

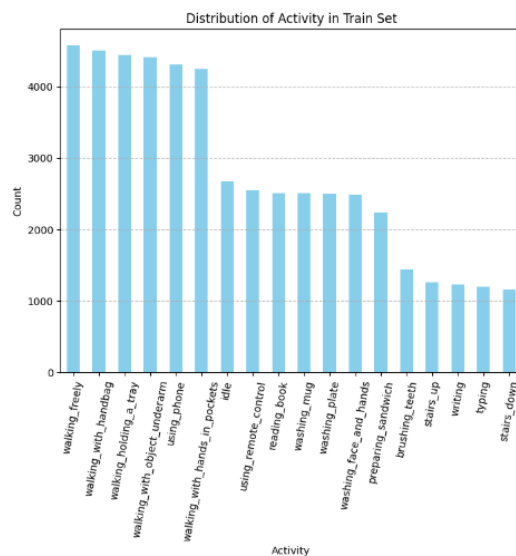
Part 1:

- a) Our data comes from motion sensors that were placed on the limbs of users (either left/right hand/leg) as they were asked to do 18 different tasks such as washing a mug, holding a tray, walking up the stairs.
- We have 2 different types of data files, some of our .csv files only contain accelerometer data and the other .csv files contain 3 different types of measurements (accelerometer, gyroscope and magnetometer data).

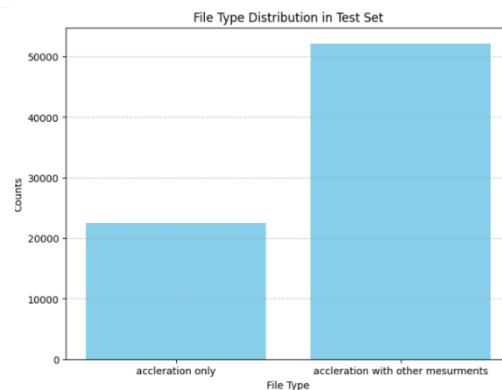
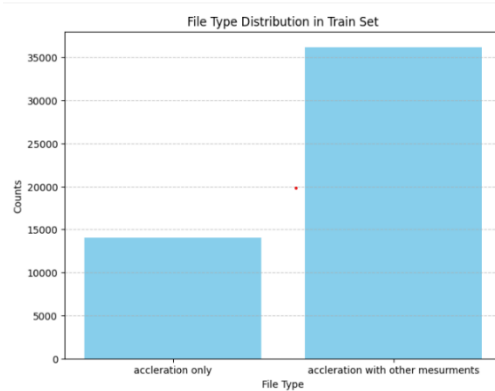
Both file formats contain time series data in 3 axis (x,y,z) with some having a sequence length of up to 4000 and some having only 3000.

The data isn't homogeneous, with some labels in the train data appearing 4578 times like walking freely and labels such as stairs down appearing only 1155 times, given that the mean is 2791, it is safe to say that the data is not homogeneous.

Label distribution in train:



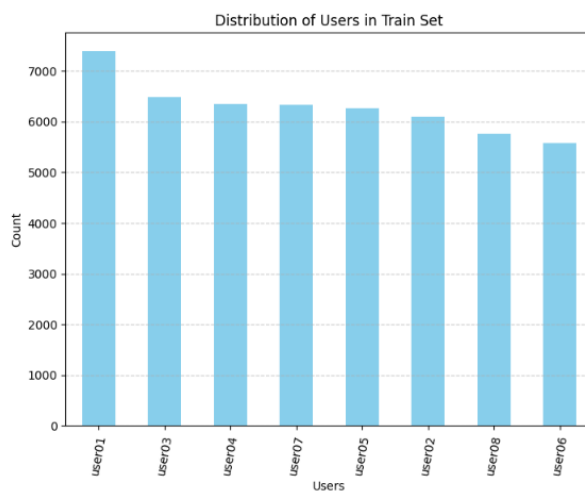
File type distribution in train and test:



Since it isn't written anywhere, we don't know exactly how the data was labeled but we can guess that the users were ordered to do the tasks (that are our labels) while wearing the sensors.

In our opinion, all the labels should be treated equally since we don't know how the distribution is in the test dataset, stairs down might have a lot more representation in the test dataset so we can't ignore it or give it less of a priority. In our training of the models we split the data randomly into training and validation so it is safe to assume that the label distribution is the same in both of them.

