EMORY UNIVERSITY

Introduction to Business Data Analytics

Homework #4 Part 2

_______Qiyu Wang_______

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

Total grade: ________ out of ___145___ points

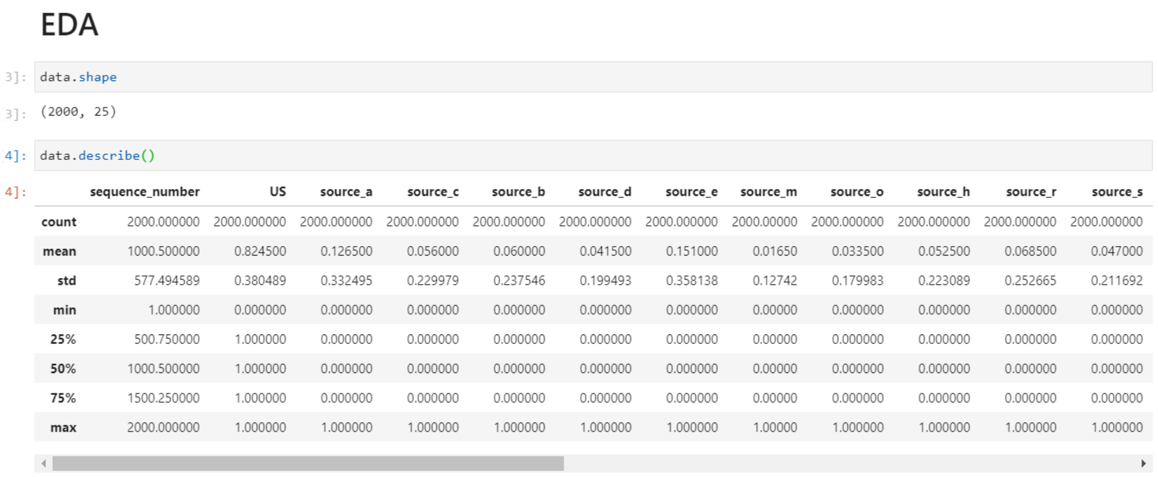
(145 points) Use numeric prediction techniques to build a predictive model for the HW4.xlsx dataset. This dataset is provided on Canvas and contains data about whether or not different consumers made a purchase in response to a test mailing of a certain catalog and, in case of a purchase, how much money each consumer spent. The data file has a brief description of all the attributes in a separate worksheet. We would like to build predictive models to predict how much will the customers spend; Spending is the target variable (numeric value: amount spent).

Use Python for this exercise.

<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]



EDA

```
3]: data.shape

3]: (2000, 25)

4]: data.describe()
```

4]:

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	source_s
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.00000	2000.000000	2000.000000	2000.000000	2000.000000
mean	1000.500000	0.824500	0.126500	0.056000	0.060000	0.041500	0.151000	0.01650	0.033500	0.052500	0.068500	0.047000
std	577.494589	0.380489	0.332495	0.229979	0.237546	0.199493	0.358138	0.12742	0.179983	0.223089	0.252665	0.211692
min	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
25%	500.750000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
50%	1000.500000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
75%	1500.250000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000
max	2000.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.00000	1.000000	1.000000	1.000000	1.000000

First, I checked the shape and descriptive table, where I found the table has 2000 rows, and data are quite well bounded. There's no significant outliers based on the MIN/MAX//mean/Std.

We can see that most variables are bounded binary from those statistics. Meanwhile, Sequence number seems to be meaningless (uniform distributed and have interval of one), and "Spending", "Freq" are the only columns that are highly skewed.

```
data.dtypes
```

```
sequence_number           int64
US                        int64
source_a                  int64
source_c                  int64
source_b                  int64
source_d                  int64
source_e                  int64
source_m                  int64
source_o                  int64
source_h                  int64
source_r                  int64
source_s                  int64
source_t                  int64
source_u                  int64
source_p                  int64
source_x                  int64
source_w                  int64
Freq                      int64
last_update_days_ago      int64
1st_update_days_ago       int64
Web order                 int64
Gender=male               int64
Address_is_res            int64
Purchase                  int64
Spending                float64
dtype: object
```

```
data.isnull().sum()
```

```
sequence_number         0
US                      0
source_a                0
source_c                0
source_b                0
source_d                0
source_e                0
source_m                0
source_o                0
source_h                0
source_r                0
source_s                0
source_t                0
source_u                0
source_p                0
source_x                0
source_w                0
Freq                    0
last_update_days_ago    0
1st_update_days_ago     0
Web order               0
Gender=male             0
Address_is_res          0
Purchase                0
Spending                0
dtype: int64
```

```
data.head()
```

