Surf **Tensor Flow KMeans Demo**

Iris Demo Project

Name: Nishanth Bhaskara **June 28, 2016**

1) Introduction We are given 150 data points, each of which are 4 - dimensional vectors, and a corresponding class which corresponds to the type of flower that it is. The goal is to use the Tensor Flow library and the KMeans algorithm to be able to cluster this data set into 3 groupings, and try to achieve accurate classification.

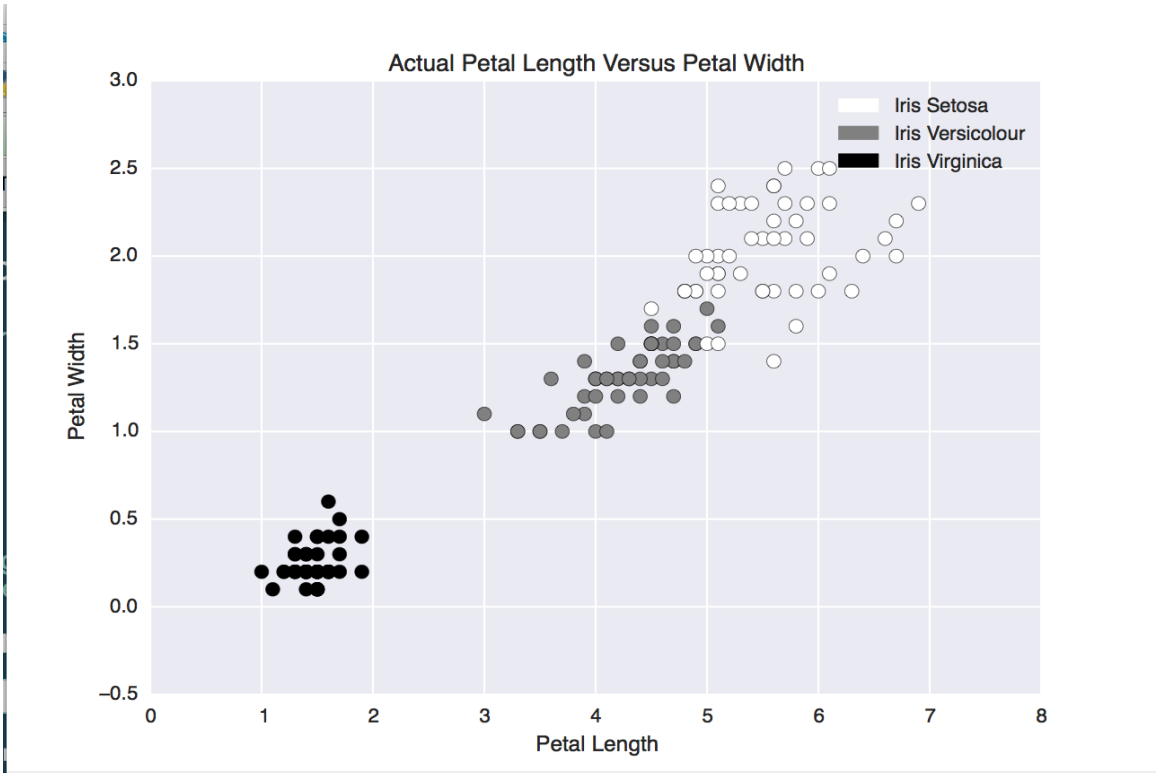
2) Code Aside from all of the formatting, and graph displaying, the most essential code written for this was the actual KMeans algorithm, which is shown below.

