Programming Assignment 2

Nishanth Ravula
Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260
nravula@buffalo.edu

1 Back Propagation

Proper training of a Neural Network is the most important aspect of making a reliable model. This training usually associated with the term 'Back propagation', which is highly vague to most process. In general back propagation is the essence of Neural net training. It is the practice of fine tuning of the weights of a neural net based on the error rates i.e., loss obtained in the iteration. If proper tuning is done properly then we can expect lower error rates and can make model reliable by increasing its generalization.

Now let us take a training set

X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

The purpose of training is to build a model that performs the XOR functionality with 2 inputs X1 and X2 and three hidden units. The training set looks like above table.

We need a activation function that gives activation value. In this case we will take an Identity Activation function.

f(a) = a

The activation function will give the activation value at every node in the neural net.

Now to determine the input to the activation function we need a hypothesis function. The Hypothesis function maps the inputs (features) to the outputs (targets). We will be taking the hypothesis function which is very popular i.e.,

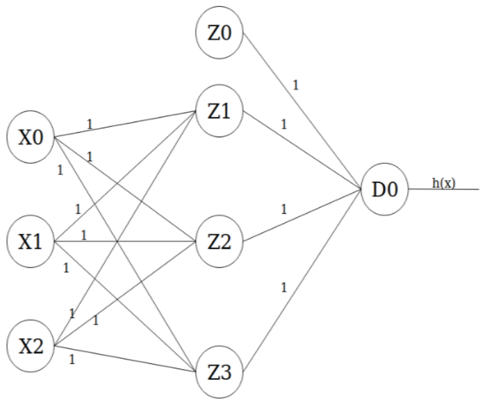
h(X) = sigma (W.X) for all (W, X)

Now let us take chose a loss function, we will be using a usual cost function of Logistic regression. Loss function is a method of evaluating how well your algorithm models your datasets. The cost function of logistic regression is a bit complex but a little simple which is as follows:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

Here we will use Batch gradient descent optimization function to determine what direction to adjust the weights to get the lower loss function. In general gradient descent is an iterative 1st order optimization algorithm used to find a local min/max of the given function. And batch gradient descent calculates the error of each ## in the training dataset, but only updates the model after all training ## have been evaluated.

1.1 Neural Model



We will be having the above neural network for the given dataset. Here X1 and X2 which is the leftmost layer are the input features and input layer. X0 is the bias term for the value 1. Z0, Z1, Z2, Z3 are the hidden layer where Z0 is the bias for the value 1. Finally, D0 is the output of the model whose activation value is the actual output of the model h(x).

1.2 Initiate the forward propagation:

Here the Forward propagation is done in two steps, the steps are as follows:

1. Getting weighted sum of inputs of a particular unit using the h(x).
2. From the step 1 plugging the value we get into the Activation function (f(a) = a) and using Activation function value as the input feature for the connected nodes in the neural layers.

As the initial nodes X0, X1, X2, X3 AND Z0 are not provided with any inputs or do not have any units connected to them, the above steps are not performed to those nodes. The following are the

Unit Z1:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a
=> z = f(1) = 1

Unit Z2:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a
=> z = f(1) = 1

Unit Z3:

h(x) = W0.X0 + W1.X1 + W2.X2
= 1 . 1 + 1 . 0 + 1 . 0
= 1 = a
z = f(a) = a
=> z = f(1) = 1

Unit D0:

h(x) = W0.Z0 + W1.Z1 + W2.Z2 + W3.Z3
= 1 . 1 + 1 . 1 + 1 . 1 + 1 . 1
= 4 = a
z = f(a) = a
=> z = f(4) = 4

Here the Activation value (z) of the final unit is that of the whole model. Therefore, our model predicted an output of 1 for the set of inputs {0, 0}.

We will find the loss/cost function for the current iteration. We can do it by finding the difference between actual_y and predict_y

Loss = actual_y - predicted_y
= 0 - 4
= -4

From here we will be doing Back propagation. We got the loss/cost that is equal to -4 instead of 1. The model prediction is not accurate because loss function is -4.

1.3 Back Propagation:

It is passing the dataset backwards to the neural nodes, here we'll be using Gradient descent so that, in the next iteration the loss/cost function will be the lower than current loss/cost function. We will be using the partial derivation. The forward propagation is done by using f(a) = a and cost function of logistic regression. So by partial derivation we will be using

f'(a) = a

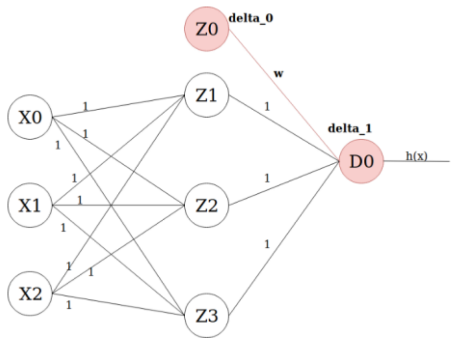
J'(w) = Z . delta

Z is just the z value we obtained from the activation function calculations from the forward feeding. Delta is the the loss/cost of the unit in the layer.

Calculating the Delta:

The loss at each unit and node of the neural network must now be determined. How come? Consider it this way: Every loss that the deep learning model encounters is really the mess that all the nodes combined into one number generated. Therefore, we must identify the node that causes the majority of the loss in each layer so that we may penalize it by assigning it a lower weight value, hence reducing the overall loss of the model. Finding the delta for each and every unit may become complex. There is a shortcut formula for the whole process. The formula is as follows:

delta_0 = w . delta_1 . f'(z)



where the values of the same unit are represented by delta 0, w, and f'(z), whereas delta 1 represents the loss of the unit on the other side of the weighted connection.
In order to calculate a node's loss (for example, Z0), we multiply the value of its corresponding f'(z) by the loss of the node it is linked to in the next layer (delta 1), by the weight of the connection between the two nodes.

In fact, this is how back-propagation functions. We do the delta calculation step at each unit to determine the loss attributable to each node/unit by back-propagating the loss into the neural network.

Now calculation of the deltas:

delta_D0 = total_loss
= -4

delta_Z0 = W . delta_D0 . f'(Z0)
= 1 . (-4) . 1
= -4

delta_Z1 = W . delta_D0 . f'(Z1)
= 1 . (-4) . 1
= -4
delta_Z2 = W . delta_D0 . f'(Z2)
= 1 . (-4) . 1
= -4
delta_Z3 = W . delta_D0 . f'(Z3)
= 1 . (-4) . 1
= -4

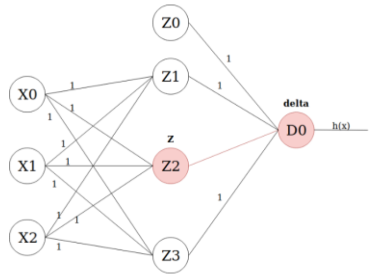
Here Loss of the last unit, or D0, is equivalent to loss of the whole model. Since it is the output unit, its loss represents the total loss of all the units put together. No matter what the input (i.e., z) is equal to, the function f'(z) will always return the value 1. This is due to the partial derivative, which is as follows, as we said earlier: f'(a) = 1. Since the input nodes/units (X0, X1, and X2) do not influence anything in the neural network, they do not have delta values. They serve solely as the neural network's connection to the data collection. Simply said, this is the reason why the entire layer is typically excluded from the layer count.

