215452079: david podvinsky

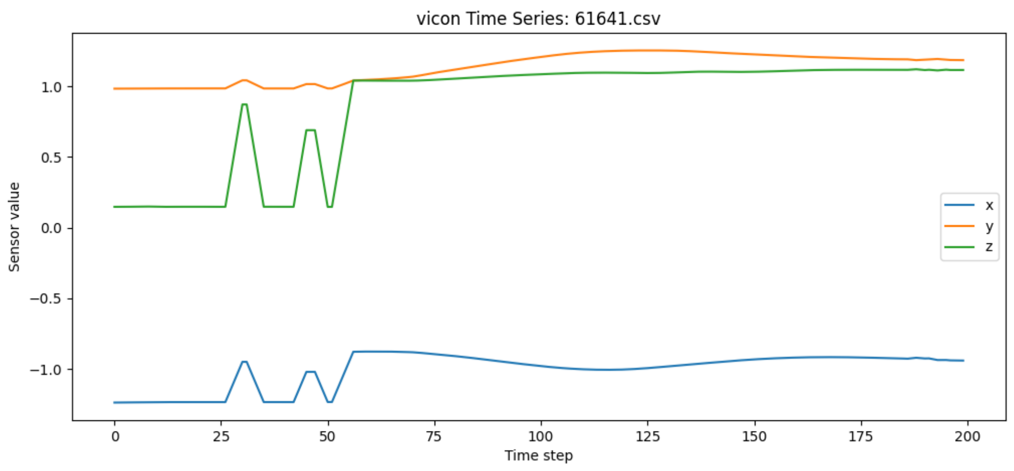
Task 1: EDA

1. We are provided 125k time series, 74744 of whom are unlabeled and whose data is in metadata.csv, and 50248 labeled data saved in train.csv.
   This dataset consists of multivariate time series.
   88373 are data from smart watches, 36619 are from vicon.
   Data from smartwatches is split into 3 types, acceleration whom we need, magnetometer and gyroscope who we don't need and will have to strip away.
   Data from vicons are pure x,y,z coordinates, whom we will have to either calculate acceleration for, or manage somehow else by having different encoders or maybe even 2 models for each type of data.
   Also a device can be on either side right or left, we will have to canonalize by mirroring the affected axis which is easy in the preprocessing step.
   106482 if the data comes from sensors on hands, 18510 comes from feet, we will probably have to inject body part in the last layer as an additional param to be able to tell them apart, as they behave differently.
   The data in train.csv is labeled by classes, meaning it's a classification problem, while in metadata.csv its unlabeled, so we will have to use it to pretrain a regression model to help it learn features before the actual classification training.
   All of that leads to the fact that the data is **heterogeneous** as our model need to learn how to treat different data, speciffecly foot and hand ts's.
   The labeled subset (train.csv) contains activity labels, e.g., walking, using phone, brushing teeth.
   Each TS in the labeled subset is assigned one activity class, so the supervised task is multiclass classification.
   All labels are treated as equally important for classification.
   Largest class: walking_freely (~4,578) Smallest class: stairs_down (~1,155) Ratio ~ 4:1, which is manageable.

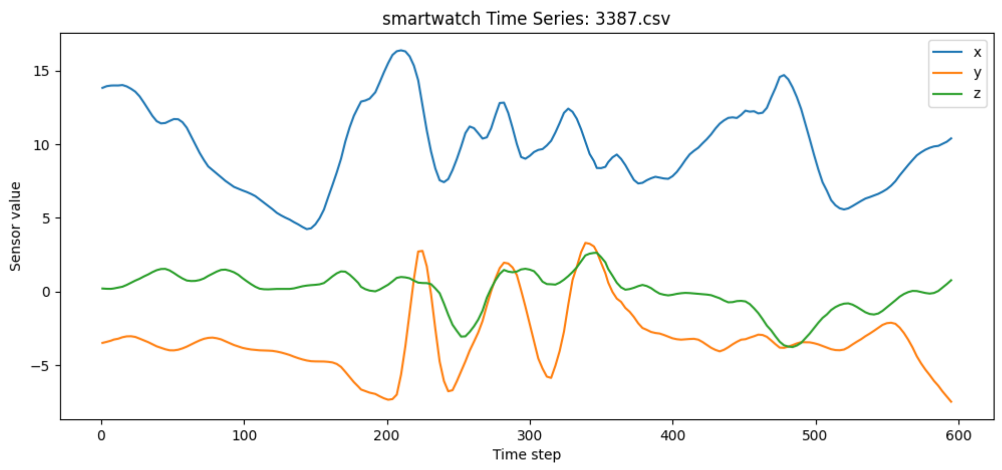
All labels are assumed validated, as they come from the competition dataset.
For better performance, class weighting or monitoring per-class metrics (F1-score) is recommended.
No idea what does do you mean by subjects in the data, there are 8 unique users, but they don't really split the data nor seem to have any correlation to the data, the test/train split is 75k to 50k, where the test set is metadata.csv, while the train set is smaller and in train.csv.

Another thing is, while all time series originatly fluctuate between 3k and 4k in length, because we only need around 1/3 of the data from smart watches, our data from them will be between 1k to 1.5k in length, which creates quite the problem as now the length difference between our data is huge.
AFTER PREPROCESSING COMING BACK HERE: some of the unlabeled data seem to include nan's which wasted around 3 hours of my life trying to find them trying to understand why my model predictions included nan.
So will have to also deal with that I guess
Maybe taking this course was a mistake as its not really my domain, or I should have waited at least until the last year where I would maybe have more experience with such thing.
As for the solutions, for each detected nan during preprocessing just duplicate the last know data and replace the nan