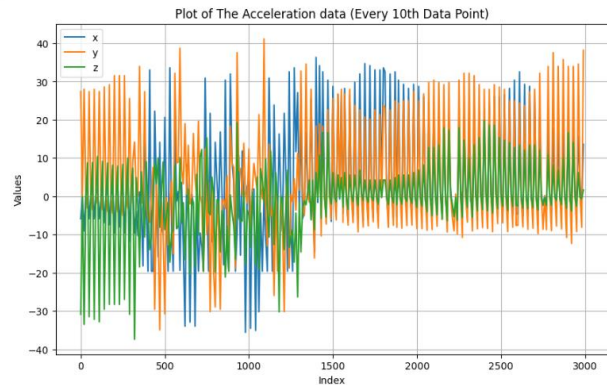
The data comes from 8 different subjects (users) where their contribution is almost equal with user01 having slightly more than the rest.



We don't exactly know how the data was split into train and test, but we know that the test data is larger than the train with the test having 74744 datapoints while the train has 50248 datapoints. However, we do know that there might be users in the test data that we haven't encountered in the training.

- b) Our task is classification, our goal is: given a sequence of accelerometer data, we want to know which one of the 18 activities the user was doing at that point in time. In hopes that after we get a good robust model we can use it to classify what the user is doing without them telling us. So in short, it is a classification task so that we can predict future events.

Plot of how the data looks:



c) The two tasks that we chose were: masking and encoding.

1) Masking:

We give the sequences to a model that masks some random parts of the data, then the model tries to predict what those masked segments were. After its training is complete, we do transfer learning, instead of passing the data to the last layer of our masking model, we feed it into our classification model.

2) Encoding:

We give the sequences to an autoencoder and use it as a feature extractor, as in we remove the last layer and feed the data into our classification model.

Our preprocessing:


Since we only took the accelerometer data from the files that had 3 different measurements, and since the sequence lengths weren't the same for all the files, we added padding to make them all of length 4000, we also added noise sampled from normal distribution with $\text{std}=0.01$, so that we don't over fit on the users themselves and make the model more generalized and robust.

Part 2:

a) For our validation strategy, we took 20% of the train data as validation data, we didn't use stratify split because the labels distribution isn't equal, so we wanted to keep the same distribution in the validation as in the train dataset.

Also, we didn't consider the users when splitting, since we handled the variation of the users in our preprocessing.

b) For our naïve solution, we chose to return the most common activity which is 'walking freely' as our classification for all the data, here is the test result from Kaggle:

 walking freely.csv
Complete · George Kanazi · 18s ago · walking freely

32.88883



The result is very bad which is not surprising.

c) The features that we extracted for our models were:

- Sequence length (without padding).
- We split each dimension of the sequences (after adding the noise) into 10 equal bins and for each bin we calculated the mean and the std.

So, in total we have 61 features that we fed into our ML models which were:

1- Logistic regression with L2 norm

```
➡ Training Acc: 0.3890, Validation Acc: 0.3774
```

2- Random forest with n_estimators = 100

```
➡ Training Acc: 0.9999, Validation Acc: 0.7811
```

3- XGBoost

```
➡ Training Acc: 0.9970, Validation Acc: 0.7747
```

Looking at the accuracies, random forest and XGBoost are both overfit while random forest had the best performance, we attribute these overfittings to the variety of the users and the data itself.

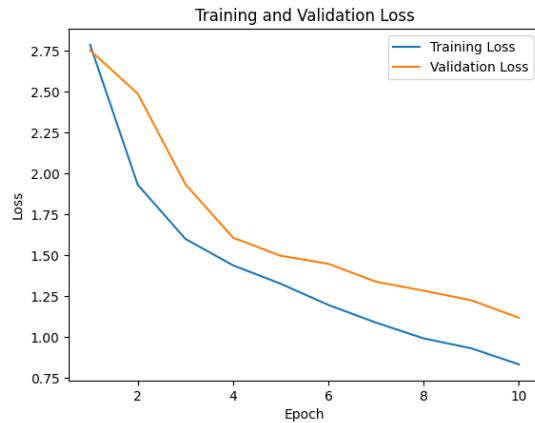
d) For our models, this is our GRU architecture:

```
GRUClassifier(  
    (gru): GRU(3, 32, num_layers=5, batch_first=True)  
    (fc): Linear(in_features=32, out_features=18, bias=True)  
)
```

The architecture stayed the same for both the normal and the improved versions of our models, we tried to change it by adding more layers, adding dropout, setting the initial weights using xavier uniform, using earlystopper, we even tried a Bidirectional GRU, however none of these changes improved the score in the Kaggle competition.

