

Assignment 1

Name: Ahmed Ashraf

Sec: 1

BN: 2

```
In [195]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(50)
```

Exercise 1.10

Here is an experiment that illustrates the difference between a single bin and multiple bins. Run a computer simulation for flipping 1,000 fair coins. Flip each coin independently 10 times. Let's focus on 3 coins as follows: c_1 is that first coin flipped; c_{rand} is a coin you choose at random; c_{min} is the coin that had the minimum frequency of heads (pick the earlier one in case of a tie). Let v_1 , v_{rand} and v_{min} be the fraction of heads you obtain for the respective three coins.

(a) What is μ for the three coins selected?

Assuming a fair coin, $P(\text{heads}) = P(\text{tails}) = 0.5$. Then, for c_1 , c_{rand} , c_{min} , $\mu = 0.5$ regardless of the outcomes of the flips.

```
In [196]: def experiment(number_of_coins, number_of_flips):
    heads = np.zeros(number_of_coins)
    for _ in range(number_of_flips):
        p = np.random.uniform(size=number_of_coins)
        heads[p > 0.5] += 1

    random_coin = np.random.choice(number_of_coins)
    minimum_coin = np.argmin(heads)
    heads /= number_of_flips

    v1 = heads[0]
    vrand = heads[random_coin]
    vmin = heads[minimum_coin]

    return v1, vrand, vmin
```

(b) Repeat the entire experiment a large number of times (e.g., 100,000 runs of the entire experiment) to get several instances of v_1 , v_{rand} , and v_{min} and plot the histograms of the distributions of v_1 , v_{rand} , and v_{min} . Notice that which coins end up being c_{rand} , and c_{min} may differ from one run to another.

```
In [197]: number_of_coins = 1000
number_of_flips = 10
number_of_runs = 100000

def print_fractions(v1, vrand, vmin):
    print("v1= ", v1)
    print("vrand= ", vrand)
    print("vmin= ", vmin)

v1, vrand, vmin = experiment(number_of_coins, number_of_flips)
print_fractions(v1, vrand, vmin)

v1s, vrand, vmins = [], [], []
for _ in range(number_of_runs):
    v1, vrand, vmin = experiment(number_of_coins, number_of_flips)
    v1s.append(v1)
    vrand.append(vrand)
    vmins.append(vmin)

v1= 0.8
vrand= 0.4
vmin= 0.0
```

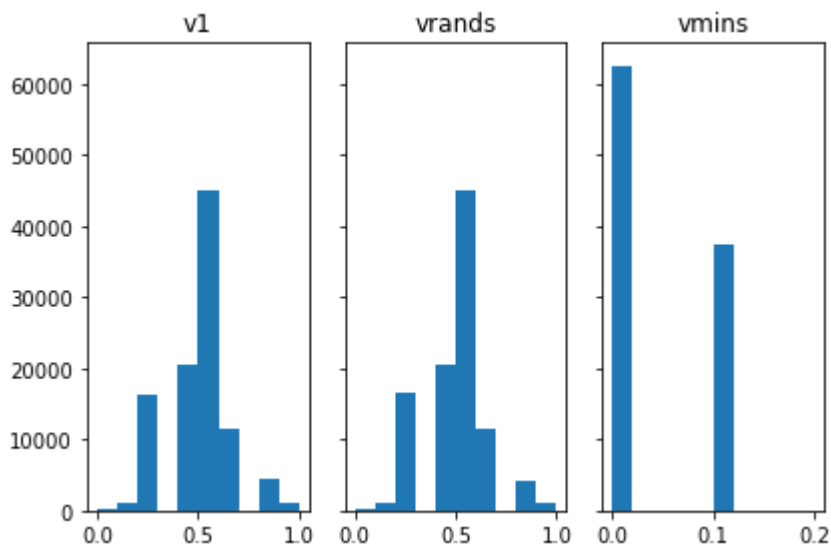
```
In [198]: def plot_fractions_hist(vls, vrand, vmins):
    n_bins = 10
    _, axs = plt.subplots(1, 3, sharey=True, tight_layout=True)

    axs[0].hist(vls, bins=n_bins)
    axs[0].set_title("v1")

    axs[1].hist(vrand, bins=n_bins)
    axs[1].set_title("vrand")

    axs[2].hist(vmins, bins=n_bins)
    axs[2].set_title("vmins")

    plot_fractions_hist(vls, vrand, vmins)
```



(c) Using (b), plot estimates of $\mathbb{P}[|\nu - \mu| > \epsilon]$ as a function of ϵ , together with the Hoeffding bound $2e^{-2\epsilon^2 N}$ (on the same graph).

```
In [199]: def hoeffding(eps, N):
    return 2 * np.exp(-2 * (eps**2) * N)
```

