CSE 601: Project 3
Fall 2015

Classification of genomic data

Report by
Group #2
Ankit Kapur, 50133149, ankitkap@buffalo.edu
Samved Divekar: 50135204, samvedch@buffalo.edu
Hrishikesh Sathe: 50134055, hsathe@buffalo.edu

Part I : Understanding of the algorithms

Neural Network

The Backpropagation neural network is a multilayered, feedforward neural network. It is one of the simplest and most general methods used for supervised training of multilayered neural networks. Backpropagation works by approximating the nonlinear relationship between the input and the output by adjusting the weight values internally.

Algorithm:

Initialize all weights with small random numbers, typically between -1 and 1
while ((maximum number of iterations < than specified) AND (Error Function is > than specified))
    for every sample in the training set
    Present the sample to the network
    //Feed forward
    for each layer in the network
        for every node in the layer
            1. Calculate the weight sum of the inputs to the node
            2. Add the threshold to the sum
            3. Calculate the activation for the node
        end for
    end for
    //Backpropagation
    for every node in the output layer
        calculate the error signal
    end for
    for all hidden layers
        for every node in the layer
            1. Calculate the node's signal error
            2. Update each node's weight in the network
    end for
    end for
    //Calculate Global Error
    Calculate the Error Function
end while

Implementation:

1. import the required libraries
2. initialize all weights
3. while weights are changing or iterations < maxIterations:
    a. call nnObjFunction - this function computes the value of the - negative log likelihood error function with regularization - given the training data, labels and lambda (regularization huper-parameter)
    b. weights = nnObjFunction.weights

    c. iterations ++
4. After weights are constant. call nnPredict - this function takes predicts the labels for the validation data and the test data as well.

When it works:

1. Works when boundaries to be learnt are non-linear

When it does not work:

1. When number of samples is too low.
2. When hyperparameters are not chosen carefully.

Advantages:

1. NNs require very less formal statistical training
2. Ability to implicitly detect complex nonlinear relationships between dependent and independent variables
3. Availability of multiple training algorithms

Disadvantages:

1. Proneness to overfitting
2. Very sensitive to the hyperparameters

Handling Categorical attributes

To deal with categorical data, we have created a function *categorical_to_numeric* that creates a set of unique values that the variable can take, and makes a dictionary using these unique values. For example if the variable can take values a, b, c, then a map is created in this way 0=a, 1=b, 2=c.
The values for the column are then replaced with the numeric values, and the model is then processed.

Decision Trees

The following section describes decision trees.

Algorithm:

There are many algorithms for implementing Decision Trees. We have used the C4.5 approach as it handles features with continuous values. The algorithm is described below:

For creating the decision tree

1. foreach unused attribute "a" in training data:
    a. Calculate Gini Index/Information Gain
    b. *maxGainAttribute* = attribute with max gain
2. Create a node *parent* that will split the data points on *maxGainAttribute* and mark the attribute as used
3. Go to 1 for each list created after splitting on *maxGainAttribute.* Multiple nodes are created. Add these as children to *parent*.
4. Do this until all attributes are used.

For classifying a data point D:

1. go to *root* of decision tree
2. Let *splitAttribute* be the attribute on which root splits
3. Go to 1 with *root* = appropriate child based on the value of *D.attributes[splitAttribute]*
4. repeat 1-3 until we reach leaf node
5. *D.label* = label at leaf node

Implementation:

```
createDecisionTree(Node root, List attributes) {
        dataPoints = root.getDataPoints
        minimumImpurity = Double.maxValue
        foreach unused attribute attr in attribtes:
                temp  = calculateGiniIndex(attr) or calculateEntropy(attr)
                if(temp < minimumImpurity) {
                        minImpAttr = attr;
                        minimumImpurity = Min(minimumImpurity, temp)
                }

        attributes.minImpAttr = used
        Let dataPointsLists = lists created when split on minImpAttr
        foreach List of datapoints in dataPointsLists:
                create child node c with datapoints
                maxMajority = max majority of a class in datapoints
                c.majority = maxMajority
        let minMajorityChild = child with minimum majority
        createDecisionTree(minMajorityChild, attributes)
        foreach remaining child c:
                createDecisionTree(c, attributes)
}

classifyDataPoint(Node root, DataPoint d) {
        label = null
        if(root is a leaf node) {
                label = root.label
        }
        attributeToSplit = root.getAttributeToSplit
        children = root.getChildren
        child_to_visit = appropriate child based on value of d.attributes[attributeToSplit]
        label = classifyDataPoints(child_to_visit, d)
        return label
}
```

When it works:

1. Decision trees work well with low number of training samples

When it doesn't work:

1. Decision trees don't work well when there are small perturbations in the data

Advantages:

1. Decision trees are easy to interpret.
2. Works well on numerical as well as categorical data.
3. Faster compared to other algorithms like ANN as less number of calculations
4. Low number of parameters to tune.
5. As they are easy to interpret, they can be used to generate Rules in the Rule Generation problem.
6. Handles noisy or incomplete data

Disadvantages:

1. Its accuracy is lower when compared to other algorithms like ANN
2. If there are small perturbations in continuous data, it can create a completely different tree.
3. It uses the greedy strategy so finding the right root is hard and an incorrect root can change results.

Handling Categorical and continuous attributes

Handling categorical data is easy in decision trees. Example:
Attribute x has categorical data with distinct values - A, B, C. When we split on x, we can have each child with distinct categorical data i.e. child 1 would have data where x = A, child 2 would have data where x = B and child 3 would have data where x = C.

For handling continuous data we are using the **Uniform approximation** method explained in "Efficient Determination of Dynamic Split Points in a Decision Tree" [1]. More about this in the other work section.

Naïve Bayes

How it works:

Naive bayes classifiers is a simple probabilistic classifier based on Bayes Rule of conditional probability. It is a type statistical classifier. It estimates the probability of a specific sample belonging to a specific class and assigns the class with highest posterior probability.

Algorithm NaiveBayes:

foreach Datapoint X in TestData
for each class Ci
Calculate the class posterior probability P(Hi | X) using following formula.

$$P(H_i \mid X) = \frac{P(H_i)\,P(X \mid H_i)}{P(X)}$$

where

P(Hi) = Class prior probability
P(X | Hi) = Descriptor posterior probability
P(X) = Descriptor Prior probability
end for
Assign X to the class Ci with highest P(Hi | X)
end for

Our implementation:

1. Parse the training data and the test data.
2. Calculate the class prior probabilities P(Hi) for each class
3. For each Data sample X in Test Data:
   for each class Ci:
   for each feature x in X:
   if feature is continuous
   calculate P(x| Hi) using probability distribution function(Assuming normal distribution of data)
   else if feature is categorical
   calculate P(x| Hi) by counting occurrences of x in given class
   end for
   Calculate Descriptor posterior probability P(X | Hi) by multiplying individual P(x| Hi)
   Calculate P(Hi) * P(X | Hi) for class Ci
   end for
   Assign X to the class with highest P(Hi) * P(X | Hi) value
   end for
4. END

When it works:

1. Whenever the assumption of conditional independence between features holds, Naive Bayes classifier works well. It also converges very fast in this case.
2. Works better than discriminative approaches when training data is less

When is does not work:

1. When assumption of conditional independence between features does not hold and features are very much correlated, then Naive Bayes classifier suffers.

