SpringBoard Capstone Project 2: Human Activity Recognition Project Report

Student: Sofy Weisenberg
Date: 06/27/20

This Capstone Project milestone report will cover the following project sub-topics:

- Background
- Data Set
- Data Preprocessing/Feature Engineering
- Exploratory Data Analysis
    - Plotting Example Sensor Data
    - Visualizing Separability of Classes
        - PCA
        - tSNE
- Statistical Analysis
    - Aggregating Data
    - Selecting Features of Interest
    - Visualizing Distributions of Aggregated Feature Statistics
    - Bootstrap Hypothesis Testing
        - Motivation
        - Performing Hypothesis Tests
- Machine Learning Preprocessing
    - Splitting the Data: Training and Test Sets
    - Feature Scaling
    - Categorical Class Encoding
- Fitting an SVM Model
    - Hyperparameter Tuning
    - Model Evaluation
    - Visualizing Misclassified Activities
- Fitting a Neural Network Model
    - Importing the Raw Data
    - Preprocessing
    - Fitting an RNN Model
    - Fitting Additional RNN Models with Convolution
    - Comparison to SVM
- Conclusions

Background

Human activity recognition (HAR) has become an in-demand capability for several growing applications over recent years. The reduced size of accelerometers and gyroscope components have made them standard in most smartphone and wearable computing devices. Additionally, sensor data collection has become less expensive (higher framerates, more channels) and data

storage options have grown (use of cloud-based storage, etc). Sensor-based activity recognition researchers believe that by empowering ubiquitous computers and sensors to monitor the behavior of agents (under consent), these computers will be better suited to act on our behalf. For example, these may include monitoring of activities of those suffering from chronic diseases (such as diabetes or COPD) or monitoring sleep and exercise patterns.

Data Set

- Data was gathered for 30 subjects, aged 19-48 years old. Each subject performed 6 activities while wearing a smartphone (Samsung Galaxy S2) on their waist: laying, sitting, standing, walking, walking upstairs, and walking downstairs.
- The smartphone contains an accelerometer and a gyroscope for measuring 3-axial linear acceleration and angular velocity respectively at a constant rate of 50 Hz, which is sufficient for capturing human body motion.
- The data was manually labeled using corresponding video recordings of the experiments.

Data Preprocessing/Feature Engineering

The dataset available through Kaggle has already been processed and organized into a format usable by various machine learning techniques. Specifically the following steps were performed:

- The sensor signals (accelerometer and gyroscope) were pre-processed by applying noise filters and then sampled in fixed-width sliding windows of 2.56 sec and 50% overlap (128 readings/window).
- The sensor acceleration signal, which has gravitational and body motion components, was separated using a Butterworth low-pass filter into body acceleration and gravity. The gravitational force is assumed to have only low frequency components, therefore a filter with 0.3 Hz cutoff frequency was used.
- From each window, a vector of features was obtained by calculating variables from the time and frequency domain.

This resulted in a dataset with the following attributes provided for each record:

- Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
- Triaxial Angular velocity from the gyroscope.
- A 561-feature vector with time and frequency domain variables.
- Its activity label.
- An identifier of the subject who carried out the experiment.

Additional feature engineering details are provided for this dataset on the UCI website (features_info.txt from the downloadable data):

The features selected for this database come from the accelerometer and gyroscope 3-axial raw signals tAcc-XYZ and tGyro-XYZ. These time domain signals (prefix 't' to denote time) were captured at a constant rate of 50 Hz. Then they were filtered using a median filter and a 3rd order low pass Butterworth filter with a corner frequency of 20 Hz to remove noise. Similarly, the acceleration signal was then separated into body and gravity acceleration signals (tBodyAcc-XYZ and tGravityAcc-XYZ) using another low pass Butterworth filter with a corner frequency of 0.3 Hz.

Subsequently, the body linear acceleration and angular velocity were derived in time to obtain Jerk signals (tBodyAccJerk-XYZ and tBodyGyroJerk-XYZ). Also the magnitude of these three-dimensional signals were calculated using the Euclidean norm (tBodyAccMag, tGravityAccMag, tBodyAccJerkMag, tBodyGyroMag, tBodyGyroJerkMag).

Finally a Fast Fourier Transform (FFT) was applied to some of these signals producing fBodyAcc-XYZ, fBodyAccJerk-XYZ, fBodyGyro-XYZ, fBodyAccJerkMag, fBodyGyroMag, fBodyGyroJerkMag. (Note the 'f' to indicate frequency domain signals).

These signals were used to estimate variables of the feature vector for each pattern:
'-XYZ' is used to denote 3-axial signals in the X, Y and Z directions.

- tBodyAcc-XYZ
- tGravityAcc-XYZ
- tBodyAccJerk-XYZ
- tBodyGyro-XYZ
- tBodyGyroJerk-XYZ
- tBodyAccMag
- tGravityAccMag
- tBodyAccJerkMag
- tBodyGyroMag
- tBodyGyroJerkMag
- fBodyAcc-XYZ
- fBodyAccJerk-XYZ
- fBodyGyro-XYZ
- fBodyAccMag
- fBodyAccJerkMag
- fBodyGyroMag
- fBodyGyroJerkMag

The set of variables that were estimated from these signals are:

- mean(): Mean value
- std(): Standard deviation
- mad(): Median absolute deviation

- max(): Largest value in array
- min(): Smallest value in array
- sma(): Signal magnitude area
- energy(): Energy measure. Sum of the squares divided by the number of values.
- iqr(): Interquartile range
- entropy(): Signal entropy
- arCoeff(): Autorregresion coefficients with Burg order equal to 4
- correlation(): correlation coefficient between two signals
- maxInds(): index of the frequency component with largest magnitude
- meanFreq(): Weighted average of the frequency components to obtain a mean frequency
- skewness(): skewness of the frequency domain signal
- kurtosis(): kurtosis of the frequency domain signal
- bandsEnergy(): Energy of a frequency interval within the 64 bins of the FFT of each window.
- angle(): Angle between two vectors.

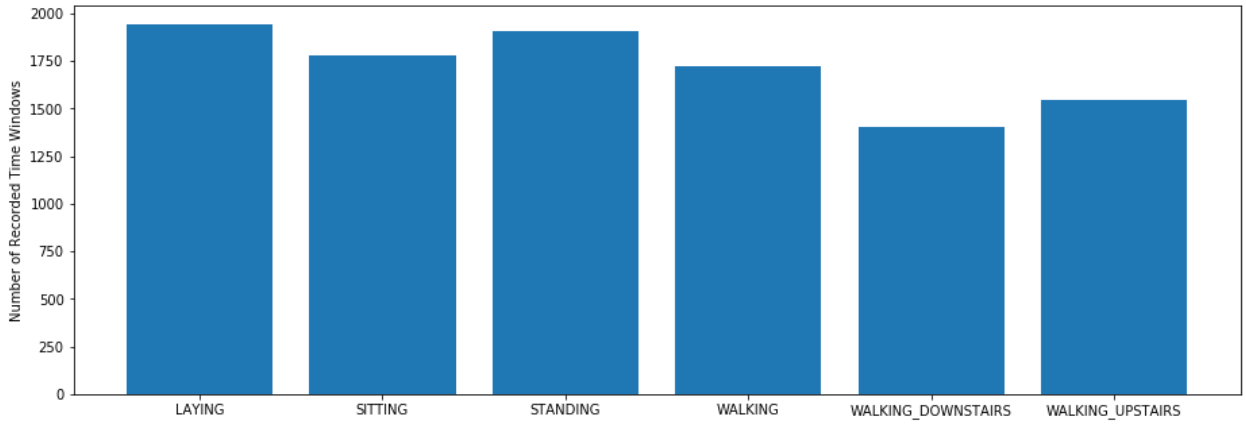
Additional vectors were obtained by averaging the signals in a signal window sample. These are used on the angle() variable:

- gravityMean
- tBodyAccMean
- tBodyAccJerkMean
- tBodyGyroMean
- tBodyGyroJerkMean

Exploratory Data Analysis

Loading the dataset, it can be seen that there are a total of 10299 rows and 563 columns, with no missing values. The feature data is all data type float64, and the last 2 columns are the subject number (int64) and the activity class (object).

The various activity classes are approximately evenly distributed, since each subject performed each of the six activities. This means that for classification modeling, this is a balanced (rather than an imbalanced) dataset.

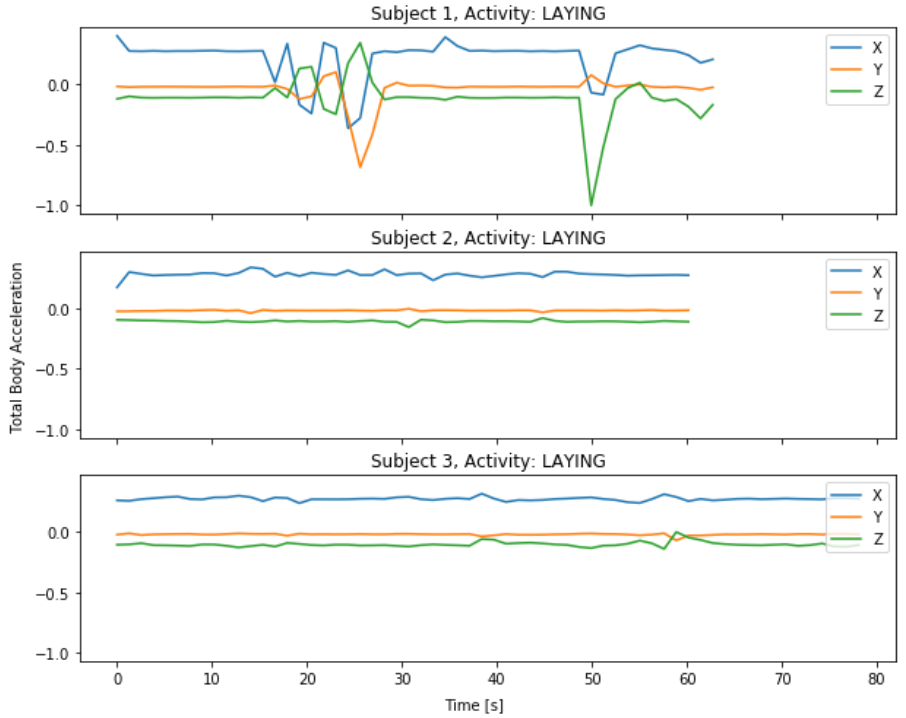


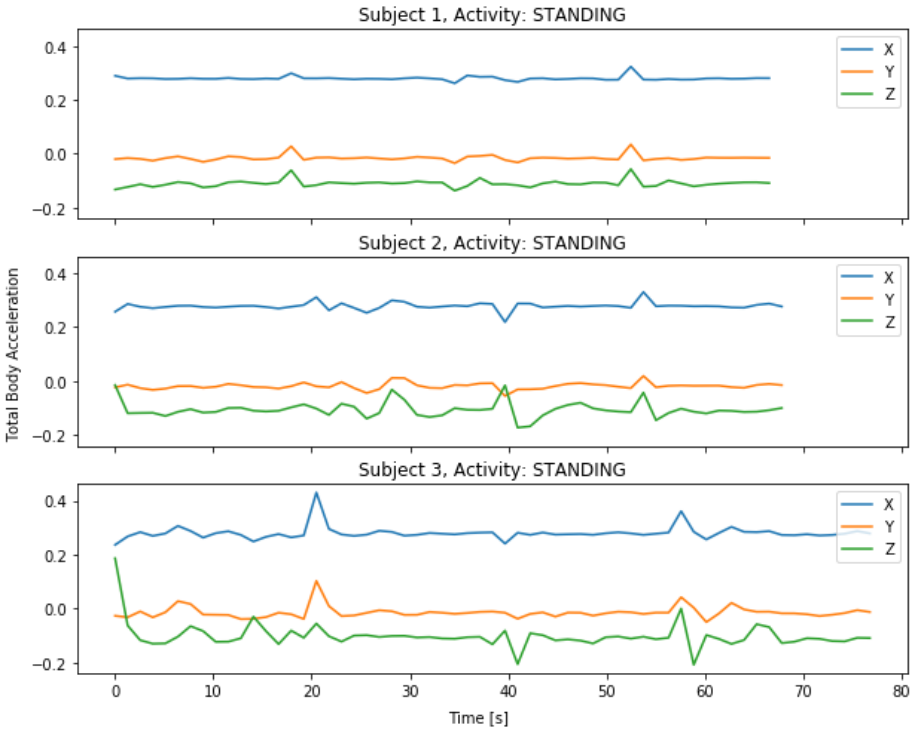
Next, it would be helpful to see a summary of the descriptive statistics for the data. This is done using the .describe() method. From the results of the .describe() method, it appears that the data has already been scaled to a range of [-1, 1] for all feature columns which will be helpful for both data visualization and machine learning. Additional scaling, for example standardization in which the mean is zero and standard deviation is 1, may be additionally performed during the modeling phase of the project.