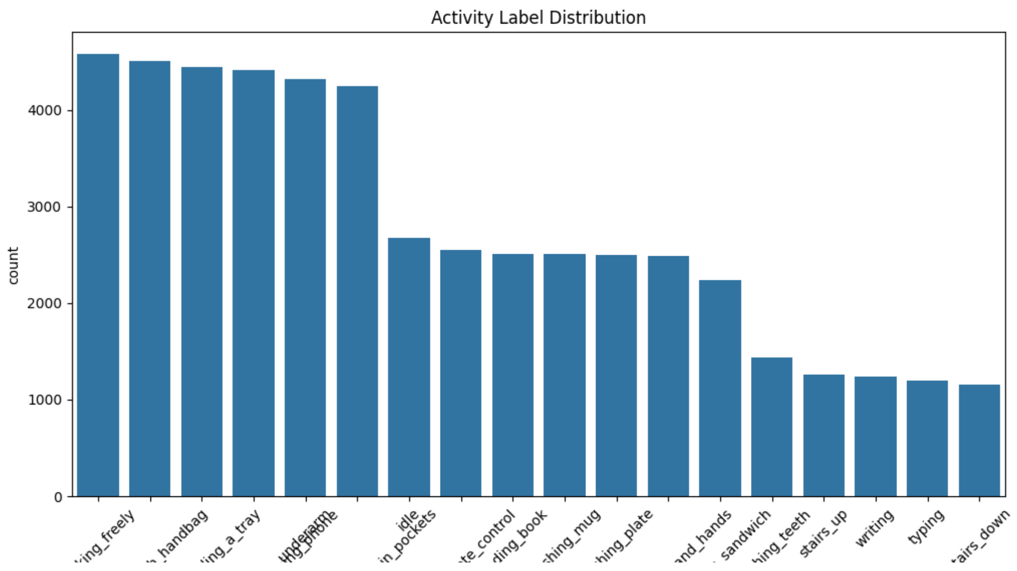
2.



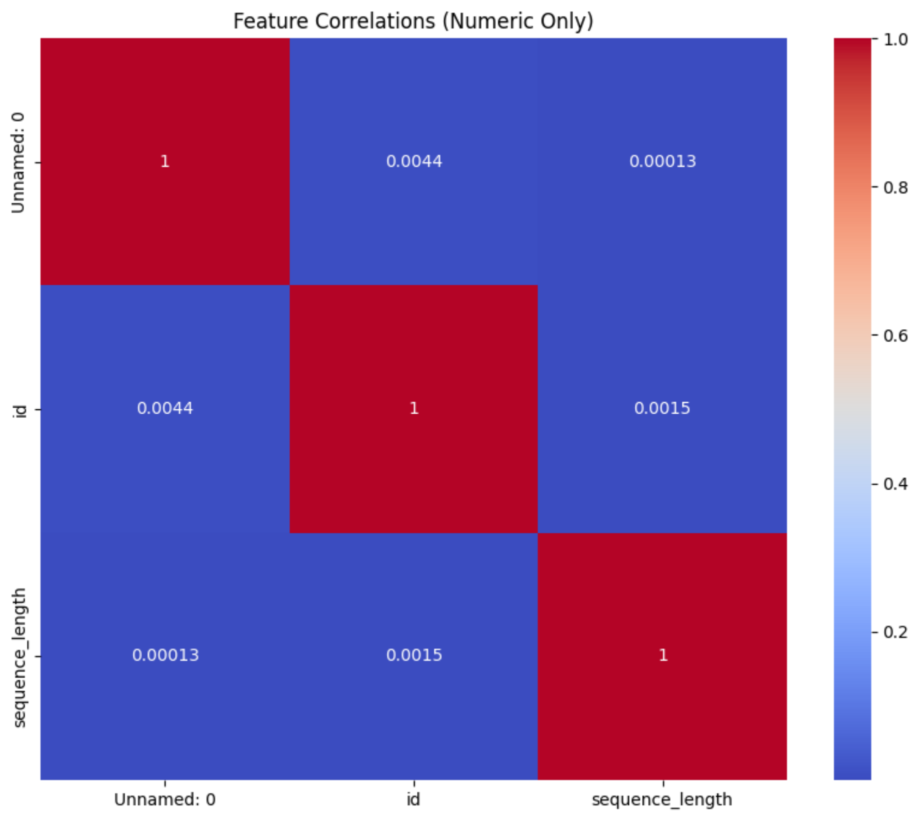
we are looking here a data snipset from a vicon on a hand while using the phone, we can see fluctuations probably while clicking on the phone in the data until coming to a steady phase probably just reading/watching the content.



here we are looking at a data from a smart watch on a hand while climbing up the stairs, we see here acceleration data, and we see a lot of jumps and downs mainly in the x and y axises, as the hand goes up and down, while the z axis remains mostly the same as the hand doesn't go side to side.



We see the labels spread, as from section 1 we can see an imbalance in the labels distributions.



gpt gave me this plot for whatever reason, we see that there is no observable correlation between the different numeric data, and im too lazy and inexperienced with pandas and data processing to actually give numbers to labels and try this once again without breaking anything.

3. We can use:
   Time Series Forecasting (Next-Step Prediction):
   We can use the unlabeled data as a bigger dataset to train a feature extractor/encoder, in order to speed up classification training in the future.
   Masked Reconstruction (Masked Prediction):
   Randomly mask parts of the time series and predict the missing values. This can help to learn contextual temporal features.

Before task 2: preprocessing of the data

125k small files is an i/o nightmare and adds a huge overhead, so we want to reduce each data split into 1 or a few bigger chunks to reduce i/o.

Also we face multiple problems: the first and the easiest to deal with is left vs right sides, we can simply flip the x axis to canonalize all the data.

Another problem which is manageable, is that our smartwatch data includes multiple measurement types, and we only want acceleration so we will have to strip it down, and then drop the measurement type colum.

After that we want to somehow deal with the data type mismatch between smartwatch that uses acceleration [m/s^2], vs the vicon that uses pure [m], so the first thing that comes to mind is for each vicon ts, calc the acceleration between each step.

Another problem is that after stripping, smartwatch data goes somewhere to 1.5k length while vicon remains around the 3.5k mark so we will have to deal with that, by something like padding and masking. A simple excel inspection gives us that max length is 4k.

