**Introduction:**

The purpose of this final project is to explore and compare 3 methods of machine learning. The 3 methods discussed will be K-Means clustering, Logistic Regression and Neural Networks. All 3 of these methods will be applied to the same data set consisting of 150 data samples of 3 different Iris flower species. Each sample consists of the measurements of the width and length of the petal and sepal of all 150 flowers. These measurements will be used as the features by which to apply a machine learning algorithm. The output of these models will be a prediction of the species of the Iris flower. The data set of 150 Iris flowers will be used for both model training and validation in k-means clustering, logistic regression and neural network machine learning methods.

**PART 1: DATA EXPLORATION AND PREPROCESSING**

# Part 1: Data Visualization

# Code from ECE 3831 Final Project Help

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import load\_iris

import pandas as pd

# Load the Iris dataset

iris = load\_iris()

iris\_df = pd.DataFrame(data = iris.data, columns = iris.feature\_names)

# Get the shape of the dataset for future use

print('The Shape of the Iris Dataset is: ', iris\_df.shape)

iris\_df['species'] = pd.Categorical.from\_codes(iris.target, iris.target\_names)

# Set the aesthetic style of the plots

sns.set(style ="whitegrid")

# Four Feature Visualization Using Histograms for Each Feature

plt.figure(figsize=(12, 8))

for index, feature\_name in enumerate(iris.feature\_names):

# Create a subplot for each feature

plt.subplot(2, 2, index + 1)

sns.histplot(iris\_df[feature\_name], kde=True, bins=20)

plt.title('Histogram of ' + feature\_name)

plt.xlabel(feature\_name)

plt.ylabel('Count')

plt.tight\_layout()

plt.show()

# Four Feature Visualization Using Scatter Plots for Each Pair of Features

# Pairplot plots the column's relationships, used in multivariate analysis

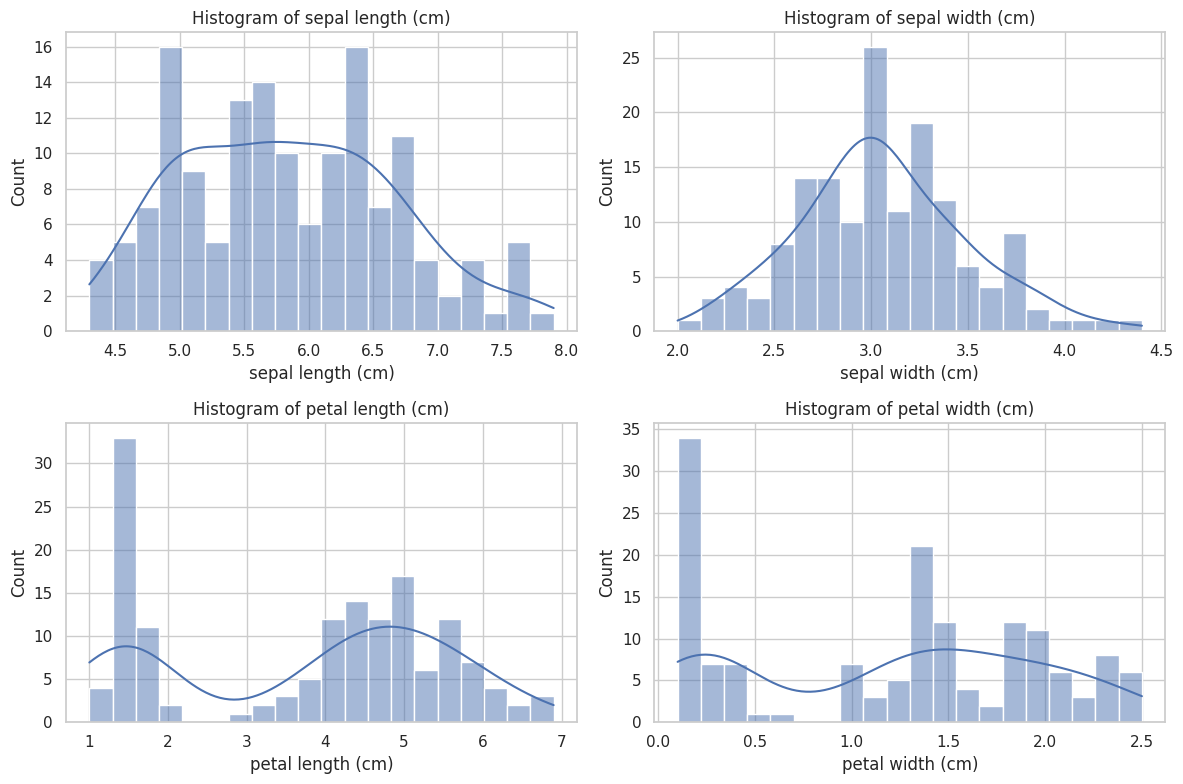
sns.pairplot(iris\_df, hue="species", markers=["o", "s", "D"])