Advantages:

1. Very simple to implement
2. The model can be modified with new training data without having to rebuild the model
3. Converges very fast when assumption of conditional independence between features holds
4. Requires very less data for training

Disadvantages:

1. 1.If no occurrences of a class label and a certain attribute value together then the frequency-based probability estimate will be zero. This could cause problems. (Though it can be corrected using Laplacian correction)

2. If normal distribution of data in features does not hold, then Gaussian NB will suffer.

Handling Categorical and continuous attributes for Naive Bayes

Handling Categorical attributes in Naive Bayes is easier, since the individual feature x's probability can be calculated by the formula given below.

$$P(xj|Hi)=ni,j/ni,$$

where ni,j is number of training examples in class Ci having value xj for attribute Aj, ni is number of training examples in Ci.
This is straight-forward, easy to implement approach hence we have used it.

For handling continuous attributes, we have assumed normal(Gaussian) distribution of data within the given class. There were two additional approaches for handling continuous data, viz. Kernel method and data discretization. Though other methods outperform normal method in terms of accuracy, we have chosen normal method because it is intuitive, easier to implement and also performs faster than other methods. (Our conclusion is based on the results mentioned in a research paper named "naive bayes classifiers that perform well with continuous variables" by Dr.Remco R. Bouckaert. Further discussion regarding this can be found in **other considerations** section.)

Random Forests

The following section describes random forests

Algorithm Random Forests:

1. Select number of Trees - T
2. Select number of features to randomly select at each node - m
3. Create T datasets from the training data by choosing data at random. We have chosen 80% of training data for dataset
4. create T decision trees by passing dataset and m features randomly selected. (Slight modification in createDecisionTree algorithm - choose m features randomly at each node of the tree).

Classifying a new sample S:

1. Pass S to T decision trees and get back labels
2. L = label with max majority among T labels which were returned from T trees
3. return L i.e. S is classified with L

Implementation:

```
randomForest() {
        treeList = list of trees
        foreach dataset in T datasets:
                root = new Node(dataset)
                tree = createDecisionTree(root, randomized(attributes))
                add tree to treeList
}

randomized(attributes) {
        return m random attributes
```

```
}

classifyUsingRandomForest(T trees, DataPoint d) {
        labels = list of labels
        for tree in T trees:
                labels[i] = classifyDataPoint(tree.root, d)

        L = label with maximum majority from labels
        return L
}
createDecisionTree(Node root, List attributes) {
        dataPoints = root.getDataPoints
        minimumImpurity = Double.maxValue
        foreach unused attribute attr in attribtes:
                temp  = calculateGiniIndex(attr) or calculateEntropy(attr)
                if(temp < minimumImpurity) {
                        minImpAttr = attr;
                        minimumImpurity = Min(minimumImpurity, temp)
                }

        attributes.minImpAttr = used
        Let dataPointsLists = lists created when split on minImpAttr
        foreach List of datapoints in dataPointsLists:
                create child node c with datapoints
                maxMajority = max majority of a class in datapoints
                c.majority = maxMajority

        let minMajorityChild = child with minimum majority
        createDecisionTree(minMajorityChild, randomized(attributes))
        foreach remaining child c:
                createDecisionTree(c, randomized(attributes))
        // after recursion
        return root
}

classifyDataPoint(Node root, DataPoint d) {
        label = null
        if(root is a leaf node) {
                label = root.label
        }
        attributeToSplit = root.getAttributeToSplit
        children = root.getChildren
        child_to_visit = appropriate child based on value of d.attributes[attributeToSplit]
        label = classifyDataPoints(child_to_visit, d)
        return label
}
```

When it works:

1. It works even when number of samples are low.
2. We can also run the random forest algorithm on distributed systems i.e. each system on the network creates a tree for the forest.

When it doesn't work:

1. It does not work well if the number of features are very low. This is because, if there are suppose 10 features and we select 20% i.e. 2 features then it doesn't work well.

Advantages:

1. Random forests handle large number of predictors very efficiently
2. Faster to train than SVMs and Neural Networks
3. More interpretable because individual trees can be visualized

Disadvantages:

1. A large number of trees may make the algorithm slow for real-time prediction.

Part II : Result Analysis

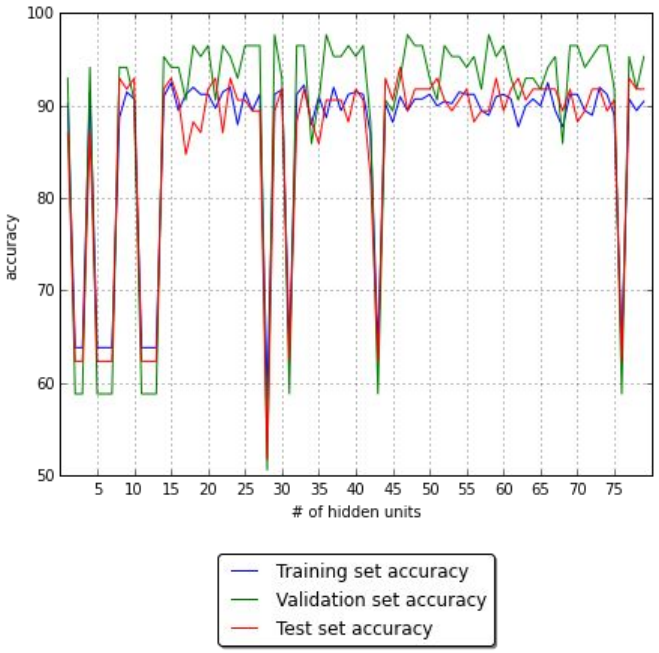
Neural Network

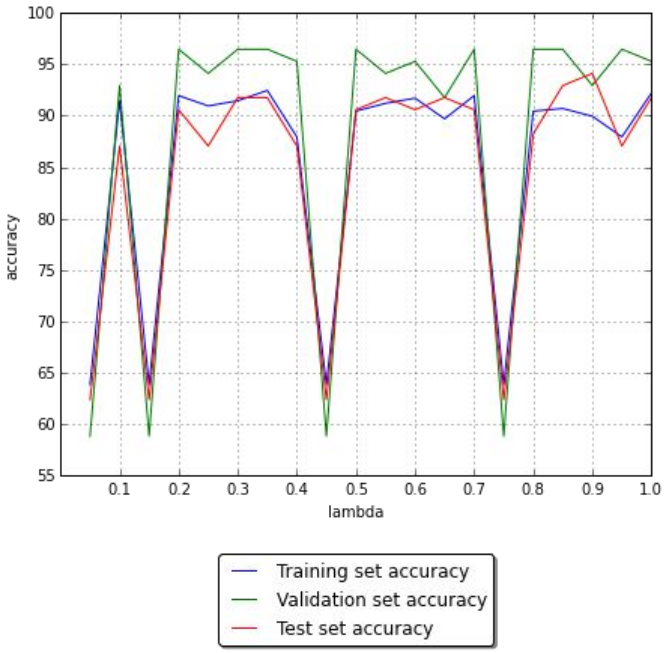
Dataset 1

Results of training the 1st dataset with the neural network model are presented below. In order to find the most optimum values of lambda and hidden_units, the values were varied over a range and the ones with the highest accuracy were chosen as the most optimum ones.

