# Final Project

December 3, 2020

Stuart Miller, Justin Howard, Paul Adams

## 1 Introduction

To be competitive within our industry, gaining rapid insights from generating useful models is not only profitable, but required for long-term success. Therefore, we have decided to implement machine learning into our operations to provide solutions at which we otherwise would be unable to arrive. The problem our business has been facing is if we should invest additional labor to resolve data we were previously unable to deliver to our clients due to quality. Our customers have agreed to purchase the data if it is corrected and delivered as needed - based on market conditions - but it must be delivered at the time it is needed or we will fail to obtain full revenue. Therefore, predicting when the client will need the data is paramount to our success.

It is important to note that if we do not act, all potential revenue from this data will remain lost. If we correct the data, we can still recover some revenue. However, if we attempt to deliver the data too early, we will lose 25 dollars on each dollar invested in correcting our data as we will need to house the additional inventory until it is useful. Conversely, if we deliver the data too late, we will lose 125 dollars because the client will no longer fully benefit from the corrected data. In other words, predicting the data is needed when it is not yet needed (a false positive) will cost us 25 dollars while predicting the data is not yet needed when it is (a false negative) will cost us 125 dollars.

When applying machine learning, there are many different models that can be used. Typically, however, models perform differently than other models when applied to different datasets. Consequently, we decided to solve our problem using three different candidate models, testing each approach with multiple configurations, to find the best solution. All modeling approaches take into consideration the fact that an early delivery costs 25 dollars and a late delivery costs 125.

### 1.1 The Objective

The objective is to minimize the our monetary loss using machine learning techniques. As stated, we currently experience a 25.00 dollar loss for each false positive prediction and a loss of 125.00 dollars for false negative predictions. Our goal in this scenario is to generate a model that minimizes the cost of doing business and provide adequate justifications for the decisions made in the final model.

## 2 The Dataset

The dataset features 160,000 records, 50 features, and a binary target variable. These features have been engineered to represent our marketplace conditions. Because of the method used for their engineering, there are no labels to explain exactly what they represent, as most are multiplicative representation of marketplace events that have been reduced for simplicity. Therefore, we must rely strictly on the interpretations of each feature and discover relationships between the features that help meet the overall objective of minimizing costs for our business.

## 3 Methods

### 3.1 Data Cleaning

The data are mostly clean, with the exception of five features that required mild transformations to make them machine readable. An example of the formatting of these features is provided.

**Table 1. Examples of Transformed Features**

| Attribute | x24 | x29 | x30 | x32 | x37 |
|---|---|---|---|---|---|
| Data Type | object | object | object | object | object |
| Data Example | asia | July | wednesday | 0.01% | $287.14 |

Features x32 and x37 were transformed into a numeric format by simply removing the non_numeric characters. Features x24, x29, and x30 were transformed into a numeric format using SciKit-Learn's LabelEncoder[1].

### 3.2 Data Imputation & Scaling

The dataset contains missing values that were judged to be missing at random. Missing values were imputed using SciKit-Learn's KNNImputer with the `n_neighbors` parameter set to three[2]. Once the data were imputed, they were normalized using Scikit-Learn's StandardScaler[3].

### 3.3 Modeling

We explored modeling this data with logistic regression, random forest, and XGBoost. Models were tuned and selected based on cost minimization. Specifically, we selected the model hyperparameters and the final model based on the cost given by the following cost function:

$$Model\ Cost = 25\,(False\ Positive\ Count) + 125\,(False\ Negative\ Count)$$

[1] https://scikit-learn.org/stable/modules/impute.html
[3] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html
[3] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

We utilized a randomized hyperparameter search with 5-fold cross-validation to select hyperparameters. 100 search iterations were executed for logistic regression and 1000 search iteration were executed for random forest and XGBoost since these models have more hyperparameters.

### 3.3.1 Logistic Regression

**Recursive Feature Selection** A baseline Logistic Regression model was built using Recursive Feature Selection to first determine an ideal number of features that emphasizes overall accuracy. Although accuracy appears to plateau after 10 features, our objective is to limit the financial loss to the customer, so as many features as are empirically necessary to limit this financial loss will be used. This procedure determined that the training and test sets could be limited to 19 out of the 50 features in the given dataset.

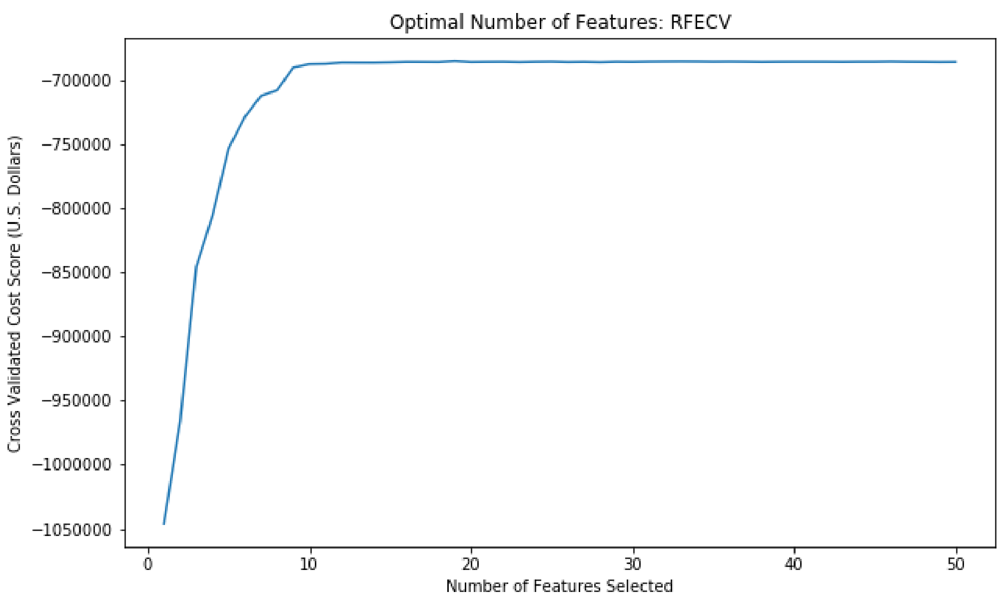



Figure 1: Feature Selection

**Hyperparameter Tuning** To complete the baseline model, a randomized grid search was conducted to find the best C value and regularization term[4]. The selected hyperparameters are shown in table 2.

**Table 2. Logistic Regression Hyperparameters**

| C | Regularization | Number of Features |
|---|---|---|
| 0.3712 | $L2$ | 19 |

[4]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

### 3.3.2 Random Forest

The following parameters were selected from the 1000-iteration randomized search.

**Table 3. Tuned Random Forest Hyperparameters**

| Parameter | Value |
|---|---|
| `n_estimators` | 132 |
| `criterion` | `'gini'` |
| `max_depth` | 83 |
| `min_samples_split` | 2 |
| `min_samples_leaf` | 83 |
| `max_features` | `'sqrt'` |

### 3.3.3 XGBoost

The following parameters were selected from the 1000-iteration randomized search.

**Table 4. Tuned XGBoost HyperParameters**

| Parameter | Value |
|---|---|
| `boosting_rounds` | 170 |
| `eta` | 0.2518 |
| `gamma` | 0.00227 |
| `max_depth` | 9 |
| `colsample_bytree` | 1.0 |
| `colsample_bylevel` | 0.8 |
| `colsample_bynode` | 0.9 |
| `subsample` | 0.9545 |
| `lambda` | 3.594 |
| `alpha` | 2.057 |

# 4 Results

Three tuned models were generated and their estimated costs to the company are visualized below. The XG Boost model is the most cost effective and accurate of the three. While this model has a limited ability to help determine the the main contributing factors to model cost, meaning, it is the most difficult to interpret, its high accuracy appears to deliver the results that are sought. The model costs are shown in Fig. 2 and the model accuracies are shown in table 5.

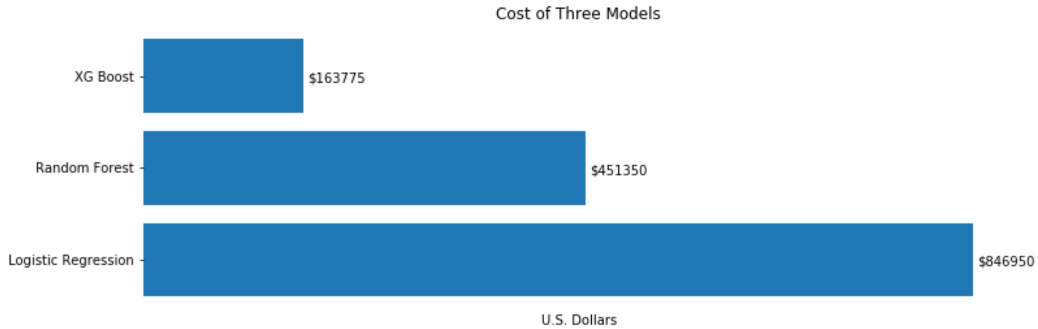


Figure 2: Cost of model predictions on an out-of-fold test set.

**Table 5. Model Performance on Test Set**

| Model | Accuracy | Estimated Loss |
|---|---|---|
| `Logistic Regression` | 70% | $846,950 |
| `Random Forest` | 85% | $451,350 |
| `XGBoost` | 93% | $163,775 |

## 5 Conclusion

First, we selected the marketplace features that were most useful. After, we generated predictions from three different models using multiple configurations. Ultimately, we found a model and configuration that yielded a 99% accuracy, minimizing the our monetary loss to 163,775 dollars. When using this approach, we are able to allocate resources to recover revenue on data that would otherwise be unprofitable. This use case provides reason to continue using machine learning within our organization to resolve ongoing problems and enable us to therefore remain competitive within the marketplace.