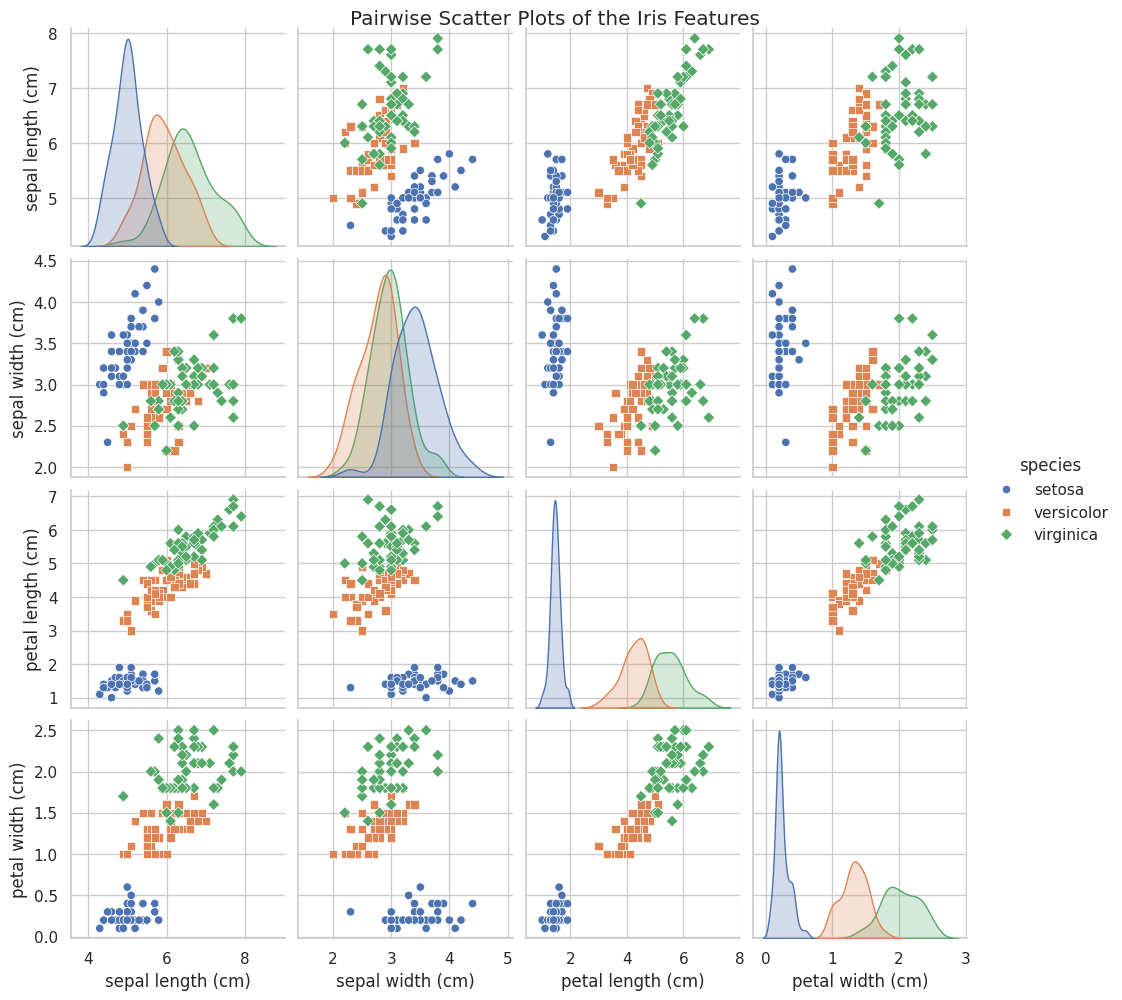
plt.suptitle('Pairwise Scatter Plots of the Iris Features', verticalalignment='bottom')

plt.show()

https://colab.research.google.com/drive/1\_2WiUXkU-vRnja4omTDIVwDeDNAmfyEC?usp=sharing