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	source_r	source_s	source_t	sc
0	1	1	0	0	1	0	0	0	0	0	0	0	0	
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	1	
3	4	1	0	1	0	0	0	0	0	0	0	0	0	
4	5	1	0	1	0	0	0	0	0	0	0	0	0	

Then I check the datatype of each column, and the number of null values in each column, and the first 5 rows.

```
del df['Purchase']
```

```
data['sequence_number'].head()
```

```
0    1
1    2
2    3
3    4
4    5
Name: sequence_number, dtype: int64
```

```
del df['sequence_number']
```

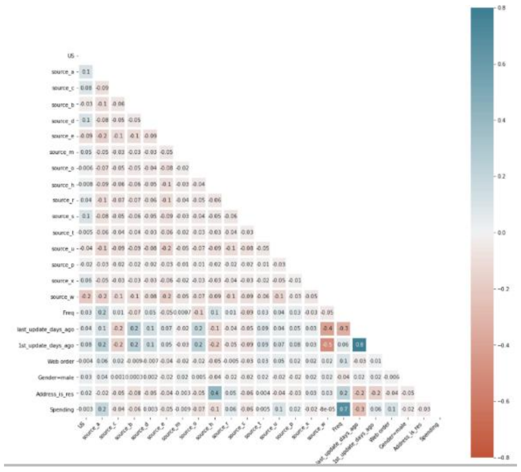
Proves me that sequence_number is just a index number, so I removed it. Also "Purchase" is not a information that's available at the point of prediction, so I removed it too.

```
sum(data.iloc[:,2:2+15].sum())
```

```
1910
```

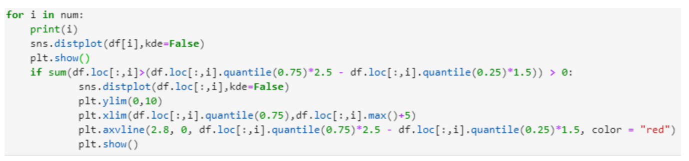
Moreover, I wasn't so sure if those 15 sources columns are dummies of a single categorical column, which might create multicolinearity problem. So I added up all the sources, turns out it's only 1910, which means there's still a base level, so it wou't be problematic.

```
mask = np.triu(np.ones_like(corr))

plt.figure(figsize=(16, 16))
ax = sns.heatmap(
    corr,
    vmin=-.8, vmax=.8, center=0,
    cmap= sns.diverging_palette(20,220, n=200),
    square = True,
    linewidth=4,
    annot = True,
    fmt='.1g',
    mask = mask

    )

ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=45,
    horizontalalignment = 'right'
    )
```



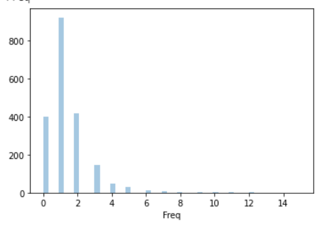
Right after that, I did a correlation matrix, and I figured out that "last_update_days_ago" and "1st_update_days_ago" are highly correlation. And "last_update_days_ago" is a better predictor for "Spending", so I removed "last_update_days_ago".

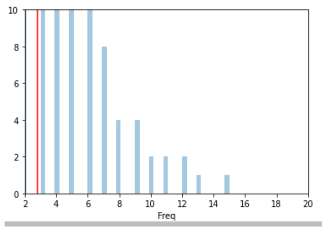
```
[19]: del df['1st_update_days_ago']
```



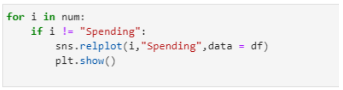
```
for i in num:
    print(i)
    sns.distplot(df[i],kde=False)
    plt.show()
    if sum(df.loc[:,i]>(df.loc[:,i].quantile(0.75)*2.5 - df.loc[:,i].quantile(0.25)*1.5)) > 0:
            sns.distplot(df.loc[:,i],kde=False)
            plt.ylim(0,10)
            plt.xlim(df.loc[:,i].quantile(0.75),df.loc[:,i].max()+5)
            plt.axvline(2.8, 0, df.loc[:,i].quantile(0.75)*2.5 - df.loc[:,i].quantile(0.25)*1.5, color = "red")
            plt.show()
```

```
Freq
```

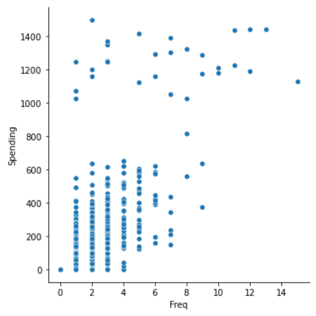




Then, I did histogram and scatterplots on variables to see the distribution, correlation, and the outliers.

