Machine Learning Engineer Nanodegree

Telstra Network Disruptions

Jeremias Binder
June 2nd, 2017

I. Definition

Project Overview

Telstra is australias biggest telecommunications and media company with an annual revenue of over 20 billion USD (A$27.1 billion).
In a recent Kaggle competition, Telstra challenged people to predict the severity of service disruptions on their network. The reason:

Telstra is on a journey to enhance the customer experience - ensuring everyone in the company is putting customers first. In terms of its expansive network, this means continuously advancing how it predicts the scope and timing of service disruptions. Telstra wants to see how kaggle users would help it drive customer advocacy by developing a more advanced predictive model for service disruptions and to help it better serve its customers.

(Telstra Network Disruptions. (2016). Retrieved and edited June 12, 2017, from the competitions page).

The data used in the competition is provided by Telstra. Service logs and data from their network nodes were prepared, anonymised and are available on Kaggle.

The competition ended at 2/29/2016, the winner was a user called 'Mario Filho'. In total, 974 teams participated in the competition.

One of the reasons i choose this project as my capstone project for the Machine Learning Nanodegree at Udacity, is that the company i am currently working at, the sovanta AG in Heidelberg, Germany, has some projects running in the domain of predictive maintenance.

Problem Statement

The problem we are about to solve, is to minimize the reaction time to interruptions in Telstras network. Long reaction times lower customer satisfaction in the long run and can therefore be costly to Telstra.
A model, that can predict accurately the network failures in advance, would be very valuable to Telstra and its customers.

The problem right now is, that Telstra has no good estimation on when one of their nodes will fail. They might have clues (certain nodes reporting an error), but no further conclusions are drawn from this data.

A predictive model can change that: The data provided by the nodes will be used to create a model, that will accurately predict failures on the network.

Since the input information is digitally obtained, each error message can be put in a certain category and is distinct. Since it's a future prediction, it's easily verifiable: After the event is predicted, the actual time and place can be observed and the degree to which the prediction is correct can be verified.

Metrics

The thing Telstra is interested in, is when and where their nodes are likely to fail. The quality of the prediction therefore matters:
To determine the quality of such a model the multi-class logarithmic loss is used. Each data row has been labeled with one true class, which represents the severity of the incident (an incident with label '0' means, there is no issue). For each row, a set of predicted probabilities is submitted (one for every fault severity). The formula is then,

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij}),$$

where N is the number of rows in the test set, M is the number of fault severity classes, log is the natural logarithm, y_{ij} is 1 if observation i belongs to class j and 0 otherwise, and p_{ij} is the predicted probability that observation i belongs to class j.
Or in layman terms: The lower the sum of wrongly predicted severities, the lower the logloss. Participants with a lower logloss on the testset get ranked higher on the public leaderboard.
Scoring a low logloss on the testset is important, since the model might overfit on the trainset.

Why does the multilogloss makes sense here?
The output of our model is the severity of a network issue ("Fault_Severity"). This severity is measured in three integer categories from zero to two. Meaning, we have exactly three labels.

But: Our model will give us predictions in the form of a probability distribution (i.e. 0: 0.212, 1: 0.573, 2:0.215). The most likely category in this example would be the second one with a chance of 0.573. If we were to measure the output label (in this example “1”), some valuable information about the quality of our model is discarded.

Now, intuitively speaking: log-loss is the cross entropy between the distribution of the true labels and the predictions. We use it to gauge the “extra noise” that comes from using a predictor as opposed to the true labels. By minimizing the cross entropy, one maximizes the accuracy of the classifier.

II. Analysis

Data Exploration

Lets take first glance at the data, which comes in six files:

train.csv - the training set for fault severity
test.csv - the test set for fault severity
event_type.csv - event type related to the main dataset
log_feature.csv - features extracted from log files
resource_type.csv - type of resource related to the main dataset
severity_type.csv - severity type of a warning message coming from the log

All the files have one attribute, that connects them all together: The column named ‘id’. Note that the ‘id’ doesn’t have the same property as a key we might now from relational databases: Sometimes one id is listed several times in one column.

