IE7374 FINAL PROJECT REPORT

Dry Bean Classification

Group 10
Rahul Dixit
Siddhartha Putti
Kirthana Chiramdasu
Remica Sequeira

dixit.ra@northeastern.edu
putti.s@northeastern.edu
chiramdasu.k@northeastern.edu
sequeira.r@northeastern.edu

Percentage of Effort Contributed by Student 1:_25%_
Percentage of Effort Contributed by Student 2:_25%_
Percentage of Effort Contributed by Student 3:_25%_
Percentage of Effort Contributed by Student 4:_25%_

Signature of Student 1:___*Rahul Dixit*____
Signature of Student 2:_____*Siddhartha Putti*___
Signature of Student 1:___*Kirthana Chiramdasu*____
Signature of Student 2:_____*Remica Sequeira*____

Submission Date:__4/25/22__

PROBLEM SETTING & DEFINITION:

There is a wide range of genetic diversity in dry beans which is the most produced among the edible legume crops in the world. Seed quality is definitely influential in crop production. Therefore, seed classification is essential for both marketing and production to provide the principles of sustainable agricultural systems. The primary objective of this study is to provide a method for obtaining uniform seed varieties from crop production, which is in the form of population, so the seeds are not certified as a sole variety. With the help of this project, we will try our best in classifying the 7 types of beans using several Machine learning techniques and algorithms.

DATA SOURCES:

Link to the UCI repository:
https://archive.ics.uci.edu/ml/datasets/Dry+Bean+Dataset Features/Predictors

DATA DESCRIPTION:

We have 16 attributes, 13,611 instances, and 7 outcome classes.

1.) Area (A): The area of a bean zone and the number of pixels within its boundaries.

2.) Perimeter (P): Bean circumference is defined as the length of its border.

3.) Major axis length (L): The distance between the ends of the longest line that can be drawn from a bean.

4.) Minor axis length (l): The longest line that can be drawn from the bean while standing perpendicular to the main axis.

5.) Aspect ratio (K): Defines the relationship between L and l.

6.) Eccentricity (Ec): Eccentricity of the ellipse having the same moments as the region.

7.) Convex area (C): Number of pixels in the smallest convex polygon that can contain the area of a bean seed.

8.) Equivalent diameter (Ed): The diameter of a circle having the same area as a bean seed area.

9.) Extent (Ex): The ratio of the pixels in the bounding box to the bean area.

10.) Solidity (S): Also known as convexity. The ratio of the pixels in the convex shell to those found in beans.

11.) Roundness (R): Calculated with the following formula: (4piA)/(P^2)

12.) Compactness (CO): Measures the roundness of an object: Ed/L

13.) ShapeFactor1 (SF1): The shape factor was determined as the ratio of pod width to thickness
14.) ShapeFactor2 (SF2): The shape factor was determined as the ratio of pod width to thickness

15.) ShapeFactor3 (SF3): The shape factor was determined as the ratio of pod width to thickness

16.) ShapeFactor4 (SF4): The shape factor was determined as the ratio of pod width to thickness
Outcome:

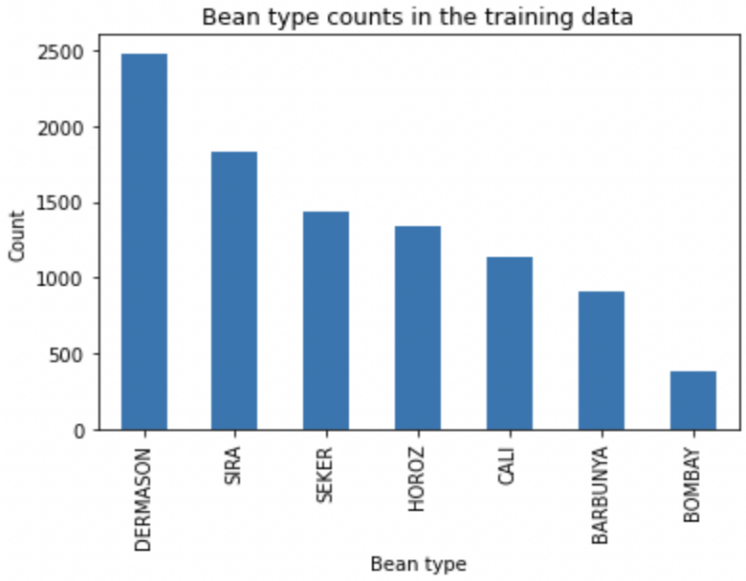
17.) Class: Seker, Barbunya, Bombay, Cali, Dermosan, Horoz,, and Sira are the various classes

SPLITTING THE DATASET

The first step we do before proceeding is to split the dataset into a train and test set. This helps us get rid of any inherent bias that we would incorporate during the Feature Selection/EDA stage. We split our data in the ratios of 70, and 30. Training 70%, Testing 30%.

EXPLORATORY DATA ANALYSIS

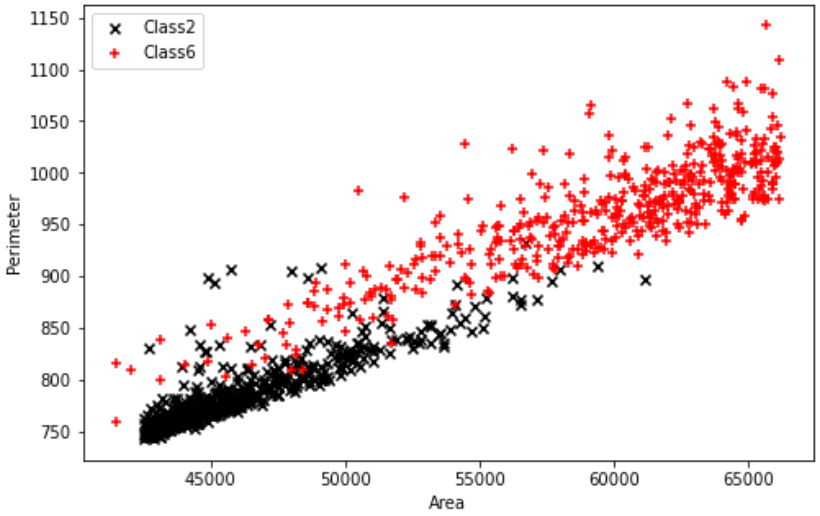
We plot a bar chart for the number of beans present in each type.

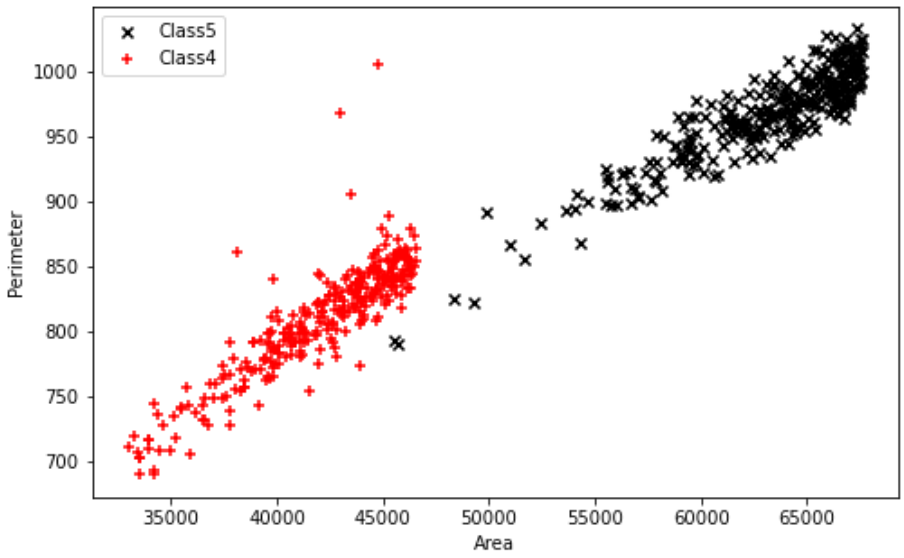


The count of each class varies as we can see from the above graph. Dermason is the most frequent class having 2496 beans. Bombay is the least frequent class having 365 beans. There is a huge difference between the 2 classes, which should be taken into consideration when building a model.