The Shape of the Iris Dataset is: (150, 4)





**DATA PREPROCESSING:** Standardize or normalize the data to prepare for subsequent analysis.

# Part 1: Data Preprocessing. Standardize the data to prepare for subsequent analysis.

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

import pandas as pd

# Load the Iris dataset

iris = load\_iris()

x = iris.data

y = iris.target

# Standardize the data

scaler = StandardScaler()

x\_scaled = scaler.fit\_transform(x)

**PART 2: K-MEANS CLUSTERING**

**Google Colab for K-Means Clustering code:**

<https://colab.research.google.com/drive/1EWCfdlWKcCucGcFp-usp9YyEuIkfglNO?usp=sharing>

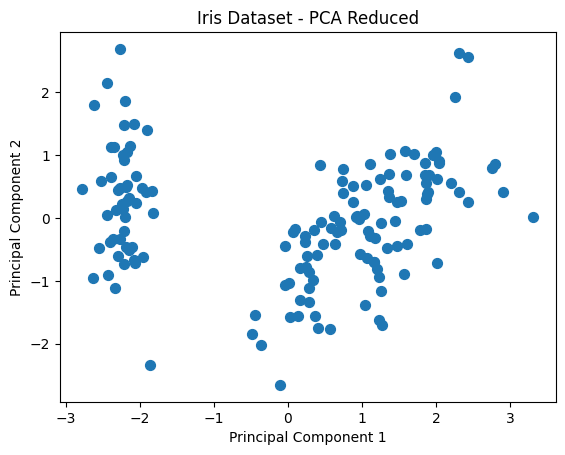
**K-Means Clustering Implementation:**

The K-means clustering algorithm was implemented as discussed in Lecture 9. The initial centroids are picked randomly from the dataset. Then it uses a distance matrix to hold the euclidean distances of each data point to each centroid and a bookkeeping matrix of zeros and ones to mark the closest centroid. The new centroids are calculated and replace the previous ones.

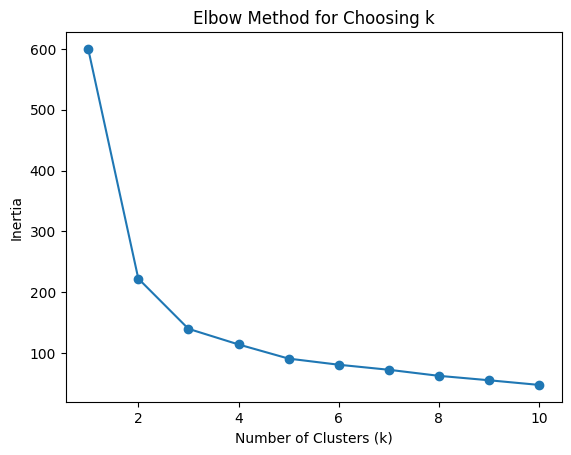
The distance and bookkeeper matrices are multiplied and the Frobenius norm is calculated. This is used as the cost function within the K-means algorithm. If the cost difference meets the stopping criteria, the run ends. Otherwise, it repeats from the distance matrix. The inertia is also calculated and stored to give a better comparison to the Scikit-Learn implementation and is used to determine the best set of centroids from multiple runs of the algorithm.

