INFO 5731- Computational Methods for Information Systems

FINAL TERM PROJECT REPORT

Waze Traffic Data Analysis

TEAM MEMBERS:	EMAIL
1. Shashank Ghanta	shashankghanta@my.unt.edu
2. Sai Varun Teja Mudumba	saivaruntejamudumba@my.unt.edu
3. Venkat Rohan Gupta Bompally	venkatrohanguptabompally@my.unt.edu

Purpose and Objective

The main purpose of the project is to apply all the knowledge and techniques that we have worked on during the coursework and build a prediction model based on the analysis made on the Waze Traffic data obtained from the Waze application. The data provided has about 60 lakhs of rows and about 30 columns which includes the columns such as Facility Name which has the names of all the roads, Event Type which has the information about the issues like road blocks, accidents, traffic jams and the timestamp at which these incidents have occurred. We pose the car accident risk prediction as a classification problem with two labels (accident and minor accident). Our objective of the project is to build a model which can predict accident occurrences based on the previously available patterns/sequences obtained from the original data after running through the required data pre-processing and data cleaning steps.

Design Description

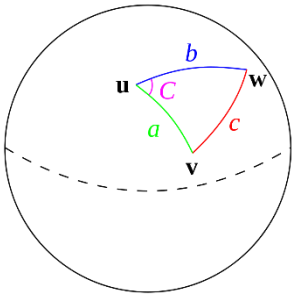
In our first observation we found that there are two types of accident events named “accidents” and “minor accidents”. We plan to separate these two events from the remaining events and create sequences with a time frame of 40 – 50 Mins (time frame will be varied to check the accuracy of the model and then finalize a time based on the best accuracy) for every event. Later the sequential data will be divided in to test, valid and training sets to train and test the model to predict accidents.

Function Description:

Haversine Formula:

The haversine formula determines the distance between two points on the surface of sphere using latitudes and longitudes. This haversine function requires four parameters. The latitude, longitude of point one and the latitude, longitude of point two. The term haversine comes from haversine function given as:

$$\text{haversine}(\theta) = \sin^2(\theta/2)$$



Sequences Generation:

The Function Sequences Generation need two parameters as requirements. The list of accidents and the list of roads. This function generates sequences with a time frame of 30 minutes. These sequences are validated and checked whether these events occur under 10-mile radius. The events that occur more that 10-mile radius are removed from these sequences. The output obtained from this function are the list that contains sequences of events occurred before the accident.

Event Dictionary:

The function event_dict requires list of unique events. These events are listed and converted into dictionary. The output produced from this function are the dictionary of events.

```
events=list(dfmain['EVENT_TYPE'].unique())
events
```

```
['minor accident',
 'other',
 'road closed',
 'stopped car on the shoulder',
 'traffic jam',
 'major event',
 'pothole',
 'missing sign on the shoulder',
 'stopped car',
 'hazard on the shoulder',
 'accident',
 'flooding',
 'road construction',
 'stopped traffic',
 'object on roadway',
 'animal struck',
 'heavy traffic',
 'small traffic jam',
 'slowdown',
 'animal on the shoulder',
 'large traffic jam',
 'traffic heavier than normal',
 'malfunctioning traffic light',
 'medium traffic jam',
 'hazard on road',
 'weather hazard',
 'road closed due to construction',
 'hail',
```

Reverse Dictionary:

The function reverse_dict is used to create a dictionary of events from the data set which will be used to form the sequences in the model.

Sequence-Padding:

We performed the sequence padding because we have the sequences of different lengths, and if we give the model as it is, the model may treat the sequence superior to another based on the length. So, we pad all the sequences of same length with zeros using the pad_sequences from keras.

Training and Validation Data:

We extract the labels from the data frame into an array and name it labels. After which we split the entire sequences into Training and Validation sets in the ratio of 80%, 20% after shuffling the sequences and labels by using the 'shuffle' method from numpy. The index for the training sample can be obtained by multiplying the length of the indices with 0.80. Similarly, the index for the validation set is obtained by subtracting the training samples from the length of indices.