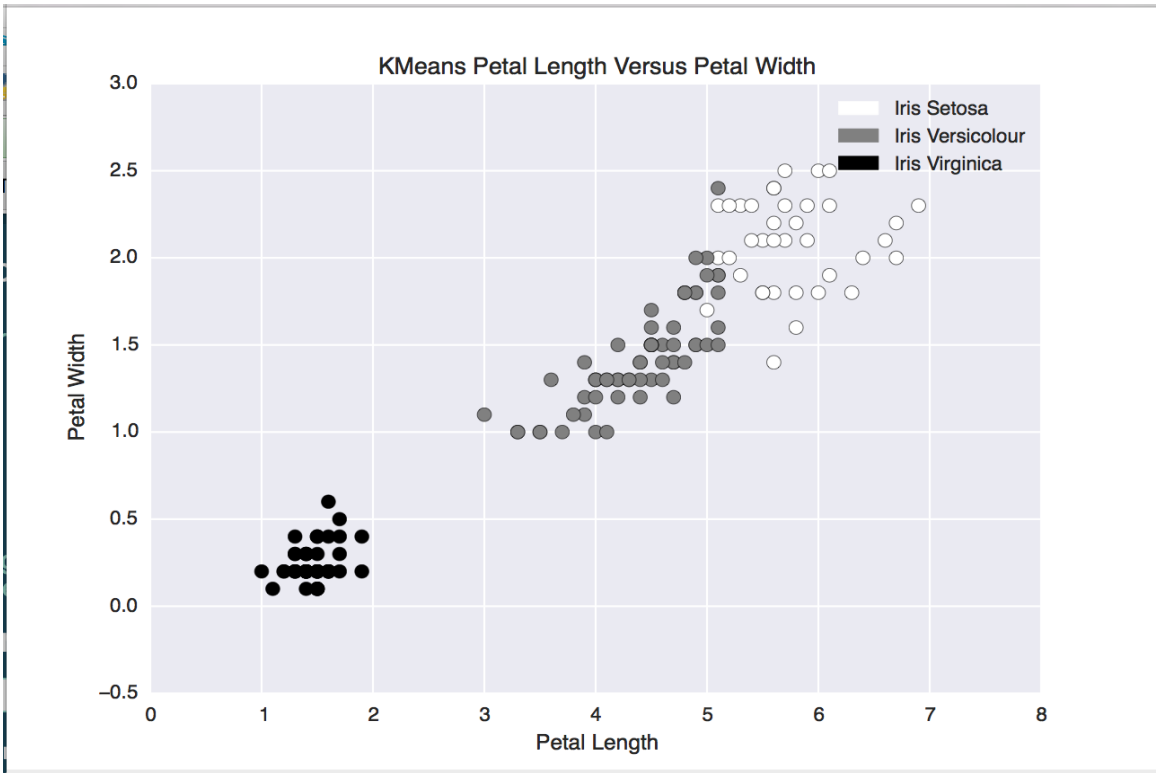
```
def TensorFlowKMeans(vectors, kClusters):

    kClusters = int(kClusters)
    assert kClusters < len(vectors)

    # First we find the dimensionality of our data set.
    dim = len(vectors[0])

    # Let's just shuffle the indices to select random centroids.

    vector_indices = list(range(len(vectors)))
    shuffle(vector_indices)

    # We initialize a new graph and set it as the default during each run
    # of this algorithm. This will allow us not to crowd our graph with
    # unused/unnecessary operations or values.

    graph = tf.Graph()

    with graph.as_default():

        sess = tf.Session()

        # First lets ensure we have a Variable vector for each centroid,
        # initialized to one of the vectors from the available data points

        centroids = [tf.Variable((vectors[vector_indices[i]]))
                     for i in range(kClusters)]

        # These nodes will assign the centroid Variables the appropriate
        # values

        centroid_value = tf.placeholder("float64", [dim])
        cent_assigns = []
        for centroid in centroids:
            cent_assigns.append(tf.assign(centroid, centroid_value))
```

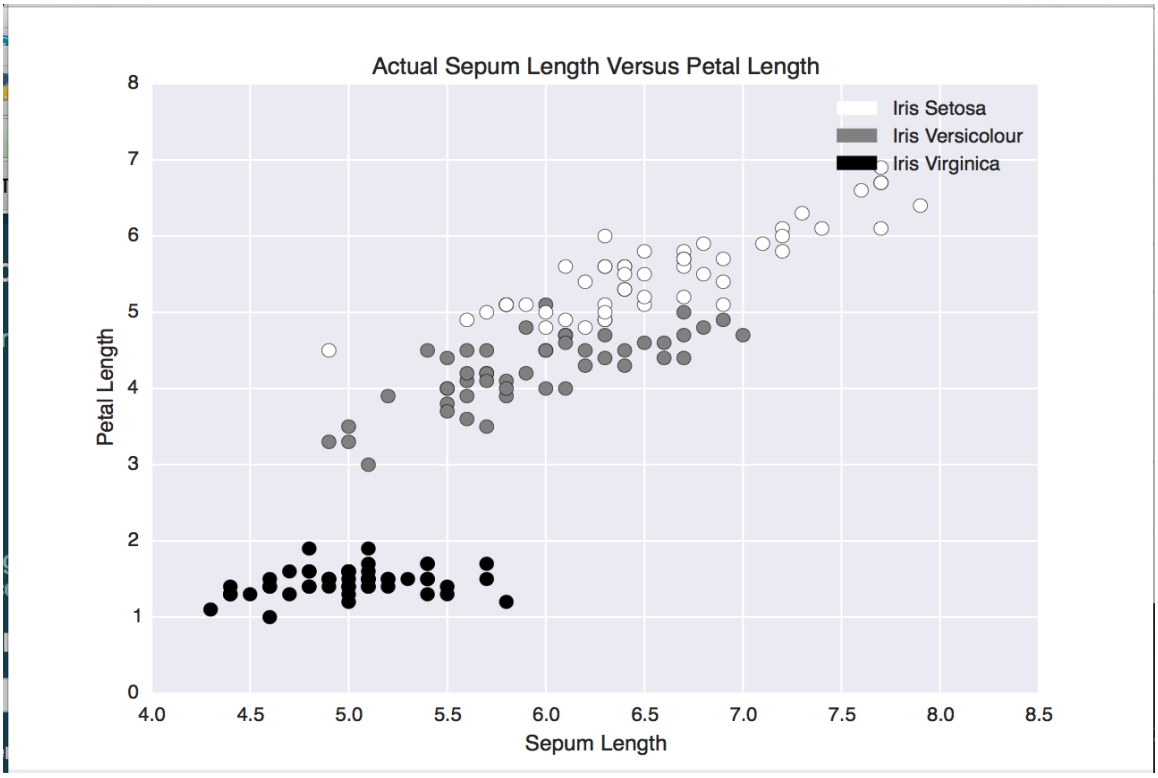
```
# Variables for cluster assignments of individual vectors(initialized
# to 0 at first)

assignments = [tf.Variable(0) for i in range(len(vectors))]

# These nodes will assign an assignment Variable the appropriate
# value.

assignment_value = tf.placeholder("int32")
cluster_assigns = []

for assignment in assignments:
    cluster_assigns.append(tf.assign(assignment,
                                     assignment_value))

# Now lets construct the node that will compute the mean
# The placeholder for the input

mean_input = tf.placeholder("float", [None, dim])

#The Node/op takes the input and computes a mean along the 0th
#dimension, i.e. the list of input vectors

mean_op = tf.reduce_mean(mean_input, 0)

# Node for computing Euclidean distances
# Placeholders for input
v1 = tf.placeholder("float", [dim])
v2 = tf.placeholder("float", [dim])
euclid_dist = tf.sqrt(tf.reduce_sum(tf.pow(tf.sub(
    v1, v2), 2)))

# This node will figure out which cluster to assign a vector to,
# based on Euclidean distances of the vector from the centroids.
# Placeholder for input

centroid_distances = tf.placeholder("float", [kClusters])
cluster_assignment = tf.argmin(centroid_distances, 0)


# This will help initialization of all Variables defined with respect
# to the graph. The Variable-initializer should be defined after
# all the Variables have been constructed, so that each of them
# will be included in the initialization.

init_op = tf.initialize_all_variables()

# Initialize all variables
sess.run(init_op)
```