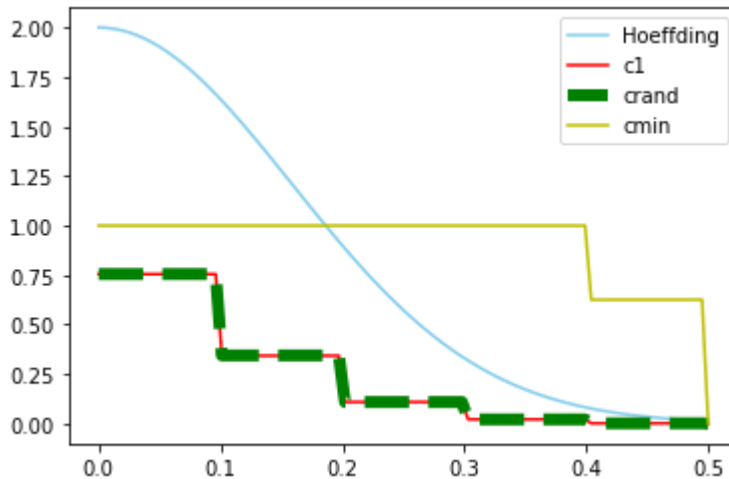
```
In [200]: mu = 0.5
eps = np.linspace(0, 5, 100) / number_of_flips
vls_diff, vrand_diff, vmins_diff = np.abs(np.array(vls) - mu), np.abs(
    np.array(vrand) - mu), np.abs(np.array(vmins) - mu)
p1, prand, pmin = [], [], []

for i in range(eps.shape[0]):
    eps_i = eps[i]
    p1.append(np.sum(vls_diff > eps_i) / number_of_runs)
    prand.append(np.sum(vrand_diff > eps_i) / number_of_runs)
    pmin.append(np.sum(vmins_diff > eps_i) / number_of_runs)

hoeffding_bounds = hoeffding(eps, number_of_flips)
```

```
In [201]: plt.plot(eps, hoeffding_bounds, color='skyblue', label='Hoeffding')
plt.plot(eps, p1, color='r', label='c1')
plt.plot(eps, prand, color='g', label='crand', linewidth=6, linestyle=
e= "--")
plt.plot(eps, pmin, color='y', label='cmin')
plt.legend()
```

Out[201]: <matplotlib.legend.Legend at 0x7f5ed7c4e4c0>



(d) Which coins obey the Hoeffding bound, which ones do not? Explain why.

The c1 and crand coins obey the Hoeffding inequality, but cmin does not. That's because cmin was chosen during the experiments runs based on the sample results which violates the Hoeffding's conditions stating that the hypothesis has been chosen before experiments runs.

Relate part (d) to the multiple bins in Figure 1.

Choosing the cmin during the experiment is equivalent to learning algorithm for the minimum frequency hypothesis. The crand and c1 are chosen before the experiment which follows the condition of Hoeffding.

Exercise 1.4

This problem leads you to explore the perceptron learning algorithm with data sets of different sizes and dimensions. You will have to generate random dataset as described in Exercise 1.4 of the book which works as follows: For a specific number of dimensions d , Choose a random target function f (randomly pick values for the $d+1$ weights of the perceptron), then generate N random data points $\{x_i \in \mathbb{R}^d \mid i \in [1, N]\}$ and label each data point $y_i = f(x_i)$ according to the selected target function f .

