Homework #4

Adil Ashish Kumar

(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 (Ai)	Predicted Credit Rating (pi)	Ei = predicted - actual	abs(Ei)	Abs(Ei)/ai
670	710	40	40	40/670
680	660	-20	20	20/680
550	600	50	50	50/550
740	800	60	60	60/740
700	600	-100	100	100/700

[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]

Mean absolute error (MAE) = sum (abs(Ei)) /n
= (40+20+50+60+100)/5 = 54

Mean absolute % error (MAPE) = sum(abs(Ei/Ai))/n

= (0.0597 +0.0294 + 0.0909 + 0.081 + 0.1429)/5 = 0.081

Root Mean Square Error = sqrt(sum (Ei^2) /n)

= sqrt((1600+400+2500+3600 +10000)/5) = 60.166

Avg error = sum (Ei) /n

= (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)
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]

```
# To write a Python 2/3 compatible codebase, the first step is to add this line to the top of each module
from __future__ import division, print_function, unicode_literals
import numpy as np # np is an alias pointing to numpy
import pandas as pd # pd is an alias pointing to pandas
pd.set_option('display.max_columns', 50) #increasing no columns to display
pd.set_option('display.width', 120) #increasing panda output window width

import sys ,warnings
if not sys.warnoptions:
    warnings.simplefilter("ignore")

df = pd.read_excel("HW4.xlsx")
print(df.shape) # no of rows and columns in data
print()
print(df.head()) # understanding the data by viewing first 5 rows
print(df.describe(include='all')) # simple statistics on each column in data
print()
print(df.isnull().any())  # to check if any column has missing values
```

First, I imported the data and imported packages needed like numpy and pandas to help with understanding the data and data transformations. Using the shape method, we see that the data has 2000 rows and 25 columns. Using the head function on the data, we get a preview of the dataset. Each row corresponds to a customer transaction as to whether or not they purchased the product and how much they spent. Using the describe method we can see the summary statistics of the columns in the dataset. The first column is an identifier. Freq, last_update_days_ago, first_update_days_ago and spending are the only numeric variables in the dataset. All others are categorical variables with values – 0,1.

```
(2000, 25)

   sequence_number  US  source_a  source_c  source_b  source_d  source_e  source_m  source_o  source_h  source_r  \
0                1   1         0         0         1         0         0         0         0         0         0
1                2   1         0         0         0         0         1         0         0         0         0
2                3   1         0         0         0         0         0         0         0         0         0
3                4   1         0         1         0         0         0         0         0         0         0
4                5   1         0         1         0         0         0         0         0         0         0

   source_s  source_t  source_u  source_p  source_x  source_w  Freq  last_update_days_ago  1st_update_days_ago  \
0         0         0         0         0         0         0     2                  3662                 3662
1         0         0         0         0         0         0     0                  2900                 2900
2         0         1         0         0         0         0     2                  3883                 3914
3         0         0         0         0         0         0     1                   829                  829
4         0         0         0         0         0         0     1                   869                  869

   Web order  Gender=male  Address_is_res  Purchase  Spending
0          1            0               1         1    127.87
1          1            1               0         0      0.00
2          0            0               0         1    127.48
3          0            1               0         0      0.00
4          0            0               0         0      0.00
       sequence_number           US     source_a     source_c     source_b     source_d     source_e     source_m  \
count      2000.000000  2000.000000  2000.000000  2000.000000  2000.000000  2000.000000  2000.000000  2000.00000
mean       1000.500000     0.824500     0.126500     0.056000     0.060000     0.041500     0.151000     0.01650
std         577.494589     0.380489     0.332495     0.229979     0.237546     0.199493     0.358138     0.12742
min           1.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.00000
25%         500.750000     1.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.00000
50%        1000.500000     1.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.00000
75%        1500.250000     1.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.00000
max        2000.000000     1.000000     1.000000     1.000000     1.000000     1.000000     1.000000     1.00000
```