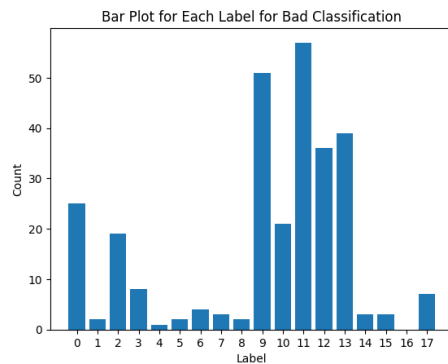
The main difference between the basic and the improved model was that we added a learning rate scheduler and that we trained the model on more epochs.

Simple GRU learning curve:

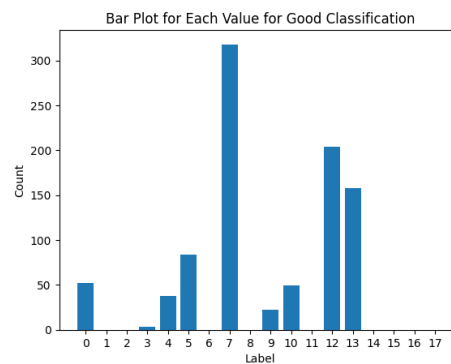


For the good and bad classifications, we created 2 lists, one for good and the other for bad classifications, we added a classification to good if it had a probability of over 0.93 and it was actually correct, and bad if it had a probability of less than 0.01 and was incorrect.

Bad:



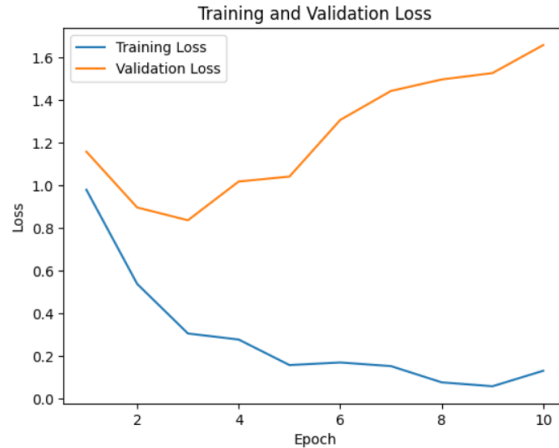
Good:



Looking at the graphs above, we can see that our model learned some labels much better than others, for example label 7 which is 'using phone' had almost no bad classifications and a lot of good classifications, while labels like 11 and 9 which are 'walking with handbag' and 'walking freely' respectively the model had a hard time learning.

The 1D-CNN model:

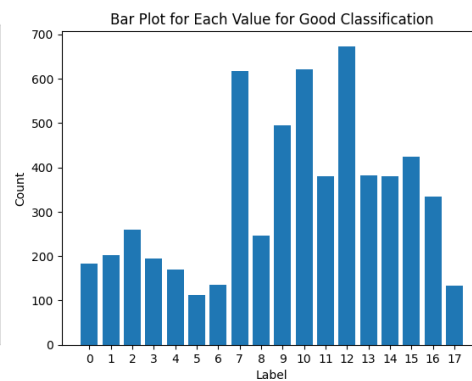
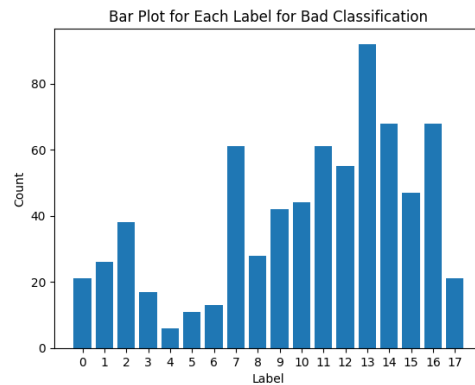
```
CNN1D(
  (conv1d): Conv1d(3, 64, kernel_size=(3,), stride=(1,), padding=(1,))
  (relu): ReLU()
  (conv1d2): Conv1d(64, 128, kernel_size=(3,), stride=(1,), padding=(1,))
  (conv1d3): Conv1d(128, 256, kernel_size=(3,), stride=(1,), padding=(1,))
  (pool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=512000, out_features=18, bias=True)
)
```



Here we set the threshold for bad to be 0.001 and for good to be 0.99.

Bad:

Good:



Looking at the graphs we can see that the model didn't necessarily have a problem with a specific class, the performances were the same across the board.

- e) We implemented the autoencoder suggestion as a self-supervised model for pretraining a GRU model for classification.

We first used the encoder part of the autoencoder for pretraining and then fed it into the GRU classifier.