Embedding Matrix:

We create an embedding matrix with rand function by passing the number of events as event list plus one as we padded zeroes and embedding dimension as 20.

```
embedding_matrix = np.random.rand(num_events, embedding_dim)
```

```
embedding_matrix
```

```
        0.78244882, 0.8624164 , 0.67383799, 0.67705388, 0.7893716 ,
        0.39578405, 0.75783668, 0.15170496, 0.69862835, 0.50314497,
        0.38008585, 0.54893888, 0.02748248, 0.376243  , 0.69696506],
       [0.59310036, 0.89443076, 0.99384781, 0.14677215, 0.14640427,
        0.66635587, 0.45424183, 0.25896566, 0.09527174, 0.21527424,
        0.59687015, 0.09948284, 0.18682151, 0.45372142, 0.14746822,
        0.27550535, 0.03028523, 0.27528396, 0.14790651, 0.5317522 ],
       [0.03314711, 0.12651555, 0.16147474, 0.52903961, 0.34952145,
        0.656561  , 0.61474799, 0.58785023, 0.32501019, 0.05701557,
        0.97878117, 0.66506606, 0.09367309, 0.54565025, 0.84521916,
        0.28744574, 0.10735615, 0.61225214, 0.57324503, 0.5659908 ],
       [0.51830664, 0.98509364, 0.00132608, 0.60667856, 0.80383634,
        0.19489587, 0.38576849, 0.49647409, 0.61739358, 0.65100771,
        0.95351063, 0.41602287, 0.0071967 , 0.34435269, 0.28966679,
        0.44135416, 0.55125725, 0.95895722, 0.7007052 , 0.7177643 ],
       [0.5739394 , 0.18899869, 0.06091098, 0.77931143, 0.06674488,
        0.86479469, 0.91622463, 0.98059582, 0.09090802, 0.40988679,
        0.79128707, 0.47332613, 0.8992363 , 0.59236118, 0.38658665,
        0.58573587, 0.33077392, 0.06799466, 0.07841327, 0.93438996],
       [0.59415626, 0.25997077, 0.16386182, 0.07273381, 0.11841677,
        0.53576933, 0.53804793, 0.25690145, 0.99394482, 0.94121718,
        0.04164026, 0.62351361, 0.77511044, 0.16060544, 0.27342057,
        0.72174011, 0.92324252, 0.78230861, 0.3991275 , 0.06016461],
       [0.99717158, 0.3304414 , 0.4439444 , 0.85758444, 0.74750753,
        0.0426144 , 0.14110584, 0.95579349, 0.32372755, 0.78830503,
        0.94078262, 0.04334779, 0.49263676, 0.47636178, 0.09558969,
        0.98258465, 0.38104738, 0.56036502, 0.87835477, 0.57701907],
       [0.38309559, 0.48158373, 0.6689119 , 0.06368938, 0.10123553,
```

Sequential Model:

We initialize the sequential model to model and then we add embedded layer followed by the Embedding layer followed by the LSTM layer. Then we add the model with the dense layers with different activations (such as ReLU and sigmoid). We then compile the model with a loss function of binary cross entropy, optimizer adam and rmsprop. Then at last we fit the training set by assigning the validation with the valid data for different epochs and batch sizes.

Python Tools and Modules

We used a variety of modules and tools for our analysis which include Pandas to create data frames and manipulate data. We will be using Google Collab with GPU or TPU enabled to run the model. Since we will have to predict the occurrence of accidents from the sequence model, we used Keras to train on the sequences that we will extract from the original data after proper data pre-processing. Matplot is used to manage the Graphical representation of data. Numpy.

Programming strategies:

Feature Engineering:

The initial stage in any Data Analysis project is about the data exploration. Knowing about data in depth should be the major priority of any data analyst. We used pandas to extract the data from the pickle which had about 30 lakh rows with around 10-12 columns. First we clean the data because to treat the missing values. And we take into consideration of only the required columns from the entire dataset and in our case our primary focus is with the Event_Type (which has Accidents, Minor Accidents), Facility_Name (name of the roads), Latitude and Longitude and the timestamp of the event. On careful observation we noticed that there were some duplicate values in the dataset which does not constitute in conveying more information about the model that we are going to predict. So, we removed the duplicates using the functions of pandas. On looking at the information of the data frame(df.info()) we noticed that the Latitude and Longitude values were of type Float for some records and String for some records. So, we changed the Latitude and Longitude to the type Float in the data frame.

