Optimization Methods for Machine Learning

Assignment # 1

1536242 - Paolo Tamagnini

24 October 2016

1 Question 1

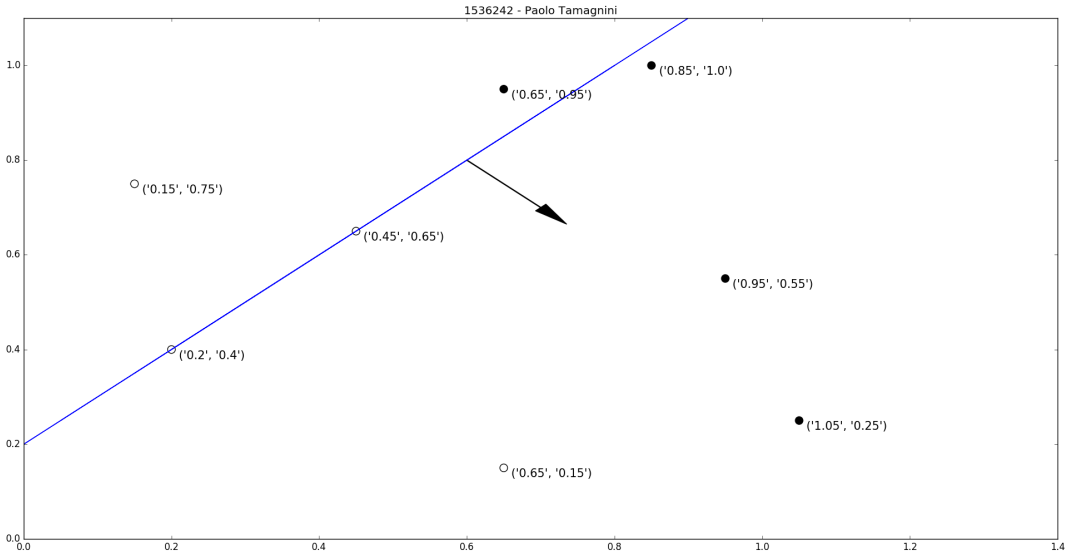
1.1 Point 1

The chosen points are the following:

```
('0.15', '0.75')
('0.2', '0.4')
('0.45', '0.65')
('0.65', '0.95')
('0.65', '0.15')
('0.85', '1.0')
('0.95', '0.55')
('1.05', '0.25')
```

1.2 Point 2

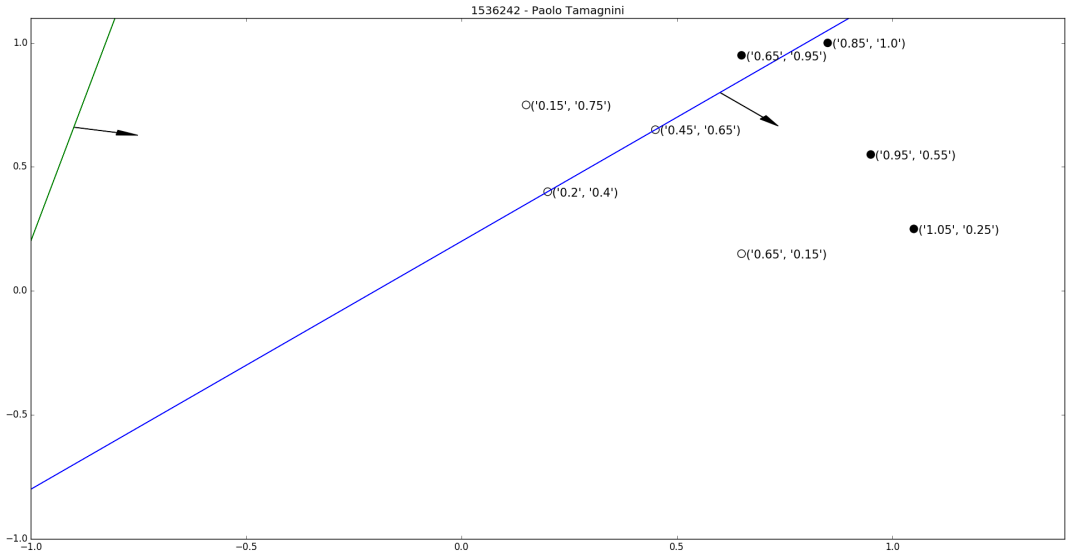
The initial perceptron with weights $b^0 = 0.2$, $w_1^0 = 1$ and $w_2^0 = -1$ is represented by the line in the following chart:



This line represents the intersection between the function plane and the plane of the graph, so it gives a visual representation on how the function is classying our points. As we can see half of the point are misclassified by this first perceptron. The misclassified samples are 4 out of the 8. The ones on which the line is passing (('0.2', '0.4') and ('0.45', '0.65')) are classified correctly because for $t \geq 0$ the classification function gives predicted label $\hat{y} = 1$. The function for those points have $t = 0$ and this means that they are classified correctly.

1.3 Point 3

The first iteration of the perceptron, enven then it takes the splitting line in a far position, keeps the number of errors to 4. As we can see from the picture the new green line is classifying uncorrectly all the black labels ($y = -1$), but at the same is classifying correctly all the white ones ($y = 1$).



To do this I used the following python script and we stop for $k = 1$ so that the weigths are updated only once.

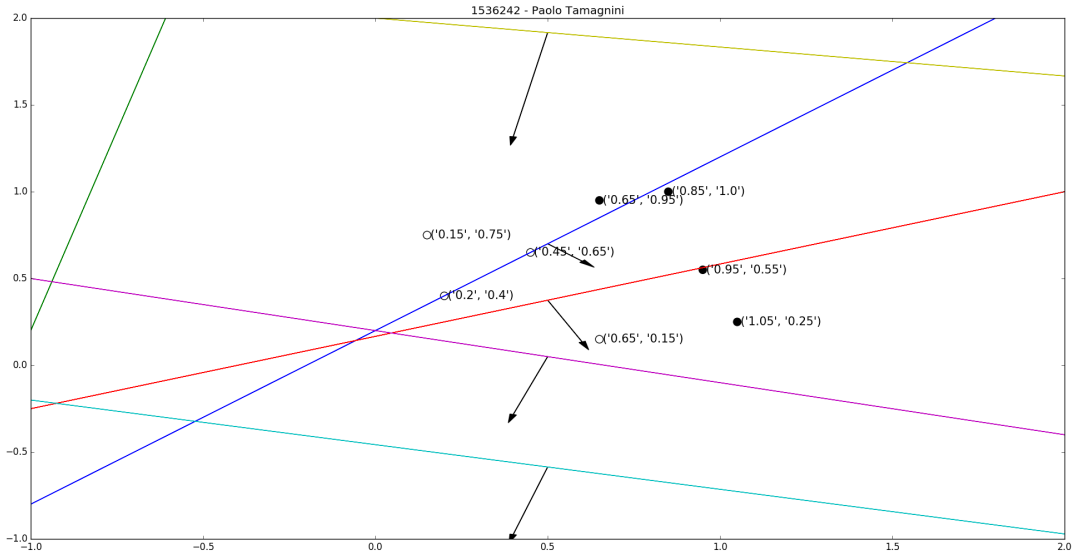
```
stop = 1
omega = np.array([1,-1])
b = 0.2
cc = 0
k = 0
omega_list = []
b_list = []
omega_list.append(omega)
b_list.append(b)
while cc < P :
    for i in range(0,P):
        t = b + np.dot(omega,x_matrix[i,:])
        if t >= 0:
            sgn = 1
        elif t < 0:
            sgn = -1
        else:
            print 'error'
        if y_vector[i]*(sgn) <= 0:
            omega = omega + y_vector[i]*x_matrix[i,:]
            omega_list.append(omega)
            b = b + y_vector[i]
            b_list.append(b)
            k = k + 1
            if k == stop:
                break


        else:
            cc +=1

    if cc < P:
        cc = 0
    if k == stop:
        break
```

1.4 Point 4

We will now use the same algorith this time with the variable $stop = 5$ (we are repeating the process four more times). With 4 more updates of the weights we get the following values.

```
step 0 : w_1=  1.00 | w_2= -1.00 | b=  0.2
step 1 : w_1=  1.15 | w_2= -0.25 | b=  1.2
step 2 : w_1=  0.50 | w_2= -1.20 | b=  0.2
step 3 : w_1= -0.45 | w_2= -1.75 | b= -0.8
step 4 : w_1= -0.30 | w_2= -1.00 | b=  0.2
step 5 : w_1= -0.10 | w_2= -0.60 | b=  1.2
```

Counting also the initialization line we will have a total of 6 lines in our chart.



