Project 4 - Regression Analysis

Jonathan Hurwitz (804258351) | Peter Kim (204271299)

March 5, 2018

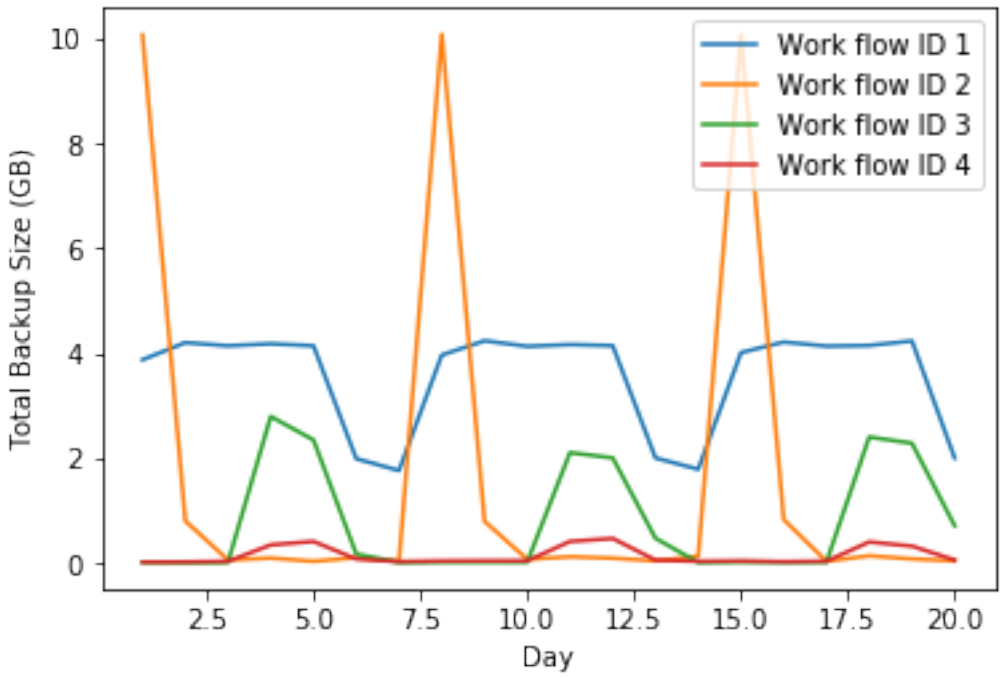
Code chunks are included in this report to help the reader understand how graphs and results were generated, and where our conclusions are coming from. However, not all code is shown. For the sake of brevity, we have removed code chunks that are redundant and essentially repeat behavior we've shown before. One example of this is in the case of k-Fold cross validation for different regressors, since the code is the same in all cases except for a few lines. Full code, as well as all figures and decision trees can be viewed in the Jupyter notebook submitted with this report, or on the GitHub reposity for this project: https://github.com/jdhurwitz/UCLA-EE219/tree/master/Project4.

1 Loading the Dataset

We first performed data preprocessing in order to convert categorical variables into numerical representations. This code is omitted for brevity. Please view the full notebook to see how the graphs below were generated.

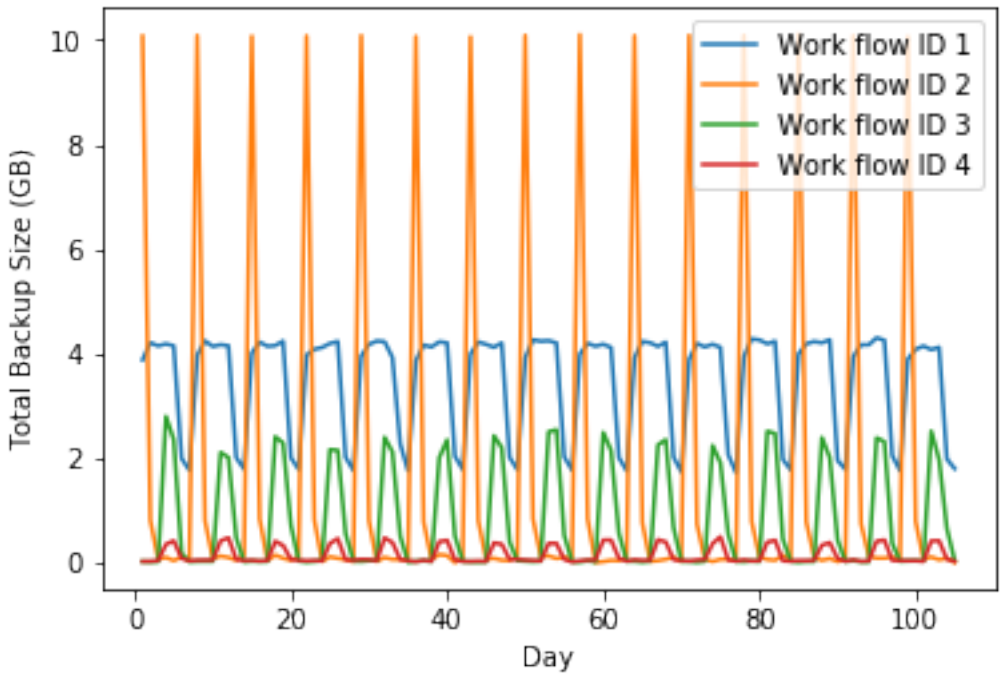
1.0.1 a) For a twenty-day period (X-axis unit is day number) plot the backup sizes for all workflows (color coded on the Y-axis)

The plot below shows the backup sizes for all workflows, plotted against 20 days. Clearly, the pattern is periodic. Backup size peaks the highest for work flow 2 at 10GB. The next highest peak is workflow 1, at 4GB. Workflow 3 peaks at around 2-2.5GB, and workflow 1 is less than 1GB.



1.0.2 b) Do the same plot for the first 105-day period. Can you identify any repeating patterns?

The graph is similar to that shown above, just on a larger time-axis. The same patterns exist. Workflow 2 has a large backup (10GB) that only occurs every ~7.5 days. The other workflows appear to have more frequent backups but of smaller sizes.



All the work flows have a very clear pattern of a spike in total backup size every week. For example, the work flow 2 consistently reaches a total backup size of 10G

2 Prediction of backup size

2.1 a) Linear Regression

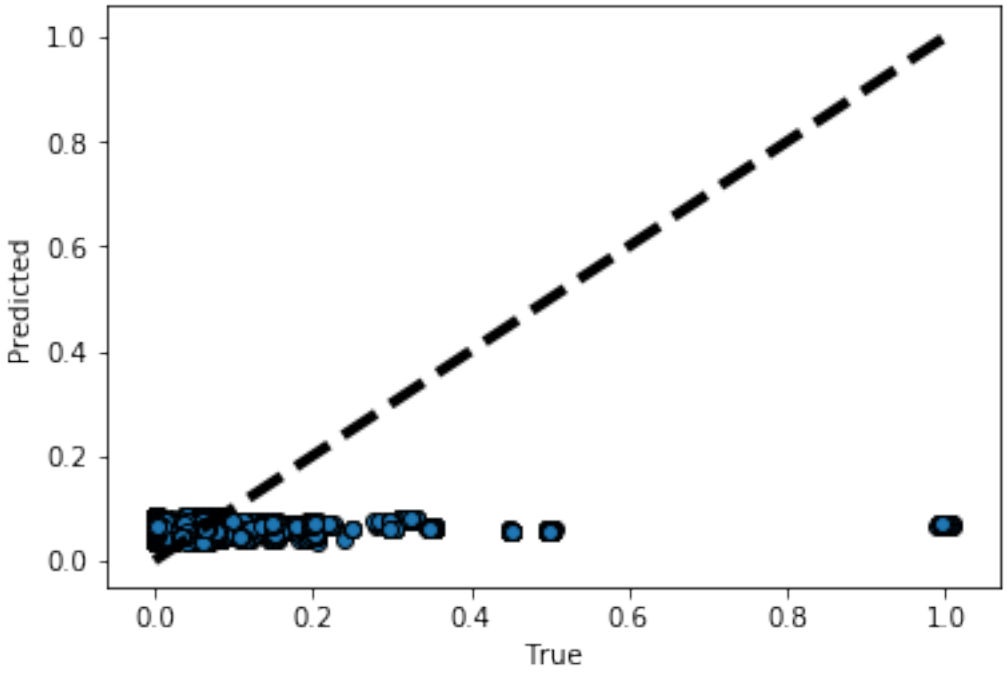
2.1.1 i) First convert each categorical feature into one dimensional numerical values using scalar encoding (e.g. Monday to Sunday can be mapped to 1-7), and then directly use them to fit a basic linear regression model.

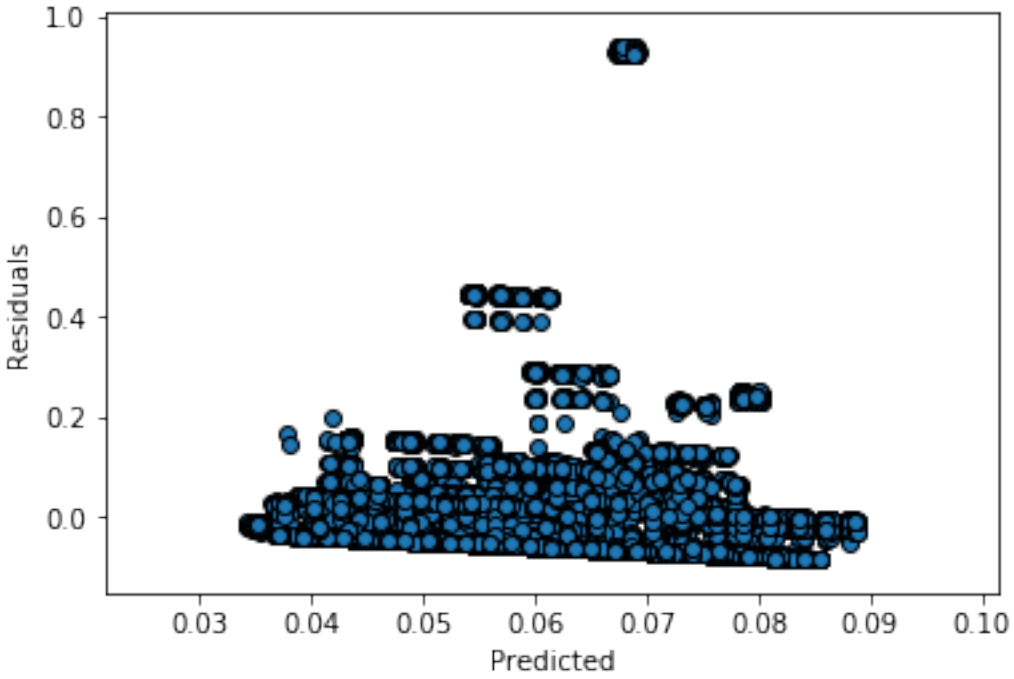
We used raw numerical encodings - Here, Monday through Sunday were mapped to 1 through 7, and the work-flow number and file-name were converted to its respective indices, i.e. the strings "work_flow_" and "File_" were removed. The results of fitting a linear model onto this dataset was as such:

```
In [238]: from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt

          X = data[:,0:5]
          y = data[:,5]

          kf = KFold(10)
```

```
k = 1
lr = linear_model.LinearRegression()
for train_index, test_index in kf.split(X): # Output 10 train test RMSE values
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    lr.fit(X_train, y_train)
    pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

    rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(mean_s
    print("Fold %i: train RMSE = %.3f, test RMSE = %.3f" % (k, rmse_train, rmse_test))
    k += 1

lr = linear_model.LinearRegression()
predicted = cross_val_predict(lr, X, y, cv=10)

fig, ax = plt.subplots()
ax.scatter(y, predicted, edgecolors=(0, 0, 0))
ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
ax.set_xlabel('True')
ax.set_ylabel('Predicted')
plt.show()

fig, ax = plt.subplots()
residuals = y - predicted
ax.scatter(predicted, residuals, edgecolors=(0, 0, 0))
ax.set_xlabel('Predicted')
ax.set_ylabel('Residuals')
plt.show()
```

```
Fold 1: train RMSE = 0.103, test RMSE = 0.107
Fold 2: train RMSE = 0.104, test RMSE = 0.100
Fold 3: train RMSE = 0.103, test RMSE = 0.107
Fold 4: train RMSE = 0.104, test RMSE = 0.100
Fold 5: train RMSE = 0.103, test RMSE = 0.107
Fold 6: train RMSE = 0.104, test RMSE = 0.100
Fold 7: train RMSE = 0.103, test RMSE = 0.107
Fold 8: train RMSE = 0.104, test RMSE = 0.100
Fold 9: train RMSE = 0.103, test RMSE = 0.107
Fold 10: train RMSE = 0.104, test RMSE = 0.100
```





The figure shows the naive model performs poorly for true backup sizes larger than 0.1 i.e. the model does not generalize well to the full range of backup sizes. However, from observing

the second residual vs. predicted graph, the model can fit well for the true backup sizes less than 0.1, which constitutes most of the examples in the dataset. It is clear that the residuals for most examples are in between 0 and 0.2.