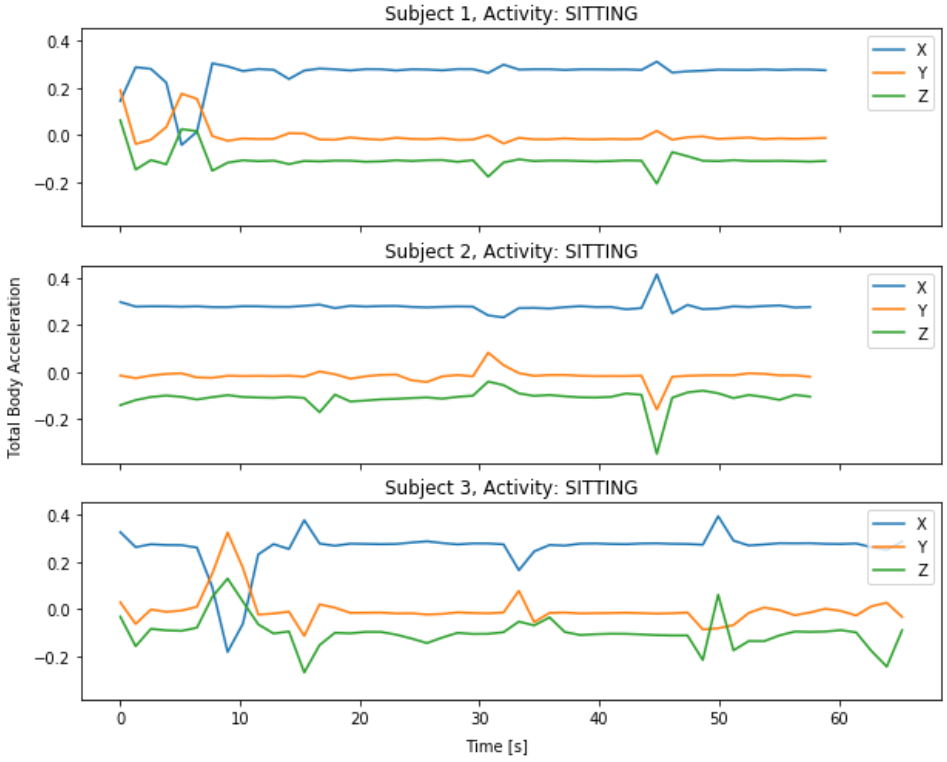
Plotting Example Sensor Data

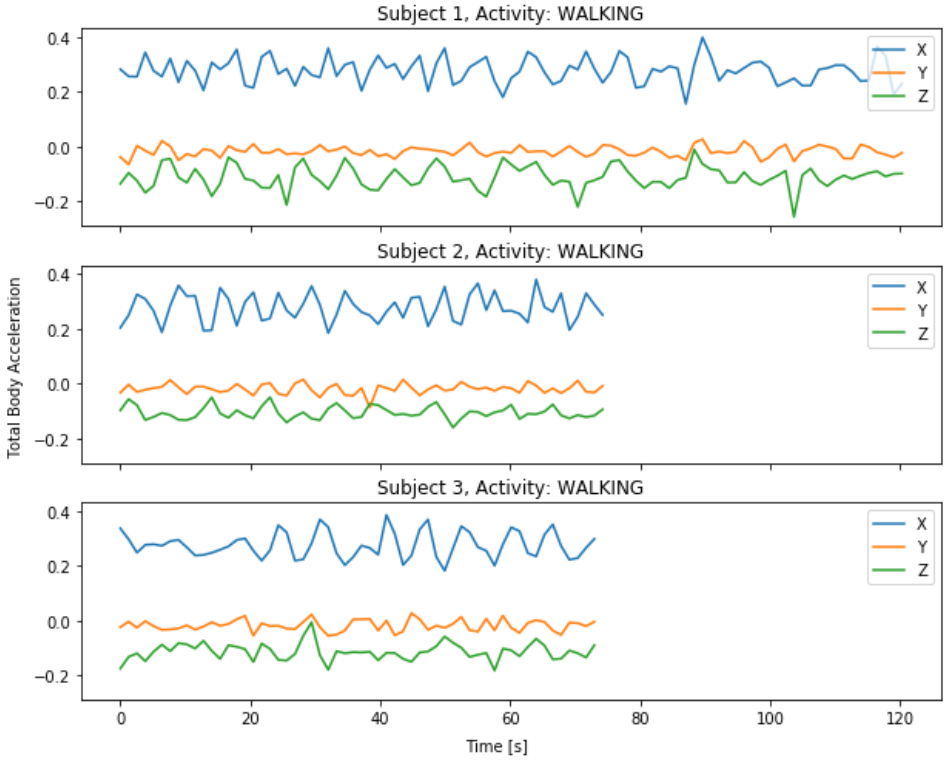
To get a feeling for the type of data, it would be helpful to plot a few examples of the various activities across several subjects. Total mean acceleration in the x,y,z directions is a good starting point.

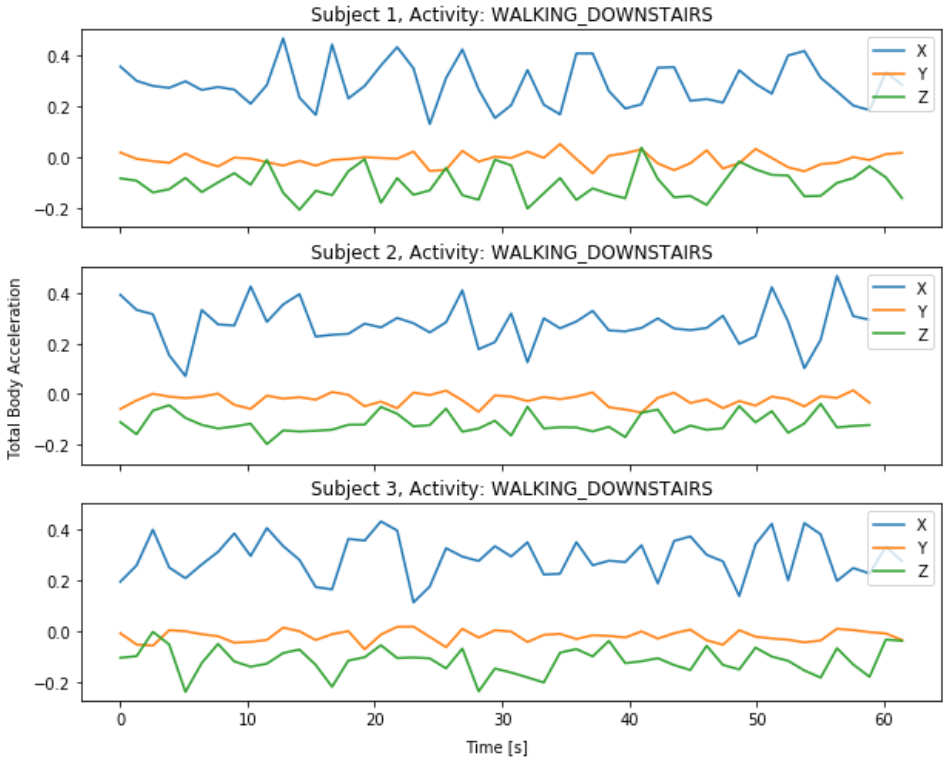
The following plots explore this feature for each of the 6 activities and each of the first 3 subjects.

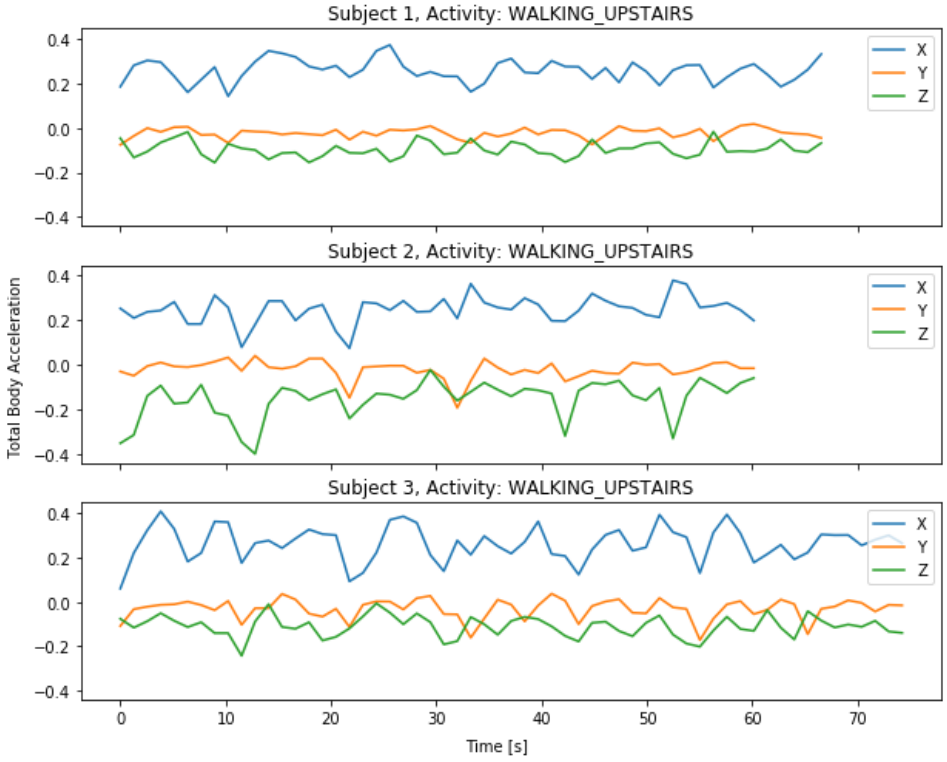





Subject 1, Activity: SITTING
Subject 2, Activity: SITTING
Subject 3, Activity: SITTING
Total Body Acceleration
X
Y
Z
0.4
0.2
0.0
−0.2
0
10
20
30
40
50
60
Time [s]


Subject 1, Activity: WALKING
Subject 2, Activity: WALKING
Subject 3, Activity: WALKING
Total Body Acceleration
X
Y
Z
0.4
0.2
0.0
−0.2
0
20
40
60
80
100
120
Time [s]


Subject 1, Activity: WALKING_DOWNSTAIRS
Subject 2, Activity: WALKING_DOWNSTAIRS
Subject 3, Activity: WALKING_DOWNSTAIRS
Total Body Acceleration
Time [s]
X
Y
Z


Subject 1, Activity: WALKING_UPSTAIRS
Subject 2, Activity: WALKING_UPSTAIRS
Subject 3, Activity: WALKING_UPSTAIRS
Total Body Acceleration
Time [s]
X
Y
Z

Some observations from the plots above:

- Each subject recorded the same activity for a different period of time.
- The laying activity, as expected, has the lowest signals (mostly flat). Except subject 1 who had an erratic (unusual) signal.
- Standing and sitting have signals of similar magnitude, which might be a source of classification error.
- Walking upstairs seems to produce more regular cycles of movement than walking downstairs.

Visualizing Separability of Classes

From the single attribute type plotted above (total body acceleration), some differences between the classes are immediately noticeable. However, it does not give a clear picture of the overall dataset and how the rest of the attributes contribute to the separability of the activity classes.

Two visualization techniques, using different dimensionality reduction methods, will be implemented and compared to get a qualitative indication of the class separability using the full dataset. The techniques are as follows (source):

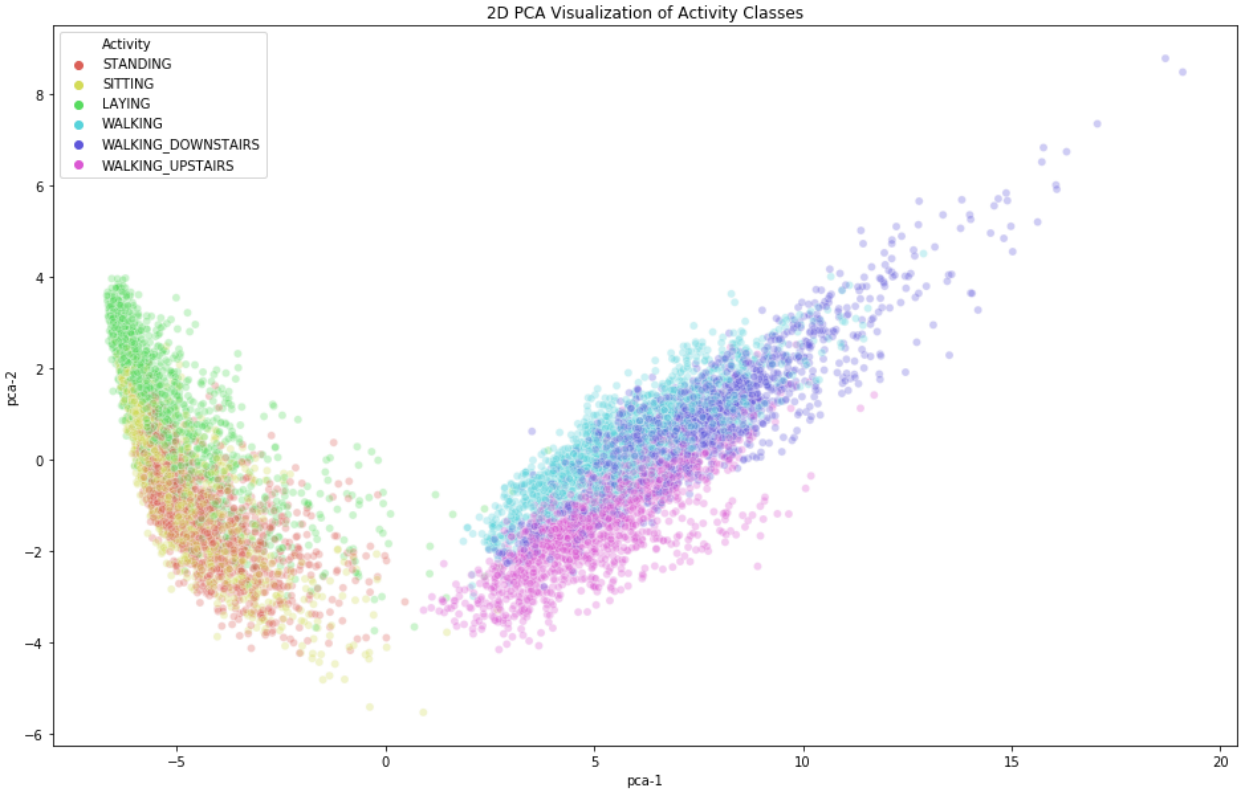
- Principal component analysis (PCA) uses the eigenvectors of the covariance matrix to reduce the dimensionality of the data. The eigenvectors point in the directions of maximum variation in a dataset, preserving as much of the variability in the data in as few dimensions as possible. Two or three of the principle components may be plotted for visualization purposes.
- t-Distributed Stochastic Neighbor Embedding (t-SNE) is a probablistic technique for dimensionality reduction, which minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding.

t-SNE is computationally heavy for very high dimensional data and will likely require an initial dimensionality reduction before execution, for example PCA to the first 50 principle components.

