Blake Ward
MLND Capstone Project

I. Definition

Project Overview

This project is derived directly from my day-to-day job. As a data engineer working on building out a Hadoop data lake, I am responsible for designing jobs that load data from various sources. These jobs run on a specific time schedules and need to be monitored for potential failures.

To date, we have not had a great way to know if a job was having problems. Specifically, we want to know if a job is "long running", indicating it may miss its window for completion. Recently we implemented a very simple algorithm to predict the run time for a job. It is based on average run times and is not very accurate. As a result, we need to build a better model for predicting run time.

Problem Statement

In this project the goal is to improve upon the predicted run time estimate for jobs that are used to load data into a data lake at my company. Our current predictions are not accurate and do not give us the ability to monitor job performance. We have a lengthy history of completed job runs and I believe this problem can be solved by building a better predictive model that will estimate run time at the start of a job.

At the end of the project, I intend to have a model that will output a predicted run time in seconds. Because the output space is continuous, this project lends itself to a regression based model.

Metrics

As a regression problem, two good evaluation metrics for goodness of fit are the r-squared score and the standard error of the regression (equations in Fig. 1). These both can be applied to the benchmark model and our new model for a comparison of performance. The r-squared score measures the fraction by which the variance of the errors is less than the variance of the dependent variable. In other words, it gives an approximation of the amount of variance in the dependent variable that the model explains. The standard error of the

Figure 1

$$S_{y.x} = \sqrt{\frac{\Sigma(y - y_{est})^2}{n}} \qquad R^2 = \frac{SSR}{SST} = \frac{\sum(\hat{y}_i - \bar{y})^2}{\sum(y_i - \bar{y})^2}$$

regression is a complementary metric for evaluating a regression model that explains the typical distance (in common units) that data points lie from the prediction.

II. Analysis

Data Exploration

The data I'm using is drawn directly for the operational database that powers our ingestion jobs and logs the job metrics. It is an output from a few of our job stats tables combined into one dataset (figure 2).

Figure 2

	min_run_time	std_run_time	avg_table_run_time	max_table_run_time	\
count	22488.000000	22488.000000	22488.000000	22488.000000	
unique	NaN	NaN	NaN	NaN	
top	NaN	NaN	NaN	NaN	
freq	NaN	NaN	NaN	NaN	
mean	270.697305	1521.408446	993.174850	4982.548502	
std	1408.747497	3191.832579	6504.959808	18919.318788	
min	1.000000	0.000000	1.000000	1.000000	
25%	7.000000	133.764025	35.484569	95.424390	
50%	31.000000	608.044673	57.313450	214.975000	
75%	166.000000	1561.765284	96.999741	1214.025316	
max	118904.000000	51423.994420	113768.375190	286177.224138	

	min_table_run_time
count	22488.000000
unique	NaN
top	NaN
freq	NaN
mean	245.750570
std	2765.321487
min	1.000000
25%	23.888489
50%	25.666667
75%	35.211538
max	65210.857143

	source	dbType	source_type	jobType	run_time	\
count	22488	22488	22488	22488	22488.000000	
unique	112	13	4	3	NaN	
top	SAP PS NA	sap	db	raw	NaN	
freq	6878	9168	20796	10140	NaN	
mean	NaN	NaN	NaN	NaN	2122.030149	
std	NaN	NaN	NaN	NaN	6500.156860	
min	NaN	NaN	NaN	NaN	1.000000	
25%	NaN	NaN	NaN	NaN	151.000000	
50%	NaN	NaN	NaN	NaN	431.000000	
75%	NaN	NaN	NaN	NaN	1713.000000	
max	NaN	NaN	NaN	NaN	196536.000000	

	total_objects_count	avg_object_run_time	avg_run_time	\
count	22488.000000	2.248800e+04	22488.000000	
unique	NaN	NaN	NaN	
top	NaN	NaN	NaN	
freq	NaN	NaN	NaN	
mean	262.018143	3.897195e+02	2081.737075	
std	2023.500230	9.510893e+03	5212.074403	
min	1.000000	0.000000e+00	1.000000	
25%	10.000000	2.860000e+01	224.566079	
50%	23.000000	4.000000e+01	507.794872	
75%	81.000000	9.291990e+01	2161.922131	
max	196470.000000	1.344507e+06	118904.000000	

```
cols = ('source', 'dbType', 'source_type', 'jobType', 'run_time', 'total_objects_count', 'avg_object_run_time',
        'avg_run_time', 'max_run_time', 'min_run_time', 'std_run_time', 'avg_table_run_time',
        'max_table_run_time', 'min_table_run_time')
```

The fields listed through time are all categorical, the rest are all continuous. The dependent variable for this project is "run_time". The data contains 22488 data points.