Sequence Generation:

Our main motto is to predict the occurrences of accidents and minor accidents based on the previous updates on the same road within a span of 30minutes on the same road which fall in a radius of 10 miles. In order to obtain the accident and minor accident causing sequences we extract the timestamps, of the records where an accident or minor accident has been recorded. We initialize it as the end time and using the time delta function from the date time module we get the start time by subtracting 30 minutes from the end time. In order to put this in a loop we dump all the roads in where at least a single accident or minor accident has been recorded. And then we put that list in the loop, obtaining the time stamps of every record that resulted in Accident (i.e. end time) thus by generating the start time by back tracing 30 minutes.

Then we mask the data records of that road, which fall in between the start time and end time into a dummy data frame for further use. After masking the dummy data frame, we use it to get the latitudes and longitudes to filter the records which fall in the ten-mile radius.

The last record in the dummy data frame is either an accident or the minor accident. So, we assign the Latitude and Longitude of that record as the center co-ordinates. By using the haversine formula we evaluate the difference between the two pairs of latitudes and longitudes.

```
from math import radians, cos, sin, asin, sqrt

def haversine(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 3956 # Radius of earth in kilometers. Use 3956 for miles
    return c * r
```

We loop through all the other records in the dummy data frame and compare it with the center co-ordinates. If the distance between them is less than 10, then we append it to the sequence. And after looping through the dummy data frame, we append the obtained list to our final sequences list.

```
seqsna=[]
etimena=[]
radius=10.00
for ix in range(len(dfsam)):
    road=dfsam['FACILITY_NAME'].iloc[ix]
    dfuse1na=df3[df3['FACILITY_NAME'] == road]
    etimena=dfsam['time'].iloc[ix]
    stimena= etimena -timedelta(minutes=30)
    maskna = ((dfuse1na['time'] >= stimena) & (dfuse1na['time'] <= etimena))
    dfusena=dfuse1na.loc[maskna]
    dfusena=dfusena.reset_index(drop=True)
    clatna=dfsam['LAT'].iloc[ix]
    clonna=dfsam['LON'].iloc[ix]
    seq1na=[]
    for i in range(len(dfusena)-1):
        lat2na=dfusena['LAT'].loc[i]
        lon2na=dfusena['LON'].loc[i]
        na=haversine(clonna, clatna, lon2na, lat2na)
        if na< radius:
            seq1na.append(dfusena['EVENT_TYPE'].loc[i])
    seqsna.append(seq1na)
```

Similarly, we need to feed the model with the sequences which resulted in events other than accidents, which helps the model to distinguish between the accident causing and non-causing patterns. As the data frame has large number of rows (30 Lakhs), of which just 45000 records being the accidents and minor accidents, it becomes difficult to generate sequences for the other events. So, we randomly select around 50,000 records from the records which did not result in accidents or minor accidents using the random function and run through the similar loop as followed for the accidents.

After generating the sequences for both Accident causing and Non causing scenarios, we observed that there are empty sequences, which do not convey any pattern and they cannot be used in prediction cause they do not possess any information about the accidents and we don't need them in the model. So, we remove the empty lists from the generated sequences. Then we create the data frame with columns events and label. We load the accident sequences under the events with label 1 and non-accident sequences as label 0.

```
cdf=pd.concat([adf, nadf])
cdf.head()
```

	label	events
0	1	[stopped car on the shoulder, stopped car on t...
1	1	[traffic jam, traffic jam, traffic jam, medium...
2	1	[stopped car on the shoulder]
3	1	[slowdown, stopped car on the shoulder, major ...
4	1	[stopped car on the shoulder, stopped car on t...

Normalization:

It is important about the data that is being used for training the model, because, the model learns purely from the data. And any misalignments in the data would result in a very bad model. And in our case since the model must predict whether a sequence results in Accident or not. We need to provide equal records of accident and non-accident scenarios. Plotting the value counts of the label column of the data frame. We initially noticed the irregular distribution of 0's and 1's so we normalise the data frame by removing the excess records randomly by using the sample method.

Categorical Encoding:

Since the machine only understands numbers, we cannot load in the list of sequences and except the desired result. So, before we train the model with the data, we encode it using the event dictionary which a unique value for every event. Using the function stated below:

```
[ ]  def event_list_to_index(event_list):
       list_idxs = []
       for event in (event_list):
         idx = reversed_dict[event]
         list_idxs.append(idx)
       return list_idxs
```

we encode the events using the dictionary.