Task 2: neural networks

1. It’s a classification problem of technically distinct and unrelated classes, so cross entropy should do the job.
   As for the split, I will simply split the training data like 75-25 where the last 25% will be used for validation, while the first 75% for training.
   For the pretraining however, ill just do forecasting so mse should do the trick as the loss function.
2. Ill go with Random-Guessing prediction as im a software engineer and I have no idea what any of them really mean, also unlike in assignment 1 where there was a giant bias of 65% for 1 class, here there isn’t as much of 1 class bias so guessing the more common class won't really work here. And since it’s a classification task, prediction based approaches like guess last wouldn’t fly here either, so random guesses is the most naïve way.

```
Train accuracy: 0.0556
Val accuracy:   0.0555
Train log loss: 2.8904
Val log loss:   2.8904
```
As expected,all I have to say is womp womp.

<u>3.</u> Ill use a random forest classifier, ill extract:

```
x.mean(),
x.std(),
x.min(),
x.max(),
np.ptp(x),
np.mean(np.abs(x)),
skew(x),
kurtosis(x)
```

and run:

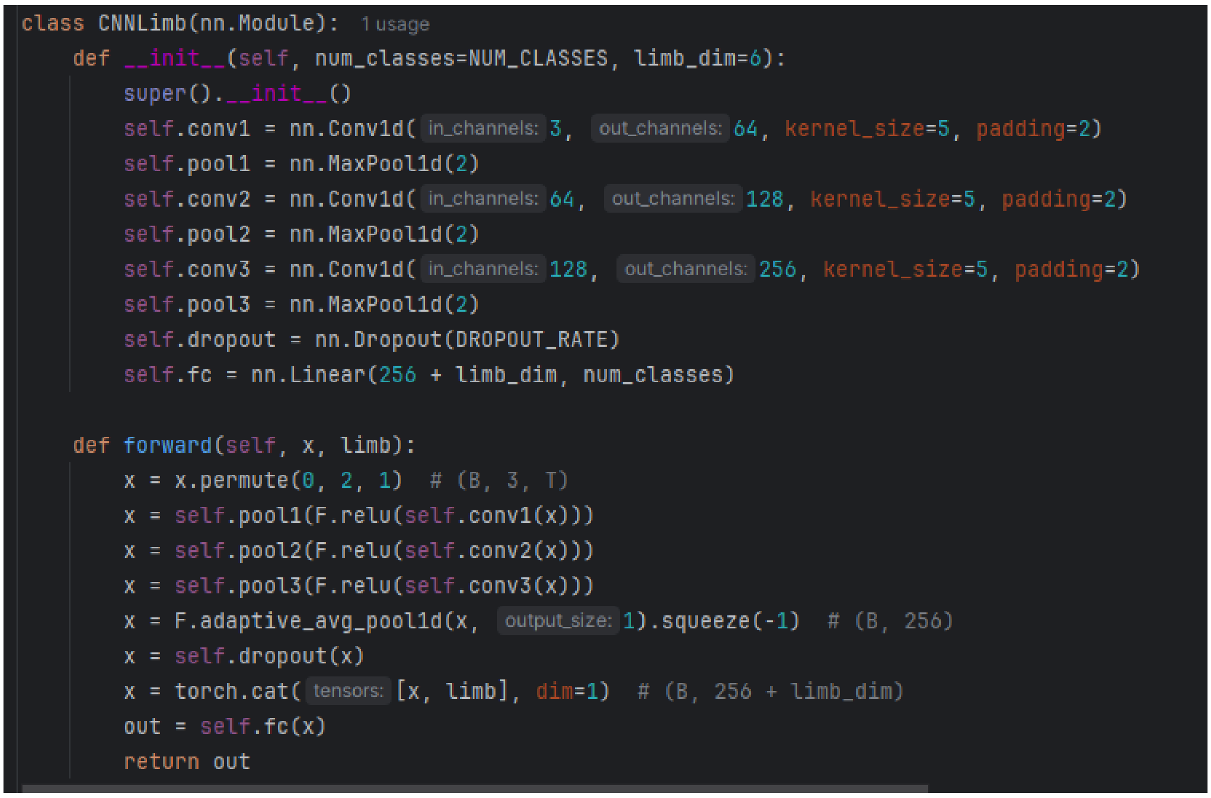
```
pipe = Pipeline([
  ("scaler", StandardScaler()),
  ("clf", RandomForestClassifier(
    n_estimators=300,
    random_state=42,
    n_jobs=-1
  ))
])
```

to get:

```
train accuracy: 1.0
train log loss: 0.06935059439913795
Validation accuracy: 0.9733333333333334
Validation log loss: 0.23821864617891111
```

Which is huge, raising the question of do I even need a DL.
After submission I got a score of 1.59795, which I guess is ok? It put me in third place, and its 2 times better than the random guesses.

<u>4. Creating a small 1d cnn:</u>



```
class CNNLimb(nn.Module):  1 usage
    def __init__(self, num_classes=NUM_CLASSES, limb_dim=6):
        super().__init__()
        self.conv1 = nn.Conv1d( in_channels: 3,  out_channels: 64, kernel_size=5, padding=2)
        self.pool1 = nn.MaxPool1d(2)
        self.conv2 = nn.Conv1d( in_channels: 64,  out_channels: 128, kernel_size=5, padding=2)
        self.pool2 = nn.MaxPool1d(2)
        self.conv3 = nn.Conv1d( in_channels: 128,  out_channels: 256, kernel_size=5, padding=2)
        self.pool3 = nn.MaxPool1d(2)
        self.dropout = nn.Dropout(DROPOUT_RATE)
        self.fc = nn.Linear(256 + limb_dim, num_classes)

    def forward(self, x, limb):
        x = x.permute(0, 2, 1)  # (B, 3, T)
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = F.adaptive_avg_pool1d(x,  output_size: 1).squeeze(-1)  # (B, 256)
        x = self.dropout(x)
        x = torch.cat( tensors: [x, limb], dim=1)  # (B, 256 + limb_dim)
        out = self.fc(x)
        return out
```

