Mobile Computing — Lab#2 (Team) Write-up

Task #1 Writeup –

In terms of how we split the data into validation and training, we decided to do a 20/80 split. We choose this split because we were told from the Assignment that there are 174 Test Cases inside the /Test folder. Dividing this by 6, it would be on *average* 29 test per activity (special note on the average! We actually don't know that, but just as a quick exercise to see how much data we are working with). Then, if we look at (116) / (116 + 29) [Or in other words, the Labeled Data per activity divided by *on average* total per activity]= 0.8, this tells us roughly 80% of data is labeled and we can break that up as we choose. Since we have 20% for testing, we figured 20% for validation, and 60% for training would be adequate (helpful source talking about this: https://www.v7labs.com/blog/train-validation-test-set)

Task #2 Writeup –

First, the function predict_stat_thresh is just opening up dataframes, and renaming information so it's easy to work with (we tried to stick to naming convention of <direction><device><measurement> such as xleft_rot meaning Left Controllers, Rotation, X axis. We *largely* looked at variance, but there are some mean measurements. After this, we also defined some constants. Here are some descriptions:

- Looked to identify what is the *minimum* variance that is required in order to justify as at least meaningful (attempts to narrow down from people just slightly swaying as standing **perfectly** still might be difficult)
- Similarly, we also defined what is considered *high* variance, as well as *very_high* variance. As per name of variables, this is precisely to attempt to capture what is meaningful changes in an activity; and an activity that largely depends on rapid movements (respectively)
- Looked at what should be the upper *vertical* distance between headset and controllers when sitting (here we are exploiting the fact that when you sit, your arms aren't fully extended → the distance is instead just the torso of your body). Here we are looking at mean.
- Looked at is considered high *controller* distance variance. Here, we are attempting to capture activities that require the controllers to change distance *relative to each other*. This is different from when controllers move *in general*, because we can move controllers up and down at the same time while maintaining same relative distance (such as chopping), but there are some activities where the distance between controllers dynamically change
- Looked at what the minimum vertical variance in headset to determine that user is moving head up and down. Here we are attempting to capture how some activities (such as running), do require your head to go up and down, meanwhile others (such as standing or sitting) don't.

After defining constants we get to the meat of the algorithm::

1. First we identify the activity of STR based on whether or not the controllers are changing position *relative to each other* (referring to the 4th bullet above). STR is the only activity

that requires controllers to go toward the torso of a user, and then also outside the torso of the user so we rely on to identify this activity.

2. Afterwards, we look at the vertical variance in headset (last bullet above) since again, we identified that JOG is the only activity that actually requires users to be *engaging* in any movement related to nodding, bobbing, etc.
3. Afterwards, we noticed that for CHP at least one controllers should high variance (referring to 2nd bullet above) across *all* axes (x, y, and z).
4. Afterwards, to identify TWS we looked at the Headset Rotation for Y axis. Here we utilized the *very high* variance threshold (referring to the 2nd bullet above) since in this activity there is noticeable rotation. To ensure a bit more robustness though, we also checked that either the X or Z axis for Headset Rotation was also across the *minimum* variance (referring to the 1st bullet above).
5. To identify STD, we tried to draw on two cases we thought about (if we just relied on one, accuracy for STD and STD would get confused). Firstly, when people sit, there is a high chance that they put their controllers on their lap rather than hanging down their sides. However, that means that their hands will be closer than when they are standing (because if standing, then hands have to be at **minimum** the distance of your entire torso, meanwhile if sitting, then hands can be closer). The issue with just relying on this is that there are different sizes of people, so to compensate for that, we also utilized the vertical distance between headset and controllers (3rd bullet above).
6. Lastly, if none of the above cases identified an activty, then we defaulted to the value of SIT (since this is the only activity left)

One thing left to say with how we determined each activity, we used one of our Group Member's code from last week (Jose) to graph out data and identify trends in data. From those trends, we had an intuitive idea of where to start but we had to tune variables (for example, what is considered "high"). We tuned by looking at results, graphs, and raw means from different datasets but tried not to overtune as otherwise it could potentially be unhelpful for future cases (which is why most variables are in base of 100, or 5s).

EXTRA Information/modifition to file —-> We noticed that predict_stat_thresh.py was having to take in an input to run, but when we wanted to look at latency, there really wasn't a need to pass in an argument. As such, we modified file so that IF there are 0 arguments passed in when calling function, then it defaults to just calling our latency function that we created.

