CDS: Machine Learning · Assignments Week 3

Nils Kimman — s1007368
Wietze Koops — s1040685
Cárolos Laméris — s1040683

1 Week 3

Extra Exercise on Perceptrons 1.

In this exercise we study the perceptron capacity

$$C(P,N) = 2\sum_{i=0}^{N-1} \binom{P-1}{i}.$$

(a): Show that all problems with $P \leq N$ are linearly separable.

Solution: Using the given formula for the perceptron capacity we find

$$\begin{aligned} C(P,N) &= 2\sum_{i=0}^{N-1} \binom{P-1}{i} = 2\sum_{i=0}^{P-1} \binom{P-1}{i} \\ &= 2\sum_{i=0}^{P-1} \binom{P-1}{i} \cdot 1^i = 2 \cdot 2^{P-1} = 2^P \end{aligned}$$

In the first line we use that $N-1 \geq P-1$ and that $\binom{P-1}{i} = 0$ for $i > P-1$. In the second line we use the Binomial Theorem $(1+\alpha)^n = \sum_{i=0}^{n} \binom{n}{i} \alpha^i$ with $n = P-1$ and $\alpha = 1$.
Since there are 2^P linearly separable problems with P patterns, it follows that all problems with $P \leq N$ are linearly separable.

(b): Show that exactly half of the problems with $P = 2N$ are linearly separable.

Solution: Note that

$$\sum_{i=0}^{N-1}\binom{2N-1}{i} = \sum_{i=0}^{N-1}\binom{2N-1}{(2N-1)-i} = \sum_{j=N}^{2N-1}\binom{2N-1}{j} = \sum_{i=N}^{2N-1}\binom{2N-1}{i},$$

where the first step holds since $\binom{k}{i} = \frac{k!}{i!(k-i)!} = \frac{k!}{(k-i)!(k-(k-i))!} = \binom{k}{k-i}$ for any positive integers k, i (this can also be seen combinatorially: the number of ways to choose i items out of k items is the same as the number of ways to *not* choose $k - i$ items). Hence,

$$C(2N, N) = 2\sum_{i=0}^{N-1}\binom{2N-1}{i} = \sum_{i=0}^{N-1}\binom{2N-1}{i} + \sum_{i=N}^{2N-1}\binom{2N-1}{i} = \sum_{i=0}^{2N-1}\binom{2N-1}{i} = 2^{2N-1}$$

by the Binomial Theorem with $n = 2N - 1$ and $\alpha = 1$.

Since there are 2^{2N} separable problems with $2N$ patterns, it follows that exactly half of the problems with $P = 2N$ are linearly separable. $\square$

Perceptron question 2

In [1]:
```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

(a): Implement the perceptron learning rule as well as a function generating suitable input for the learning rule.

Solution.

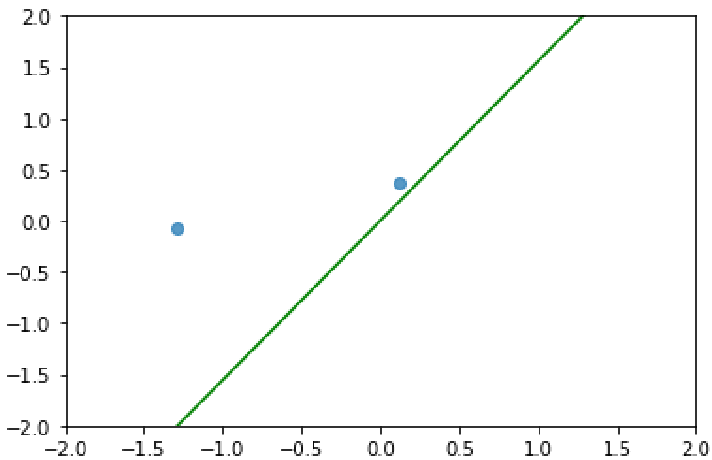
In [2]:
```
# input:
#   range, a real number
#   N, P natural numbers
# output:
#   P vectors of size N, with random entries ranging between [-range, range)
#   a random vector of size P of 1's and -1's, denoting the class of the
#   previous set of random vectors

