Homework #4

Carl Xi

(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]

Key component of most measures is difference between actual outcome and predicted outcome (i.e., error)

- RMSE: Root Mean Squared Error
- MAE: Mean Absolute Error
- MAPE: Mean Absolute Percentage Error
- Total SSE: Total Sum of Squared Errors
- Average error
  - Gives an idea of systematic over- or under-prediction

- For each record i $(i = 1, \ldots, n)$ in validation dataset:
  - a_i – actual value of the outcome variable (from data)
  - p_i – predicted value of the outcome variable (from model)
- Error for data record i: $e_i = \hat{y} - y = p_i - a_i$
- Mean absolute error (**MAE**) = $\frac{\sum_{i=1}^{n} |e_i|}{n}$
- Mean absolute % error (**MAPE**) = $\frac{\sum_{i=1}^{n} |e_i/a_i|}{n}$
- Root mean squared error (**RMSE**) = $\sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n}}$
- Average error = $\frac{\sum_{i=1}^{n} e_i}{n}$ (not very informative)

$$MAE = \frac{|(710-670)|+|(660-680)|+|(600-550)|+|(800-740)|+|(600-700)|}{5} = 54$$

$$MAPE = \frac{\left|\frac{(710-670)}{670}\right|+\left|\frac{(660-680)}{680}\right|+\left|\frac{(600-550)}{550}\right|+\left|\frac{(800-740)}{740}\right|+\left|\frac{(600-700)}{700}\right|}{5} = 0.08079 = 8.0792\%$$

$$\text{RMSE} = \sqrt{\frac{(710-670)^2+(660-680)^2+(600-550)^2+(800-740)^2+(600-700)^2}{5}} = 60.1664$$

$$Average\ Error = \frac{(710-670)+(660-680)+(600-550)+(800-740)+(600-700)}{5} = 6$$

abs((710-670))+abs((660-680))+abs((600-550))+abs((800-740))+abs((600-700))
abs((710-670)/670)+abs((660-680)/680)+abs((600-550)/550)+abs((800-740)/740)+abs((600-700)/700)

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

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

[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]

As usual, this homework follows the standard homework/model building procedure. First, we imported all the necessary libraries, set the root directory and set the random state to 42. We will also be using a new library for stepwise feature selection so please make sure you install the library indicated on line 3 of the screenshot.

In [2]:

```
1  ##################Libraries, Data Import, Cleaning, Exploration##########################
2
3  # MAKE SURE YOU RUN "pip install mlxtend --user" in your anaconda prompt or python prompt!!!!!!!!!!!
4
5  # To write a Python 2/3 compatible codebase, the first step is to add this line to the top of each module
6  from __future__ import division, print_function, unicode_literals
7  from IPython.display import Image
8  from sklearn import linear_model, neighbors, datasets, metrics, tree, preprocessing, utils        # The sklearn.linear_mode
9  from sklearn.linear_model import LogisticRegression, LinearRegression, Lasso, Ridge
10 from matplotlib.colors import ListedColormap # Learn more about matplotlib.colors here https://matplotlib.org/3.1.1/api/col
11 import matplotlib.pyplot as plt #pyplot is matplotlib's plotting framework https://matplotlib.org/users/pyplot_tutorial.htm
12 import matplotlib
13 import scipy as sp # sp is an alias pointing to scipy
14 import numpy as np
15 import pandas as pd # pd is an alias point to pandas
16 import warnings
17 warnings.filterwarnings("ignore")
18 from sklearn.model_selection import train_test_split, cross_val_score, validation_curve, GridSearchCV, KFold, StratifiedKFo
19 from sklearn.preprocessing import PolynomialFeatures, StandardScaler, LabelEncoder, MinMaxScaler
20 # StandardScaler Standardize features by removing the mean and scaling to unit variance
21 # LabelEncoder Encode labels with value between 0 and n_classes-1
22 # Cross_val_score Evaluate a score by cross-validation
23
24 # The sklearn.metrics module includes score functions, performance metrics and pairwise metrics
25 # and distance computations.
26 # https://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics
27 from sklearn.metrics import mean_absolute_error, mean_squared_error
28 from sklearn.metrics import confusion_matrix, classification_report, accuracy_score, f1_score, cohen_kappa_score,  roc_curv
29 from sklearn.tree import export_graphviz, DecisionTreeRegressor
30 from sklearn.neighbors import KNeighborsClassifier
31 # If you don't have graphviz package, you need to install it https://anaconda.org/anaconda/graphviz
32 # How to install Graphviz with Anaconda 1
33 # conda install -c anaconda graphviz
34 !pip install graphviz
35 import itertools
36 import graphviz
37 import os
38 #---------------------
39 # Seed the generator to make this notebook's output stable across runs
40 np.random.seed(42)
41
42 # To plot pretty figures
43 %matplotlib inline
44
45 # Dynamically change the default rc settings in a python script
46 # See documentation for a complete list of parameters https://matplotlib.org/users/customizing.html
47 plt.rcParams['axes.labelsize'] = 14  # fontsize of the x any y labels
48 plt.rcParams['xtick.labelsize'] = 12 # fontsize of the tick labels
49 plt.rcParams['ytick.labelsize'] = 12 # fontsize of the tick labels
50 #-----------------
51 # Matplotlib inline allows the output of plotting commands will be displayed inline
52 %matplotlib inline
53 # Root Directory
54 PROJECT_ROOT_DIR = "C:\\Users\\carlj\\OneDrive\\Documents\\School-MSBA\\Classes\\Fall\\Intro. to Business Analytics\\HW4"
```

executed in 1.43s, finished 13:59:17 2019-10-08

```
Requirement already satisfied: graphviz in c:\programdata\anaconda3\lib\site-packages (0.13)
```

I then imported the data and looked at the dataset. There are 25 columns and 2000 rows. I looked at the standard deviation, min, max, etc. of my variables. Many columns are binary variables.

1.2 Data import

In [3]:
```
######################################## Imports ########################################
hw4data = pd.read_excel("HW4.xlsx")#, header=None, names=["id","diagnosis","radius_mean","radius_stderror","radius_worst","
#hw2data.rename(columns={0:"id",1:"diagnosis"},inplace=True)
```
executed in 644ms, finished 13:59:19 2019-10-08

1.3 Pre-Clean Data Exploration

In [4]:
```
#Looking at how big our dataset is
hw4data.shape
```
executed in 12ms, finished 13:59:21 2019-10-08

Out[4]: (2000, 25)

In [5]:
```
#taking a peek at how the data looks like
hw4data.head()
```
executed in 22ms, finished 13:59:22 2019-10-08

Out[5]:

	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_day
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

In [6]:
```
#Looking at any potential problems/outliers
hw4data.describe()
```
executed in 61ms, finished 13:59:23 2019-10-08

Out[6]:

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...	source
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	...	2000.0000
mean	1000.500000	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	...	0.0180
std	577.494589	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	...	0.1329
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0000
25%	500.750000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0000
50%	1000.500000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0000
75%	1500.250000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	...	0.0000
max	2000.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	...	1.0000

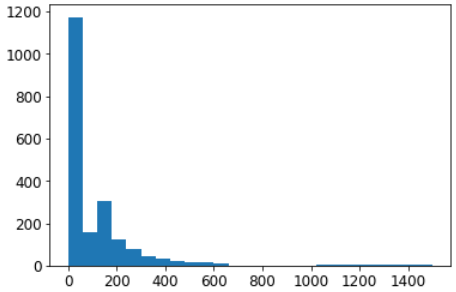
8 rows × 25 columns

I made a histogram of the target variable (spending in our case) to visualize its distribution. From the looks of it, our target variable is heavily skewed to the left.