## A Code

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression


# pd.options.display.max_columns = 100
random_state = 42
```

[10]:
```
data = pd.read_csv('./final_project.csv')
data.head()
```

[10]:
```
         x0        x1         x2        x3        x4         x5         x6  \
0 -0.166563 -3.961588   4.621113  2.481908 -1.800135   0.804684   6.718751
1 -0.149894 -0.585676  27.839856  4.152333  6.426802  -2.426943  40.477058
2 -0.321707 -1.429819  12.251561  6.586874 -5.304647 -11.311090  17.812850
3 -0.245594  5.076677 -24.149632  3.637307  6.505811   2.290224 -35.111751
4 -0.273366  0.306326 -11.352593  1.676758  2.928441  -0.616824 -16.505817

          x7        x8        x9  ...        x41       x42       x43  \
0 -14.789997 -1.040673 -4.204950  ...  -1.497117  5.414063 -2.325655
1  -6.725709  0.896421  0.330165  ...  36.292790  4.490915  0.762561
2  11.060572  5.325880 -2.632984  ...  -0.368491  9.088864 -0.689886
3 -18.913592 -0.337041 -5.568076  ...  15.691546 -7.467775  2.940789
4  27.532281  1.199715 -4.309105  ... -13.911297 -5.229937  1.783928

        x44       x45        x46       x47       x48        x49  y
0  1.674827 -0.264332  60.781427 -7.689696  0.151589  -8.040166  0
1  6.526662  1.007927  15.805696 -4.896678 -0.320283  16.719974  0
2 -2.731118  0.754200  30.856417 -7.428573 -2.090804  -7.869421  0
3 -6.424112  0.419776 -72.424569  5.361375  1.806070  -7.670847  0
4  3.957801 -0.096988 -14.085435 -0.208351 -0.894942  15.724742  1

[5 rows x 51 columns]
```

[11]:
```
y = data['y']
X = data.drop(['y'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=random_state, stratify=y)
print('X_train: ', X_train.shape,
      '\ny_train: ', y_train.shape,
```

```python
        '\nX_test: ', X_test.shape,
        '\ny_test: ', y_test.shape)
```

```
X_train:  (128000, 50)
y_train:  (128000,)
X_test:  (32000, 50)
y_test:  (32000,)
```

[4]:
```python
# pd.options.display.max_columns = 49
X_train.describe()
```

[4]:
```
                  x0             x1             x2             x3  \
count  127982.000000  127981.000000  127972.000000  127970.000000
mean       -0.000233       0.015927      -1.137009      -0.030134
std         0.371378       6.338409      13.287193       8.067211
min        -1.592635     -26.053979     -59.394048     -33.864827
25%        -0.250854      -4.239255     -10.166609      -5.459399
50%        -0.001322       0.028105      -1.312739      -0.031448
75%         0.249459       4.291164       7.865764       5.429540
max         1.600849      27.988178      57.908998      38.906025

                  x4             x5             x6             x7  \
count  127983.000000  127969.000000  127979.000000  127975.000000
mean       -0.005189       0.017988      -1.650412      -7.632300
std         6.381621       7.673899      19.318144      30.583165
min       -28.467536     -33.822988     -86.354483    -181.506976
25%        -4.315710      -5.153559     -14.779614     -27.238063
50%         0.009683       0.028314      -1.905278      -6.873548
75%         4.298284       5.196676      11.439981      12.284980
max        26.247812      35.550110      84.195332     149.150634

                  x8             x9  ...            x40            x41  \
count  127981.000000  127974.000000  ...  127975.000000  127968.000000
mean       -0.036771       0.013169  ...      -2.280161       6.696548
std         8.898180       6.347198  ...      17.047011      18.697780
min       -37.691045     -27.980659  ...     -74.059196     -82.167224
25%        -6.036748      -4.248533  ...     -13.918753      -5.804080
50%        -0.014964       0.004454  ...      -2.672047       6.824208
75%         5.957489       4.302156  ...       9.000087      19.273821
max        39.049831      27.377842  ...      88.824477     100.050432

                 x42            x43            x44            x45  \
count  127980.000000  127971.000000  127968.000000  127979.000000
mean       -1.823620      -0.004985      -0.007610       0.000887
std         5.106297       1.534255       4.157975       0.396842
min       -27.933750      -6.876234     -17.983487      -1.753221
25%        -5.147121      -1.041030      -2.808371      -0.266998
```

```
50%        -1.917862      -0.007160      -0.010351       0.001770
75%         1.465863       1.029794       2.777005       0.268573
max        22.668041       6.441093      17.007392       1.669205

                 x46            x47            x48            x49
count  127976.000000  127972.000000  127971.000000  127973.000000
mean      -12.757091       0.021462       0.000821      -0.664870
std        36.606157       4.794483       1.936761      15.055081
min      -201.826828     -21.086333      -8.490155     -65.791191
25%       -36.444478      -3.232455      -1.322622     -10.937141
50%       -12.977831       0.021841      -0.012518      -0.572639
75%        11.484705       3.266336       1.319282       9.679463
max       150.859415      20.836854       8.206509      66.877604

[8 rows x 45 columns]
```

[5]:
```python
# assess missingness in data
names = X_train.columns
for i in names:
    print(i, ': ', 'Train-', sum(pd.isna(X_train[i])),
          'Test-', sum(pd.isna(X_test[i])))
```

```
x0 :  Train- 18 Test- 8
x1 :  Train- 19 Test- 6
x2 :  Train- 28 Test- 10
x3 :  Train- 30 Test- 7
x4 :  Train- 17 Test- 9
x5 :  Train- 31 Test- 6
x6 :  Train- 21 Test- 5
x7 :  Train- 25 Test- 2
x8 :  Train- 19 Test- 2
x9 :  Train- 26 Test- 4
x10 :  Train- 33 Test- 10
x11 :  Train- 23 Test- 7
x12 :  Train- 31 Test- 5
x13 :  Train- 26 Test- 5
x14 :  Train- 29 Test- 5
x15 :  Train- 29 Test- 6
x16 :  Train- 19 Test- 7
x17 :  Train- 19 Test- 8
x18 :  Train- 35 Test- 5
x19 :  Train- 24 Test- 11
x20 :  Train- 34 Test- 4
x21 :  Train- 21 Test- 8
x22 :  Train- 21 Test- 6
x23 :  Train- 38 Test- 9
x24 :  Train- 24 Test- 4
```

```
x25 :  Train- 19 Test- 3
x26 :  Train- 28 Test- 8
x27 :  Train- 26 Test- 4
x28 :  Train- 29 Test- 6
x29 :  Train- 19 Test- 11
x30 :  Train- 23 Test- 7
x31 :  Train- 34 Test- 5
x32 :  Train- 26 Test- 5
x33 :  Train- 35 Test- 6
x34 :  Train- 36 Test- 5
x35 :  Train- 25 Test- 5
x36 :  Train- 21 Test- 6
x37 :  Train- 17 Test- 6
x38 :  Train- 26 Test- 5
x39 :  Train- 16 Test- 7
x40 :  Train- 25 Test- 11
x41 :  Train- 32 Test- 8
x42 :  Train- 20 Test- 6
x43 :  Train- 29 Test- 8
x44 :  Train- 32 Test- 8
x45 :  Train- 21 Test- 8
x46 :  Train- 24 Test- 7
x47 :  Train- 28 Test- 9
x48 :  Train- 29 Test- 3
x49 :  Train- 27 Test- 5
```

[6]:
```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 128000 entries, 111609 to 57443
Data columns (total 50 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   x0      127982 non-null  float64
 1   x1      127981 non-null  float64
 2   x2      127972 non-null  float64
 3   x3      127970 non-null  float64
 4   x4      127983 non-null  float64
 5   x5      127969 non-null  float64
 6   x6      127979 non-null  float64
 7   x7      127975 non-null  float64
 8   x8      127981 non-null  float64
 9   x9      127974 non-null  float64
 10  x10     127967 non-null  float64
 11  x11     127977 non-null  float64
 12  x12     127969 non-null  float64
 13  x13     127974 non-null  float64
 14  x14     127971 non-null  float64
```

```
 15  x15     127971 non-null  float64
 16  x16     127981 non-null  float64
 17  x17     127981 non-null  float64
 18  x18     127965 non-null  float64
 19  x19     127976 non-null  float64
 20  x20     127966 non-null  float64
 21  x21     127979 non-null  float64
 22  x22     127979 non-null  float64
 23  x23     127962 non-null  float64
 24  x24     127976 non-null  object
 25  x25     127981 non-null  float64
 26  x26     127972 non-null  float64
 27  x27     127974 non-null  float64
 28  x28     127971 non-null  float64
 29  x29     127981 non-null  object
 30  x30     127977 non-null  object
 31  x31     127966 non-null  float64
 32  x32     127974 non-null  object
 33  x33     127965 non-null  float64
 34  x34     127964 non-null  float64
 35  x35     127975 non-null  float64
 36  x36     127979 non-null  float64
 37  x37     127983 non-null  object
 38  x38     127974 non-null  float64
 39  x39     127984 non-null  float64
 40  x40     127975 non-null  float64
 41  x41     127968 non-null  float64
 42  x42     127980 non-null  float64
 43  x43     127971 non-null  float64
 44  x44     127968 non-null  float64
 45  x45     127979 non-null  float64
 46  x46     127976 non-null  float64
 47  x47     127972 non-null  float64
 48  x48     127971 non-null  float64
 49  x49     127973 non-null  float64
dtypes: float64(45), object(5)
memory usage: 49.8+ MB
```

[12]:
```
objects = X_train.select_dtypes(['O'])
objects_test = X_test.select_dtypes(['O'])
objects.head()
```

[12]:

| | x24 | x29 | x30 | x32 | x37 |
|---|---|---|---|---|---|
| 111609 | asia | Apr | wednesday | -0.0% | $100.73 |
| 3785 | asia | May | wednesday | 0.01% | $1005.31 |
| 64066 | asia | July | wednesday | 0.0% | $-1406.52 |
| 103309 | asia | Jun | wednesday | 0.01% | $-1287.29 |

```
9084     asia    Aug     tuesday   0.02%  $-1670.43
```

[8]:
```
objects.describe()
```

[8]:

| | x24 | x29 | x30 | x32 | x37 |
|---|---|---|---|---|---|
| count | 127976 | 127981 | 127977 | 127974 | 127983 |
| unique | 3 | 12 | 5 | 12 | 107696 |
| top | asia | July | wednesday | 0.01% | $237.4 |
| freq | 111198 | 36384 | 81253 | 32630 | 6 |

[13]:
```
# fix spelling error
X_test['x24'] = X_test['x24'].str.replace('euorpe', 'europe')
# remove  %
X_test['x32'] = pd.to_numeric(X_test['x32'].str.replace('%', ''))
# remove $
X_test['x37'] = pd.to_numeric(X_test['x37'].str.replace('$', ''))
# repeat process for training set
X_train['x24'] = X_train['x24'].str.replace('euorpe', 'europe')
X_train['x32'] = pd.to_numeric(X_train['x32'].str.replace('%', ''))
X_train['x37'] = pd.to_numeric(X_train['x37'].str.replace('$', ''))
# remake objects
objects = X_train.select_dtypes(['O'])
objects_test = X_test.select_dtypes(['O'])
objects.describe()
```

[13]:

| | x24 | x29 | x30 |
|---|---|---|---|
| count | 127976 | 127981 | 127977 |
| unique | 3 | 12 | 5 |
| top | asia | July | wednesday |
| freq | 111198 | 36384 | 81253 |

[14]:
```
# imputing with mode from training data
X_train['x24'].fillna('asia', inplace=True)
X_train['x29'].fillna('July', inplace=True)
X_train['x30'].fillna('wednesday', inplace=True)

X_test['x24'].fillna('asia', inplace=True)
X_test['x29'].fillna('July', inplace=True)
X_test['x30'].fillna('wednesday', inplace=True)
for i in objects.columns:
    print(i, sum(pd.isna(X_train[i])), '\t', sum(pd.isna(X_test[i])))
```

```
x24 0    0
x29 0    0
x30 0    0
```

```
[15]: # label encode all string values
from sklearn.preprocessing import LabelEncoder

names = [i for i in list(objects.columns)]

le = LabelEncoder()
for i in names:
    le.fit(objects[i].astype(str))
    X_train[i] = le.transform(X_train[i])
    X_test[i] = le.transform(X_test[i])
```

```
[16]: from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler

KNNimp = KNNImputer(n_neighbors=3)
X_train = KNNimp.fit_transform(X_train)
X_test = KNNimp.transform(X_test)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
[17]: print(np.isnan(X_train).any())
print(np.isnan(X_test).any())
```

```
False
False
```

**Custom Loss Function**

To make our model fit the parameters of the scenario, we made a custom loss function and used it to score our grid search results.