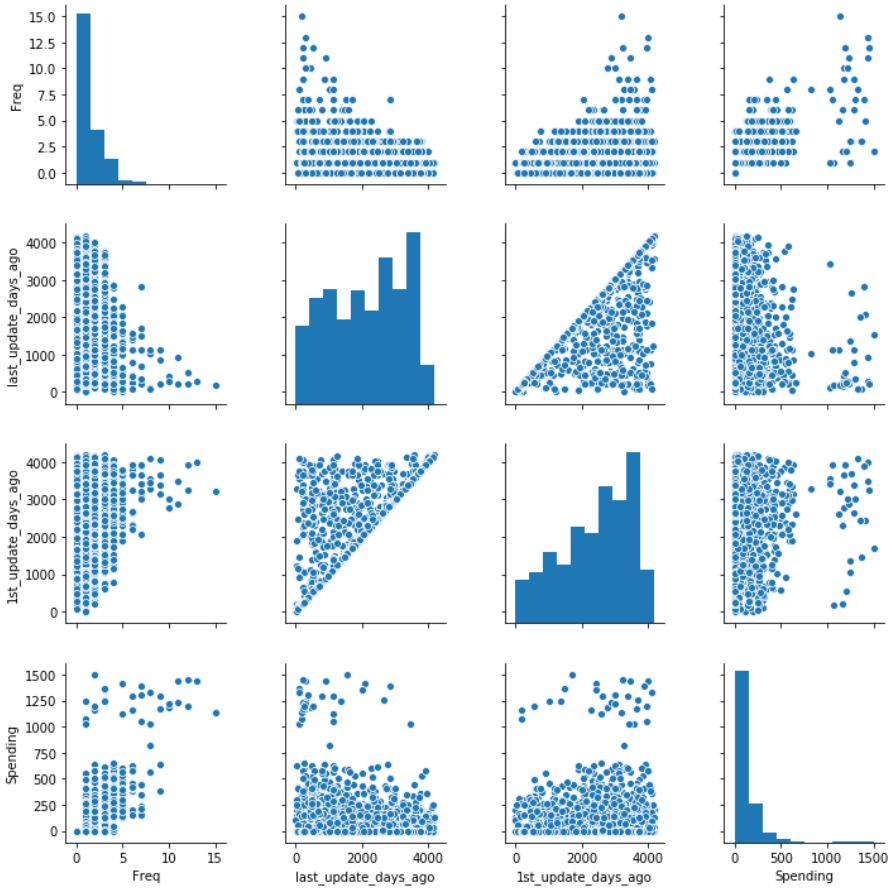
```
          source_o     source_h     source_r     source_s    source_t     source_u     source_p     source_x  \
count  2000.000000  2000.000000  2000.000000  2000.000000  2000.00000  2000.000000  2000.000000  2000.000000
mean      0.033500     0.052500     0.068500     0.047000     0.02150     0.119000     0.006000     0.018000
std       0.179983     0.223089     0.252665     0.211692     0.14508     0.323869     0.077246     0.132984
min       0.000000     0.000000     0.000000     0.000000     0.00000     0.000000     0.000000     0.000000
25%       0.000000     0.000000     0.000000     0.000000     0.00000     0.000000     0.000000     0.000000
50%       0.000000     0.000000     0.000000     0.000000     0.00000     0.000000     0.000000     0.000000
75%       0.000000     0.000000     0.000000     0.000000     0.00000     0.000000     0.000000     0.000000
max       1.000000     1.000000     1.000000     1.000000     1.00000     1.000000     1.000000     1.000000

          source_w         Freq  last_update_days_ago  1st_update_days_ago    Web order  Gender=male  Address_is_res  \
count  2000.000000  2000.000000           2000.000000          2000.000000  2000.000000  2000.000000     2000.000000
mean      0.137500     1.417000           2155.101000          2435.601500     0.426000     0.524500        0.221000
std       0.344461     1.405738           1141.302846          1077.872233     0.494617     0.499524        0.415024
min       0.000000     0.000000              1.000000             1.000000     0.000000     0.000000        0.000000
25%       0.000000     1.000000           1133.000000          1671.250000     0.000000     0.000000        0.000000
50%       0.000000     1.000000           2280.000000          2721.000000     0.000000     1.000000        0.000000
75%       0.000000     2.000000           3139.250000          3353.000000     1.000000     1.000000        0.000000
max       1.000000    15.000000           4188.000000          4188.000000     1.000000     1.000000        1.000000

          Purchase     Spending
count  2000.000000  2000.000000
mean      0.500000   102.560745
std       0.500125   186.749816
min       0.000000     0.000000
25%       0.000000     0.000000
50%       0.500000     1.855000
75%       1.000000   152.532500
max       1.000000  1500.060000
```

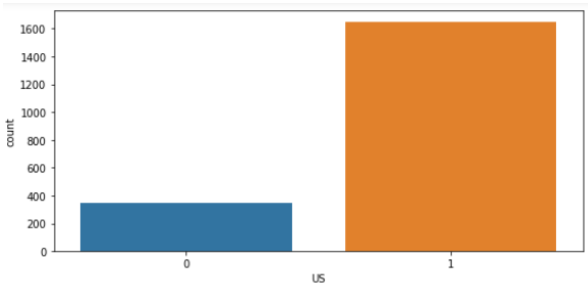
Below we see that there are no missing values in any column, so no missing value treatment is needed.