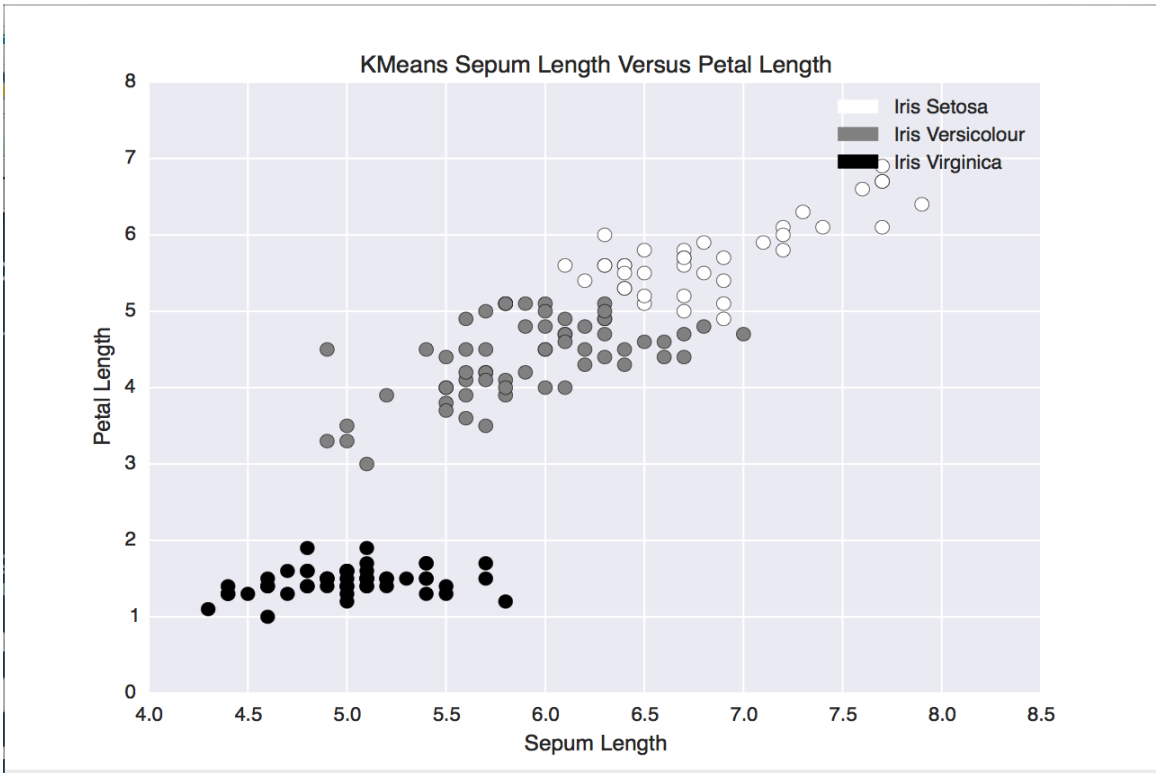
```
# Now perform the Expectation-Maximization steps of K-Means clustering
# iterations. To keep things simple, we will only do a set number of
# iterations, instead of using a Stopping Criterion.
kIterations = 100
for iteration_n in range(kIterations):

    # Based on the centroid locations till last iteration, compute
    # the _expected_ centroid assignments.
    # Iterate over each vector
    for vector_n in range(len(vectors)):
        vect = vectors[vector_n]
        # Compute Euclidean distance between this vector and each
        # centroid. Remember that this list cannot be named
        # 'centroid_distances', since that is the input to the
        # cluster assignment node.
        distances = [sess.run(euclid_dist, feed_dict={
            v1: vect, v2: sess.run(centroid)})
                     for centroid in centroids]
        # Now use the cluster assignment node, with the distances
        # as the input
        assignment = sess.run(cluster_assignment, feed_dict = {
            centroid_distances: distances})
        # Now assign the value to the appropriate state variable
        sess.run(cluster_assigns[vector_n], feed_dict={
            assignment_value: assignment})

    # Based on the expected state computed from the Expectation Step,
    # compute the locations of the centroids so as to maximize the
    # overall objective of minimizing within-cluster Sum-of-Squares
    for cluster_n in range(kClusters):
        # Collect all the vectors assigned to this cluster
        assigned_vects = [vectors[i] for i in range(len(vectors))
                          if sess.run(assignments[i]) == cluster_n]
        # Compute new centroid location
        new_location = sess.run(mean_op, feed_dict={
            mean_input: array(assigned_vects)})
        # Assign value to appropriate variable
        sess.run(cent_assigns[cluster_n], feed_dict={
            centroid_value: new_location})

#Return centroids and assignments
centroids = sess.run(centroids)
assignments = sess.run(assignments)
return centroids, assignments
```