The autoencoder architecture:

```
Autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=3, out_features=2, bias=True)
    (1): ReLU()
  )
  (decoder): Sequential(
    (0): Linear(in_features=2, out_features=3, bias=True)
    (1): Sigmoid()
  )
)
```

The autoencoder loss:

```
Epoch 1, Train Loss: 0.25259091478181994
Epoch 1, Vall Loss: 1.010313392278567
Epoch 2, Train Loss: 0.2362423291745616
Epoch 2, Vall Loss: 0.9449223032994056
Epoch 3, Train Loss: 0.23137598162510187
Epoch 3, Vall Loss: 0.9254578815289398
Epoch 4, Train Loss: 0.23021152750331553
Epoch 4, Vall Loss: 0.9208002967739579
Epoch 5, Train Loss: 0.22958764095639486
Epoch 5, Vall Loss: 0.9183048747428021
Epoch 6, Train Loss: 0.22920372812443926
Epoch 6, Vall Loss: 0.9167692998155432
Epoch 7, Train Loss: 0.22904100186303217
Epoch 7, Vall Loss: 0.9161184271532504
Epoch 8, Train Loss: 0.22879889029294995
Epoch 8, Vall Loss: 0.9151500290543286
Epoch 9, Train Loss: 0.2287840709039077
Epoch 9, Vall Loss: 0.9150907544472917
Epoch 10, Train Loss: 0.22874353349407525
Epoch 10, Vall Loss: 0.9149286128751081
```

The classifier architecture:

```
GRUModel(
  (gru): GRU(2, 32, num_layers=5, batch_first=True)
  (fc): Linear(in_features=32, out_features=18, bias=True)
)
```

The classifier loss:

```
Classifier Epoch 1, Train Loss: 0.08730029141086211
Epoch 1, Vall Loss: 0.08731330186128616
Classifier Epoch 2, Train Loss: 0.08506526921043242
Epoch 2, Vall Loss: 0.0846875011920929
Classifier Epoch 3, Train Loss: 0.0842891372942297
Epoch 3, Vall Loss: 0.08469255268573761
Classifier Epoch 4, Train Loss: 0.08423917668631696
Epoch 4, Vall Loss: 0.08462991565465927
Classifier Epoch 5, Train Loss: 0.0778668419574231
Epoch 5, Vall Loss: 0.06890568137168884
```

Even with such good results on the validation set, it failed miserably in the test, but not as bad as the 1DCNN.

	transfer.csv Complete · George Kanazi · 4h ago	2.27738	<input type="checkbox"/>
	1D-CNN.csv Complete · George Kanazi · 2d ago	3.91817	<input type="checkbox"/>
	GRUoutput.csv Complete · George Kanazi · 3d ago	1.82355	<input type="checkbox"/>

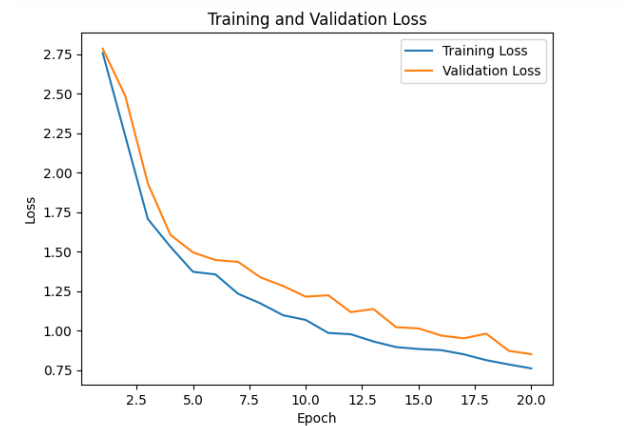
Somehow the GRU classifier after the encoder is overfitting a lot, we might attribute this error to the problem of the latent space of the encoder.

- f) For the GRU model we got good results and found that we were on the right track so the improvements that we suggested were to complicate the model further, by:
- 1- Adding more epochs.
 - 2- Adding a learning rate scheduler.
 - 3- Adding more layers.

However, for the CNN model, since we found that the model was overfitting a lot, we suggested adding:

- 1- Dropout.
- 2- Batch normalization.
- 3- Regularization.