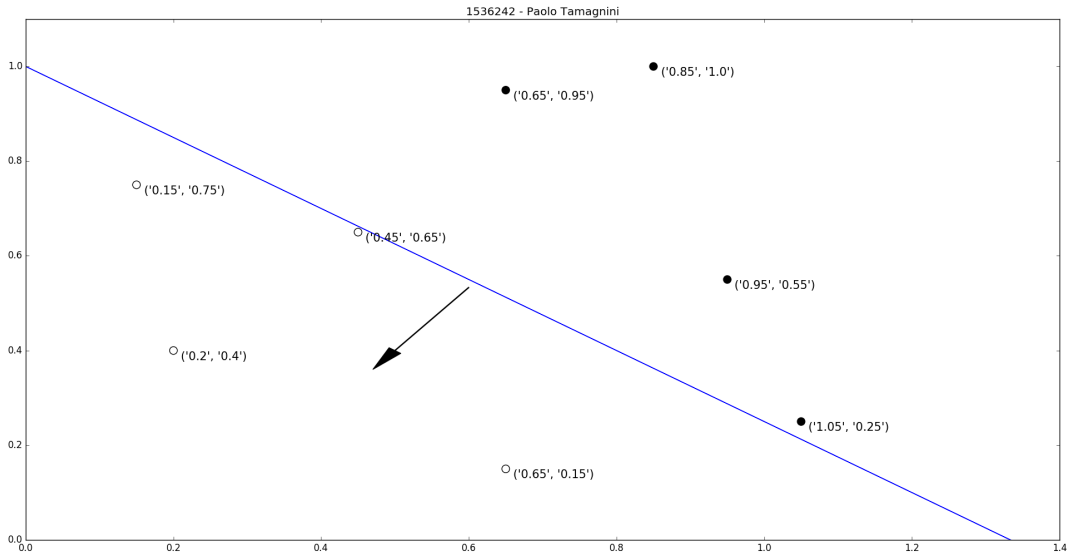
2 Question 2

2.1 Point 1

The script to let the program run until it reaches convergence is the same one as before just without the *break* command. Anyway to make sure the computer doesn't get stuck in the cycle, for example in case the data points are not linearly separable, we leave the *break* command set with $stop = 1000$. This is not our case where we reach convergence at the 9th update for a total of 10 lines. The number of steps required is therefore 9.

```
step 0 : w_1 =  1.00 | w_2 = -1.00 | b =  0.2 |
step 1 : w_1 =  1.15 | w_2 = -0.25 | b =  1.2 |
step 2 : w_1 =  0.50 | w_2 = -1.20 | b =  0.2 |
step 3 : w_1 = -0.45 | w_2 = -1.75 | b = -0.8 |
step 4 : w_1 = -0.30 | w_2 = -1.00 | b =  0.2 |
step 5 : w_1 = -0.10 | w_2 = -0.60 | b =  1.2 |
step 6 : w_1 = -0.75 | w_2 = -1.55 | b =  0.2 |
step 7 : w_1 = -0.10 | w_2 = -1.40 | b =  1.2 |
step 8 : w_1 = -1.05 | w_2 = -1.95 | b =  0.2 |
step 9 : w_1 = -0.90 | w_2 = -1.20 | b =  1.2 |
```

We see here the result of the last 9th step (the 10th line) where every point is correctly classified.