Updating the Weights:

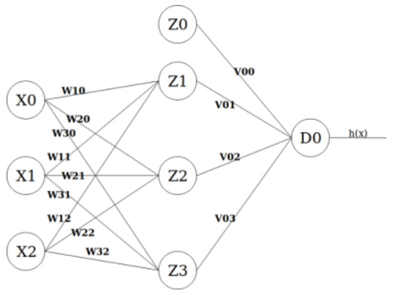
By using the Batch gradient descent formula we need to update al the weights we have in the neural net. The formula is as follows:

W := W - alpha . J'(W)

In this case, alpha is equal to 0.5, the learning rate, and J'(W) is the partial derivative of the cost function J(W) with respect to W. Here, W stands for the current weight. Again we will be using the partial derivation function J'(w) = Z . delta . Where delta is the loss at the unit on the other end of the weighted connection and Z is the Z value gained from forward propagation:
.



Using the partial derivative values we acquire at each step, we now apply the Batch Gradient Descent weight update to all the weights. It is important to note that the input nodes (X0, X1, and X2) have Z values of 1, 0, and 0, respectively. The zeros really represent the feature input values taken from the data set, whereas the 1 represents the bias unit's value. Last but not least, adjusting the weights happens in any sequence. As long as you avoid changing any weight more than once within the same iteration, you may update them in whatever order you choose.



In order to calculate the new weights, let's give the links in our neural nets names as in the fig to the right.

New weight calculation will be as follows:

W10 := W10 - alpha . Z_X0 . delta_Z1
= 1 - 0.1 . 1 . (-4)
= 1.4
W20 := W20 - alpha . Z_X0 . delta_Z2
= 1 - 0.1 . 1 . (-4)
= 1.4
. . . . .
. . . . .
. . . . .
W30 := 1.4

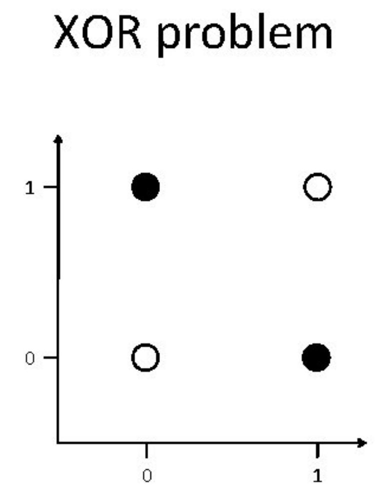
W11 := 1.4
W21 := 1.4
W31 := 1.4
W12 := 1.4
W22 := 1.4
W32 := 1.4V00 := V00 - alpha . Z_Z0 . delta_D0
    = 1 - 0.1 . 1   .   (-4)
    = 1.4
V01 := 1.4
V02 := 1.4
V03 := 1.4

As we just back-propagated via one sample from the training set, it is critical to highlight that the model is not yet adequately trained. Repeating what we did for all the data will result in a model that is more accurate as we go, attempting to come closer to the minimal loss/cost at each stage.

You might not understand why all the weights now have the same value. The nodes will have varied weights based on their contributions to the overall loss if the model is repeatedly trained on various samples.

1.4 Dataset

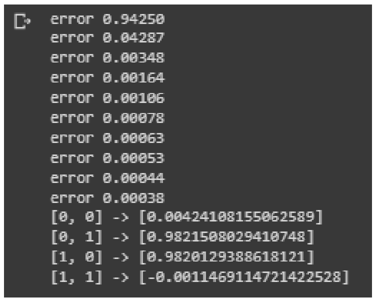
The given dataset for training is known as the XOR problem. The task at hand is to find the decision boundary that accurately distinguishes between all the available classes. It can be clearly seen that the dataset requires a non-linear decision boundary. By changing the activation function in the output layer, the NN can solve this problem.
The dataset also relates closely to the XOR gate used in electronics.



1.5 Experiment and Result

Once the training process is completed, the training loss reduces at every iteration. This can be seen as the output of the given code.

The decrease in the training loss indicates that the model is training continuously at every epoch. At the end of iterations, the model attains local minima and can be used to predict on the test dataset.



```
error 0.94250
error 0.04287
error 0.00348
error 0.00164
error 0.00106
error 0.00078
error 0.00063
error 0.00053
error 0.00044
error 0.00038
[0, 0] -> [0.00424108155062589]
[0, 1] -> [0.9821508029410748]
[1, 0] -> [0.9820129388618121]
[1, 1] -> [-0.0011469114721422528]
```

References

[1] How Does Back-Propagation in Artificial Neural Networks Work?. (2019). Retrieved 21 October 2022, from https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7#:~:text=Back%2Dpropagation%20is%20the%20essence,reliable%20by%20increasing%20its%20generalization.

[2] Wikimedia Foundation. (2022, October 11). Backpropagation. Wikipedia. Retrieved October 30, 2022, from https://en.wikipedia.org/wiki/Backpropagation

[3] Changyou Chen Lecture Slides.

[4] Nielsen, M. A. (1970, January 1). Neural networks and deep learning. Retrieved October 30, 2022, from http://neuralnetworksanddeeplearning.com/chap2.html

[5] Brownlee, J. (2021, October 21). How to code a neural network with backpropagation in Python (from scratch). Machine Learning Mastery. Retrieved October 30, 2022, from https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

2 Support Vector Machine (SVM)

Another straightforward method that any machine learning expert should know how to use is the support vector machine. Many people choose the support vector machine because it offers notable accuracy while using less processing resources. SVM, or Support Vector Machine, is a tool that may be used for both classification and regression problems. However, it is often employed in classification goals.

What is SVM?

Finding a hyperplane in an N-dimensional space (N is the number of features) that categorizes the data points clearly is the goal of the support vector machine method.
There are a variety of different hyperplanes that might be used to split the two classes of data points. Finding a plane with the greatest margin—that is, the greatest separation between data points from both classes—is our goal. Maximizing the margin distance adds some support, increasing the confidence with which future data points may be categorized.

Hyperplanes and Support Vectors

Decision boundaries known as hyperplanes assist in categorizing the data points. Different classes can be given to the data points that fall on each side of the hyperplane. Additionally, the amount of features affects how big the hyperplane is. The hyperplane is essentially a line if there are just two input features. The hyperplane turns into a two-dimensional plane if there are three input features. When there are more than three features, it gets harder to imagine. Closer-to-the-hyperplane data points called support vectors have an impact on the hyperplane's location and orientation. We increase the classifier's margin using these support vectors. The location of the hyperplane will vary if the support vectors are deleted. These are the ideas that support how we construct our SVM.

Large margin intuition

In logistic regression, we use the sigmoid function to condense the value of the output of the linear function to the range [0,1]. If the squished value exceeds a threshold value (0.5), we label it as 1, else we label it as 0. In SVM, we take the output of the linear function and, if it is larger than 1, we classify it into one category, and, if it is less than 1, we classify it into a different category. Since the SVM threshold values are now 1 and -1, we get this reinforcing range of values ([-1,1]) that serves as a margin.