1.4 Target Variable Distribution Histogram

In [106]:
```
matplotlib.pyplot.hist(hw4data.Spending, bins=25, range=None, density=None, weights=None)
```
executed in 149ms, finished 16:42:59 2019-10-08

Out[106]:
```
(array([1.174e+03, 1.590e+02, 3.030e+02, 1.210e+02, 8.000e+01, 4.400e+01,
        3.400e+01, 1.900e+01, 1.400e+01, 1.400e+01, 8.000e+00, 0.000e+00,
        0.000e+00, 1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 4.000e+00,
        2.000e+00, 6.000e+00, 5.000e+00, 3.000e+00, 3.000e+00, 2.000e+00,
        4.000e+00]),
 array([   0.    ,   60.0024,  120.0048,  180.0072,  240.0096,  300.012 ,
         360.0144,  420.0168,  480.0192,  540.0216,  600.024 ,  660.0264,
         720.0288,  780.0312,  840.0336,  900.036 ,  960.0384, 1020.0408,
        1080.0432, 1140.0456, 1200.048 , 1260.0504, 1320.0528, 1380.0552,
        1440.0576, 1500.06  ]),
 <a list of 25 Patch objects>)
```



I double checked to make sure that there are no NaN or null values in our dataset, which luckily there isn't. From looking at the descriptions of the variables, sequence number is just an index number, and **Purchase** is a leakage problem because we cannot predict if a person will purchase or not until the moment of purchase, which is also the moment we know how much they spent. Thus, these two variables will be removed from our dataset. Our post-clean dataset has 23 variables (including target variable) and still retains 2000 records.

Codelist				
Var. #	**Variable Name**	**Description**	**Variable Type**	**Code Description**
1.	US	Is it a US address?	binary	1: yes 0: no
2 - 16	Source_*	Source catalog for the record (15 possible sources)	binary	1: yes 0: no
17.	Freq.	Number of transactions in last year at source catalog	numeric	
18.	last_update_days_ago	How many days ago was last update to cust. record	numeric	
19.	1st_update_days_ago	How many days ago was 1st update to cust. record	numeric	
20.	Web_order	Customer placed at least 1 order via web	binary	1: yes 0: no
21.	Gender=mal	Customer is male	binary	1: yes 0: no
22.	Address_is_res	Address is a residence	binary	1: yes 0: no
23.	Purchase	Person made purchase in test mailing	binary	1: yes 0: no
24.	Spending	Amount spent by customer in test mailing ($)	numeric	

1.5 Data Cleaning

In [7]:
```
#data cleaning
print(hw4data.isnull().sum())
hw4data.drop(['Purchase','sequence_number'], inplace=True, axis=1)
```

executed in 12ms, finished 13:59:26 2019-10-08

```
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
```

1.6 Post-Clean Data Exploration

In [8]:
```
hw4data.shape
```

executed in 10ms, finished 13:59:29 2019-10-08

Out[8]: (2000, 23)

In [9]:
```
hw4data.head()
```

executed in 20ms, finished 13:59:30 2019-10-08

Out[9]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	...	source_p	source_x	source_w	Freq	last_update_da
0	1	0	0	1	0	0	0	0	0	0	...	0	0	0	2	
1	1	0	0	0	0	1	0	0	0	0	...	0	0	0	0	
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	2	
3	1	0	1	0	0	0	0	0	0	0	...	0	0	0	1	
4	1	0	1	0	0	0	0	0	0	0	...	0	0	0	1	

5 rows × 23 columns

Just to be sure, I checked the summary statistics of the variables of my cleaned dataset again. Nothing seems to be out of the ordinary. I then printed a correlation matrix of my cleaned variables to see if I can identify any notable correlations.

In [10]:
```
1 hw4data.describe()
```
executed in 56ms, finished 13:59:31 2019-10-08

Out[10]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	...	source_p	2(
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	2000.000000	...	2000.000000	
mean	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	0.068500	...	0.006000	
std	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	0.252665	...	0.077246	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
25%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
50%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
75%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	1.000000	...	1.000000	

8 rows × 23 columns

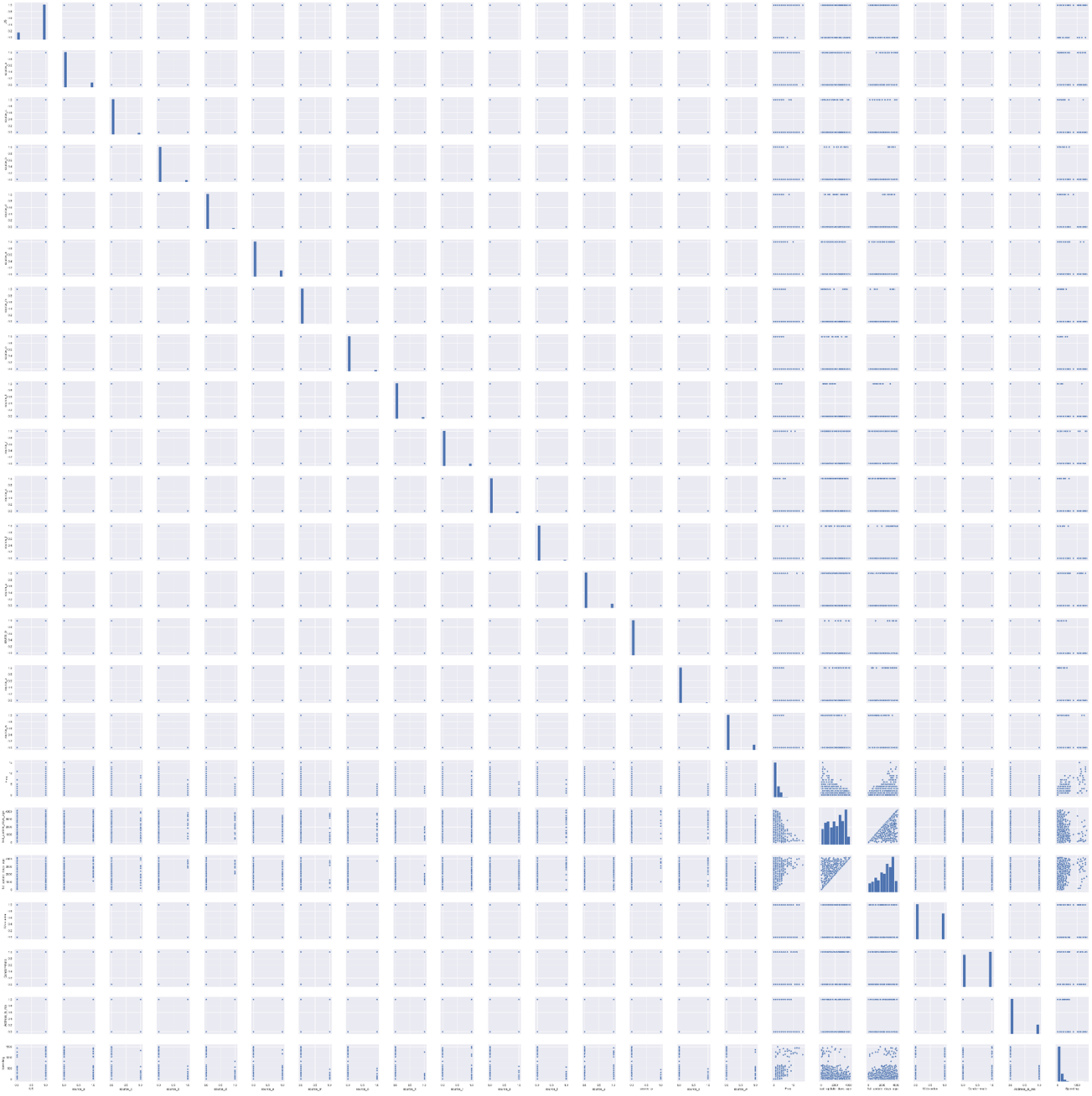
1.7 Correlation matrix

In [32]:
```
# Seaborn is a Python data visualization library based on matplotlib.
# Seaborn documentation can be found here https://seaborn.pydata.org/generated/seaborn.set.html
import seaborn as sns # sns is an alias pointing to seaborn
sns.set(color_codes=True) #Set aesthetic parameters in one step. Remaps the shorthand color codes (e.g. “b”, “g”, “r”, etc.,
from scipy import stats #Documentation stats package of scipy https://docs.scipy.org/doc/scipy/reference/stats.html#module-s

sns.pairplot(hw4data, size=2.5)         #df[cols]     # Plot pairwise relationships in a dataset
plt.tight_layout()                                    # Tight_layout automatically adjusts subplot params
                                                      # so that the subplot(s) fits in to the figure area.
# plt.savefig('housing_dataset.png', dpi=300) # Saves the figure in our local disk
plt.show()                                            # Display figure

```
executed in 57.5s, finished 17:38:27 2019-10-07

Unfortunately, there weren't any outstanding correlations or distributions.