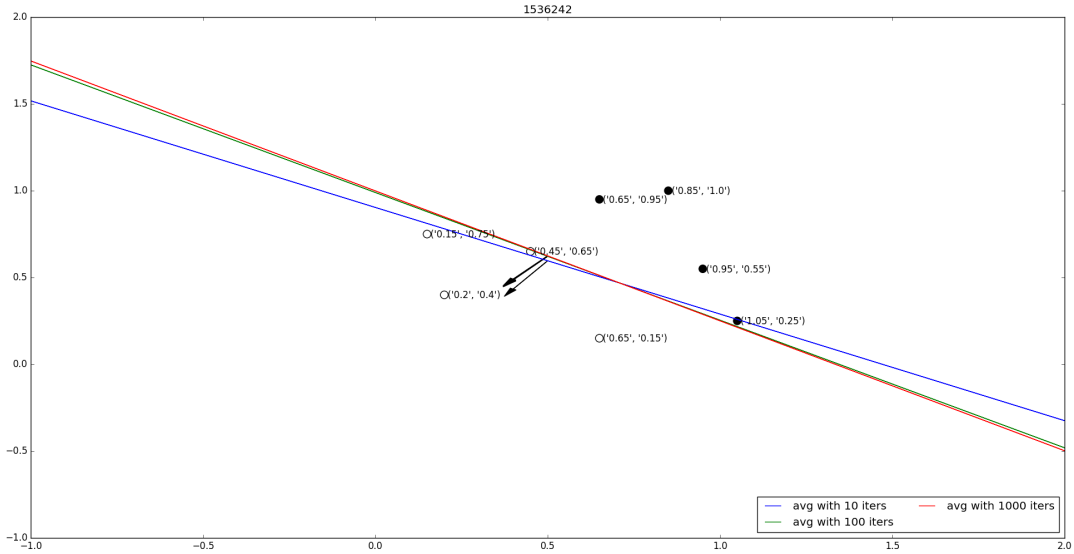
2.2 Point 2

The average perceptron algorithm was implemented in the following way:

```
stop = 100
omega = np.array([1,-1])
b = 0.2
k = 0
omega_avg = np.array([0,0])
b_avg = 0
v = 0

omega_list = []
b_list = []
omega_list.append(omega)
b_list.append(b)

omega_list_avg = []
b_list_avg = []
omega_list_avg.append(omega_avg)
b_list_avg.append(b_avg)

v_list = []
v_list.append("go")
its = 0
while its <= stop :
    for i in range(0,P):
        t = b + np.dot(omega,x_matrix[i,:])
        if t >= 0:
            sgn = 1
        elif t < 0:
            sgn = -1
        else:
            print 'error'
```

```
            if y_vector[i]*(sgn) <= 0:
                omega = omega + y_vector[i]*x_matrix[i,:]
                omega_list.append(omega)

                b = b + y_vector[i]
                b_list.append(b)

                omega_avg = omega_avg + v*omega_list[k]
                omega_list_avg.append(omega_avg)

                b_avg = b_avg + v*b_list[k]
                b_list_avg.append(b_avg)

                k = k + 1
                v_list.append(v)
                v = 1

            else:
                v = v + 1
        its = its + 1

#last update
omega_avg = omega_avg + v*omega_list[k]
omega_list_avg.append(omega_avg)

b_avg = b_avg + v*b_list[k]
b_list_avg.append(b_avg)
v_list.append(v)
```

It is important to underline that here the *stop* variable is not optional like before since the convergence will not be reached unless we don't hit the desired number of iterations *stop*. The following results are achieved with *stop* equal to 10, 100 and 1000. In the previous algorithm *stop* was used to count the number of weight updates computed. In this one it is different since we are counting the number of *while* cycles performed.



The number of lines updates performed in the *while* cycle is always 9. The difference achieved in

