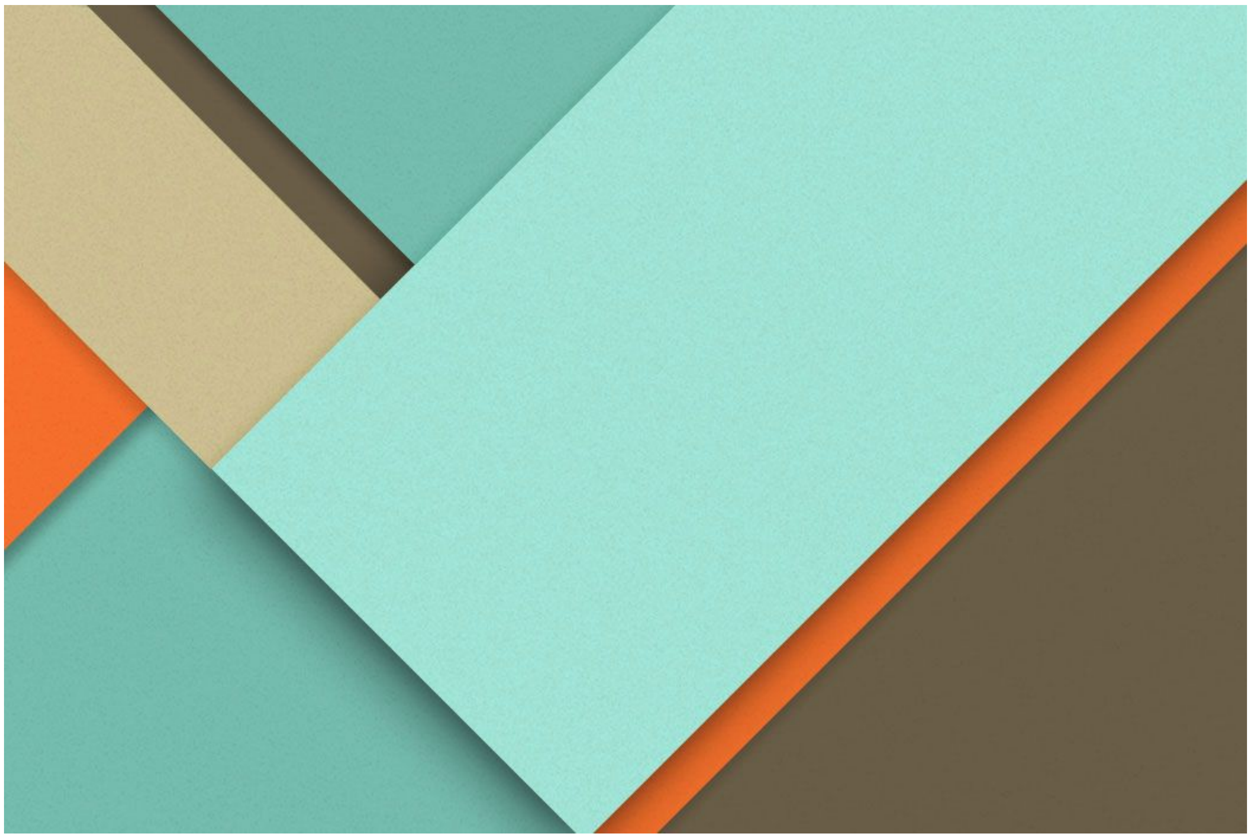


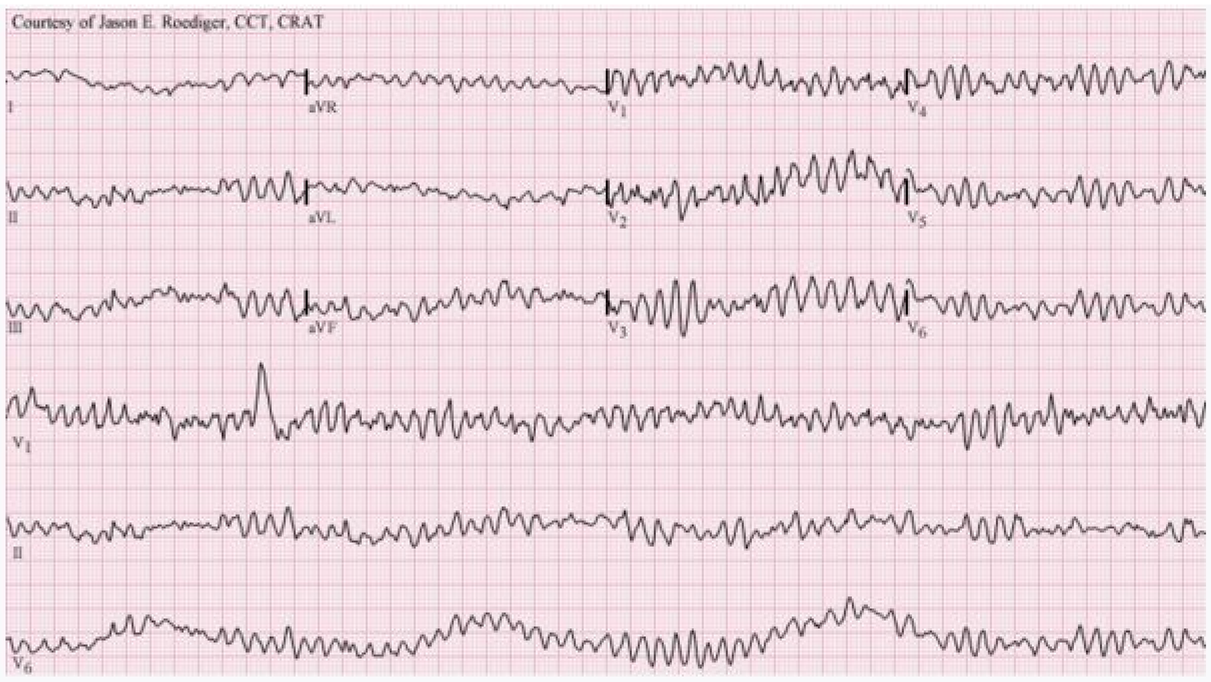
Arrhythmia Classification

07.19.18

Ishu Kalra
Machine Learning Advance Nanodegree

Overview

Heart arrhythmia (also known as **arrhythmia**, **dysrhythmia**, or **irregular heartbeat**) is a group of conditions in which the heartbeat is irregular, too fast, or too slow. A heart rate that is too fast – above 100 beats per minute in adults – is called tachycardia and a heart rate that is too slow – below 60 beats per minute – is called bradycardia. Many types of arrhythmia have no symptoms. When symptoms are present these may include palpitations or feeling a pause between heartbeats.

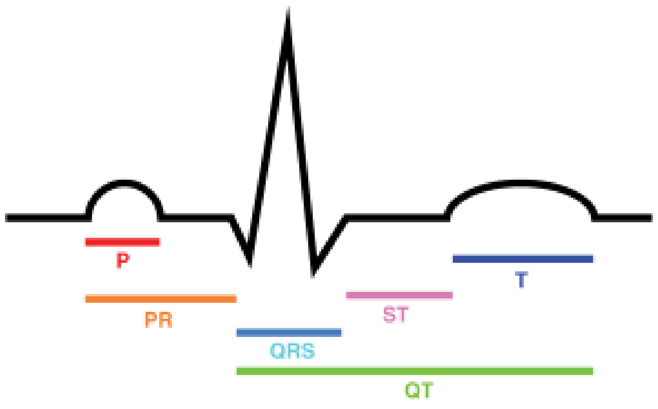


Ventricular fibrillation (VF) showing disorganized electrical activity producing a spiked tracing on an electrocardiogram (ECG).

Here, I am going to MIT-BIH Arrhythmia dataset to implement a neural network based approach to classify the arrhythmia type. The dataset has total 50 classes that needs to be classified.

I will be using CNN (Convolutional Neural Networks) to classify, based on the input ECG QRS complex, the type of the signal.

Link to the dataset: https://physionet.org/physiobank/database/mitdb/



The QRS complex is the peak like feature shown. This peaks helps in identifying cardiac arrhythmias.

CNN are extremely good for feature extraction and hence this is the reason for using them. They will extract the QRS feature and help in classification. This is the Deep Learning Approach.

The Machine Learning approach will use ensemble learning to classify using algorithms like SVN, k-Neighbors, Logistic Regression, Extra Trees etc.

Problem Statement - Deep Learning

Problem: *Arrhythmia classification based on MIT-BIH Arrhythmia dataset.* The dataset has 48 patients ECG in digitized form. This digital form (stores y coordinates) can be easily used to reconstruct the ECG signal image.

Strategy: Using the digital data, the signal is again formed and stored in a variable (holds the coordinates to reform the signal). This variable is passed to a CNN model to extract the features, specifically, the QRS complex which helps in determining the type of arrhythmia. The trained model is then used to check accuracy and validation accuracy.

Model Inputs/Outputs: Output is the class of predicted arrhythmia. The input is the QRS complex as shown above. The model trains with the QRS and the class of Arrhythmia it represents. Then for prediction, a new QRS complex signal is fed and the model then determines the class of arrhythmia.

Metrics: The validation accuracy and accuracy and validation loss and loss are used as a metric to measure the performance of the trained network.

Metric Justification: Validation Accuracy represents how good the model performed on the validation set (not train set). The model learns through the training set and is tested on the validation set (the set which the model has never seen before to ensure that it does not literally memorize the data) to check its performance. This is usually done on all Deep Learning models and as far as I know, is a standard in industry.

The training set is split into train test and test test. The model is trained only on the train test and validated on the test set to get the output. The keras model used here, model.fit itself outputs a history object. This can be easily used to make validation accuracy and loss graphs. This has been implemented in the notebook and can be viewed at the end.

Data exploration

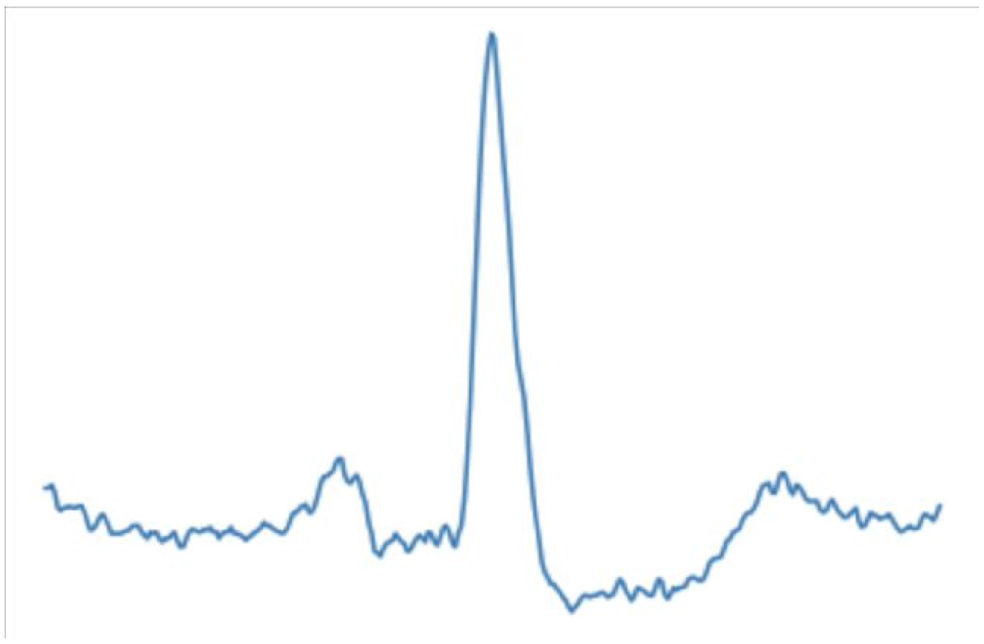
The data is of 48 patients, digitized at 360 frequency, that is, 360 annotations per second in 10mV range. Their are overall 110000 beat annotations and total of 50 cardiac arrhythmias.

https://www.physionet.org/physiobank/annotations.shtml discusses about the varios arrhythmias and their meanings.

All the data exploration is done in the notebook itself with comments. Please refer the notebook for further details.

