Virginia Tech
Blacksburg, VA, 24061
CS 4624: Multimedia, Hypertext, and Information Access
Spring 2020

Deep Learning Predicting Accidents

Final Report

Elias Gorine, Farnaz Khagani, Jacob Smethurst
{egorine, farnazk, jcsmeth}@vt.edu
Professor: Dr. Edward A. Fox

May 6, 2020

Table of Contents

Table of Figures……..3

Table of Tables……...4

Abstract / Executive Summary……...5

Introduction……..6

Requirements……..7

Accident Detection Model……..7

Accident Detection……...7

Design and Implementation……8

Traffic Data Collection……..……..8

Weather Data Collection……..…13

Data Organization and Preprocessing……..…16

Deep Learning Model…….22

Testing and Evaluation……...23

User Manual……..25

Git and File Structure…….……...25

Running the Project Code…….……...26

Interpreting Results…….……...27

Developer Manual……29

Database Setup…….……29

Data Collection…….……29

Database Schema…….……30

Processing with SQL……31

Database Extensibility……31

Database Indexing Strategy……33

Bidirectional Model Overview……...34

Bidirectional Model Extensions……...36

Project Methodology……..37

Lessons Learned / Conclusion……42

Project Timeline……..42

Problems Encountered……..42

Future Work……43

What We Learned……..……...…44

Acknowledgments……..……45

References……46

Table of Figures

Figure 1. PeMS homepage……...8

Figure 2. Example of traffic speed data……...9

Figure 3. Example of traffic incident data……..……11

Figure 4. MesoWest homepage……..……...……..…13

Figure 5. Example of weather data file……..……14

Figure 6. Map visual describing location scope of the project……...19

Figure 7. Python dictionary with combined data sets……...21

Figure 8. Incidents found in the database that match up to a captured anomaly……..23

Figure 9. Loading speed data using parameterized queries…….……...32

Figure 10. Threshold values……..35

Figure 11. The Bidirectional model capturing anomalous data points…….……..35

Figure 12. Workflow for Goal 1……38

Figure 13. Workflow incorporating Goal 2……39

Table of Tables

Table 1. Traffic speed data file column descriptions……..……..……....10

Table 2. Traffic incident data file column descriptions……..12

Table 3. Description of headers in weather data files……..……14

Table 4. Description of columns in weather data files……...15

Table 5. *Incidents_raw* schema……..……17

Table 6. *Speed_raw* schema……..17

Table 7. *Flow_raw* schema……18

Table 8. Final traffic and weather data schema……..……....20

Table 9. A 5 x 21 table of speed and time data……34

Abstract / Executive Summary

The Deep Learning Predicting Accidents project was completed during the Spring 2020 semester as part of the Computer Science capstone course CS 4624: Multimedia, Hypertext, and Information Access. The goal of the project was to create a deep learning model of highway traffic dynamics that lead to car crashes, and make predictions as to whether a car crash has occurred given a particular traffic scenario. The intended use of this project is to improve the management and response times of Emergency Medical Technicians so as to maximize the survivability of highway car crashes.

Predicting the occurrence of a highway car accident any significant length of time into the future is obviously not feasible, since the vast majority of crashes ultimately occur due to unpredictable human negligence and/or error. Therefore, we focused on identifying patterns in traffic speed, traffic flow, and weather that are conducive to the occurrence of car crashes, and using anomalies in these patterns to detect the occurrence of an accident.

This project's model relies on: traffic speed, which is the average speed of highway traffic at a certain location and time; traffic flow, which is a measure of total traffic volume at a certain location and time that takes into account speed and number of cars; and the weather at all of these locations and times. We train and evaluate using traffic incident data, which contains information about car crashes on all California interstate highways. This data is obtained from government sources.

The relevant data for this project is stored in a SQLite database, and both the code for data organization and preprocessing, as well as the deep learning model, are written in Python. The source code for the project is available at https://github.com/Elias222/DeepLearningPredictingAccidents.

Introduction

Reducing the response times of Emergency Medical Technicians to car crashes is key in increasing the survivability of the crash for those involved. According to studies, counties across the United States with a response time of more than 12 minutes had a motor vehicle collision mortality rate nearly twice that of counties with a response time of less than 7 minutes. [1][2]

For this reason, even a decrease in EMT response times on the order of fractions of a minute can prove life-saving, especially in serious collisions at highway speeds. If transportation officials in a state have any advanced notice or warning as to which areas of the state's interstate highways are most likely to have an accident at certain times of the day, a decrease in response time might be obtained.

This project seeks to find such relationships between traffic patterns and collisions based on data from California interstate highways. Obviously, one could expect more accidents to occur during rush hour periods on highways. This project seeks to use deep learning techniques to identify more granular patterns than "busy" and "not busy" periods, including and especially patterns that are not easily identifiable by human analysis of highway data sets.

Since this project is completely new and has not been carried over from a previous semester, the current work plan does not include any front-end interface for using the final product. Rather, we are seeking to begin exploring the use of a deep learning model to detect the occurrence of traffic accidents, and the creation of helpful visualizations that show which traffic factors make an accident more likely.

Requirements

The goals and stretch goals of this project were established during early meetings with the client. We established the main goal of the project to be an accident detection model, and the stretch goal to be an accident prediction codebase.

Accident Detection Model

The main goal for this project is an accident detection deep learning model. The relevant deliverables are the complete data set used in analysis, the codebase for the model, and the classifications made by the model. The presentation of these deliverables is open to interpretation and can be in whichever form best supports the goal of the project.

Accident Prediction

The stretch goal of the project was determined to be the creation of an accident prediction codebase. The main difference between this problem and the problem of accident detection is that detection relies on finding trends in data from before, during, and after an incident. Accident prediction on the other hand, only looks at data from before an incident. For this reason, an accident detection codebase represents a more straightforward classification problem, whereas an accident prediction codebase relies on much more nuanced patterns in a smaller set of input data.

Unfortunately, due to time constraints and delays in our progress largely caused by the university's transition to online learning in wake of the COVID-19 outbreak in the United States, we were unable to proceed with our stretch goal of an accident prediction codebase.

Design and Implementation

Our project design and implementation began with data collection, preprocessing, and organization. We then moved into integrating our dataset with our deep learning model.

Traffic Data Collection

Our traffic data collection begins with California's system of highway loop detectors. Loop detectors are electromagnetically conductive loops that are embedded in the highway. When cars and other vehicles pass over the detector, the undercarriage of the vehicle interacts with the detector to create a disturbance in its electromagnetic field. The detectors are able to classify the type of vehicle based on the profile of this disturbance. They report data such as the vehicle speed and the traffic flow at the detectors. In California, loop detectors are found on all interstate highways at an interval of about 0.1 miles, and collect data about every 5 minutes [3]. However, we found the available traffic data to have a resolution of about every 0.5 miles, which could be due in part to the loop detector system being incomplete or requiring maintenance.

Our project uses the collections of this loop detector traffic data found on the Freeway Performance Measurement System, or PeMS. As shown in Figure 1, PeMS is a publicly available online portal for accessing all types of data provided by the California Department of Transportation (Caltrans).

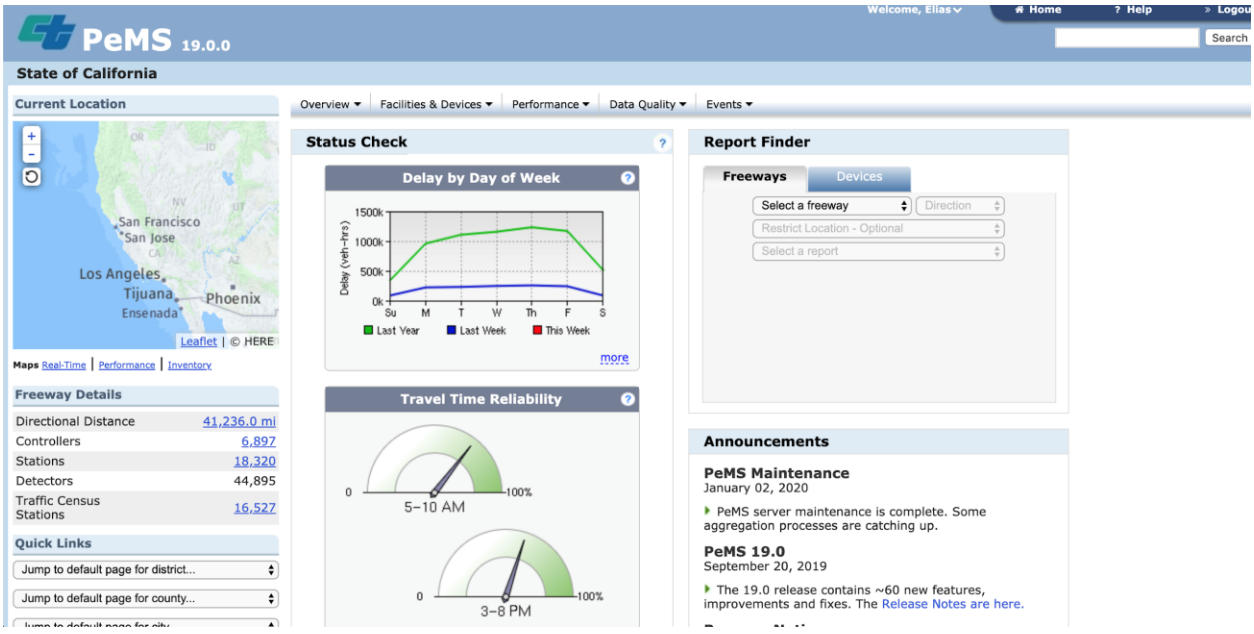

Figure 1. PeMS homepage, available at http://pems.dot.ca.gov/

PeMS provides Comma Separated Value (CSV) files of traffic speed, flow, and incident data.

Traffic speed data is grouped by day and interstate highway, and includes 7 pieces of data (see Figure 2) per loop detector reading: Time, Absolute Postmile, Caltrans Postmile, Vehicle Detector Station, Aggregate Speed, Number of Lane Points, and Percent Observed.