Data Dictionary:
source: Business name for the source database
dbType: Database technology type (oracle, sap, mssql, etc.)
source_type: database, flat file, sftp, web api
jobType: raw - ingestion of raw data from a source; hive – conversion of raw data into usable data tables; cdc – conversion of delta raw data into usable data tables

run_time: a job's run time in seconds
total_objects_count: the number of objects included in the job's current run
avg_object_run_time: the actual average run time for objects in the current run
avg_run_time: the average run time for the job over all its runs
min_run_time: the minimum run time of the job for all its runs
std_run_time: the standard deviation of the job's run time over all its runs
avg_table_run_time: the average run time of for a single table over all of the runs
max_table_run_time: the maximum run time for any table over all the runs
min_table_run_time: the minimum run time for any table over all the runs

Data Sample:

	source	dbType	source_type	jobType	run_time	total_objects_count	avg_object_run_time	avg_run_time	max_run_time	min_run_time	std_run_time	avg_table_run_time	max_table_run_time	min_table_run_time
0	FX Rates	oracle	db	raw	2581	169	146.0828	2012.735593	4843.0	443.0	622.421159	71.593881	879.426316	43.569378
1	FX Rates	oracle	db	hive	42	1	38.0000	68.511945	632.0	1.0	56.772380	61.028540	72.927350	49.129730
2	FX Rates	oracle	db	raw	2448	169	127.8994	2012.735593	4843.0	443.0	622.421159	71.593881	879.426316	43.569378
3	EBS North America	oracle	db	hive	927	38	86.0263	1165.729730	4235.0	564.0	615.313660	17912.706182	50080.357143	32.625000
4	EBS North America	oracle	db	raw	11235	38	548.1842	9709.923077	13453.0	1490.0	2149.466133	502.728079	6587.309091	

A preliminary look at the raw data show that many of the continuous variables have a skewed distribution and wide ranges. This leads me to believe there are some outliers in the data. This isn't unexpected, as I didn't try and filter out records when creating this dataset. As a result, I know it includes records that definitely have issues that would lead to inappropriate outliers. These could be the result of server crashes or other technological glitches, which don't represent the "true" run time of a job. While it will be hard to determine exactly which records these are, I plan to be somewhat conservative but still remove outliers. I'm going to do this in order to be on the safe side of getting "real" runs and not early runs with few tables or runs that had to be paused, etc.

Also, job type is an important variable in this data set, as the job types really represent three different distributions of data. The three types of jobs are "raw", "cdc", and "hive". Looking at some of the descriptive statistics grouped by job type (Fig. 3) shows this fact.

Figure 3

cdc

	run_time	total_objects_count	avg_object_run_time	avg_run_time	\
count	2568.000000	2568.000000	2568.000000	2568.000000	
mean	2743.000000	118.828660	489.974629	2461.211469	
std	4135.150711	3895.722033	1710.986177	1508.110894	
min	5.000000	1.000000	26.000000	139.126126	
25%	378.250000	13.000000	142.810750	2161.922131	
50%	1692.500000	16.000000	296.159700	2161.922131	
75%	2571.750000	46.000000	557.748775	3226.277228	
max	75935.000000	196470.000000	81671.308200	6458.510490	

hive

	run_time	total_objects_count	avg_object_run_time	avg_run_time	\
count	9780.000000	9780.000000	9.780000e+03	9780.000000	
mean	485.403783	44.819530	2.281442e+02	491.066919	
std	1176.094665	249.419084	1.365814e+04	815.535829	
min	2.000000	1.000000	7.000000e+00	20.166667	
25%	120.000000	4.000000	2.818937e+01	189.057504	
50%	229.000000	23.000000	3.500000e+01	312.307407	
75%	470.000000	49.000000	4.875000e+01	601.529820	
max	25002.000000	22806.000000	1.344507e+06	20634.000000	