```
[18]: def cost_score(y, y_pred, fp_cost=25, fn_cost=125):
    '''
    '''
    # get the misclassifications
    misclass_idx = np.where(np.equal(y, y_pred) == False)[0]
    # get the false positives
    fp_idx = np.where(y_pred[misclass_idx] == 1)[0]
    # get the false negatives
    fn_idx = np.where(y_pred[misclass_idx] == 0)[0]
    # calc the misclassification cost
    misclassification_cost = fp_idx.size * fp_cost + fn_idx.size * fn_cost

    return misclassification_cost

cost_scorer = make_scorer(cost_score, greater_is_better=False)
```

[19]:
```
from sklearn.feature_selection import RFECV
from sklearn.linear_model import LogisticRegression

# define the estimator
logistic = LogisticRegression()
# provide the parameters of the feature selection process
feature_selector = RFECV(logistic,
          scoring=cost_scorer,
          step = 1,
          min_features_to_select= 1,
          cv = 5,
          n_jobs = -1)
feature_selector = feature_selector.fit(X_train, y_train)
```

[20]:
```
import matplotlib.pyplot as plt
# visualize the results
print('Optimal number of features : %d' % feature_selector.n_features_)
plt.figure(figsize=(10,6))
plt.title('Optimal Number of Features: RFECV')
plt.xlabel('Number of Features Selected')
plt.ylabel('Cross Validated Cost Score (U.S. Dollars)')
plt.plot(range(1,len(feature_selector.grid_scores_) +1), feature_selector.
 ↪grid_scores_)
caption = 'Figure 1: Feature Selection'
plt.figtext(0.5, 0.001, caption, wrap=True, horizontalalignment='center',␣
 ↪fontsize=14);
plt.savefig('feature_selection.png')
plt.show()
```

```
Optimal number of features : 19
```

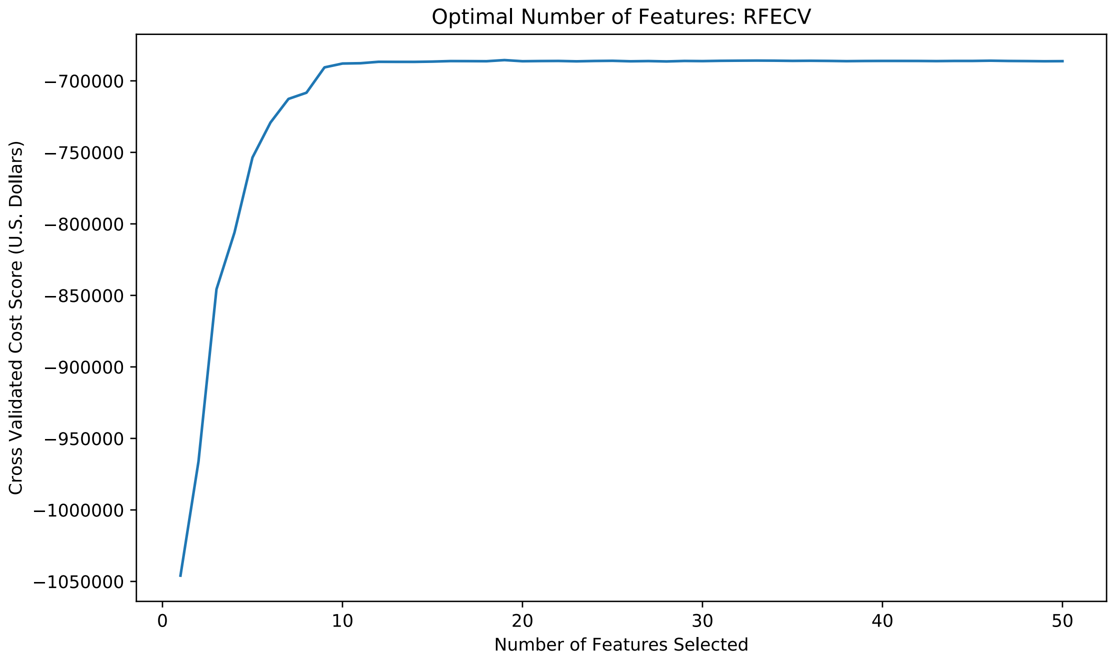


Figure 1: Feature Selection

That's a lot of features with just a little return, but we're trying to minimize costs, so we can afford to put them in in the model.

[22]:
```
X_train = feature_selector.transform(X_train)
X_test = feature_selector.transform(X_test)
print('X_train shape: ', X_train.shape,
      '\nX_test shape: ', X_test.shape)
```

```
X_train shape:  (128000, 33)
X_test shape:  (32000, 33)
```

[20]:
```
# randomized grid search to fine best parameters for logistic regression
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
# instantiate estimator
logistic = LogisticRegression(solver='saga',
                              tol=1e-2,
                              max_iter=200,
                              random_state=random_state)
# define search space
penalty = ['l1', 'l2']
C = uniform(loc=0, scale=4)
hyperparameters = dict(C=C, penalty=penalty)
# instantiate grid search
clf = RandomizedSearchCV(logistic,
```

```
                             hyperparameters,
                             scoring=cost_scorer,
                             random_state=1,
                             n_iter=100,
                             cv=5,
                             verbose=0,
                             n_jobs=-1)

# search for best parameters
search = clf.fit(X_train, y_train)
```

[14]:
```
print('Best Penalty:', search.best_estimator_.get_params()['penalty'])
print('Best C:', search.best_estimator_.get_params()['C'])
```

```
Best Penalty: l1
Best C: 0.3846890417818467
```

[15]:
```
from sklearn.metrics import classification_report
preds = search.predict(X_test)

print(classification_report(y_test, preds))
```

```
              precision    recall  f1-score   support

           0       0.72      0.83      0.77     19161
           1       0.67      0.52      0.58     12839

    accuracy                           0.70     32000
   macro avg       0.70      0.67      0.68     32000
weighted avg       0.70      0.70      0.70     32000
```

[16]:
```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, preds)
cm
```

[16]:
```
array([[15900,  3261],
       [ 6193,  6646]], dtype=int64)
```

[17]:
```
# [[TP, FP],
#  [FN, TN]]

# precision = TP / TP + FP
# recall = TP / TP + FN
# accuracy = TP + TN / total
FN = cm[1][0]
```

```
FP = cm[0][1]
loss = 25*FP + 125*FN
loss
```

[17]: 855650

[22]:
```
import numpy as np
kwargs = dict(delimiter=",",
              skip_header=1,
              dtype='float64'
             )
df = np.genfromtxt('./final_project.csv',**kwargs)
X2 = df[:,:-1]
y2 = df[:,-1]
```

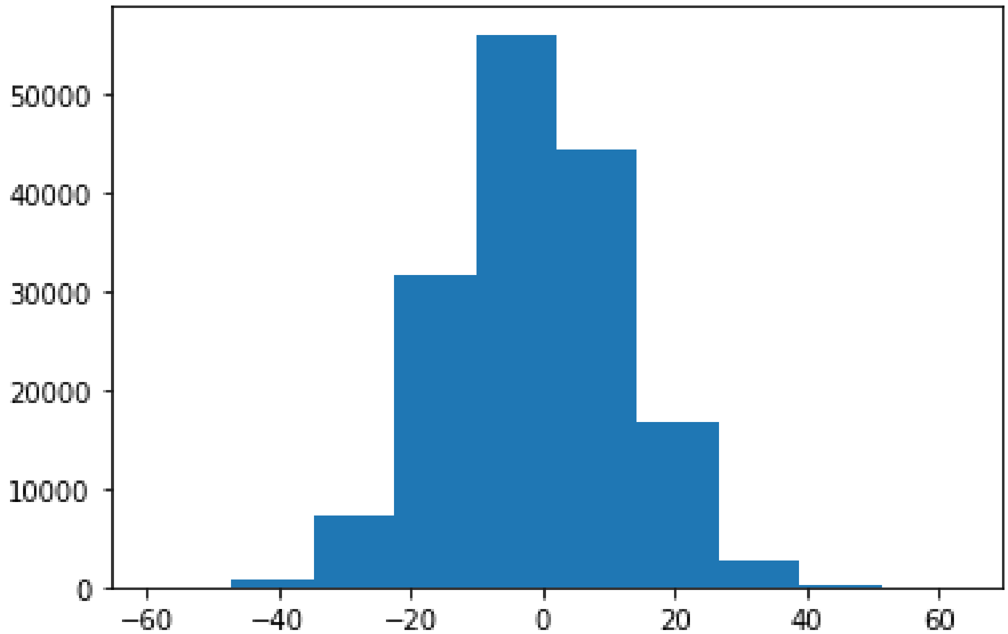
**Below, we used the Anderson Darling Test to determine if features are normally distributed or not. This is useful for our imputation strategy. Where the critical value is exceeded by the test statistic under the assumption of normality, we reject the null hypothesis and conclude violaiton. Therefore, for these, we apply a median imputation because while the distributions appear normal visually, there are outliers influencing normality at a 95% confidence level.**

**Also, we identify columns that are unable to be tested for having insufficient values required for the test.**

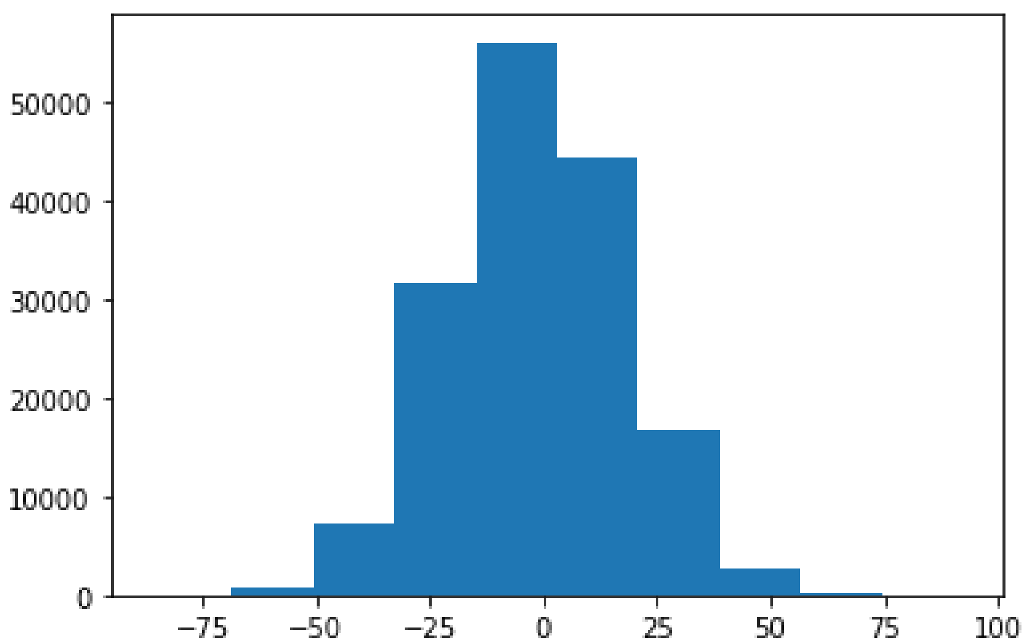
[23]:
```
# https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.anderson.html
from scipy import stats
import warnings
warnings.filterwarnings('ignore')

non_parametric=[]
bad_cols=[]

for i in np.arange(0,np.shape(X2)[1]):
    try:
        statistic = stats.anderson(X2[~np.isnan(X2[:,i]),i], dist='norm').
↪statistic
        crit_val = stats.anderson(X2[~np.isnan(X2[:,i]),i], dist='norm').
↪critical_values[2] # 5% significance, 95% confidence)

        if (crit_val < statistic):
            print("Column {} is not normally distributed at a 95% level of␣
↪confidence. Statistic: {} and Critical Value: {}".format(i, statistic,␣
↪crit_val));
            non_parametric.append(i)
            plt.hist(X2[~np.isnan(X2[:,i]),i])
            plt.show()
    except:
        bad_cols.append(i)
```

