EEE 485 SPRING 2021

Project Final Report

Barış Aşkın Yavuz Yarıcı

1. Task

The availability of affordable wearable equipment and smartphones allows us to collect huge amount of data including motion, location, and environment information. Since smartphones are widespread and they have an accelerometer, gyroscope, and GPS sensors built in them, predicting human activity using smartphone data becomes an active research area. These predictions are especially useful for health care and monitoring applications today.

The goal of this task is to create models that can accurately predict human activity using a smartphone's embedded accelerometer and gyroscope data. Our models will classify activity into six simple activities: walking, walking upstairs, walking downstairs, sitting, standing, and laying.

2. Dataset

Our dataset was prepared and made available by Davide Anguita, et al. from the University of Genova, Italy, and is described in full in their 2013 paper "A Public Domain Dataset for Human Activity Recognition Using Smartphones." [1] Data is collected from 30 volunteers within the age bracket of 19-48. Each participant of the experiment performed six basic activities (walking, walking upstairs, walking downstairs, sitting, standing, and laying.) wearing smartphones. Using the embedded accelerometer and gyroscope of the smartphones, 3-axial linear acceleration and 3-axial angular velocity is captured.

In total the dataset consists of 10301 samples. The volunteers from whom data is collected are randomly separated into two groups where 70% of them are selected for the training and 30% of them are selected for the test.

The data captured from accelerometer and gyroscope sensors were pre-processed by applying noise filters and then sampled in fixed-width sliding. The data from the acceleration sensor signal was separated into body acceleration and gravity using a Butterworth low-pass filter. From each window, a vector of 561 features was obtained by calculating variables from the time and frequency domain. Examples of these features are the mean of the acceleration in x, the correlation between x and y, and the angle between gravity and acceleration. The final data set is available at UCI machine learning repository. [2]

Distribution of the dataset to the classes plotted on the figure 1. Although, there is some imbalance in the distribution, classes are balanced enough for our purposes.

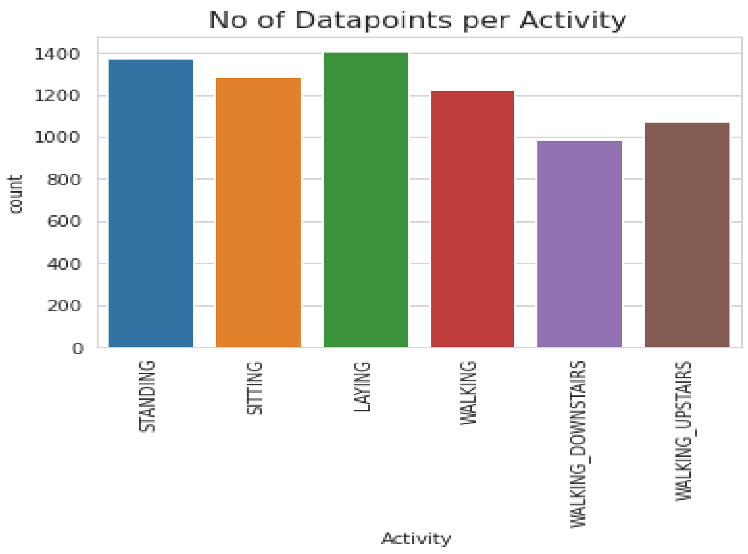


Figure 1**:** Number of Dataset of Each Class

2.1 Data Preprocessing (Normalization and Principal Component Analysis):

Since different features of our data have different scales, which are different means and variances, the dataset is normalized. During normalization, all feature vectors are scaled such that they have zero mean and unit variance. Sample mean and sample variance are calculated on training data. Then using these values, both training and test data are normalized. Below, the mathematical procedure for normalization of feature j is shown where n represents the number of samples in train set:

$$\hat{\mu}_j \coloneqq \frac{\sum_{i=1}^{n} X_{train,i,j}}{n} \; and \; \widehat{\sigma_j} \coloneqq \sqrt{\frac{\sum_{i=1}^{n}\left(X_{train,i,j} - \hat{\mu}_j\right)^2}{n}} \tag{1}$$

$$x_{train,i,j} \coloneqq \frac{x_{train,i,j} - \hat{\mu}_j}{\widehat{\sigma_j}}, x_{test,i,j} \coloneqq \frac{x_{test,i,j} - \hat{\mu}_j}{\widehat{\sigma_j}}, \forall x_{train,i,j} \in X_{train}, \forall x_{test,i,j} \in X_{test} \tag{2}$$

After normalization Principal Component Analysis (PCA) is applied to reduce dimension of our data. Since each sample has 561 features, it is reasonable to think that some features have high correlation. While using PCA, a parameter for the number of selected principal components (PC) should be determined. For this parameter selection, the effect of number of principal components on total variance explained (TVE) is observed. A threshold is determined as 90% of TVE. We could reach 90% TVE with 63 PCs. The graph could be seen in Figure 2:

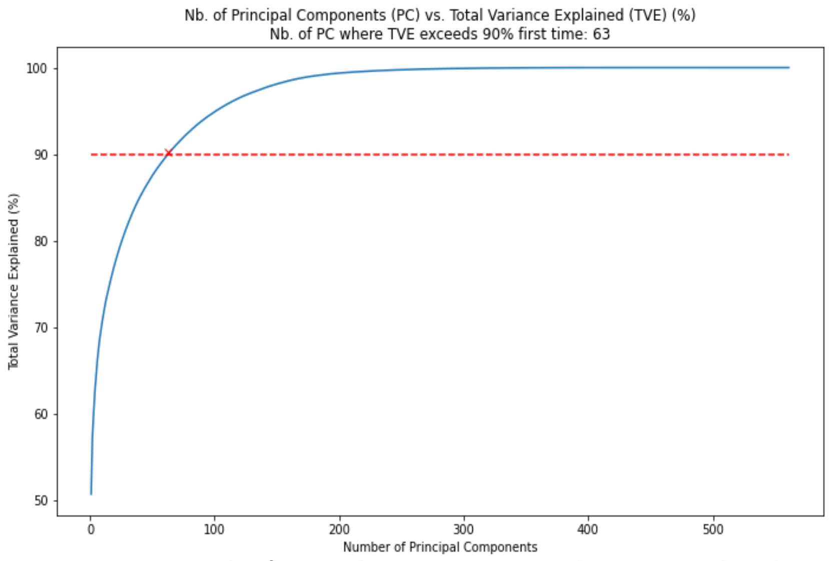


Figure 2: Nb. of Principal Components vs. Total Variance Explained

3. Methods

3.1 How Methods are Chosen

To select the methods for our problem, whether the methods are linear or nonlinear is considered. We selected SoftMax classifier, which is a linear method and K – Nearest Neighbors (kNN) which is a nonlinear method. For the third method, we selected Multiclass Support Vector Machine since it can be both a linear method with linear kernel and nonlinear method with other kernel tricks.

Softmax Regression and Multiclass Support Vector Machine (SVM) with linear kernel are both linear multiclass classification methods. Since our dataset consists of high dimensional feature space (each sample has 561 features in dataset), we thought it is highly possible that there might be a linear relationship between some features and classes. Moreover, the methods we chose for linear classifiers have a distinct difference. While Softmax Regression gives the estimated class probabilities as output, SVM only returns the estimated class type as output. Knowing class probabilities makes output more interpretable. Hence, we think that using both of these linear methods enriches our results.

As nonlinear classifiers, we used SVM with nonlinear kernel (Gaussian kernel) and kNN. kNN gives complex shaped decision boundaries. Also, one of the reasons why we chose kNN is that our dataset consists of 7352 training samples and it is known that kNN may work well with large datasets [12]. Moreover, nonlinear kernels which we used for SVM tries to map non-linear separable dataset into another feature space in which dataset is linearly separable with linear methods [11]. Hence, we thought that it would be beneficial to try kernel trick.

3.2 Multinomial Logistic Regression (SoftMax Regression):

Logistic regression is used when the target variable is binary. It is a transformation of the liner model to the probabilistic model. Thus, it can be considered as a special case of liner regression. In logistic regression main assumption is that the log-odds (logit is) are linearly dependent to the data [3]. For transformation from the liner model to the probabilistic model, logistic regression uses logistic function as in equation 3.

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1+e^{\beta_0+\beta_1 X}} \tag{3}$$

This function gives values between 0 and 1. Using this function as probabilistic model logarithm of odds ratio gives liner model as in equation 4.

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X \tag{4}$$

Classical logistic regression classifies the data into two class 1 or 0. When there are multiple classes larger than 2, SoftMax regression is used. SoftMax regression is a generalized version of the logistic regression when there are multiple classes. We want our model to estimate the probability of $P(y = k\ |X)$ for each value of k. For each of the classes, we want to get the probability of the labels taking on each of the k different values. SoftMax regression hypothesis can be seen in equation 5. Notice that there are k probabilities and they sum up to one [4].

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix} \tag{5}$$

Since our dataset has 6 different classes, this method is appropriate for our classification task. To implement this method, we treated correct class as 1 and others as 0 and we found corresponding probability for that class. For each class, we trained different models. To implement logistic regression, gradient descent is used to find maximum likelihood function.
Loss function of the SoftMax can be given as:

$$L(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^i = k\} * \log \frac{\exp\left(\theta^{k^T} x^i\right)}{\sum_{j=1}^{K} \exp\left(\theta^{k^T} x^i\right)}\right] \tag{6}$$

Where K is the number of class and m is the number of samples. Taking the gradient of this loss function we get:

$$\nabla L(\theta) = \sum_{i}^{m} x^i \left(1\{y^i = k\} - P\left(y^i = k \middle| x^i;\theta\right)\right) \tag{7}$$

Gradient Descent update rule can be written as:

$$\theta = \theta - \alpha.\nabla L(\theta) \tag{8}$$

$'\alpha'$ is learning rate parameter. This parameter can be constant or it can be decaying function of iteration number. One of the common learning rate decay can be given with equation 9.

$$\alpha = learning\ rate \cdot \frac{1}{1 + decay \cdot iteration\ number} \tag{9}$$

<u>Our SoftMax Regressions Implementation:</u>

In SoftMax regression implementation, loss function is calculated as mentioned above. For each iteration, gradients are calculated and weights are updated according to equation 8. In equation 8, alpha is the learning rate for our implementation and we used decaying learning rate according to equation 9. To find values for learning rate parameter and decay parameter, 10-fold cross validation method is used. The details of validation are discussed in section 4 Validation Methods. Optimum decay rate and learning rate is found with grid search. The results of the grid search can be seen in figure 3. Highest accuracy is achieved with learning rate 0.04 and decay rate 0.05. To prevent overfitting L2 regularization is used. Regularization parameter is chosen with cross validation same as before. As it can be seen in figure 4, highest accuracy is achieved with 0.05 regularization parameter. With these parameters, model is trained until change in loss function becomes less than 0.01%. Stopping condition is reached after 720 iterations. Stopping condition vs iteration can be seen in figure 5. We extracted the accuracy after each iteration for both validation sets and training set. After training is completed, we take the average of 10 training for 10 folds. Also, we found the test set accuracy as 92.8% with trained model. Results are given in section 7. Lastly, we extracted the confusion matrix for test dataset. Accuracy metric that we used this project is the ratio of the correct predictions to the total number of predictions.