Time	Postmile (Abs)	Postmile (CA)	VDS	Agg Speed	# Lane Points	% Observed
00:00	56.481	R6.52	400573	67.7	3	100
00:00	53.831	3.87	400322	66.8	4	50
00:00	53.401	3.44	400575	69.2	4	100
00:00	53.071	3.11	401486	68.5	4	100
00:00	52.601	2.64	400676	68.8	4	100
00:00	52.011	2.05	400156	70.1	5	80
00:00	51.481	1.52	404618	69.7	4	100

Figure 2. Example of traffic speed data

A description of the schema of traffic speed data files is given in Table 1.

Column name	Description
Time	The Time value begins at 00:00, representing 12:00 AM, and continues throughout the day, using a 24 hour clock.
Absolute Postmile	The Absolute Postmile is a measure of the location of the reading, using the statewide mile markers for the particular highway.
Caltrans Postmile	The Caltrans Postmile is a more convoluted measure of the location of the reading; it measures the distance from the county line of the county in which the reading occurs.
VDS	The Vehicle Detector Station, or VDS, is the unique identifier of the station to which the loop detector belongs
Aggregate Speed	The Aggregate Speed value is a valuable piece of traffic data that we utilize; it is an average of the speeds of cars passing over the loop detector
Number of Lane Points	The Number of Lane Points, which is simply the width of the highway at this particular location.
Percent Observed	The Percent Observed is the percentage of lanes that recorded the presence of a vehicle. For instance, a point on a four-lane highway that has three cars side-by-side has a Percent Observed of 75%.

Table 1. Traffic speed data file column descriptions

For the purposes of this project, the most important pieces of data for each speed reading are the Time value, the Absolute Postmile, and the Aggregate Speed value.

Traffic flow data is organized in almost the exact same way, with the main difference being the inclusion of an Aggregate Flow value instead of an Aggregate Speed value. Similar to the speed readings, the most important pieces of data for each flow reading are the Time value, the Absolute Postmile, and the Aggregate Flow value.

The traffic incident data, also available on PeMS, has 10 pieces of data per entry (see Figure 3): Incident ID, Start Time, Duration, Freeway, Caltrans Postmile, Absolute Postmile, Source, Area, Location, and Description.

Incident Id	Start Time	Duration (mins)	Freeway	CA PM	Abs PM
18326292	11/01/18 09:11	416	I405-N	29.128	52.9
18326295	11/01/18 09:14	-2	I405-N	22.63	22.4
18326296	11/01/18 09:12	24	I405-N	22.63	22.4
18326424	11/01/18 10:16	22	I405-N	24.128	47.9
18326444	11/01/18 10:29	12	I405-N	24.128	47.9
18326445	11/01/18 10:28	20	I405-N	44.728	68.5
18326449	11/01/18 10:26	16	I405-N	23.73	23.5
18326491	11/01/18 10:51	47	I405-N	20.128	43.9
18326492	11/01/18 10:50	18	I405-N	20.128	43.9

Source	AREA	LOCATION	DESCRIPTION
CHP	West LA	I405 N / National Blvd	1182-Trfc Collision-No Inj
CHP	Orange County	I405 N / Seal Beach Blvd	1125-Traffic Hazard
CHP	Westminster	I405 N / Seal Beach Blvd	1125-Traffic Hazard
CHP	West LA	I405 N / La Tijera Blvd Ofr	1182-Trfc Collision-No Inj
CHP	LAFSP	I405 N / La Tijera Blvd Ofr	1182-Trfc Collision-No Inj
CHP	West Valley	I405 N / Nordhoff St	1125-Traffic Hazard
CHP	Orange County	I405 N / I405 N I605 N Con	1125-Traffic Hazard
CHP	LAFSP	I405 N / I405 N I105 Con	1179-Trfc Collision-1141Enrt
CHP	West LA	I405 N / I405 N I105 Con	1179-Trfc Collision-1141 Enrt

Figure 3. Example of traffic incident data

A description of the schema for traffic incident data files is given in Table 2.

Column name	Description
Incident ID	The Incident ID is a unique identifier for the incident.
Start Time	The Start Time is a 24-hour clock timestamp of the form MM/DD/YY HH:MM that represents when the incident occurred.
Duration	The Duration is a value in minutes that represents for how long the incident was occurring, as assessed by the police department reporting the incident.
Freeway	The Freeway is the name and travel direction of the interstate highway, for example, “I405-N”.
Caltrans Postmile	The Caltrans Postmile is a measure of the location of the incident and uses the same convention as discussed for the traffic speed and flow data, in Table 1.
Absolute Postmile	The Absolute Postmile is a measure of the location of the incident and uses the same convention as discussed for the traffic speed and flow data, in Table 1.
Source	The Source is typically “CHP”, which represents the California Highway Patrol.
Area	The Area is the county in which the incident took place.
Location	The Location includes the information from the Freeway value, as well as the nearest cross-street.
Description	The Description is a text description of the incident that also includes a four-digit incident code. For example, the entry for a traffic collision with no injuries is “1182-Trfc Collision-No Inj”.

Table 2. Traffic incident data file column descriptions

For the purposes of this project, the most important pieces of data for each incident report are the Start Time, Duration, Freeway, Absolute Postmile, and Description.

Our project makes use of raw traffic data files covering all of Interstate 5 North in California from July 1, 2018 through November 30, 2018.

Weather Data Collection

Unfortunately, the PeMS portal does not have any available weather data sets. To collect weather data, we used the University of Utah's MesoWest portal, which aggregates data from weather observation stations across the United States.

Figure 4. MesoWest homepage, available at https://mesowest.utah.edu/

MesoWest also provides Comma Separated Value (CSV) files of weather data on a per-station basis, for any length of time. The resolution of each weather reading is slightly more frequent than every five minutes. An example weather data file from University Airport outside Sacramento, California is pictured in Figure 5.

The provisional data available here are intended for diverse user applications.
For data requir review the information
available from the NCEI (https://www.ncdc.noaa.gov/customer-support/certification-data)
or consult a CCM (http://www.nicm.org).
STATION: KEDU
STATION NAME: University Airport
LATITUDE: 38.5315
LONGITUDE: -121.7865
ELEVATION [ft]: 68
STATE: CA

Station_ID	Date_Time	air_temp_set_1	wind_speed_set	precip_accum_o	cloud_layer_1_c	cloud_layer_2_c	visibility_set_1	weather_cond_c	cloud_layer_3_c	weather_condition_set_1d
		Celsius	m/s	Millimeters	code	code	Statute miles	code	code	Code
KEDU	2018-07-01T00:15:00Z	38	5.66		1		10			Clear
KEDU	2018-07-01T00:35:00Z	38	5.14		1		10			Clear
KEDU	2018-07-01T00:55:00Z	38	5.66		1		10			Clear
KEDU	2018-07-01T01:15:00Z	38	4.63		1		10			Clear
KEDU	2018-07-01T01:35:00Z	38	3.6		1		10			Clear
KEDU	2018-07-01T01:55:00Z	37	3.09		1		10			Clear
KEDU	2018-07-01T02:15:00Z	37	2.57		1		10			Clear
KEDU	2018-07-01T02:35:00Z	37	2.06		1		10			Clear
KEDU	2018-07-01T02:55:00Z	35	0		1		10			Clear
KEDU	2018-07-01T03:15:00Z	33	1.54		1		10			Clear
KEDU	2018-07-01T03:35:00Z	32	2.06		1		10			Clear
KEDU	2018-07-01T03:55:00Z	30	3.09		1		10			Clear
KEDU	2018-07-01T04:15:00Z	29	3.09		1		10			Clear
KEDU	2018-07-01T04:35:00Z	28	2.57		1		10			Clear
KEDU	2018-07-01T04:55:00Z	27	4.12		1		10			Clear

Figure 5. Example of weather data file

Each weather data file contains 6 informational items at the top of the file. These are described in Table 3.

Header name	**Description**
STATION	This is the official call sign or identifier for the weather station that recorded the data in the file. The weather stations we surveyed were most frequently located at airports.
STATION NAME	This is the common name of the weather station.
LATITUDE	This is the numerical latitude location of the weather station.
LONGITUDE	This is the numerical longitude location of the weather station.
ELEVATION	This is the elevation above sea level of the weather station, measured in feet.
STATE	This is the state containing the weather station. For the purposes of this project, we only explored weather stations located in California.

Table 3. Description of headers in weather data files

Beneath each file's headers are the weather readings for each point in time. The 11 columns per weather reading are described in Table 4.

Column name	Description
Station_ID	This is a duplicate of the STATION field contained in the headers at the top of each data file.
Date_Time	This is a timestamp of the form YYYY-MM-DD HH:MM:SS that gives the time of the reading.
Air_temp_set_1	Given in degrees Celsius, this is the air temperature recorded at the station.
Wind_speed_set_1	Given in meters per second, this is the wind speed recorded at the station. Note that there is no wind direction given, only speed.
Precip_accum_one_hour_set_1	Given in millimeters, this is the amount of precipitation that has accumulated at the station in the hour leading up to the time of the reading.
Cloud_layer_1_code_set_1	This is a code value describing the first cloud layer in the area. This field is more likely to have a value on cloudy days.
Cloud_layer_2_code_set_1	This is a code value describing the second cloud layer in the area.
Visibility_set_1	Given in miles, this is the visibility length from the weather station.
Weather_cond_code_set_1	This is a code value describing the weather conditions.
Cloud_layer_3_code_set_1	This is a code value describing the third cloud layer in the area. This field is more likely to have a value on heavily clouded days.
Weather_condition_set_1	This is a text description of what the weather is like at the given time. Some examples of values for weather_condition_set_1 are “Clear,” “Partly Cloudy,” and “Smoke”

Table 4. Description of columns in weather data files

Data Organization and Preprocessing

After we had collected traffic data for Interstate 5 North over the period of July 1, 2018 through November 30, 2018, we began implementing Python modules to parse the raw data files and load them into a SQLite3 database. We accomplished this by creating one Python script to parse and load each traffic data type: speed, flow, and incidents. *Speed_reader.py* gives an overview of our parsing process. We begin by defining a "Speed" class that contains fields corresponding to each column of a traffic speed reading, as given in Table 1. Then, for each file in our speed data file directory, we open the file and begin reading data in line by line. For each line in each file, we use the spacing of the columns to isolate each data point and extract its value. Since the timestamp in traffic speed and traffic flow files is given as HH:MM, we use the naming convention of the files to our advantage. To create a timestamp, we combine the date contained in the file name with the time value extracted from the file itself to create a timestamp of the form MM-DD-YYYY HH:MM. Once we have extracted each data point from the current line in the file, we instantiate a new "Speed" object with fields corresponding to each parsed data point. We append each line's Speed object to a list.