def giveRandom(range, N, P):
    vecs = (range * 2 * (np.random.rand(P, N) - 1 / 2))
    class_vec = 2 * np.random.binomial(1, 1 / 2, P) - 1
    return vecs, class_vec
```

In [3]:
```
  # perceptron learningrule:
def PLR(vecs, classes):
    xs = np.multiply(vecs.T, classes).T
      # range for choosing initital position w:
    w = np.zeros(xs.shape[1])
      # learning rate:
    eta = 0.01
      # terminate if no solution is found within 1000 updates
    for idx in range(1000):
        signs = np.sign(np.dot(xs, w))
        update_idxs = np.equal(signs, -1) + np.equal(signs, 0)
        if np.all(update_idxs == 0):
            return w, True, idx
        else:
            w = w + eta * np.sum(np.multiply(xs.T, update_idxs).T, axis = 0)
    return w, False, idx
```

Following is not part of the exercise, but just for testing the implementation:

In [4]:
```
  # testing 2D case:
bound = 2
a, b = giveRandom(bound,2,2)
sol, succes, steps = PLR(a, b)

C1 = a[np.equal(b, 1)]
C2 = a[np.equal(b, -1)]

  # plotting:
if C1.size != 0:
    if C1.shape[0] == 1:
        plt.scatter(C1[0][0], C1[0][1], label =  "1's", alpha = .75)
    else:
        plt.scatter(C1[:,0],C1[:,1], label =  "1's", alpha = .75)
```

```
if C2.size != 0:
    if C2.shape[0] == 1:
        plt.scatter(C2[0][0], C2[0][1], label =  "-1's", alpha = .75)
    else:
        plt.scatter(C2[:,0],C2[:,1], label = "-1's", alpha = .75)
    x = np.linspace(-2,2,100)
if succes:
    plt.plot(x, -sol[0]/sol[1]*x, '-g')
else:
    plt.plot(x, -sol[0]/sol[1]*x, '-r')
plt.axis([-bound, bound, -bound, bound])
plt.margins(0.2)
plt.show()
```



(b): Verify that if $P < 2N$ the rule converges almost always, and for $P > 2N$ converges almost never.

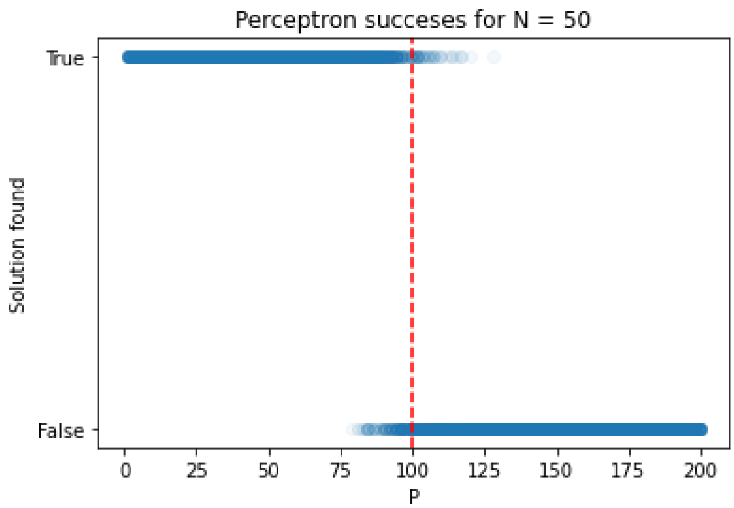
Solution. We'll make a plot for 20 repetitions per P if the algorithm found a solution or not, for P ranging from 1 to 200.

In [5]:

```
N = 50
rep = 20
bound = 2
list_of_succes = []
list_of_P = sum([[P for idx in range(rep)] for P in range(1, 4 * N + 1)], [])
for P in range(1, 4 * N + 1):
    for idx in range(rep):
        vecs, classes = giveRandom(bound, N, P)
        fin, succes, steps = PLR(vecs, classes)
        list_of_succes.append(succes)
```

In [6]:

```
  # plotting:
plt.scatter(list_of_P, list_of_succes, alpha = .05)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Perceptron succeses for N = 50')
plt.xlabel("P")
plt.ylabel("Solution found")
plt.yticks([0,1], ["False", "True"])
plt.show()
  # red line is the given border
```



The red line denotes the border $2N = 100$, where the algorithm should start to fail.

(c): For $P \leq 2N = 200$ plot the

(i) := mean succes rate.

(ii) := mean error rate, standard deviation in the error rate.

(iii) := average number of steps taken.

Solution.

In [7]:

```
  # calculating error rate:
def calc_err(sol, vecs, classes):
    vecs = np.multiply(vecs.T, classes).T
    signs = np.sign(np.dot(vecs, sol))
    update_idxs = np.equal(signs, -1) + np.equal(signs, 0)
    return np.sum(update_idxs) / vecs.shape[1]
```

In [8]:

```
# test from 2d case:
calc_err(sol, a, b)
```

Out[8]:

```
0.0
```

In [9]:

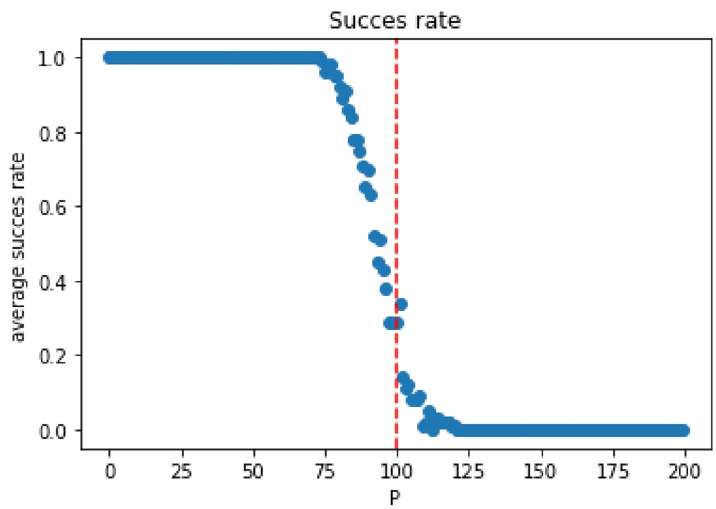
```
N = 50
rep = 100
bound = 2
# (i):
list_of_succes = []

# (ii):
list_of_mean    = []
list_of_std     = []

# (iii):
list_of_mean_steps  = []
list_of_std_steps  = []


list_of_P = np.arange(0, 4 * N, 1)

for P in range(1, 4 * N + 1):
    # (i):
```

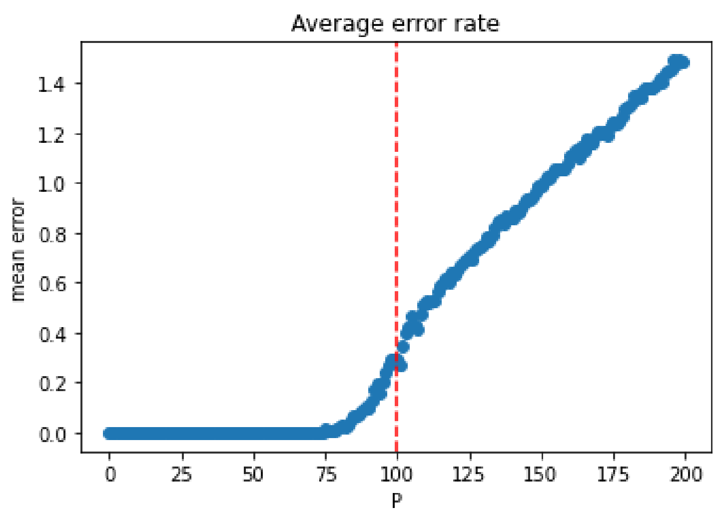
```python
    temp_arr_suc = np.array([])

    # (ii):
    temp_arr_err = np.array([])

    # (iii):
    temp_arr_step = np.array([])

    for idx in range(rep):
        vecs, classes = giveRandom(bound, N, P)
        fin, succes, steps = PLR(vecs, classes)
        # (i):
        temp_arr_suc = np.append(temp_arr_suc, succes)

        # (ii):
        if succes:
            temp_arr_err = np.append(temp_arr_err, 0)
        else :
            temp_arr_err = np.append(temp_arr_err, calc_err(fin, vecs, classes))