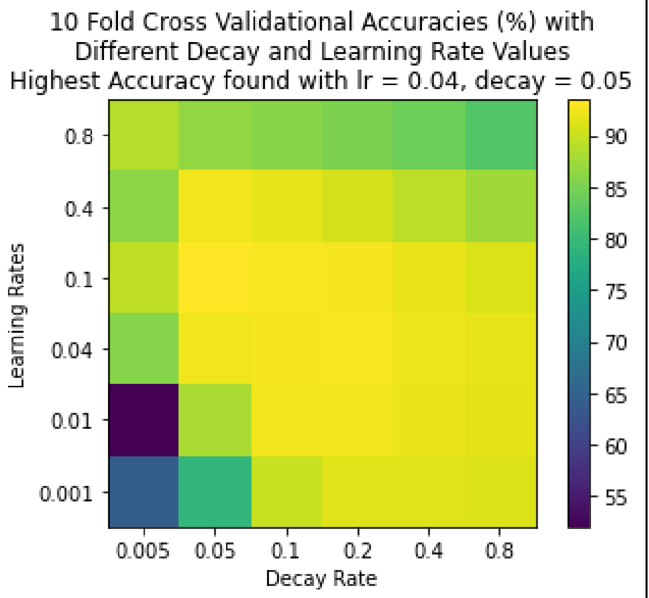


Figure 3: 10-Fold CV Results for SoftMax Parameter Selection

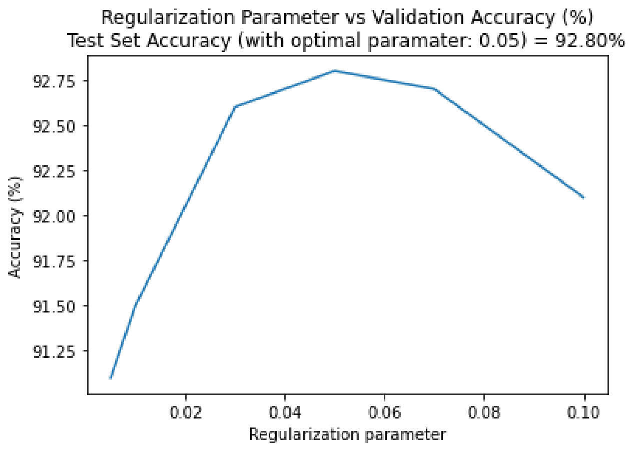


Figure 4: 10-Fold CV Results for SoftMax Regularization Selection

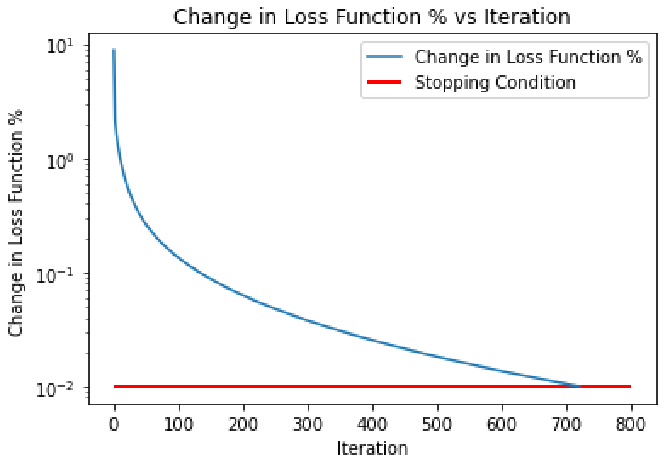


Figure 5: Stopping Condition for SoftMax Regression

3.3 Multiclass Support Vector Machine

The aim of the support vector machine (SVM) classifier is to find a hyperplane which separates data samples in N dimensional space. Here, N is the number of features of each data sample. There are many decision boundaries can be found which may classify the points. SVM tries to maximize the minimum distances between decision plane and nearest data samples, which are called support vectors, to this plane. This distance is called "margin" [5].

The hyperplane of SVM classifier is learnt with data. It is represented with "weights" of classifier. Weight is the vector for hyperplane and represented with "w". In multiclass SVM, the aim is to find hyperplanes which each of them classifies one distinct class. Namely, multiclass SVM tries to make correct class's score higher than other classes [6]. Here, score is inner product of the hyperplane weight and data sample vectors, which represents where the data sample lies with respect to hyperplane. Throughout the training process, hinge loss is used and can be seen in equation (10). The loss function penalizes wrong classifications, which happen if score of wrong class is higher than correct one. However, if the score of wrong class is smaller than score of correct class with some margin "Δ", it does not contribute the loss. A contribution of one data instance to the loss is below where x_i is data instance, y_i is correct class, w_j is the weight of j^{th} class and w_{y_i} is the weight of correct class.

$$L_i = \sum_{j \neq y_i} \max\left(\Delta + x_i^T w_j - x_i^T w_{y_i}, 0\right) \tag{10}$$

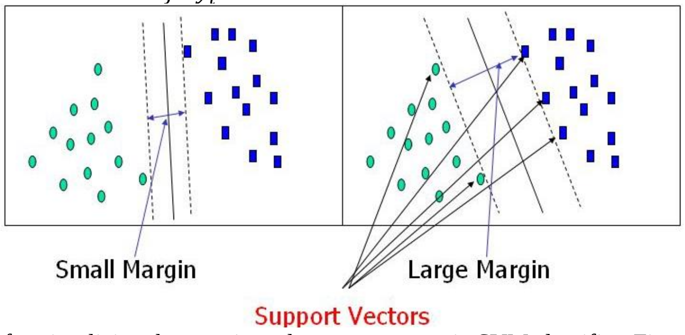


Figure 6: Figure for visualizing the margin and support vectors in SVM classifier. Figure is retrieved from [5]

Also, L2 norm weight regularization is applied for adjusting flexibility of weights with loss given below:

$$R(W) = \sum_{j=1}^{\#of\ classes} \sum_{k=1}^{\#of\ features} w_{j,k}^2 \tag{11}$$

The relation between SVM loss and regularization is controlled with parameter C and total loss becomes:

$$Loss = C\frac{1}{N}\sum_i L_i + R(W) = \frac{C}{N}\sum_i \sum_{j \neq y_i} \max\left(\Delta + x_i^T w_j - x_i^T w_{y_i}, 0\right) + \sum_{j=1}^{\#of\ classes} \sum_{k=1}^{\#of\ features} w_{j,k}^2 \tag{12}$$

To minimize the loss function, gradient descent method is used as in logistic regression with similar update rule in equation (8). To calculate gradient, one need to consider whether it is correct class's weight or not:

$$\nabla_{w_j} Loss = \begin{cases} \frac{C}{N}\sum_i -x_i \sum_{j \neq y_i} 1\left(x_i^T w_j - x_i^T w_{y_i} + \Delta > 0\right) + 2w_j \ , if\ j = y_i \\ \frac{C}{N}\sum_i 1\left(x_i^T w_j - x_i^T w_{y_i} + \Delta > 0\right) + 2w_j \ , if\ j \neq y_i \end{cases} \tag{13}$$

where 1 is considered as indicator function. Then using α step size, update rule becomes:

$$w_j = w_j - \alpha \,.\, \nabla_{w_j} Loss\ (W)$$

Our SVM Classifier Implementation:

As one can notice, there are two parameters C and Δ. They actually control the same tradeoff between regularization loss and data loss [6]. So, in our implementation Δ is fixed as 1 and optimal C value is selected. Moreover, learning rate (lr (α)) is a parameter. C and α parameters are selected with 10-fold cross validation method. The details of validation are discussed in section 4 Validation Methods. We basically run grid search by trying different C and α parameters and find the optimal parameters as seen in Figure 7. After optimal parameters are found as 10^{-3} for learning rate and 1000 for C, we trained our SVM model. We extracted the accuracy after each iteration for both validation sets and training set. After training is completed, we take the average of 10 training for 10 folds. Also, we found the test set accuracy as 89.38% with trained model. Results are given in section 7. Lastly, we extracted the confusion matrix and other evaluation scores for test dataset which is discussed in result section.

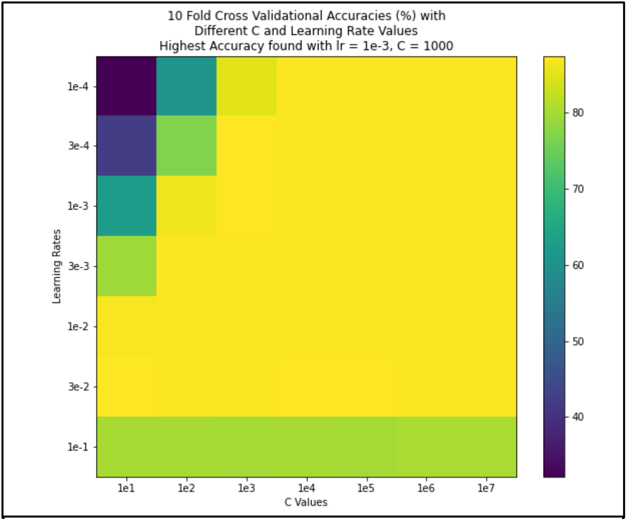


Figure 7: 10-Fold CV Results for SVM Parameter Selection

Moreover, another trial with SVM is done with a kernel trick which is Gaussian Kernel (Radial Basis Function). Kernel tricks are frequently used with SVM to make the model nonlinear [11]. In our project, since we have 2 linear classifiers (SVM and SoftMax Regression), it would be beneficial to see the nonlinear kernel's effect. With Gaussian Kernel, test set accuracy is found as 78.21%. As seen, Gaussian Kernel gives worse test set accuracy than linear kernel in our problem. The reason for this situation is discussed in Results section.

Total iteration number of SVM classifier is selected with trials. We plotted and observed the error rates change for different C and learning rate parameters for all folds. We concluded that 100 iterations are enough for our purpose.