I also printed a heatmap just to make sure I didn't miss anything from the correlation matrix.

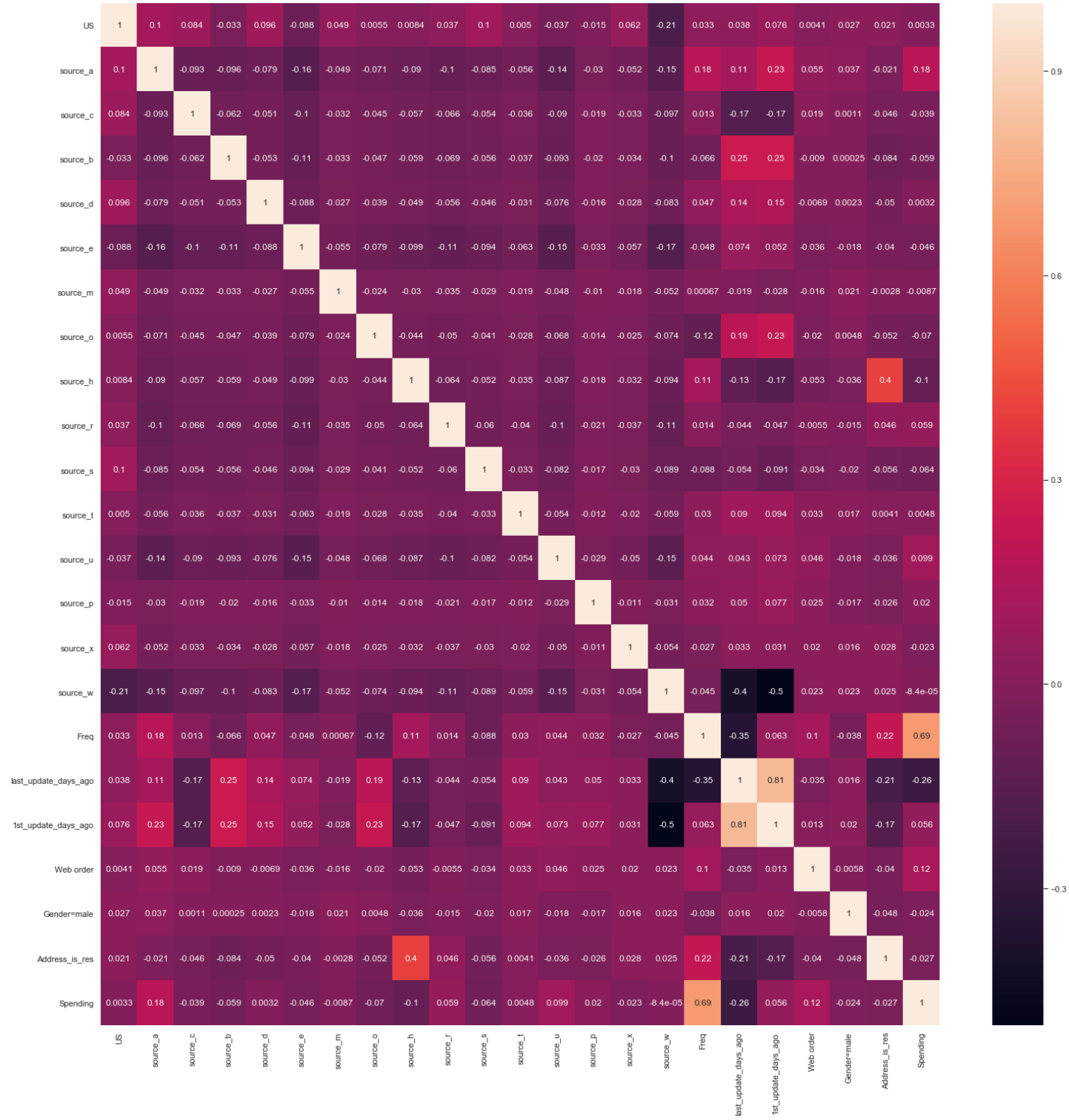
1.8 Heatmap

In [34]:
```
#plot heat map

corrmatrix = hw4data.corr()
top_correlated_features = corrmatrix.index
plt.figure(figsize=(25,25))
plot=sns.heatmap(hw4data[top_correlated_features].corr(),annot=True)
```

executed in 2.50s, finished 17:38:32 2019-10-07

It’s easier to see correlations on this heatmap, and I was able to find correlations as high as 0.81 positive or -0.5 negative.



After exploration, I isolated the target variable from the dataset. In the end, we have 1 target variable (y) and 22 predictor variables (X). I then created 3 10-fold cross validation splits, with 2 for parameter tuning/feature selection's inner and outer (non-nested/nested) cross validation, and the last one for our initial models for part a). I also created standardized versions of my X variable for knn. While I did create a train/test 70-30 split versions of my x and y, and also standardized versions of them, I **DID NOT USE THEM IN THE END, SINCE ALL OUR EVALUATIONS CALLS FOR NESTED CROSS VALIDATION.**

1.9 Isolating Target Variable & Train/Test Split

In [11]:
```
######################################## Isolating the Target Variable ########################################
# Retrieving Attributes
X = hw4data.iloc[:,:-1].values#or remove .values?
# Retriving Target Variable
y = hw4data.iloc[:,-1].values#or remove .values?
################################## Split the Data into 70% training and 30% test ###################################
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42)#, stratify=y)
```
executed in 8ms, finished 13:59:36 2019-10-08

In [12]:
```
print(X.shape)
print(y.shape)
```
executed in 5ms, finished 13:59:37 2019-10-08

```
(2000, 22)
(2000,)
```

In [13]:
```
#Creating 10-fold for model Crosss Validation
inner_cv = KFold(n_splits=10, shuffle=True, random_state=42)
outer_cv = KFold(n_splits=10, shuffle=True, random_state=42)
a_cv = KFold(n_splits=10, shuffle=True, random_state=42)
```
executed in 10ms, finished 13:59:37 2019-10-08

1.10 Standardizing Variables

In [14]:
```
################### Standardize Training and Testing X using metrics from Training X ############################
sc = StandardScaler()
sc.fit(X_train) # Compute the mean and std to be used for later scaling.
X_train_std = sc.transform(X_train) # Perform standardization of train set X by centering and scaling
X_test_std = sc.transform(X_test) # Perform standardization of test set Xby centering and scaling

sc = StandardScaler()
sc.fit(X)
X_std = sc.transform(X)
```
executed in 9ms, finished 13:59:41 2019-10-08

Here comes the fun part, modeling. I want to make a point that I have included a parameter for all models just for consistency AND because the instructions did not say that our models for part a) CANNOT have any parameters.

Our first model is a barebone simple linear regression. I will be discussing why I chose RMSE as my performance evaluation metric and how this model compares to other models below. Simple Linear Regression has a mean RMSE of 127.75.

2.1 Linear Regression Model

In [15]:
```
################################## Fit a Linear Regression Model (NTK) ##################################
linmodel = LinearRegression()    # Linear Regression class
linmodel =linmodel.fit(X, y)   # Fit Model to data

print()
cross_val_lin = cross_val_score(linmodel, X=X, y=y, cv=a_cv, scoring="neg_mean_squared_error") #or X=X_train and y=y_train?
print("CV MSE Performance: ", cross_val_lin.mean(), " +/- ", cross_val_lin.std())
print("CV RMSE Performance: ", np.sqrt(abs(cross_val_lin.mean())), " +/- ", (np.sqrt(abs(cross_val_lin.mean())+cross_val_li
cross_val_lin = cross_val_score(linmodel, X=X, y=y, cv=a_cv, scoring="neg_mean_absolute_error")
print("CV MAE Performance: ", cross_val_lin.mean(), " +/- ", cross_val_lin.std())
```
executed in 86ms, finished 13:59:47 2019-10-08

```
CV MSE Performance:  -16320.793058297533  +/-  5104.501643201018
CV RMSE Performance:  127.75285929597636  +/-  18.62095906474876
CV MAE Performance:  -76.9588899818126  +/-  6.563491294080922
```

Our second model is a linear regression with lasso. Since the function for lasso is independent from the traditional linear regression, I will be treating it as an individual model. Like with the linear regression model, I will be discussing why I chose RMSE as my performance evaluation metric and how this model compares to other models below. Lasso has a mean RMSE of 127.64.

