EMORY UNIVERSITY

# Introduction to Business Data Analytics

**Homework #4**

______Jie Zhu______

(put your name above (incl. any nicknames))

Total grade: ________ out of ___165___ points

***Please answer all the questions and submit your assignment as a single PDF by uploading it on Canvas.***

**1) (20 points) Assume that you built a model for predicting consumer credit ratings and evaluated it on the test dataset of 5 records. Based the following 5 actual and predicted credit ratings (see table below), calculate the following performance metrics for your model: MAE, MAPE, RMSE, and Average error.**

| **Actual Credit Rating** | **Predicted Credit Rating** |
|---|---|
| 670 | 710 |
| 680 | 660 |
| 550 | 600 |
| 740 | 800 |
| 700 | 600 |

[4 points MAE – 1 point for formula and 3 points for correct application of formula
4 points MAPE– 1 point for formula and 3 points for correct application of formula
4 points RMSE– 1 point for formula and 3 points for correct application of formula
4 points Average Error– 1 point for formula and 3 points for correct application of formula]

$$e_i = \hat{y} - y = p_i - a_i$$

$$\textbf{(MAE)} = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

MAE = (|710-670| + |660-680| + |600-550| + |800-740| + |600-700|) / 5 = (40+20+50+60+100) / 5 = 54

$$\textbf{(MAPE)} = \frac{\sum_{i=1}^{n} |e_i/a_i|}{n}$$

MAPE = (40/670 + 20/680 + 50/550 + 60/740 + 100/700) / 5 = 8.08%

$$\textbf{(RMSE)} = \sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n}}$$

RMSE = SQRT((40^2 + (-20)^2 + 50^2 + 60^2 + (-100)^2)/5) = SQRT((1600+400+2500+3600+10000)/5) = SQRT(3620) = 60.17

$$\text{Average error} = \frac{\sum_{i=1}^{n} e_i}{n}$$

Average Error = (40 + (-20) + 50 + 60 + (-100)) / 5 = 6

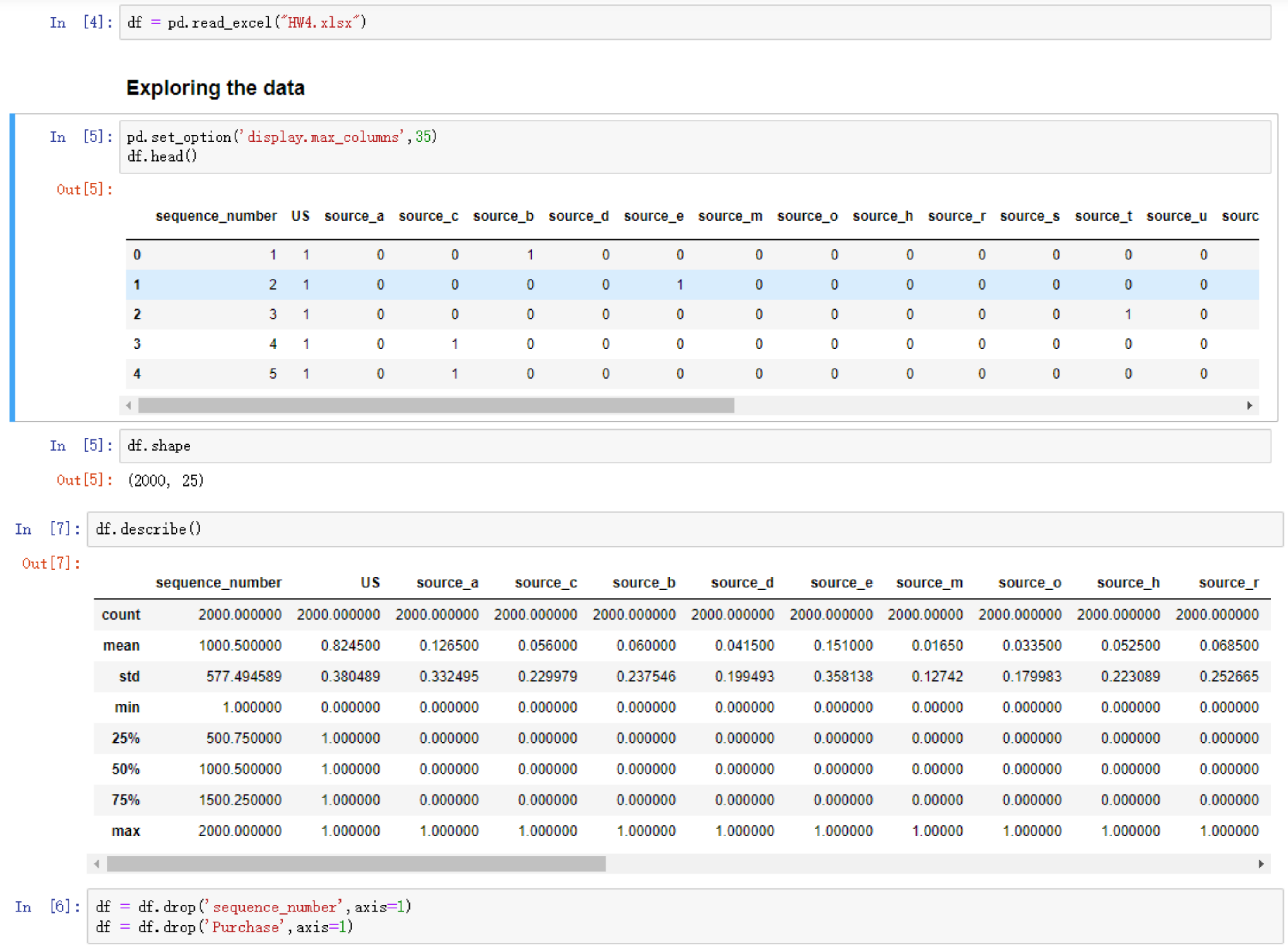
**2) (145 points) Use numeric prediction techniques to build a predictive model for the HW4.xlsx dataset. This dataset is provided on Canvas and contains data about whether or not different consumers made a purchase in response to a test mailing of a certain catalog and, in case of a purchase, how much 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)



```
In  [4]: df = pd.read_excel("HW4.xlsx")
```

### Exploring the data

```
In  [5]: pd.set_option('display.max_columns',35)
         df.head()
Out[5]:
```

| | sequence_number | US | source_a | source_c | source_b | source_d | source_e | source_m | source_o | source_h | source_r | source_s | source_t | source_u | sourc |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 1 | 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 2 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| 3 | 4 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 4 | 5 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

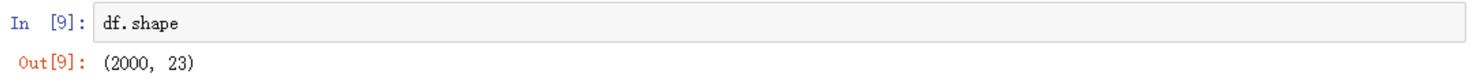
```
In  [5]: df.shape
Out[5]: (2000, 25)
```

```
In  [7]: df.describe()
Out[7]:
```

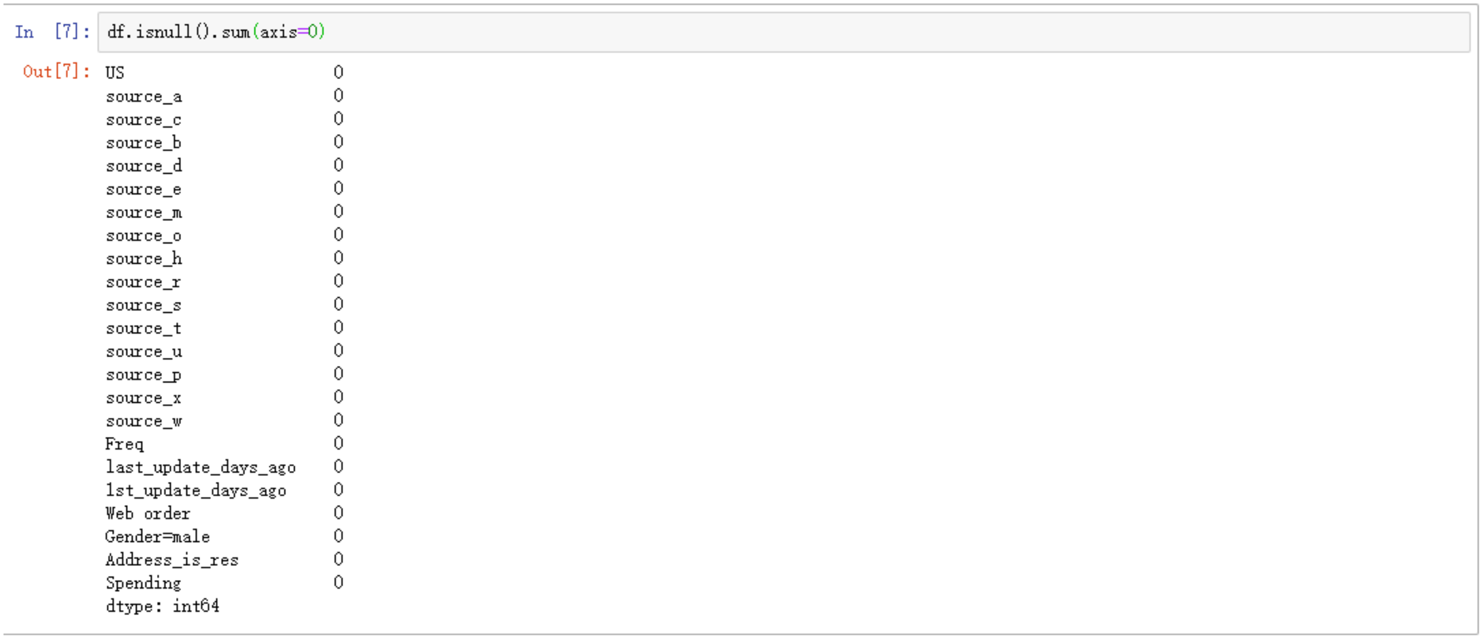
| | sequence_number | US | source_a | source_c | source_b | source_d | source_e | source_m | source_o | source_h | source_r |
|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 2000.000000 | 2000.000000 | 2000.000000 | 2000.000000 | 2000.000000 | 2000.000000 | 2000.000000 | 2000.00000 | 2000.000000 | 2000.000000 | 2000.000000 |
| mean | 1000.500000 | 0.824500 | 0.126500 | 0.056000 | 0.060000 | 0.041500 | 0.151000 | 0.01650 | 0.033500 | 0.052500 | 0.068500 |
| std | 577.494589 | 0.380489 | 0.332495 | 0.229979 | 0.237546 | 0.199493 | 0.358138 | 0.12742 | 0.179983 | 0.223089 | 0.252665 |
| min | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 | 0.000000 | 0.000000 | 0.000000 |
| 25% | 500.750000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 | 0.000000 | 0.000000 | 0.000000 |
| 50% | 1000.500000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 | 0.000000 | 0.000000 | 0.000000 |
| 75% | 1500.250000 | 1.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.00000 | 0.000000 | 0.000000 | 0.000000 |
| max | 2000.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.00000 | 1.000000 | 1.000000 | 1.000000 |

```
In  [6]: df = df.drop('sequence_number',axis=1)
         df = df.drop('Purchase',axis=1)
```

Drop the ‘sequence_number’ column because it’s an ID column.
Drop the ‘Purchase’ column because it’s data leakage.