Soft Margin

The hinge loss function is useful for extending SVM to situations when the data are not easily separable linearly.

$$\max\left(0, 1 - y_i(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i - b)\right).$$

Note that y_i is the i-th target (i.e., in this case, 1 or −1), and $w^T x_i$ - b is the i-th output.
This function is zero if the constraint in $y_i(w^T x_i - b) >= 1$, for all $1 <= i <= n$ is satisfied, in other words, if x_i lies on the correct side of the margin. For data on the wrong side of the margin, the function's value is proportional to the distance from the margin.
The goal of the optimization then is to minimize

$$\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\mathbf{w}^{\mathsf{T}}\mathbf{x}_i - b)\right)\right],$$

where the parameter $\lambda > 0$ determines the trade-off between increasing the margin size and ensuring that the x_i lie on the correct side of the margin. Thus, for sufficiently small values of lambda , it will behave similar to the hard-margin SVM, if the input data are linearly classifiable, but will still learn if a classification rule is viable or not. (This parameter lambda is often also called C, e.g. in LIBSVM, but usually refers to the inverse of Lambda.)

Cost Function and Gradient Updates

The goal of the SVM method is to increase the distance between the data points and the hyperplane. Hinged loss is the loss function that aids in maximizing the margin.

There is no cost if the expected value and the actual value have the same sign. If not, we proceed to calculate the loss amount. We also provide a regularization parameter to the cost function. The purpose of the regularization parameter is to find a balance between loss minimization and margin maximization. When the regularization parameter has been applied, the cost functions are as follows.

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

We take partial derivatives with regard to the weights now that we know the loss function to discover the gradients. Our weights may be updated using the gradients.

$$\frac{\delta}{\delta w_k} \lambda \parallel w \parallel^2 \quad = 2\lambda w_k$$

$$\frac{\delta}{\delta w_k} \big(1 - y_i \langle x_i, w \rangle\big)_+ \quad = \begin{cases} 0, & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ -y_i x_{ik}, & \text{else} \end{cases}$$

When there is no misclassification, that is, when our model predicts the class of our data point accurately, all that is required of us is to update the gradient based on the regularization parameter.

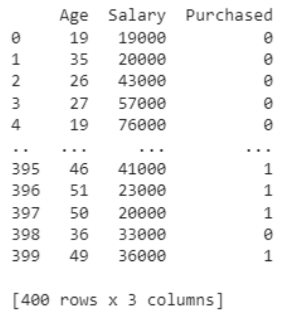
$$\boldsymbol{w = w - \alpha \cdot (2\lambda w)}$$

In order to execute a gradient update when there is a misclassification, or when our model incorrectly predicts the class of a data point, we add the loss along with the regularization parameter.

$$\boldsymbol{w = w + \alpha \cdot (y_i \cdot x_i - 2\lambda w)}$$

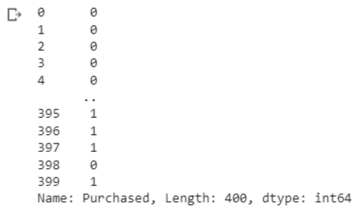
2.1 Dataset

Let's use a binary dataset to train our model. In this case, we will use customer data about whether the customer had purchased a product or not. The below is the sample dataset with 27 rows.



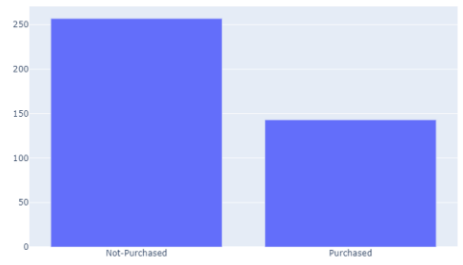
```
     Age  Salary  Purchased
0     19   19000          0
1     35   20000          0
2     26   43000          0
3     27   57000          0
4     19   76000          0
..   ...     ...        ...
395   46   41000          1
396   51   23000          1
397   50   20000          1
398   36   33000          0
399   49   36000          1

[400 rows x 3 columns]
```

We can print out the target/output class to verify that the data we are using is a binary set (containing only two output categories). Notice that the output class contains either 0 or 1, showing whether the customer had purchased the product or not.



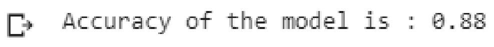
```
0      0
1      0
2      0
3      0
4      0
      ..
395    1
396    1
397    1
398    0
399    1
Name: Purchased, Length: 400, dtype: int64
```

The image on the right gives that more people have not purchased the product from the giving dataset.



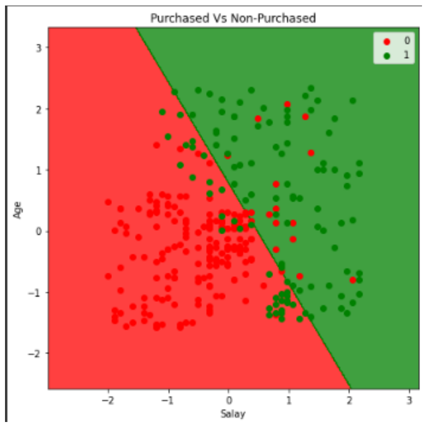
2.2 Experiment and Result:

Once we are done with the pre-processing of the data, we can move into the splitting part to divide the data into the testing and training parts. We have assigned 25% of the data to the testing and 75% to the training parts. That means our model will use 75% of the original data for training, and the remaining portion will be used to test the model to know how accurate our model predicts the output class. Before feeding the training data to our model, we need to scale the given data so that the outlier will not affect the output class. scaling is only applied to the input/independent variables. Once the scaling is done, our data is then ready to be used to train our model. After the training, we need to provide the testing data to see how well our model predicts. We're storing predicted outputs in the **y_pred** variable. We can then use these predicted outputs to find the accuracy of our model. And running the accuracy snippet we got an accuracy of 88 percent which is good, which means our model is predicting good data.

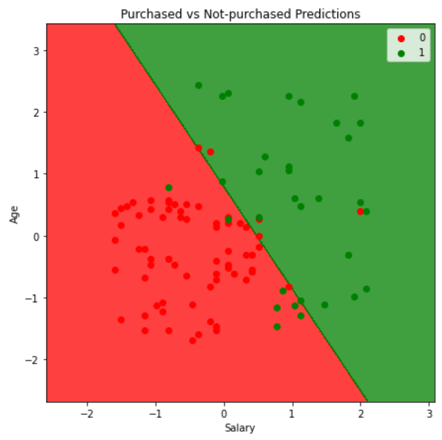


```
Accuracy of the model is : 0.88
```

Let's visualize the model trained by the Linear Kernel to see how the model has been trained visually. Notice that there is a linear boundary between the two classes because we have specified the Kernel to be linear.



Similarly, we can also visualize the predictions of our model, bypassing the testing dataset. You can consider any testing point in the red area as Not-purchased and any point in the green area as Purchased.



