(y_test, y_test_pred,squared=False)))
print('MAE train: %.3f, test: %.3f' % (
        mean_absolute_error(y_train, y_train_pred),
        mean_absolute_error(y_test, y_test_pred)))
```

```
RMSE train: 111.031, test: 149.480
MAE train: 63.197, test: 75.044
```

In [129]:
```
cv=KFold(n_splits=10,shuffle=True,random_state=42)
scores=cross_val_score(tree,X,y,scoring='neg_root_mean_squared_error',cv=cv)
print("Mean RMSE: %0.3f (+/- %0.3f)" % (abs(np.mean(scores)),np.std(scores)))
print(scores)
```

```
Mean RMSE: 135.345 (+/- 28.193)
[-117.57007427 -141.50433912 -130.49993483 -107.90514885 -134.39019933
 -118.1232428  -178.20465379  -86.61204741 -169.58136532 -169.05995912]
```

When choosing models, we would like to choose the one with a smaller test RMSE and mean RMSE after cross validation and the difference between tain RMSE and test RMSE,which possibly indicates overfitting if the difference is too large . Test MAE is also considered but not an important role.
According to the results of this step, we decided to choose Lasso regression since it had lowest test RMSE and lowest mean RMSE after applying cross validation.Ridge regression had similar test RMSE,mean RMSE after cross validation and test MAE. However, the difference between train RMSE and test RMSE was large for Ridge regression which indicates fitting.



	train RMSE	test RMSE	CV-RMSE	test MAE
Lasso	125.344	128.776	125.705	74.157
Ridge	111.031	129.823	125.908	74.235
kNN	190.177	181.839	170.715	119.626
Tree	111.031	149.48	135.345	75.044

(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]

Part B: Feature Engineering

Feature engineering is used to improve model performance, for example, by transforming or combining features. This is also a good point in the model-building process to address issues like data leakage.

New Feature and Dropping Columns:

The *1st update* and *last update* columns were highly correlated. We decided that their difference, *account age*, was more informative from a business perspective, so we replaced the two update columns with *account age*.

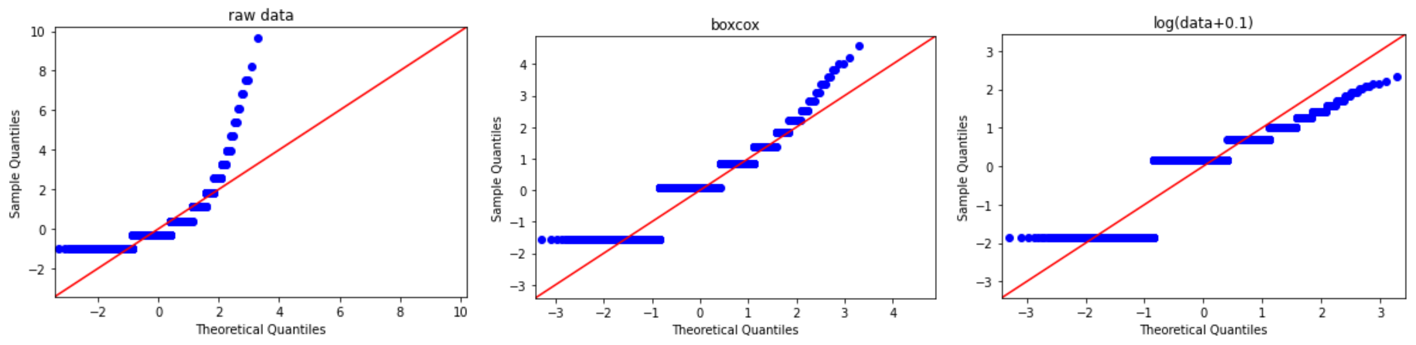
We also dropped the *Purchases* column as it created a data leakage problem. Customers who made a purchase spent money, and our aim was to predict how much a customer would spend.

We feel confident in these choices: as each of these columns were dropped, we re-ran our models, and test RMSE improved.

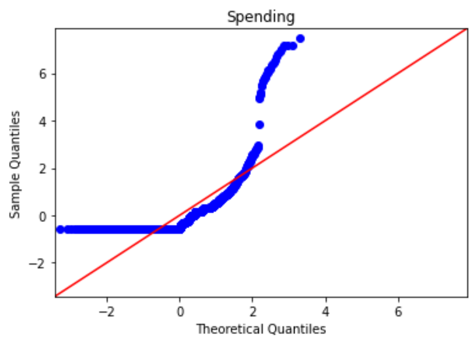
Feature Transformation:

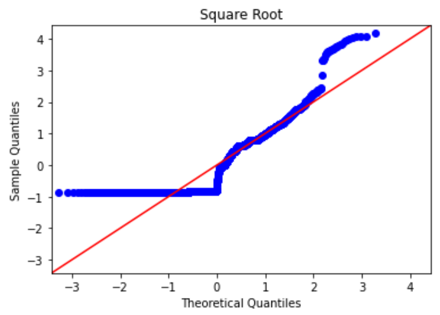
In the words of George Easton "if your model uses distance, it is better to transform attributes so they are normally distributed." Or, to quote Jesse Bockstedt, "if it's more predictive, go for it."

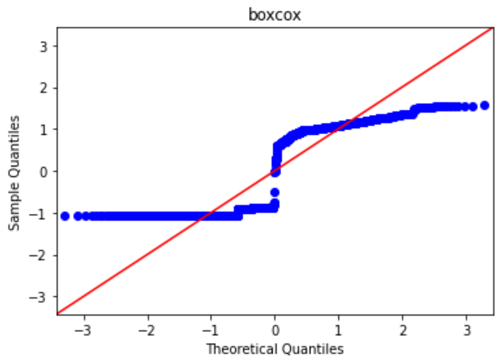
