**Weekly Report**

**Week 1**

**Date:** 11/09/2020 – 11/13/2020.

**Total time contributed**: 12 hours

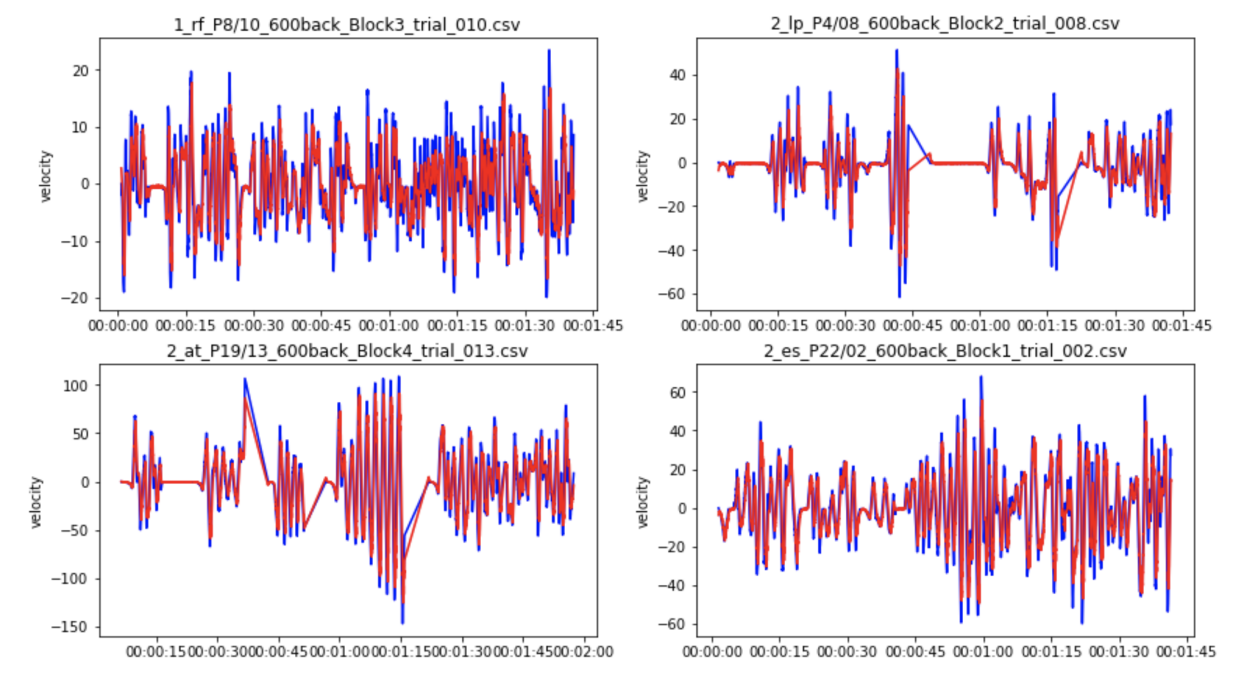
**Total time allowed**: 10 hours

In this week, I mainly did data cleaning and data manipulation work to understand the data.

**Data manipulation**

The data includes 34 people two day’s experiments. There are 1360 csv data sets in total. I tried several methods to load all the data using python. First I tried used two simple for loops to read the data, however, it took me so much time to run it. Then I used a list and concatenate all the csv data sets into one big data set. I added a new column called ‘peopleTrialKey’ which could identify the data from which people in which trial and in which day. Also I included a datetime format column called ‘datetimeNew’, which could be used to do date-related manipulation in the future work. Another new column I added is called ‘calculated\_vel’, which is the new velocity and derived from current velocity. The formula is ‘calculated\_vel’ = (Position\_t - Position\_t-1)/ (Seconds\_t - Seconds\_t-1). The reason I did this is because there’s a delay of the machine while it records the velocity. The delay is around 30-40ms. In the end I wrote the new big dataset into a csv file called ‘data\_all.csv’.

Since we have a calculated velocity, we need to validate it. So I did some visualization on both velocities. I randomly selected some ‘peopleTrialKey’. Below are the graphs. The bule line is original machine velocity, while the red one is the calculated velocity.



Both machine velocity and calculated velocity are quite similar over all, even though there might be some cases in machine velocity which have big gap compare to calculated velocity.

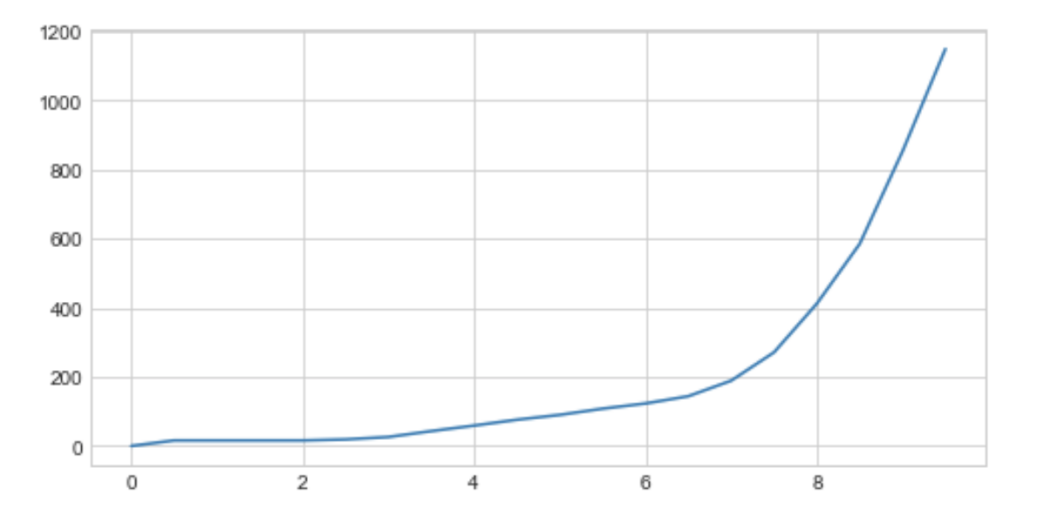
**Data exploration**

Since there’s a column called ‘trialPhase’, which represents the status of the machine. ‘1’ means under the machine control, usually means the machine is resetting after a crash. We don’t know those data, so I **excluded** all of them when I do the future work. ‘3’ means under human control. ‘4’ means there’s a crash happening.

Since there’s another criteria which is if the position is beyond 60 degree or - 60 degree, we say there’s a crash. The first criteria I mentioned above when the ‘trialPhase’ = ‘4’, there’s a crash. However, when I used the 60 degree criteria, I found there’s a difference from the ‘trialPhase’ criteria. The conclusion is that: there are 4300 crash events based on +- 60 degree criteria; and there are 6251 crash events based on ‘trialPhase’. We can see there’s a huge difference. The reason is because there’s tiny bias of the machine when recording the position and the degree of that position. In the end, we decided to use the ‘trialPhase’ criteria.