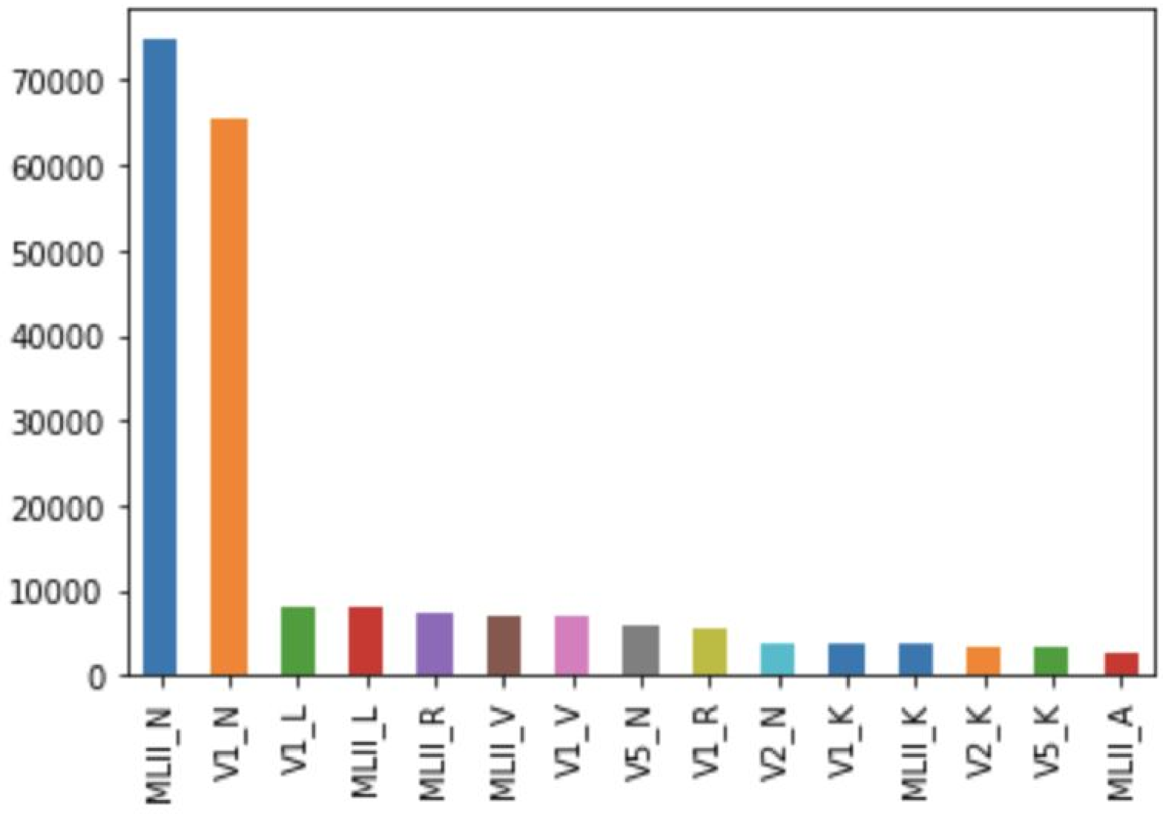
Data Visualization

Data has been visualized in the jupyter notebook itself. The reconstructed signal has been shown in a cell. Here, I gave a snip of reconstructed ECG QRS concerning a particular annotation.



The visualization has been done using the stored coordinates of the signal itself contained in each instance.

Since each signal is highly varied, it is impossible to generate a signal visualization of whole dataset here. Therefore, I have generated the bar graphs of top 10 most frequent QRS type here.



Data Preprocessing

The glob library is used to extract the .dat files containing all the signal with the labels. Then by using the signal processing library of python named wfdb, we extract the signal (the x and y coordinates of time series data) along with its label and qrs_type. The qrs type is length is taken as 300 (The half qrs is 150 in the notebook). These steps are marked out clearly in the notebook to make it clear.

Benchmarks

The following paper has been used a benchmark
http://cs229.stanford.edu/proj2014/Samuel%20McCandlish,%20Taylor%20Barrella,%20Identifying%20Arrhythmia%20from%20Electrocardiogram%20Data.pdf

They have used a MLP and a Machine learning approach. Here, the MLP is the benchmark. They have used a single hidden layer and feeding the data in four forms (raw data, FFT, Haar wavelets and DB3 wavelets). Of these four forms, the correct benchmark should be raw data since we are also feeding it in the same form. Using this they have secured a generalization error of 0.25 compared with the baseline error 0f 0.521. I believe, my model with more hidden layers and batch normalization, dropouts etc. performs much better.

Algorithms and Techniques - The Theory

Input

The input is the coordinates of the signal. This is essentially the signal as shown above which is fed to the network.

CNN

CNN are used because they are best to use for any image data. They easily extract the features from the signal. The complexity of these extracted signals depends upon the layer of the CNN. For example, a start layer can at best identify the edges or lines. The next second layer will then extract the more complex features like shapes. The next layer will handle even more complex features like artifacts formed from those shapes. As such, the complexity of extracted features increase with the depth of the layers. Also, they are kind of immune to vanishing gradient problem. That is more the layers, the better the network trains unlike other neural networks.

Fully Connected Layer

These are the layers that are present in your normal Neural Networks. Here the input (taken from CNNs) is multiplied by the weights and bias is added. The result is forward passed to the next layer and later through backpropagation, the value of these weights are modified.

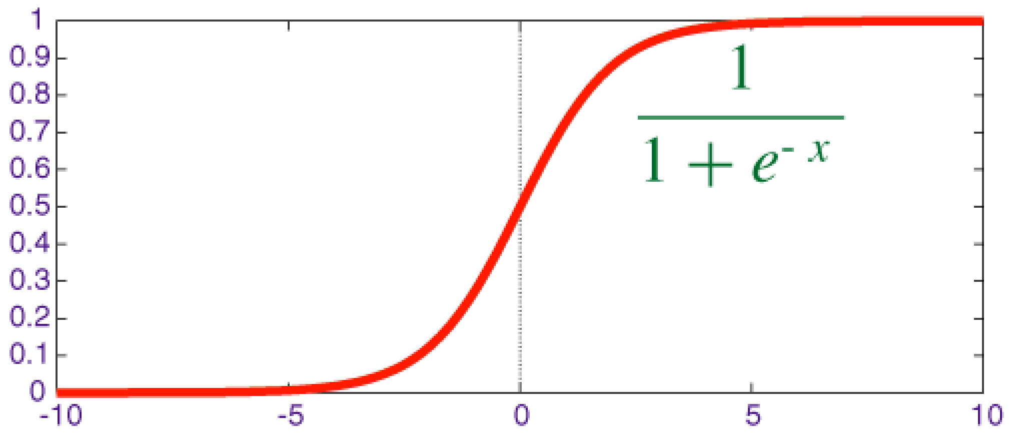
Output Layer

The output layer is the final layer and has nodes equal to class labels that needs to be classified. For example, say I want to classify in one of 50 types of arrhythmia, so the class label is 50 here. Thence, number of nodes is 50. Since we want only one class label to be predicted (single class output and not multi class output), the softmax or sigmoid function can be used. Here, I have used the softmax function.

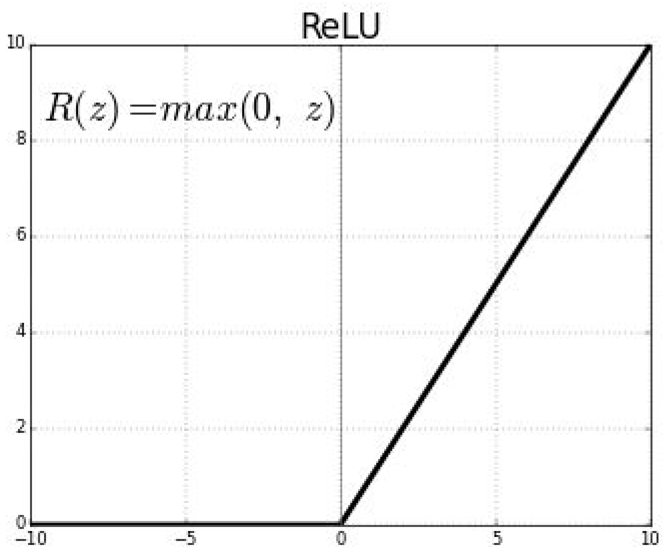
The softmax outputs the class probabilities for each label (such that their sum is 1). The highest of these is the predicted class.

Another use of class probabilities is in predict_proba() method of scikit-learn that is essential to use their ensemble learning module.

Below, I have attached the graph and formula of softmax. The x there is the output.



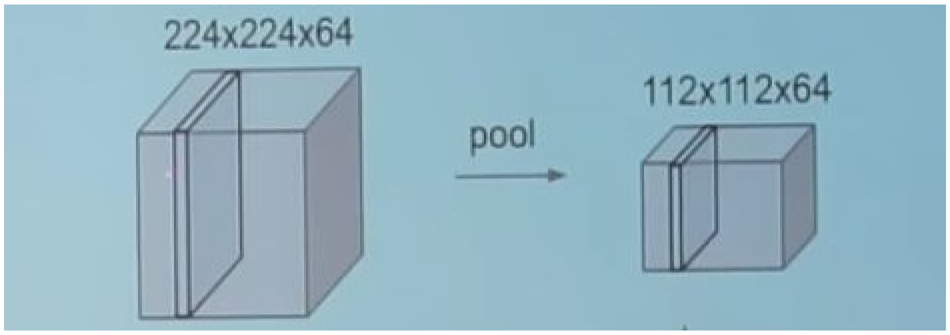
Another function used is relu. Relu stands for **Re**ctifier **l**inear **u**nit and is one of most used function in NN. It is a quite simple function.



The z is the input here. The function and its derivative both are monotonic.

Pooling

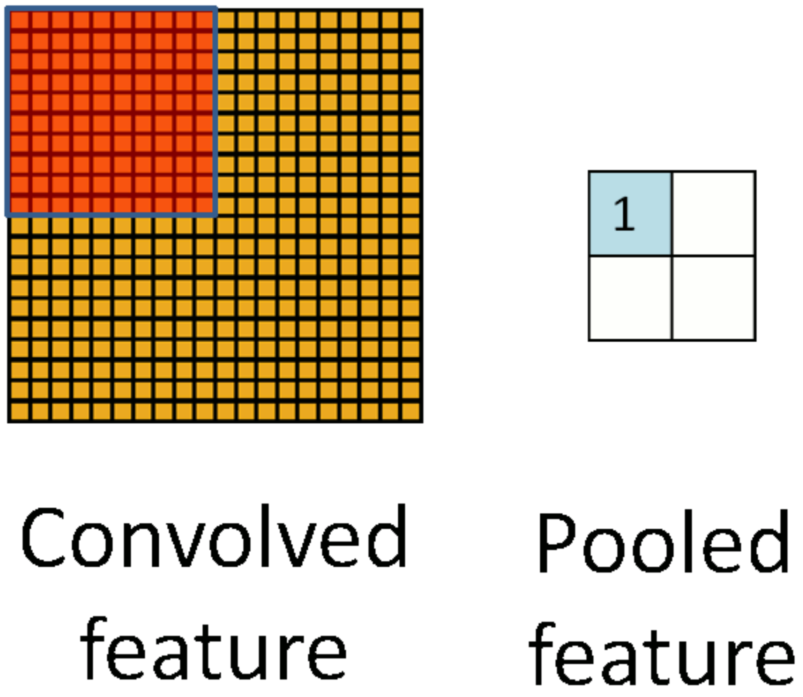
A pooling layer is another building block of a CNN.



Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Pooling layer operates on each feature map independently.

The most common approach used in pooling is *max pooling*.

The max pooling searches for the max number in the area (red) and outputs it.