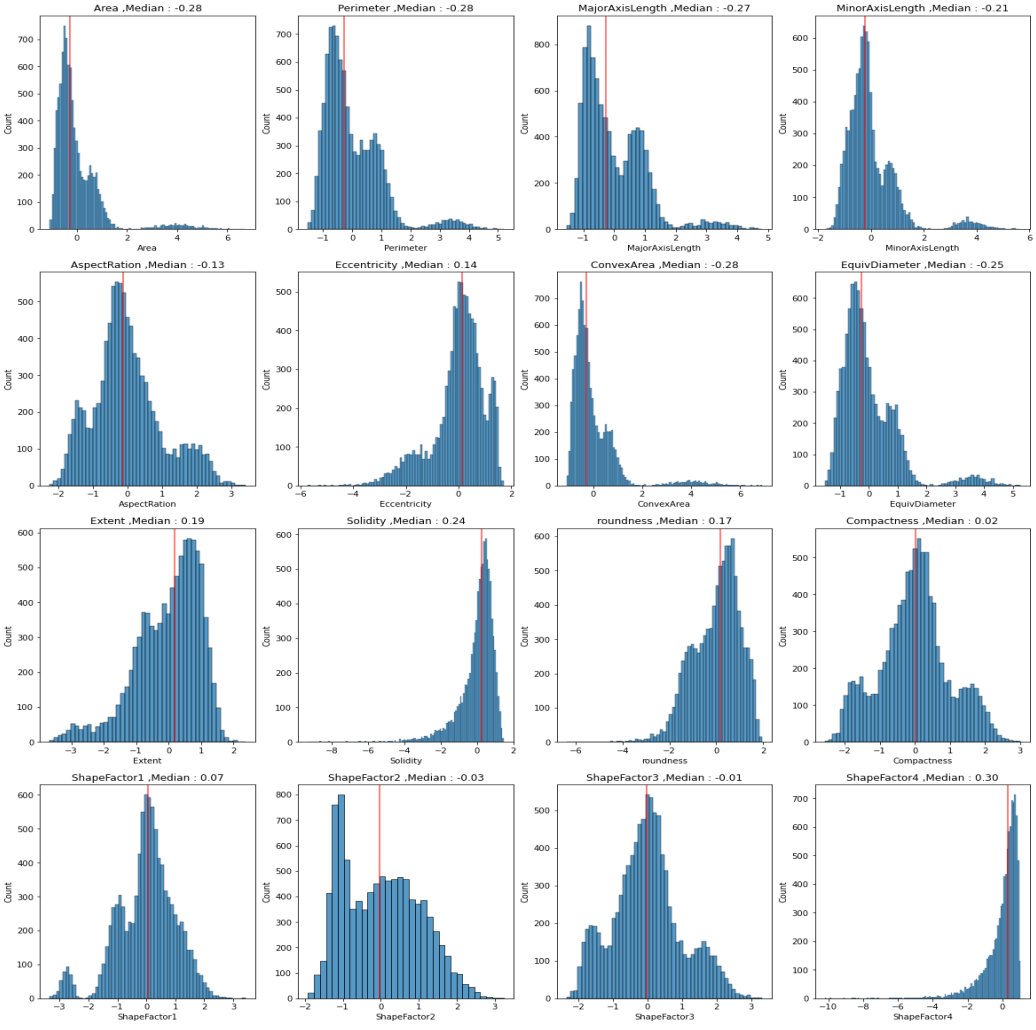
We see that the Equivalent Diameter of the bean type ‘SIRA’ is the maximum

Checking if the data exists in a Linear Sub-space:
We picked out some rows from different Classes and 2 major contributing features and checked if they are separable. From the below plots we can say that the major contributing features are linearly separable for different classes.



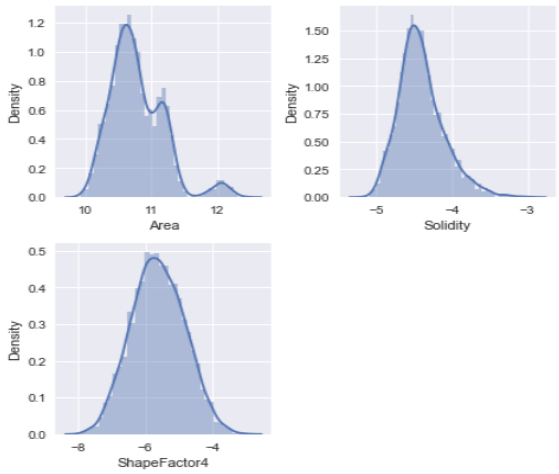


Univariate Analysis

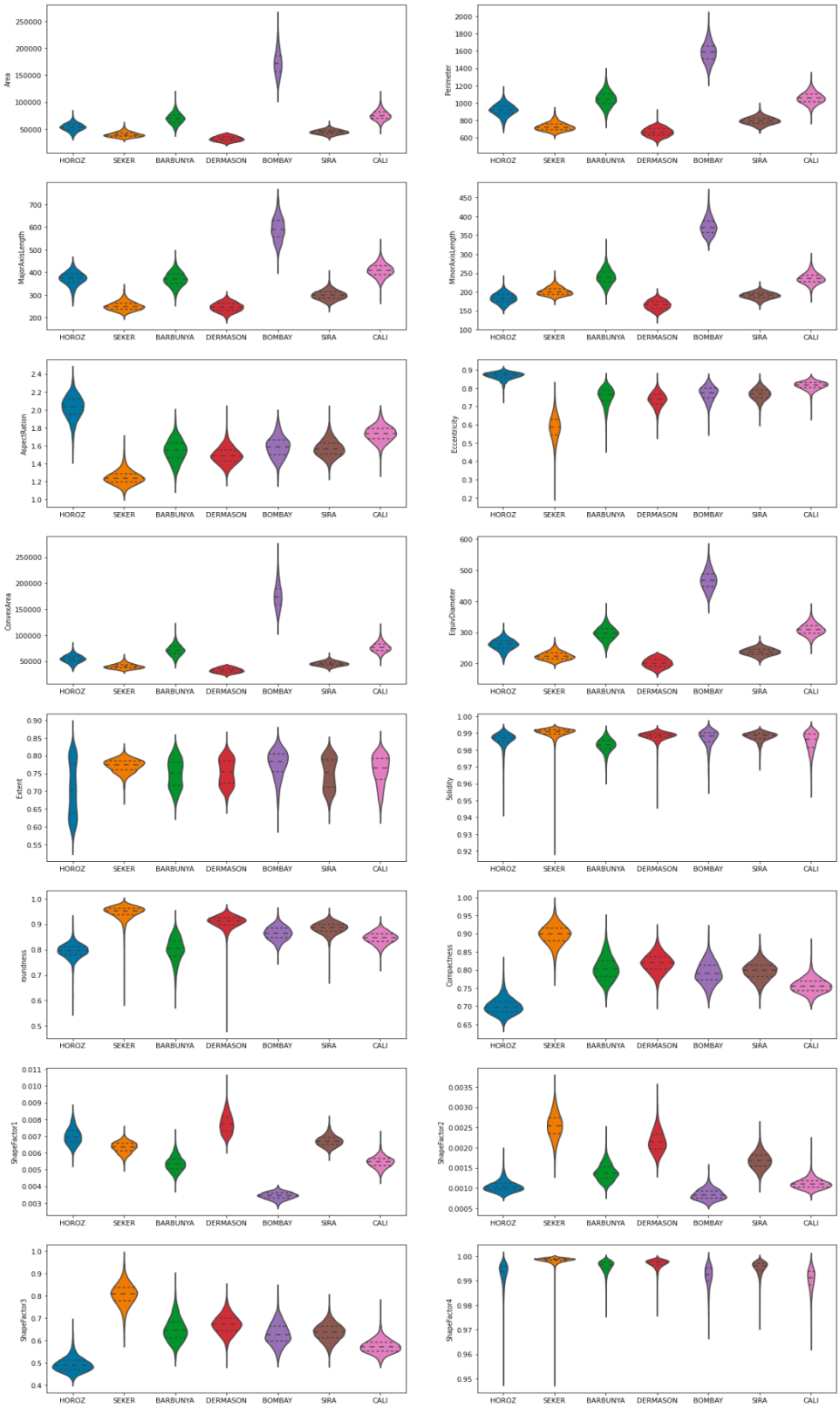


A lot of features show skewness and outliers in their distribution, these points may resemble a unique class of dry beans. We further investigate the effect of some of the highly effective columns and apply a log transform to some of the columns since it is being passed into the Naive Bayes model which assumes Gaussian Distribution.