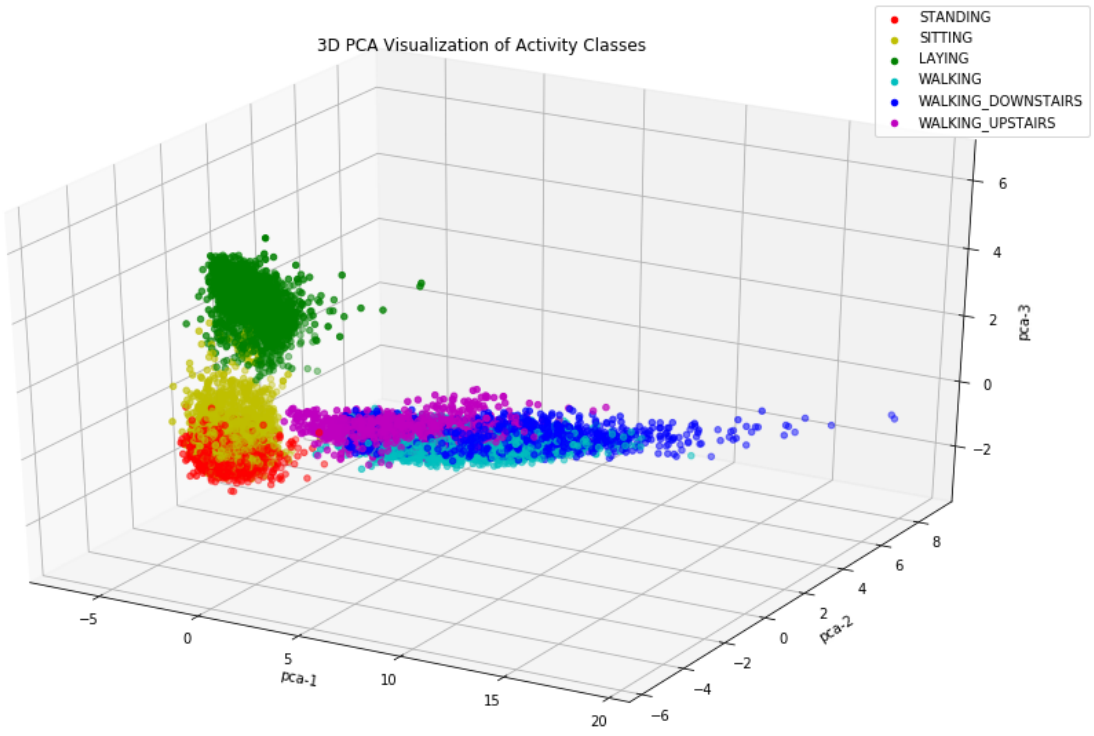
PCA

The PCA analysis was quite fast (~0.20 s), requiring relatively low computational resources to execute. The first 3 principle components represent nearly 71% of the overall variation in the dataset. This is quite high and should hopefully allow for producing some useful visualizations.

The following figures are two rotations of the 2D PCA plots (with 1st/2nd and 1st/3rd principle components) and the 3D plot of all 3 principle components):


2D PCA Visualization of Activity Classes
Activity
STANDING
SITTING
LAYING
WALKING
WALKING_DOWNSTAIRS
WALKING_UPSTAIRS
pca-2
pca-1
8
6
4
2
0
−2
−4
−6
−5
0
5
10
15
20


2D PCA Visualization of Activity Classes (Rotated)
Activity
STANDING
SITTING
LAYING
WALKING
WALKING_DOWNSTAIRS
WALKING_UPSTAIRS
pca-3
pca-1
6
4
2
0
−2
−5
0
5
10
15
20



As can be seen from the above plots, the “active” classes (walking, walking upstairs, walking downstairs) are very separable from the “non-active” classes (laying, standing, sitting).

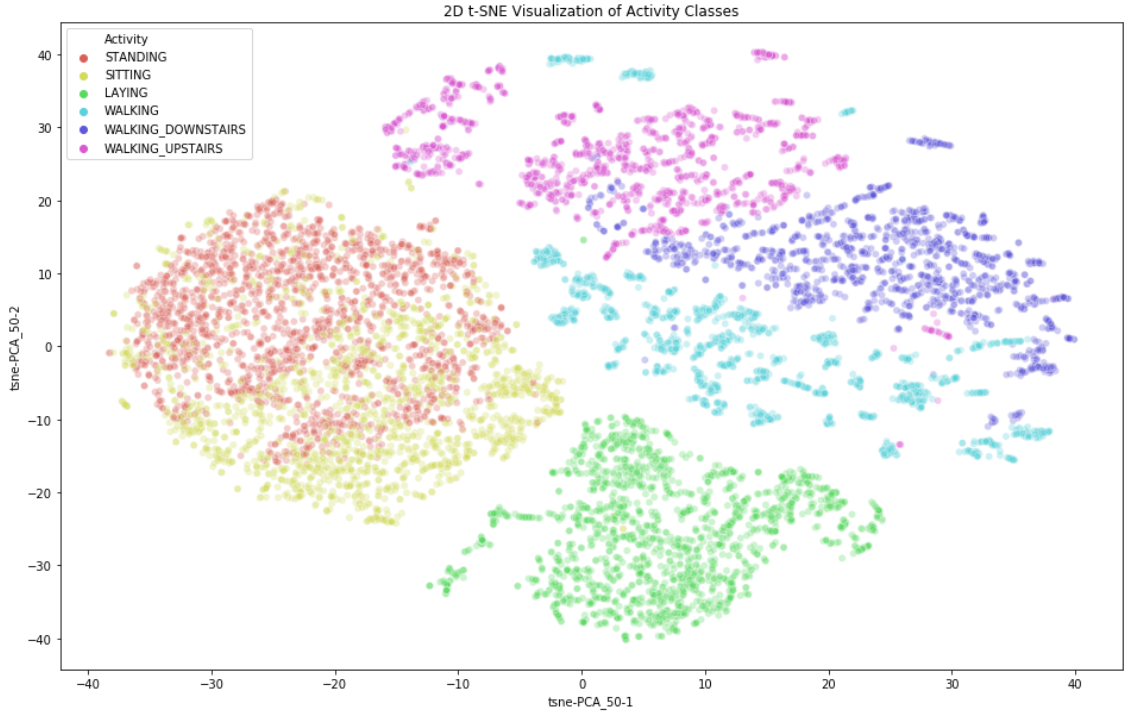
- Within the non-active group, the laying class is much more separated than the sitting and standing classes.
- Within the active group, the classes are less separated, with the walking class bridging the other two.

tSNE

The t-SNE approach to class separability visualization took nearly 600x the runtime of PCA. Several combinations of perplexity and n_iterations were tried with no significant difference in the overall visualization appearance or KL divergence. The algorithm appears to have converged and produced some plausible clusterings, which offer a bit more visual separation than PCA, though standing and sitting classes still appear heavily intermixed. However, the drawback of t-SNE is in its interpretability: since Euclidean distance between points and between clusters is not necessarily indicative of relative correlation strengths. Overall, it seems PCA could be a sufficient technique for visualization in this case.

The following visualization was produced with this method:

Additionally, it is possible to combine the two techniques (PCA with first 50 principle components and t-SNE) and examine the result. The cumulative explained variation for 50 principal components is 92%. The following is the resulting plot:



When combining the two techniques, the runtime of t-SNE is reduced by about 68%. The visualizations are very comparable, with some expected rotation. There does not appear to be any significant loss of relational/probablistic robustness.

This would indicate that there is a computational advantage to running PCA first and then t-SNE. However, with PCA offering better interpretability than t-SNE, as well as being faster and deterministic, it seems to be the best option for this analysis.

Statistical Analysis

Aggregating Data

For statistical analysis purposes and in order to visualize some of the distributions of the different classes and subjects, each activity for a given subject will be summarized (time, mean, median, std, max, min) across the full activity time window for all features. Each of these summary statistics will be calculated in a new dataframe "separated_df_X".

Additionally, it was observed in the Exporatory Data Analysis, that the active and passive classes were more distinguishable as "super-clusters" containing the individual activity class clusters. This means that additional statistical analysis can be done by assigning a new super-class label to each observation. This label will be in a new column "superClass" and will either be 0 for passive or 1 for active.

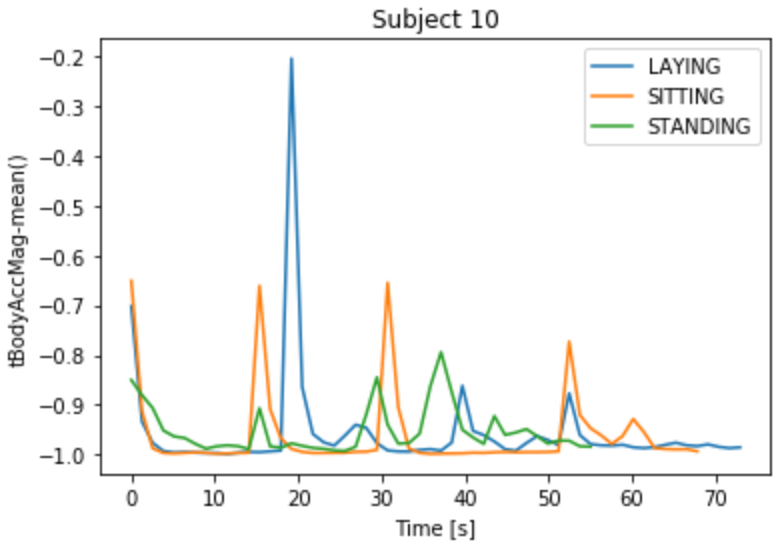
Again, each of the summary statistics will be calculated in a new dataframe "superClass_df_X" -- this time separated by superClass.

Selecting Features of Interest

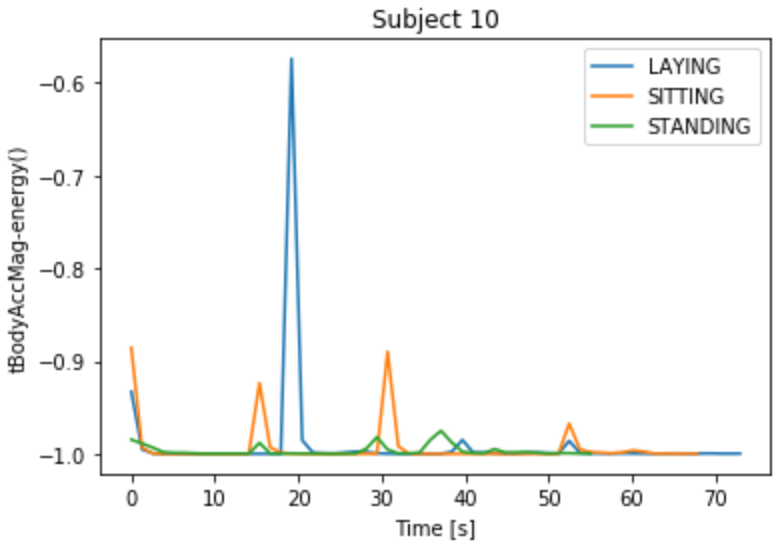
To get an idea of the usefulness of this newly aggregated data, it is helpful first to select and plot single features of interest across Activities and group by superClass. A plotting function is created and called for a random individual subject. This function call may be executed multiple times to get a visual concept of variability amongst subjects.