In this dataset, *Spending* and *Freq* (number of transactions in the previous year) were heavily right-skew, or, for the non-technical reader, 'smooshed together.' This can be an issue for algorithms like k-NN or linear regression, which use distance between points as part of their prediction. Using QQ plots as a visual aid, we experimented with different transformations for these variables. First, we worked on *Freq*:



The red line represents the normal distribution, and the blue points represent our data. We decided the log transformation gave the closest results. Note that our precise transformation was log(*Freq*+0.1), as we cannot take the log of a zero value. Next, we repeated this process with *Spending*:







Initially, the square root appeared better, however, after building models on both the square root and Box-Cox transformations, we found superior performance for all three models using Box-Cox. One thought as to why this was the case: square root is a less drastic transformation, and given how highly skewed our data was, perhaps this was a factor in Box-Cox's superior performance.

Technical aside: transforming the target variable (*Spending*) can lead to difficulties interpreting model performance, unless the predictions are inverted back to their original units. For example, we can invert log(x) by exponentiating our predictions (ie, $e^{prediction}$). This technical hurdle was overcome using the wrapper function TransformedTargetRegressor from sklearn, and an inverted Box-Cox function.

More on Box-Cox: developed by English Statisticians George Box and David Cox, Box-Cox is a type of power transformation (below), where *lambda* is estimated via a profile likelihood function and goodness-of-fit tests (source). This technique is used for a number of reasons, including stabilizing variance. In our case, it was employed to make the target variable more normal distribution-like.

$$y_i^{(\lambda)} = \begin{cases} \dfrac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln y_i & \text{if } \lambda = 0, \end{cases}$$

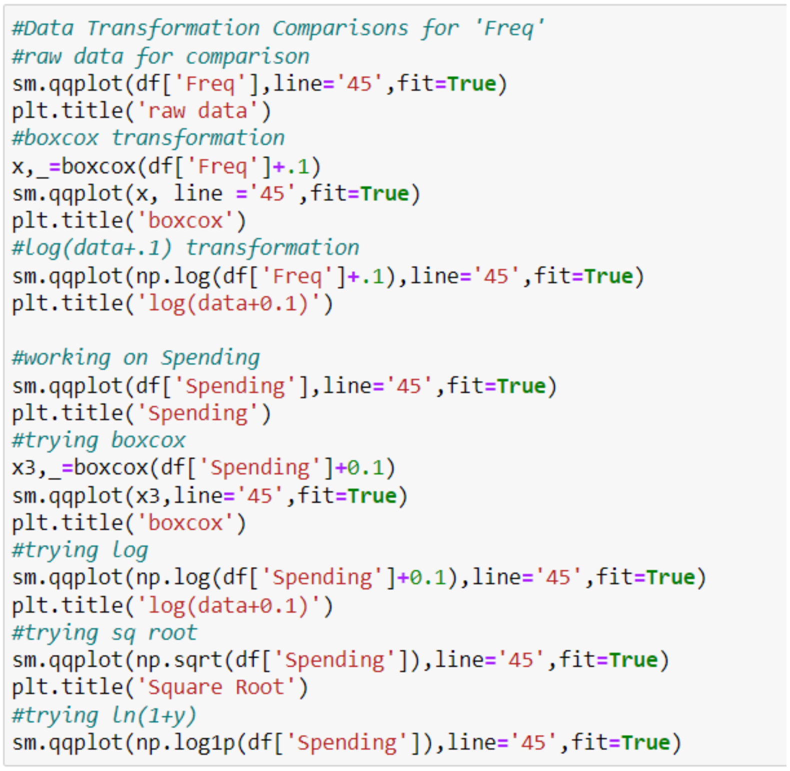
Scaling Features:

Some of the models we are testing are indifferent to normalization (decision tree), however, other models are improved by it (k-NN). Because of this, we decided to normalize all attributes, improving some model's performance while not harming others. We used z-score normalization, where $Z = \frac{x_i - \mu}{\sigma}$, so that our features had the same scale. If we had not, features with larger ranges could have more impact than those with smaller ranges.

Code:

```python
# ===========================================================================
# Feature Engineering, Transform, and Scale Data
# ===========================================================================
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from scipy.stats import boxcox
from scipy.special import inv_boxcox
import pandas as pd
import numpy as np

#copying df
df_fe=df.copy()