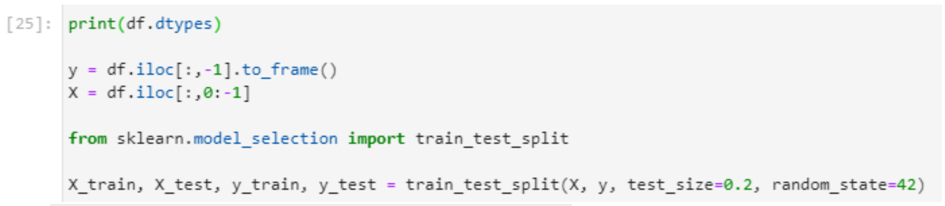


```
for i in num:
    if i != "Spending":
        sns.relplot(i,"Spending",data = df)
        plt.show()
```



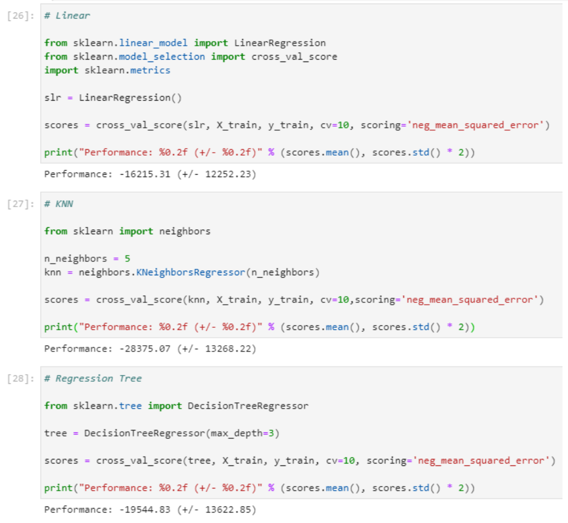
The red line shows 1.5 IQR from 75% Quartile which might be considered as outliers, however, it's all continuous/continuity, so I didn't feel that simple removal will be a good choice, so I kept them. And it happens to "spending" too.

Based on Scatterplot, we see that there's a bit of clustering in the conner, and might be overall polynomial relationships, so I definitely want to create features on "Freq" later on.

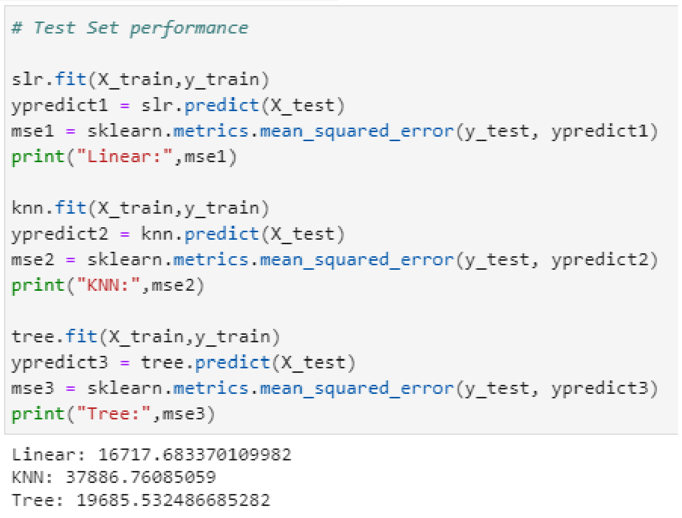


```
[25]: print(df.dtypes)

      y = df.iloc[:,-1].to_frame()
      X = df.iloc[:,0:-1]

      from sklearn.model_selection import train_test_split

      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Here I splited test/train set.