(a) Generate a linearly separable data set of size 20 where $d=2$. Plot the examples $\{x_n\}$ as well as the target function f on a plane. Be sure to mark the examples from different classes y differently and add y s to the axes of the plot.

```
In [202]: def generate_data(N, d):
           X = np.concatenate((np.random.randn(N, d), np.ones((N, 1))), axis=1)
           return X

           def generate_target_function(X, d):
               f = np.random.randn(d+1)
               y = np.sign(np.dot(X, f)).reshape((X.shape[0], 1))
               return f, y
```

```
In [203]: def plot_line(ax, X, w, color, label):
           xTarget = np.linspace(int(np.min(X[:, 0])*100), int(np.max(X[:, 0])+0.5)*100, num=1000)/100
           yTarget = -(w[2]+w[0]*xTarget)/w[1]
           ax.plot(xTarget, yTarget, c=color, label=label)

           def plot_data(X, y, f, w=None):
               posX = X[np.where(y > 0)[0]]
               negX = X[np.where(y < 0)[0]]

               _, ax = plt.subplots(figsize=(10, 6))
               ax.scatter(posX[:, 0], posX[:, 1], marker="+", s=50, c='red', label='y=+1')
               ax.scatter(negX[:, 0], negX[:, 1], marker="$-$", s=50, c='blue', label='y=-1')

               plot_line(ax, X, f, "k", "Target")

               if w is not None:
                   plot_line(ax, X, w, "c", "Perceptron")

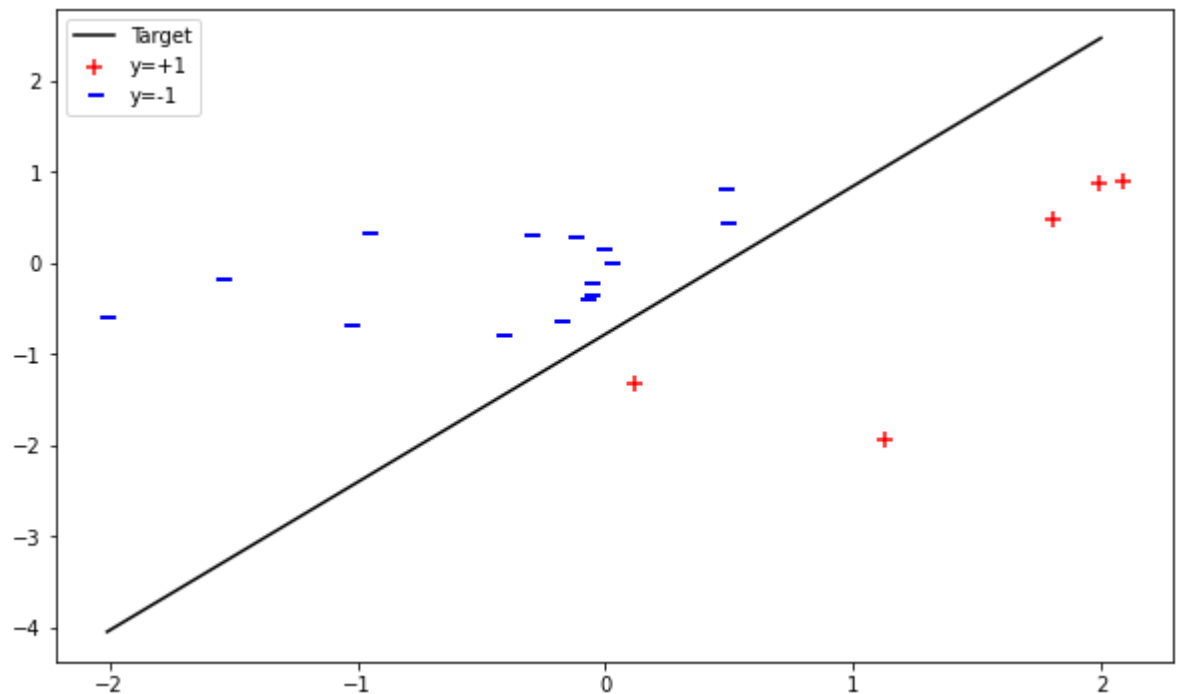
               plt.legend(loc='best')
               plt.show()
```

```

In [238]: d = 2
N = 20

X = np.zeros(N)
y = np.ones((N, 1))
while np.all(y == 1) or not np.any(y == 1):
    X = generate_data(N, d)
    f, y = generate_target_function(X, d)
plot_data(X, y, f)

```



(b) Run the perceptron learning algorithm on the data set above. Report the number of updates that the algorithm takes before converging. Plot the examples, the target function f and the final hypothesis g in the same figure. Comment on whether f is close to g .

```

In [205]: def perceptron_experiment(X, f, y, plot = True, random_pick = False
):
    w = np.ones(X.shape[1])

    noIterations = 0
    wrongPreds = True
    maxIterations = 10000
    while wrongPreds and noIterations < maxIterations:
        if random_pick:
            t = np.arange(X.shape[0])
            np.random.shuffle(t)
            X, y = X[t, :], y[t]

            wrongPreds = False
            for i, label in enumerate(y):
                if label * np.dot(X[i, :], w) < 0:
                    wrongPreds = True
                    w = w + X[i, :]*label
                    break

            noIterations += 1

        if plot:
            plot_data(X, y, w, f)

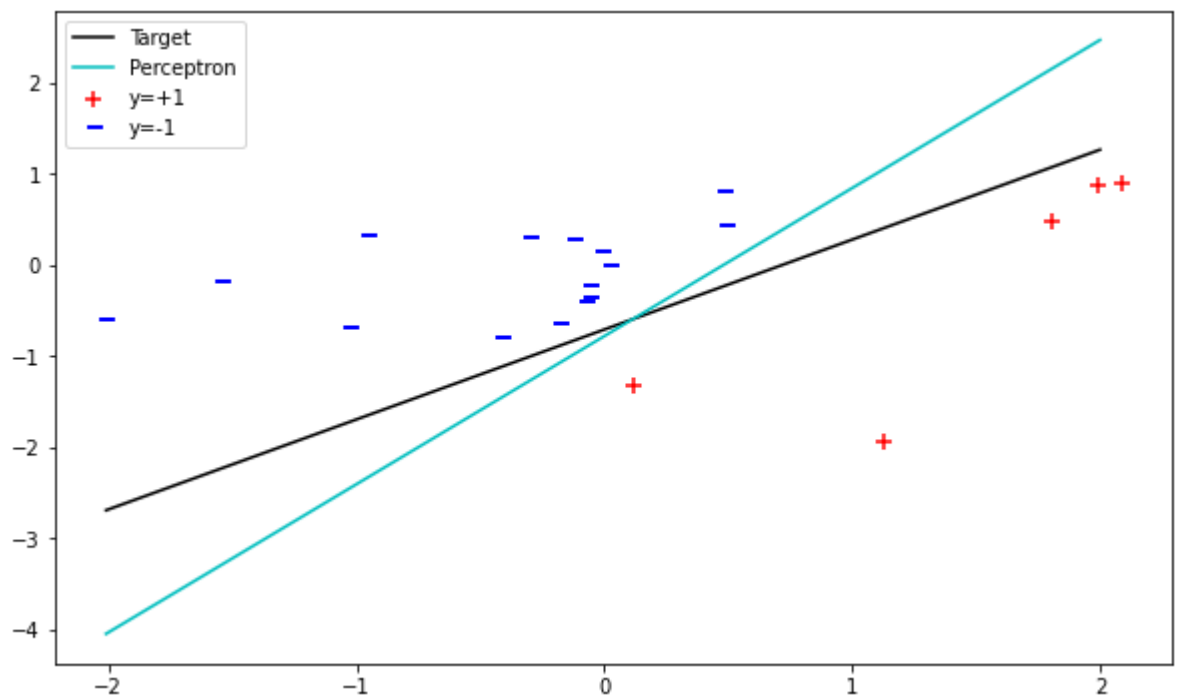
    return w, noIterations