```
print("Columns that are not numeric: {}".format(bad_cols))
```

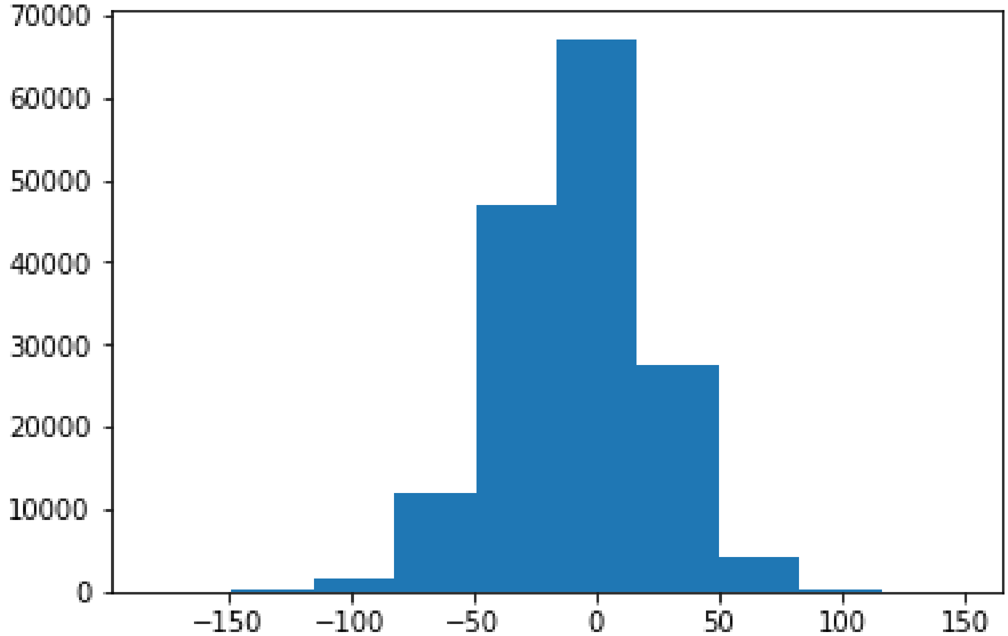
```
Column 2 is not normally distributed at a 95% level of confidence. Statistic:
10.220688929781318 and Critical Value: 0.787
```



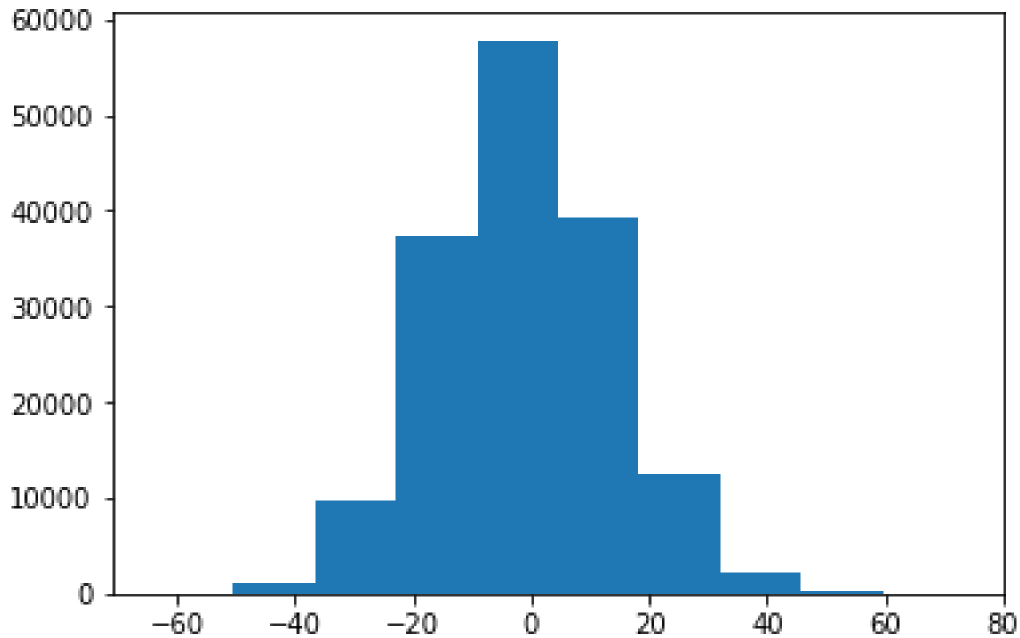