```
[26]: # Linear

      from sklearn.linear_model import LinearRegression
      from sklearn.model_selection import cross_val_score
      import sklearn.metrics

      slr = LinearRegression()

      scores = cross_val_score(slr, X_train, y_train, cv=10, scoring='neg_mean_squared_error')

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

      Performance: -16215.31 (+/- 12252.23)

[27]: # KNN

      from sklearn import neighbors

      n_neighbors = 5
      knn = neighbors.KNeighborsRegressor(n_neighbors)

      scores = cross_val_score(knn, X_train, y_train, cv=10,scoring='neg_mean_squared_error')

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

      Performance: -28375.07 (+/- 13268.22)

[28]: # Regression Tree

      from sklearn.tree import DecisionTreeRegressor

      tree = DecisionTreeRegressor(max_depth=3)

      scores = cross_val_score(tree, X_train, y_train, cv=10, scoring='neg_mean_squared_error')

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

      Performance: -19544.83 (+/- 13622.85)
```



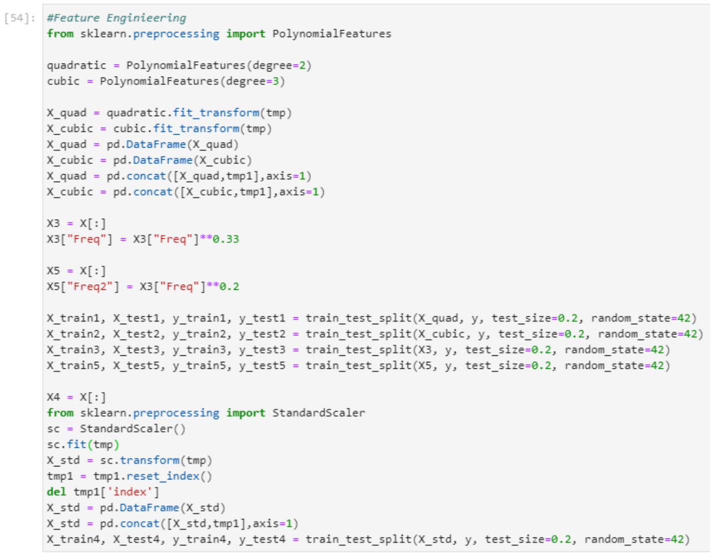
```
# Test Set performance

slr.fit(X_train,y_train)
ypredict1 = slr.predict(X_test)
mse1 = sklearn.metrics.mean_squared_error(y_test, ypredict1)
print("Linear:",mse1)

knn.fit(X_train,y_train)
ypredict2 = knn.predict(X_test)
mse2 = sklearn.metrics.mean_squared_error(y_test, ypredict2)
print("KNN:",mse2)

tree.fit(X_train,y_train)
ypredict3 = tree.predict(X_test)
mse3 = sklearn.metrics.mean_squared_error(y_test, ypredict3)
print("Tree:",mse3)

Linear: 16717.683370109982
KNN: 37886.76085059
Tree: 19685.532486685282
```

Here, I trained three simple model with all variables remained and default parameters. Linear performed way better than the other two based on MSE scores. Linear model has lowest average error (MSE) base on 10-fold cv, and lowest variance on performance too.

(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



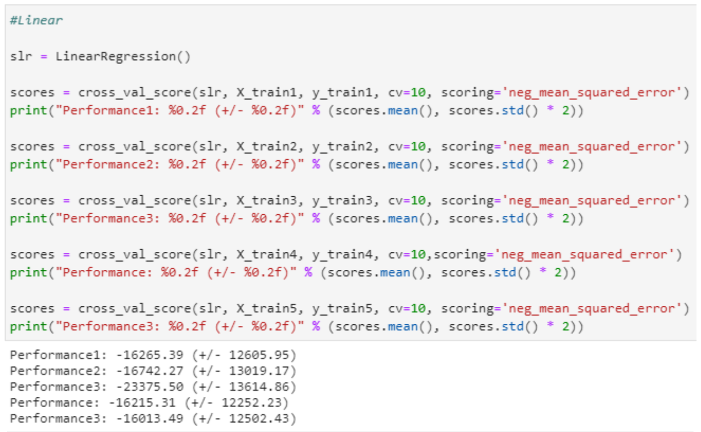
```
[54]: #Feature Enginieering
from sklearn.preprocessing import PolynomialFeatures

quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)

X_quad = quadratic.fit_transform(tmp)
X_cubic = cubic.fit_transform(tmp)
X_quad = pd.DataFrame(X_quad)
X_cubic = pd.DataFrame(X_cubic)
X_quad = pd.concat([X_quad,tmp1],axis=1)
X_cubic = pd.concat([X_cubic,tmp1],axis=1)