After applying log transform our features look like this:

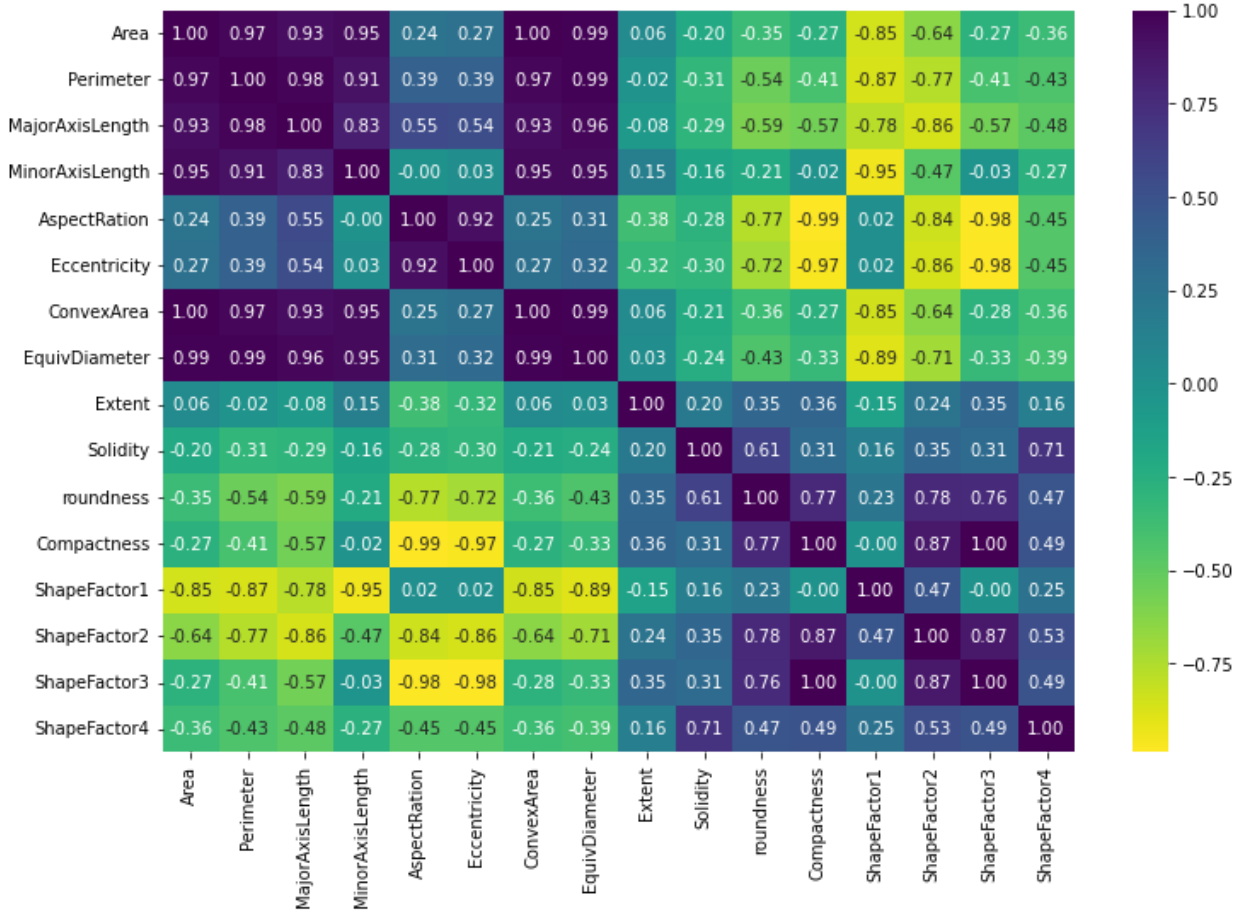


Violin plot to check the distribution of data for different classes



By plotting a violin chart, we observe that some features have a highly skewed distribution with long tails (eccentricity, solidity, roundness, shape factor2, shape factor4). Bombay class differs greatly from other classes, it has a larger area and perimeter. It can be clearly distinguished from other classes based on Minor Axis Length and Shape Factor 1.

Multi-Variate Analysis



By plotting a heat map, we try to figure out what features are correlated.

- The area is highly correlated with Equivalent Diameter, Major axis, and Minor Axis length
- Perimeter is highly correlated with Equi Diameter.
- Aspect Ratio is highly correlated with Compactness and Shape Factor3.
- We Drop perimeter, MajorAxisLength, MinorAxisLength, EquivDiameter, Compactness, and ShapeFactor3.
- *This is done for some of the models*

UNI VARIATE FEATURE SELECTION

Mutual information:
Mutual information between two random variables is a non-negative value, which measures the dependency between the variables. It is equal to zero if and only if two random variables are independent, and higher values mean higher dependency.

	features	Scores
1	Perimeter	1.060967
6	ConvexArea	1.036974
0	Area	1.035343
7	EquivDiameter	1.035111
2	MajorAxisLength	0.991314
13	ShapeFactor2	0.948655
12	ShapeFactor1	0.932884
3	MinorAxisLength	0.929724
14	ShapeFactor3	0.826614
11	Compactness	0.826610
5	Eccentricity	0.822355
4	AspectRation	0.822312
10	roundness	0.798797
15	ShapeFactor4	0.374618
9	Solidity	0.229328
8	Extent	0.197554

We observe that "Solidity", "ShapeFactor4", and "Extent" have the lowest scores, which means that they are less likely to depend on the target variable.

ANOVA / F-VALUE

F-test estimates the degree of linear dependency between feature and the target variable. F-test estimates the degree of linear dependency between two random variables.
On the other hand, mutual information methods can capture any kind of statistical dependency, but being nonparametric, they require more samples for accurate estimation.

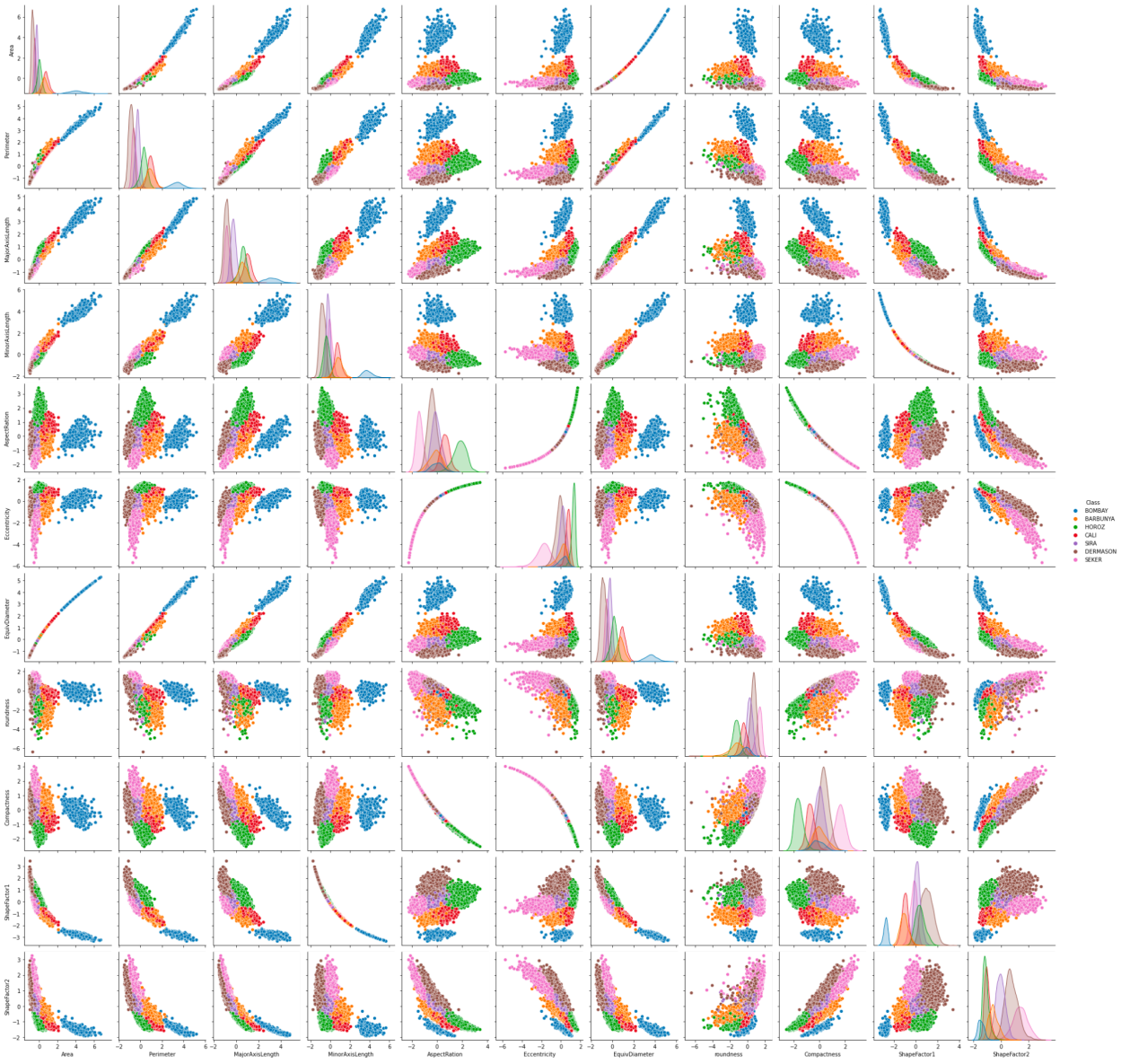
	features	Scores
0	Area	20832.039984
6	ConvexArea	20767.223264
7	EquivDiameter	18287.084343
1	Perimeter	17382.307034
3	MinorAxisLength	16351.774938
2	MajorAxisLength	15359.513660
13	ShapeFactor2	8641.824438
12	ShapeFactor1	8641.763125
4	AspectRation	7352.794983
11	Compactness	7174.827023
14	ShapeFactor3	6978.170291
5	Eccentricity	5790.736955
10	roundness	4325.857754
15	ShapeFactor4	881.488214
9	Solidity	454.404598
8	Extent	307.825667

Both f-test and mutual information have the lowest scores for "Solidity", "ShapeFactor4", and "Extent".

