Exercise 1. Clustering

a) Write a program that implements the k-means clustering algorithm on the iris data set. You should use the objective function and learning rule you derived in W4 Q1 (and your implementation will help you uncover any errors, so it is helpful to do this before you turn in W4.)

```
def kmeans(data_set, k):
    centroids = initialize_centroids(data_set, k)
    stopping_criteria = False

    list_of_centroids = [centroids]
    num_steps = 0
    while not stopping_criteria:
        previous_centroids = centroids
        labels = get_labels(data_set, centroids)
        centroids = objective_function(data_set, labels, k)
        list_of_centroids.append(centroids)
        num_steps += 1
        if stop(previous_centroids, centroids):
            stopping_criteria = True

    fig = plt.figure(1, (10, 9))
    fig.subplots_adjust(0.1)

    print_progress(fig.add_subplot(221), data_set, list_of_centroids[0], "Initial Cluster", 0)
    print_progress(fig.add_subplot(222), data_set, list_of_centroids[int(num_steps/2)], "Intermediate Cluster", int(num_steps/2))
    print_progress(fig.add_subplot(223), data_set, list_of_centroids[int(num_steps)], "Converged Cluster", int(num_steps))
    plt.show()
```

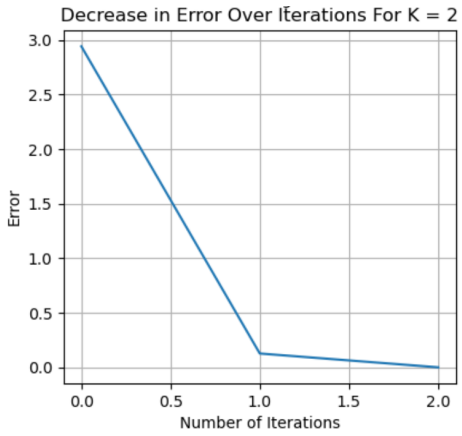
- get_labels function labels each data_set and returns as array of labels
- Initialize_centroids function takes data_set and produce k number of random centroids to begin with
- update_centroids function takes data set, labels, and number of clusters k to update the coordinates of centroids accordingly

b) Plot the value of the objective function as a function of the iteration.

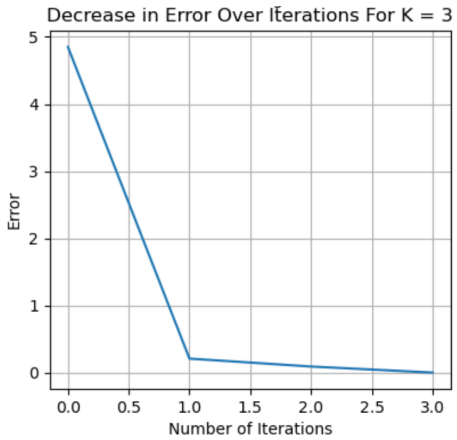
$$D = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{n,k} || x_n - \mu_k ||^2$$

to show that your learning rule and implementation minimizes this expression.

- K = 2:



- K=3:

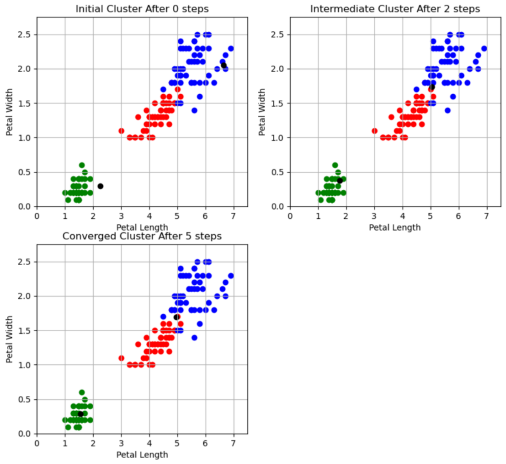


- Source Code to compute the error:

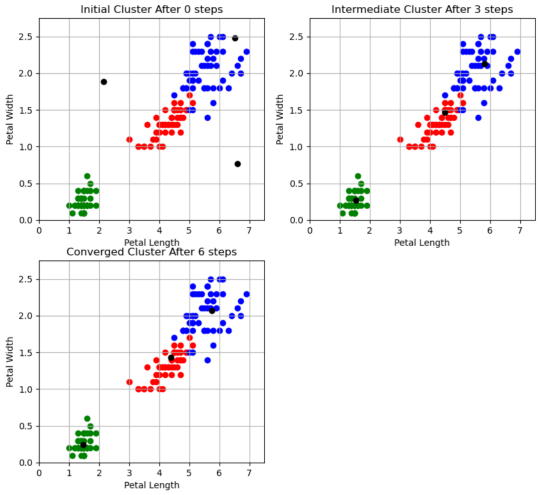
```
def errors(previous, next):
    movement = 0
    for prev, nxt in zip(previous, next):
        movement += ((prev[0] - nxt[0]) ** 2 + (prev[1] - nxt[1]) ** 2 +
                     (prev[2] - nxt[2]) ** 2 + (prev[3] - nxt[3]) ** 2) ** 0.5
    return movement < 0.0000001, movement
```

c) Plot the results of the learning process by showing the initial, intermediate, and converged cluster centers overlaid on the data for k = 2 and k = 3.

- k=2

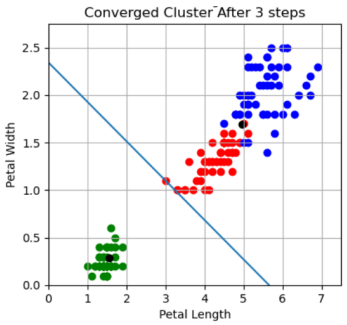


- k=3

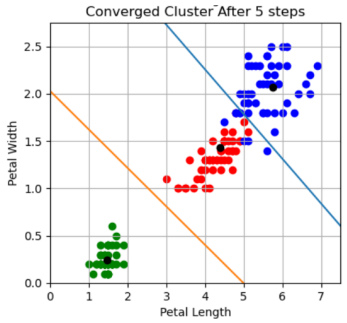


d) Devise a method to plot the decision boundaries for this dataset using the optimized parameters. Explain your approach and plot your results.

- k=2



- k=3

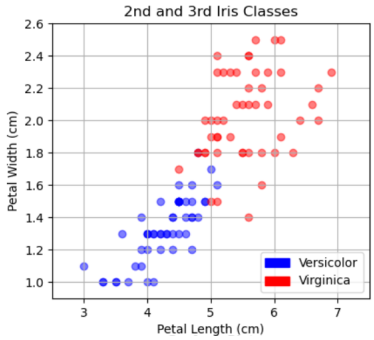


- I compute the equation of the linear function by calculating the points that are equidistant from a set of pairs of the closest centroids which represents the decision boundaries for the given iris data set.
- The equation for computing the decision boundaries is given by the code below:

```
def equidistance(pairs):
    coordinates = []
    t = np.linspace(-2, 2, 100)
    for pair in pairs:
        x = (pair[1][2] + pair[0][2]) / 2 + (pair[0][2] - pair[1][2]) * t
        y = (pair[1][3] + pair[0][3]) / 2 - (pair[0][3] - pair[1][3]) * t
        coordinates.append((x, y))
    return coordinates
```

Exercise 2. Linear decision boundaries

a) Inspect irisdata.csv which is provided with the assignment file on Canvas. Write a program (or use a library) that loads the iris data set and plots the 2nd and 3rd iris classes, similar to the plots shown in lecture on neural networks (i.e. using the petal width and length dimensions).



- Source Code:

```
def scatter_plot(plot, iris_objects, names, title):
    red_patch = patches.Patch(color='red', label='Virginica')
    blue_patch = patches.Patch(color='blue', label='Versicolor')

    for name in names_:
        plot.scatter(iris_objects[name].petal_lengths, iris_objects[name].petal_widths, c=name, alpha=0.5)

    plot.legend(handles=[blue_patch, red_patch], loc='lower right')
    plot.set_ylim([0.9, 2.6])
    plot.set_xlim([2.5, 7.5])
    plot.set_title(title)
    plot.set_ylabel('Petal Width (cm)')
    plot.set_xlabel('Petal Length (cm)')
    plot.grid(True)
```

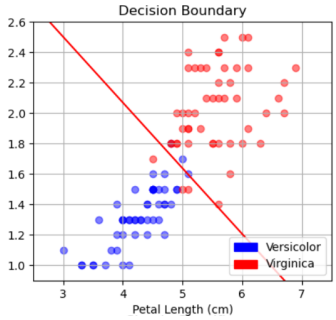
b) Define a function that computes the output of a simple one-layer neural network using a sigmoid nonlinearity (linear classification with sigmoid nonlinearity).