There are also some cases that two consecutive crashed happened during a very short time. Since we will take the previous time point data as our features, we need to know how is the distributions of crash events in different time scales. Below is a graph showing the distribution:

The x-axis is the time scale between two consecutive crashes, while the y-axis is the total number of crash events. We can see the majority of crash events have enough time scale (2 seconds).



In the following work, I excluded all the crash events that have less than 2 seconds time gap. There are 12 crash events are removed. 5 seconds, 90 crash events.

**Next Step:** continue do data manipulation and start to do feature engineering.

**Week 2**

**Date:** 11/16/2020 – 11/20/2020.

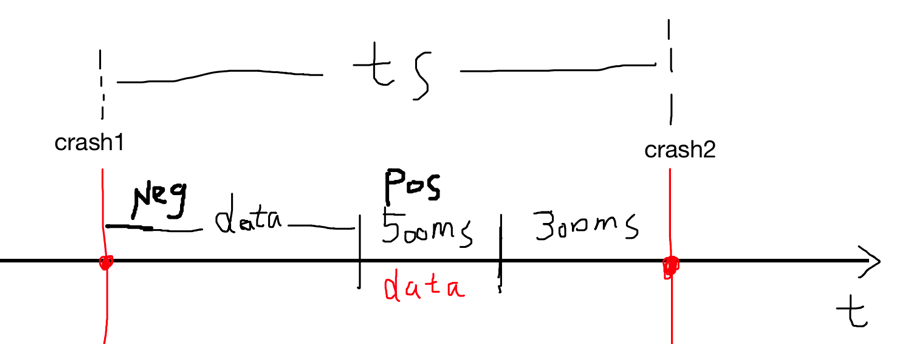
**Total time contributed**: 8 hours

**Total time allowed**: 10 hours

This week, I continued doing data manipulation and also start to do feature engineering.

**Data manipulation & feature engineering**

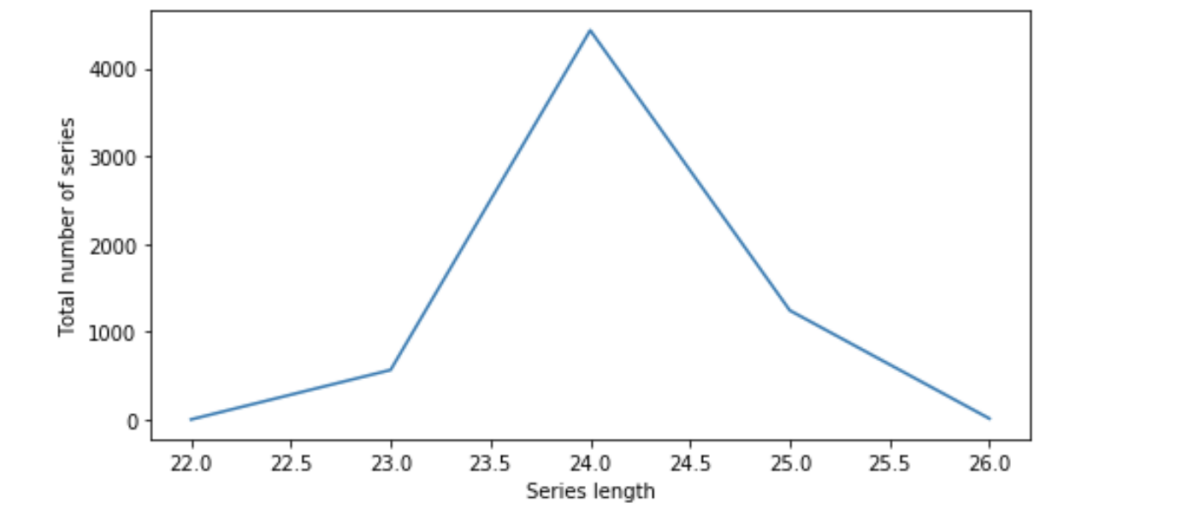
After last week’s work, I have already got somehow cleaned data. Then I tried to create the data for training which includes the features and labels. When label equals to 1, it represents there’s a crash, when label equals to 0, it represents non-crash events. In terms of the features part, below is a graph I drew.



We first decided to predict 300 ms in advance if there’s a crash happen or not. So we leave the data 300 ms close to a crash event out. Then we take time scale = 500 ms as our feature for that crash event. For non-crash events, I took all the data between two crash events except for the 300 ms close to a crash and 500 ms that used as crash event feature (**positive**). In the graph, there’s a time line called ‘data’, it represents the data we will used for non-crash events (**negative**). However, for non-crash event, we decided to use a sliding window to get the features, with 500 ms time scale as features and 300 ms as jumping step. In this way, our number of crash events (**positive**) only have 6251cases, while our number of non-crash events (**negative**) is way more than our positive cases. So our data is very **imbalanced**. It’s not good for our model. And we need to utilized techniques to deal with this issue.

300ms, 500ms, 700ms, 900ms, 1100ms, 1300ms

After we get all the features and labels, I found that the sampling rate is quite different for both crash and non-crash events. I drew a sampling rate distribution of crash event in below graph.



We can see that the sampling rate is different, which could be an **issue** for our future LSTM model. So we need to make all samples’ sampling rate the same, so that we could feed into our model.

**Next step:** solve the imbalanced data issue and also the inconsistent sampling rate.

**Week 3**

**Date:** 11/23/2020 – 11/27/2020.

**Total time contributed**: 8 hours

**Total time allowed**: 10 hours

In this week, I solved the two issues I mentioned last week. One is the imbalanced data, one is the inconsistent sampling rate.

**Imbalanced data**

I first used a under sampling method to solve the imbalanced data. To be more specific, since the number of crash events is around 6000, I randomly chose 6000 non-crash events among all the negative cases. In this way, we could have a balanced data set. However, the drawback of this method is we will miss lots of information on non-crash event side. So it’s not a good idea.