**Analysis:**

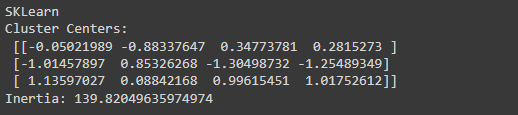
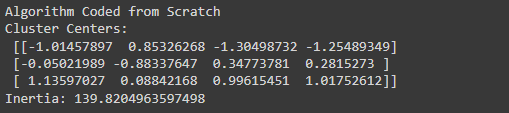
In order to better visualize the Iris data, which is in 4 dimensions, here is the normalized data plotted in 2 dimensions:



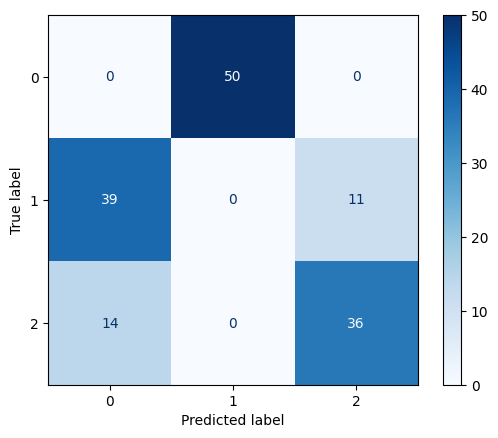
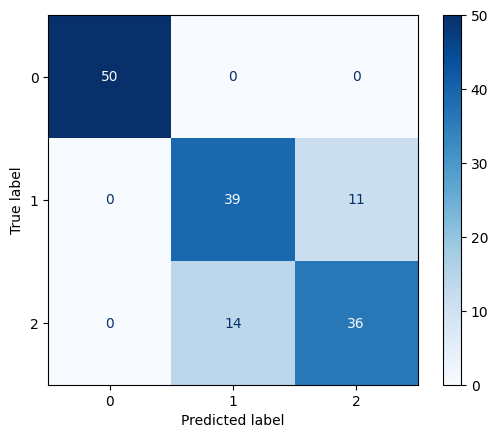
At first glance, it looks to be 2 distinct clusters. We know that it is in actuality 3 clusters, but if we did not know this beforehand, we would determine the best *k* value using an elbow curve:

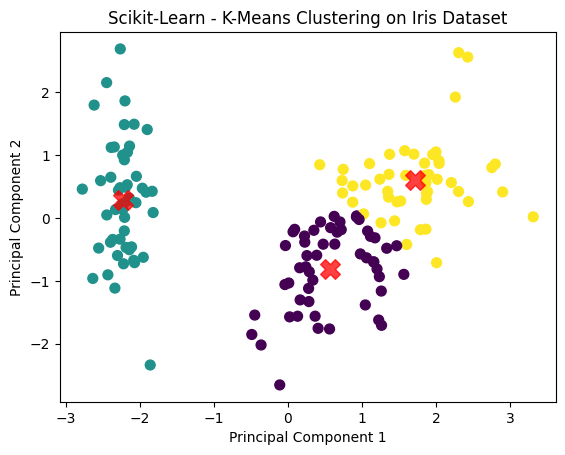
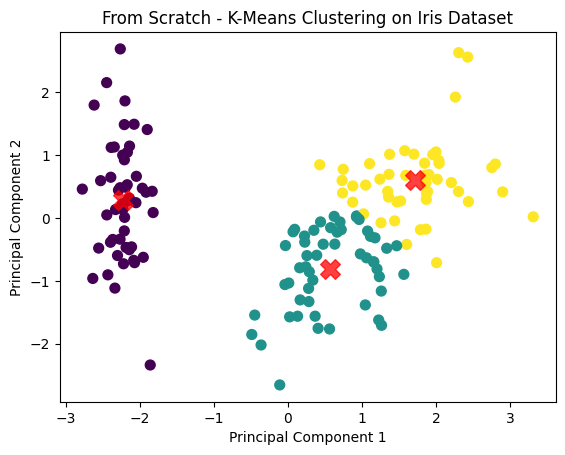


We choose the value for *k* at which the elbow bends, i.e. where both the inertia difference from the previous value is high and the difference to the next value is low. In this case, it is 3. Using this *k*, we cluster the dataset using both the algorithm coded from scratch and Scikit-Learn. Here are the results of the clustering from running each implementation 10 times and selecting the best by inertia:

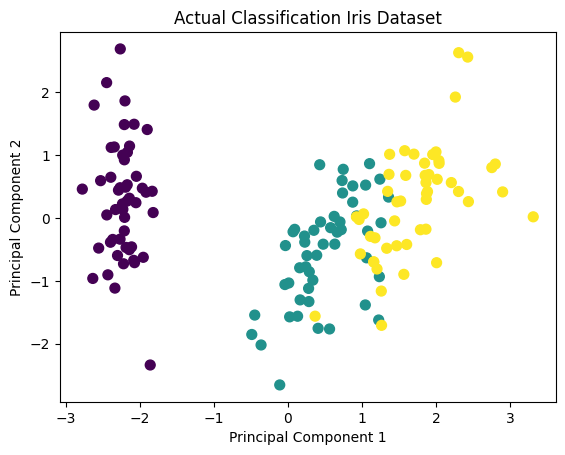


Corresponding confusion matrices:

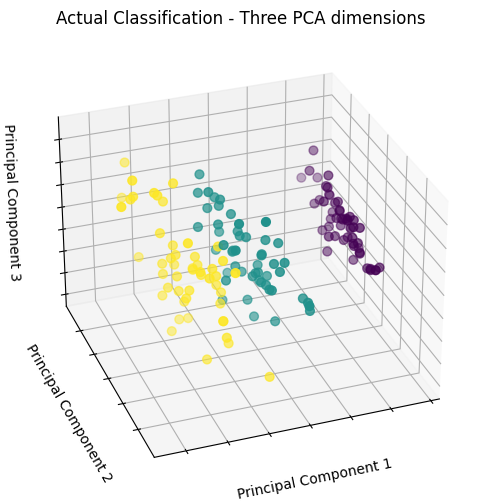
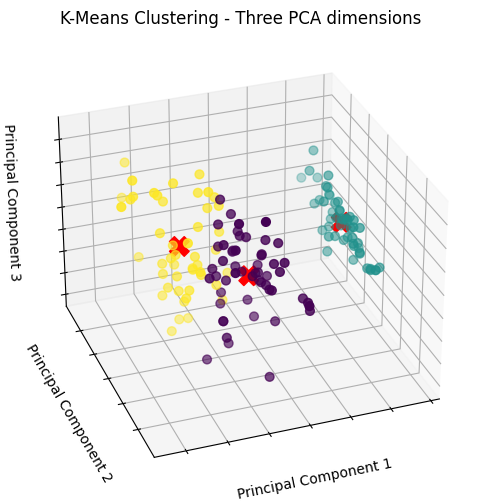




As we can see, both implementations found the same cluster centers and thus have the same prediction results. One of the clusters was predicted 100% accurately as shown in the confusion matrices, but the other two clusters are more ambiguous. Here is the actual classification of the dataset:



From the 2D representation, we can see some overlap, which explains why the algorithm would have trouble distinguishing between the two clusters. Here is also a 3D representation, which can give us a better idea of how the data is actually grouped:



This highlights one of the disadvantages of the K-means clustering algorithm. Since it assigns points to clusters based on their direct distance from a center point, the clusters will tend towards a “spherical” shape. With the Iris dataset, the actual groupings are more “oblong”, which K-means is not ideally equipped to handle.

**Mathematical Foundations:**

K-means clustering is an unsupervised machine learning algorithm that aims to find groups of similar data within a dataset. These groups are formed by setting a *k* number of clusters and assigning data points to the nearest cluster based on their euclidean distance to the center point/centroid of the cluster.

The goal of K-means clustering is to find the best centroids. This is achieved by repeating the two-step process of assigning data points to the closest centroid, then updating the centroid based on the resulting cluster. The new centroid is simply the sum of the cluster’s d-dimensional vector data points divided by the number of data points, i.e. the mean.

The algorithm iterates through these two steps until a stopping criterion is reached. This can be either an indicated limit on the number of iterations or when the algorithm has converged. Convergence is determined according to the difference in cost from the previous iteration. If the cost difference passes below a set threshold, then it is unlikely to meaningfully further improve and can be said to have converged.

**PART 3: LOGISTIC REGRESSION-BASED CLASSIFICATION**

**Google Colab Notebook**:

https://colab.research.google.com/drive/1rW2d47cVPjXmq-9MCCFkk7gmHUarH\_sf?usp=sharing

**Code and logistic regression explanation:**

The code in this notebook starts by importing and normalizing the Iris dataset (as described in part one) in the “import necessary packages and data” section.