2.2 Lasso Regression

In [16]:
```
######################## Regularized Linear Regression Model - Lasso (NTK) ########################
lasso = Lasso(alpha=0.1,random_state=42)   # Lasso
lasso = lasso.fit(X, y) # Fit model to data

print()
cross_val_las = cross_val_score(lasso, X=X, y=y, cv=a_cv, scoring="neg_mean_squared_error") #or X=X_train and y=y_train?
print("CV MSE Performance: ", cross_val_las.mean(), " +/- ", cross_val_las.std())
print("CV RMSE Performance: ", np.sqrt(abs(cross_val_las.mean())), " +/- ", (np.sqrt(abs(cross_val_las.mean())+cross_val_la
cross_val_las = cross_val_score(lasso, X=X, y=y, cv=a_cv, scoring="neg_mean_absolute_error")
print("CV MAE Performance: ", cross_val_las.mean(), " +/- ", cross_val_las.std())
```

executed in 142ms, finished 13:59:49 2019-10-08

```
CV MSE Performance:  -16292.410880330895  +/-  5112.325195466684
CV RMSE Performance:  127.64172860131163  +/-  18.66184646026224
CV MAE Performance:  -76.90887732834692  +/-  6.516866312190457
```

Our third model is a linear regression with ridge. Since the function for ridge is independent from the traditional linear regression, I will be treating it as an individual model. Like with the linear regression model, I will be discussing why I chose RMSE as my performance evaluation metric and how this model compares to other models below. Ridge has a mean RMSE of 127.64.

2.3 Ridge Regression

In [17]:
```
######################## Regularized Linear Regression Model - Ridge (NTK) ########################

ridge = Ridge(alpha=1.0,random_state=42) # Regularization strength; must be a positive float.
                        # Larger values specify stronger regularization.
                        # Alpha corresponds to C^-1 in other linear models such as LogisticRegression
ridge = ridge.fit(X, y) # Fit Model

print()
cross_val_rid = cross_val_score(ridge, X=X, y=y, cv=a_cv, scoring="neg_mean_squared_error") #or X=X_train and y=y_train?
print("CV MSE Performance: ", cross_val_rid.mean(), " +/- ", cross_val_rid.std())
print("CV RMSE Performance: ", np.sqrt(abs(cross_val_rid.mean())), " +/- ", (np.sqrt(abs(cross_val_rid.mean())+cross_val_ri
cross_val_rid = cross_val_score(ridge, X=X, y=y, cv=a_cv, scoring="neg_mean_absolute_error")
print("CV MAE Performance: ", cross_val_rid.mean(), " +/- ", cross_val_rid.std())


#Irrelevant features become very small but not 0
```

executed in 62ms, finished 13:59:50 2019-10-08

```
CV MSE Performance:  -16316.402356048007  +/-  5105.889487197948
CV RMSE Performance:  127.73567378006821  +/-  18.627886723377983
CV MAE Performance:  -76.96954012114239  +/-  6.559151935463092
```

Our fourth model is a regressor tree. Like with the linear regression model, I will be discussing why I chose RMSE as my performance evaluation metric and how this model compares to other models below. Regressor tree has a mean RMSE of 137.50.

2.4 Regressor Tree

In [18]:
```
1  ################################### Regressor Tree - Numeric Prediction (NTK) ###################################
2
3  # Decision Tree Regressor
4  # Supported criteria are "mse" for the mean squared error, which is equal to variance
5  # reduction as feature selection criterion and minimizes the L2 loss using the mean of
6  # each terminal node, "friedman_mse", which uses mean squared error with Friedman's
7  # improvement score for potential splits, and "mae" for the mean absolute error,
8  # which minimizes the L1 loss using the median of each terminal node.
9  tree = DecisionTreeRegressor(max_depth=3,random_state=42)
10 tree = tree.fit(X, y)
11
12 print()
13 cross_val_tree = cross_val_score(tree, X=X, y=y, cv=a_cv, scoring="neg_mean_squared_error") #or X=X_train and y=y_train?
14 print("CV MSE Performance: ", cross_val_tree.mean(), " +/- ", cross_val_tree.std())
15 print("CV RMSE Performance: ", np.sqrt(abs(cross_val_tree.mean())), " +/- ", (np.sqrt(abs(cross_val_tree.mean())+cross_val_
16 cross_val_tree = cross_val_score(tree, X=X, y=y, cv=a_cv, scoring="neg_mean_absolute_error")
17 print("CV MAE Performance: ", cross_val_tree.mean(), " +/- ", cross_val_tree.std())
```

executed in 64ms, finished 13:59:53 2019-10-08

```
CV MSE Performance:  -18907.985804927077  +/-  6843.38022165636
CV RMSE Performance:  137.50631187304487  +/-  22.96600981073783
CV MAE Performance:  -79.8288810488542  +/-  9.224992174469058
```

We need to be careful about choosing an appropriate value for the depth of the tree to not overfit or underfit the data.

Our fifth and last model is a regressor tree. Like with the linear regression model, I will be discussing why I chose RMSE as my performance evaluation metric and how this model compares to other models below. Regressor tree has a mean RMSE of 150.53.

2.5 kNN Regressor

In [19]:
```
1  ################################### kNN Regressor Example (NTK) ###################################
2  #3NN regressor
3  knn_regressor = neighbors.KNeighborsRegressor(n_neighbors = 3)
4  knn_regressor = knn_regressor.fit(X_std, y)  #fit the model
5
6  print()
7  cross_val_knn = cross_val_score(knn_regressor, X=X_std, y=y, cv=a_cv, scoring="neg_mean_squared_error") #or X=X_train and y=
8  print("CV MSE Performance: ", cross_val_knn.mean(), " +/- ", cross_val_knn.std())
9  print("CV RMSE Performance: ", np.sqrt(abs(cross_val_knn.mean())), " +/- ", (np.sqrt(abs(cross_val_knn.mean())+cross_val_kn
10 cross_val_knn = cross_val_score(knn_regressor, X=X_std, y=y, cv=a_cv, scoring="neg_mean_absolute_error")
11 print("CV MAE Performance: ", cross_val_knn.mean(), " +/- ", cross_val_knn.std())
```

executed in 278ms, finished 13:59:57 2019-10-08

```
CV MSE Performance:  -22661.044630683333  +/-  7362.640473913422
CV RMSE Performance:  150.53585828859292  +/-  22.737581986011065
CV MAE Performance:  -80.45898166666666  +/-  11.197520258780639
```

Among all measures of the difference between actual and predicted outcome, RMSE (Root Mean Squared Error) is the best in our case. It is calculated using the formula below:

$$\text{Root mean squared error (RMSE)} = \sqrt{\frac{\sum_{i=1}^{n} e_i^2}{n}}$$

I chose to use mean RMSE as my model evaluation metric because intuitively, RMSE represents how much the predicted values diverge from the actual values on average. Comparing and choosing the smallest RMSE model means it is also the model that is the best at predicting the data, as its predictions are on average the closest to the actual data points.

RMSE is defined as the standard deviation of the residuals (prediction errors). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are. In other words, it tells you how concentrated the data is around the line of best fit.

All in all, in our case where we are trying to predict a numeric variable with regression, RMSE is the best metric. Here is a summary table of the performances of all three models from above.

	Linear Regression	Lasso	Ridge	Regressor Tree	*k*NN
RMSE (Original)	127.75	127.64	127.74	137.51	150.54

Lasso, being the model with the lowest RMSE at **127.64**, is by far the best performing model. In fact, since it is basically a variation of linear regression and it outperformed both simple linear regression and ridge, I will be using Lasso in place of linear regression for parts b) and c).