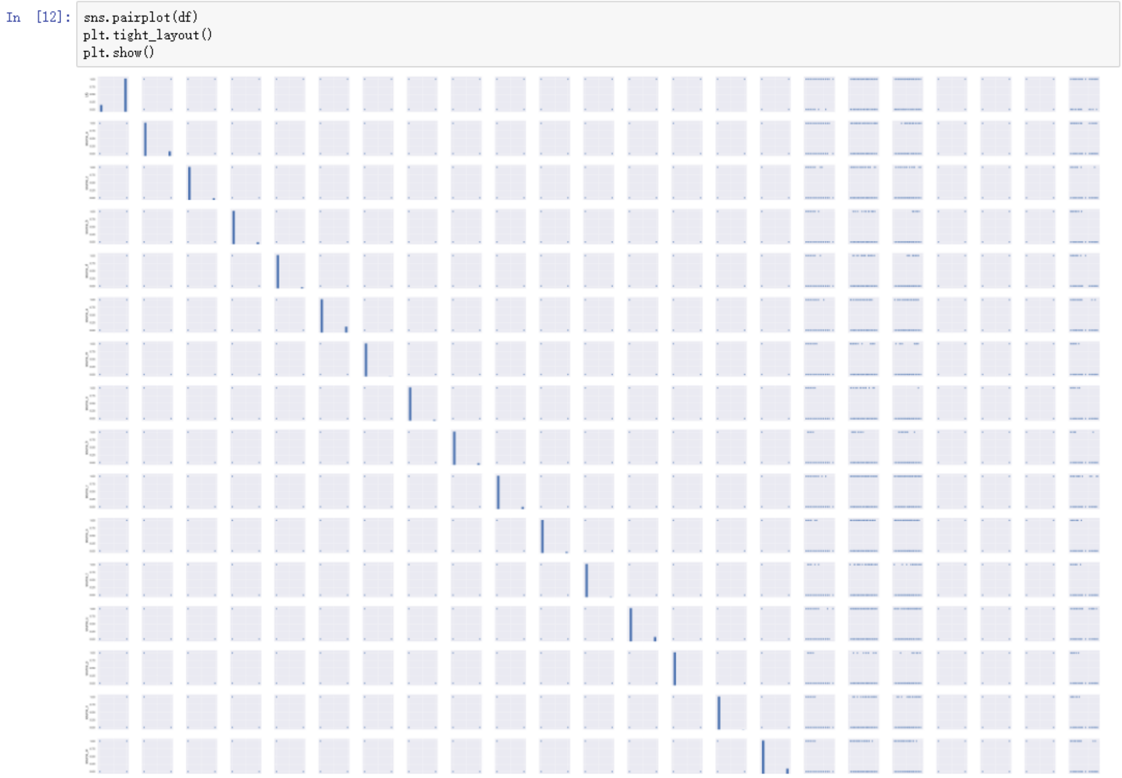


```
In  [9]: df.shape

Out[9]: (2000, 23)
```



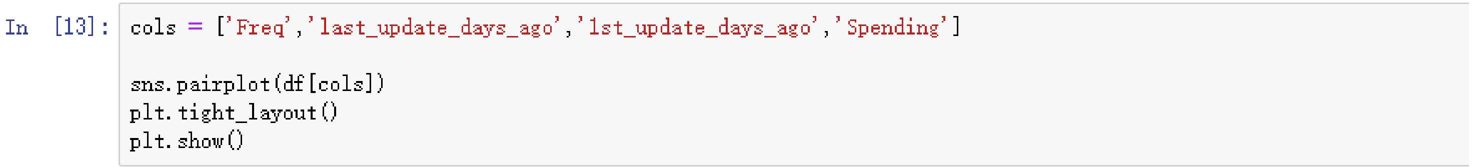
```
In  [7]: df.isnull().sum(axis=0)

Out[7]: 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
        Spending                0
        dtype: int64
```

Check data.frame.shape after dropping two columns.
Check if there’s any null data.

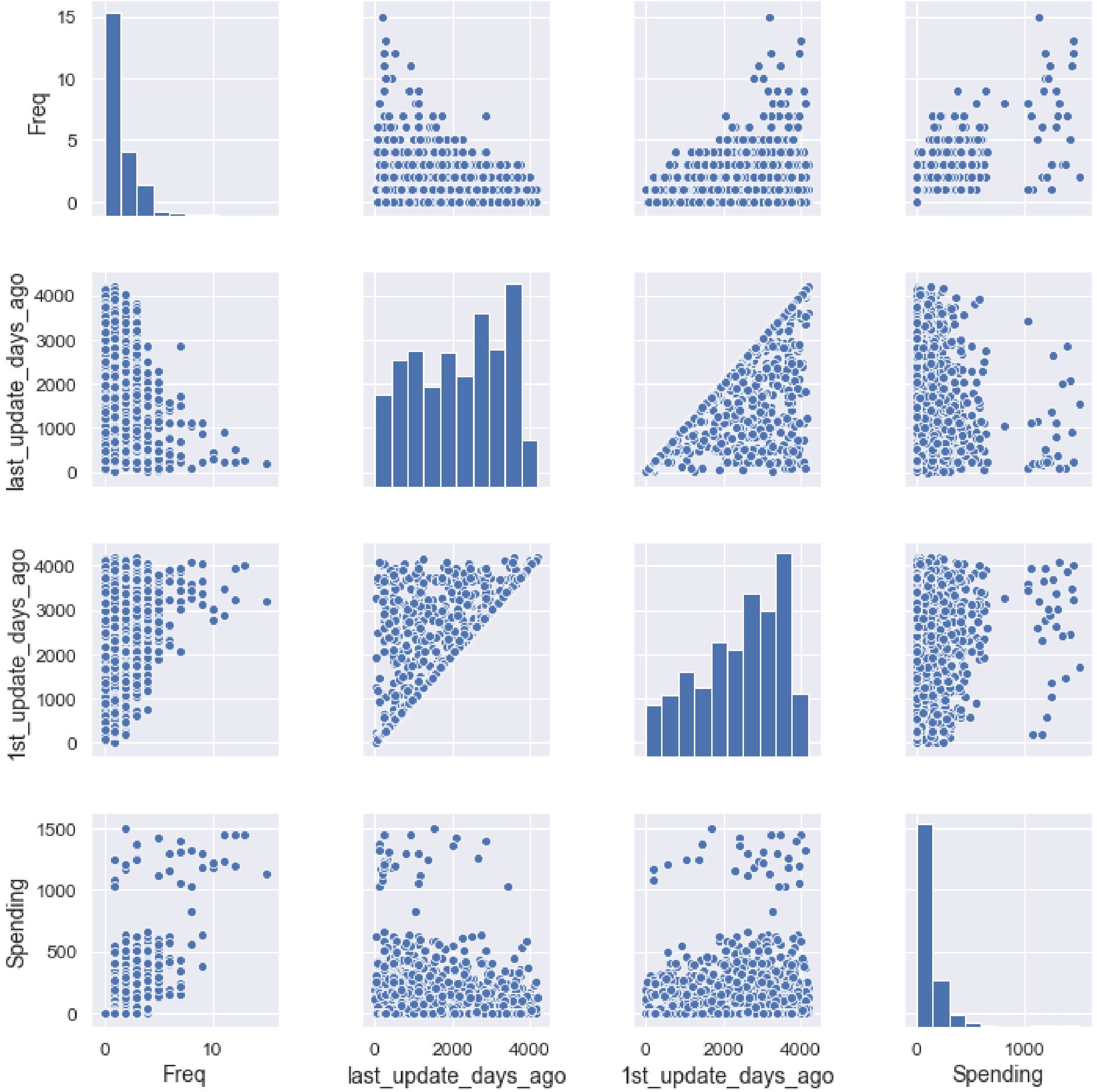


```
In  [12]: sns.pairplot(df)
          plt.tight_layout()
          plt.show()
```

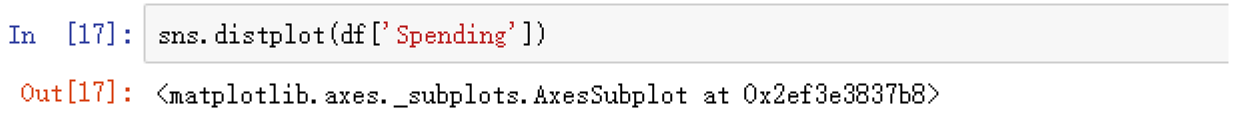
Visualization of pairplot of the data.frame.