In the “Logistic regression from scratch” and “Logistic regression from scratch function caller” sections, a series of functions are defined that will be used during logistic regression. These functions include a prediction, cost, gradient descent, weight update, and weight initialization functions. Finally, a training function calls the previously mentioned functions in the correct order to train the model. The training data can then be input into the training function to determine the weights needed for accurate prediction using logistic regression.

The “Logistic regression with Scikit-Learn” section uses the Scikit-Learn package in python to perform logistic regression. The Scikit-Learn package has a self-contained and user-friendly logistic regression model which simply takes in the training data along with some basic model parameters and preferences. This results in a model that can be used to predict classes of further validation data sets.

The “Logistic regression evaluation” section evaluates the precision, recall, f1 score, support, and accuracy for both the logistic regression from scratch and logistic regression using Scikit-Learn results. It also produces the confusion matrix of both, which helps illustrate when data points are being misclassified.

The final section “Cross-validation to validate results” uses k-fold cross validation to verify the model performs acceptably across all the data in the data set and isn’t overtrained to the training data.

Some code snippets and outputs will be displayed below, but the reader is encouraged to view the Colab notebook and read the comments there for a more thorough understanding of the given code.

**The error function:**

The error function is a function used to calculate the error resulting from the given model weights applied across all training data points. The error function is the negative sum of the natural log of the probability associated with the correct data point. Simply, if, for a given data point, the probability associated with the correct class is low (for example around 0.3), then the natural log will be smaller (or larger after being multiplied by -1) than if the probability associated with the correct class is perfectly correct at 1.0 (which results in a natural log of zero). The equation and its code implementation can be seen below.

From Lecture 11:

Where

Where

In python code:

def error\_func(prob\_array, pred\_class, correct\_class):

num\_data\_points = prob\_array.shape[0] #used to loop through all data points

total\_error = 0 #initialize total error as 0

for n in range(num\_data\_points): #loop through data points, adding errors

total\_error += -1 \* np.log(prob\_array[n, correct\_class[n]]) #Error function

return total\_error

**The Gradient Descent Function:**

The gradient descent function is a function used to determine the gradient of the error function. This information is then used to update the weights in a way that tends to move the error function lower and lower in steps proportional to the learning rate. This reduction in error is the means by which the machine learning algorithm is trained. The gradient descent function is the sum of the difference between the predicted probability of a given class and the true class multiplied with the data vector across all data points. The equation and code are given below.

From Lecture 11:

where

and

η = learning rate

python code:

def grad\_desc(data, prob\_array, correct\_class):

T\_arr = np.zeros(np.shape(prob\_array)) #initialize T matrix

data = np.append(data, np.ones((len(data), 1)), 1) #add column of ones to the data matrix for bias term

for i in range(len(data)): #loop through data points

T\_arr[i, correct\_class[i]] = 1 #populate T matrix at correct class

del\_W = np.dot(np.transpose(data), (prob\_array - T\_arr)) #gradient calculation

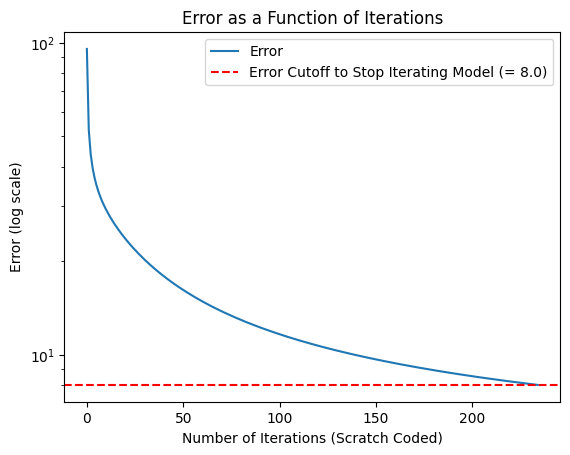
return del\_W

def update\_weights(weights, del\_W, learning\_rate):

new\_weights = weights - (learning\_rate \* del\_W) #New weights are the old weights minus the result of gradient descent times the learning rate

return new\_weights

**Discussion:**

Completing logistic regression from scratch as well as using the Scikit-Learn package allowed for effective learning of the key concepts of logistic regression as well as familiarization with the user-friendly and highly capable Scikit-Learn package. The error as a function of model iteration that was collected during the model execution of “logistic regression from scratch” showed a smooth decay in error with each iteration. This indicates a properly sized learning rate and correct execution of the gradient descent function as the model converges on the chosen error cutoff which stops model iteration. The error cutoff was chosen at 8 as this resulted in a quick convergence time and lower error targets did not seem to meaningfully decrease the real output of the model. A plot of this behavior is shown below.  
  