The process for parsing data out of traffic flow and traffic incident files is very similar. These processes occur in *flow_reader.py* and *incident_reader.py*. In fact, the process for building an incident timestamp is even easier, since the text timestamps in each file already contain the date of the incident.

The three data loader scripts are all called from an overarching script, *db_loader.py*. Once all three scripts have been run, *db_loader.py* has access to lists containing Python objects of each data point from every data file available. It then uses SQLite3's Python library to create and populate the tables *incidents_raw*, *speed_raw,* and *flow_raw.*

The schema of *incidents_raw* is given in Table 5.

Column name	Data type
incident_id	integer
time	timestamp
duration	integer
freeway	text
ca_pm	text
abs_pm	real
source	text
area	text
location	text
description	text

Table 5. *Incidents_raw* schema

The schema of *speed_raw* is given in Table 6.

Column name	Data type
time	timestamp
pm_abs	real
pm_ca	text
vds	integer
agg_speed	real
lane_points	integer
pct_obs	real

Table 6. *Speed_raw* schema

The schema of *flow_raw* is given in Table 7.

Column name	Data type
time	timestamp
pm_abs	real
pm_ca	text
vds	integer
agg_flow	real
lane_points	integer
pct_obs	real

Table 7. *Flow_raw* schema

After inserting each data reading into the appropriate table, the final size of *raw_data.db* is roughly 827 MB. Using our preliminary database loader scripts to load our full traffic data sets took roughly 8 hours, and did not yet include weather data. As the scope of our application began to take shape and we settled on a final desired schema for our traffic and weather data, we reconstructed our database loader into a new script, *DataLoader.py*.

The first step to constructing a new database loader was finalizing the time and location scope for the data we wished to analyze. Though we had access to data from all of Interstate 5 North, we chose to narrow down our dataset to only included readings between absolute postmiles 504.223 and 520.744. There are 30 loop detector stations contained in this span. The full list of absolute postmile locations for these stations is [504.223, 504.793, 506.383, 507.504, 507.953, 508.463, 509.013, 510.094, 510.293, 510.643, 511.341, 511.543, 512.073, 512.435, 512.753, 513.503, 513.998, 514.662, 515.173, 515.973, 516.593, 517.093, 517.916, 518.543, 518.864, 519.193, 519.571, 519.863, 519.874, 520.744]. Since we had yet to collect weather data, we decided to source our weather data from Sacramento Executive Airport. The location of this weather data collection station in relation to our chosen stretch of Interstate 5 North is shown in Figure 6.

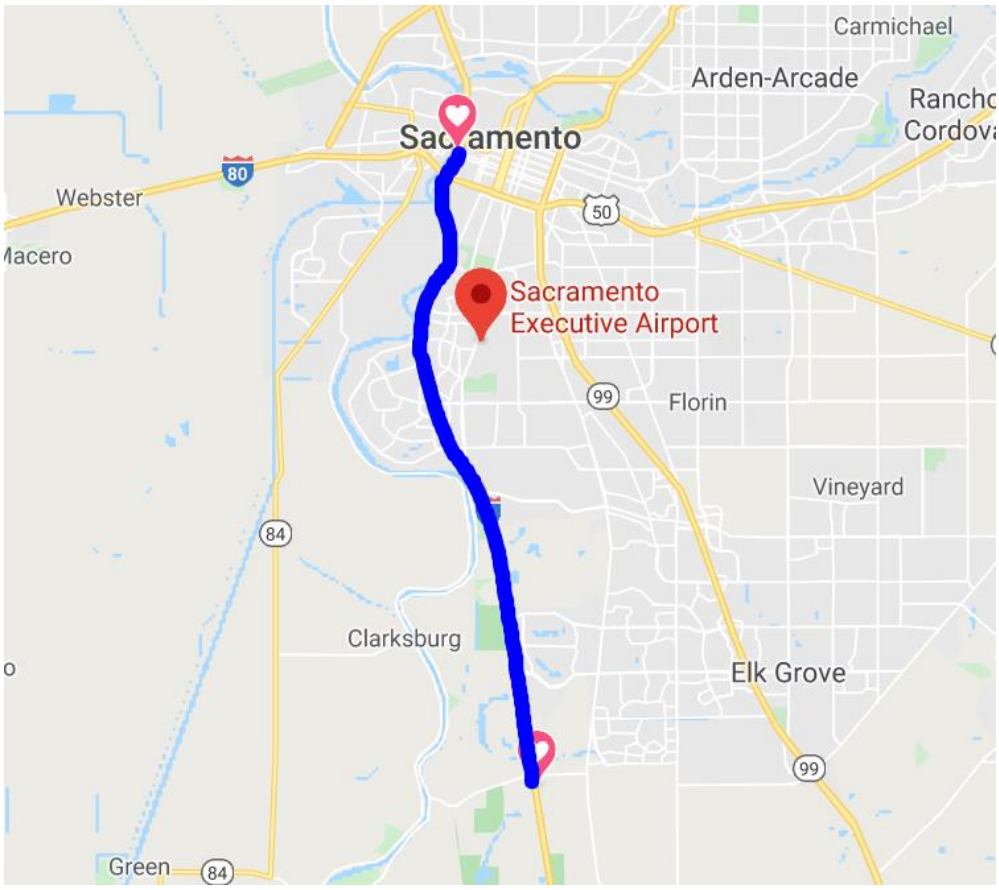

Figure 6. Map visual describing location scope of the project

The time scope of our project remained the same. We are considering data from July 1, 2018 through November 30, 2018.

The most major change to our data loading process once we selected the location scope for the project was to the schema of the final database. Since we had a predetermined list of absolute postmile locations, we could match up traffic speed, traffic flow, and weather data readings at each location in 5 minute intervals. For traffic speed data, we chose to ignore many of the irrelevant fields in the raw data files and focus only on aggregate speed. We did the same for flow, only considering aggregate flow, and for weather, where we chose to consider temperature, wind speed, and hourly precipitation accumulation. The weather data points at a given time are duplicated for all absolute postmile station locations. Our final schema includes a column for each of these values at each absolute postmile value in our list, as well as a timestamp for each set of readings. The final schema is shown in Table 8.

Column name	Data type	Description
time	timestamp	This is the YYYY-MM-DD HH:MM timestamp for the readings contained in every other column of this row.
s504223	real	This represents the traffic speed reading at absolute postmile 504.223.
f504223	real	This represents the traffic flow reading at absolute postmile 504.223.
t504223	integer	This represents the temperature reading at absolute postmile 504.223.
w504223	real	This represents the wind speed reading at absolute postmile 504.223.
p504223	real	This represents the hourly precipitation accumulation reading at absolute postmile 504.223.
...	...	...
...	...	For each other absolute postmile location, the schema contains sets of speed, flow, temperature, wind, and precipitation columns.
...	...	...
s520744	real	This represents the traffic speed reading at absolute postmile 520.744.
f520744	real	This represents the traffic flow reading at absolute postmile 520.744.
t520744	integer	This represents the temperature reading at absolute postmile 520.744.
w520744	real	This represents the wind speed reading at absolute postmile 520.744.
p520744	real	This represents the hourly precipitation accumulation reading at absolute postmile 520.744.

Table 8. Final traffic and weather data schema

With a speed, flow, temperature, wind, and precipitation column for each of the 30 traffic data collection stations, plus a timestamp column, the final schema has 151 columns.

The new database loader to support this new schema is implemented in *DataLoader.py*. The script begins by reading through one file of speed data, which contains data for a single day. For each line of the file, it adds an entry to a Python dictionary that maps the HH:MM timestamp of the line to the absolute postmile of the line, and then to the aggregate speed reading of the line. The same procedure then occurs with the traffic flow data file and weather data file for that particular day. After the weather data is added into the data dictionary for a given day, the dictionary has the form shown in Figure 7 at each point in time.

```
1710: {
    speed: 66.1,
    flow: 271,
    temp: 20,
    wind: 2.57,
    precip: 0
},
```

Figure 7. Python dictionary with combined data sets

Figure 7 shows the data for an arbitrary day at time 1710, or 5:10 PM. By using and accessing Python dictionaries, the script is able to match up the relevant data points at each moment in time. Now, the data from the current day can be inserted into the new database. This process repeats for each day's data files until the database is populated with all the proper data points at each point in time, according to the schema given in Table 8.

DataLoader.py also helps to ensure data quality in the case of missing or invalid data points. If a speed or flow data point is missing at time *t*, then it is replaced by the value at the next closest downstream traffic data collection station at time *t.* If a weather data point is missing at time *t*, then it is replaced by the weather data point at time *t + 5 minutes.* Replacing missing data points ensures that the deep learning model does not produce errors or inaccuracies due to null or invalid data points.

By ignoring irrelevant data fields and focusing only on the relevant absolute postmile locations along Interstate 5 North, *DataLoader.py* has better time performance than the previous data loader scripts, and produces a database, *combined.db* that is only 34.9 MB.

Once *combined.db* is built, exporting the combined data to a CSV file for use by the deep learning model is trivial. The procedure for exporting the combined data is described in the User Manual.

Deep Learning Model

Before we discuss the model used in this project, let us first introduce the concept of Long Short Term Memory networks (LSTMs). LSTMs are a type of Recurrent Neural Network (RNN) which are an effective solution to sequence prediction problems. [4] LSTMs are able to selectively remember patterns for long durations. Additionally, an LSTM model was successfully used in a 2018 application that predicts the number of traffic incidents in different regions of the state of Iowa [5].

