Machine Learning Engineer Nanodegree   Capstone project

Telstra Network Disruptions

December 15, 2018   Chieko Natori

1. Definition

1.1 Project Overview

Information and Communications Technology (ICT), including the Internet and smartphones, is playing an important role in our modern society. Most of our daily activities are supported by the swift and diverse communication through the networks. Therefore, reliable network management is crucial. When the network suffers any disruption or fault, the company providing the network access needs to fix it as soon as possible. However, as the networks grow larger and more complex along with the development of our communication environment, solving network fault becomes more difficult. To satisfy customers' expectation in this challenging situation, telecommunications companies started to turn their attention to the latest big data and machine learning techniques. They expect to predict future faults by making use of data from their networks and avoid serious fault or disruption before it actually affects customers.

In this project, I would like to build a model that predicts the level of fault severity at a time and a location from network log data. I obtained the dataset from Kaggle's past competition "Telstra Network Disruptions"[1]. The dataset was originally provided by Telstra, Australia's largest telecommunications company. The dataset I use for this project consists of 5 data files shown below:

● train.csv:	the main dataset for fault severity with respect to a location and time
● event_type.csv:	event type number related to the main dataset
● log_feature.csv:	feature number and volume 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 files have been recorded in CSV format. In "train.csv", each row represents a location and a time point. It consists of 'id', 'location', and 'fault_severity'. 'id' is unique to the row. 'fault_severity' (0, 1, or 2) is a measurement of reported faults, where 0 means no fault, 1 means only a few, and 2 means many. Although the detail of the data will be explained in the later section, it has been known that the class distribution is imbalanced. Around 65% of data points belong to fault severity 0, whereas there are only 10% of samples labeled as fault severity 2.

Other 4 files contain additional information related to 'id'. The information was extracted from Telstra's log files and other sources. Every 'id' has all corresponding features, and there is no missing data observed. I will further discuss my dataset in the later section.

1.2 Problem Statement

My problem is predicting a class of fault severity at a location and a time given by using the dataset shown above. Possible fault severity is 0, 1, or 2, therefore this is a multi-class classification problem that has three classes.

To solve this problem, I will build a predictive model using a supervised machine learning algorithm. To train my model, I use 'fault_severity' in "train.csv" as a label, and information in other files as features for a specific location and time.

I read data from csv files, and merge all information into one table using ‘id’ as a key. The data should be properly preprocessed before the models are trained. In this project, I examine three algorithms: Random Forest, LightGBM, and Deep Neural Networks. After I train these models with training data, I test the model with testing data and evaluate the performance. At last I compare the results of these models. In this process, I also try two different data encoding methods. My goal is to determine the best model with the best encoding method that can predict the fault severity at the specific time and location.

1.3 Metrics

1.3.1 Multi-class Log Loss

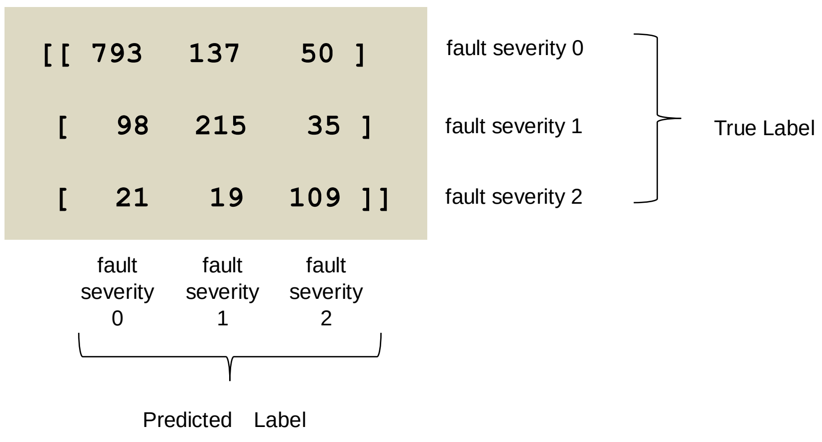
I evaluate my model using multi-class log loss. As I mentioned earlier, one of the significant characteristics of this problem is that class distribution is obviously imbalanced. Although one of the common metrics for classification is accuracy, it is inappropriate to use it in this case because it would still obtain 65% accuracy if the model predicted all samples as fault severity 0. In this problem, it would be meaningless if the model does not detect higher fault severity rather than 0 at all. While accuracy only counts the number of samples that are correctly classified, multi-class log loss takes into account the predicted probability of how much likely the data point belongs to the class. Multi-class log loss works in multi-class classification problems, especially when the number of samples is imbalanced between classes like this case. Multi-class log loss is defined as below:

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

where N is the number of data points, 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(This formula is taken from Kaggle’s competition page [1]). Smaller log loss indicates that the model is performing better.

1.3.2 Confusion Matrix

As well as mult-class log loss, I also use the confusion matrix to evaluate the quality of the model’s performance [2]. A confusion matrix shows how many points in the class are classified into each class. In this project, a confusion matrix looks like below:

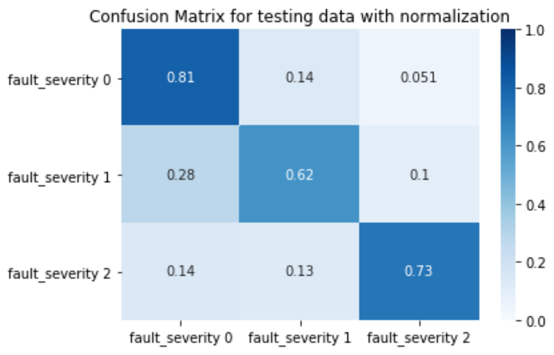


```
[[ 793   137    50 ]
 [   98   215    35 ]
 [   21    19   109 ]]
```

This example shows, of all 980 data points that are originally labeled as fault severity 0, 793 data points were correctly predicted as fault severity 0 whereas 137 were mistakenly classified into fault severity 1 and 50 were into fault severity 2. The same things go with other classes. The diagonal elements represent the number of points that are classified into the class that is same as its true label. The higher values in the diagonal elements mean the model performed better.

1.3.3 Confusion Matrix with Normalization

To see the confusion matrix in more interpretable way, I also apply normalization by class support size [3]. Each element is divided by the total number of samples in each true class. We can see how many samples in a class are actually classified into each class in proportion, which is helpful to understand the quality of the classification especially in the case of imbalanced class. The visualization example of a confusion matrix with normalization is shown below:



2. Analysis

2.1 Data Exploration

2.1.1 Contents of Dataset

My dataset consists of 5 csv files. The table below shows the contents of data files.