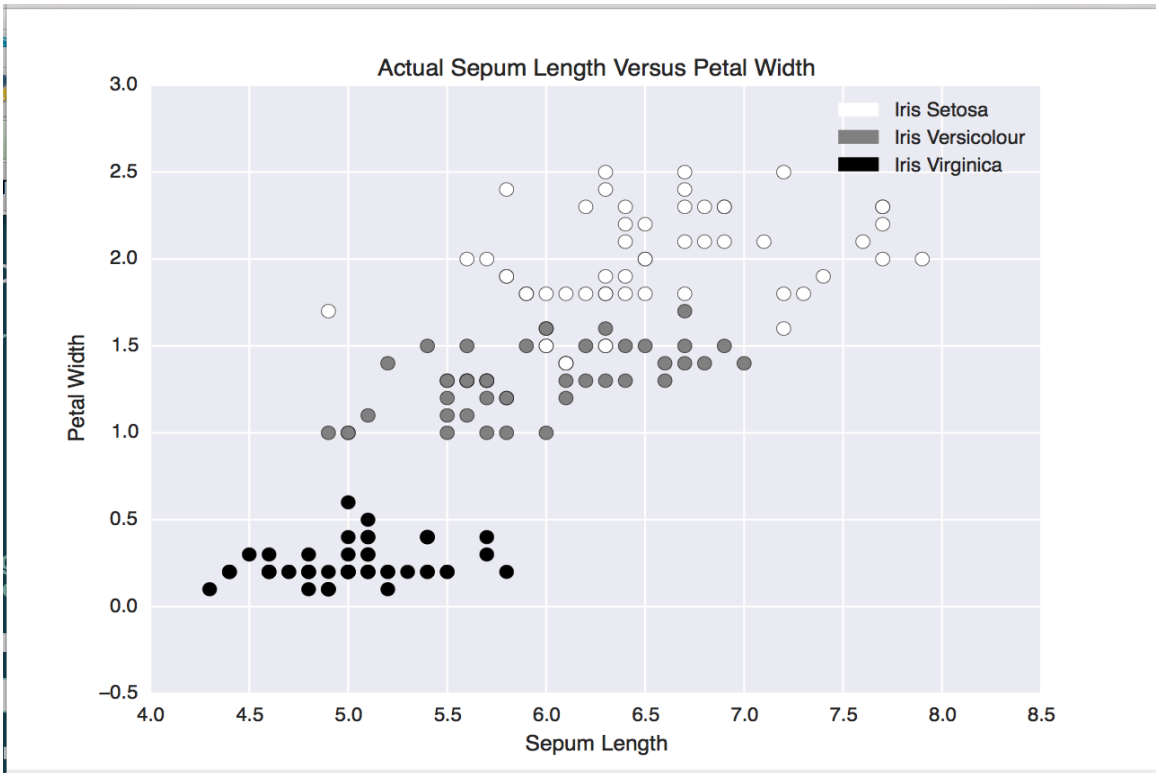
3) Results The result is best depicted by graphs. As we are only able to display 2 dimensions at a time and we have 4 dimensional data points, we need $\binom{4}{2} = 6$ graphs to depict our results. For each of these 6 graphs, there is also a corresponding graph generated by kMeans.