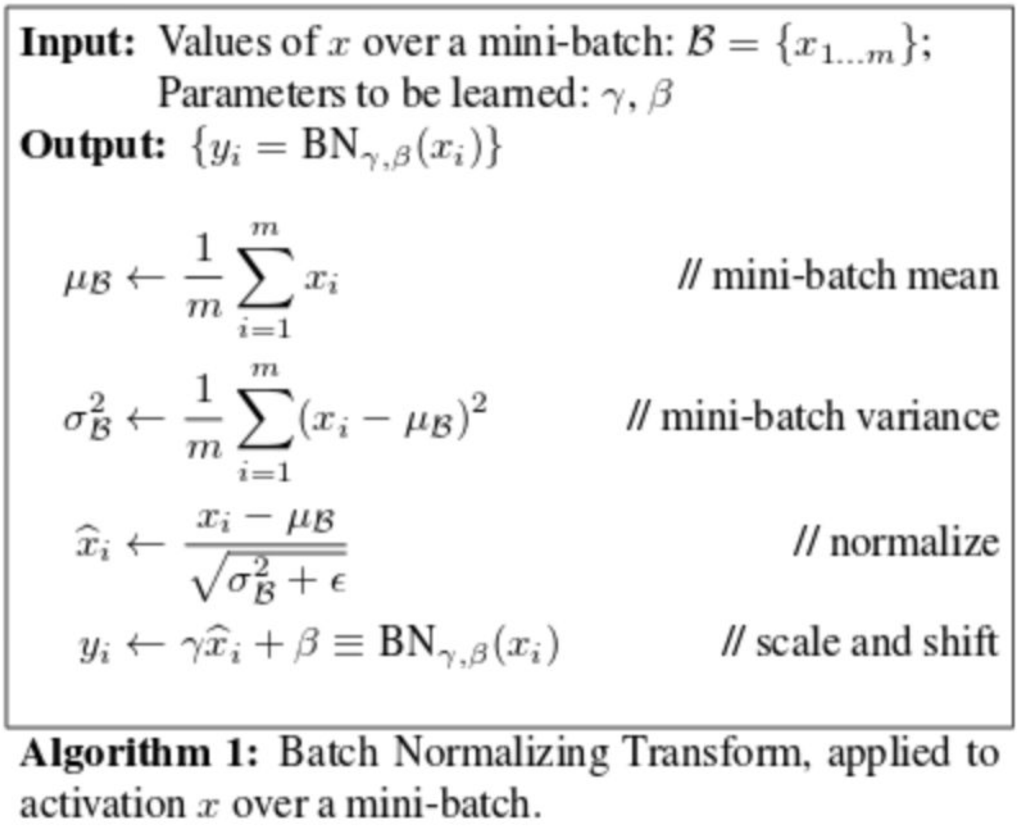
Another approach is to use average pooling or min pooling. Here, I have used max pooling.

Flattening

Flattening squashes our matrix (nxn) into a vector form (n^2x1). This is essential since we need to feed our data to a fully connected layer (and not a a CNN layer!).

Batch Normalization

As the name suggests, it is 'normalizing' the inputs with respect to each batch of the fed data. It is done by adjusting and scaling the activations. It helps in reducing covariance shift and gives better result overall. In essence, it allows each layer to learn by itself a little more independently of other layers.



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$    // mini-batch mean
    $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$    // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$    // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$    // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

From the original batch-norm paper

Algorithms and Techniques - The Implementation

In preprocessing steps, I did not encountered any problem or complications. When Making the CNN model, I tried to make it efficient and fast. That is why I settled with 3 CNN layer and 3 hidden layer.

I am using relu activation function which has now become quite a standard now. In the final layer, I am using softmax function since I want class probabilities. Other than that, I am using dropout as 0.25 which is again the standard or default value usually used.

Evaluation Metrics

The model.fit methods outputs a matplotlib object which is used to check the validation loss and accuracy. The graphs has been plotted in the notebook itself. Please refer it. Using this method, we have reached a validation accuracy of ~89.1%

Result

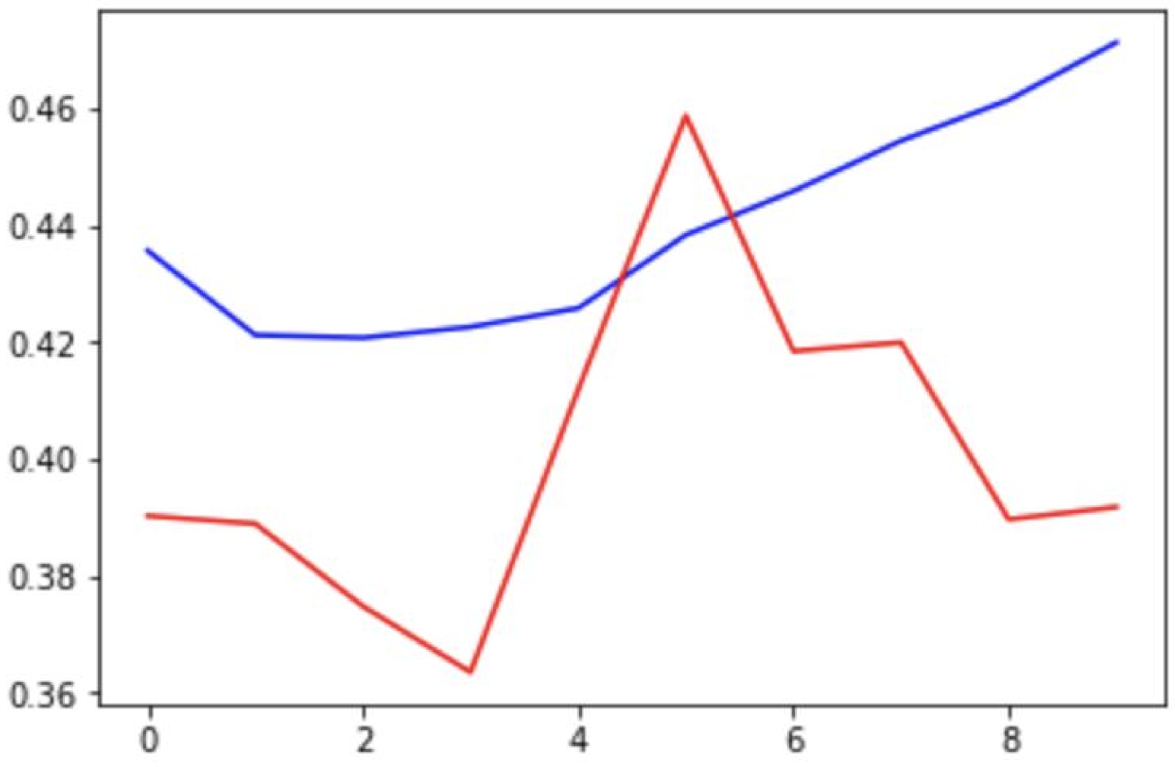
The model.fit methods outputs a matplotlib object which is used to check the validation loss and accuracy. The graphs has been plotted in the notebook itself. Please refer it. Using this method, we have reached a validation accuracy of ~87.5%. The benchmark had an error of 0.25 and our model of course, with more CNN layers and hidden layers performs much better.

Note that I am comparing this with the raw data part that they fed (Like I have done) into the MLP and not Transformed Signals like DB3.

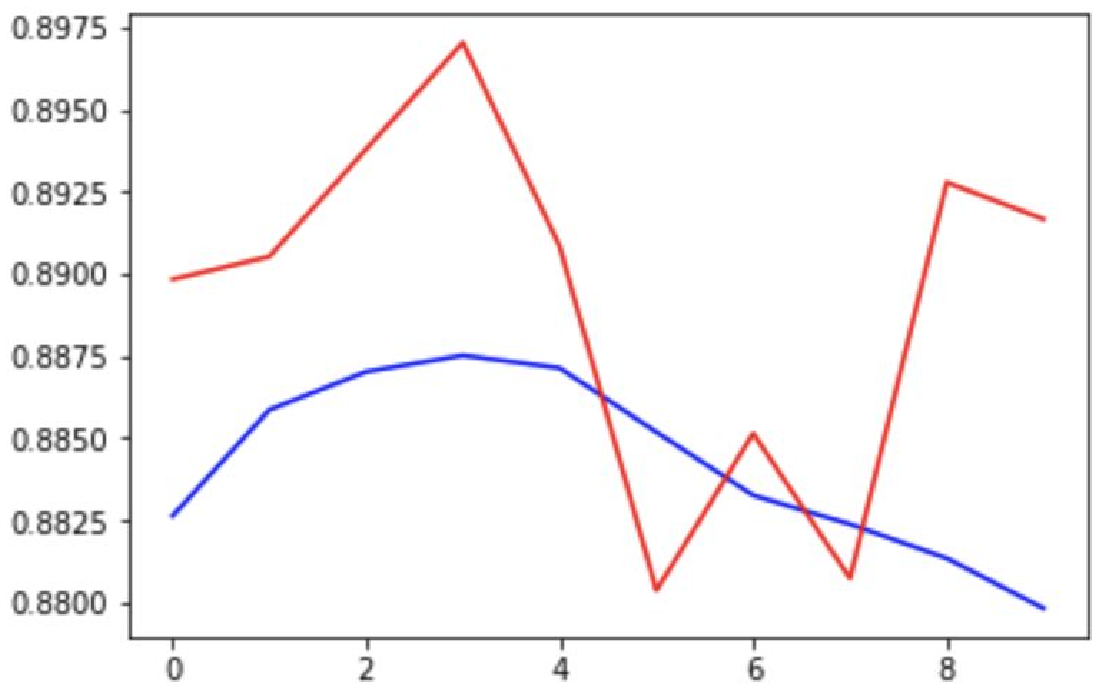
Conclusion

Visualization of the results:

1. Validation Loss (red) vs Training Loss (blue)



2. Validation Accuracy (red) vs Training Accuracy (blue)



The validation accuracy can be improved by implementing the following:

1. More CNN layers: In CNN, the more the layers, the better is the performance without any vanishing gradient problem
2. Cleaner Data: The data provided has a lot of background noise which skews the result. This interferes with the learning of Neural Network. Perhaps instead a long signal, short burst of signal with the anomaly of arrhythmia contained in it will help in better classification.
3. More Data: In deep learning, more the data, the merrier. This enables very accurate prediction. The MIT-BIH currently has very less data points.

Things I found interesting about this project:

1. How the many epochs in training actually decreases the efficiency of the model. I guess this can be countered by checking the loss at every epoch and stopping the model when this stops to better the model (early stopping and warm start)

2. Signal complexity: Usually it takes years of training to predict the correct arrhythmia class from the ecg signal by the cardiologists. I am surprised by the fact that how with the limited and complex data, our model is giving such good results.

Summary - end to end problem solution

Here, I have taken the ECG recordings (raw data) with x and y coordinates of the signal and fed it to a MLP with 3 CNN and 3 hidden layer to predict the class of arrhythmia. The weights of the layers are adjusted till the right output is not predicted. Moreover, I have used ReLu activation functions and Dropouts and Batch Normalization to improve my model.

Problem Statement -Machine Learning

Problem: *Arrhythmia classification based on UCI Arrhythmia dataset.* This has 452 instances that needs to be classified into 16 classes of arrhythmia.

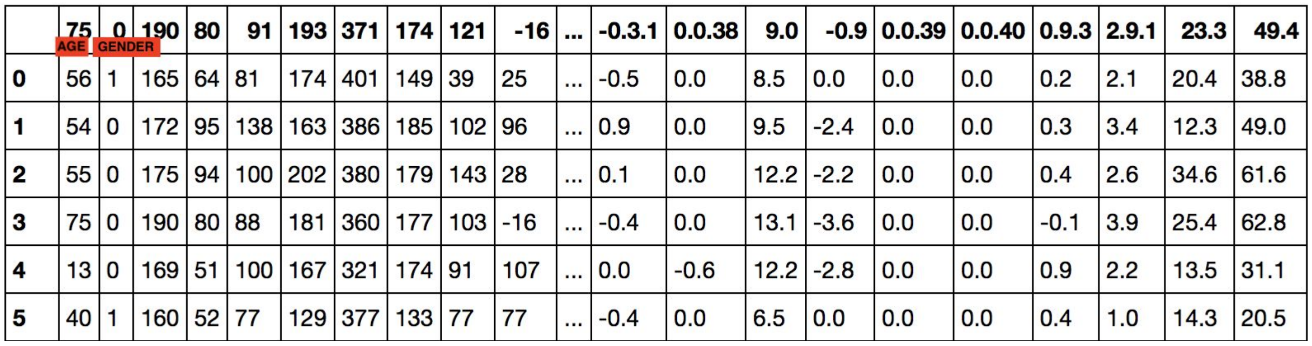
Strategy: Using the digital data, the data is first adjusted by handling the missing values. This is done using Imputer Class of scikit learn. Then the data is scaled using the inbuilt library of scikit. Scaling is necessary as some algorithms like k-Neighbors are sensitive to it

Metrics: Evaluation is done by using the predict method after fitting the data.

Note: All warnings in this iPython notebook has been disabled.