File Name	Columns	Type	Description
train.csv	id	int	ID that is specific to a data point. This stands for a certain time and location. e.g.) 14121
	location	object(string)	A location that is represented by a number e.g.) “location 118”
	fault_severity	int	Class of fault severity. This is the target variable. 0, 1, or 2.
log_feature.csv	id	int	ID that is linked to ID in train.csv
	log_feature	object(string)	Log feature that is represented by a number e.g.) “feature 68”
	volume	int	Volume that corresponds to the log feature e.g.) 6
severity_type.csv	id	int	ID that is linked to ID in train.csv
	severity_type	object(string)	Severity type of a warning message, which is represented by a number e.g.) “severity_type 2”
event_type.csv	id	int	ID that is linked to ID in train.csv
	event_type	object(string)	Event type that is represented by a number e.g.) “event_type 11”
resource_type.csv	id	int	ID that is linked to ID in train.csv
	resource_type	object(string)	Resource type that is represented by a number e.g.) “resource_type 8”

Data type of ‘id’ (in all data files), ‘fault_severity’ (in “train.csv”), and ‘volume’ (in “log_feature.csv”) is integer. ‘id’ is a unique number to identify each data point, so it will not be used in training. ‘fault_severity’ is a label. ‘volume’ seems to indicate a quantity of something. According to the Wikipedia [4], **traffic volume** is “a measure of the total work done by a resource or facility”. I could guess that the ‘volume’ in this data might mean that, although I do not have any further clue to understand what it is.

All other features are string data, which is written in a form of *“<feature name> <number>”*. From this data form and the explanation in Kaggle’s completion page, those features are considered to be categorical variables.

For each file, I computed the number of rows, the number of unique ‘id’, and the minimum and maximum number of unique type of a feature that is related to one ‘id’.

	Number of rows	Number of unique ‘id’	Feature	Minimum number of feature categories	Maximum number of feature categories
train.csv	7381	7381	location, fault_severity	1	1
log_feature.csv	58671	18552	log_feature, volume	1	20
severity_type.csv	18552	18552	severity_type	1	1
event_type.csv	34082	18552	event_type	1	20
resource_type.csv	22876	18552	resource_type	1	10

The numbers of rows among files are inconsistent. “train.csv” contains 7381 rows and other files contain obviously more rows and more unique ‘id’s than that, which is because there are ‘id’s that are not in “train.csv”. Of all these data points, I only use samples that are connected to 7381 unique ‘id’ being present in “train.csv”. In “log_feature.csv”, “event_type.csv”, and “resource_type.csv”, the number of rows are larger than the number of unique ‘id’. This means multiple types in a feature can be linked to one ‘id’. To be more precise, I calculated the minimum and maximum number of feature categories to one ‘id’. For a certain ‘id’, there are a maximum of 20 ‘log_feature’ and ‘volume’ pairs, 20 ‘event_type’s, and 10 ‘resource_type’s. An ‘id’ would have just one ‘log_feature’, or three ‘event_type’s, and so on.

2.1.2 Samples

To see what actual data is like, I took a sample per class (fault_severity) randomly. We can see what we have seen above more clearly.

	id	location	fault_ severity	log_feature, volume	severity_ type	event_type	resource_type
SAMPLE 0	15637	location 493	0	feature 312, 1 feature 232, 1	severity_type 2	event_type 34 event_type 35	resource_type 2
SAMPLE 1	16866	location 821	1	feature 82, 8 feature 71, 4 feature 193, 2 feature 203, 3	severity_type 2	event_type 15	resource_type 8
SAMPLE 2	5928	location 1008	2	feature 203, 4 feature 201, 3 feature 82, 6 feature 80, 4	severity_type 1	event_type 15 event_type 11	resource_type 8

It is observed that ‘log_feature’ and ‘volume’, ‘event_type’ has multiple types of features to one ‘id’. There is only one ‘resource_type’ in each sample, but it is considered to have happened by accident.

2.1.3 Class Distribution

It is important to make sure of the class distribution. Referring to the table below, I can clearly recognize that the class distribution is imbalanced. The number of samples in fault severity 0 is the highest, and it accounts for about 65% of the dataset. It is reasonable because being without a fault is a normal status for the network. We will see the visualization of class distribution in the following section.

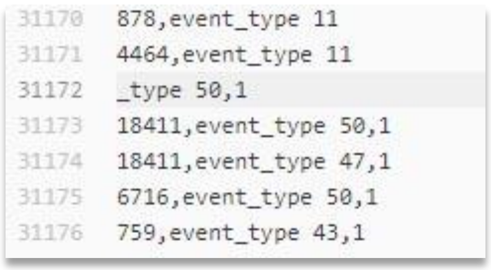
	Number of samples	**Proportion**
Fault severity 0	4784	64.82%
Fault severity 1	1871	25.35%
Fault severity 2	726	9.84%
Total	7381	100%

2.1.4 Data Abnormalities

I discovered two issues that have to be addressed before I start implementation.

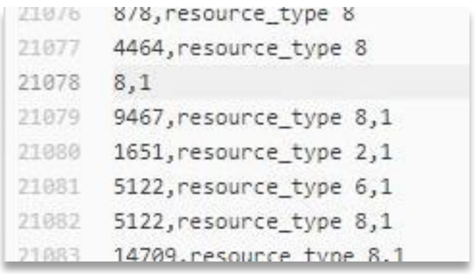
1) Invalid row format

In “event_type.csv” and “resource_type.csv”, there is a row that is written in inappropriate format in each file.



```
878,event_type 11
4464,event_type 11
_type 50,1
18411,event_type 50,1
18411,event_type 47,1
6716,event_type 50,1
759,event_type 43,1
```

event_type.csv



```
878,resource_type 8
4464,resource_type 8
8,1
9467,resource_type 8,1
1651,resource_type 2,1
5122,resource_type 6,1
5122,resource_type 8,1
14709,resource_type 8,1
```

resource_type.csv

These lines have to be discarded from the dataset.

2) Additional column

I noticed that the lines under the invalid row I mentioned above have an additional column which is not specified in the header of those csv files. In the screenshot of raw data file shown above, I can see that the lines in lower part have additional ‘1’ each. After exploration, I realized that this column was added when the specific ‘id’ had the same type of feature twice. For example, a sample with ‘id’ being 18411 has two “event_type 50”, and the ‘1’ is added to the end of the line where it appears for the second time.

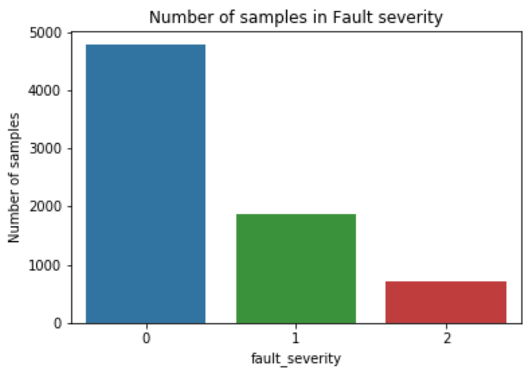
Other files do not have any problem to be addressed.

2.2 Exploratory Visualization

I would like to explain some characteristics of my data using plots.