MODEL-BASED FEATURE SELECTION

There are many ways by which features can be selected using a model and the L1 penalty is one of them. We consider the L1 penalty to reduce the dimensions of the data using linear_SVM, in linear_SVM the parameter C decides the level of penalty imposed. If C is lower, fewer features will be selected.

with C being 0.01, the dimensions are reduced from 16 to 13



- We can see a linear trend between many features.
- Notice that the Bombay class is mostly separated from other classes in some features, which means that despite having a low count in the dataset, a model may still be able to correctly classify it.

- The classes are clearly clustered within some scatterplots, mainly between the area and perimeter features with all other features.
- Even though we can see clusters for each class, there's some overlap between them, mainly between Dermason and Sira classes.
- Some features (aspect ratio, eccentricity, compactness) seem to hit abound when plotted against roundness, which indicates that (in the given data) no outliers occur above that bound.
- Most of the classes are linearly separable with respect to features.
- We can use SVM_linear and Naive Bayes as algorithms for linearly separable data but there are also some nonlinear trends observed in the data with respect to features, so we tried KNN, and multinomial logistic(softmax), and Neural Networks.

DIMENSIONALITY REDUCTION

1. Principal Component Analysis:

   Principal component analysis selects the components that have the highest variance captured. The components are calculated by projecting values onto the eigenvectors of the respective covariance matrices. We do eigenvalue decomposition to find the eigenvalues and eigenvectors. If the data set is not a square matrix, we can decompose using singular value decomposition.

   The first n components from the singular values decomposition, when calculated the maximum variance captured from the formula:

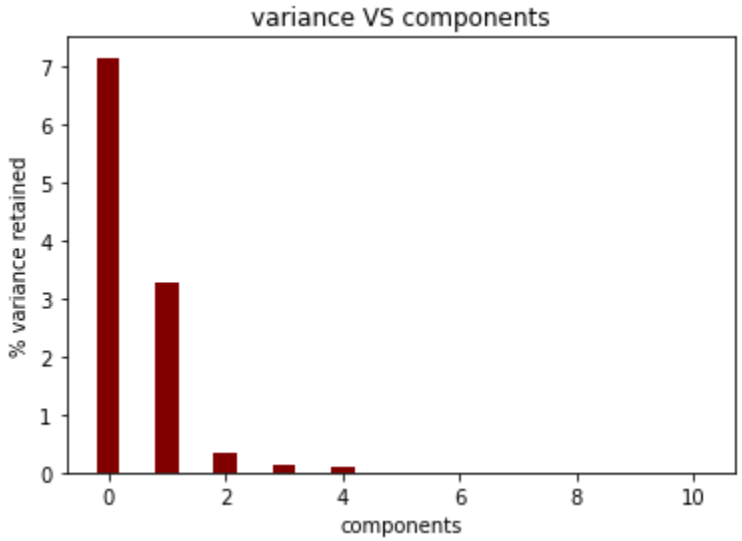
$$\frac{\sum_{i=1}^{n} s_i}{\sum_{i=1}^{m} s_i}$$

   n = first n eigenvalues
   M = sum of all eigenvalues

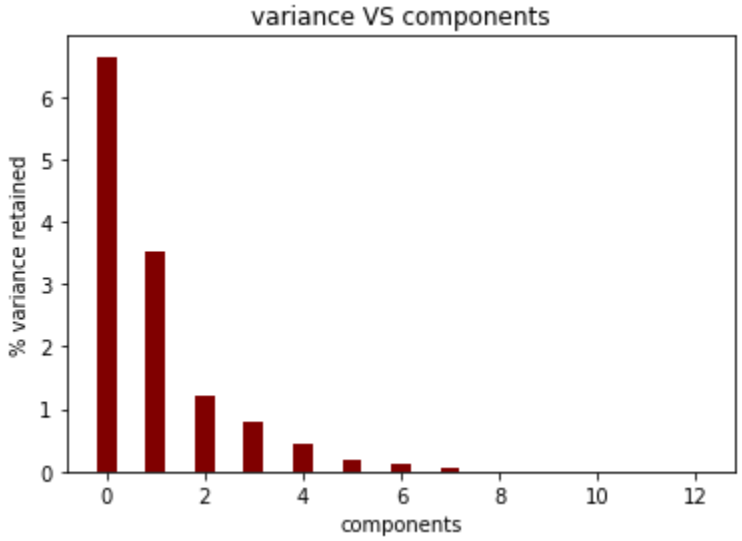
   This returns the variance captured by first n components.

   Step1: make the values mean-centered
   Step2: find the covariance of the matrix
   Step3: find eigenvalues and eigenvectors of the covariance matrix
   Step4: sort eigenvectors
   Step6: pick top n eigenvectors
   Step4: project data onto the eigenvectors

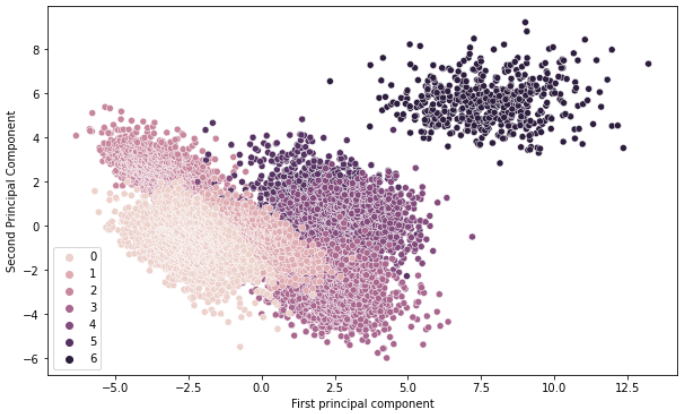
The below plot shows the percentage of variance captured VS a number of components; we can see that the first 2-3 components cover most of the variance. The data selected over this plot is from univariate feature selection methods.



We have also performed the same PCA over model-based feature selection (used SVM to select features) shown in the below chart.

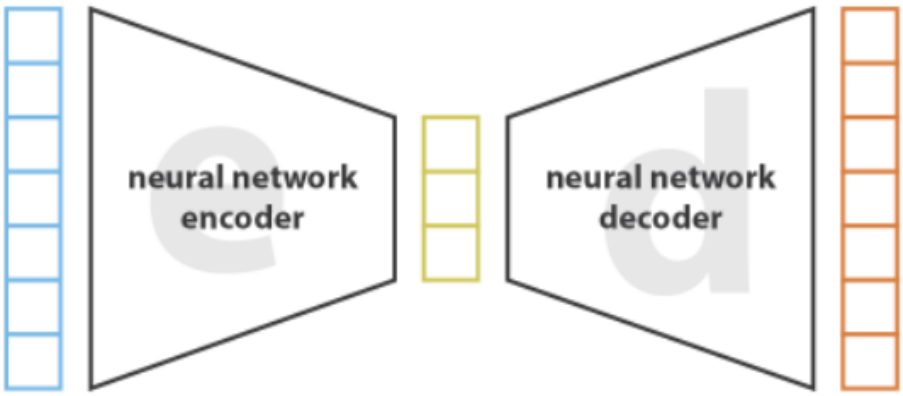


We can see that it requires almost 5 components to have most of the variance. So, we are selecting univariate feature selection methods data to move forward. 97% percent of the variance is captured by the 5 components.
The plot of the PC1 vs PC2 is as follows:



2. Variational AutoEncoders

The general idea of autoencoders is simple and consists in setting an encoder and a decoder as neural networks and learning the best encoding-decoding scheme using an iterative optimization process. The basic architecture of AutoEncoders is as follows



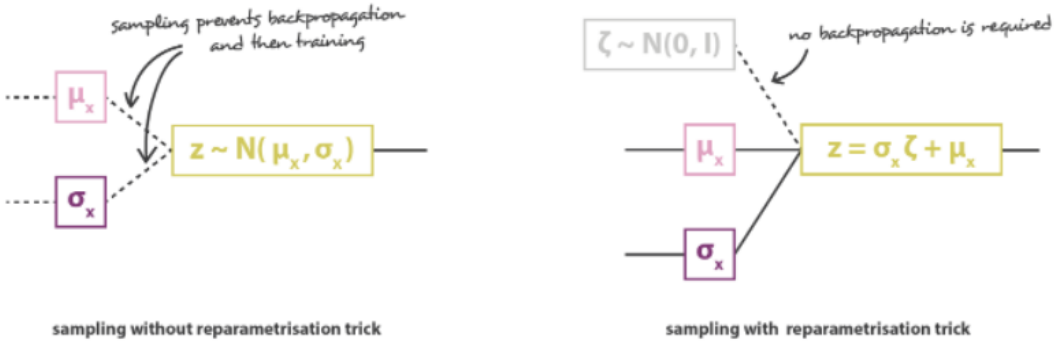
In PCA, we are looking for the best linear subspace to project data on with as little information loss as possible when doing so. Encoding and decoding matrices obtained with PCA define naturally one of the solutions we would be satisfied to reach by gradient descent, but we should outline that this is not the only one. Indeed, **several basis can be chosen to describe the same optimal subspace,** and, so, several encoder/decoder pairs can give the optimal reconstruction error. Moreover, for linear autoencoders and contrarily to PCA, the new features we end up do not have to be independent.

In variational AutoEncoders, the high degree of freedom of the autoencoder that makes it possible to encode and decode with no information loss (despite the low dimensionality of the latent space) **leads to severe overfitting** implying that some points of the latent space will give meaningless content once decoded. We define our sub-parametric optimization property in autoencoders to overcome overfitting.

Instead of encoding an input as a single point, we encode it as a distribution over the latent space. The model is then trained as follows:

- first, the input is encoded as a distribution over the latent space
- second, a point from the latent space is sampled from that distribution
- third, the sampled point is decoded, and the reconstruction error can be computed
- finally, the reconstruction error is back propagated through the network

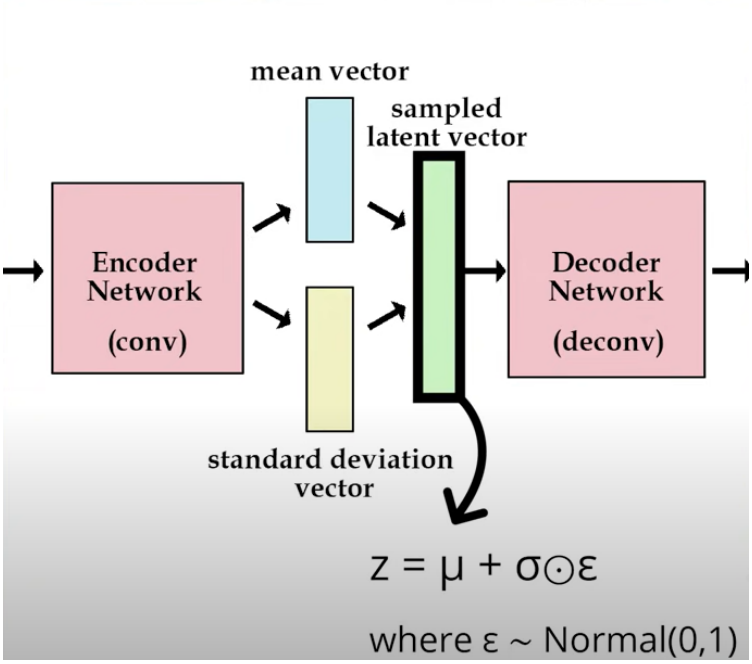
Here we are finding a sample Gaussian distribution of encoded data that can be directly expressed in terms of means and covariances of two distributions. The regularization used in this method is KL Divergence. This basically calculates the distance between the distributions.



sampling without reparametrisation trick

sampling with reparametrisation trick

The encoded part is divided into or copied into two subparts, one as a sample distribution of encoded data and the other for the backpropagation. The sampled part is then sent to the decoding such that the input is not mimicked on the decoded layer. This needs a cost function that can handle the Gaussian distributions where KL Divergence is used as follows:



Here epsilon is going to be standard gaussian normal with mean 0 and standard deviation 1 when multiplied with SD and added to the mean vector giving the latent vector for decoding.

In our case, we considered the tuned STD to be 0.5.

```
def reparameterize(self, mu, logvar):
    if self.training:
        std = logvar.mul(0.5).exp_()
        eps = Variable(std.data.new(std.size()).normal_())
        return eps.mul(std).add_(mu)
    else:
        return mu
```

And the loss function = loss + KLD_loss

```
def forward(self, x_recon, x, mu, logvar):
    loss_MSE = self.mse_loss(x_recon, x)
    loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return loss_MSE + loss_KLD
```

We randomly chose a mini-batch to check the original VS reconstructed data

Original data

```
standardizer.inverse_transform(data[65].cpu().numpy())
```

```
array([-0.8626753 , -1.1429613 , -1.1140032 , -0.9822785 , -0.65055466,
       -0.43116057, -1.0952591 ,  1.0912019 ,  0.6105686 ,  1.3247869 ,
        1.2285852 ], dtype=float32)
```

Reconstructed data

```
standardizer.inverse_transform(recon_batch[65].cpu().numpy())
```

```
array([-0.8003157 , -1.0243723 , -1.0071269 , -0.8843839 , -0.5607413 ,
       -0.31931147, -0.9844946 ,  0.78466815,  0.49616048,  1.1781172 ,
        1.01199   ], dtype=float32)
```

The data Reconstructed is very similar with some small errors. Overall, the error is decreasing as the epochs increase over time. We can now use the latent vector as dimensionally reduced data and feed it to our models.

MODEL IMPLEMENTATION

We believe that we can solve this classification problem using various Machine Learning algorithms. Some of which are:

1. Multi-Class Logistic Regression
2. Support Vector Machines
3. KNN
4. Naive Bayes
5. Neural Networks

Multi-class logistic regression

Multinomial logistic regression is an extension of logistic regression for multi-class classification using softmax activation. It is used when we want to predict more than 2 classes which provide the probability of belonging to different classes.

Algorithm:

1) We have used softmax in logistic regression where the dependent variable is more than two classes. We used the softmax function since we have 7 classes.

```
def softmax(self,z):
    z = z - np.max(z)
    z = np.exp(z)
    return np.divide(z , z.sum(axis=0))
```

2) The categorical cross-entropy loss function calculates the loss of an example by computing the following sum:

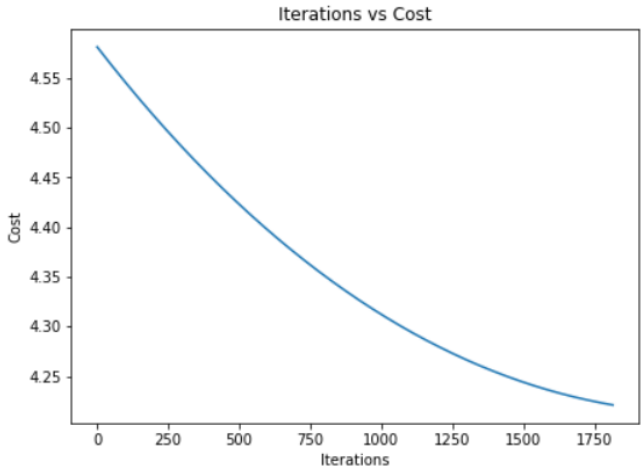
$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

3) In the specific (and usual) case of Multi-Class classification the labels are one-hot, so only the positive class Cp keeps its term in the loss. There is only one element of the Target vector t which is not zero ti=tp So discarding the elements of the summation which are zero due to target labels, we can write:

$$\text{Loss} = -\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

```
def costFunction(self,X,y):
    y_hat = self.softmax(X.dot(self.w)) # y_hat = predicted labels
    cost  = - np.sum(np.log(y_hat) * y, axis=1) # - sum(  y * log ( y_hat )  )
    if self.reg:
      cost  = cost + (self.reg_param * ( np.sum(np.square(self.w)) ) )
    return 0.5* np.mean(cost)
```

