Lab 2 Report

Kyu Park, Dingyi Sun, Kendrick Xie

1. In our `creation_validation_set.py`, the function looks like this:

```python
# Divides the CSV files in Data/Lab2/Labeled into Train or Validation directory
given the ratio.
# Assumes all CSV files are valid files named correctly (ACT_###.csv).

# The ratio is a ratio of train to validation. Defaults at 0.8:
# 80% of the data set will be classified for training, 20% for validation.

# Included optional checking procedure to ensure all activities are included in
# both training and validation sets.
def create_validation_set(ratio:float = 0.8, check:bool = True):
```

   Basically, what it does is given a ratio (defaults at 0.8 since it is known as an ideal ratio), we randomly distribute the csv files into the Train or Validation directory. CSV files are not modified, and keep the same name and the same contents. We also do optional checks to ensure that both Train and Validation directories have at least one activity CSV.

   Simply running `create_validation_set()` can automatically distribute the files to the predefined Train and Validation directories.

   (Note that we assume that this file is run in Python directory, not the main directory, given our predefined directory route)

2. The algorithm design is already described and implemented in Lab1. In Lab2, we used exactly the same code in classify.py, but updated the definition of "standard vectors" using the data from the training set. We re-adjusted the weights for a few attributes such as head_pos_var to improve its accuracy.

   Using the same evaluation method described in detail in task 3, we characterize the performance of the statistical thresholding method:

	accuracy	precision	recall	f1-score	latency
Overall	0.6137	-	-	-	-
JOG	0.7189	0.79	0.79	0.77	~ ms

OHD	0.7312	0.67	0.66	0.74	~ ms
SIT	0.5320	0.52	0.52	0.70	~ ms
STD	0.5176	0.51	0.51	0.71	~ ms
STR	0.7233	0.77	0.77	0.75	~ ms
TWS	0.8041	0.82	0.82	0.79	~ ms

Overall, it took about 3 seconds to classify the whole entire Validation sets, which contains around 173 sets, thus each set taking around 0.2 seconds for the classification.

3. We model the paper “Sensing Meetings Mobile Social Networks: The Design, Implementation and Evaluation of the CenceMe Application” for our ML model, and use mean, variation and peaks of significant features we have decided from the statistical threshold method from Lab 1. We use a decision tree classifier. We use such a design because the paper clarified that mean, variation and peaks would yield promising results if used in decision trees, and we believed we could also yield promising results if we use our significant features from Lab 1. The algorithm to construct a decision tree from Training sets created from Part 1 is as follows:

Python

```
# returns a Decision Tree Model
# using features selected from classify.py
# feel free to modify features -Kyu
def learn() -> tree.DecisionTreeClassifier:
   train_df = createDF()
   columns = train_df.columns

   # features from classify.py; I included all mean, variance, peaks for these
-Kyu
   features_cols = [columns[num * 3 + alpha] for alpha in (0,1,2) for num in
(6,7,8,15,21,33,27,19,30)]

   X = train_df[features_cols]
   y = train_df.category
   clf = tree.DecisionTreeClassifier()
   clf = clf.fit(X, y)

   # valid_df = createDF("Data/Lab2/Validation/")
   # x_valid, y_valid = valid_df[features_cols], valid_df.category
   # y_pred = clf.predict(x_valid)
   # print("Accuracy:",metrics.accuracy_score(y_valid, y_pred))

   return clf
```

However, unlike our features for Lab 1, we use all means, standard deviations and peaks for features we thought were significant, regardless of what they were initially chosen for. For example, `head_pos.y` was selected for its mean; we not only included its mean in the decision tree but also standard variation and peaks, hoping that including more statistics of significant features would improve accuracy (which it did).

We trained the decision tree using the Training sets from Part 1, and validated the accuracy and precision using the Validation sets from Part 1. The result is as follows:

	accuracy	precision	recall	f1-score	support
JOG	0.96969697	0.97	0.97	0.97	33
OHD	0.96	0.89	0.96	0.92	25
SIT	1	0.88	1	0.94	23
STD	0.91304348	1	0.91	0.95	23
STR	0.76470588	0.93	0.76	0.84	34
TWS	1	0.92	1	0.96	34
Avg accuracy	0.93			0.93	172
macro avg		0.93	0.93	0.93	172
weighted avg		0.93	0.93	0.93	172

Overall, we saw 93% average accuracy and 93% precision in predicting the Validation sets, validating that our decision tree method is effective in predicting each activity. Training with the Training sets and predicting Validation sets takes around 3.5 seconds, so around 0.2 seconds per set.

4. We used the same algorithm from classify.py. The only difference is that it does not calculate the standard vectors itself, but read a "/Assets/Resources/standard.txt" file that contains the standard vector for each activity output by the python programs in task 2. This is done in Start().

   We added a few new methods to the ActivityDetection class as helper functions:

   - GetData: extract data from `attributes` into a `updatedData`, a list of list of floats to keep a history of real time data from the current VR session
   - ProcessData: turn the list above into a list of mean and var like we did in classify.py
   - Classify: calculate the distance between standard and current values, find the smallest

Each frame, the GetData method is called to extract the real time VR data from the current frame. Then, every 2.9 seconds, the GetCurrentActivity method uses the ProcessData and Classify methods above to produce a prediction. Finally, the lists of floats in `updatedData` are cleared so each new guess is based on the activity from the most recent 2.9 seconds.

Jogging, arms stretching, arms overhead, and twisting were almost always identified correctly. Sitting was often confused with arms overhead or jogging and standing was often confused with jogging. These results are close to what we expected since sitting and standing had the lowest accuracy, precision, and recall in Task 2. Our algorithm works best at identifying distinct movement rather than different stationary poses.

Based on our experience with activity detection in Labs 1 and 2, we believe that activity detection could effectively distinguish between exercises that involve movement. We plan to implement this idea in our course project in which we aim to create an exercise assistant. Another application that could take advantage of activity detection are VR videogames. For example, a magic videogame could trigger different spells based off of player motions.

Contribution:

Kyu Park: Task 1, Task 3, little bit of Task 4 (all Unity, no contribution to algorithm)

Dingyi Sun: Entire task 2. GetCurrentActivity and associated helper methods in task 4.

Kendrick Xie: Task 4 implementation, testing, and debugging.

Note: Isaaq Khader has one push on our branch which was due to the wrong email being logged in on GitHub Desktop on the CSIL windows PC.