Our project utilizes a bidirectional LSTM which is an extension of the traditional LSTM. The Bidirectional model is a type of recurrent neural network, which essentially puts together two independent neural networks (RNNs). The same input sequence is fed to one RNN in a normal time-order, and to another RNN in reverse time order and at each step the outputs of both RNNs are concatenated.

“[Bidirectional LSTMs] can provide additional context to the network and result in faster and even fuller learning on the problem.”

-Jason Brownlee PhD.[4]

Our bidirectional model uses Keras, a high-level neural network Python API along with a TensorFlow backend. TensorFlow allows for building and training neural networks to detect patterns and correlations. The model also uses Pandas for data manipulation and Pylab/Matplotlib for plotting data points.

The entire model is presented with Jupyter Notebook and can be found within the repository in *Bidirectional model.ipynb* along with detailed code, outputs and explanations.

Testing and Evaluation

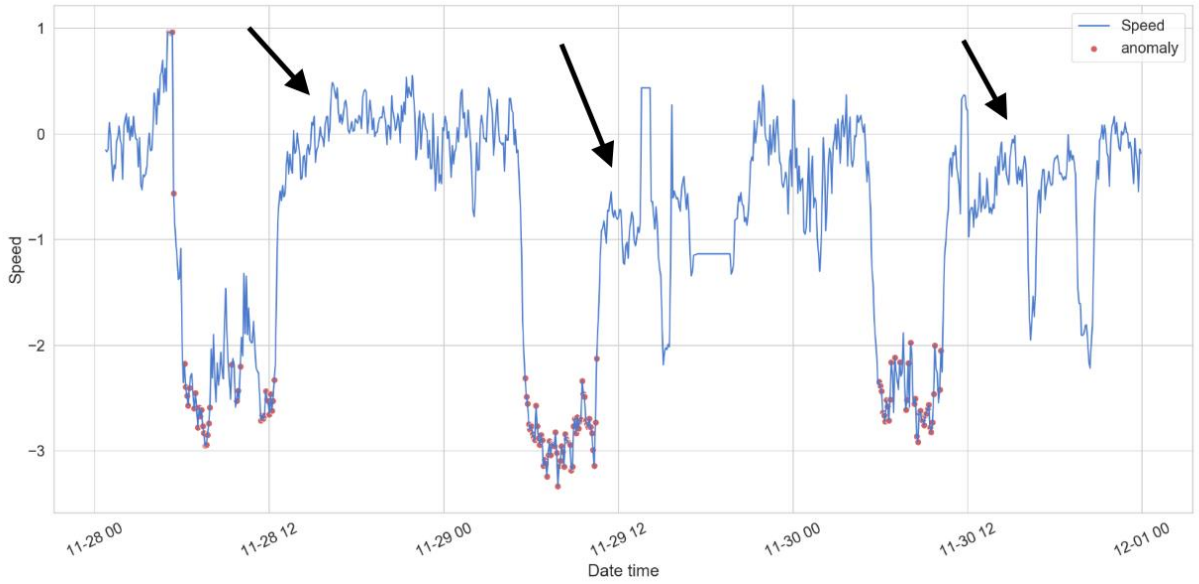

Figure 8. The Bidirectional model capturing anomalous data points

Figure 8 illustrates how the Bidirectional model captures anomalies from a data sequence. Each anomaly captured by the model is indicated by a red dot. As these captured points are grouped together, each trough in the graph can be considered as a single anomaly. The black arrows on Figure 8 indicate locations in which the database contains incidents responsible for the captured anomalies reported at later times.

To numerically interpret the results, two helpful metrics can be used: Detection Rate and False Alarm Rate.

Detection Rate represents the percentage of real incidents that were successfully detected. For example, if 4 incidents occur at a particular postmile location, yet the model detects 3 anomalies that match up to these incidents, a Detection Rate of 75% has been achieved.

A model that produces good results will maximize the Detection Rate. In a sample set of performance results for postmiles 516.593, 515.173, 513.503, 511.543, 510.293, and 508.463, we saw a maximum Detection Rate of 87.0% at postmile 513.503, and a minimum Detection Rate of 65.0% at postmile 511.543. A low detection rate at a particular location can be due in part to the fact that some accidents do not result in major traffic delays. Whether serious or not, we can intuitively assume that an incident that does not block travel lanes will result in less delays, and therefore will be less likely to be detected.

False Alarm Rate represents the ratio of incorrect model-detected incidents to the total number of model-detected incidents. For example, if the model detects 10 anomalies and claims them as incidents, but one of the detected incidents was a false alarm and never actually occurred, a False Alarm Rate of 10% has been achieved.

A model that produces good results will minimize the False Alarm Rate. In our sample set of performance results, we saw a minimum False Alarm Rate of 13.6% at postmile 516.593, and a maximum False Alarm Rate of 29.9%, at postmile 513.503.

Higher False Alarm Rates can be explained much like lower Detection Rates, by considering the different types of incidents. The traffic incident data, given in Table 2, includes incidents beyond standard traffic accidents. For example, Animal Hazards and general Traffic Hazards are also included as possible incident descriptions. Since the goal of the project is to detect standard traffic incidents, we excluded these extra incident types. Unfortunately, it is likely that some of the non-accident occurrences did cause delays that were detected by the model and classified as false-alarm incidents. For example, an animal crossing would be defined as an Animal Hazard, which we excluded, but could very well force drivers to reduce their speed greatly and cause a slowdown.

In the future, maximizing the model's Detection Rate and minimizing its False Alarm Rate will depend greatly on the types of incidents included, as well as the threshold value that controls which anomalies ultimately get detected. The effects of changing the threshold value are discussed further in the Developer Manual.

User Manual

Git and File Structure

The first step to running the project is forking and cloning the code repository, found at https://github.com/Elias222/DeepLearningPredictingAccidents. Access to the repository is available upon request. The project requires SQLite3, which should be installed by default on MacOS. On GNU Linux, it is readily available for installation through any standard package manager. It also requires Python 3 or later and Jupyter Notebook.

The repository includes the relevant project code as well as all the data used to train the model. For this reason, cloning into the repository could take quite some time, depending on your network connection. For help with the basics of using Git to fork or clone a Git repository, please use the official Git documentation, found at https://git-scm.com/docs.

The repository has some notable files in the following structure:

- Bidirectional model.ipynb
- DataLoader.py
- combined.db
- combined.csv
- data (directory)
    - flows (directory)
    - incidents (directory)
    - speeds (directory)
    - weather (directory)

Bidirectional model.ipynb is the Jupyter Notebook that is responsible for training the deep learning model and generating results.

combined.db and combined.csv both contain the full dataset used for training the model. The current data contained in these files comes from all the traffic and weather data files contained in the data/flows, data/speeds, and data/weather directories.

Finally, DataLoader.py is the script responsible for aggregating data from these raw data files into combined.db. When new data is added to the project, the first step to running the project is deleting combined.db and combined.csv and rebuilding the aggregated data set using this script.

Running the Project Code

To run the script, execute python DataLoader.py from the command line. Successful script execution should take ten to twenty minutes depending on your hardware and produce the following output in the terminal:

```
~ $ python DataLoader.py

Database created and successfully connected to SQLite

Creating table: combined_data

~ $
```

Running the script creates combined.db, which contains the full, unsorted dataset. To produce combined.csv for use in training and evaluating the model, some basic manipulation of the dataset using SQL is required. First, launch the SQLite command line interface:

```
~ $ sqlite3
```

Then, use the following commands to open combined.db and export the sorted contents to combined.csv:

```
.open combined.db

.headers on

.mode csv

.output combined.csv

SELECT * from combined_data ORDER BY time;

.quit
```

Exporting the sorted data should take less than a minute. After completion, it is advisable to briefly browse combined.csv to ensure there are no major issues with data quality. At this point, Bidirectional model.ipynb can be run using your preferred Jupyter Notebook manager.

Interpreting Results

To view a full explanation of the bidirectional model, be sure to run the *Bidirectional model.ipynb* Jupyter Notebook file. If you have not run Jupyter Notebook before, we recommend Installing Jupyter Notebook with Anaconda. A number of Python libraries are required for full functionality. You will need the following Python modules installed

and preferably Python version 3.3 or higher for the installation of Jupyter Notebook and the Python libraries.

- numpy
- tensorflow
- pandas
- seaborn
- pylab
- matplotlib
- Sklearn.externals

To detect anomalies, run the *Bidirectional model.ipynb* and view the generated graph at the bottom of the notebook file. It should be similar to Figure 8.

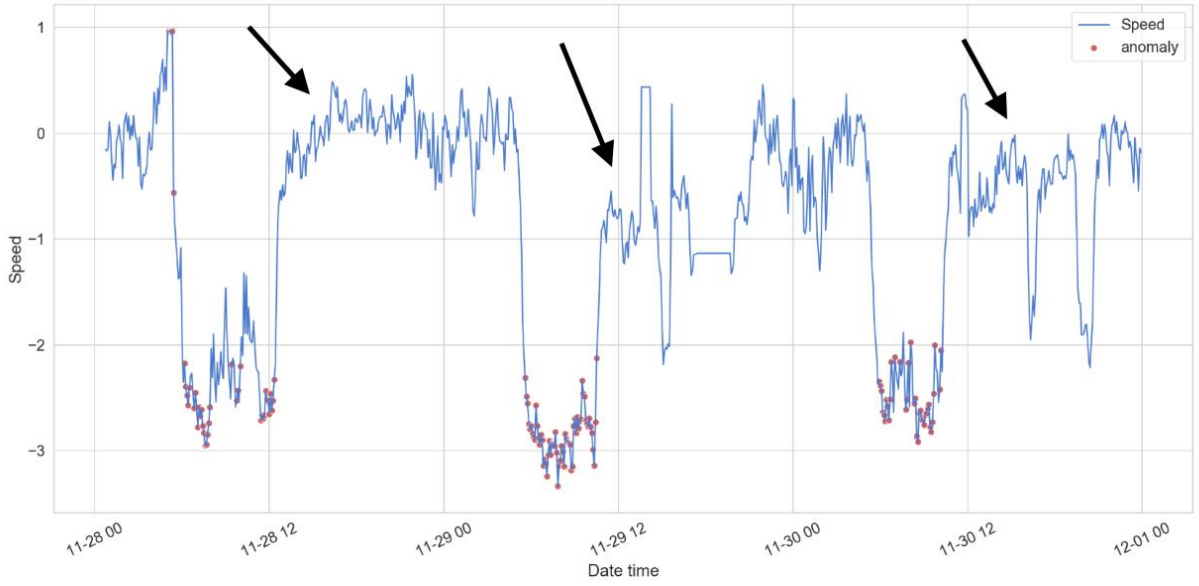

Figure 8. The Bidirectional model capturing anomalous data points

The black arrows in this image will not be generated. They are drawn here for your reference to mark the locations of reported incidents in the database. Upon generating a model you will notice a number of anomalies captured by the model, as indicated by the red dots. You can consider each trough in the graph as a single anomaly. To determine what sort of incident caused the anomaly to be captured, note the timestamp of the anomaly and search the database that you loaded in the previous step. The database should already be indexed by location (absolute postmiles) and timestamp, so search queries should execute quickly.