Sequence Padding:

As discussed above the sequences are of different lengths due which the results of the model will be affected. To avoid the issue, we need to make the sequences of equal length. We can achieve this by padding method. In the padding method we pass in the maximum sequence length as the max length.

```
data = pad_sequences(sequences, maxlen=maxlen)
data
```

```
array([[ 0,  0,  0, ...,  0,  4,  4],
       [ 0,  0,  0, ..., 19, 14, 17],
       [ 0,  0,  0, ...,  0,  0,  4],
       ...,
       [ 0,  0,  0, ...,  5,  5,  5],
       [ 0,  0,  0, ...,  0,  0,  5],
       [ 0,  0,  0, ...,  0,  5,  5]], dtype=int32)
```

Model Building:

As we padded all the sequences to the equal length, the model will now have to judge between length of event list + 1 number of events. Taking the embedding dimension as 20 we develop the embedding matrix with shape (eventlist+1, embedding dimension). We initialize the sequential model as model and then we add Embedding layer followed by the LSTM layer of 32 units. Then we add the model with the dense layers with different activations (such as

ReLU and sigmoid). We then compile the model with a loss function of binary cross entropy, optimizer adam and rmsprop. Then at last we fit the training set by assigning the validation with the valid data for different epochs and batch sizes.

We built 4 different models with different optimizers, layers, epoch cycles, batch sizes.

Model 1:

LSTM Units = 32, Optimizer: Adam, Epoch: 50, Batch size: 512, Trainable: False

Result:

For the model with these parameters we achieved training accuracy of 68.9% and loss of 57.8% similarly validation accuracy and loss were 68.9%, 58.2%

Looking at the accuracy and loss plots for training and validation there is no significant trend of convergence between them.

Model 2:

LSTM Units = 32, Optimizer: Adam, Epoch: 50, Batch size: 512, Trainable: True

Result:

For the model with these parameters we achieved training accuracy of 69.4% and loss of 57.2% similarly validation accuracy and loss were 68.9%, 58%

Looking at the accuracy and loss plots for training and validation the plot seems smoother than before. The loss plot trend of training and validation seem to diverge, and this results in overfitting after the 5^{th} epoch cycle.

Model 3

LSTM Units = 32, Dropout= 0,2, Optimizer: Adam, Epoch: 50, Batch size: 512

Result:

For the model with these parameters we achieved training accuracy of 69.3% and loss of 57.4% similarly validation accuracy and loss were 68.8%, 57.9%

Looking at the accuracy and loss plots for training and validation the plot seems smoother than before. The loss plot trend of training and validation shows that overfitting is reduced compared to the previous model and the diverge after 30 epoch cycles.

Model 4

LSTM Units = 32, Dropout= 0.5, recurrent dropout = 0.5, Optimizer: Adam, Epoch: 50, Batch size: 512

Result:

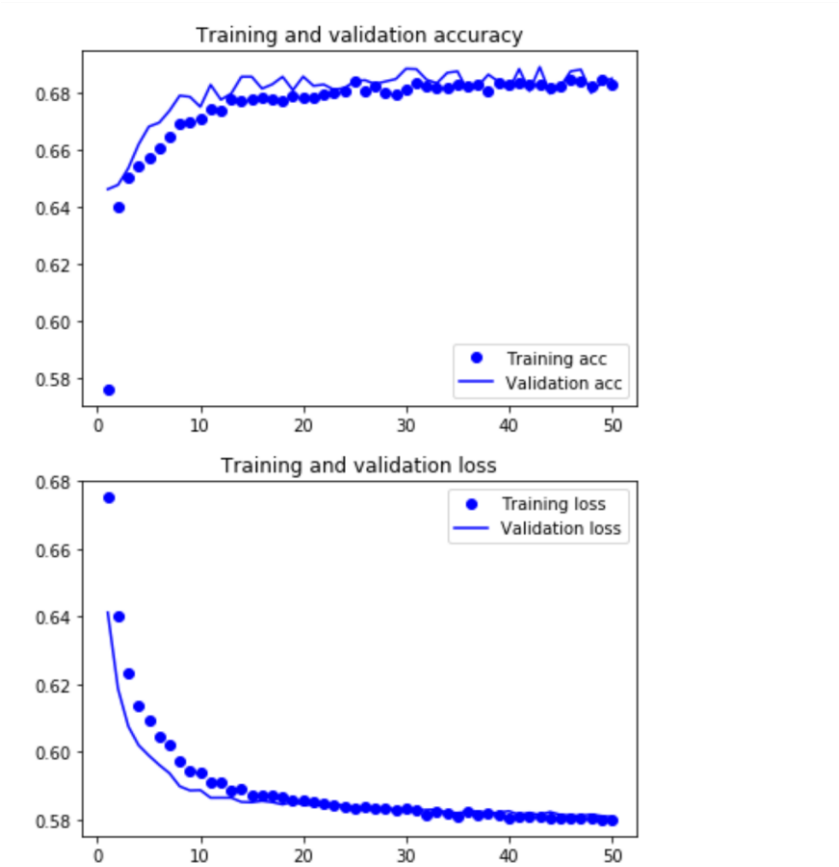
For the model with these parameters we achieved training accuracy of 68.2% and loss of 58.01% similarly validation accuracy and loss were 68.4%, 58%.