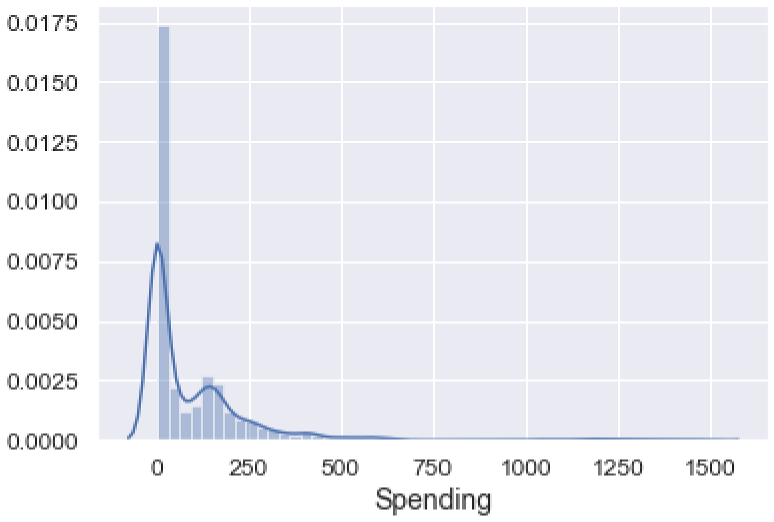
```
In  [13]: cols = ['Freq','last_update_days_ago','1st_update_days_ago','Spending']

          sns.pairplot(df[cols])
          plt.tight_layout()
          plt.show()
```



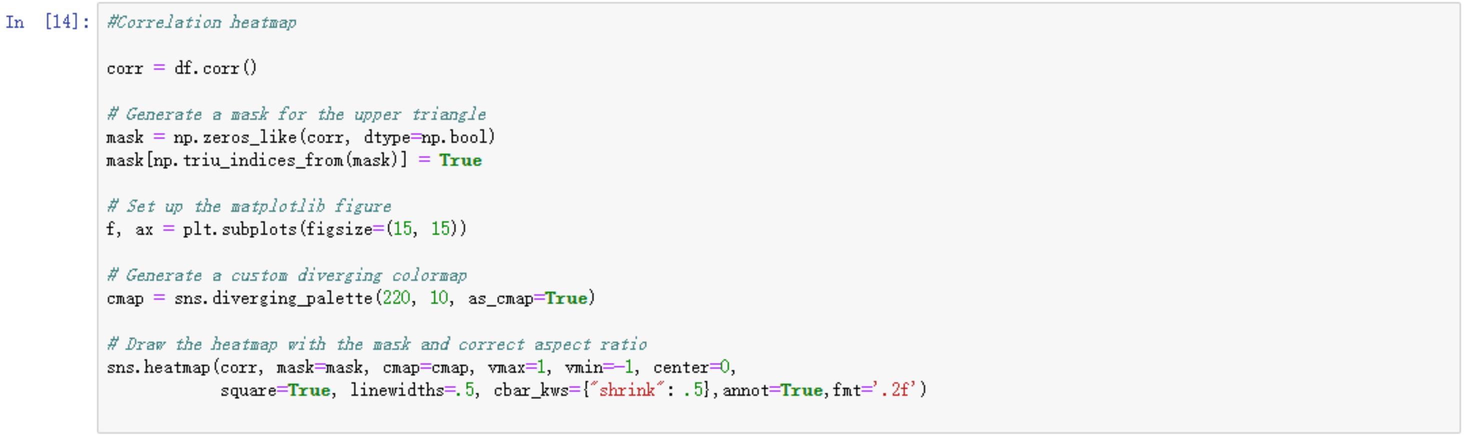
Visualization of pairplot of 4 non-binary numeric variables.



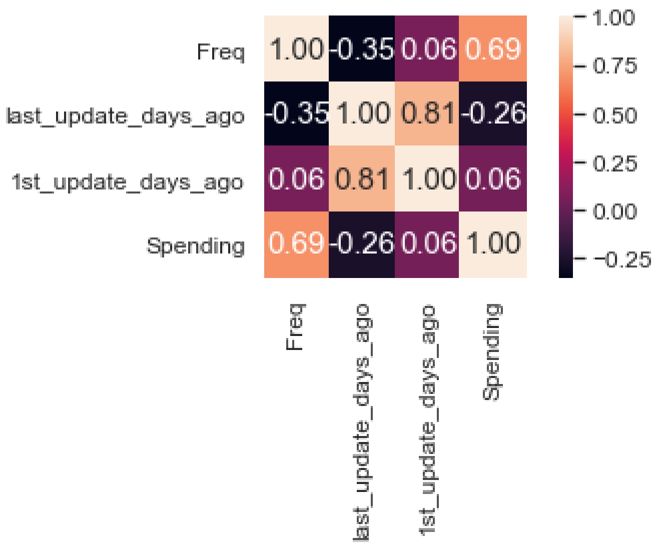
```
In  [17]: sns.distplot(df['Spending'])

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x2ef3e3837b8>
```



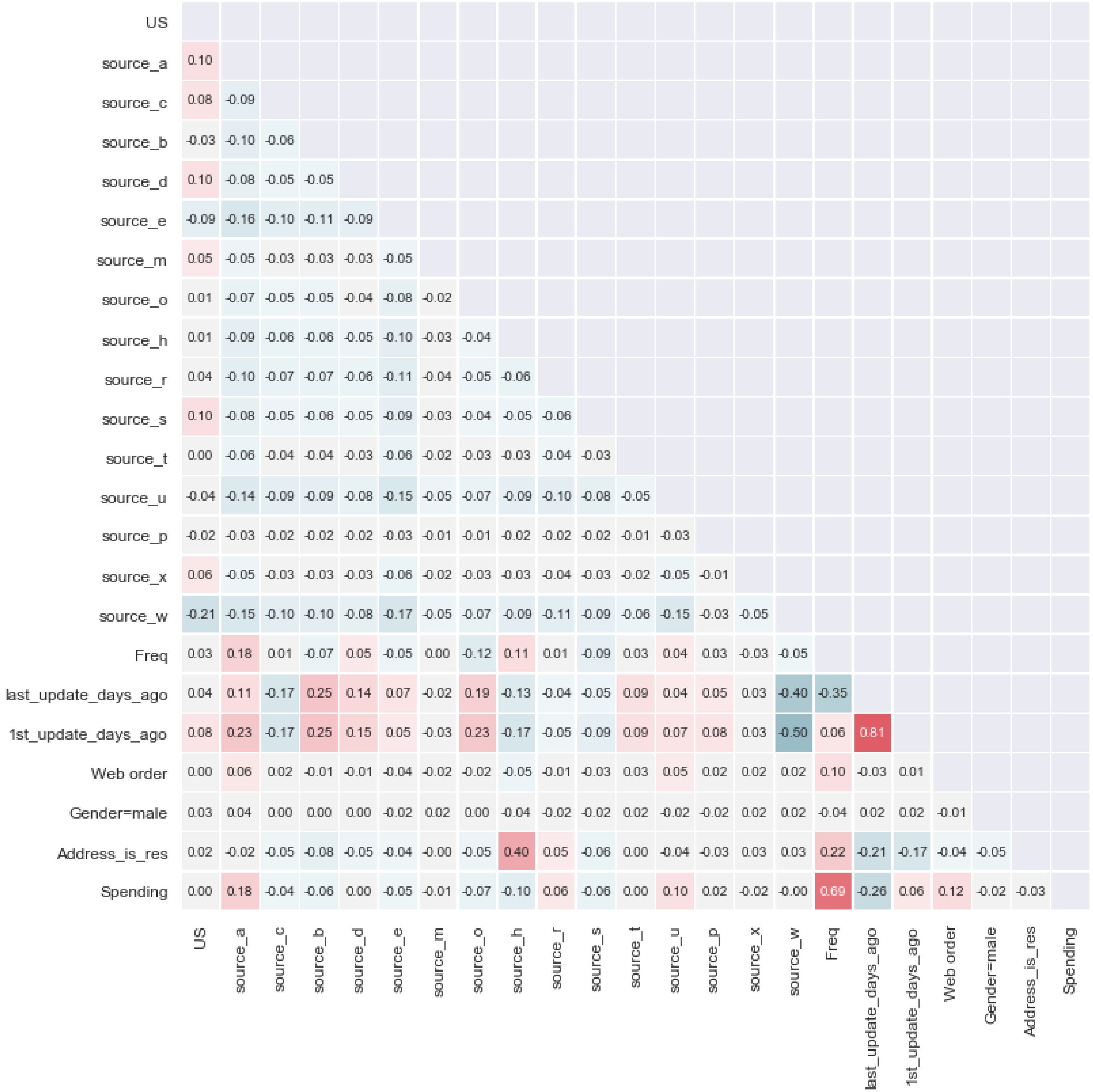
Visualization of target variable distribution.



```
In  [14]: #Correlation heatmap

          corr = df.corr()

          # Generate a mask for the upper triangle
          mask = np.zeros_like(corr, dtype=np.bool)
          mask[np.triu_indices_from(mask)] = True

          # Set up the matplotlib figure
          f, ax = plt.subplots(figsize=(15, 15))

          # Generate a custom diverging colormap
          cmap = sns.diverging_palette(220, 10, as_cmap=True)

          # Draw the heatmap with the mask and correct aspect ratio
          sns.heatmap(corr, mask=mask, cmap=cmap, vmax=1, vmin=-1, center=0,
                      square=True, linewidths=.5, cbar_kws={"shrink": .5},annot=True,fmt='.2f')
```



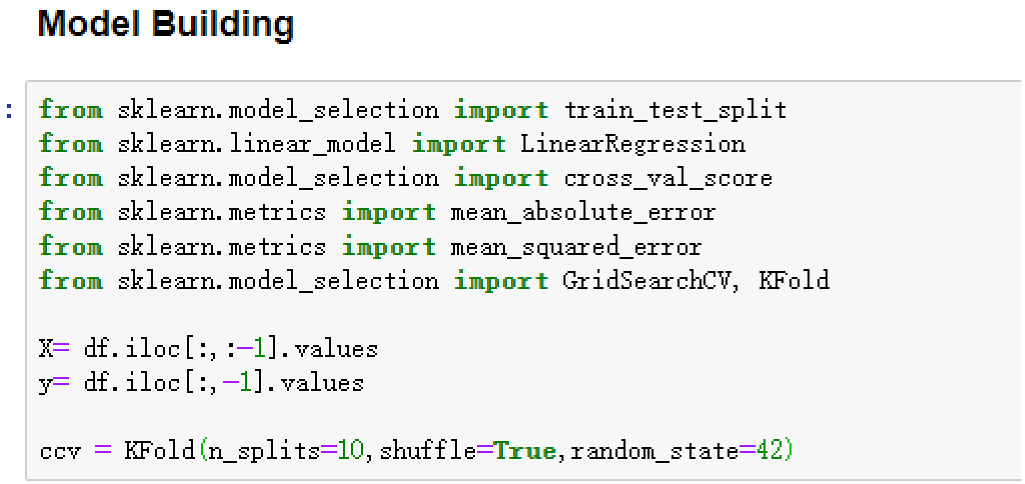
Visualization of correlation matrix for 4 non-binary numeric variables.



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

Visualization of correlation matrix for the whole data.frame.