Lambda was varied from 0.05 to 1, in increments of 0.05
Number of hidden units was varied from 1 to 80, in increments of 1

Optimum # of hidden units = **46**
Optimum lambda value = **0.9**
Execution time: 0.496 seconds.
Accuracy on the test dataset = **94.12%**





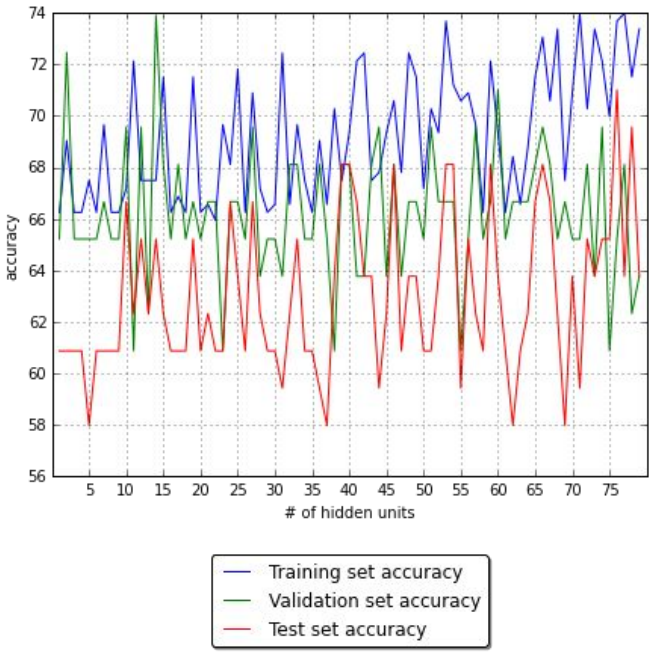
10-fold cross validation results:
Precision = 0.897102263468
Recall = 0.876222857813
Accuracy = 0.896470588235
F-measure = 0.886325928179

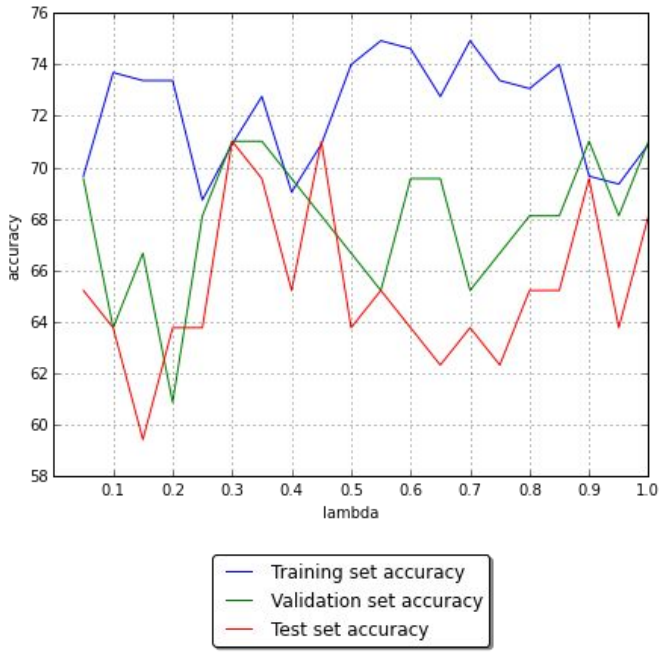
Dataset 2

Results of training the 2nd dataset with the neural network model. In order to find the most optimum values of lambda and hidden_units, the values were varied over a range and the ones with the highest accuracy were chosen as the most optimum ones.

Lambda was varied from 0.05 to 1, in increments of 0.05
Number of hidden units was varied from 1 to 80, in increments of 1

Optimum # of hidden units = **76**
Optimum lambda value = **0.3**
Execution complete. Total execution time: 0.152 seconds.
Accuracy on the test dataset = **71.01%**





10-fold cross validation results:
Precision = 0.566521645573
Recall = 0.547961556292
Accuracy = 0.657971014493
F-measure = 0.552801184895

Decision Trees

We have performed 10-fold cross validation on given datasets.
The tables below show how decision tree performs when we use K split points for each continuous feature.

Using Entropy/Information Gain:

Dataset 1

K	Accuracy	Precision	Recall	F-1 Measure	Time taken (ms)
1	0.924945055	0.927353882	0.912694895	0.919892279	221
2	**0.934395604**	**0.936697483**	**0.924472291**	**0.930450645**	**131**
3	0.934304029	0.935403604	0.924893016	0.93003308	64
4	0.934072802	0.93495701	0.924462159	0.929582344	59
5	0.931175824	0.9310776	0.921893256	0.926350181	72

Dataset 2

K	Accuracy	Precision	Recall	F-1 Measure	Time taken (ms)
1	0.69048913	0.691029612	0.585327005	0.631567672	90
2	0.702400362	0.68931454	0.622828901	0.652469432	25
3	**0.709903382**	**0.693350326**	**0.633882171**	**0.660852579**	**25**
4	0.694723732	0.645034221	0.60411725	0.621366513	24
5	0.686503623	0.632167318	0.586309968	0.604325858	22