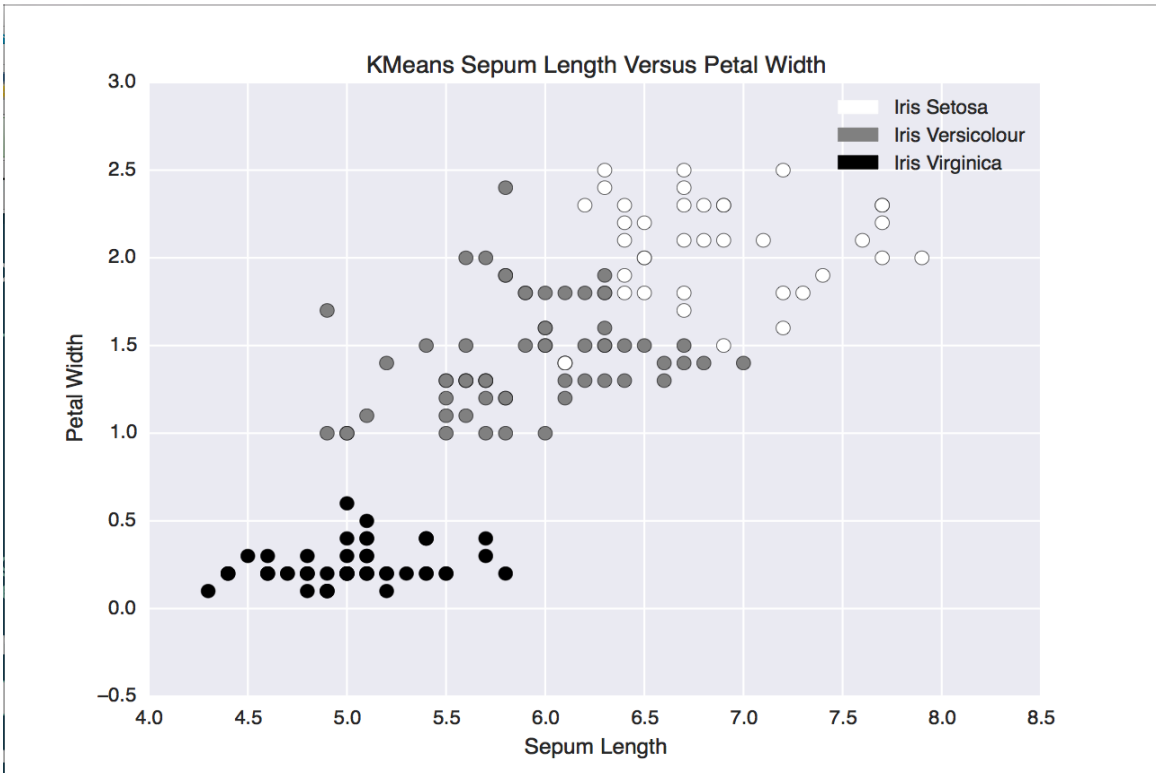

Actual Petal Length Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Petal Length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7
8

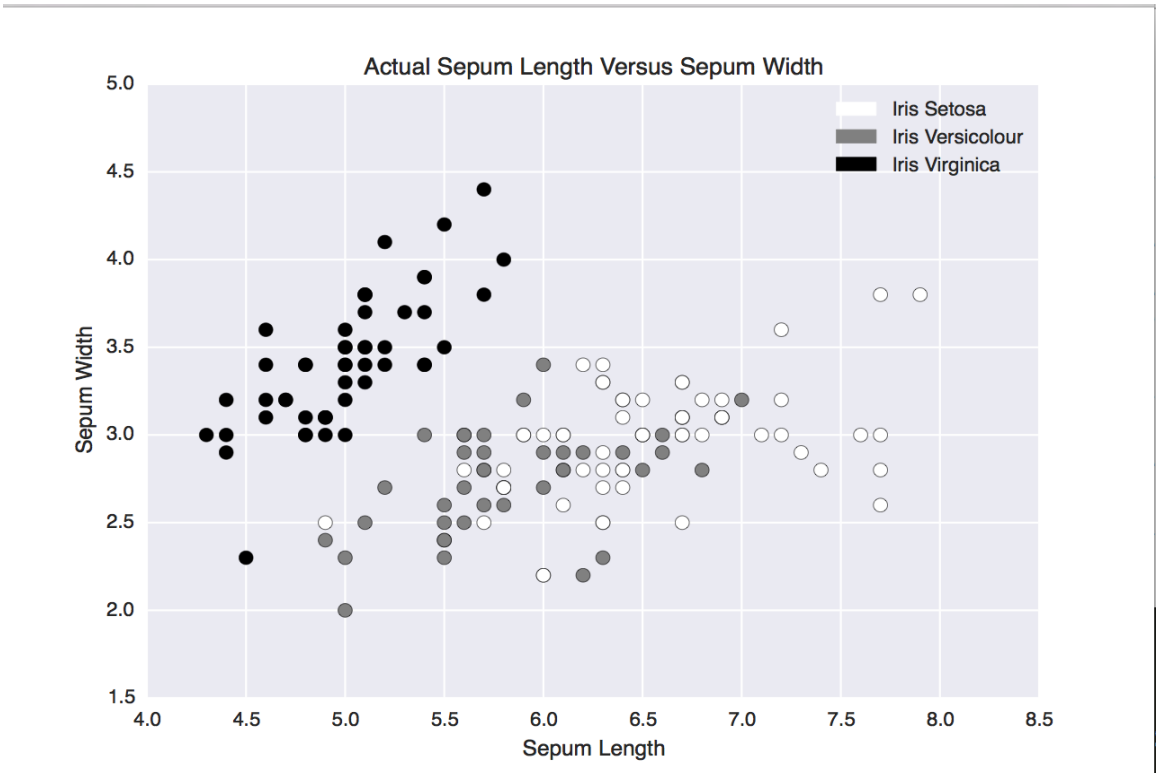

KMeans Petal Length Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Petal Length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
2
3
4
5
6
7
8