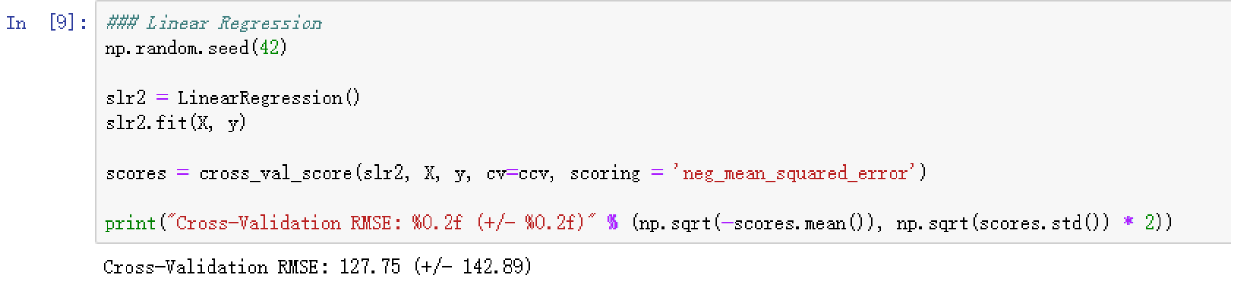
10 points for correctly building linear regression model - provide screenshots and explain what you are doing and the corresponding results



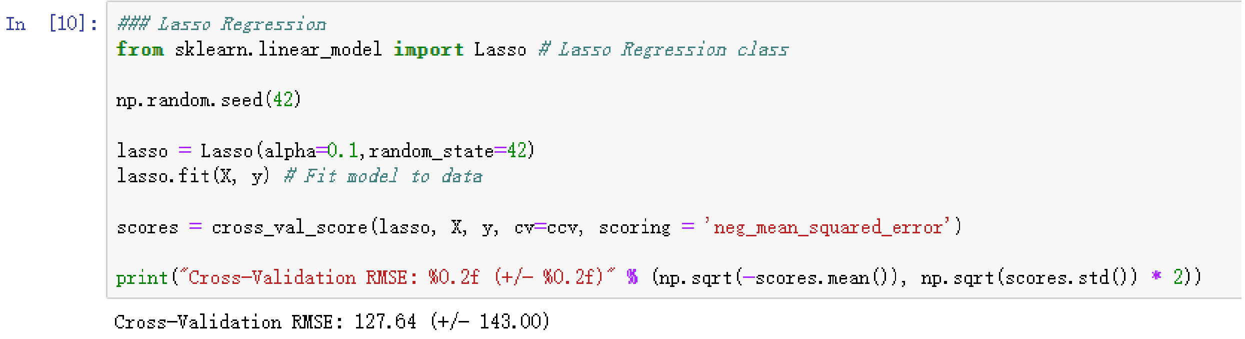
## Model Building

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV, KFold

X= df.iloc[:, :-1].values
y= df.iloc[:,-1].values

ccv = KFold(n_splits=10,shuffle=True,random_state=42)
```

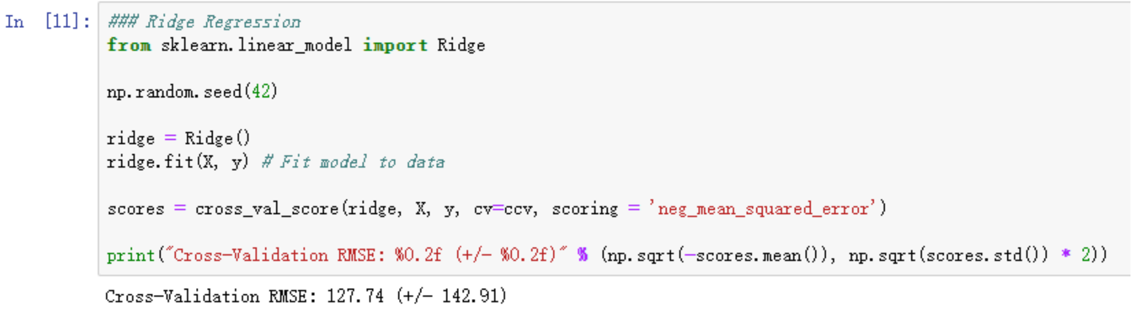
Set up X and y, specify random_state=42 for 10-fold cross validation.
Since my target variable “Spending” in this problem is numeric rather than categorical, I use Root Mean Squared Error (RMSE) to evaluate the performance of different models in terms of prediction error.
In addition, I use 10-fold cross validation to measure generalization performance.
X and y here are from original dataset after dropping 2 variables (‘sequence_number’, ‘Purchase’).



```
In [9]: ### Linear Regression
        np.random.seed(42)

        slr2 = LinearRegression()
        slr2.fit(X, y)

        scores = cross_val_score(slr2, X, y, cv=ccv, scoring = 'neg_mean_squared_error')

        print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

        Cross-Validation RMSE: 127.75 (+/- 142.89)
```



```
In [10]: ### Lasso Regression
         from sklearn.linear_model import Lasso # Lasso Regression class

         np.random.seed(42)

         lasso = Lasso(alpha=0.1,random_state=42)
         lasso.fit(X, y) # Fit model to data

         scores = cross_val_score(lasso, X, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 127.64 (+/- 143.00)
```



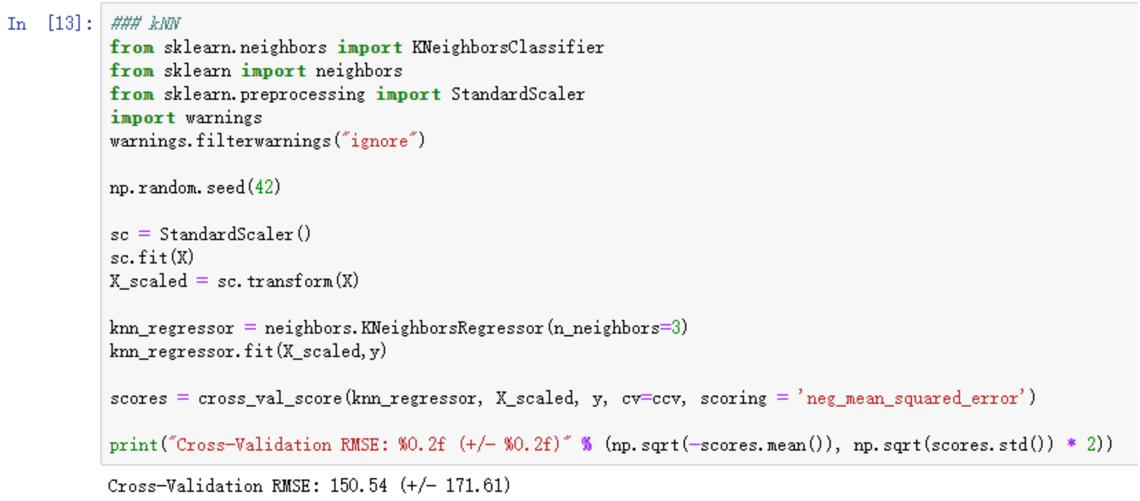
```
In [11]: ### Ridge Regression
         from sklearn.linear_model import Ridge

         np.random.seed(42)

         ridge = Ridge()
         ridge.fit(X, y) # Fit model to data

         scores = cross_val_score(ridge, X, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 127.74 (+/- 142.91)
```

For linear regression, I build 3 models, linear regression without penalty, lasso regression, and ridge regression. Their performance evaluated by 10-fold cross-validation are:

| | **RMSE** |
|---|---|
| Simple Linear | 127.75 |
| Lasso Linear | 127.64 |
| Ridge Linear | 127.74 |

Lasso Regression with alpha = 0.1 specified is slightly better than the other two models.

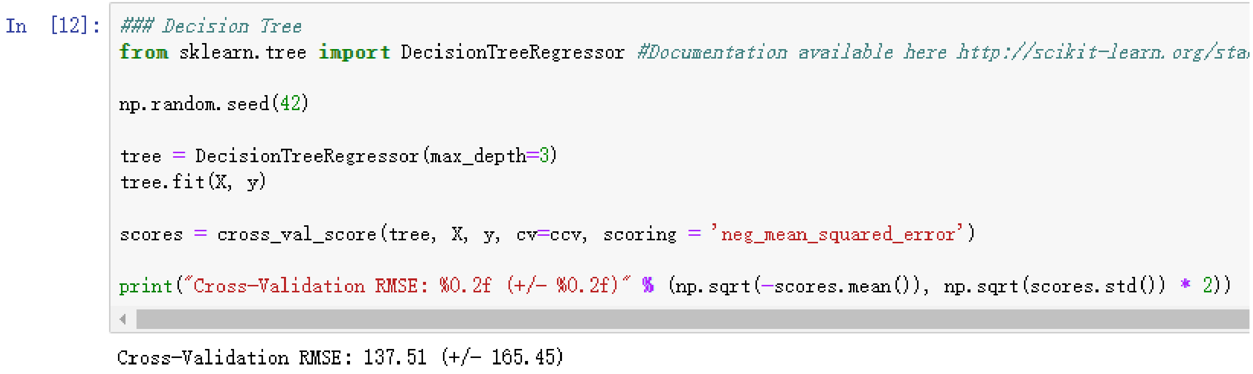
10 points for correctly building k-NN model - provide screenshots and explain what you are doing and the corresponding results



```
In [13]: ### kNN
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn import neighbors
         from sklearn.preprocessing import StandardScaler
         import warnings
         warnings.filterwarnings("ignore")

         np.random.seed(42)

         sc = StandardScaler()
         sc.fit(X)
         X_scaled = sc.transform(X)

         knn_regressor = neighbors.KNeighborsRegressor(n_neighbors=3)
         knn_regressor.fit(X_scaled,y)

         scores = cross_val_score(knn_regressor, X_scaled, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 150.54 (+/- 171.61)
```

For kNN model, I use StandardScaler() to standardize X into X_scaled. I then use X_scaled and y to train the model. In addition, I specify n_neighbors = 3 to make this kNN a 3NN model. The 10-fold cross-validation performance is RMSE = 150.54.

10 points for correctly building regression tree model - provide screenshots and explain what you are doing and the corresponding results



```
In [12]: ### Decision Tree
         from sklearn.tree import DecisionTreeRegressor #Documentation available here http://scikit-learn.org/sta

         np.random.seed(42)

         tree = DecisionTreeRegressor(max_depth=3)
         tree.fit(X, y)