```

```

In [239]: w, noIterations = perceptron_experiment(X, f, y)
print("Number of iterations: ", noIterations)

```



Number of iterations: 7

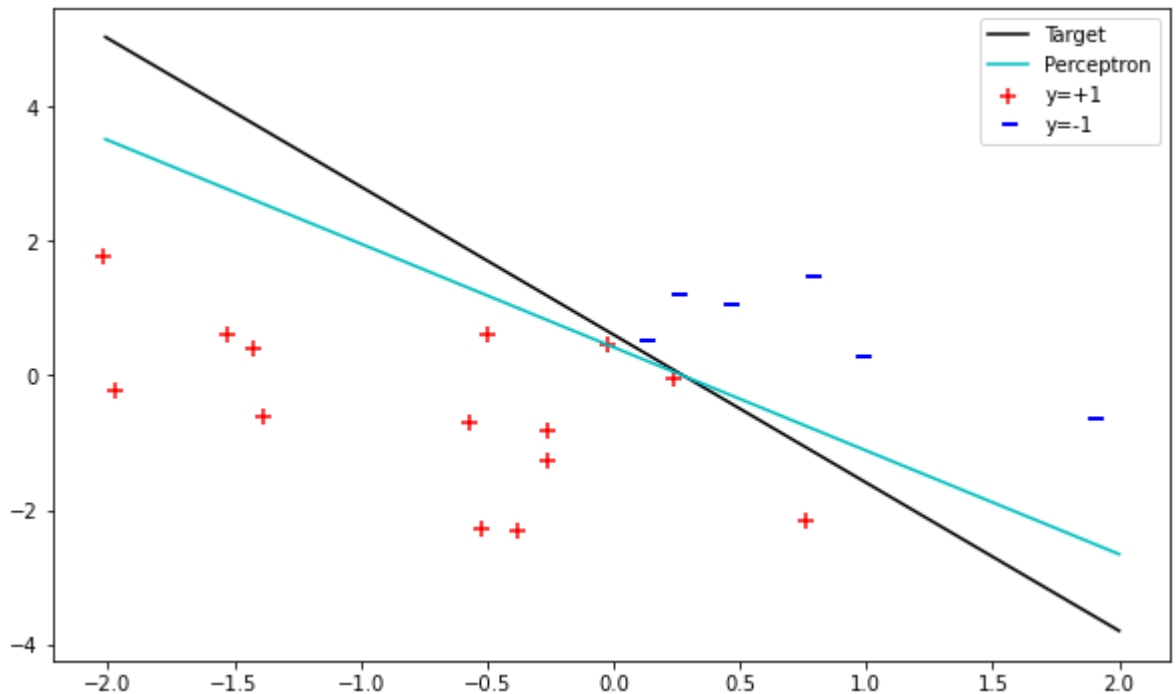
Usually, the number of iterations is very small (tens of iterations) which is proportionally means that it converts very fast but the generated hypothesis, g , doesn't necessarily close to the true target function, f .

