CSC 448 Program part #1 – Due 25 September

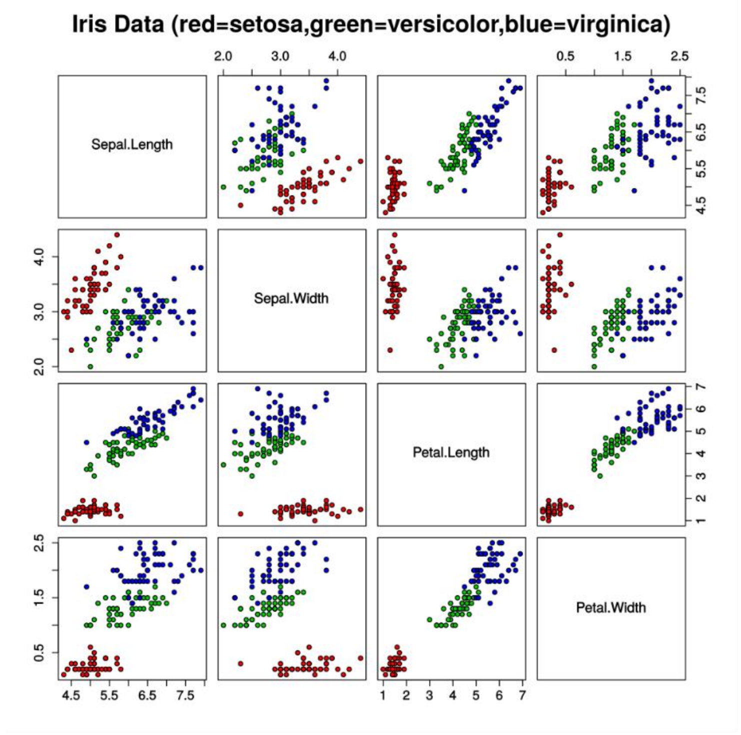
Perceptron Model

My hope is that during the semester we will build ourselves a library of machine learning tools. Let us call it `ML.py`. So our programming assignments, part of exam projects, and part of the final will be to add functionalities to this library.

Our first task will be to add the perceptron part. We will use `pandas, numpy` and `mathplotlib`, so let us import those at once. I showed yesterday how to load the **Iris** data set into a *DataFrame* object.

```
>>> import pandas as pd, numpy as np, matplotlib.pyplot as plt
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-
learning-databases/iris/iris.data', header=None)
```

You can read more about the dataset here Iris flower data set. We have three species of Iris flowers, associated with four features. Notice that I omit the header, we do not need it for this project. If we look at a scatter plot of the data we see:



In class I used *setosa* and *versicolor*, but I could equally easy has used any two (We have to use two and only two as our classification has to be binary). So our first task will be to extract the 100 first class labels.

```
>>> y = df.iloc[0:100, 4].values
>>> y
array(['Iris-setosa', …
       …,'Iris-versicolor'], dtype=object)
```

The *Perceptron* model we studied used *Y = {-1, 1}* so we better convert our label to those integer class labels

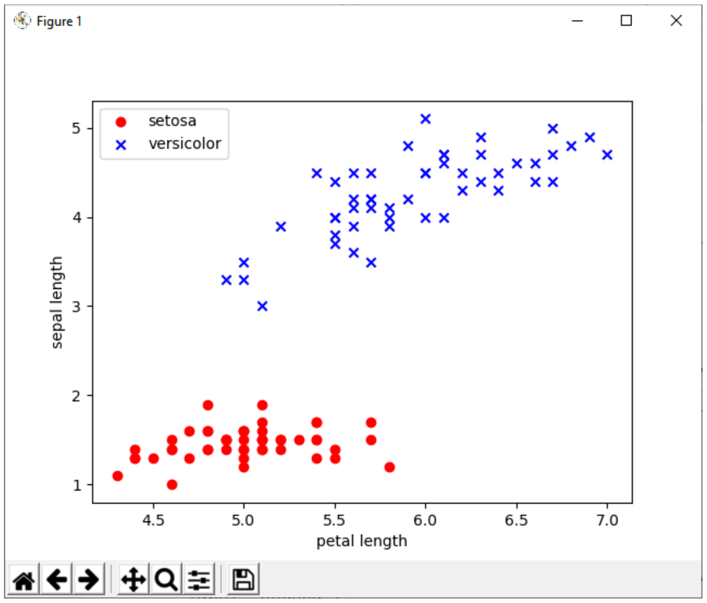
```
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> y
array([-1, -1, -1, . . . , 1,  1,  1])
```

Next we need to extract the features. We could use all four features, but as I mentioned the classes has to be separable, which meant we had to be able to cut the data into two clean groups with a hyperplane. So we are just going to use two features *sepal length* (the first feature) and *petal length* (the third feature).

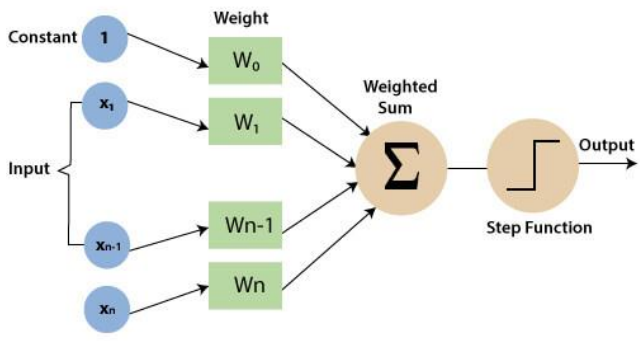
```
>>> X = df.iloc[0:100, [0, 2]].values
>>> X
   Squeezed text (100 lines).
```

Let us visualize the data, to make sure that it is separable:

```
>>> plt.scatter(X[:50, 0], X[:50, 1], color='red', marker='o', 
label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1], color='blue', 
marker='x', label='versicolor')
>>> plt.xlabel('petal length')
>>> plt.ylabel('sepal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```



We can now build and train our *perceptron* on the dataset we just extracted. While we are at it we will make sure that our *perceptron* is generic enough that we can have any number of features.



The only difference from this picture is that we are going to feed-back the error, so we can do a *weight* update. Also notice that we feed a constant into w_0 this is what we in class called the bias, so we will need our *w*-vector to one element larger than the number of features in *X*.

Our goal is to be able to do the following:

```
>>> from ML import Perceptron
>>> pn = Perceptron(0.1, 10)
```

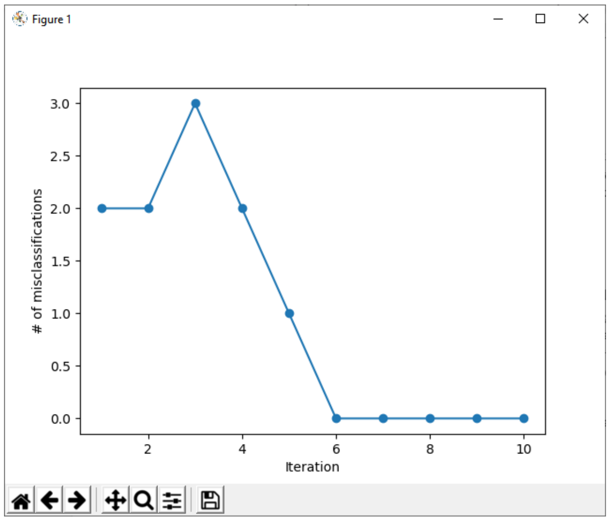
I am importing class `Perceptron` from `ML.py` and then I am initializing it. The first argument is the *learning rate* and this is a number between 0 and 1. A larger value makes the weight changes more volatile. The second argument is the number of iterations we want to do. This to make sure we halt in case we will not converged to a solution (i.e. we still have training errors after n iterations). This page (Perceptron) pretty much explains what we want to do (don't you just love how much I use Wiki 😉).

After we run the *fit*-function on the *perceptron*, we can extract (so we are able to plot) the misclassification errors:

```
>>> pn.fit(X, y)
>>> pn.errors
[2, 2, 3, 2, 1, 0, 0, 0, 0, 0]
```

Here I fitted *X* to *y* i.e. the algorithm found the appropriate values for the weights (*w*). I also printed the error for each iteration.

```
>>> plt.plot(range(1, len(pn.errors) + 1), pn.errors,
marker='o')
>>> plt.xlabel('Iteration')
>>> plt.ylabel('# of misclassifications')
>>> plt.show()
```



The perceptron converged after 6 iterations, and from there and on we should have been able to classify all training samples with a zero error rate.

Time to look what I have in the `perceptron` class:

`pn.errors`	(ok, this is the array of errors in each iteration)
`fit(X,y)`	(the function that creates our hypothesis (*w*) from *X* and *y*)
`net_input(X)`	(the weighted sum of an object or all objects in ***X***: $\boldsymbol{x_i}\cdot\boldsymbol{w}$ + *bias*)
`niter`	(number of iterations)
`predict(X)`	(returns the predicted class label for an object, every object in X)
`rate`	
`weight`	(the current weight array, [w_0, w_1, …, w_n])

```
>>> pn.net_input(X)
array([-1.32 , -1.184, . . . ,  1.592,  3.186])
```

```
>>> pn.predict(X)
array([-1, -1, . . . , 1,  1])
```

As I said earlier, our weight vector creates a hyper-plane or a decision boundary between our two classes. We can look at our *w*-array

```
>>> pn.weight
array([-0.4 , -0.68,  1.82])
```

As this is a 2D example we should be able to visualize this boundary. There is no real good function in `Matplotlib` to do this, so we are going to create a helper function and put that in our `ML.py` library. Ok, what we need to do is:

1. Define a number of colors and markers. We can do that using the ListedColormap in `matplotlib`
2. Find the `min` and the `max` for our features.

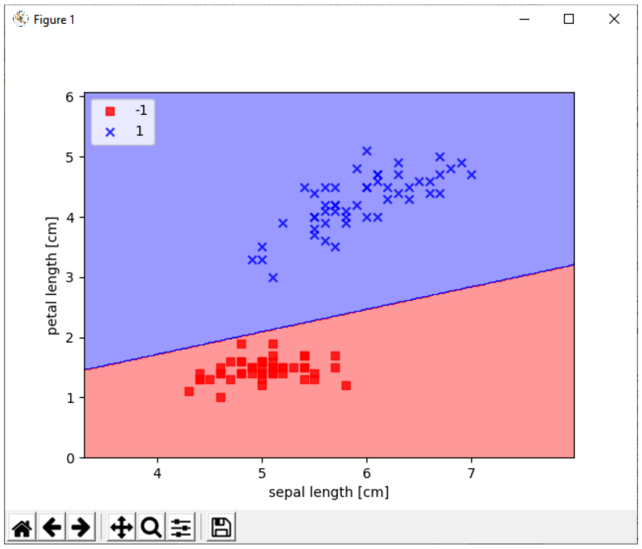
3. Use the features to create two grid arrays (using `meshgrid` and `arrange` from `numpy`).
4. Our perceptron was trained on two features, so our grid array has to be flatten into a matrix with the same number of columns as our *X*, so we can use the predict method to predict the class label (*Z*) on our grid points.
5. Reshape *Z* (our predicted class labels) into the same dimensions as our gridpoints.
6. Draw our contour plot (using `countorf` from `matplotlib`)

Ok, we will need to import both `numpy` and `matplotlib` into our `ML.py` to make sure we encounter no problems.

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
   # setup marker generator and color map
   markers = ('s', 'x', 'o', '^', 'v')
   colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
   cmap = ListedColormap(colors[:len(np.unique(y))])

   # plot the decision surface
   x1_min, x1_max = X[:,  0].min() - 1, X[:, 0].max() + 1
   x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
   xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
   np.arange(x2_min, x2_max, resolution))
   Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
   Z = Z.reshape(xx1.shape)
   plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
   plt.xlim(xx1.min(), xx1.max())
   plt.ylim(xx2.min(), xx2.max())

   # plot class samples
   for idx, cl in enumerate(np.unique(y)):
      plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
      alpha=0.8, c=cmap(idx),
     marker=markers[idx], label=cl)
```



Programming Assignment:

This is the first part of your machine learning library. We will focus on the perceptron.

1. Create the perceptron class:
    a. A new perceptron instance should be created with a rate and the numbers of iterations.
    b. It should be generic enough that it can take a training set of any size. My write-up only uses two features, but my code can take any number of features.
    c. The class has four functions:
        i. `__init__()`
        ii. `fit()`
        iii. `net_input()`
        iv. `predict()`
2. The example in the text always runs the provided number of iterations, even if it has converged long before that. Fix that problem (i.e. in the text case it should run only 6 iterations).
3. Fix and put the `plot_decison_regions` into your ML-library, so you can use it.
4. Try your implementation on other pairs, and other triples of the Iris-data set.

```
# ML.py

import numpy as np

class Perceptron(object):
   def __init__(self, rate = 0.01, niter = 10):
      self.rate = rate
      self.niter = niter

   def fit(self, X, y):
      """Fit training data
      X : Training vectors, X.shape : [#samples, #features]
      y : Target values, y.shape : [#samples]"""

      # weights: create a weights array of right size
      # and initialize elements to zero

      # Number of misclassifications, creates an array
      # to hold the number of misclassifications

      # main loop to fit the data to the labels
      for i in range(self.niter):
         # set iteration error to zero
         # loop over all the objects in X and corresponding y element
         for xi, target in zip(X, y):
            # calculate the needed (delta_w) update from previous step
            # delta_w = rate * (target – prediction current object)

            # calculate what the current object will add to the weight

            # set the bias to be the current delta_w

            # increase the iteration error if delta_w != 0

         # Update the misclassification array with # of errors in iteration

      # return self

   def net_input(self, X):

      """Calculate net input"""
      # return the return the dot product: X.w + bias

   def predict(self, X):
      """Return class label after unit step"""
      return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Notice: In the `Fit`-function, we predict just a single object (x_i) and calculate the net input for just that object, but as I showed in my write-up we can predict an entire array of objects, and calculate the net input array for each object.