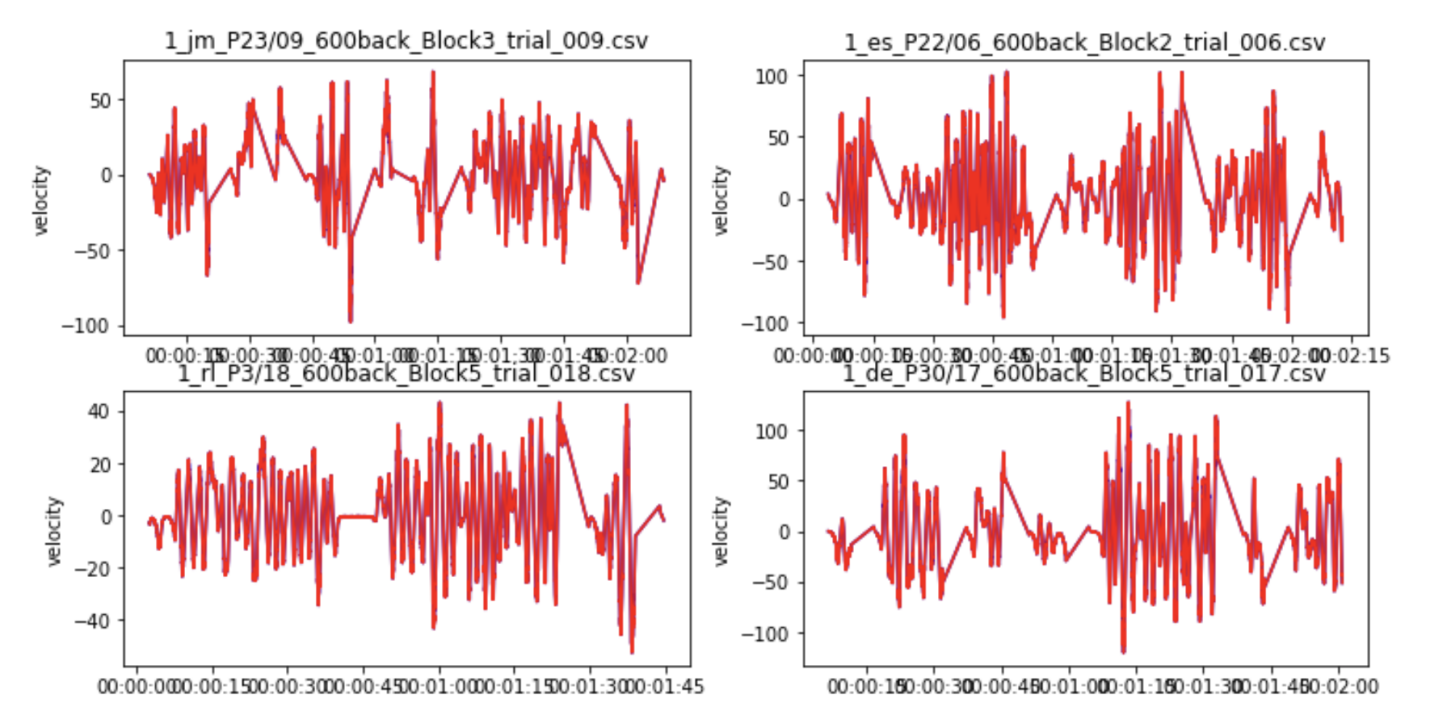
Another method which is corresponding to under sampling method is called over sampling. This method is to copy all the crash events till the total number of it reaches the same number of non-crash events. This method’s defects are there are so many copies.

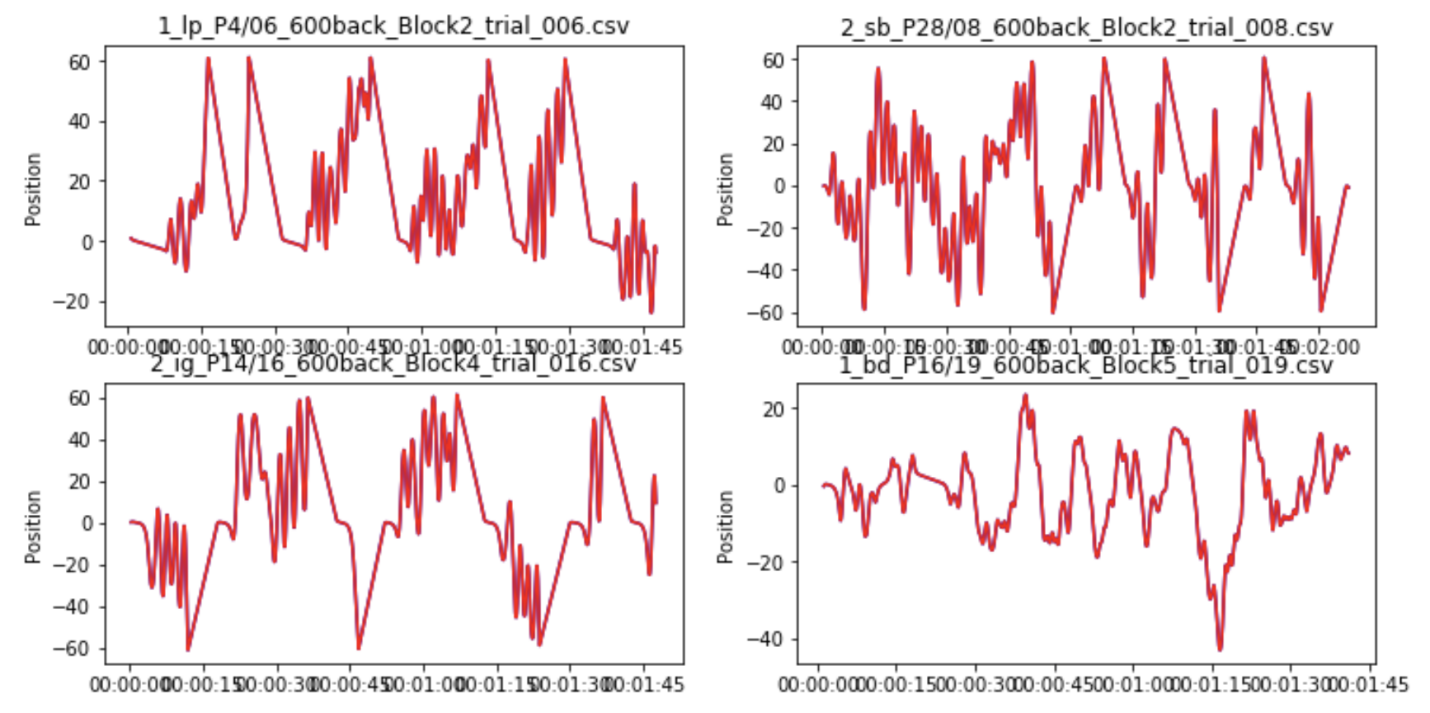
The last method I tried is based on our model. We can assign different weights to different classes. Usually we would assign higher weights to positive cases, since we value it much more. I decided to try 1:1, 1:10, 1:50, 1:100 ratios.

**Inconsistent sampling rate**

To solve this problem, we usually use resampling and interpolation methods.

**Pandas resampling:** there’s a function in pandas used for time series data called resample and interpolate. Using this method, we could have a very good fit before and after resampling. Below are some visualizations included resampled velocity and positions. I randomly select 4 different ‘peopleTrialKey’. Blue stands for original data, and red stands for resampled data.