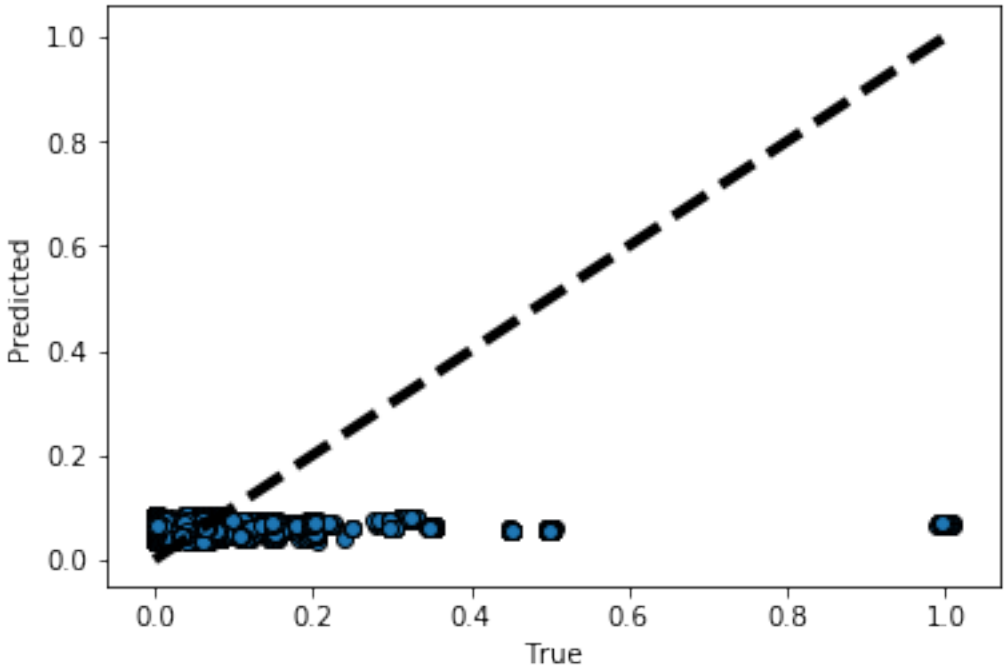
2.1.2 ii) Data Preprocessing

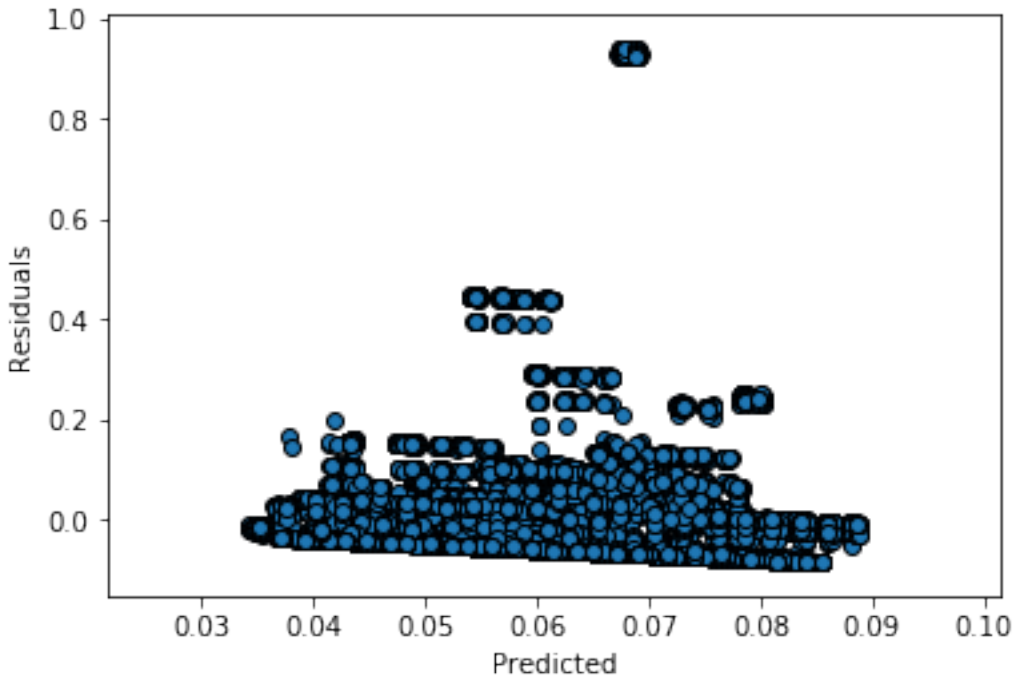
Here, we standardize numerical features to have zero mean and unit variance. The results are as such:

```
In [239]: from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt
          from sklearn.preprocessing import StandardScaler

          X = data[:,0:5]
          y = data[:,5]
          scaler = StandardScaler()
          X = scaler.fit_transform(X, y)


          kf = KFold(10)
          k = 1
          lr = linear_model.LinearRegression()
          for train_index, test_index in kf.split(X): # Output 10 train test RMSE values
              X_train, X_test = X[train_index], X[test_index]
              y_train, y_test = y[train_index], y[test_index]
              lr.fit(X_train, y_train)
              pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

              rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(mean_s
              print("Fold %i: train RMSE = %.3f, test RMSE = %.3f" % (k, rmse_train, rmse_test))
              k += 1

          lr = linear_model.LinearRegression()
          predicted = cross_val_predict(lr, X, y, cv=10)

          fig, ax = plt.subplots()
          ax.scatter(y, predicted, edgecolors=(0, 0, 0))
          ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
          ax.set_xlabel('True')
          ax.set_ylabel('Predicted')
          plt.show()

          fig, ax = plt.subplots()
          residuals = y - predicted
          ax.scatter(predicted, residuals, edgecolors=(0, 0, 0))
```

```
        ax.set_xlabel('Predicted')
        ax.set_ylabel('Residuals')
        plt.show()
```

```
Fold 1: train RMSE = 0.103, test RMSE = 0.107
Fold 2: train RMSE = 0.104, test RMSE = 0.100
Fold 3: train RMSE = 0.103, test RMSE = 0.107
Fold 4: train RMSE = 0.104, test RMSE = 0.100
Fold 5: train RMSE = 0.103, test RMSE = 0.107
Fold 6: train RMSE = 0.104, test RMSE = 0.100
Fold 7: train RMSE = 0.103, test RMSE = 0.107
Fold 8: train RMSE = 0.104, test RMSE = 0.100
Fold 9: train RMSE = 0.103, test RMSE = 0.107
Fold 10: train RMSE = 0.104, test RMSE = 0.100
```





The standardization of features does not affect the regression performance as one can see from the graphs above (they stay highly similar to the those of (i)). Overall, the fitting result does not change.

2.1.3 iii) Feature Selection

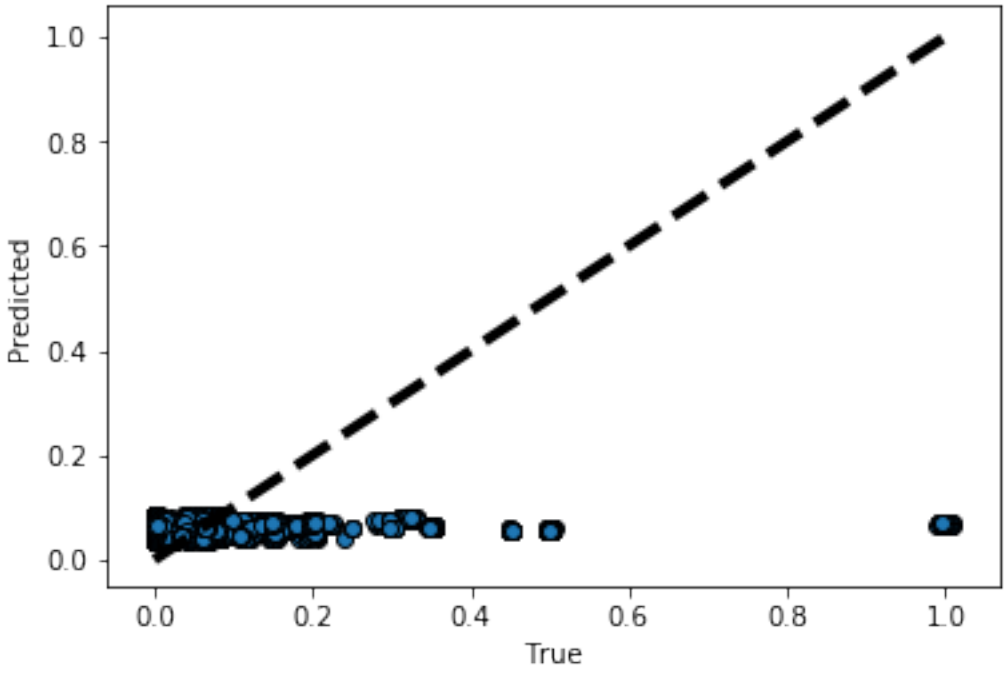
We will now determine the three most important features that influence the backup size, using f regression and mutual information regression measure.

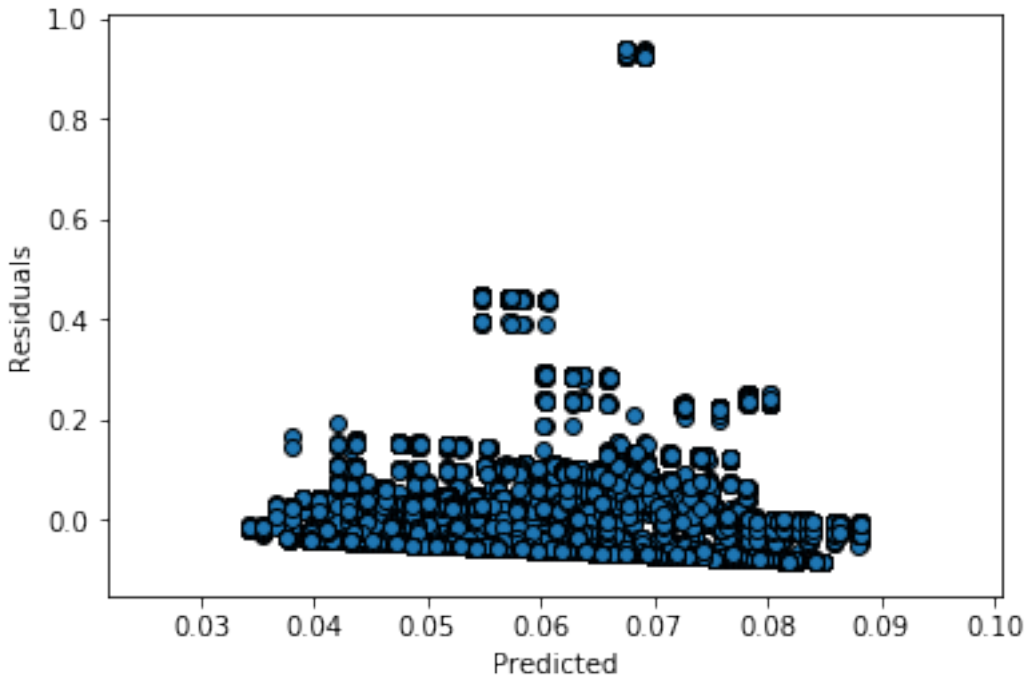
```
In [240]: from sklearn.feature_selection import SelectKBest, f_regression, mutual_info_regressio
          skb_f = SelectKBest(f_regression, k=3)
          X_new_f = skb_f.fit_transform(X, y)
          print("f_regression: " + str(skb_f.get_support()))
          skb_m = SelectKBest(mutual_info_regression, k=3)
          X_new_m = skb_m.fit_transform(X, y)
          print("mutual_info_regression: " + str(skb_m.get_support()))
```

```
f_regression: [False  True  True  True False]
mutual_info_regression: [False False  True  True  True]
```

Hence, the three most important features determined using f_regression are day of week, hour of day, and the work flow id. Conversely, the three most important features determined using mutual info regression are hour of day, work flow id, and the file id. Now we will use only these three features for each regression technique to compare the model performance. The same method of k-Fold cross-validation used above in the previous sections was used here. The only difference is that the train and test splits were generated on X_new_f (our transformed input) rather than x.

```
Fold 1: train RMSE = 0.103, test RMSE = 0.107
Fold 2: train RMSE = 0.104, test RMSE = 0.100
Fold 3: train RMSE = 0.103, test RMSE = 0.107
Fold 4: train RMSE = 0.104, test RMSE = 0.100
Fold 5: train RMSE = 0.103, test RMSE = 0.107
Fold 6: train RMSE = 0.104, test RMSE = 0.100
Fold 7: train RMSE = 0.103, test RMSE = 0.107
Fold 8: train RMSE = 0.104, test RMSE = 0.100
Fold 9: train RMSE = 0.103, test RMSE = 0.107
Fold 10: train RMSE = 0.104, test RMSE = 0.100
```

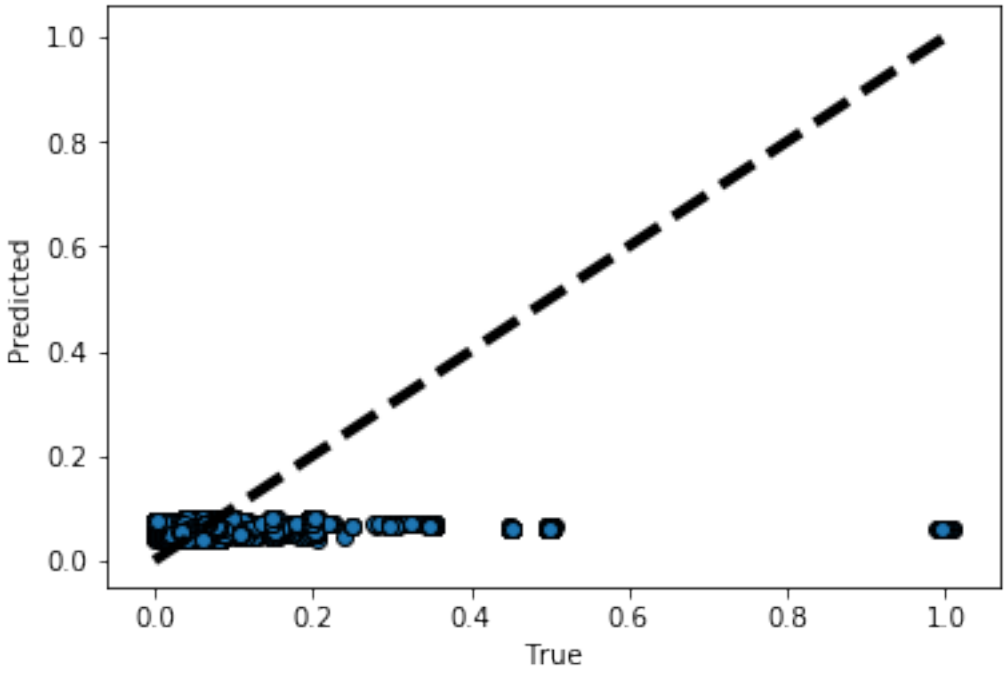


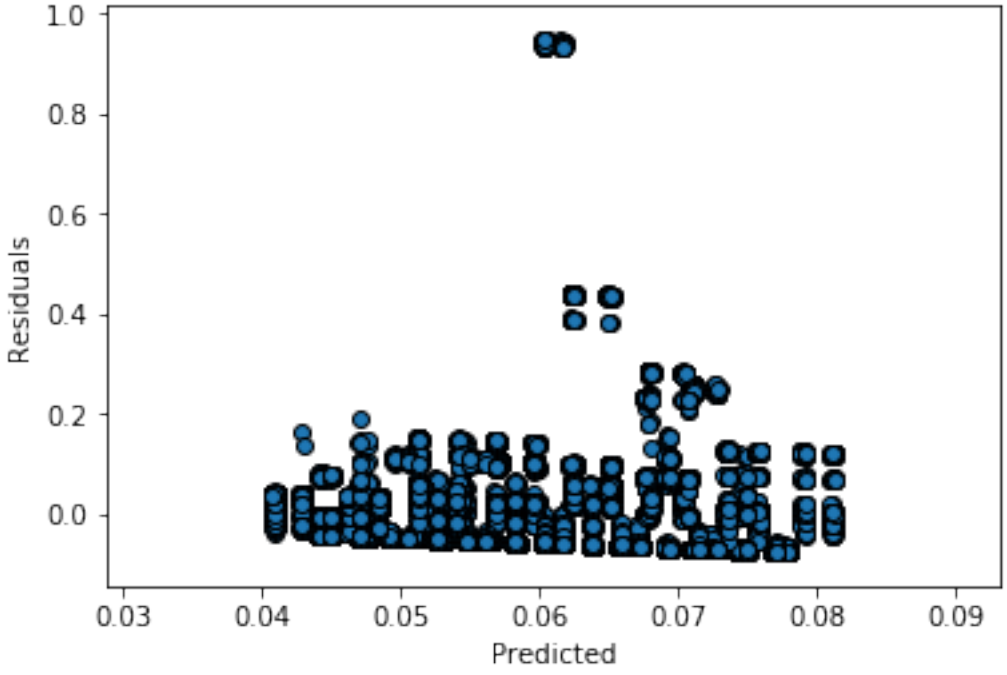


Using f regression, the model's performance is basically identical to those from (i) and (ii). This observation can be used as evidence to believe that the original linear model only weighs these three features to make the final prediction, essentially almost ignoring the other two features as they do not contribute as much. This can be viewed as good behavior when evaluating models, because it means that the machine learning model is able to figure out which features are most important and weigh those on its own. Models that perform feature selection effectively can also provide insights into the data when unpacking the model and looking at the weights.