Next, two features have been chosen as those that are both human interpretable and likely to show larger differences among the classes, based on a rough understanding of the processed input signals:

- tBodyAccMag-mean() is the total body acceleration magnitude (Euclidean norm of the three-dimensional signal), averaged over each 2.56 s time window
- tBodyAccMag-energy() the total body acceleration magnitude energy measure (sum of the squares divided by the number of values)


Subject 10
LAYING
SITTING
STANDING
tBodyAccMag-mean()
Time [s]


Subject 10
WALKING
WALKING_UPSTAIRS
WALKING_DOWNSTAIRS
tBodyAccMag-mean()
Time [s]



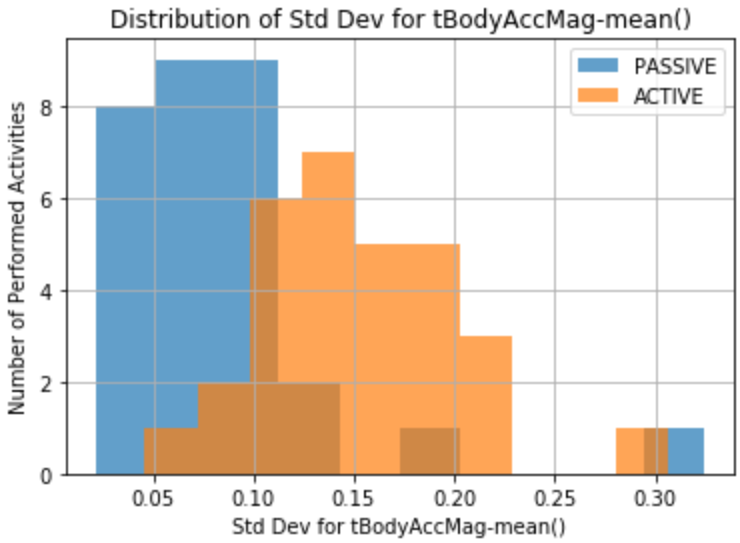
From the plots above, it can be seen that the various activities appear to be somewhat separable in the range of their values.

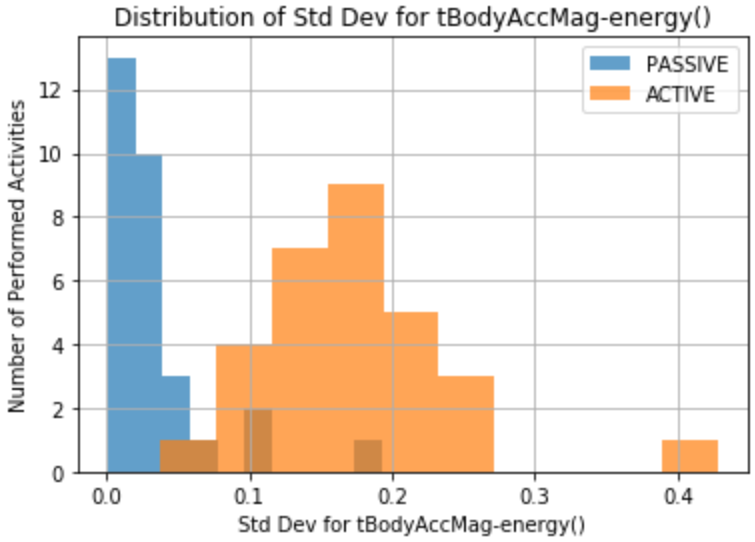
As an additional exercise, summarizing the data contained in the bandsEnergy() features (energy of a frequency interval within the 64 bins of the FFT of each window) could likely give some Activity class-specific differentiators. This would require a different aggregation strategy as the Fourier transformed variables contain 42 bandsEnergy() features, each in its own column. The assumption is that one or more of these frequency bands would consistently stand out as a frequency "signature" for a given Activity class. This has not been attempted here.

Visualizing Distributions of Aggregated Feature Statistics

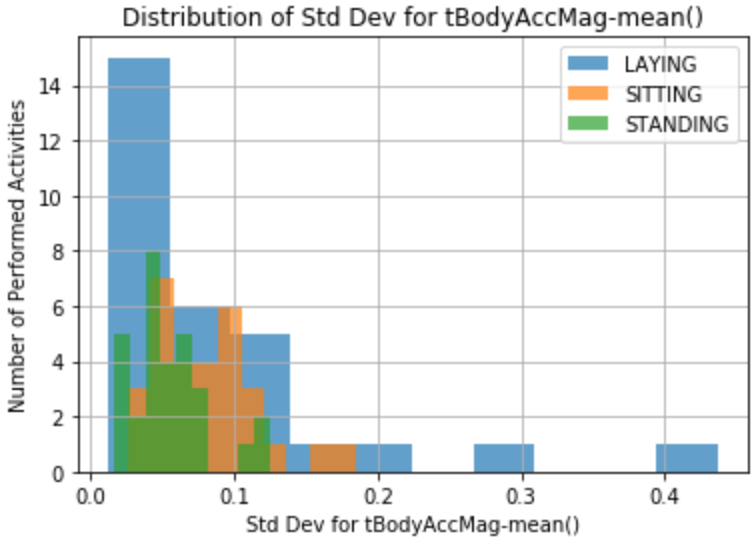
Next, in statistically trying to distinguish the classes, it may be useful to visually compare the distributions of the above plotted features across each of the various activities. For example, it is likely that the std dev of tBodyAccMag signals from the active classes will be generally higher than those of the passive classes.

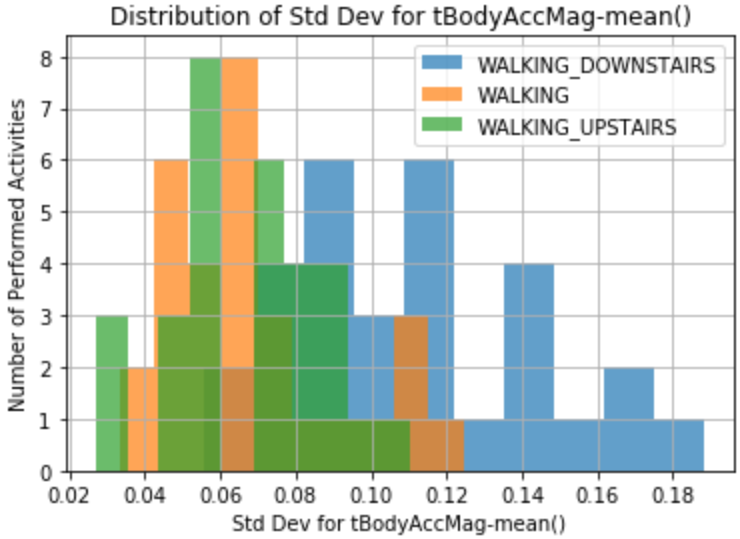
Histograms of the standard deviation for the 2 variables of interest may be plotted to compare by the superClass distinction.

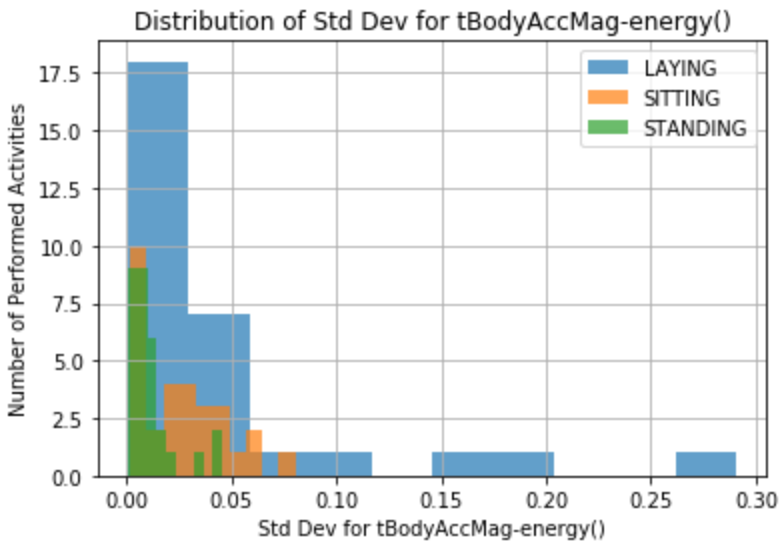


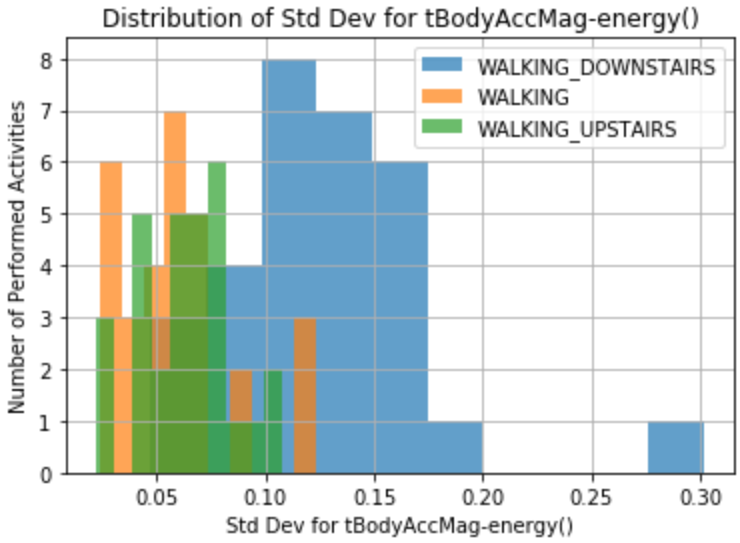


Plotting similar histograms for these feature summary statistics (focusing on std deviation here for conciseness), this time with each Activity separated out individually. The differences of these distributions are more nuanced than those of the aggregated superClasses, as is to be expected.


Distribution of Std Dev for tBodyAccMag-mean()
LAYING
SITTING
STANDING
Number of Performed Activities
Std Dev for tBodyAccMag-mean()


Distribution of Std Dev for tBodyAccMag-mean()
WALKING_DOWNSTAIRS
WALKING
WALKING_UPSTAIRS
Number of Performed Activities
Std Dev for tBodyAccMag-mean()


Distribution of Std Dev for tBodyAccMag-energy()
LAYING
SITTING
STANDING
Number of Performed Activities
Std Dev for tBodyAccMag-energy()
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.00
0.05
0.10
0.15
0.20
0.25
0.30


Distribution of Std Dev for tBodyAccMag-energy()
WALKING_DOWNSTAIRS
WALKING
WALKING_UPSTAIRS
Number of Performed Activities
Std Dev for tBodyAccMag-energy()
0
1
2
3
4
5
6
7
8
0.05
0.10
0.15
0.20
0.25
0.30

Bootstrap Hypothesis Testing

Motivation

One hypothesis worth testing is that Activity or superClass distinctions can be made based on the distribution of one or more of their single feature's summary statistics, across a subject sample of 30 participants.

For example, one hypothesis: passive and active classes have statistically significantly different mean std. dev for the tBodyAccMag-mean() and tBodyAccMag-energy() features. The histogram points to the possibility that this hypothesis is true, but a hypothesis test would be required to prove it with statistical significance. It is also possible that statistically significant differences can be proven between superClasses, but might fail to proven between certain similar activities within a superClass (WALKING and WALKING_UPSTAIRS, for example).

Since the distributions of std. dev. for either superClass do not meet the normality assumption of traditional frequentist inference and are relatively small (N=90, 30 subjects performing each of 3 activities), this a good candidate for bootstrapping. A bootstrap resampling of these distributions would allow for statistical inference testing of the difference of means.

If this hypothesis test fails, it would strengthen the business case for creating a machine learning classification model. If, however statistically siginificant differences CAN be found amongst single features, a more lightweight approach to classification based on statistical inference may make more sense, particularly if the productionized classifier needs to run in real time with limited processing power. This is often the case for wearable devices.