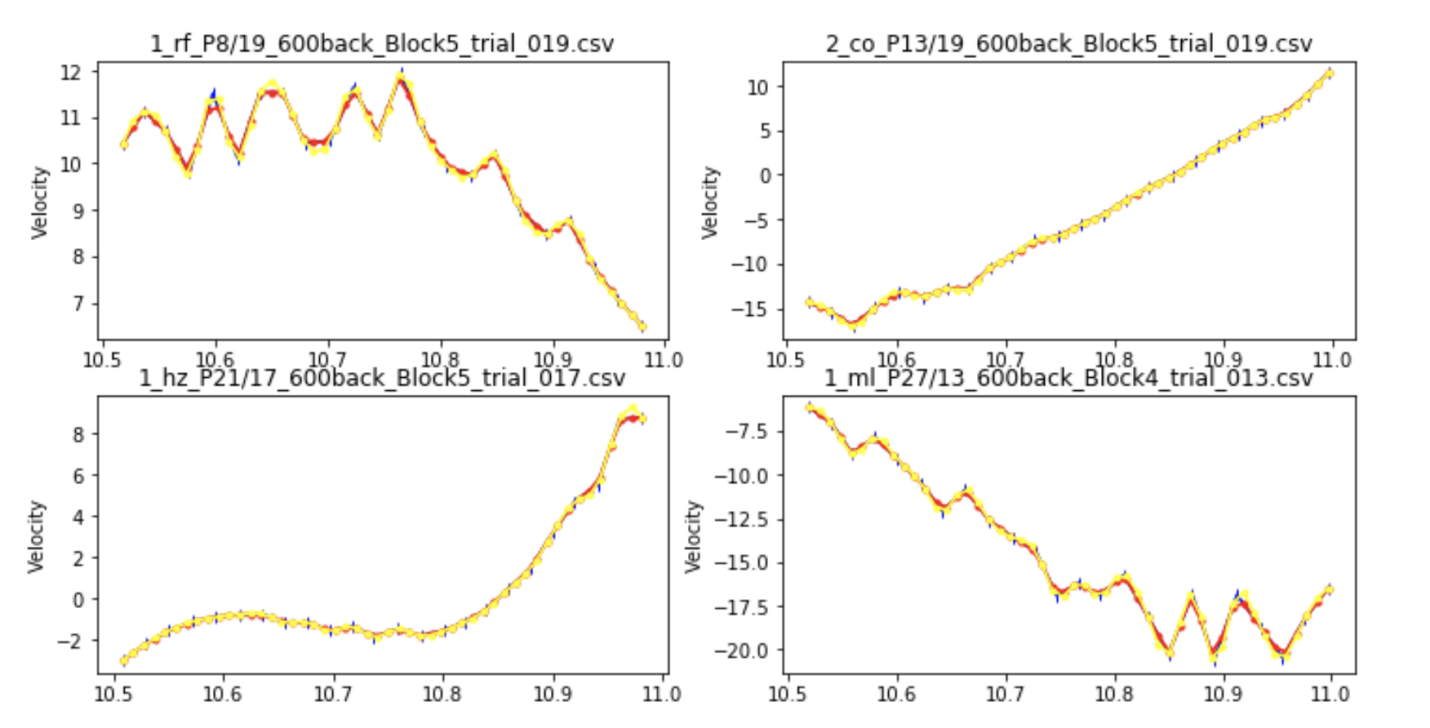
Even though this method can have a very good matched between before and after data, the drawback of this is it still cannot solve the different sampling rate issue.

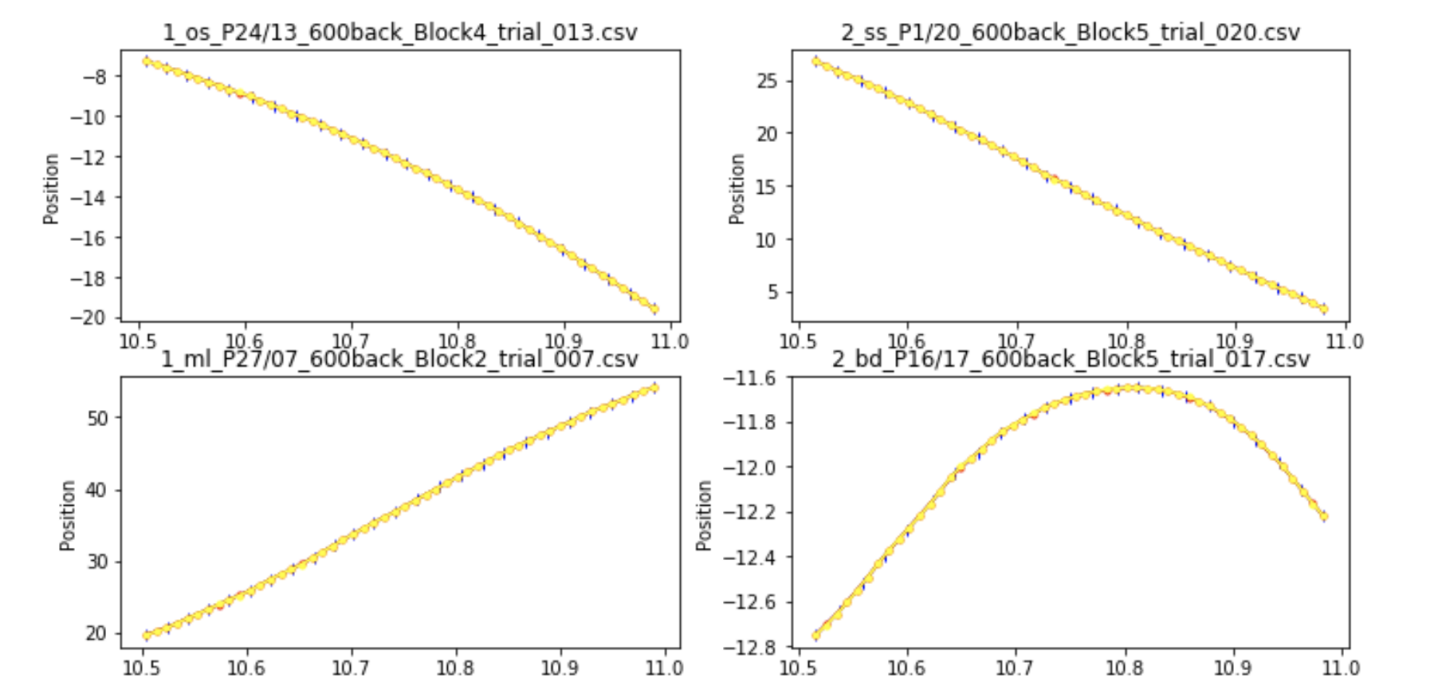
Because:

e.x.1: from 1.003 seconds to 1.500 seconds, there would be 498 sampling rate;

e.x.2: from 1.005 seconds to 1.500 seconds, there would be 496 sampling rate.

**Numpy & scipy resampling:** this methods can sovle the different sampling rate issue since numpy.linspace could help us have a fixed length. **Here we choose sampling rate of 50**. Another thing to be noticed is that we could use different method while do resampling. For example, we could use ‘linear’, or ‘quadratic’. To identify which method is better, I drew both. Below are graphs of before and after resampling. Blue is the original data, red stands for ‘linear’ resampling method, and yellow line stands for ‘quadratic’ resampling method.