3.4 K Nearest Neighbors

K nearest neighbor (kNN) is a supervised classification algorithm which regards stored data as a data space and using it for future data predictions. It can be used for both regression and classification tasks [8]. In our project, it is used for classification. The algorithm works as follows: When new data sample comes, the distance between this sample and all the stored data samples are calculated. Then the closest k stored data samples are selected, and new data sample's class is estimated based on closest k data samples. Different distance metrics can be used.

Moreover, in this method, k is a free parameter, and it determines the tradeoff between bias and variance. As 'k' increases, the bias increases while variance decreases since the number of samples used for prediction increases. On the contrary, decrement in k results in lower bias [9]. How classification is affected by k parameter can be seen in Figure 8 with a binary classification task.

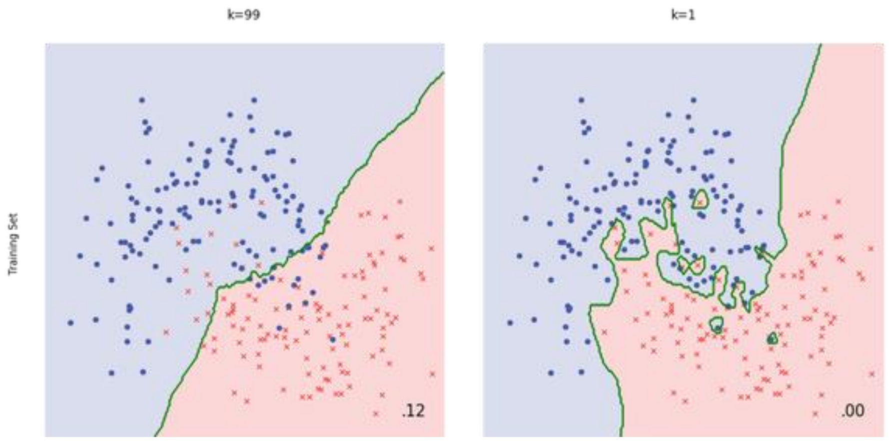


Figure 8: Bias - Variance tradeoff is controlled with k value. As seen when k = 1, bias is low but variance is high. It is reverse when k = 99. Figure is retrieved from [10].

<u>Our kNN Classifier Implementation:</u>

In kNN, as mentioned above, the distance metric is free to choose. In our task, we choose two of the most popular distance metrics which are L1 (Manhattan Distance) and L2 (Euclidian Distance). Their formulas can be seen in equation 14.

To implement kNN, first issue we handle is to select k parameter. We used 10-fold cross validation method to choose this parameter for both of distance metrics. Same method is followed with parameter selection of previous methods. K-Fold Cross Validation is discussed more detailly in next section. For every k value, we run 10-fold cross validation and get the average accuracy for 10 validation folds. Then we plotted the graph for k vs. validation accuracy. It can be seen in Figure 9. Then k is selected as 3 for L1 norm and 9 for L2 norm. After selecting k, we run the kNN algorithm with our test data. Test accuracy is found as 84.46% with L1 and 86.66% with L2. Finally, we extracted the confusion matrix for test dataset. The results are discussed in Results section.

$$d_{l2}(x_1, x_2) = \|x_1 - x_2\|_2 = \sqrt{\sum_p \left(x_{1,p} - x_{2,p}\right)^2} \; and \; d_{l1}(x_1, x_2) = \|x_1 - x_2\|_1 = \sum_p \left|\left(x_{1,p} - x_{2,p}\right)\right| \quad (14)$$

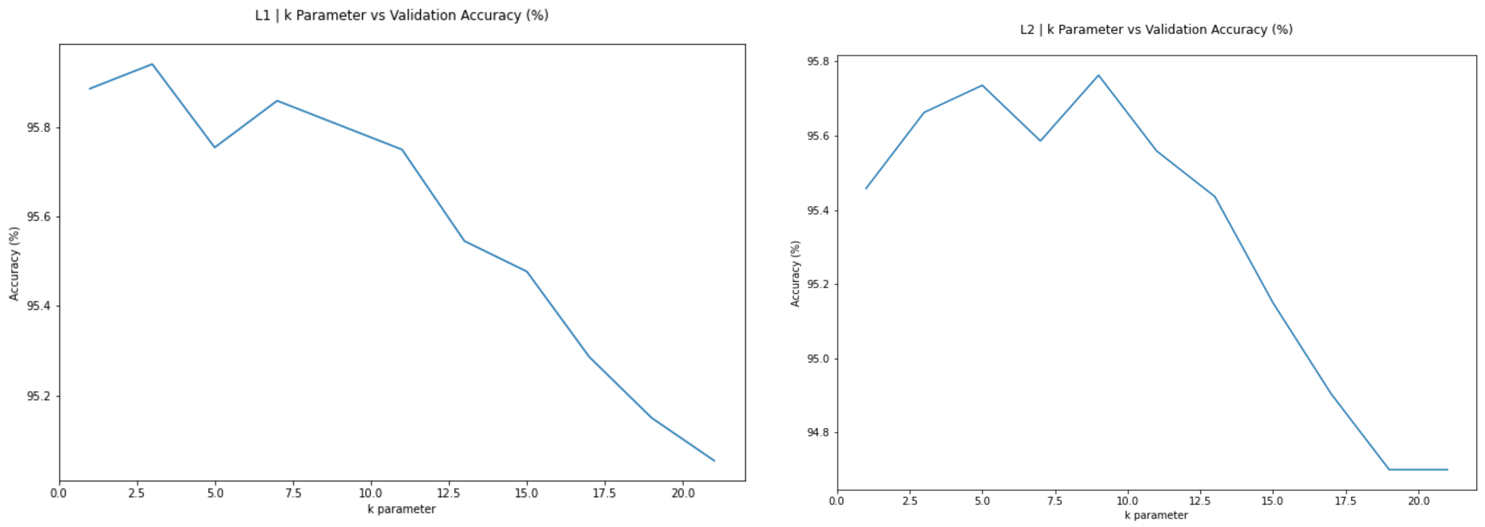


Figure 9: 10-Fold CV Results for kNN Parameter Selection with L1 (left) and L2 (right) norms. k=3 and k=9 are selected for L1 and L2 respectively.

4. Validation Method

As a validation method, we used 10-fold cross validation method. We shuffled training set and divide it into 10 different folds. To implement 10-fold cross validation, for each of the kth "folds", a new model is trained using 9 training folds and the obtained model is validated on the remaining 1 validation fold. After that, the performance measures are calculated by taking the average of all folds' separate performances. After training/validation process is completed, test performance measures are calculated with unseen test dataset.

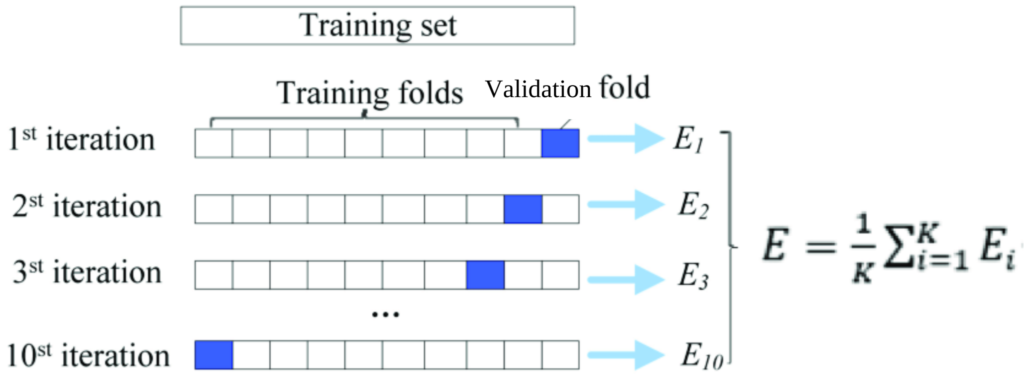
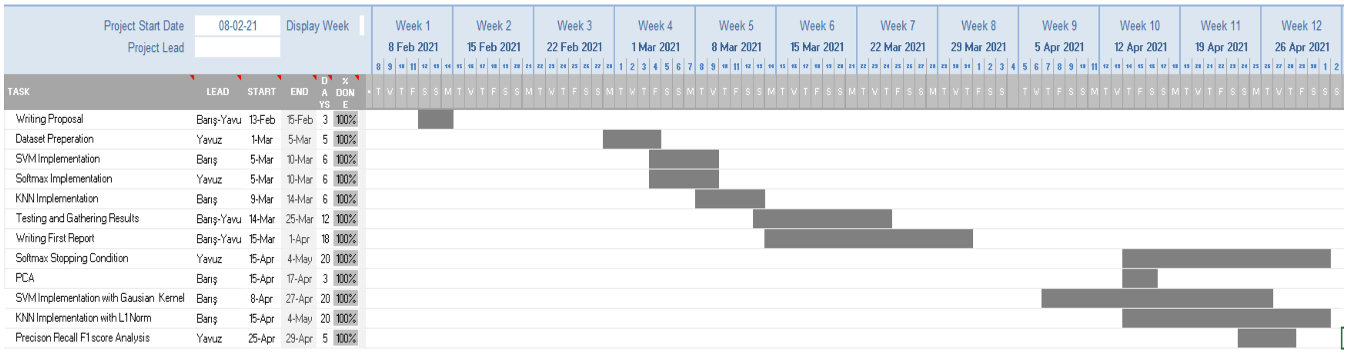


Figure 10: 10-Fold Cross Validation

5. Gannt Chart

The Gannt chart of our project can be seen in Figure 11.



	Project Start Date	08-02-21	Display Week		
	Project Lead				
TASK	LEAD	START	END	DAYS	% DONE
Writing Proposal	Barış-Yavu	13-Feb	15-Feb	3	100%
Dataset Preperation	Yavuz	1-Mar	5-Mar	5	100%
SVM Implementation	Barış	5-Mar	10-Mar	6	100%
Softmax Implementation	Yavuz	5-Mar	10-Mar	6	100%
KNN Implementation	Barış	9-Mar	14-Mar	6	100%
Testing and Gathering Results	Barış-Yavu	14-Mar	25-Mar	12	100%
Writing First Report	Barış-Yavu	15-Mar	1-Apr	18	100%
Softmax Stopping Condition	Yavuz	15-Apr	4-May	20	100%
PCA	Barış	15-Apr	17-Apr	3	100%
SVM Implementation with Gausian Kernel	Barış	8-Apr	27-Apr	20	100%
KNN Implementation with L1 Norm	Barış	15-Apr	4-May	20	100%
Precison Recall F1 score Analysis	Yavuz	25-Apr	29-Apr	5	100%

Figure 11: Gannt Chart of Work Packages

6. Results

All simulations are done on CPU, intel core i7 6700HQ processor. Python 3 is used as programing languages for building all models. In this section, each methods' individual results are shown and discussed separately. After that, they are discussed together.