Using Gini Index

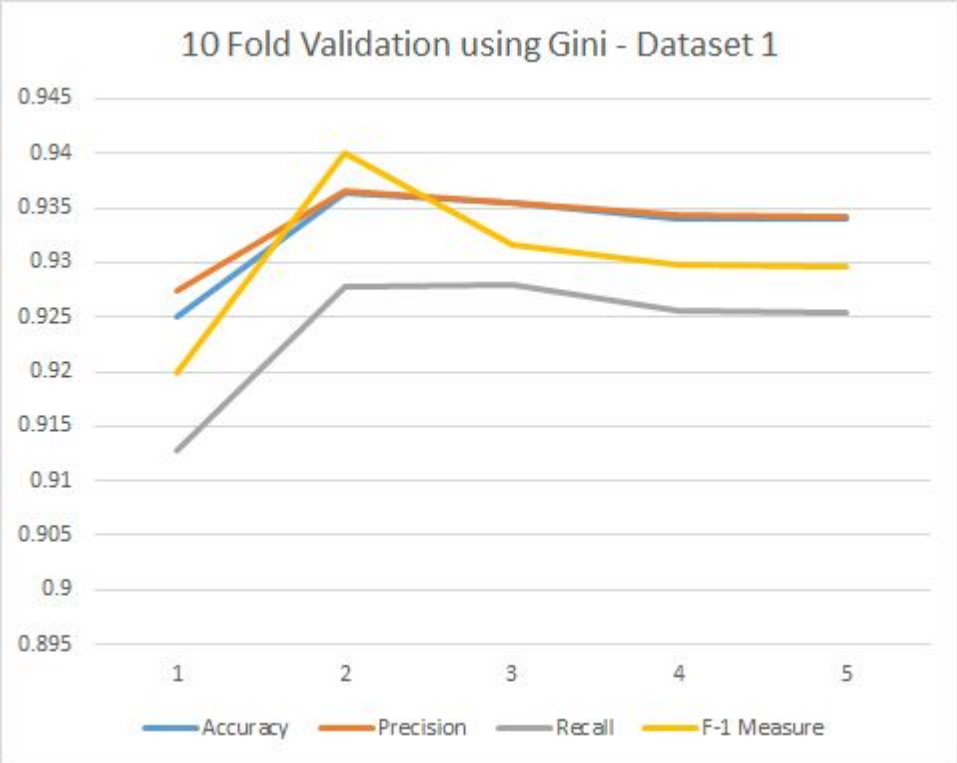
Dataset 1

K	Accuracy	Precision	Recall	F-1 Measure	Time taken
1	0.924945055	0.927353882	0.912694895	0.919892279	216
2	**0.936395604**	**0.936497543**	**0.927706054**	**0.939967715**	**123**
3	0.935494505	0.935558441	0.927886042	0.931602039	65
4	0.934072802	0.934410544	0.925564517	0.929848911	58
5	0.934032967	0.934144335	0.925451162	0.929674824	66

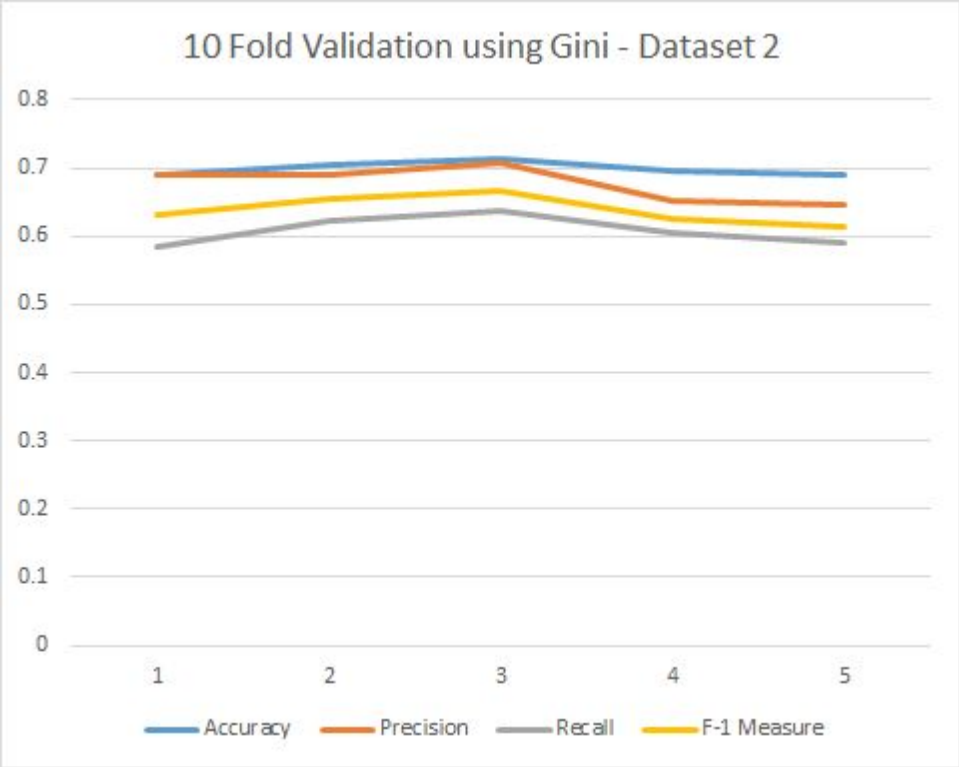
Dataset 2:

K	Accuracy	Precision	Recall	F-1 Measure	Time taken (ms)
1	0.69048913	0.691029612	0.585327005	0.631567672	80
2	0.703487319	0.691024946	0.624019377	0.653904176	25
3	**0.71352657**	**0.708738545**	**0.638379526**	**0.665768188**	**20**
4	0.695810688	0.651279138	0.606438182	0.62599973	22
5	0.689076087	0.644623083	0.590414189	0.612806867	20

Graphs for Gini Impurity:



As we can see from the above graph, the **validation measures are the highest when K = 2** i.e. 2 split points. Hence when the continuous features are split into three intervals we get the highest accuracy for Dataset 1.



As we can see from the above graph, the **validation measures are the highest when K = 3** i.e. 3 split points. Hence when the continuous features are split into four intervals we get the highest accuracy for Dataset 2.

For both the datasets, the Gini index performs slightly better than entropy/ information gain both in terms of accuracy as well as time taken. Entropy is slower because it has logarithmic computations. Even though it is a very small dataset, we can see that Gini is slightly faster than entropy.

Dataset 1:

Impurity Measure	**Accuracy**	**Precision**	**Recall**	**F-1 Measure**	**Time taken(ms)**
Gini	0.936395604	0.936497543	0.927706054	0.939967715	123
Entropy	0.934395604	0.936697483	0.924472291	0.930450645	131

Dataset 2:

Impurity Measure	**Accuracy**	**Precision**	**Recall**	**F-1 Measure**	**Time taken(ms)**
Gini	0.71352657	0.708738545	0.638379526	0.665768188	20
Entropy	0.709903382	0.693350326	0.633882171	0.660852579	25

Naive Bayes

We performed 10-fold cross validation on given data sets and summarized the results. Following tables show the observed results.

For Data set 1:

Following table shows the accuracy, precision, recall and F1-measure for dataset 1 for each fold. The last row of the table shows the average values for each of the performance measure over 10 folds.

	Accuracy	**Precision**	**Recall**	**F1-Measure**
Iteration 1	1	1	0.999999986	0.999999993
Iteration 2	0.910714286	0.894385027	0.919630144	0.906831921
Iteration 3	0.946428571	0.938095238	0.947222209	0.942636631
Iteration 4	0.964285714	0.95	0.973684198	0.9616963
Iteration 5	0.910714286	0.90952381	0.902406403	0.905951128
Iteration 6	0.910714286	0.915483871	0.910714268	0.913092841
Iteration 7	0.839285714	0.895348837	0.795454531	0.842450758
Iteration 8	0.928571429	0.911111111	0.932748526	0.921802863
Iteration 9	0.910714286	0.894025605	0.904970748	0.899464881
Iteration 10	0.984615385	0.987179487	0.981481468	0.984322232
Average	**0.930604396**	**0.929515299**	**0.926831248**	**0.927824955**

For Data set 2:

Following table shows the accuracy, precision, recall and F1-measure for dataset 1 for each fold. The last row of the table shows the average values for each of the performance measure over 10 folds.