This method works very well too! And one benefit of this is that we could make the sampling rate fixed. So I decided to use this method while do feature engineering. Also I tried both **linear** and **quadratic** interpolate methods, but there's **no big difference**, even though the quadratic is a little bit smoother. In the end, I choose the linear method to do interpolation.

**Week 4**

**Date:** 11/30/2020 – 12/04/2020.

**Total time contributed**: 2 hours

**Total time allowed**: 2 hours

This week, since it’s my final week, so I didn’t work too much. However, I spent some time building up the model and ran some experiments.

The model I used is the long and short term memory (LSTM), which is designed for time series analysis.

**Next step:** run the experiments and analyze the results.

**Week 5**

**Date:** 12/07/2020 – 12/11/2020.

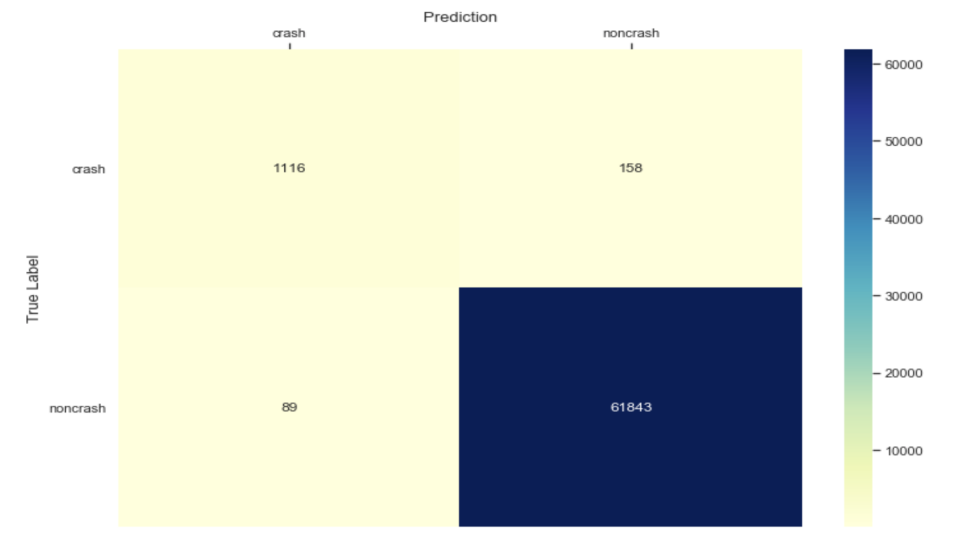
**Total time contributed**: 12 hours

**Total time allowed**: 20 hours

This week, I spent time to build up the model and run lots of experiment.

I tried different weights while running the experiments. In terms of the model, I first used a LSTM layer with 128 units, followed by a dropout layer with rate = 0.5. Then there are two dense layers. One is with 128 units and activation function is ‘relu’. One is with output shape = 1 and a ‘softmax’ activation. The loss I used is 'categorical\_crossentropy' and optimizer is 'adam'.

After I got the results, I drew the confusion matrix for our results. Later on, I calculated accuracy, precision and recall.



Below are the summarized results.

Accuracy = (1116 + 61843)/ (1116 + 61843+158+89)

Recall = 1116/(1116+158) we value the crash event a lot!

Precision = 1116/(1116 + 89) showing our model’s ability to predict the positive samples.

Type I error : 158

Type II error : 89

|  |  |  |  |
| --- | --- | --- | --- |
| **Class\_weights** | **Accuracy** | **Precision** | **Recall** |
| 1:1 | 99.61% | 92.61% | 87.60% |
| 1:10 | 99.28% | 92.61% | 87.60% |

**Next step:** try both calculated velocity and original velocity in the models to see if there’s difference among results. Also, assign more class\_weights to do comparison.

**Week 6**

**Date:** 12/14/2020 – 12/18/2020.

**Total time contributed**: 10 hours

**Total time allowed**: 20 hours

This week, I removed all the machine control data and ran the experiments again with weights of {1:1, 1:10, 1:50, 1:100}. Also, I wrote reports for the previous weeks.

The results are showing below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Velocity type** | **Class\_weights** | **Accuracy** | **Precision** | **Recall** |
| Calculated | 1:1 | 99.59% | 90.65% | 95.53% |
| 1:10 | 98.80% | 70.44% | 99.83% |
| 1:50 | 98.67% | 68.37% | 99.83% |
| 1:100 | 96.72% | 46.62% | 100% |
| Original | 1:1 | 99.66% | 92.08% | 96.27% |
| 1:10 | 99.66% | 90.49% | 98.43% |
| 1:50 | 98.45% | 64.83% | 99.83% |
| 1:100 | 97.35% | 51.87% | 99.92% |

From the results we can see that, both calculated and original velocity feature performed well when class weight is 1 to 1. However, when the class weight is 1 to 10, the precision of calculated velocity performed much worse than original one. Another thing I noticed is when the class weights increase, the performance is becoming worse, except for the 1 to 10 weight with original velocity.

Here comes to our conclusions:

1. The class weight 1 to 1 is a good choice for both feature types.

2. Increase the weights is not a good idea except for the original features till 1 to 10 weight.

Issues: I am having difficulty using GPU on HPCC to run the experiments.

Next step: increase the time scale.

EEG,

FMRI

**Week 7**

**Date:** 12/21/2020 – 12/25/2020.

**Total time contributed**: 10 hours

**Total time allowed**: 10 hours

This week, I increased the in advanced time length to do prediction with weights of {1:1, 1:10, 1:50, 1:100}. Last week I only tried 300ms in advance, for this week, I tried 500ms, 700ms, 900ms, 1100ms, 1300ms, 1500ms, 1900ms separately with weights of {1:1, 1:10, 1:50, 1:100} on negative and positive samples.

Before we run our model, we need first to generate the data in advance. Here you choose any time length in advance. Then we can start to run models.

Since there are so many experiments and each experiments will take 10 hours to run in local computer, I ran paralleled experiments with school’s HPCC. It cuts down the time spent on one experiments to 3 hours.