After quickly scrolling through the data in MS Excel, i decided to do the following:
In order to just process numeric data, i dropped the strings from all the csv files containing the description (‘event_type ‘, ‘log_feature ‘, ‘resource_type’) manually with the excel replace function.

id	event_type
62093	~~event_type~~ 15

The thing i did then was calculating basic statistics of the data from the files to get a good first understanding.
What i found out was:

- Most of the interruptions in the *train* data are just a temporary network glitch (Fault Severity = 0), but 1 and 2 do seem to cause problems quite frequently (around 35% of the time).
- The *event types* 11,35 and 34, 15 and 2 seem occur a lot together with a network incident
- *Resource types* 8 and 2 are the most prevalent. There is a significant gap between these two resources and the rest.
- *Severity type* of warning messages 1 and 2 seem to be the most frequent ones.
- *Log features* is actually tricky to deal with: we have the ‘id’ column, then the ‘log feature’ column and a ‘volume’ column, which indicates how often a ‘log feature’ appeared with an network incident. The fact that there are 386 different ‘log features’ became a real sticking point for this problem.

Note that this is just a univariant analysis of the features provided in the single files. The initial plan was to provide a multivariant analyis of the data, but this became almost impossible, since the data is quite sparse.

So the challenge in the first place was to merge the files into one coherent frame to work with. This turned out to be more difficult than expected, since the the *log_feature* table contained 453 categorical values, which couldn’t be merged easily together with the ids of the other files.

Also, each id usually had just a few features from *log_feature*, so i decided to one-hot encode the information into the frame. Actually, i tried two versions of one-hot-encoding to solve this problem:
At first, i did a binary one-hot-encoding and discarded the *volume* information of the features, then i tried something else and used the *volume* value of each feature instead of just a one.

I ended up with a dataframe with a total of 453 features, which are quite difficult to visualize even pandas

```
print train.head(3)
```