	Accuracy	**Precision**	**Recall**	**F1-Measure**
Iteration 1	0.652173913	0.621505376	0.614604449	0.61803565
Iteration 2	0.673913043	0.603225806	0.617647048	0.610351254
Iteration 3	0.717391304	0.658441558	0.636160701	0.647109397
Iteration 4	0.760869565	0.74009324	0.714583319	0.727114602
Iteration 5	0.760869565	0.727678571	0.719354825	0.723492758
Iteration 6	0.630434783	0.555944056	0.553571417	0.5547552
Iteration 7	0.782608696	0.771825397	0.771825382	0.771825389

Iteration 8	0.760869565	0.763403263	0.724206333	0.7432884
Iteration 9	0.717391304	0.70979021	0.678571413	0.693829818
Iteration 10	0.625	0.59375	0.588888877	0.591309448
Average	**0.708152174**	**0.674565748**	**0.661941376**	**0.668111191**

The lower values of performance measures can be attributed to the fact that the data in dataset 1 fits pretty well in a normal distribution than for data set 2. (Since we have assumed normal distribution of data for float values of features.) Following are the plots that demonstrate the distribution of features.

Random Forests

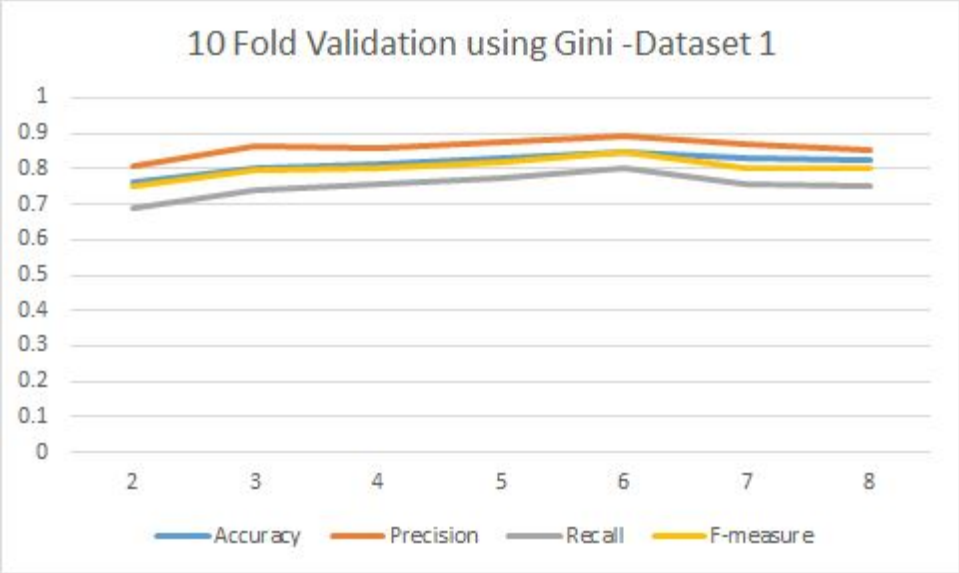
We have performed 10-fold cross validation on given datasets.

Varying number of trees for the datasets:
The tables below show how random forests perform when we use T Trees. We are using 70% of the training data randomly for each subtree and 70% of attributes randomly chosen at each node of the tree when varying the number of trees. We are also using Gini as it performed better than entropy
.

Dataset 1 Using Gini varying number of trees:

Number of Trees	Accuracy	Precision	Recall	F-measure
2	0.763186813	0.804981484	0.689855618	0.752837157
3	0.803887363	0.86245644	0.741314437	0.793709746
4	0.810888278	0.861222603	0.75459786	0.801511364
5	0.828427198	0.873672539	0.776676845	0.81991043
6	**0.84560989**	**0.89002523**	**0.802345065**	**0.846219502**
7	0.828182234	0.869076547	0.754704157	0.801547312
8	0.82639325	0.850252959	0.751475118	0.800192547

Graph for the above table:

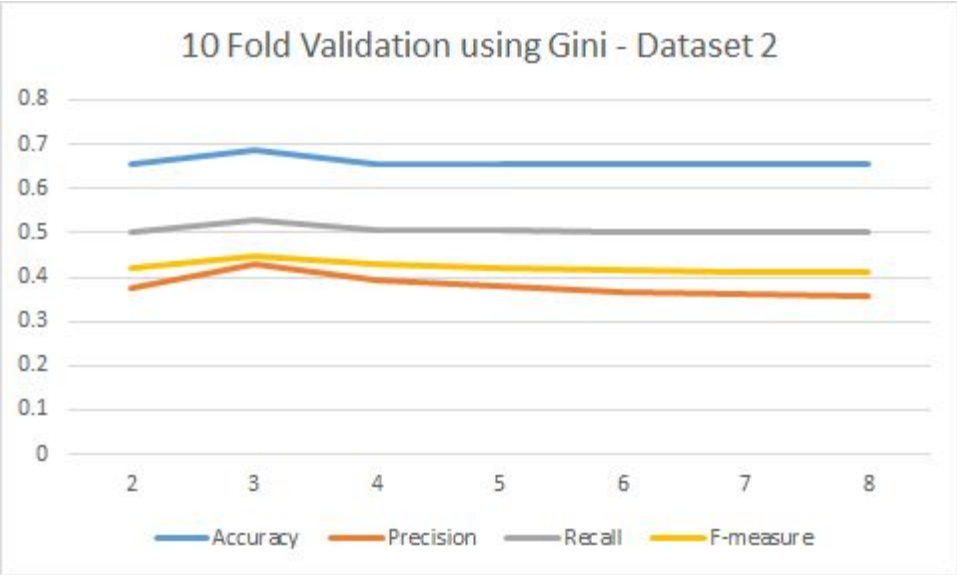


As we can see from the above graph, there is a **slight increase in accuracy when number of trees = 6**. Hence we get the best results for Dataset 1 using gini impurity when number of trees = 6

Dataset 2 using Gini varying number of trees:

Number of Trees	Accuracy	Precision	Recall	F-measure
2	0.655706522	0.377370169	0.502380952	0.421285789
3	**0.688967391**	**0.42869428**	**0.530589599**	**0.447283262**
4	0.657155797	0.394718288	0.506393066	0.429487318
5	0.65625	0.377730292	0.504794799	0.420589346
6	0.655706522	0.367537495	0.50383584	0.415250563
7	0.655344203	0.360742296	0.503196533	0.411691374
8	0.655085404	0.355888583	0.502739885	0.409149097

Graph for the above table:



As we can see from the above graph, the **highest accuracy is when number of trees = 3.** Hence for dataset 2, when number of trees = 3, we get the highest accuracy.