X3 = X[:]
X3["Freq"] = X3["Freq"]**0.33

X5 = X[:]
X5["Freq2"] = X3["Freq"]**0.2

X_train1, X_test1, y_train1, y_test1 = train_test_split(X_quad, y, test_size=0.2, random_state=42)
X_train2, X_test2, y_train2, y_test2 = train_test_split(X_cubic, y, test_size=0.2, random_state=42)
X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y, test_size=0.2, random_state=42)
X_train5, X_test5, y_train5, y_test5 = train_test_split(X5, y, test_size=0.2, random_state=42)

X4 = X[:]
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(tmp)
X_std = sc.transform(tmp)
tmp1 = tmp1.reset_index()
del tmp1['index']
X_std = pd.DataFrame(X_std)
X_std = pd.concat([X_std,tmp1],axis=1)
X_train4, X_test4, y_train4, y_test4 = train_test_split(X_std, y, test_size=0.2, random_state=42)
```

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]

Here, I engineered 5 different sets of features using different methods:

1. Quadratic set using PolynomialFeature package.
2. Cubic set using PolynomialFeature package.
3. Transformation on Freq to make it more normal distributed.
4. Standardized dataset (mainly for KNN and Tree)
5. Manually created a quadratic term for Freq.

Then I runed each model with default parameters on those five sets of features and to see which set gets the best performance on each model.

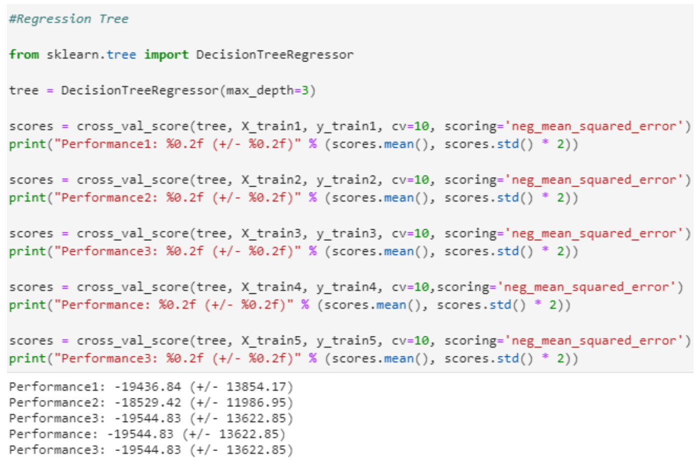


```
#Linear

slr = LinearRegression()

scores = cross_val_score(slr, X_train1, y_train1, cv=10, scoring='neg_mean_squared_error')
print("Performance1: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(slr, X_train2, y_train2, cv=10, scoring='neg_mean_squared_error')
print("Performance2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(slr, X_train3, y_train3, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(slr, X_train4, y_train4, cv=10,scoring='neg_mean_squared_error')
print("Performance: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(slr, X_train5, y_train5, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Performance1: -16265.39 (+/- 12605.95)
Performance2: -16742.27 (+/- 13019.17)
Performance3: -23375.50 (+/- 13614.86)
Performance: -16215.31 (+/- 12252.23)
Performance3: -16013.49 (+/- 12502.43)
```

First, I runed the five sets using Linear.

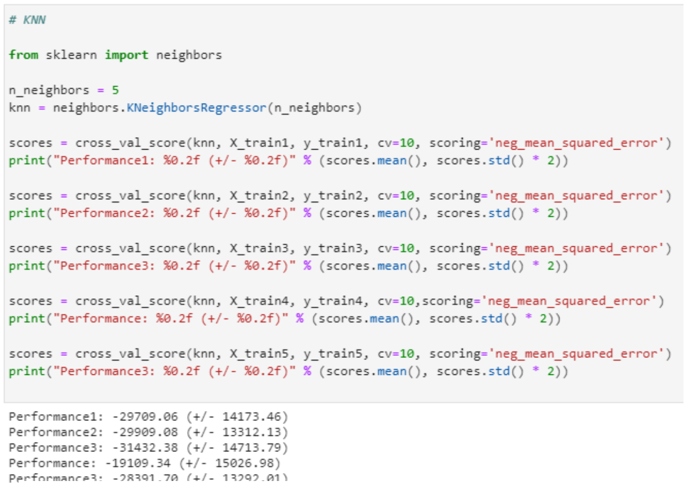
There's slight improvement over the last model in term of error, but the models become less stable (std).