(c) Repeat everything in (b) with another randomly generated data set of size 20. Compare your results with (b).

```
In [207]: def run_perceptron_experiment(N, d):
          X = np.zeros(N)
          y = np.ones((N, 1))
          while np.all(y == 1) or not np.any(y == 1):
              X = generate_data(N, d)
              f, y = generate_target_function(X, d)

          w, iterations = perceptron_experiment(X, f, y)
          print("Number of iterations: ", iterations)
          return X, y, f, w, iterations
```

```
In [248]: X, y, f, w, iterations = run_perceptron_experiment(20, 2)
```

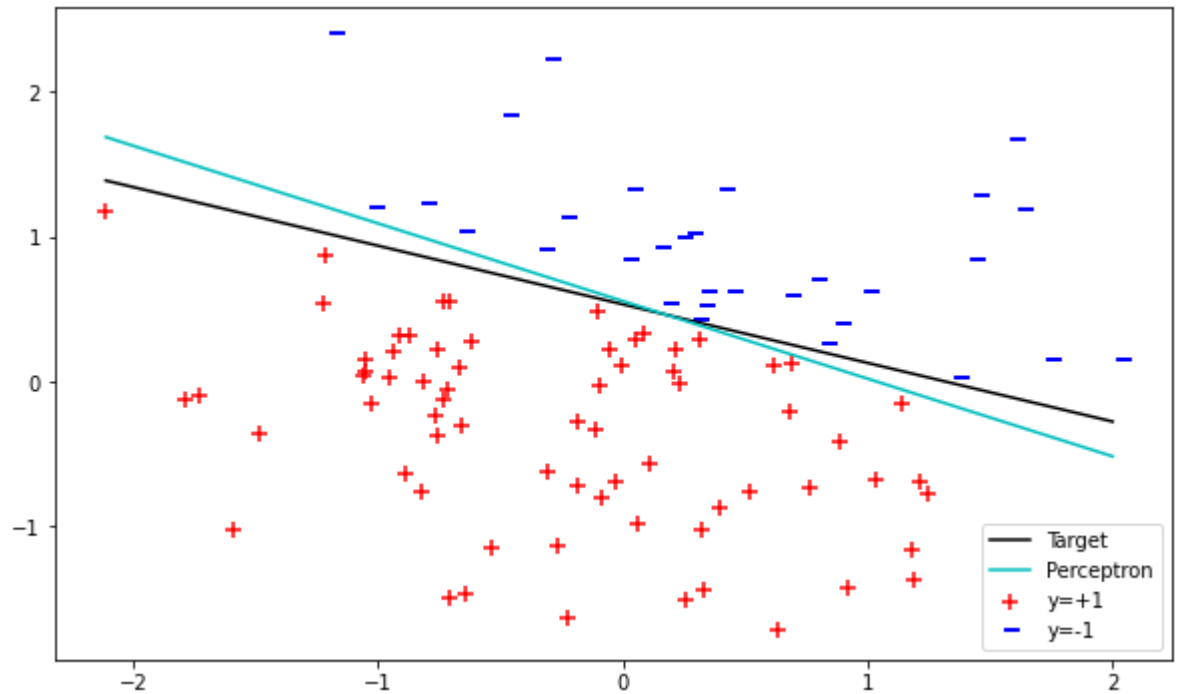


Number of iterations: 5

Similar results to the results obtained in (b).

(d) Repeat everything in (b) with another randomly generated data set of size 100. Compare your results with (b).


```
In [232]: X, y, f, w, iterations = run_perceptron_experiment(100, 2)
```