With hyper params:

BATCH_SIZE = 64   # batch size for training

NUM_EPOCHS = 30   # number of training epochs

LEARNING_RATE = 1e-3   # Adam optimizer learning rate

DROPOUT_RATE = 0.3

we get:

Epoch 1: Train Loss 1.7518, Acc 0.3693 | Val Loss 1.5125, Acc 0.4599

Epoch 2: Train Loss 1.4510, Acc 0.4744 | Val Loss 1.3455, Acc 0.5164

Epoch 3: Train Loss 1.3271, Acc 0.5246 | Val Loss 1.2529, Acc 0.5472

Epoch 28: Train Loss 0.4843, Acc 0.8277 | Val Loss 0.4276, Acc 0.8536

Epoch 29: Train Loss 0.4713, Acc 0.8321 | Val Loss 0.4196, Acc 0.8523

Epoch 30: Train Loss 0.4638, Acc 0.8348 | Val Loss 0.4127, Acc 0.8591

which is technically good, yet no idea if we overfit or not, also a very nice bonus is that each epoch takes around 3s, and the vram usage is around 1.6GB, meaning I can easly increase the batch size and employ kfold as its extremely fast.

Changing hyper params to:

```
BATCH_SIZE = 512        # batch size for training
NUM_EPOCHS = 30         # number of training epochs
LEARNING_RATE = 1e-4     # Adam optimizer learning rate
DROPOUT_RATE = 0.3       # dropout in CNN
NUM_CLASSES = 18
```

We now use 4.2 GB of vram, and I again forgot to include runtime analytics and the plots

We now get:

```
Epoch 1: Train Loss 2.0245, Acc 0.2986 | Val Loss 1.7197, Acc 0.3708
Epoch 2: Train Loss 1.6763, Acc 0.3897 | Val Loss 1.5492, Acc 0.4609
Epoch 3: Train Loss 1.5429, Acc 0.4441 | Val Loss 1.4525, Acc 0.4986
Epoch 28: Train Loss 0.8823, Acc 0.6891 | Val Loss 0.8211, Acc
0.7147
Epoch 29: Train Loss 0.8688, Acc 0.6944 | Val Loss 0.8152, Acc
0.7220
Epoch 30: Train Loss 0.8529, Acc 0.7008 | Val Loss 0.7882, Acc
0.7294
```

Which is noticeably worse, but its mainly because we have a lower learning rate so the model didn’t reach a good point yet.

Changing hyper params to:

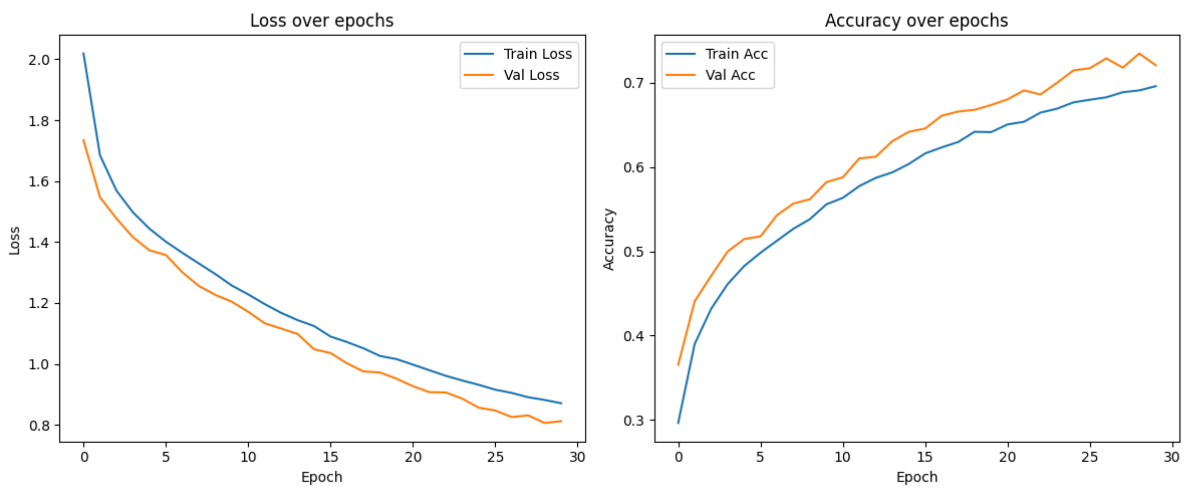
```
BATCH_SIZE = 512        # batch size for training
NUM_EPOCHS = 30         # number of training epochs
LEARNING_RATE = 1e-3     # Adam optimizer learning rate
DROPOUT_RATE = 0.3       # dropout in CNN
NUM_CLASSES = 18
```

We get:

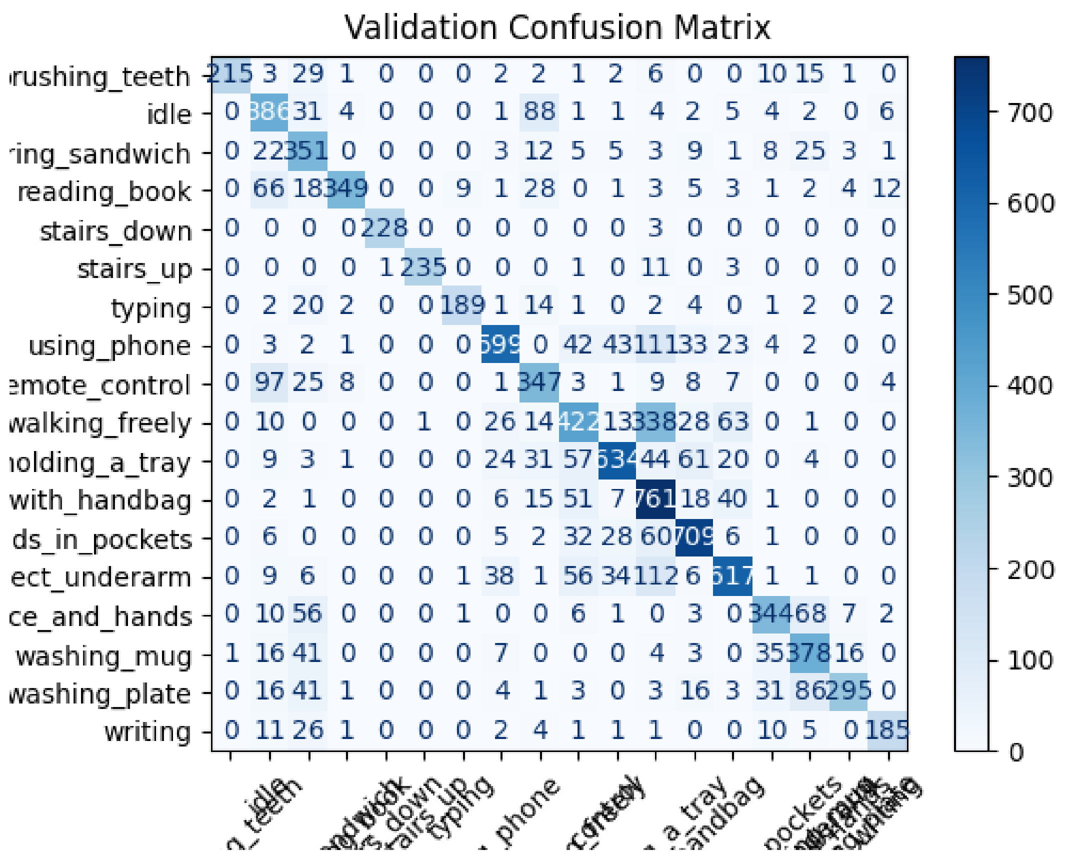
```
Epoch 1: Train Loss 2.0187, Acc 0.2962 | Val Loss 1.7341, Acc 0.3653

Epoch 2: Train Loss 1.6848, Acc 0.3899 | Val Loss 1.5469, Acc 0.4407

Epoch 3: Train Loss 1.5687, Acc 0.4317 | Val Loss 1.4771, Acc 0.4709

Epoch 28: Train Loss 0.8907, Acc 0.6889 | Val Loss 0.8310, Acc 0.7181

Epoch 29: Train Loss 0.8816, Acc 0.6911 | Val Loss 0.8062, Acc 0.7349
```

Epoch 30: Train Loss 0.8710, Acc 0.6961 | Val Loss 0.8121, Acc 0.7210

Which is still a bit on the higher end…



looks good, but since val comes from the same set, overfitting may pretty much carry on



Going back to: BATCH_SIZE = 64   # batch size for training
NUM_EPOCHS = 30   # number of training epochs
LEARNING_RATE = 1e-3   # Adam optimizer learning rate
DROPOUT_RATE = 0.3   # dropout in CNN
NUM_CLASSES = 18

We again see a great result:

```
Epoch 1: Train Loss 1.7386, Acc 0.3781 | Val Loss 1.4645, Acc 0.4819
Epoch 2: Train Loss 1.4273, Acc 0.4848 | Val Loss 1.3227, Acc 0.5271
Epoch 3: Train Loss 1.2995, Acc 0.5312 | Val Loss 1.2134, Acc 0.5715
Epoch 28: Train Loss 0.4752, Acc 0.8330 | Val Loss 0.4109, Acc
0.8618
Epoch 29: Train Loss 0.4397, Acc 0.8455 | Val Loss 0.3982, Acc
0.8654
Epoch 30: Train Loss 0.4388, Acc 0.8459 | Val Loss 0.3679, Acc
0.8785
```

Which is great, if I had to guess whats going on, id say that lower batch size allows better generalization, and allows us to more easly avoid local minimas leading to a more steady progress.

now ill submit the predictions to see where we stand.

Turns out my model outputs 1 number instead of the soft max, so need to change it and retrain.

I absolutely have gpt, im getting scientific notations and gpt sucks so much he cant help me get a proper prediction numbers.

Got a score of 3.8 which I assume is the worst possible score, but its most likely because of label encoding misalignment, leading to while an ok predictions in papers, outputting it under the wrong labels

Final run with a unified deterministic label encoder:

```
BATCH_SIZE = 64        # batch size for training
NUM_EPOCHS = 30         # number of training epochs
LEARNING_RATE = 1e-3     # Adam optimizer learning rate
DROPOUT_RATE = 0.3       # dropout in CNN
NUM_CLASSES = 18        # number of activity classes
```

Leads to:

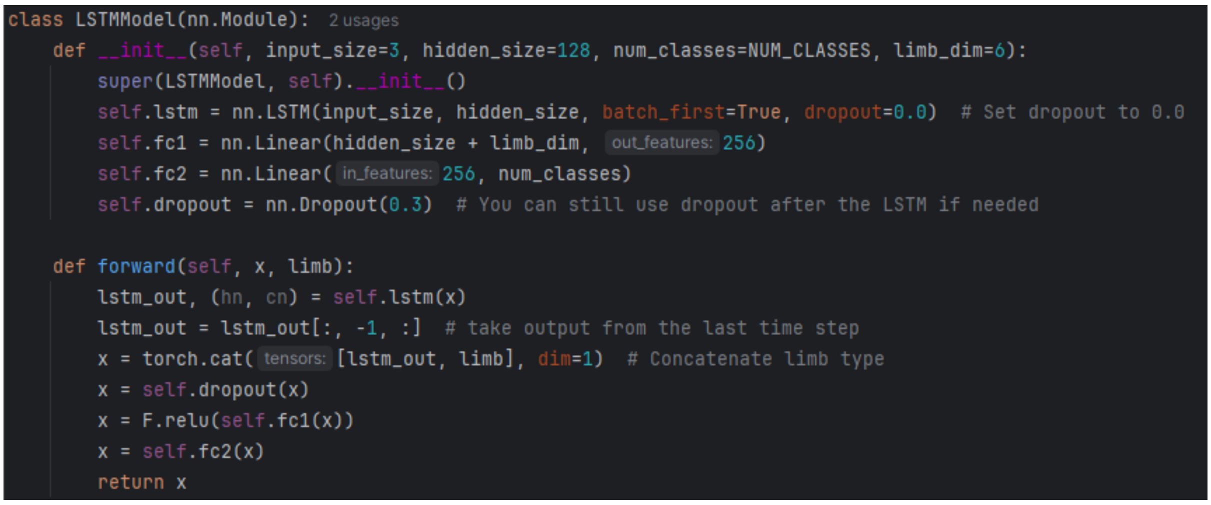
```
Epoch 1: Train Loss 1.7698, Acc 0.3656 | Val Loss 1.5339, Acc 0.4499
Epoch 2: Train Loss 1.4559, Acc 0.4766 | Val Loss 1.3179, Acc 0.5274
Epoch 3: Train Loss 1.3171, Acc 0.5224 | Val Loss 1.2024, Acc 0.5730
Epoch 28: Train Loss 0.4543, Acc 0.8388 | Val Loss 0.3996, Acc 0.8585
Epoch 29: Train Loss 0.4373, Acc 0.8449 | Val Loss 0.3902, Acc 0.8691
Epoch 30: Train Loss 0.4289, Acc 0.8492 | Val Loss 0.3716, Acc 0.8711
```