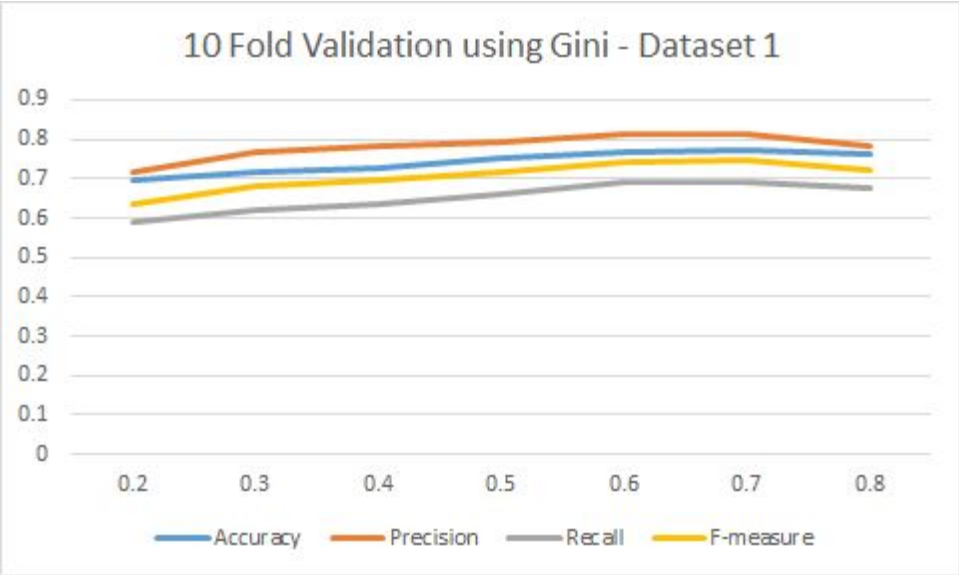
Varying number of attributes to be selected for each node:

We keep the number of trees constant and vary the number of features. The tables and graphs are given below:

Dataset 1 using Gini - Varying number of features:

% of features	Accuracy	Precision	Recall	F-measure
0.2	0.694973	0.716202	0.590389	0.635794
0.3	0.719203	0.767169	0.621753	0.679343
0.4	0.72826	0.780936	0.634749	0.694409
0.5	0.750817	0.795506	0.662207	0.717425
0.6	0.769264	0.814919	0.689448	0.742428
0.7	**0.770746**	**0.815827**	**0.692813**	**0.744818**
0.8	0.760844	0.783586	0.677738	0.720042

Graph for the above table:

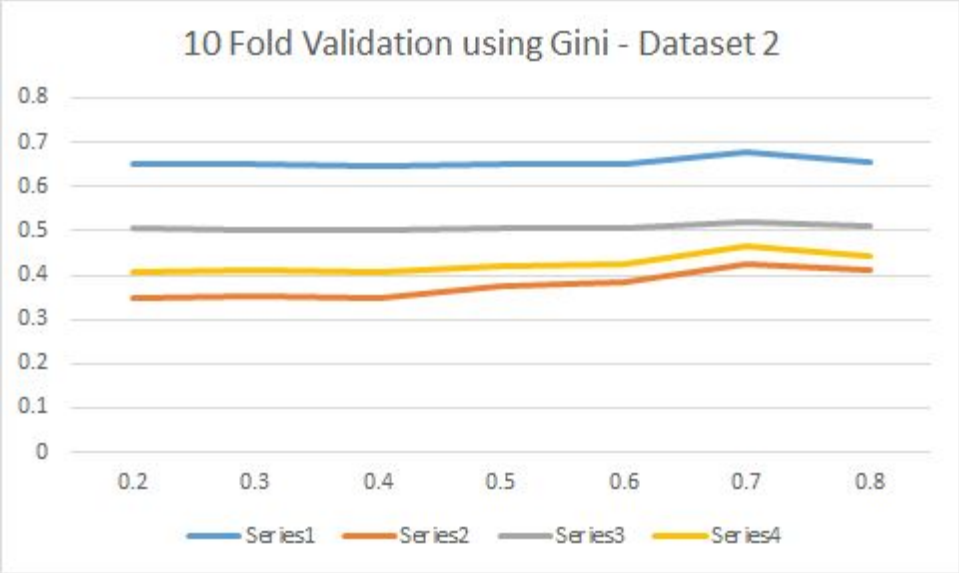


As we can see from the above graph, the **highest accuracy is achieved when number of features selected is 70%** for dataset 1.

Dataset 2 using Gini - Varying number of features:

% of features	Accuracy	Precision	Recall	F-measure
0.2	0.649185	0.348814	0.503958	0.408727
0.3	0.650272	0.350681	0.503157	0.409621
0.4	0.64846	0.347891	0.50284	0.407663
0.5	0.650815	0.374348	0.504379	0.420357
0.6	0.651793	0.386163	0.505776	0.426953
0.7	**0.676069**	**0.425184**	**0.520844**	**0.463976**
0.8	0.655707	0.413393	0.509921	0.442731

Graph for the above table:



As we can see from the above graph, the highest accuracy is achieved when number of features selected is 70% for dataset 2.

Part III: Comparative Analysis

Here, we are comparing all the four models of classifications that we have implemented.

1. Neural Networks
2. Decision Trees
3. Naive Bayes
4. Random Forests

After comparing results for all the four models for all four performance measures:

For Dataset 1:

Decision trees outperform other three methods for optimum value of number of splits(=2)(F1-measure = 0.930450645). Naive Bayes closely follows(F1-measure = 0.927824955), because NB classifier works well even with less number of samples and our assumption that feature values follow a normal distribution also holds true.

For Dataset 2:

Naive Bayes classifier outperforms decision tree algorithm by a very small margin.(f1-measure 0.668111191 vs 0.665768188). Other two algorithms also closely follow the similar results.
However these values are quite less as compared to those for dataset 1. This can be due to, features in dataset 2 do not fit normal distribution curve as those in dataset 1.(For Naive Bayes). Random forests would have performed better, if there were enough number of samples and more number of features.

While evaluating performance, we had some interesting findings in the Decision Tree and Random Forest algorithms.

1. For datasets 1 and 2, decision tree gave best performance for 2 and 3 number of splits respectively.
2. For datasets 1 and 2, random forest algorithm gave best performance for 6 and 3 number of trees respectively.

Part IV : Classification strategies for dataset 3

Approach

Since the # of training samples available to us was extremely small, our first concern was to prevent overfitting of the data. Secondly, with the large number of features - 7004 - another step was feature reduction. We reduced the dimensionality of the data using PCA.

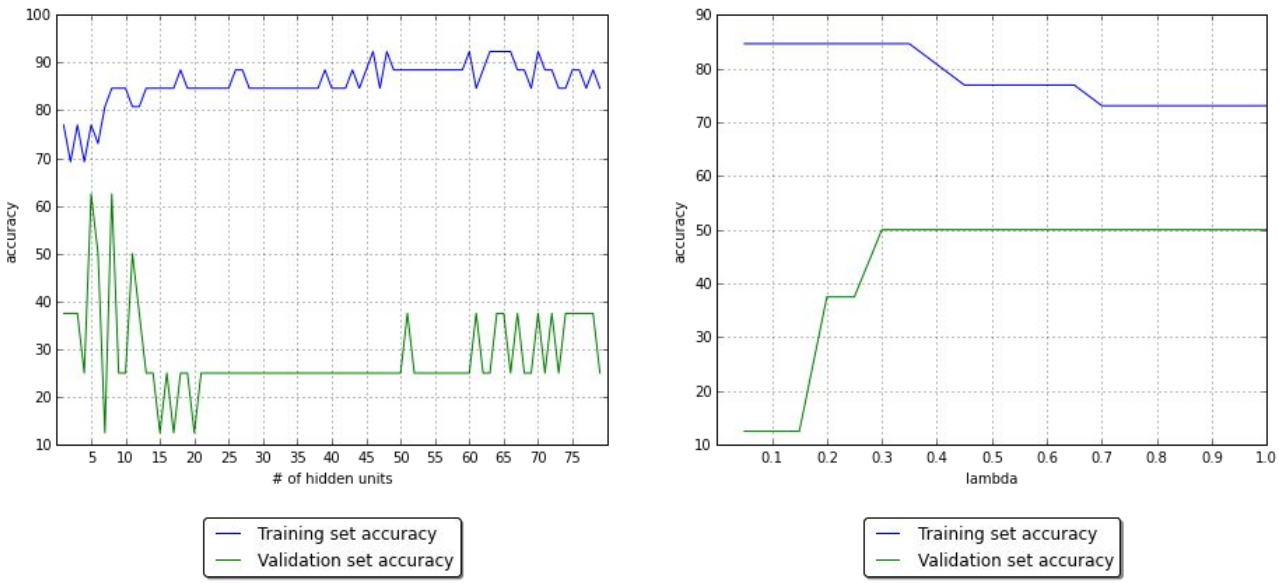
Feature reduction
We reduced the dimensionality of the data from 7004 to 35 using PCA. We chose 35 features because we found it descriptive enough without losing differentiating information.