Following is part of output that occurs after calling <python3 Python/predict_stat_thresh.py> with no arguments!! (telling us all the latency information that we are wanting looking across all the validation set):

★ Currently looking at the activity: STD took 0.000365s on average for calculation
    with average accuracy:20.0 out of:23.0
★ Currently looking at the activity: SIT took 0.000396s on average for calculation
    with average accuracy:23.0 out of:23.0
★ Currently looking at the activity: JOG took 0.000365s on average for calculation
    with average accuracy:23.0 out of:23.0

- ★ Currently looking at the activity: CHP took 0.000373s on average for calculation
  with average accuracy:23.0 out of:23.0
- ★ Currently looking at the activity: STR took 0.000422s on average for calculation
  with average accuracy:23.0 out of:23.0
- ★ Currently looking at the activity: TWS took 0.000445s on average for calculation
  with average accuracy:21.0 out of:23.0

===> Total Accuracy is 96.377%

Task #3 Writeup –

Model Design:
For Task #3, we implemented and tested two different non-deep learning classifiers: a random forest classifier and a histogram-based gradient boosting classifier. To pre-compute the relevant model weights from the raw data, we created the file shallow_weights_accuracy_latency.py for our computations. The first function in this file, get_model_weights(), reads the CSV files from the training data and appends them to a dataframe. The train_test_split() function is used to split 80% of the data to be a training subset and the remaining 20% to be a testing subset. The array y is used to account for the activity labels of the lines appended to the dataframe. In our final solution, a histogram-based gradient boosting classifier is trained on the training data and then evaluated on the validation data using score() function. The trained classifier is then retrained on the entire dataset and saved using joblib.dump() into the model_weights.joblib file.

Model Selection:
In deciding on the learning classifier to use, we researched different classifiers and took into consideration factors including accuracy and applicability to the dataset (helpful sources we used: https://www.activestate.com/blog/top-10-python-machine-learning-algorithms/ and https://www.simplilearn.com/10-algorithms-machine-learning-engineers-need-to-know-article). Random forest classification and gradient boosting classification were chosen for their relatively high accuracy and their ability to handle large datasets. We learned that histogram-based gradient boosting classification is much faster than gradient boosting classification and therefore decided to use the former.

Model Explanation:
Random forest classification uses a collection of decision trees to classify an object based on its attributes. Each tree is classified and the tree “votes” for that class. The forest chooses the classification having the most votes. The classification starts off by using multiple trees with slightly different training data. The predictions from all of the trees are then averaged out, resulting in better performance than any single tree in the model.

Histogram-based gradient boosting first creates histograms of the data. Each histogram represents a subset of the data, and the histogram bin size and number of bins can be adjusted to control the trade-off between bias and variance. The performance of the weak model is evaluated and a new model is built to correct the errors made by the previous model. The predictions made by the previous model are updated to include the predictions made by the new model. The updated predictions are used to train the next model. This model is repeated so the overall model improves with each iteration. This can be more efficient than traditional gradient boosting, especially when working with large data sets, and can require less decision trees.

Model Comparison:
We compared the two classifiers in terms of algorithm precision for each activity, and the average precision and average latency across the entire validation set. This is calculated using the function accuracy_latency() in the shallow_weights_accuracy_latency.py file. A precision score of 1 indicates perfect accuracy and latency is set to equal the sum of training latency and inference latency. The results are shown in the table below:

	Gradient Boosting Classifier		Random Forest Classifier	
	Precision	Latency	Precision	Latency
STD	1.0000	0.223837s	1.0000	0.307465s
SIT	1.0000	0.154222s	1.0000	0.428894s
JOG	1.0000	0.171166s	1.0000	0.298418s
CHP	1.0000	0.227557s	1.0000	0.678103s
STR	1.0000	0.515507s	1.0000	0.386971s
TWS	1.0000	0.476301s	1.0000	0.336182s
Average	1.0000	0.294765s	1.0000	0.406005s

From our results, we observed that accuracy was comparably the same but latency was, on average, 38% greater using the random forest classifier. Therefore, we proceeded to use the gradient boosting classifier in calculating our model weights.

Task #4 Writeup –

For Task 4, our algorithm is quite simple. It starts by reading in the data from the .csv as a Pandas dataframe and calculating a few crucial values: the data length in seconds, and the number of iterations we should perform (equal to how many 3-second intervals with a 1-second sliding window). Subsequently, we take a sub-dataframe of the first 3 seconds, call our shallow ML function on this 3-seconds worth of data, then repeat on 3 seconds starting at 1 second (so the data is of length 3 seconds, from 1 second to 4 seconds of values). Each evaluation is stored in an array and this array is returned. We decided on this method because we believed it was the most efficient algorithm that we could feasibly implement. At any point in time, we only have two dataframes: the one with all the data, and one with a 3 second portion of the data. We were contemplating splitting the dataframe into a list of dataframes with each member of the list representing 3 seconds worth of data, however, this seemed as if it would store too much data. An alternative that we also considered was splitting the dataframe into a list of dataframes, each member of the list containing 1 second worth of data, and re-joining these dataframes to create 3-second dataframes, however, we thought that this might have been slightly less efficient than our current implementation due to the constant re-joining of dataframes.