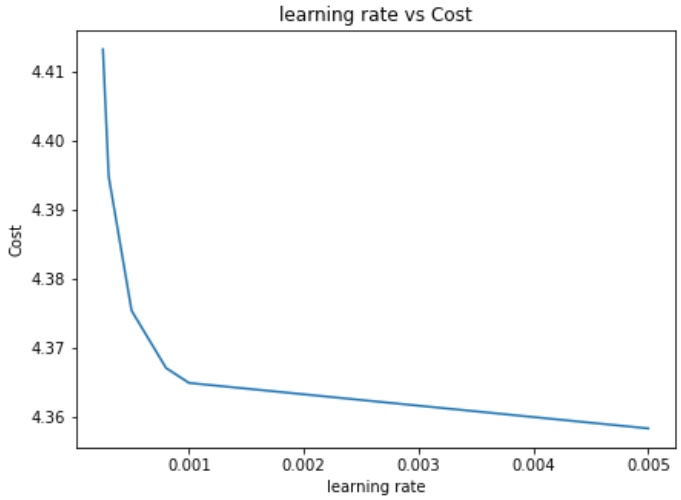
4) The following graph displays cost along with its accuracy, train errorand test error.



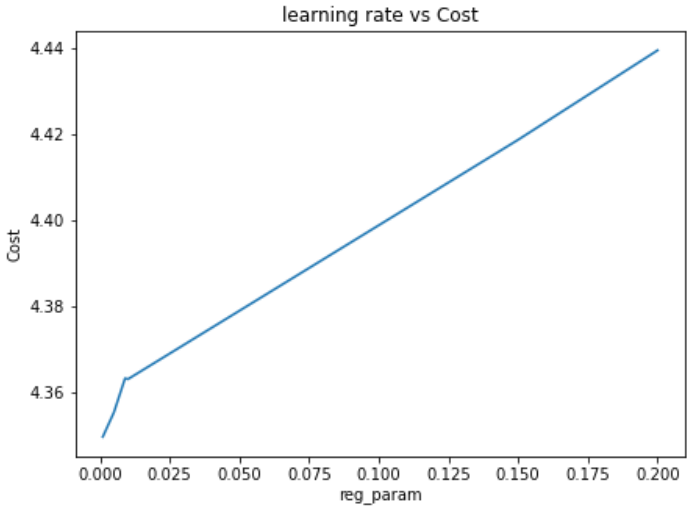

```
accuracy: 0.8457394711067581
test error: 3.800297094089643
```

As the number of iterations increases the cost decreases using L2 regularization.

The below plot is the learning rate VS average loss across the iterations, the average loss decreases as the learning rate increases.



The below plot is the Regularization parameter value (L2 penalization) and cost. As the value of penalization increases the loss also increases, because the greater the penalty term, the weight vector is forced to become zero, which makes the model more biased. In contrast, if the regularization parameter is very low value, the cost is also very low which indicates the model is overfitting the data.



SUPPORT VECTOR MACHINE

A support vector machine (SVM) is a machine learning algorithm that analyzes data for classification and regression analysis. SVM is a supervised learning method that looks at data and sorts it into one of two categories. An SVM outputs a map of the sorted data with the margins between the two as far apart as possible while keeping the distance between the data point and the margin minimum. Here, we used Soft Margin SVM and since the dataset has multiple classes, we performed one vs all.

Algorithm:

1. The soft margin SVC allows some of the points to misclassify using the hyperparameter C. if C is large it is a hard margin SVC, on the contrary, if C is small we allow some points to be misclassified by forcing $\alpha's$ them to be in a specific range.

This can be achieved through complementary slackness, which is the 4th rule of KKT conditions to be satisfied. They are

$$\alpha_i(1 - y_i f(x_i) - \varepsilon) = 0$$

$$(\frac{c}{n} - \alpha_i)\varepsilon = 0$$

- If $y_i f(x_i) > 1$ then the margin loss is $\varepsilon = 0$ and we get $\alpha_i = 0$
- If $y_i f(x_i) < 1$ then the margin loss is $\varepsilon > 0$ so $\alpha_i = \frac{c}{n}$
- If $\alpha_i = 0$ then $\varepsilon = 0$ which implies no loss, so $y_i f(x_i) >= 1$
- If $\alpha_i \epsilon (0, \frac{c}{n})$ then $\varepsilon = 0$ which implies $1 - y_i f(x_i) = 0$

2.We used Gaussian radial basis function as kernel to address the non linearity in data. The Gaussian radial basis function measures the gaussian distance between the data points in a non-linear space. This is also addressed as the similarity between the data points.

$$K(X, Y) = exp(-\frac{||x-y||2}{2\sigma^2})$$

```
def GRBF(x1, x2):
    diff = x1 - x2
    return np.exp(-np.dot(diff, diff) * len(x1) / 2)
```

3. As we are dealing with multiple classes, we implemented One VS all, taking one class at a time as ‘1’ and other classes as ‘-1’.

4. The support vector machines are taking longer than expected to execute the model, thus we implemented one VS all for one class at a time by sampling our data.
The result for the same is as follows:

```
one_vs_all(X_train_UV_pca,y_train_UV,X_test_UV_pca, y_test_UV)
```

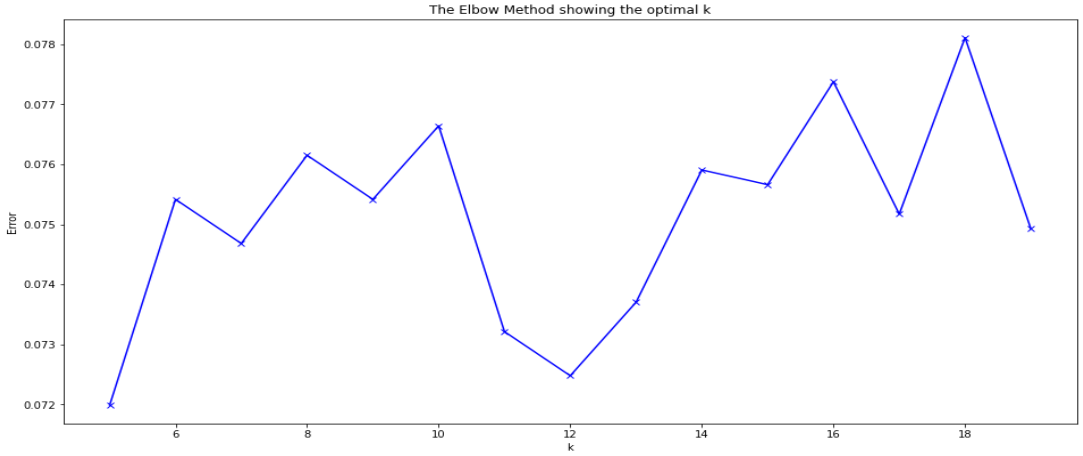
```
92 corrcted out of 100:
```

K-Nearest Neighbors

k-Nearest Neighbors (KNN) is a supervised machine learning algorithm that can be used for either regression or classification tasks. KNN is non-parametric, which means that the algorithm does not make assumptions about the underlying distributions of the data.

Algorithm:

1. Use the distance function to get the distance between a test point and all train points. Yes, the computation is intense but that's how the algorithm works. We have used euclidean distance as the distance function but this is one of many different ones.
2. We concatenate the distances with the training set.
3. Sort distance measurements to find the points closest to the test point by choosing K.
4. To choose an Optimal K, we ran a loop to test how the model performs. The metric used was how many data points were correctly classified by total data points



   Looking at this we chose the Optimal K value to be 12.
5. We use majority class labels of those closest points to predict the label of the test point. This is done by checking the frequency of the label.
6. We repeat steps 1 through 5 until all test data points are classified

Results - KNN

The classification report for KNN is as follows:

```
misclassification rate 0.07247796278158668
testing error 7.247796278158668
              precision    recall  f1-score   support

         0.0       0.90      0.94      0.92      1057
         1.0       0.89      0.87      0.88       830
         2.0       0.95      0.95      0.95       579
         3.0       0.97      0.96      0.96       594
         4.0       0.92      0.97      0.94       501
         5.0       0.97      0.87      0.92       389
         6.0       1.00      1.00      1.00       134

    accuracy                           0.93      4084
   macro avg       0.94      0.94      0.94      4084
weighted avg       0.93      0.93      0.93      4084
```

The testing error = 7.247%
Test Time: 0:16:47.268353 (16 minutes, 47 seconds)
We proceed to the training error next to advocate for the Bias Variance Tradeoff. This is done by running the model using the train set.

```
misclassification rate 0.06864700325390995
training error 6.864700325390995
              precision    recall  f1-score   support

         0.0       0.92      0.93      0.93      2489
         1.0       0.87      0.89      0.88      1806
         2.0       0.96      0.95      0.95      1448
         3.0       0.97      0.95      0.96      1334
         4.0       0.93      0.96      0.94      1129
         5.0       0.97      0.90      0.93       933
         6.0       1.00      1.00      1.00       388

    accuracy                           0.93      9527
   macro avg       0.94      0.94      0.94      9527
weighted avg       0.93      0.93      0.93      9527
```

The training error = 6.86%

Gaussian Naive Bayes

Gaussian Naive Bayes is a Machine Learning Generative Algorithm that makes an assumption that the data is Gaussian. We feed in a separate data frame because of this assumption. Some columns of which have been transformed due to the Gaussian Assumption. It considers the data to follow a gaussian distribution.