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

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 137.51 (+/- 165.45)
```

For regression tree model, I specify max_depth =3. The 10-fold cross-validation performance is RMSE = 137.51.

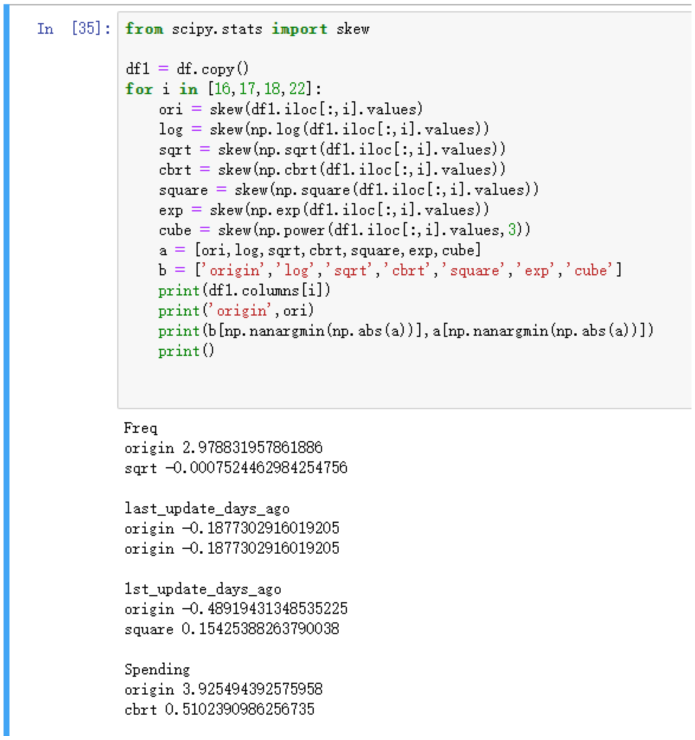
5 points for discussing which of the three models yields the best performance]

| | RMSE |
|---|---|
| Simple Linear | 127.75 |
| Lasso Linear | 127.64 |
| Ridge Linear | 127.74 |
| k-NN | 150.54 |
| Regression Tree | 137.51 |

Compiling the RMSE results from previous models, I find that using 10-fold cross-validation, all of the three linear regression models perform better than k-NN and regression tree. However, within the three linear models, Lasso Regression wins slightly with an RMSE of 127.64. Thus, the best performing model in term of RMSE is Lasso Regression with 10-fold cross-validation and original dataset.

**(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.**

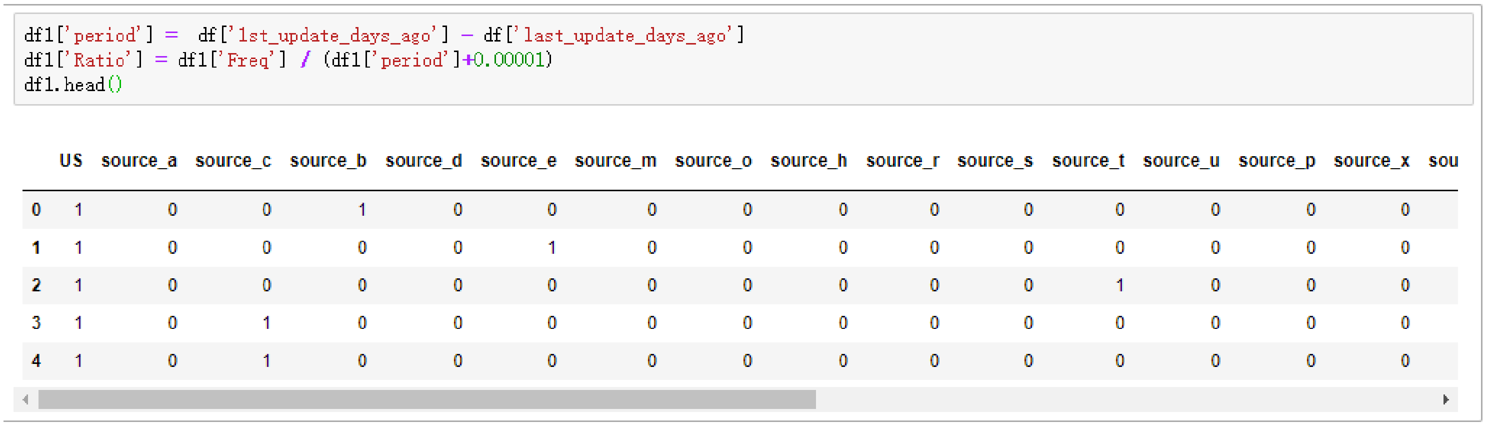
[part a is worth 50 points in total:
10 points for correctly building the new linear regression model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results



```
In  [35]: from scipy.stats import skew

          df1 = df.copy()
          for i in [16,17,18,22]:
              ori = skew(df1.iloc[:,i].values)
              log = skew(np.log(df1.iloc[:,i].values))
              sqrt = skew(np.sqrt(df1.iloc[:,i].values))
              cbrt = skew(np.cbrt(df1.iloc[:,i].values))
              square = skew(np.square(df1.iloc[:,i].values))
              exp = skew(np.exp(df1.iloc[:,i].values))
              cube = skew(np.power(df1.iloc[:,i].values,3))
              a = [ori,log,sqrt,cbrt,square,exp,cube]
              b = ['origin','log','sqrt','cbrt','square','exp','cube']
              print(df1.columns[i])
              print('origin',ori)
              print(b[np.nanargmin(np.abs(a))],a[np.nanargmin(np.abs(a))])
              print()
```

```
Freq
origin 2.978831957861886
sqrt -0.0007524462984254756

last_update_days_ago
origin -0.1877302916019205
origin -0.1877302916019205

1st_update_days_ago
origin -0.48919431348535225
square 0.15425388263790038

Spending
origin 3.925494392575958
cbrt 0.5102390986256735
```

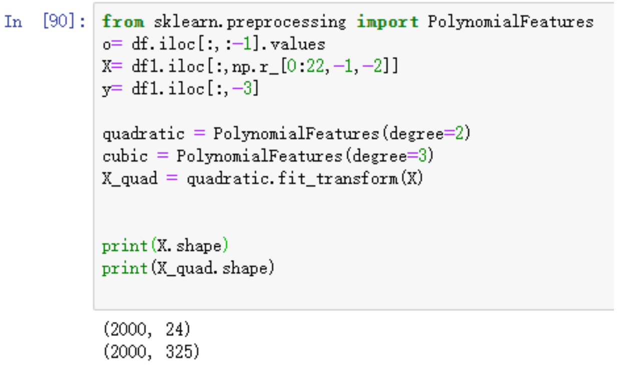
Before I build any new model, I run a FOR loop to find the current skewness of 4 non-binary numeric variables in the dataset (‘Freq’, ‘last_update_days_ago’, ‘1st_update_days_ago’, ‘Spending’) and the skewness after using log, sqrt, and cbrt. After the transformation of these 4 variables, the RMSEs of different models only get bigger. So I start to look for a new approach.



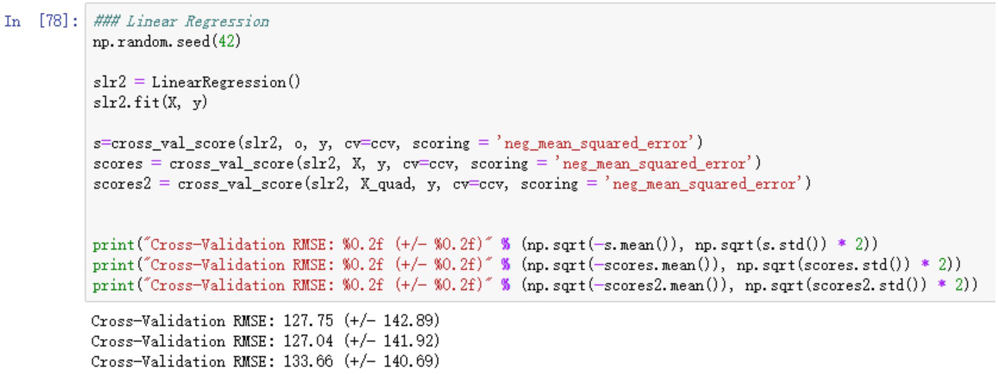
```
df1['period'] =  df['1st_update_days_ago'] - df['last_update_days_ago']
df1['Ratio'] = df1['Freq'] / (df1['period']+0.00001)
df1.head()
```

|   | 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 | sou |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **1** | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **2** | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | |
| **3** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **4** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

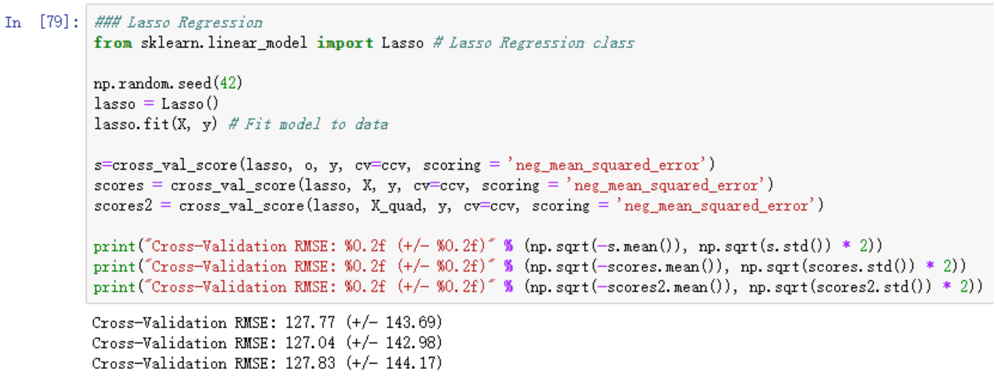
I first create a new variable ‘period’ to measure the difference between ‘1$^{st}$_update_days_ago’ and ‘lsat_update_days_ago’. Then, I create another variable ‘Ratio’ to get ‘Freq’ per day. Here, due to a large number of zeros in ‘period’, I add 0.00001 to ‘period’ to avoid inf values in ‘Ratio’.



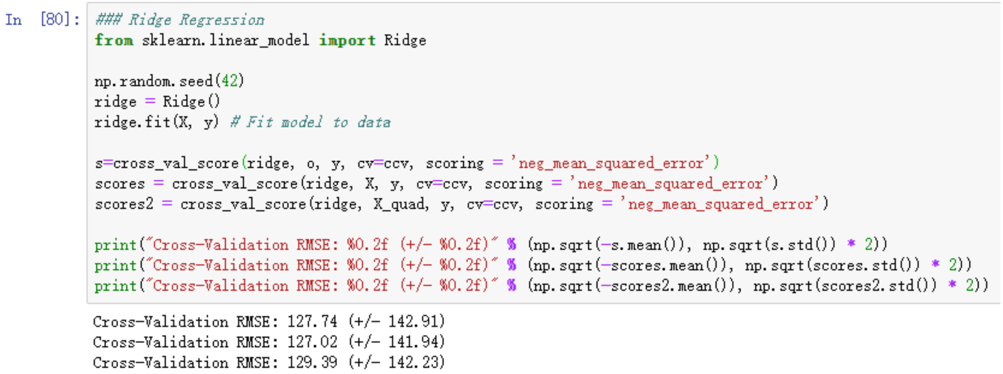
```
In [90]: from sklearn.preprocessing import PolynomialFeatures
         o= df.iloc[:, :-1].values
         X= df1.iloc[:,np.r_[0:22,-1,-2]]
         y= df1.iloc[:,-3]

         quadratic = PolynomialFeatures(degree=2)
         cubic = PolynomialFeatures(degree=3)
         X_quad = quadratic.fit_transform(X)


         print(X.shape)
         print(X_quad.shape)

