Machine Learning Engineer Nanodegree

Capstone Project

To find an Optimum Machine Learning Model for Human Activity Recognition Using Smartphones Sensor Data

Arunabh Saikia
November 10, 2017

1. Definition

1.1 Project Overview

Nowadays smartphones have become an inseparable part of daily human life. May the purpose be for communicating with others, playing games, constantly be on social network or getting news updates about what is going on around the globe. But still the all the possible uses of smartphone are not exhausted. One major study regarding smartphone is how to understand human activities using smartphones and use it for health monitoring. Nowadays all the smartphones are equipped with built in sensors such as gyroscope, GPS, accelerometer, compass, barometer etc. Smartphone-based activity tracking is an active area of research because they can lead to new types of mobile applications. Applications that can record the user's regular activities and reports when there is any anomaly. It can also help in elder care support and rehabilitation assistance, thus saving huge number of resources that are being used currently.

The problem was discussed by **Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra** and **Jorge L. Reyes-Ortiz** in their paper titled "*Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine*" published in *2012*, which uses a support vector machine classifier to tackle the problem. Also, a number of other open source analysis has been performed using various other machine learning algorithms. The dataset we are considering here is an open source dataset from UCI Machine Learning Repository and can be found under Kaggle's datasets section (Link: Kaggle: *https://www.kaggle.com/uciml/ human-activity-recognition-with-smartphones*)

1.2 Problem Statement

The problem in question is a multi-class Classification problem. The input data consists 3-axial linear acceleration and 3-axial angular velocity readings from the gyroscope of a Samsung Galaxy S II at a constant rate of 50Hz accelerometer. The input data consists of 561 which we need to analyze and our goal is to predict if a user is doing any of the following activities: WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING from their smartphone activity.

I will be approaching this as any other multi-class classification problem and after analyzing the dataset and doing all the needful pre-processing on the dataset. I will apply Naive Bayes Algorithm to do an initial classification to check how the algorithm works on the dataset. After that I plan to use a Logistic Regression Classifier, a Support Vector Classifier followed by an ensemble using LightGBM and finally a Keras Neural Network to check which model performs best for the problem at hand. The solution will be classifying a user's activity to any of the following six activities: (WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING or LAYING based on the input feature vector from his smartphone's sensor readings.

My approach is to find an optimal model which can solve the task in hand accurately and quickly. So, I will be using few classification algorithms on the dataset to see which one performs better and then use the one that provides the best output and work on that model further to fine tune its parameters to reach an optimum model.

My first step after importing the dataset would be a quick visualization of how the records are distributed against the Activity labels. This will allow us to visualize if the distribution of the dataset based on activities is balanced or some activities are preferred over the others. The next step I would consider would be pre-processing the dataset before feeding it into our algorithms. In our case, the dataset doesn't contain any missing values. Also, the dataset is normalized in the range [-1, 1]. So, I would look for any feature that is not necessary for predicting the activity and I will remove those form the dataset. Then I will store the labels separately by encoding the activities to numerical categories by using Label Encoder.

Once the pre-processing is done, I will fit a Naïve Bayes model to check the prediction score. After that I will iterate through the following models to find the one that provides the highest prediction accuracy and then work on that to improve its performance.

- Logistic Regression.
- SVC.
- LightGBM.
- Keras Neural Network.

1.3 Metrics

To measure the model performance, it is very important to choose the right evaluation metrics for the problem. As our problem is a classification one, we will consider our model is "Right" when it correctly predicts the class/label of a test instance, and "wrong" when it incorrectly predicts a class/label. To measure this, there are a variety of accuracy metrics that implement this idea in one way or another, such as precision, recall, F1, AUC scores. For our problem, we will use F1 Score as our evaluation metrics. F1 takes into consideration both the precision and the recall of the test to compute the score. Precision is the number of correct positive results divided by the number of all positive results, and Recall is the number of correct positive results divided by the number of positive results that should have been returned. The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. Another accuracy metric we are going to use is Logarithmic Loss which measures the probability for each class, rather than just the most likely class. Log Loss heavily penalizes classifiers that are confident about an incorrect classification. Minimizing the Log Loss is basically equivalent to maximizing the prediction confidence of the classifier.

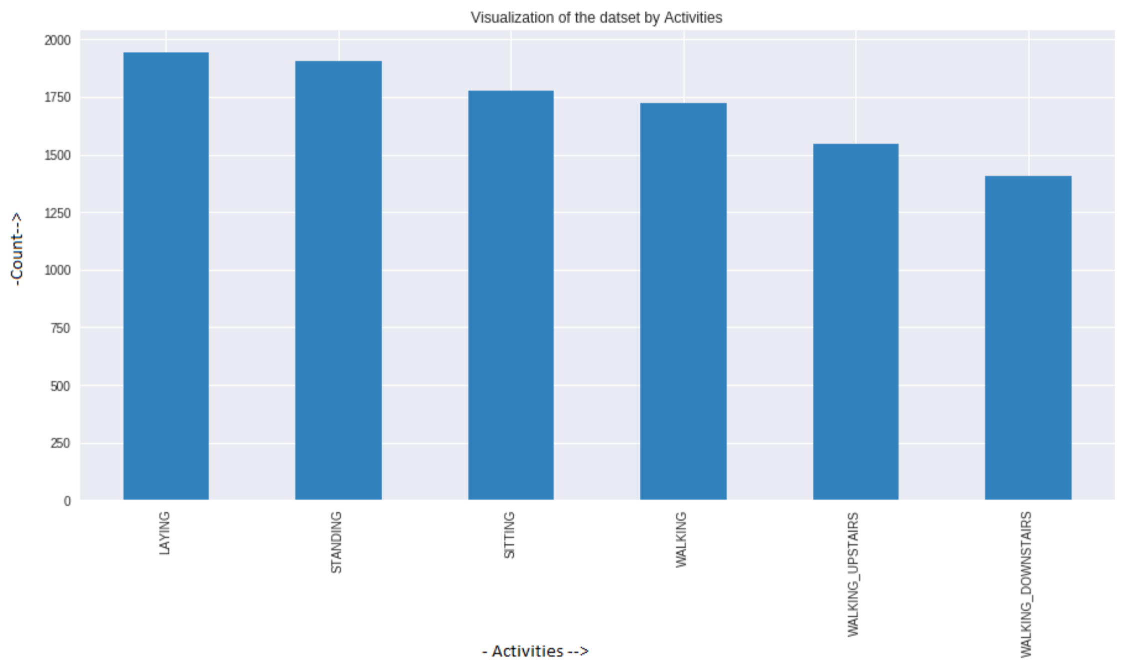


Figure 1

From the above visualization, we can see that our dataset is quite evenly distributed. There are no major fluctuations among count of the activities. So, we could have just taken accuracy as our evaluation metric instead of F1 and log-loss function. However, even though the dataset is fairly distributed among the classes, it is not equally distributed. Therefore, taking a weighted average of precision and recall which is F1 score rather than simply considering accuracy is a better metric for our model. Log loss on the other hand will help improve the confidence of our model for a classification. e.g. If for a binary classification, a model predicts probability of a class A as 51% and that of B as 49%, the model will choose class A. But in this case the log loss is very high. Therefore, even if the accuracy of the model is high, it is not very confident about its prediction. However, if the model predicts probability of a class A is 99% and that of B is 01%, the prediction results will be same as earlier, but this time the model will be more confident about its prediction.

2. Analysis

2.1 Data Exploration

The dataset I am going to use is an open source dataset which can be found in UCI Machine Learning repository as well as under Datasets section in Kaggle. (Link*: https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones/data*).

The data set was collected from an experiment which was carried out with a group of 30 volunteers within an age bracket of 19-48 years. They performed a series of activities standing, sitting, lying, walking, walking downstairs and walking upstairs. The experiment also included postural transitions that occurred between the static postures which are: stand-to-sit, sit-to-stand, sit-to-lie, lie-to-sit, stand-to-lie, and lie-to-stand. All the participants were wearing a smartphone (Samsung Galaxy S II) on the waist during the experiment execution. Data was captured with 3-axial linear acceleration and 3-axial angular

velocity at a constant rate of 50Hz using the embedded accelerometer and gyroscope of the device. The experiments were video-recorded to label the data manually. The obtained dataset was randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data.
(Reference: *http://archive.ics.uci.edu/ml/datasets/Smartphone-Based+Recognition+of+Human+Activities+and+Postural+Transitions*).
The data-set consists of the following features:

- A 561-feature vector with time and frequency domain variables.
- Its associated activity labels (WALKING, WALKING_UPSTAIRS, WALKING_ DOWNSTAIRS, SITTING, STANDING or LAYING).
- An identifier of the subject who carried out the experiment.

The Activity feature of the dataset is categorical as well as the subject feature. All the other features of the dataset are normalized and bounded within [-1,1]. The dataset has no missing values. We will remove the subject feature from our dataset and use all the remaining features to train a model and predict the activity a user is performing.

2.2 Exploratory Visualization

We know that the dataset we are using is normalized and bounded within [1, -1]. Also from *Figure-1* displayed earlier we can see our dataset is quite evenly distributed among all six activities even though it is not equally divided. Now let us try to extract more information about our dataset by exploring its different features visually. [Implementation]

Mean Body Acceleration in X-Direction and Y- Direction vs Activities: Let us use seaborn Strip-plot to see if mean body acceleration in X-Direction has any distinct relationship with any of the activities. We will be randomly selecting activities recorded by subject 10 for the visualizations.

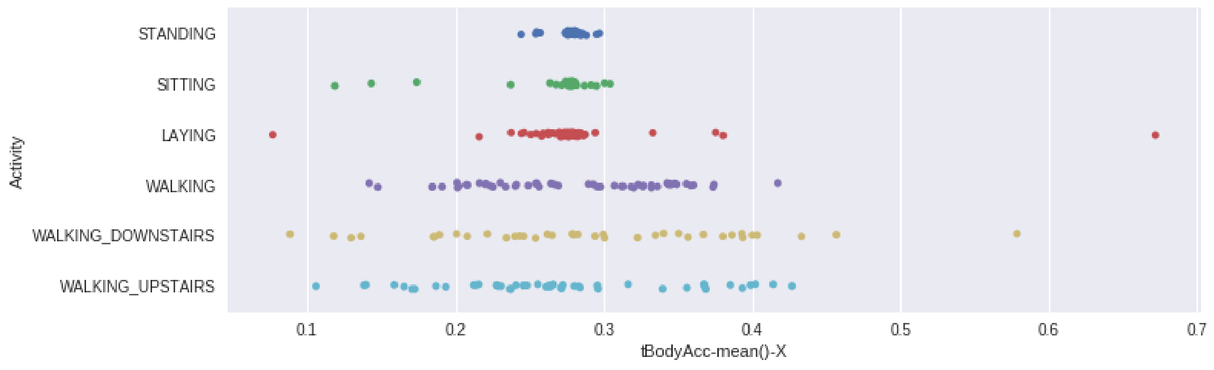


Figure 2.2 (a)

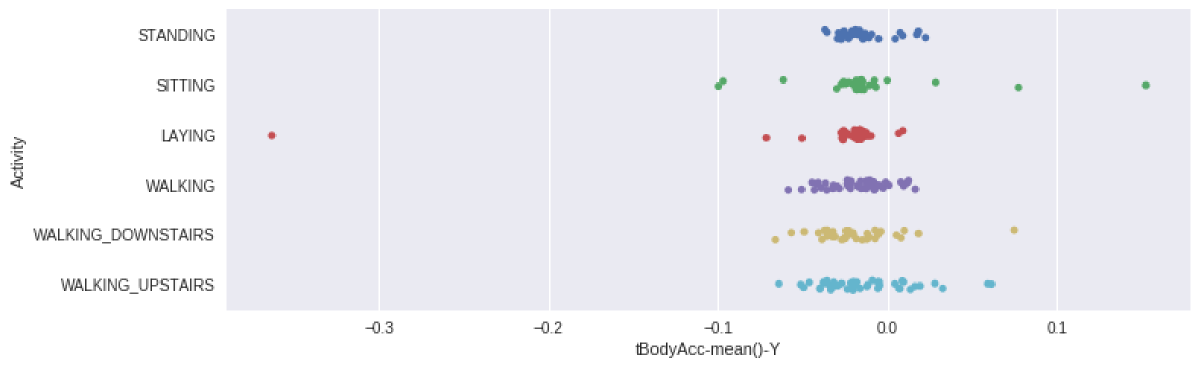


Figure 2.2(b)

From the above visualizations [Figure 2.2(a) and Figure 2.2(b)], we can see that the body acceleration in X-direction varies more than that in Y-direction. Also, if we observe closely, for the active activities: WALKING, WALKING_UPSTAIRS and WALKING_DOWNSTAIRS the variation in X-axis is more compared the passive activities: STANDING, SITTING and LAYING.

Now let us plot a dendrogram using seaborn to check if we can find any structure of Mean Body acceleration in X, Y and Z direction with Activities recorder by Subject-10.

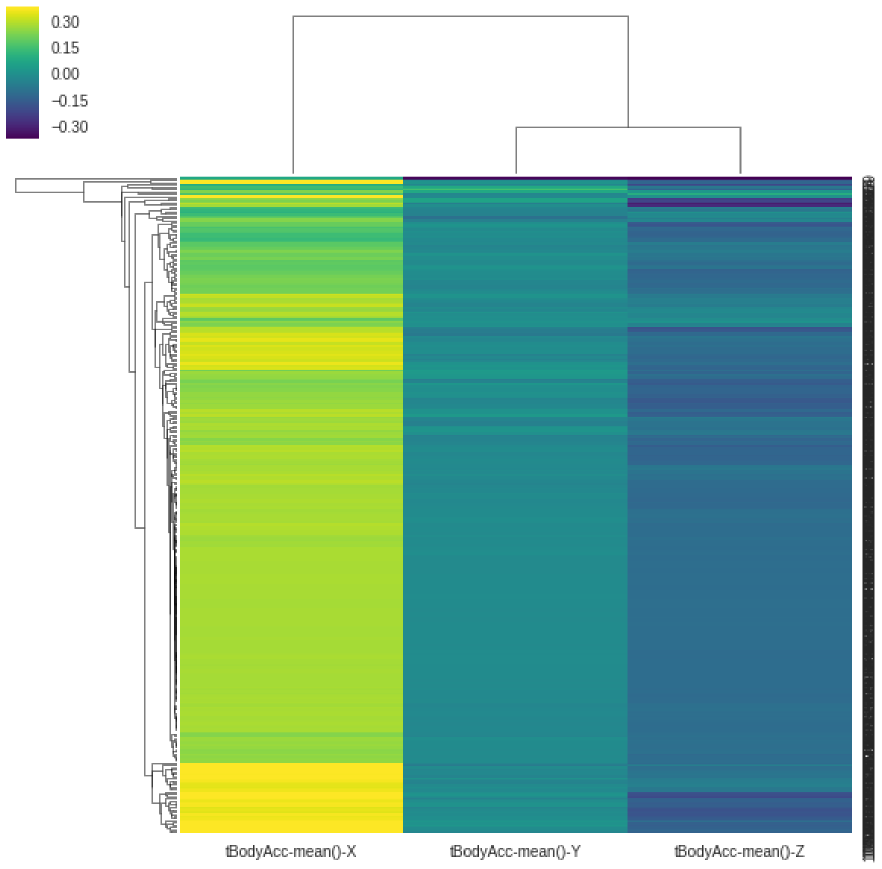
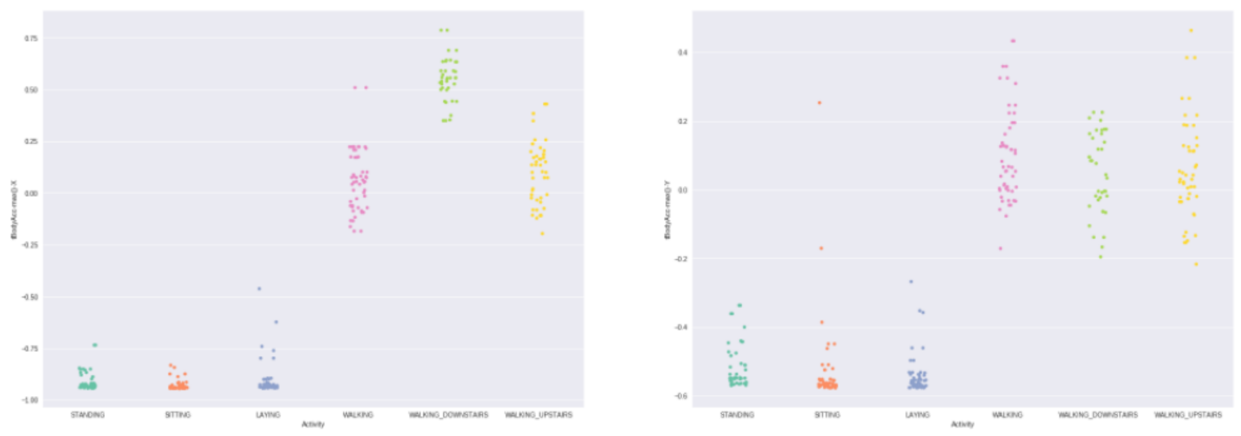


Figure 2.2 (c)

In the above visualization [Figure 2.2(c)], we can see that most of the plot is homogeneous. In X and Z direction, only a few spots of the map are dark, Y being constantly homogeneous all the time throughout. If we recall from our previous plot, for the active activities mean body acceleration in X has a high variation and the few dark spots here might be due to that. From our mean body acceleration plot we cannot conclude anything strongly.

Let's take another feature to check if we can find any structure.

Max Body Acceleration in X- Y-Z Direction vs Activities: By using seaborn Strip Plot again for the visualizations for MaxBodyAcceleration Vs Activities we get the below plots.



<u>Figure 2.2(d)</u>

Comparing the above Strip-plots, from *Figure 2.2(d)*, we can clearly see that for Max Acceleration in both X and Y direction, the Passive activities are way below the active ones and for the active one the variation is also huge. Which makes sense, because acceleration in X and Y directions should be there when any active activities are performed. If we notice in the first of the two plots, WALKING_DOWNSTAIRS has a greater value for Max body accleration. This can be explained as when a person walks down stairs, his body moves comparatively faster compared to when he is steadily walking or Walking_Upstairs.

Now, let's plot the cluster map for maximum body acceleration with Activities.

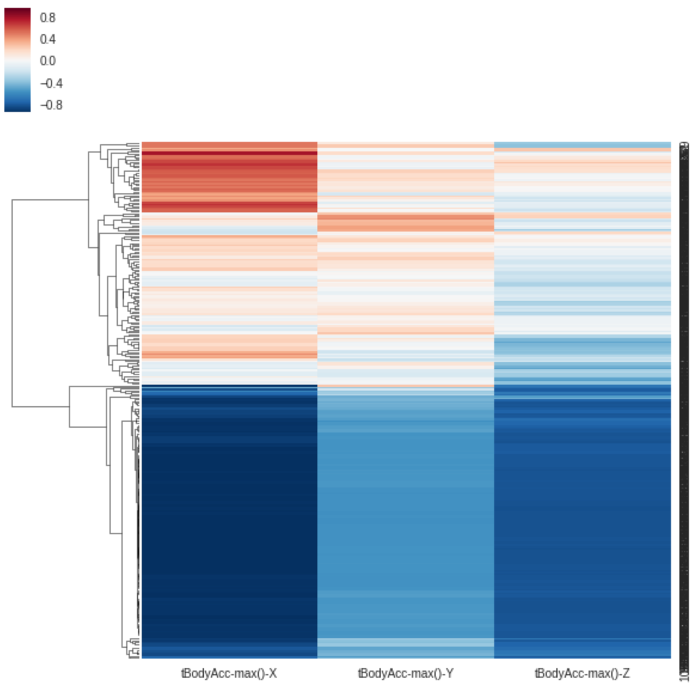


Figure 2.2(e)

From *Figure 2.2(e),* we can now see the difference in the distribution between the active and passive activities with the walking down activity clearly distinct from all others especially in the X-direction. The passive activities are indistinguishable and present no clear pattern in any direction (X, Y, Z). So, we can say that Max Body Accleration in X direction has a direct influence on the Active Activities.

From our visualization of the two features *MeanBodyAcc* and *MaxBodyAcc* with in its all three directions X, Y and Z against Activities, below are our findings:

- Max Body Accleration in X-direction is greatly affected by all three WALKING activities.
- The Passive activities are not affected by Max Body Accleration in any-direction.

2.3 <u>Algorithms and Techniques:</u>

Before pre-processing our dataset, I would first fit a Naïve Bayes model to see how much accuracy we can obtain without any pre-processing and how much runtime is required so that after pre-processing we can compare the results and see if the pre-processing has helped us in increasing the accuracy or in reducing the model train time or not.

For the pre-processing step, as our dataset is already normalized and scaled in the range [1, -1], we can move on to feature selection step. We can see that the 'subject' column which is the subject who has taken reading for this dataset has no real effect on the output activity as all the subjects has performed the same activities and predicting an activity is not dependent on the subject in any way, we can discard that feature. Before starting to work on the model, I will first do some visualization on the dataset to see which features are more important and strongly related. After removing 'Subject' and storing 'Activity' separately, we will be left with 561 features, and I plan to use all of these features while training and predicting with our model.

The main goal of this project is to find the model which best performs with our problem and for that the Machine Learning algorithms we considering are:

- SVC
- Logistic Regression.
- LightGBM.
- Keras Neural Network.

SVC or Support Classifier is a supervised learning algorithm. I will use the Radial Basis Function(RBF) as the kernel for our classification problem. For RBF kernel, SVC has two parameters that needs tuning, gamma and C. Gamma can be thought of as the 'spread' of the decision region. When gamma is low, the 'curve' of the decision boundary is very low and thus the decision region is very broad. When gamma is high, the 'curve' of the decision boundary is high, which creates islands of decision-boundaries around data points. Whereas, the C parameter is the penalty for misclassifying a data point. When C is small, the classifier is okay with misclassified data points (high bias, low variance). When C is large, the classifier is heavily penalized for misclassified data and therefore bends over backwards avoid any misclassified data points (low bias, high variance). Tuning gamma and C by using "Grid Search CV" method we can improve the model accuracy. As our problem is a classification problem we are using SVC for the task.

Logistic regression is basically a regression method with the exception that it provides probability output of each class rather than actual predicted class. Classification using logistic regression is a supervised learning method, and therefore requires a labeled dataset. Our problem is a multiclass classification problem. Here, I will use the "one-vs-all binary logistic regression" approach and then fine tune its parameters to attain maximum F1 and lowest Log Loss for our problem. A one-vs-all binary logistic regression works on the assumption that for **N** output classes there are **N** independent classification, one for each class.

LightGBM is a fast gradient boosting framework based on decision tree algorithm, used for classification problems. Light GBM grows the tree vertically (leaf wise) while other algorithms grow trees horizontally (level wise). I am choosing LGBM over XGBoost for its higher speed. Even though basic implementation of LGBM is easy, it has a huge number of parameters which needs tuning. I will try to tune the parameters using Grid Search CV to obtain the highest accuracy for our model. For our classification task we will be using LGBMClassifier from LightGBM.

Keras Neural Net is a high-level neural networks API with fast prototyping which supports both convolutional networks and recurrent networks, as well as combinations of the two. We will use the simplest type of keras model which is the Sequential model, a linear stack of layers.

The Neural Network I am going to consider is a Keras Model for Multiclass Classification. Our sequential model will have a dense and an activation layer. Below is the basic architecture of our model. The very first layer of our model is a Dense layer with 64 Filters and Activation as relu. As this is the first layer, input shape is also provided using 'input_shape = 561'. Then we add a dropout layer using the ".add" method. A dropout layer prevents overfitting. The Dense(64) is a fully connected layer with 64 hidden units. The output layer is a dense layer with 6 nodes which corresponds to our 6 class outputs. Activation used in this layer is 'softmax' which will give the probability of each class. The optimizer I will use is a stochastic Gradient Descent Optimizer (SGD) with Categorical Cross-entropy as loss function and Accuracy as evaluation metric.

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=561))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(6, activation='softmax'))
```

After the initial prediction with the base model, I will use checkpoint and early stopping to find an optimum epoch and batch size which gives maximum accuracy for our model.

Finally, I will compare the performance between these models and take the one with the better F1 Score and low Log Loss value with shortest runtime. This will be our optimum model.

2.4 Benchmark

We already have many solutions for our problem. The model we will be considering as a benchmark is a SVC which obtains an Accuracy: 96.34 and Precision: 96.44.

(Link: *https://github.com/gornes/Human_Activity_Recognition*). I will try to exceed this score using the above-mentioned machine learning models.

3. Methodology

3.1 Data Preprocessing

Data Preprocessing is a technique that is used to convert raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not directly feasible for the analysis. Therefore, certain steps are executed to convert the data into a small clean data set. It includes Data Cleaning, Data Integration, Data Transformation and Data Reduction.

Before feeding the data into our machine learning algorithms, we should preprocess the data. For this we will perform the following steps:

- Removing any Missing or NA values from the dataset.
- Rescaling the dataset
- Normalizing the dataset

However, the dataset we are using does not contain any missing values. Also, for the dataset features are already Features are normalized and bounded within [-1,1]. So, the above-mentioned steps are not required in our case.

Now let us see if we can get rid of any feature from our dataset which is not relevant to our dataset. In other words, any feature, removing which will not affect performance of our model on the dataset, rather will work more effectively due to decrease in the size of the dataset and in turn improving its performance.

Out of all 563 features of our dataset, *Activity* and *Subject* are categorical and the other 561 features are continuous. Just by looking at the dataset we feel that *'Subject'* is not a relevant feature for predicting the *Activity*. *Subjects* are the people who have undertaken the experiment for capturing our dataset and each one of 30 subjects have performed the same six activities. Let's visualize the distribution of activity performed by the subjects for all 6 activities to see if there exists a relationship between *subject* feature and the *activity* feature which might affect our model performance if we remove the *Subject* feature.

Let us use *Matplotlib* to plot total number of activities performed by each subject. From *Figure 3.1(a)*, we can see that total no of activities performed by the subjects vary slightly and there is no distinct relationship between these two features. [Implementation]

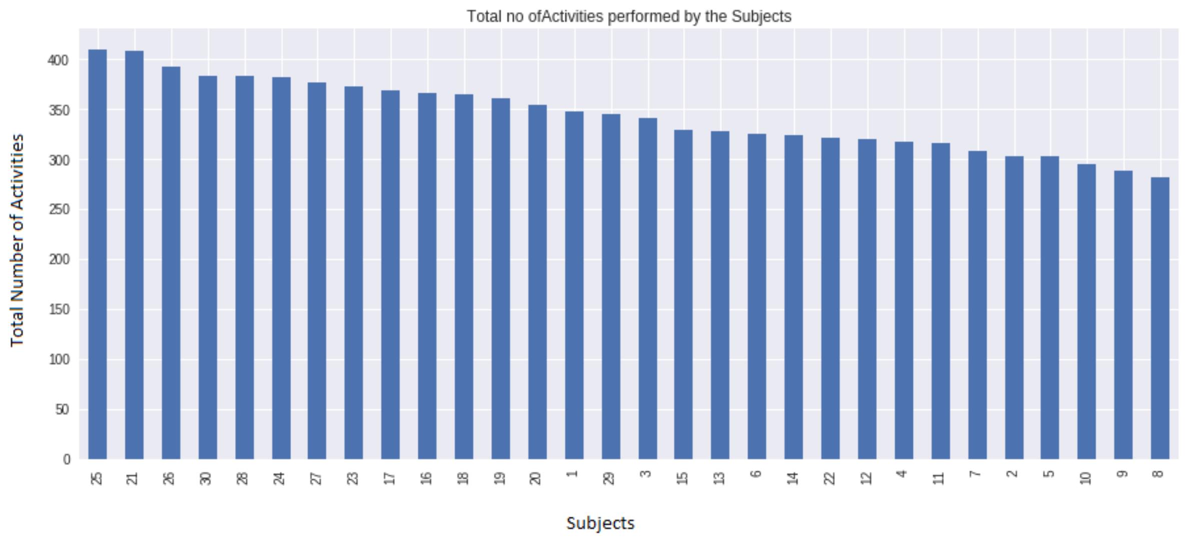


Figure 3.1(a)

Now, let us dig deeper by using Seaborn count plot to visualize individual activities performed by the subjects. We are plotting Subject on X -axis vs Activities on the Y-axis.

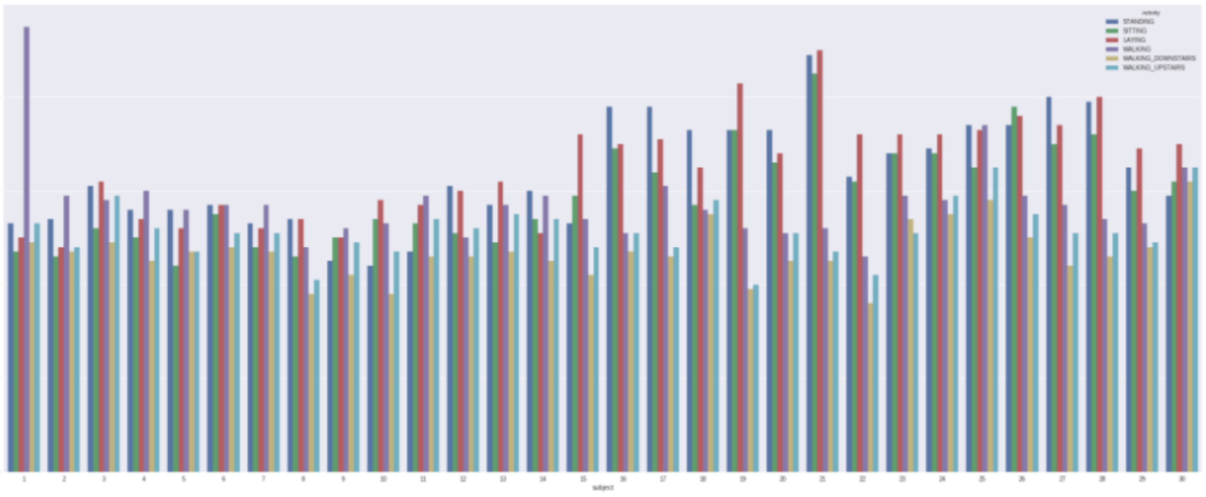


Figure 3.1(b)

As we can see above, from Figure 3.1(b) activities performed by the subjects are similar. Except for a few cases (e.g. subject 1), LAYING being the maximum no of activity performed by all users. We can conclude that there is no direct relationship which will help us determine an activity just by looking at the Subject. So, we can remove the 'Subject' feature from our dataset.

Now we are left with 561 features and we will be using all these features to evaluate our models. From our dataset, we can see that the 'Activity' feature is non-numeric. ML algorithms expect the input data to be numeric, so we shall convert the 'Activity' feature into something Numerical. Let us use 'Label Encoder' for this purpose. Label Encoder will encode each of the 6 Activities to numerical categories in the range (0, 5). We will store encoded Activity as ‘target’.

Finally, as our test data is now preprocessed and the target is separated and encoded as numerical categories. Let us proceed with splitting the dataset as training and testing set using ‘train_test_split’ taking ‘test_size as’ 30% of the total dataset and considering a ‘random_state’ for reproducible results. Our dataset is split as below:

- Training set has 7209 samples.
- Testing set has 3090 samples.

One thing to keep in mind that, log loss can work only when the target variable is binary. But in our case the target class *‘Activities’* has numerical categorical values [0, 1, 2, 3, 4, 5]. We will use One Hot Encoding to the convert the numerical categorical values into binary categorical values.

3.2 Implementation

Now our training and testing data are ready to be fed into out ML algorithms. First, we will use the Naive Bayes predictor to see how it is performing against the unprocessed and our processed dataset.

For pre-processing data-set, we have dropped one feature ‘Subject’ as shown:

```
# Drop the column 'subject'
data = data.drop(['subject'], axis = 1)   # No of column decreases by 1 (562 Columns)
label = data['Activity']
data = data.drop(['Activity'], axis = 1)  # 561 Columns
```

Also, the subject feature is categorically encoded using label encoder.

```
# Use Label Encoder and transform 'Activity' into numerical catagories
LE = LabelEncoder()
target = LE.fit_transform(label)
```

For both the unprocessed and processed models, a NB classifier is defined, fit and then used to evaluate performance on the test dataset. Below is the code showing the implementation

```
# Define a classifier and use it to fit and predict the data
clf_nb= GaussianNB()
start = time.clock() # Store the start time
clf_nb.fit(X_train, y_train) # Fit the NB classifier
pred_nb = clf_nb.predict(X_test) # Use the fitted NB classifier to predict on the Test set
end = time.clock() # Store the end execution time for the model
runtime_raw = end-start  # Save the runtime
raw_acc = accuracy_score(y_test, pred_nb) # store the model aacuracy score as raw_accuracy
print('Accuracy Score for NB on the unprocessed dataset is: {}'.format(raw_acc)) #Print out accuracy score
print('Runtime required: {}'.format(runtime_raw))  # Print out runtime
```

Model	Accuracy Score	Runtime
Naïve Bayes with unprocessed data	0.690614886731	0.193971
Naïve Bayes with processed data	0.690614886731	0.167511

As we can see from our above comparison, accuracy score in case of both processed and unprocessed is same. However, for the same accuracy, the raw Naive Bayes model is taking more runtime than the model with the pre-processed dataset. So, for all the further models, we will consider the preprocessed dataset with 561 Features for implementation.

Now, let us use the various models that we selected on the dataset and see which one performs the best.

Support Vector Classifier (SVC): [Implementation]

"Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features we have) with the value of each feature being the value of a coordinate. Then, we perform classification by finding the hyper-plane that differentiates the classes well. Support Vectors are simply the co-ordinates of individual observation. Support Vector Machine (in our case the SVC we are selecting) is a frontier which best segregates the two classes. Below is a simple representation of a binary SVC classifier [*Figure 3.2 (a)*].

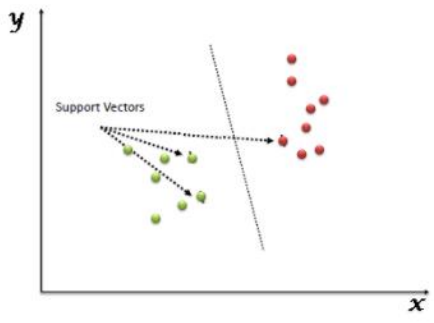


Figure 3.2(a)

We are using the Radial Basis function as kernel for our classifier. The (Gaussian) radial basis function kernel, or RBF kernel on two samples x and x', represented as feature vectors in some input space, is defined as:

$$\mathbf{K(X, X') = exp(-\ \frac{-|X-X'|^2}{2\,\sigma^2})}$$

$|\boldsymbol{X}-\boldsymbol{X'}|$ is the squared Euclidean distance between the two feature vectors. Sigma(σ) is a free parameter. An equivalent, but simpler, definition involves a parameter $\gamma = \frac{1}{2\,\sigma^2}$

Since the value of the RBF kernel decreases with distance and ranges between zero (in the limit) and one (when x = x'), it has a ready interpretation as a similarity measure (*similarity function* is a real-valued function that quantifies the similarity between two objects). The feature space of the kernel has an infinite number of dimensions. Fitting a SVC with its default parameters, our model performs exceptionally well in terms of F1 score and Log Loss Score.

For implementing SVC, our initial approach is to work with a default SVC and then optimize its two parameters so that to achieve higher performance. Initially we have imported the model from sklearn and then used our processed dataset to fit the model and then used it to predict on the test dataset. The model uses default values of C and gamma. Our unoptimized SVC has the following results:

Model: Unoptimized SVC	
Accuracy Score	0.944336569579
F1 Score	0.944145032047
Log Loss Score	0.124973620049
Runtime	57.966881

Once we are set with the basic model, we have used Grid Search to optimize its two parameters C and Gamma in iterations. First, we have taken a large logarithmic grid which is the initial coarse search for the parameters. This step helps us to find an approximate value for the parameters. Below we can see the code for doing initial coarse Grid Search involving C and Gamma.

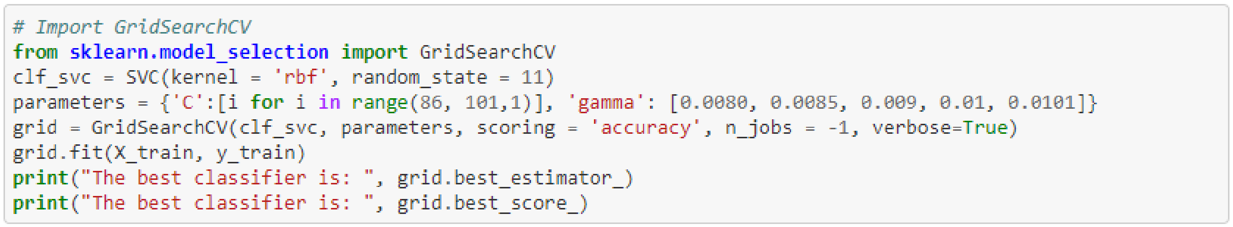
```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
clf_svc = SVC(kernel = 'rbf')
parameters = {'C':10.**np.arange(-3, 8), 'gamma': 10.**np.arange(-5, 4)}
grid = GridSearchCV(clf_svc, parameters, scoring = 'accuracy', n_jobs = -1, verbose=True)
```

```
grid.fit(X_train, y_train)
print("The best classifier is: ", grid.best_estimator_)
```

The above blocks of code do the following:

a. Imports GridSearchCV from sklearn
b. Defines a dictionary for logarithmic ranges of C and gamma.
c. Defines the Grid passing the SVC classifier, the parameters dictionary and a scoring function (accuracy) with n_jobs = -1 for parallel processing. Verbose is set to true so that we can reproduce the results later.
d. In the final step, we fit the Grid Search which iterates through every combination of the two parameters and provides us with the best set for this range.

After the initial search, which gives C = 100 and gamma = 0.01, we check the model's performance using these values against the test data set and further move on with fine tuning the parameters using a different range around the output of first Grid Search.



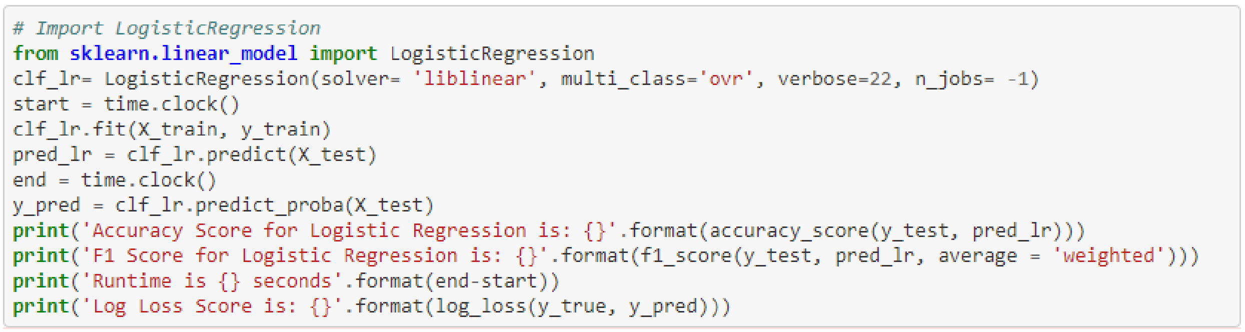
```python
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV
clf_svc = SVC(kernel = 'rbf', random_state = 11)
parameters = {'C':[i for i in range(86, 101,1)], 'gamma': [0.0080, 0.0085, 0.009, 0.01, 0.0101]}
grid = GridSearchCV(clf_svc, parameters, scoring = 'accuracy', n_jobs = -1, verbose=True)
grid.fit(X_train, y_train)
print("The best classifier is: ", grid.best_estimator_)
print("The best classifier is: ", grid.best_score_)
```

From the first parameter search, C gave a value of 100 and while searching in range [... 1, 10, 100, 1000, ...], so we know that our C is within the range (10 < C < 100). Therefore, this time we choose a range more around 100. We have chosen a small range, this is because of processing runtime. Our aim is to if this search gives a C value which tends to move towards 86(our lowest boundary value), we will consider the next grid towards that value and vice versa. Same method is applied for gamma.

Logistic Regression Classifier: [Implementation]

Here, we are using the "one-vs-all binary logistic regression" approach. A one-vs-all binary logistic regression works on the assumption that for **N** output classes there are **N** independent classification, one for each class. The model will select the class as most likely which has the highest probability. The unoptimized model uses *solver* = *'liblinear'* and *multi_class* = *'ovr'*. Setting multi_class parameter to 'ovr' allows us to implement 'one vs all' binary option. The default solver is selected which is 'liblinear'. We will use gridsearch to optimize these parameters in the Refinement step. The steps involving implementation of the basic model are as below:

a. Importing the model form sklearn.model_selection module
b. Defining the classifier. The solver is set to default value and multi class to 'ovr' as this is a multi-class classification problem.
c. The classifier is then fit on the training dataset and predicted on the test dataset.
d. The evaluation metrics are then calculated to check the model performance. Also predict_proba function is used to predict the probabilities rather than classes as log loss only works with probabilistic values.

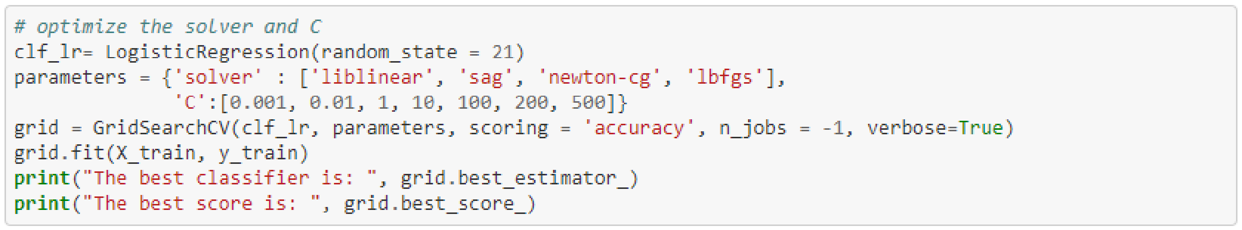


```python
# Import LogisticRegression
from sklearn.linear_model import LogisticRegression
clf_lr= LogisticRegression(solver= 'liblinear', multi_class='ovr', verbose=22, n_jobs= -1)
start = time.clock()
clf_lr.fit(X_train, y_train)
pred_lr = clf_lr.predict(X_test)
end = time.clock()
y_pred = clf_lr.predict_proba(X_test)
print('Accuracy Score for Logistic Regression is: {}'.format(accuracy_score(y_test, pred_lr)))
print('F1 Score for Logistic Regression is: {}'.format(f1_score(y_test, pred_lr, average = 'weighted')))
print('Runtime is {} seconds'.format(end-start))
print('Log Loss Score is: {}'.format(log_loss(y_true, y_pred)))
```

Below is the performance of our basic model.

Model: Unoptimized Logistic Regression	
Accuracy Score	0.981877022654
F1 Score	0.981834941184
Log Loss Score	0.0623213805757
Runtime	6.374008

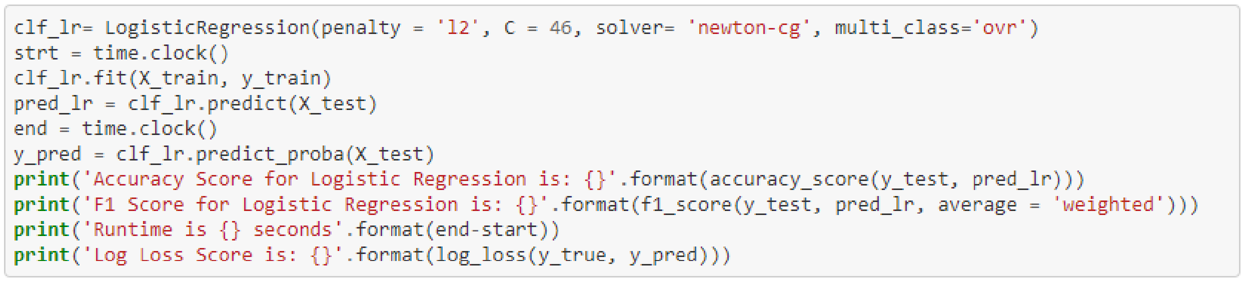
Approach to tune the parameters are same as implemented in case of the above SVC model, the only difference being Logistic Regression has more parameters that needs tuning. For parameters, we have decided to tune solver followed by C as seen in the below code block.



```python
# optimize the solver and C
clf_lr= LogisticRegression(random_state = 21)
parameters = {'solver' : ['liblinear', 'sag', 'newton-cg', 'lbfgs'],
              'C':[0.001, 0.01, 1, 10, 100, 200, 500]}
grid = GridSearchCV(clf_lr, parameters, scoring = 'accuracy', n_jobs = -1, verbose=True)
grid.fit(X_train, y_train)
print("The best classifier is: ", grid.best_estimator_)
print("The best score is: ", grid.best_score_)
```

This Grid Search gives us *newton-cg* to be best for our problem. Hence, we move on with tuning C to the finest value. Using Grid Search same as above gives us our optimum parameter values and we use that to evaluate our model on the test set. The final model is fit as below

a. The classifier is defined passing the optimum parameters.
b. The classifier is fit on the training data.
c. Prediction is made on the test set and model is evaluated against F1and Log loss metrics.



```python
clf_lr= LogisticRegression(penalty = 'l2', C = 46, solver= 'newton-cg', multi_class='ovr')
strt = time.clock()
clf_lr.fit(X_train, y_train)
pred_lr = clf_lr.predict(X_test)
end = time.clock()
y_pred = clf_lr.predict_proba(X_test)
print('Accuracy Score for Logistic Regression is: {}'.format(accuracy_score(y_test, pred_lr)))
print('F1 Score for Logistic Regression is: {}'.format(f1_score(y_test, pred_lr, average = 'weighted')))
print('Runtime is {} seconds'.format(end-start))
print('Log Loss Score is: {}'.format(log_loss(y_true, y_pred)))
```

LightGBM Classifier: [Implementation]

LightGBM is a fast gradient boosting framework based on decision tree algorithm, works well for classification problems. The default model uses an out of box LGBM classifier with the default parameter values, which gives the below results for its initial fit.

Model: Unoptimized LightGBM Classifier	
Accuracy Score	0.96957928802589
F1 Score	0.96951559163176
Log Loss Score	0.59567670869609
Runtime	406.51

The next step we implement is using Grid Search to optimize 'num_ierations', 'learning_rate' followed by optimizing *‘num_leaves’*, *‘min_data_in_leaf’*, *‘max_depth’*. Let’s set learning rate as 0.1 and see what optimum value we get for *num_iterations*. Our objective is to obtain a num_iterations value closer to 100 and then for the final score evaluation we will increase this consequently decreasing *‘learning_rate*’. This will help the model learn slowly over time. Thus, the model will not learn aggressively. Below is the initial implementation of our grid search.

```
params = {'num_iterations' : [i*10 for i in range(6, 15, 2)],
          'learning_rate' : [0.5]}
GSCV = GridSearchCV(clf_lgbm, params, verbose = True, n_jobs = -1)
GSCV.fit(X_train, y_train)
```

Iterating through the Grid Search for all the parameters let us find the optimum values for our parameters. Once we find our optimum parameters we used the below code to train and evaluate our model.

a. Define the Model passing the tuned parameters.
b. Fit the model to the train data and evaluate it against the evaluation metrics.

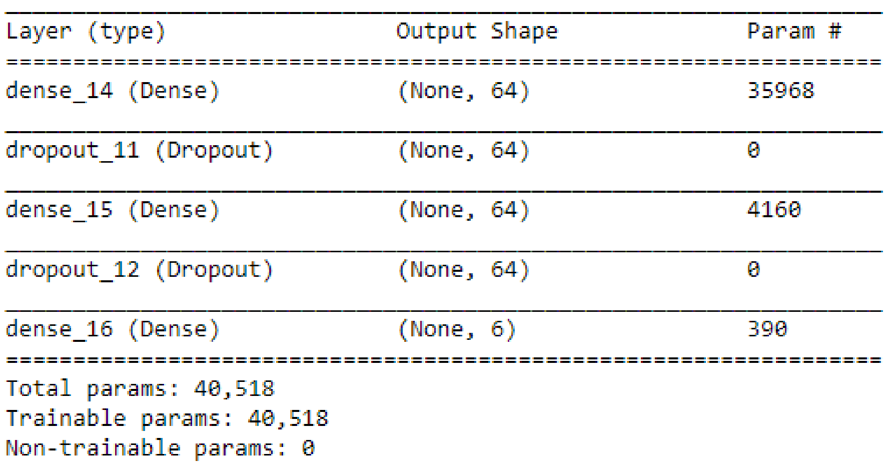
```
clf_lgbm = LGBMClassifier(boosting_type='gbdt', objective = 'multiclass', num_class = 6,
                          learning_rate = 0.05, num_iterations = 1000, num_leaves = 8,
                          min_data_in_leaf = 229, max_depth = 7, seed = 31)

# Fit to the training data
start = time.clock()
clf_lgbm.fit(X_train, y_train)
## make predictions
preds = clf_lgbm.predict(X_test)
end = time.clock()
y_pred = clf_lgbm.predict_proba(X_test)
print('Accuracy Score for LGBM Classifier is: {}'.format(accuracy_score(y_test, preds)))
print('F1 Score for LGBM Classifier is: {}'.format(f1_score(y_test, preds, average = 'weighted')))
print('Runtime is {} seconds'.format(end-start))
print('Log Loss Score is: {}'.format(log_loss(y_true, y_pred)))
```

Keras Neural Network: [Implementation]

Before fitting our dataset to Keras, we need to convert it into a format that Keras accepts. Currently we have our data as X_train, X_test, y_train and y_test with all numerical values as pandas dataframe. Keras takes input data in Numpy Array format. So, we will convert our dataset to a numpy array. As our datset is 2-dimensional, we can either use 'numpy.asarray()' or use '.as_matrix' for our conversion. We are using the later for our implementation. Furthermore, we have used one hot encoder to convert y_train and save it as y_train_ohe to use later.

We started with a basic small model with only 6 layers as shown below.

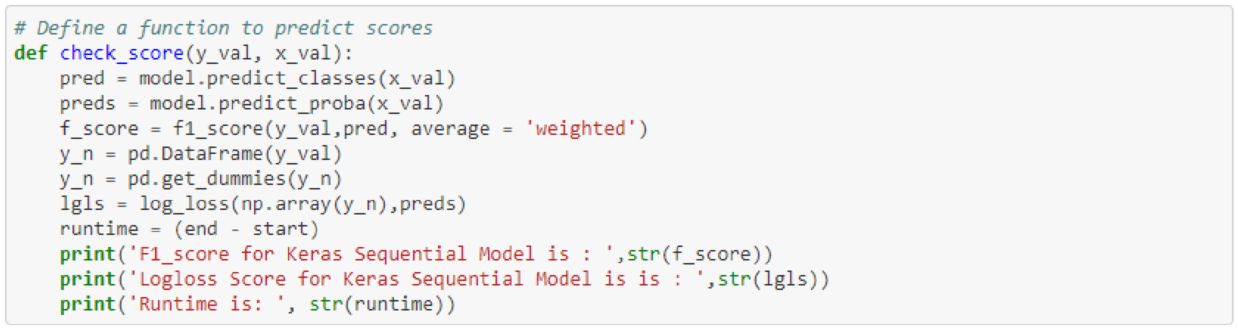


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_14 (Dense)             (None, 64)                35968     
_________________________________________________________________
dropout_11 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_15 (Dense)             (None, 64)                4160      
_________________________________________________________________
dropout_12 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_16 (Dense)             (None, 6)                 390       
=================================================================
Total params: 40,518
Trainable params: 40,518
Non-trainable params: 0
```

Our Base model uses SGD optimizer and accuracy as metric. Categorical Cross-entropy is used for the loss function as our problem is a multi-class classification problem. We have defined a function *'check_score'* which will check the F1 score, log loss and runtime for our model. Our base model performance is shown below. We have also used model checkpoint to store the best weight over the epochs.

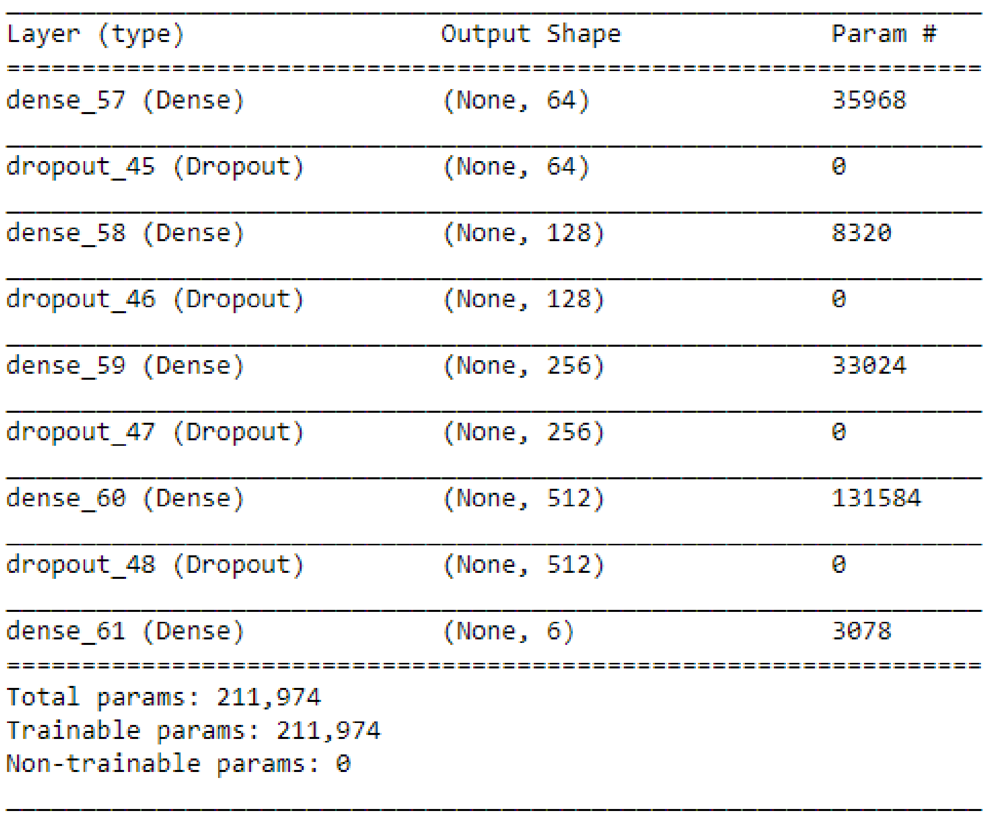
Model: Base Keras Model	
F1 Score	0.950770315993
Log Loss Score	0.131412646997
Runtime	7.04

We have defined a function which takes in arguments and provides the model performance. Below is the function implementation.



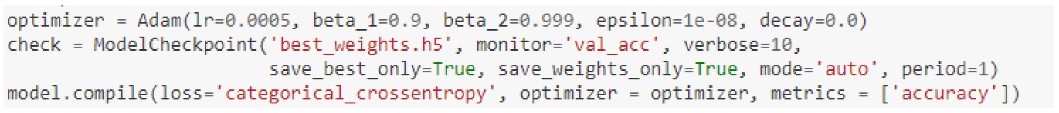
```python
# Define a function to predict scores
def check_score(y_val, x_val):
    pred = model.predict_classes(x_val)
    preds = model.predict_proba(x_val)
    f_score = f1_score(y_val,pred, average = 'weighted')
    y_n = pd.DataFrame(y_val)
    y_n = pd.get_dummies(y_n)
    lgls = log_loss(np.array(y_n),preds)
    runtime = (end - start)
    print('F1_score for Keras Sequential Model is : ',str(f_score))
    print('Logloss Score for Keras Sequential Model is is : ',str(lgls))
    print('Runtime is: ', str(runtime))
```

To improve our model accuracy, we have added more layers and tuning the parameters. (epochs, batch size). After iterative changes, by changing the optimizer, epochs, batch size, we reached at the below final model architecture. Our final model has achieved an F1 score of 98.213 which is far below than our optimized SVC score, which is 99.22. Hence, we chose to stop fine tuning our Keras further.

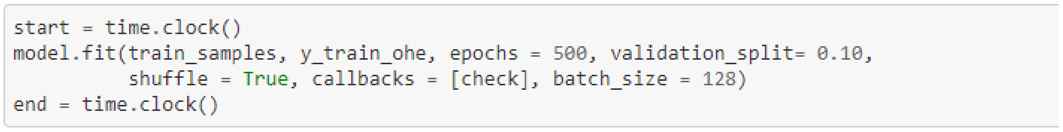


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_57 (Dense)             (None, 64)                35968     
_________________________________________________________________
dropout_45 (Dropout)         (None, 64)                0         
_________________________________________________________________
dense_58 (Dense)             (None, 128)               8320      
_________________________________________________________________
dropout_46 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_59 (Dense)             (None, 256)               33024     
_________________________________________________________________
dropout_47 (Dropout)         (None, 256)               0         
_________________________________________________________________
dense_60 (Dense)             (None, 512)               131584    
_________________________________________________________________
dropout_48 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_61 (Dense)             (None, 6)                 3078      
=================================================================
Total params: 211,974
Trainable params: 211,974
Non-trainable params: 0
_________________________________________________________________
```

The final Keras model is defined and fit by shown below. It has an Adam optimizer with a slow learning rate and hence we have increased the epochs respectively while training.



```
optimizer = Adam(lr=0.0005, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
check = ModelCheckpoint('best_weights.h5', monitor='val_acc', verbose=10,
                        save_best_only=True, save_weights_only=True, mode='auto', period=1)
model.compile(loss='categorical_crossentropy', optimizer = optimizer, metrics = ['accuracy'])
```



```
start = time.clock()
model.fit(train_samples, y_train_ohe, epochs = 500, validation_split= 0.10,
          shuffle = True, callbacks = [check], batch_size = 128)
end = time.clock()
```

An interesting thing to notice is that out of all our basic models, we see that Logistic Regression performs best with 98.18% F1 Score and 0.062 Log-Loss score taking only 6 seconds of runtime. Also, the basic model itself outperforms our benchmark model. But this model's performance does not improve that much significantly and the final model is an optimized SVC and not a Logistic Regression Model.

The primary complication that I faced during coding was that choosing the batch size for keras. I had set the batch size = 1000 for my basic model. My kernel kept dying and for a long time I could not get my code to work. Finally, I figured out the problem, which was due to my low ram size and so I decreased the batch size.

3.3 Refinement

Once we have fitted all 4 basic models, now it is time for us to optimize the parameters of each of the models. Below are detailed explanation of parameter tuning for each model

Support Vector Classifier: [Implementation]

We are using the Radial Basis Function(RBF) as the kernel for our classification problem. A SVC has two parameters that's need tuning: Gamma and C. Let's use GridSearchCV from sklearn to tune our parameters. Our initial search involves a logarithmic grid for both parameters. A dictionary containing ranges of gamma and C in the following range are passed

- C: [0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000]
- Gamma: [0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

For default cv of 3 folds, this Grid-Search of 99 parameters for each fit; totaling 297 fits, gives an optimum C = 100 and gamma = 0.01. Now as we know the range of C and Gamma, let us fine tune C and Gamma again. After a few turns of optimizing the parameters using Grid search we found optimum values for C and Gamma as

- C: 99
- Gamma: 0.0085

Our Optimum SVC gives the below results on the test data:

Model: Optimized SVC	
Accuracy Score	0.992233009709
F1 Score	0.992227107078
Log Loss Score	0.0256282742758
Runtime	18.730906

A simple stacked plot is shown below to compare the performance of the initial and final model. It can be seen clearly that our optimized model outperforms our base model in all aspects. (Please note that for representation purpose the values are scaled to fit the same plot, *Figure 3.2(b)*)

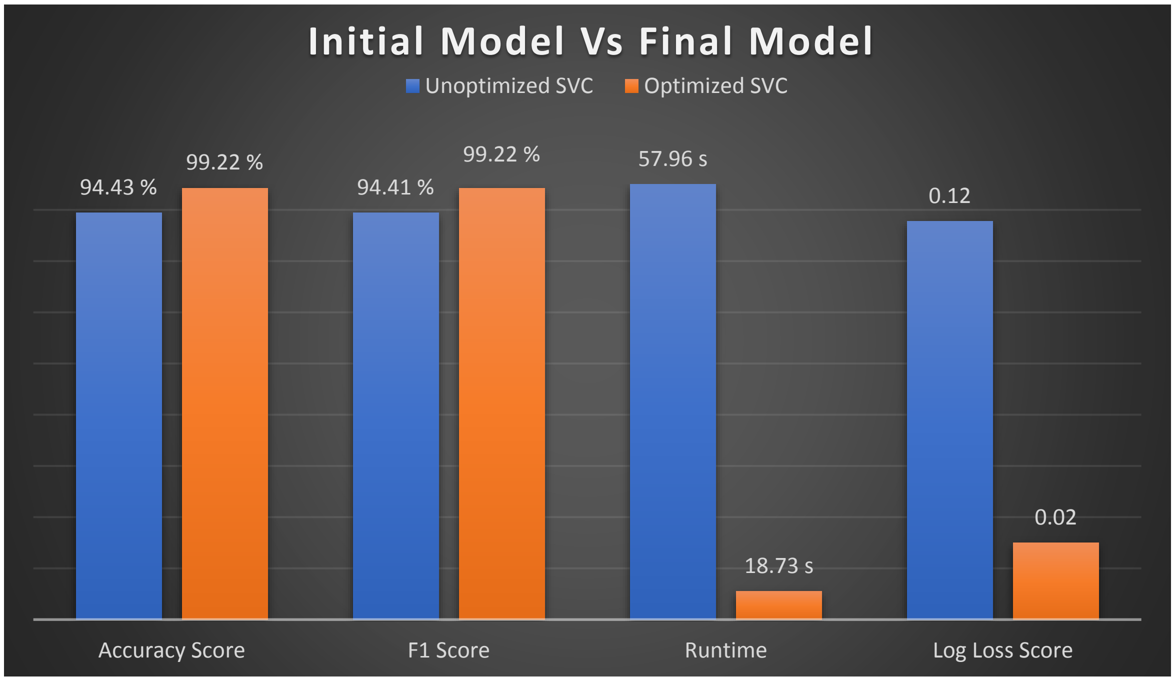


Figure 3.2(b)

Logistic Regression: [Implementation]

Here we have two parameters that we need to optimize. The solver and C. Solver refers to the optimization method to be used to find the optimum objective function. The C parameter refers to as the inverse of regularization strength. Similar to Support vector machines, smaller values refer to stronger regularization. We will pass the following dictonary for GridSearch to optimize the solver followed by C.

- solver : ['liblinear', 'sag', 'newton-cg', 'lbfgs']
- C : [0.001, 0.01, 1, 10, 100, 200, 500]

From this grid Search, we get solver = 'newton-cg' and C = 100 as our best choice. Let us tune C again in a range of 60 to 100 with step 10. This time we got C as 60, so lets check in the range 10 to 60 with step 10. This time best C is 50. So lets check in the range 45 to 55 with step as 1. We get C to be best at 46. One final time, let us fine tune C in range 45.5 to 46.5 with step 0.1. After a series of Grid Search, our optimum parameters are

- Solver = newton-cg
- C = 46

. Below is the performance of our optimized model.

Model: Optimized LR	
Accuracy Score	0.988025889968
F1 Score	0.988019092014
Log Loss Score	0.040566206566
Runtime	202.429314

LightGBM Classifier: [Implementation]

LGBM has a lots of parameters which needs tuning. We will tune parameters for the leaf-wise (Best-first) Tree. LightGBM uses the leaf-wise tree growth algorithm, while many other popular tools use depth-wise tree growth. Compared with depth-wise growth, the leaf-wise algorithm can convenge much faster. However, the leaf-wise growth may be over-fitting if not used with the appropriate parameters. To get good results using a leaf-wise tree, below are some important parameters that we should consider:

1. **num_leaves** : This is the main parameter to control the complexity of the tree model. Theoretically, we can set *num_leaves = 2^(max_depth)* to convert from depth-wise tree. However, this simple conversion is not good in practice. The reason is, when number of leaves are the same, the leaf-wise tree is much deeper than depth-wise tree. As a result, it may be over-fitting. Thus, when trying to tune the num_leaves, we should let it be smaller than *2^(max_depth).*
2. **min_data_in_leaf** : This is a very important parameter to deal with over-fitting in leaf-wise tree. Its value depends on the number of training data and *num_leaves*. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting.
3. **max_depth** : Used to limit the tree depth explicitly.

Now we will select '*objective*' as multiclass and *'numclass'* as 6 and then do parameter tuning for *'num_ierations'* with *'learning_rate'* as 0.5 followed by the tuning of above mentioned three parameters. Initially we will use a higher learning rate for faster runtimer while doing gridsearch of the other hyperparameters and finally we will set learning rate to a lower value and correspondingly increasing the num_iterations. Using Grid Search with cv = 3, we will do intital optimization for *'num_iterations*' in range 60 to 150 with step 20. This gives a value of 100 for num iterations. Now lets set *num_iterations= 100, learning_rate=0.5* and move on with tuning the other parameters. A series of gridsearch outputs the following values for our parameters:

- num_leaves : 8
- min_data_in_leaf : 229
- max_depth: 7

We are setting a '*seed'* value for reproducible results and selecting boosting type as '*gdbt*'. *'gdbt'* is a traditional Gradient Boosting Decision Tree. Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models.

Our final LGBM model is trained with learning rate 0.05 and hence increasing the 'num_iterations ' from 100 to 1000. Below are the performance of our final model:

Model: Optimized LGBMClassifier	
Accuracy Score	0.9912621359223301
F1 Score	0.9912578592647582
Log Loss Score	0.02471739179437413
Runtime	264.508288

Keras NN: [Implementation]

From Keras library the model we are selecting is simplest model, a Sequential class which is a linear stack of Layers. The first layer in our model must specify the shape of the input, hence we are setting *input_shape=561* where 561 being our total no of input features. Keras supports a range of standard neuron activation function, such as: relu, softmax, rectifier, tanh and sigmoid. The role of the activation function in a neural network is to produce a non-linear decision boundary via non-linear combinations of the weighted inputs. Initially, we are selecting our activation function as ‘relu’. Later we will se how other activation functions works with our model. ReLU stands for Rectified Linear Unit which is nonlinear in nature. ReLU can be expressed as a function: F(x) = 0 when X<= 0 and F(x) = x, when x> 0. It gives an output x if x is positive and 0 otherwise as shown below. [Figure 3.2(c)].

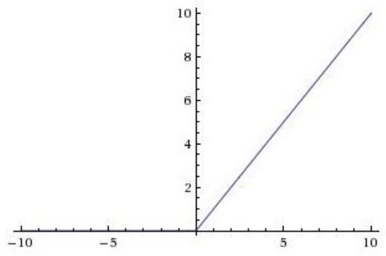


Figure 3.2(c)

Using ReLu prevents from activating all the neurons of our network at the same time and thus making the activation process more efficient and sparse. Moreover, ReLu works most of the time as a general approximator, and it is computationally less expensive as it involves simpler mathematical operations. Hence we are going with ReLU for our model. We have two dropout layer with 0.5 value each. Dropout can looked at as neurons randomly dropped out of the network during training, and other neurons stepping in to handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network. Dropout is implemented by randomly selecting nodes to be dropped-out with a given probability. In our case Dropout with 0.5 means probability of each node getting dropped is 50%. Also our model has dense layers with 64 nodes. Dense is a fully connected MLP layer. Our final dense layer uses 6 nodes for our 6 output classes and activation function as ‘Softmax’. The output of a Softmax function is used to represent a categorical distribution, a probability distribution over **N** different possible outcomes. The main advantage of using Softmax is the output probabilities range (0 to 1), and the sum of all the probabilities will be equal to one. If the softmax function is used for multi-class classification problem, it will return the probability of each class and the predicted target class will have the highest probability.

$$F(x_i) = \frac{exp(x_i)_{i=0,1,2,3....n}}{\sum_{j=0}^{k} \exp(x_j)}$$

The above formula computes the exponential of the given input value and the sum of exponential values of all the values in the inputs. Then the ratio of the exponential of the input value and the sum of exponential values is the output of the softmax function.

We have choose our optimizer as SGD with learning rate = 0.01, momentum = 0.9 and decay = 1e-6. Stochastic gradient descent (SGD), is a stochastic approximation of the gradient descent optimization tries to find minima or maxima by iteration. Our loss is 'categorical_crossentropy' as it is a multiclass classification problem and accuracy as metric.

For 20 epochs, our basic model with 40,518 trainable params gives 95.7% F1 score. Let us add a few more layers to check the performance. By adding another dense layer and changing nodes in each layer from 64 to 128 with a dropout(0.2), 128 to 256 with dropout(0.3), 256 to 512 with dropout(0.4) and final output layer with dropout(0.5), for 20 epochs our model performance detoriates. After increasing the no of epochs to 40 for this modified model, the performance of the model improved a lot. Form 95% to 97.41% F1 score.

Now, lets try to imrpove it further by tuning other parameters. Let's use 'relu' with optimizer as 'adam' with learning rate 0.001 for 100 epochs. Adam can be used instead of the classical stochastic gradient descent procedure to update network weights as iterative based in training data. Adam is well suited for data having large no of parameters (we have 561 features) and the hyper-parameters typically require little tuning. Unlike Stochastic gradient descent which maintains a single learning rate for all weight updates and the learning rate does not change during training, in case of Adam learning rate is maintained for each network weight and separately adapted as learning unfolds. With optimizer as Adam our accuracy has improved to 97.99%. Now, let's change the no of epochs, learning rate and batch size to improve the score further. First, increasing batch size to 128 from 64 and epoch to 200 improves score to 97.9%. Again, decreasing learning rate to 0.0005 and increasing epoch to 500 gives a F1 score of 98.213% with logloss 0.07. We can see that our validation accuracy did not improve so much over time over 200 epochs. This is our optimum keras model. Below are the model's performance:

Model: Optimized Keras NN	
F1 Score	0.982137574772
Log Loss Score	0.0742955556088
Runtime	374.608259

4. Results

4.1 Model Evaluation and Validation

We have evaluated all our 4 algorithms by optimizing their hyperparameters and below is the performance comparison of the best model of each algorithm.

Models	**F1 Score**	**Log Loss**
SVC	0.992227107078	0.0256282742758
LR	0.988019092014	0.040566206566
LGBM	0.9912578592647582	0.02471739179437413
Keras	0.985080830908	0.0558832170514

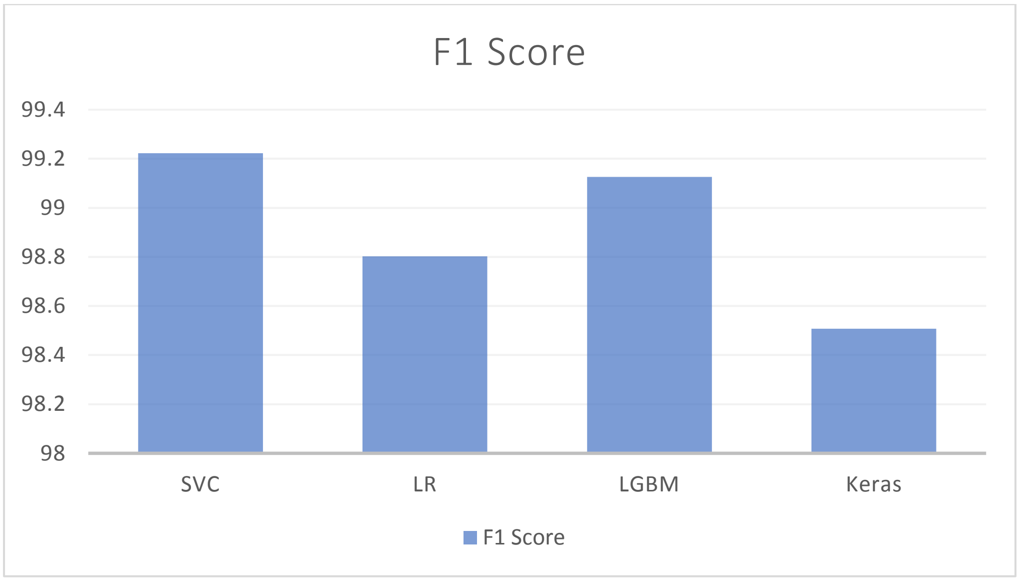


Figure 4.1(a)

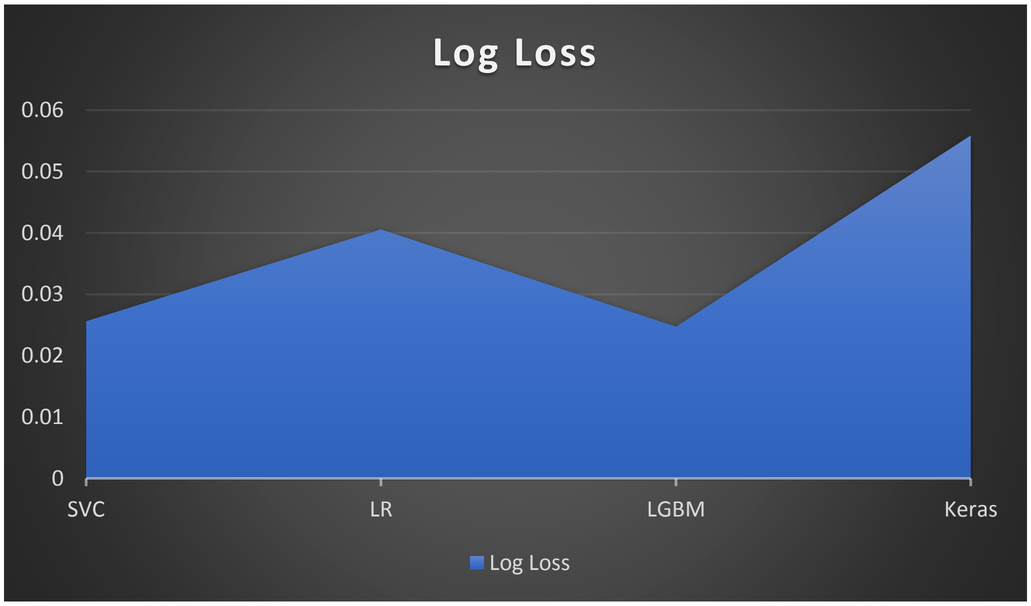


Figure 4.1(b)

From *Figure 4.1(a) & (b)* out of all 4 models only SVC and LGBM has a close competition in terms of their performance. LGBM has a slightly better score of log loss with 0.0247 against SVC's 0.0256. However, F1 of SVC is 99.22 where the same for LGBM is 99.12. It is interesting to see both the models perform nearly same. It is confusing to choose one over the other. So, let's compare the runtime of each of these two models.

- SVC: 18.73 Seconds
- LGBM: 264.51 Seconds

SVC outperforms LGBM in terms of runtime. Also, the F1 Score of SVC is better than that of LGBM. Hence, we choose SVC as our best model.

Our best model is a SVC with C and gamma 99 and 0.085 respectively. We started with a base model with default C and gamma and the performed logarithmic gridsearch for these two parameters with RBF kernel. After finding C =99 and gamma 0.85 further gridsearch gives a C = 120 with gamma 0.8. These values of C and overfits the data as we can see from our test F1 score and log loss as compared to our earlier values. The values and C and gamma should be chosen carefully such that it does not overfit the data. Our optimum model takes a mere 18 seconds of runtime and gives a 99.22% of F1 score with logloss as low as 0.025 which can be considered as a good model.

Our final SVC model can be considered a robust one since it gives pretty good results on the unseen test data set with a very little runtime. The logarithmic loss score is also quite small which shows our model is confident about its predictions.

4.2 Justification

Our model surpasses the benchmark in every aspect of performance. Our benchmark model which was also an SVC obtained an Accuracy: 96.34, Precision: 96.44. Whereas in our model where we considered a harmonic average of precision and recall obtained a score of 99.22, which can be considered an significant improvement over the benchmark's performance. Furthermore, we considered log loss as another metric for our model evaluation which reached a minimum of 0.025 which suggests our model is confident about its predictions.

4. Conclusion

4.1 Free-Form Visualization

From the below learning plot [*Figure 4.1(a)*], we can see that training score remains constant throughout. However, the Cross-validation score starts at the lowest when training examples are less and as training examples increases CV score also gradually increases, reaching its maximum value at maximum no of training examples. From the plot, it is evident that validation score can further be increased with increase in training examples. So, adding more data points to our training set will improve our model accuracy further. [Implementation]

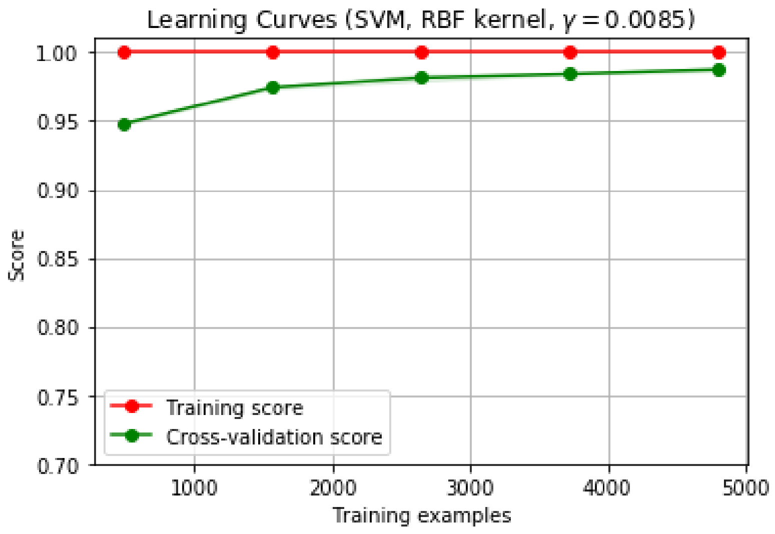


Figure 4.1 (a)

4.2 Reflection

I started this project with a reading a lot of work related to this and similar problems. There are a vast number of well documented solutions for this particular problem using various ML algorithms. I came across various implementations such as CNN, LSTM, tensor-flow, XGBoost. This is when I decided to try a few of my own implementations starting with the basic model and then optimizing the models to find an optimum one. I decided to use LightGBM instead of XGBoost as I never used LGBM before and it has earned a reputation to work as good as XGBoost with lesser runtimes. I have learned a lot about LGBM and how its parameters work while implementing the same for this project. The purpose of the project was to find a model among the 4 selected algorithms, which outperforms the other three while also challenging the already well-established implementations in terms of performance. And from the final SVM that we optimized, it is safe to say that we developed a model which can confidently predict an activity based on the recorded smartphone sensor data.

While exploring the dataset the first thing that I found quite interesting was that the dataset was already mostly preprocessed with all the values scaled and normalized. Also, the dataset was quite equally distributed among the six classes. Another interesting fact was both SVC and LGBM performed quite neck to neck after optimization with SVC surpassing LGBM in terms of F1 score. However, LGBM took a long time to do the same job which SVC did with a better accuracy in less time.

The major challenge I faced was while working with LGBM. This was my first implementation of LGBM and to select which parameters to give more importance out of a list of available parameters was quite a challenge. I had to read documents, already implemented solutions for similar problems to understand which parameters I should give more importance, how to set parameters values and tuning which parameters would help us prevent overfitting while improving accuracy.

4.3 Improvement

Even though our model has reached an F1 score of 99.22% with log loss 0.025 and the model seems nearly perfect, there are still room for improvement. Below are a few points we should consider which will improve the model accuracy:

- C and Gamma can further be fine-tuned to improve performance.
- Adding more data points to the training dataset will increase our model performance as evident from the learning curve plot. [*Figure 4.1 (a)*]

References:

1. Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz. Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine. International Workshop of Ambient Assisted Living (IWAAL 2012). Vitoria-Gasteiz, Spain. Dec 2012
2. Oscar D. Lara and Miguel A. Labrador. A Survey on Human Activity Recognition using Wearable Sensors, IEEE COMMUNICATIONS SURVEYS & TUTORIALS, VOL. 15, NO. 3, THIRD QUARTER 2013
3. https://github.com/guillaume-chevalier/LSTM-Human-Activity-Recognition.