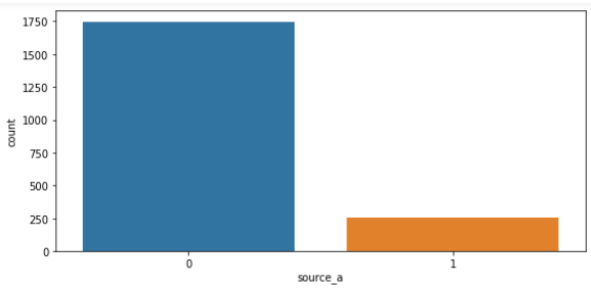
```
sequence_number         False
US                      False
source_a                False
source_c                False
source_b                False
source_d                False
source_e                False
source_m                False
source_o                False
source_h                False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

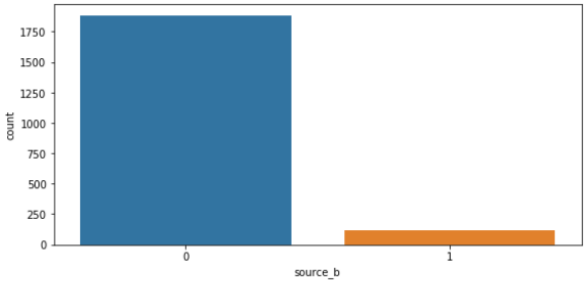


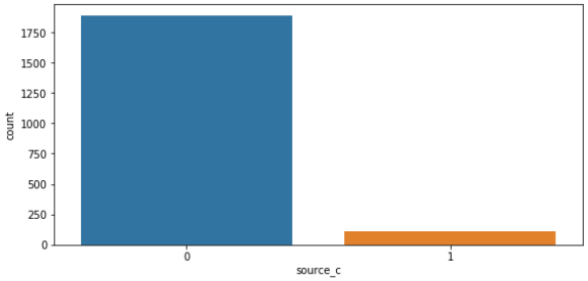
For the numeric variables in the data, the above plot shows us the distributions and correlations between them. The last plot on the bottom right shows us the target variable distribution – “Spending”. We can see that there are a lot of cases where spending is very low. The distribution is right skewed. This is an important point and maybe we can use transformations to transform the target variable. This could be of high value since we are going to use linear models in the modeling process.

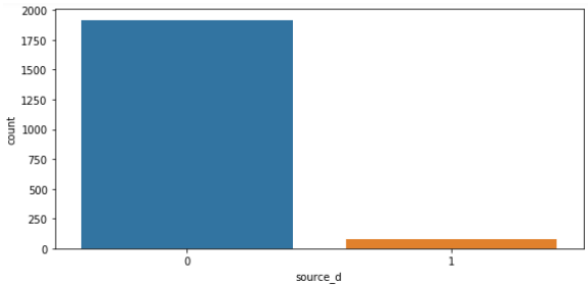
Below are the count plots for all the categorical variables. For most of the variables we notice that there is a huge imbalance in the distribution

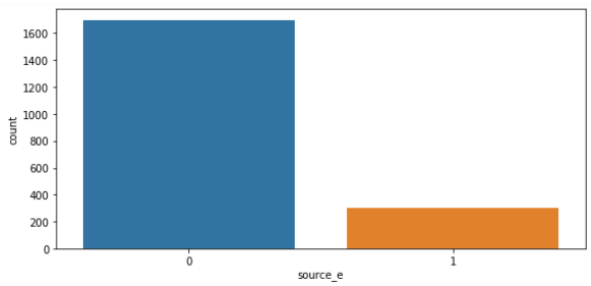


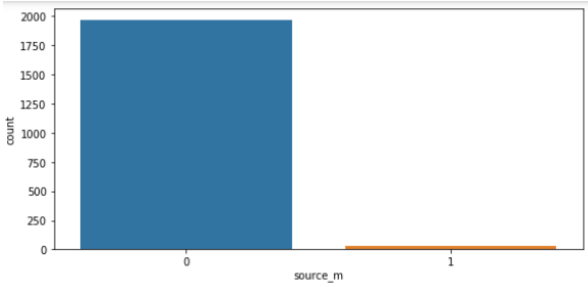


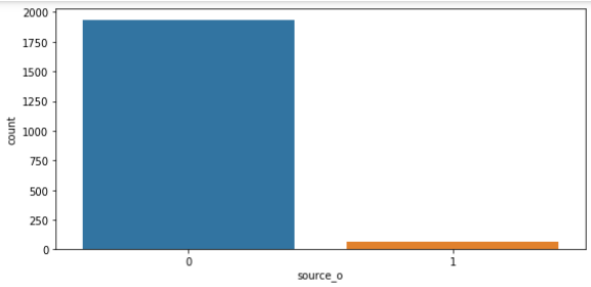


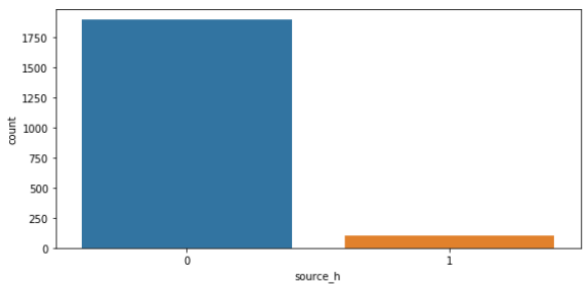


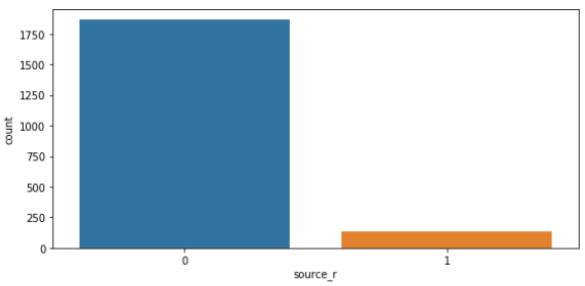


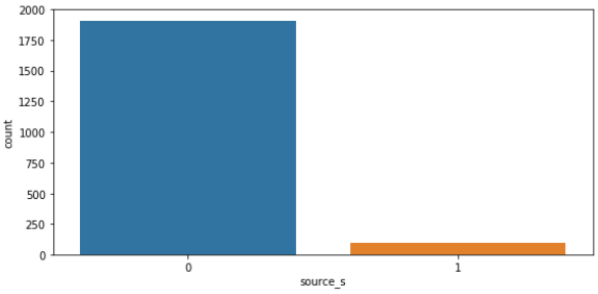


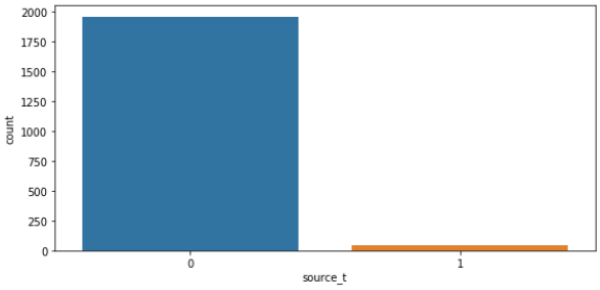


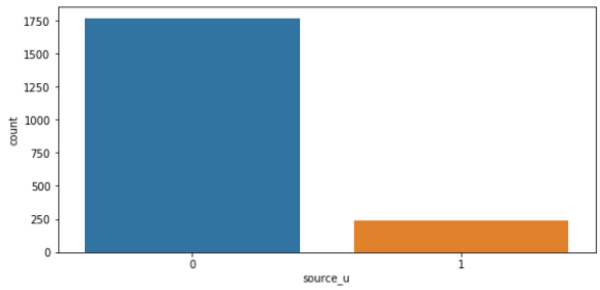


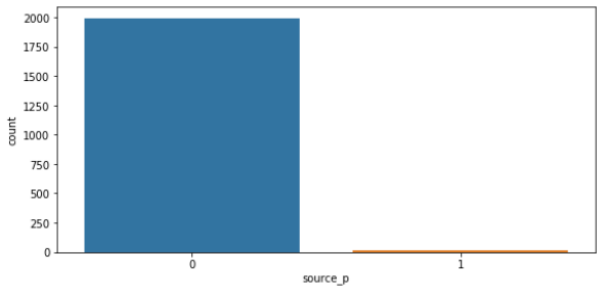


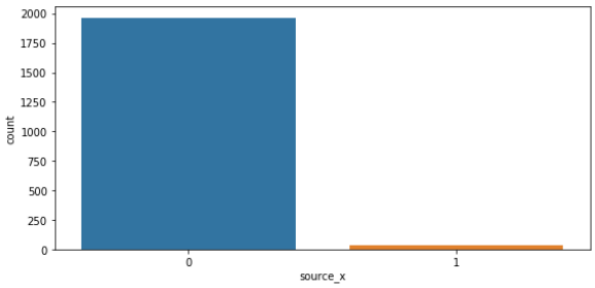


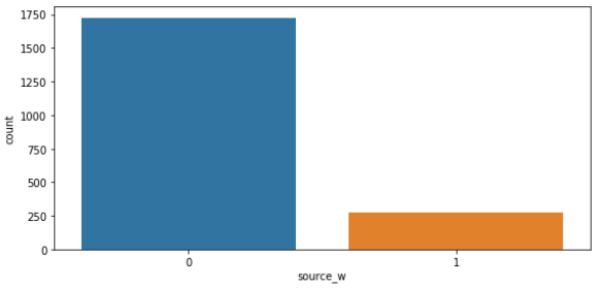


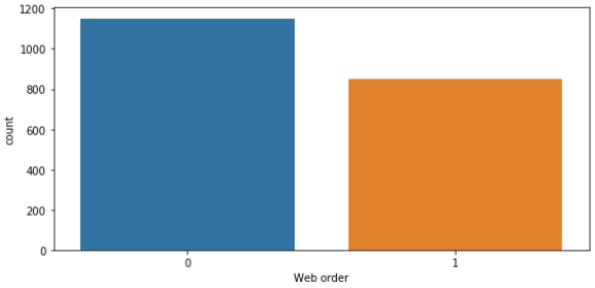


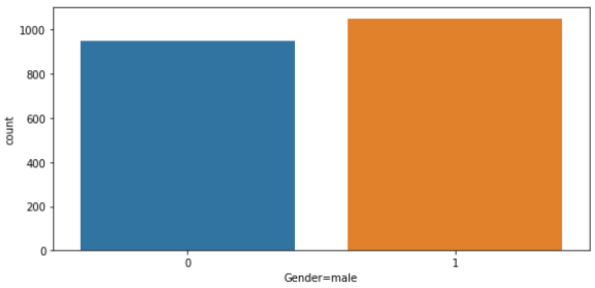


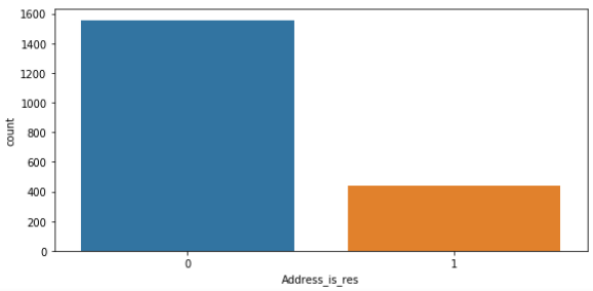


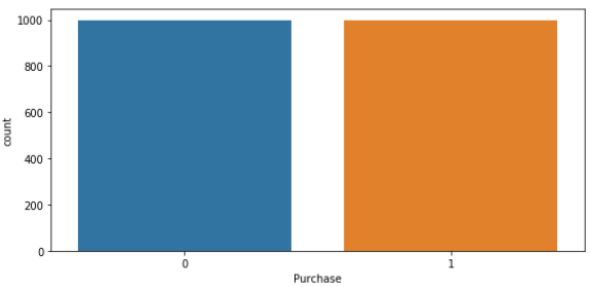




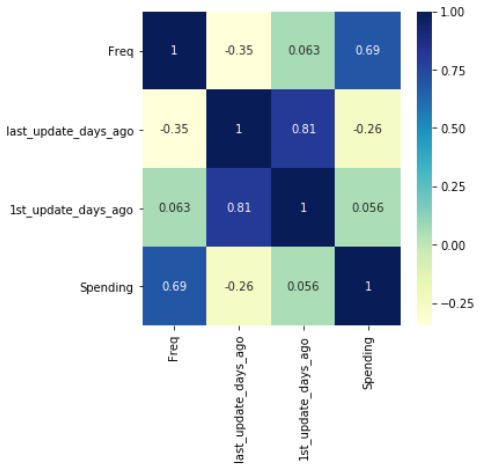








For the variables – gender and purchase we see that the distribution is almost 50:50. The web order variable is slightly imbalanced.



Looking at the correlation heat map for the numeric variables, we see that there are no major high correlation values. Thus, there is no issue of multicollinearity. However, it is important to note that the variable purchase could introduce the problem of leakage into the model. Therefore, I decided to exclude it from the dataset for modeling process.

Modeling:

```
#Linear Regression Model
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split # Split validation class

X = df.iloc[:, 1:-2].values # Use all features as attributes except last 2 column
y = df['Spending'].values      # Set last column as target variable

lr = LinearRegression()

scores_lr = -cross_val_score(lr, X, y, cv=10,scoring ='neg_mean_squared_error')
scores_lr = np.sqrt(scores_lr)
print("Performance: %0.3f (+/- %0.3f)" % (scores_lr.mean(), scores_lr.std() * 2))
```

```
Performance: 125.549 (+/- 49.981)
```

```
#Lasso Regression Model
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=0.1,random_state=42)
scores_lasso = -cross_val_score(lasso, X, y, cv=10,scoring ='neg_mean_squared_error')
scores_lasso = np.sqrt(scores_lasso)
print("Performance: %0.3f (+/- %0.3f)" % (scores_lasso.mean(), scores_lasso.std() * 2))
```

```
Performance: 125.469 (+/- 50.081)
```

```
#Ridge Regression Model
from sklearn.linear_model import Ridge # Ridge Regression class

ridge = Ridge(alpha=1.0,random_state=42)
scores_ridge = -cross_val_score(ridge, X, y, cv=10,scoring ='neg_mean_squared_error')
scores_ridge = np.sqrt(scores_ridge)
print("Performance: %0.3f (+/- %0.3f)" % (scores_ridge.mean(), scores_ridge.std() * 2))
```

```
Performance: 125.528 (+/- 50.027)
```

```
#KNN Regression Model
from sklearn import neighbors
from sklearn.preprocessing import StandardScaler
# Fit regression model
n_neighbors = 5
#Normalize Data
sc = StandardScaler()
#sc.fit(X_train)
sc.fit(X)
X_std = sc.transform(X)

knn = neighbors.KNeighborsRegressor(n_neighbors, weights='distance') #Regression based
scores_knn = -cross_val_score(knn, X_std, y, cv=10,scoring ='neg_mean_squared_error')
scores_knn  = np.sqrt(scores_knn)
print("Performance: %0.3f (+/- %0.3f)" % (scores_knn.mean(), scores_knn.std() * 2))
```