        # (iii):
        temp_arr_step = np.append(temp_arr_step, steps)

    # (i):
    list_of_succes.append(np.mean(temp_arr_suc))

    # (ii):
    list_of_mean.append(np.mean(temp_arr_err))
    list_of_std.append(np.std(temp_arr_err))

    # (iii):
    list_of_mean_steps.append(np.mean(temp_arr_step))
    list_of_std_steps.append(np.std(temp_arr_step))
```

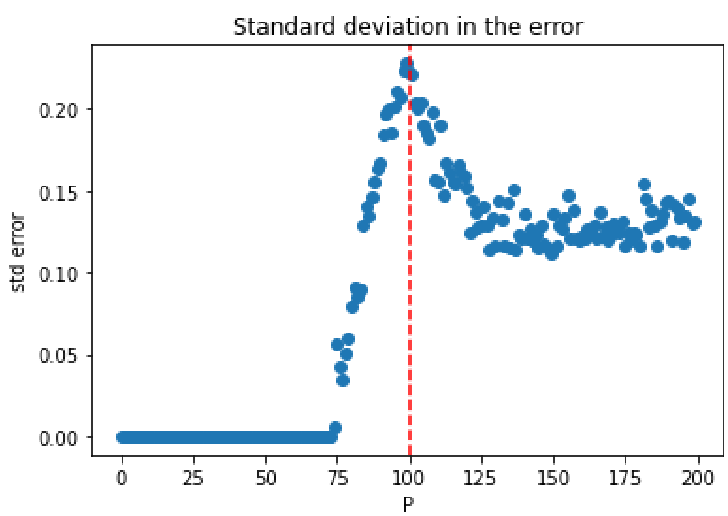
In [10]:

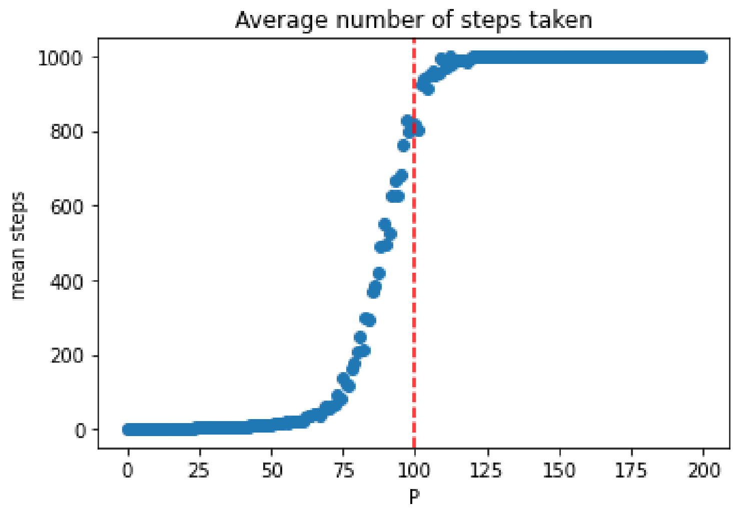
```python
  # plotting succes rates:
plt.scatter(list_of_P, list_of_succes)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Succes rate')
plt.xlabel("P")
plt.ylabel("average succes rate")
plt.show()

  # plotting mean error rate:
plt.scatter(list_of_P, list_of_mean)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Average error rate')
plt.xlabel("P")
plt.ylabel("mean error")
plt.show()

  # plotting std of error rate:
plt.scatter(list_of_P, list_of_std)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Standard deviation in the error')
plt.xlabel("P")
plt.ylabel("std error")
plt.show()

  # plotting average number of steps:
plt.scatter(list_of_P, list_of_mean_steps)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Average number of steps taken')
plt.xlabel("P")
plt.ylabel("mean steps")
```

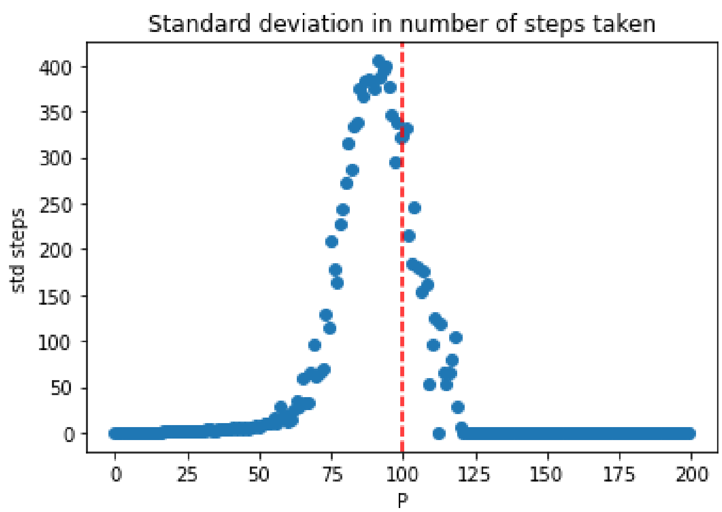
```
plt.show()