Below table is showing all the results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Time advanced | Velocity | Weights | Accuracy | Precision | Recall |
| 300ms | Calculated | 1:1 | 99.59 | 90.65 | 95.53 |
| 1:10 | 98.80 | 70.44 | 99.83 |
| 1:50 | 98.67 | 68.37 | 99.83 |
| 1:100 | 96.72 | 46.62 | 100 |
| Original | 1:1 | 99.66 | 92.08 | 96.27 |
| 1:10 | 99.66 | 90.49 | 98.43 |
| 1:50 | 98.45 | 64.83 | 99.83 |
| 1:100 | 97.35 | 51.87 | 99.92 |
| 500ms | Calculated | 1:1 | 97.73 | 57.92 | 86.64 |
| 1:10 | 97.19 | 51.76 | 95.41 |
| 1:50 | 93.89 | 32.90 | 99.18 |
| 1:100 | 92.58 | 28.75 | 99.18 |
| Original | 1:1 | 98.46 | 74.51 | 74.26 |
| 1:10 | 97.62 | 56.22 | 94.84 |
| 1:50 | 96.09 | 43.28 | 96.89 |
| 1:100 | 92.04 | 27.34 | 99.26 |

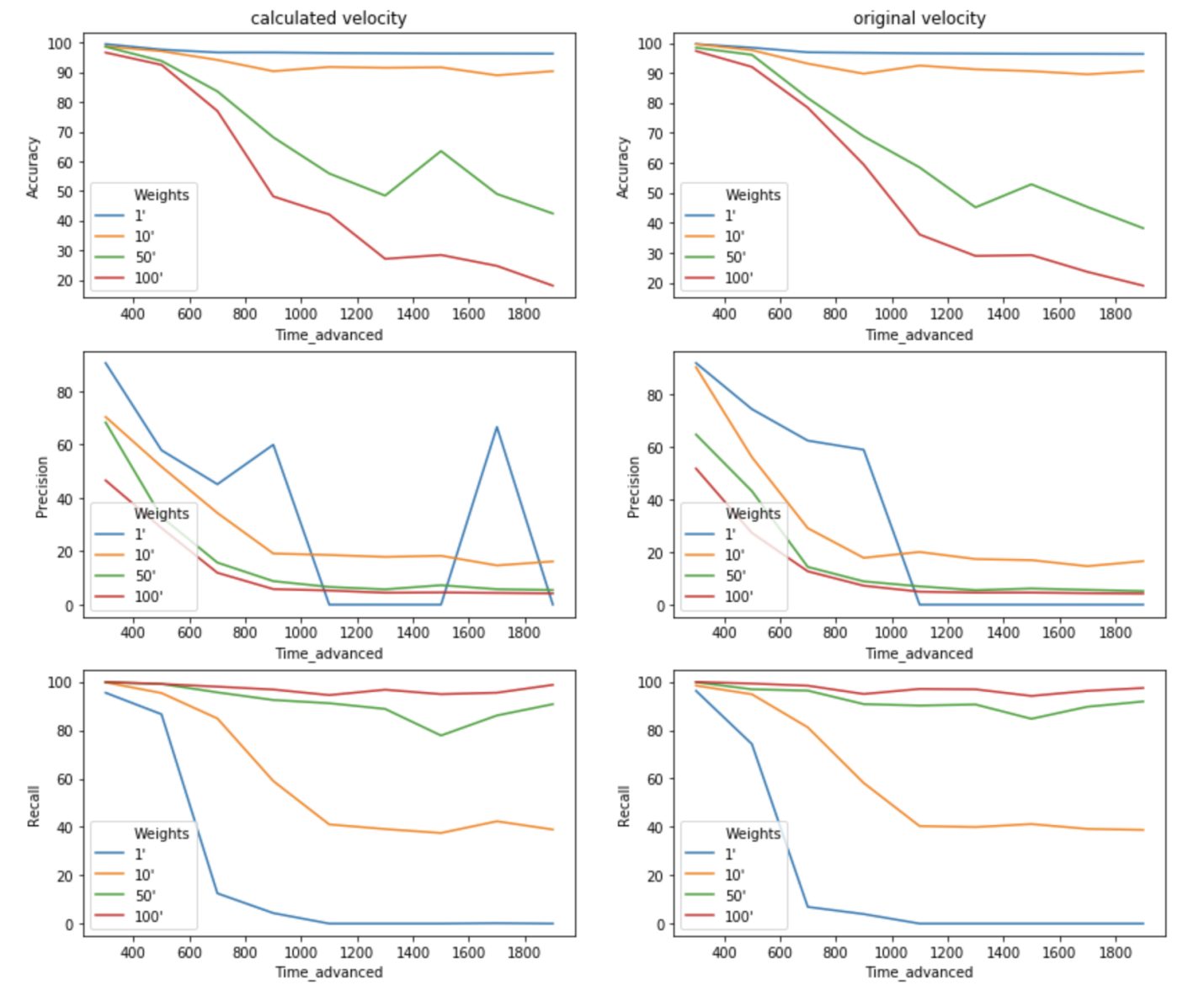
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time advanced** | **Velocity** | **Weights** | **Accuracy** | **Precision** | **Recall** |
| 700ms | Calculated | 1:1 | 96.75 | 45.16 | 12.51 |
| 1:10 | 94.23 | 34.39 | 84.89 |
| 1:50 | 83.63 | 15.77 | 95.69 |
| 1:100 | 77.00 | 11.95 | 98.05 |
| Original | 1:1 | 96.91 | 62.50 | 6.90 |
| 1:10 | 93.13 | 29.13 | 81.15 |
| 1:50 | 81.65 | 14.37 | 96.34 |
| 1:100 | 78.44 | 12.69 | 98.38 |
| 900ms | Calculated | 1:1 | 96.74 | 60.00 | 4.34 |
| 1:10 | 90.42 | 19.17 | 58.99 |
| 1:50 | 68.12 | 8.81 | 92.53 |
| 1:100 | 48.17 | 5.83 | 96.87 |
| Original | 1:1 | 96.74 | 59.04 | 3.94 |
| 1:10 | 89.76 | 17.83 | 58.15 |
| 1:50 | 68.85 | 8.86 | 90.76 |
| 1:100 | 59.44 | 7.21 | 94.94 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time advanced** | **Velocity** | **Weights** | **Accuracy** | **Precision** | **Recall** |
| 1100ms | Calculated | 1:1 | 96.58 | 0 | 0 |
| 1:10 | 91.86 | 18.63 | 41.04 |
| 1:50 | 55.91 | 6.64 | 91.20 |
| 1:100 | 42.10 | 5.30 | 94.56 |
| Original | 1:1 | 96.58 | 0 | 0 |
| 1:10 | 92.48 | 20.07 | 40.32 |
| 1:50 | 58.50 | 6.96 | 90.16 |
| 1:100 | 36.11 | 4.94 | 97.04 |
| 1300ms | Calculated | 1:1 | 96.49 | 0 | 0 |
| 1:10 | 91.57 | 17.91 | 39.12 |
| 1:50 | 48.44 | 5.74 | 88.84 |
| 1:100 | 27.11 | 4.46 | 96.79 |
| Original | 1:1 | 96.49 | 0 | 0 |
| 1:10 | 91.23 | 17.37 | 39.92 |
| 1:50 | 45.17 | 5.51 | 90.60 |
| 1:100 | 28.96 | 4.57 | 96.87 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time advanced** | **Velocity** | **Weights** | **Accuracy** | **Precision** | **Recall** |
| 1500ms | Calculated | 1:1 | 96.40 | 0 | 0 |
| 1:10 | 91.73 | 18.33 | 37.49 |
| 1:50 | 63.49 | 7.28 | 77.83 |
| 1:100 | 28.42 | 4.57 | 94.95 |
| Original | 1:1 | 96.40 | 0 | 0 |
| 1:10 | 90.62 | 16.96 | 41.16 |
| 1:50 | 52.87 | 6.15 | 84.68 |
| 1:100 | 29.25 | 4.59 | 94.13 |
| 1700ms | Calculated | 1:1 | 96.39 | 66.67 | 0.17 |
| 1:10 | 89.05 | 14.71 | 42.35 |
| 1:50 | 49.00 | 5.80 | 86.13 |
| 1:100 | 24.73 | 4.39 | 95.55 |
| Original | 1:1 | 96.39 | 0 | 0 |
| 1:10 | 89.56 | 14.65 | 39.16 |
| 1:50 | 45.31 | 5.63 | 89.66 |
| 1:100 | 23.66 | 4.36 | 96.22 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Time advanced** | **Velocity** | **Weights** | **Accuracy** | **Precision** | **Recall** |
| 1900ms | Calculated | 1:1 | 96.35 | 0 | 0 |
| 1:10 | 90.43 | 16.21 | 38.92 |
| 1:50 | 42.41 | 5.47 | 90.81 |
| 1:100 | 18.06 | 4.22 | 98.80 |
| Original | 1:1 | 96.35 | 0 | 0 |
| 1:10 | 90.65 | 16.57 | 38.75 |
| 1:50 | 38.21 | 5.17 | 91.84 |
| 1:100 | 19.04 | 4.21 | 97.42 |