Data exploration

All the data is converted into a panda Data Frame and is explored in the notebook itself using methods like info(), describe() etc. The data has the patient age in column 1, his gender in column 2 and clinical data in rest of the columns.



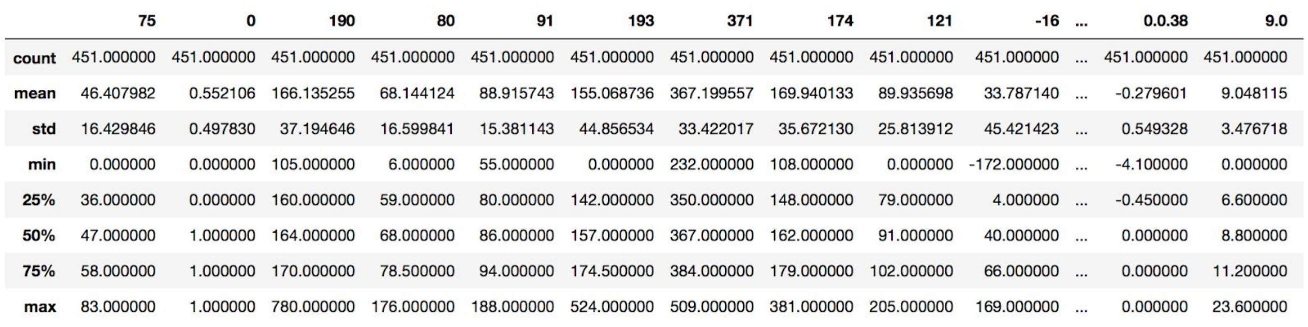
	75	0	190	80	91	193	371	174	121	-16	...	-0.3.1	0.0.38	9.0	-0.9	0.0.39	0.0.40	0.9.3	2.9.1	23.3	49.4
0	56	1	165	64	81	174	401	149	39	25	...	-0.5	0.0	8.5	0.0	0.0	0.0	0.2	2.1	20.4	38.8
1	54	0	172	95	138	163	386	185	102	96	...	0.9	0.0	9.5	-2.4	0.0	0.0	0.3	3.4	12.3	49.0
2	55	0	175	94	100	202	380	179	143	28	...	0.1	0.0	12.2	-2.2	0.0	0.0	0.4	2.6	34.6	61.6
3	75	0	190	80	88	181	360	177	103	-16	...	-0.4	0.0	13.1	-3.6	0.0	0.0	-0.1	3.9	25.4	62.8
4	13	0	169	51	100	167	321	174	91	107	...	0.0	-0.6	12.2	-2.8	0.0	0.0	0.9	2.2	13.5	31.1
5	40	1	160	52	77	129	377	133	77	77	...	-0.4	0.0	6.5	0.0	0.0	0.0	0.4	1.0	14.3	20.5

The 0 denotes Male and 1 denotes female in column 2. For detailed info about each column, refer https://archive.ics.uci.edu/ml/machine-learning-databases/arrhythmia/arrhythmia.names

This link is provided by data repository host and contains disambiguations about each column.

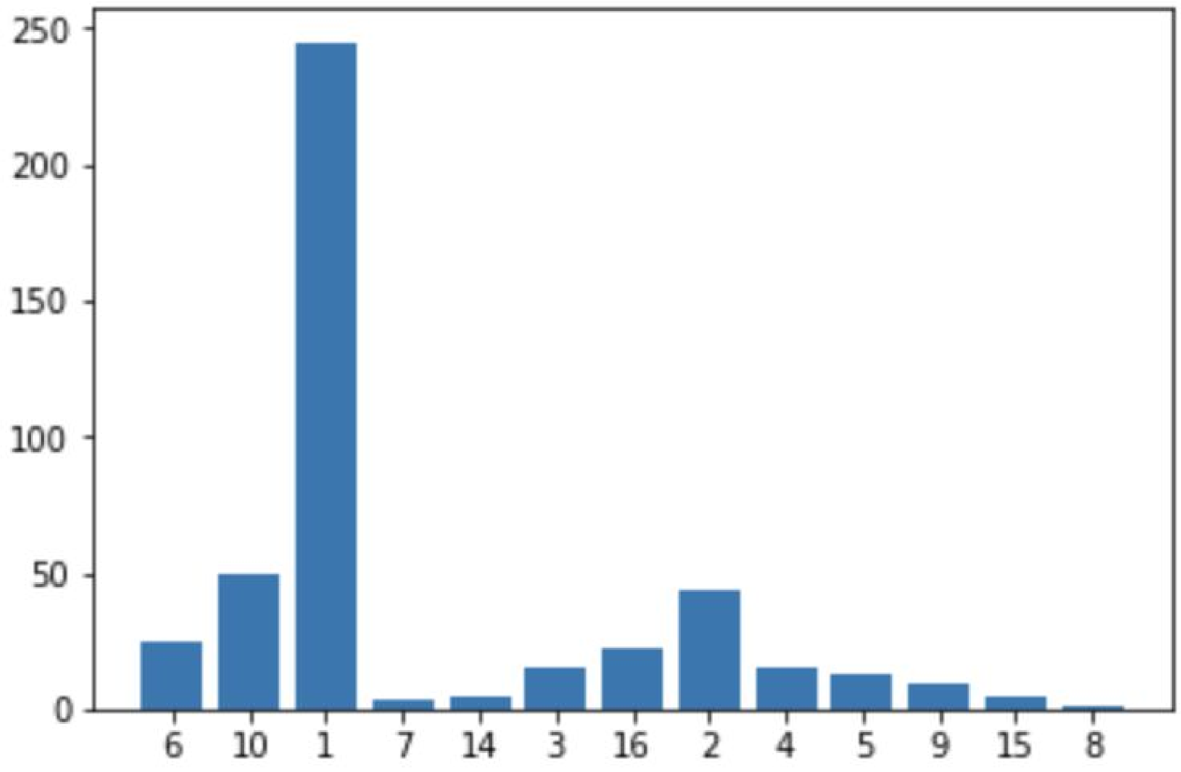
Data visualization

A summary of the whole dataset with their counts, mean, standard deviations, 25 percentile, 50 percentile, 75 percentile and max value for each column is shown below. This was generated using df.describe() method of pandas library.



	75	0	190	80	91	193	371	174	121	-16	...	0.0.38	9.0
count	451.000000	451.000000	451.000000	451.000000	451.000000	451.000000	451.000000	451.000000	451.000000	451.000000	...	451.000000	451.000000
mean	46.407982	0.552106	166.135255	68.144124	88.915743	155.068736	367.199557	169.940133	89.935698	33.787140	...	-0.279601	9.048115
std	16.429846	0.497830	37.194646	16.599841	15.381143	44.856534	33.422017	35.672130	25.813912	45.421423	...	0.549328	3.476718
min	0.000000	0.000000	105.000000	6.000000	55.000000	0.000000	232.000000	108.000000	0.000000	-172.000000	...	-4.100000	0.000000
25%	36.000000	0.000000	160.000000	59.000000	80.000000	142.000000	350.000000	148.000000	79.000000	4.000000	...	-0.450000	6.600000
50%	47.000000	1.000000	164.000000	68.000000	86.000000	157.000000	367.000000	162.000000	91.000000	40.000000	...	0.000000	8.800000
75%	58.000000	1.000000	170.000000	78.500000	94.000000	174.500000	384.000000	179.000000	102.000000	66.000000	...	0.000000	11.200000
max	83.000000	1.000000	780.000000	176.000000	188.000000	524.000000	509.000000	381.000000	205.000000	169.000000	...	0.000000	23.600000

A bar graph showing instances of each arrhythmia is plotted. Since it is plotted using a dictionary, they (the x values) are not in order. Nonetheless, it shows that which are most frequent classes of repeated arrhythmia in the dataset.



Benchmark

Please refer the following link to get the benchmark model
http://cs229.stanford.edu/proj2014/Manas%20Karandikar,%20Giulia%20Guidi,%20Classification%20Of%20Arrhythmia%20Using%20ECG%20Data.pdf

Here thy have used only two most frequent classes of arrhythmia (class 1 and class 2). They have used SVN and Random Forests, and achieving max accuracy of 86%. I believe my model, with ensemble and Linear SVM performs much better.

Algorithm	Accuracy	Training speed	Testing speed
Decision Trees	78%	0.11s	0.001s
SVM for 2 classes and 279 features	80%	1.4s	0.033s
SVM for 2 classes and 11 features	86%	0.3s	0.011s
Naive Bayes	68%	8.5s	1.55s
Random Forest	78%	23s	0.1s

Data Preprocessing

The data is read and the missing annotations denoted by ? are replaced by 0 to preserve the inherent nature of whole data and later prevent the breaking of the data. This part is elucidated in the notebook itself. I have only kept those labels 1 and 2 and have drop the rest from the panda data frame. Then the imputing is done using the most-frequent or the median metric. This essentially replaces the missing values with the median or the mode of the value. Usually, in this case the mode should work the best. This is then made into a pipeline. Then we do the standard scaling of data to normalize it. Finally we have the preprocessed data stored in a nice Panda Frame.

Algorithms and Techniques - The Theory

Splitting the data set

When splitting the dataset, using Stratified Sampling is preferred since it preserves the percentage of sample of each class. Unfortunately, this is not possible with the UCI dataset since some classes have instances of only one (it requires minimum 2 instances)

So normal splitting using model_selection.test_train_split() has been done

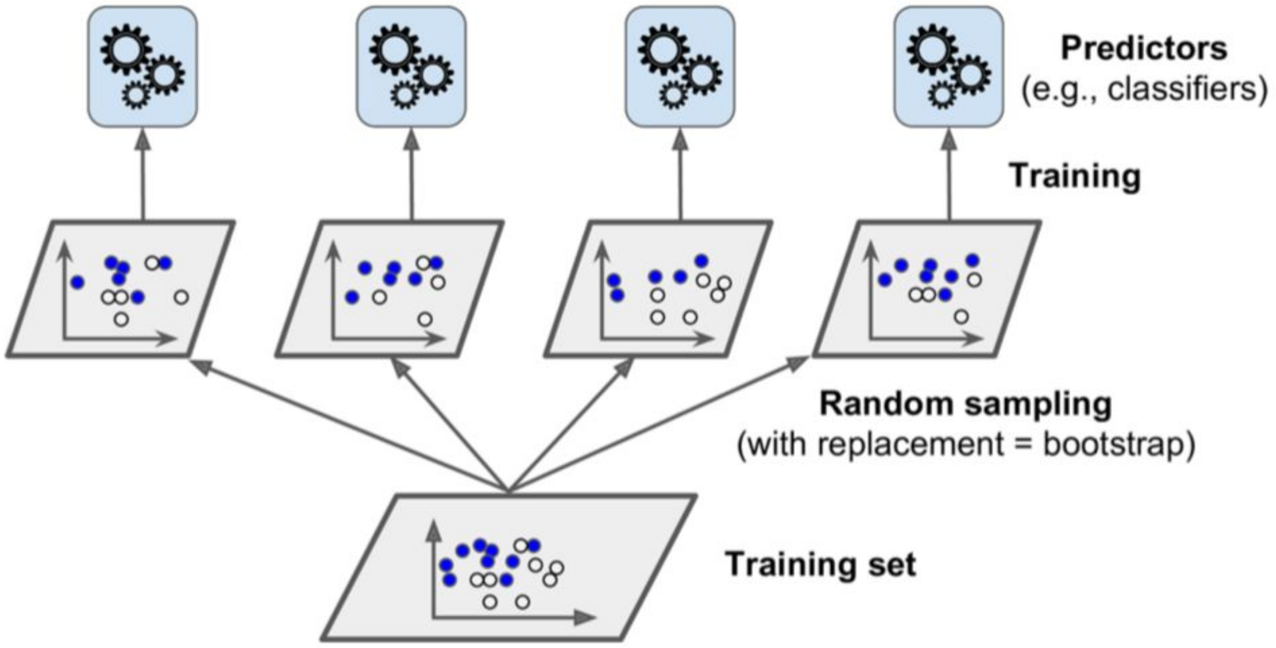
Ada Boosting

The dataset was boosted using Adaptive Boosting (AdaBoost) and Gradient Boosting. Boosting refers to any Ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor under-fitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost

Bagging & Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as will be done. Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When sampling is performed *with* replacement, this method is called *bagging*1 (short for *bootstrap aggregating*2). When sampling is performed *without* replacement, it is called *pasting*. Image is attached below. Here, since I have not passed any base-estimator, the scikit defaults to using decision trees as an estimator.