#account age variable
df_fe['account_age']=df['1st_update_days_ago']-df['last_update_days_ago']
#freq transformation
df_fe['Freq']=np.log(df['Freq']+0.1)
#spending transformation
df_fe['Spending'],_=boxcox(df['Spending']+0.1)
#dropping sequence num. column, unneccesary
df_fe=df_fe.drop(columns=['sequence_number'])
#dropping Purchase column, data leakage
df_fe=df_fe.drop(columns=['Purchase'])
#dropping Last Update column, multicollinearity
df_fe=df_fe.drop(columns=['last_update_days_ago'])
#dropping 1st Update column, multicollinearity
df_fe=df_fe.drop(columns=['1st_update_days_ago'])

#scaling all but target variable
original_colnames=df_fe.columns

col_names=['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',
           'Web order','Gender=male', 'Address_is_res', 'account_age']

ct = ColumnTransformer([('attributes', StandardScaler(), col_names)], remainder='drop')
df_fe[col_names]=ct.fit_transform(df_fe)

#function to undo y transformation
def y_xform(ypred):
    return np.round(inv_boxcox(ypred,-0.02965434077579322)-.1,5)
```

```python
# ===========================================================================
# Splitting Data, Building Models
# ===========================================================================
from sklearn.model_selection import train_test_split # Split validation class
X=df_fe.loc[:, df_fe.columns != 'Spending']# no spending
y=df_fe['Spending']
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3, random_state=42)
```



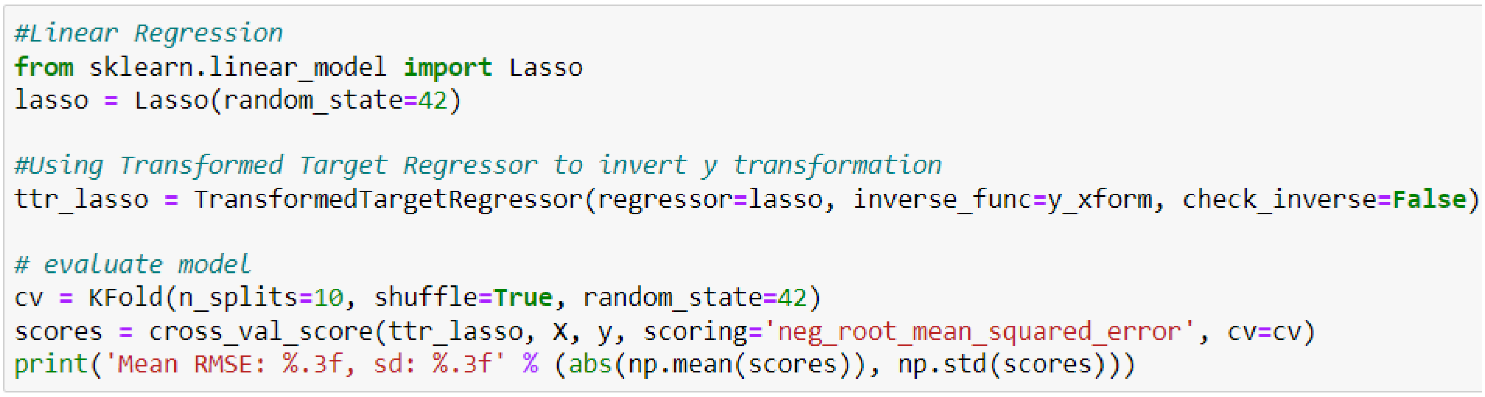
```python
#Data Transformation Comparisons for 'Freq'
#raw data for comparison
sm.qqplot(df['Freq'],line='45',fit=True)
plt.title('raw data')
#boxcox transformation
x,_=boxcox(df['Freq']+.1)
sm.qqplot(x, line ='45',fit=True)
plt.title('boxcox')
#log(data+.1) transformation
sm.qqplot(np.log(df['Freq']+.1),line='45',fit=True)
plt.title('log(data+0.1)')