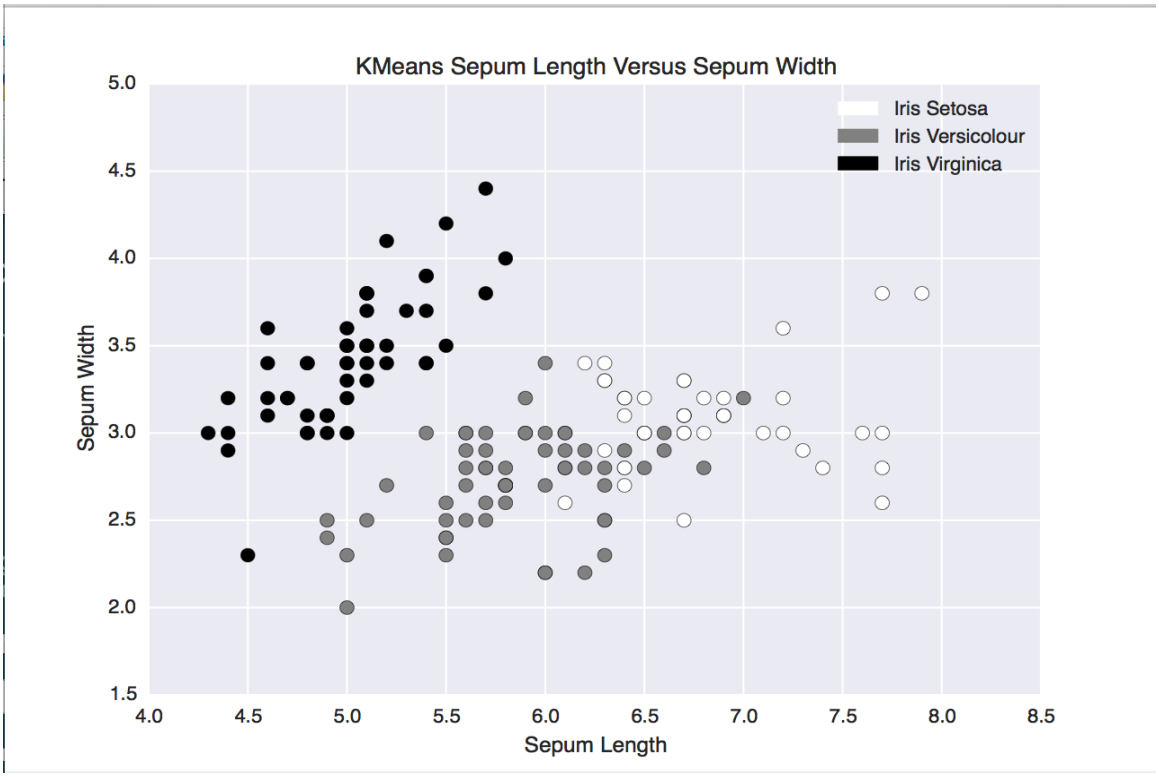

Actual Sepum Length Versus Petal Length
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Length
Sepum Length
0
1
2
3
4
5
6
7
8
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5

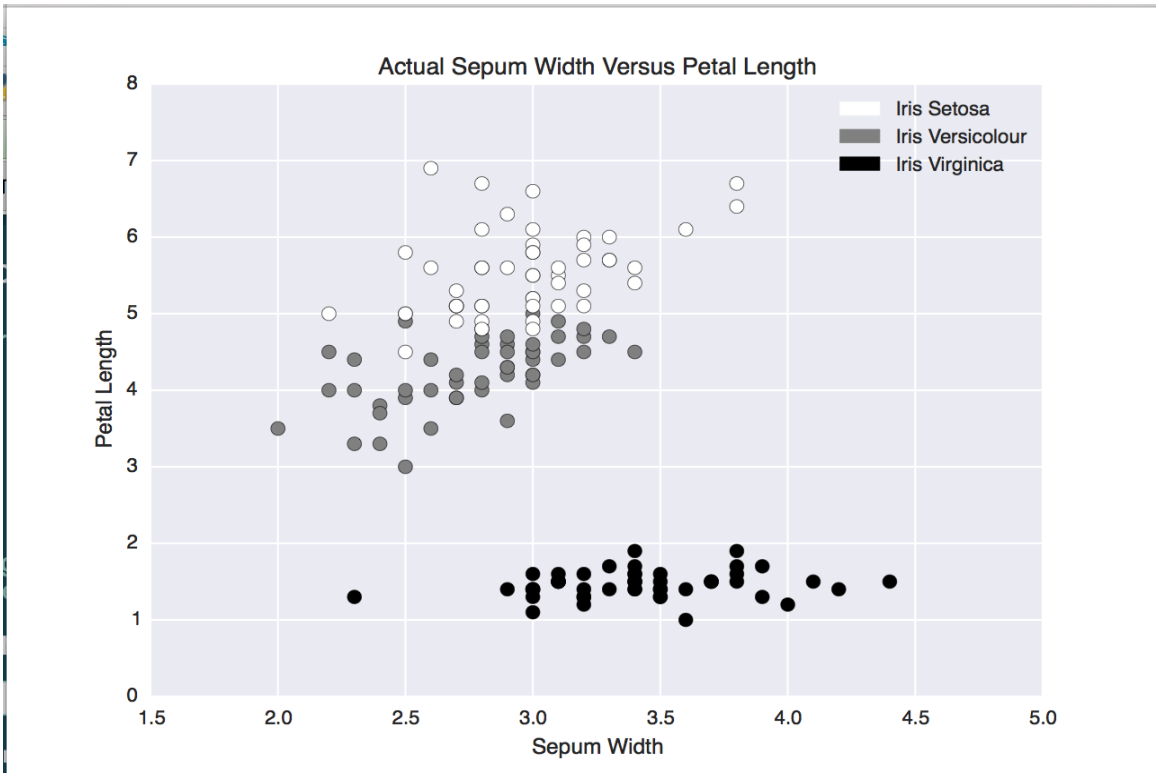

KMeans Sepum Length Versus Petal Length
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Length
Sepum Length
0
1
2
3
4
5
6
7
8
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5