Performing Hypothesis Tests

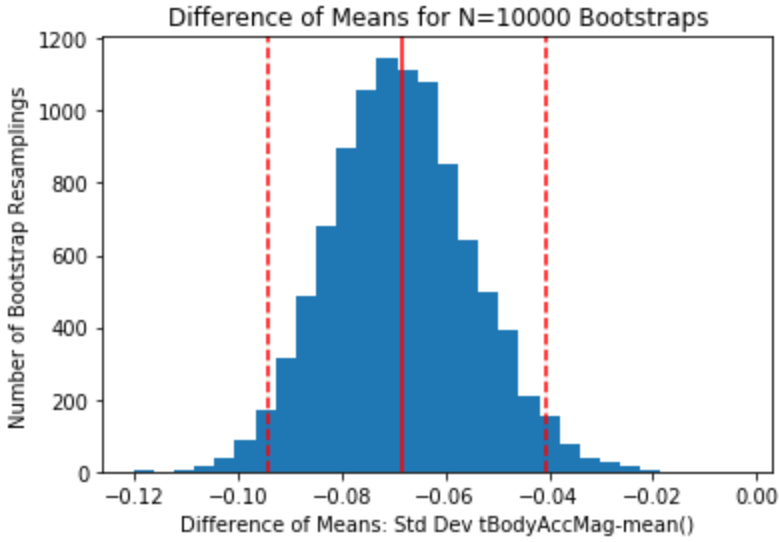
Using the above defined function, a bootstrap hypothesis test may be performed. To start, the following hypothesis will be tested: passive and active classes have statistically significantly different mean std. dev for the tBodyAccMag-mean() and tBodyAccMag-energy() features. This statement actually contains 2 hypothesis tests, one for each feature. The hypothesis tests are defined as follows, with null and alternative hypotheses:

Hypothesis 1
Ho: The difference of mean standard deviation for the tBodyAccMag-mean() between passive and active classes is equal to zero.
Ha: The difference of mean standard deviation for the tBodyAccMag-mean() between passive and active classes is not equal to zero.

Hypothesis 2
Ho: The difference of mean standard deviation for the tBodyAccMag-energy() between passive and active classes is equal to zero.
Ha: The difference of mean standard deviation for the tBodyAccMag-energy() between passive and active classes is not equal to zero.
Significance level for both tests is 95%. The number of bootstrap samples is N=10000.

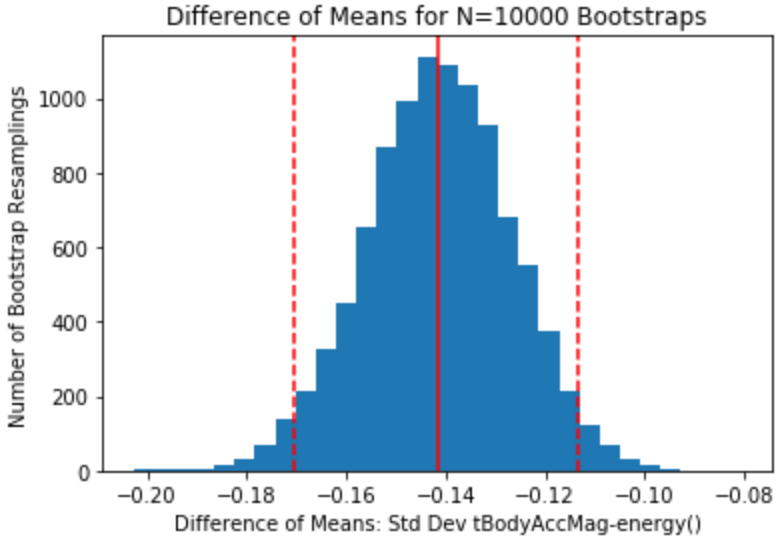
Hypothesis 1

The bootstrap 95% CI is [-0.09417189, -0.04060474].



Hypothesis 2

The bootstrap 95% CI is [-0.17066601, -0.11332069]



For both of the above hypothesis tests, the N=10000 bootstrap 95% confidence interval does not contain zero. This means that the null hypothesis can be rejected. Therefore, the hypothesis that passive and active classes have statistically significantly different mean std. dev for the tBodyAccMag-mean() and tBodyAccMag-energy() features can be accepted with a statistical significance level of 95%.

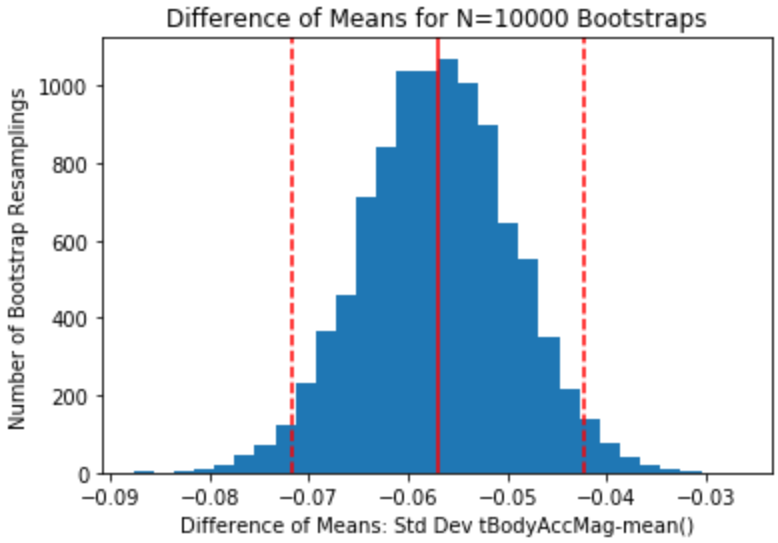
In other words, the distributions of these feature summary statistics are different enough to correctly classify a random Activity as passive or active based only on the std. deviation of one of these features with 95% confidence.

This result would most likely be the case also for examining two activities that are the most "dissimilar". For example, STANDING and WALKING_DOWNSTAIRS. However, similar activities within the same superClass (WALKING and WALKING_UPSTAIRS, for example) may not have statistically different enough distributions.

Hypothesis 3
Ho: The difference of mean standard deviation for the tBodyAccMag-mean() between STANDING and WALKING_DOWNSTAIRS is equal to zero.
Ha: The difference of mean standard deviation for the tBodyAccMag-mean() between STANDING and WALKING_DOWNSTAIRS is not equal to zero.

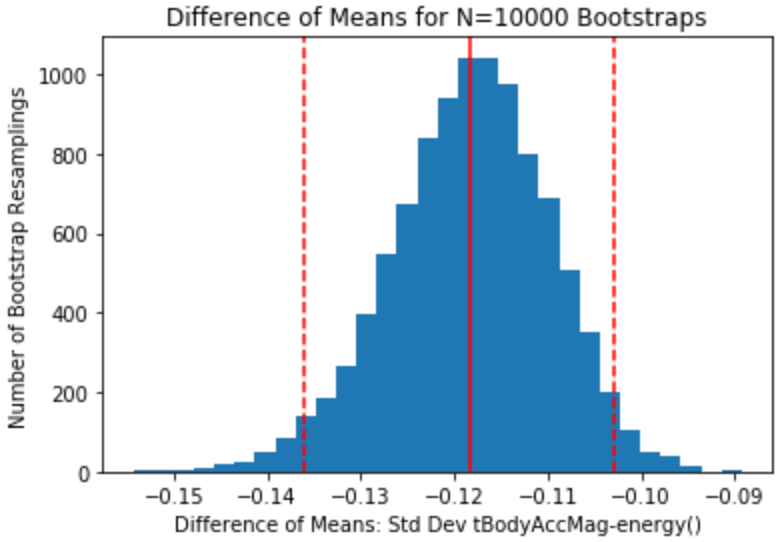
Hypothesis 4
Ho: The difference of mean standard deviation for the tBodyAccMag-energy() between STANDING and WALKING_DOWNSTAIRS is equal to zero.
Ha: The difference of mean standard deviation for the tBodyAccMag-energy() between STANDING and WALKING_DOWNSTAIRS is not equal to zero.
Significance level for both tests is 95%. The number of bootstrap samples is N=10000.

Hypothesis 3
The bootstrap 95% CI is [-0.071769, -0.04226481]



Hypothesis 4

The bootstrap 95% CI is [-0.13603891, -0.10288204]



Hypothesis 5

Ho: The difference of mean standard deviation for the tBodyAccMag-mean() between STANDING and SITTING is equal to zero.

Ha: The difference of mean standard deviation for the tBodyAccMag-mean() between STANDING and SITTING is not equal to zero.

Hypothesis 6

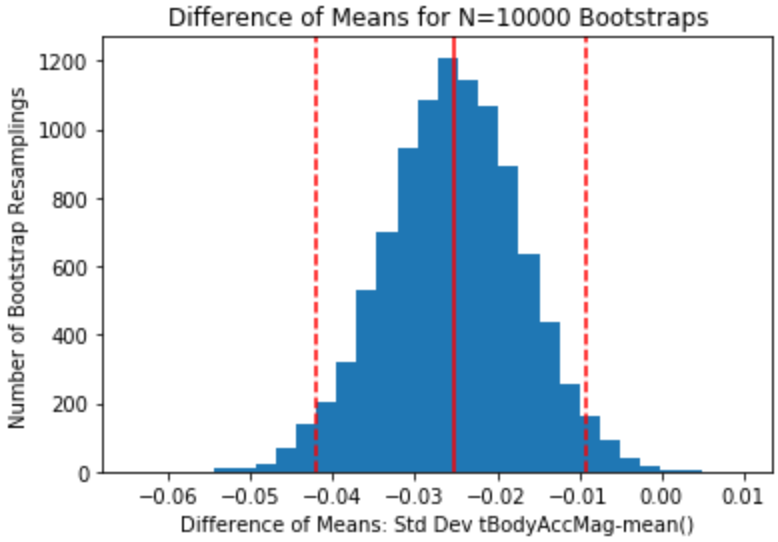
Ho: The difference of mean standard deviation for the tBodyAccMag-energy() between STANDING and SITTING is equal to zero.

Ha: The difference of mean standard deviation for the tBodyAccMag-energy() between STANDING and SITTING is not equal to zero.

Significance level for both tests is 95%. The number of bootstrap samples is N=10000.

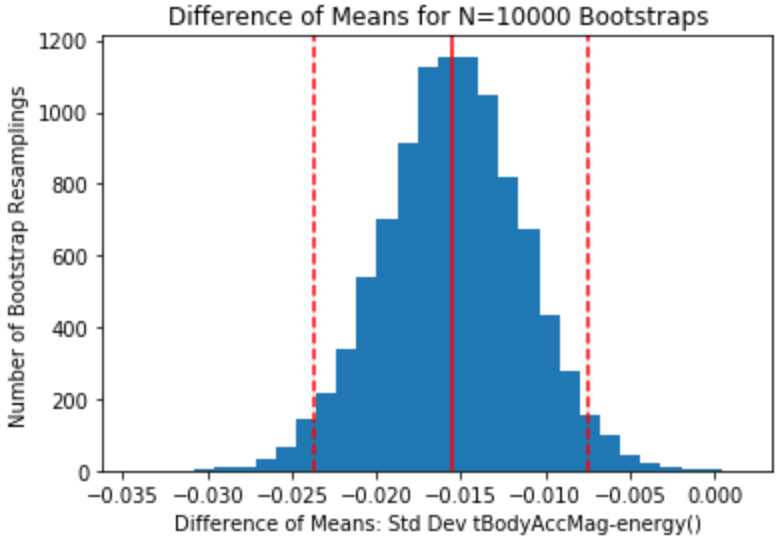
Hypothesis 5

The bootstrap 95% CI is [-0.04202838, -0.00906252]



Hypothesis 6

The bootstrap 95% CI is [-0.02366883 -0.00743425]



Both null hypotheses can be rejected at a 95% statistical significance level.

This is a surprising result: even for highly overlapping std. dev. distributions like for SITTING and STANDING, there is enough difference to say with statistical significance that a random std. dev value for the tested variables belongs to one distribution over another. This confirms the high class separability that was seen in the PCA and t-SNE visualizations. It makes this dataset a strong candidate for both statistical inference approaches as well as machine learning classifiers such as SVM. These will be explored in the next section.

Machine Learning Preprocessing

Splitting the Data: Training and Test Sets

In order to allow for evaluation of the machine learning (ML) model, it is critical to split the full data set into a training set (used to train the ML model) and a test set (used to evaluate the model's performance).

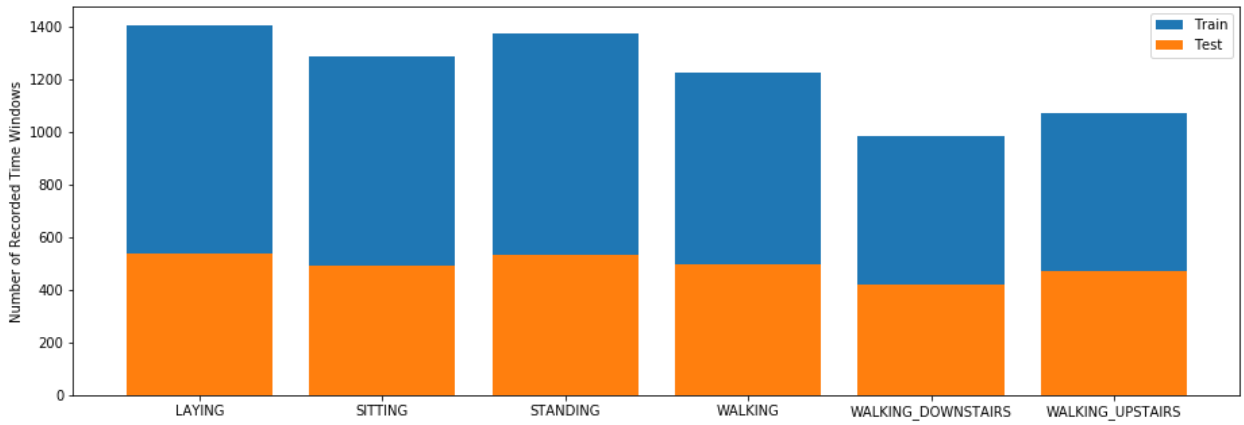
In this case, a stratified split is important (i.e. making sure that the classes are represented in both the training and test sets relatively equally). Also, it must be taken into account that entire activities and subject sets should be kept together. Movement patterns of the same subject should be segregated so as not to bias the test set.