(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]

Create some features. Build new models. improving the performance as much as possible. Different models can handle different features in different ways.
You only need to show the BEST performing model and discuss what happened. Make sure you are consistent. E.g. you can't compare feature a from model x and feature y from model y (or was it metric a from model x and metric b for model y).

The first step I did was to create a new 'X_transform' version of my X variables to make sure that my un-transformed and transformed Xs stay independent from each other. Talking a look at the the head, everything seems to be the same. I then created two new features.

- The first one is 'day_diff', which is the difference between the first and most recent day the customer's records are updated. Because we don't know if a customer will be returning in the future or will never come back, we can only confidently say that the customer has been with us for the duration that they first and last updated their records.
- The second variable I created was "freqday", which takes customer duration (day_diff) and dividing it by the "freq" variable (number of transactions in the past year). This should give us an idea as to their customer lifetime as a relation to the number of purchases.

With these two variables, we now have 24 X variables.

In [67]:
```
#Creating a new X for transformations
X_transform=hw4data.iloc[:,:-1]
X_transform.head()
```

executed in 18ms, finished 15:00:50 2019-10-08

Out[67]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	...	source_u	source_p	source_x	source_w	Freq	las
0	1	0	0	1	0	0	0	0	0	0	...	0	0	0	0	2	
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	0	0	0	2	
3	1	0	1	0	0	0	0	0	0	0	...	0	0	0	0	1	
4	1	0	1	0	0	0	0	0	0	0	...	0	0	0	0	1	

5 rows × 22 columns

In [68]:
```
#Creating a new variable "day diff" that calculates customer lifetime
X_transform['day_diff']=X_transform['1st_update_days_ago']-X_transform['last_update_days_ago']
X_transform['freqday']=(X_transform['day_diff']/X_transform['Freq']).replace([np.inf, -np.inf], np.nan).fillna(0)

#X_transform.head()

X_transform.describe()
```

executed in 57ms, finished 15:00:51 2019-10-08

Out[68]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	...	source_x	2(
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	2000.000000	...	2000.000000	
mean	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	0.068500	...	0.018000	
std	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	0.252665	...	0.132984	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
25%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
50%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
75%	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	...	0.000000	
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	1.000000	...	1.000000	

8 rows × 24 columns

Next, I tried playing around with polynomials by creating squared and cubed versions of my X variables. After running both, I decided to only use cubic for all my models, as cubic includes all the variables for squared AND performed slightly better. This increased my number of X variables to 2925.

There are many columns where it's one binary dummy variable multiplying another (for source), and the result would be a whole column of 0s. This would do nothing to improve my model, I dropped all columns that have a standard deviation of 0. This brought the number of variables down to 1206.

I also dropped all duplicate columns, as a binary column would still be the exact same if we square or cube it. This brought the number of variables to 902.

In [69]:
```
# Generate polynomial and interaction features.
# PolynomialFeatures = Generate a new feature matrix consisting of all polynomial
# combinations of the features with degree less than or equal to the specified degree.
# For example, if an input sample is two dimensional and of the form [a, b], the degree-2
# polynomial features are [1, a, b, a^2, ab, b^2].

squared = PolynomialFeatures(degree=2) # degree = the degree of the polynomial features (default = 2)
cubic = PolynomialFeatures(degree=3) #We know that cube
squaredx = squared.fit_transform(X_transform)
cubicx = cubic.fit_transform(X_transform)

targetfeaturenames = squared.get_feature_names(X_transform.columns)
squaredx = pd.DataFrame(squaredx, columns=targetfeaturenames )
targetfeaturenames = cubic.get_feature_names(X_transform.columns)
cubicx = pd.DataFrame(cubicx, columns=targetfeaturenames )

print(squaredx.shape)
print(cubicx.shape)
```

executed in 142ms, finished 15:00:52 2019-10-08

```
(2000, 325)
(2000, 2925)
```

In [70]:
```
#remove all columns with only 0s
cubicx = cubicx.loc[:, cubicx.std()!=0]
squaredx = squaredx.loc[:, squaredx.std()!=0]
print(squaredx.shape)
print(cubicx.shape)
```

executed in 92ms, finished 15:00:59 2019-10-08

```
(2000, 219)
(2000, 1206)
```

In [71]:
```
#remove all duplicate columns (e.g. x and X^2 if x is a binary variable)
cubicx = cubicx.T.drop_duplicates().T
squaredx = squaredx.T.drop_duplicates().T
print(squaredx.shape)
print(cubicx.shape)
```

executed in 647ms, finished 15:01:00 2019-10-08

```
(2000, 197)
(2000, 902)
```

Because all my numeric columns have the string ‘day’ in it, I wanted to see if I can further transform/improve all columns containing numeric values by making the distribution more symmetric. However, I quickly realized that this only works for linear models with a linear relationship. Because both kNN and regressor tree are not linear, and we don’t know if the linear regression model is truly linear, I in the end decided to give up on using log, squareroot or cuberoot to normalize the distribution of my numeric variables. I think another possible source of influence for this worsening performance is that there are underlying interactive factors between the variables that we are unable to find.

Below, you can see me finding all columns containing ‘day’ (so related to a numeric variable), finding the most skewed variables and trying to perform log and/or cuberoot on it. I did not include this in my final model runs because it did not work well, but an attempt is an attempt.

In [94]:
```
#Finding all columns that are influenced by day variables (numeric)
daycols = [col for col in cubicx.columns if 'day' in col]
daycols[1:10]
```

executed in 6ms, finished 16:15:34 2019-10-08

Out[94]:
```
['1st_update_days_ago',
 'day_diff',
 'freqday',
 'US last_update_days_ago',
 'US 1st_update_days_ago',
 'US day_diff',
 'US freqday',
 'source_a last_update_days_ago',
 'source_a 1st_update_days_ago']
```

In [95]:
```
#seeing most skew variables
cubicx[daycols].skew().sort_values().head()
```

executed in 44ms, finished 16:15:40 2019-10-08

Out[95]:
```
1st_update_days_ago       -0.489562
US 1st_update_days_ago    -0.296689
last_update_days_ago      -0.187871
US last_update_days_ago   -0.000086
1st_update_days_ago^2      0.154370
dtype: float64
```

In [72]:
```
#Get the cuberoot of all day (numeric) related variables
transx = cubicx
#transx[daycols] = np.cbrt(transx[daycols]) #np.log(+0.01)
#transx.isnull().sum().head()
```

executed in 4ms, finished 15:01:05 2019-10-08

In [73]:
```
transx.head()
```

executed in 22ms, finished 15:01:07 2019-10-08

Out[73]:

	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	...	Gender=male day_diff^2	Gender=male day_diff freqday	Gender=male freqday^2	Address_ day_
0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
1	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
2	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
3	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	
4	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	

5 rows × 902 columns

I then encoded the transformed x for the X sets I will be using for my models. The first cell below allows me to change my X for my models on the fly without having to go back and changing the X in all of my respective models. However, I ended up using the same transformed X for all models to keep things consistent. Additionally, trying to juggle hundreds of different variables became too hectic, and because I will be using step-wise feature selection below.

In [74]:
```
#Encode transformations for each model
X_lin_transform = transx
X_tree_transform = transx
X_kNN_transform = transx
```
executed in 6ms, finished 15:01:09 2019-10-08

In [75]:
```
#Fuction that allows us to directly get results of the 2econd or 3rd degree of our features
#explore non-linear relationships/patterns in our data

print(X.shape) # Data size after polynomial transformation
print(X_lin_transform.shape)
print(X_tree_transform.shape)
print(X_kNN_transform.shape)

#print(X[1:5,:]) # Preview data after polynomial transformations
#print(X_lin_transform.iloc[1:5,:])
#print(X_tree_transform.iloc[1:5,:])
#print(X_kNN_transform.iloc[1:5,:])
```
executed in 3ms, finished 15:01:11 2019-10-08

```
(2000, 22)
(2000, 902)
(2000, 902)
(2000, 902)
```

Because we know that the goal of selecting/subsetting predictors is to find the parsimonious model (i.e. the simplest model that performs sufficiently well), I decided to utilize a stepwise feature selection package/module for this task. YOU MUST INSTALL THE CORRECT PACKAGE mentioned at the beginning of part a) for this module to work. But in a nutshell, I asked the module to use my previous models and find the best “k_features” (number of features) model that has the smallest mean squared error. I ran this for all three models, with different number of features for all of them.