6.1 Multinomial Logistic Regression (SoftMax Regression):

SoftMax regression performed very well for our task. We get train accuracy of 93.62% test accuracy of 92.81 % with the optimum parameters chosen with 10-fold cross validation as explained section 3.2. Training of the logistic regression is slower compared to other models. Training time can be decreased by changing stopping condition or increasing learning rate. Increasing the stopping condition limit will decrease iterations and decrease our iteration. However, when we increase learning rate, the learning process starts to oscillate and it overshoots the optimum point which result in lower accuracy. In the final phase, we applied PCA, our weight matrices dimension is decreased and training time of the SoftMax regression is decreased as well.

In order to see the classification of the samples in test set we plotted confusion matrix for SoftMax regression on figure 13. As it can be seen in this figure, our model has confusion on sitting-standing and walking-walking downstairs-walking upstairs. Since, the data taken from sensors are similar for these actions, our models are not able to differentiate these classes properly as expected. Precision, recall and F1 scores for SoftMax regression are tabulated on Table 1. As it can be seen in this table, recall for Laying is 1 which shows that all samples that has laying as true label correctly classified as Laying. Since action of laying is different than other classes in terms of accelerometer and gyroscope data, it is expected that these samples are easily differentiated by our model.

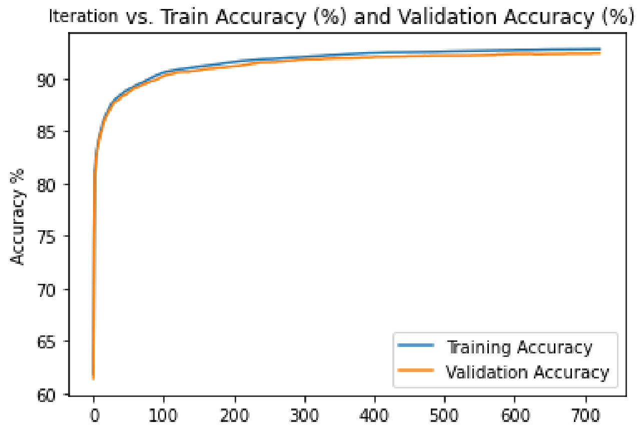
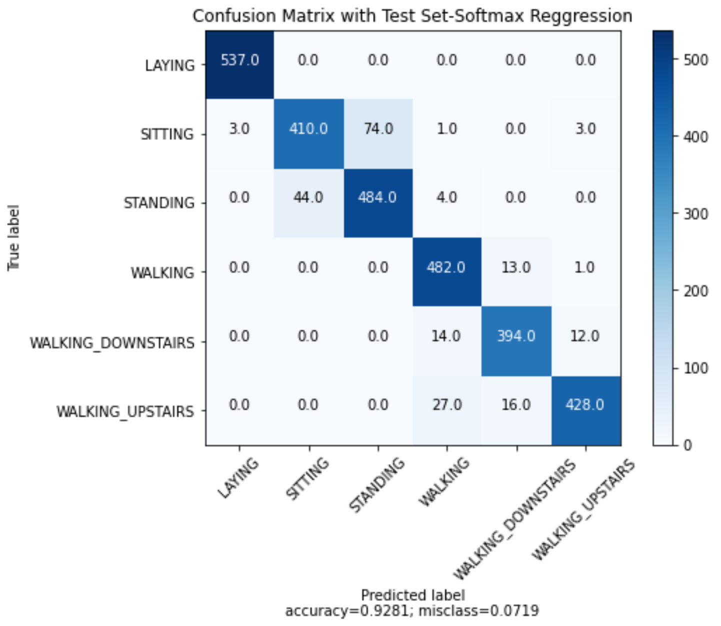


Figure 12: Iteration vs Train Accuracy for SoftMax Classifier



	LAYING	SITTING	STANDING	WALKING	WALKING_DOWNSTAIRS	WALKING_UPSTAIRS
LAYING	537.0	0.0	0.0	0.0	0.0	0.0
SITTING	3.0	410.0	74.0	1.0	0.0	3.0
STANDING	0.0	44.0	484.0	4.0	0.0	0.0
WALKING	0.0	0.0	0.0	482.0	13.0	1.0
WALKING_DOWNSTAIRS	0.0	0.0	0.0	14.0	394.0	12.0
WALKING_UPSTAIRS	0.0	0.0	0.0	27.0	16.0	428.0

Figure 13: Test Set Confusion Matrix with SoftMax Classifier

Classes	Precision	Recall	F1 Score
Laying	**0.99**	**1**	**0.99**
Sitting	**0.90**	**0.84**	**0.87**
Standing	**0.87**	**0.91**	**0.89**
Walking	**0.91**	**0.97**	**0.94**
Walking Downstairs	**0.93**	**0.94**	**0.93**
Walking Upstairs	**0.96**	**0.91**	**0.94**

Table 1: Precision, Recall and F1 Score for SoftMax Regression

6.2 Support Vector Machine

Support vector machine with linear kernel reaches the test accuracy 89.38% with the optimum parameters chosen with 10-fold cross validation as explained before. When the Gaussian Kernel is used with same procedure, test accuracy becomes 78.21%. Hence, while comparing SVM method with other methods, we consider the results of linear kernel.

SVM with linear kernel gives high test accuracy. It might be claimed that our dataset is linearly well separable after applying PCA. On the other hand, Gaussian Kernel could not reach as high accuracy as linear kernel. This might result from the size of the dataset. Gaussian Kernel calculates the similarity between sample points according to the Gaussian similarity function and use them as features of dataset. However, our dataset consists of 7352 training samples and this results in very high feature dimension with kernel. Therefore, while nonlinearity is added with kernel, model accuracy might drop due to the huge size of dataset.

For linear kernel, the graph of training and validation accuracy for each iteration can be seen in Figure 14. Train, validation and test accuracies are obtained as 88.4%, 87.6%, 89.4% respectively. One may claim that SVM method is able to capture the possible different distribution of data because trained model without seeing test set could reach high accuracy on test set as well as train set. Furthermore, the confusion matrix of SVM method on test set estimations is extracted and can be seen in Figure 15. Also, precision, recall and F1 scores are given for each class in the table below. It could be seen that laying is the least confused class. It is an expected result because our dataset comprises of mobile phone sensor data and laying activity involves most dissimilar actions from others. Comparing the precision scores of each class, it could be said that "Standing" is most likely to be labeled as false positive. By looking confusion matrix, one can see that 88 of 491 "sitting" actions are labeled as "standing". About recall scores, it could be said that "Sitting" and "Walking Downstairs" are the worst classes in terms of false negatives. In confusion matrix, it is seen that these classes are frequently confused with "Standing" and "Walking/Walking Upstairs" respectively.

Classes	Precision	Recall	F1 Score
Laying	**0.99**	**0.97**	**0.98**
Sitting	**0.87**	**0.81**	**0.84**
Standing	**0.84**	**0.90**	**0.87**
Walking	**0.89**	**0.95**	**0.92**
Walking Downstairs	**0.87**	**0.82**	**0.84**
Walking Upstairs	**0.90**	**0.90**	**0.90**

Table 2: Precision, Recall and F1 Score for SVM

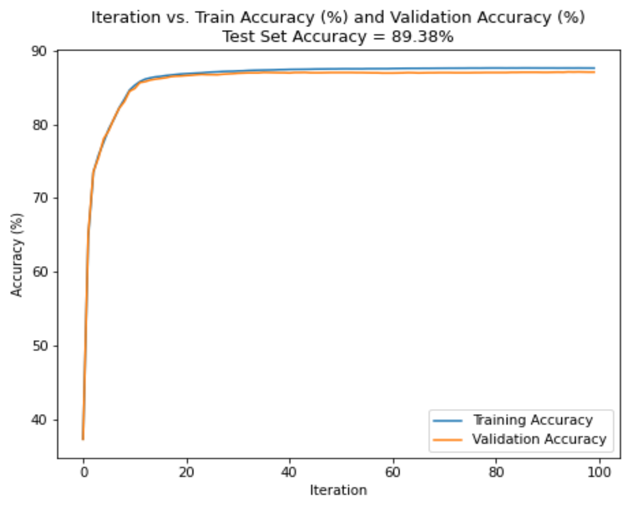
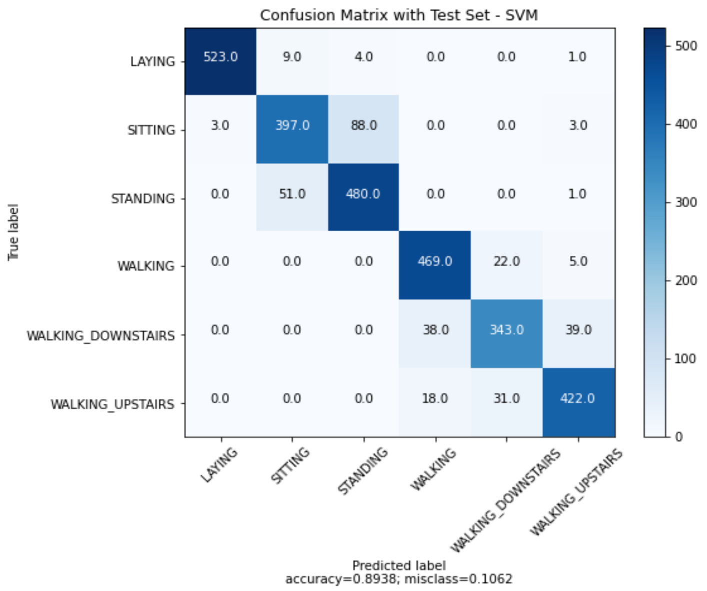


Figure 14: Iteration vs Train Accuracy for (Linear) SVM Classifier



	LAYING	SITTING	STANDING	WALKING	WALKING_DOWNSTAIRS	WALKING_UPSTAIRS
LAYING	523.0	9.0	4.0	0.0	0.0	1.0
SITTING	3.0	397.0	88.0	0.0	0.0	3.0
STANDING	0.0	51.0	480.0	0.0	0.0	1.0
WALKING	0.0	0.0	0.0	469.0	22.0	5.0
WALKING_DOWNSTAIRS	0.0	0.0	0.0	38.0	343.0	39.0
WALKING_UPSTAIRS	0.0	0.0	0.0	18.0	31.0	422.0

Figure 15: Test Set Confusion Matrix with trained (Linear) SVM Classifier

6.3 K nearest Neighbors

K – nearest neighbor algorithm could reach 84.46% and 86.66% accuracies on test set using all training set as a reference with L1 and L2 distance metrics respectively. For both metrics, the parameter k is selected with 10-Fold Cross Validation as explained in Section 3.4 kNN – Our Implementation. For both metrics it is observed that there is a difference in accuracy for train/validation set and test set. This difference may be seen unusual for someone who do not know the structure of dataset. In our dataset, samples are collected from 30 volunteers. Train and test sets are separated as 21 – 9 people's samples [2]. During validation learning, to obtain fold's validation sets, we randomly shuffled training set. It means that, in train and validation set, there are samples taken from the same volunteers. However, in test set, completely different volunteers' samples are used. Then, since kNN directly measures the samples distances in feature space, it is expected that test accuracy might be significantly lower than validation accuracy like in our case.