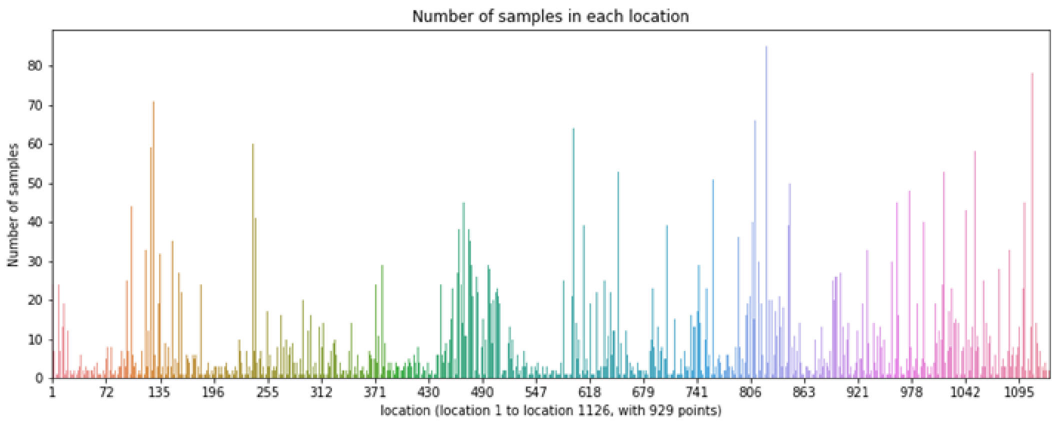
1) Distribution of fault severity

The figure below shows the number of data points in each class. Fault severity 0 holds data points most, whereas fault severity 1 has less than half of it. The number of samples of fault severity 2 is the least of all, which is less than half of fault severity 1. It is visually clear that the class distribution is highly imbalanced.

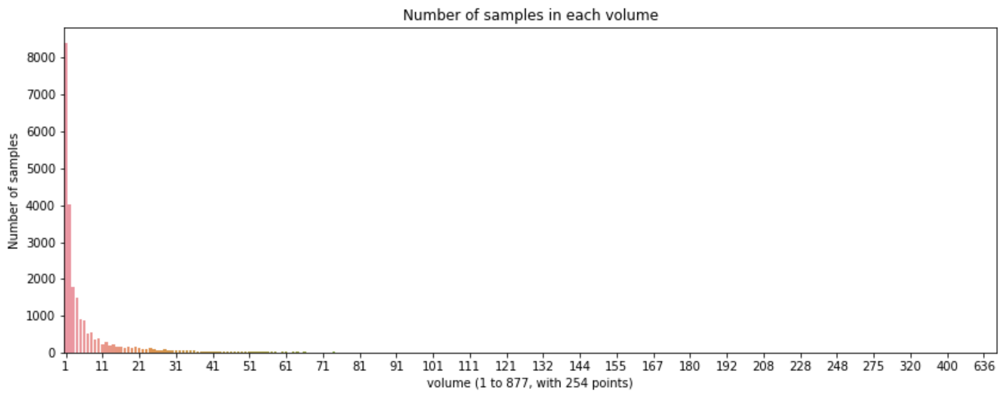


2) Distribution of other features

I also visualized the counts of other features. Each feature expresses distinctive distribution, but generally, there are some categories that have a lot of data points whereas others have only few, and it does not indicate any linear relationship between the feature number (e.g. log_feature N) and the sample count. As a representative, I put a plot of ‘location’ below.



Among those, however, the only exception is ‘volume’. The volume that holds most data points is ‘1’, the second most frequent volume is ‘2’, and it becomes less as the number gets higher.



Note that x ticks in these plots are sorted by feature number, but it does not necessarily mean that the numbers are perfectly continuous. In fact, there are some missing numbers, and the label of x axis describes the range of the feature number and the count of those points. Visualizations for other features are displayed in “Telstra_project_dataexploration.ipynb”.

2.3 Algorithms and Techniques

The algorithms I examine in this project are, Random Forest, LightGBM, and Deep Neural Networks. These algorithms are applicable to the multi-class classification problem, and all of them have a hyper-parameter of ‘class_weight’.
As I discussed earlier, this problem is an imbalanced multiclass classification task. With imbalanced class, the model tends to classify most samples as the majority class. To address this difficulty, I need to set an appropriate ‘class_weight’ parameter.

- **Random Forest**
  Random Forest is one of the most common algorithms that can work well on a range of classification problems with categorical variables. This algorithm is an ensemble of Decision Tree, where sub-samples that are drawn at random from all samples are used to train each decision tree. In default setting of the classifier, the size of sub-samples is the same as input sample size, but the samples are drawn with replacement (by default). In each decision tree, sub-samples are input to the first node and the node calculates the best threshold that does a binary split. The split repeats until all ends of the tree (leaves) are pure, or until the maximum depth of the tree (with no default value) or the minimum number of samples required in the node (default is 2) meets the condition specified by classifier's parameter. At each

node of a tree, random subset of all features are used to determine the best split (default is sqrt(number of features)). The number of trees that are made in training can be set by the parameter (default is 10). At a prediction, the Random Forest takes a majority vote of all trees. This algorithm is good at controlling overfitting, and generally better handle multiclass classification problem. So I decided to use Random Forest.

I will apply parameter tuning to the parameters below:

- The depth of the trees (max_depth)
- The minimum number of samples required to split an internal node (min_samples_split)
- The number of trees (n_estimators)

Other parameters will be set as below. The class_weight specification is especially important.

- Weights associated with classes (class_weight) : 'balanced' *1
- Random State (random_state) : 45

*1) 'balanced' is the same as the class weight defined as:

Weight of class C = number of all samples / (number of classes * number of samples in class C)

- **LightBGM**

LightGBM is also an ensemble of Decision Trees, but the difference from Random Forest is the way of ensemble. LightGBM employs a Gradient Boosting method. In the Gradient Boosting, the first tree performs the binary split, then the next tree looks for the better split than that aiming to minimize the loss function. This procedure is iterated so that it eventually reaches a strong learner. The improvement is based on the previous weak trees, which is the major difference from Random Forest. The number of iterations is specified by the parameter of the classifier (default is 100).

As well as that, the characteristic of LightGBM algorithm is that it grows trees leaf-wise whereas other Decision Tree algorithms including Random Forest grow trees depth-wise. It chooses the leaf with max delta loss to grow. According to the LightBGM document [5], it tends to achieve lower loss than level-wise algorithm. In this algorithm, the size of the growth of a tree is controlled by setting maximum number of leaves in one tree (default is 31). Another difference from Random Forest is that LightBGM has a parameter that specifies feature fraction. LightGBM randomly chooses a subset of features in the ratio set by the feature fraction in each iteration, and use only them for training (default is 1.0).

LightGBM is known to achieve better accuracy than any other boosting models, so I chose this model to try. However, I should be careful it tends to be overfitting when the number of samples is not large. To avoid it, minimal number of data in one leaf should be set to an optimal value (default is 20). Also maximum depth of the tree could be set like Random Forest. Reduction of feature fraction can contribute to avoiding overfitting because it increases the randomness of features.