Therefore of the 30 participating subjects' data, 30% (9 subjects) will be segregated into the test set and saved for the evaluation phase. These subjects have been selected randomly (Kaggle has already provided the data in train/test split for this dataset.)

The resulting feature (X) and label (y) arrays are sized as follows:

Array	Num. Rows	Num. Columns
X_train	7352	561
y_train	7352	1
X_test	2947	561
y_test	2947	1

And the class distribution among the resulting test/train split is relatively even:



Feature Scaling

To improve performance of the machine learning model(s) to be applied to this classification problem, first the feature data should be appropriately scaled.

Scaling must be performed both on the training and test data sets in the same manner. Best practice is to use only the training set to identify the correct scaling, and then blindly apply the same transformation to the test set. In this case, the feature matrix will be standardized (i.e. the mean of each feature set to zero and the std dev set to 1) using the sklearn method StandardScaler(). This results in feature sets X_train_scaled, X_test_scaled .

- For each column in the X_train_scaled feature matrix, the mean is set to zero and the std dev is set to 1
- For each column in the X_test_scaled feature matrix, the mean is ***NOT*** set to zero and the std dev is ***NOT*** set to 1, since a transformation matrix (generated from normalizing the training feature matrix range) was applied to the test feature matrix.

Categorical Class Encoding

Since the class labels (Activity column) for this data are in a categorical string format, they must first be encoded to a numerical format useable for supervised machine learning. Simple integer encoding (i.e. translating each class label into an integer label 1-6) is normally would not effective on its own if the categorical variable was in the feature set, since such an encoding implies an ordinal relationship between the categories, where in reality there is none. This may result in poor model performance or unexpected results.

However, in this case where the categorical variable is the dependent variable (classifier label y), the second step of one-hot encoding is not necessary. In fact the SVC() method in sklearn will not accept a one-hot encoded multi-class array as the label vector.

Fitting an SVM Model

In previous published papers (most notably: https://upcommons.upc.edu/bitstream/handle/2117/101769/IWAAL2012.pdf), a support vector machine (SVM) was used successfully to harness the engineered features to implement a lightweight multi-class machine learning classifier.

As a baseline for comparison of deep learning techniques that will be implemented later in this project, the SVM model is fit to the training data and evaluated. Deep learning models can effectively work on the raw time series data, but require heavier computational resources, while traditional machine learning models perform better when feature engineering is performed beforehand with sufficient domain knowledge applied. This will be done using the support vector classifier SVC() method in sklearn.

Hyperparameter Tuning

In order to optimize performance of the SVC() model, a grid search across several model hyperparameters is performed. The hyperparameters chosen for this search have the following value ranges, based on commonly selected values in the literature:

- Regularization parameter **C**: [1, 10, 1000, 10000]
- Kernel coefficient **gamma**: [1e-6, 1e-4, 1e-2]
- A linear kernel vs. (non-linear) RBF kernel, since it is not known a-priori if the classes are linearly separable

The GridSearchCV() method is used to scan across hyperparameter combinations using a 5-fold cross validation to evaluate a mean model performance for each combination. The best performing set of hyperparameters is chosen for the final model. Since the classes are more or less evenly distributed in the train/test sets, the accuracy metric is suitable for performance evaluation. The results of the grid search are as follows:

Best Score (accuracy): 0.9383
Best Params: {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}

Model Evaluation

The SVC() model is fit using the optimized parameter set and the .predict() method is used on the fitted model. The confusion matrix and classification report will be used to compare model to future models. First, evaluating the method on the training features results in the following:

Confusion matrix for SVM evaluation on ***training*** set:

	LAY	**SIT**	**STA**	**WALK**	**WALK_D**	**WALK_U**
LAY	1407	0	0	0	0	0
SIT	0	1276	10	0	0	0
STA	0	13	1361	0	0	0
WALK	0	0	0	1226	0	0
WALK_D	0	0	0	0	986	0
WALK_U	0	0	0	0	0	1073

Classification report for SVM evaluation on ***training*** set:

	precision	**recall**	**f1-score**	**support**
LAY	1	1	1	1407
SIT	0.99	0.99	0.99	1286
STA	0.99	0.99	0.99	1374
WALK	1	1	1	1226
WALK_D	1	1	1	986
WALK_U	1	1	1	1073
accuracy			1	7352
macro avg	1	1	1	7352
weighted avg	1	1	1	7352

The above evaluation shows a nearly perfect classification, as expected on the data set used to train the model. There is some misclassification between the SITTING and STANDING classes.

Next, the model is evaluated on the test set, which it has never encountered.

Confusion matrix for SVM evaluation on test set:

	LAY	**SIT**	**STA**	**WALK**	**WALK_D**	**WALK_U**
LAY	537	0	0	0	0	0
SIT	0	436	54	0	0	1
STA	0	15	517	0	0	0
WALK	0	0	0	493	3	0
WALK_D	0	0	0	5	398	17
WALK_U	0	0	0	16	2	453

Classification report for SVM evaluation on test set:

	precision	recall	f1-score	support
LAY	1	1	1	537
SIT	0.97	0.89	0.93	491
STA	0.91	0.97	0.94	532
WALK	0.96	0.99	0.98	496
WALK_D	0.99	0.95	0.97	420
WALK_U	0.96	0.96	0.96	471
accuracy			0.96	2947
macro avg	0.96	0.96	0.96	2947
weighted avg	0.96	0.96	0.96	2947

The evaluation on the test set shows some additional misclassifications, but overall appears to be a strong classifier with many classes being perfectly distinguished from one another (no misclassifications). Overall model accuracy is 0.96.

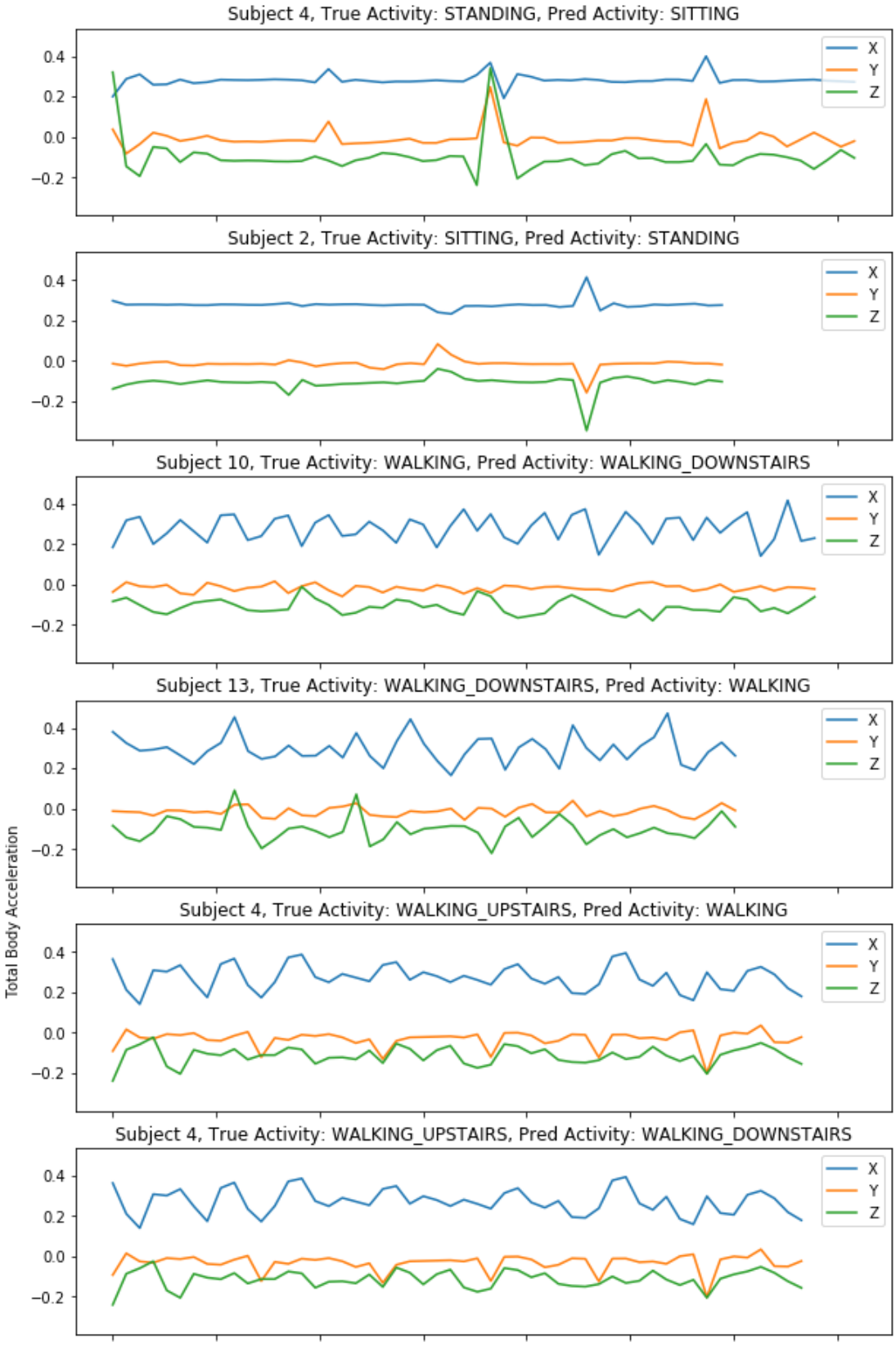
The SVM model is deterministic so the model performance should be the same across executions.

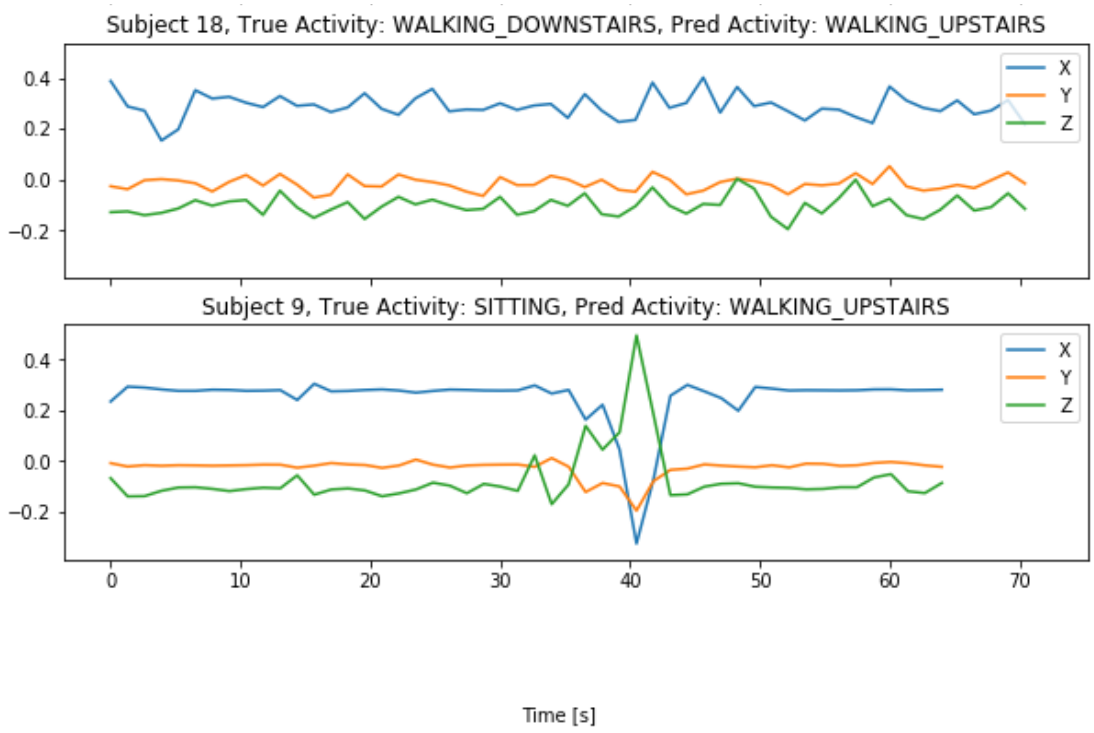
Visualizing Misclassified Activities

To better understand the limitations of the evaluated SVM model, it is helpful to visualize some of the signal data from activity instances that were misclassified by the model. These may be compared to activities that were correctly predicted. Since each window is predicted separately, however, it may not be straightforward to see these patterns of misclassification.

Here are examples from the training set misclassifications:

And a few examples from the test set misclassifications:





Fitting a Neural Network Model

Importing the Raw Data

In previous sections, the dataset used had undergone feature engineering (with 9 signal time series transformed into 561 features for each 2.56 s window of the time series). Implementing a deep learning approach requires switching to the raw data in order to allow the network to find its own features in the data.