Moreover, an accuracy difference between kNNs with L1 and L2 norms is encountered. While L2 norm takes the squares of difference of each feature, L1 norm takes absolute value of differences. It might be said that compared to the L1 norm, L2 norm penalizes outlier samples more because there are far away from others and L2 norm applies taking square of these large differences. Since our feature dimension is high, it is hard to anticipate which norm works better in our problem. Hence, we tried both of them and saw that L2 norm works better.

Furthermore, we extracted confusion matrices, recall, precision and F1 scores for both trials. They could be seen below. Both results show that kNN works poor in terms of recall score for "Sitting" and "Walking Downstairs". In confusion matrices, it is seen that "Sitting" is mostly false classified as "Standing" and "Walking downstairs" mostly false classified as "Walking" and "Walking Upstairs".
Since kNN with L2 norm works better in our problem, we will use its results for comparison with other methods.

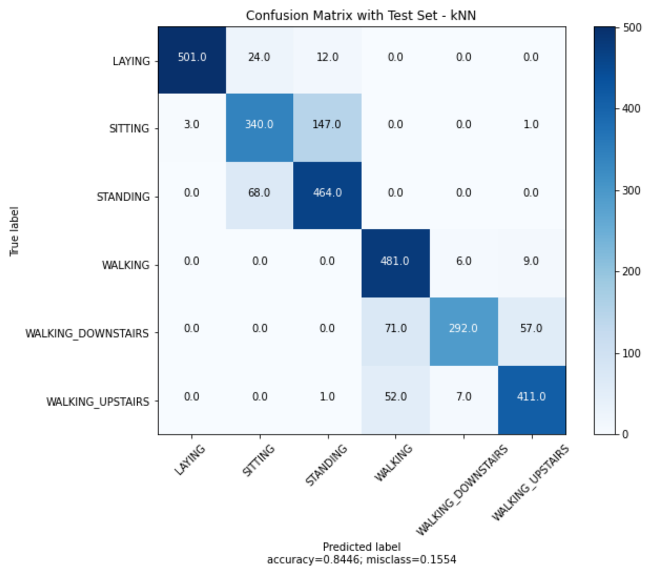


Figure 16: Test Set Confusion Matrix with kNN Classifier (L1)

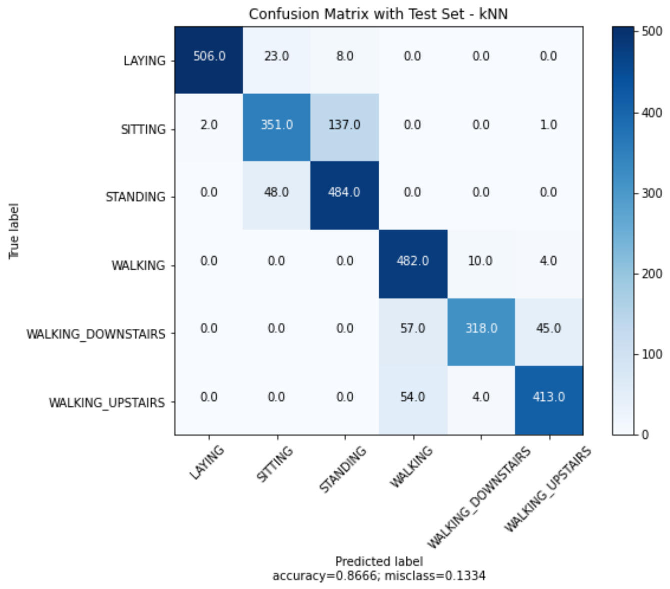


Figure 17: Test Set Confusion Matrix with kNN Classifier (L2)

	kNN with L1 distance metric			**kNN with L2 distance metric**		
Classes	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Laying	**0.994**	**0.933**	**0.963**	**0.996**	**0.942**	**0.968**
Sitting	**0.787**	**0.692**	**0.737**	**0.832**	**0.715**	**0.769**
Standing	**0.744**	**0.872**	**0.803**	**0.769**	**0.910**	**0.834**
Walking	**0.796**	**0.970**	**0.875**	**0.813**	**0.972**	**0.885**
Walking Downstairs	**0.957**	**0.700**	**0.810**	**0.958**	**0.757**	**0.846**
Walking Upstairs	**0.860**	**0.873**	**0.866**	**0.892**	**0.877**	**0.884**

Table 3: Precision, Recall and F1 Score for kNN

Classes	Metrics	SoftMax Regression	SVM	kNN
Laying	True Positives	537	523	506
	False Positives	3	3	2
	False Negatives	0	14	31
Sitting	True Positives	410	397	351
	False Positives	44	60	71
	False Negatives	81	91	140
Standing	True Positives	484	480	484
	False Positives	74	92	145
	False Negatives	48	52	48
Walking	True Positives	482	469	482
	False Positives	46	56	111
	False Negatives	14	27	14
Walking Downstairs	True Positives	394	343	318
	False Positives	29	53	14
	False Negatives	26	77	102
Walking Upstairs	True Positives	428	422	413
	False Positives	16	52	50
	False Negatives	43	49	58
Training Time (s)		38.4	0.41	-
Execution Time (s)		0.25	0.17	13.2
Training Accuracy (%)		93.7	88.4	-
Validation Accuracy (%)		92.9	87.6	95.76
Test Accuracy (%)		92.8	89.38	86.66

Table 4: Comparison of all Results. In the table, True Negatives are intentionally not given since our task is multiclass. For SVM, Linear Kernel's and for kNN, L2 Norm's Results are given in table because these kernel and norm choices gave the best accuracies for these methods as explained before.

6.4 Discussion of All Methods

Both SoftMax and SVM methods have close training and test accuracies which shows that overfitting or underfitting do not occur for these methods. Test accuracy for SoftMax regression is 92.8%, With this accuracy SoftMax regression outperformed other methods. SVM also performed very well by reaching the test accuracy 89.4%. Since our dataset is high dimensional, linear models like linear SVM and SoftMax is expected to perform well. As seen in Section 3.3 Our SVM Implementation, SVM with Gaussian Kernel could not reach as much test accuracy as linear SVM. This may result from the large size of our dataset. After we used Gaussian Kernel to make the model nonlinear classifier, our data get higher dimensional features, and this dropped the test accuracy to 78.21%. kNN is the best classifier in terms of Validation set accuracy. We have not defined the training accuracy of kNN because we thought that it is meaningless. Since kNN looks at data points in train set to estimate the label of data, accuracy over train set may reach 100% by simply picking k = 1. A significant gap is observed between the validation and test accuracy of kNN. This is due to our dataset's characteristic and this issue is discussed detailly in Section 6.3 – Results of kNN.

Training of SoftMax takes 38.4 seconds, and Training of SVM takes 0.25 seconds. Training time of SoftMax regression is much more than SVM. SVM is converging much faster than SoftMax. One of the reasons for that is the learning rate of the SoftMax regression. When learning rate of the SoftMax regression is large, it cannot converge to local minimum, it overshoots and oscillates around local minimum. Thus, learning rate of the SoftMax regression is chosen as small and decay is applied. In this way more robust converging is obtained. For kNN model, there is no training but estimating whole test set takes 13.2 seconds. Since kNN calculates distance to every train sample for each test sample, it is expected to take more than 10 seconds. However, execution time of SoftMax and SVM are less than 1 seconds. Since estimation of test set just requires multiplying test samples with trained weight, execution time is expected to be low. Our Original dataset has 561 dimensions, and this is reduced to 63 with PCA. Reducing the dimension significantly decreased our both training and execution time.

Looking at the confusion matrices of all models, we realized similarities. All models have similar misclassification. Classification of the Laying action is the most successful classification. Although for kNN some of the laying actions classified as sitting, they are mostly classified correctly. For SoftMax recall value for laying is 1 which shows that all actions are classified correctly, and precision is 0.99 due to some of the misclassified sitting actions. Another similarity is the confusion between sitting and standing actions. Since these actions are inverse of each other it is expected that data taken from the gyroscope is similar for these actions and they are misclassified. Because of this misclassification, recall and precision values for these two classes are the lowest for all models. Another similarity between confusion matrix is the confusion between walking, walking downstairs and walking upstairs. All three actions are misclassified as each other. Especially walking downstairs and walking upstairs are classified as walking, which also can be seen in the precision values of the walking. Precision values of the walking is significantly lower especially for kNN model.

7. Conclusion

In the final phase, we successfully run three models for our task. Test performance of our models reached 90% accuracy which is higher than what we expected. During this project, we have learned lots of thing about the topics that we have covered in class. Since we have not used any machine learning libraries for building our models, we have learned how to implement mathematical models that we learned in class. To build our models, we have used our knowledge about the linear algebra and probability. We have used SVM and kNN which are not covered in this class; hence, we have been pushed to learn these methods ourselves and implement them. This was a great experience since it improves both our knowledge about Statistical Learning and our self-learning abilities.

For feature improvements, different learning methods and different feature selection methods might be used.

References

[1] D. Garcia-Gonzalez, D. Rivero, E. Fernandez-Blanco, and M. R. Luaces, “A Public Domain Dataset for Real-Life Human Activity Recognition Using Smartphone Sensors,” *Sensors*, vol. 20, no. 8, p. 2200, Apr. 2020.

[2] “Human Activity Recognition Using Smartphones Data Set,” *UCI Machine Learning Repository: Human Activity Recognition Using Smartphones Data Set*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones. [Accessed: 14-Feb-2021].

[3] J. Brownlee, "Multinomial Logistic Regression With Python", *Machine Learning Mastery*, 2021. [Online]. Available: https://machinelearningmastery.com/multinomial-logistic-regression-with-python/. [Accessed: 14- Feb- 2021].

[4] “SoftMax Regression,” deeplearning.stanford. [Online]. Available: http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression. [Accessed: 10-Mar-2021].

[5] R. Gandhi, "Support Vector Machine — Introduction to Machine Learning Algorithms", *Medium*, 2018. [Online]. Available: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47. [Accessed: 25- Mar- 2021].