```
Column 6 is not normally distributed at a 95% level of confidence. Statistic:
10.270658839464886 and Critical Value: 0.787
```



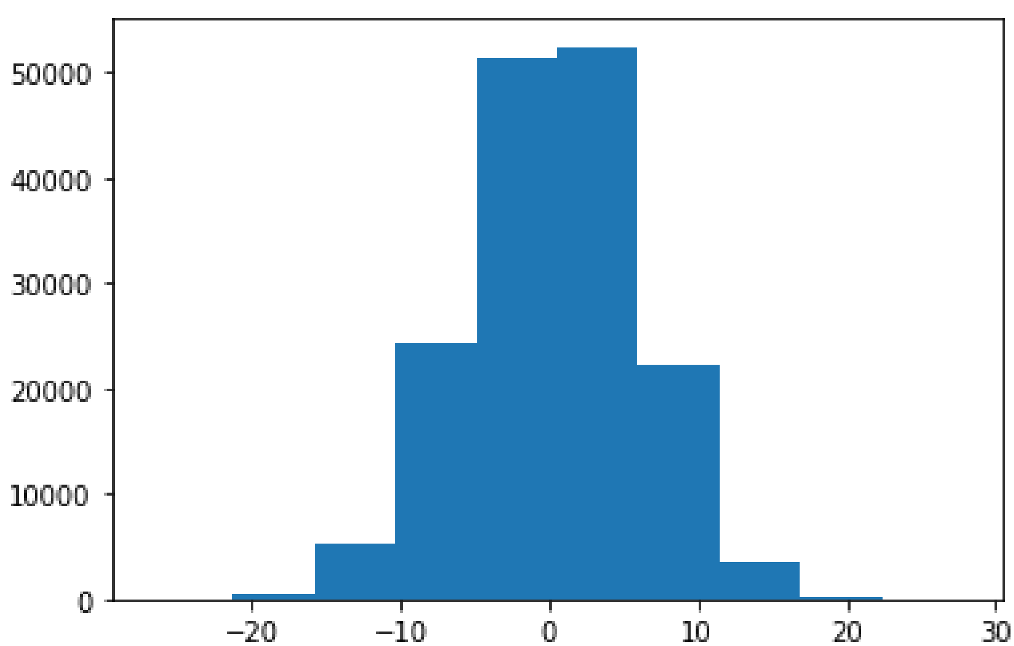
Column 7 is not normally distributed at a 95% level of confidence. Statistic: 58.931357158726314 and Critical Value: 0.787



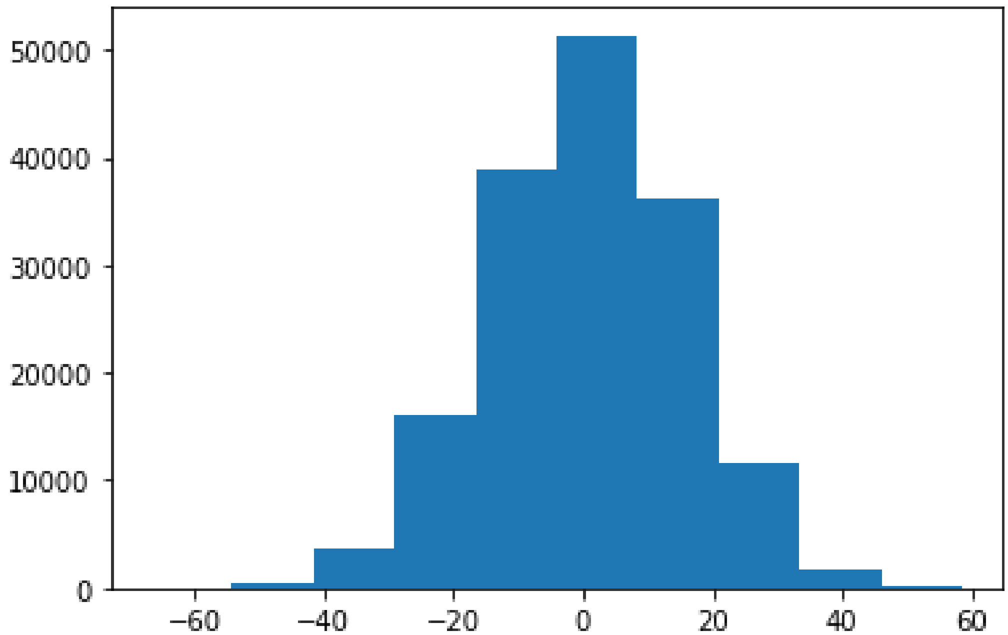

Column 12 is not normally distributed at a 95% level of confidence. Statistic: 26.41666220035404 and Critical Value: 0.787



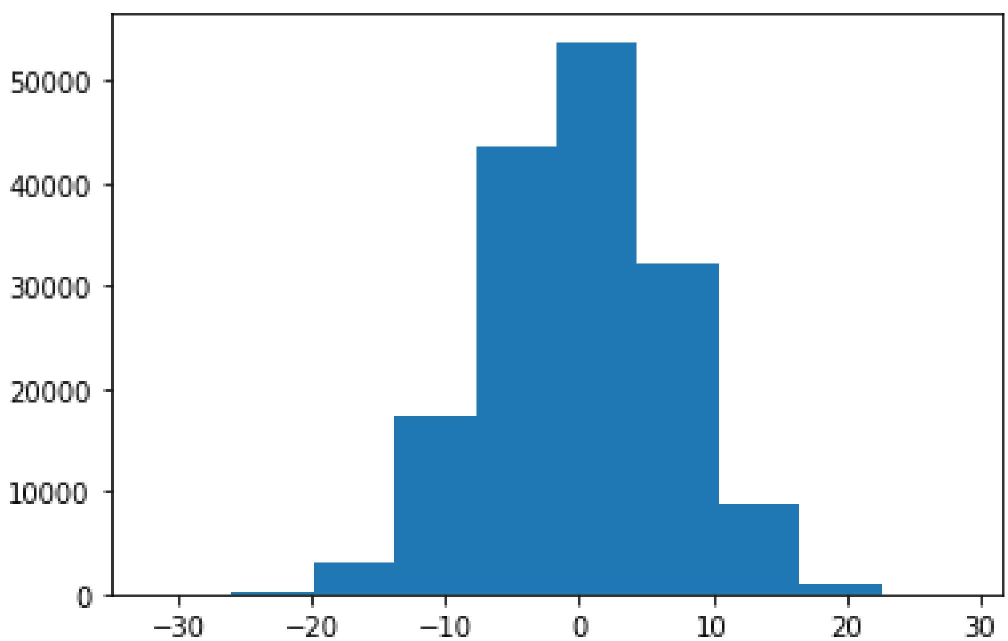

Column 20 is not normally distributed at a 95% level of confidence. Statistic: 29.789376930275466 and Critical Value: 0.787



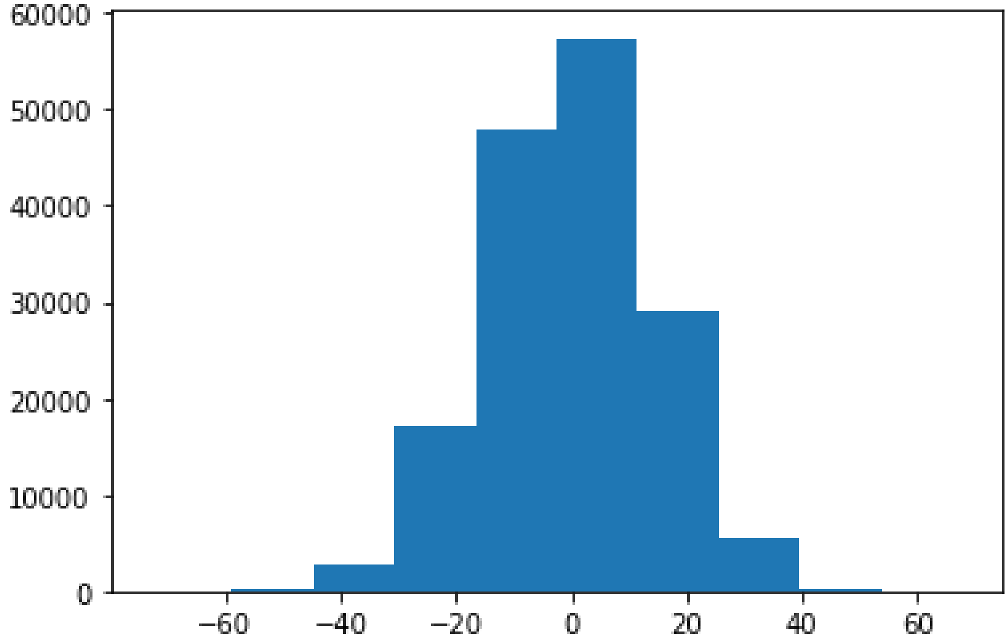

Column 23 is not normally distributed at a 95% level of confidence. Statistic: 29.42702754313359 and Critical Value: 0.787



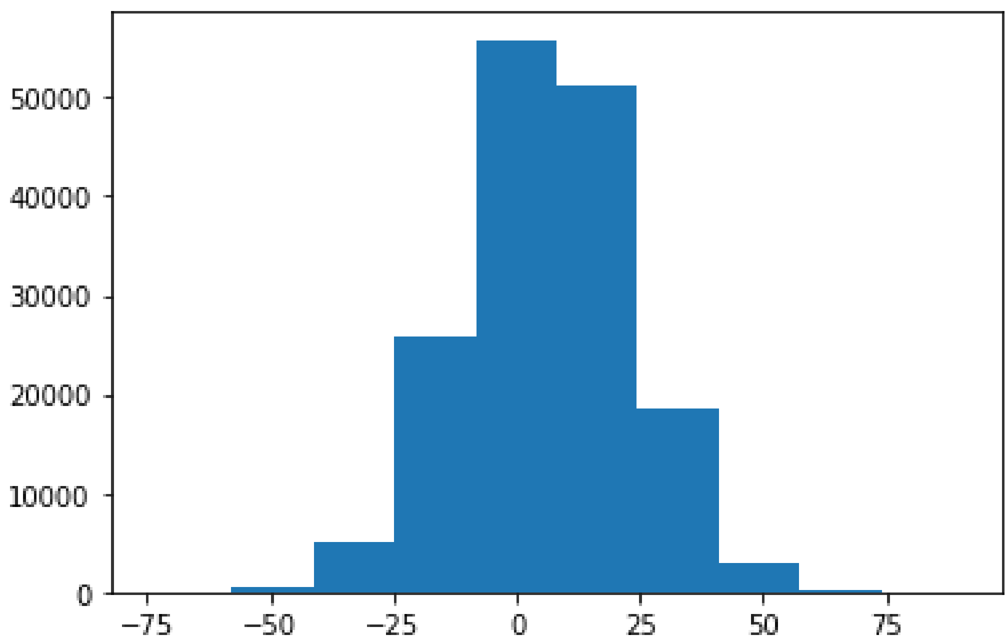

Column 27 is not normally distributed at a 95% level of confidence. Statistic: 5.4526740521832835 and Critical Value: 0.787



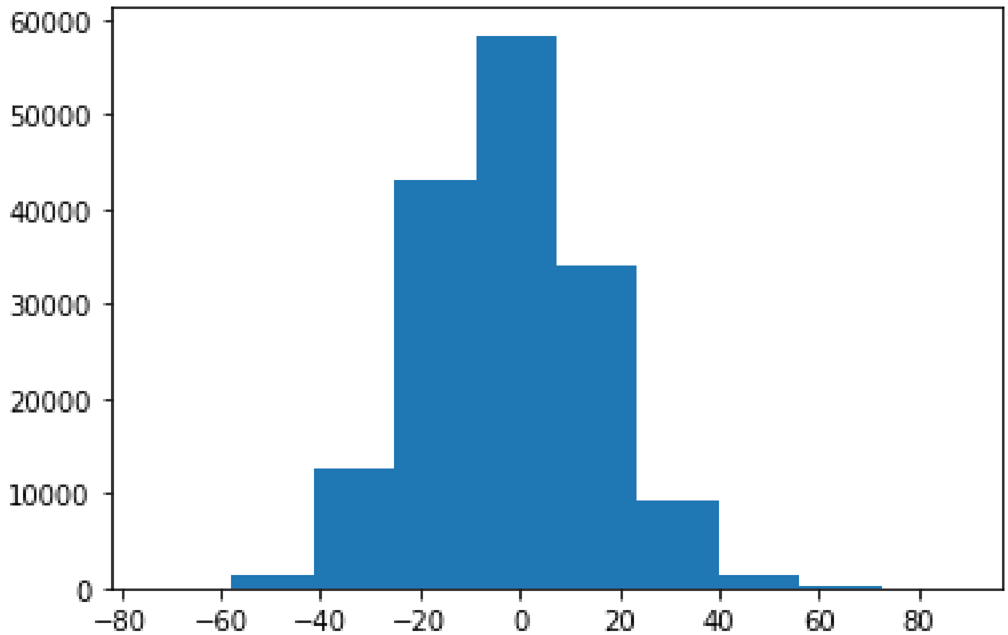

Column 28 is not normally distributed at a 95% level of confidence. Statistic: 13.03100506181363 and Critical Value: 0.787



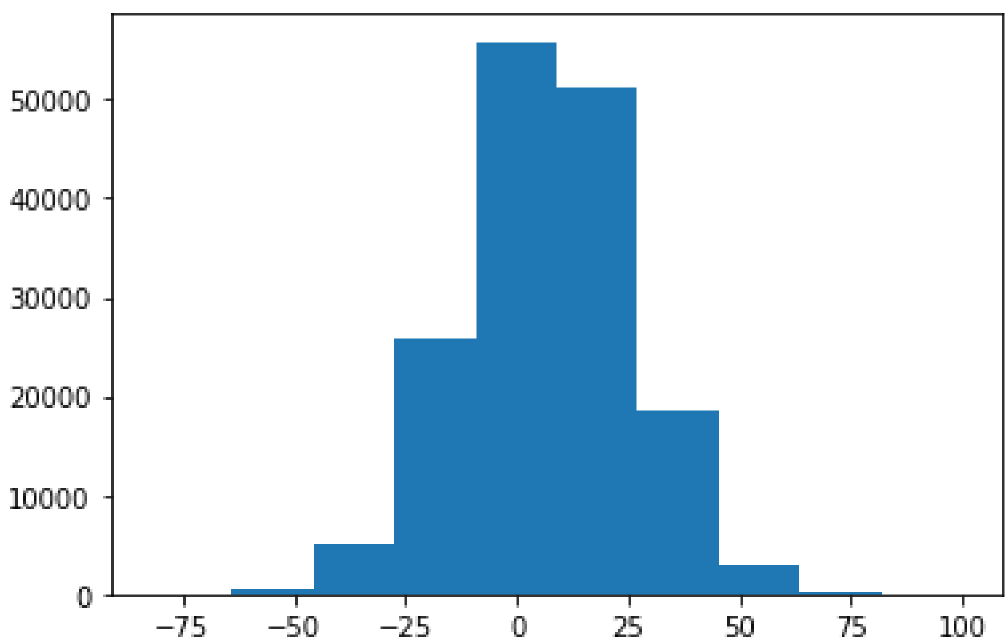

Column 38 is not normally distributed at a 95% level of confidence. Statistic: 3.5536952081602067 and Critical Value: 0.787



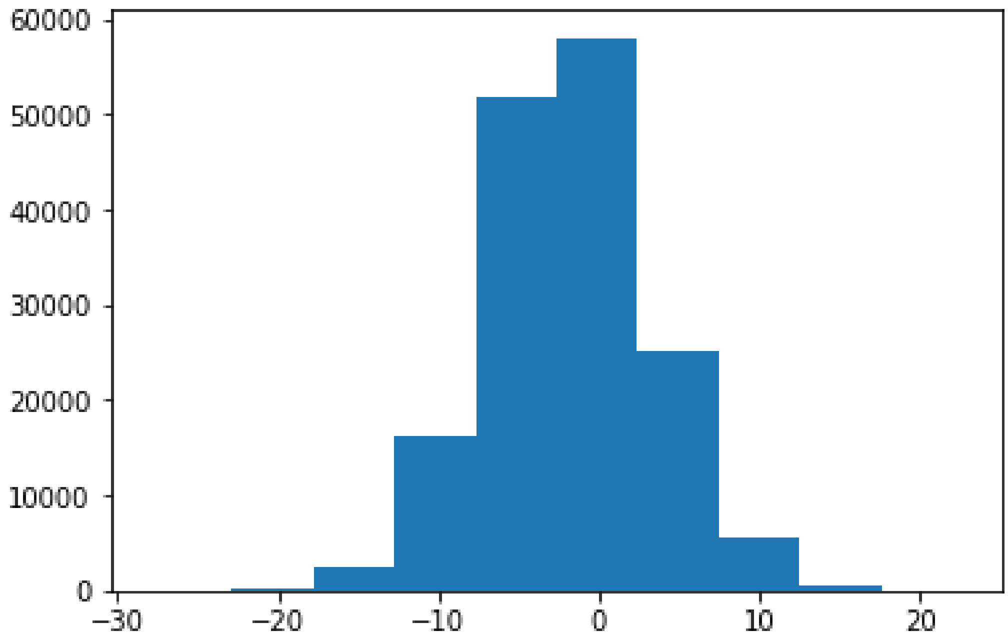

Column 40 is not normally distributed at a 95% level of confidence. Statistic: 26.42135159339523 and Critical Value: 0.787



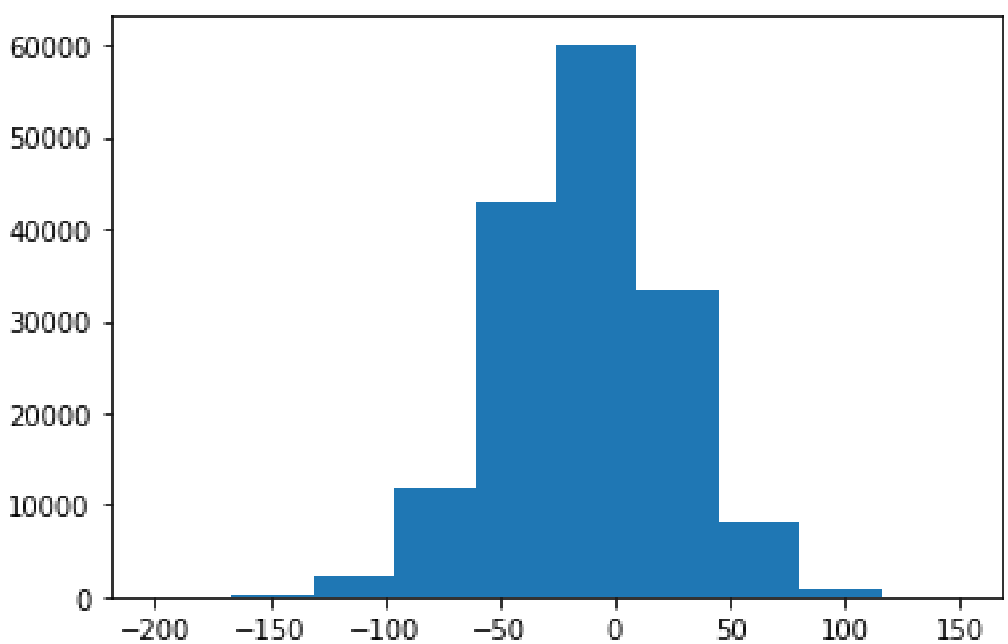

Column 41 is not normally distributed at a 95% level of confidence. Statistic: 3.595784921606537 and Critical Value: 0.787



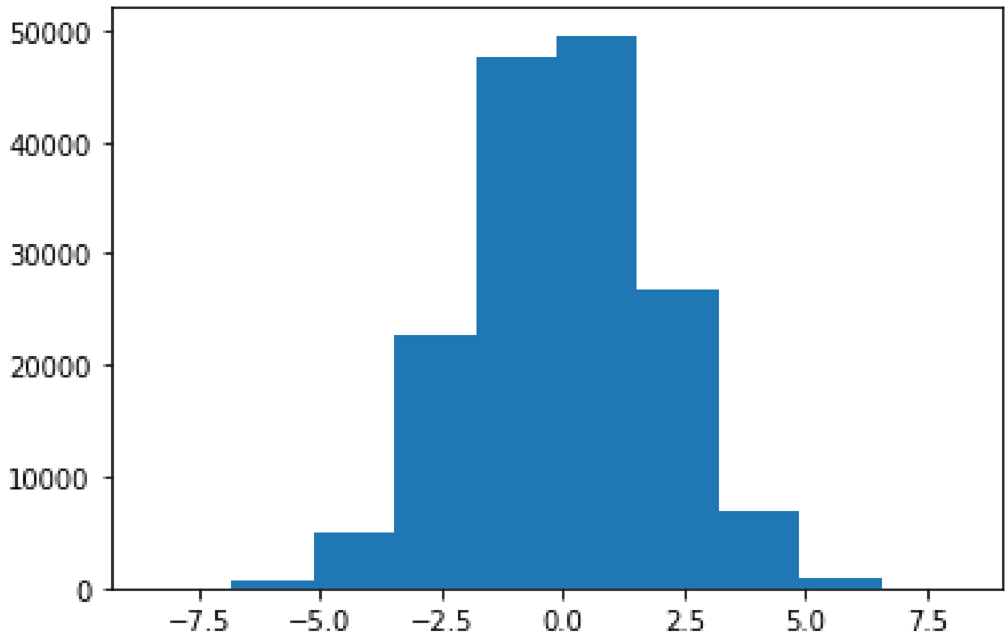

Column 42 is not normally distributed at a 95% level of confidence. Statistic: 47.59360718273092 and Critical Value: 0.787



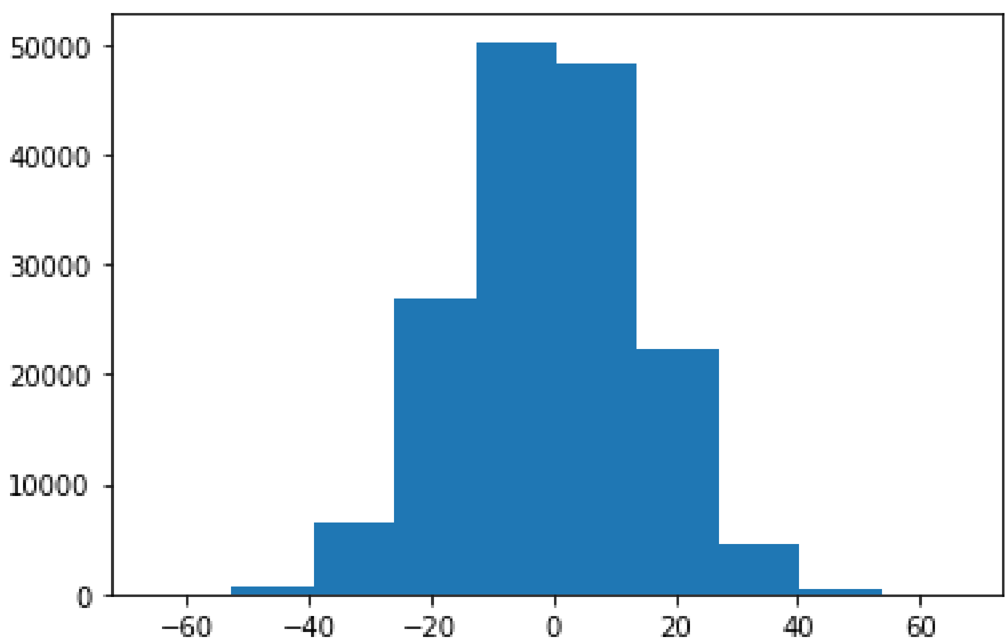

Column 46 is not normally distributed at a 95% level of confidence. Statistic: 31.06127203576034 and Critical Value: 0.787




Column 48 is not normally distributed at a 95% level of confidence. Statistic: 2.897343336167978 and Critical Value: 0.787