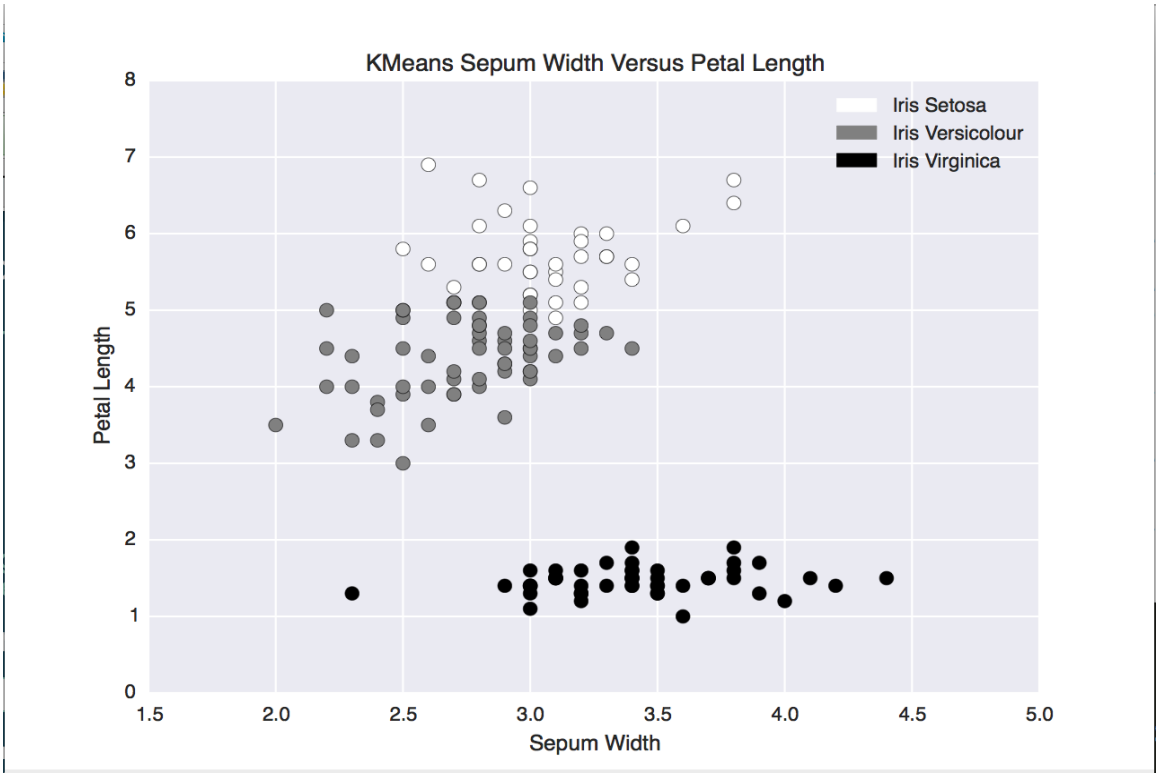

Actual Sepum Length Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Sepum Length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5