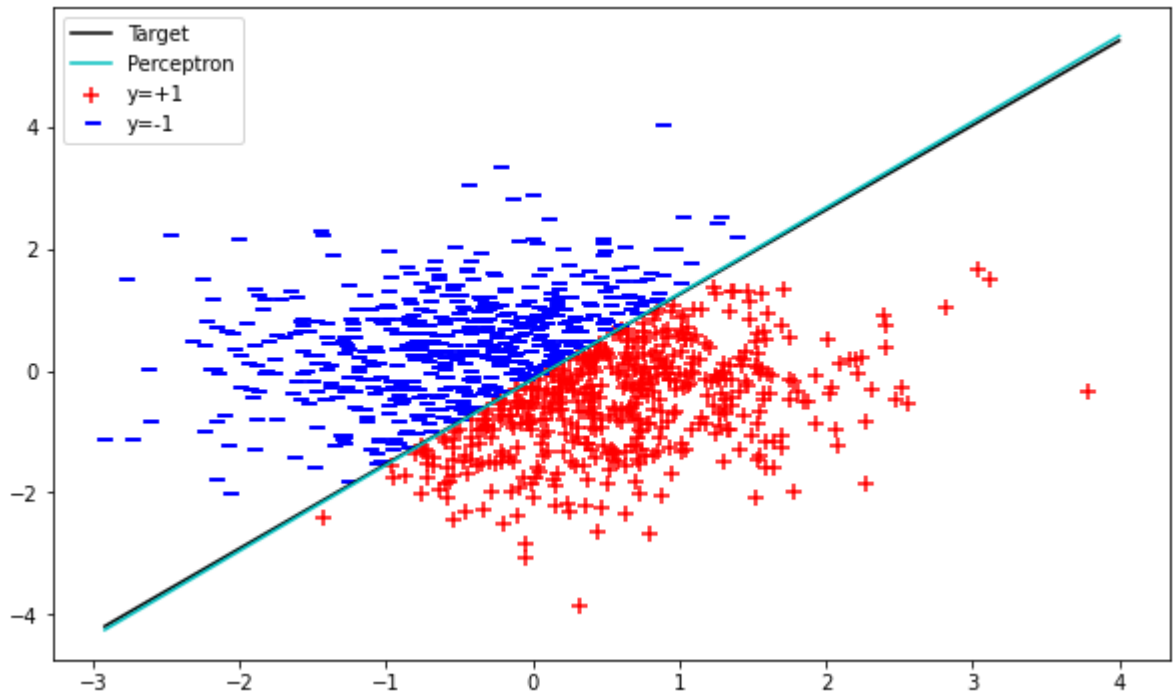


Number of iterations: 20

Still converges relatively fast with relatively small number of iterations but its accuracy is better than $N=10$.

(e) Repeat everything in (b) with another randomly generated data set of size 1,000. Compare your results with (b).

```
In [233]: X, y, f, w, iterations = run_perceptron_experiment(1000, 2)
```

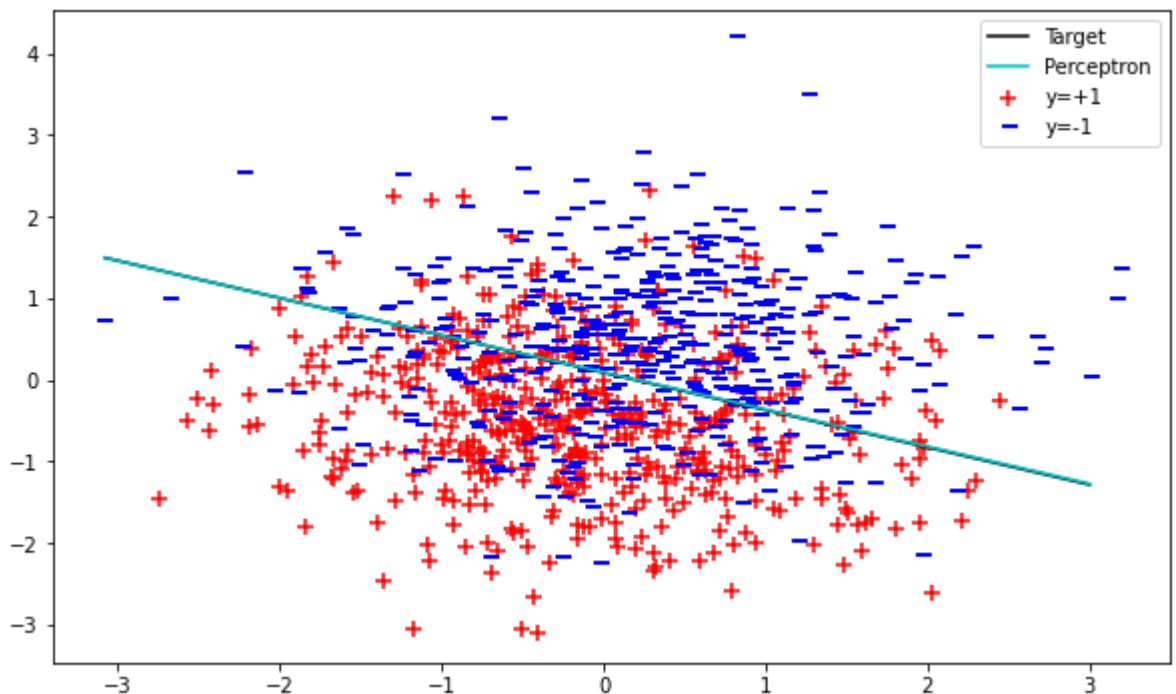


Number of iterations: 598

It's relatively slow with relatively large number of iterations (hundreds of iterations) but has much better accuracy. It's generated hypothesis, g , is almost the same as true target function, f .

(f) Modify the algorithm such that it takes $d=10$ instead of 2. Randomly generate a linearly separable data set of size 1,000 and feed the data set to the algorithm. How many updates does the algorithm take to converge?

```
In [234]: X, y, f, w, iterations = run_perceptron_experiment(1000, 10)
```



Number of iterations: 8200

It's usually takes thousands of iterations to converge.