```
Column 49 is not normally distributed at a 95% level of confidence. Statistic:
6.4221460196131375 and Critical Value: 0.787
```




```
Columns that are not numeric: [24, 29, 30, 32, 37]
```

**The columns that are not normally distributed to not appear to be in wild violation of normality. Therefore, we can assume the SimpleImputer would work reasonable well using the mean.**

Drop the non-numeric columns:

[24]:
```
X_numeric = np.delete(X2, bad_cols, axis=1)
```

[25]:
```
np.shape(X_numeric)
```

[25]:
```
(160000, 45)
```

[26]:
```
for i in np.arange(np.shape(X_numeric)[1]):
    print("Percent of column {} having missing values: {}%".
 ↪format(i,round(100*(np.count_nonzero(np.isnan(X_numeric[:,i]))/np.
 ↪shape(X_numeric[:,i])[0]),2)))
```

```
Percent of column 0 having missing values: 0.02%
Percent of column 1 having missing values: 0.02%
Percent of column 2 having missing values: 0.02%
Percent of column 3 having missing values: 0.02%
Percent of column 4 having missing values: 0.02%
Percent of column 5 having missing values: 0.02%
Percent of column 6 having missing values: 0.02%
Percent of column 7 having missing values: 0.02%
Percent of column 8 having missing values: 0.01%
Percent of column 9 having missing values: 0.02%
Percent of column 10 having missing values: 0.03%
Percent of column 11 having missing values: 0.02%
Percent of column 12 having missing values: 0.02%
Percent of column 13 having missing values: 0.02%
Percent of column 14 having missing values: 0.02%
Percent of column 15 having missing values: 0.02%
Percent of column 16 having missing values: 0.02%
Percent of column 17 having missing values: 0.02%
Percent of column 18 having missing values: 0.03%
Percent of column 19 having missing values: 0.02%
Percent of column 20 having missing values: 0.02%
Percent of column 21 having missing values: 0.02%
Percent of column 22 having missing values: 0.02%
Percent of column 23 having missing values: 0.03%
Percent of column 24 having missing values: 0.01%
Percent of column 25 having missing values: 0.02%
Percent of column 26 having missing values: 0.02%
Percent of column 27 having missing values: 0.02%
Percent of column 28 having missing values: 0.02%
Percent of column 29 having missing values: 0.03%
Percent of column 30 having missing values: 0.03%
```

```
Percent of column 31 having missing values: 0.02%
Percent of column 32 having missing values: 0.02%
Percent of column 33 having missing values: 0.02%
Percent of column 34 having missing values: 0.01%
Percent of column 35 having missing values: 0.02%
Percent of column 36 having missing values: 0.03%
Percent of column 37 having missing values: 0.02%
Percent of column 38 having missing values: 0.02%
Percent of column 39 having missing values: 0.03%
Percent of column 40 having missing values: 0.02%
Percent of column 41 having missing values: 0.02%
Percent of column 42 having missing values: 0.02%
Percent of column 43 having missing values: 0.02%
Percent of column 44 having missing values: 0.02%
```

[27]:
```
from sklearn.impute import SimpleImputer
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(X_numeric)
X_numeric = imp.transform(X_numeric)
```

### A.0.1 Check to confirm numeric imputation was successful:

[28]:
```
for i in np.arange(np.shape(X_numeric)[1]):
    print("Percent of column {} having missing values: {}%".
 ↪format(i,round(100*(np.count_nonzero(np.isnan(X_numeric[:,i]))/np.
 ↪shape(X_numeric[:,i])[0]),2)))
```

```
Percent of column 0 having missing values: 0.0%
Percent of column 1 having missing values: 0.0%
Percent of column 2 having missing values: 0.0%
Percent of column 3 having missing values: 0.0%
Percent of column 4 having missing values: 0.0%
Percent of column 5 having missing values: 0.0%
Percent of column 6 having missing values: 0.0%
Percent of column 7 having missing values: 0.0%
Percent of column 8 having missing values: 0.0%
Percent of column 9 having missing values: 0.0%
Percent of column 10 having missing values: 0.0%
Percent of column 11 having missing values: 0.0%
Percent of column 12 having missing values: 0.0%
Percent of column 13 having missing values: 0.0%
Percent of column 14 having missing values: 0.0%
Percent of column 15 having missing values: 0.0%
Percent of column 16 having missing values: 0.0%
Percent of column 17 having missing values: 0.0%
Percent of column 18 having missing values: 0.0%
```