As I have discussed above, I have chosen lasso to represent linear regression models from here on out because a) lasso IS linear regression just with automatic feature selection, b) lasso performed better than linear regression, and c) lasso has 2 parameters that I can tune for part c), while linear regression does not. Lasso with alpha set to 0 IS linear regression. I know lasso MSE will decrease but in smaller and smaller increments as we reach a certain number of variables, so I decided to set a relatively higher number of features at 35. In the end, I was able to get an RMSE as low as 117.98 for lasso. I will discuss/compare all the results together and justify why we saw an improvement below.

3.1 T-Lasso Regression

In [89]:

```
############################LASSO SFS#########################################
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
lassoSFS = Lasso(alpha=0.1,random_state=42)
lassoSFSfinal = SFS(lassoSFS, k_features = 35, forward = True, #edit k_features
            floating = False, verbose = 2,
            scoring = 'neg_mean_squared_error',
            cv=inner_cv, n_jobs = -1)

lassoSFSfinal = lassoSFSfinal.fit(X_lin_transform, y)
pd.DataFrame.from_dict(lassoSFSfinal.get_metric_dict()).T

lassoSFSresult = pd.DataFrame.from_dict(lassoSFSfinal.get_metric_dict()).T
lassoSFSfeatures = lassoSFSresult[lassoSFSresult.avg_score==max(lassoSFSresult.avg_score)].feature_names
lassoSFSfeaturesFinal = list(list(lassoSFSfeatures)[0])
lassoSFSfeaturesFinal = X_lin_transform[lassoSFSfeaturesFinal]

```

executed in 19m 14s, finished 16:08:25 2019-10-08

```
[Parallel(n_jobs=-1)]: Done 870 out of 870 | elapsed:   59.1s finished

[2019-10-08 16:06:01] Features: 33/35 -- score: -13959.892115455412[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8
concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    4.1s
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed:   12.5s
[Parallel(n_jobs=-1)]: Done 349 tasks      | elapsed:   26.4s
[Parallel(n_jobs=-1)]: Done 632 tasks      | elapsed:   48.9s
[Parallel(n_jobs=-1)]: Done 869 out of 869 | elapsed:  1.2min finished

[2019-10-08 16:07:13] Features: 34/35 -- score: -13939.830056457762[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8
concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    4.7s
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed:   14.4s
[Parallel(n_jobs=-1)]: Done 349 tasks      | elapsed:   31.1s
[Parallel(n_jobs=-1)]: Done 632 tasks      | elapsed:   53.0s
[Parallel(n_jobs=-1)]: Done 868 out of 868 | elapsed:  1.2min finished

[2019-10-08 16:08:25] Features: 35/35 -- score: -13919.356209470401
```

In [90]:

```
######################## Linear Regression Model - Lasso (NTK) ########################
lasso_SFS = Lasso(alpha=0.1,random_state=42)
lasso_SFS = lasso_SFS.fit(lassoSFSfeaturesFinal, y) # Fit model to data

lassoSFS_CV = cross_val_score(lasso_SFS, X=lassoSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_squared_error") #or X=
print("CV MSE Performance: ", lassoSFS_CV.mean(), " +/- ", lassoSFS_CV.std())
print("CV RMSE Performance: ", np.sqrt(abs(lassoSFS_CV.mean())), " +/- ", (np.sqrt(abs(lassoSFS_CV.mean()))+lassoSFS_CV.std(
lassoSFS_CV = cross_val_score(lasso_SFS, X=lassoSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_absolute_error")
print("CV MAE Performance: ", lassoSFS_CV.mean(), " +/- ", lassoSFS_CV.std())
```

executed in 847ms, finished 16:08:32 2019-10-08

```
CV MSE Performance:  -13919.356186854382  +/-  4614.861346863342
CV RMSE Performance:  117.98032118473988  +/-  18.16011194854248
CV MAE Performance:  -69.57585491201988  +/-  6.306007824923059
```

From experimentation, I found that MSE for regressor tree will increase but smaller and smaller as we add more relevant variables, so I decided to set a relatively lower number of features at 25. RMSE stabilized at around 129.65 and did not move much, or if at all lower than that. In the end, I was able to get an RMSE of 129.65 for regressor tree. I will discuss/compare all the results together and justify why we saw an improvement below.

3.2 T-Regressor Tree

In [29]:

```
############################Regressor Tree SFS#############################################
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
treeSFS = DecisionTreeRegressor(max_depth=3,random_state=42)
treeSFSfinal = SFS(treeSFS, k_features = 25, forward = True, #edit k_features
            floating = False, verbose = 2,
            scoring = 'neg_mean_squared_error',
            cv=inner_cv, n_jobs = -1)

treeSFSfinal = treeSFSfinal.fit(X_tree_transform, y)
pd.DataFrame.from_dict(treeSFSfinal.get_metric_dict()).T

treeSFSresult = pd.DataFrame.from_dict(treeSFSfinal.get_metric_dict()).T
treeSFSfeatures = treeSFSresult[treeSFSresult.avg_score==max(treeSFSresult.avg_score)].feature_names
treeSFSfeaturesFinal = list(list(treeSFSfeatures)[0])
treeSFSfeaturesFinal = X_tree_transform[treeSFSfeaturesFinal]
```

executed in 2m 46s, finished 14:03:39 2019-10-08

```
[2019-10-08 14:01:05] Features: 3/25 -- score: -17183.97657088242[Parallel(n_jobs=-1)]: Using backend LokyBack
end with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    2.1s
[Parallel(n_jobs=-1)]: Done 644 tasks      | elapsed:    4.0s
[Parallel(n_jobs=-1)]: Done 710 out of 710 | elapsed:    4.3s finished

[2019-10-08 14:01:10] Features: 4/25 -- score: -16810.449517855137[Parallel(n_jobs=-1)]: Using backend LokyBac
kend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    2.0s
[Parallel(n_jobs=-1)]: Done 644 tasks      | elapsed:    4.5s
[Parallel(n_jobs=-1)]: Done 709 out of 709 | elapsed:    4.8s finished

[2019-10-08 14:01:15] Features: 5/25 -- score: -16810.249756220455[Parallel(n_jobs=-1)]: Using backend LokyBac
kend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    2.4s
[Parallel(n_jobs=-1)]: Done 408 tasks      | elapsed:    4.3s
[Parallel(n_jobs=-1)]: Done 708 out of 708 | elapsed:    5.8s finished

[2019-10-08 14:01:21] Features: 6/25 -- score: -16810.249756220455[Parallel(n_jobs=-1)]: Using backend LokyBac
```

In [30]:

```
################################### Regressor Tree - Numeric Prediction (NTK) ##############################
tree_SFS = DecisionTreeRegressor(max_depth=3,random_state=42)
tree_SFS = tree_SFS.fit(treeSFSfeaturesFinal, y)

treeSFS_CV = cross_val_score(tree_SFS, X=treeSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_squared_err
print("CV MSE Performance: ", treeSFS_CV.mean(), " +/- ", treeSFS_CV.std())
print("CV RMSE Performance: ", np.sqrt(abs(treeSFS_CV.mean())), " +/- ", (np.sqrt(abs(treeSFS_CV.mean())+treeS
treeSFS_CV = cross_val_score(tree_SFS, X=treeSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_absolute_er
print("CV MAE Performance: ", treeSFS_CV.mean(), " +/- ", treeSFS_CV.std())
```

executed in 95ms, finished 14:07:56 2019-10-08

```
CV MSE Performance:  -16810.249756220455  +/-  5694.209432160333
CV RMSE Performance:  129.65434723224845  +/-  20.36051599263598
CV MAE Performance:  -75.411174580852  +/-  8.287117672517184
```

From experimentation, I found that MSE for kNN decrease in a similar fashion as lasso, but since kNN takes forever to run, I decided to keep the number of features at 25. In the end, I was able to get an RMSE of 124.61 for kNN. I will discuss/compare all the results together and justify why we saw an improvement below.