The algorithm is as follows:

1. We first create a function to fit the data in a gaussian distribution to find the likelihood. This is done by finding the mean and the standard deviation and these 2 values are fed into the 'norm' function that scipy provides us. It calculates the probability density for us for each dimension.

$$f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$$

2. Next we proceed to find the priors for each class. This is defined as the number of rows that give us a class 'K' divided by the total number of rows. We find the priors for all 7 classes.

3. We then fit distributions for each dimension for each of the classes. So, in our case, we have 10 dimensions (after reduction) and 7 classes so we compute 70 distributions. This is done using the help of a dictionary and a single for loop. Thus, 70 likelihoods have been computed.

Having computed the priors and the likelihoods for each class and dimension, we then are left with feeding in the test dataset to compute the probability of a certain row belonging to a certain class. This is done with the help of nested loops of an array and dictionary

```
def probability(self,X,prior,distribution):
    self.num_features = self.X_test.shape[1]
    res = []
    for j in range(self.num_features):
        for k,v in distribution.items():
            res.append(v.pdf(X[j]))
    res = res[0::self.num_features+1]
    a = np.prod(res, axis=None, dtype=None, out=None)
    return a*prior
```

```
for sample, target in zip(self.X_test, self.y_test):

    py0 = self.probability(sample,self.prior_y0,self.dist1)
    py1 = self.probability(sample,self.prior_y1,self.dist2)
    py2 = self.probability(sample,self.prior_y2,self.dist3)
    py3 = self.probability(sample,self.prior_y3,self.dist4)
    py4 = self.probability(sample,self.prior_y4,self.dist5)
    py5 = self.probability(sample,self.prior_y5,self.dist6)
    py6 = self.probability(sample,self.prior_y6,self.dist7)
```

5. We compare each of the probabilities and find the maximum and then assign that input to that class.

6. We continue this process until we compute it for the whole test set.

Results-Gaussian Naive Bayes

Error Metric: Number of correct classifications/ Total Classification
Count in the below image is defined as the number of instances misclassified as some other class

```
testing error : 8.349657198824682
count for class 0 : 107
count for class 1 : 97
count for class 2 : 35
count for class 3 : 27
count for class 4 : 41
count for class 5 : 49
count for class 6 : 6
```

Test Time: 05:08.129370 (5 minutes, 8 seconds)

```
training Error: 9.257898603967671
count for class 0 : 299
count for class 1 : 232
count for class 2 : 83
count for class 3 : 62
count for class 4 : 95
count for class 5 : 126
count for class 6 : 6
```

DECISION TREE

Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. A **node** is the building block in the decision tree.

We strive to make splits to classify each instance. The splits are governed by how much Information gained is achieved. Information gain can be quantified by the Entropy/Gini Index. The purity of a node is directly proportional to an increase in purity. By comparing the reduction in impurity across all possible splits in all possible predictors, the next split is chosen.

$$Gini = 1 - \sum_j p_j^2$$

Pj is the probability of that class occurring

The algorithm is as follows:

1. We start by striving to get the best split based on a dimension that gives the least impurity. We use the initial Gini for the base input and then sort values by using a moving average. This function is called the best_split. It recursively calculates the left, and right Gini index and then a weighted Gini. We keep checking the split is giving the most information gain and if it is, we set it to be the best gain. This returns the best feature and its value.

2. We move to the grow tree function where we find the recursively keep growing tree by calling the previous best split function. We keep splitting until Gini is to be gained and save the best split to the current node.

3. There is a function we have created to showcase what the tree would look like:

```
Root
   | GINI impurity of the node: 0.83
   | Class distribution in the node: {5.0: 924, 0.0: 2484, 4.0: 1127, 2.0: 1424, 6.0: 372, 1.0: 1828, 3.0: 1368}
   | Predicted class: 0.0
|-------- Split rule: ConvexArea <= 40192.0
          | GINI impurity of the node: 0.47
          | Class distribution in the node: {0.0: 2345, 2.0: 772, 3.0: 46, 1.0: 258}
          | Predicted class: 0.0
|---------------- Split rule: ShapeFactor1 <= 0.007
                  | GINI impurity of the node: 0.16
                  | Class distribution in the node: {2.0: 714, 0.0: 43, 1.0: 22}
                  | Predicted class: 2.0
|----------------------- Split rule: AspectRation <= 1.332
                         | GINI impurity of the node: 0.04
                         | Class distribution in the node: {2.0: 694, 0.0: 12, 1.0: 3}
                         | Predicted class: 2.0
|------------------------------- Split rule: ShapeFactor4 <= 0.995
                                 | GINI impurity of the node: 0.64
                                 | Class distribution in the node: {0.0: 4, 1.0: 2, 2.0: 4}
                                 | Predicted class: 2.0
|------------------------------- Split rule: ShapeFactor4 > 0.995
                                 | GINI impurity of the node: 0.03
                                 | Class distribution in the node: {2.0: 690, 0.0: 8, 1.0: 1}
                                 | Predicted class: 2.0
|----------------------- Split rule: AspectRation > 1.332
                         | GINI impurity of the node: 0.65
                         | Class distribution in the node: {1.0: 19, 2.0: 20, 0.0: 31}
                         | Predicted class: 0.0
|------------------------------- Split rule: ShapeFactor2 <= 0.002
                                 | GINI impurity of the node: 0.51
                                 | Class distribution in the node: {1.0: 14, 0.0: 8, 2.0: 1}
                                 | Predicted class: 1.0
|------------------------------- Split rule: ShapeFactor2 > 0.002
                                 | GINI impurity of the node: 0.59
                                 | Class distribution in the node: {2.0: 19, 1.0: 5, 0.0: 23}
                                 | Predicted class: 0.0
|---------------- Split rule: ShapeFactor1 > 0.007
                  | GINI impurity of the node: 0.23
                  | Class distribution in the node: {0.0: 2302, 2.0: 58, 3.0: 46, 1.0: 236}
                  | Predicted class: 0.0
|----------------------- Split rule: ConvexArea <= 36683.5
                         | GINI impurity of the node: 0.11
                         | Class distribution in the node: {0.0: 1964, 2.0: 56, 1.0: 47, 3.0: 21}
                         | Predicted class: 0.0
|------------------------------- Split rule: ShapeFactor1 <= 0.007
                                 | GINI impurity of the node: 0.47
                                 | Class distribution in the node: {2.0: 47, 0.0: 25, 1.0: 1}
                                 | Predicted class: 2.0
|------------------------------- Split rule: ShapeFactor1 > 0.007
                                 | GINI impurity of the node: 0.07
                                 | Class distribution in the node: {0.0: 1939, 1.0: 46, 3.0: 21, 2.0: 9}
                                 | Predicted class: 0.0
|----------------------- Split rule: ConvexArea > 36683.5
                         | GINI impurity of the node: 0.51
                         | Class distribution in the node: {0.0: 338, 3.0: 25, 1.0: 189, 2.0: 2}
                         | Predicted class: 0.0
|------------------------------- Split rule: ShapeFactor2 <= 0.001
                                 | GINI impurity of the node: 0.44
```

4. We feed in the testing sets based on the splits done above.

Results- Decision Tree

Testing set

Test Time : Test Time: 18:06.49370 (18 Minutes, 6 seconds)