References:
[1] Support vector machines — soft margin formulation and kernel trick. (n.d.). Retrieved October 31, 2022, from https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe

[2] Wikimedia Foundation. (2022, October 19). Support Vector Machine. Wikipedia. Retrieved October 30, 2022, from https://en.wikipedia.org/wiki/Support_vector_machine
[3] Alam, B. A. T. is B. (2022, June 26). Implementation of Support Vector Machine (SVM) using Python. Hands. Retrieved October 30, 2022, from https://hands-on.cloud/implementation-of-support-vector-machine-svm-using-python/

[4] Nair, A. (2021, June 1). Understanding the basics of SVM with example and python implementation. Analytics India Magazine. Retrieved October 30, 2022, from https://analyticsindiamag.com/understanding-the-basics-of-svm-with-example-and-python-implementation/

[5] Changyou Chen Lecture Slides

[6] Support Vector Machine — introduction to machine learning algorithms ... (n.d.). Retrieved October 31, 2022, from https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[7] 1.4. Support Vector Machines. scikit. (n.d.). Retrieved November 1, 2022, from https://scikit-learn.org/stable/modules/svm.html

3 Navies Bayes Classification

The Bayes' Theorem is the foundation of the Naive Bayes classifier, which has been modified for use with various machine learning issues. These consist of network analysis, categorization, and clustering. The usage of Naive Bayes for classification issues that fall under the supervised branch of the machine learning tree.

Naive The core element of Bayes' theory is that the predictors—the qualities or independent variables—are independent of one another. This is a huge assumption since it is simple to demonstrate that there is frequently at least some connection between variables in actual life. Bayes classification is considered "naive" because to this assumption of independence.

Despite the independence requirement, it has been repeatedly demonstrated that the Naive Bayes algorithm excels at classification issues. In addition, it is a quick technique since it scales well to incorporate several predictors without needing to deal with multi-dimensional correlations.

Conditional Probability

Assume we have a bucket filled with red and black balls. In total, there are 15 balls: 7 red and 8 black.
The probability of randomly picking a red ball out of the bucket is 7/15. You can write it as

P(red) = 7/15.

If we were to draw balls one at a time without replacing them, what is the probability of getting a black ball on a second attempt after drawing a red one on the first attempt?

You can see that the above question is worded to provide us with the condition that needs to be satisfied first before the second attempt is made. That condition says that a red ball must be drawn during the first attempt.
As stated earlier, the probability of getting a red ball on the first attempt

(P(red)) is 7/15.

That leaves 14 balls inside a bucket with 6 red and 8 black.
Hence, the probability of getting a black ball next is
8/14 = 4/7.

We can write this as a conditional probability:
P(black|red) = 4/7. (read: probability of black given red)
We can also see that
P(red and black) = P(red) * P(black|red)
= 7/15 * 8/14 = 4/15.
Similarly,
P(black and red) = P(black) * P(red|black)
= 8/15 * 7/14 = 4/15.

Bayes' theorem

The Bayes' theorem helps us calculate conditional probabilities of an event when we know the likelihood of a reverse event. Using the example above, we would write it as follows:

$$\underset{\text{Posterior}}{P(black \mid red)} = \frac{\overset{\text{Prior}}{P(black)} * \overset{\text{Likelihood}}{P(red \mid black)}}{\underset{\text{Evidence}}{P(red)}}$$

Naive Bayes classifier

Let’s now take the above equation and change the notation to make it more relevant for classification problems.

$$P(C_k \mid x) = \frac{P(C_k) * P(x \mid C_k)}{P(x)}$$

- P(C|x) is the posterior probability of class C (target variable) given the predictor x (attribute / independent variable);
- P(C) is the prior probability of class C;
- P(x|C) is the likelihood, which is the probability of the predictor x given class C;
- P(x) is the prior probability of the predictor x;
- Little k is just the notation to distinguish between different classes as you would have at least 2 separate classes in the classification scenario (e.g., spam / not-spam, red ball / black ball).

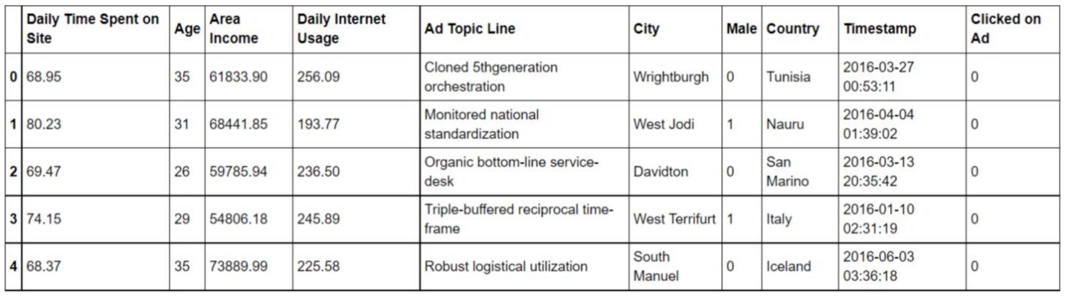
Since the denominator of the aforementioned equation does not depend on C, in reality, only the numerator is of relevance. Additionally, the denominator is virtually a constant because all of the values for the property x are known.

In light of the foregoing, assuming independence, and accounting for numerous predictors, the classification equation is as follows:

$$P(C_k \mid X) = P(C_k) * P(x_1 \mid C_k) * P(x_2 \mid C_k) * \ldots * P(x_n \mid C_k)$$
$$P(C_k \mid X) = P(C_k) * \prod_{i=1}^{n} P(x_i \mid C_k)$$

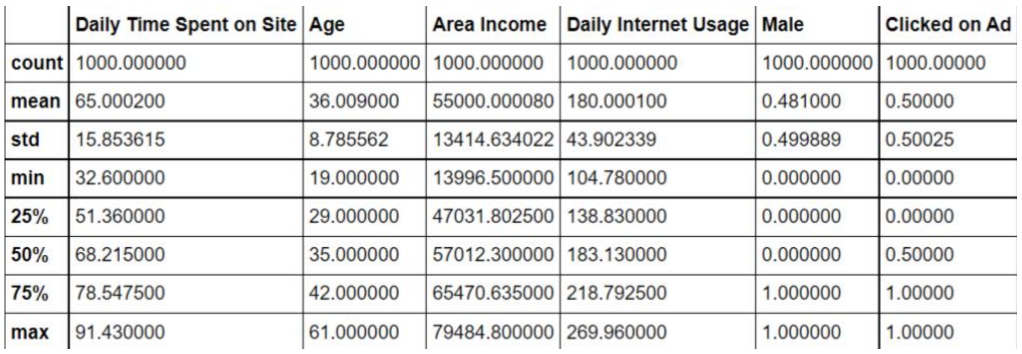
3.1 Dataset and Experiment:

We are using the Advertisement clicking dataset (about users clicking the ads or not)
The dataset has the following features,
Daily Time Spent on Site — Amount of time spent on the website
Age — User’s Age
Area Income — Avg revenue of the Users
Daily Internet Usage — Avg usage of internet daily
Ad Topic Line — Topic text of the advertisement
City — City of the Users
Male — gender of the users(male or female)
Country — Country of the users
Timestamp — Time clicked on the Ad
Clicked on Ad — 0 or 1, 0-not clicked,1-clicked.