raw

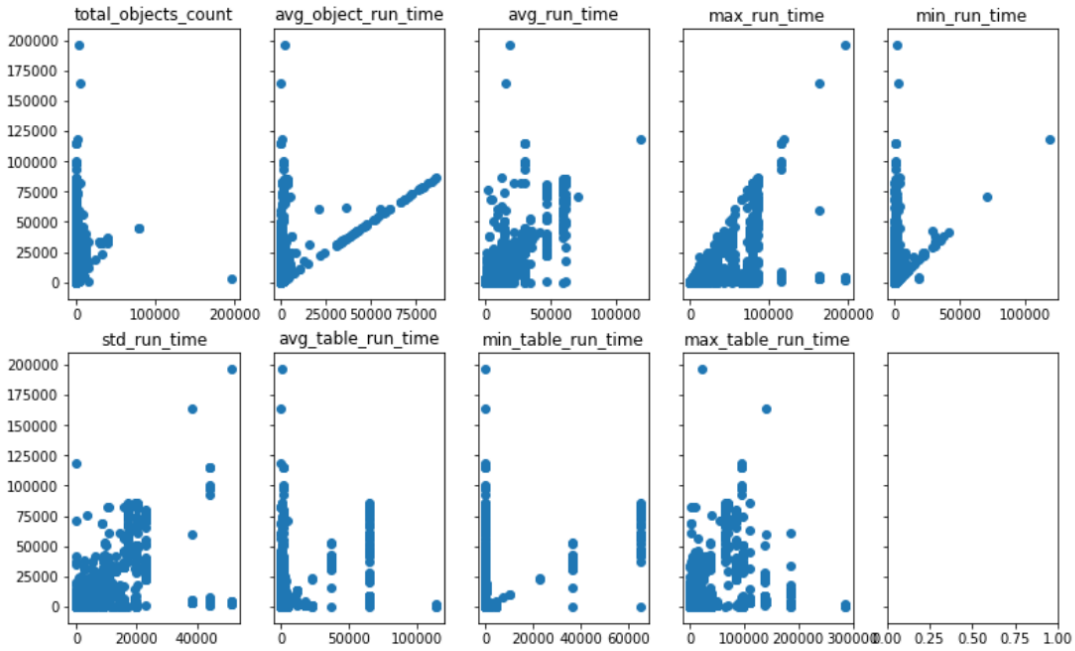
	run_time	total_objects_count	avg_object_run_time	avg_run_time	\
count	10140.000000	10140.000000	10140.000000	10140.000000	
mean	3543.288067	507.768935	520.168462	3519.830159	
std	9129.332410	2251.146058	4462.613034	7381.553393	
min	1.000000	1.000000	0.000000	1.000000	
25%	284.000000	19.000000	27.962275	436.814867	
50%	962.500000	32.000000	38.816300	1429.245665	
75%	2910.750000	329.000000	99.798575	3224.836066	
max	196536.000000	80084.000000	86308.000000	118904.000000	

This made me attempt to break the data into three subsets to train separate models, but this didn't increase performance enough to compensate for the overhead. The wide spread of some of these features also required me to standardize them when attempting certain algorithms like the multi-layer perceptron.

Exploratory Visualization

As the output I need for this project is a continuous variable, my first thought was toward a simple linear model. In order to get a sense of how this might work, I created a visual (Fig 4) to plot all of the numeric features (x-axis) vs the target variable (y-axis). All units are in seconds.

Figure 4

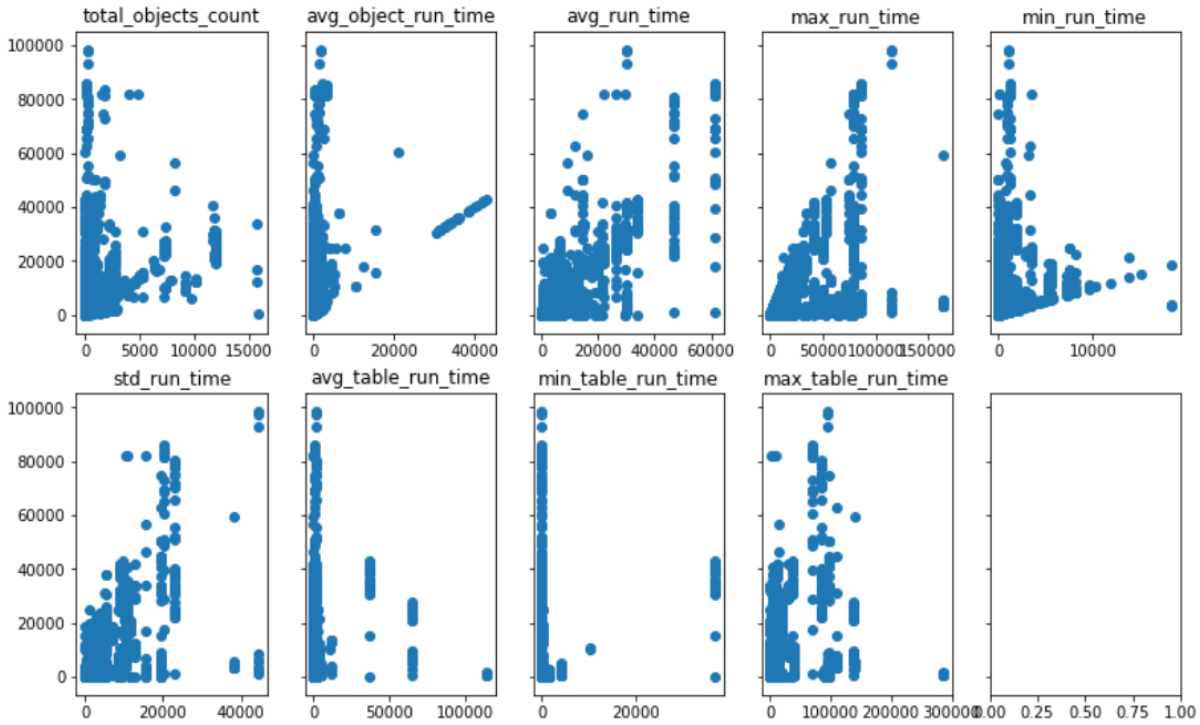


Although there are some semblances of a linear relationship between some of these features and the target, there is nothing that stands out as a clear linear or even polynomial relationship. There are a number of relationships that look like the letter "V". Looking at this data led me to believe that an algorithm like a decision tree would probably be more successful.

These visualizations also clearly show some of the outliers. The features avg_ run_time and avg_table_run_time are good examples. In both of these there are a couple of data points well outside the rest. Looking at this I decided to remove outliers based on a conservative 10+/- standard deviation rule for the distributions of each feature variable broken down by job type. For the target variable I also chose to remove values outside of 10 standard deviations. I chose these because I felt I had enough data to accurately capture the statistical nature of "true" runs even if I removed some true runs at the extreme ends of the spectrums. I was more interested in getting rid of the garbage data that would really cause havoc in a model.

After cleaning outliers, the revised scatter plots (Fig. 5) show a slightly different distribution by feature.

Figure 5



Algorithms and Techniques

The requirement for this project is a continuous output variable, so any algorithm I use will need to meet that need. Essentially I will be looking at algorithms that have regression options. My plan to identify the best algorithm to use for this task is to explore a variety of regression techniques with the default parameters and see how they perform. At that point, I will narrow my selection down to one or two and do more optimization to come up with a final model.

Looking at the data above, my instinct is that a linear regression model will not be effective due to the complicated relationship between the features and the target. I believe that a decision tree or an ensemble method will probably work better.

All of the algorithms I tested are listed below with an intuitive explanation provided. A detail explanation is provided of the random forest algorithm, which provided the final result.

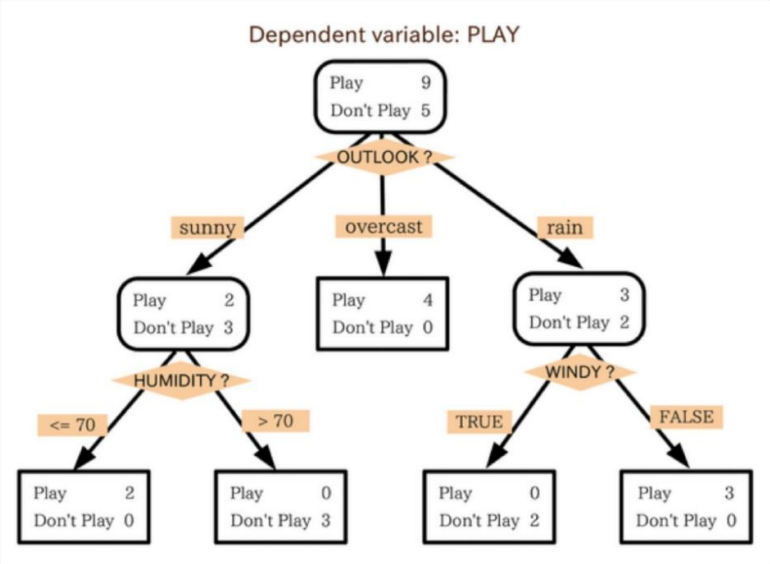
<u>Regression Algorithms</u>:

- Linear Model with Ordinary Least Square
  This is a standard linear model that fits a line to the data by minimizing the sum of the squared errors between the dependent variables and the predictions. This line is often described as the line of best fit and is representation of the relationship between the independent and dependent variables.

- Decision Tree

  Decision trees are a non-parametric method of solving regression (and classification) problems that use nodes and leafs to segment data into smaller and smaller units until the units that remain at the "bottom" of the tree are homogenous enough to be assigned to an output variable. The algorithm searches through the data to find the best split at each leaf by minimizing a split function, generally mean squared error or mean absolute error for regression problems.

  A simple example of a decision tree (from blog.cigizennet.com):



- Random Forest

  Random Forests are an ensemble method using decision trees as the base estimators. Ensemble methods general outperform their base estimators for something I like to think of as the "power of the crowd". Instead of a single model being trained, ensemble methods constitute a number of models combined together to produce more robust estimates.

  Specifically, a random forest is a collection of "weak learners". These are predictive models, and this case decision trees (see explanation above), whose threshold of performance is only that they perform better than a random estimate. In constructing each weak learner, a random forest creates a decision tree with a subset of the total training data. Then at each node, the algorithm only picks from a random subset of available features. This adds variance into each of the weak learners, making for a more robust final model because each weak learner will model the data in a slightly different manner.

  Random forests produce a final result by combining the output of each of the weak learners. This can be done in a variety of manners, but for classification tasks it can be a voting mechanism where each learner has a vote. In a regression problem like this, it can be done using an averaging algorithm.

  A random forest is a nice algorithm because it is fast and can also handle unbalanced data. This latter fact makes it a nice choice for my particular dataset. One downside is the tendency to over it noisy data, and that is something I will be cautious about when I run this in production.

- Gradient Boosted Tree
- Ada Boosted Tree

  These two algorithms are each also ensemble methods for regression using decision trees as the base estimator. As ensembles, they both produce a set of weak learner decision trees. In adaptive boosting, the algorithm starts with stump trees with a single split and then successive trees are added that focus on the more and more difficult pieces of data. In this way the ensemble of tree gets built out with an emphasis on the difficult data sets. In gradient boosting, successive trees are added to the ensemble that minimize a given loss function through gradient descent.

- Support Vector Regression

  SVMs use support vectors - the samples from the training set that are closest to the boundary line - to determine a hyperplane that will separate the data into two categories. At first sight, the data doesn't have to be linearly separable, because SVMs can also use a kernel trick to move the data to a higher dimension where it may be separable. SVMs work on the idea of optimizing the maximal distance from the closest support vectors to the boundary line.

- Multi Layer Perceptron

  MLP is a form of neural network that uses layers (at least three) of perceptron nodes to output a continuous variable. Each perceptron takes an input, applies a weight and a bias, and then uses an activation function to output the results either as a final result or to the next perceptron.

Each of these algorithms can output a continuous dependent variable. Also, because many of the feature have skewed distributions, I am learning toward methods that are not parametric and won't be sensitive to the population distributions.

Benchmark

We are currently implementing a simple algorithm to predict run time. This can serve as the benchmark model to compare the performance of our new model. The current model takes two inputs and outputs a predicted run time. The first input is the average table run time – based on all previous runs – for all the tables to be loaded. The second input is the number of tables to be loaded for the current run. These two inputs are multiplied to get the job's predicted run time in seconds.

The benchmark algorithm does not perform well, from anecdotal experience and also looking at our evaluation metrics. For some jobs it is dead on, but for the jobs where there isn't a linear relationship between those two variables, it can be widely drastic. The R2 score below (Fig. 6) show in general the model does worse than picking a constant.

Figure 6

```
from sklearn.metrics import r2_score

r2_simple = r2_score(clean_data.run_time, simple_target)
se = st_err(clean_data.run_time, simple_target)

print("Benchmark R2-Score: " + str(r2_simple))
print("Benchamrk Standard Error:" + str(se))

plt.scatter(simple_target, clean_data[target])
```

```
Benchmark R2-Score: -26.6180088569
Benchamrk Standard Error:27859.3835965
```

III. Methodology

Data Preprocessing

As I was looking to narrow down my model choices, I did only the basic preprocessing at first. In fact there were only three steps (Fig. 7):

1. Split feature data and target data
2. One hot encode categorical variables in feature data
3. Split data into test and training sets

Figure 7

PRE-PROCESS DATA

In [17]:
```
#Separating feature and target datasets
feature_data = clean_data[features]
target_data = clean_data[target]

#one-hot-encoding categorical features
feature_data = pd.get_dummies(feature_data, columns=cat_features)
```

PREPARE TRAINING DATA

In [18]:
```
from sklearn.model_selection import train_test_split

# create train and test data with 80% split
X_train, X_test, y_train, y_test = train_test_split(feature_data, target_data, train_size = .8, random_state = 42)

print(X_train.shape)
```

```
(17391, 138)
```

Implementation

In this implementation I primarily used modules from sklearn (version 0.18.1). I started implementing by applying my test and training data to each of the regression algorithms listed above using the default parameters (Fig. 8). This was fairly straightforward, except for the multi-layer perceptron, where I standardized the features to values between 0 and 1. At first I didn't standardize the values and the MLP was performing very badly. I didn't know why until I did a little more research and found that

MLPs are sensitive to unscaled data. I also tried scaling the data for the SVM, but it didn't change the results much.

Figure 8

LINEAR REGRESSION

```
from sklearn.linear_model import LinearRegression

# Most basic linear regression
lm = LinearRegression()
lm.fit(X_train, y_train)
y_pred = lm.predict(X_test)
r2_lm = r2_score(y_pred, y_test)
print("Basic LR R2-Score: " + str(r2_lm))
```

```
Basic LR R2-Score: 0.565938910277
```

DECISION TREE

Going to start with a basic decision tree without any parameter tuning or going to an ensemble method for comparsion

```
from sklearn.tree import DecisionTreeRegressor, export_graphviz

# Decision Tree Regressor
dtr = DecisionTreeRegressor(random_state=42)
dtr.fit(X_train, y_train['run_time'])
```

```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_split=1e-07,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, presort=False, random_state=42,
           splitter='best')
```

```
y_pred = dtr.predict(X_test)

r2 = r2_score(y_test['run_time'], y_pred)
print("Basic R2-Score from Decision Tree: " + str(r2))
```

After running all the regression algorithms listed above, I found that the random forest produced the best result. I also found that only a subset of the features were having an impact on my results. At that point I limited features and focused on optimizing that algorithm using the sklearn grid search module (Fig. 9). I tried a variety of options (in small groups to limit training time). This ultimately produced only one variable that needed to be changed from the default, which was oob_score.

Figure 9

```
from sklearn.grid_search import GridSearchCV

param_grid = {
                #'n_estimators': [10] #[50, 70, 90], #[25, 35, 50], #[20, 25, 35, 50], #[13, 14, 15], #[10, 15, 20] #70 best
                #'max_depth': [13, 15, 19], #[15, 16, 17], #[15, 17, 19], None best
                #'max_features': ['auto', 'sqrt'], #['auto', 'sqrt'] auto best
                #'min_samples_leaf': [1, 10], #default of 1 was best
                #'criterion': ['mse', 'mae'],
                'oob_score': ['True'] #['True', 'False'] True best
             }

rfGS = GridSearchCV(RandomForestRegressor(random_state=42), param_grid, cv=8)
rfGS.fit(X_train, y_train)
y_pred = rfGS.predict(X_test)

rfGS_r2 = r2_score(y_test, y_pred)
```

The details of how I got to this final model and a full explanation of the refinement steps are given in the next section.

Refinement

To choose a final model for this project, I started by running the basic clean data through a number of different regression based models. My goal in this was to do the minimal amount of pre-processing and feature engineering to see where I should spend my time focusing more effort. I used the R2 score to evaluate a number of different models using their default sklearn values. The results are listed below. The only

algorithm where I did any extra tweaks to the data was for the multi-layer perceptron, where I scaled the data as that algorithm is very sensitive to unscaled data.

R2 Scores
Linear Regression (OLS): .56
Decision Tree: .865
Random Forest: .916
Gradient Boosted Tree: .877
Ada Boosted Tree: .846
Support Vector Regression: -.07
Multi-Layer Perceptron: .738

As I suspected, the random forest ensemble method had the best output. That is the algorithm that I chose to optimize, but I didn't jump directly to that step.

Before I moved on to optimization, I did take a look at the feature importance output from my basic decision tree. The data I used for the test above had 138 feature after the one-hot encoding. I had a sense that not all of these might have been important in the model and wanting to keep things as simple as possible for when this needs to be put into production in a working cluster, I was hoping to be able to reduce the feature space somewhat and possibly remove categorical variables altogether if it made sense.

The feature importance from the decision tree produced the following output (only features with an importance score above .0001 shown):

avg_run_time	0.7242
avg_object_run_time	0.1246
total_objects_count	0.0855
std_run_time	0.0301
max_run_time	0.0168
source_JCH	0.0066
min_run_time	0.0029
min_table_run_time	0.0024
source_Maximo	0.0017
dbType_sqlserver	0.0016
avg_table_run_time	0.0011
max_table_run_time	0.0009
source_JCH	0.0006
source_Service	0.0003
source_Avaya	0.0002
dbType_Web	0.0001
source_Macpac	0.0001
source_MfgPro	0.0001

Fortunately there were only five features contributing most of the predictive power to the model. As using this in our actual system will be easier if there are less features I need to feed the model, I decided to slim down my feature set to just those five features.

The last thing I did before tuning models was attempted to build individual models for each of the job types. I was wondering if this would significantly increase the predictive power of the models (although I was a little skeptical as the job type feature didn't seem to be adding a lot of importance). The results are shown below.

```
Random Forests R2 by job type: raw 0.891490590678
Random Forests R2 by job type: hive 0.952891276412
Random Forests R2 by job type: cdc 0.739457684564
```

Although it did increase predictive power for the hive job type, the increase to me wasn't worth the overhead that would be required to maintain three models in production and filter data to the correct model at run time.

After choosing to ultimately use a random forest algorithm, I chose to optimize the model using the grid search module in sklearn with cross fold validation. One of the nice things about the random forest regressor is the fact it doesn't have a lot of parameters that need to be tuned. I focused on the following parameters (brief explanation provided):

- n_estimators
  The number of trees to use in the ensemble. The default value is 10.
- max_depth
  The maximum depth to be used for each tree. Deeper trees are more susceptible to overfitting. The default value is None, which splits a tree until all the leafs are pure or are below the minimum number of samples need to split.
- max_features
  The number of features to look at when determining each split. The default is to look at all the features available. Lowering the number of features creates a level of randomness in each tree the helps build the robustness of the model.
- min_samples_leaf
  The minimum number of samples to be at a leaf node. The default is 1. This is a stopping criteria that helps overfitting.
- criterion
  The criterion which determines the best splits. The minimization of this criterion determines which features and values represent the best splits at each level of the tree. The default is mean squared error.
- oob_score

This determines whether to use out-of-bag samples (bootstrapping) to estimate the R2 on unseen data. This is another way to prevent overfitting and to validate model by using a random subset of data in the training and testing phases.

I tried a lot of iterations with different values of these parameters. The resulting R2 scores all varied between .85 and .91. At the end of this process my final model had a slightly improved R2 score over using the base default values. I produced that result using cross fold validation, which also means the final product is slightly more robust to fluctuations in input data. At the end there weren't a lot of parameters that needed to be changed from the default. I only switched the oob_score parameters from False to True and the max_features parameters to sqrt – meaning the algorithm only could pick from a few randomly selected features at each node (Fig. 10).

Figure 10

```
param_grid = {
                #'n_estimators': [10] #[50, 70, 90], #[25, 35, 50], #[20, 25, 35, 50], #[13, 14, 15], #[10, 15, 20] #70 best
                #'max_depth': [13, 15, 19], #[15, 16, 17], #[15, 17, 19], None best
                #'max_features': ['auto', 'sqrt'], #['auto', 'sqrt'] auto best
                #'min_samples_leaf': [1, 10], #default of 1 was best
                #'criterion': ['mse', 'mae'],
                'oob_score': ['True'] #['True', 'False'] True best
            }
```

IV. Results

Model Evaluation and Validation

After using Grid Search and a cross fold validation, my final model produced the evaluation metrics below.

```
Optimized Forest R2: 0.916415097106
Optimized Forest Standard Error: 1374.66314683
```

I am happy with the final model's results and feel that it will be robust enough for our needs. The reason I feel confident in the robustness of the result is the use of cross fold validation. Cross fold validation essentially breaks down my training data into k-number of folds (in this project I used 8 folds). A model is then trained on a rotating group of 7 of those folds with one left out as a cross-validation "test" set. The results of this showed that the model performs reasonable well across all the folds, meaning it seems to be resilient to slight fluctuations in data (fig. 11).

Figure 11

split0_test_score	0.866304
split0_train_score	0.990088
split1_test_score	0.95515
split1_train_score	0.987765
split2_test_score	0.924443
split2_train_score	0.989368
split3_test_score	0.948673
split3_train_score	0.987865
split4_test_score	0.883989
split4_train_score	0.989869
split5_test_score	0.952162
split5_train_score	0.989633
split6_test_score	0.918237
split6_train_score	0.987528
split7_test_score	0.900221
split7_train_score	0.985487

Justification

The final model for this project performed much better than the benchmark model and will be a major improvement for our operational team. The comparison results are listed below.

```
Benchmark R2-Score: -26.6180088569
Benchmark Standard Error: 27859.3835965

Optimized Forest R2: 0.916415097106
Optimized Forest Standard Error: 1374.66314683
```

The standard error improvement is an area where I am the most impressed. Although some of our predictions will obviously be off, an average error of 22 minutes is well with a reasonable margin if we take a conservative approach to determining when a job has run long.

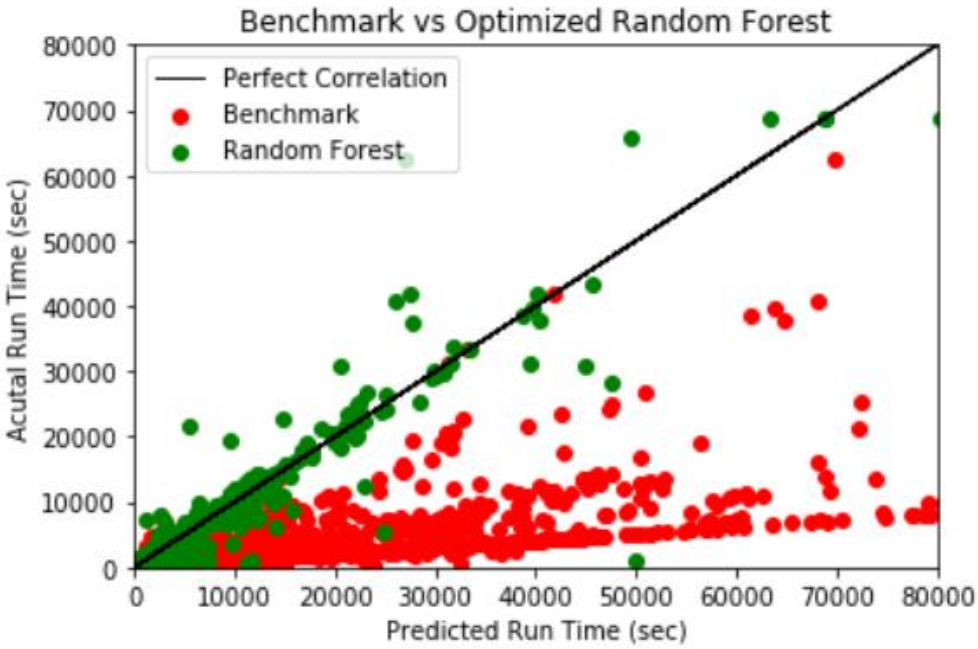
V. Conclusion

Free-Form Visualization

The visualization I have chosen (fig. 12) shows the comparison of my benchmark results versus my optimized model. The actual run times are on the y-axis and the predicted run times are on the x-axis.

Looking at the graphic it is clear that the correlation between actual and predicted (and hence a smaller R2) is better for the optimized model. I noticed one outlier that falls around the 50,000 predicted run time seemed very off from the rest. Looking at that data element in detail it looks like a job that was an early test with only one table where the rest of times the job ran it had hundreds of tables. As a test, I ran an R2 score removing that from the test data and it went up to 93%.

Figure 12



Reflection

This project was an excellent chance for me to take a real-world scenario through the entire machine learning process from early data exploration to optimizing a final model. It wasn't totally smooth, but at every step I was happy with the result.

In the data exploration, when I first looked at the scatter plots of the various features to the target, I wasn't sure that any model was going to work because there seemed to be no relationship. The data seemed totally random with a lot of outliers. I was a little hesitant to remove outliers, but looking closely at the data samples I removed made me much more comfortable.

In selecting a model, I was happy to find that the non-parametric models like a decision tree worked well. The final model was a random forest, and I had a hunch going in that an ensemble model would probably be the best bet for me. I enjoyed trying my data on all the different regression algorithms, both as practice and to be able to compare the different implementations and results.

The most difficult aspect – although at the same time a nice finding – was that optimizing the parameters in grid search didn't improve my model much. I spent quite a bit of time trying new parameters and different combinations and in the end the defaults didn't need much tweaking.

Overall I am still a little worried whether the model accurately captured the relationship between my features and my dependent variable. I think my dataset was

comprehensive enough to cover all the job “templates”, but if a job that has a totally different flavor comes through I am curious how the model will perform. After running this in production for some time I think I will get my answer.

Improvement

I think the final result of my model could always be improved. Although the final evaluation metrics more than met my success criteria, my biggest thought for improvement would be more feature engineering or finding new features about my jobs. Although I had a limited amount of options based on the statistics we were capturing, I wonder about excluding the categorical variables. It didn’t affect performance, and will make my life a lot easier, but perhaps there is a way I could manipulate these features to add value. I also wonder if there is a better algorithm I could use for this particular type of dataset. I am familiar with the basics of the ones covered in our course, but I don’t feel I have the expertise yet to see my dataset and immediately know which algorithms make the most sense.