Looking at the accuracy and loss plots for training and validation the plot seems to converge and smoother. The loss plot trend of training and validation shows that there are no signs of overfitting.

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense, LSTM

units = 32

model = Sequential()
model.add(Embedding(num_events, embedding_dim))
model.add(LSTM(units))
model.add(Dense(1, activation='sigmoid'))
```



From all the above models we found that the model 4 has the best possible result because there are no traces of over fitting and the trends of training and Validation accuracy and losses seem to converge.

The accuracy, precision, recall and F score values for the above model are obtained as below:

```
[53]  # accuracy: (tp + tn) / (p + n)
      accuracy = accuracy_score(y_valid, yhat_classes)
      print('Accuracy: %f' % accuracy)
      # precision tp / (tp + fp)
      precision = precision_score(y_valid, yhat_classes)
      print('Precision: %f' % precision)
      # recall: tp / (tp + fn)
      recall = recall_score(y_valid, yhat_classes)
      print('Recall: %f' % recall)
      # f1: 2 tp / (2 tp + fp + fn)
      f1 = f1_score(y_valid, yhat_classes)
      print('F1 score: %f' % f1)
```

```
Accuracy: 0.688430
Precision: 0.713440
Recall: 0.613296
F1 score: 0.659589
```

Challenges and Lessons Learned:

Sequence Generation:

The major issue we faced during the sequence generation is getting the list of events which occurred on the same date on the same road within 30 minutes from the occurrence of an accident. Since the time given is a time stamp object, we had to research for a solution that gives us the timestamp object earlier by 30 minutes. And since the accidents occur due to the previous incidents on the same road. It makes sense to take into consideration of only the sequences on the same road in which an accident has occurred. It was a big learning curve for us to sort and apply confined loops to meet multiple conditions. The next challenge was to set up a radial filter since the accidents can never depend on the incident that has been recorded far away on the same road. So, we had to apply the haversine formula which we obtained from the online sources. Though, it seemed so easy to calculate the difference between a pair of latitudes and longitudes, it was a difficult task to extract all the latitudes and longitudes of the records within 30 minutes. So, for this, we had to use a data frame to store the latitudes and longitudes along with the event type and a loop to extract each latitude and longitude one by one.

Model Fine Tuning:

The most difficult part was to build a model without compromising on accuracy, rather omitting over fitting the model. For this we had to play around with the hyper parameters. And since there are no pre-defined parameter values for achieving the best model. The only available solution for this was to apply and test the model each time for every change in the hyper parameter. And since we are not much into neural networks and sequential modelling, we learned a lot about the ways of building a Sequential model based on the output required and the different optimisers that serve our purpose and the layers that needs to be added.

Future Study and Improvements:

It is always important that a model serves better with higher accuracy in order to serve the real-world issues. We wish we could build a model with a better accuracy and a promising model. We wish we had a lot more time to learn about Sequential Modelling and Neural Networks which would have helped us with the model building which could achieve higher accuracy and with minimal losses.

References and Links:

Tools Links:

Fast ai: https://www.fast.ai/

Keras: https://keras.io/

Pandas: https://pandas.pydata.org/

Numpy: https://www.numpy.org/

Matplotlib: https://matplotlib.org/

References

https://towardsdatascience.com/how-to-predict-severe-traffic-jams-with-python-and-recurrent-neural-networks-e53b6d411e8d

https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/