  # plotting standard deviation in number of steps:
plt.scatter(list_of_P, list_of_std_steps)
plt.axvline(x = 2 * N, color = 'r', linestyle='--')
plt.title('Standard deviation in number of steps taken')
plt.xlabel("P")
plt.ylabel("std steps")
plt.show()
```











Again, the red line denotes the border $2N = 100$, where the algorithm should start to fail.

Perceptron question 3

Plot $C(N, P)$ and the bound for $N = 50$ and $P \leq 200$.

Solution.

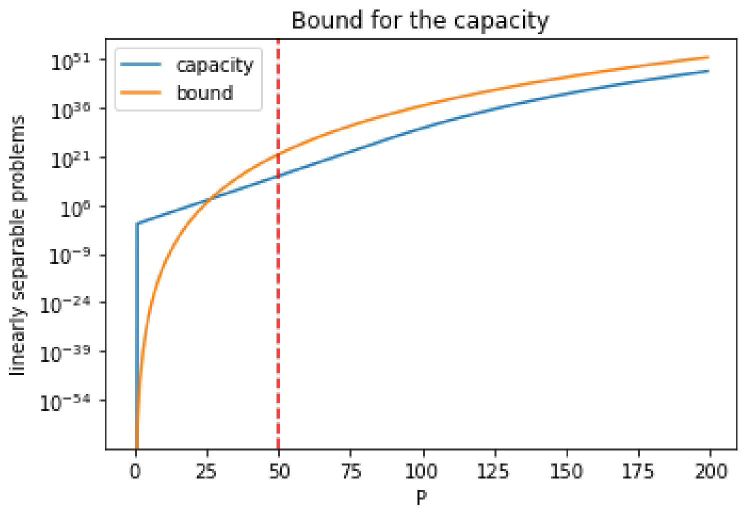
In [11]:

```
def p_cap(N, P):
    def bin_coef(a, b):
        if 0 <= b and b <= a:
            return np.math.factorial(a) / (np.math.factorial(b) * \
                                           np.math.factorial(a - b))
        return 0
    return 2 * sum([bin_coef(P - 1, idx) for idx in range(N)])

def bound_cap(N, P):
    return (np.exp(1) * P / N) ** N
```

In [12]:

```
N = 50
arr_of_P = np.arange(0, 200, 1)
arr_of_cap = np.array([p_cap(N, P) for P in arr_of_P])
arr_of_bound = np.array([bound_cap(N, P) for P in arr_of_P])
```

In [13]:
```
plt.plot(arr_of_P, arr_of_cap, label = 'capacity')
plt.plot(arr_of_P, arr_of_bound, label = 'bound')
plt.axvline(x = N, color = 'r', linestyle='--')
plt.yscale('log')
plt.title('Bound for the capacity')
plt.xlabel('P')
plt.ylabel('linearly separable problems')
plt.legend()
plt.show()
```



The red line denotes the border $N = 50$, from which point on we know the bound to be correct.

Extra Exercise on Perceptrons 4. Consider $\delta = 4m(2P)\exp\left(-\frac{\epsilon^2 P}{8}\right)$

(a:) Rewrite the formula into an expression of ϵ in terms of N and P taking $\delta = 0.01$

Solution: We plug in $\delta = 0.01$ and solve for ϵ:

$$\begin{aligned} & 0.01 = 4 \cdot m(2P) \cdot \exp\left(-\frac{\epsilon^2 P}{8}\right) \\ \implies \quad & 0.01 \leq 4\left(\frac{e \cdot 2P}{N}\right)^N \exp\left(-\frac{\epsilon^2 P}{8}\right), \end{aligned}$$

where we used that $m(2P) \leq \left(\frac{e \cdot 2P}{N}\right)^N$.

$$\begin{aligned} \implies \quad & \frac{0.01}{4(\frac{e \cdot 2P}{N})^N} \leq \exp\left(-\frac{\epsilon^2 P}{8}\right) \\ \implies \quad & \log\left(\frac{0.01}{4(\frac{e \cdot 2P}{N})^N}\right) \leq -\frac{\epsilon^2 P}{8} \\ \implies \quad & 8\log\left(\frac{0.01}{4(\frac{e \cdot 2P}{N})^N}\right) \leq -\epsilon^2 P \end{aligned}$$

$$\begin{aligned} \implies \epsilon^2 & \leq -\frac{8}{P}\log\left(\frac{0.01}{4(\frac{e \cdot 2P}{N})^N}\right) \\ \implies \epsilon^2 & \leq -\frac{8}{P}\left(\log(0.01) - \log(4) - \log\left(\left(\frac{e \cdot 2P}{N}\right)^N\right)\right) \\ \implies \epsilon^2 & \leq -\frac{8}{P}\left(\log(1/400) - N\log\left(\frac{e \cdot 2P}{N}\right)\right) \\ \implies \epsilon^2 & \leq \frac{8}{P}\left(\log(400) + N(\log(2e) + \log P - \log N)\right) \\ \implies \epsilon^2 & \leq \frac{8}{P}\left(\log(400) + N(\log(2e) + \log P - \log N)\right) \\ \implies \epsilon \ & \leq \sqrt{\frac{8}{P}\left(\log(400) + N(\log(2e) + \log P - \log N)\right)} \end{aligned}$$

Note that the formula is only correct bound for $P \gg N$. Although this equation also has a solution with $P < N$, this is actually not a correct solution since the approximation from Exercise 3 is only correct for $P \gg N$. The following code is used for plotting the function and finding the number of patterns P.

R Code

```r
f <- function(P, N) {
  return(sqrt(8/P * (log(400) + N * (1+log(2) + log(P) - log(N)))))
}

require(ggplot2)
s <- seq(1, 800000, by = 100)
ggplot() + geom_line(aes(x = s, y = f(s, 10), color = "N=10")) +
  geom_line(aes(x = s, y = f(s, 20), color = "N=20")) +
  geom_line(aes(x = s, y = f(s, 30), color = "N=30")) +
  geom_line(aes(x = s, y = f(s, 40), color = "N=40")) +
  geom_line(aes(x = s, y = f(s, 50), color = "N=50")) +
  xlab("P") + ylab("epsilon") + scale_color_discrete("") +
  ggtitle("Generalization bound") +
  theme(text = element_text(size=12)) + ylim(0, 0.5)