In terms of accuracy, our algorithm was not quite what we expected. Considering that it is based on the shallow ML model in Task 3, we expected very high accuracy, however, the ML model struggled when given only 3 seconds of data. For JOG=>STR, the algorithm correctly identifies this, for STD => CHP, the algorithm detects JOG=>CHP, and for TWS=>SIT, the algorithm detects TWS the entire time. There do not appear to be any logical errors in our algorithm, thus we were unable to pin down the cause of these inaccuracies.

Task #5 Writeup

The algorithm works very well in detecting each activity. We had all team members test out, to account for how different users might have collected data in a different way (note that we also wanted to try and find some edge cases and it worked out: for example - we tested out trying to see if we slouched while standing vs. standing upright). However, it struggles sometimes transitioning from certain activities to others. For example, assume static activities are SIT and STD, and dynamic activities are CHP, JOG, STR, and TWS. Detecting a transition from a static activity to a dynamic activity is almost instantaneous, which is excellent. This is also the case when detecting a static activity to another static activity. However, detecting a transition from a dynamic activity to a static activity takes quite long (about 3 seconds). Detecting a transition from a dynamic activity to another dynamic activity can vary, but generally is around 2 seconds. This is what we would expect due to phenomena related to our model for activity detection from Task #2. Consider a case where we have 3 seconds worth of data, containing 1 second of a dynamic activity with 2 seconds of a static activity (either one can be performed first). Static activities do not have movement, while dynamic activities do have movement. Because the model for this task is based on thresholds on the **mean** and **variance** of different measurements, the threshold can be reached in this scenario (quick reference back from Task #2 description - there are thresholds set for Variances which we rely on to identify the activity). With this understanding, we can see how transitions from static to dynamic can be quick, but from dynamic to static are slow.

One potential remedy to this would be to place a higher weight on more recent data when computing mean and variance statistics. For example, we could double or triple the weight of the last 1.5 seconds as compared to the first 1.5 seconds. Doing so would allow for quicker transitions, as data that was collected more recently would dominate the average and variance calculations. Another potential remedy, though more susceptible to tuning rather than methodical changes, is to place an upper bound of what is *useful* variance (as mentioned above, going from a dynamic activity to static will result in a *massive* variance, but this

The accuracy is not that surprising, as each activity can be quite easily distinguished from the other when an in-depth analysis is performed and subsequently implemented. Nonetheless, it was still a great feeling when we were performing the activity and we saw that the activity detected so quickly, especially when transitioning in some scenarios.

Based on our experiences with activity detection in Labs 1 and 2, we propose an additional application leveraging real-time VR sensing data to detect healthy exercise forms, such as running and weight lifting. There are a plethora of potential applications of each of these types of activity detection, however, our focus will be on determining whether someone's form and posture are consistent, which would help us determine whether the exercise is being done in a healthy manner or not. When analyzing someone in a running state, we would first provide them with the VR controllers to collect data. Then, we would train an ML model based on "healthy" running forms, and then pinpoint specific attributes to look for with regards to bad running form. For example, someone with bad running form may lean slightly to one side, and we could detect this through rotation values and positional values of the controllers.

Now, if we focus on weightlifting, we would give the user the VR headset to track data. For example, if the user were doing squats, we would be focused on the pitch and roll axes to see if the individual is moving their head a lot or if their neck is tilted. This would likely be more focused on statistical analytical methods (i.e. detecting head tilting and shaking) as opposed to a shallow ML-based model that we would use for detecting poor running form.

We also thought of another application that would leverage real-time VR sensing data: first-person VR video games such as first-person shooters. These types of games involve crouching, jumping, leaning, running, and other activities. Quickly detecting activities is crucial, as users want to feel as if they are immersed in the game. This would leverage many of the similar activity detection mechanisms that we explored in this Lab, such as shallow ML and continuous detection.

Contributions:

Matt:

Task 4: All of Task 4, except it had some code copied from task 3 to run the shallow ML model.

Task 5: All of Task 5, except all of Task 5 code related to the activity detection model was copied from Task 2 and reinterpreted into C#. Thus, Task 5 new code mainly focused on implementing the sliding window.

Michelle:

Task 3 & Debugging: Testing two different Machine Learning models (random forest and histogram gradient boosting classifier)

Jose:

Task 1 & 2 - Code for Task 2 was used later on in Task 5 as well as giving an insight into what type of columns we wanted for Training Model (and what columns we did not)