Which is insane, now we need to hope that we will get somewhere close with our submition.

Submission score is 3.54411, worse than before and I have no idea how to explain it, since its worse than random guessing

lowering learning rate to 1e-5 improved the score to 1.8, meaning we are overfitting

Single directional LTSM:



```
class LSTMModel(nn.Module):  2 usages
    def __init__(self, input_size=3, hidden_size=128, num_classes=NUM_CLASSES, limb_dim=6):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True, dropout=0.0)  # Set dropout to 0.0
        self.fc1 = nn.Linear(hidden_size + limb_dim,  out_features: 256)
        self.fc2 = nn.Linear( in_features: 256, num_classes)
        self.dropout = nn.Dropout(0.3)  # You can still use dropout after the LSTM if needed

    def forward(self, x, limb):
        lstm_out, (hn, cn) = self.lstm(x)
        lstm_out = lstm_out[:, -1, :]  # take output from the last time step
        x = torch.cat( tensors: [lstm_out, limb], dim=1)  # Concatenate limb type
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

BATCH_SIZE = 64

NUM_EPOCHS = 30

LEARNING_RATE = 1e-5

NUM_CLASSES = 18

Epoch 1: Train Loss 2.8762, Acc 0.0849 | Val Loss 2.8597, Acc 0.0859

Epoch 2: Train Loss 2.8102, Acc 0.0881 | Val Loss 2.7876, Acc 0.0901

Epoch 3: Train Loss 2.7883, Acc 0.0883 | Val Loss 2.7863, Acc 0.0898

Epoch 8: Train Loss 2.7862, Acc 0.0884 | Val Loss 2.7853, Acc 0.0902

The learning rate is obviously way too small, as we barely move at all, so early stopping and adjusting.

Also I would like to complain, how come batch 64 saturate 7.5 GB of vram, is slower and utilizes more gpu? Isn't ltsm supposed to be less parallelable leading to smaller gpu utilization, yet here it uses more gpu and still give slower speeds?

BATCH_SIZE =128

NUM_EPOCHS = 30

LEARNING_RATE = 5e-5

Epoch 1: Train Loss 2.8371, Acc 0.0901 | Val Loss 2.7867, Acc 0.0890

Epoch 2: Train Loss 2.7878, Acc 0.0894 | Val Loss 2.7856, Acc 0.0908

Epoch 3: Train Loss 2.7864, Acc 0.0907 | Val Loss 2.7851, Acc 0.0873

Epoch 8: Train Loss 2.7835, Acc 0.0889 | Val Loss 2.7824, Acc 0.0908

Still the same problem.

No matter what I try, what I change, even after making the model a lot more complex I get the same results, which indicate that most likely my pipeline is wrong somewhere, maybe I don't return raw logits or something(although it seems that the model should work fine and no ai was able to help me), and since I value my time and unwilling to kill hours training an arch that will likely be worse than the 1d cnn so ill leave it there.

5. Trying to fine tune tsai, turns out it's a relic of the past with no support for the latest pytorch, and im already very annoyed by this job since we need to work with so many relics of the pasts that simply refuse to work...

```
BATCH_SIZE = 128
NUM_EPOCHS = 50
LEARNING_RATE = 3e-4
DROPOUT_RATE = 0.3
NUM_CLASSES = 18
Epoch 1: Train Acc 0.3708 | Val Acc 0.4530
Epoch 2: Train Acc 0.5026 | Val Acc 0.4739
Epoch 3: Train Acc 0.5715 | Val Acc 0.5236
Epoch 4: Train Acc 0.6171 | Val Acc 0.5377
Epoch 5: Train Acc 0.6549 | Val Acc 0.4189
Epoch 6: Train Acc 0.6801 | Val Acc 0.1845
```

Great start and it wend south fast...

ill lower learning rate to 1e-5 and increase dropout to 0.5, lets see where it takes us.