The results below are shown using the same k-Fold cross validation procedure as above. However, the train and test splits are generated on X_new_m rather than X_new_f or X.

```
Fold 1: train RMSE = 0.103, test RMSE = 0.107
Fold 2: train RMSE = 0.104, test RMSE = 0.100
Fold 3: train RMSE = 0.103, test RMSE = 0.107
Fold 4: train RMSE = 0.104, test RMSE = 0.100
Fold 5: train RMSE = 0.103, test RMSE = 0.107
Fold 6: train RMSE = 0.104, test RMSE = 0.100
Fold 7: train RMSE = 0.103, test RMSE = 0.107
Fold 8: train RMSE = 0.104, test RMSE = 0.100
Fold 9: train RMSE = 0.103, test RMSE = 0.107
Fold 10: train RMSE = 0.104, test RMSE = 0.100
```





In the case of using mutual info regression, the two graphs above finally differs from the previous ones. The graphs indicate a decrease in regression performance as many residuals are now above 0, as there is no giant cluster of residuals ceneterd around 0.

2.1.4 iv) Feature Encoding

Here, we utilize 32 feature encodings that encapsulate all the feature combinations generated from 5 features (2^5 = 32). Plot the average training RMSE and test RMSE for each combination (in range 1 to 32). Which combinations achieve best performance? Can you provide an intuitive explanation?

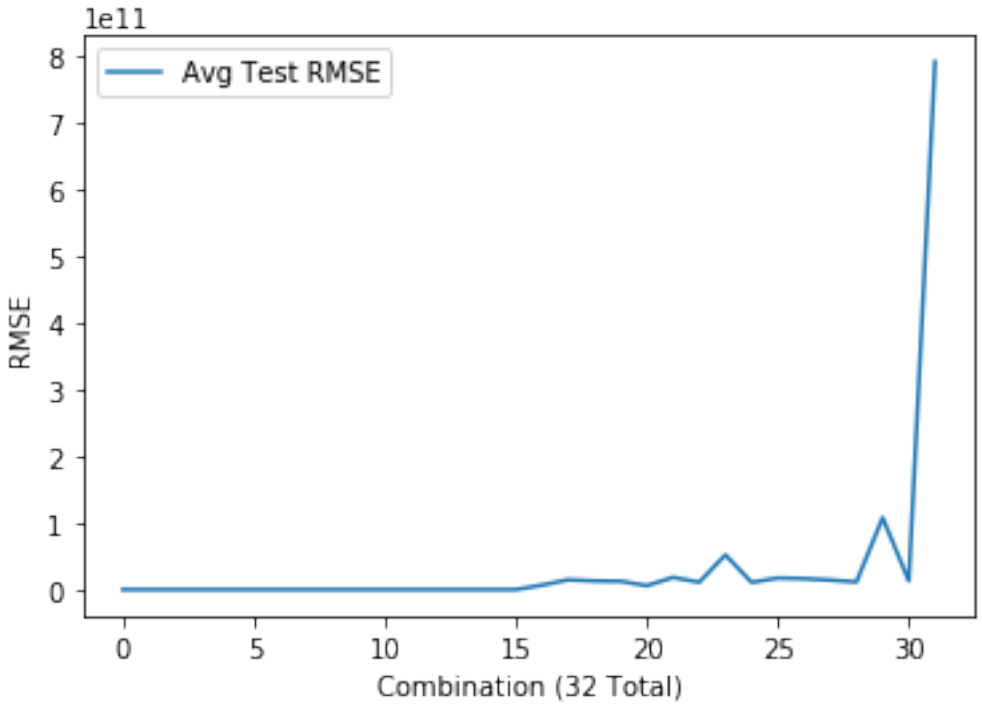
```
In [243]: import itertools
          from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt
          from sklearn.preprocessing import StandardScaler
          from sklearn.preprocessing import OneHotEncoder

          X = data[:,0:5]
          y = data[:,5]
          num_cols = 5
          table = list(itertools.product([False, True], repeat=num_cols))
          #print(table)
          avg_training_rmse = []
          avg_test_rmse = []
          for c, combination in enumerate(table):
              print(str(c) + " " + str(combination))
              enc = OneHotEncoder(categorical_features=combination, sparse=False)
              X_new = enc.fit_transform(X)

              kf = KFold(10)
              k = 1

              training_rmse = []
              test_rmse = []
              for train_index, test_index in kf.split(X_new): # Output 10 train test RMSE values
                  X_train, X_test = X_new[train_index], X_new[test_index]
                  y_train, y_test = y[train_index], y[test_index]
                  lr = linear_model.LinearRegression()
                  lr.fit(X_train, y_train)
                  pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  training_rmse.append(rmse_train)
                  test_rmse.append(rmse_test)
                  k += 1

              avg_training_rmse.append(np.mean(training_rmse))
              avg_test_rmse.append(np.mean(test_rmse))
```

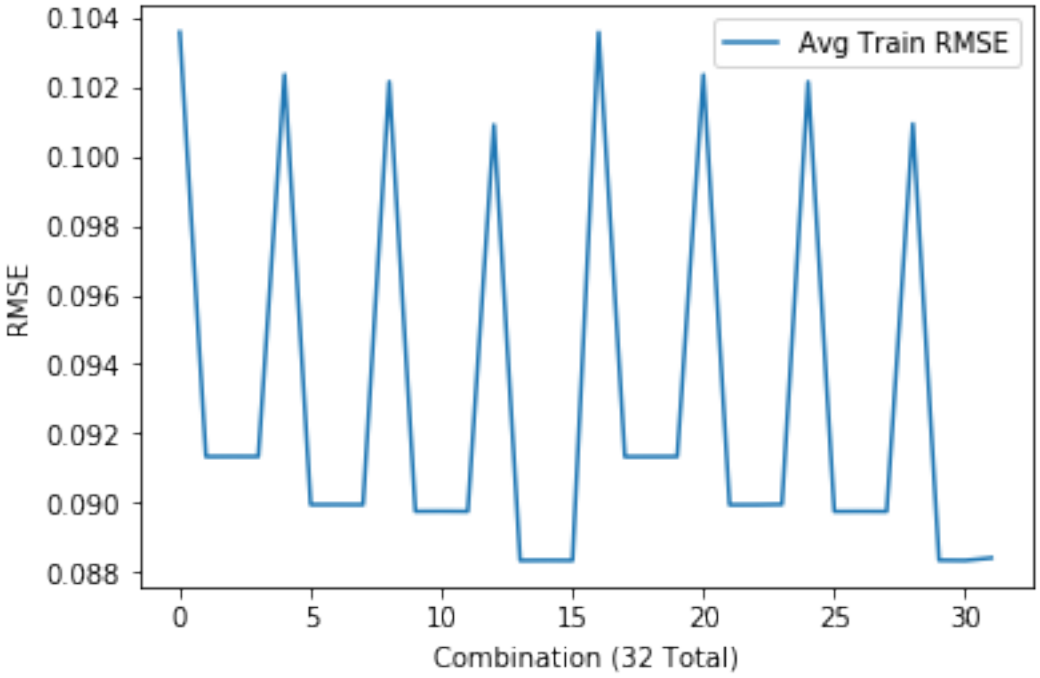
```
          print(np.argmin(avg_training_rmse), np.argmin(avg_test_rmse))
          print(avg_training_rmse[13], avg_test_rmse[13])
```

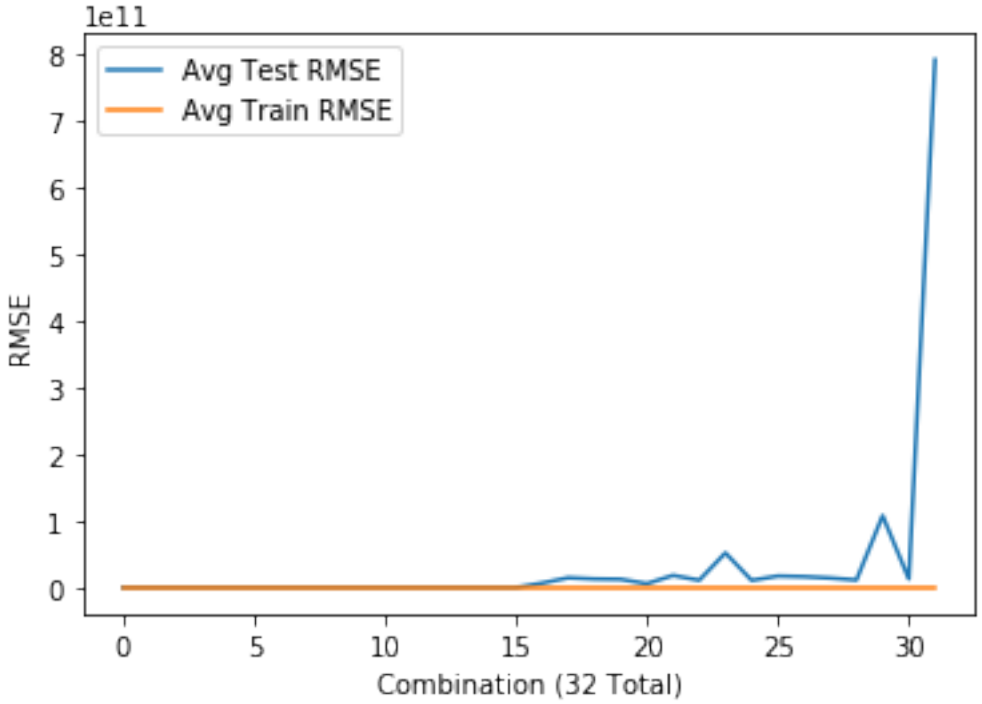
```
0 (False, False, False, False, False)
1 (False, False, False, False, True)
2 (False, False, False, True, False)
3 (False, False, False, True, True)
4 (False, False, True, False, False)
5 (False, False, True, False, True)
6 (False, False, True, True, False)
7 (False, False, True, True, True)
8 (False, True, False, False, False)
9 (False, True, False, False, True)
10 (False, True, False, True, False)
11 (False, True, False, True, True)
12 (False, True, True, False, False)
13 (False, True, True, False, True)
14 (False, True, True, True, False)
15 (False, True, True, True, True)
16 (True, False, False, False, False)
17 (True, False, False, False, True)
18 (True, False, False, True, False)
19 (True, False, False, True, True)
20 (True, False, True, False, False)
21 (True, False, True, False, True)
22 (True, False, True, True, False)
23 (True, False, True, True, True)
24 (True, True, False, False, False)
25 (True, True, False, False, True)
26 (True, True, False, True, False)
27 (True, True, False, True, True)
28 (True, True, True, False, False)
29 (True, True, True, False, True)
30 (True, True, True, True, False)
31 (True, True, True, True, True)
13 13
0.0883364063914 0.0883701240192
```

In [245]:
```
#Figure 1
fig, ax = plt.subplots()
ax.plot(avg_test_rmse, label="Avg Test RMSE")

ax.set_xlabel('Combination (32 Total)')
ax.set_ylabel('RMSE')
plt.legend()
plt.show()
```

```
#Figure 2
fig, ax = plt.subplots()
ax.plot(avg_training_rmse, label="Avg Train RMSE")
ax.set_xlabel('Combination (32 Total)')
ax.set_ylabel('RMSE')
plt.legend()
plt.show()

#Figure 3
fig, ax = plt.subplots()
ax.plot(avg_test_rmse, label="Avg Test RMSE")
ax.plot(avg_training_rmse, label="Avg Train RMSE")

ax.set_xlabel('Combination (32 Total)')
ax.set_ylabel('RMSE')
plt.legend()
plt.show()
```







From the graphs above, it is clear that for all combiations, the train RMSE stays consistently low around 0.08. However, the test RMSE shows the true predictive performance of each com-

bination Combination 16, (False, True, True, True, True), had the lowest test RMSE of 0.0884. In this combination, every feature, except the week #, was converted into a 1hot categorical representation. Intuitively, this makes sense as converting the week # into a 1hot representation is not reasonable as week # will continously increase since it is a timeseries feature (one would have to add one extra column for every single new week we observe). Every other feature, however, can clearly be converted into a 1hot representation as each feature is naturally categorical e.g. day of week and hour of day.

2.2 v) Controlling ill-conditioning and over-fitting

You should have found obvious increases in test RMSE compared to training RMSE in some combinations, can you explain why this happens? Observe those fitted coefficients. To solve this problem, you can try the following regularizations with suitable parameters.