```
Performance: 147.119 (+/- 57.200)
```

```
#Decision tree Regression Model
from sklearn.tree import DecisionTreeRegressor
tree = DecisionTreeRegressor(max_depth=5,random_state=42)
scores_tree = -cross_val_score(tree, X, y, cv=10,scoring ='neg_mean_squared_error')
scores_tree  = np.sqrt(scores_tree)
print("Performance: %0.3f (+/- %0.3f)" % (scores_tree.mean(), scores_tree.std() * 2))
```

```
Performance: 135.742 (+/- 63.272)
```

I ran a total of 5 models on the raw data after excluding the purchase variable. First, I split the data into X and y variables for ease of modeling. In the first model, I ran linear regression and did not specify any parameters, indicating that the model will use default values for the parameters. I then used a 10 fold cross validation to validate generalization performance. The model performance metric being used here is RMSE. Linear regression model gave a mean RMSE of 125.549 +- std deviation of 49.981.

I then ran the lasso regression model and specified alpha as 0.1. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the lasso regression model has a mean RMSE of 125.469 with a std deviation of 50.081.

For ridge regression model, I specified alpha as 1.0. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the ridge regression model has a mean RMSE of 125.528 with a std deviation of 50.027.

For KNN regression model, I first standardized the data using z score scaling. This means each variable column will be subtracted from its mean and divided by then std deviation of the variable. It is important to standardize the variable for KNN so that the distance measure is not affected by the scale of the variables. I specified K= 5 ,which means the model will look for 5 nearest neighbors, and specified weights = 'distance" to make sure that nearer neighbors have more weightage than further neighbors. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the KNN regression model has a mean RMSE of 147.119 with a std deviation of 57.2.

For decision tree regression model, I specified max_depth = 5, indicating maximum no of splits in the tree as 5. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the decision tree regression model has a mean RMSE of 135.656 with a std deviation of 63.245.

Comparing all the models, the lasso regression model has the least RMSE. Lower the RMSE, better the model. The lasso regression model has an RMSE of 125.469, the lowest among all the models. Linear regression and ridge regression models had almost similar RMSE values as lasso. Since lasso also uses auto feature selection to use only useful features, I would prefer to use this model. KNN had the worst RMSE value of 147.119. Decision tree model RMSE of about 135.656 was better than KNN but still higher than the linear models.

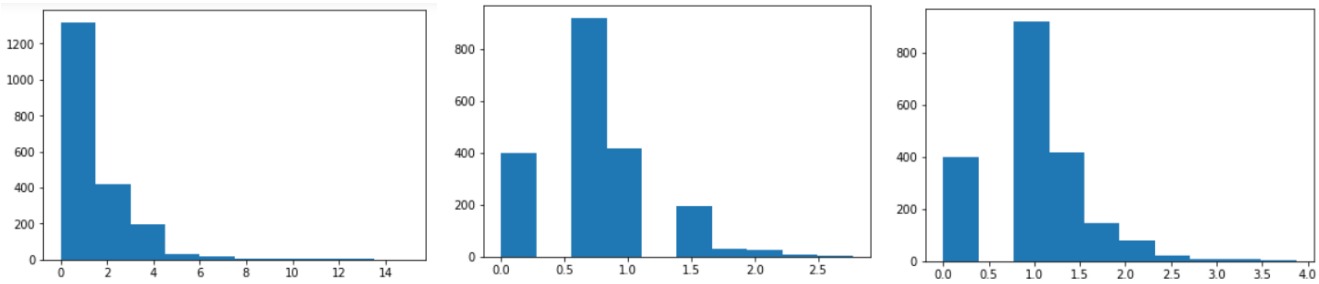
(b) (50 points) Engage in feature engineering (i.e., create new features based on existing features) to optimize the performance of linear regression, k-NN, and regression tree techniques. Present the results for each of the three techniques (choose the best performing model for each technique in case 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]

For feature engineering, I created a duration variable by subtracting 1st_update_days ago and last_update_days ago. I felt this variable could be important as it will capture the time between updates. I also created 2 ratios – last_update_days_ago/Freq and 1st_update_days_ago. I felt these 2 ratios could be useful predictors as they would indicate the updates as a fraction of the frequency.

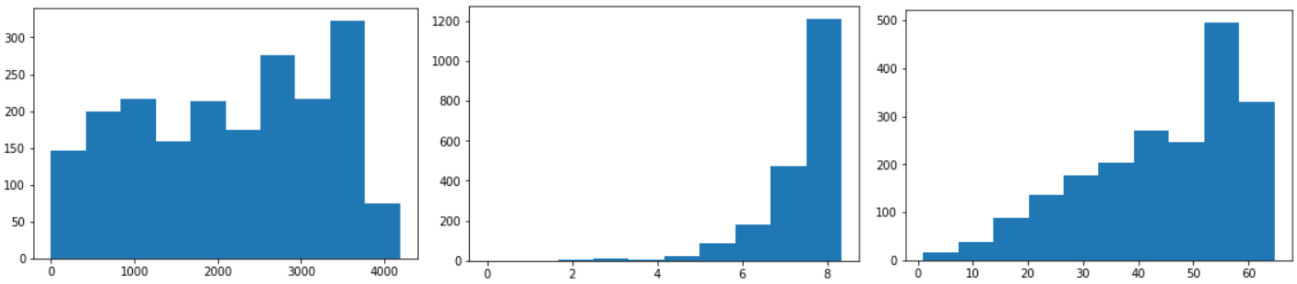
I also observed the distributions of the numeric variables. I then used the log and square root transformations on them. I tried it on the predictor variables – Freq, last_update_days_ago, first_update_days_ago .

For Freq, I noticed that the distribution was right skewed. Using Log and square root, I noticed that the square root transformation had the best effect on the distribution as it looked more normal than the original distribution. So I decided to use the square root of Freq in the model.

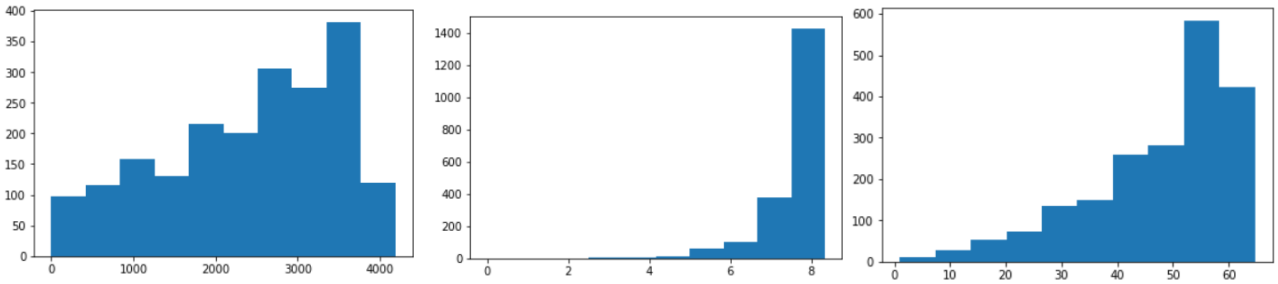


The first histogram plot is freq, second is log(freq+1) and third is sqrt(Freq)

For the variables last_update_days_ago and 1st_update_days_ago, the distribution was not as skewed. I still tried both the log and square transformations, but they made the distributions left skewed. So I decided to leave these variables as is.



From above, first histogram plot is last_update_days_ago, second is log(last_update_days_ago) and third is sqrt(last_update_days_ago)



From above, first histogram plot is 1st_update_days_ago, second is log(1st_update_days_ago) and third is sqrt(1st_update_days_ago)

I then tried the same models on this transformed dataset to see if there was an improvement in performance.

Feature engineering and transformations