[6] L. Miranda, "Implementing a multiclass support-vector machine", 2017. [Online]. Available: https://ljvmiranda921.github.io/notebook/2017/02/11/multiclass-svm/. [Accessed: 24- Mar- 2021].

[7] D. Wilimitis, "The Kernel Trick", *Medium*, 2018. [Online]. Available: https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f. [Accessed: 27- Mar- 2021].

[8] O. Harrison, "Machine Learning Basics with the K-Nearest Neighbors Algorithm", *Medium*, 2018. [Online]. Available: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761. [Accessed: 25- Mar- 2021].

[9] T. Lin, "Day 3 — K-Nearest Neighbors and Bias–Variance Tradeoff", *Medium*, 2018. [Online]. Available: https://medium.com/30-days-of-machine-learning/day-3-k-nearest-neighbors-and-bias-variance-tradeoff-75f84d515bdb. [Accessed: 26- Mar- 2021].

[10] "Misleading modelling: overfitting, cross-validation, and the bias-variance trade-off", *Cambridge Coding Academy*, 2016. [Online]. Available: https://cambridgecoding.wordpress.com/2016/03/24/misleading-modelling-overfitting-cross-validation-and-the-bias-variance-trade-off/. [Accessed: 24- Mar- 2021].

[11] Wilimits, D. (2018). *The Kernel Trick in Support Vector Classification*. Medium. Retrieved 17 April 2021, from https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f.

[12] Kankatala, S. (2015). Performance Analysis of kNN on large datasets using CUDA & Pthreads comparing between CPU & GPU.

Appendix

Softmax Regression

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time


#Data Import and preprocess ######################################################

train = pd.read_csv('train.csv/train.csv')
train=train.sample(frac = 1)     #shufle
X_train= train.iloc[:,:-2].values  #7351x561
y_train= train.iloc[:, -1].values #7352




test = pd.read_csv('test.csv/test.csv')
test=test.sample(frac = 1)  #shufle
X_test= test.iloc[:,:-2].values #2946x561
y_test= test.iloc[:, -1].values #2946


##################################Onehotencode
classes=['LAYING','SITTING', 'STANDING','WALKING' , 'WALKING_DOWNSTAIRS','WALKING_UPSTAIRS']
y_train_onehot = np.zeros((len(y_train), 6))
y_test_onehot = np.zeros((len(y_test), 6))

for i in range(0, 6):
    for j in range(0, len(y_train)):
        if y_train[j] == classes[i]:
            y_train_onehot[j, i] = 1
        else:
            y_train_onehot[j, i] = 0

for i in range(0, 6):
    for j in range(0, len(y_test)):
        if y_test[j] == classes[i]:
            y_test_onehot[j, i] = 1
            else:
                y_test_onehot[j, i] = 0

y_train=y_train_onehot
y_test=y_test_onehot

##########################################



p=np.size(X_train,1) #feature len
n=np.size(X_train,0)


################################# Feature Scaling
mean=X_train.mean(axis=0)
var=X_train.std(axis=0)
for i in range(np.size(X_train,1)):
     X_train[:,i] = (X_train[:,i] - mean[i]) / var[i]
for i in range(np.size(X_test,1)):
     X_test[:,i] = (X_test[:,i] - mean[i]) / var[i]


######################'######################
##PCA
sample_cov_mat = (1/n)*X_train.T@X_train
eig_vals, eig_vecs = np.linalg.eig(sample_cov_mat)
eig_vals = np.real(eig_vals)
eig_vecs = (np.real(eig_vecs)) #Type conversion
total_variance = (1/n)*(np.linalg.norm(X_train,ord='fro'))**2
#Graph of total variance explained vs k
total_var_explained_wrt_k = np.zeros((len(eig_vals),1))
total_var_explained = 0
project_X_train = X_train@eig_vecs
temp = list()
limit = 95 # %
for kk in range(len(eig_vals)):
    total_var_explained += (1/n)*(project_X_train[:,kk].
       T@project_X_train[:,kk]) / total_variance
    total_var_explained_wrt_k[kk] = (total_var_explained)
    if total_var_explained > limit/100:
        temp.append(kk+1)
how_many_eig_vec_required = temp[0]
plt.plot(np.linspace(1,p,p),100*total_var_explained_wrt_k)
plt.title('Nb. of Principal Components (PC) vs. Total Variance
   Explained (TVE) (%)\n'
        'Nb. of PC where TVE exceeds {0:d}% first time: {1:d}'.
           format(limit,how_many_eig_vec_required))
plt.xlabel('Number of Principal Components')
plt.ylabel('Total Variance Explained (%)')
plt.plot(how_many_eig_vec_required,100*total_var_explained_wrt_k[
   how_many_eig_vec_required],'rx')
plt.plot(np.linspace(1,p,p),limit*np.ones(p),'r--')

u = eig_vecs[:,:how_many_eig_vec_required]
X_train = X_train@u
X_test = X_test@u
p = how_many_eig_vec_required

##########################################

def confusion_matrix(true, pred):
  confusion = np.zeros((6, 6))
  for i in range(len(true)):
    confusion[true[i]][pred[i]] += 1
    confusion.astype(int)
  return confusion

train_acc_list=[]
val_acc_list=[]
past_cost=0
difference_list=[]



def accuracy(X,y,weight):
    output = []
    for l in range(0, 6):
        h = sigmoid(weight[:,l], X)
        output.append(h)
    output=np.array(output)

    predict=np.argmax(output,axis=0)
    true_label=np.argmax(y,axis=1)

    accuracy = 0

    for row in range(len(y)):
        if true_label[row]==predict[row]:
            accuracy += 1

    accuracy = accuracy/len(X)
    return accuracy,predict

def sigmoid(weight,X):
    y = np.dot( weight,X.T)
    return 1 / (1 + np.exp(-y))




def gradient_descent(X, y, weight, learning_rate,decay):
    iteration=0
    past_cost=-10
    while True:
        iteration=iteration+1
        tr_acc=0
        val_acc=0

        cost=0
        for fold in range(0,cv_fold):


            X_val=X[735*fold:735*(fold+1)]
            y_val=y[735*fold:735*(fold+1)]

            X_tr=np.concatenate((X[0:735*(fold)],X[735*(fold+1):])
                ,axis=0)
            y_tr=np.concatenate((y[0:735*(fold)],y[735*(fold+1):])
                ,axis=0)



            for j in range(0, 6):


                h = sigmoid(weight[:,j,fold], X_tr)
                cost=cost+(np.sum(y_tr[:,j]*np.log(h+0.00001) +
                    (1-y_tr[:,j])*np.log(1-h+0.00001))+0.00001)*1/
                    len(X_tr)

                for k in range(0, p):
                    weight[k, j,fold] -= (learning_rate/p)*(1/(1+
                        decay*i)) * (np.sum((h-y_tr[:,j])*X_tr[:, k
```

```
                            ])+ lam*weight[k, j,fold] )



                acc,_=accuracy(X_tr,y_tr,weight[:,:,fold])
                tr_acc=tr_acc+acc
                acc,_=accuracy(X_val,y_val,weight[:,:,fold])
                val_acc=val_acc+ acc
            print('----------cost-----')
            cost=cost/60
            difference= (cost-past_cost)/abs(cost)
            print(difference)
            past_cost=cost
            difference_list.append(difference)
            tr_acc=tr_acc*10  #percent scale
            val_acc=val_acc*10
            train_acc_list.append(tr_acc)
            val_acc_list.append(val_acc)
            print('------------------------------')
            print('Iteration= %d'%iteration)
            print('train accuracy=%f'%(tr_acc))
            print('validation accuracy=%f'%(val_acc))


            if difference <0.00007:
                break
        return weight

cv_fold=10
lam=0.05


######################################################################
    Grid Search
'''
lr_list = [0.002,0.003,0.004,0.005,0.006]
decay_list = [0.1,0.2,0.3,0.4,0.5,0.6]
grid_search_acc = np.zeros((len(lr_list),len(decay_list)))
for lr_idx in range(len(lr_list)):
    for C_idx in range(len(decay_list)):
        learning_rate = lr_list[lr_idx]
        decay = decay_list[C_idx]
        weight = np.zeros([p, 6,cv_fold])
        weight = gradient_descent(X_train, y_train, weight,
            learning_rate ,decay)
        tes_acc,prediction=accuracy(X_test,y_test,weight[:,:,0])
        grid_search_acc[lr_idx,C_idx] =tes_acc
        print("LR_idx: {0}, C_idx: {1}".format(lr_idx, C_idx))


'''
##################################################################


learning_rate=0.04
decay=0.05



weight = np.zeros([p, 6,cv_fold])
weight = gradient_descent(X_train, y_train, weight,learning_rate ,
    decay)
tes_acc,prediction=accuracy(X_test,y_test,weight[:,:,0])



print('test')
print(tes_acc)

true_label=np.argmax(y_test,axis=1)

##############################Confusion Matrix

confusion = confusion_matrix(true_label, prediction)


######################Precision recall F1 score #########

precision = np.zeros(6)
recall = np.zeros(6)
f1_score = np.zeros(6)

for j in range(6):
    precision[j] = confusion[j,j] / sum(confusion[:,j])
    recall[j] = confusion[j,j] / sum(confusion[j,:])
    f1_score[j] = 2*(precision[j]*recall[j])/(precision[j]+recall[
        j])

print('precision:')
print(precision)

print('recall:')
print(recall)

print('f1_score:')
print(f1_score)

################################################################





plt.figure();
plt.title('Iteration vs. Train Accuracy (%) and Validation
    Accuracy (%)')
plt.xlabel("Iteration")
plt.ylabel("Accuracy %")
plt.plot(train_acc_list)
plt.plot(val_acc_list)
plt.legend(['Training Accuracy','Validation Accuracy'])
plt.show()

difference_list=np.array(difference_list)*100

plt.figure();
plt.title('Change in Loss Function % vs Iteration')
plt.xlabel("Iteration")
plt.ylabel("Change in Loss Function % ")
plt.yscale('log')
plt.hlines(y=0.01, xmin=0, xmax=800, linewidth=2, color='r')
plt.plot(difference_list[1:])
plt.legend(['Change in Loss Function %','Stopping Condition'])
plt.show()
```

SVM

```
#%%
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time
import os

#%%
```

```
#Data Import and preprocess
    ###############################################################

train = pd.read_csv('train.csv/train.csv')
train=train.sample(frac = 1)    #shufle
X_train= train.iloc[:,:-2].values  #7351x561
y_train= train.iloc[:, -1].values #7352

test = pd.read_csv('test.csv/test.csv')
test=test.sample(frac = 1)  #shufle
X_test= test.iloc[:,:-2].values #2946x561
y_test= test.iloc[:, -1].values #2946
tr=y_train
tst=y_test