2.2.1 Ridge Regularization

In [246]:
```
#Find the best combination for use with ridge regression
from sklearn.linear_model import Ridge

X = data[:,0:5]
y = data[:,5]
num_cols = 5
table = list(itertools.product([False, True], repeat=num_cols))
avg_training_rmse = []
avg_test_rmse = []
for c, combination in enumerate(table):
    print(str(c) + " " + str(combination))
    enc = OneHotEncoder(categorical_features=combination, sparse=False)
    X_new = enc.fit_transform(X)

    kf = KFold(10)
    k = 1

    training_rmse = []
    test_rmse = []
    for train_index, test_index in kf.split(X_new): # Output 10 train test RMSE values
        X_train, X_test = X_new[train_index], X_new[test_index]
        y_train, y_test = y[train_index], y[test_index]
        rr = Ridge()
        rr.fit(X_train, y_train)
        pred_train, pred_test = rr.predict(X_train), rr.predict(X_test)

        rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
        training_rmse.append(rmse_train)
        test_rmse.append(rmse_test)
        k += 1
    avg_training_rmse.append(np.mean(training_rmse))
    avg_test_rmse.append(np.mean(test_rmse))
```

```
In [247]: print("Min index training RMSE: ", np.argmin(avg_training_rmse), "Min index test RMSE:

Min index training RMSE:  31 Min index test RMSE:  14


In [248]: print(avg_training_rmse[31], avg_test_rmse[14])

0.0883330325949 0.0883677632029
```

For the ridge regression, the minimum test RMSE occurs for combination 15 (zero indexed, so 14). This combination is: (False, True, True, True, False). This encoding scheme will be used for the optimizations.

```
In [249]: kf = KFold(10)
          k = 1
          n_alphas = 200
          alphas = np.logspace(-10, -2, n_alphas)

          #set up X
          X = data[:,0:5]
          enc = OneHotEncoder(categorical_features=(False, True, True, True, False), sparse=Fals
          X_new = enc.fit_transform(X)

          avg_training_rmse = []
          avg_test_rmse = []
          for alpha in alphas:
              training_rmse = []
              test_rmse = []
              for train_index, test_index in kf.split(X_new): # Output 10 train test RMSE values
                  X_train, X_test = X_new[train_index], X_new[test_index]
                  y_train, y_test = y[train_index], y[test_index]
                  rr = Ridge(alpha=alpha)
                  rr.fit(X_train, y_train)
                  pred_train, pred_test = rr.predict(X_train), rr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  training_rmse.append(rmse_train)
                  test_rmse.append(rmse_test)
                  k += 1
              avg_training_rmse.append(np.mean(training_rmse))
              avg_test_rmse.append(np.mean(test_rmse))

In [250]: best_alpha_train = alphas[np.argmin(avg_training_rmse)]
          best_alpha_test = alphas[np.argmin(avg_test_rmse)]
          print("Best alpha (train)", best_alpha_train)
          print("Best alpha (test)", best_alpha_test)
```

```
Best alpha (train) 1e-10
Best alpha (test) 0.01
```

The best performing combination of one-hot vectors and scalar features for Ridge regression was (False, True, True, True, False). Using this, the best alpha value was optimized and 10-fold cross-validation was used. Alphas were generated via np.logspace(-10, -2, n_alphas), where n_alphas=200. The best alpha for Ridge regression was 0.01. Smaller values of alpha would indicate that we prefer to apply more emphasis to the weights rather than the regularizer, meaning we "trust" our data more.

2.2.2 Lasso Regularization

```
In [251]: #Find the best combination for use with lasso regression
          from sklearn.linear_model import Lasso

          X = data[:,0:5]
          y = data[:,5]
          num_cols = 5
          table = list(itertools.product([False, True], repeat=num_cols))
          avg_training_rmse = []
          avg_test_rmse = []
          for c, combination in enumerate(table):
              print(str(c) + " " + str(combination))
              enc = OneHotEncoder(categorical_features=combination, sparse=False)
              X_new = enc.fit_transform(X)

              kf = KFold(10)
              k = 1

              training_rmse = []
              test_rmse = []
              for train_index, test_index in kf.split(X_new): # Output 10 train test RMSE values
                  X_train, X_test = X_new[train_index], X_new[test_index]
                  y_train, y_test = y[train_index], y[test_index]
                  lr = Lasso()
                  lr.fit(X_train, y_train)
                  pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  training_rmse.append(rmse_train)
                  test_rmse.append(rmse_test)
                  k += 1
              avg_training_rmse.append(np.mean(training_rmse))
              avg_test_rmse.append(np.mean(test_rmse))


In [252]: print("Min index training RMSE: ", np.argmin(avg_training_rmse), "Min index test RMSE:
          print(avg_training_rmse[0], avg_test_rmse[0])
```

```
Min index training RMSE:  0 Min index test RMSE:  0
0.104189007948 0.104139930717
```

```
In [253]: #Determine the best alpha for lasso regression
          kf = KFold(10)
          k = 1
          n_alphas = 200
          alphas = np.logspace(-10, -2, n_alphas)

          #set up X
          X = data[:,0:5]
          enc = OneHotEncoder(categorical_features=(False, False, False, False, False), sparse=F
          X_new = enc.fit_transform(X)
          print(X_new.shape)

          avg_training_rmse = []
          avg_test_rmse = []
          for alpha in alphas:
              training_rmse = []
              test_rmse = []
              for train_index, test_index in kf.split(X_new): # Output 10 train test RMSE values
                  X_train, X_test = X_new[train_index], X_new[test_index]
                  y_train, y_test = y[train_index], y[test_index]
                  lr = Lasso(alpha=alpha)
                  lr.fit(X_train, y_train)
                  pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  training_rmse.append(rmse_train)
                  test_rmse.append(rmse_test)
                  k += 1
              avg_training_rmse.append(np.mean(training_rmse))
              avg_test_rmse.append(np.mean(test_rmse))
```

```
(18588, 5)
```

```
In [254]: best_alpha_train = alphas[np.argmin(avg_training_rmse)]
          best_alpha_test = alphas[np.argmin(avg_test_rmse)]
          print("Best alpha (train)", best_alpha_train)
          print("Best alpha (test)", best_alpha_test)
```

```
Best alpha (train) 1e-10
Best alpha (test) 0.00108436596869
```

The best performing combination for Lasso regularization was (False, False, False, False, False), meaning the regularized model performs better with all features as scalars and none transformed

to be one-hot vectors. Using this, the best alpha was determined with 10-fold cross validation. Alpha values were generated via np.logspace(-10, -2, n_alphas), with n_alphas set to 200. The best value was alpha=0.001.

2.2.3 Compare the values of the estimated coefficients for these regularized good models, with the un-regularized best model.

From cross-validation, the best alpha value for the ridge regularizer (L2) is 0.01. For the lasso regularizer (L1), it is 0.001. These differ by an order of magnitude. Since L1 regularization results in more sparsity, it's possible that coefficients smaller than those used for L2 are appropriate in order to reconcile sparsity and weight sizes.

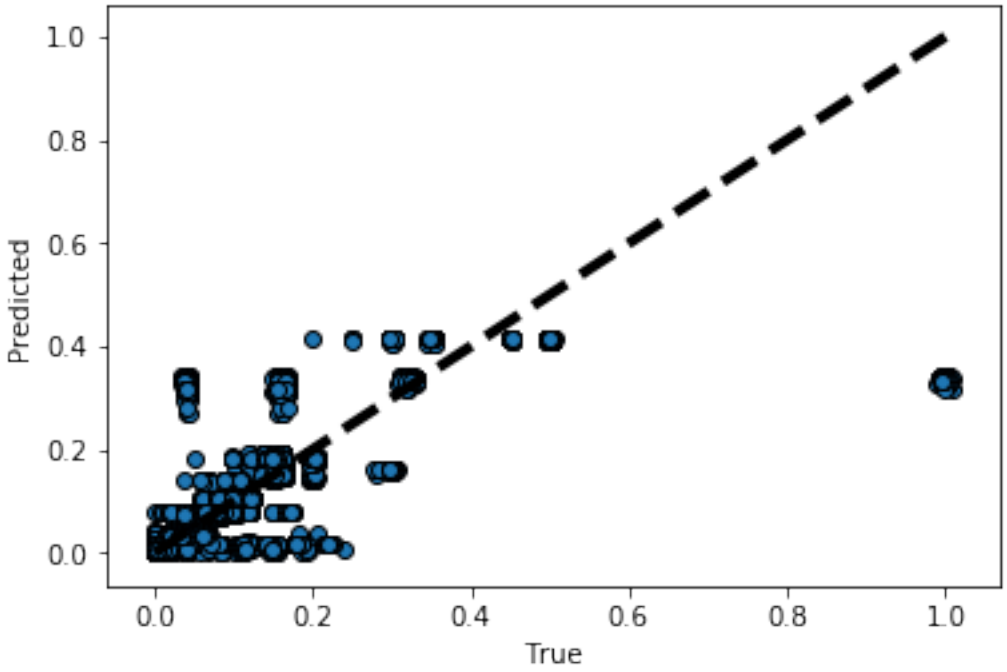
2.3 b) Random Forest Regression

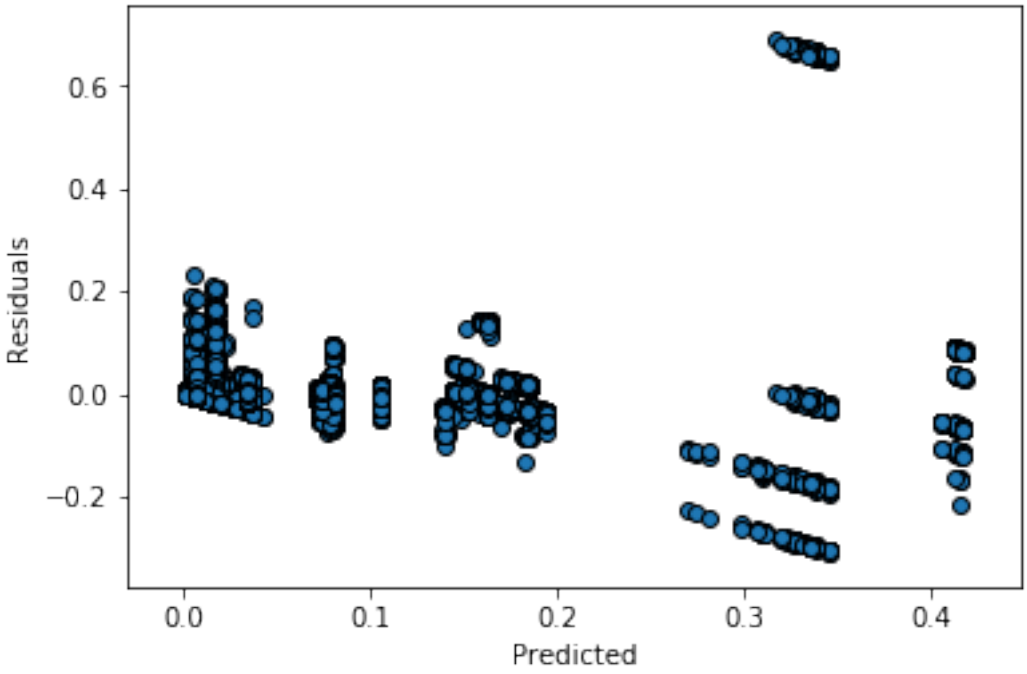
2.3.1 i) Report training and average test RMSE from 10 fold cross validation, as well as out of bag error.

```
In [255]: from sklearn.ensemble import RandomForestRegressor

          rf = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=4,
                                     bootstrap=True, oob_score=False, n_jobs=-1)
          kf = KFold(10)

          k = 1
          for train_index, test_index in kf.split(X):
              X_train, X_test = X[train_index], X[test_index]
              y_train, y_test = y[train_index], y[test_index]
              rf.fit(X_train, y_train)

              pred_train, pred_test = rf.predict(X_train), rf.predict(X_test)

              rmse_train = sqrt(mean_squared_error(y_train, pred_train))
              rmse_test = sqrt(mean_squared_error(y_test, pred_test))

              print("Fold %i: train RMSE = %.3f, test RMSE = %.3f" % (k, rmse_train, rmse_test))
              k += 1

          rf = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=4,
                                     bootstrap=True, n_jobs=-1)
          predicted = cross_val_predict(rf, X, y, cv=10)

          rf = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=4,
                                     bootstrap=True, oob_score=True, n_jobs=-1)
          rf.fit(X, y)
          oob = rf.oob_score_
          print("Out of bag error: ", oob)

          fig, ax = plt.subplots()
```

```
        ax.scatter(y, predicted, edgecolors=(0, 0, 0))
        ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
        ax.set_xlabel('True')
        ax.set_ylabel('Predicted')
        plt.show()

        fig, ax = plt.subplots()
        residuals = y - predicted
        ax.scatter(predicted, residuals, edgecolors=(0, 0, 0))
        ax.set_xlabel('Predicted')
        ax.set_ylabel('Residuals')
        plt.show()
```

```
Fold 1: train RMSE = 0.060, test RMSE = 0.067
Fold 2: train RMSE = 0.060, test RMSE = 0.052
Fold 3: train RMSE = 0.060, test RMSE = 0.067
Fold 4: train RMSE = 0.061, test RMSE = 0.053
Fold 5: train RMSE = 0.060, test RMSE = 0.067
Fold 6: train RMSE = 0.060, test RMSE = 0.053
Fold 7: train RMSE = 0.060, test RMSE = 0.068
Fold 8: train RMSE = 0.061, test RMSE = 0.052
Fold 9: train RMSE = 0.060, test RMSE = 0.067
Fold 10: train RMSE = 0.062, test RMSE = 0.053
Out of bag error:  0.6632403926
```



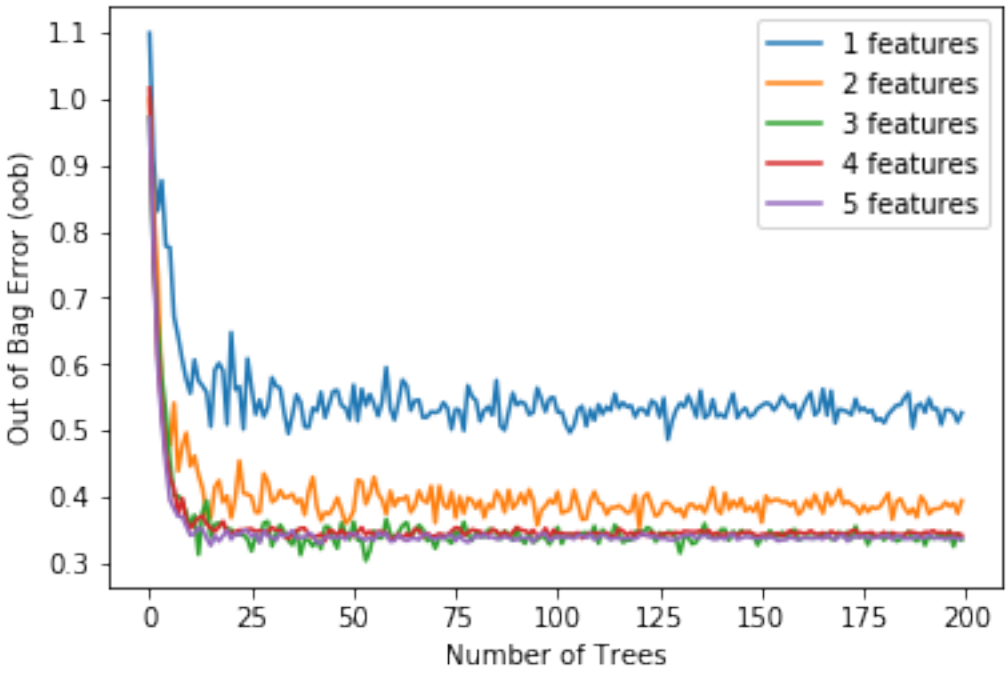


2.3.2 ii) Sweep over number of trees from 1 to 200 and maximum number of features from 1 to 5. Plot out of bag error (y-axis) vs. number of trees (x-axis) as figure 1. Plot average test RMSE (y-axis) against number of trees (x axis).

```
In [256]: import warnings
          warnings.filterwarnings('ignore')

          num_trees_min = 1
          num_trees_max = 200
          num_features_min = 1
          num_features_max = 5

          bag_errors = []
          rmses = []
          for num_features in range(num_features_min, num_features_max+1):
              bag_errors_for_cur_feature = []
              rmses_cur_feature = []
              for num_trees in range(num_trees_min, num_trees_max+1):
                  rf = RandomForestRegressor(n_estimators=num_trees, max_features=num_features,
                                             bootstrap=True, oob_score=True, n_jobs=-1)
                  rf.fit(X_train, y_train)

                  y_pred = rf.predict(X_test)
                  bag_errors_for_cur_feature.append(1-rf.oob_score_)
```