```
df1= df.copy()
df1['Duration'] = df1['1st_update_days_ago'] - df1['last_update_days_ago']
df1['ratio1'] = df1['1st_update_days_ago']/df1['Freq']
df1['ratio2'] = df1['last_update_days_ago'] /df1['Freq']
df1['Freq'] = np.sqrt(df1['Freq'])
#df1['Spending'] = np.log(df1["Spending"]+1)
y1= df1['Spending'].values      # Set last column as target variable
df1.drop(['Purchase','Spending'],axis=1, inplace=True)

df1 = df1.replace([np.inf, -np.inf],0)

X1 = df1.iloc[:, 1:].values # Use all features as attributes except last but one column
```

Models on transformed data

```
scores_lr = -cross_val_score(lr, X1, y1, cv=10,scoring = 'neg_mean_squared_error')
scores_lr = np.sqrt(scores_lr)
print("Performance linear: %0.3f (+/- %0.3f)" % (scores_lr.mean(), scores_lr.std() * 2))

scores_lasso = -cross_val_score(lasso, X1, y1, cv=10,scoring = 'neg_mean_squared_error')
scores_lasso = np.sqrt(scores_lasso)
print("Performance lasso: %0.3f (+/- %0.3f)" % (scores_lasso.mean(), scores_lasso.std() * 2))

scores_ridge = -cross_val_score(ridge, X1, y1, cv=10,scoring = 'neg_mean_squared_error')
scores_ridge = np.sqrt(scores_ridge)
print("Performance ridge: %0.3f (+/- %0.3f)" % (scores_ridge.mean(), scores_ridge.std() * 2))

sc.fit(X1)
X1_std = sc.transform(X1)
scores_knn = -cross_val_score(knn, X1_std, y1, cv=10,scoring = 'neg_mean_squared_error')
scores_knn = np.sqrt(scores_knn)
print("Performance knn: %0.3f (+/- %0.3f)" % (scores_knn.mean(), scores_knn.std() * 2))

scores_tree = -cross_val_score(tree, X1, y1, cv=10,scoring = 'neg_mean_squared_error')
scores_tree = np.sqrt(scores_tree)
print("Performance rtree: %0.3f (+/- %0.3f)" % (scores_tree.mean(), scores_tree.std() * 2))
```

```
Performance linear: 132.049 (+/- 50.172)
Performance lasso: 131.964 (+/- 50.323)
Performance ridge: 132.025 (+/- 50.241)
Performance knn: 152.273 (+/- 60.407)
Performance rtree: 141.109 (+/- 69.546)
```

I ran a total of 5 models on the transformed data. In the first model, I ran linear regression and did not specify any parameters, indicating that the model will use default values for the parameters. I then used a 10 fold cross validation to validate generalization performance. The model performance metric being used here is RMSE. Linear regression model gave a mean RMSE value of 132.049+- std deviation of 50.172.

I then ran the lasso regression model and specified alpha as 0.1. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the lasso regression model has a mean RMSE of 131.964 with a std deviation of 50.323.

For ridge regression model, I specified alpha as 1.0. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the ridge regression model has a mean RMSE of 132.025 with a std deviation of 50.241.

For KNN regression model, I first standardized the data using z score scaling. This means each variable column will be subtracted from its mean and divided by then std deviation of the variable. It is important to standardize the variable for KNN so that the distance measure is not affected by the scale of the variables. I specified K= 5 ,which means the model will look for 5 nearest neighbors, and specified weights = ‘distance” to make sure that nearer neighbors have more weightage than further neighbors. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the KNN regression model has a mean RMSE of 152.273 with a std deviation of 60.407.

For decision tree regression model, I specified max_depth = 5, indicating maximum no of splits in the tree as 5. I left all other parameters unspecified, which means the model will use default values for those parameters. Using 10 fold cross validation for generalization performance, the decision tree regression model has a mean RMSE of 141.109 with a std deviation of 69.546.

In comparison to the previous models on raw data, all these models on transformed variables did not perform as well previously. In each model there was an increase in mean RMSE values as compared to previous models on raw data. The transformations did not have an improvement in model performance – sometimes this does happen because of the way the data is. I expected that the transformations would benefit the linear models- linear regression, lasso and ridge regression, since the transformations made the variable distributions more normal. However, it was not the case, maybe because the distributions of the variables were not perfectly normal. It could also be that the cumulative effect of these variables had an influence on the result. I did not expect much change in the knn and decision tree models, since they handle non linearity well. In general, transformations do not have an effect on non linear models like KNN and decision trees.

Comparing all the new models, the lasso regression model has the least RMSE value. The linear and ridge regression models have RMSE almost the same as lasso. Decision tree model mean RMSE of about 141.109 was better than KNN but still higher than the linear regression and ridge models. KNN had the highest mean RMSE value. Among these models, I would pick the lasso regression model since it has the least mean RMSE value. It also has the added advantage of using automatic feature selection.

(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]

In order to carry out parameter tuning, I used the grid search method along with nested cross validation. First I used an inner 5 fold cross validation to find optimum hyperparameters for each model. Then I used an outer 5 fold cross validation to estimate the generalization performance of that model with optimum hyperparameters.
First I ran the grid search for Linear regression model. I chose parameters to tune as normalize= True or False, which indicates whether to normalize the data or not as part of the modeling process. All other model parameters for the model are left unspecified, so the model will consider default values. As a result of grid search, the optimum parameter is Normalize = False and it has a nested cross validation performance of 134.135 mean RMSE.
I then ran a linear regression model with the optimum parameters from grid search with a 10 fold cross validation set to estimate generalization performance. The model had a mean RMSE of 132.049.