The parameters below will be tuned in the implementation step:

- Minimum number of data in one leaf (min_data_in_leaf)
- Maximum number of leaves in one tree (num_leaves)

The parameters listed below will be set in order to fit the model with this problem:

- The percentage of features that will be randomly selected before training each tree (feature_fraction) : 0.5
- Maximum number of bins that feature values will be bucketed in (max_bin) : 512

- Weights associated with classes (class_weight) : 'balanced' *1
- Random State (random_state) : 45

- **Deep Neural Network**

Deep Neural Networks (DNN) are another candidate for this problem. My dataset has only 6 features, but some of them have more than hundreds of varieties, and how many feature categories in a feature belong to a sample is inconsistent. My data structure is complex, so I expected DNN could sort it out.

DNN is made with multiple layers of neural network. DNN consists of an input layer, a hidden layer that contains more than one neural network, and an output layer. In each layer, the input is turned into the output by a combination of linear equations. The result of each equation is combined and is input to activation function. The output of activation function is then input to the next network. DNN is called a feed forward network, because the input data goes through each layer towards the output layer in a one-way direction. The output layer outputs the final score (probability) for each class. Once the prediction is evaluated by the error function, weights of the equations in each layer are modified in order to minimize the error based on the gradient descent (the algorithm of gradient descent can be selected by the parameter 'optimizer'). This process is called back propagation. In training, feed forward and back propagation is repeated the number of times specified as epochs (default is 1). In this problem, the output layer should predict the probabilities of three classes for a sample. Therefore the number of the output node should be three, and the Softmax function should be used for the activation function of output layer. For loss function, categorical cross-entropy is set for multi-class classification.

Dropout can be set for each layer, which specifies how many nodes in the layer will be excluded from training in one epoch. It randomizes the use of nodes, prevents overfitting, and helps the model generalize the learning better.

To apply DNN on my problem, I have to change categorical variables into numerical variables by one-hot encoding. The dataset is supposed to have 1578 features after one-hot encoding is applied.

Architecture

I define an input layer and three consecutive fully connected layers in the hidden layer. The output layer outputs probabilities of three classes (fault severity) with respect to each sample. The number of units (output dimensions) and activation function for each layer are stated below:

1. Input layer (1578 input dimensions, 512 units, ReLU activation function, and dropout = 0.5)
2. Fully connected layer (128 units, ReLU activation function, and dropout = 0.5)
3. Fully connected layer (64 units, ReLU activation function, and dropout = 0.5)
4. Fully connected layer (16 units, ReLU activation function, and dropout = 0.5)
5. Output layer (3 units, Softmax activation function)

Parameters

Loss function and optimizer has to be specified to train the DNN model.

- Loss function (loss) : categorical_crossentropy
- Optimizer (optimizer) : RMSprop

In training, also the parameters shown below is set.

- Class weight (class_weight) : will be tuned in the implementation step
- Whether to shuffle the training data before each epoch (shuffle) : True

Parameters specified above mostly came from my pre-experiments on this dataset. For Random Forest and LightGBM, I will perform Grid Search technique to determine the final set of parameters. In Grid Search, training data is split into training set and validation set. Training is performed with every combination of expected parameters, and the result is validated to identify the best set of parameters.

2.4 Benchmark

As a benchmark model, I use Random Forest model without parameter tuning. The number of estimators (n_estimators = 100) is exclusively specified because it performed terribly without the parameter, which I thought was not enough to use as a benchmark model. Also random state (random_state = 45) is set for the reproducibility of the result.

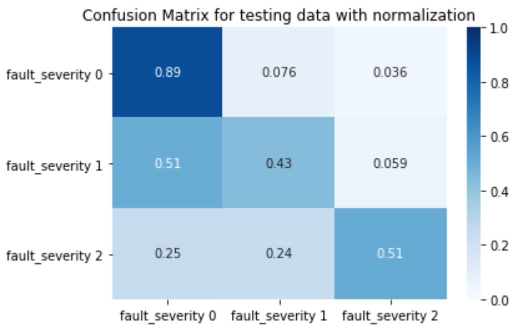
I acquired scores shown below by testing this model with input data which encoded in two ways (Refer to **3.1 Data Preprocessing**). Throughout this project, I aim to build a model that clearly outperforms these results in terms of both log loss and the quality of the confusion matrix.

- **One-hot encoding**

log loss = 0.6290

confusion matrix

```
[[838  72  34]
 [198 170  23]
 [ 35  34  73]]
```

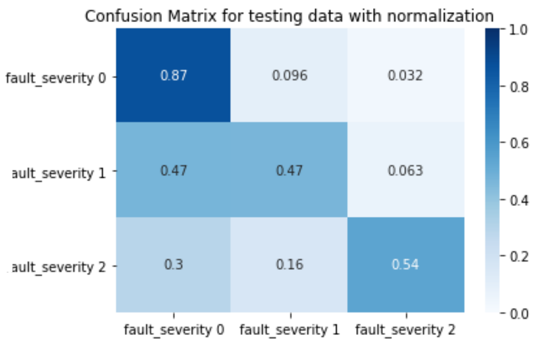


- **Numeric encoding**

log loss = 0.7212

confusion matrix

```
[[855  94  31]
 [164 162  22]
 [ 44  24  81]]
```



3. Methodology

3.1 Data Preprocessing

3.1.1 Reading data

First of all, I had to address issues I mentioned in **3.1.1 Data Abnormalities**. Because of the additional column that is not specified in the header in the CSV file, I got an error when I tried to read “event_type.csv” and “resource_type.csv”. Therefore, for those files, I decided to define the header manually so that the additional column is recognized as a normal column (I named it ‘event_flag’ and ‘resource_flag’ respectively.). For the lines that have no additional ‘1’, I temporarily padded with NaN to read these files. After the CSV files were successfully read, the abnormal lines, which was explained in **1) Invalid row format**, had to be discarded. To detect those invalid lines, I implemented the function that checks if the ‘id’ is numeric and the feature name is correctly contained in the column which represents the feature. For example, if a cell in the ‘resource_type’ column does not contain a string ‘resource_type’, it is judged to be an inappropriate line.

These operations have been implemented in read_tables() and check_bad_col() in “telstra_helper.py”.

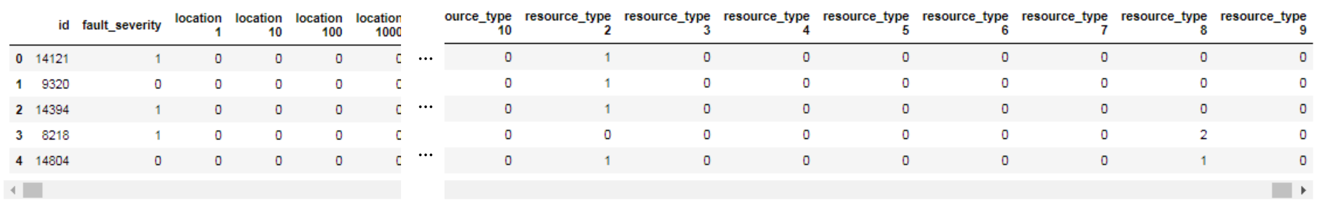
3.1.2 Encoding and Merging

Next, I merged all features into one table to make a final dataset. Categorical variables in there had to be encoded into numerical variables in this stage in order to input them into my models. In this project, I tested two ways of table structures and encoding methods.