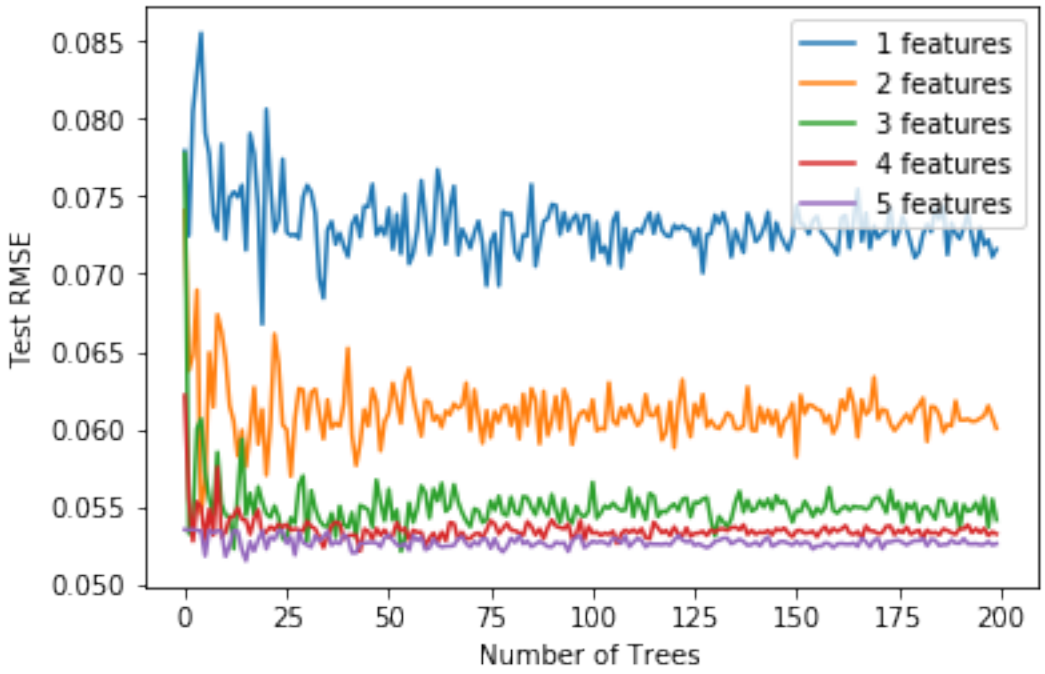
```
            rmses_cur_feature.append(sqrt(mean_squared_error(y_test, y_pred)))

        rmses.append(rmses_cur_feature)
        bag_errors.append(bag_errors_for_cur_feature)
```

In [257]:
```
#Figure 1
fig, ax = plt.subplots()
i = 0
for oob in bag_errors:
    ax.plot(oob, label=str(i+1)+" features")
    i+=1

ax.set_xlabel('Number of Trees')
ax.set_ylabel('Out of Bag Error (oob)')
plt.legend()
plt.show()
```



In [258]:
```
fig, ax = plt.subplots()

i=0
for rmse in rmses:
    ax.plot(rmse, label=str(i+1)+" features")
```

```
    i+=1

ax.set_xlabel('Number of Trees')
ax.set_ylabel('Test RMSE')
plt.legend()
plt.show()
```



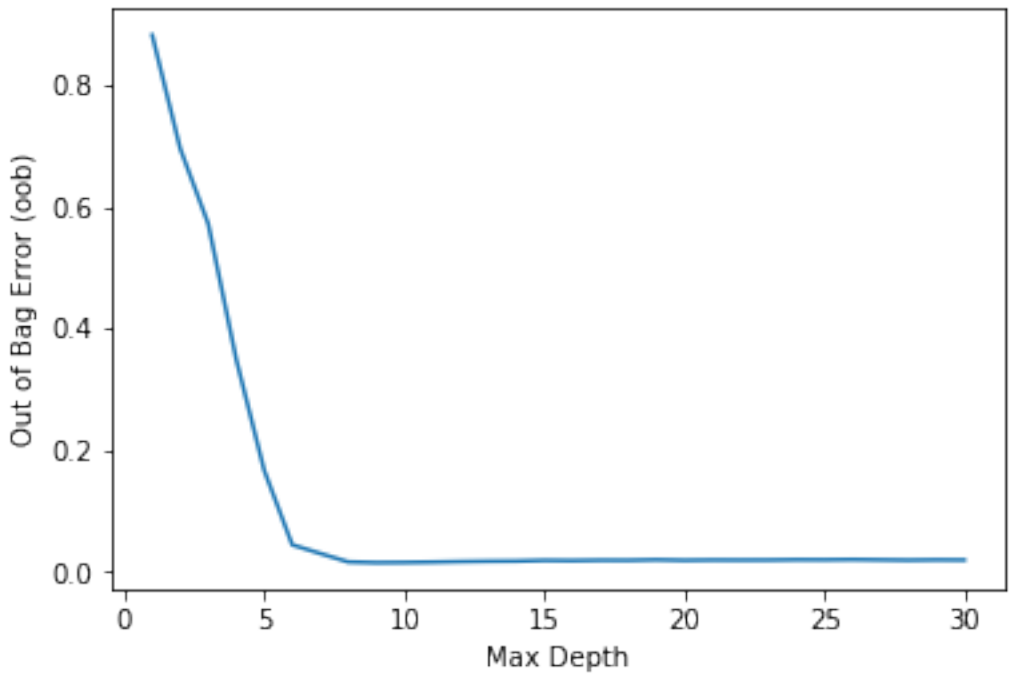
2.3.3 iii) Choose another parameter to tune

max_depth was chosen as another parameter to tune. Our hypothesis is that increasing this value too much will cause the random forest to overfit the data. This will be shown in the plots and is denoted by progressively worse and worse test errors. However, it's possible that test error may actually reduce up to a certain point, at which overfitting will cause it to increase.

```
In [259]: import warnings
          warnings.filterwarnings('ignore')

          min_depth = 1
          max_depth = 30

          bag_errors = []
          test_rmses = []
          train_rmses = []
          depths = np.linspace(1, max_depth, max_depth)
```

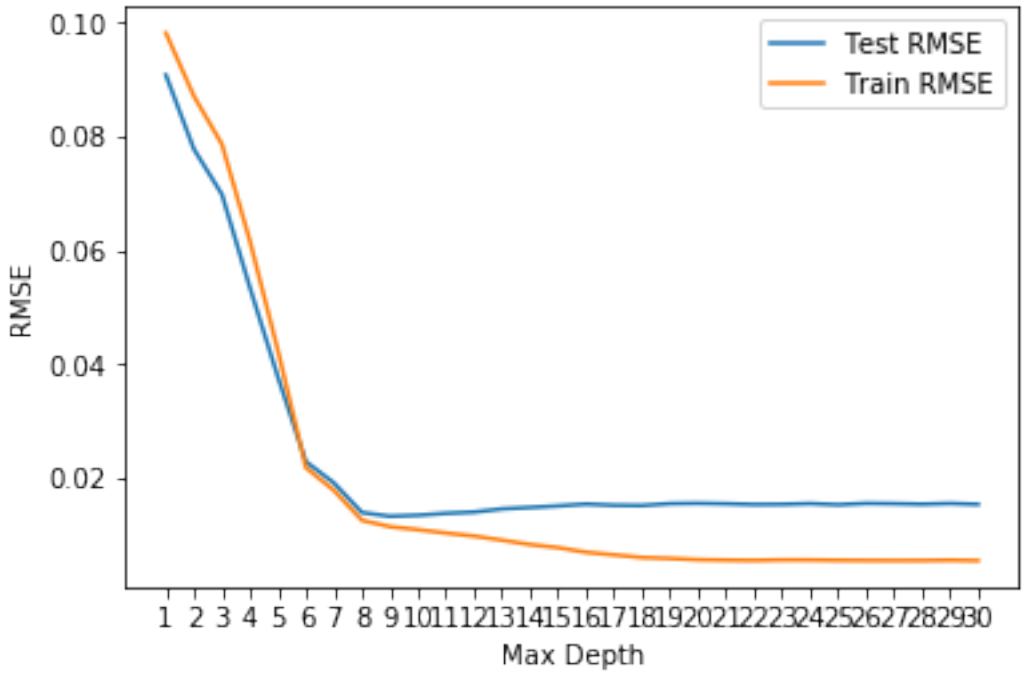
```
        for depth in range(min_depth, max_depth+1):
            rf = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=depth,
                                       bootstrap=True, oob_score=True, n_jobs=-1)
            rf.fit(X_train, y_train)

            y_pred = rf.predict(X_test)
            y_pred_train = rf.predict(X_train)
            bag_errors.append(1-rf.oob_score_)
            train_rmses.append(sqrt(mean_squared_error(y_train, y_pred_train)))
            test_rmses.append(sqrt(mean_squared_error(y_test, y_pred)))
```

In [260]:
```
#Figure 1
fig, ax = plt.subplots()
ax.plot(depths, bag_errors)

ax.set_xlabel('Max Depth')
ax.set_ylabel('Out of Bag Error (oob)')
plt.legend()
plt.show()
```



In [261]:
```
fig, ax = plt.subplots()

ax.plot(depths, test_rmses, label="Test RMSE")
ax.plot(depths, train_rmses, label="Train RMSE")
```

```
ax.set_xticks(depths)
ax.set_xlabel('Max Depth')
ax.set_ylabel('RMSE')
plt.legend()
plt.show()
```



For this test, n_estimators was set to 20 and max_features to 5. This was because these values appeared to have good performance in the section above. The graphs for random forests with max_features=5 had the lowest out of bag error as well as test RMSE. With 5 features, there is also less variation with respect to the number of trees used in the ensemble. By looking at the plot of OOB vs. max depth and Test RMSE vs. max depth, it's clear that adding more and more depth to the random forest causes overfitting. The plot showing both test RMSE and train RMSe vs. max depth can help us choose an optimal depth. Max depth of 8 appears ot be best for this data set. After 8, there is a spread between the test and train RMSE. This spread is characteristically overfitting, as indicated by test RMSE increasing while train RMSE is decreasing.

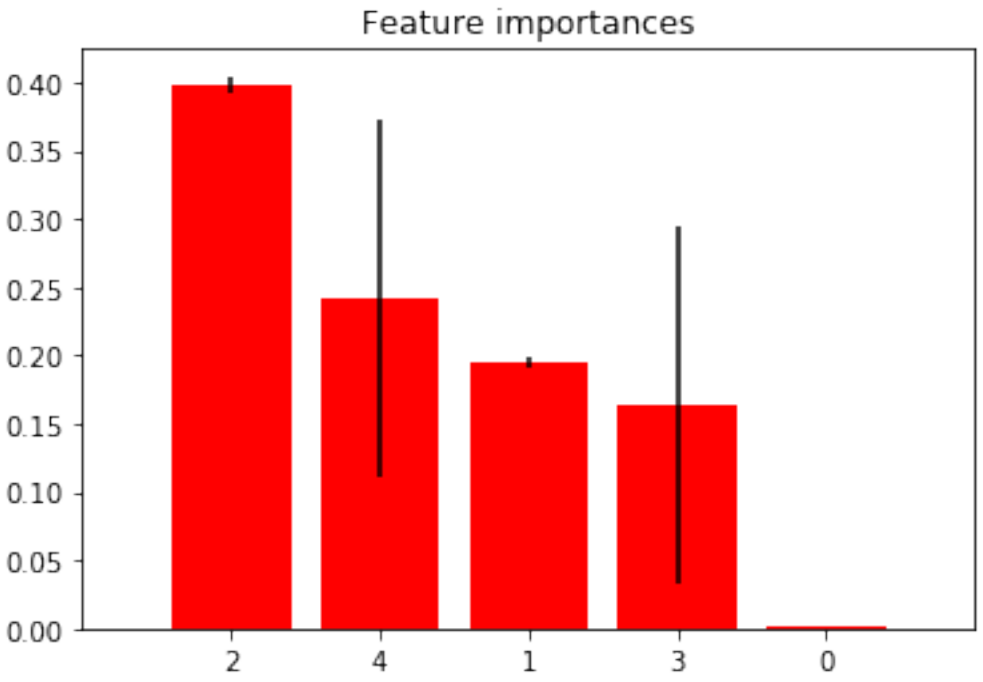
2.3.4 iv) Report the feature importances you got from the best random forest regression.