#working on Spending
sm.qqplot(df['Spending'],line='45',fit=True)
plt.title('Spending')
#trying boxcox
x3,_=boxcox(df['Spending']+0.1)
sm.qqplot(x3,line='45',fit=True)
plt.title('boxcox')
#trying log
sm.qqplot(np.log(df['Spending']+0.1),line='45',fit=True)
plt.title('log(data+0.1)')
#trying sq root
sm.qqplot(np.sqrt(df['Spending']),line='45',fit=True)
plt.title('Square Root')
#trying ln(1+y)
sm.qqplot(np.log1p(df['Spending']),line='45',fit=True)
```

Linear Regression:

Of the linear regression models tested in Part A, Lasso (least absolute shrinkage and selection operator) performed the best. As mentioned earlier, non-invariant models like Lasso benefit from attribute scaling. The post-feature-engineering improvement was *very dramatic* (see below). Linear regressions like Lasso are also a good choice for data like ours with a large number of dimensions, as it performs **automatic feature selection**.

Initial Lasso Average Test RMSE	125.705
Feature Engineering Lasso Average Test RMSE	6.183

Code:



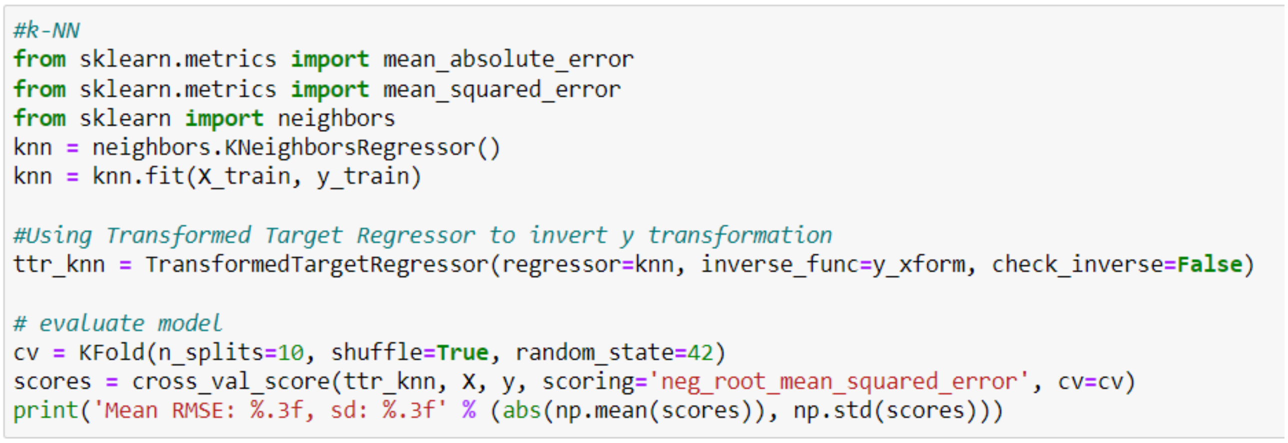
```python
#Linear Regression
from sklearn.linear_model import Lasso
lasso = Lasso(random_state=42)

#Using Transformed Target Regressor to invert y transformation
ttr_lasso = TransformedTargetRegressor(regressor=lasso, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_lasso, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

k-NN:

Similar to Lasso regression, k-NN benefits from normalized data. It is little surprise, then, that the model improved significantly (see below). In comparison to our Lasso regression's results, it appears the absence of automatic feature selection reduces k-NN's efficacy on high-dimensional datasets like ours.

Initial k-NN Average Test RMSE	170.715
Feature Engineering k-NN Average Test RMSE	88.862

Code:



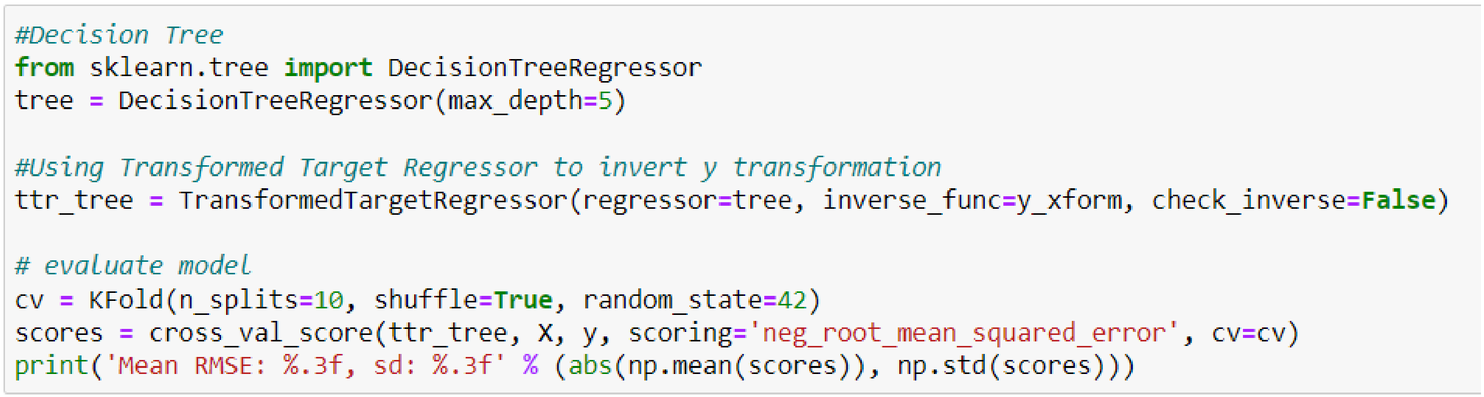
```
#k-NN
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn import neighbors
knn = neighbors.KNeighborsRegressor()
knn = knn.fit(X_train, y_train)

#Using Transformed Target Regressor to invert y transformation
ttr_knn = TransformedTargetRegressor(regressor=knn, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_knn, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

Regression Tree:

Of our three models, the Regression tree improved the least. This, however, is not particularly surprising. Its performance improved when we dropped columns that were redundant or causing data leakage, and performance improved when we transformed the y-variable (creating better decision boundaries), however, scaling did *not* improve the regression tree, and thus, these improvements feel lackluster compared to the previous two models.

Initial Regression Tree Average Test RMSE	135.345
Feature Engineering Regression Tree Average Test RMSE	112.151

Code:



```
#Decision Tree
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(max_depth=5)

#Using Transformed Target Regressor to invert y transformation
ttr_tree = TransformedTargetRegressor(regressor=tree, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_tree, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

Why Generalization Performance Improved:

Our feature engineering results feel a bit like the fairy tale *Big, Bad, Wolf*. The scaling-sensitive Lasso blew our house down, k-NN improved nicely, and the Decision Tree Regression was anticlimactic. To summarize the reasons for this, all models benefited from dropping columns and creating the account age column, k-NN and Lasso benefited from *both* variable transformation and scaling, while the Regression Trees saw a smaller benefit from variable transformations.