	Daily Time Spent on Site	Age	Area Income	Daily Internet Usage	Ad Topic Line	City	Male	Country	Timestamp	Clicked on Ad
0	68.95	35	61833.90	256.09	Cloned 5thgeneration orchestration	Wrightburgh	0	Tunisia	2016-03-27 00:53:11	0
1	80.23	31	68441.85	193.77	Monitored national standardization	West Jodi	1	Nauru	2016-04-04 01:39:02	0
2	69.47	26	59785.94	236.50	Organic bottom-line service-desk	Davidton	0	San Marino	2016-03-13 20:35:42	0
3	74.15	29	54806.18	245.89	Triple-buffered reciprocal time-frame	West Terrifurt	1	Italy	2016-01-10 02:31:19	0
4	68.37	35	73889.99	225.58	Robust logistical utilization	South Manuel	0	Iceland	2016-06-03 03:36:18	0

The above table represents the head of the data set and the below table gives the description of the loaded dataset.



	Daily Time Spent on Site	Age	Area Income	Daily Internet Usage	Male	Clicked on Ad
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.00000
mean	65.000200	36.009000	55000.000080	180.000100	0.481000	0.50000
std	15.853615	8.785562	13414.634022	43.902339	0.499889	0.50025
min	32.600000	19.000000	13996.500000	104.780000	0.000000	0.00000
25%	51.360000	29.000000	47031.802500	138.830000	0.000000	0.00000
50%	68.215000	35.000000	57012.300000	183.130000	0.000000	0.50000
75%	78.547500	42.000000	65470.635000	218.792500	1.000000	1.00000
max	91.430000	61.000000	79484.800000	269.960000	1.000000	1.00000

To get a better Naïve Bayes Model we need to drop some of the unnecessary columns. We will be dropping the following columns

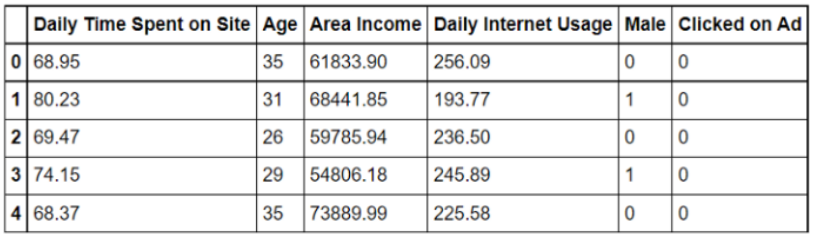
'Ad Line Topic'

'City',

'Country' and

Timestamp.

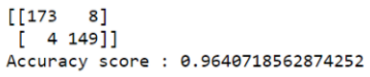
After deletion of the data the table looks as follows,



	Daily Time Spent on Site	Age	Area Income	Daily Internet Usage	Male	Clicked on Ad
0	68.95	35	61833.90	256.09	0	0
1	80.23	31	68441.85	193.77	1	0
2	69.47	26	59785.94	236.50	0	0
3	74.15	29	54806.18	245.89	1	0
4	68.37	35	73889.99	225.58	0	0

Next, we can then split the dataset into train and test and the test size is fixed as 1/3 or 0.33.

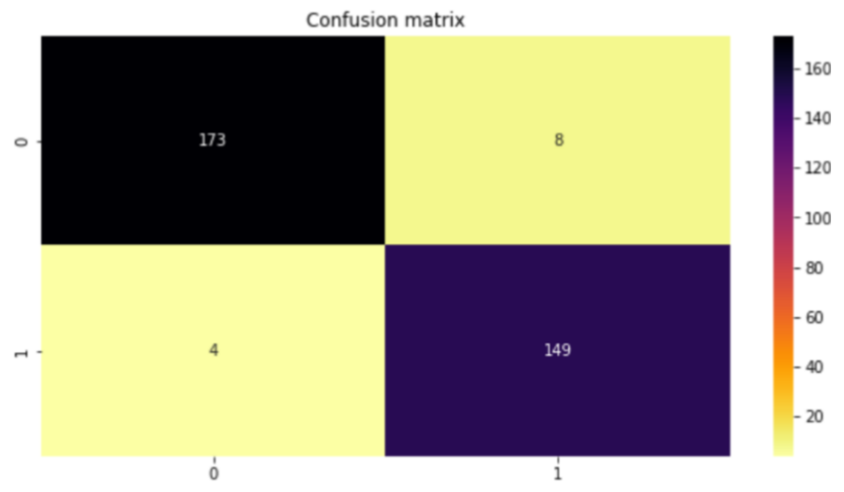
Running the accuracy snippet, we got an accuracy of 96 percent which is good, which means our model is predicting good data. The Naive Bayes Classification model was performed well for the advertisement dataset.



```
[[173   8]
 [  4 149]]
Accuracy score : 0.9640718562874252
```

From the result of confusion matrix, Let’s visualize the confusion matrix using heatmaps. The representation of confusion matrix will look as following.