```
Epoch 1: Train Loss 2.6804 | Train Acc 0.2112 | Val Loss 2.5019 | Val Acc 0.3136
Epoch 2: Train Loss 2.4191 | Train Acc 0.3018 | Val Loss 2.2841 | Val Acc 0.3279
Epoch 3: Train Loss 2.2318 | Train Acc 0.3223 | Val Loss 2.1093 | Val Acc 0.3618
```

For got to document las epochs, but for 22 epochs val loss was:1.17, Ill just tell the test score: 1.60623

which is almost as good as the ML, ill check if with an earlier epoch we get a better result to adjust for overfitting, and to know if I maybe need to train it further

21 epoch gives, val 1.20 : 1.59494

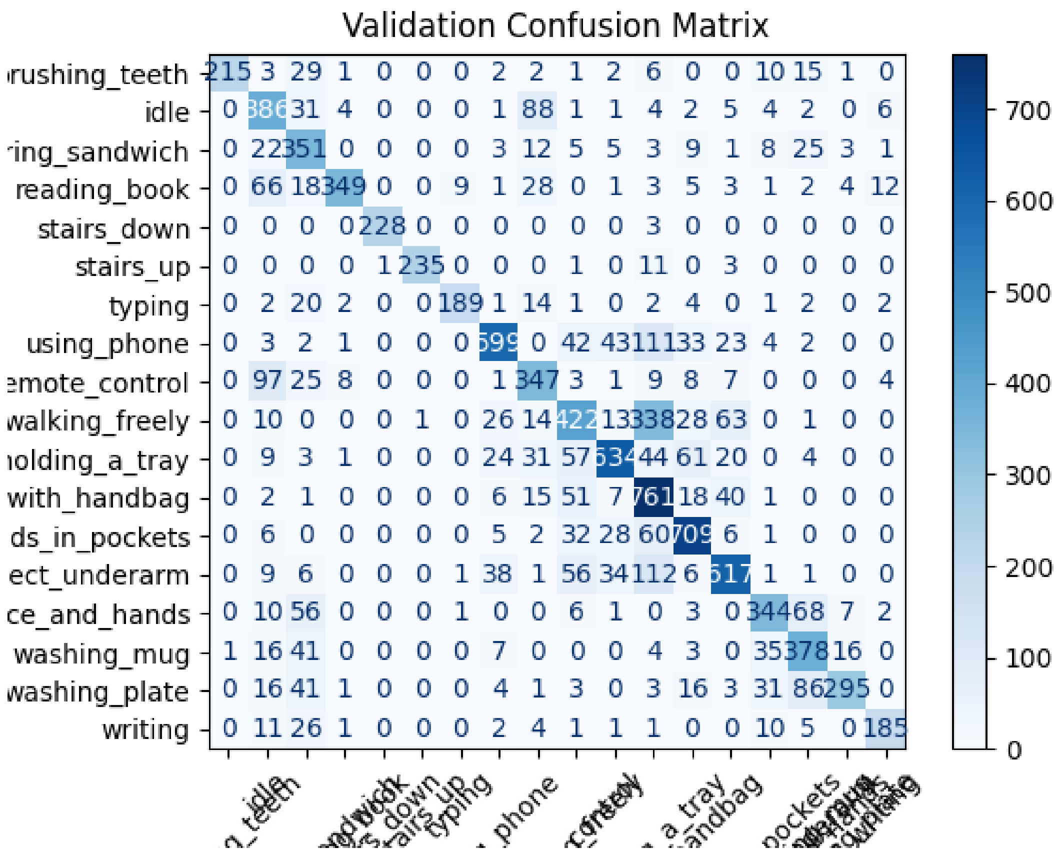
Meaning we most likely started entering the overfit territory considering the difference between the eval scores are 0.03.

Ill try with an even earlier epoch, it has val of 1.24, 19: 1.61929

So it seems like we have stopped in time.

Overall inception gave us the best score so far of 1.595 which puts it ahead of ML.

6. Since for almost all cases I have early stopped mannualy without any graphs, ill just use the graphs from my 1d cnn, which gets a score of 1.8 with early stopping before it grossly overfits:



we can see that its good at predicting steady hands like with objects in hands, where hands barely move and they all differentiate in hand orientation, and it struggles in places where hands move in a repeated pattern as most of those actions look the same, like writing and brushing teeth, where it’s a repeated move that look alike.

Improvements:

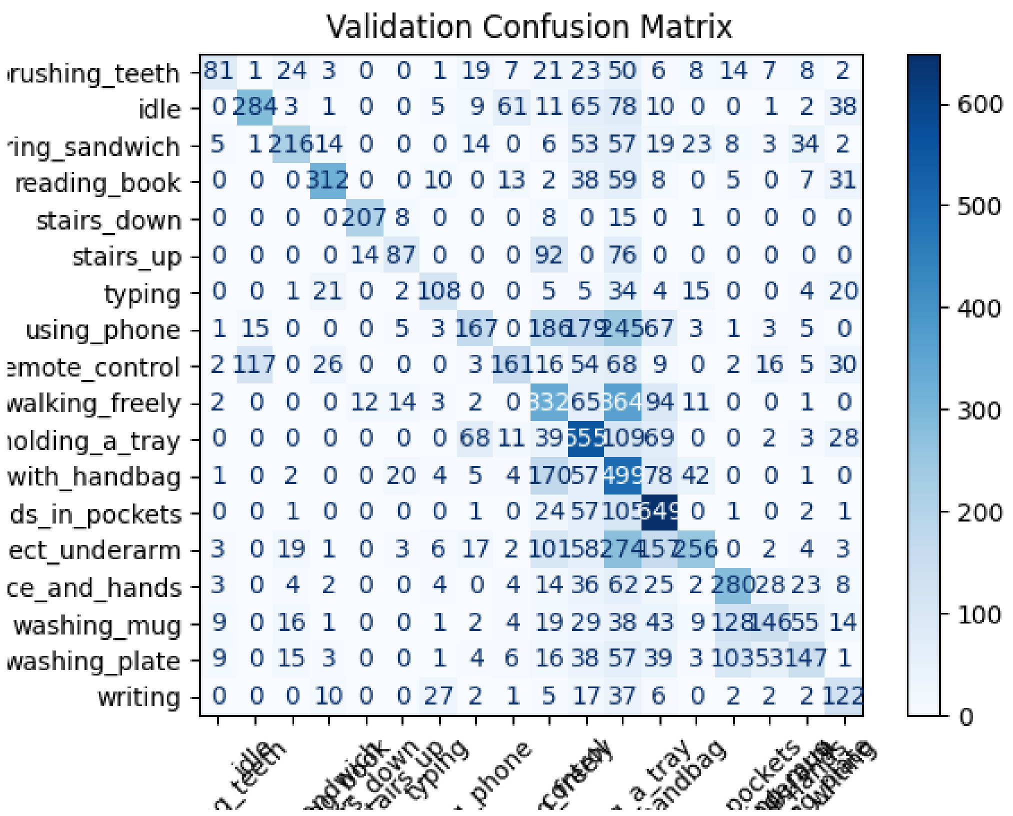
a. We can notice that we start overfitting very fast, meaning that data augmentation is a must.
b. While the data is not heavily biased towards 1 singular class, it is still quite unbalanced with some classes having 4 times less representation than other, so adding class weights may help identifying them.
c. We need more data, no matter how you look at it, 50k sequences is not really that much, especially for a 1d cnn that ill probably be using as it's the fastest.
d. Hyperparameter tuning will not hurt
e. Pretraining on the unlabeled data, and then fine tuning the classifer will probably help a lot especially to learn pattern before learning how to interpret them.

7. Adding augmentations in the form of: jittering, scaling and time wrapping.