**Results:**  
Both methods (scratch and Scikit-Learn) yielded the same results for the random split of training data used, as shown in their classification reports and confusion matrices below. The models predicted the Setosa species perfectly, which could be expected given that it showed a wide separation and well defined cluster when compared to the other 2 species in the feature charts from part 1. The Veriscolor and Viriginica species were predicted fairly well, but were not 100% correct. They both showed a precision, recall f1, and accuracy in the 90% range. Precision indicates that of the data points predicted at these species, most were true positives with few false positives. The recall score indicating that there were similarly few false negatives of both species. As an interesting note, all the false positives of one species are false negatives of the other species for versicolor and virginica, resulting in the precision of one species being equal to the recall of another. Due to this, the f1 scores (a function of both precision and recall) are the same for both species. The support column simply indicates the number of species per class in the given dataset. Finally, the accuracy indicates that across all data points, 96% of the data points were predicted correctly.

Classification Report for Logistic Regression Coded from Scratch:

precision recall f1-score support

setosa 1.00 1.00 1.00 26

versicolor 0.92 0.96 0.94 24

virginica 0.96 0.92 0.94 25

accuracy 0.96 75

macro avg 0.96 0.96 0.96 75

weighted avg 0.96 0.96 0.96 75

Classification Report for Logistic Regression Using SKLearn:

precision recall f1-score support

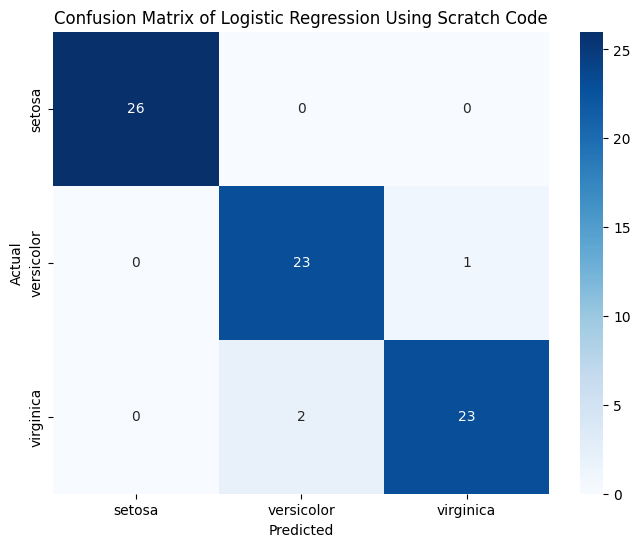
setosa 1.00 1.00 1.00 26

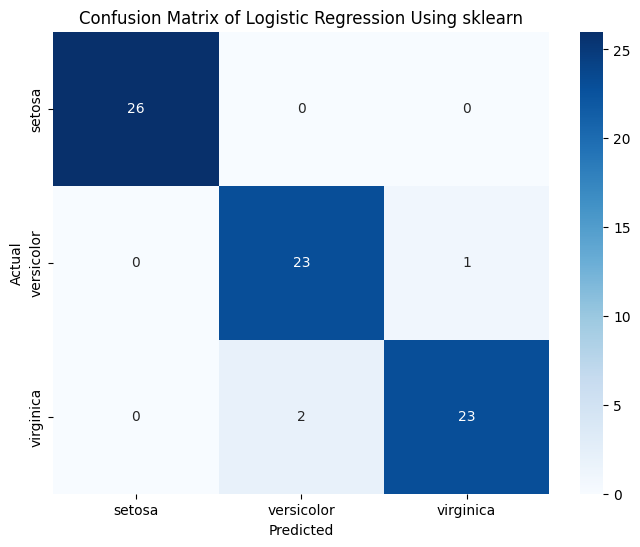
versicolor 0.92 0.96 0.94 24

virginica 0.96 0.92 0.94 25

accuracy 0.96 75

macro avg 0.96 0.96 0.96 75

weighted avg 0.96 0.96 0.96 75



Performing K-Fold cross-validation on the model showed that with 10 folds, the average accuracy was 97% with a standard deviation of 3%. These results would suggest that the model is well trained on the data, but is not overly trained to a specific subset.