Out of Bag Evaluation

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a Bagging Classifier samples *m* training instances with replacement (bootstrap=True), where *m* is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor (as m grows, the ratio approached 1 - exp(-1) = 63.212%) The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation. (as done here)

Gradient Boosting

Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

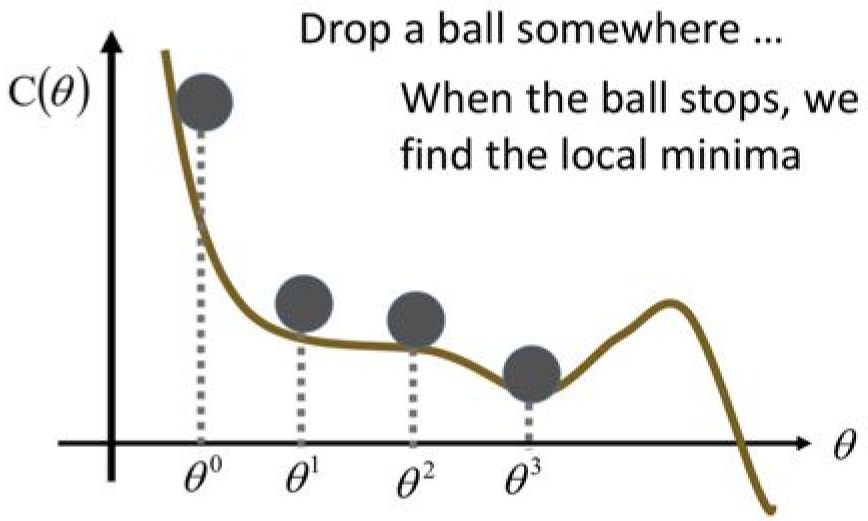
Stochastic Gradient Descent (SGD)

Gradient Descent is a very generic optimisation algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimise a cost function. Concretely, you start by filling θ with random values (this is called *random initialisation*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum. We tweak a hyper-parameter, learning rate to converge the algorithm to the global minimum.

To begin, we formulate a “**Cost Function**”(or called loss function) to evaluate how bad the parameter set of the neural network is, represented as C(θ). We have to find out the θ^* which minimizes the cost C(θ^*).

$\theta^* = \arg\min C(\theta)$

Imagine that dropping a ball somewhere, we can then get the local minimun where the ball stop moving just as the picture show.



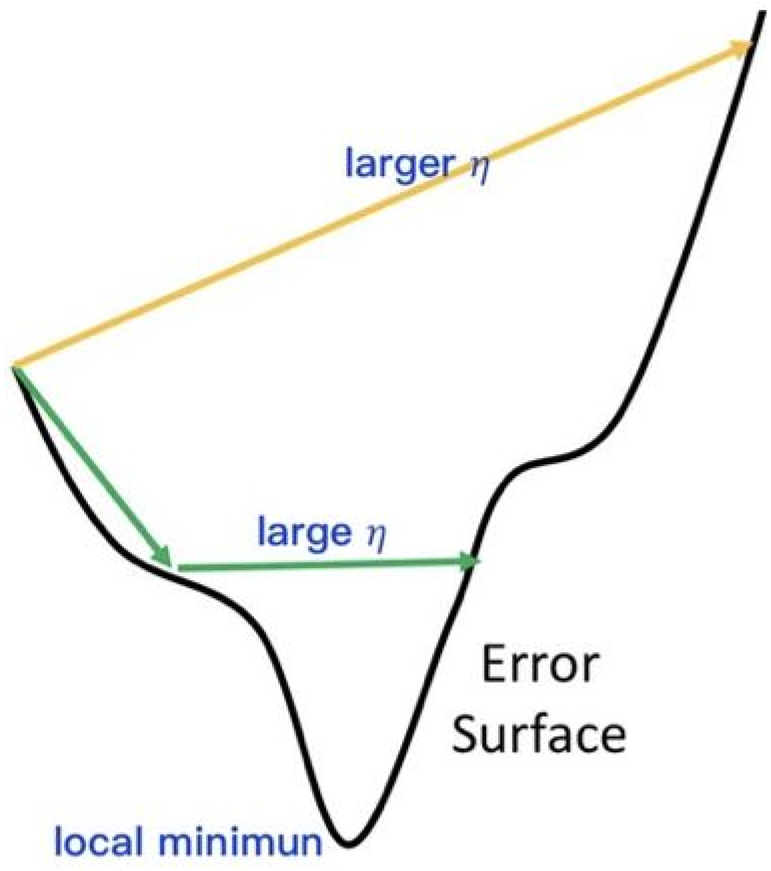
Randomly start at $\theta^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$

Compute the gradients of $C(\theta)$ at θ^0: $\nabla C(\theta^0) = \begin{bmatrix} \partial C(\theta_1^0)/\partial\theta_1 \\ \partial C(\theta_2^0)/\partial\theta_2 \end{bmatrix}$

Update parameters

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \partial C(\theta_1^0)/\partial\theta_1 \\ \partial C(\theta_2^0)/\partial\theta_2 \end{bmatrix} \Rightarrow \theta^1 = \theta^0 - \eta \nabla C(\theta^0)$$

Compute the gradients of $C(\theta)$ at θ^1: $\nabla C(\theta^1) = \begin{bmatrix} \partial C(\theta_1^1)/\partial\theta_1 \\ \partial C(\theta_2^1)/\partial\theta_2 \end{bmatrix}$



We repeat the calculations of gradient at different places until the gradient approaches to zero.

Random Forest and Extra Trees

It uses a decision Tree classifier but is more dense than this. Decision Trees are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multi-output tasks. They are very powerful algorithms, capable of fitting complex datasets. It starts at root node (0) and grows to many leaf nodes (1 or greater) with only 2 leaf/child nodes per node.

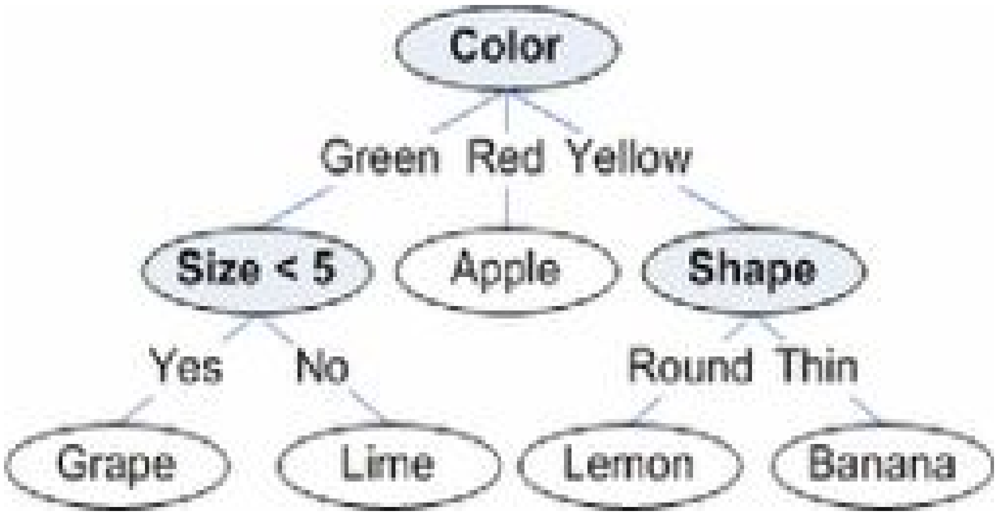
One of the many qualities of Decision Trees is that they require very little data preparation. In particular, they don’t require feature scaling or centering at all. It uses GINI measure to calculate the accuracy of a particular node. GINI equation is:

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}{}^2$$

- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

Scikit-Learn uses the CART (Classification and Regression and Tree) algorithm, which produces only *binary trees*: non-leaf nodes always have two children (that is, questions only have yes/no answers). However, other algorithms such as ID3 (Iterative Dichotomiser 3) can produce Decision Trees with nodes that have more than two children.

An example of a decision Trees is given below



The Random Forest algorithm introduces extra randomness when growing trees; instead of searching for the best feature when splitting a node, it searches for the best feature among a random subset of features. This results in a greater tree diversity, which (once again) trades a higher bias for a lower variance, generally yielding an overall better model.

Extra Trees Classifier: It performs much better than either of random Trees Classifier and Decision Trees Classifier. The Extra-Tree method (standing for **Ext**remely **Ra**ndomised **Tree**s) was proposed, with the main objective of further randomising tree building in the context of numerical input features, where the choice of the optimal cut-point is responsible for a large proportion of the variance of the induced tree.

With respect to random forests, the method drops the idea of using bootstrap copies of the learning sample, and instead of trying to find an optimal cut-point for each one of the K randomly chosen features at each node, it selects a cut-point at random.

Logistic Regression

Logistic Regression (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labeled "0"). This makes it a binary classifier.

A Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result. The logistic—also called the *logit*, noted *σ*(·)—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1. The model estimated probability in vectorized form: is:

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma\left(\theta^T \cdot \mathbf{x}\right)$$

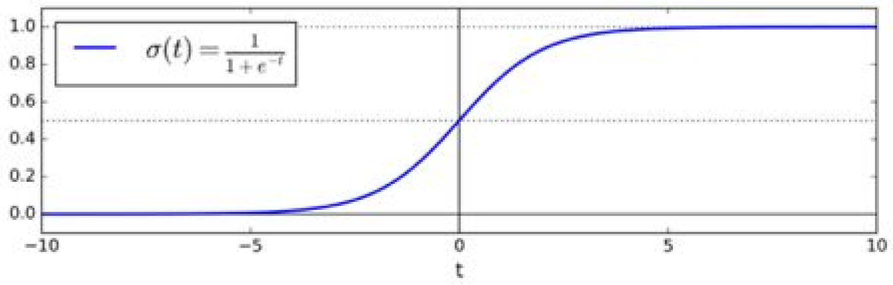
σ: Sigmoid function

Ɵ: The input weights. The transpose of weight matrix is multiplied by inputs

x: The inputs.

hƟ: The whole combined function with x as inputs

The sigmoid function is used because it squashes the parameter to lie between 0 and 1 and therefore helps to give class prediction. The sigmoid function is shown below.



Support Vector Machines

An SVM is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. SVMs are particularly well suited for classification of complex but small or medium-sized datasets.

Linear SVM has no method to predict class probabilities and therefore cannot be used for Ensemble Learning. SVC (Single Vector Clustering) though can be used for Ensemble. Both of them have been implemented in the project.

k-Nearest Neighbors

Linear SVM has no method to predict class probabilities and therefore cannot be used for Ensemble Learning. SVC (Single Vector Clustering) though can be used for Ensemble. Both of them have been implemented in the project.

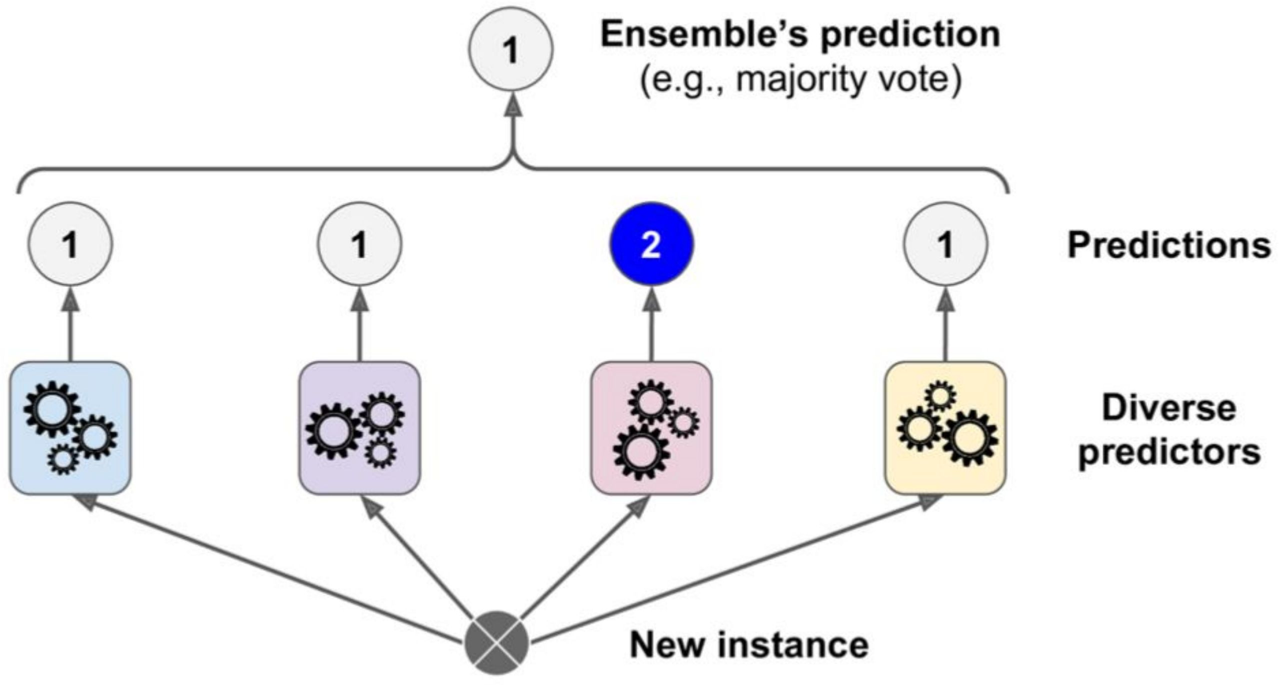
k-Nearest Neighbors:In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression:

- In k-NN classification, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.
- In k-NN regression, the output is the property value for the object. This value is the average of the values of its k nearest neighbors.