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f19695eb5d0>
```



Reference:

[1] Wikimedia Foundation. (2022, October 29). Naive Bayes classifier. Wikipedia. Retrieved October 30, 2022, from https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[2] Changyou Chen Lecture Slides

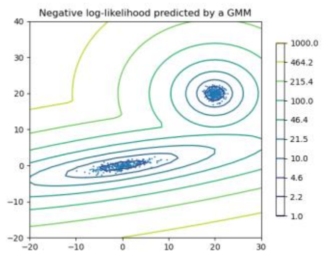
[3] Alam, B. A. T. is B. (2022, June 26). Implementing naive Bayes classification using Python. Hands. Retrieved October 30, 2022, from https://hands-on.cloud/implementing-naive-bayes-classification-using-python/

[4] Naive Bayes classification - python in plain english. (n.d.). Retrieved October 31, 2022, from https://python.plainenglish.io/introduction-to-naive-bayes-classification-and-implementation-in-python-fc7858287fea

[5] Sawla, S. (2018, June 9). Introduction to naive bayes for classification. Medium. Retrieved October 30, 2022, from https://medium.com/@srishtisawla/introduction-to-naive-bayes-for-classification-baefefb43a2d

4 Gaussian Mixture Model

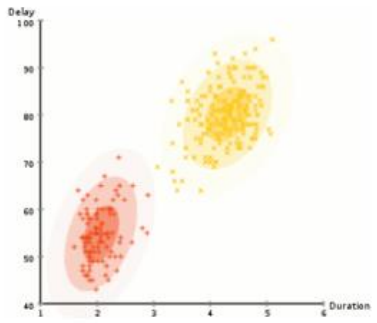
The Gaussian Mixture Models (GMM) algorithm is an unsupervised learning algorithm since we do not know any values of a target feature. Further, the GMM is categorized into the clustering algorithms, since it can be used to find clusters in the data. A Gaussian mixture model is a probabilistic model that presupposes that the data points were produced through a combination of a limited number of Gaussian distributions with unidentified characteristics. Mixture models may be thought of as a generalization of k-means clustering that takes into account both the centers of the latent Gaussian ellipses and the covariance structure of the data. The image in the left is the Two-component Gaussian mixture model.



The biggest limitation of K-Means clustering is that each cluster has the same diagonal covariance matrix. This produces spherical clusters that are quite inflexible in terms of the types of distributions they can model. GMM solves this issue by forming clusters of different elongated widths.
Another point to note is that K-Means clustering follows hard clustering i.e each point in our dataset can belong to only one cluster. It does not give the probability distribution of each point belonging to all clusters.

In order to fit a mixture of Gaussian models, the Gaussian Mixtures object uses the exception-maximization (EM) algorithm. Additionally, it can compute the Bayesian Information Criterion to determine how many clusters there are in the data and create confidence ellipsoids for multivariate models. We must add a pseudo-parameter as required by the EM technique in order to model the unknown latent variables Ci. Since Ci can have k discrete values, this new parameter will be a n x k matrix with wij as each member. Each element of this matrix represents the likelihood that the i-th data point originated from cluster j. This pseudo-parameter is not a real parameter of the model because it is only utilized during model fitting and then ignored.
The EM algorithm then proceeds iteratively, with each iteration being divided into two steps: the E-step and the M-step. In the E-step, we use our current best knowledge of the centers and shapes of each cluster to update our estimates of which data point came from which class. Concretely, we hold μj and Σj fixed and update wij and ϕ. In the M-step, we use our current best knowledge of which class each point belongs to to update and improve our estimates for the center and shape of each cluster. Concretely, we use wij as sample weights when updating μj and Σj by taking weighted averages over X. For example, if w11=0.01 and w11=0.99 we know that the data point X1 is unlikely to be in class 1, but very likely to be in class 2. Therefore, when estimating the center of the first class μ1 we give X1 almost negligible weight, but when estimating the center of the second class μ2 we give X1 almost full weight. This “pulls” the center of each cluster towards those data points which are considered likely to be part of that cluster.

Visually, the iterative process looks something like this:



E-step
Given the that centroid μj and covariance matrix Σj for class j is fixed, we can update wij by simply calculating the probability that Xi came from each class and normalizing:

$$w_{ij} = \frac{P(X_i|K=j)}{P(K_i)} = \frac{P(X_i|K=j)}{\sum_{l=1}^{k} P(X_i|K=l)}$$

The conditional probablity P(Xi|K=j) is simply the multivariate normal distribution Xi N(μi,Σi) so we can use equation Xi~N(μCi,ΣCi) above to calculate the probability density for each class, and then divide through by the total to normalize each row of X to 1. This gives us a concrete formula for the update to wij:

$$w_{ij} = \frac{f_{\mathcal{N}(\mu_i,\Sigma_i)}(\mathbf{X}_i)}{\sum_{l=1}^{k} f_{\mathcal{N}(\mu_l,\Sigma_l)}(\mathbf{X}_i)}$$

The probability of each class ϕ can then be estimated by averaging over all examples in the training set:

$$\phi_j = \sum_{i=1}^{n} w_{ij}$$

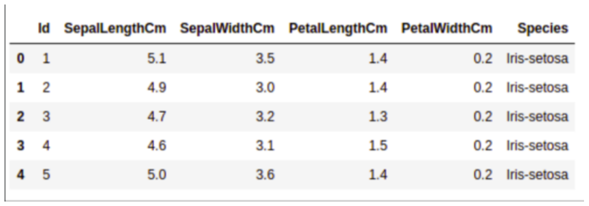
M-step