The plot below shows which features are the most informative and which are not. Clearly, feature 2 is the most informative, followed by features 4, 1, and 3. Feature 0 i sthe least informative. This graph would indicate that an optimal model in terms of efficiency and capacity would use features 1-4 but omit feature 0.

```
In [262]: forest = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=8,
                                bootstrap=True, oob_score=True, n_jobs=-1)
          forest.fit(X, y)
```

```
importances = forest.feature_importances_

std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(X.shape[1]), importances[indices],
       color="r", yerr=std[indices], align="center")
plt.xticks(range(X.shape[1]), indices)
plt.xlim([-1, X.shape[1]])
plt.show()
```

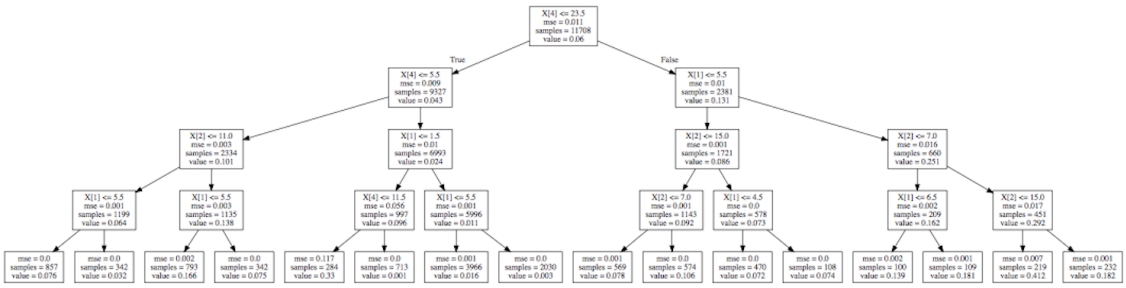
```
Feature ranking:
1. feature 2 (0.398167)
2. feature 4 (0.242020)
3. feature 1 (0.195136)
4. feature 3 (0.163444)
5. feature 0 (0.001233)
```



2.3.5 v) Visualize your decision trees. Pick any tree (estimator) in best random forest (with max depth=4) and plot its structure, which is the root node in this decision tree? Is it the most important feature according to the feature importance reported by the regressor?

```
In [273]: from sklearn.tree import export_graphviz
          import graphviz

          #export_graphviz()
          forest = RandomForestRegressor(n_estimators=20, max_features=5, max_depth=4,
                              bootstrap=True, oob_score=True, n_jobs=-1)
          forest.fit(X, y)
          estimator = forest.estimators_[0]

          export_graphviz(estimator, out_file='tree.dot')
          with open("tree.dot") as f:
              dot_graph = f.read()

          #graphviz.Source(dot_graph)
```

The 0-th tree in the random forest is shown below. A screenshot of it is included in the folder we have submitted, and it's also viewable in the Jupyter notebook at full size. Clearly, we can see that this is a tree of depth 4, as specified in the random forest regressor model declaration.



2.4 c) Neural Network Regression

```
In [264]: from sklearn.preprocessing import OneHotEncoder
          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=


          enc = OneHotEncoder()
          train_encoded = enc.fit_transform(X_train)
          train_encoded_arr = train_encoded.toarray()
```

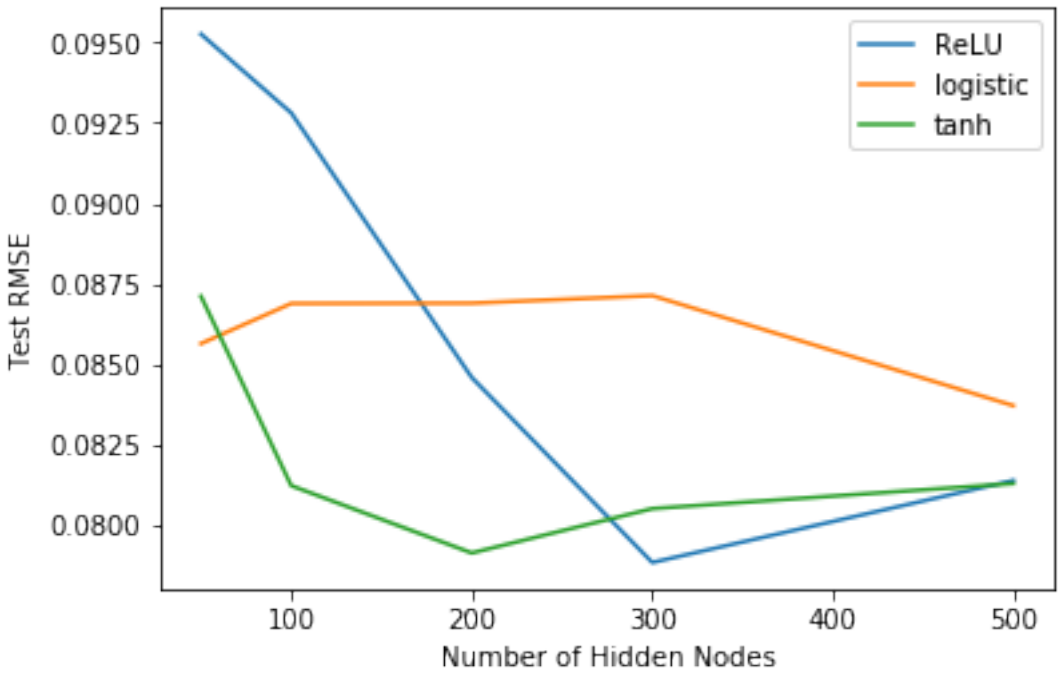
```
        test_encoded = enc.transform(X_test)
        test_encoded_arr = test_encoded.toarray()
```

In [265]:
```
        print(train_encoded_arr.shape)
        print(test_encoded_arr.shape)
```

```
(14870, 63)
(3718, 63)
```

In [266]:
```
from sklearn.neural_network import MLPRegressor


        hidden_configs = [50, 100, 200, 300, 500]
        activation_types = ['relu', 'logistic', 'tanh']

        rmses = {}
        for combo in list(itertools.product(activation_types,hidden_configs)):
            num_hidden = (combo[1] , )
            activation = combo[0]

            mlp = MLPRegressor(hidden_layer_sizes=num_hidden, activation=activation)
            mlp.fit(X_train, y_train)

            y_pred = mlp.predict(X_test)
            rmses[combo] = sqrt(mean_squared_error(y_test, y_pred))
```

In [269]:
```
        #plot RMSE (y-axis) vs. num hidden layers. This plot will have 3 curves, one for each
        fig, ax = plt.subplots()

        ax.plot(relu_x, relu_y, label='ReLU')
        ax.plot(logistic_x, logistic_y, label='logistic')
        ax.plot(tanh_x, tanh_y, label='tanh')

        ax.set_xlabel('Number of Hidden Nodes')
        ax.set_ylabel('Test RMSE')
        plt.legend()
        plt.show()
```



Several combinations were tested. Hidden layer possibilities included: 50, 100, 200, 300, and 1000. The activation types tested included ReLU, tanh, and logistic. From the graph above, it's clear that adding more nodes increases the expressive capacity of the network only when paired with certain activation functions. In general, the logistic activation function (sigmoid) suffers from a number of issues. Since sigmoid is not-zero centered, there can be zig-zagging gradients during gradient descent when the sign switches. In addition, at extreme values this saturates and has 0 gradient. On the plus side, around x=0 sigmoid behaves linearly and is differentiable everywhere. Tanh attempts to deal with some of these issues and is zero-centered. It shares the same benefits of sigmoid, but also saturates. ReLU has very simple derivatives and converges quickly. It can still experience zig-zagging gradients and the derivative is not defined at x=0, but we can use the subgradient. For examples with zero-activation, no learning will occur. One important point to mention is that ReLu is not bounded, but we can use weight regularization to deal with growing weights.

The lowest Test RMSE was achieved for a network with 300 hidden nodes and a ReLU activation function.

3 d) Prediction of backup sizes for each workflows

In this section, we predicted the back up sizes for each individual workflows.

3.0.1 i) Using linear regression model. Explain if the fit is improved?

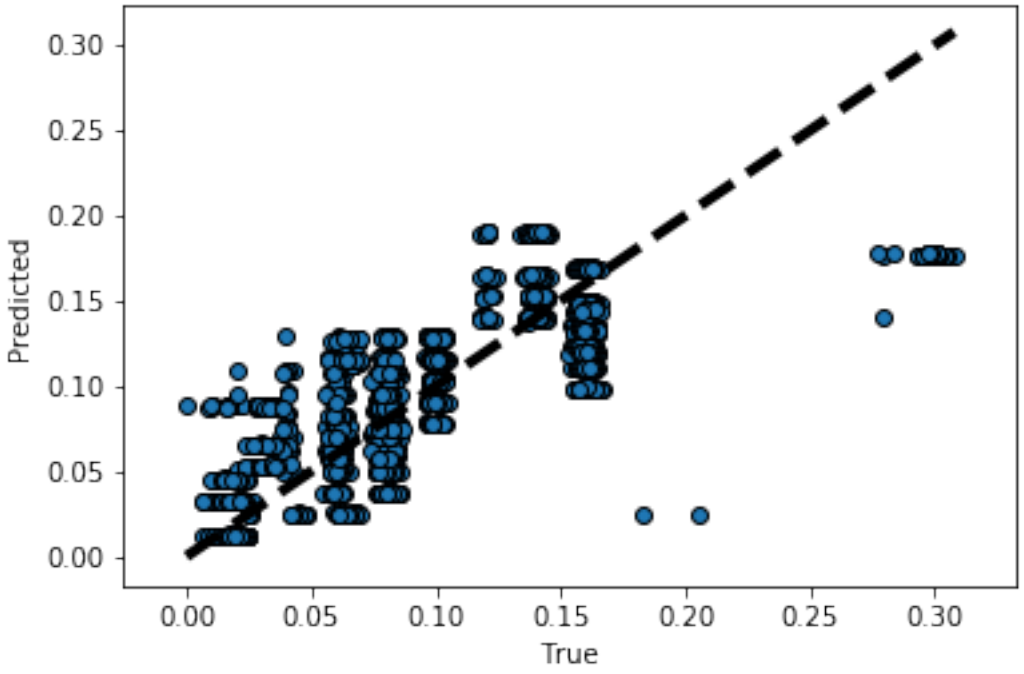
We first utilized a simple linear regression model for predicting backup sizes for each workflow. For our preprocessing step, we normalize our data by centering the mean to 0 and setting the variance to 1. Afterwards, we apply a linear regression model using 10-fold cross validation. For

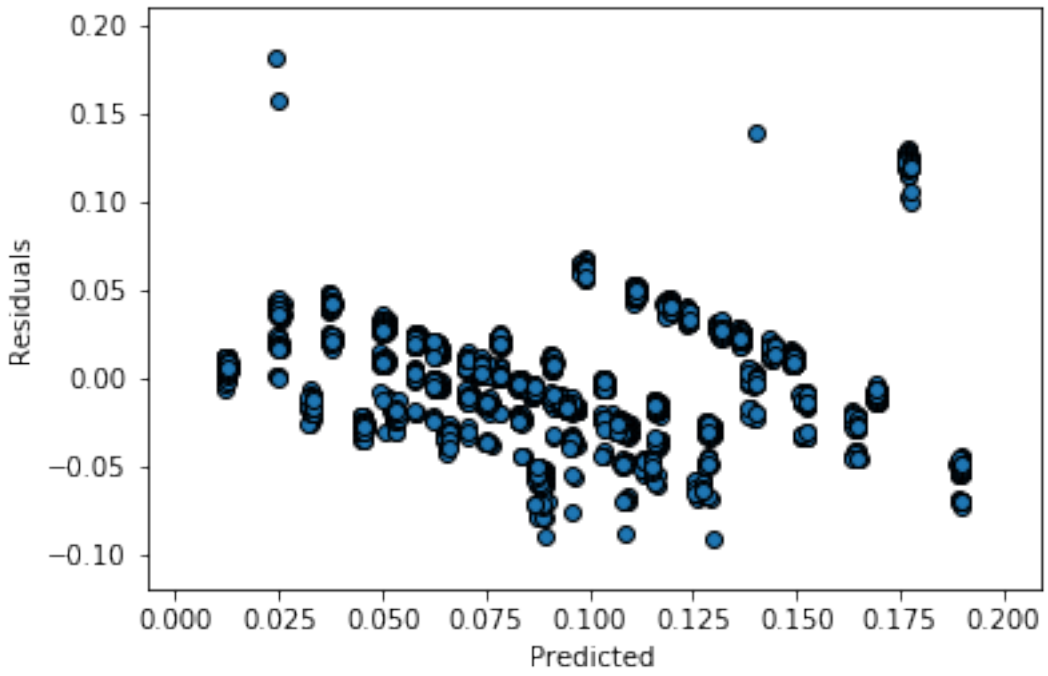
each work flow, we plot the respective fitted values vs. true values, and the residuals vs. fitted values.

```
In [270]: from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt
          from sklearn.preprocessing import StandardScaler

          X = data[:,0:5]
          y = data[:,5]

          for wid in range(5):
              print("Work Flow %i" % wid)
              wid_data = data[(data[:,3] == wid)]
              X_wid = np.delete(wid_data[:,0:5], 3, axis=1)
              y_wid = wid_data[:,5]
              scaler = StandardScaler()
              X_wid = scaler.fit_transform(X_wid, y_wid)

              kf = KFold(10)
              k = 1
              lr = linear_model.LinearRegression()
              for train_index, test_index in kf.split(X_wid): # Output 10 train test RMSE values
                  X_train, X_test = X_wid[train_index], X_wid[test_index]
                  y_train, y_test = y_wid[train_index], y_wid[test_index]
                  lr.fit(X_train, y_train)
                  pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  print("Fold %i: train RMSE = %.3f, test RMSE = %.3f" % (k, rmse_train, rmse_te
                  k += 1

              lr = linear_model.LinearRegression()
              predicted = cross_val_predict(lr, X_wid, y_wid, cv=10)

              fig, ax = plt.subplots()
              ax.scatter(y_wid, predicted, edgecolors=(0, 0, 0))
              ax.plot([y_wid.min(), y_wid.max()], [y_wid.min(), y_wid.max()], 'k--', lw=4)
              ax.set_xlabel('True')
              ax.set_ylabel('Predicted')
              plt.show()

              fig, ax = plt.subplots()
              residuals = y_wid - predicted
              ax.scatter(predicted, residuals, edgecolors=(0, 0, 0))
              ax.set_xlabel('Predicted')
```

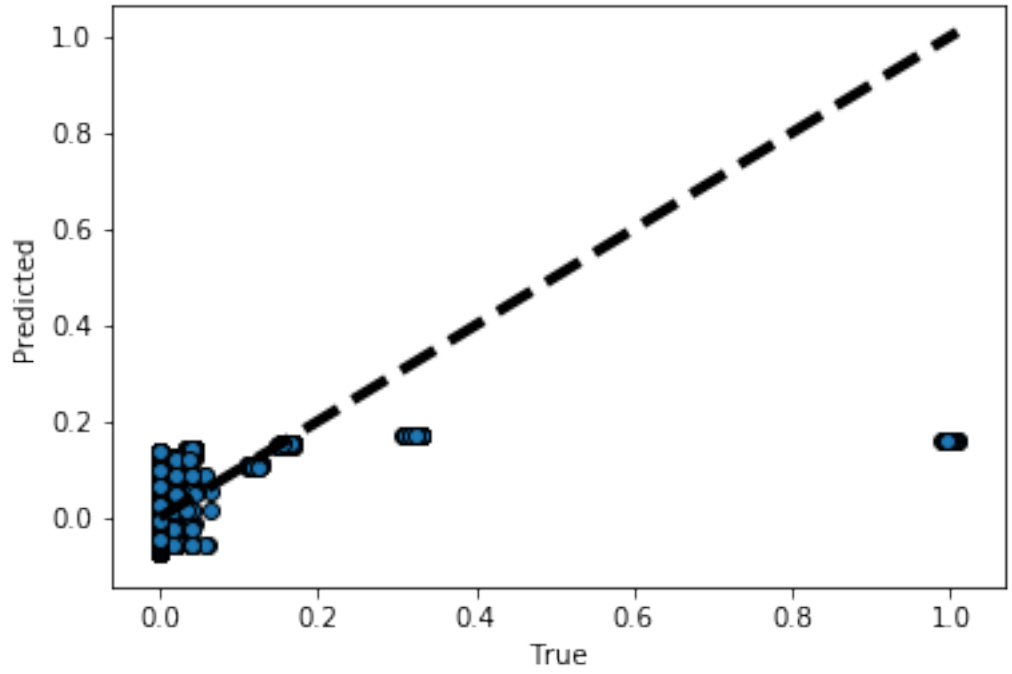
```
            ax.set_ylabel('Residuals')
            plt.show()
```