```
# linear regression parameter tuning
from sklearn.model_selection import GridSearchCV,KFold
inner_cv = KFold(n_splits=5, shuffle=True,random_state=42)
outer_cv = KFold(n_splits=5, shuffle=True,random_state=42)
parameters = {'normalize':['False','True']}      # range of parameters for the depth of the tree
gs_dt = GridSearchCV(LinearRegression(), parameters, n_jobs=5,scoring = 'neg_mean_squared_error') #GridS
gs_dt.fit(X1, y1) # Fit model

print(" Parameter Tuning")
print("Non-nested Performance: ", np.sqrt(-gs_dt.best_score_))
print("Optimal Parameter: ", gs_dt.best_params_)    # Parameter setting that gave the best results on th
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. esti
# Outer CV
nested_score_gs_dt = -cross_val_score(gs_dt, X=X, y=y, cv=outer_cv,scoring = 'neg_mean_squared_error')
nested_score_gs_dt = np.sqrt(nested_score_gs_dt)
print("Nested CV Performance: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
 Parameter Tuning
Non-nested Performance:  134.6281967746648
Optimal Parameter:  {'normalize': 'False'}
Optimal Estimator:  LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize='False')
Nested CV Performance:  134.13533646927027  +/-  8.977786085973621
```

```
scores_lr = -cross_val_score(LinearRegression(normalize=False), X1, y1, cv=10,scoring='neg_mean_squared_
scores_lr = np.sqrt(scores_lr)
print("Performance: %0.3f (+/- %0.3f)" % (scores_lr.mean(), scores_lr.std() * 2))
```

```
Performance: 132.049 (+/- 50.172)
```

For Lasso regression model, I chose parameters to tune as normalize= True or False, which indicates whether to normalize the data or not as part of the modeling process. I also gave a range of values for alpha as input to tune. All other model parameters for the model are left unspecified, so the model will consider default values. As a result of

grid search, the optimum parameter is Normalize = False, alpha =0.001 - and it has a nested cross validation performance of 134.119 mean RMSE. I then ran a lasso regression model with the optimum parameters from grid search with a 10 fold cross validation set to estimate generalization performance. The model had a mean RMSE of 132.048.

Parameter tuning - lasso regression

```
# Lasso regression parameter tuning
parameters = {'normalize':['False','True'],'alpha':[1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 1, 5, 10]}      # range of paramet
gs_dt = GridSearchCV(Lasso(random_state=42), parameters, n_jobs=5,scoring='neg_mean_squared_error') #GridSearchCV
gs_dt.fit(X1, y1) # Fit model

print(" Parameter Tuning")
print("Non-nested Performance: ", np.sqrt(-gs_dt.best_score_))
print("Optimal Parameter: ", gs_dt.best_params_)    # Parameter setting that gave the best results on the hold out data.
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest
# Outer CV
nested_score_gs_dt = -cross_val_score(gs_dt, X=X1, y=y1, cv=outer_cv,scoring='neg_mean_squared_error')
nested_score_gs_dt = np.sqrt(nested_score_gs_dt)
print("Nested CV Performance: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
 Parameter Tuning
Non-nested Performance:  134.6239906372296
Optimal Parameter:  {'alpha': 0.001, 'normalize': 'False'}
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 Performance:  134.11910809403417  +/-  8.989408321094544
```

```
scores_la = -cross_val_score(Lasso(normalize=False,alpha= 0.001,random_state=42), X1, y1, cv=10,scoring='neg_mean_squared_error'
scores_la= np.sqrt(scores_la)
print("Performance: %0.3f (+/- %0.3f)" % (scores_la.mean(), scores_la.std() * 2))
```

```
Performance: 132.048 (+/- 50.173)
```

For ridge regression model, I chose parameters to tune as normalize= True or False, which indicates whether to normalize the data or not as part of the modeling process. I also gave a range of values for alpha as input to tune. All other model parameters for the model are left unspecified, so the model will consider default values. As a result of grid search, the optimum parameter is Normalize = False, alpha =0.0001 - and it has a nested cross validation performance of 134.07 mean RMSE. I then ran a model with the optimum parameters from grid search with a 10 fold cross validation set to estimate generalization performance. The model had a mean RMSE of 132.049.

Parameter tuning - Ridge regression