##################################Onehotencode
classes=['LAYING','SITTING', 'STANDING','WALKING' , '
    WALKING_DOWNSTAIRS','WALKING_UPSTAIRS']
y_train_onehot = np.zeros((len(y_train), 6))
y_test_onehot = np.zeros((len(y_test), 6))

for i in range(0, 6):
    for j in range(0, len(y_train)):
        if y_train[j] == classes[i]:
            y_train_onehot[j, i] = 1
        else:
            y_train_onehot[j, i] = 0

for i in range(0, 6):
    for j in range(0, len(y_test)):
        if y_test[j] == classes[i]:
            y_test_onehot[j, i] = 1
        else:
            y_test_onehot[j, i] = 0

y_train=y_train_onehot
y_test=y_test_onehot
del y_train_onehot, y_test_onehot
####################################################

p=np.size(X_train,1) #feature len
n=np.size(X_train,0)
# Feature Scaling

```

```
################################### Feature Scaling
mean=X_train.mean(axis=0)
var=X_train.std(axis=0)
for i in range(np.size(X_train,1)):
     X_train[:,i] = (X_train[:,i] - mean[i]) / var[i]
for i in range(np.size(X_test,1)):
     X_test[:,i] = (X_test[:,i] - mean[i]) / var[i]
###########################################

#######################'#########################
##PCA
usePCA = True
if usePCA:
    sample_cov_mat = (1/n)*X_train.T@X_train
    eig_vals, eig_vecs = np.linalg.eig(sample_cov_mat)
    eig_vals = np.real(eig_vals)
    eig_vecs = (np.real(eig_vecs)) #Type conversion
    total_variance = (1/n)*(np.linalg.norm(X_train,ord='fro'))**2
    #Graph of total variance explained vs k
    total_var_explained_wrt_k = np.zeros((len(eig_vals),1))
    total_var_explained = 0
    project_X_train = X_train@eig_vecs
    temp = list()
    limit = 90 # %
    for kk in range(len(eig_vals)):
        total_var_explained += (1/n)*(project_X_train[:,kk].T@project_X_train[:,kk]) / total_variance
        total_var_explained_wrt_k[kk] = (total_var_explained)
        if total_var_explained > limit/100:
            temp.append(kk+1)
    how_many_eig_vec_required = temp[0]
    plt.plot(np.linspace(1,p,p),100*total_var_explained_wrt_k)
    plt.title('Nb. of Principal Components (PC) vs. Total Variance Explained (TVE) (%)\n'
              'Nb. of PC where TVE exceeds {0:d}% first time: {1:d}'.format(limit,how_many_eig_vec_required))
    plt.xlabel('Number of Principal Components')
    plt.ylabel('Total Variance Explained (%)')
    plt.plot(how_many_eig_vec_required,100*total_var_explained_wrt_k[how_many_eig_vec_required],'rx')
    plt.plot(np.linspace(1,p,p),limit*np.ones(p),'r--')

    u = eig_vecs[:,:how_many_eig_vec_required]
    X_train = X_train@u
    X_test = X_test@u
```

```python
    p = how_many_eig_vec_required
    del project_X_train , eig_vals , eig_vecs , temp

# ####################################################################################################
use_rbf_kernel = False
if use_rbf_kernel:
    gamma = 1/(np.var(X_train)*X_train.shape[1])
    p = newFeatureNb = 250
    new_data_train = np.zeros((n,newFeatureNb))
    new_data_test = np.zeros((X_test.shape[0],newFeatureNb))
    f_idx = 0
    for sample_idx in np.random.choice(range(X_train.shape[0]), newFeatureNb,replace=False):
        print('Sample idx:',sample_idx)
        l = X_train[sample_idx,:]
        distanceSqrSum = np.sum((X_train-l)**2,axis=1)
        new_data_train[:,f_idx] = np.exp(-distanceSqrSum*gamma)
        distanceSqrSum = np.sum((X_test-l)**2,axis=1)
        new_data_test[:,f_idx] = np.exp(-distanceSqrSum*gamma)
        f_idx += 1
    X_train = new_data_train
    X_test= new_data_test
    del new_data_train,new_data_test

feature_scaling_for_rbf = True
if use_rbf_kernel and feature_scaling_for_rbf:
    del mean, var
    mean=X_train.mean(axis=0)
    var=X_train.std(axis=0)
    for i in range(np.size(X_train,1)):
         X_train[:,i] = (X_train[:,i] - mean[i]) / var[i]
    for i in range(np.size(X_test,1)):
         X_test[:,i] = (X_test[:,i] - mean[i]) / var[i]

# ####################################################################################################
def confusion_matrix(true, pred):
  confusion = np.zeros((6, 6))
  for i in range(len(true)):
    confusion[true[i]][pred[i]] += 1
    confusion.astype(int)
  return confusion
```

```

## Hyper Parameters
C = 1000#C for loss of svm
how_many_fold = 10
fold_size = n//how_many_fold
epoch = 100
nb_of_class = y_train.shape[1]
d = 1 #delta is constant chosen as 1, not trying to optimize, C
    will be optimized instead
lr = 1e-3

#weights is defined as (p x nb_of_class)
#Cost Fnc: C * hinge loss + 1/2 ||w||^2 is used
def calculate_loss(weights, C, X, Y):

    nb_of_sample = X.shape[0]

    #Finding scores for each class
    Y_idxed = np.reshape(np.argmax(Y,1),-1)
    scores = X @ weights
    scores_of_true_class = scores[Y>0.5]
    marjin = np.maximum(d + scores - scores_of_true_class[:,np.
        newaxis],0)

    loss_margin = np.sum(marjin) / n
    return (C*loss_margin + 0.5*np.sum(weights*weights))

def calculate_grad(weights, C, X, Y):

    nb_of_sample = X.shape[0]

    #Finding scores for each class
    Y_idxed = np.reshape(np.argmax(Y,1),-1)
    scores = X @ weights
    scores_of_true_class = scores[Y>0.5]
    marjin = np.maximum(d + scores - scores_of_true_class[:,np.
        newaxis],0)

    loss_margin = np.sum(marjin) / n

    #We need to determine where the margin is greaterr than 0
    how_many_greater_than_zero_margin = np.sum(marjin >0,axis=1)
    X_modified_for_grad = (marjin>0).astype(float)
    X_modified_for_grad[Y>0.5] = -
        how_many_greater_than_zero_margin
    grad = C * X.T @ X_modified_for_grad / n + weights
    return grad


def accuracy(X,y,weights):
    estimates = np.argmax(X @ weights,axis=1)
    real = np.argmax(y,axis=1)
    true_labeled = np.sum(real==estimates)
    return true_labeled/X.shape[0]


#%%
# 
    ##########################################################################
#Training with entire train set for final results
initial_time = time.time()
weights = np.random.randn(X_train.shape[1],y_train.shape[1])
for epoch_nb in range(1,1+epoch):
    weights -= lr*calculate_grad(weights, C, X_train, y_train)
elapsed = time.time()-initial_time
print("Elapsed Time: {0:.5f}".format(elapsed))
print(accuracy(X_test,y_test,weights))


#%%
# 
    ##########################################################################
## Cross Validational Training for epoch based graph

train_acc_list=[]
val_acc_list=[]

#Shuffling the data
order_of_data = np.linspace(0,n-1,n,dtype=int)
np.random.shuffle(order_of_data)
shuffled_X_train = X_train[order_of_data,:]
shuffled_y_train = y_train[order_of_data,:]
for fold_nb in range(1,how_many_fold+1):
    train_acc_fold = list()
    val_acc_fold = list()
    #Divide validational fold and training part
    fold_train_x = np.concatenate((shuffled_X_train[:(fold_nb-1)*
       fold_size,:],shuffled_X_train[fold_nb*fold_size:,:]))
    fold_train_y = np.concatenate((shuffled_y_train[:(fold_nb-1)*
        fold_size,:],shuffled_y_train[fold_nb*fold_size:,:]))
    fold_val_x = shuffled_X_train[(fold_nb-1)*fold_size:fold_nb*
        fold_size-1,:]
    fold_val_y = shuffled_y_train[(fold_nb-1)*fold_size:fold_nb*
        fold_size-1,:]
    weights = np.random.randn(fold_val_x.shape[1],fold_val_y.shape
        [1])
    for epoch_nb in range(1,1+epoch):
        weights -= lr*calculate_grad(weights, C, fold_train_x,
            fold_train_y)
        val_acccc = accuracy(fold_val_x,fold_val_y,weights)
        train_acc_fold.append(accuracy(fold_train_x,fold_train_y,
            weights))
        val_acc_fold.append(val_acccc)
        print("Fold {0}, epoch {1} val accuracy: {2:.3f}".format(
            fold_nb, epoch_nb,val_acccc))
    train_acc_list.append(train_acc_fold)
    val_acc_list.append(val_acc_fold)


test_acc = accuracy(X_test,y_test,weights)

print('test')
print(test_acc)

# true_label=np.argmax(y_test,axis=1)
# confusion = confusion_matrix(true_label, prediction)

# plot_confusion_matrix(cm=confusion, target_names=['LAYING','
    SITTING', 'STANDING','WALKING' , 'WALKING_DOWNSTAIRS','
    WALKING_UPSTAIRS' ], title='Confusion Matrix')

plt.figure();
plt.title('Iteration vs. Train Accuracy (%) and Validation
    Accuracy (%)\nTest Set Accuracy = {0:.2f}%'.format(100*test_acc
    ))
plt.xlabel("Iteration")
plt.ylabel("Accuracy (%)")


avg_train_acc = np.zeros((len(train_acc_list[0]),1))
for i in range(len(train_acc_list)):
    avg_train_acc += np.reshape(np.array(train_acc_list[i]),(-1,1)
        )
```

```
avg_train_acc /= len(train_acc_list)

avg_val_acc = np.zeros((len(val_acc_list[0]),1))
for i in range(len(val_acc_list)):
    avg_val_acc += np.reshape(np.array(val_acc_list[i]),(-1,1))
avg_val_acc /= len(val_acc_list)


plt.plot(100*avg_train_acc)
plt.plot(100*avg_val_acc)
plt.legend(['Training Accuracy','Validation Accuracy'])
plt.show()