Therefore, the raw data needs to be loaded into an appropriately sized feature array X. The label vector y from previous sections remains the same.

Each row/observation still represents a single 2.56 s window, and these windows still have 50% overlap per the design of the collected data. However, now each row contains 128 time steps of raw signal data (50 Hz x 2.56 s). A row with 128 columns exists for each of the 9 signal channels:

- body acceleration (x,y,z)
- body gyroscope (x,y,z)
- total acceletation (x,y,z)

Therefore, when combining all the signal data, each row should actually have 128 x 9 = 1152 columns or features. The 9 channels of data are stored in 9 separate files and are already separated into training and test sets (see "Splitting the Data: Training and Test Sets" in the previous section).

Preprocessing

The raw data again requires preprocessing before a neural network can be trained on it. This consists of the following steps:

1. **Categorical Class Encoding**: Since the class labels (Activity column) for this data are in a categorical string format, they must first be encoded to a numerical format useable for supervised machine learning. Integer encoding followed by one-hot encoding is the preferred method to transform the categorical data. Each integer class label is assigned to a new binary (0/1) column of "dummy variables". Each observation in the dataset is then be labeled with a 1 in only one of these "dummy variable" columns and the rest are labeled with a 0, resulting in sparse matrices of class labels for y_train and y_test.
2. **Feature Scaling:** To improve performance of the machine learning model(s) to be applied to this classification problem, first the feature data should be appropriately scaled. In the SVM model, [0,1] was successfully applied to the feature engineered dataset. However, after evaluating the neural network model with [0,1] scaling applied to the raw data, poor performance was observed (approx. 0.78 accuracy). MinMaxScaler() with range [1,1] scaling performed much better (approx. 0.89 accuracy) and StandardScaler() with mean centered around 0 and std. dev = 1 performed even slightly better (approx. 0.91 accuracy). Therefore, the StandardScaler() method was selected for feature scaling.
3. **Reshaping the Input:** Since the feature data contains sequential information from 9 different channels with 128 timesteps per sample, it must first be reshaped in order to be supplied to an LSTM input layer. This type of layer requires input of the shape ($N_{samples}$, $N_{timesteps}$, $N_{features}$). In the case of this dataset this will result in a training feature array of size (7352, 128, 9) and test feature array of size (2947, 128, 9).

Fitting an RNN Model

Now that the data has been loaded and preprocessing completed, a deep learning model is trained on the data. The first model that will be implemented is a Recurrent Neural Network (RNN) which leverages the sequential nature of the time series signal data from the 9 different channels. The model is implemented using the tf.keras, TensorFlow's high-level API for building and training deep learning models.

Specifically, the LSTM (long short term memory) layer of the neural network, unlike feed forward neural network architectures, has feedback mechanisms that allow previous observations in the sequence to be "remembered" by the network. The LSTM layer in this architecture is “stateless” (vs. “stateful”, in which the previous epoch’s predictions are used as an initial guess for the next epoch) due to its inherent stability and translatability into a production environment.

First, a sequential neural network model architecture is established. Then several network hyperparameters will be optimized using a grid search optimization. For the baseline RNN model, the network consists of the following layers and network parameters (weights):

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 200)               168000
_________________________________________________________________
dropout (Dropout)            (None, 200)               0
_________________________________________________________________
dense (Dense)                (None, 200)               40200
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 1206
=================================================================
Total params: 209,406
Trainable params: 209,406
Non-trainable params: 0
_________________________________________________________________
```

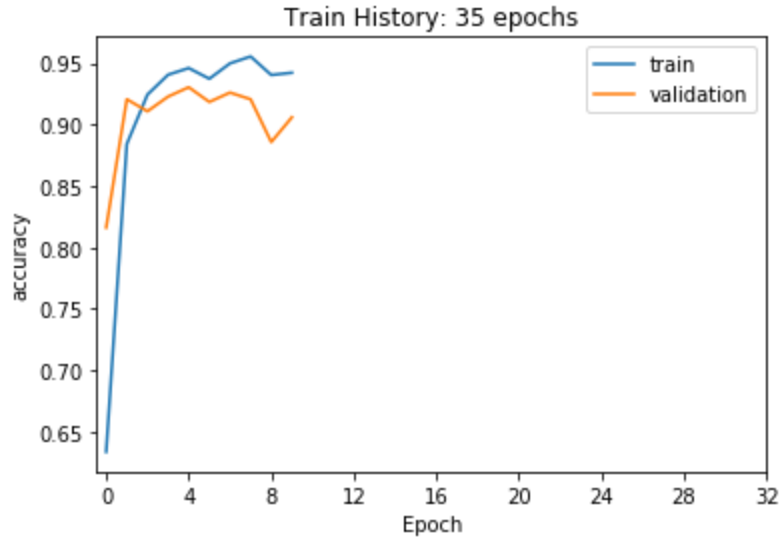
The following hyperparameter ranges are searched, using a grid search with 3-fold cross validation, to find the combination resulting in optimal model performance. These were selected as important model parameters (from the larger set of all model parameters) given the neural network type and architecture.

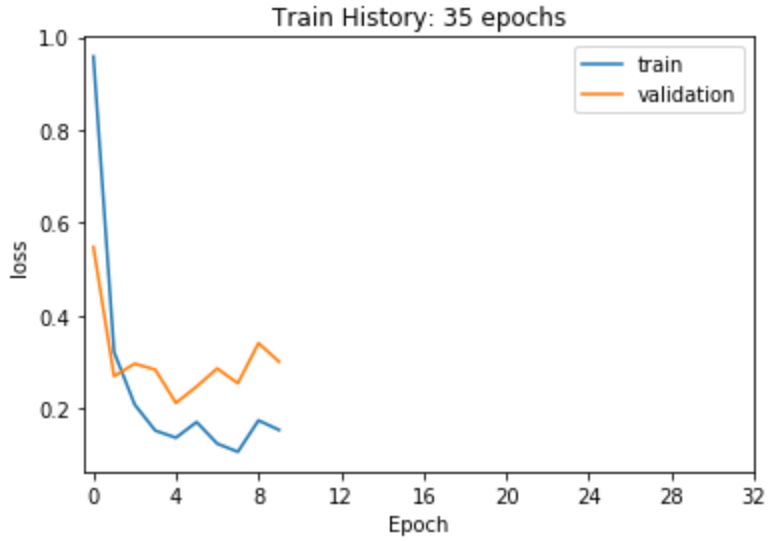
- **batch_size** = [32, 64, 128]
  Batch size controls the accuracy of the estimate of the error gradient (in gradient descent optimization) when training neural networks. This can affect the convergence of the learning algorithm.
- **dropout** = [0.1,0.3,0.5]
  Dropout is an effective regularization method to reduce overfitting and improve generalization error in neural networks of all kinds. This parameter affects the number of nodes randomly dropped during training.
- **neurons** = [50, 100, 200]
  The number of neurons in each hidden layer (LSTM, Dense) has an impact on under/overfitting and on training time.

The hyperparameters with best performance on a 25% validation set are:
{'batch_size': 128, 'dropout': 0.1, 'neurons': 200}

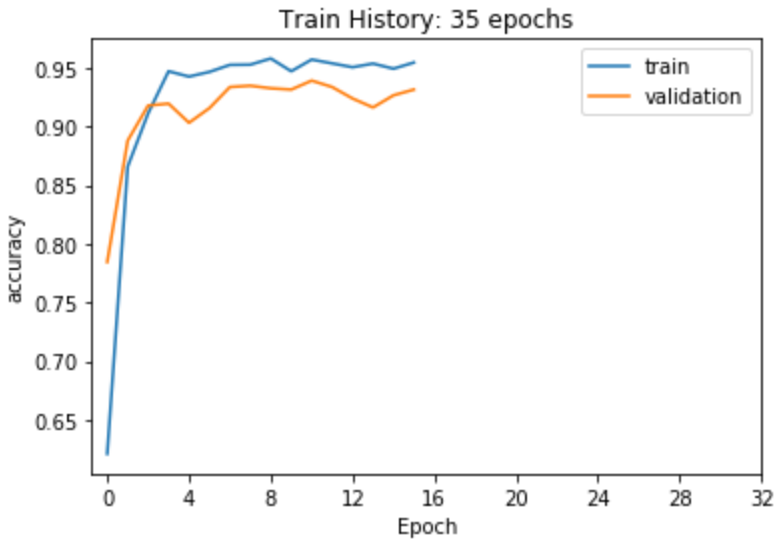
The RNN model with optimized hyperparameter set is fit to the training data and allowed to train for 35 epochs, with an early stopping algorithm to stop training prior to 35 epochs if no improvement in the loss (crossentropy) is observed over any 5 consecutive epochs. The training history is monitored (with a 20% validation set) and plotted (see below).

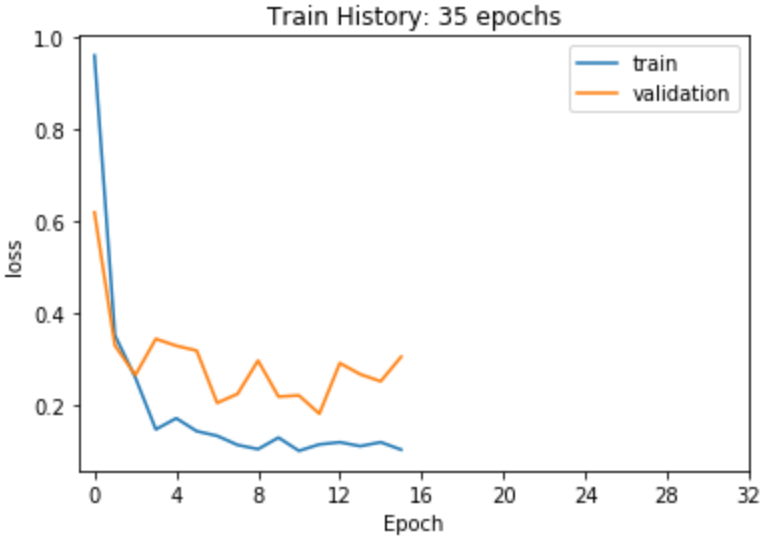
RNN Training history





When left to train for additional epochs, the performance remains relatively stable without obvious signs of overfitting. See below.


Train History: 35 epochs
train
validation
accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
4
8
12
16
20
24
28
32
Epoch


Train History: 35 epochs
train
validation
loss
1.0
0.8
0.6
0.4
0.2
0
4
8
12
16
20
24
28
32
Epoch

The RNN model is then used to predict the labels for the test set of 2947 samples (see results below).

Confusion matrix for RNN evaluation on test set:

	LAY	SIT	STA	WALK	WALK_D	WALK_U
LAY	460	13	23	0	0	0
SIT	0	450	20	0	0	1
STA	0	3	417	0	0	0
WALK	1	10	0	353	127	0
WALK_D	0	0	0	46	486	0
WALK_U	0	27	0	0	0	510

Classification report for RNN evaluation on test set:

	precision	recall	f1-score	support
LAY	1.00	0.93	0.96	496
SIT	0.89	0.96	0.92	471
STA	0.91	0.99	0.95	420
WALK	0.88	0.72	0.79	491
WALK_D	0.79	0.91	0.85	532
WALK_U	1.00	0.95	0.97	537
accuracy			0.91	2947
macro avg	0.91	0.91	0.91	2947
weighted avg	0.91	0.91	0.91	2947

Fitting Additional RNN Models with Convolution

Now that an RNN model with a single LSTM layer has been evaluated, additional variations on this model may be tested for potential evaluation improvements. In particular, combining the LSTM with a convolutional neural network architecture is promising, as the sequences of 128 time steps can be subsampled and explored through convolution, a powerful pattern finding tool.

CNN LSTM

The CNN LSTM model is often implemented in machine learning tasks with input that has both temporal and spatial characteristics, such as sequences of images (video), or sequences of words/sentences in text. The CNN-LSTM architecture uses CNN layer(s) for feature extraction on the raw input data and uses subsequent LSTM layer(s) to support sequence prediction.

In order to create input for the initial CNN convolutional layer, the training data array is reshaped such that each time window of 128 samples is broken into 4 blocks or "steps" of 32 samples each, across each of the 9 features. The entire CNN model is wrapped in a TimeDistributed layer to allow the model to read in each of the four steps in the window. The features extracted by the CNN are then flattened and used as input to the LSTM model, which then extracts its own set of temporal features and then returns a final class prediction through the output layer.

Again, the model architecture is defined and a grid search used to evaluate hyperparameter combinations to optimize the model. Then the model with the best parameter combination is fit to the training dataset and evaluated on the test dataset. See the model architecture and evaluation below.

Layer (type)	Output Shape	Param #
time_distributed (TimeDistri	(None, None, 30, 64)	1792
time_distributed_1 (TimeDist	(None, None, 28, 64)	12352
time_distributed_2 (TimeDist	(None, None, 28, 64)	0
time_distributed_3 (TimeDist	(None, None, 14, 64)	0
time_distributed_4 (TimeDist	(None, None, 896)	0
lstm_1 (LSTM)	(None, 200)	877600
dropout_2 (Dropout)	(None, 200)	0
dense_2 (Dense)	(None, 200)	40200
dense_3 (Dense)	(None, 6)	1206

==========
Total params: 933,150
Trainable params: 933,150
Non-trainable params: 0
_____

This outcome for the training history is challenging to understand at first, since normally an increase in validation loss would be expected to be accompanied by a decrease in validation accuracy. However, that is not the case here as the accuracy stays relatively constant but the loss increases significantly.

A possible explanation:

Cross entropy is not a bounded loss, meaning that a few "very" wrong predictions can potentially make the loss grow very quickly. It is possible that there are a few outliers that are classified

extremely badly and that are making the loss explode, but the model is still learning on the rest of the dataset and continuing to correctly predict the other samples consistently.

The growing loss could be a combination of 2 competing phenomena:

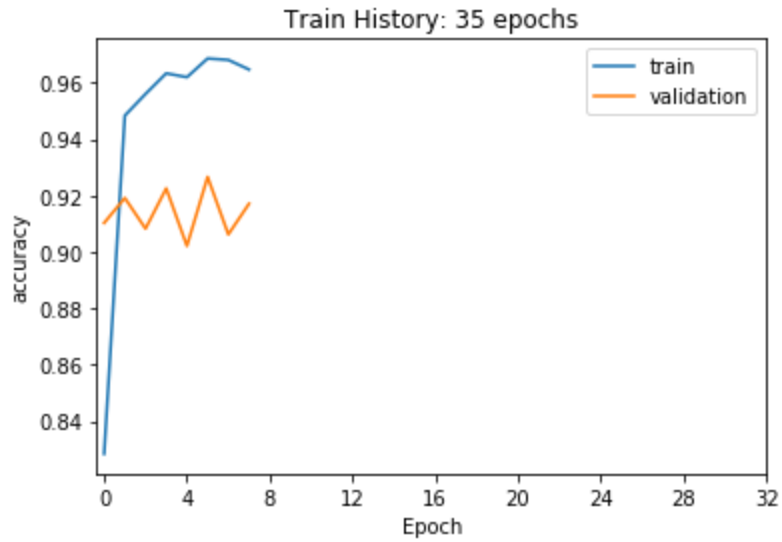
- Some examples with borderline predictions get predicted better and so their output class changes (e.g. a SITTING sample predicted at 0.4 to be SITTING and 0.6 to STANDING in the next epoch gets predicted at 0.4 to be STANDING and 0.6 to be SITTING). In this case, accuracy increases while loss decreases.
- Some examples with poor predictions keep getting worse (e.g. a SITTING sample predicted at 0.8 to be to be STANDING becomes predicted at 0.9 to be STANDING) and/or, potentially more probable for multi-class problems, some examples with very good predictions get a little worse (e.g. a SITTING sample predicted at 0.9 to be SITTING becomes predicted at 0.8 to be SITTING). In this case, loss increases while accuracy stays the same.

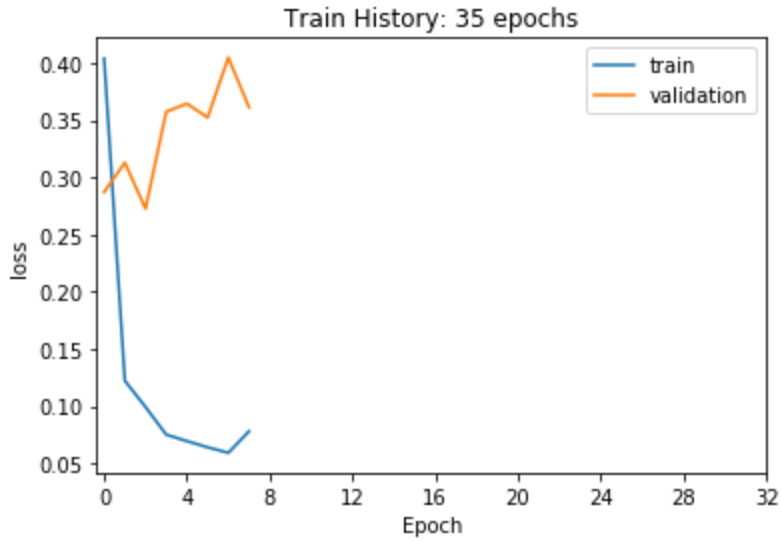
If the second case occurs on lots of examples (e.g. for a specific class which is not well captured by the model, such as SITTING or WALKING classes) and cannot be compensated enough by the first case, then the loss will continually increase.

Several attempts at creating a converging or decreasing loss were attempted, with no success for this particular model architecture:

- increasing the number of epochs
- decreasing the batch size
- decreasing and increasing the "step" (subsequence) length

Therefore to avoid loss of accuracy in both the CNN-LSTM and ConvLSTM models, training was stopped when no improvement to the validation loss could be observed (usually before epoch 8). Example training history (with early stopping) and prediction performance using the CNN-LSTM model are shown below.





Confusion matrix for CNN-LSTM evaluation on test set:

	LAY	SIT	STA	WALK	WALK_D	WALK_U
LAY	432	32	31	1	0	0
SIT	0	455	16	0	0	0
STA	0	32	388	0	0	0
WALK	0	3	0	428	60	0
WALK_D	0	3	0	115	414	0
WALK_U	0	41	0	0	0	496

Classification report for CNN-LSTM evaluation on test set:

	precision	recall	f1-score	support
LAY	1.00	0.87	0.93	496
SIT	0.80	0.97	0.88	471
STA	0.89	0.92	0.91	420
WALK	0.79	0.87	0.83	491
WALK_D	0.87	0.78	0.82	532
WALK_U	1.00	0.92	0.96	537

accuracy			0.89	2947
macro avg	0.89	0.89	0.89	2947
weighted avg	0.89	0.89	0.89	2947

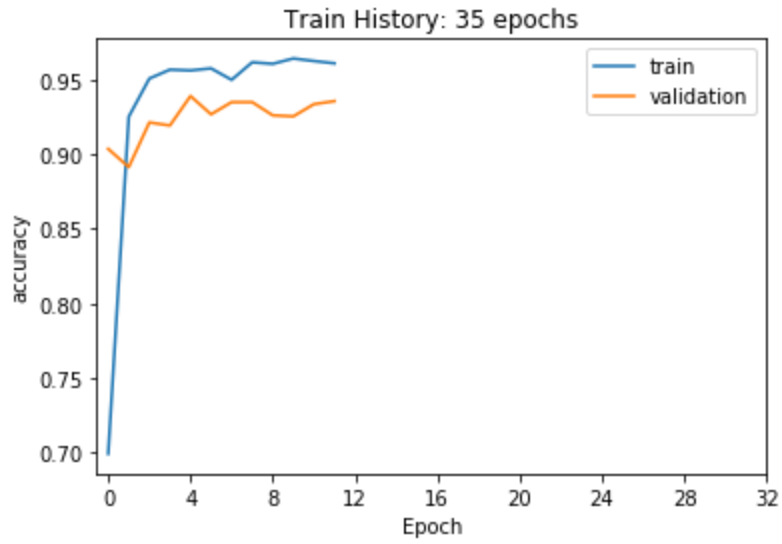
ConvLSTM Model

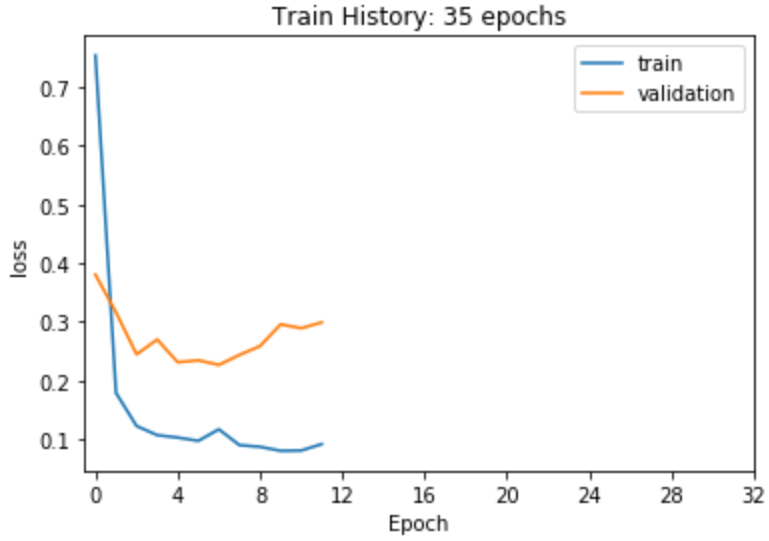
The ConvLSTM is similar in function to the CNN-LSTM. In ConvLSTM, the matrix multiplication calculation of the input with the LSTM cell is replaced by a convolution operation; the convolution is essentially embedded in the architecture. In contrast, CNN-LSTM architecture concatenates the CNN and LSTM architectures together externally.

Again, the model architecture will be defined in a function and a grid search used to evaluate hyperparameter combinations to optimize the model. Then the model with the best parameter combination will be fit to the training dataset and evaluated on the test dataset. See the model architecture and evaluation below.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv_lst_m2d_82 (ConvLSTM2D) (None, 1, 30, 64)         56320
_________________________________________________________________
dropout_85 (Dropout)         (None, 1, 30, 64)         0
_________________________________________________________________
flatten_83 (Flatten)         (None, 1920)              0
_________________________________________________________________
dense_168 (Dense)            (None, 50)                96050
_________________________________________________________________
dense_169 (Dense)            (None, 6)                 306
=================================================================
Total params: 152,676
Trainable params: 152,676
Non-trainable params: 0
_________________________________________________________________
```