```
#Regression Tree

from sklearn.tree import DecisionTreeRegressor

tree = DecisionTreeRegressor(max_depth=3)

scores = cross_val_score(tree, X_train1, y_train1, cv=10, scoring='neg_mean_squared_error')
print("Performance1: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(tree, X_train2, y_train2, cv=10, scoring='neg_mean_squared_error')
print("Performance2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(tree, X_train3, y_train3, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(tree, X_train4, y_train4, cv=10,scoring='neg_mean_squared_error')
print("Performance: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(tree, X_train5, y_train5, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Performance1: -19436.84 (+/- 13854.17)
Performance2: -18529.42 (+/- 11986.95)
Performance3: -19544.83 (+/- 13622.85)
Performance: -19544.83 (+/- 13622.85)
Performance3: -19544.83 (+/- 13622.85)
```

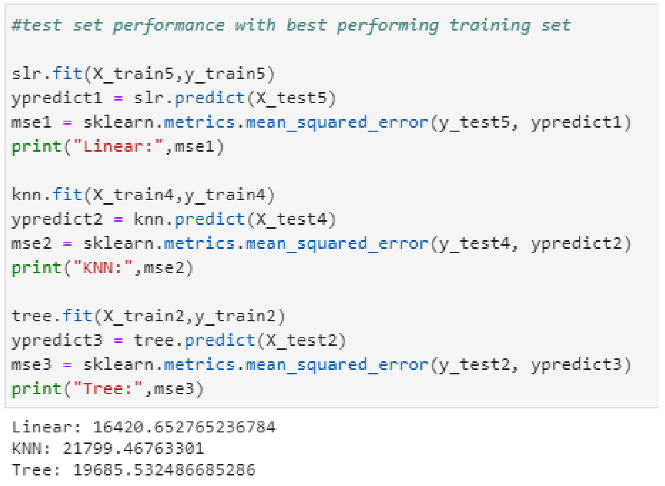
Decision tree has larger improvement than linear. On the set, where I created two quadratic feature with PolynomialFeature, the MSE got lower by 1000, and the model become more stable, where the std lowered by 1400.



```
# KNN

from sklearn import neighbors

n_neighbors = 5
knn = neighbors.KNeighborsRegressor(n_neighbors)

scores = cross_val_score(knn, X_train1, y_train1, cv=10, scoring='neg_mean_squared_error')
print("Performance1: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(knn, X_train2, y_train2, cv=10, scoring='neg_mean_squared_error')
print("Performance2: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(knn, X_train3, y_train3, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(knn, X_train4, y_train4, cv=10,scoring='neg_mean_squared_error')
print("Performance: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

scores = cross_val_score(knn, X_train5, y_train5, cv=10, scoring='neg_mean_squared_error')
print("Performance3: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
Performance1: -29709.06 (+/- 14173.46)
Performance2: -29909.08 (+/- 13312.13)
Performance3: -31432.38 (+/- 14713.79)
Performance: -19109.34 (+/- 15026.98)
Performance3: -28391.70 (+/- 13292.01)
```

KNN has the largest improvement over the three models, with the 9000 increase on standardized data in MSE, however the model become a bit more unstable by increasing the std of performance.



```
#test set performance with best performing training set

slr.fit(X_train5,y_train5)
ypredict1 = slr.predict(X_test5)
mse1 = sklearn.metrics.mean_squared_error(y_test5, ypredict1)
print("Linear:",mse1)

knn.fit(X_train4,y_train4)
ypredict2 = knn.predict(X_test4)
mse2 = sklearn.metrics.mean_squared_error(y_test4, ypredict2)
print("KNN:",mse2)

tree.fit(X_train2,y_train2)
ypredict3 = tree.predict(X_test2)
mse3 = sklearn.metrics.mean_squared_error(y_test2, ypredict3)
print("Tree:",mse3)
```