**PART 4: NEURAL NETWORK-BASED CLASSIFICATION**

Neural Network for Effectively Classifying the Iris Species

Link to Code for Coded Neural Network. Created to Take any Number of Layers and Any Number of Neurons Per Layer

<https://drive.google.com/file/d/1QiJcQegdK5YNYkWYSDWN0rNtoQRrIBuF/view?usp=sharing>

Link to Code for Sklearn Coded Neural Network. Created to take any number of layers and any number of neurons per layer.

<https://drive.google.com/file/d/1-fzoVEgTM2BiJ-QjGbExPgYLIQC09IcM/view?usp=sharing>

Looking at the cross-validation accuracies for the Neural Network created from scratch, the Sigmoid activation function saw the lowest fold validation accuracies, regardless of the number of layers or neurons per layer. Using the ReLU activation function averaged the highest validation accuracies, with the TanH activation function sitting in the middle of the two. The ReLU activation function may have had a more effective means of combating the vanishing gradient problem, improving its overall performance. That being said, for most of the activation functions, increasing the learning rate had more of an effect on the learning curve than increases in either the number of hidden layers or the number of neurons per layer. While this is not always the case, at times missing the mark in convergence to the optimal weights, on average got to the optimal rate much more quickly than a lower learning rate.

Each of the activation function’s learning curves saw strange differences in their learning curves when more layers were introduced. Some a nice slope that suddenly jutted out, while others looked completely different and unrecognizable. This was witnessed in both the from scratch neural network as well as the Sklearn neural network. While the from scratch neural network’s learning curve was more recognizable across similar activation functions, the Sklearn neural network’s learning curve shape was relatively consistent regardless of activation function, number of hidden layers, number of neurons per layer, or learning rate. Although the learning curve became a jittery mess when two layers, each with ten neurons, and a learning rate of 0.3, all of the other curves increased or decreased in relative amounts. This may have been due to the fact that Sklearn had a library and better implementation in place versus what we attempted from scratch.

On average, two hidden layers, with five to ten neurons per layer remained a great deal more stable for both the from scratch neural network, as well as the Sklearn neural network, even without considering the learning rate or activation function. This is due to the fact that the first hidden layer is able to partition the input space and allows the second layer to improve approximations. These do so without affecting any other areas. The networks with three hidden layers suffered from overfitting, while one hidden layer had the opposite issue.

For a closer look at each graph, please visit the following link-

<https://docs.google.com/document/d/1x94emf3WUlet2dXiRmjcNysMmbEjW8Qv/edit?usp=sharing&ouid=110994036319626462080&rtpof=true&sd=true>

**Conclusion:**

Throughout this project, 3 different types of machine learning were explored: k-means clustering, logistic regression and neural networks. All 3 showed different performances across the data set given their strengths and weaknesses. For k-means clustering, the “spherical” nature of the clusters resulted in several false-positives and false-negatives between the versicolor and virginica species, resulting in around 70% accuracy for both species. These two species features are closely grouped, so this behavior is to be expected. K-means clustering did predict the setosa species very well, with 100% accuracy. Further, k-means clustering may be the simplest to implement and may take fewer iterations to converge on this small dataset. Logistic regression improves on the results of k-means clustering, still predicting the setosa species perfectly, while also doing a better job at predicting the viriginica and versicolor species, at around 95% for both. However, this comes at the cost of more complex implementation and a much larger number of iterations needed to converge on an answer. Finally, neural networks require the most complex mathematical implementation and most computationally taxing training sequence. However, the result of this should be a further improvement on the logistic regression methods. Overall, the best method for creating a machine learning model for this data would depend on the accuracy needed and the computational resources at hand. For this data set, k-means clustering may suffer with accuracy of 2 species. Logistic regression offers a significant improvement upon this, but may take more computational power. Neural networks may be the most accurate, but it is also considerably more computationally taxing than the other 2 methods.