From above results, we can see that with the increase of the in advance time length, our model performance is decreasing. When the in advance time is less than 700ms, the performance is literally fine. However, when it is more than 700ms, the performance is super poor.



**Conclusions:**

For the accuracy metric, for both velocity types, it’s going down for ratio 1:50 and 1:100 when we increase in advance time length. It’s relatively stable and high score when ratio is 1:1 and 1:10.

For precision metric, for original velocity, it performs good when ratio is 1:1 and 1:10, with less than 400ms second in advance. And then it decreases sharply while we increase the time length. For calculated velocity, it performs good when ratio is 1:1. And then it decreases as the time length increases.

For recall metric, when the ratio is 1:50 and 1:100, for all time lengths, it performs well which makes sense, since we put much more weights on positive samples. However, for the ratio 1:1, recall is high when the time in advance is less than 500ms. And for the ratio 1:10, recall is high when the time in advance is less than 700ms.

For both precision and recall metrics, both perform good when we have less than 500ms time in advance on ratios 1:1 and 1:10.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time advanced | Weights | Accuracy | Precision | Recall |
| 500ms | 1:1 | 98.46 | 74.51 | 74.26 |
| 1:10 | 97.62 | 56.22 | 94.84 |

Methods to improve:

1. we use 500ms time scale to predict, probably use more than 500ms

2. different model

3. Dive into: False positive samples vs false negative samples

**Week 7**

**Date:** 12/28/2020 – 01/04/2021.

**Total time contributed**: 10 hours

This week I tried different training time scales with different time in advance to predict. Based on last week’s conclusion, I only used weights: 1:1, 1:10, 1:50 and original velocity. Below are details:

Time scales to train: 500ms, 1000ms, 1500ms, 2000ms (4 cases)

Time in advance: 500ms, 700ms, 900ms, 1100ms, 1300ms (5 cases)

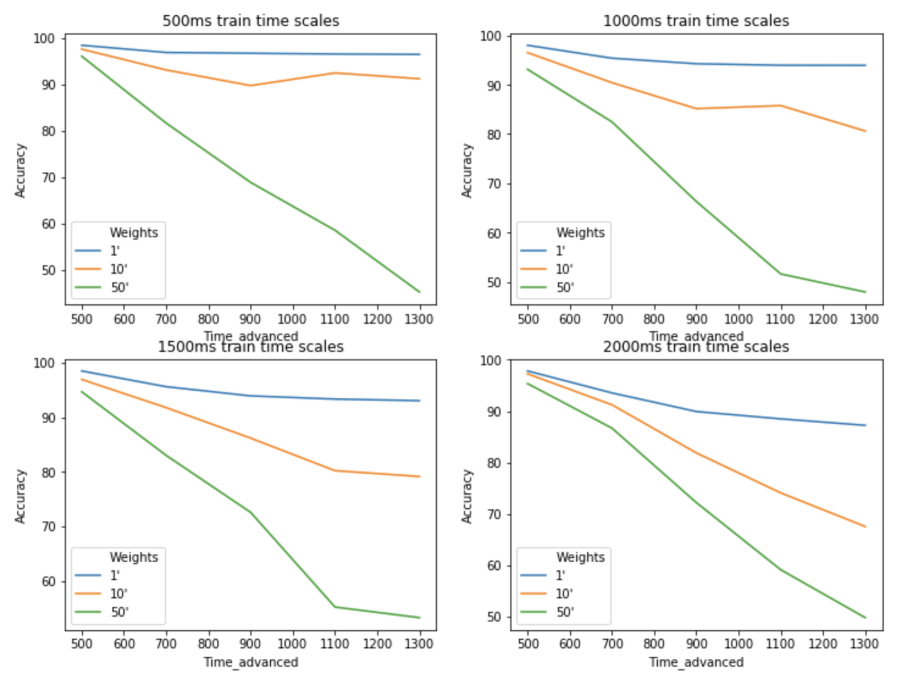
Original velocity

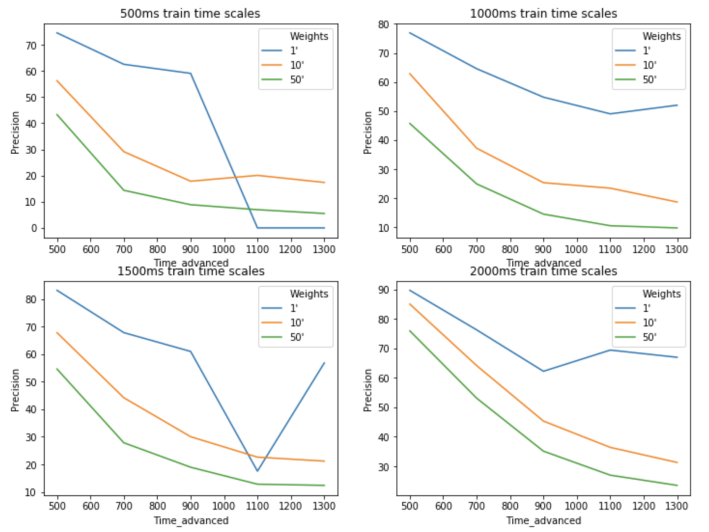
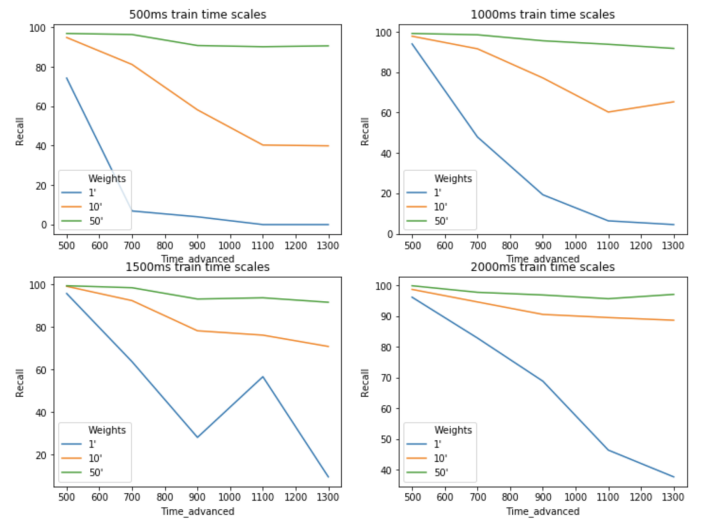
Weights: 1:1, 1:10, 1:50