Example training history (with early stopping) and prediction performance using the ConvLSTM model are shown below.





Confusion matrix for ConvLSTM evaluation on test set:

	LAY	SIT	STA	WALK	WALK_D	WALK_U
LAY	472	6	18	0	0	0
SIT	8	446	17	0	0	0
STA	2	18	400	0	0	0
WALK	0	10	0	399	82	0
WALK_D	0	0	0	76	456	0
WALK_U	0	27	0	0	0	510

Classification report for ConvLSTM evaluation on test set:

	precision	recall	f1-score	support
LAY	0.98	0.95	0.97	496
SIT	0.88	0.95	0.91	471
STA	0.92	0.95	0.94	420
WALK	0.84	0.81	0.83	491
WALK_D	0.85	0.86	0.85	532
WALK_U	1.00	0.95	0.97	537
accuracy			0.91	2947
macro avg	0.91	0.91	0.91	2947
weighted avg	0.91	0.91	0.91	2947

Comparison to SVM

Both the CNN-LSTM and ConvLSTM performed approximately the same, and not much better than the LSTM alone for this classification problem. None of the deep learning models implemented on the raw data were able to outperform the SVM baseline on the feature engineered data. However, their strong performance on the raw data is impressive given the knowledge required to perform the original feature engineering. Below is a summary of the performance evaluations of all of the tested models. NN performance is an average of 10 runs, as these are non-deterministic.

Hyperparameter Optimization

Model	Hyperparameters Selected	Total Trainable Parameters
SVM	C: 1000 gamma: 0.0001 kernel: rbf	561
LSTM	batch_size: 128 dropout: 0.1 neurons: 200	209,406
CNN-LSTM	batch_size: 64 dropout: 0.1 neurons: 200	933,150
ConvLSTM	batch_size: 128 dropout: 0.3 neurons: 50	152,676

Prediction Time Summary

Model	Prediction Time [s]
SVM	1.45
LSTM	2.48
CNN-LSTM	0.74
ConvLSTM	0.63

Overall Model Accuracy Summary

Model	Average Model Accuracy (+/- Std. Dev.)
SVM	96.17%
LSTM	91.39% (+/-1.06%)
CNN-LSTM	92.01% (+/-1.27%)
ConvLSTM	92.74% (+/-0.56%)

Conclusions

Human activity recognition will continue to be an evolving field for cutting edge signal processing and machine classification methods. As personalized computing devices become smaller and faster, the race to implement the next generation of real-time activity recognition algorithms will continue.

The classification problem investigated in this project consisted of classifying data into 6 highly separable classes. Class separability was observed with various visualization techniques as well as with statistical inference. High class separability was also observed in the very good classification performance of the traditional machine learning model (SVM) and neural network models (RNN, CNN-LSTM, ConvLSTM) implemented.

Each modeling methodology has its benefits and limitations:

- SVM is a lightweight, deterministic ML model that can be quickly trained and deployed. However it requires extensive feature engineering, much of which is not computationally cheap to achieve and makes the code more cumbersome to update if changes on the signal processing side are required. With the proper feature engineering and hyperparameter tuning, it results in very highly accurate and repeatable classification.
- RNNs (alone and in combination with convolutional layers) perform almost as well as the SVM, but require no feature engineering. This makes their code potentially more versatile, though the large number of parameters result in longer training and prediction times. Still, the prediction times are low enough that they could feasibly be used for real-time classification. Another limitation is classification repeatability, as NNs are non-deterministic.