```
BATCH_SIZE = 128       # batch size for training
NUM_EPOCHS = 50         # number of training epochs
LEARNING_RATE = 5e-5     # Adam optimizer learning rate
DROPOUT_RATE = 0.3       # dropout in CNN
NUM_CLASSES = 18
Epoch 1: Train Loss 2.6283, Acc 0.1591 | Val Loss 2.3891, Acc 0.2205
Epoch 2: Train Loss 2.3670, Acc 0.2140 | Val Loss 2.2190, Acc 0.2490
Epoch 3: Train Loss 2.2353, Acc 0.2392 | Val Loss 2.1176, Acc 0.2522
Epoch 48: Train Loss 1.5865, Acc 0.4375 | Val Loss 1.5093, Acc 0.4605
Epoch 49: Train Loss 1.5828, Acc 0.4382 | Val Loss 1.5065, Acc 0.4591
Epoch 50: Train Loss 1.5726, Acc 0.4424 | Val Loss 1.5063, Acc 0.4586
Runtime: 831.65 seconds
```

We do see consistent improvements up until the end, meaning that we need more epochs, also the improvements are somewhat slow from the very beginning, meaning too low of lr.

Got score: 1.87394, earlier epoch got score: 1.92125 meaning we are still in active learning.



BATCH_SIZE = 128   # batch size for training

NUM_EPOCHS = 100   # number of training epochs

LEARNING_RATE = 1e-4   # Adam optimizer learning rate

DROPOUT_RATE = 0.3   # dropout in CNN

NUM_CLASSES = 18

Epoch 1: Train Loss 2.5098, Acc 0.1839 | Val Loss 2.2195, Acc 0.2426

Epoch 2: Train Loss 2.2030, Acc 0.2406 | Val Loss 2.0518, Acc 0.2882

Epoch 3: Train Loss 2.0796, Acc 0.2689 | Val Loss 1.9547, Acc 0.3033

Epoch 87: Train Loss 1.2760, Acc 0.5534 | Val Loss 1.1734, Acc 0.5912

Epoch 88: Train Loss 1.2678, Acc 0.5562 | Val Loss 1.2128, Acc 0.5741

Epoch 89: Train Loss 1.2810, Acc 0.5549 | Val Loss 1.1883, Acc 0.5767

 --> No improvement for 3 epochs

Epoch 90: Train Loss 1.2942, Acc 0.5495 | Val Loss 1.1602, Acc 0.6004
Epoch 91: Train Loss 1.2662, Acc 0.5588 | Val Loss 1.1554, Acc 0.5976
Epoch 92: Train Loss 1.2578, Acc 0.5619 | Val Loss 1.1605, Acc 0.5993
Early stopped, still learning very slowly and we do see a strange pattern of train getting worse before a huge jump, which could indicate we are overfitting.
Score: 1.84339 still awful.

MODEL(best version)	Test score	Train score	Val score
1d-cnn pre adjustments	1.87233	0.8492	0.8711
LTSM/GRU pre adjustments	No idea	2.7835	2.7824
1d-cnn after augmentation +class wieghts	1.84339	1.2578	1.1605
LSTM/GRU	I didn't have the courage	To reattempt it as it killed to much time	And Im not enjoying it
RF+ hand extracted features	1.40447	0.4883	0.5824
Inception	1.595	Didn't save but probably on par with this 1.17	1.205
Inception feature extractor+	1.60918	0.2713	0.4214

xgboost classifier			

SUMMARY:

Im now sure of what path in life im not interested in as while it is interesting, its not something I enjoy doing, and I will probably just learn it as a useful skill/hobby like many other things I find interesting but not see them as a valid career path for me.

I did in the end try to get inception feature extractor + xgboost classifier, and the scores did get quite close between the 2, so probably if I train or find a proper feature extractor I will be able to feed better features to an ML that will yield a better score, but im really indifferent to it and lack the time.

Didn’t figure how to make the rnn run, no matter how or what I've tried, it simply refused to learn and it was painfully slow so I just gave up on this relic.

Didn’t like working with time series for 2 main reasons:

They are way harder to estimate performance on, since in sequential split train and val come from the same sample, it’s a lot harder to detect over fitting without a dedicated val/test set, but because the data is already small enough splitting it will not only require me to do even more data science, but also lose valuable data and considering all but the inception models overfit very quickly, I cant really afford having even a smaller train set.

The second reason is that I simply don’t care, im interested in DL simply because im interested in stable diffusion, image vision and image processing, and because im interested in how LLM's work and how to hack them, and I really don’t care for ts and I don’t see myself doing anything ts related as im more in cyber and vulnerabilities anyway.

Another thing I have to say is that LTSM/GRU use more compute power than 1d-ccn's, are a lot slower, use substantially more memory and are more hurtful to work with, and trying to figure out why neither an LTSM or a GRU agreed to work without even having a proper example having to relies on MML's for the code that also didn’t know whats going on was really a pain in the ass.

Hope other works will be image or LLM related and wont require me to suffer as much.

Probably my best way to get a higher score would be pretraining a proper feature extractor on the unlabaled data, fine tune it with the labeled to generalize it better, and finally use it as a feature extractor for and ensamble of RF+XGBOOST or something.