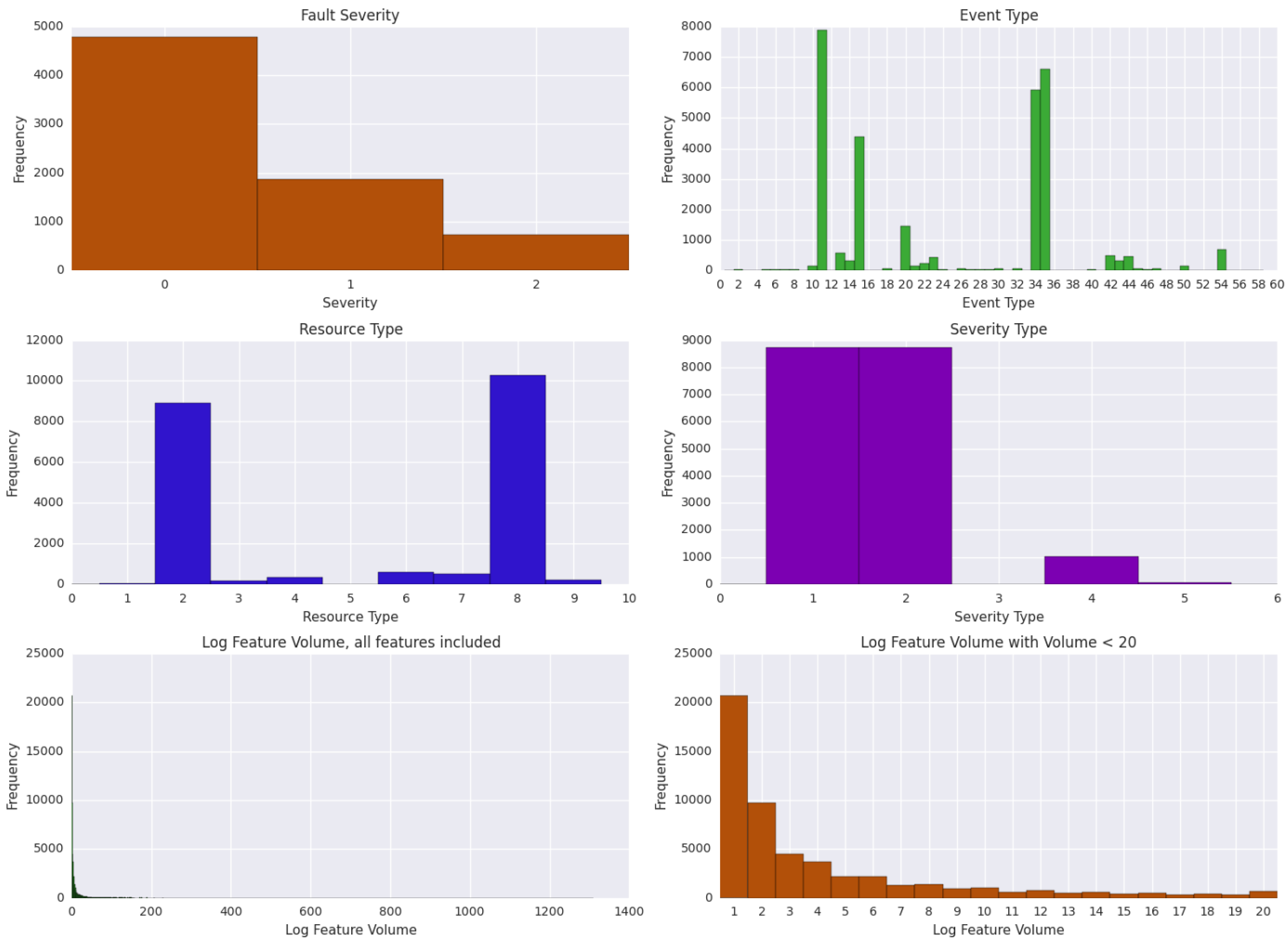
prints out a quite extensive output:

```
      id  location  severity_type  severity_order  resource_type_1  \
0  14121       118              2        0.170009                0
1   9320        91              2        0.912947                0
2  14394       152              2        0.231458                0

   resource_type_2  resource_type_3  resource_type_4  resource_type_5  \
0                1                0                0                0
1                1                0                0                0
2                1                0                0                0

   resource_type_6       ...         log_feature_377  log_feature_378  \
0                0       ...                       0                0
1                0       ...                       0                0
2                0       ...                       0                0

   log_feature_379  log_feature_380  log_feature_381  log_feature_382  \
0                0                0                0                0
1                0                0                0                0
2                0                0                0                0

   log_feature_383  log_feature_384  log_feature_385  log_feature_386
0                0                0                0                0
1                0                0                0                0
2                0                0                0                0

[3 rows x 453 columns]
```

Note that the log_feature columns contain a lot of zero values. This makes the dataset as a whole quite sparse, i assume we would have difficulties using a strong learner with this problem.

Exploratory Visualization

As already explained above, due to the sparse nature of the data, a multivariate analysis of the data is not easy to provide. However, i plotted the features in histograms, which seems like a good entry point for further discussions.



The histograms shown above give an impression on how sparse the data actually is:
While some features are distributed across all issues (Severity, Severity type) others occur unevenly. After reading about internet providers service disruptions on the internet, these graphs made much more sense to me: A guy described that usually, if a node has a issue with one feature at a time, this usually doesn't break the connection. This happens only if a certain combination of issues occur at the same time.

There are few facts about these graphs that really stick out:

- As we can see, most of the interruptions are just a temporary network glitch (Fault Severity = 0), but 1 and 2 do seem to cause problems quite frequently (around 35% of the time).
- The event types 11,35 and 34, 15 and 2 seem occur a lot together with a network incident.
- The resource types 8 and 2 are used most. There is a significant gap between these two resources and the rest.
- The severity type of warning messages 1 and 2 seem to be the most frequent ones.
- The log features behave a little different here: We have a bunch of different features, which occur in a different Frequency (Volume). The two graphs above show just how often a given feature appears with an issue. Since the feature with the largest volume occurs 1350 times, the graph on the bottom left corner is highly skewed.