3.3 T-kNN Regression

In [79]:
```
################### Standardizing Transformed X for kNN ############################
sc = StandardScaler()
sc.fit(X_kNN_transform)
X_kNN_trans_std = sc.transform(X_kNN_transform)
X_kNN_trans_std = pd.DataFrame(X_kNN_trans_std, columns=X_kNN_transform.columns)
```

executed in 62ms, finished 15:21:54 2019-10-08

In [85]:
```
################################kNN SFS#########################################################
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
knnSFS = neighbors.KNeighborsRegressor(n_neighbors = 3)
knnSFSfinal = SFS(knnSFS, k_features = 25, forward = True, #edit k_features
            floating = False, verbose = 2,
            scoring = 'neg_mean_squared_error',
            cv=inner_cv, n_jobs = -1)

knnSFSfinal = knnSFSfinal.fit(X_kNN_trans_std, y)
pd.DataFrame.from_dict(knnSFSfinal.get_metric_dict()).T

knnSFSresult = pd.DataFrame.from_dict(knnSFSfinal.get_metric_dict()).T
knnSFSfeatures = knnSFSresult[knnSFSresult.avg_score==max(knnSFSresult.avg_score)].feature_names
knnSFSfeaturesFinal = list(list(knnSFSfeatures)[0])
knnSFSfeaturesFinal = X_kNN_trans_std[knnSFSfeaturesFinal]
```

executed in 11m 52s, finished 15:39:46 2019-10-08

```
[Parallel(n_jobs=-1)]: Done 880 out of 880 | elapsed:   37.9s finished

[2019-10-08 15:38:18] Features: 23/25 -- score: -15538.996851172222[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8
concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    3.6s
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed:    8.7s
[Parallel(n_jobs=-1)]: Done 349 tasks      | elapsed:   17.3s
[Parallel(n_jobs=-1)]: Done 632 tasks      | elapsed:   30.3s
[Parallel(n_jobs=-1)]: Done 879 out of 879 | elapsed:   41.3s finished

[2019-10-08 15:38:59] Features: 24/25 -- score: -15536.6093452[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concu
rrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed:    4.3s
[Parallel(n_jobs=-1)]: Done 146 tasks      | elapsed:    9.9s
[Parallel(n_jobs=-1)]: Done 349 tasks      | elapsed:   19.4s
[Parallel(n_jobs=-1)]: Done 632 tasks      | elapsed:   34.1s
[Parallel(n_jobs=-1)]: Done 878 out of 878 | elapsed:   45.9s finished

[2019-10-08 15:39:46] Features: 25/25 -- score: -15528.760769800001
```

In [86]:
```
################################### kNN Regressor Example (NTK) ###################################
knn_SFS = neighbors.KNeighborsRegressor(n_neighbors = 3)
knn_SFS = knn_SFS.fit(knnSFSfeaturesFinal, y)  #fit the model

KnnSFS_CV = cross_val_score(knn_SFS, X=knnSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_squared_error") #or X=X_tra
print("CV MSE Performance: ", KnnSFS_CV.mean(), " +/- ", KnnSFS_CV.std())
print("CV RMSE Performance: ", np.sqrt(abs(KnnSFS_CV.mean())), " +/- ", (np.sqrt(abs(KnnSFS_CV.mean())+KnnSFS_CV.std())-np.
KnnSFS_CV = cross_val_score(knn_SFS, X=knnSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_absolute_error")
print("CV MAE Performance: ", KnnSFS_CV.mean(), " +/- ", KnnSFS_CV.std())
```

executed in 299ms, finished 15:40:00 2019-10-08

```
CV MSE Performance:  -15528.760769800001  +/-  4006.0541060065852
CV RMSE Performance:  124.61444847929954  +/-  15.152553619931979
CV MAE Performance:  -71.93142666666667  +/-  5.039852740712433
```

Here are the best 35, 25 and 25 variables for lasso, regressor tree and kNN respectively:

Lasso Stepwise-Feature-Selection Variables:

```
In [108]: lassoSFSfeaturesFinal.columns
executed in 5ms, finished 20:21:36 2019-10-08

Out[108]: Index(['source_c', 'Freq', 'US 1st_update_days_ago',
       'source_c 1st_update_days_ago', 'source_u Gender=male',
       'source_w Gender=male', 'Freq 1st_update_days_ago',
       'Freq Address_is_res', 'last_update_days_ago Address_is_res',
       'Web order Address_is_res', 'US source_u freqday',
       'US source_w 1st_update_days_ago', 'US Freq last_update_days_ago',
       'US Freq Web order', 'US Web order Address_is_res',
       'source_a Freq last_update_days_ago', 'source_a Web order day_diff',
       'source_c Freq Web order', 'source_c Freq Gender=male',
       'source_d freqday^2', 'source_e Freq Gender=male',
       'source_e last_update_days_ago day_diff',
       'source_o Gender=male day_diff', 'source_h Freq last_update_days_ago',
       'source_r Web order Gender=male', 'source_u Web order Address_is_res',
       'source_u Web order day_diff', 'Freq^2 last_update_days_ago',
       'Freq last_update_days_ago Web order', 'Freq Web order Address_is_res',
       'last_update_days_ago^2 Address_is_res',
       'Web order Address_is_res freqday', 'Web order day_diff^2',
       'Gender=male Address_is_res freqday', 'Address_is_res day_diff^2'],
      dtype='object')
```

Regressor Tree Stepwise-Feature-Selection Variables:

```
In [109]: treeSFSfeaturesFinal.columns
executed in 5ms, finished 20:21:48 2019-10-08

Out[109]: Index(['US', 'source_a', 'source_c', 'source_b', 'source_d', 'source_e',
       'source_m', 'source_o', 'source_r', 'source_t', 'source_x', 'source_w',
       'Freq', 'Web order', 'US source_c', 'US source_b',
       'source_a 1st_update_days_ago', 'source_x 1st_update_days_ago',
       'US source_m Address_is_res', 'source_p Gender=male day_diff',
       'Freq^2 last_update_days_ago', 'Freq^2 1st_update_days_ago',
       'Freq Web order Address_is_res', 'Freq Gender=male day_diff'],
      dtype='object')
```

kNN Regression Stepwise-Feature-Selection Variables:

```
In [110]: knnSFSfeaturesFinal.columns
executed in 5ms, finished 20:22:02 2019-10-08

Out[110]: Index(['source_m Freq', 'source_o Address_is_res', 'US source_m Freq',
       'US source_r freqday', 'source_a Web order Address_is_res',
       'source_c Freq Gender=male', 'source_c Web order Address_is_res',
       'source_c Address_is_res day_diff', 'source_b last_update_days_ago^2',
       'source_b last_update_days_ago day_diff',
       'source_b 1st_update_days_ago^2',
       'source_e last_update_days_ago Gender=male', 'source_m Freq^2',
       'source_o Freq last_update_days_ago', 'source_o Freq Address_is_res',
       'source_h Freq Gender=male', 'source_h Freq Address_is_res',
       'source_h 1st_update_days_ago Web order',
       'source_h Address_is_res freqday', 'source_s Freq Address_is_res',
       'Freq^2 Address_is_res', 'Freq^2 day_diff',
       'Freq 1st_update_days_ago Address_is_res',
       'last_update_days_ago Address_is_res freqday',
       'last_update_days_ago day_diff^2'],
      dtype='object')
```

All RMSE are from 10-fold cross validation. Here are the performances after feature engineering compared to the base models:

	Linear Regression	**Lasso**	**Ridge**	**Regressor Tree**	***k*NN**
RMSE (Original)	127.75	127.64	127.74	137.51	150.54
RMSE (Feature Engineering)	Eliminated	117.98	Eliminated	129.65	124.61

Across the board, all three models improved after feature engineering, with lower RMSEs. Again, lasso outperformed the other two models. I have already discussed what RMSE means above, so we know lower is better. I think the reason why generalization performance improved across the board is because having good features allow us to most accurately represent the underlying structure of the data and therefore create the best model. By cubing, we created almost every single variable interaction possible. By then selecting the best variables from this list of all possible, we are effectively selecting the best possible variables for our models.