Partitioning the dataset

With a dataset this small it's important to figure out the right balance for training & validation partitions. We tried various combinations and found that and 80-20 balance works the best.

1. 75% - 25% partition

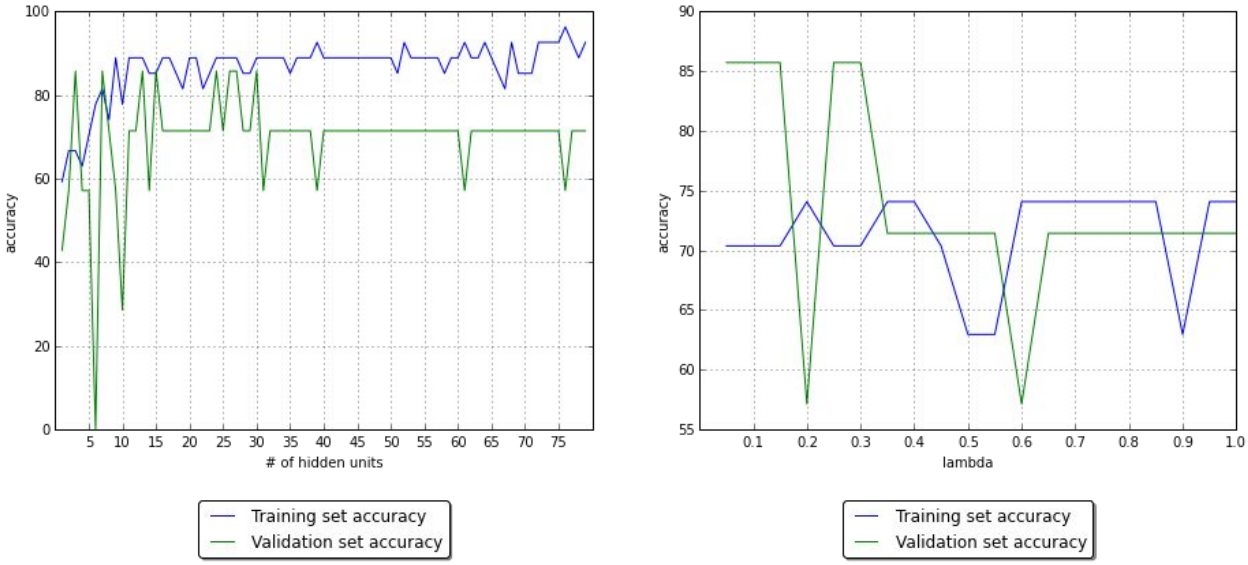
   Poor accuracy of on the validation dataset = 55.56%
   High overfitting: The training set accuracies are much, much higher than validation set accuracies.



2. 80% - 20% partition

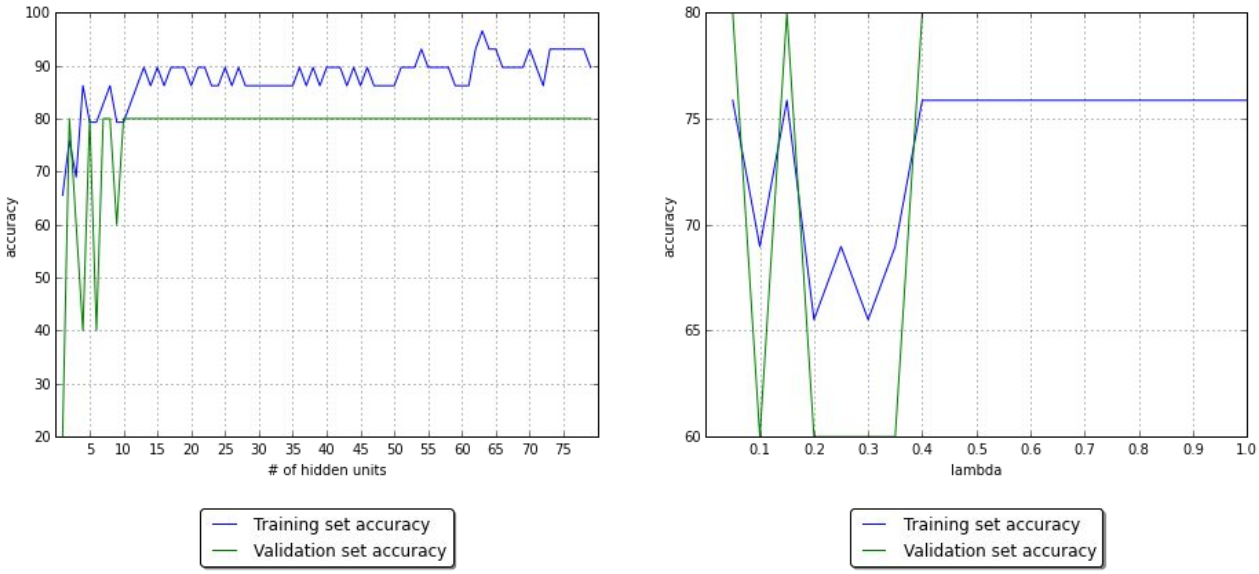
   Decent accuracy on the validation dataset = 85.71%

Not too much overfitting. This is a good partitioning scheme.



3. 85% - 15% partition

Lesser accuracy of on the validation dataset = 80.00%
Higher overfitting: The training set accuracies are much, much higher than validation set accuracies.



Based on the observations above we have decided to choose an 80%-20% partitioning for the small dataset of size 35, giving us 27 test samples and 8 validation samples.

Results

Before making predictions on the test data for dataset 3, we have first evaluated various classification models for their performance with a subset of the training data (which we used as a validation set).