Fortunately, this is not the end of our story. In the next section, we will optimize our model parameters. Perhaps the true benefit of these transformations will be realized after parameter tuning, especially in the case of the Regression Tree?

Part C: Parameter Tuning

(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.**

[part a is worth 35 points in total:
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
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
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
5 points for discussing which of the three models yields the best performance

Part C: Parameter Tuning

General Process:

While producing different results, the parameter tuning process was the same for all three models. First, using a cross-validated grid search, we assessed which model parameters achieve the best RMSE. Next, we used those parameters and cross validation to generalize model performance.

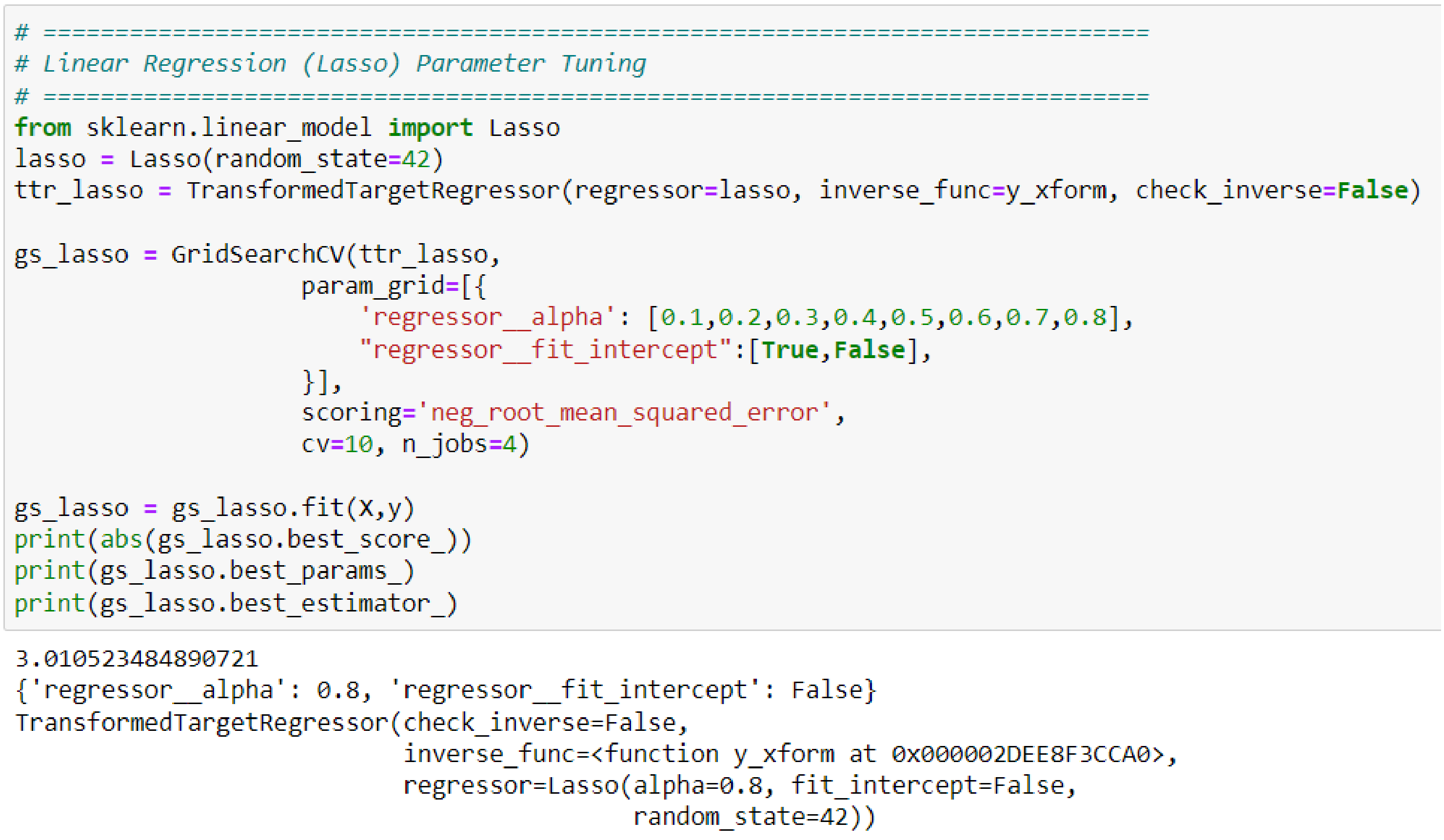
Linear Regression:

In Part B, we saw fantastic results using the Lasso linear regression. For further improvements, we followed the following process. First, we used the wrapper function TransformedTargetRegressor, which allows us to train and test our models on a transformed target variable, but communicate results using the target variable's original unit, which is more interpretable. In other words, it allows us to look at the RMSE in dollars, rather than a Box Cox transformation of dollars. Next, using cross-validated grid search, we tuned the following parameters: alpha, which increases the number of zero coefficients the algorithm applies to features, and fit intercept, which either calculates the y-intercept for the model or not.