         (2000, 24)
         (2000, 325)
```



```
In [78]: ### Linear Regression
         np.random.seed(42)

         slr2 = LinearRegression()
         slr2.fit(X, y)

         s=cross_val_score(slr2, o, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores = cross_val_score(slr2, X, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores2 = cross_val_score(slr2, X_quad, y, cv=ccv, scoring = 'neg_mean_squared_error')


         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-s.mean()), np.sqrt(s.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores2.mean()), np.sqrt(scores2.std()) * 2))

         Cross-Validation RMSE: 127.75 (+/- 142.89)
         Cross-Validation RMSE: 127.04 (+/- 141.92)
         Cross-Validation RMSE: 133.66 (+/- 140.69)
```



```
In [79]: ### Lasso Regression
         from sklearn.linear_model import Lasso # Lasso Regression class

         np.random.seed(42)
         lasso = Lasso()
         lasso.fit(X, y) # Fit model to data

         s=cross_val_score(lasso, o, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores = cross_val_score(lasso, X, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores2 = cross_val_score(lasso, X_quad, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-s.mean()), np.sqrt(s.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores2.mean()), np.sqrt(scores2.std()) * 2))

         Cross-Validation RMSE: 127.77 (+/- 143.69)
         Cross-Validation RMSE: 127.04 (+/- 142.98)
         Cross-Validation RMSE: 127.83 (+/- 144.17)
```



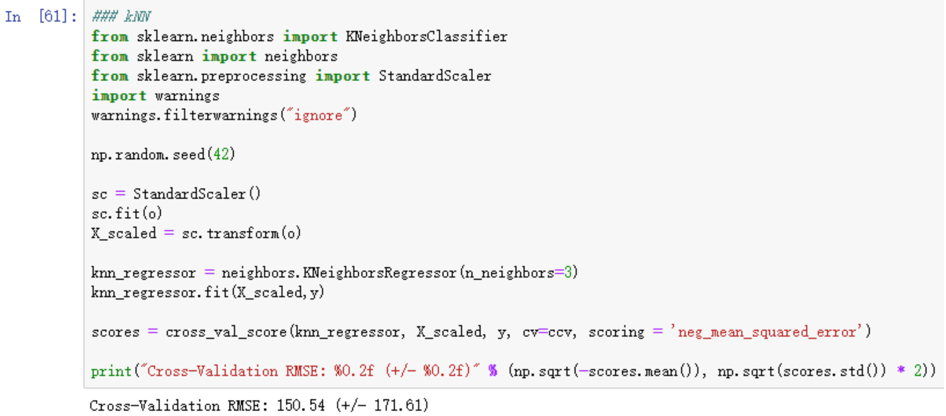
```
In [80]: ### Ridge Regression
         from sklearn.linear_model import Ridge

         np.random.seed(42)
         ridge = Ridge()
         ridge.fit(X, y) # Fit model to data

         s=cross_val_score(ridge, o, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores = cross_val_score(ridge, X, y, cv=ccv, scoring = 'neg_mean_squared_error')
         scores2 = cross_val_score(ridge, X_quad, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-s.mean()), np.sqrt(s.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))
         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores2.mean()), np.sqrt(scores2.std()) * 2))

         Cross-Validation RMSE: 127.74 (+/- 142.91)
         Cross-Validation RMSE: 127.02 (+/- 141.94)
         Cross-Validation RMSE: 129.39 (+/- 142.23)
```

For each model, it reports RMSE for original dataset, RMSE with 2 new variables ('period', 'Ratio'), and RMSE after quadratic transformation. Their performance evaluated by 10-fold cross-validation are:

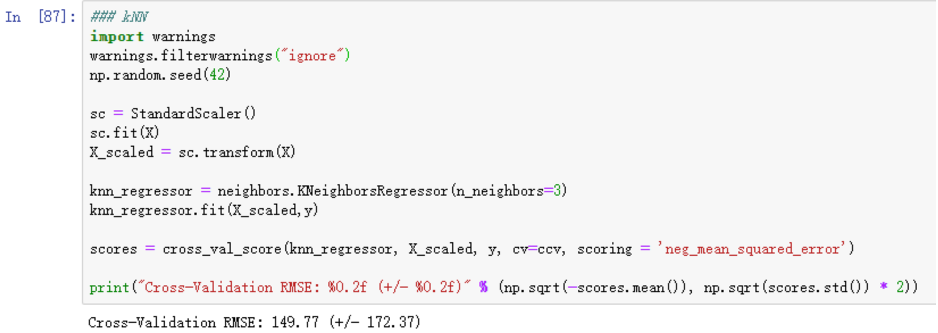
| | Original RMSE | RMSE with 2 new variables | RMSE with quadratic transformation |
|---|---|---|---|
| Simple Linear | 127.75 | 127.04 | 133.66 |
| Lasso Linear | 127.64 | 127.04 | 127.83 |
| Ridge Linear | 127.74 | 127.02 | 129.39 |

The best performance happens when using Ridge Regression and dataset with 2 new variables. The best performing RMSE is 127.02, better than the original RMSE.

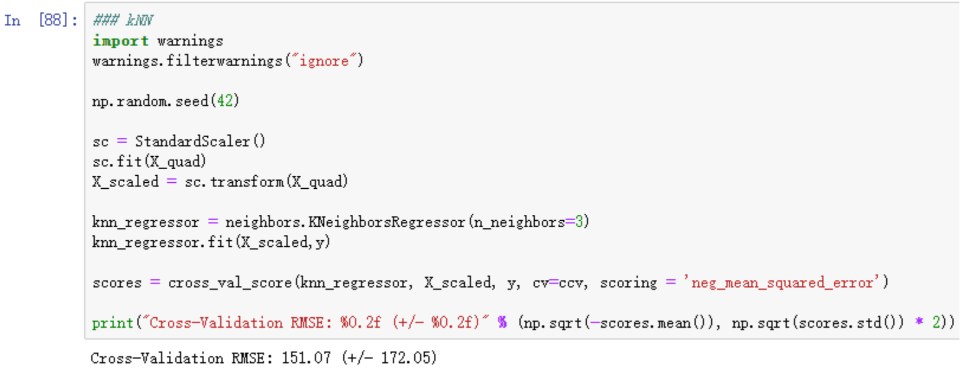
10 points for correctly building the new k-NN model and improving the performance as much as possible - provide screenshots and explain what you are doing and the corresponding results



```
In [61]: ### kNN
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn import neighbors
         from sklearn.preprocessing import StandardScaler
         import warnings
         warnings.filterwarnings("ignore")

         np.random.seed(42)

         sc = StandardScaler()
         sc.fit(o)
         X_scaled = sc.transform(o)

         knn_regressor = neighbors.KNeighborsRegressor(n_neighbors=3)
         knn_regressor.fit(X_scaled,y)

         scores = cross_val_score(knn_regressor, X_scaled, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 150.54 (+/- 171.61)
```



```
In [87]: ### kNN
         import warnings
         warnings.filterwarnings("ignore")
         np.random.seed(42)

         sc = StandardScaler()
         sc.fit(X)
         X_scaled = sc.transform(X)

         knn_regressor = neighbors.KNeighborsRegressor(n_neighbors=3)
         knn_regressor.fit(X_scaled,y)

         scores = cross_val_score(knn_regressor, X_scaled, y, cv=ccv, scoring = 'neg_mean_squared_error')

         print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

         Cross-Validation RMSE: 149.77 (+/- 172.37)
```

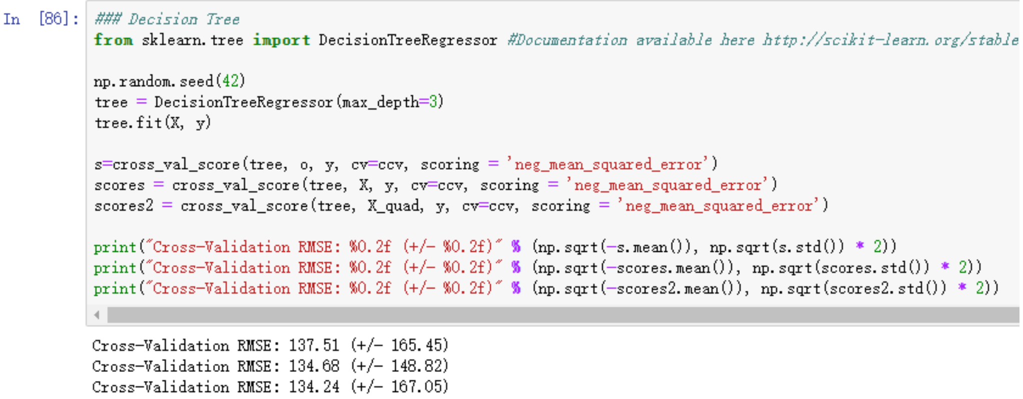