Be sure that you search for the correct highway, location, and timestamp. From there, you will manually determine which reported incident is likely to be the cause of the anomaly. Consider that the reported incident will occur *after* the time of the captured

anomaly. Also note that the incident can be reported hours after the anomaly. However, given the ratio of feature data to incident data, it should not be too difficult to determine which incident was responsible for each anomaly, as there will not be a large number of incidents within the same time and location to choose from. The version available in the repository for execution is designed to capture only the most severe incidents such as car crashes, which will make it easier to find the offending incident for each anomaly. If you are interested in capturing less extreme incidents than car accidents, we discuss how to alter this in the *Developer Manual* section.

Developer Manual

Database Setup

Our data is collected and inserted into a database by way of Python modules that use the aggregated .txt data files as inputs. Python files *flow_reader.py*, *incident_reader.py, speed_reader.py,* and *weather_reader.py* are used to read data from the files and parse it. Each incident, flow data point, weather data point, etc., is converted to separate Python classes which are then loaded as tuples into a SQLite database by another Python module *db_loader.py*.

The SQLite database file *raw_data.db* aggregates a number of raw data tables. i.e., *incidents_raw, flow_raw*, etc. From this point, SQL queries are executed to join tables in meaningful ways. For example, columns can be joined on time stamps, which are available as Python datetime objects in each tuple, or by location -- or an inner join can be done using both metrics. The new tables can be grouped by columns which indicate the type of incident, the weather patterns, or any number of factors that could be deemed important to the model.

Data Collection

All data to be loaded into the SQLite database comes from publicly available, verified sources. We picked sources which would account for the highest levels of data integrity, data validity and source trustworthiness. All data comes from government or educational institutions which have made it publicly available, requiring no more than a user account in most cases.

For our purposes, we downloaded all of the required data in .txt form, and used Python's CSV library to parse the data before loading it into the database.

In-depth discussion of these sources is found in the section titled "Design and Implementation" under the heading "Data Collection and Organization". This section will outline the primary sources of data from which our schema is devised.

PeMs — CalTrans Performance Measurement System
Provided by the California Department of Transportation, PeMs offers downloadable data in .csv and .txt forms. We used PeMs to gather data from California's vast array of traffic loop detectors which operate on the interstate system. For our project PeMs provided data on speed and flow as well as locations and timestamps for each reading.

CAHP — California Highway Patrol
From the California Highway Patrol we obtained data on reported incidents. This data also came in .txt form, similar to the PeMs data. The CAHP provides incident report files, with each file representing a month's worth of reported traffic incidents on the interstates. These files included incident ids, locations, descriptions and timestamps. For a more detailed look at the format and fields of the data see the Traffic Data Collection subsection of the Design and Implementation section**.**

MesoWest
Provided by the University of Utah, MesoWest archives current and past weather observations from around the country. We used MesoWest to obtain archived weather data to add to the database. Our project currently includes temperature, wind and precipitation data, but this can easily be extended to include other types of weather data if we see fit.

Database Schema

The database schema is constructed so that for every five minute interval between 12 AM on July 1, 2018 and 11:55 PM on November 30, 2018, there is data for traffic speed, traffic flow, and weather (air temperature, wind speed, and precipitation) at each of the 30 data collection stations from mile marker 504 and mile marker 520 on California's Interstate 5 North. The full list of locations for the traffic data collection stations is [504.223, 504.793, 506.383, 507.504, 507.953, 508.463, 509.013, 510.094, 510.293, 510.643, 511.341, 511.543, 512.073, 512.435, 512.753, 513.503, 513.998, 514.662, 515.173, 515.973, 516.593, 517.093, 517.916, 518.543, 518.864, 519.193, 519.571, 519.863, 519.874, 520.744].

The first column in the database schema is a timestamp in the form MM/DD/YYYY HH:MM. Then, for each of the listed data collection station locations (x), there is a column for each of the data types in the form *sx, fx, tx, wx,* and *px*. Column *sx* represents the traffic speed at station location *x*, column *fx* represents the traffic flow at station location *x*, and so on with *t* = air temperature, *w* = wind speed, and *p* = precipitation. With five columns per data collection station, plus the timestamp column, the final schema has a total of 151 columns:

time,s504223,f504223,t504223,w504223,p504223,s504793,f504793,t504793,w504793,p504793,s506383,f506383,t506383,w506383,p506383,s507504,f507504,t507504,w507504,p507504,s507953,f507953,t507953,w507953,p507953,s508463,f508463,t508463,w508463,p508463,s509013,f509013,t509013,w509013,p509013,s510094,f510094,t510094,w510094,p510094,s510293,f510293,t510293,w510293,p510293,s510643,f510643,t510643,w510643,p510643,s511341,f511341,t511341,w511341,p511341,s511

543,f511543,t511543,w511543,p511543,s512073,f512073,t512073,w512073,p512073,s512435,f512435,t512435,w512435,p512435,s512753,f512753,t512753,w512753,p512753,s513503,f513503,t513503,w513503,p513503,s513998,f513998,t513998,w513998,p513998,s514662,f514662,t514662,w514662,p514662,s515173,f515173,t515173,w515173,p515173,s515973,f515973,t515973,w515973,p515973,s516593,f516593,t516593,w516593,p516593,s517093,f517093,t517093,w517093,p517093,s517916,f517916,t517916,w517916,p517916,s518543,f518543,t518543,w518543,p518543,s518864,f518864,t518864,w518864,p518864,s519193,f519193,t519193,w519193,p519193,s519571,f519571,t519571,w519571,p519571,s519863,f519863,t519863,w519863,p519863,s519874,f519874,t519874,w519874,p519874,s520744,f520744,t520744,w520744,p520744

Processing with SQL

SQL statements are used to filter the large amount of data we have to work with in this project. When training our model, we worked with subsets of the data, primarily training on data collected from a stretch of I-5N, spanning from the southernmost suburbs of Sacramento to downtown. This data can be found in the repository in a new database, *combined.db,* or you can view it in CSV format as *combined.csv*. The absolute postmiles (i.e. location information) of sensor and incident data falls into the range 504.223 - 520.744. Since all tables include postmile data, and this data is simply stored as *real* (floating point) values, it can easily be searched with SQL. Once appropriate views are made, the data can be easily exported in CSV format for digestion by the bidirectional model.

Considering that future users of this project may wish to use only parts of the data they collect (either for modeling or training), writing SQL queries is an effective way of filtering out extraneous data, searching for relevant data or combining data sets into new tables.

Database Extensibility

The *flow_reader.py*, *incident_reader.py*, and *speed_reader.py* scripts are made to aggregate every value in the raw data files into a SQLite database. These scripts take longer to execute than *DataLoader.py*, which organizes data specifically in the schema used by our model, but they conveniently aggregate *all* the data so that it can be manipulated for any desired application. That is to say, *DataLoader.py* is less extensible and is built directly for our application, whereas the *flow_reader.py*, *incident_reader.py*, and *speed_reader.py* scripts are more extensible, but perform more slowly and parse possibly unneeded data from the raw files.

We attempted to make our database population scripts modular to promote extensibility for future developers. We recognize that successful accident detection models may include many more factors than our code does, such as the vertical gradient of the road, a measure of the road degradation, or a measure of the tightness of a curve in the road. If a developer is seeking to add a new factor to the model, they should first find a high-quality source for the data and ensure they have a way of finding the location and time of each data point in the form used by our project.

To create a new database reader module, a developer can follow the example of *flow_reader.py*, *incident_reader.py*, or *speed_reader.py*. These routines create a class with fields corresponding to the different columns in the CSV data file. They then use Python's CSV reader capabilities to parse the files and populate an array of the new class objects. Depending on the data used by the developer, there may be additional work required to create a timestamp in the same format as the rest of the data entries.

Then, *db_loader.py* can be easily adapted to load this list of Python objects into the SQLite3 database.

All insert and update queries to the database that are managed by the *db_loader.py* script are protected by parameterized queries. The insert queries in *db_loader.py* are prepared within prepared statements. Then, as each datapoint is read out of its associated data point list, the information is broken down by column and each column is loaded into the prepared statement as a dynamic parameter.

```
for data_point in speed_list:
    c.execute('INSERT INTO speed_raw VALUES (?,?,?,?,?,?,?)',
              (data_point.time, data_point.pm_abs, data_point.pm_ca, data_point.vds, data_point.agg_speed,
               data_point.lane_points, data_point.pct_obs))
    conn.commit()
```

Figure 9. Loading speed data using parameterized queries

In part, this decision was made given the fact that it is not possible to verify the integrity of *all* of the data that is being loaded into the database. Rather than verifying that each datapoint contains no hidden SQL queries within them, we instead chose to use prepared statements and dynamic parameters (bind variables). The primary goal was to prevent SQL injection attacks which may come from unverified data sources. Although all of the data sources for this iteration of the project are typically government-based and trustworthy - should the project be extended, future developers may attempt to pull data from various sources. These sources may be compromised, and if a malicious SQL statement ends up within the source-data, unpredictable outcomes are possible. For this reason, we highly recommend that you follow the same practices in the provided source code and utilize parameterized queries and bind variables.