Algorithms and Techniques

So, after merging the files in one single frame, i could finally get started with my very first model. Reading through some of the forums on the web and in specific kaggle, i decided to take a closer look on the xgboost algorithm.

The xgboost algorithm is a flexible, performant, and scalable gradient boosting algorithm, which gained a lot of popularity on recent kaggle competitions. Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made.

Lets have a look on the most important default parameters set in our xgboost classifier (or read the full list):

- eta [default=0.3, alias: learning_rate]: The learning rate of algorithm, Makes the model more robust by shrinking the weights on each step.
- gamma [default=0, alias: min_split_loss]: A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
- max_depth [default=6]: This describes the maximum depth of a decision tree. The parameter is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- min_child_weight [default=1]: here we define the minimum sum of weights of all observations required in a child node of a DT. Its also a measure to prevent overfitting, it avoids relations which might be highly specific to a certain sample.

The xgboost algorithm creates a bunch of weak learners, which perform very poorly (just slightly above chance) on their own. But then comes boosting, in which we start by looking over the training data and generate some distributions, then find some set of Weak Learners (classifiers) with low errors, and each learner outputs some hypothesis, H_x. This generates some Y(class label), and at the end combines the set of good hypotheses to generate a final hypothesis.

Benchmark

To determine the quality of our model the multiclass logarithmic loss function is used. Each data row has been labeled with one true class, which represents the severity of the incident (an incident with label '0' means, there is no issue). For each row, a set of predicted probabilities is submitted (one for every fault severity). The formula is then,

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij}),$$

where N is the number of rows in the test set, M is the number of fault severity classes, log is the natural logarithm, y_{ij} is 1 if observation i belongs to class j and 0 otherwise, and p_{ij} is the predicted probability that observation i belongs to class j.
Or in layman terms: The lower the sum of wrongly predicted severities, the lower the logloss. Participants with a lower logloss on the testset get ranked higher on the public leaderboard.
Scoring a low logloss on the testset is important, since the model might overfit on the trainset.

Once the model is fitted, we will run the built-in xgboost function 'predict_proba' to predict the probabilities for each severity category in the 'test.csv' file and then upload them on the kaggle submission page, to calculate against the kaggle testset.

To get a feeling on what score to expect:
The first place on the Telstra leaderboard, a user called Mario Filho, scored a log loss of 0.3954.
Once i submitted the complete dataset & ran xgboost with default settings, i scored a logloss of 0.58323. So the final prediction will probably be in the range between 0.4 and 0.55.

III. Methodology

Data Preprocessing

1. Merging the data.

As explained in the 'Data Exploration' Section, the problem with our dataset is, that the length of the different data files attributes have different length. Not every id in the one table corresponds to every id in the other tables. Some tables vary in length. In order to fit the data to the xgboost classifier, the dataframe first has to be put in shape.

To unify this information, i decided to one-hot encode all the features. While this was quite straightfoward with *events*, *severity*, and *resource type*, i had decided to change plans with *log features* at first:
I decided to drop the information of the *volume* column and make *log feature* binary.

Say for a given *id* the volume of feature 2 is 7, while log features 1-n are all zero. You can see the pure “one-hot encoded” approach in the first row, and the “volume-hot-encoded” approach in the second row.

approach	id	log feature 1	log feature 2	…	log feature n
one-hot encode	62093	0	1	…	0
volume-hot encode	62093	0	7	…	0

Now i had all the features one hot encoded in different tables.
Time, to unify them into one table. In order to do so, i merged the files one by one into the ‘train’ pandas frame. Here’s a code snippet:

```
train = pd.merge(train, pd.DataFrame(severity, columns = ['id', 'severity_type']), on='id',how='left')
```

As you can see, i am merging the ‘train’ frame with the ‘severity’ frame on their id, by using keys from the train frame, similar to a SQL left outer join. The operation preserves the key order.