- Use an output of 0 to represent the iris color green and an output 1 to represent the blue iris classification.
- In the below equation, the y represents the output of a one-layer neural network utilizing sigmoid nonlinearity.

$$y = \sigma\left(\sum_{i=0}^{M} w_i x_{i,n}\right) = \sigma(w^T x_n) = \frac{1}{1 + e^{w_i x_n}}$$

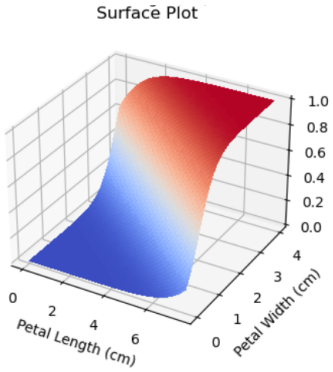
c) Write a function that plots the decision boundary for the non-linearity above overlaid on the iris data. Choose a boundary (by setting weight parameters by hand) that roughly separates the two classes. Use an output of 0 to indicate the 2nd iris class, and 1 to indicate the 3rd.

- I initialized the weight set = [-11, 1.25, 2.9] because it separated the two classes nicely.



d) Use a surface plot from a 3D plotting library (e.g. in matlab or using matplotlib in python) to plot the output of your neural network over the input space. This should be similar to the learning curve shown in fig 19.17 (18.17 in 3rd ed.) of the textbook.

- Surface Plot:

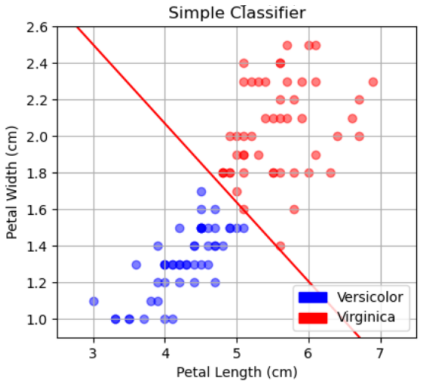


- Source Code:

```
def surface_plot(plot, weights, title):
    width = np.linspace(0, 4, 100)
    length = np.linspace(0, 7.5, 100)
    x_axis, y_axis = np.meshgrid(length, width)
    outputs = []

    for x in range(0, 100):
        outputs.append([])
        for y in range(0, 100):
            outputs[x].append(single_layer([1, x_axis[x][y], y_axis[x][y]], weights))

    graph_outputs = np.array(outputs)

    plot.set_zlim([0, 1])
    plot.plot_surface(x_axis, y_axis, graph_outputs, cmap=cm.coolwarm, linewidth=0, antialiased=False)
    plot.set_ylabel('Petal Width (cm)')
    plot.set_xlabel('Petal Length (cm)')
    plot.set_title(title)
```

e) Show the output of your simple classifier using examples from the 2nd and 3rd Iris classes. Choose examples that are unambigious as well as those that are near the decision boundary.

- As you can observe in the plots below, some of the unambigious data points are located in the very left bottom corner or at the very top right corner, such as at location (3, 1.15), (4, 1.0), and (4, 1.2). On the other hand, some of the points are right at the decision boundary. Some of these examples include points at (5, 1.7), (5.7, 1.4), and (5.01, 1.6)
- Simple Classifier Plot



- Source Code:

```
def simple_classifier(iris_objects, weights, names, desired_class)_:
    decided_class = {names[0]_: Iris(names[0], [], []), names[1]_: Iris(names[1], [], [])}
    for name in names_:
        for (x, y) in zip(iris_objects[name].petal_lengths, iris_objects[name].petal_widths)_:

            if single_layer([1, x, y], weights) > 0.5_:
                decided_class[desired_class['1']].petal_lengths.append(x)
                decided_class[desired_class['1']].petal_widths.append(y)
            else_:
                decided_class[desired_class['0']].petal_lengths.append(x)
                decided_class[desired_class['0']].petal_widths.append(y)

    return decided_class
```

Exercise 3. Neural networks

a) Write a program that calculates the mean-squared error of the iris data for the neural network defined above. The function should take three arguments: the data vectors, the parameters defining the neural network, and the pattern classes.

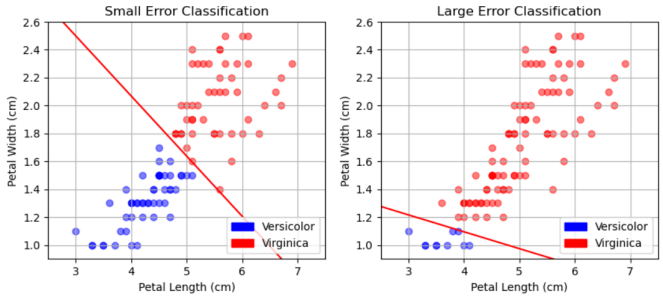
$$MSE = \frac{1}{N}\sum_{n=1}^{N}(\sigma(w^T x_n) - c_n)^2$$

- Above is the equation to compute the mean-squared error of the iris data for the neural network, where MSE stands for mean-squared error, and c_n represents the targeted vector. In addition, since this is a mean-squared error, it is divided by the number of data points.

b) Compute the mean squared error for two different settings of the weights (i.e. two different decision boundaries). Like above, select these by hand and choose settings that give large and small errors respectively. Plot both boundaries on the dataset as above.