1) One-hot encoding and frequency table

Before merging features, I had to think of how to deal with the additional column I mentioned in **2) Additional column** in **3.1.1 Data Abnormalities**. After some examinations, I decided to perform one-hot encoding, with giving 2 instead of 1 to the feature categories that happen twice to a certain ‘id’. In order to do that, I generated a frequency table where each column corresponds to each feature category. This method also solved the problem that one ‘id’ could have multiple feature categories. I applied this method to ‘log_feature’, ‘volume’, ‘event_type’, and ‘resource_type’. To other features, I just applied one-hot encoding. Then I joined each feature table to the main table that came from “train.csv” based on ‘id’.

The image below is a screenshot of the part of the final data structure for this encoding method. It has 7381 rows and 1580 columns.



	id	fault_severity	location 1	location 10	location 100	location 1000	…	ource_type 10	resource_type 2	resource_type 3	resource_type 4	resource_type 5	resource_type 6	resource_type 7	resource_type 8	resource_type 9
0	14121	1	0	0	0	C	…	0	1	0	0	0	0	0	0	0
1	9320	0	0	0	0	C		0	1	0	0	0	0	0	0	0
2	14394	1	0	0	0	C	…	0	1	0	0	0	0	0	0	0
3	8218	1	0	0	0	C		0	0	0	0	0	0	0	2	0
4	14804	0	0	0	0	C	…	0	1	0	0	0	0	0	1	0

This process has been implemented in jupyter notebook “Telstra_project_onehot.ipynb”.

2) Numeric encoding

One thing I was concerned about the one-hot encoding method above is that the data has too many feature dimensions and data is too sparse. I have one more idea of how to assemble the table, although I was not sure whether it would really work.

In this idea, I would merge features into one table, where their feature numbers were simply transformed into integer as representation of categories. However, a tricky part was how to arrange multiple feature categories in one row. Referring to the second table in **2.1.1 Contents of Dataset**, I could find how many types of features in one feature one 'id' could have at maximum. So I decided to make multiple columns for one feature so that the number of columns is the same as the maximum number of feature categories one 'id' can hold. If the number of feature categories in the 'id' is less than the maximum number (most of cases are like this), I padded '0' in the rest of columns based on the fact that every feature number starts with 1. The table below explains how those features are arranged.

id	location	fault_ severity	…	event_ type-1	event_ type-2	…	event_ type-20	…	resource _type-1	resource _type-2	…	resource_ type-10
1	601	1	…	11	13	…	0	…	8	6	…	0
5	460	0	…	34	35	…	0	…	2	0	…	0
6	332	1	…	34	0	…	0	…	2	0	…	0

20 columns for 'event_type'

10 columns for 'resource_type'

There are 74 columns in the final table. Note that this encoding is only applicable to Random Forest and LightGBM among my candidates. The implementation is in jupyter notebook "Telstra_project_numeric.ipynb".

3.2 Implementation

3.2.1 Split data into training data and testing data

After creating the input data, I excluded 'id' and 'fault_severity' to leave feature variables, and also extracted 'fault_severity' to make an array of labels. These datasets were split into training data and testing data by train_test_split() in sklearn. From the whole dataset, 5904 samples are obtained for traing data and 1477 samples for testing data.

3.2.2 Train models

Expected models were trained with the training dataset I made in the previous step.

- **RandomForest**
  RandomForestClassifier from sklearn library was used as a classifier. Grid Search was conducted by GridSearchCV() in sklearn to research optimal parameters. Parameters listed below was examined by GridSearchCV() for both data encodings.

  - max_depth : 10, 20, and 50
  - min_samples_split : 5, 10, and 20
  - n_estimators : 100, 200, and 300

  After that, I created the RandomForestClassifier with the best parameters including the parameters explained in **2.3Algorithms and Techniques**, and then train the model with training data. The screenshot below shows the final parameters I set for the classifier.

One-hot encoding

```
# train the model with the best parameters
rf_clf_best = RandomForestClassifier(n_estimators=200, max_depth=50, min_samples_split=5, \
                                     class_weight='balanced', random_state=45)
rf_clf_best.fit(X_train, y_train)
```

Numeric encoding

```
# train the model with the best parameters
rf_clf_best = RandomForestClassifier(n_estimators=300, max_depth=20, min_samples_split=5, \
                                     class_weight='balanced', random_state=45)
rf_clf_best.fit(X_train, y_train)
```

- **LightGBM**,

  To create a LightGBM classifier, LGBMClassifier in lightgbm package was loaded. The process after that is the same as Random Forest. I applied GridSearchCV() to the parameters listed below to determine the optimal setting.

  One-hot encoding

  - min_data_in_leaf : 5, 6, and 7
  - num_leaves : 50, 55, 60, and 70

  Numeric encoding

  - min_data_in_leaf : 6, 7, and 8
  - num_leaves : 45, 50, and 55

Then I set all parameters into the LGBMclassifier, and then fit the model to the training data. I show my implementation with optimized parameters below.

One-hot encoding

```
lgb_best_clf = lgb.LGBMClassifier(feature_fraction=0.5, min_data_in_leaf=5, num_leaves=60, \
                                  max_bin=512, class_weight='balanced', random_state=45)

lgb_best_clf.fit(X_train, y_train)
```

Numeric encoding

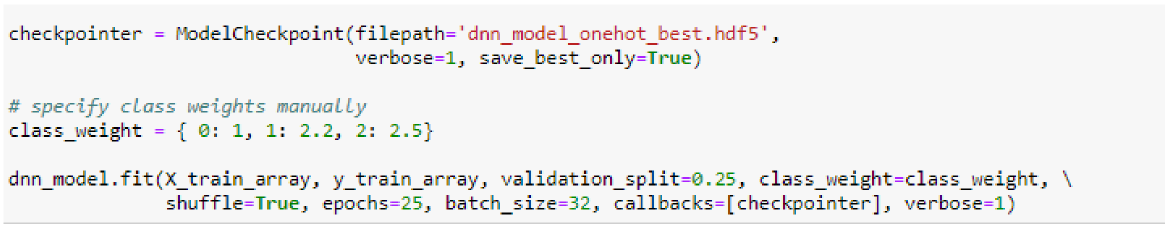
```
lgb_best_clf = lgb.LGBMClassifier(feature_fraction=0.5, min_data_in_leaf=6, num_leaves=55, \
                                  max_bin=512, class_weight='balanced', random_state=45)

lgb_best_clf.fit(X_train, y_train)
```

- **Deep Neural Networks**

  Only one-hot encoding version was used as input data for this model. I built my DNN model using Keras library. Before passing my dataset into the DNN model, I had to transform the array of labels into binary class matrix with 3 columns and rows of samples using to_categorical(). I also casted feature variables to numpy array.