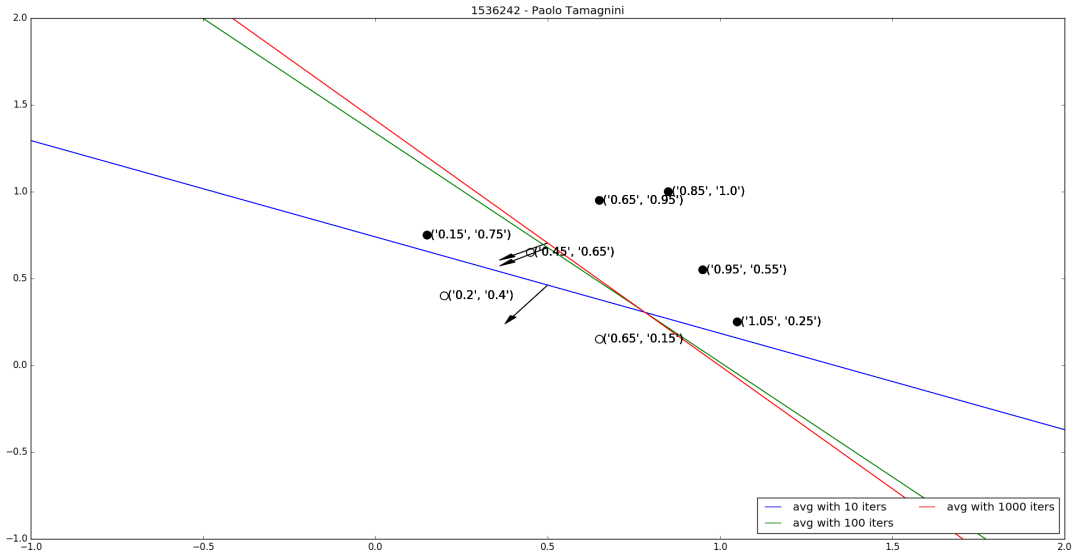
using different values for the *stop* variable is in the very last update outside the *while* cycle of w_{avg} and b_{avg}. In the following table we can see how the different in varying the maximum number of iterations influences just the 10th step while everything else does not change.

	v_stop10	v_stop100	v_stop1000	w_avg_stop10	w_avg_stop100	w_avg_stop1000	b_avg_stop10	b_avg_stop100	b_avg_stop1000
step_0	go	go	go	[0, 0]	[0, 0]	[0, 0]	0.0	0.0	0.0
step_1	0	0	0	[0, 0]	[0, 0]	[0, 0]	0.0	0.0	0.0
step_2	3	3	3	[3.45, -0.75]	[3.45, -0.75]	[3.45, -0.75]	3.6	3.6	3.6
step_3	3	3	3	[4.95, -4.35]	[4.95, -4.35]	[4.95, -4.35]	4.2	4.2	4.2
step_4	2	2	2	[4.05, -7.85]	[4.05, -7.85]	[4.05, -7.85]	2.6	2.6	2.6
step_5	1	1	1	[3.75, -8.85]	[3.75, -8.85]	[3.75, -8.85]	2.8	2.8	2.8
step_6	2	2	2	[3.55, -10.05]	[3.55, -10.05]	[3.55, -10.05]	5.2	5.2	5.2
step_7	1	1	1	[2.8, -11.6]	[2.8, -11.6]	[2.8, -11.6]	5.4	5.4	5.4
step_8	2	2	2	[2.6, -14.4]	[2.6, -14.4]	[2.6, -14.4]	7.8	7.8	7.8
step_9	2	2	2	[0.5, -18.3]	[0.5, -18.3]	[0.5, -18.3]	8.2	8.2	8.2
step_10	72	792	7992	[-64.3, -104.7]	[-712.3, -968.7]	[-7192.3, -9608.7]	94.6	958.6	9598.6

The last computed averaged weights are really different but the relative lines are almost overlapping. This is because the weight ratios that are needed to draw the lines are not that different. Anyway the difference in the value is related to the amount v used to compute the last update. Indeed the more iteration we decide to do, the more will increase v and the bigger will be the value of the weight computed for last. This way the last update will have more influence over all the previous one. This is desired because the previous updates are not as precise and stable, since they don't last long, and we want to reduce their influence by increasing the contribution of updates that are lasting longer. Once we reach convergence on the 9th step, we don't update the weights anymore and we wait to reach the desired number of iterations and by doing so we increase v.
By comparing the algorith of the previous question and this average one we see how the oscillations in the average one are less strong thanks to v (click here).

2.3 Point 3

In the case where the point cannot be linearly shattered we do not reach convergence but by increasing the number of maximum iteration we get weights for the best suitable line. If we were using the normal perceptron the line would have oscillated around the position of the average one instead of stopping there.



	stop_10	stop_100	stop_1000
w1	-112.30	-2195.05	-23018.95
w2	-202.25	-1662.20	-16265.70
b	149.60	2224.60	22968.60