Our best results used a high alpha value. This means that more aggressive feature selection produced the most predictive results. This is unsurprising given the high dimensionality of the data. Finally, we used cross validation in order to generalize model performance.

Initial Lasso Average Test RMSE	125.705
Feature Engineering Lasso Average Test RMSE	6.183
Parameter Tuning Lasso Average Test RMSE	3.006

<u>Code:</u>



```
# ============================================================================
# Linear Regression (Lasso) Parameter Tuning
# ============================================================================
from sklearn.linear_model import Lasso
lasso = Lasso(random_state=42)
ttr_lasso = TransformedTargetRegressor(regressor=lasso, inverse_func=y_xform, check_inverse=False)

gs_lasso = GridSearchCV(ttr_lasso,
                    param_grid=[{
                        'regressor__alpha': [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8],
                        "regressor__fit_intercept":[True,False],
                    }],
                    scoring='neg_root_mean_squared_error',
                    cv=10, n_jobs=4)

gs_lasso = gs_lasso.fit(X,y)
print(abs(gs_lasso.best_score_))
print(gs_lasso.best_params_)
print(gs_lasso.best_estimator_)
```

```
3.010523484890721
{'regressor__alpha': 0.8, 'regressor__fit_intercept': False}
TransformedTargetRegressor(check_inverse=False,
                           inverse_func=<function y_xform at 0x000002DEE8F3CCA0>,
                           regressor=Lasso(alpha=0.8, fit_intercept=False,
                                           random_state=42))
```

```python
#Lasso Regression
lasso = Lasso(random_state=42,alpha=0.8,fit_intercept=False)

#Using Transformed Target Regressor to invert y transformation
ttr_lasso = TransformedTargetRegressor(regressor=lasso, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_lasso, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

```
Mean RMSE: 3.006, sd: 0.103
```

k-NN:

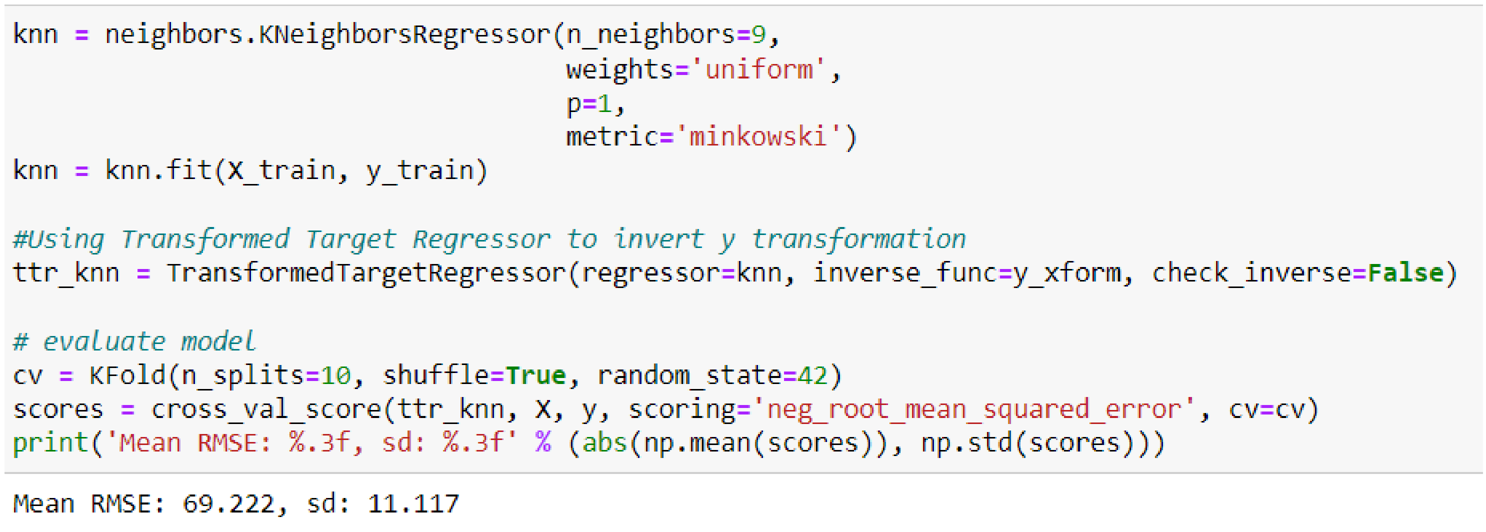
Parameter tuning for k-NN followed the same process as Lasso: apply wrapper function, then perform cross validated grid search. In the case of k-NN, we wanted to tune the following parameters: the number of neighbors, the type of distance weighting, the k-NN algorithm type, and the type of distance measure. For our dataset, 9 neighbors performed the best, with uniform weighting using Manhattan distance. Finally, we used cross-validation to generalize model performance.

Initial k-NN Average Test RMSE	170.715
Feature Engineering k-NN Average Test RMSE	88.862
Parameter Tuning k-NN Average Test RMSE	69.222

Code:

```python
# ============================================================================
# k-NN Parameter Tuning
# ============================================================================
from sklearn.model_selection import GridSearchCV
k_list = list(range(1, 10))
gs_knn = GridSearchCV(estimator=ttr_knn,
                      param_grid=[{'regressor__n_neighbors':k_list,
                                  'regressor__weights' : ['uniform','distance'],
                                  'regressor__algorithm': ['auto', 'ball_tree','kd_tree', 'brute'],
                                  'regressor__p':[1,2]}],
                                  scoring='neg_root_mean_squared_error',cv=10,n_jobs=4)

gs_knn = gs_knn.fit(X,y)
print(abs(gs_knn.best_score_))
print(gs_knn.best_params_)
print(gs_knn.best_estimator_)
```

```
71.11255445139105
{'regressor__algorithm': 'auto', 'regressor__n_neighbors': 9, 'regressor__p': 1, 'regressor__weights': 'uniform'}
TransformedTargetRegressor(check_inverse=False,
                           inverse_func=<function y_xform at 0x000002DEE8F3CCA0>,
                           regressor=KNeighborsRegressor(n_neighbors=9, p=1))
```



```
knn = neighbors.KNeighborsRegressor(n_neighbors=9,
                                    weights='uniform',
                                    p=1,
                                    metric='minkowski')
knn = knn.fit(X_train, y_train)

#Using Transformed Target Regressor to invert y transformation
ttr_knn = TransformedTargetRegressor(regressor=knn, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_knn, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

```
Mean RMSE: 69.222, sd: 11.117
```

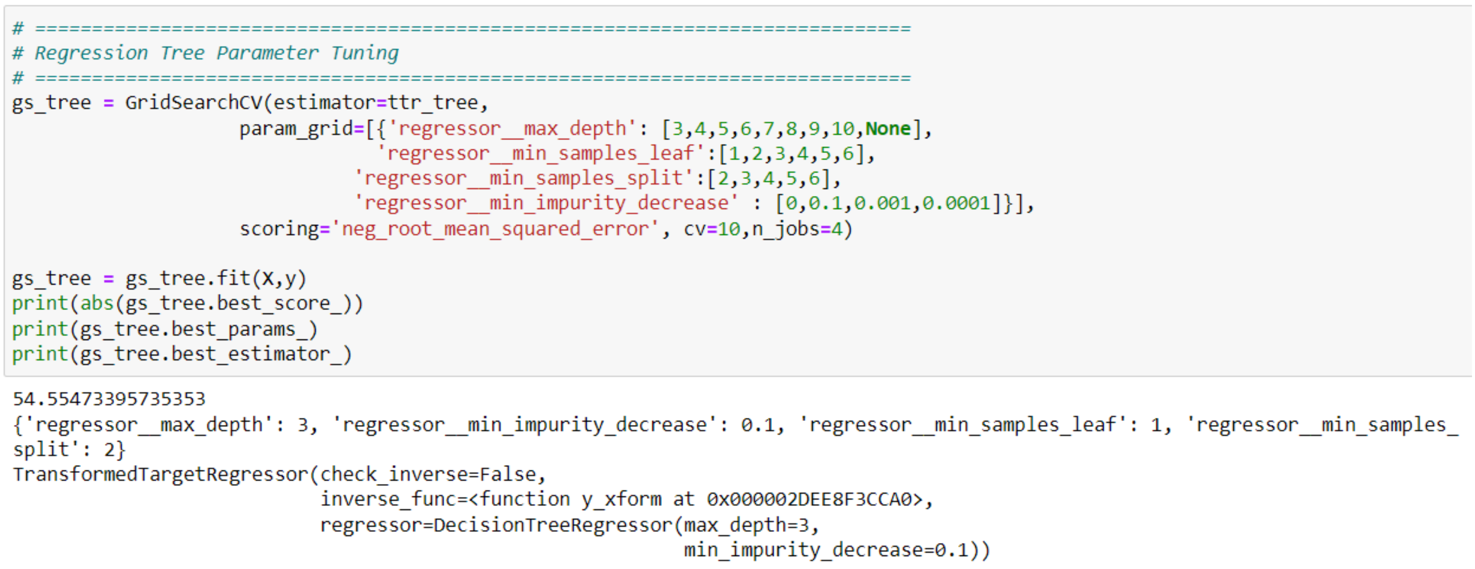
Regression Tree:

For our third model, we continued our parameter tuning process. In this case, we wanted to tune the following parameters: maximum tree depth, minimum leaf samples, minimum split samples, and minimum impurity decrease. These parameters guide when our tree is allowed to split, and how deep the tree will go. Our best model had a relatively small maximum depth, perhaps because larger depths overfit the training data.