- Small weight sets: [-11, 1.25, 2.9]
  - This small weight set is chosen by picking the random set, plotting the graph with the decision boundary line, and checking if the line clearly separates the two classes reasonably. As a result, this weight set separates the two classes clearly with the smallest error.
  - Small error calculated by error_computation function: 0.058029372297171686

- Large Weight Sets: [-3.0, 0.23, 1.9]
  - Large error set was chosen by manually computing the minimum squared error using error_computation function.
  - Large error calculated by error_computation function: 0.2046404135803611



- Source Code:

```
def error_computation(iris, weight, names, output):
    sum = 0
    num_points = 0
    for name in names:
        for petal_length, petal_width in zip(iris[name].petal_lengths, iris[name].petal_widths):
            num_points += 1
            diff = single_layer([1, petal_length, petal_width], weight) - output[name]
            sum += diff ** 2
    return sum / num_points
```

c) Give a mathematical derivation the gradient of the objective function above with respect to the neural network weights. You will have to use chain rule and the derivative of the sigmoid function as discussed in class. Use w_o to represent the bias term. You should show and explain each step.

- We have to take the derivative of the objective function with respect to the individual weights. As explained in part a, c_n represents the targeted vector.

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(\sigma(w^T x_n) - c_n)^2 = \frac{1}{N}\sum_{n=1}^{N}(\sigma(\sum_{i=0}^{M} w_i x_{i,n}) - c_n)^2$$

  - Taking the derivative of the above function with respect to the individual weights w_i:

$$\frac{\delta MSE}{\delta W} = [\frac{\delta MSE}{\delta W_0}, \frac{\delta MSE}{\delta W_1}, \ldots, \frac{\delta MSE}{\delta W_M}]$$

$$= [\frac{\delta}{\delta W_0}\frac{1}{N}\sum_{n=1}^{N}(\sigma(\sum_{i=0}^{M}(w_i x_{i,n}) - c_n)^2, \frac{\delta}{\delta W_1}\frac{1}{N}\sum_{n=1}^{N}(\sigma(\sum_{i=0}^{M} w_i x_{i,n}) - c_n)^2, \ldots, \frac{\delta}{\delta W_k}\frac{1}{N}\sum_{n=1}^{N}(\sigma(\sum_{i=0}^{M} w_i x_{i,n}) - c_n)^2$$

$$\frac{\delta MSE}{\delta W} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(w^T x_n) - c_n)\sigma(w^T x_n)(1 - \sigma(w^T x_n))[x_{0,n}, x_{1,n}, \ldots, x_{k,n}]$$

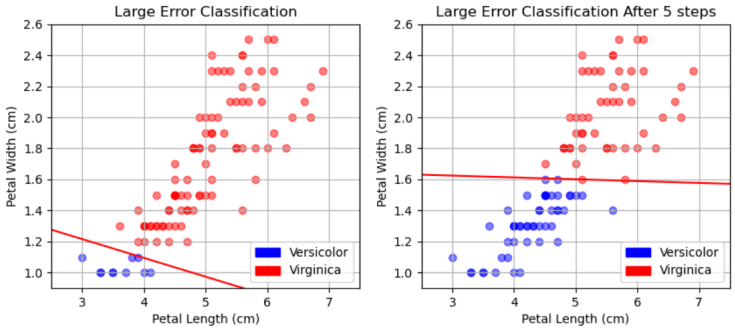
d) Show how the gradient can be written in both scalar and vector form.

$$\frac{\delta MSE}{\delta w_i} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(w^T x_n) - c_n)\sigma(w^T x_n)(1 - \sigma(w^T x_n))x_{i,n}$$

$$\frac{\delta MSE}{\delta w} = \frac{2}{N}\sum_{n=1}^{N}(\sigma(w^T x_n) - c_n)\sigma(w^T x_n)(1 - \sigma(w^T x_n))x_n$$

e) Write code that computes the summed gradient for an ensemble of patterns. Illustrate the gradient by showing (i.e. plotting) how the decision boundary changes for a small step.

- Large Error before the gradient steps: 0.2046404135803611
- Large Error after 15 gradient steps: 0.1274534669730427
- Large Weight sets before the gradient steps:
  - [-3.0, 0.23, 1.9]
- Large Weight sets after 5 gradient steps:
  - [-3.091363260585498, -0.0008727280300413337, 1.8538765031651765]
- The plot after the 5 gradient steps with the new large weight sets

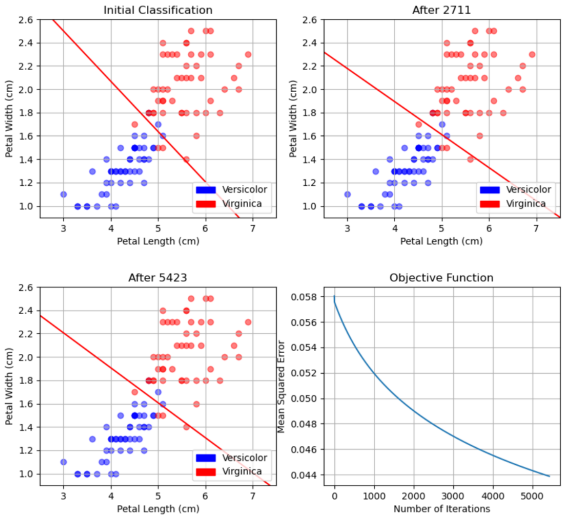


Exercise 4. Learning a decision boundary through optimization

a) Using your code above, write a program that implements gradient descent to optimize the decision boundary for the iris dataset.

```
def gradient_descent(iris, weights, colors, outputs):
    errors = [0.00001 + error_computation(iris, weights, colors, outputs), error_computation(iris, weights, colors, outputs)]
    list_of_weights = [weights, weights]
    steps = 1

    while errors[steps - 1] - errors[steps] > 0.000001:
        previous = list_of_weights[steps].copy()
        updated_weights = gradient_computation(iris, previous, colors, outputs)
        list_of_weights.append(updated_weights)
        errors.append(error_computation(iris, updated_weights, colors, outputs))
        steps += 1

    return list_of_weights, steps, errors
```

b) In your program, include code that shows the progress in two plots: the first should show the current decision boundary location overlaid on the data; the second should show the learning curve, i.e. a plot of the objective function as a function of the iteration.



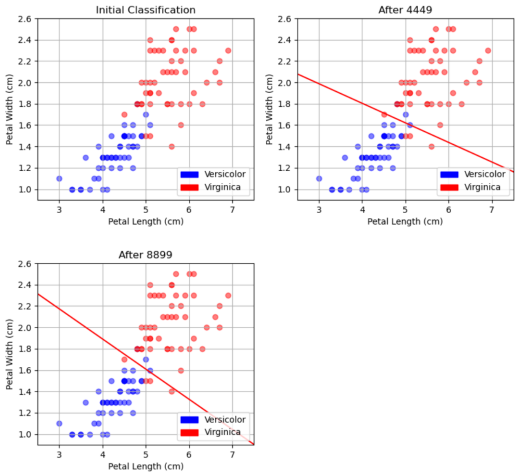
- weight_set = [-11, 1.25, 2.9]

c) Run your code on the iris data set starting from a random setting of the weights. Note: you might need to restrict the degree of randomness so that the initial decision boundary is visible somewhere in the plot. In your writeup, show the two output plots at the initial, middle, and final locations of the decision boundary.

- A set of random weights was produced by the function below. I restricted the degree of randomness as you can see below to make sure that the decision boundary was always visible somewhere in the plot on the initial plot.

```
def random_weight_generator():
    return [-1 - random.random() * 11, random.random() * 2, random.random() * 6]
```

- Plot of the initial, intermediate, and final:



d) Explain how you chose the gradient step size.

- I set the gradient step size epsilon to be 0.35. When I set it to be something like 0.1, it converges too quickly and when I set it to be something like 0.5, the step size is too high. I tested by starting from step size 0.1 and increasing the step size by 0.05 to make sure that the number of iterations keeps increasing and found that between step size 0.35 and 0.4, the number of iterations starts to decrease. As a result, I set the step size to be 0.35 which produces a reasonable result.

e) Explain how you chose a stopping criterion

- In my code, I set the stopping criteria to be 1e-5. I experimented by setting the initial weights to a set of predefined weights which is [-11, 1.25, 2.9] and checked the number of iterations. When I set it to be 1e-4, it converges too quickly and when I set it to be 1e-6, it takes about 10 times the iterations of when I set the criteria to be 1e-5.