The DNN model architecture was implemetend as I defined in **2.3 Algorithms and Techniques**.

I trained this model with training data. I tried training the model several times with changing ‘class_weight’ and ‘epochs’, and finally defined all parameters as shown in the code below.



```
checkpointer = ModelCheckpoint(filepath='dnn_model_onehot_best.hdf5',
                               verbose=1, save_best_only=True)

# specify class weights manually
class_weight = { 0: 1, 1: 2.2, 2: 2.5}

dnn_model.fit(X_train_array, y_train_array, validation_split=0.25, class_weight=class_weight, \
              shuffle=True, epochs=25, batch_size=32, callbacks=[checkpointer], verbose=1)
```

‘checkpointer’ was set to record the best model so that it would be loaded later for the test.

All the procedure in this section was coded in “Telstra_project_numeric.ipynb” for numeric encoding, and “Telstra_project_onehot.ipynb” for one-hot encoding.

3.2.3 Evaluation Metrics

The model I created was tested with testing data. Training data was also predicted because it is useful to see whether the model was overfitting. I evaluated the result of the prediction by multi-class log loss and a confusion matrix as I explained in **1.3 Metrics**. log_loss() in sklearn was used for multi-class log loss, and confusion_matrix() in sklearn for confusion matrix to implement metrics.

1) Random Forest and LightGBM

In order to calculate multi-class log loss, the prediction should be made in the form of probabilities. I used classifier’s predict_proba() to get the probabilities, and then input it with label data in log_loss(). On the other hand, confusion_matrix() takes in a predicted class for each sample, which is output by classifier’s predict() method.

I also calculated the confusion matrix with normalization for the prediction with testing data. I visualized this result by heatmap() in seaborn.

These metrics were implemented in logloss_confmat() in “telstra_helper.py”.

2) Deep Neural Networks

For the DNN model, metrics were implemented basically in the same way as Random Forest and LightGBM, In my DNN architecture, however, the output of the model’s predict() method is probabilities like predict_proba() above. Thus, with respect to each sample, I had to pick up the class with the highest probability from the DNN’s output in order to input it into confusion matrix(). Also the binary class matrix of class labels was transformed back to the array of labels to be used in the confusion matrix. This part of implementation required me a lot of care and precise understanding of input and output data structure of neural networks. It was especially important because if this part had been implemented in a wrong way, I could not have evaluated the DNN model properly.

Other parts of implementation are the same as Random Forest and LightGBM. The metrics for the DNN was implemented in logloss_confmat_dnn() in “telstra_helper.py”.

3.3 Refinement

After seeing those models' performances, I attempted to improve the quality of prediction. Because I had seen that the classification for fault severity 1 was relatively poor, my first aim was to improve that. I also expected overall improvement. I chose LightGBM model with data of numeric encoding, which exhibited the desirable performance so far, and applied two methods on it.

1) Set class_weight manually

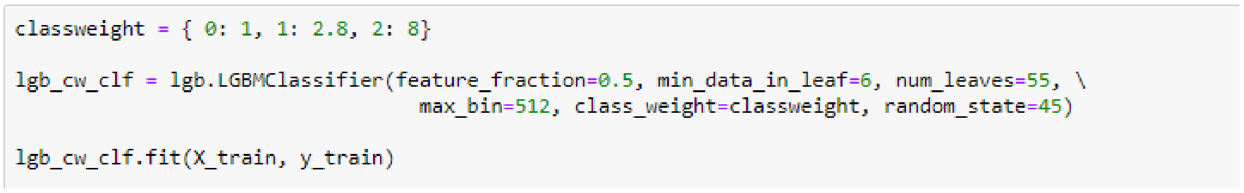
In order to deal with the imbalanced class, I set "class_weight = 'balanced'" for the LightGBM classifier. I calculated actual class weights of 'balanced' based on the formula I mentioned in **2.3 Algorithms and Techniques**, and in the case of this dataset, it turned out that the classifier set class weights as shown below:

fault severity	weight	ratio to fault severity 0
0	0.5174	1.0
1	1.2922	2.4977
2	3.4107	6.5927

From there, I varied the value of class weights manually so that the classification of fault severity 1 got better. Also I had to pay attention to the balance between classes to ensure the whole quality of prediction. After adjustments, I determined my final class weights as below:

fault severity	weight
0	1
1	2.8
2	8

Implementation was done as shown in the code below.



```
classweight = { 0: 1, 1: 2.8, 2: 8}

lgb_cw_clf = lgb.LGBMClassifier(feature_fraction=0.5, min_data_in_leaf=6, num_leaves=55, \
                                max_bin=512, class_weight=classweight, random_state=45)

lgb_cw_clf.fit(X_train, y_train)
```

2) Select important features

As I mentioned above, models seemed they particularly confused samples in fault severity 1. I suspected it might be because some features gave irrelevant information to the model, which acted as noise and made the model confused. I acquired feature importance from feature_importances_ attribute of the trained LightGBM classifier to see which features had more impact on the model's classification.

Top 20 features with higher feature importance are listed below:

```
*** LightGBM  (class weight='balanced')  ***
FEATURE: FI VALUE, PERCENT (CUMULATIVE VALUE [PERCENT])
--------------------------------------------------
(1) location: 2763, 17.06% (2763 [17.06%])
(2) log_feature-1: 1631, 10.07% (4394 [27.12%])
(3) log feature-2: 1460, 9.01% (5854 [36.14%])
(4) volume-2: 1359, 8.39% (7213 [44.52%])
(5) volume-1: 1183, 7.30% (8396 [51.83%])
(6) volume-3: 958, 5.91% (9354 [57.74%])
(7) log feature-3: 923, 5.70% (10277 [63.44%])
(8) log_feature-4: 782, 4.83% (11059 [68.27%])
(9) volume-4: 694, 4.28% (11753 [72.55%])
(10) event type-1: 595, 3.67% (12348 [76.22%])
(11) event_type-2: 513, 3.17% (12861 [79.39%])
(12) log feature-5: 454, 2.80% (13315 [82.19%])
(13) severity type: 344, 2.12% (13659 [84.31%])
(14) resource_type-2: 332, 2.05% (13991 [86.36%])
(15) volume-5: 313, 1.93% (14304 [88.30%])
(16) log feature-6: 286, 1.77% (14590 [90.06%])
(17) event_type-3: 286, 1.77% (14876 [91.83%])
(18) resource type-1: 259, 1.60% (15135 [93.43%])
(19) volume-6: 233, 1.44% (15368 [94.86%])
(20) event_type-4: 166, 1.02% (15534 [95.89%])
```

'location', 'log_feature-1', 'log_feature-2', 'volume-1', and 'volume-2' account for 50 % of all feature importance. Basically, the lower the number of 'log_feature-N' or 'volume-N' is, the denser the data is. I recreated the input data only with these features and some other selected columns as variations, and train and test the model. Through examination, my final dataset ended up to consist of 13 columns shown below:

- location
- log_feature-1, log_feature-2, log_feature-3, log_feature-4, log_feature-5,
- volume-1, volume-2, volume-3, volume-4, volume-5, volume-6
- severity_type

The procedure of refinement was implemented in jupyter notebook "Telstra_project_numeric.ipynb".