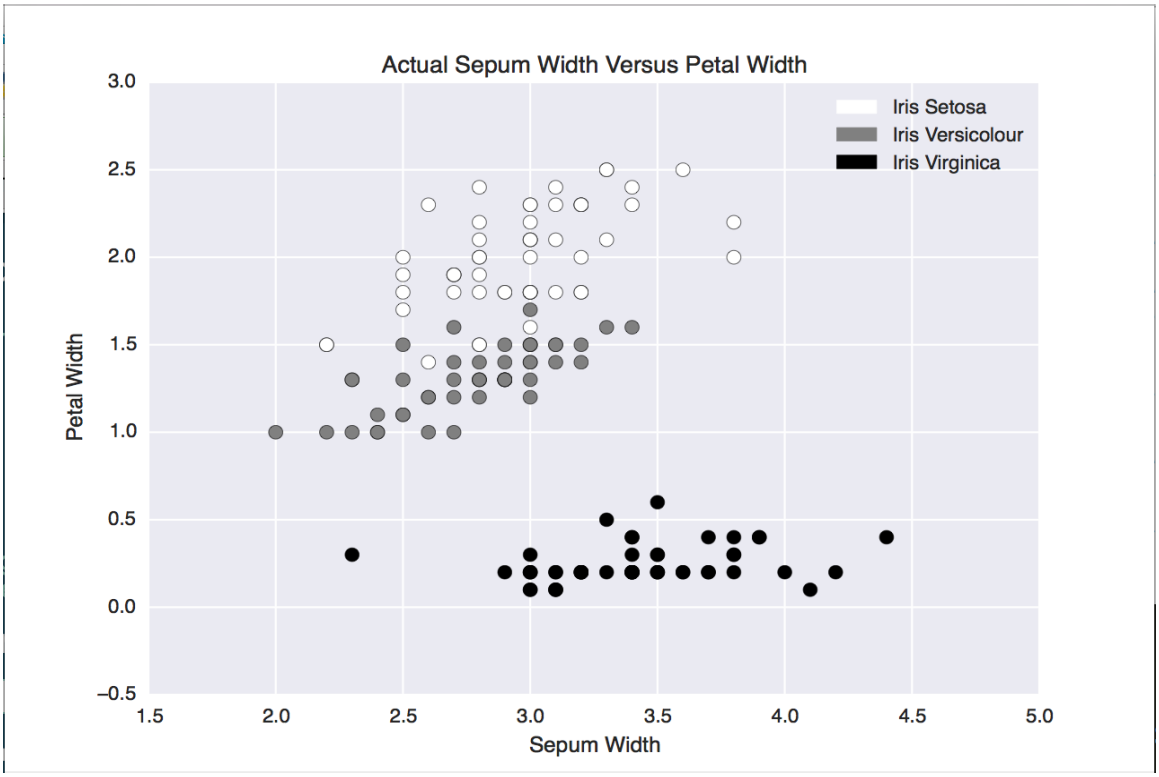

KMeans Sepum Length Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Sepum Length
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5

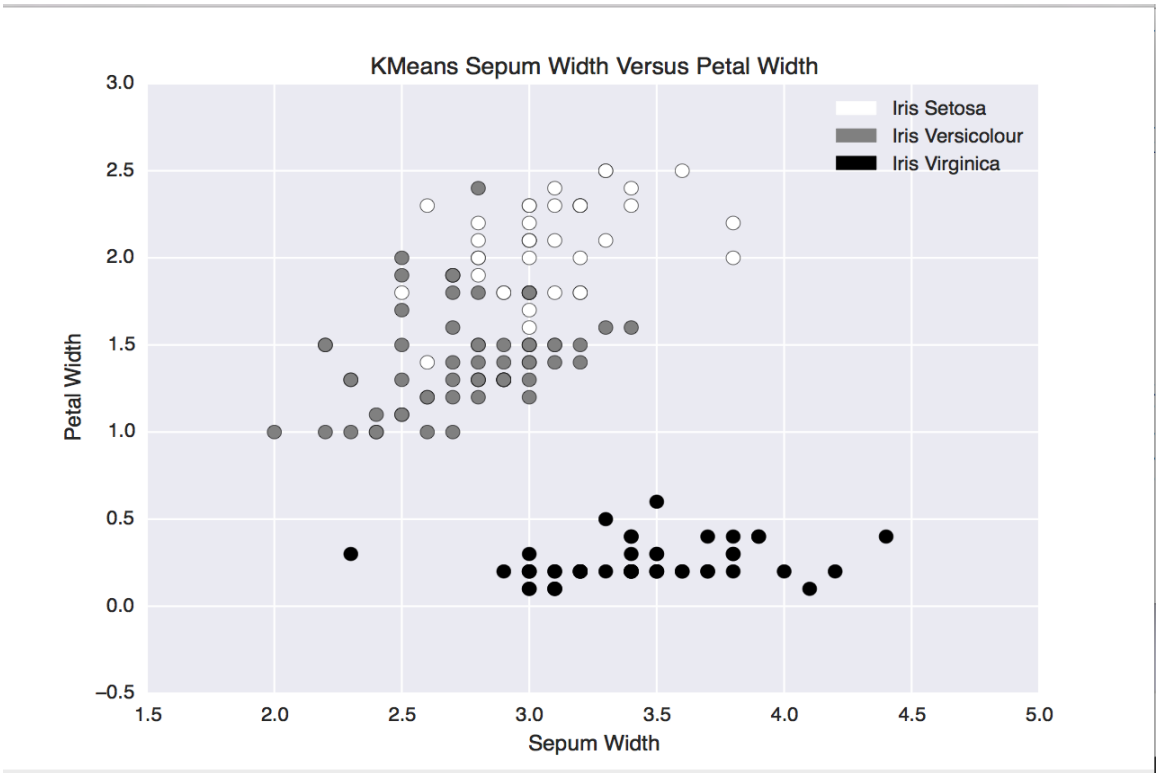

Actual Sepum Length Versus Sepum Width
Iris Setosa
Iris Versicolour
Iris Virginica
Sepum Width
Sepum Length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5


KMeans Sepum Length Versus Sepum Width
Iris Setosa
Iris Versicolour
Iris Virginica
Sepum Width
Sepum Length
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5


Actual Sepum Width Versus Petal Length
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Length
Sepum Width
0
1
2
3
4
5
6
7
8
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


KMeans Sepum Width Versus Petal Length
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Length
Sepum Width
0
1
2
3
4
5
6
7
8
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Actual Sepum Width Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Sepum Width
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


KMeans Sepum Width Versus Petal Width
Iris Setosa
Iris Versicolour
Iris Virginica
Petal Width
Sepum Width
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

4) Conclusion

```
The accuracy for kMeans for the first class is 1.0
The accuracy for kMeans for the second class is 0.96
The accuracy for kMeans for the third class is 0.96
```

The results were surprisingly good. As expected, the Iris Virginica had a success rate of 100% as it can be linearly separated by a hyperplane from the other two classes (as we can see in the graphs). Despite the fact that Iris Versicolour and and Iris Setosa were not as clearly separable, our algorithm was still able to achieve an accuracy of 96%.