```
Linear: 16420.652765236784
KNN: 21799.46763301
Tree: 19685.532486685286
```

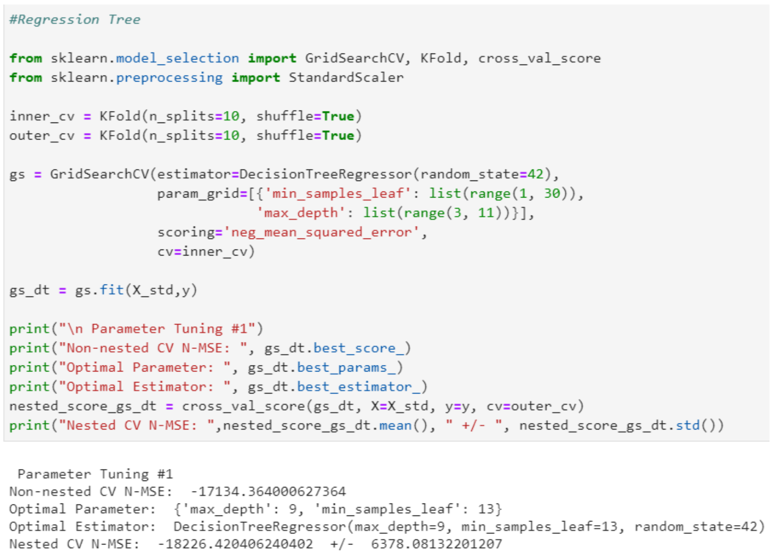
So I go ahead and trained everything on entire training set using their individual best performance feature engineering set.

The Linear model is still the best performer although didn't improve much in this stage. I Suppose that the relationships before X's and Y were already quite linear, so that complex features doesn't provide much more information. Or the provided feature didn't capture that pattern well.

For the KNN, Standardization improved it's similarity metrics by a lot. For Tree, the performance dropped on test set, and went back to original level, so it might be something associated with the randomly generated validation.

(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]

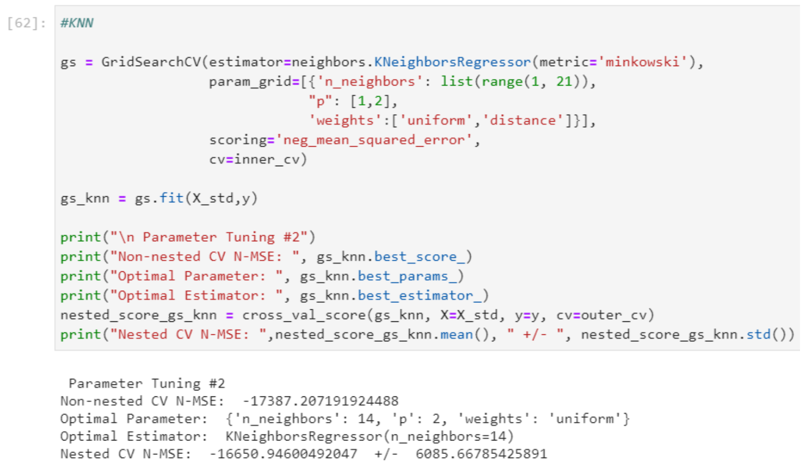


```
#Regression Tree

from sklearn.model_selection import GridSearchCV, KFold, cross_val_score
from sklearn.preprocessing import StandardScaler

inner_cv = KFold(n_splits=10, shuffle=True)
outer_cv = KFold(n_splits=10, shuffle=True)

gs = GridSearchCV(estimator=DecisionTreeRegressor(random_state=42),
                  param_grid=[{'min_samples_leaf': list(range(1, 30)),
                               'max_depth': list(range(3, 11))}],
                  scoring='neg_mean_squared_error',
                  cv=inner_cv)

gs_dt = gs.fit(X_std,y)

print("\n Parameter Tuning #1")
print("Non-nested CV N-MSE: ", gs_dt.best_score_)
print("Optimal Parameter: ", gs_dt.best_params_)
print("Optimal Estimator: ", gs_dt.best_estimator_)
nested_score_gs_dt = cross_val_score(gs_dt, X=X_std, y=y, cv=outer_cv)
print("Nested CV N-MSE: ",nested_score_gs_dt.mean(), " +/- ", nested_score_gs_dt.std())
```

```
 Parameter Tuning #1
Non-nested CV N-MSE:  -17134.364000627364
Optimal Parameter:  {'max_depth': 9, 'min_samples_leaf': 13}
Optimal Estimator:  DecisionTreeRegressor(max_depth=9, min_samples_leaf=13, random_state=42)
Nested CV N-MSE:  -18226.420406240402  +/-  6378.08132201207
```

For this part, I runed 10-fold nested cv for parameter tuning. Where I pick the best performing dataset in Question 2 as the dataset for each individual model.

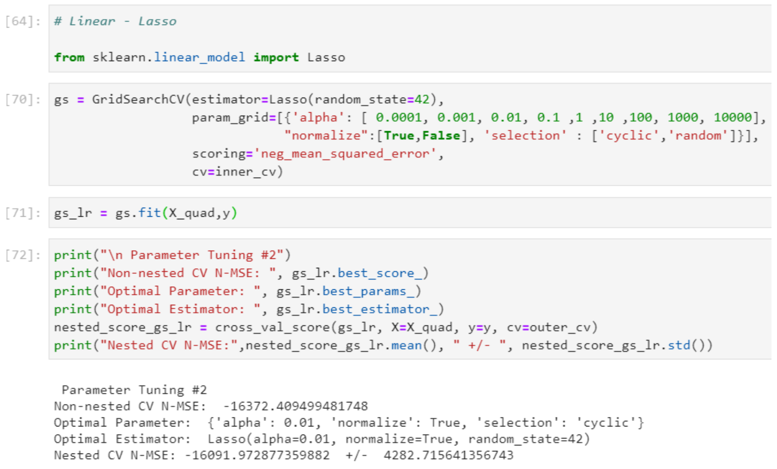
For the first model grid search returns a model with {'max_depth': 9, 'min_samples_leaf': 13} on the standardized data.
As setting caps on depths and min leaf sample, we probably eliminate some overfitting issues, which returns us a much better MSE (3000 lower than pervious). And more important, the model is times more stable than before with std of MSE of only 6000.