4. Results

4.1 Model Evaluation and Validation

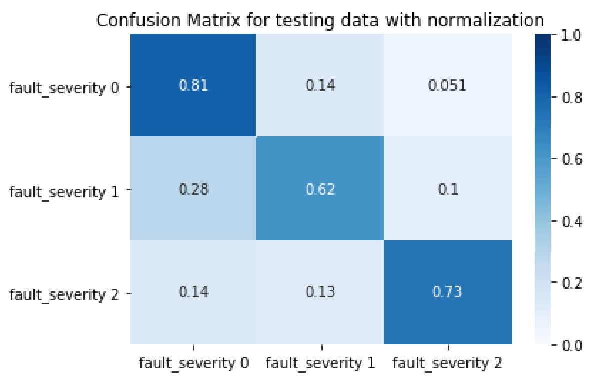
The performance of each model I trained was evaluated by metrics I stated in **1.3 Metrics**.I show log loss and the confusion matrix of all models below.

Model	log loss		confusion matrix	
	train	test	train	test
Input Data: One-hot encoding				
Random Forest	0.3445	0.5840	[[3459 270 111] [73 1376 31] [0 5 579]]	[[774 110 60] [118 229 44] [8 30 104]]
LightGBM	0.3829	0.5994	[[3128 548 164] [96 1324 60] [1 6 577]]	[[736 146 62] [85 258 48] [11 21 110]]
DNN	0.6070	0.6777	[[2589 1066 185] [187 1142 151] [19 103 462]]	[[623 263 58] [70 272 49] [9 31 102]]
Input Data: Numeric encoding				
Random Forest	0.3218	0.5707	[[3486 209 109] [54 1426 43] [1 1 575]]	[[793 137 50] [98 215 35] [21 19 109]]
LightGBM	0.3165	0.5730	[[3221 443 140] [62 1424 37] [1 0 576]]	[[758 165 57] [67 243 38] [12 23 114]]
LightGBM + class weight	0.3266	0.5816	[[3148 507 149] [46 1441 36] [1 0 576]]	[[734 185 61] [58 252 38] [11 22 116]]
LightGBM + Feature Selection	0.3008	0.5654	[[3275 403 126] [47 1443 33] [0 0 577]]	[[752 170 58] [70 236 42] [15 18 116]]

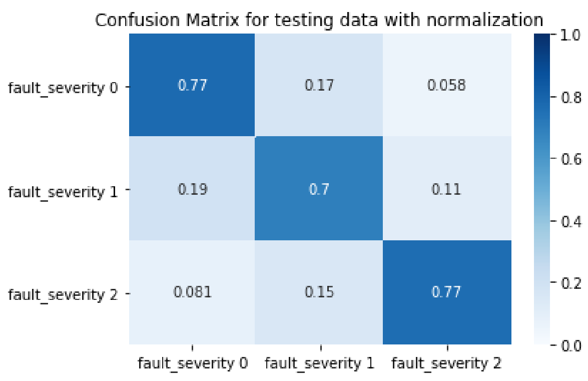
One thing I have to point out is all models seem to be overfitting. I tried to optimize the parameters to avoid overfitting, but I could not fix it.

Looking at the test result, I can say that the numeric encoding generally seems better than one-hot encoding, in terms of data encoding. Comparing log loss values, both Random Forest and LightGBM with numeric encoding hold lower log loss (the lower value means it performs better) than those with one-hot encoding. DNN model ended up in the worst log loss of all models. When we look at log loss, LightGBM with Feature Selection seems to be the best model.

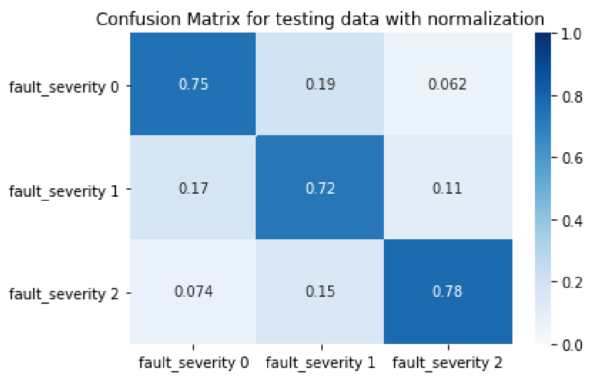
However, a confusion matrix with normalization gives us a different aspect of the performance quality. I would like to show confusion matrices with normalization of four models with numeric encoding.



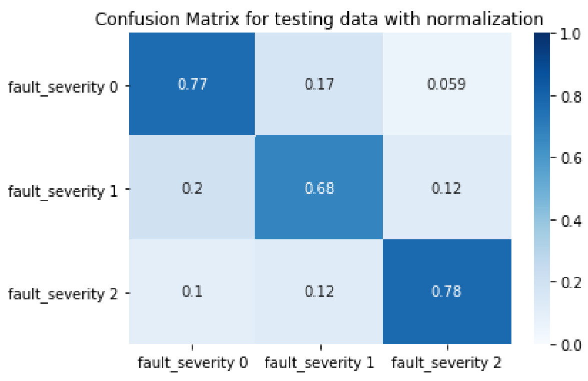
Random Forest



LightGBM



LightGBM + Class Weight



LightGBM + Feature Selection

From these matrices, we can see the proportion of samples that are classified correctly (or incorrectly) with respect to each fault severity class, which helps us grasp the model's behavior more accurately and intuitively. From my point of view, both LightGBM and LightGBM with class weight seem to be the best referring to these heat maps. It is because, in both cases, the number of samples that were classified correctly reached to more than 70% of the members of that class, and all classes attained that level. The other two models have some better points, but the classification of fault severity 1 is not satisfactory.

From the result of log loss and the appearance of the confusion matrix, I will choose LightGBM (with class_weight='balanced') as my final model. The methods I tried in **3.3 Refinement** did not contribute to a significant improvement.

To verify the model's quality, I fed it new data which were never used during the whole process I have documented so far. I had one more file called "test.csv" from Kaggle's competition page, as well as what I listed in **1.1Project Overview**. It contains 'id' and 'location', where any id is not included in "train.csv". I connected all other features to the 'id' in "test.csv" as I did on "train.csv" by numeric encoding, and input it to the model. Although I could not evaluate its result with metrics because "test.csv" did not have labels ('fault_severity' column), still it gave me an insight about the model's performance to unseen samples. I calculated the proportion of samples that were classified into each class, with respect to test data (used in model evaluation) and this new data. I obtained the result as shown below:

data	fault severity 0	fault severity 1	fault severity 2
test data	837 (56.67%)	431 (29.18%)	209 (14.15%)
new data	5998 (53.69%)	3521 (31.52%)	1652 (14.79%)

For both datasets, the model split samples into the similar ratio among three classes. Supposing that the underlying distribution between two datasets are resemble, it can be said that the model proved its consistency to the data the model was never exposed to.