(g) Repeat the algorithm on the same data set as (f) for 100 experiments. In the iterations of each experiment, pick $x(t)$ randomly instead of deterministically. Plot a histogram for the number of updates that the algorithm takes to converge.

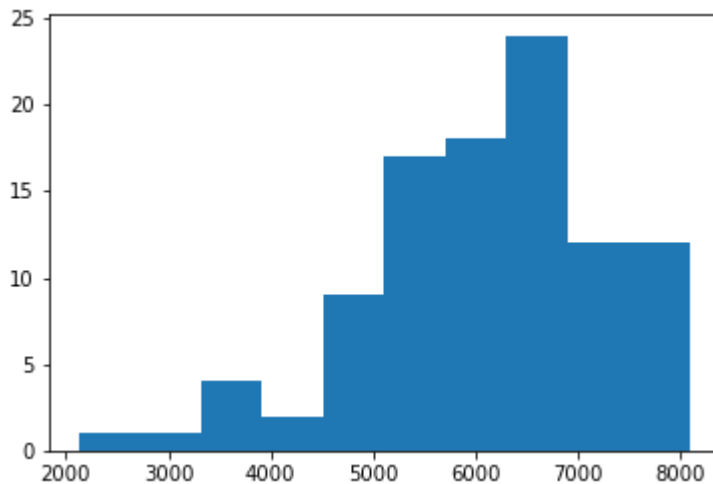
```
In [235]: iters = []
for _ in range(100):
    w, iterations = perceptron_experiment(X, f, y, False, True)
    iters.append(iterations)

print(iters)
```

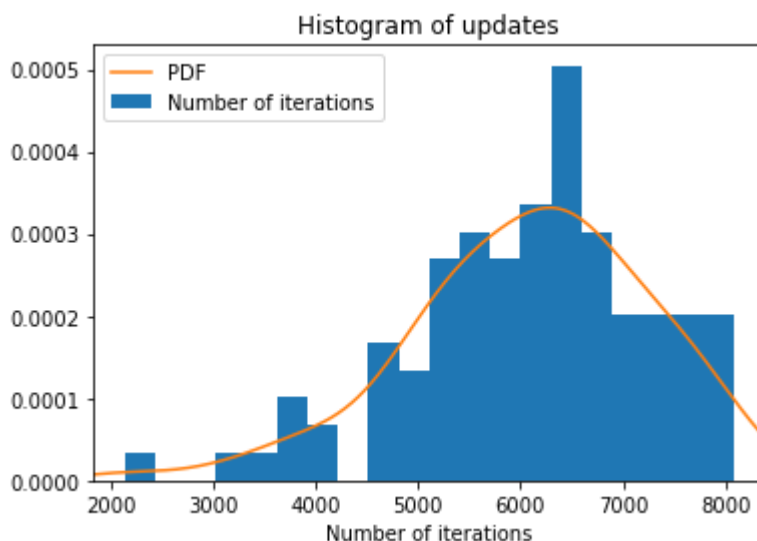
```
[6041, 6858, 6320, 2133, 6503, 5148, 6390, 6922, 5862, 7220, 8080, 575
1, 6082, 6538, 7041, 5506, 5730, 5674, 6075, 6772, 6039, 4667, 5887, 5
457, 7126, 5540, 6140, 6818, 6268, 8056, 6856, 6543, 7347, 5743, 7806,
4596, 6929, 6453, 3888, 6335, 5150, 5137, 6100, 6339, 5086, 4848, 390
0, 5602, 8062, 6232, 6027, 5347, 6333, 4096, 6747, 6553, 7835, 7358, 4
208, 6596, 7501, 5410, 5551, 4732, 4752, 7508, 4964, 3792, 5006, 7285,
5285, 6197, 5354, 5128, 5289, 5728, 7155, 3056, 5551, 7554, 6698, 745
6, 7050, 7999, 6384, 6523, 7643, 7546, 5783, 6476, 6354, 5408, 6797, 5
867, 7777, 4740, 6326, 7309, 6807, 3414]
```

```
In [236]: plt.hist(iters)
```

```
Out[236]: (array([ 1.,  1.,  4.,  2.,  9., 17., 18., 24., 12., 12.]),  
          array([2133. , 2727.7, 3322.4, 3917.1, 4511.8, 5106.5, 5701.2, 6295.  
9,  
          6890.6, 7485.3, 8080. ]),  
          <BarContainer object of 10 artists>)
```



```
In [237]: import scipy.stats as st  
plt.hist(iters, density=True, bins=20, label="Number of iterations")  
mn, mx = plt.xlim()  
plt.xlim(mn, mx)  
kde_xs = np.linspace(mn, mx, 300)  
kde = st.gaussian_kde(iters)  
plt.plot(kde_xs, kde.pdf(kde_xs), label="PDF")  
plt.legend(loc="upper left")  
plt.xlabel("Number of iterations")  
plt.title("Histogram of updates");
```



(h) Summarize your conclusions with respect to accuracy and running time as a function of N and d .