Another issue to consider is performance. Consider that it is in the nature of the *db_loader.py* script to load hundreds of thousands - or millions - of records into the DB. SQLite, like other databases, builds execution plans to determine the best strategy to execute a query. An execution plan can be stored in an execution plan cache. However, this only works if the SQL statement to be executed is the exact same every time - this is not the case with our loads since the data parameters vary. For this reason, the database would handle each insert operation by building a whole new execution plan. This is certainly not ideal as it wastes a lot of time and performance suffers as a result.

By using bind variables, the actual values of our data points are not being written - instead, the bind variables act as placeholders within the prepared statement. This means that the SQL statements will not change and the same execution plan can be reused, thus improving performance. Considering the volume of data that can potentially be loaded for extensions of this project, it is recommended to follow the aforementioned practices to ensure that the performance of the database does not suffer in the build phase.

Database Indexing Strategy

Indexes are added to the frequently accessed columns and tables of our database. An index is a data structure which helps improve performance of queries. By default, SQLite uses B-trees (balanced trees, *not* binary trees) to organize indexes. The general rule we followed was adding indexes to the most frequently searched and accessed columns of each table. While it is not necessary, we recommend that any tables added as extensions to the project have indexes added where appropriate. If the table has columns that are frequently searched or joined upon, then an index may be added to improve performance.

Keep in mind that indexes have some overhead themselves. They occupy space on the disk and in the DB memory itself. Moreover, with each update/insertion/deletion, the index will also have to be updated. Having too many indexes, or indexes on unnecessary columns, can introduce performance issues. For these reasons, we added indexes to columns of tables which would be joined together or searched frequently. For example, if we needed to join the *flow_raw* and *speed_raw* tables using the pm_abs (absolute postmiles) columns, it would be wise to create indexes on these columns and tables. In SQLite this is quite easy:

```
CREATE INDEX postmiles_index_flow ON flow_raw(pm_abs);

CREATE INDEX postmiles_index_speed ON speed_raw(pm_abs);
```

Bidirectional Model Overview

Preprocessing

First data is loaded into the Pandas dataframe. Then a short feature engineering step is accomplished to break down the timestamps of each data point into minutes, hours, days of the week, and months. Before the model can be utilized features are scaled and sequences are created. Each sequence contains 10 data steps from the history. Notice that for each time stamp there are a number of data points recorded along a section of highway mile markers.

	Speed	116.151	116.601	116.833	117.343	117.743	118.193	118.663	119.433	119.913	...	121.213
Date time												
2018-11-01 00:00:00	0.756303	0.490439	0.386792	0.133333	0.214286	0.238095	0.238095	0.214286	0.238095	0.311475	...	0.081883
2018-11-01 00:05:00	0.890756	0.242970	0.278302	0.200000	0.238095	0.238095	0.238095	0.238095	0.238095	0.344262	...	0.040942
2018-11-01 00:10:00	1.100840	0.206974	0.212264	0.233333	0.238095	0.261905	0.261905	0.238095	0.261905	0.327869	...	0.069601
2018-11-01 00:15:00	0.983193	0.269966	0.259434	0.200000	0.190476	0.214286	0.214286	0.190476	0.214286	0.327869	...	0.085977
2018-11-01 00:20:00	0.815126	0.094488	0.132075	0.166667	0.190476	0.214286	0.214286	0.190476	0.214286	0.295082	...	0.077789

Table 9. A 5 x 21 table of speed and time data

Choosing a Threshold

The threshold is a constant value we pick that determines what the model will consider to be an anomaly. In our case, setting a higher threshold means that the model will capture more extreme events as anomalies. These are incidents like car accidents and crashes. Setting a lower threshold value will result in the Bidirectional LSTM capturing “smaller” events as anomalies, for example, hazards in the roadway or animal-related incidents. We use a threshold value of 1.5, which allows for major incidents like accidents to be captured, whilst ignoring smaller incidents.

When choosing your threshold, consider the impact of potential loss of data by not capturing enough information, or potential “muddying” of data by capturing too much. Should you choose to replicate this project, we suggest adjusting the threshold value to see what sorts of results are returned by the model. Also consider that the incident database is loaded with not only accident data, but data about *any* reported incident by

the CHP. For this reason the threshold can easily be adjusted from 1.5 to a smaller value to capture smaller incidents, and those incidents can be looked up in the incident database.

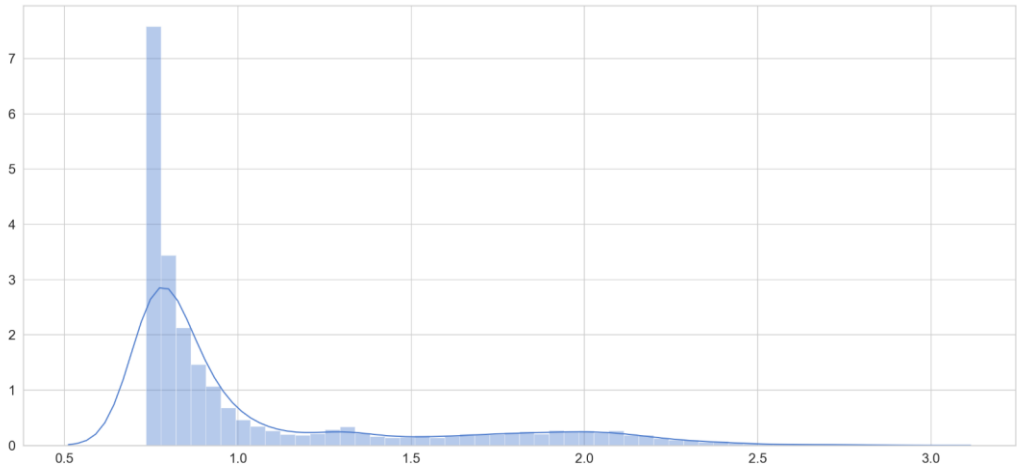

Figure 10. Threshold values

Capturing Anomalies and Interpreting Results

The model pictured in Figure 11 captures a number of points as anomalies. Notice that each trough in the graph is made up of multiple captured points. The entire drop and rise of each trough can be considered as a single anomaly, made up of many captured points from the data set.

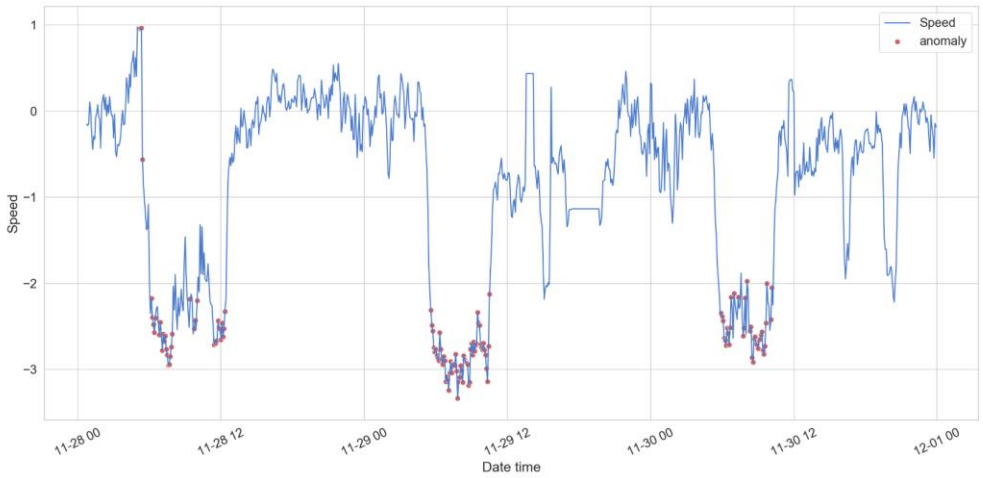

Figure 11. The Bidirectional model capturing anomalous data points

For each captured anomaly, we search the database for reported incidents downstream. With the current threshold value set to 1.5, the model captures “large” anomalies. These larger incidents tend to be things like car accidents, rather than smaller reported incidents like road hazards which do not result in such a large discrepancy in captured speed data. Let us once again look at Figure 8, the reported incidents are indicated by black arrows, somewhere in the downstream of the dataset.

It is important to note that the difference between the timestamp of a captured anomaly and the timestamp of a reported incident that matches it can vary greatly. For some anomalies, a resulting major incident can be reported within 30 minutes of the captured anomaly. For others, the incident may be reported hours after the captured anomaly. Since incident reports are generated by the CHP, times of incidents can at best be an officer’s estimation, or even the time that the scene was cleared. For this reason, there is no way to easily determine exactly *when* an incident occurred to cause an anomaly to be captured. However, there will very likely be a reported incident in the database within a number of hours of the captured anomaly assuming we search the correct highway, highway direction, location, and timestamp of the incident table.