```
[62]: #KNN

      gs = GridSearchCV(estimator=neighbors.KNeighborsRegressor(metric='minkowski'),
                        param_grid=[{'n_neighbors': list(range(1, 21)),
                                     "p": [1,2],
                                     'weights':['uniform','distance']}],
                        scoring='neg_mean_squared_error',
                        cv=inner_cv)

      gs_knn = gs.fit(X_std,y)

      print("\n Parameter Tuning #2")
      print("Non-nested CV N-MSE: ", gs_knn.best_score_)
      print("Optimal Parameter: ", gs_knn.best_params_)
      print("Optimal Estimator: ", gs_knn.best_estimator_)
      nested_score_gs_knn = cross_val_score(gs_knn, X=X_std, y=y, cv=outer_cv)
      print("Nested CV N-MSE: ",nested_score_gs_knn.mean(), " +/- ", nested_score_gs_knn.std())
```

```
 Parameter Tuning #2
Non-nested CV N-MSE:  -17387.207191924488
Optimal Parameter:  {'n_neighbors': 14, 'p': 2, 'weights': 'uniform'}
Optimal Estimator:  KNeighborsRegressor(n_neighbors=14)
Nested CV N-MSE:  -16650.94600492047  +/-  6085.66785425891
```

KNN model is also applied on standardized data.
Similar thing happens to KNN, as adding parameter to fix overfitting issues, the performance scores on validation and test are significantly higher and more stable. The grid search returns a model with 14 k:
{'n_neighbors': 14, 'p': 2, 'weights': 'uniform'}
It shows that it yields even better performance than regression tree, where KNN yields lower errors and less variance.



```
[64]: # Linear - Lasso

      from sklearn.linear_model import Lasso

[70]: gs = GridSearchCV(estimator=Lasso(random_state=42),
                        param_grid=[{'alpha': [ 0.0001, 0.001, 0.01, 0.1 ,1 ,10 ,100, 1000, 10000],
                                     "normalize":[True,False], 'selection' : ['cyclic','random']}],
                        scoring='neg_mean_squared_error',
                        cv=inner_cv)

[71]: gs_lr = gs.fit(X_quad,y)

[72]: print("\n Parameter Tuning #2")
      print("Non-nested CV N-MSE: ", 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=X_quad, y=y, cv=outer_cv)
      print("Nested CV N-MSE:",nested_score_gs_lr.mean(), " +/- ", nested_score_gs_lr.std())
```

```
 Parameter Tuning #2
Non-nested CV N-MSE:  -16372.409499481748
Optimal Parameter:  {'alpha': 0.01, 'normalize': True, 'selection': 'cyclic'}
Optimal Estimator:  Lasso(alpha=0.01, normalize=True, random_state=42)
Nested CV N-MSE: -16091.972877359882  +/-  4282.715641356743
```

For the linear regression, I did a lasso regression, because I added quadratic features to it, and also there's 15 binary classes for sources. So I would like to know if any of those doesn't make sense to be included and cause overfitting. It returns me a lamda of 0.01, and normalized model. The performance is the best out of three, more accurate and more stable.

{'alpha': 0.01, 'normalize': True, 'selection': 'cyclic'}

I suppose the reason linear model is doing so well on this problem is that the relationship in this model is relatively more linear. We have almost all our variables as binary, and only two numeric, which doesn't have a distinct polynomial shape in scatterplots also. Moreover, having all the binary can potentially limits the performance of KNN and Trees, because instead of split on mult-categories, now they have to split on binary, which limits the amounts of information included in each splits. However, Linear model works great with binarys.