```
Percent of column 19 having missing values: 0.0%
Percent of column 20 having missing values: 0.0%
Percent of column 21 having missing values: 0.0%
Percent of column 22 having missing values: 0.0%
Percent of column 23 having missing values: 0.0%
Percent of column 24 having missing values: 0.0%
Percent of column 25 having missing values: 0.0%
Percent of column 26 having missing values: 0.0%
Percent of column 27 having missing values: 0.0%
Percent of column 28 having missing values: 0.0%
Percent of column 29 having missing values: 0.0%
Percent of column 30 having missing values: 0.0%
Percent of column 31 having missing values: 0.0%
Percent of column 32 having missing values: 0.0%
Percent of column 33 having missing values: 0.0%
Percent of column 34 having missing values: 0.0%
Percent of column 35 having missing values: 0.0%
Percent of column 36 having missing values: 0.0%
Percent of column 37 having missing values: 0.0%
Percent of column 38 having missing values: 0.0%
Percent of column 39 having missing values: 0.0%
Percent of column 40 having missing values: 0.0%
Percent of column 41 having missing values: 0.0%
Percent of column 42 having missing values: 0.0%
Percent of column 43 having missing values: 0.0%
Percent of column 44 having missing values: 0.0%
```

Read the non-numeric columns only:

[29]:
```
kwargs = dict(delimiter=",",
              skip_header=1,
              dtype="|U5",
              usecols=bad_cols,
              autostrip=True
             )
df_cat = np.genfromtxt('./final_project.csv',**kwargs)
```

Some of the non-numeric columns are numeric with special characters that need to be dropped while others are names of continents, months, and days, which can be encoded:

[30]:
```
df_cat
```

[30]:
```
array([['euorp', 'July', 'tuesd', '0.0%', '$1313'],
       ['asia', 'Aug', 'wedne', '-0.02', '$1962'],
       ['asia', 'July', 'wedne', '-0.01', '$430.'],
       ...,
       ['asia', 'Jun', 'wedne', '-0.0%', '$687.'],
       ['asia', 'May', 'wedne', '-0.02', '$439.'],
```

```
       ['asia', 'Aug', 'tuesd', '0.02%', '$-122']], dtype='<U5')
```

[31]:
```
continents = df_cat[:,0]
months = df_cat[:,1]
days = df_cat[:,2]

# convert sept. to Sept to be consistent with the other months and remove the␣
 ↪period character
months[months=='sept.']='Sept'
```

Check to see the percent of missing values in categorical data to assess risk of imputation. The missing volume is very small less than 0.02% so highly likely to be insignificant. After inspection, the missing values appear to be completely at random as well so the risk of any imputation is most likely non-impactful.

[32]:
```
print("Percent of days with missing values: {}%".format(round(100*np.
 ↪count_nonzero(continents=='')/np.shape(continents)[0],2)))
print("Percent of days with missing values: {}%".format(round(100*np.
 ↪count_nonzero(months=='')/np.shape(months)[0],2)))
print("Percent of days with missing values: {}%".format(round(100*np.
 ↪count_nonzero(days=='')/np.shape(days)[0],2)))
```

```
Percent of days with missing values: 0.02%
Percent of days with missing values: 0.02%
Percent of days with missing values: 0.02%
```

There are also very few missing values in the dollars and percents columns as well. Imputation here - along with all the other features - will also be minimal.

[33]:
```
df_cat[185,4]
```

[33]:
```
''
```

Set a placeholder of 4444 in place of the missing values so we can convert the array to float, then convert 4444 back to nan:

[34]:
```
# To convert the dollar and percent columns to float, we need to strip special␣
 ↪characters and convert to float.
# If it fails to convert to float, we set the value to 4444 because this is␣
 ↪happening for instances that have no value.
# We then convert all 4444 values to np.nan and join the values back to the␣
 ↪numeric data. We then encode true categorical data.
for i in np.arange(0,len(df_cat)):
    try:
        df_cat[i,3] = float(df_cat[i,3].replace('%', ''))
        df_cat[i,4] = float(df_cat[i,4].replace('$', ''))
    except:
        #pass
```

```
        df_cat[i,3] = 4444 #4444 is not a real percent value used so will be the␣
↪placeholder for nan conversion; +/-0.05% is the max
        df_cat[i,4] = 4444 #4444 is not a real dollar value used so will be the␣
↪placeholder for nan conversion; $999 is the max

percents = df_cat[:,3].astype(np.float)
percents[percents==4444] = 'nan'
dollars = df_cat[:,4].astype(np.float)
dollars[dollars==4444] = 'nan'

cat_to_num = np.hstack((percents.reshape(160000,1),dollars.reshape(160000,1)))
```

[35]:
```
for i in np.arange(np.shape(cat_to_num)[1]):
    print("Percent of column {} having missing values: {}%".
↪format(i,round(100*(np.count_nonzero(np.isnan(cat_to_num[:,i]))/np.
↪shape(cat_to_num[:,i])[0]),2)))
```

```
Percent of column 0 having missing values: 0.03%
Percent of column 1 having missing values: 0.03%
```

**Now combine all numeric data, run an imputer on it and use those values to predict the missing values of the numeric data converted from cats (dollars and percents)**

All features that are truly numeric are horizontally stacked together here:

[36]:
```
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
imp.fit(cat_to_num)
cat_to_num = imp.transform(cat_to_num)
```

**Check to confirm dollar and percent imputations were successful:**

[37]:
```
for i in np.arange(np.shape(cat_to_num)[1]):
    print("Percent of column {} having missing values: {}%".
↪format(i,round(100*(np.count_nonzero(np.isnan(cat_to_num[:,i]))/np.
↪shape(cat_to_num[:,i])[0]),2)))
```

```
Percent of column 0 having missing values: 0.0%
Percent of column 1 having missing values: 0.0%
```

**Add the dollars and percents 2d array into the original numeric 2d array:**

[38]:
```
X_nums = np.hstack((X_numeric, cat_to_num))
```

Confirm no missing values:

[39]:
```
for i in np.arange(np.shape(X_nums)[1]):
    print("Percent of column {} having missing values: {}%".
↪format(i,round(100*(np.count_nonzero(np.isnan(X_nums[:,i]))/np.shape(X_nums[:
↪,i])[0]),2)))
```

```
Percent of column 0 having missing values: 0.0%
Percent of column 1 having missing values: 0.0%
Percent of column 2 having missing values: 0.0%
Percent of column 3 having missing values: 0.0%
Percent of column 4 having missing values: 0.0%
Percent of column 5 having missing values: 0.0%
Percent of column 6 having missing values: 0.0%
Percent of column 7 having missing values: 0.0%
Percent of column 8 having missing values: 0.0%
Percent of column 9 having missing values: 0.0%
Percent of column 10 having missing values: 0.0%
Percent of column 11 having missing values: 0.0%
Percent of column 12 having missing values: 0.0%
Percent of column 13 having missing values: 0.0%
Percent of column 14 having missing values: 0.0%
Percent of column 15 having missing values: 0.0%
Percent of column 16 having missing values: 0.0%
Percent of column 17 having missing values: 0.0%
Percent of column 18 having missing values: 0.0%
Percent of column 19 having missing values: 0.0%
Percent of column 20 having missing values: 0.0%
Percent of column 21 having missing values: 0.0%
Percent of column 22 having missing values: 0.0%
Percent of column 23 having missing values: 0.0%
Percent of column 24 having missing values: 0.0%
Percent of column 25 having missing values: 0.0%
Percent of column 26 having missing values: 0.0%
Percent of column 27 having missing values: 0.0%
Percent of column 28 having missing values: 0.0%
Percent of column 29 having missing values: 0.0%
Percent of column 30 having missing values: 0.0%
Percent of column 31 having missing values: 0.0%
Percent of column 32 having missing values: 0.0%
Percent of column 33 having missing values: 0.0%
Percent of column 34 having missing values: 0.0%
Percent of column 35 having missing values: 0.0%
Percent of column 36 having missing values: 0.0%
Percent of column 37 having missing values: 0.0%
Percent of column 38 having missing values: 0.0%
Percent of column 39 having missing values: 0.0%
Percent of column 40 having missing values: 0.0%
Percent of column 41 having missing values: 0.0%
Percent of column 42 having missing values: 0.0%
Percent of column 43 having missing values: 0.0%
Percent of column 44 having missing values: 0.0%
Percent of column 45 having missing values: 0.0%
Percent of column 46 having missing values: 0.0%
```

[40]:
```
# For mapping X_train and X_test values to the missing and non-missing␣
↪categorical targets for imputing
cont_to_impute = list(np.where(continents==''))
months_to_impute = list(np.where(months==''))
days_to_impute = list(np.where(days==''))
```

[41]:
```
# y_train_continents = np.delete(continents, cont_to_impute, axis=0)
# y_train_months = np.delete(months, months_to_impute, axis=0)
# y_train_days = np.delete(days, days_to_impute, axis=0)

# y_test_continents = continents[cont_to_impute]
# y_test_months = months[months_to_impute]
# y_test_days = days[days_to_impute]
```

[42]:
```
# y_train_continents = continents[continents!='']
# y_train_months = months[months!='']
# y_train_days = days[days!='']

# y_test_continents = continents[continents=='']
# y_test_months = months[months=='']
# y_test_days = days[days=='']
```

**Encoding categorical data to numeric using dictionaries**

[43]:
```
import string
from collections import Counter

continent_dict = Counter()
month_dict = Counter()
day_dict = Counter()

for continent in np.delete(continents, cont_to_impute, axis=0):
    continent_dict[continent]+=1

for month in np.delete(months, months_to_impute, axis=0):
    month_dict[month]+=1

for day in np.delete(days, days_to_impute, axis=0):
    day_dict[day]+=1

continent_map = dict(enumerate(continent_dict.keys(),1))
month_map = dict(enumerate(month_dict.keys(),1))
day_map = dict(enumerate(day_dict.keys(),1))

reverse_continent_map = dict([(value, key) for (key, value) in continent_map.
↪items()])
reverse_month_map = dict([(value, key) for (key, value) in month_map.items()])
```

```
reverse_day_map = dict([(value, key) for (key, value) in day_map.items()])

# encoded_continents = np.vectorize(reverse_continent_map.
 ↪get)(y_train_continents)
# encoded_months = np.vectorize(reverse_month_map.get)(y_train_months)
# encoded_days = np.vectorize(reverse_day_map.get)(y_train_days)

# y_train_cont_final = encoded_continents
# y_train_month_final = encoded_months
# y_train_day_final = encoded_days
```

[44]:
```
continents[continents!=''] = np.vectorize(reverse_continent_map.
 ↪get)(continents[continents!=''])
months[months!=''] = np.vectorize(reverse_month_map.get)(months[months!=''])
days[days!=''] = np.vectorize(reverse_day_map.get)(days[days!=''])
```

[45]:
```
X_train_continents = np.delete(X_nums, cont_to_impute, axis=0)
X_train_months = np.delete(X_nums, months_to_impute, axis=0)
X_train_days = np.delete(X_nums, days_to_impute, axis=0)

# X_test_continents = X_nums[cont_to_impute,:]
# X_test_months = X_nums[months_to_impute,:]
# X_test_days = X_nums[days_to_impute,:]