For this purpose we have used the following classification models:

1. Logistic regression,
2. SVM
    a. SVM with poly kernel (degree=3),
    b. SVM with RBF kernel,
    c. SVM with Sigmoid kernel,
    d. SVM with linear kernel,
3. Random forest classifier,
4. Gaussian Naive Bayes classifier

For a fair comparison of the models we performed a 10-fold cross validation on multiple permutations of the training dataset. The results we obtained are discussed below:

Using model: **Logistic regression** Cross validation complete. Precision = 0.587857142857 Recall = 0.62 Accuracy = 0.485714285714 F-measure = 0.589598617757 Predicted labels for test data: [1 1 0 1 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 1 1]
Using model: **SVM with poly kernel (degree=3)** Cross validation complete. Precision = 0.446428571429 Recall = 0.541666666667 Accuracy = 0.7 F-measure = 0.475152162652 Predicted labels for test data: [0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1]
Using model: **SVM with RBF kernel** Cross validation complete. Precision = 0.328571428571 Recall = 0.5 Accuracy = 0.657142857143 F-measure = 0.385052447552 Predicted labels for test data: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
Using model: **SVM with Sigmoid kernel**

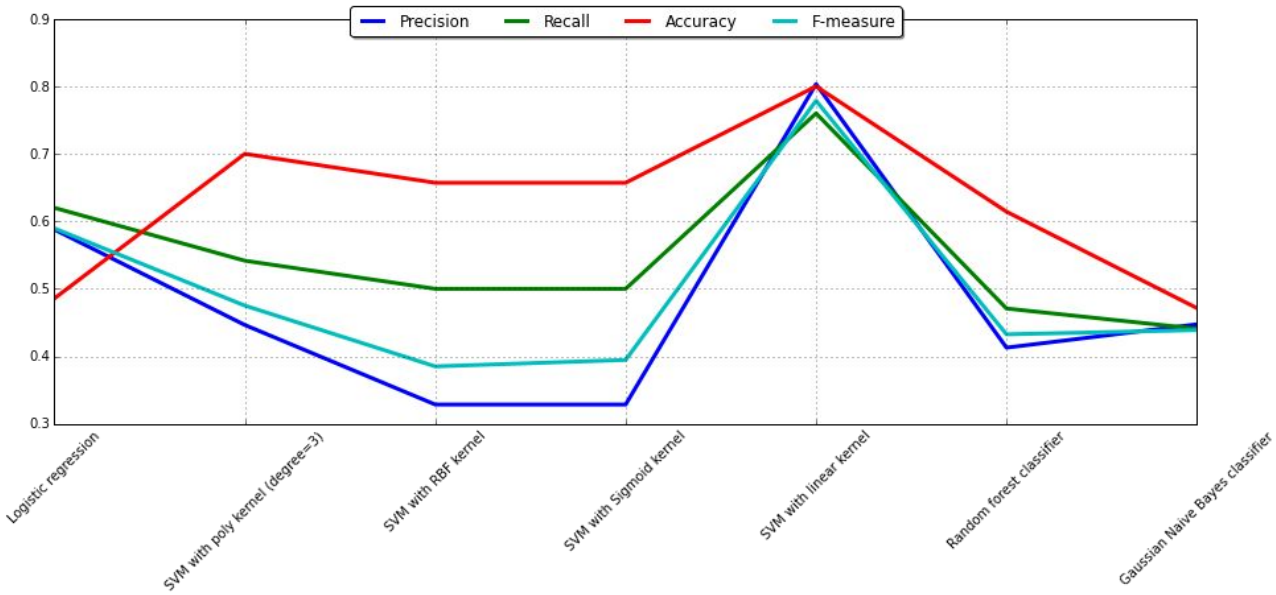
Cross validation complete.
Precision = 0.328571428571
Recall = 0.5
Accuracy = 0.657142857143
F-measure = 0.394638694639
Predicted labels for test data: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

Using model: **SVM with linear kernel**
Cross validation complete.
Precision = 0.803333333333
Recall = 0.76
Accuracy = 0.8
F-measure = 0.778691986031
Predicted labels for test data: [1 1 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1]

Using model: **Random forest classifier**
Cross validation complete.
Precision = 0.412857142857
Recall = 0.470833333333
Accuracy = 0.614285714286
F-measure = 0.432575757576
Predicted labels for test data: [0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0]

Using model: **Gaussian Naive Bayes classifier**
Cross validation complete.
Precision = 0.447380952381
Recall = 0.440833333333
Accuracy = 0.471428571429
F-measure = 0.438823264026
Predicted labels for test data: [0 0 0 0 0 1 0 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]

A visualization of these results is displayed below:



It is evident from the results that the **SVM model with a linear kernel** gives the best measures of all evaluation parameters. So our predictions for the test data are the ones predicted by this model:

[1 1 0 1 0 0 0 0 1 0 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1]

Part V: Other considerations:

Choosing probability function for continuous data in Naive Bayes:

While looking for different approaches for calculating probabilities for continuous data, we came across a research paper titled, "**Naive Bayes Classifiers that Perform Well with Continuous Variables**" by Dr.Remco R. Bouckaert. He had compared three different approaches for handling continuous variables in naive Bayes classifiers.

1. Normal method

   The classical method that approximates the distribution of the continuous variable using a parameterized distribution such as the Gaussian.

2. Kernel Method

   non-parameterized approximation of probabilities

3. Discretization

   The method which first discretizes the continuous variables into discrete ones

He has evaluated all the three methods using repeated k-fold cross validation on a number of datasets. According to the results of his evaluation, Kernel methods usually outperform normal methods in terms of accuracy however they are complex to implement. Also after, 10 times 10-fold cross evaluation, he found out that all the three methods have their strengths and weaknesses, and none of the three methods systematically outperforms the others on all problems they tested. In addition, the learning rate of the normal method can be slightly better than that of the kernel method if the generating distribution is indeed Gaussian.

Hence we chose normal method over other two types of methods.

Splitting continuous attributes using Uniform Approximation:

For a decision tree, deciding how many children a node should have is difficult for continuous attributes. This is because, the data in such attributes is distinct. Suppose a continuous attribute has N distinct values, it doesn't make sense to make N children for the node that splits on this attribute. Hence we used the technique know as **Uniform Approximation**. In this technique we define **k** split points for a continuous attribute. Hence if we have **c1 to ck split points**, the value of each split point is given by the formula:

$$c_i = \min(j) + i \cdot \frac{\max(j) - \min(j)}{k+1}$$

In our implementation, we have stored a list of midPoints for each node of the tree and we calculate the value of each mid point based on the above formula.

Initially we had used the binary split and later we implemented this method. There was a decent increase in the accuracies for both the datasets after this implementation. We have mentioned the **optimal number of split points for dataset 1 as 2 and for dataset 2 as 3 in our graphs above.**

Pruning in decision trees:

As decision trees tend to grow a lot until all the attributes are used, we have used a majority based pruning method while creating the tree. It works as follows:

Suppose the values for an attribute in the training data are - A,A,B,B,A,B,A,A,B,B,B . Now when we split on this attribute we get two children - left child with samples - A, A, A, A, A and right child with samples - B, B, B, B, B.

Now suppose the datapoints at the left child are 80% of Class 0 and datapoints at B are 60% Class 1.

We say that if the majority of a child if >= threshold (in this case say 80%) then don't bother splitting it further, instead split the one with 60% majority as we can't decide its class yet. This method helps in pruning the tree and guarantees that the child that is more impure will be split first.

In our code, we have used a threshold of 96% i.e. if the majority at a node is greater than 96 then there's no need to split the node further.