This algorithm can be modified to do multi label and multi output classification. There are many ways to evaluate a multi label classifier, and selecting the right metric really depends on your project. For example, one approach is to measure the F1 score for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score.

Ensemble Learning

If you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and Ensemble Learning algorithm is called an *Ensemble method*. A picture for reference of ensemble is attached.



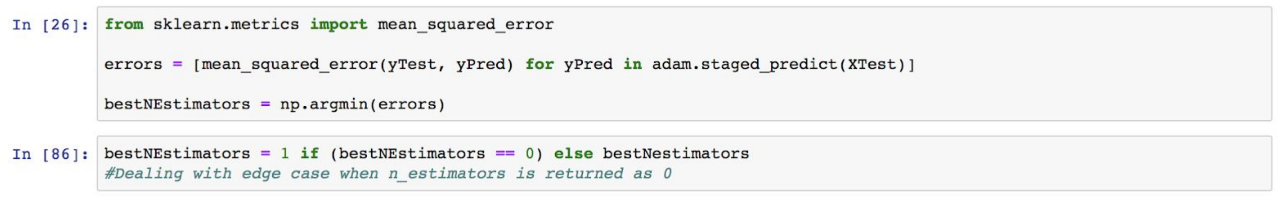
Methodology - Algorithms and Techniques - The Implementation

Imputing: This process to deal with the missing data has been used with two separate pipelines. One using the mode and the other for median. Here, I was in dilemma which one to use. I went with mode since the ecg signal tends to repeat itself at regular intervals (the QRS complex will be repeated in 1s approximately)

Scaling: It was required since I am using scale sensitive algorithm like k-Nearest Neighbors. It, using default values, standardize features by removing the mean and scaling to unit variance.

Splitting: I wanted to use Stratified Splitting to better represent the data but the way this data is structured, it cannot be used :(I have documented this in the notebook itself.

Boosting: It was confusing to decide which Boosting algorithm to select. Finally I went with AdaBoost and Gradient Boosting. The AdaBoost is affected by the number of estimator being used. I have tried to select the best estimator using this code snippet:



```
In [26]: from sklearn.metrics import mean_squared_error

         errors = [mean_squared_error(yTest, yPred) for yPred in adam.staged_predict(XTest)]

         bestNEstimators = np.argmin(errors)

In [86]: bestNEstimators = 1 if (bestNEstimators == 0) else bestNestimators
         #Dealing with edge case when n_estimators is returned as 0
```

The algorithm I am using is SAMME which is used for classification tasks (For regression we have SAMME.R). Same thing has been done for Gradient Boosting.

Bagging and Out-of-Bag Evaluation: Then, I have used Bagging to predict the values and see how it is developing. I have used oob to check its accuracy.

Random Forest and Extra Trees: Here, I am taking warm-start as True to make sure it stops early and get the best possible accuracy.

SGD: The I modified the loss metric as modified huber loss to make sure that it can be easily used with Ensemble. The default loss was hinge which is not supported by Ensemble.

Logistic Regression: The Tolerance is taken as 0.0001 means that the algorithm stops when it comes close the value, essentially when it lies between [value - 0.0001, value + 0.0001]. All default values suffice here and is shown in the notebook.

SVM: We are trying here to fit a 2 degree polynomial here. A higher degree can lead to overfitting whereas a low degree can underfit. 2 degree seems to work the best.

Linear SVC: It works great!, even better than ensemble. Unfortunately it cannot be used in our ensemble since it does not outputs class probabilities :(

KNN: The scaling was essential for this to work. A brute force approach is used. It slows down the processing and it takes time but it gives the best result. 3 neighbors seems to give the best results here. Distance as weights gives overfitting here. Therefore using Euclidean distance and not Minkowski Distance metric. Brute Force is used because the data is sparse.

Ensemble: I have applied equal weightage to each algorithm and am using soft voting to better give the results.

Hyperparameter Tuning - Grid Search CV

I first made a model without hyperparameter tuning which was the intermediate model. Then I did hyperparameter tuning to make the final model. Both of these models are documented in the notebook. The Final model lies at the end whereas intermediate is in the middle.

Table for GridSearchCV

Algorithm	Parameter-Grid (Dictionary)	Best Parameters

Random Forest	param_grid = { 'n_estimators': [200, 300], 'max_depth' : [4,5,6,7,8], }	{'max_depth': 6, 'n_estimators': 300}
Logistic Regression	param_grid = { 'C': [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0], }	{'C': 0.01}
Linear SVC	param_grid = {'SVC__C':np.arange(0.01,100, 10)}	{'SVC__C': 10.01}
K Neighbors	param_grid = {'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]}	{'n_neighbors': 3}
Extra Trees	param_grid = { 'n_estimators': [200, 300], 'max_depth' : [4,5,6,7,8], }	{'max_depth': 7, 'n_estimators': 200}

AdaBoost and Gradient Boosting are also fine tuned without using GridSearchCV. They are documented in the notebook at the middle.

k-Fold Validation

Used 12, 10 and 5 fold Cross Validation here. The code snippet is at the very end of jupyter notebook just before the graphs visualization. Please see that. The Variance is really low in all of them (<0.05) which makes sure that our model is robust.

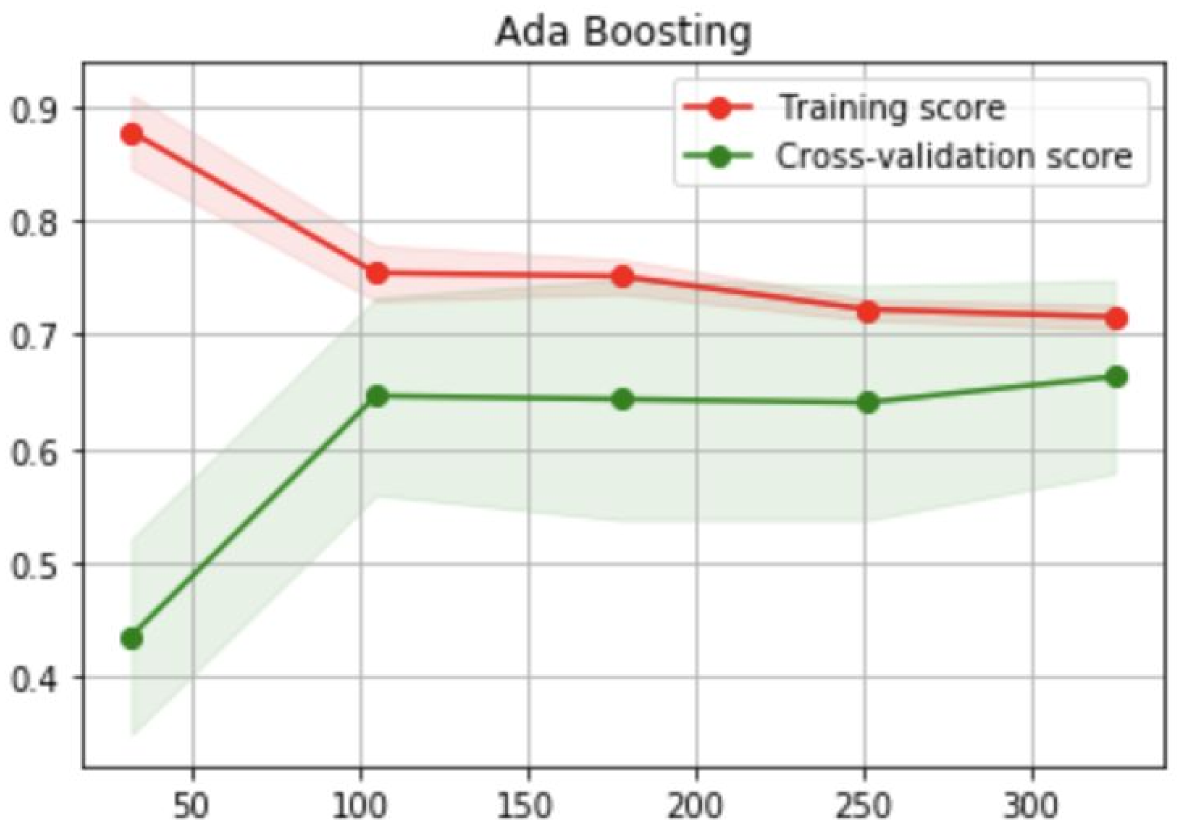
Algorithm	**12 Cross-Validated Accuracy (Variance)**	**10 Cross-Validated Accuracy (Variance)**	**5 Cross-Validated Accuracy (Variance)**
Ada Boost	1.00 (+/- 0.00)	1.00 (+/- 0.00)	1.00 (+/- 0.00)
Gradient Boosting	0.85 (+/- 0.02)	0.85 (+/- 0.01)	0.85 (+/- 0.01)
Stochastic Gradient Descent	0.88 (+/- 0.05)	0.89 (+/- 0.04)	0.87 (+/- 0.03)
Random Forest	0.94 (+/- 0.04)	0.95 (+/- 0.03)	0.95 (+/- 0.02)
Logistic Regression	0.88 (+/- 0.06)	0.89 (+/- 0.03)	0.88 (+/- 0.01)
SVC	0.90 (+/- 0.05)	0.89 (+/- 0.03)	0.89 (+/- 0.03)
Linear SVC	1.00 (+/- 0.01)	1.00 (+/- 0.01)	0.99 (+/- 0.01)
k-Nearest Neighbors	0.86 (+/- 0.04)	0.87 (+/- 0.04)	0.86 (+/- 0.03)
Extra Trees	0.97 (+/- 0.03)	0.97 (+/- 0.03)	0.98 (+/- 0.02)
Ensemble	0.94 (+/- 0.03)	0.93 (+/- 0.04)	0.92 (+/- 0.04)