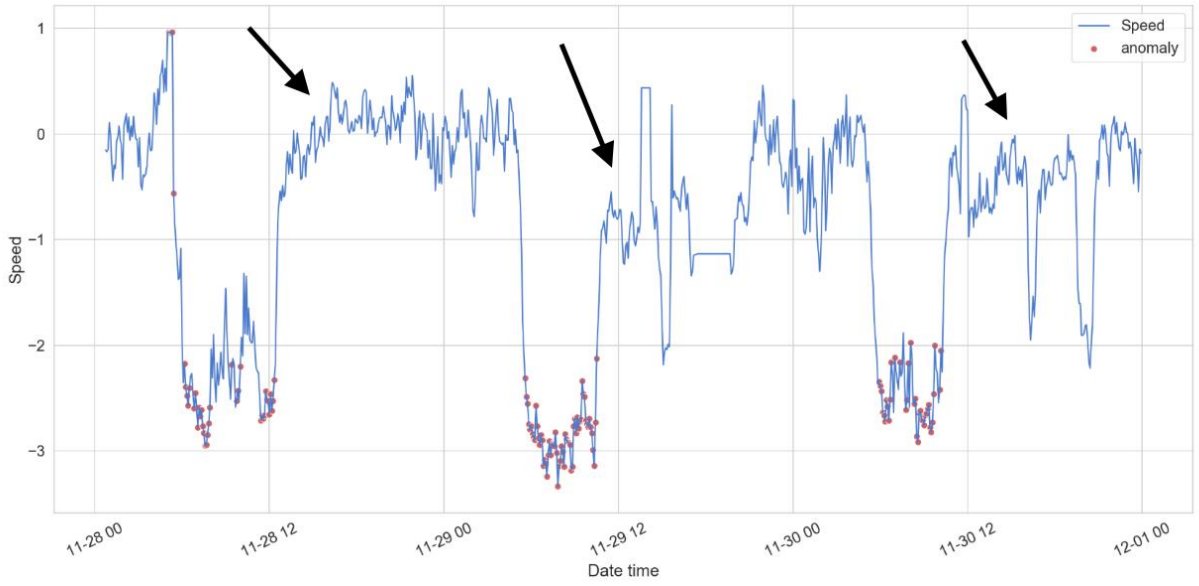

Figure 8. Incidents found in the database that match up to a captured anomaly

Bidirectional Model Extensions

Like the database, the modeling section of this project can also be extended. Currently the incident database must be manually searched to match up reported incidents with captured anomalies. As previously mentioned the time at which an incident is reported compared to the time when an anomaly is captured by the model varies unpredictably.

Usually, an incident is reported within a number of hours after the time at which an anomaly is captured. This time frame can vary. In our own experiments we found incidents that were reported within 30 minutes of the captured anomaly it associates with as well as incidents several hours after their associated anomalies.

For this reason, we suggest an extension to the project in which captured anomalies are first aggregated into a single anomaly and stored within a mapping of timestamps and anomalies. Then an automated process can search the incident database with the correct parameters (location, date, time, highway, highway direction, etc) and find a likely match between reported incidents and the stored anomaly/time value pairs.

There is currently not an agreed-upon threshold for how far we should look for an incident after the captured anomaly and automation could provide for further data in deciding how far along to scan in the incident database to find a match.

Project Methodology

Included below are notes taken on the methodology of the project, including its users, the goals, tasks, and subtasks that make up the project, the project represented as a set of workflows.

Who are the users?
Emergency services responding to collisions and accidents.
Civil engineers designing and improving roadways.

What are their goals?
Emergency services workers have the goal of improving response times to accidents and collisions.
Highway engineers have the goal of improving safety conditions along major highways and roadways to prevent accidents from occurring in the first place.

Goal 1: Improve accident response time for first-responders

Subtask 1 - Collect PeMs data on collision incidents from CHP

Subtask 2 - Collect PeMs data on traffic flow and speed from highway loop detectors

Subtask 3- Collect weather data from California RWIS

Subtask 4- Merge datasets into a single database

Subtask 5- Filter out irrelevant/noisy incident data

Subtask 6- Combine dataset tables with SQL queries based on time and location of incidents

Subtask 7- Train ConvLSTM model on subset of sanitized data

Subtask 8- Use model to predict when and where accidents will occur

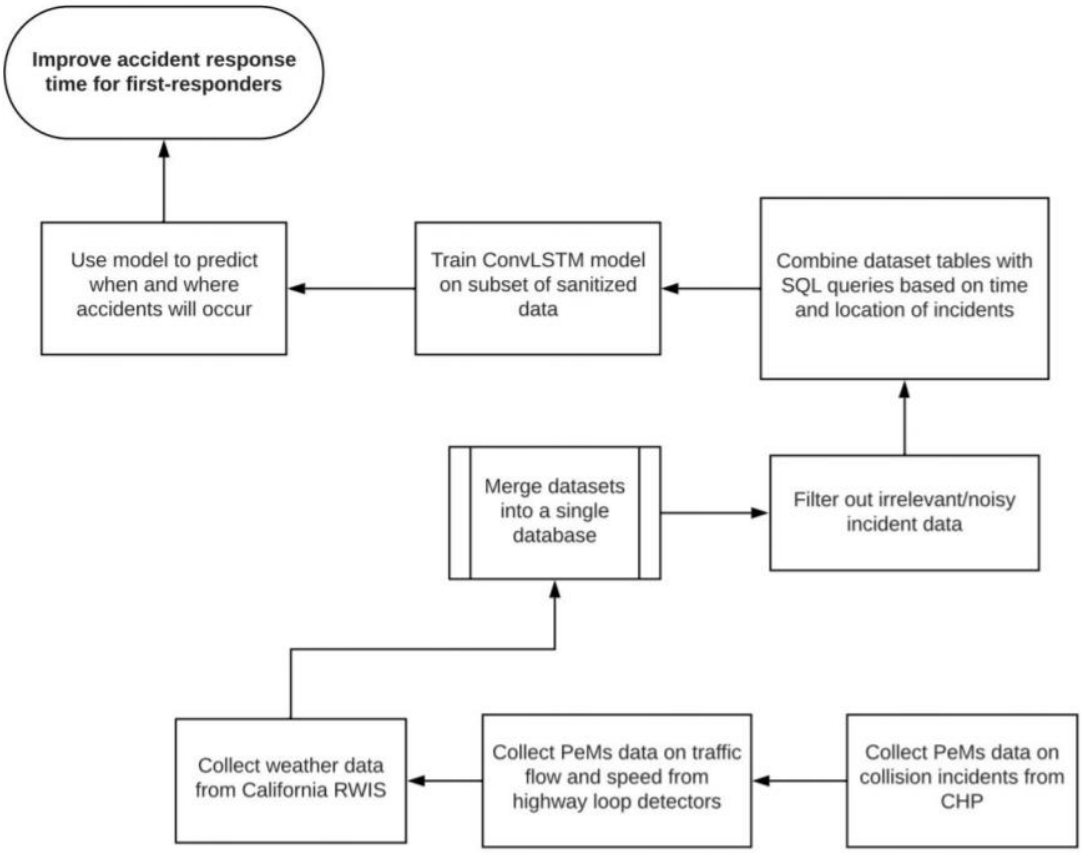

Figure 12. Workflow for Goal 1

Goal 2: Design safer roadway systems

Subtasks 1-8 - remain the same as in Goal 1

Subtask 9 - Accident prediction model used to identify patterns that correlate with incidents

Subtask 10 - use identified patterns from Subtask 9 to train model that detects incidents

Subtask 11 - Analysis on sections of roadways with highest occurrences of incidents

Subtask 12 - Inspection of why incidents occur in higher volume on those sections

Subtask 13 - Focus on fixing/improving conditions along those sections

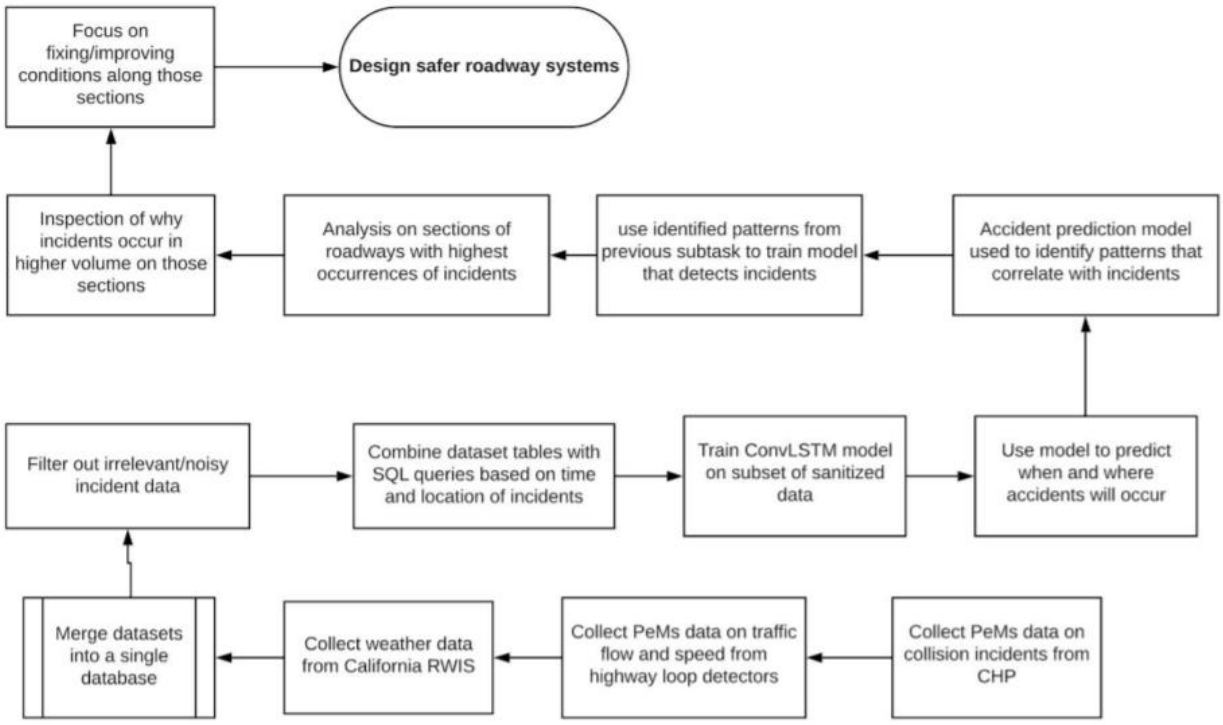

Figure 13. Workflow incorporating Goal 2

Implementation-based service description for each task

Initial data collection for Subtasks 1 and 2 is quite simple. The California Department of Transportation (CalTrans) offers this data for free via the CalTrans Performance Measurement System (PeMs). All that is needed is to make an account on the http://pems.dot.ca.gov/ website.

Incident data can be downloaded in either .csv or .txt format. It is provided via PeMs and is collected from California Highway Patrol (CHP) incident reports. For our project, we have downloaded months worth of incident data in .txt format. The flow, speed and other PeMs data comes from California's robust loop detector system, which has hundreds to thousands of loop detectors set up along each highway in the state. The number of loop detectors varies from highway to highway. This data can be acquired through the same means as the incident data.