```
# ridge regression parameter tuning
parameters = {'normalize':['False','True'],'alpha':[1e-15, 1e-10, 1e-8, 1e-5,1e-4, 1e-3,1e-2, 1, 5, 10]}      # range of paramet
gs_dt = GridSearchCV(Ridge(random_state=42), parameters, n_jobs=5,scoring='neg_mean_squared_error') #GridSearchCV
gs_dt.fit(X1, y1) # Fit model

print(" Parameter Tuning")
print("Non-nested Performance: ", np.sqrt(-gs_dt.best_score_))
print("Optimal Parameter: ", gs_dt.best_params_)    # Parameter setting that gave the best results on the hold out data.
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest
# Outer CV
nested_score_gs_dt = -cross_val_score(gs_dt, X=X1, y=y1, cv=outer_cv,scoring='neg_mean_squared_error')
nested_score_gs_dt = np.sqrt(nested_score_gs_dt)
print("Nested CV Performance: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
 Parameter Tuning
Non-nested Performance:  134.626022572998
Optimal Parameter:  {'alpha': 0.0001, 'normalize': 'False'}
Optimal Estimator:  Ridge(alpha=0.0001, copy_X=True, fit_intercept=True, max_iter=None,
   normalize='False', random_state=42, solver='auto', tol=0.001)
Nested CV Performance:  134.07593270844978  +/-  8.994662596704257
```

```
scores_r = -cross_val_score(Ridge(normalize=False,alpha= 0.0001,random_state=42), X1, y1, cv=10,scoring='neg_mean_squared_error'
scores_r =np.sqrt(scores_r)
print("Performance: %0.3f (+/- %0.3f)" % (scores_r.mean(), scores_r.std() * 2))
```

```
Performance: 132.049 (+/- 50.172)
```

For decision tree regression model, in the parameters to optimize I have mentioned maximum depth of tree with range of values from 3 to 50, minimum no of samples to split an internal node with range of values= (2,3,4,5,6,7,8,9,10) and minimum no of samples for a leaf node with range of values= (1,2,3,4,5,6,7,8,9,10). All other model parameters for the decision tree model are left unspecified, so the model will consider default values. As a result of grid search, the optimum parameters are max_depth=10, min_samples_leaf=9, min_samples_split=8 - and it has a nested cross validation performance of 133.43 mean RMSE. I then ran a model with the optimum parameters from grid search with a 10 fold cross validation set to estimate generalization performance. The model had a mean RMSE of 136.368.

```
# Tree model parameter tuning

parameters = {'max_depth':range(3,50),'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10],
                           'min_samples_split':[2,3,4,5,6,7,8,9,10]}      # range of parameters for the depth of the tree
gs_dt = GridSearchCV(DecisionTreeRegressor(random_state=42), parameters, n_jobs=4,scoring='neg_mean_squared_error') #GridSearchC
gs_dt.fit(X1, y1) # Fit model

print(" Parameter Tuning")
print("Non-nested Performance: ", np.sqrt(-gs_dt.best_score_))
print("Optimal Parameter: ", gs_dt.best_params_)    # Parameter setting that gave the best results on the hold out data.
print("Optimal Estimator: ", gs_dt.best_estimator_) # Estimator that was chosen by the search, i.e. estimator which gave highest
# Outer CV
nested_score_gs_dt = -cross_val_score(gs_dt, X=X1, y=y1, cv=outer_cv,scoring='neg_mean_squared_error')
nested_score_gs_dt = np.sqrt(nested_score_gs_dt)
print("Nested CV Performance: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
 Parameter Tuning
Non-nested Performance:  138.11584197120405
Optimal Parameter:  {'max_depth': 10, 'min_samples_leaf': 9, 'min_samples_split': 2}
Optimal Estimator:  DecisionTreeRegressor(criterion='mse', max_depth=10, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=9,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=42, splitter='best')
Nested CV Performance:  133.4362162864122  +/-  18.20458380612317
```

```
scores_dt = -cross_val_score(DecisionTreeRegressor(max_depth=10,min_samples_leaf=9,min_samples_split=2), X1, y1, cv=10,scoring='
scores_dt = np.sqrt(scores_dt)
print("Performance: %0.3f (+/- %0.3f)" % (scores_dt.mean(), scores_dt.std() * 2))
```

```
Performance: 136.368 (+/- 47.338)
```

For KNN regression model, in the parameters to optimize I have mentioned a range of values for no of neighbors and weights: uniform and distance. All other model parameters for the model are left unspecified, so the model will consider default values. As a result of grid search, the optimum parameters k=17, weights = uniform - and it has a nested cross validation performance of 148.01 mean RMSE. I then ran a model with the optimum parameters from grid search with a 10 fold cross validation set to estimate generalization performance. The model had a mean RMSE of 145.49.

```
# Choosing k for kNN AND type of distance
gs_knn = GridSearchCV(estimator=neighbors.KNeighborsRegressor(p=2,
                           metric='minkowski'),
                  param_grid=[{'n_neighbors': [1,3,5,7,9,11,13,15,17,19,21,23,25,27,29],
                               'weights':['uniform','distance']}],
                  cv=inner_cv,
                  n_jobs=5,scoring='neg_mean_squared_error')

gs_knn = gs_knn.fit(X1_std,y1)
print("\n Parameter Tuning - KNN algorithm")
print("Non-nested CV F1 Score: ", np.sqrt(-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 highes
nested_score_gs_knn = -cross_val_score(gs_knn, X=X1_std, y=y1, cv=outer_cv,scoring='neg_mean_squared_error')
nested_score_gs_knn = np.sqrt(nested_score_gs_knn)
print("Nested CV F1 Score: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
```

```
 Parameter Tuning - KNN algorithm
Non-nested CV F1 Score:  148.19420733280924
Optimal Parameter:  {'n_neighbors': 17, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=17, p=2,
          weights='uniform')
Nested CV F1 Score:  148.01883783740783  +/-  14.254301643338572
```

```
scores_kn = -cross_val_score(neighbors.KNeighborsRegressor(n_neighbors=17,weights='uniform',p=2,metric='minkowski'), X1_std, y1,
scores_kn = np.sqrt(scores_kn)
print("Performance: %0.3f (+/- %0.3f)" % (scores_kn.mean(), scores_kn.std() * 2))
```

```
Performance: 145.490 (+/- 63.781)
```

Comparing all the models, lasso model had the least mean RMSE value of 132.048. The linear, ridge regression models had similar RMSE values as lasso. The KNN model had the highest mean RMSE value of 145.49. Among all models I would choose to use lasso model. It has the lowest mean RMSE value among all models and it also has the advantage of auto feature selection.