I finally ended up with a 7381x453 dataframe, which contained all possible features, one-hot encoded.
So here’s a simplified version of how the table looks after all the operations. Note that it containts 453 columns (shortened here with 1…n)

id	severity type 1 … n	event type 1 … n	log feature 1 … n	resource type 1 … n	fault severity
62093	1	0	1	1	0
58261	1	0	0	…	2
…	…	…	…	…	…
28413	0	1	0	0	1

This dataset is now ready for a first fitting iteration with xgboost.

Implementation

Now that i finally got the dataframe set up, we’re ready to run the first iteration of our fault severity prediction.
As described above, i choose the xgboost algorithm as classifier. Lets split our dataset into a train- and testset, and see what logloss we can achieve with this rough first draw:

```
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(train, y, test_size=test_size, random_state=7)
model_1 = xgboost.XGBClassifier(eval_metric='mlogloss', max_depth = 5)
model_1.fit(X_train, y_train.values.ravel())
```

Note that the initial parameters for our classifier are the default settings, i just changed *max_depth* to *5*, since the default value of *3* appeared very small for such a sparse dataset. Parameter tuning and model improvement will be the topic of the refinement chapter.

The moment of truth came, the classifier accepted the frame without an error and fitted the first classifier.

```
from sklearn.metrics import log_loss
y_pred = model_1.predict_proba(X_test)
print("Logloss: %.3f" % log_loss(y_test,y_pred))
----
Logloss: 0.584
```

A value of 0.584 on the personal testset and 0.578 on the kaggle board is a quite acceptable value for the first submission without any tuning.

Refinement

1. Feature importances

The initial model is now set up, and gives us a stable result we can further build upon.
As in the preprocessing chapter described, there are a lot of attributes we included, resulting in a very sparse dataset. In order to improve the result, it is obvious to me, that the feature space needs to be reduced.

Luckily, xgboost provides us a simple tool for that:

```
model.feature_importances
```

returns us a list of how each feature contributed to the prediction. Since we have more than 400 features, it could makes sense that not all these features contribute equally to the prediction. Some might even have a zero impact.
Since this creates only noise and makes further progress harder, it does make sense to remove them.

So i tested the model with different thresholds (meaning a different amount of features) and found out, that i could reduce the feature space from n=452 down to n=133, with just a very slight increase in the log loss.

2. Feature engineering
Since i tried very hard to decrease the log loss by feature reduction, but couldn't succeed eventually, i tried to take a shot for feature engineering. Maybe i can deduce new information, based on a logic or combination of already existing data.

After reading this interesting post about proper feature engineering, i took a closer look on the data and crafted two new features:

- a ranking sorted by severity type (0.0 for the sample with the lowest int, 1.0 for the highest)
- a counter for the location (how often a particular location occurred)

This probhably made the biggest difference in the prediction process. After parameter tuning (see next paragraph) i brought down the log loss to just slightly above 0.50.

Note: If i had to continue working on the project to improve the log loss, more detailed feature engineering would probably the only way to improve significantly.

3. Tuning Parameters
Finally, after deciding on which features to use, i decided to tune my algorithm by choosing the right parameters for the xgboost algorithm. At the beginning i didn't expect it to be such a time-, and calculation intense process. Luckily, a friend from work had a AWS-coupon for me, so this allowed me to do the computations much quicker.

After i read this article on parameter tuning with xgboost, i decided to tune the following parameters (max. two at a time):

- {'num_boosting_rounds'}: numbers of trees used.
- {'max_depth' , 'min_child_weight' }:
  The maximum depth of a tree and the minimum sum of weights of all observations required
- {'gamma'}: specifies the minimum loss reduction required to make a split
- {'subsample', 'colsample_bytree'}:
  The fraction of observations to be randomly samples for each tree and fraction of columns to be randomly samples for each tree.