```
In  [88]: ### kNN
          import warnings
          warnings.filterwarnings("ignore")

          np.random.seed(42)

          sc = StandardScaler()
          sc.fit(X_quad)
          X_scaled = sc.transform(X_quad)

          knn_regressor = neighbors.KNeighborsRegressor(n_neighbors=3)
          knn_regressor.fit(X_scaled,y)

          scores = cross_val_score(knn_regressor, X_scaled, y, cv=ccv, scoring = 'neg_mean_squared_error')

          print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))

          Cross-Validation RMSE: 151.07 (+/- 172.05)
```

Within all three k-NN models (original, original+2 new variables, original+2 new variables+quadratic transformed), the best performing model is the one with 2 new variables but not transformed quadratically. The best 10-fold cross-validation RMSE is 149.77, better than the original RMSE.

| | **Original RMSE** | **RMSE with 2 new variables** | **RMSE with quadratic transformation** |
|---|---|---|---|
| k-NN | 150.54 | 149.77 | 151.07 |

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



```
In  [86]: ### Decision Tree
          from sklearn.tree import DecisionTreeRegressor #Documentation available here http://scikit-learn.org/stable

          np.random.seed(42)
          tree = DecisionTreeRegressor(max_depth=3)
          tree.fit(X, y)

          s=cross_val_score(tree, o, y, cv=ccv, scoring = 'neg_mean_squared_error')
          scores = cross_val_score(tree, X, y, cv=ccv, scoring = 'neg_mean_squared_error')
          scores2 = cross_val_score(tree, X_quad, y, cv=ccv, scoring = 'neg_mean_squared_error')

          print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-s.mean()), np.sqrt(s.std()) * 2))
          print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores.mean()), np.sqrt(scores.std()) * 2))
          print("Cross-Validation RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-scores2.mean()), np.sqrt(scores2.std()) * 2))

          Cross-Validation RMSE: 137.51 (+/- 165.45)
          Cross-Validation RMSE: 134.68 (+/- 148.82)
          Cross-Validation RMSE: 134.24 (+/- 167.05)
```

Within all three regression tree models (original, original+2 new variables, original+2 new variables+quadratic transformed), the best performing model is the one with 2 new variables and transformed quadratically. The best 10-fold cross-validation RMSE is 134.24, better than the original RMSE.

| | **Original RMSE** | **RMSE with 2 new variables** | **RMSE with quadratic transformation** |
|---|---|---|---|
| Regression Tree | 137.51 | 134.68 | 134.24 |

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]

| | Original RMSE | RMSE with 2 new variables | RMSE with quadratic transformation |
|---|---|---|---|
| Simple Linear | 127.75 | 127.04 | 133.66 |
| Lasso Linear | 127.64 | 127.04 | 127.83 |
| Ridge Linear | 127.74 | 127.02 | 129.39 |
| k-NN | 150.54 | 149.77 | 151.07 |
| Regression Tree | 137.51 | 134.68 | 134.24 |

RMSE with 2 new variables (‘period’, ‘Ratio’):

Under 10-fold cross-validation, all models have improvement in RMSE, although most of the improvements are limited (less than 1). For regression tree, the improvement is bigger (almost 3). This is because the 2 new variables capture more information from the original 3 variables(‘Freq’, ‘1st_update_days_ago’, ‘last_update_days_ago’). This increase in information helps models to predict more accurately.

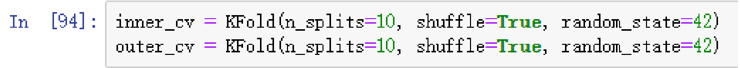
RMSE with 2 new variables and quadratic transformation:

Under 10-fold cross-validation, most models have worse performance than original RMSE. This is probably because quadratic transformation generates redundant variables that affects the accuracy and robustness of the original model. However, for regression tree, RMSE with 2 new variables and quadratic transformation performs better than both original RMSE and RMSE with 2 new variables but without quadratic transformation. This is probably because that after quadratic transformation, the new model has more variables and information to perform more partition of the data to predict more accurately.

Overall, the best performing model is Ridge Regression with 2 new variables, yielding an RMSE of 127.02.

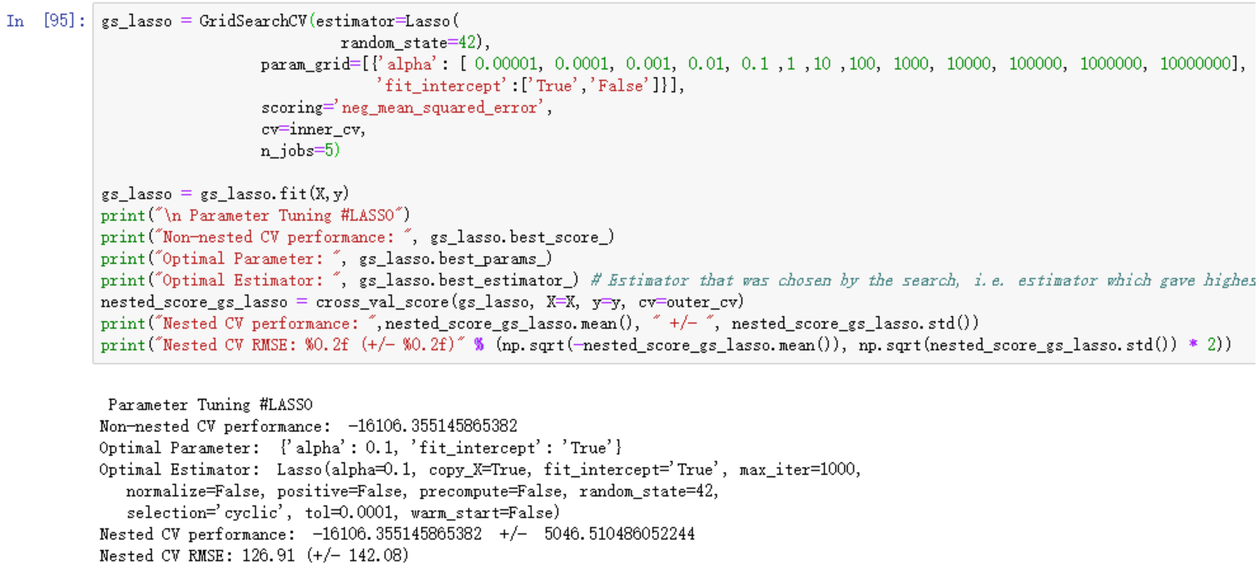
**(c) (35 points) Engage in parameter tuning to optimize the performance of linear regression, k-NN, and regression tree techniques. Use cross-validations with 10 folds to estimate the generalization performance. Present the results for each of the three techniques and discuss which one yields the best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



```
In [94]: inner_cv = KFold(n_splits=10, shuffle=True, random_state=42)
         outer_cv = KFold(n_splits=10, shuffle=True, random_state=42)
```

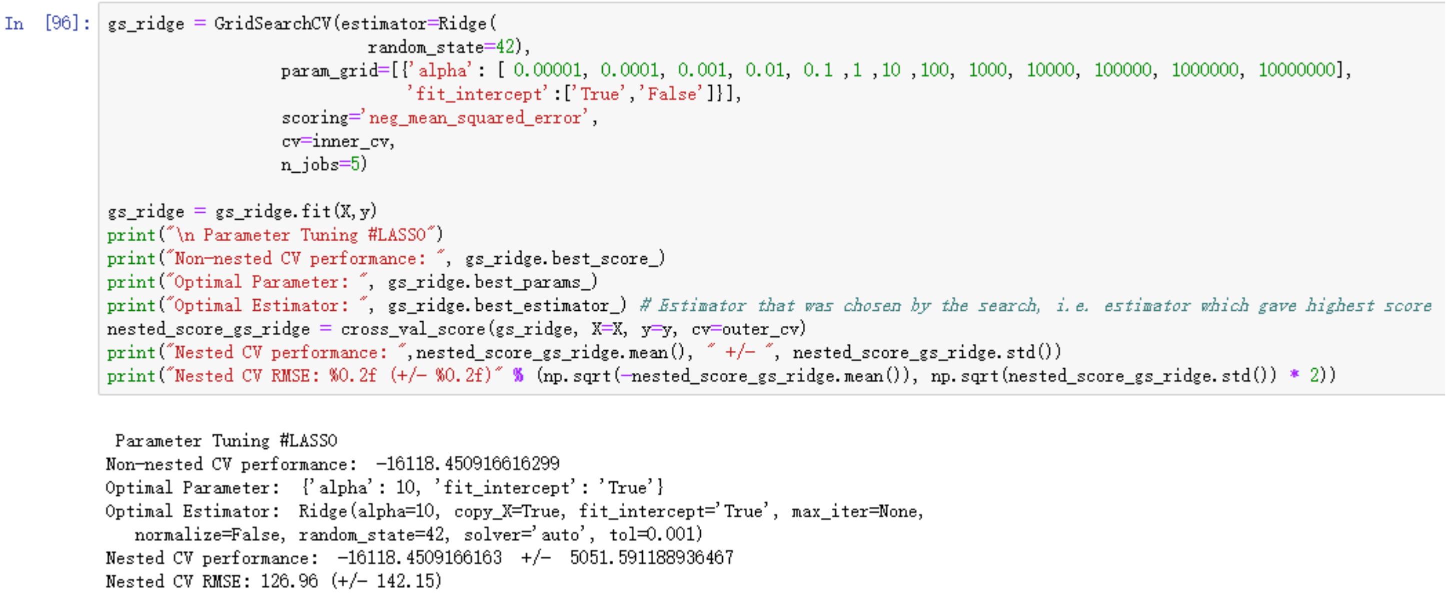
Specify random_state = 42 for both inner_cv and outer_cv.
For all models, I use original dataset + 2 new variables as X. For the k-NN model, I then standardize X into X_std.