X_test_continents = X_nums[cont_to_impute]
X_test_months = X_nums[months_to_impute]
X_test_days = X_nums[days_to_impute]
```

[46]:
```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
```

### Imputing Continents

[47]:
```
neigh.fit(X_train_continents, continents[continents!=''])
continents[continents==''] = neigh.predict(X_test_continents)
```

[48]:
```
continents[continents=='']
```

[48]:
```
array([], dtype='<U5')
```

[49]:
```
continents = continents.astype('float64')
```

[50]:
```
continents
```

[50]:
```
array([1., 2., 2., ..., 2., 2., 2.])
```

### Imputing Months

[51]:
```
neigh.fit(X_train_months, months[months!=''])
months[months==''] = neigh.predict(X_test_months)
```

[52]:
```
months[months=='']
```

[52]:
```
array([], dtype='<U5')
```

[53]:
```
months = months.astype('float64')
```

[54]:
```
months
```

[54]:
```
array([1., 2., 1., ..., 3., 4., 2.])
```

### Imputing Days

[55]:
```
neigh.fit(X_train_days, days[days!=''])
days[days==''] = neigh.predict(X_test_days)
```

[56]:
```
days[days=='']
```

[56]:
```
array([], dtype='<U5')
```

[57]:
```
days = days.astype('float64')
```

[58]:
```
days
```

[58]:
```
array([1., 2., 2., ..., 2., 2., 1.])
```

### Add the categorical-encoded data to the numeric data

[59]:
```
X_new = np.hstack((X_nums, continents.reshape(160000,1), months.
 ↪reshape(160000,1), days.reshape(160000,1)))
```

[60]:
```
np.shape(X_new)
```

[60]:
```
(160000, 50)
```

[61]:
```
np.shape(y)
```

[61]:
```
(160000,)
```

[62]:
```
y[np.isnan(y)]
```

[62]:
```
Series([], Name: y, dtype: int64)
```

[63]:
```
np.unique(y)
```

[63]:
```
array([0, 1])
```

[124]:
```python
from sklearn.metrics import make_scorer, accuracy_score
from sklearn.ensemble import RandomForestClassifier
```

[98]:
```python
def cost_score(y, y_pred, fp_cost=25, fn_cost=125):
    '''
    '''
    # get the misclassifications
    misclass_idx = np.where(np.equal(y, y_pred) == False)[0]
    # get the false positives
    fp_idx = np.where(y_pred[misclass_idx] == 1)[0]
    # get the false negatives
    fn_idx = np.where(y_pred[misclass_idx] == 0)[0]
    # calc the misclassification cost
    misclassification_cost = fp_idx.size * fp_cost + fn_idx.size * fn_cost

    return misclassification_cost
```

[99]:
```python
cost_score(np.array([1,1,1,0,0,0]), np.array([0,1,1,0,1,1]))
```

[99]: 175

[100]:
```python
cost_scorer = make_scorer(cost_score, greater_is_better=False)
```

[127]:
```python
# instantiate estimator
logistic = LogisticRegression(solver='saga',
                              tol=1e-2,
                              max_iter=200,
                              random_state = random_state)
# define search space
penalty = ['l1', 'l2']
C = uniform(loc=0, scale=1000)
hyperparameters = dict(C=C, penalty=penalty)
# instantiate grid search
clf = RandomizedSearchCV(logistic,
                         hyperparameters,
                         random_state=random_state,
                         scoring=cost_scorer,
                         n_iter=100,
                         cv=5,
                         verbose=0,
                         n_jobs=-1)

# search for best parameters
search = clf.fit(X_train, y_train)

best_linear = search.best_estimator_
```

[128]:
```
print('Best model Score:', -search.best_score_) # negate since␣
↪'greater_is_better=False'
print('Best model Accuracy:', accuracy_score(y_train, best_linear.
↪predict(X_train)))
print('Best Penalty:', search.best_estimator_.get_params()['penalty'])
print('Best C:', search.best_estimator_.get_params()['C'])
```

```
Best model Score: 675705.0
Best model Accuracy: 0.70190625
Best Penalty: l1
Best C: 374.54011884736246
```

[130]:
```
y_pred = best_linear.predict(X_test)
linear_cost = cost_score(y_test, y_pred)
linear_acc = accuracy_score(y_test, y_pred)
print('Best Linear Model Test Cost', linear_cost)
print('Best Linear Model Test Accuracy', linear_acc)
```

```
Best Linear Model Test Cost 846700
Best Linear Model Test Accuracy 0.70275
```

[113]:
```
random_generator = np.random.RandomState(42)
```

[117]:
```
# instantiate estimator
rf = RandomForestClassifier(random_state=random_state)

rf_params = {
    'n_estimators': random_generator.randint(10, 2000, size=1000),
    'max_depth': random_generator.uniform(1, 1000, size=1000),
    'min_samples_split': random_generator.randint(2, 100, size=1000),
    'min_samples_leaf': random_generator.uniform(1e-6, 0.5, size=1000),
    'max_features': ['auto', 'sqrt', 'log2'],

}

# instantiate grid search
clf = RandomizedSearchCV(rf,
                         rf_params,
                         random_state=random_state,
                         scoring=cost_scorer,
                         n_iter=100,
                         cv=5,
                         verbose=0,
                         n_jobs=-1)

# search for best parameters
search = clf.fit(X_train, y_train)
```

[121]:
```
best_rf = search.best_estimator_
```

[125]:
```
print('Best model Score:', -search.best_score_) # negate since␣
↪'greater_is_better=False'
print('Best model Accuracy:', accuracy_score(y_train, best_rf.predict(X_train)))
print('n_estimators:', search.best_estimator_.get_params()['n_estimators'])
print('max_depth:', search.best_estimator_.get_params()['max_depth'])
print('min_samples_split:', search.best_estimator_.
↪get_params()['min_samples_split'])
print('min_samples_leaf:', search.best_estimator_.
↪get_params()['min_samples_leaf'])
print('max_features:', search.best_estimator_.get_params()['max_features'])
```

```
Best model Score: 350110.0
Best model Accuracy: 0.861453125
n_estimators: 10
max_depth: 551.5901948585391
min_samples_split: 81
min_samples_leaf: 0.0015881459179534207
max_features: sqrt
```

[126]:
```
y_pred = best_rf.predict(X_test)
rf_cost = cost_score(y_test, y_pred)
rf_acc = accuracy_score(y_test, y_pred)
print('Best RF Model Test Cost', rf_cost)
print('Best RF Model Test Accuracy', rf_acc)
```

```
Best RF Model Test Cost 451350
Best RF Model Test Accuracy 0.8504375
```

```
xgb_params = {
    'n_estimators': np.arange(100, 500, 10, dtype='int'),
    'learning_rate': np.linspace(0.01, 1, num=1000, dtype='float'),
    'gamma':np.geomspace(0.001, 10, num=1000, dtype='float'),
    'max_depth':[d for d in range(1, 11)],
    'subsample':np.linspace(0.1, 1, num=100, dtype='float'),
    'colsample_bytree':[0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'colsample_bylevel':[0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'colsample_bynode':[0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
    'lambda': np.geomspace(0.001, 10, num=100, dtype='float'),
    'alpha': np.geomspace(0.001, 10, num=100, dtype='float')
}

xgb = XGBClassifier(booster='gbtree',
                    early_stopping_rounds=10,
                    random_state=random_state)

xgb_search = RandomizedSearchCV(xgb,
```

```
                                xgb_params,
                                random_state=random_state,
                                scoring=cost_scorer,
                                n_iter=100,
                                cv=5,
                                verbose=0,
                                n_jobs=-1)

xgb_search.fit(X_train, y_train)

y_pred = xgb_search.best_estimator_.predict(X_train)

print('\n\n\nTraining Performance')

print('Best model Score:', -xgb_search.best_score_) # negate since 'greater_is_better=False'
print('Best model Accuracy:', accuracy_score(y_train, y_pred) )



y_pred = xgb_search.best_estimator_.predict(X_test)

test_cost = cost_score(y_test, y_pred)
test_acc = accuracy_score(y_test, y_pred)

print('\n\n\nTest Performance')

print('Best Model Test Cost', test_cost)
print('Best Model Test Accuracy', test_acc)

print('\n\n\nBest Parameters')
print(xgb_search.best_params_)
```

```
XGBoost Performance

Training Performance
Best model Score: 133475.0
Best model Accuracy: 0.9934453125



Test Performance
Best Model Test Cost 163775
Best Model Test Accuracy 0.93603125



Best Parameters
{'subsample': 0.9545454545454545, 'n_estimators': 170, 'max_depth': 9, 'learning_rate': 0.251801
```

[6]:
```python
# visualizing results
import matplotlib.pyplot as plt
import numpy as np
res_dict = {
    'Logistic Regression': 846950,
    'Random Forest': 451350,
    'XG Boost': 163775
}

bar_labels = ['$' + str(i) for i in res_dict.values()]
x_labels = np.arange(len(res_dict.keys()))

fig, ax = plt.subplots(figsize = (12,4))
y_pos = np.arange(len(res_dict.keys()))
ax.barh(y_pos, res_dict.values())
ax.set_yticks(y_pos)
ax.set_yticklabels(res_dict.keys(), minor=False)
ax.set_xlabel('U.S. Dollars')
ax.set_xticklabels([])
ax.set_title('Cost of Three Models')
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['bottom'].set_visible(False)
ax.set_xticks([])
rects = ax.patches

for rect, label in zip(rects, bar_labels):
    width = rect.get_width()
    ax.annotate(label,
            xy=(rect.get_x() + rect.get_width(), rect.get_y() + .33 ),
            xytext=(3, 3),
            textcoords='offset points',
            ha='left', va='center')

caption = 'Figure 2: Cost of model predictions on an out-of-fold test set.'
plt.figtext(0.5, 0.01, caption, wrap=True, horizontalalignment='center',␣
 ↪fontsize=14);

fig.savefig('./Model_Cost.png')
```

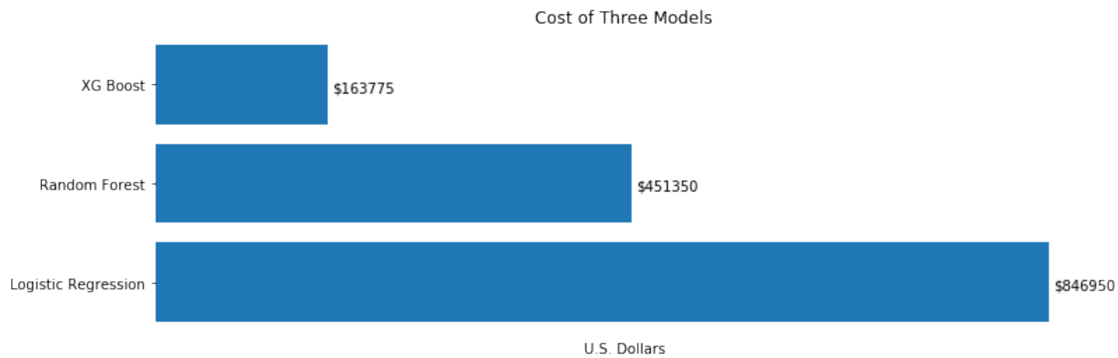


Figure 2: Cost of model predictions on an out-of-fold test set.

[ ]: