Carson Zhao, carsonzhao333@gmail.com ;
Leon Luc, lluc507@g.ucla.edu ; Andreas Hierl, hierl.andy@gmail.com

Team Albino Snow Leopards: Stats 101C Final Report

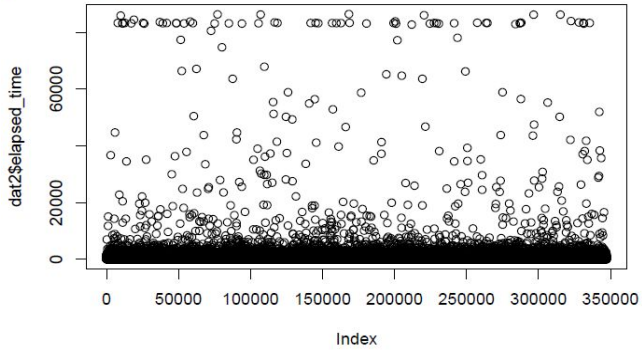
Before we could begin to fit any model, we first had to clean and transform the data. We started by renaming the variables in the training and testing data for our ease. For example, “First in District” became “first_district” and “Emergency Dispatch Code” became “EDC”. We duplicated the training and testing data into respective objects, *train* and *test*, and then removed the columns, “row.id” and “incident.ID”, because they were just identification variables, and “EDC” because all of the observations were called “Emergency”. We tried separating “incident.ID” by whether it began with FD26028 or FD26029, but a t-test indicated that there was no significant difference between the mean responses of the two groups.

There were 462,831 NA values in *train*, but since it had 2.77 million observations in total, we thought it would be best to simply delete these cases and still have 2.3 million observations to work with. Afterwards, we made the variables, “ICT” and “dispatch_sequence”, numeric. “ICT”, originally a time variable, was converted to seconds. We initially wanted to make “dispatch_sequence” categorical, but it would have had over 300 factor levels and be too computationally intensive, and so we made it numeric. We dropped “dispatch_sequence” values in *train* greater than 156 because *test* did not have any such values. We also dropped unused factor levels, “Fire Chief”, and “Planning Trailer”, that were in *train* for the same reason. Furthermore, to solve an error when predicting response values for test, we merged train unit types “Emergency Lighting” and “Swift Water Coordinator” with unit type “Command Post Unit” since they had similar mean response values (elapsed time). The remaining variables “year”, “first_district”, “dispatch_status”, “unit_type” and “PPE” were converted to factors in both *train* and *test*. These factors had 4, 102, 12, 41, and 2 levels respectively.

```
> str(train)
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':       2314921 obs. of  8 variables:
 $ year            : Factor w/ 4 levels "2013","2014",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ first_district  : Factor w/ 102 levels "1","2","3","4",..: 85 85 85 12 12 12 52 6 6 31 ...
 $ dispatch_sequence: num  1 2 3 1 3 4 1 1 2 1 ...
 $ dispatch_status  : Factor w/ 12 levels "AVI","CAV","ENR",..: 9 9 10 9 10 6 9 9 9 10 ...
 $ unit_type        : Factor w/ 38 levels "Administrative Non-Emergency",..: 35 15 26 15 15 26 26 15 26 28 ...
 $ PPE              : Factor w/ 2 levels "EMS","Non-EMS": 1 1 1 1 1 1 1 1 1 1 ...
 $ ICT              : num  82839 82839 82839 85843 85843 ...
 $ elapsed_time     : int  263 263 367 248 216 203 495 413 493 613 ...
```

After cleaning and transforming the data, we took a random sample of 30% of *train.* We wanted to get a smaller subset of *train* so that our models could run in R along with not having to wait a long time to get results. When we were trying out models in the beginning, we used an even smaller sample of 20% of *train*, but we increased this once we had a better idea of what methods were best.

```
# plot of response variable
plot(dat2$elapsed_time)
```



There were also NA values in *test* that we had to address. We couldn’t drop these because the Kaggle submission required 530,352 predictions. After realizing that all these values were from “dispatch_sequence”, we created a regression model to predict it using half of *train* with all of

the other variables besides elapsed time. Finally, since there was no unit type of “RP - Rehab Plug Buggy” in *train,* we decided to replace the one observation in *test* with that type to “RA - ALS Rescue Ambulance” since it was one of the most common unit types in *train*. Even if the two unit types are not similar, there was only one observation for “RP” so one single drastically wrong value would not cause too much trouble. We did not add any new variables.

The model that we ended up choosing was made through XGBoost (Extreme Gradient Boosting) from the “xgboost” package. It is a boosted tree algorithm that trains relatively weak tree models, but creates new tree models based off of the errors of subsequent trees. This ends up creating a strong model based off of the many previously-created, weaker models. These strong models then all contribute in determining the final prediction. The parameters used in XGBoost include: nrounds, max_depth, eta, gamma, colsample_bytree, min_child_weight, and subsample. Nrounds controls the maximum number of iterations. Max_depth describes the maximum depth of the tree. Eta helps control the learning rate by shrinking the feature weights after every round. Gamma controls the penalization of larger coefficients that don’t improve the model's importance. Colsample_bytree is related to the portion of predictors that a tree will be trained with. Min_child_weight is the minimum sum of weights of all observations required in a child. Sub_sample is the proportion of observations from *train* that will be used to train a tree.

When we first saw success with XGBoost, we decided to spend more time in tuning these parameters to get the best results. We first used 3-fold CV with 20% of *train* (and lastly 5-fold CV with 30% of *train*) with the default parameters. We then continued to change the parameter values in response to overfitting/underfitting. We changed subsample, max_depth, min_child_weight, eta, and colsample_bytree, as they help to control overfitting and underfitting the most. Eventually we settled on a certain way of cleaning our data as explained previously and the tuning parameters, as this combination resulted in one of our lowest Kaggle MSEs (polynomial + step regression was our other lowest Kaggle MSE). The final parameter values were: Eta = 0.050, Max_depth = 3, Gamma = 0, Colsample_bytree = 0.8, Min_child_weight = 5, Subsample = 1, Nrounds = 100. The best training MSE that we had with that XGBoost model was 1292198, the private and public Kaggle scores were 1429982 and **1403603**, respectively.

```
##    eta max_depth gamma colsample_bytree min_child_weight subsample nrounds
## 1 0.01         1     0              0.3                1         1     100
## 2 0.01         1     0              0.3                3         1     100
## 3 0.01         1     0              0.3                5         1     100
## 4 0.01         1     0              0.6                1         1     100
## 5 0.01         1     0              0.6                3         1     100
## 6 0.01         1     0              0.6                5         1     100
##       RMSE  Rsquared   RMSESD RsquaredSD
## 1 1260.437 0.1359836 107.4423 0.05659243
## 2 1259.415 0.1362742 109.4965 0.05642819
## 3 1263.218 0.1346914 104.1247 0.05050397
## 4 1242.409 0.1489076 108.2420 0.05358146
## 5 1239.317 0.1508061 111.6155 0.05593897
## 6 1241.921 0.1487158 108.4417 0.05431952
##
##  The training MSE is 1292198
```

XGBoost works well because it combines the strengths of other excellent regression techniques such as trees and boosting. Furthermore, we believe we were successful with XGBoost because of it has many tuning parameters. This allows for countless combinations that enables the already powerful tree-boosting algorithm to train off of the data better, since many of the parameters control for less overfitting/underfitting and be more conservative. In our data, this especially helps the cases where some of the factor levels didn’t have many observations (this would usually cause overfitting) or the cases where we observed some significantly higher elapsed time values but our model would fail to predict high enough values (underfitting). Even though we would encounter these problems for certain parameters in XGBoost, we had the luxury of continually adjusting them and come up with a model that we were content with.

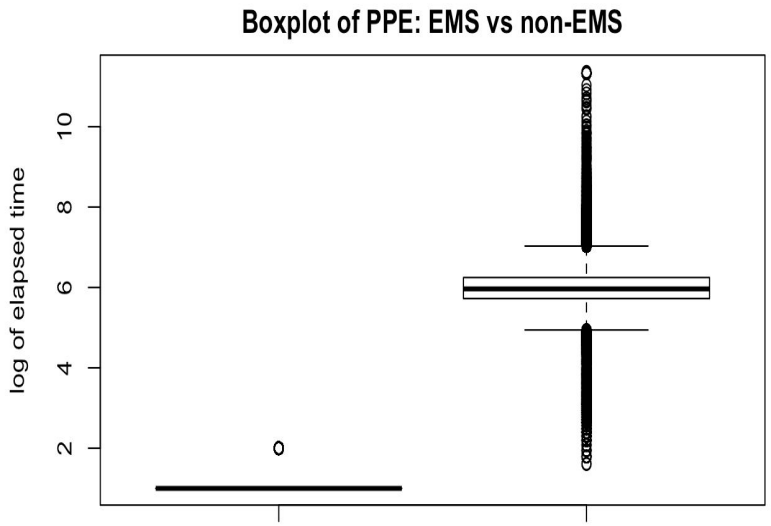


Figure 1: Boxplot of type of PPE vs log of elapsed time

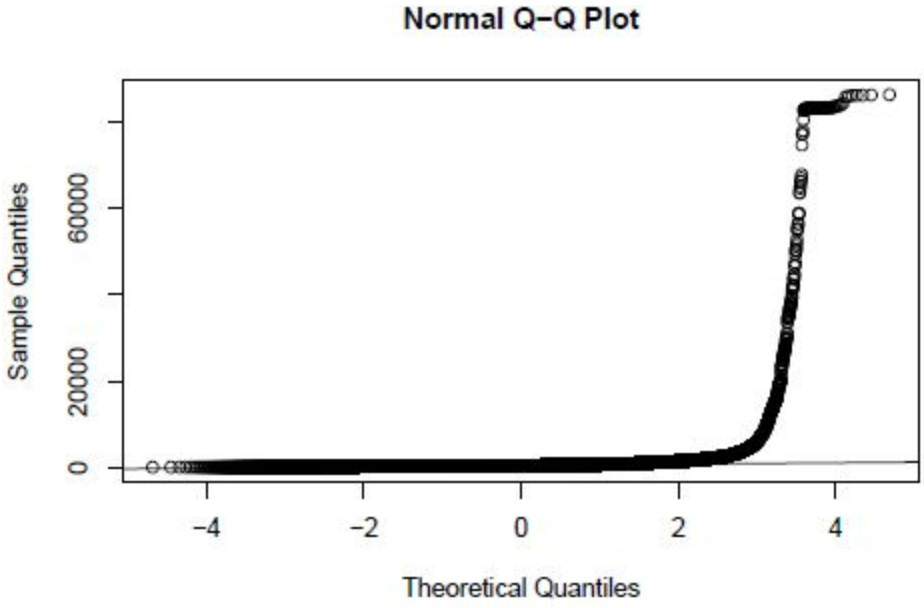


Figure 2: Normal Q-Q Plot for the response variable elapsed time