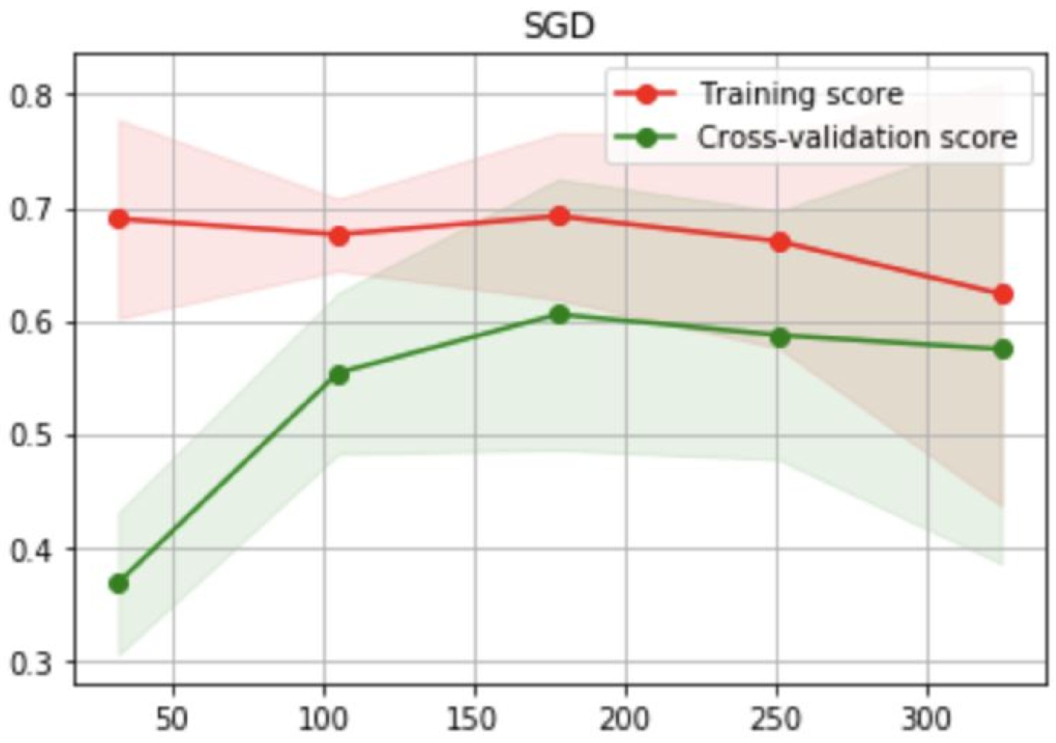
Result

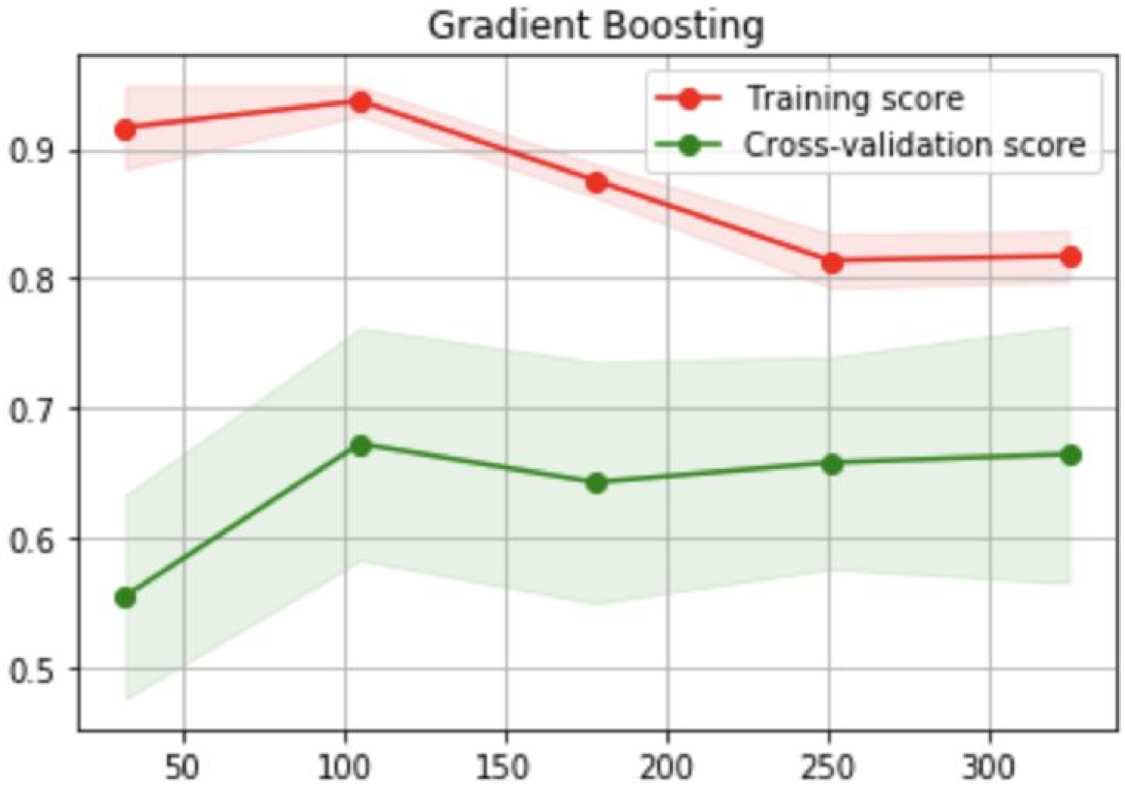
By seeing the cross validation score above, the best algorithm is Linear SVC with 100% accuracy and it even beats the ensemble (which has 94% accuracy). The benchmark model had the highest accuracy of 86% with SVC. My model gives an accuracy of 90% with SVC. I guess the little change of 4% is being caused due to different hyperparameters. Overall, the other algorithms that I have used performs much better than even SVC like LinearSVC or AdaBoosting or Extra Trees.

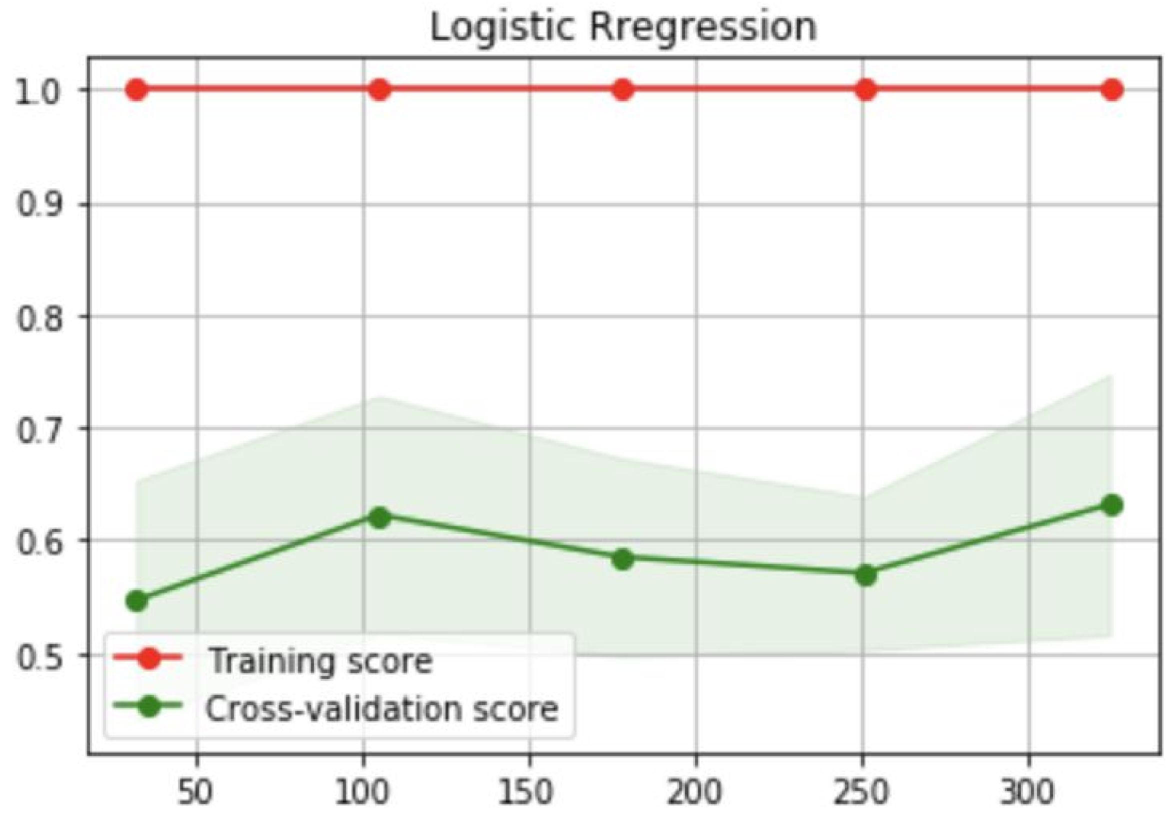
Conclusion

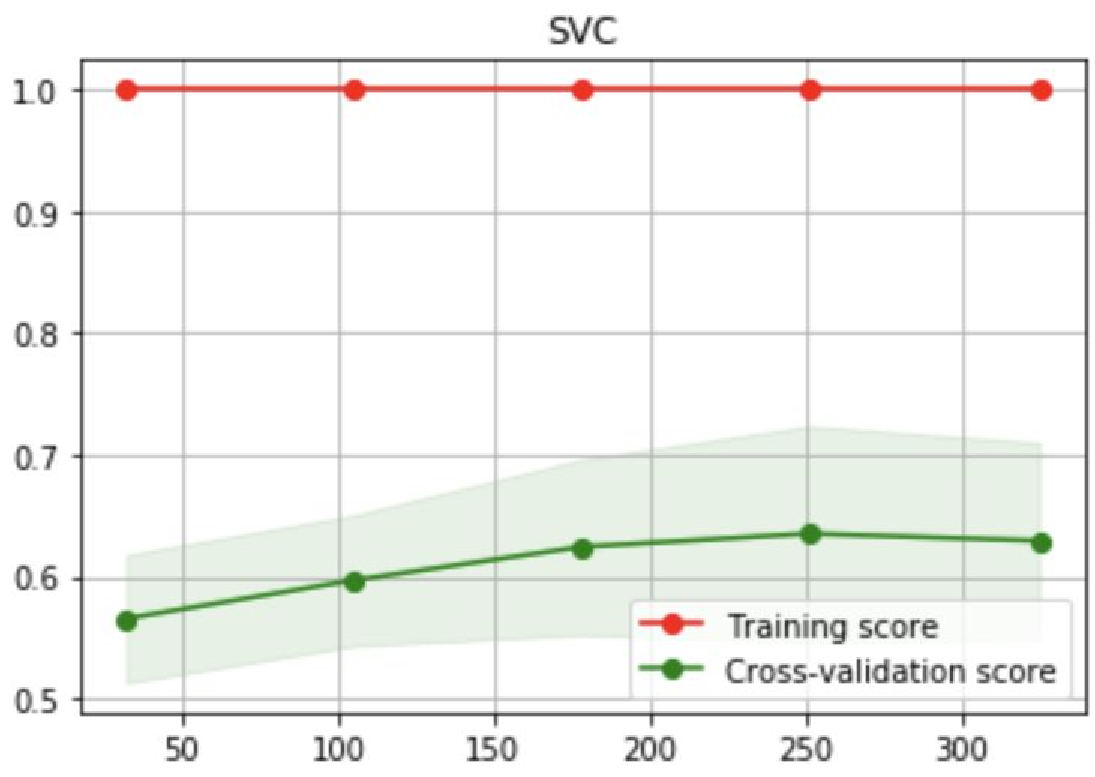
Visualization of the Results (Some of the overfit on the train set but by manually tuning the hyperparameters, I found that these values give best results in the cross validation score)