This is a more dramatic improvement than our k-NN model. One reason for this is the difference in the number of parameters we can tune: the default regression tree parameters were very different from the optimal, leading to a large improvement in predictive capability

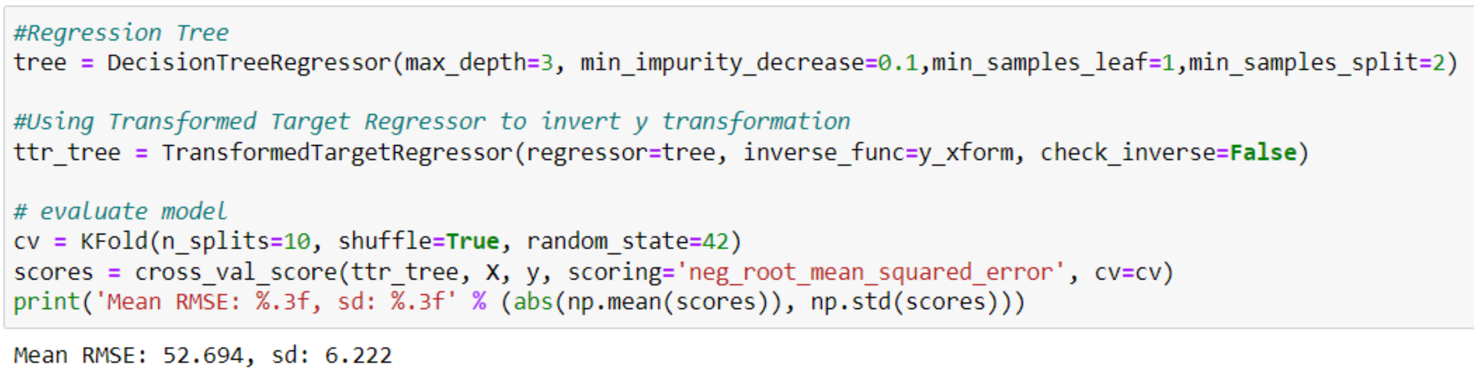
Initial Regression Tree Average Test RMSE	135.345
Feature Engineering Regression Tree Average Test RMSE	112.151
Parameter Tuning Regression Tree Average Test RMSE	52.694

Code:



```
# ==============================================================================
# Regression Tree Parameter Tuning
# ==============================================================================
gs_tree = GridSearchCV(estimator=ttr_tree,
                       param_grid=[{'regressor__max_depth': [3,4,5,6,7,8,9,10,None],
                                    'regressor__min_samples_leaf':[1,2,3,4,5,6],
                                  'regressor__min_samples_split':[2,3,4,5,6],
                                  'regressor__min_impurity_decrease' : [0,0.1,0.001,0.0001]}],
                       scoring='neg_root_mean_squared_error', cv=10,n_jobs=4)

gs_tree = gs_tree.fit(X,y)
print(abs(gs_tree.best_score_))
print(gs_tree.best_params_)
print(gs_tree.best_estimator_)
```

```
54.55473395735353
{'regressor__max_depth': 3, 'regressor__min_impurity_decrease': 0.1, 'regressor__min_samples_leaf': 1, 'regressor__min_samples_
split': 2}
TransformedTargetRegressor(check_inverse=False,
                           inverse_func=<function y_xform at 0x000002DEE8F3CCA0>,
                           regressor=DecisionTreeRegressor(max_depth=3,
                                                           min_impurity_decrease=0.1))
```



```
#Regression Tree
tree = DecisionTreeRegressor(max_depth=3, min_impurity_decrease=0.1,min_samples_leaf=1,min_samples_split=2)

#Using Transformed Target Regressor to invert y transformation
ttr_tree = TransformedTargetRegressor(regressor=tree, inverse_func=y_xform, check_inverse=False)

# evaluate model
cv = KFold(n_splits=10, shuffle=True, random_state=42)
scores = cross_val_score(ttr_tree, X, y, scoring='neg_root_mean_squared_error', cv=cv)
print('Mean RMSE: %.3f, sd: %.3f' % (abs(np.mean(scores)), np.std(scores)))
```

```
Mean RMSE: 52.694, sd: 6.222
```

Discussion of Model Performance:

Model RMSE	Lasso	k-NN	Regression Tree
Part A	125.705	170.715	135.345
Part B	6.183	88.862	112.151
Part C	3.006	69.222	52.694

Initially, the lasso model was our best performer, however, it was fairly similar to the regression tree. After feature engineering, this changed drastically, with lasso far out-performing the other models. Feature engineering also dramatically improved the k-NN model, leaving the regression tree as the least predictive model at the end of Part B. Finally, after parameter tuning, the performance of our models shifted again: while lasso is still by far the best performer, the regression tree with tuned parameters performed better than k-NN.

Were we to deploy a model from this project, we would unequivocally recommend the lasso regression. It is able to predict *Spending* within a handful of dollars, in part because it effectively automates many of the challenges of a high-dimensional dataset such as ours.