There is a direct proportional relationship between the time to converge with N and d . Higher N and d , more time to converge. As the data is separable, the perceptron algorithm will converge and more N usually means better accuracy.

Problem 1.6

Consider a sample of 10 marbles drawn independently from a bin that holds red and green marbles. The probability of a red marble is μ . For $\mu = 0.05$, $\mu = 0.5$, and $\mu = 0.8$, compute the probability of getting no red marbles ($v = 0$) in the following cases.

(a) We draw only one such sample. Compute the probability that $v = 0$.

$P(v = 0)$ = probability that all the marbles are green = $(1 - \mu)^{10}$:

1. $\mu = 0.05 \rightarrow 0.95^{10} = 0.5987369392$
2. $\mu = 0.5 \rightarrow 0.5^{10} = 0.0009765625$
3. $\mu = 0.8 \rightarrow 0.2^{10} = 0.0000001024$

(b) We draw 1,000 independent samples. Compute the probability that (at least) one of the samples has $v = 0$.

$P(\text{At least one sample has } v = 0) = 1 - P(\text{All samples has } v > 0) = 1 - (P(\text{One sample has } v > 0))^{1000} = 1 - (1 - P(v = 0))^{1000} = 1 - (1 - (1 - \mu)^{10})^{1000}$:

1. $\mu = 0.05 \rightarrow 1 - (1 - (1 - 0.05)^{10})^{1000} = 1$
2. $\mu = 0.5 \rightarrow 1 - (1 - (1 - 0.5)^{10})^{1000} = 0.6235762019$
3. $\mu = 0.8 \rightarrow 1 - (1 - (1 - 0.8)^{10})^{1000} = 0.0001023947625$

(c) Repeat (b) for 1,000,000 independent samples.

1. $\mu = 0.05 \rightarrow 1 - (1 - (1 - 0.05)^{10})^{1000000} = 1$
2. $\mu = 0.5 \rightarrow 1 - (1 - (1 - 0.5)^{10})^{1000000} = 1$
3. $\mu = 0.8 \rightarrow 1 - (1 - (1 - 0.8)^{10})^{1000000} = 0.09733159265$

Problem 1.8

The Hoeffding Inequality is one form of the law of large numbers. One of the simplest forms of that law is the Chebyshev Inequality, which you will prove here.

(a) If t is a non-negative random variable, prove that for any $\alpha > 0$, $\mathbb{P}[t \geq \alpha] \leq \mathbb{E}(t)/\alpha$.

- Consider a random variable, $I(t \geq \alpha)$, equals 1 if $t \geq \alpha$, 0 otherwise.
- If t is a non-negative random variable and $\alpha > 0$, then $\alpha I(t \geq \alpha) \leq t$.
- Apply expectation to both sides: $\mathbb{E}(\alpha I(t \geq \alpha)) \leq \mathbb{E}(t)$.
- We have $\mathbb{E}(\alpha I(t \geq \alpha)) = \alpha \mathbb{P}[t \geq \alpha] + 0 \cdot \mathbb{P}[t < \alpha] = \alpha \mathbb{P}[t \geq \alpha]$.
- Then, $\alpha \mathbb{P}[t \geq \alpha] \leq \mathbb{E}(t)$.
- And finally, $\mathbb{P}[t \geq \alpha] \leq \mathbb{E}(t)/\alpha$.

(b) If u is any random variable with mean μ and variance σ^2 , prove that for any $\alpha > 0$, $\mathbb{P}[(u - \mu)^2 \geq \alpha] \leq \sigma^2/\alpha$. [Hint: Use (a)]

- Consider $(u - \mu)^2 \geq 0$.
- Then, $\alpha \mathbb{P}[(u - \mu)^2 \geq \alpha] \leq \mathbb{E}((u - \mu)^2)$.
- We have $\mathbb{E}((u - \mu)^2) = \sigma^2$.
- Then, $\mathbb{P}[(u - \mu)^2 \geq \alpha] \leq \sigma^2/\alpha$.

(c) If u_1, \dots, u_N are iid random variables, each with mean μ and variance σ^2 , and $u = (1/N) \sum_{n=1}^N u_n$, prove that for any $\alpha > 0$, $\mathbb{P}[(u - \mu)^2 \geq \alpha] \leq \sigma^2/(N\alpha)$.

- We have $\mathbb{E}(u) = (1/N) \sum_{n=1}^N \mu = \mu$.
- We have $\text{Var}(u) = (1/N^2) \sum_{n=1}^N \sigma^2 = \sigma^2/N$.
- Substitute in the formula obtained in (b), then we get $\mathbb{P}[(u - \mu)^2 \geq \alpha] \leq \sigma^2/(N\alpha)$.