Below are the results:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Train scale | Time advanced | Weights | Accuracy | Precision | Recall | Rolling step |
| 500ms | 500ms | 1:1 | 98.46 | 74.51 | 74.14 | 200ms |
| 1:10 | 97.62 | 56.22 | 94.68 |
| 1:50 | 96.09 | 43.28 | 96.73 |
| 700ms | 1:1 | 96.91 | 62.50 | 6.74 |
| 1:10 | 93.13 | 29.13 | 81.02 |
| 1:50 | 81.65 | 14.37 | 96.17 |
| 900ms | 1:1 | 96.74 | 59.04 | 3.78 |
| 1:10 | 89.76 | 17.83 | 58.05 |
| 1:50 | 68.85 | 8.86 | 90.66 |
| 1100ms | 1:1 | 96.58 | 0 | 0 |
| 1:10 | 92.48 | 20.07 | 40.22 |
| 1:50 | 58.50 | 6.96 | 90.06 |
| 1300ms | 1:1 | 96.49 | 0 | 0 |
| 1:10 | 91.23 | 17.37 | 39.82 |
| 1:50 | 45.17 | 5.51 | 90.50 |
| 1000ms | 500ms | 1:1 | 98.02 | 76.90 | 92.66 | 300ms |
| 1:10 | 96.53 | 62.84 | 96.41 |
| 1:50 | 93.14 | 45.71 | 97.78 |
| 700ms | 1:1 | 95.40 | 64.56 | 47.20 |
| 1:10 | 90.43 | 37.18 | 90.25 |
| 1:50 | 82.47 | 24.90 | 97.08 |
| 900ms | 1:1 | 94.27 | 54.76 | 18.35 |
| 1:10 | 85.15 | 25.32 | 75.64 |
| 1:50 | 66.30 | 14.50 | 94.16 |
| 1100ms | 1:1 | 93.97 | 49.02 | 06.06 |
| 1:10 | 85.77 | 23.43 | 58.14 |
| 1:50 | 51.58 | 10.50 | 92.06 |
| 1300ms | 1:1 | 93.95 | 52.00 | 03.91 |
| 1:10 | 80.62 | 18.67 | 63.94 |
| 1:50 | 47.92 | 09.75 | 90.17 |
| 1500ms | 500ms | 1:1 | 98.48 | 83.11 | 94.33 | 700ms |
| 1:10 | 96.92 | 67.75 | 97.67 |
| 1:50 | 94.66 | 54.58 | 97.92 |
| 700ms | 1:1 | 95.59 | 67.77 | 62.77 |
| 1:10 | 9176 | 44.17 | 91.04 |
| 1:50 | 83.00 | 27.84 | 97.01 |
| 900ms | 1:1 | 93.91 | 60.95 | 27.66 |
| 1:10 | 86.17 | 30.04 | 77.38 |
| 1:50 | 72.57 | 18.98 | 92.17 |
| 1100ms | 1:1 | 93.32 | 17.55 | 55.83 |
| 1:10 | 80.20 | 22.63 | 75.19 |
| 1:50 | 55.20 | 12.82 | 92.58 |
| 1300ms | 1:1 | 93.03 | 56.78 | 08.61 |
| 1:10 | 79.13 | 21.19 | 69.89 |
| 1:50 | 53.26 | 12.40 | 90.65 |
| 2000ms | 500ms | 1:1 | 97.83 | 89.72 | 94.77 | 1000ms |
| 1:10 | 97.27 | 85.05 | 97.27 |
| 1:50 | 95.37 | 75.96 | 98.44 |
| 700ms | 1:1 | 93.55 | 76.30 | 81.69 |
| 1:10 | 91.24 | 64.17 | 93.25 |
| 1:50 | 86.69 | 53.12 | 96.35 |
| 900ms | 1:1 | 89.92 | 62.24 | 67.82 |
| 1:10 | 81.86 | 45.33 | 89.52 |
| 1:50 | 72.18 | 35.12 | 95.84 |
| 1100ms | 1:1 | 88.49 | 69.44 | 45.42 |
| 1:10 | 74.07 | 36.41 | 88.50 |
| 1:50 | 59.09 | 26.98 | 94.62 |
| 1300ms | 1:1 | 87.24 | 66.99 | 36.71 |
| 1:10 | 67.52 | 31.30 | 87.63 |
| 1:50 | 49.76 | 23.52 | 96.03 |
|  |  |  |  |  |  |  |

.





Conclusion:

In terms of accuracy, there’s a declining trend for all the weights and training time scales when time in advance to predict is increasing. This makes sense since the longer the time in advance, the harder to predict.

As for recall, when the train time scale increases, we can see there’s an increase and scores tend to be more stable especially for weights equal to 1:10. The score is acceptable even when the time in advance to predict is around 900ms when we have 2000ms train time scales.

For the precision, there’s also an improvement when we increase the train time scales. The score is acceptable when the time in advance is around 700ms and the train time scale is 2000ms with weight equal to 1:1.

Combining both recall and precision, we find that the best results are when we have less than 700ms time in advance on ratios 1:1 and 1:10 with 2000ms train time scale.

Basically, increase the train time scale does help!

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Train scale | Time advanced | Weights | Accuracy | Precision | Recall |
| 2000ms | 500ms | 1:1 | 97.83 | 89.72 | 96.12 |
| 1:10 | 97.27 | 85.05 | 98.65 |
| 700ms | 1:1 | 93.55 | 76.30 | 82.84 |
| 1:10 | 91.24 | 64.17 | 94.57 |

Next step: 1. How to predict those crashes happened in a short time. Use the same split.

2. look at which person tends to add more error to the model. Which trial, which crash, which day. Cluster those people. False positive and false negative.

3. destabilized joystick deflection cases. (5-6%) ----- relationship with different people.

Velocity, joystick, and machine direction the same.

**Week 8**

**Date:** 01/05/2021 – 01/11/2021

**Total time contributed**: 10 hours

This week, I generated false positive cases. Also, I corrected all the results by considering those crash cases that were excluded. So now the results make sense statistically.