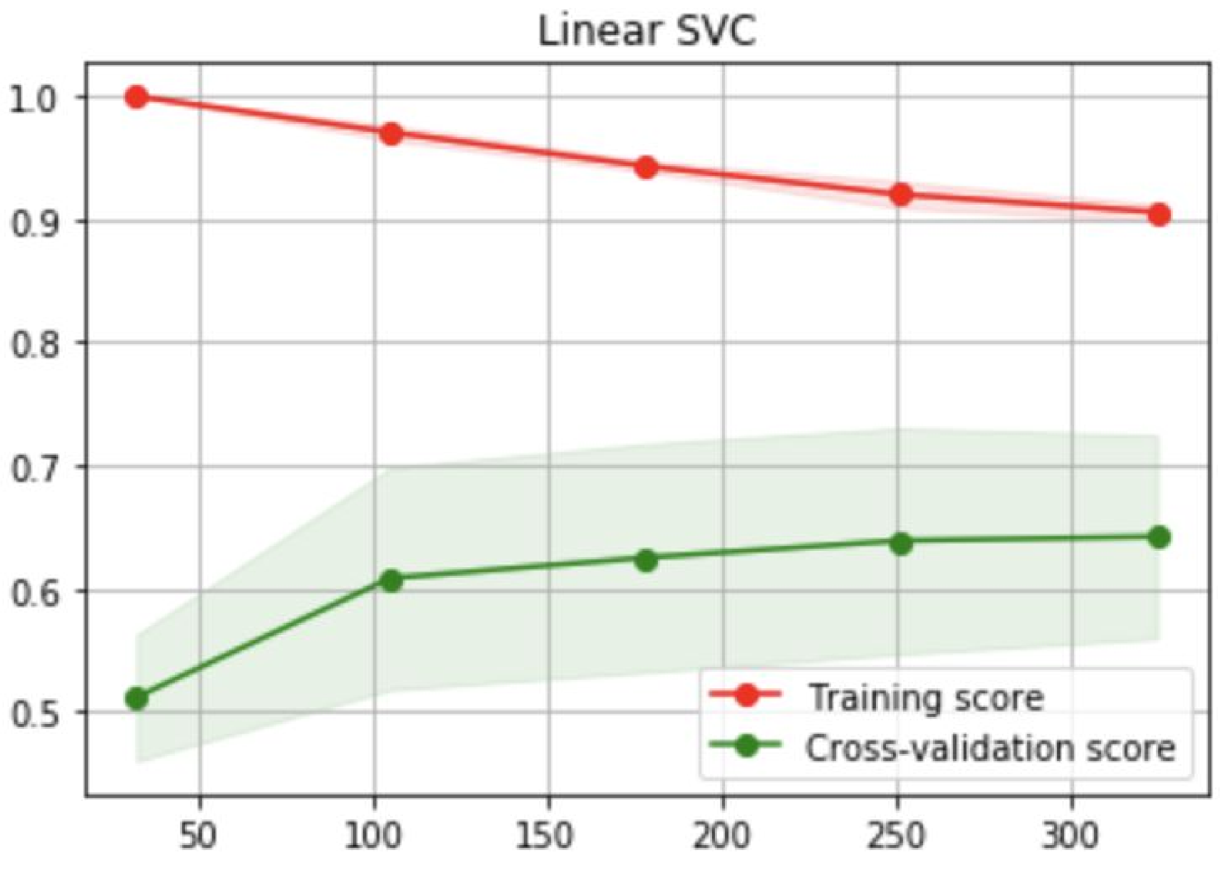

Ada Boosting
Training score
Cross-validation score
0.9
0.8
0.7
0.6
0.5
0.4
50
100
150
200
250
300

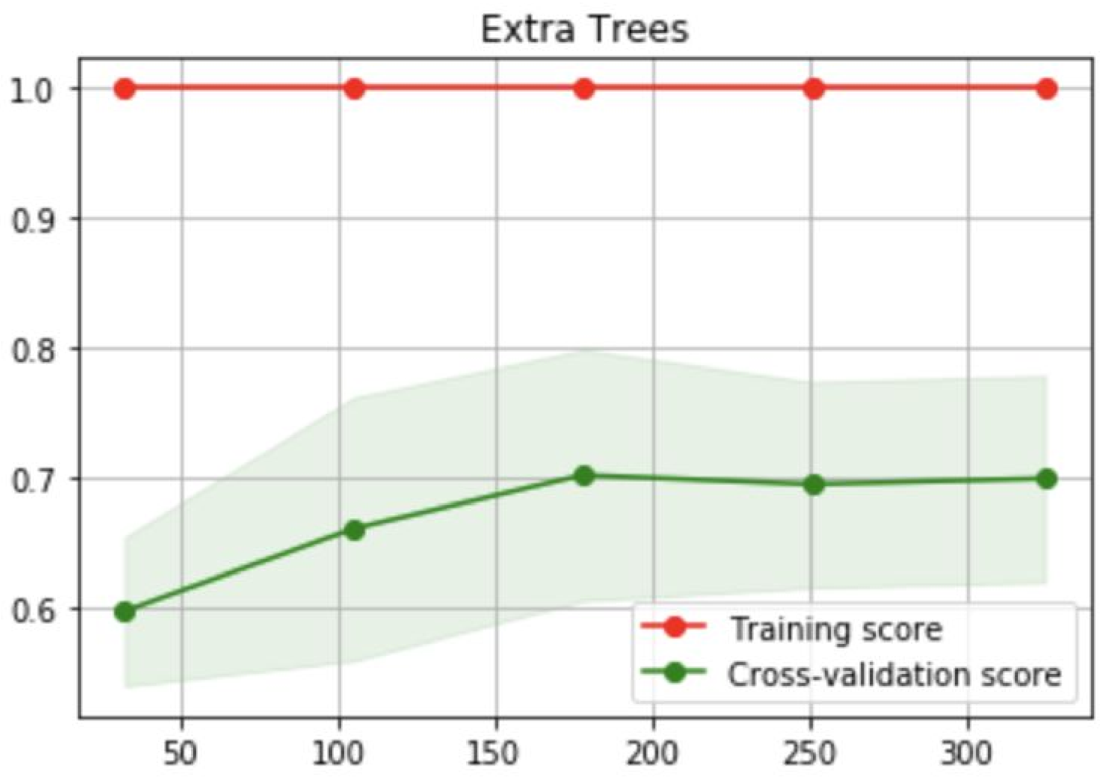

SGD
Training score
Cross-validation score
0.8
0.7
0.6
0.5
0.4
0.3
50
100
150
200
250
300

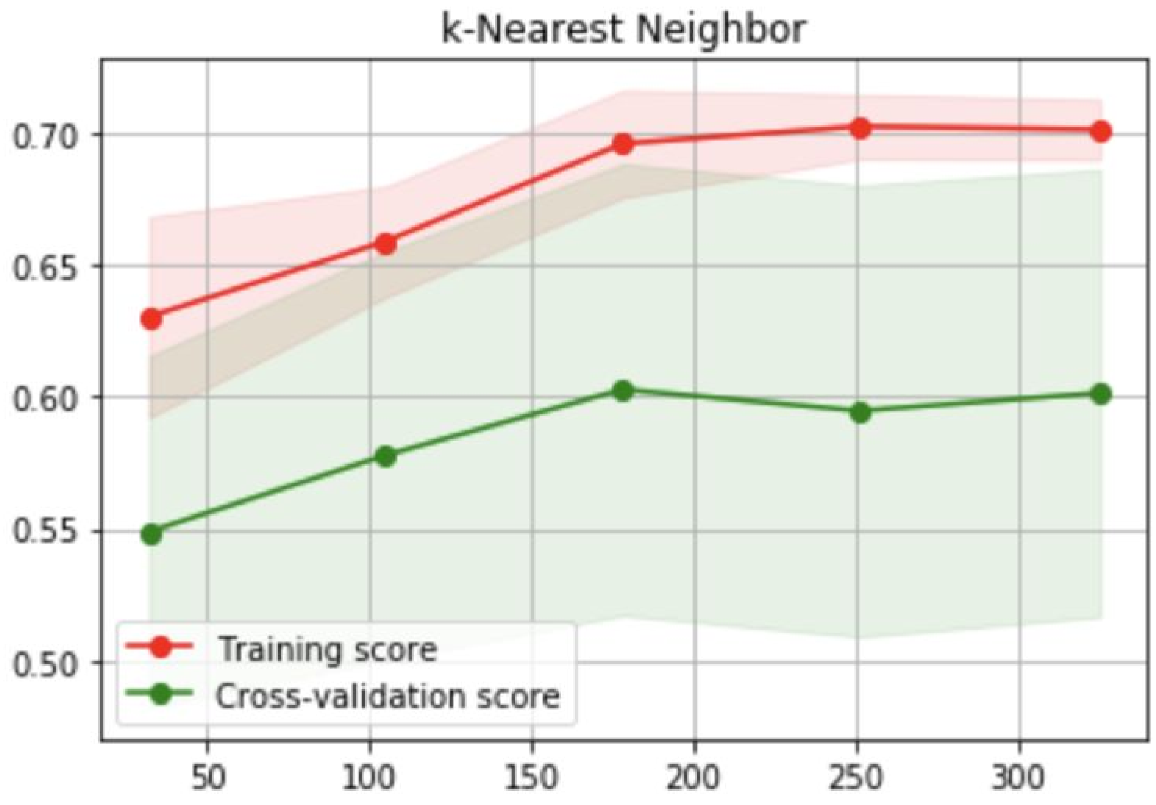

Gradient Boosting
Training score
Cross-validation score
0.9
0.8
0.7
0.6
0.5
50
100
150
200
250
300

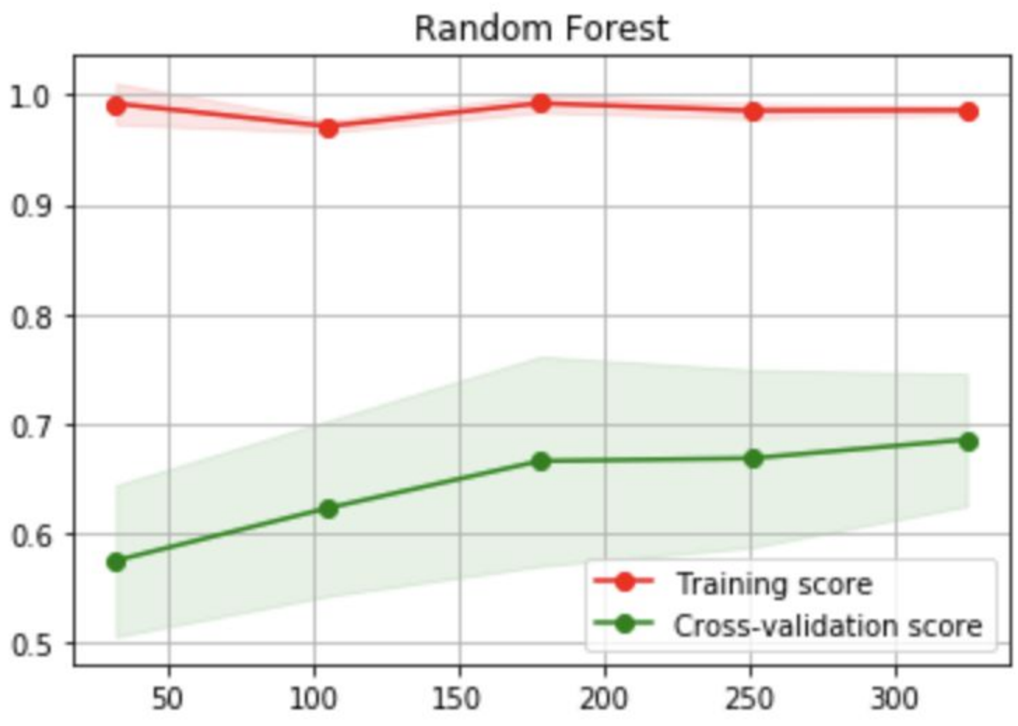

Logistic Rregression
1.0
0.9
0.8
0.7
0.6
0.5
50
100
150
200
250
300
Training score
Cross-validation score


SVC
1.0
0.9
0.8
0.7
0.6
0.5
50
100
150
200
250
300
Training score
Cross-validation score


Linear SVC
1.0
0.9
0.8
0.7
0.6
0.5
50
100
150
200
250
300
Training score
Cross-validation score


Extra Trees
1.0
0.9
0.8
0.7
0.6
50
100
150
200
250
300
Training score
Cross-validation score





The result can be enhanced by:

1. More Data: Currently, the data fed is very poor. The top 3 arrhythmias dominate the instances and about 10 other arrhythmias class have less than 5 instances. As such, removing them and only taking, say top 4 or 5 most repeated class will dramatically enhance the results here.

2. Incorporating Linear SVC in Ensemble: Currently, the ensemble learning in scikit learn requires a predict_proba() method (to obtain class probabilities) and thus form a voting classifier. This predict_proba() method only supports modified huber loss and log loss. Linear SVC has only hinge loss and modified hinged loss and thus cannot be incorporated in the ensemble thus made. Using other libraries to workaround this will dramatically improve the accuracy.

3. Using Genetic Algorithms: Genetic Algorithms can randomize the connection of different pipeline with the very pipelines being tweaked to provide the best possible results.

Things I found interesting about the project:

1. I expected ensemble to perform the best. I was pleasantly surprised when extra trees surpassed it and Linear SVC came close to it.

2. That I could not incorporate Linear SVC in the Ensemble. Later, after researching I found out that there are libraries that tweaks this function and provides a workaround thus to be able to incorporate it hence in ensemble now.

Summary - end to end problem solution

I have taken only 2 most frequent arrhythmia type from the dataset and after scaling and imputing it, I boost it with AdaBoost and Gradient Boosting. Then, I have made use of algorithm like SGD, Random Forest, Logistic Regression, SVC and Linear SVC, k Neighbors, Extra Trees and Ensemble to make a very robust classification system. During this process, I have also fine tuned the hyperparameters to ensure best possible results.