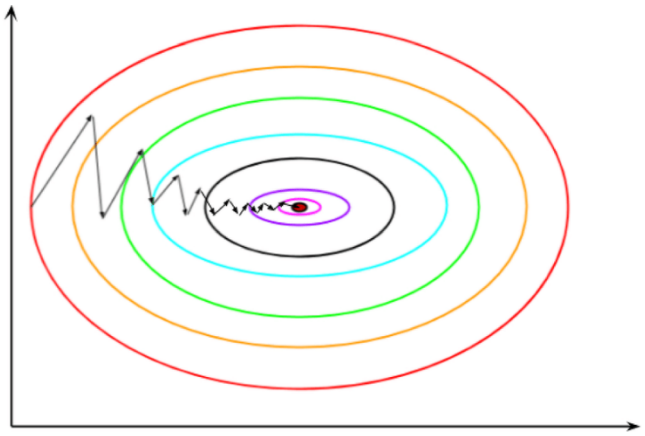
	precision	recall	f1-score	support
0	0.90	0.93	0.91	1062
1	0.83	0.87	0.85	808
2	0.94	0.92	0.93	603
3	0.98	0.88	0.93	560
4	0.55	0.98	0.70	503
5	0.00	0.00	0.00	398
6	1.00	0.97	0.98	150
accuracy			0.83	4084
macro avg	0.74	0.79	0.76	4084
weighted avg	0.78	0.83	0.79	4084

Training Set

	precision	recall	f1-score	support
0	0.89	0.93	0.91	2484
1	0.81	0.84	0.83	1828
2	0.95	0.92	0.94	1424
3	0.98	0.90	0.94	1368
4	0.55	0.98	0.70	1127
5	0.00	0.00	0.00	924
6	1.00	0.99	0.99	372
accuracy			0.83	9527
macro avg	0.74	0.79	0.76	9527
weighted avg	0.77	0.83	0.79	9527

Neural Networks

We are implementing a few different approaches to selecting the best parameters with Neural networks. Firstly, we are focused on maintaining the speed of neural networks. We chose Mini batches of size 64 to make the matrix multiplication easier. We presented the results of training time and testing time in the table nn1.1.
Secondly, to converge faster over gradient descent we chose different optimizers like Adam and RMSprop to take the weighted averages of weight vectors.
Intuitively, the RMSprop (root mean squared prop) goes as follows,
To understand RMSprop we need to have a basic understanding over gradient descent with momentum and exponentially weighted averages.



Source: researchgate

1. Momentum: When we look at the gradient descent contours, we need our steps to be slower in the vertical direction and faster in the horizontal direction to reach the minimum. To achieve this for every iteration we compute the momentum of each gradient as below.

$$V_{dw} = \beta * V_{dw} + (1 - \beta)d_w$$

$$V_{db} = \beta * V_{db} + (1 - \beta)d_b$$

$$W = W - \alpha * V_{dw}$$

$$b = b - \alpha * V_{db}$$

2. Essentially, this smooths out the update steps, implementing this same process using RMSprop

$$S_{dw} = \beta * S_{dw} + (1 - \beta) d_w^2$$

$$S_{db} = \beta * S_{db} + (1 - \beta) d_b^2$$

$$W = W - \alpha * \frac{d_w}{\sqrt{S_{dw}}}$$

$$b = b - \alpha * \frac{d_b}{\sqrt{S_{db}}}$$

From the original research as suggested, we use $\beta = 0.99$, and α is learning rate as usual. To slow down the update in the vertical direction we divide the b update with the relatively large number when compared to W. i.e., slop is larger in the b direction. Sometimes we also consider a small ε value in the denominator under square root, just in case if the denominator is not 0, to ensure numerical stability.

3. Adam Optimization (Adaptive moment estimation): Adam performs both momentum and RMSprop and combines them as follows

$$V_{dw} = \beta_1 * V_{dw} + (1 - \beta_1) d_w$$

$$V_{db} = \beta_1 * V_{db} + (1 - \beta_1) d_b$$

$$S_{dw} = \beta_2 * S_{dw} + (1 - \beta_2) d_w^2$$

$$S_{dw} = \beta_2 * S_{dw} + (1 - \beta_2) d_w^2$$

In a typical method like Adam, we do a bias correction before we can do a gradient update. With that following,

$$V_{dw}^{corrected} = \frac{V_{dw}}{1 - \beta_1^t}$$

$$V_{db}^{corrected} = \frac{V_{db}}{1 - \beta_1^t}$$

$$S_{dw}^{corrected} = \frac{S_{dw}}{1 - \beta_2^t}$$

$$S_{db}^{corrected} = \frac{S_{db}}{1 - \beta_2^t}$$

$$W = W - \alpha * \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$$

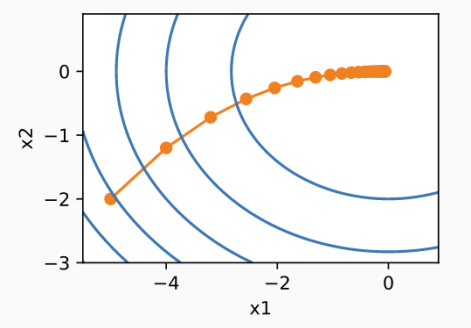
$$b = b - \alpha * \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

This algorithm combines the gradient descent with momentum and gradient descent with RMSprop is commonly used approach for faster convergence with many different types of Neural Networks. There are many hyperparameters in this algorithm,
Basically, the frequently used parameter values are

- $\alpha =$ needs to be tuned
- $\beta_1 = 0.9$ (Moving averages for dw, computes past 10 values averaged)
- $\beta_2 = 0.999$ (Moving averages of dw2)
- $\varepsilon = 10^{-8}$ (Usually not required to use)

One of the things that might speed up learning algorithms Is learning rate decay, to slowly reduce learning rate over time, which is by iterations or by time. We used learning rate decay to reduce the learning rate as the number of mini-batches and iterations increases.

The intuition behind slowly reducing the learning rate is, that initially you can take bigger steps and as the iterations pass you can slow down the learning process to get near to the converging point.



Source: diveintodeeplearning

$$\alpha = \frac{1}{1 + decay_{rate} * epoch} * \alpha_o$$

Here decay rate is also a hyperparameter you can tune, all the above-mentioned hyperparameters and methods can also be used to not stuck at the saddle point on the plateau while updating. The saddle point is one of the local optima, when stuck at the saddle point the Adam makes sure that it slips off the saddle point while learning.
For the problem we are working on, we implemented Neural networks from scratch using NumPy and tuned different parameters to obtain maximum accuracy on test results.

Results:

Accuracy on test data for different methods and data sets

	Accuracy using PCA and Univariate FS	Auto Encoders
Adam_using_weight_decay	86.95%	87.32%
RMSprop_using_L2	87.39%	89.64%
Adam_with_out_mini_batches	87.27%	89.5%
RMSprop_with_out_mini_batches	87.27%	89.52%
Base_model_without_optim_and_minibatches	57.64%	37.32%

Comparing training time and testing time for the different approaches used in project.

	Train Time	Test Time
Adam_using_weight_decay	0:00:02.677103	0:00:00.011885
RMSprop_using_L2	0:00:02.318765	0:00:00.012852
Adam_with_out_mini	0:06:00.423093	0:00:00.016991
RMSprop_with_out_mini_batches	0:06:31.845121	0:00:00.012960
Base_model_without_optim_and_minibatches	0:06:33.429350	0:00:00.017121

“Adam_using_weight_decay” refers to Using Adam optimization with mini-batch gradient descent. Similarly, “RMSprop_using_L2” refers to using RMSprop as an optimizer and L2 regularization.

All the Adam methods are using the Weight Decay method but not L2 regularization. The major difference between L2 regularization **&** weight decay is while the former modifies the gradients to add lambda*W, **weight decay** does not modify the gradients but instead, subtracts learning_rate*lambda*W from the weights in the update step.

Inference:

From the above tables we can infer that using dimensionality reduction techniques such as Variational Autoencoders, we can significantly increase the accuracy of the model.

Performing mini-batches for over 100 iterations/epochs and without mini-batches with 1000 iterations, we can see the difference in train time and test time. It takes about 6 minutes to run for 1000 epochs and less than 2 seconds for 100 epochs of over 100 batches.

The base model learning_rate is set to 0.01 and for RMSprop and Adam, the learning_rate is set to standard and popular value of 3e-4.

We limited it to 1000 iterations to compare the performance of the models, if a greater number of iterations were dedicated to the base model, it would have also given better results. But using optimizers like Adam and RMSprop shows a significant change in latency and accuracy.

FINAL RESULTS

Based on comparisons of all models based on the results and time taken,we would like to propose that Neural Networks,Naive Bayes and Decision Tree perform the best. The reasons being:

1. Time taken for NN is the least with great results.
2. The Decision Tree makes no assumptions about the data and finds optimal numbers for dividing instances.
3. Naive Bayes model is quick and calculates the class probability and can be used in the future for further analysis