Weather data for Subtask 3 comes from California RWIS (road weather information system) and is available publicly in CSV, JSON, TXT and XML formats.

For Subtask 4, Python modules are created using the aggregated .txt data files as inputs. Python files flow_reader.py, incident_reader.py and weather_reader.py are used

to read data from the files and parse it. Each incident, flow data point, weather data point, etc, is converted to separate Python classes which are then loaded as tuples into a SQLite database by another Python module db_loader.py.

Within this phase, Subtask 5 is also completed. Since not all of the data is uniform or consistent, data preprocessing is necessary before data is loaded into the database. All data points are normalized to improve data integrity and reduce redundant data. As an example, some incidents reported by the CHP can be duplicates, as officers from different departments may file an incident report on the same incident. Therefore, duplicate incidents may suggest an increased incident-rate for a certain location and time and must be accounted for. In order to match incidents with loop detector and weather data, timestamps for all data points are converted to Python datetime objects so that Subtask 6 can be easier to do.

Subtask 6 uses the SQLite database file raw_data.db to aggregate a number of raw data tables like incidents_raw, flow_raw, etc. From this point, SQL queries are executed to join tables in meaningful ways. For example, columns can be joined on time stamps, which are available as Python datetime objects in each tuple, or by location or an inner join can be done using both metrics. The new tables can be grouped by columns which indicate the type of incident, the weather patterns or any number of factors that could be deemed important to the model.

For Subtasks 7 and 8, the deep learning model will be a Convolutional LSTM model which extends a typical long short-term memory neural network to have convolutional structures. This phase is not complete yet, so specific implementation details will have to be added on in the future.

The subsequent Subtasks can be considered extensions to this project, likely done by civil engineers, researchers or personnel related to emergency services. Patterns from the ConvLSTM model can be analyzed to better understand what factors are most likely to result in traffic accidents. This data can be extrapolated upon for the accident detection model, which will help researchers understand what sections of roadways are most prone to accidents. From there a mixture of ground-work and further data analysis can tell researchers why accidents occur at higher-than-average rates in a particular location, and corrections to the infrastructure, signage, road conditions, etc., can be made.

Lists of workflows covering each goal

Goal 1 (Improving accident response times) = Data collection + data aggregation + data parsing and filtering + database build + data querying + deep learning model training +

deep learning model predictions + identifying problem areas + taking action to better respond to those areas

Goal 2 (Designing safer roadways) = Data collection + data aggregation + data parsing and filtering + database build + data querying + deep learning model training + deep learning model predictions + identifying characteristics + training new (identifying) model + analysis and inspection of roadways + actual (physical) improvements to roadways

Note: The final steps of Goal 2 should be implemented as a typical design cycle. Once real-world (physical) improvements are made to the roadways -- through improvements to infrastructure, signage, etc., then the model should be used to reevaluate the same locations and compare new incident rates to rates measured before the improvements. If incident rate remains the same, or at unacceptable levels - new characteristics should be identified and further improvements should be made.

Lessons Learned / Conclusion

Project Timeline

A timeline with important milestones and events relating to our project is detailed below.

- January 23 - Project Launch
- January 31 - First meeting with client, establishing weekly cadence of meetings
- February 22 - First presentation / progress update to class
- March 31 - Second presentation / progress update to class
- April 6 - Team methodology assignment
- April 6 - Interim Report
- April 26 - Draft of final VTechWorks submission uploaded
- April 28 - Final presentation to class
- May 5 - Team peer evaluations
- May 6 - Final Report

Problems Encountered

This section discusses some issues we encountered in building this project, as well as the solutions we discovered to rectify those issues.

Autoencoder vs. Bidirectional LSTM

Initially the model developed was an LSTM Autoencoder. The autoencoder took speed as an input feature and we trained the model accordingly. However, when more features were added, such as downstream speed, the model began to fail. In fact, an autoencoder model is not well suited to deal with multiple features, and given the number of features we consider in this project, a switch was needed. For this reason the bidirectional model took over as the primary LSTM used in our detection model. The bidirectional LSTM is better suited to multi-featured inputs and is faster in training compared to the autoencoder.

Sluggish Database

As our data requirements grew and more features needed to be added to the database, we found our table sizes to be growing at a very fast rate. We decided not to utilize a remote server since we did not want to add the time cost of sending that data over the network. Instead, we approached this problem with a number of solutions. Since each member was building the database on their personal machines, processing power was a limiting factor. Although in theory the database only needed to be built once - as we came up with new data sources and added new tables, the Python scripts would have

to be rerun for each member and the database would need to be rebuilt. A simple workaround was uploading the already-built database straight into the repository. There would be no reason for any future developers to have to rebuild the database unless they choose to modify the schema in an extension. However, even if the database is rebuilt, it only needs to happen once, as the primary goal of the database is to provide support for the Bidirectional LSTM model.

We also faced challenges in having slow-running SQL queries. To improve performance an indexing strategy was added. The details of this strategy are available in the developer manual under the subheading *Database Indexing Strategy*. Moreover, we found that the slowest-running queries were actually not required for the model, and we were able to scrap them. The primary function of the SQL statement portion of the project was to collect subsets of the data to train the model on only portions of the highways. The indexing strategy worked very well in this case, as the primary key associated with highway locations is a *real* (float) value. SQLite had no problem in building indexes for these columns and searching was very fast. From there, the relevant subsets of data were exported to CSV format so that the queries would not need to be re-run. Furthermore the CSV format was more convenient to use within the modelling section of the application.

Remote Work

We encountered challenges and delays in our project progress due to the spread of COVID-19 in the United States and the cancellations and logistic changes it caused. We attempted to work around these delays by maintaining team contact via Zoom meetings. Meetings continued between group members remotely and further efforts to communicate were made over email and Zoom rather than in-person meetings. However, time was still lost as we had to reorganize our efforts in the confusion that followed Virginia Tech’s campus shutdown. The time lost decreased our likelihood to complete our stretch goal of an accident detection codebase.

Future Work

We designed this project with the goal of keeping it easily extensible for future developers. We understand that the goal of this project is primarily research-based, rather than developing a finished product to deploy for use on an extensive scale. Rather, we assume that this project could be best used by researchers and data scientists who have the goal of improving road conditions and helping emergency services rapidly respond to accidents.

For these reasons, the database-building component of the project is modular and can be easily extended. Future work may include adding new data types to the database - and by extension, the bidirectional model. Furthermore, further automation is needed to match anomalies captured by the bidirectional model with reported incidents in the database. We discuss in detail the types of extensions that can be made in the Developer Manual portion of this report under subheadings *Database Extensibility* and *Bidirectional Model Extensions*.

What We Learned

We have planned and worked on this project throughout the semester and have certainly been forced out of our comfort zones for much of it. Going into this project we had no previous exposure to any Deep Learning concepts, and very limited experience with databases. Overall our understanding of databases and recurrent neural networks has greatly expanded. We have also never used Jupyter Notebook before. We learned that it is an excellent tool, especially in the field of data science and analytics, for researchers to share and explain their knowledge and code bases.

Another challenging aspect of this project was having to plan and build it all from scratch. We received guidance from statistics experts at the library, our client Farnaz Khagani, and our professor Dr. Edward Fox. We had to come up with the best approaches to dealing with the problem Farnaz was trying to solve and this took a lot of planning and forethought as to how the project would look in the future. It was not an easy task by any means. Although other CS classes have made us start projects from scratch, typically the materials and required knowledge to complete the project was covered within the class. For this project, we had to deal with new technologies and concepts while also planning for the future of the project and starting it from scratch. We certainly learned a lot about problem solving and planning for the future; we likely could not have achieved such a monumental task without Farnaz’s guidance.

Finally, we learned the importance of receiving incremental feedback on a long-running project. The presentations we gave to the class allowed for ample opportunities to correct our mistakes and forced us to consider which aspects of our project we should stick to, abandon, or rework. Without these consistent reviews, the quality of our work would have suffered, and we likely would not have reached the finish line.

Acknowledgments

The team would like to thank Farnaz Khagani, the project client. Farnaz is a graduating Ph.D. student in Civil Engineering. She was crucial in every stage of the project planning and development. Her knowledge relevant to traffic data sources, as well as deep learning applications in the realm of traffic-related applications was indispensable. Farnaz can be contacted via her email, farnazk@vt.edu.

We would also like to acknowledge Dr. Edward A. Fox, our CS 4624 professor. Dr. Fox's guidance during team discussion sessions was vital in our journey to learn about deep learning techniques, as well as possible challenges we might face in maximizing our data quality.

Finally, we would like to thank our CS 4624 classmates for their constructive feedback on our first and second class presentations. This feedback was helpful as we sought to explain and present the details of our project in a clear and effective way.

References

[1] Byrne JP et al. Association between emergency medical service response time and motor vehicle crash mortality in the United States. *JAMA Surg* 2019 Feb 6; [e-pub]. Retrieved February 12, 2020, from https://doi.org/10.1001/jamasurg.2018.5097

[2] Crandall M. Rapid emergency medical services response saves lives of persons injured in motor vehicle crashes. *JAMA Surg* 2019 Feb 6; [e-pub]. Retrieved February 12, 2020, from https://doi.org/10.1001/jamasurg.2018.5104

[3] Caltrans. 'Signals, Signs and Sensors' High on Fix-It List. Retrieved February 12, 2020, from https://dot.ca.gov/programs/public-affairs/mile-marker/winter-2019-2020/signals-signs-sensors-high-on-fix-it-list.

[4] Brownlee, J. (2020, January 7). How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras. Retrieved April 29, 2020, from https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/.

[5] Zhuoning Yuan, Xun Zhou, Tianbao Yang. 2018. Hetero-ConvLSTM: A Deep Learning Approach to Traffic Accident Prediction on Heterogeneous Spatio-Temporal Data. In KDD ’18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom. ACM, New York, NY, USA, 9 pages. Retrieved April 29, 2020, from https://doi.org/10.1145/3219819.3219922.

[6] Srivastava, P. (2017, December 23). Essentials of Deep Learning : Introduction to Long Short Term Memory. Retrieved April 29, 2020, from https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/