I expected the most impact of the to come from the 'max_depth' and 'num_boosting_rounds' variable, since they have the biggest impact on over- or underfitting.

After running the long and processing power intense tuning i could improve the log loss from …. down to ….
This is not a huge leap, but as i already mentioned, the biggest reduction in the log loss comes in this problem from choosing and engineering the right set of features.

IV. Results

Model Evaluation and Validation

So, i finally reduced the log loss from 0.5877 down to 0.5032 (on the public kaggle leaderboard). I was more than happy, since this was my first real-world application of the stuff i learned in my nanodegree. Considering the leaderboard stats, i have the impression that i chose an appropriate algorithm and covered the most important steps to achieve a good solution.

At the time i ran the fitting for the final model, i made sure to shuffle the set and cross-validate the predictor to eliminate any bias from the training set.

Also, at each step i checked samples of the data to make sure that the operations i did on the set where accurate and no problems occurred during the preprocessing and merging on the set.

The final and most important factor, that proves the robustness of the model is the fact, that it is tested against unseen data on the kaggle page. Therefore its safe to assume, that the model can be trusted and actually performs well on unseen data.

Justification

The results from the final model are improved by 0.08 (from 0.58 down to 0.50), or around 16%. For comparison: The first place on the kaggle leaderboard would have improved by around 33% by this measure.
The biggest change in log loss was achieved through implementing the ranks on the features with the highest feature importance.

Feature selection, feature engineering and parameter tuning have been discussed thoroughly, at this point its safe to say that a satisfactory solution has been found.

However, with more effort into feature engineering, i believe this solution could be further improved.

The final solution has the following major differences compared to the initial solution:

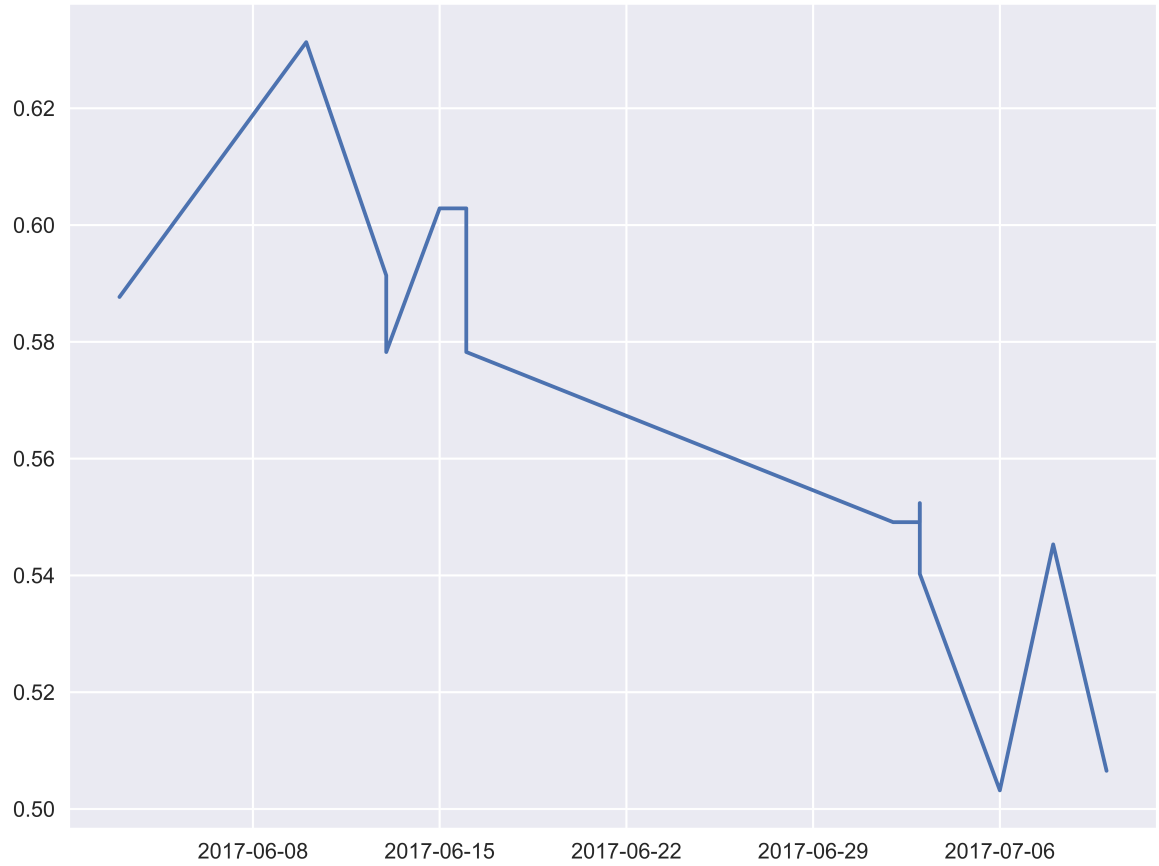
- the xgboost parameters have been tuned
- Features such as rank have been added