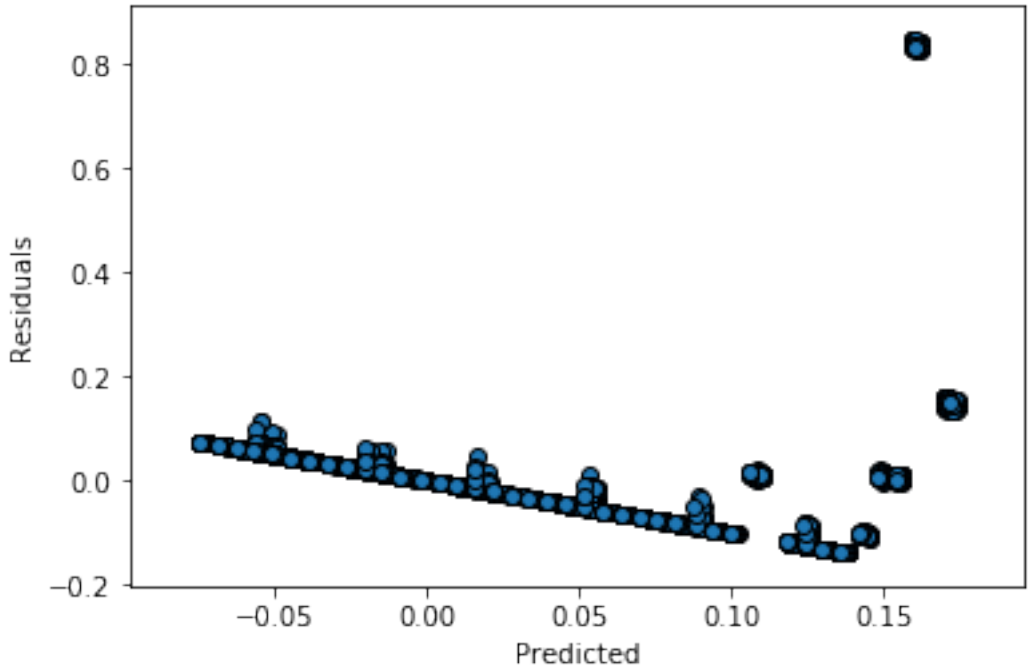
```
Work Flow 0
Fold 1: train RMSE = 0.036, test RMSE = 0.037
Fold 2: train RMSE = 0.036, test RMSE = 0.035
Fold 3: train RMSE = 0.036, test RMSE = 0.037
Fold 4: train RMSE = 0.036, test RMSE = 0.037
Fold 5: train RMSE = 0.036, test RMSE = 0.036
Fold 6: train RMSE = 0.036, test RMSE = 0.034
Fold 7: train RMSE = 0.036, test RMSE = 0.037
Fold 8: train RMSE = 0.036, test RMSE = 0.034
Fold 9: train RMSE = 0.036, test RMSE = 0.037
Fold 10: train RMSE = 0.036, test RMSE = 0.034
```





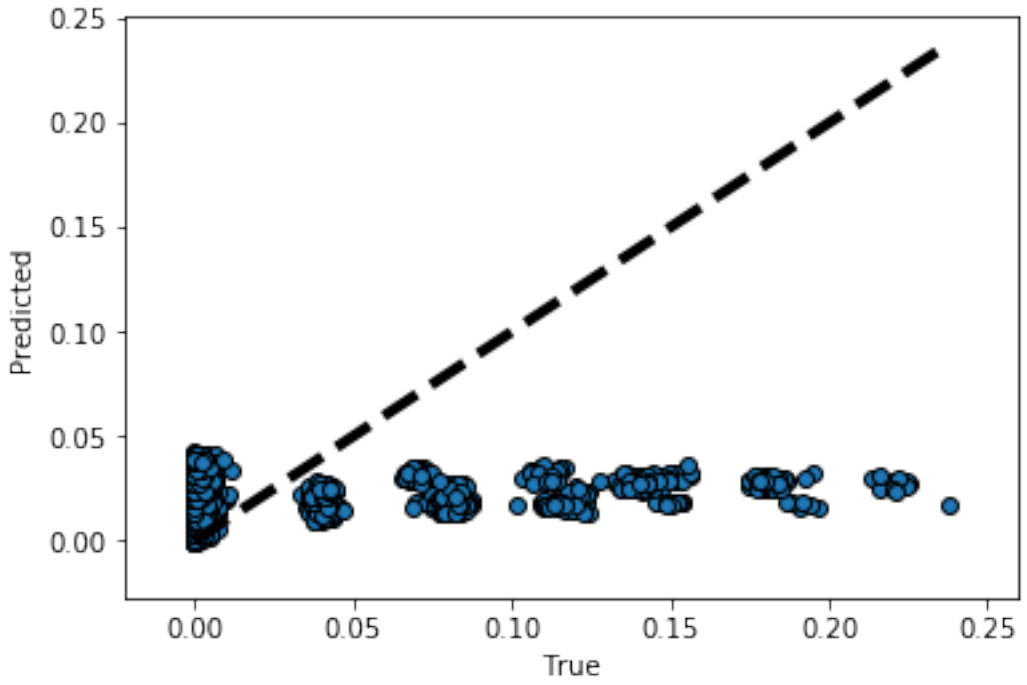

```
Work Flow 1
Fold 1: train RMSE = 0.146, test RMSE = 0.170
Fold 2: train RMSE = 0.151, test RMSE = 0.124
Fold 3: train RMSE = 0.146, test RMSE = 0.170
Fold 4: train RMSE = 0.151, test RMSE = 0.124
Fold 5: train RMSE = 0.146, test RMSE = 0.170
Fold 6: train RMSE = 0.151, test RMSE = 0.124
Fold 7: train RMSE = 0.146, test RMSE = 0.170
Fold 8: train RMSE = 0.151, test RMSE = 0.124
Fold 9: train RMSE = 0.146, test RMSE = 0.170
Fold 10: train RMSE = 0.151, test RMSE = 0.124
```

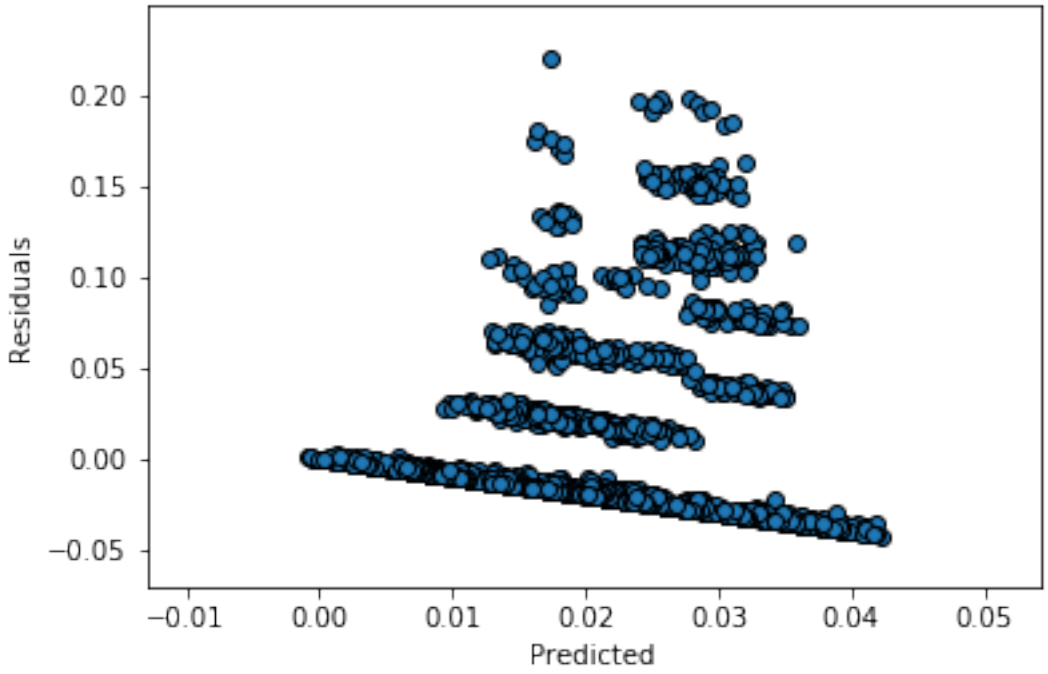




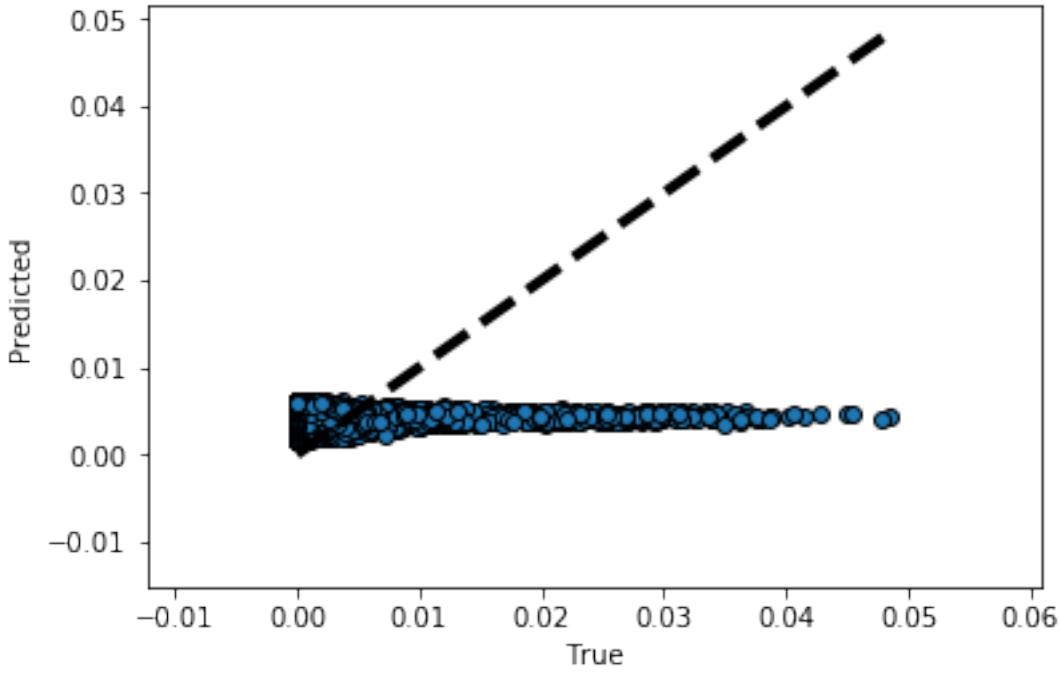

```
Work Flow 2
Fold 1: train RMSE = 0.044, test RMSE = 0.036
```

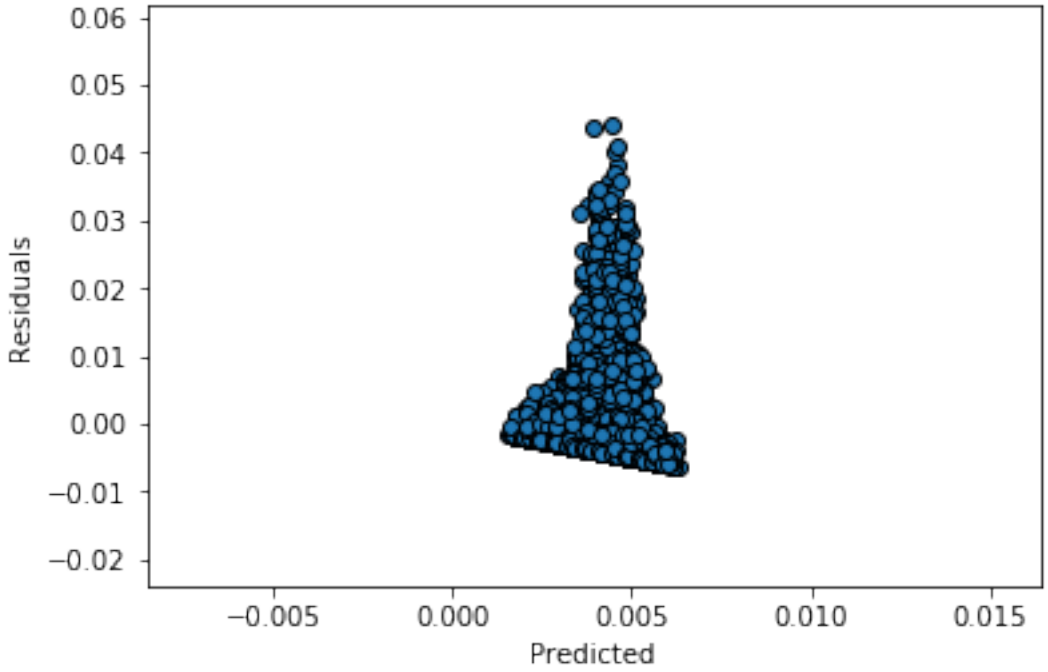
```
Fold 2: train RMSE = 0.042, test RMSE = 0.048
Fold 3: train RMSE = 0.044, test RMSE = 0.036
Fold 4: train RMSE = 0.042, test RMSE = 0.047
Fold 5: train RMSE = 0.044, test RMSE = 0.036
Fold 6: train RMSE = 0.042, test RMSE = 0.053
Fold 7: train RMSE = 0.043, test RMSE = 0.038
Fold 8: train RMSE = 0.042, test RMSE = 0.048
Fold 9: train RMSE = 0.044, test RMSE = 0.035
Fold 10: train RMSE = 0.042, test RMSE = 0.049
```