#
    ##############################################################################

#%%
## Cross Validational Training for Parameter Selection
epoch = 100
train_acc_list=[]
val_acc_list=[]
lr_list = [1e-4,3e-4,1e-3,3e-3,1e-2,3e-2,1e-1]
C_list = [1e1,1e2,1e3,1e4,1e5,1e6,1e7]
all_accs = list()
#Shuffling the data
order_of_data = np.linspace(0,n-1,n,dtype=int)
np.random.shuffle(order_of_data)
shuffled_X_train = X_train[order_of_data,:]
shuffled_y_train = y_train[order_of_data,:]
grid_search_acc = np.zeros((len(lr_list),len(C_list)))
for lr_idx in range(len(lr_list)):
    for C_idx in range(len(C_list)):
        lr = lr_list[lr_idx]
        C = C_list[C_idx]
        val_acc= list()
        for fold_nb in range(1,how_many_fold+1):
            #Divide validational fold and training part
            fold_train_x = np.concatenate((shuffled_X_train[:(
                fold_nb-1)*fold_size,:],shuffled_X_train[fold_nb*
                fold_size:,:]))
            fold_train_y = np.concatenate((shuffled_y_train[:(
                fold_nb-1)*fold_size,:],shuffled_y_train[fold_nb*
                fold_size:,:]))
            fold_val_x = shuffled_X_train[(fold_nb-1)*fold_size:
```

```python
                fold_nb*fold_size -1,:]
            fold_val_y = shuffled_y_train[(fold_nb-1)*fold_size:fold_nb*fold_size -1,:]
            weights = np.random.randn(fold_val_x.shape[1], fold_val_y.shape[1])
            temp_acc = list()
            for epoch_nb in range(1,1+epoch):
                weights -= lr*calculate_grad(weights, C, fold_train_x, fold_train_y)
                temp_acc.append(accuracy(fold_val_x, fold_val_y, weights))
            val_acc.append(accuracy(fold_val_x, fold_val_y, weights))
            all_accs.append(temp_acc)
        grid_search_acc[lr_idx,C_idx] = (sum(val_acc)/len(val_acc))
        print("LR_idx: {0}, C_idx: {1}".format(lr_idx, C_idx))


#Imshow
fig, ax = plt.subplots(1,1)

img = ax.imshow(100*grid_search_acc)

plt.xlabel("C Values")
plt.ylabel("Learning Rates")
plt.title("10 Fold Cross Validational Accuracies (%) with\nDifferent C and Learning Rate Values\nHighest"+
          " Accuracy found with lr = 1e-3, C = 1000")
x_label_list = ["","1e1","1e2","1e3","1e4","1e5","1e6","1e7"]
y_label_list = ["","1e-4","3e-4","1e-3","3e-3","1e-2","3e-2","1e-1"]
ax.set_xticklabels(x_label_list)
ax.set_yticklabels(y_label_list)

fig.colorbar(img)
# lr = 1e-3, C = 1000 selected


##################################################
### Confusion Matrix

prediction = np.argmax(X_test @ weights,axis=1)
true_label=np.argmax(y_test,axis=1)
confusion = confusion_matrix(true_label, prediction)
```

```
########################Precision recall F1 score

precision = np.zeros(6)
recall = np.zeros(6)
f1_score = np.zeros(6)

for j in range(6):
    precision[j] = confusion[j,j] / sum(confusion[:,j])
    recall[j] = confusion[j,j] / sum(confusion[j,:])
    f1_score[j] = 2*(precision[j]*recall[j])/(precision[j]+recall[
       j])

print('precision:')
print(precision)

print('recall:')
print(recall)

print('f1_score:')
print(f1_score)
#####################################################
```

KNN

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import mode
import time




#%%
#Data Import and preprocess
   #####################################################

train = pd.read_csv('train.csv/train.csv')
train=train.sample(frac = 1)    #shufle
X_train= train.iloc[:,:-2].values  #7351x561
y_train= train.iloc[:, -1].values #7352
```

```

test = pd.read_csv('test.csv/test.csv')
test=test.sample(frac = 1)  #shufle
X_test= test.iloc[:,:-2].values #2946x561
y_test= test.iloc[:, -1].values #2946
tr=y_train
tst=y_test

#############################Onehotencode
classes=['LAYING','SITTING', 'STANDING','WALKING' , '
    WALKING_DOWNSTAIRS','WALKING_UPSTAIRS']
y_train_onehot = np.zeros((len(y_train), 6))
y_test_onehot = np.zeros((len(y_test), 6))

for i in range(0, 6):
    for j in range(0, len(y_train)):
        if y_train[j] == classes[i]:
            y_train_onehot[j, i] = 1
        else:
            y_train_onehot[j, i] = 0

for i in range(0, 6):
    for j in range(0, len(y_test)):
        if y_test[j] == classes[i]:
            y_test_onehot[j, i] = 1
        else:
            y_test_onehot[j, i] = 0

y_train=y_train_onehot
y_test=y_test_onehot

##########################################


p=np.size(X_train,1) #feature len
n=np.size(X_train,0)
# Feature Scaling

################################## Feature Scaling
mean=X_train.mean(axis=0)
var=X_train.std(axis=0)
for i in range(np.size(X_train,1)):
     X_train[:,i] = (X_train[:,i] - mean[i]) / var[i]
for i in range(np.size(X_test,1)):
     X_test[:,i] = (X_test[:,i] - mean[i]) / var[i]
```

```python


########################'##########################
##PCA
sample_cov_mat = (1/n)*X_train.T@X_train
eig_vals, eig_vecs = np.linalg.eig(sample_cov_mat)
eig_vals = np.real(eig_vals)
eig_vecs = (np.real(eig_vecs)) #Type conversion
total_variance = (1/n)*(np.linalg.norm(X_train,ord='fro'))**2
#Graph of total variance explained vs k
total_var_explained_wrt_k = np.zeros((len(eig_vals),1))
total_var_explained = 0
project_X_train = X_train@eig_vecs
temp = list()
limit = 90 # %
for kk in range(len(eig_vals)):
    total_var_explained += (1/n)*(project_X_train[:,kk].T@project_X_train[:,kk]) / total_variance
    total_var_explained_wrt_k[kk] = (total_var_explained)
    if total_var_explained > limit/100:
        temp.append(kk+1)
how_many_eig_vec_required = temp[0]
plt.plot(np.linspace(1,p,p),100*total_var_explained_wrt_k)
plt.title('Nb. of Principal Components (PC) vs. Total Variance Explained (TVE) (%)\n'
          'Nb. of PC where TVE exceeds {0:d}% first time: {1:d}'.format(limit,how_many_eig_vec_required))
plt.xlabel('Number of Principal Components')
plt.ylabel('Total Variance Explained (%)')
plt.plot(how_many_eig_vec_required,100*total_var_explained_wrt_k[how_many_eig_vec_required],'rx')
plt.plot(np.linspace(1,p,p),limit*np.ones(p),'r--')

u = eig_vecs[:,:how_many_eig_vec_required]
X_train = X_train@u
X_test = X_test@u
p = how_many_eig_vec_required
############################################

y_test = np.argmax(y_test,axis=1)
y_train = np.argmax(y_train,axis=1)
#Labels are hold as an integer 0 to 6
#
##################################################################################
```

```

def confusion_matrix(true, pred):
  confusion = np.zeros((6, 6))
  for i in range(len(true)):
    confusion[true[i]][pred[i]] += 1
    confusion.astype(int)
  return confusion

## Hyper Parameters
k = 3

def knnRun(X_stored_data,y_stored_data, X_for_prediction, k,
   metric='l2'):
    output = np.zeros((X_for_prediction.shape[0],1))
    for i in range(X_for_prediction.shape[0]):
        data_point = X_for_prediction[i,:]
        distances = X_stored_data-data_point
        if metric == 'l2':
            euc_distances = np.sum(distances**2, axis = 1)
        else:
            euc_distances = np.sum(np.abs(distances), axis = 1)
        output[i] = int(mode(y_stored_data[euc_distances.argsort()
            [0:k]])[0][0])
    return output.T
def accuracy(y_true, y_estimate):
    true_labeled = np.sum(y_true==y_estimate)
    return true_labeled/len(y_true)
#%%
#
    ##############################################################################

## Cross Validational Training for k - parameter selection
val_acc_list=[]
how_many_fold = 10
fold_size = n//how_many_fold
#Shuffling the data
order_of_data = np.linspace(0,n-1,n,dtype=int)
np.random.shuffle(order_of_data)
shuffled_X_train = X_train[order_of_data,:]
ks = np.arange(1,22,2)
shuffled_y_train = y_train[order_of_data]
val_acc_k_selection = list()
for k in ks:
    accs_for_one_k = list()
    initial_time = time.time()
    for fold_nb in range(1,11):
        #Divide validational fold and training part
        fold_train_x = np.concatenate((shuffled_X_train[:(fold_nb
           -1)*fold_size ,:],shuffled_X_train[fold_nb*fold_size
           :,:]))
        fold_train_y = np.concatenate((shuffled_y_train[:(fold_nb
           -1)*fold_size],shuffled_y_train[fold_nb*fold_size:]))
        fold_val_x = shuffled_X_train[(fold_nb-1)*fold_size:
           fold_nb*fold_size-1,:]
        fold_val_y = shuffled_y_train[(fold_nb-1)*fold_size:
           fold_nb*fold_size-1]
        y_estimate = knnRun(fold_train_x, fold_train_y, fold_val_x
           , k)
        accs_for_one_k.append(accuracy(fold_val_y, y_estimate))
        print(k)
        elapsed = time.time()-initial_time
        print("Elapsed Time: {0:.0f}".format(elapsed))
    val_acc_k_selection.append(sum(accs_for_one_k)/len(
       accs_for_one_k))

optimal_k = ks[np.argmax(val_acc_k_selection)]

test_acc = accuracy(y_test,knnRun(X_train, y_train, X_test, k))

plt.figure();
plt.title('L2 | k Parameter vs Validation Accuracy (%)\nTest Set
   Accuracy (with optimal k: {1}) = {0:.2f}%'.format(100*test_acc,
    optimal_k))
plt.xlabel("k parameter")
plt.ylabel("Accuracy (%)")
plt.plot(ks, 100*np.array(val_acc_k_selection))
#%%

#%%
##### Final time
initial_time = time.time()
prediction_test_y= np.asarray(knnRun(X_train, y_train, X_test, 9),
   int)
elapsed = time.time()-initial_time
print("Elapsed Time: {0:.0f}".format(elapsed))
#####################################################
### Confusion Matrix
true_label = y_test
confusion = confusion_matrix(true_label, prediction_test_y.T)
```

```
########################Precision recall F1 score

precision = np.zeros(6)
recall = np.zeros(6)
f1_score = np.zeros(6)

for j in range(6):
    precision[j] = confusion[j,j] / sum(confusion[:,j])
    recall[j] = confusion[j,j] / sum(confusion[j,:])
    f1_score[j] = 2*(precision[j]*recall[j])/(precision[j]+recall[
       j])

print('precision:')
print(precision)

print('recall:')
print(recall)

print('f1_score:')
print(f1_score)
#####################################################
```