uniroot(function(P) {f(P, 10) - 0.1}, c(80000, 800000))
uniroot(function(P) {f(P, 20) - 0.1}, c(80000, 800000))
uniroot(function(P) {f(P, 30) - 0.1}, c(80000, 800000))
uniroot(function(P) {f(P, 40) - 0.1}, c(80000, 800000))
uniroot(function(P) {f(P, 50) - 0.1}, c(80000, 800000))
```

The results are as follows:

N	P
10	91293
20	177324
30	263350
40	349375
50	435399

Table 1: Number of patterns required

The following plot shows the value of ε as function of P:

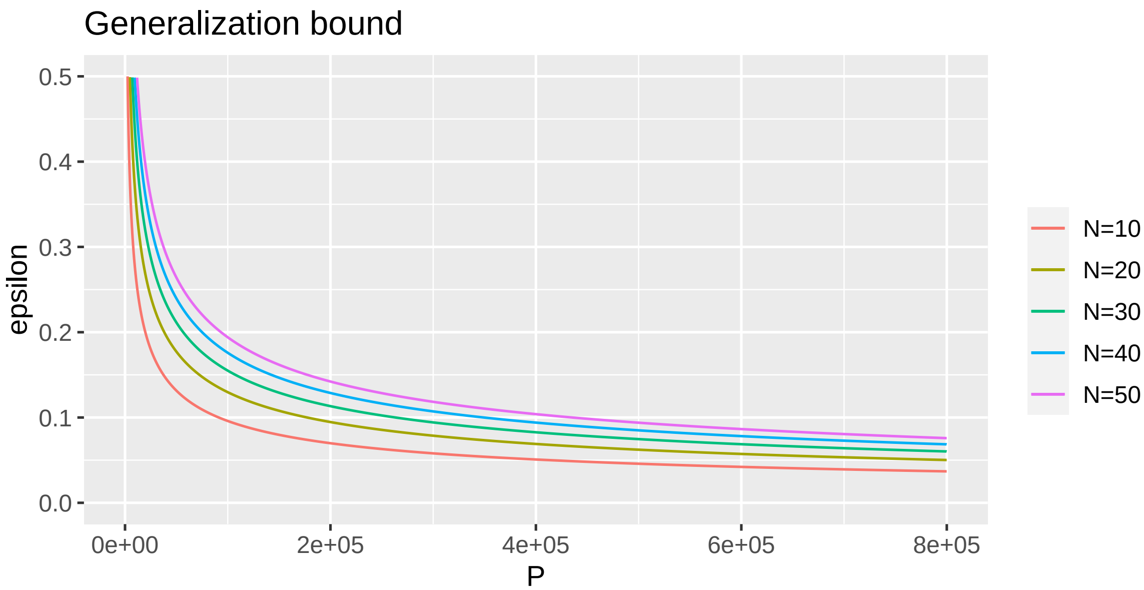


Figure 1: Plot of ε as a function of P.

Perceptron 4(b):

In [1]:

```
import numpy as np
import math
```

In [2]:

```
np.random.seed(37)

def get_data(N, P, w):
    """
    Creates data, initial weights, and corresponding labels

    Parameters:
    N : number of samples
    P : number of patterns

    Returns:
    xi : np matrix size (N, P), binary -1 or 1
    labels : list length P, indicating the truth labels of xi * w0
    """

      # (i). input data xi
    xi = np.random.choice([0, 1], size=(N, P))
    xi[xi == 0] = -1

      # (iii). teacher labels
    labels = []
    for j in range(P):
        labels.append(np.sign(np.dot(xi[:,j], w0)))

    return xi, labels


def train(xi, labels, eta=0.01, n_epoch=100):

    """Trains the perceptron

    Parameters:
    xi : input data, size (N, P)
    labels, size (P, )
    n_epoch : maximum number of iterations to train for
    eta: learning rate

    Returns:
    w : weight parameters after training
    """

    x = np.multiply(xi, labels).T
    w = np.zeros(xi.shape[0])

    for epoch in range(n_epoch):
        signs = np.sign(np.dot(x, w))
        update_idxs = np.equal(signs, -1) + np.equal(signs, 0)
        if np.all(update_idxs == 0):
            return w
        w = w + eta * np.sum(np.multiply(x.T, update_idxs), axis = 1)
    print("Perceptron did not converge")
    return w

def error(N, P):
    """Calculates the error according to the derived formula in (a)"""
    return math.sqrt(8/P * (math.log(400) + N * \
                                (1 + math.log(2) + math.log(P) - math.log(N))))

def numerical_error(x, w, y):
```

```python
    """Calculates the numerical error on the test set"""
    return sum(1 - np.equal(np.sign(np.dot(x.T, w)), y))/len(y)

N = 10
Ps = [10, 50, 100, 500, 1000]

  # hyperparameters
n_learning_runs = 100
eta = 0.01

  # (ii). teacher vector
w0 = np.random.randn(N)

  # (iv). test set
P_test = 10000
test_xi, test_labels = get_data(N, P_test, w0)

print("num. error\t generalization bound")
for P in Ps:
    train_xi, train_labels = get_data(N, P, w0)

      # (v). train weights
    w = train(train_xi, train_labels, eta, n_learning_runs)

      # (vi). print error on test set and generalization bound
    epsilon = error(N, P)
    ner = numerical_error(test_xi, w, test_labels)
    print(ner, "\t\t", "{:.4f}".format(epsilon))
```

```
num. error        generalization bound
0.2719            4.2823
0.108             2.4986
0.0726            1.9173
0.0063            0.9963
0.0026            0.7428
```