All the other things done in the notebook can be considered as experiments, which attempted to improve the score but eventually failed to do so.

V. Conclusion

Free-Form Visualization

As a small add on, i decided to track the improvement i did on the kaggle submission page as a graph. Note that sometimes, i really thought that i could improve the log loss (for example after i did an extensive grid search and parameter tuning) but ended up slightly worse than before. I really struggled with finding out what went wrong, so i had to rewrite parts of the notebook, which was a quite painful process.



You can see, that i had some initial troubles at the beginning, when i worked with just a fraction of the availiable features. The log loss improved after the 15th of june, when i ran xgboost on the whole dataset and experimented with feature engineering.

Reflection

The project can be summarized in four steps:
1. Feature preprocessing
2. Fitting
3. Predicting
4. Improving

The first step was to get a quick first solution that works. That was achieved by throwing all the available information into a model that seemed to be fitting to this kind of problem. So after the data was merged into one set, i fitted it immediately, without removing or adding any information. Then i checked the result and improved the solution gradually from there on.

But why not working slowly up to one “perfect solution”?
In my opinion, it can be very dangerous to spend a lot of time on one particular part of the model building process. One might easily get lost or run out of time without providing a basic solution. Thats the reason i was a big fan of gradual improvement rather than building the right model right from the bottom up.

The thing i had the most trouble with was the fact, that there were so many features, and i couldn’t find a good way to visualize them to get the most information out of it.
Also i had no domain knowledge, or in general any idea about network disruptions and how they occur. This made it kind of difficult to extract meaningful information out of the provided features. In the real world, i would prefer to have someone explaining the problem and the relations between the features to get a feeling on how the system works in general. This would make feature engineering (the thing that made the biggest improvement in the algorithm) a lot easier.

I really liked working with the xgboost algorithm. It seems to me, that this algorithm fits very good on problems with sparse data. Also, there is a great community with great ideas on kaggle, github and stackoverflow. Its very likely to find help out in the internet if one needs more information about a specific feature of the xgboost package.

Improvement

If i had to further improve the algorithm, i would definitely spent more time on feature engineering and less time on tuning parameters. This would maybe even enable me to get a score that is ranked in the top 100 of the competition. There are many creative ways to extract informations out of all the attributes and i consider it as an art to really think creatively on what features to construct.
Actually, working on this project made me curious to further dig into literature which covers the topic of feature engineering.

Also, i wish i knew ways to better visualize the data. This somehow is related to feature engineering, since visualization can really give new inspirations to what feature to engineer in the next step.

Lastly, i wish that i kept my jupyter notebook cleaner, reused functions more often and parameterize them instead of rewriting whole blocks. In the end i had trouble navigating to the whole book and finding what i was looking for. It also took longer to change certain points in the prediction pipeline. This would make it easier for third persons to understand as well.