Forget about the past estimates we had for μj or Σ. Unlike gradient descent, the EM algorithm does not proceed by making small changes to the previous iteration's parameter estimates - instead, it makes a bold leap all the way to the exact estimate - but only in certain dimensions. In the M-step, we will calculate the ML estimates for μj or Σ assuming that wij is held constant. we have a matrix of n observations Xi with weights wi which we believe came from a multivariate distribution N($\vec{\mu}$,Σ) That means we can use the familiar formulas:

$$\boldsymbol{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}\mathbf{X}_i$$

$$\Sigma_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}(\mathbf{X}_i - \boldsymbol{\mu}_j)(\mathbf{X}_i - \boldsymbol{\mu}_j)^T$$

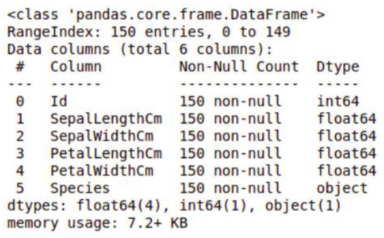
4.1 Dataset:

We will be using be using the famous iris dataset with five rows and four columns. The columns are sepal length in cm, sepal width in cm, petal length in cm, petal width in cm. Iris is a flowering plant, the researchers have measured various features of the different iris flowers and recorded them digitally.



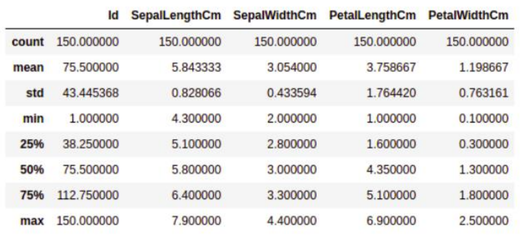
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

Now, let's also the columns and their data types. For this, we will use the "info()" method.



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             150 non-null    int64  
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object 
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

We can see that only one column has categorical data and all the other columns are of the numeric type with non-Null entries. Let's get a quick statistical summary of the dataset using the describe() method. The describe() function applies basic statistical computations on the dataset like extreme values, count of data points standard deviation, etc. Any missing value or NaN value is automatically skipped. describe() function gives a good picture of the distribution of data.

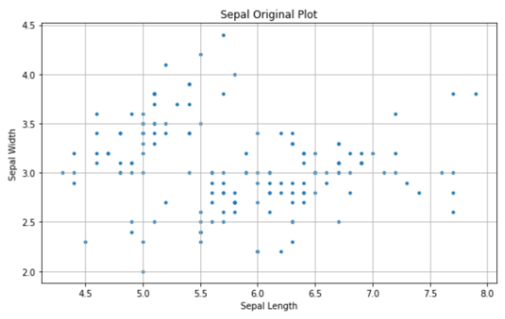


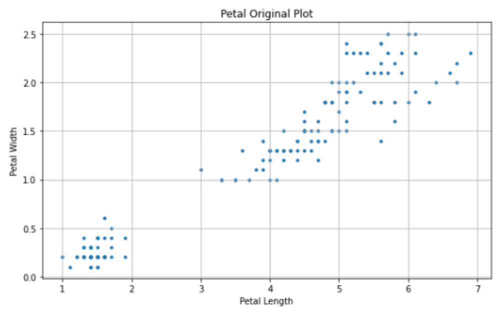
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000

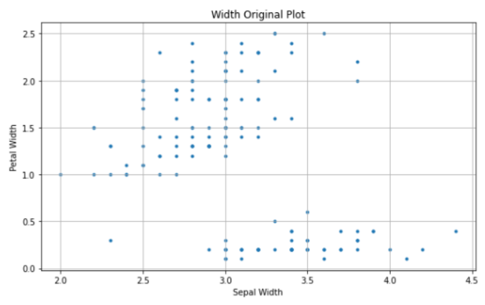
We can see the count of each column along with their mean value, standard deviation, minimum and maximum values.

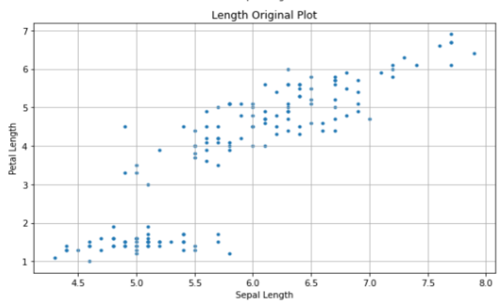
4.2 Process

The working implementation is straight forward. The below program corresponds almost one-to-one (one line of code for one equation) with the above mathematics. The equations $w_{ij} = \frac{f_{\mathcal{N}(\mu_i,\Sigma_i)}(\mathbf{X}_i)}{\sum_{l=1}^{k} f_{\mathcal{N}(\mu_l,\Sigma_l)}(\mathbf{X}_i)}$, $\phi_j = \sum_{i=1}^{n} w_{ij}$ are used in the e_step() method and equations $\boldsymbol{\mu}_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}\mathbf{X}_i$ and $\Sigma_j = \frac{1}{n}\sum_{i=1}^{n} w_{ij}(\mathbf{X}_i - \boldsymbol{\mu}_j)(\mathbf{X}_i - \boldsymbol{\mu}_j)^T$ are used in the m_step() method. One detail I did not treat above is initialization - while ϕ and wij can use simple uniform initialization, for μ it is better to choose a random index ij uniformly from 1 to n for each class and then initialize μj=Xij. This ensures that each cluster centroid is inside the support of the underlying distribution and that they are initially spread out randomly throughout the space. Now using the plot we will plot the clusters of the each column petal, sepal, width and length. Initially we will plot the original plots of the dataset. The following are the plots of the original datasets.







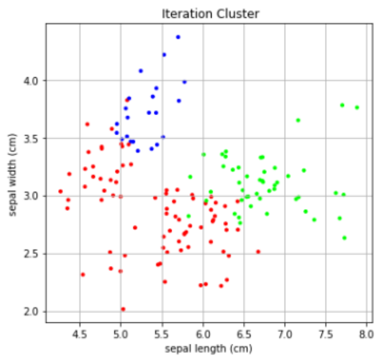


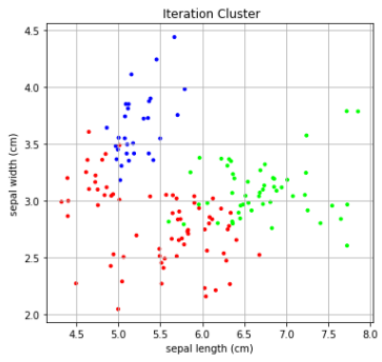
From the original plot we can find that the data is not properly clustered so, by using GMM we will cluster the dataset with proper accuracy. As it is unsupervised, we will be using GMM to classify the dataset into proper clusters. After iterating all the plots using the algorithm, we will do some predictions. Those predicted data will give the accuracy of the permuted prediction against target before the iteration. Now we'll fit a mixture of Gaussians to this data using our implementation of the EM algorithm. As with k-means, it is important to ask how we obtain an initial configuration of mixing weights and component parameters. In this simple case, we'll take three points to be the initial cluster means, use the empirical covariance of the data to be the initial covariance in each cluster (a clear overestimate), and set the initial mixing weights to be uniform across clusters. Those results look as follows

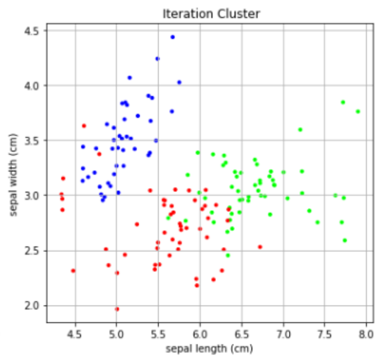
The following sections will evaluate the results by asking the following questions:
Convergence: How did the log likelihood change across iterations? Did the algorithm achieve convergence?
Uncertainty: How did cluster assignment and uncertainty evolve?
Interpretability: Can we view some example images from each cluster? Do these clusters correspond to known image categories?

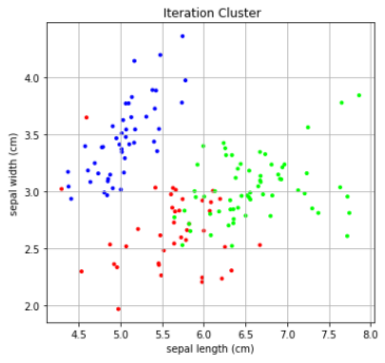
```
The accuracy of the permuted prediction against target before iteration  1:  0.6266666666666667