4.2 Justification

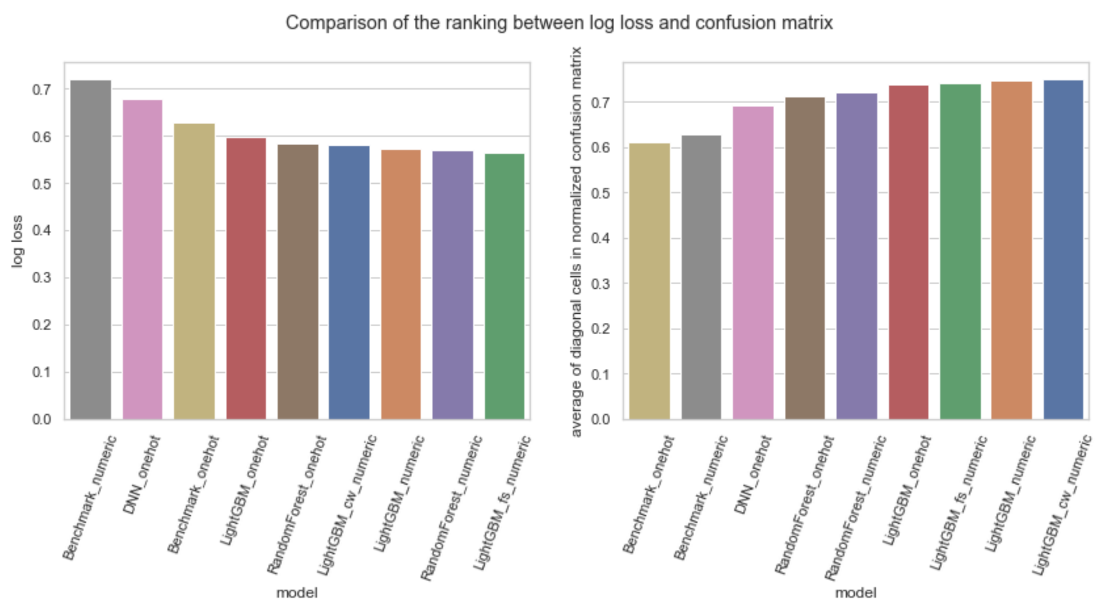
I compare the performance of my final model to the benchmark I defined in **2.4 Benchmark**. Regarding to log loss, my final model achieved significantly better log loss than that of the benchmark. The confusion matrix also displays a remarkable difference. In both confusion matrices of the benchmark model, samples of fault severity 1 are mixed up. In the default setting, Random Forest algorithm does not seem capable enough of dealing with this problem. In contrast, my final LightGBM model expresses its ability to handle this imbalanced class problem relatively more appropriately. Classification on fault severity 1 and 2 is significantly outperforming the benchmark, even though 70% and 77% are not a perfect level of prediction. Taking all into consideration, I can say my final model predicted fault severity with acceptable quality.

5. Conclusion

5.1 Free-Form Visualization

In this project, one thing that required me a lot of thoughts was about how to choose the best model. I used multi-class log loss and confusion matrix, and in some cases these two metrics did not agree with each other.

In the figure below, I would like to compare how the ranking of models are different between the value of log loss and the proportion of correct classification in the confusion matrix.



In these bar plots, models are sorted in descending order, where the models get better from left to right. The left figure illustrates the ranking of models by log loss. In the right figure, each bar represents the average of diagonal cells in the confusion matrix with normalization. Throughout the evaluation process, I mainly noticed the value of those cells to judge whether the model sorted out the problem properly or not.

When we simply see log loss, LightGBM and feature selection with numeric encoding (green bar) is the best model, but in the right figure, it drops to the third place. Log loss of Random Forest with numeric encoding (purple bar) is in the second place, but the confusion matrix of it looks dull. LightGBM and class weight adjustment with numeric encoding (blue bar) seems the best referring to the confusion matrix, although it is not as good as that in log loss.

At last, I ended up to choose the LightGBM model with numeric encoding (orange bar), based on how well the classification was accomplished on all three classes, as well as the value of log loss. The point I would like to emphasize is that I felt just one metric might not be enough to evaluate the quality of a model. One of the most challenging parts in this project for me was how to find an optimal point between these metrics when they disagreed and how I interpret the outcome.

5.2 Reflection

My problem in this project was to predict the level of fault severity in the networks at any given time and location from log data. I aimed to solve this problem by building a supervised learning model. After data exploration, I decided to preprocess data in two ways: one-hot encoding and numeric encoding. Then I trained Random Forest, LightGBM and Deep Neural Networks models by the encoded data. I evaluated the performance of each model by multi-class log loss and a confusion matrix. At last, I concluded that LightGBM with numeric encoding was my best model to solve this problem.

Overall, this problem was very challenging for me. I discovered that handling categorical variables was really tricky. In this project, some features had hundreds of possible categories, which made the data extremely sparse when it was encoded by one-hot encoding. I was disappointed that DNN did not work as well as I expected. Generally, numeric encoding worked better than one-hot encoding with any algorithm in my experiment, which was another fascinating discovery. In real-world data, categorical features with so many variables like this project could be possible. This project brought me a lot of insight about categorical data. Even though my model needs further improvement to be applied to a real situation, I believe machine learning techniques have a great potential to solve this kind of problem effectively.

5.3 Improvement

One thing I wanted to try in this project (but I could not do) was Feature Engineering. I saw the model classify training data very well, so I thought the algorithm itself has an enough ability to identify samples. However, it also meant overfitting. When I did feature selection to refine my model, I recognized that ‘location’, ‘volume’, and ‘log_feature’ were the important features. But these features have a large number of feature categories. I assumed that these features had so many categories, which made data too specific to each samples, that it would not be easy for the model to generalize the classification. I wanted to discover the possible segments in features, and replace real features to the represented segments to simplify the categorical variables. In this dataset, I only had feature numbers to identify a category in a feature, and there was no way to recognize any groups or similarities only from those numbers. But I suspect there should be some underlying groups in the features. For example, I only had the ‘location N’ in ‘location’, and I had no idea whether the number (N) represents the actual geographic relationship, like location N and location N+1 are neighbors in the network. However, it might be possible that the locations that are physically close to each other would have similar tendency to have traffic accidents or disruptions. If I could do clustering, I would be able to reduce the number of feature categories and the data would more represent the characteristics of features. I hope it would reduce overfitting and significantly improve the model’s performance.

Reference

[1] Kaggle (2016) *Telstra Network Disruptions* (https://www.kaggle.com/c/telstra-recruiting-network)

[2] sikit-learn *3.3.2.5. Confusion matrix* (https://scikit-learn.org/stable/modules/model_evaluation.html#confusion-matrix)

[3] sikit-learn *Confusion matrix* (https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html)

[4] Wikipedia *Network traffic* (https://en.wikipedia.org/wiki/Network_traffic)

[5] LightGBM *Features* (https://lightgbm.readthedocs.io/en/latest/Features.html)