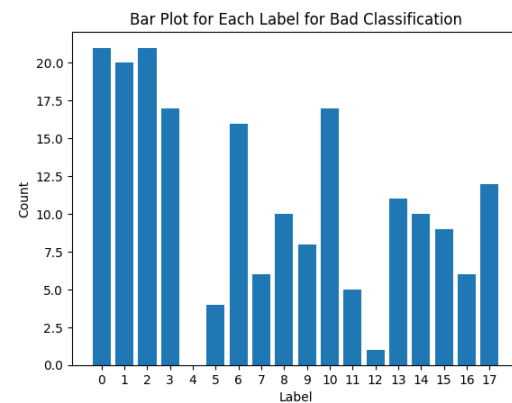
- g) For the GRU model we implemented our first 2 suggestions and got these results.
Improved GRU learning curve:



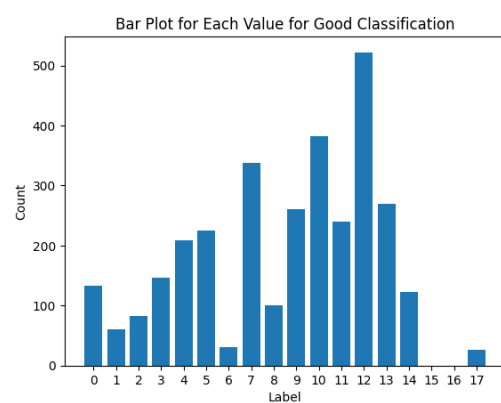
This is the model that we chose for the competition.

Here we set the good threshold to be 0.93 and the bad threshold 0.05.

Bad:



Good:



From the graphs above we can see that our model learned some classes slightly better than others, like classes 12 and 4 for example our model did exceptionally well which are

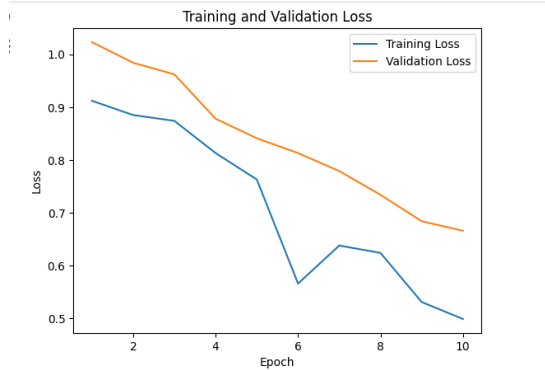
'stairs up' and 'walking with hands in pocket', while in classes 6, 15 and 16 it did bad, which are 'typing', 'washing mug' and 'washing plate' respectively.

For the CNN model after implementing our first two suggestions (which were Dropout and batch norm):

The improved CNN model architecture:

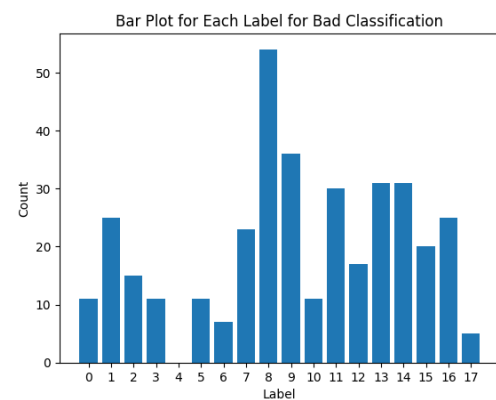
```
CNN1DIMP(
    (conv1d): Conv1d(3, 64, kernel_size=(3,), stride=(1,), padding=(1,))
    (bn1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv1d2): Conv1d(64, 128, kernel_size=(3,), stride=(1,), padding=(1,))
    (bn2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv1d3): Conv1d(128, 256, kernel_size=(3,), stride=(1,), padding=(1,))
    (bn3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (pool): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=512000, out_features=18, bias=True)
)
```

Learning curve:

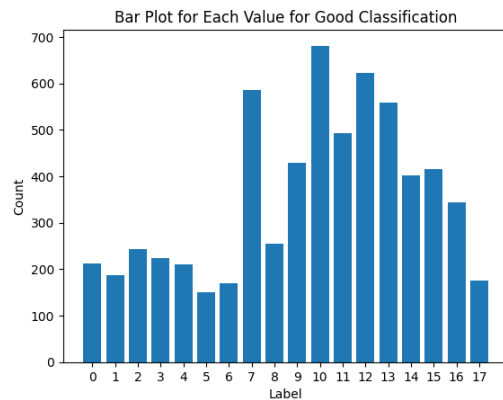


Here we kept the thresholds the same:

Bad:



Good:



Looking at the graphs, we can say that our model had problems learning class 8 and learned class 4 perfectly, while for the rest of the classes it did about the same more or less.

Model summary table:

Model name	Validation loss	Test loss
Simple 1D-CNN	1.658	3.91817
Simple GRU	0.924	1.82355
Self-supervised	0.068	2.27738
Improved 1D-CNN	0.651	2.89809
Improved GRU	0.838	1.62254