```
In [95]: gs_lasso = GridSearchCV(estimator=Lasso(
                                  random_state=42),
                      param_grid=[{'alpha': [ 0.00001, 0.0001, 0.001, 0.01, 0.1 ,1 ,10 ,100, 1000, 10000, 100000, 1000000, 10000000],
                                   'fit_intercept':['True','False']}],
                      scoring='neg_mean_squared_error',
                      cv=inner_cv,
                      n_jobs=5)

         gs_lasso = gs_lasso.fit(X,y)
         print("\n Parameter Tuning #LASSO")
         print("Non-nested CV performance: ", gs_lasso.best_score_)
         print("Optimal Parameter: ", gs_lasso.best_params_)
         print("Optimal Estimator: ", gs_lasso.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highes
         nested_score_gs_lasso = cross_val_score(gs_lasso, X=X, y=y, cv=outer_cv)
         print("Nested CV performance: ",nested_score_gs_lasso.mean(), " +/- ", nested_score_gs_lasso.std())
         print("Nested CV RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-nested_score_gs_lasso.mean()), np.sqrt(nested_score_gs_lasso.std()) * 2))
```

```
 Parameter Tuning #LASSO
Non-nested CV performance:  -16106.355145865382
Optimal Parameter:  {'alpha': 0.1, 'fit_intercept': 'True'}
Optimal Estimator:  Lasso(alpha=0.1, copy_X=True, fit_intercept='True', max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=42,
   selection='cyclic', tol=0.0001, warm_start=False)
Nested CV performance:  -16106.355145865382  +/-  5046.510486052244
Nested CV RMSE: 126.91 (+/- 142.08)
```

For the Lasso Model, optimal parameters are alpha = 0.1, fit_intercept = 'True'. The 10-fold cross-validation performance is RMSE = 126.91.

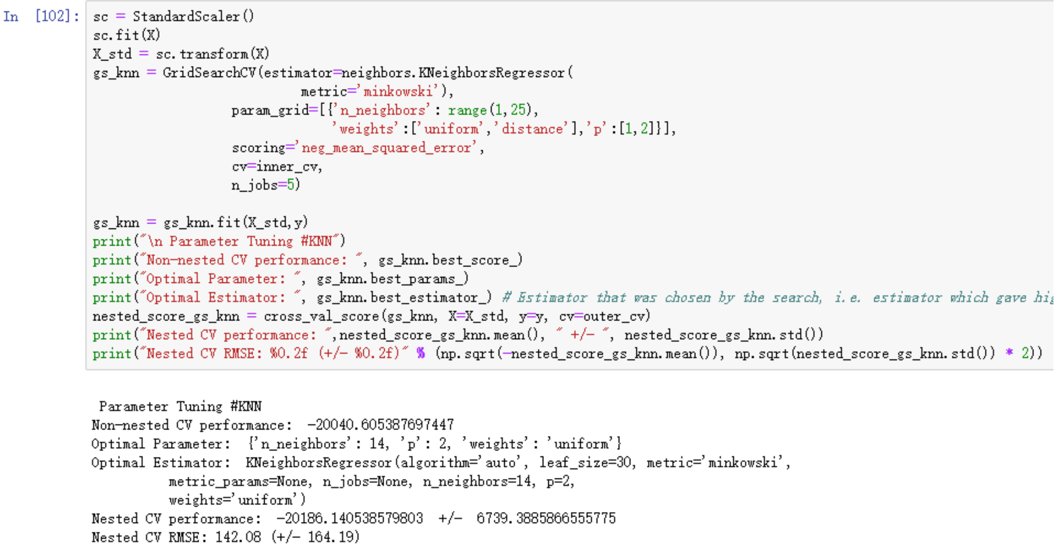


```
In [96]: gs_ridge = GridSearchCV(estimator=Ridge(
                                  random_state=42),
                      param_grid=[{'alpha': [ 0.00001, 0.0001, 0.001, 0.01, 0.1 ,1 ,10 ,100, 1000, 10000, 100000, 1000000, 10000000],
                                   'fit_intercept':['True','False']}],
                      scoring='neg_mean_squared_error',
                      cv=inner_cv,
                      n_jobs=5)

         gs_ridge = gs_ridge.fit(X,y)
         print("\n Parameter Tuning #LASSO")
         print("Non-nested CV performance: ", gs_ridge.best_score_)
         print("Optimal Parameter: ", gs_ridge.best_params_)
         print("Optimal Estimator: ", gs_ridge.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest score
         nested_score_gs_ridge = cross_val_score(gs_ridge, X=X, y=y, cv=outer_cv)
         print("Nested CV performance: ",nested_score_gs_ridge.mean(), " +/- ", nested_score_gs_ridge.std())
         print("Nested CV RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-nested_score_gs_ridge.mean()), np.sqrt(nested_score_gs_ridge.std()) * 2))
```

```
 Parameter Tuning #LASSO
Non-nested CV performance:  -16118.450916616299
Optimal Parameter:  {'alpha': 10, 'fit_intercept': 'True'}
Optimal Estimator:  Ridge(alpha=10, copy_X=True, fit_intercept='True', max_iter=None,
   normalize=False, random_state=42, solver='auto', tol=0.001)
Nested CV performance:  -16118.4509166163  +/-  5051.591188936467
Nested CV RMSE: 126.96 (+/- 142.15)
```

For the Ridge Model, optimal parameters are alpha = 10, fit_intercept = 'True'. The 10-fold cross-validation performance is RMSE = 126.96.

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

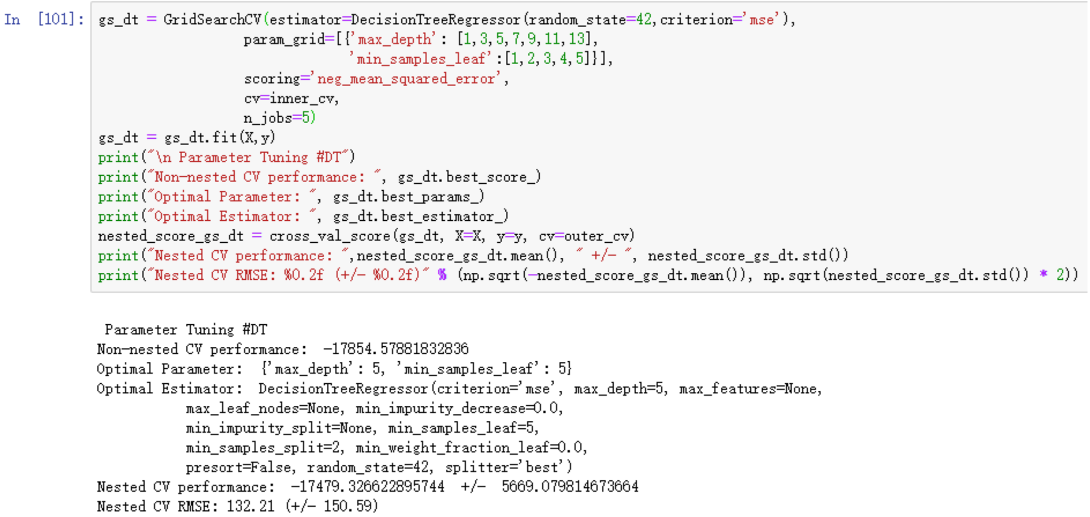


```
In [102]: sc = StandardScaler()
          sc.fit(X)
          X_std = sc.transform(X)
          gs_knn = GridSearchCV(estimator=neighbors.KNeighborsRegressor(
                                    metric='minkowski'),
                          param_grid=[{'n_neighbors': range(1,25),
                                       'weights':['uniform','distance'],'p':[1,2]}],
                          scoring='neg_mean_squared_error',
                          cv=inner_cv,
                          n_jobs=5)

          gs_knn = gs_knn.fit(X_std,y)
          print("\n Parameter Tuning #KNN")
          print("Non-nested CV performance: ", gs_knn.best_score_)
          print("Optimal Parameter: ", gs_knn.best_params_)
          print("Optimal Estimator: ", gs_knn.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave hi
          nested_score_gs_knn = cross_val_score(gs_knn, X=X_std, y=y, cv=outer_cv)
          print("Nested CV performance: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
          print("Nested CV RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-nested_score_gs_knn.mean()), np.sqrt(nested_score_gs_knn.std()) * 2))
```

```
 Parameter Tuning #KNN
Non-nested CV performance:  -20040.605387697447
Optimal Parameter:  {'n_neighbors': 14, 'p': 2, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=14, p=2,
          weights='uniform')
Nested CV performance:  -20186.140538579803  +/-  6739.3885866555775
Nested CV RMSE: 142.08 (+/- 164.19)
```

For the k-NN Model, optimal parameters are n_neighbors = 14, p = 2, weights = ‘uniform’. The 10-fold cross-validation is RMSE = 142.08.

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



```
In [101]: gs_dt = GridSearchCV(estimator=DecisionTreeRegressor(random_state=42,criterion='mse'),
                            param_grid=[{'max_depth': [1,3,5,7,9,11,13],
                                         'min_samples_leaf':[1,2,3,4,5]}],
                            scoring='neg_mean_squared_error',
                            cv=inner_cv,
                            n_jobs=5)
          gs_dt = gs_dt.fit(X,y)
          print("\n Parameter Tuning #DT")
          print("Non-nested CV performance: ", gs_dt.best_score_)
          print("Optimal Parameter: ", gs_dt.best_params_)
          print("Optimal Estimator: ", gs_dt.best_estimator_)
          nested_score_gs_dt = cross_val_score(gs_dt, X=X, y=y, cv=outer_cv)
          print("Nested CV performance: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
          print("Nested CV RMSE: %0.2f (+/- %0.2f)" % (np.sqrt(-nested_score_gs_dt.mean()), np.sqrt(nested_score_gs_dt.std()) * 2))
```

```
 Parameter Tuning #DT
Non-nested CV performance:  -17854.57881832836
Optimal Parameter:  {'max_depth': 5, 'min_samples_leaf': 5}
Optimal Estimator:  DecisionTreeRegressor(criterion='mse', max_depth=5, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=5,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=42, splitter='best')
Nested CV performance:  -17479.326622895744  +/-  5669.079814673664
Nested CV RMSE: 132.21 (+/- 150.59)
```

For the regression tree model, optimal parameters are max_depth = 5, min_samples_leaf = 5. The 10-fold cross-validation is RMSE = 132.21.

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

| | Original RMSE | RMSE with 2 new variables | RMSE with quadratic transformation | RMSE after parameter tuning |
|---|---|---|---|---|
| Simple Linear | 127.75 | 127.04 | 133.66 | - |
| Lasso Linear | 127.64 | 127.04 | 127.83 | 126.91 |
| Ridge Linear | 127.74 | 127.02 | 129.39 | 126.96 |
| k-NN | 150.54 | 149.77 | 151.07 | 142.08 |
| Regression Tree | 137.51 | 134.68 | 134.24 | 132.21 |

Overall, linear regression models perform much better than k-NN model and regression tree model. One possible reason is the dataset is relatively small with only 2000 instances. Within linear regression models, the difference is minimal among linear regression without penalty, lasso regression, and ridge regression. Ridge regression wins by only 0.05.