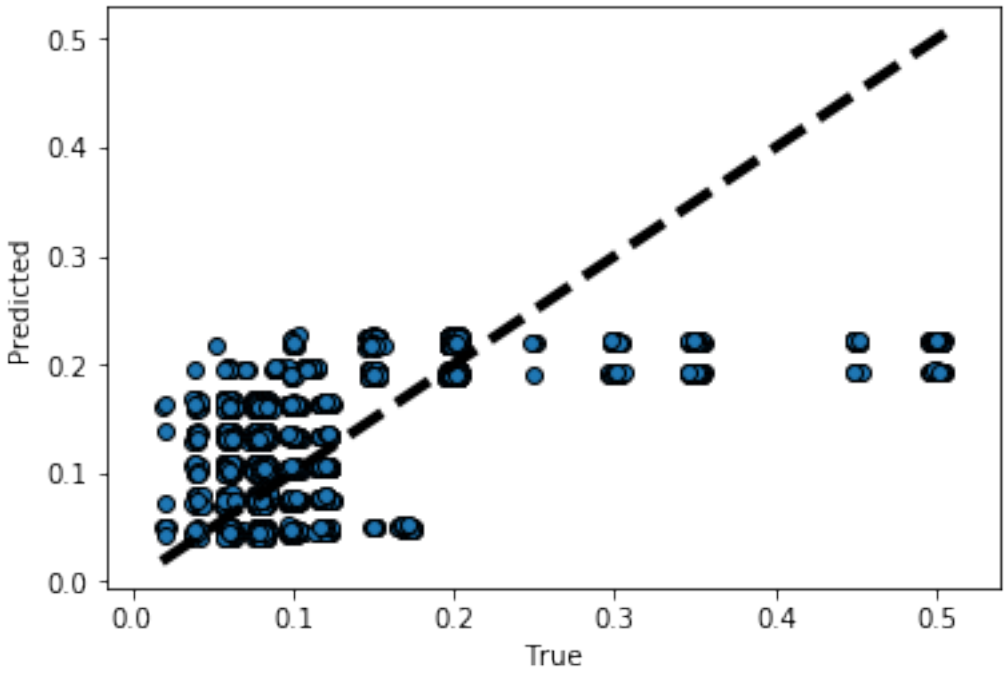

```
Work Flow 3
Fold 1: train RMSE = 0.007, test RMSE = 0.006
Fold 2: train RMSE = 0.007, test RMSE = 0.008
Fold 3: train RMSE = 0.007, test RMSE = 0.006
Fold 4: train RMSE = 0.007, test RMSE = 0.008
Fold 5: train RMSE = 0.007, test RMSE = 0.006
Fold 6: train RMSE = 0.007, test RMSE = 0.008
Fold 7: train RMSE = 0.007, test RMSE = 0.007
Fold 8: train RMSE = 0.007, test RMSE = 0.009
Fold 9: train RMSE = 0.007, test RMSE = 0.006
Fold 10: train RMSE = 0.007, test RMSE = 0.008
```

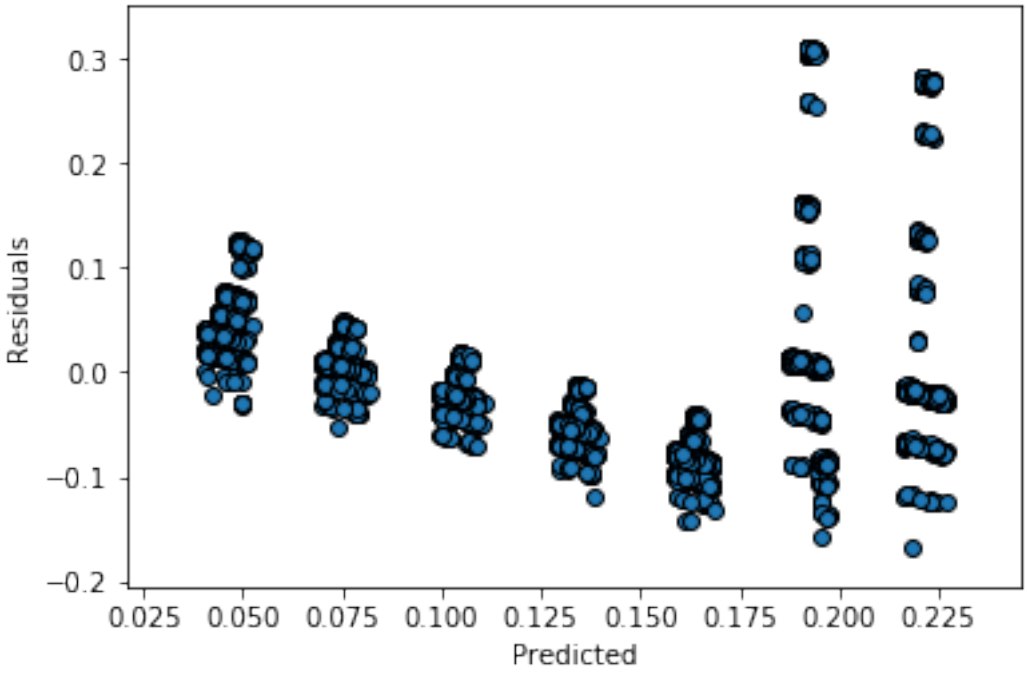





```
Work Flow 4
Fold 1: train RMSE = 0.087, test RMSE = 0.074
```

```
Fold 2: train RMSE = 0.085, test RMSE = 0.096
Fold 3: train RMSE = 0.087, test RMSE = 0.075
Fold 4: train RMSE = 0.085, test RMSE = 0.097
Fold 5: train RMSE = 0.087, test RMSE = 0.076
Fold 6: train RMSE = 0.085, test RMSE = 0.095
Fold 7: train RMSE = 0.087, test RMSE = 0.075
Fold 8: train RMSE = 0.085, test RMSE = 0.096
Fold 9: train RMSE = 0.087, test RMSE = 0.075
Fold 10: train RMSE = 0.085, test RMSE = 0.096
```





Predicting each work flow separately does increase the predictive performance for most of the workflows. For Work Flows {0,2,3,4}, the models shows consistently lower RMSEs compared to our original linear model from part (a).

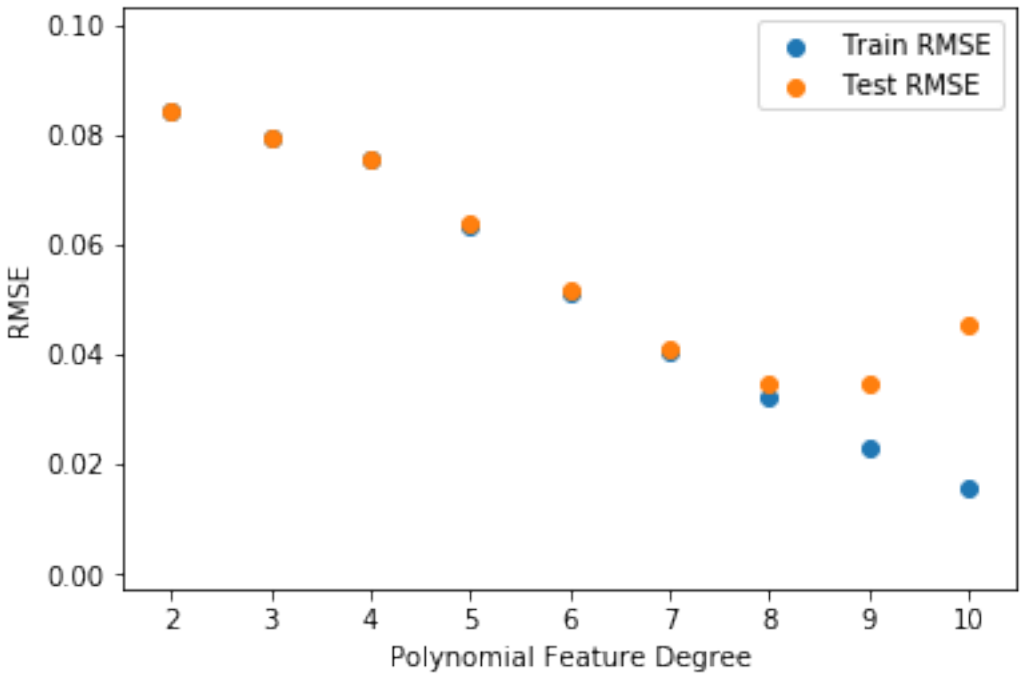
3.0.2 ii) Try fitting a more complex regression function to your data. You can try a polynomial function of your variables. Try increasing the degree of the polynomial to improve your fit. Again, use a 10 fold cross validation to evaluate your results. Plot the average train and test RMSE of the trained model against the degree of the polynomial you use. Can you find a threshold on the degree of the fitted polynomial beyond which the generalization error of your model gets worse? Can you explain how cross-validation helps controlling the complexity of your model?

To improve our fit even further, we now applied more complex regression techniques by building polynomial functions with our original features. For example, if an input sample is two dimensional and of the form [x, y], the degree-2 polynomial features are [1, x, y, x^2, y^2, xy]. Adding model complexity allows the model to be more expressive. In theory, a model of arbitrarily high polynomial degree can only achieve better test performance than a model of lower degree on the same data. This is because the more complex model is a superset of the simpler model. We can reduce the more complex model to the simpler one by zeroing out certain parameters.

The downside is overfitting. Overfitting for linear regression models generally occurs when higher order polynomials are added to the model. Adding higher order polynomials allows the learning algorithm to best fit the training data, but at the expense of complexity. Our inductive bias is such that simpler models are better, and from that we get the idea that we should keep weights minimal. The easiest way to keep weights as small as possible is by penalizing large weights through regularization.

For our analysis, we increased the degree of the polynomial and plotted the average train and test RMSEs against the degrees.

```
In [271]: from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt
          from sklearn.preprocessing import StandardScaler, PolynomialFeatures

          X = data[:,0:5]
          y = data[:,5]

          degrees = range(2, 11)
          avgTrainRMSEs, avgTestRMSEs = [], []
          for p in degrees:
              X_p = PolynomialFeatures(p).fit_transform(X)
              y_p = y

              scaler = StandardScaler()
              X_p = scaler.fit_transform(X_p, y_p)

              kf = KFold(10)
              totalTrainRMSE, totalTestRMSE = 0, 0
              for train_index, test_index in kf.split(X_p): # Output 10 train test RMSE values
                  X_train, X_test = X_p[train_index], X_p[test_index]
                  y_train, y_test = y_p[train_index], y_p[test_index]
                  lr = linear_model.LinearRegression()
                  lr.fit(X_train, y_train)
                  pred_train, pred_test = lr.predict(X_train), lr.predict(X_test)

                  rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                  totalTrainRMSE += rmse_train
                  totalTestRMSE += rmse_test
              avgTrainRMSEs.append(totalTrainRMSE / 10)
              avgTestRMSEs.append(totalTestRMSE / 10)

          fig, ax = plt.subplots()
          ax.scatter(degrees, avgTrainRMSEs, label="Train RMSE")
          ax.scatter(degrees, avgTestRMSEs, label="Test RMSE")
          ax.set_xticks(degrees)
          ax.set_xlabel('Polynomial Feature Degree')
          ax.set_ylabel('RMSE')
          plt.legend()
          plt.show()
```



The threshold to which the generalization performance suffers is when degree = 8, as clearly the train RMSE is comparatively lower than the test RMSE.

Cross validation is generally used to measure the performance of a prediction model. Setting the polynomial degree of freedom to higher numbers can result in overfitting. Thus, cross validation helps us to acheive the best fit without overfitting on the data.

3.0.3 e) Use k-nearest neighbor regression and find the best parameter.

Our aim here is to find the optimal k for kNN. Smallest values of k will cause overfitting. Large values of k will have a smoothing or averaging effect. A balance is in the middle, where the smoothing is not too excessive. Too much smoothing results in underfitting and performance will decrease.

```
In [272]: from sklearn.model_selection import KFold
          from sklearn import linear_model
          from sklearn.neighbors import KNeighborsRegressor
          from sklearn.model_selection import cross_val_predict
          from sklearn.metrics import mean_squared_error
          from math import sqrt
          from sklearn.preprocessing import StandardScaler

          X = data[:,0:5]
          y = data[:,5]

          for nn in range(2,7):
              print("%i neighbors" % nn)
```

```
            kf = KFold(10)
            k = 1
            neigh = KNeighborsRegressor(n_neighbors=nn)
            totalTrainRMSE, totalTestRMSE = 0, 0
            for train_index, test_index in kf.split(X): # Output 10 train test RMSE values
                X_train, X_test = X[train_index], X[test_index]
                y_train, y_test = y[train_index], y[test_index]
                neigh.fit(X_train, y_train)
                pred_train, pred_test = neigh.predict(X_train), neigh.predict(X_test)

                rmse_train, rmse_test = sqrt(mean_squared_error(y_train, pred_train)), sqrt(me
                totalTrainRMSE += rmse_train
                totalTestRMSE += rmse_test
            print("\tavg train rmse = %.3f, avg test rmse = %.3f" % (totalTrainRMSE / 10, tota
```

```
2 neighbors
        avg train rmse = 0.029, avg test rmse = 0.033
3 neighbors
        avg train rmse = 0.030, avg test rmse = 0.036
4 neighbors
        avg train rmse = 0.028, avg test rmse = 0.037
5 neighbors
        avg train rmse = 0.027, avg test rmse = 0.043
6 neighbors
        avg train rmse = 0.028, avg test rmse = 0.047
```

It is clear that setting number of neighbors, or k, to 2 is the most optimal parameter that yields the lowset test RMSE and best generalization performance.

4 Analysis

Overall, the best performing model was the Random Forest Regression model with max_depth = 8 as the test RMSE was 0.0143. The basic linear regression models with no modifications made did not perform relatively well, with average test RMSE values on the order of 0.1. However, several tricks that we explored and applied to other models made them more effective. For example, predicting each workflow separately was effective at reducing test RMSE for the linear regression model. Ridge regression (L2 regularization) was effective at reducing average test RMSE from the order of 0.1 to ~0.08. We observed that L2 regularization performed better than L1 regularization, which did not improve average test RMSE by a significant amount given the combinations of one-hot encodings. In fact, for most cases it made average test RMSE worse. This is probably due to the fact that L1 regularization generates sparsity. We discovered that k-NN was an effective regressor, boasting an average test RMSE of 0.033 for k=2. The basic neural network model was effective at regression as well, but was heavily dependent on hyperparamter tuning. Given the fact that all of the features were one-hot encoded, the input to the MLP had a lot of sparsity. It was able to handle this better than the linear regression model with L1 regularization, boasting an average test RMSE of ~0.088, which is better than that of the Lasso regressor. Its best performance was achieved for a ReLU activation function with 300 nodes in the hidden layer.