(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]

With the best features/variables selected, parameter tuning is the last step of the journey, and it's very straight forward. We throw our list of features from each of the three step-wise parameter selection calls into the tuner with two parameters to tune. We also make sure to use INNER CV for parameter tuning and OUTER CV for performance testing to produce non-nested and nested CV performances. We made sure to set random state to 42 and metric to mean_squared_error, which is just RMSE squared.

For Lasso, I decided to tune alpha because alpha controls the constant that multiplies the L1 term (aka how much penalty to put). There aren't that many parameters to tun for linear regression/lasso, but I also decided to tune fit_intercept, which controls whether to calculate the intercept of this model, with false meaning that no intercept will be used in calculations (e.g. data is expected to be already centered) to fufill the 2 parameters requirement. The final RMSE for the best lasso is 117.94, with alpha set to 0.001 and fit_intercept set to true. This is marginally better than the previous tree model (post stepwise feature selection).

4.1 Lasso Param Tuning

In [101]:

```
from sklearn.linear_model import Lasso # Lasso Regression class
 # Lasso                   # alpha : constant that multiplies the L1 term.
                           # Defaults to 1.0. alpha = 0 is equivalent to an ordinary least square,
                           # solved by the LinearRegression object.

gs_lr = 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, 1000
                                'fit_intercept': ['True', 'False']}],
                  scoring='neg_mean_squared_error',
                  cv=inner_cv, n_jobs=4)

gs_lr = gs_lr.fit(lassoSFSfeaturesFinal, y)
print("\n Linear Regression (Lasso) Parameter Tuning")
print("Non-nested CV F1-Score: ", gs_lr.best_score_)
print("Optimal Parameter: ", gs_lr.best_params_)
print("Optimal Estimator: ", gs_lr.best_estimator_)
nested_score_gs_lr = cross_val_score(gs_lr, X=lassoSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_squared_error")
print("Nested CV MSE:",nested_score_gs_lr.mean(), " +/- ", nested_score_gs_lr.std())
print("Nested CV RMSE: ", np.sqrt(abs(nested_score_gs_lr.mean())), " +/- ", (np.sqrt(abs(nested_score_gs_lr.mean())+nested_

#feature selection, some of the features coefficients become 0
```

executed in 41.3s, finished 16:32:26 2019-10-08

```
 Linear Regression (Lasso) Parameter Tuning
Non-nested CV F1-Score:  -13908.994802753288
Optimal Parameter:  {'alpha': 0.001, 'fit_intercept': 'True'}
Optimal Estimator:  Lasso(alpha=0.001, 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 MSE: -13909.745085337738  +/-  4555.392573989813
Nested CV RMSE:  117.93958235188785  +/-  17.946905789235103
```

Tuning for Regressor Tree is a similar to tuning for classification tree. Max_depth controls the max depth of the tree, min_samples_leaf controls the minimum number of samples required to be a leaf node, and min_sample_split controls the minimum number of samples required to split an internal node. The optimized parameters for decision tree after our first parameter tuning are a max depth of 3, max_samples_leaf of 1 and minimal sample split of 2. I have tried extending the possible range of the parameters up to 10 (e.g. min_samples_split tested for 2-10) but the results stayed the same. As well, the prompt asks for a minimum of 2 parameters to tune, which we have clearly surpassed. The best regressor tree has a RMSE of 131.2857, which is actually worse than the previous tree model (post stepwise feature selection).

4.2 Tree Param Tuning

In [102]:
```
gs_dt = GridSearchCV(DecisionTreeRegressor(random_state=42, criterion= 'mse'),
                     param_grid=[{'max_depth': [1,2,3,4,5],
                                  'min_samples_leaf':[1,2,3,4,5],
                                  'min_samples_split':[2,3,4,5]}],
                     scoring="neg_mean_squared_error", # Specifying multiple metrics for evaluation
                     cv=inner_cv, n_jobs=4) #GridSearchCV

gs_dt = gs_dt.fit(treeSFSfeaturesFinal,y)
print("\nRegressor Tree Parameter Tuning")
print("Non-nested RMSE Performance: ", gs_dt.best_score_)
print("Optimal Parameter: ", gs_dt.best_params_)    # Parameter setting that gave the best results on the hold
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator
nested_score_gs_dt = cross_val_score(gs_dt, X=treeSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_square
print("Nested CV MSE: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
print("Nested CV RMSE: ", np.sqrt(abs(nested_score_gs_dt.mean())), " +/- ", (np.sqrt(abs(nested_score_gs_dt.me

```

executed in 40.1s, finished 16:33:06 2019-10-08

```
Regressor Tree Parameter Tuning
Non-nested RMSE Performance:  -16810.24975622045
Optimal Parameter:  {'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeRegressor(criterion='mse', max_depth=3, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=42, splitter='best')
Nested CV MSE:  -17235.934483967627  +/-  5594.700815155602
Nested CV RMSE:  131.28569794142706  +/-  19.812400322989305
```

The last model to be tuned is kNN. Again, this is similar to classification model tuning, just with scoring changed to MSE. I played around with the number of neighbors (number of neighbors to use), weights (how neighbors of different distance are weighted) and p (power parameter for the minkowski metric, with 1=manhattan_distance, 2= Euclidean_distance and any other arbitrary p the minkowski_distance of (l_p) is used). Our optimization call yielded a p of 2 (manhattan_distance), uniform weighting, and 3 neighbors. The best performing RMSE is 132.137, which is again worse than the previous knn model(post stepwise feature selection).

4.3 kNN Param Tuning

In [103]:
```
gs_knn = GridSearchCV(estimator=neighbors.KNeighborsRegressor(
                            metric='minkowski'),
                      param_grid=[{'n_neighbors': [1,3,5,7,9,11,13,15],
                                   'weights':['uniform','distance'],
                                   'p':[1,2,3]}],
                      scoring="neg_mean_squared_error",
                      cv=inner_cv, n_jobs=4)

gs_knn = gs_knn.fit(knnSFSfeaturesFinal,y)  #X_std
print("\n kNN Parameter Tuning")
print("Non-nested RMSE 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
nested_score_gs_knn = cross_val_score(gs_knn, X=knnSFSfeaturesFinal, y=y, cv=outer_cv, scoring="neg_mean_squar
print("Nested CV MSE: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
print("Nested CV RMSE: ", np.sqrt(abs(nested_score_gs_knn.mean())), " +/- ", (np.sqrt(abs(nested_score_gs_knn.

```

executed in 1m 32.0s, finished 16:34:39 2019-10-08

```
 kNN Parameter Tuning
Non-nested RMSE Performance:  -15528.7607698
Optimal Parameter:  {'n_neighbors': 3, 'p': 2, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                    weights='uniform')
Nested CV MSE:  -17460.102614695345  +/-  4924.116377933236
Nested CV RMSE:  132.13668156380857  +/-  17.47688397297216
```

Here are the generalization performances across all a), b) and c) for comparison.

	Linear Regression	Lasso	Ridge	Regressor Tree	*k*NN
RMSE (Original)	127.75	127.64	127.74	137.51	150.54
RMSE (Feature Engineering)	Eliminated	117.98	Eliminated	129.65	124.61
RMSE (Parameter Tuning)	Eliminated	117.94	Eliminated	131.29	132.14

Across the board, Lasso is the only model that improved every single time. Both regressor tree and kNN improved after feature engineering (and stepwise feature-selection), then performed worse after parameter tuning. Lasso, with an RMSE of 117.94, was the unchallenged champion amongst the three model. I am not sure how to explain why the performances for tree and knn were worse for parameter tuning, as technically it is supposed to find the parameters that minimize MSE. We did not learn about this in class. However, as the prompt ONLY asked us to discuss which model was the best AND NOT WHY if any models became worse, I think I have fulfilled what the question is asking for in its entirety.