The accuracy of the permuted prediction against target before iteration  2:  0.68

The accuracy of the permuted prediction against target before iteration  3:  0.7866666666666666

The accuracy of the permuted prediction against target before iteration  4:  0.8533333333333334

The accuracy of the permuted prediction against target before iteration  5:  0.86

The accuracy of the permuted prediction against target before iteration  6:  0.8733333333333333

The accuracy of the permuted prediction against target before iteration  7:  0.8866666666666667

The accuracy of the permuted prediction against target before iteration  8:  0.9

The accuracy of the permuted prediction against target before iteration  9:  0.9466666666666667

The accuracy of the permuted prediction against target before iteration  10:  0.9733333333333334

The accuracy of the permuted prediction against target before iteration  11:  0.96

The accuracy of the permuted prediction against target before iteration  12:  0.9533333333333334

The accuracy of the permuted prediction against target before iteration  13:  0.9666666666666667

The accuracy of the permuted prediction against target before iteration  14:  0.9666666666666667
```

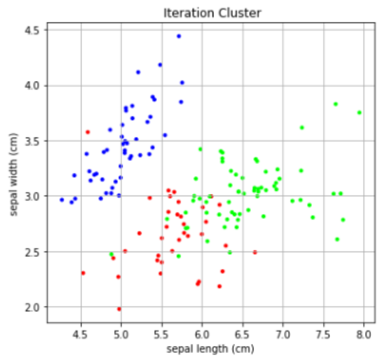
Lets also visualize the plot of each iteration where we can find the different clusters with red, blue and green colors in the plot. Where the x-axis is sepal length in centimeter and y-axis is sepal width in centimeters. We provide a function to calculate log likelihood for mixture of Gaussians. The log likelihood quantifies the probability of observing a given set of data under a particular setting of the parameters in our model. We will use this to assess convergence of our EM algorithm; specifically, we will keep looping through EM update steps until the log like-hood ceases to increase at a certain rate.

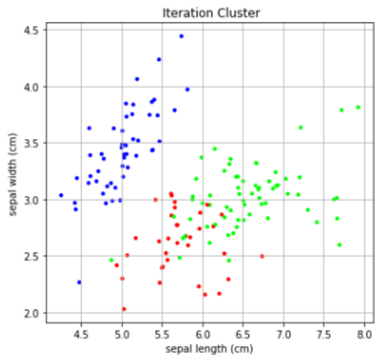


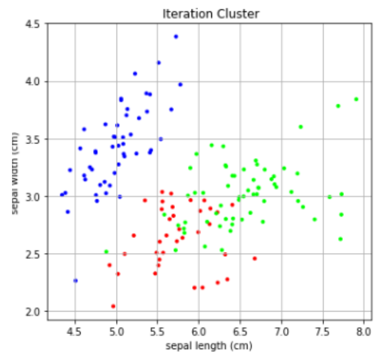


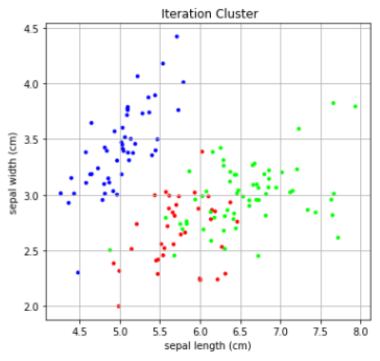


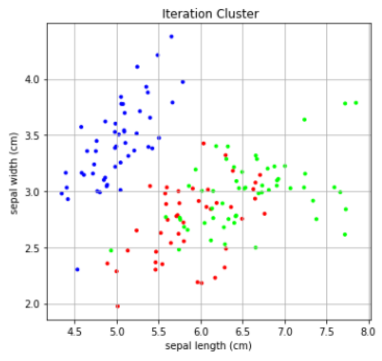


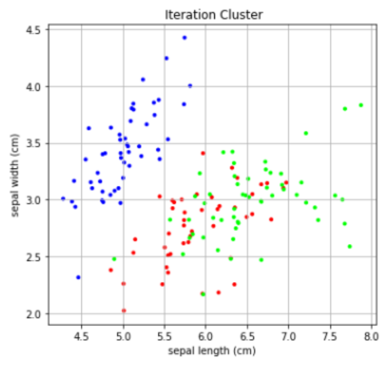


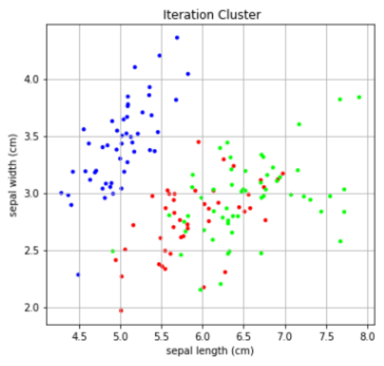


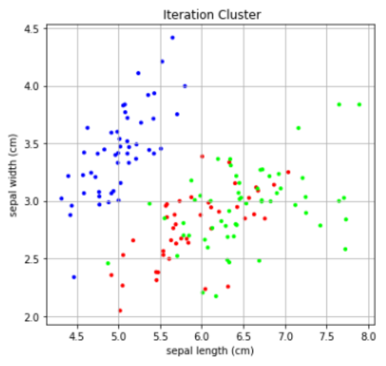


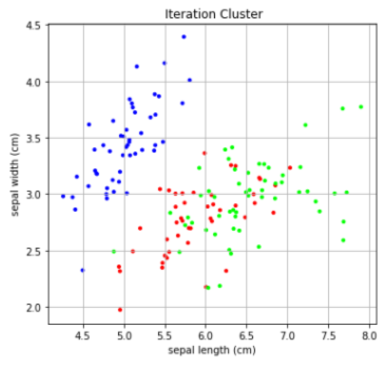


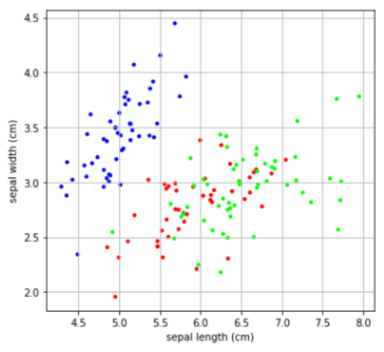




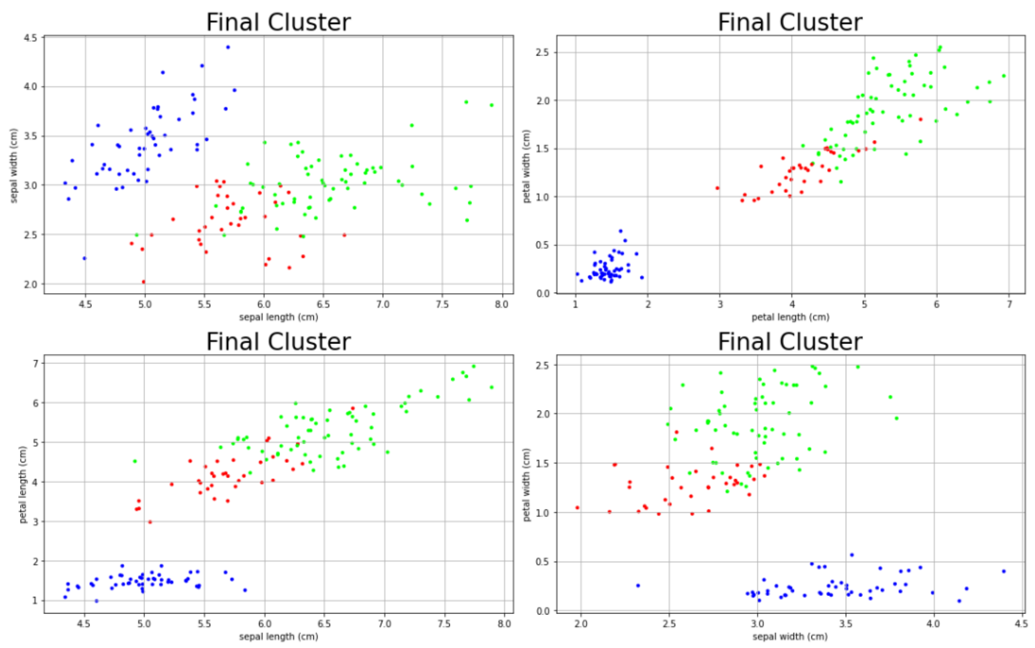








The final plot of the model prediction is as follows we are getting proper clusters for dataset which means our modal is prediction is good. The final plot contains both sepal and petal plots.



Reference:

[1] Looney, O. (2019, June 5). ML from scratch, part 5: Gaussian mixture models. OranLooney.com. Retrieved October 31, 2022, from http://www.oranlooney.com/post/ml-from-scratch-part-5-gmm/

[2] Changyou Chen Lecture Slides

[3] Wikimedia Foundation. (2022, October 20). Mixture model. Wikipedia. Retrieved October 31, 2022, from https://en.wikipedia.org/wiki/Mixture_model

[4] 2.1. gaussian mixture models. scikit. (n.d.). Retrieved October 31, 2022, from https://scikit-learn.org/stable/modules/mixture.html

[5] Nirek, R. (2020, June 23). Gaussian mixture models(gmm). Medium. Retrieved October 31, 2022, from https://medium.com/swlh/gaussian-mixture-models-gmm-1327a2a62a

[6] Gaussian mixture models with Python - Towards Data Science. (n.d.). Retrieved October 31, 2022, from https://towardsdatascience.com/gaussian-mixture-models-with-python-36dabed6212a