Exercise 03: The randomised perceptron learning rule

We consider a more general case of a perceptron in this exercise.

- First of all, we consider a **teacher-student** scenario, which is more general and allows us to better explain the nuts and bolts of the learning process.
- Also, we set an N generic number of (obviously even) elements of the input vector.
- We make learning noisy, by adding a random Gaussian term to the learning rule.

Step 1: Use the perfect perceptron as T

The basic setting is the following: we have a teacher perceptron, which in our case will be the already seen *perfect perceptron*. The perceptron is given by

$$\sigma(\vec{\xi}^{\mu}, \vec{J}) = \text{sign}(\vec{\xi}^{\mu} \cdot \vec{J})$$

and the weights of the perfect perceptron are $J_i^* = 2^{N/2-i}$ in the first half of the weights vector and the same numbers but of the opposite sign in the rest.

Step 2: Initialise a student perceptron with random weights

In the **training** part, the initial weights will be initialised at random for the student perceptron. It will be attempting learning from the teacher perceptron by applying a learning rule, which is in our case

$$\vec{J}(t+1) = \vec{J} + \frac{1}{\sqrt{N}} \sigma_T^{\mu} \vec{\xi}^{\mu} (1 + r)$$

where $r \sim \mathcal{N}(0, \sqrt{50})$.

```
Currently N is 20 and the weights of the perfect perceptron are:
[ 512.  256.  128.   64.   32.   16.    8.    4.    2.    1. -512. -256.
 -128.  -64.  -32.  -16.   -8.   -4.   -2.   -1.]
```

Step 3 and 4: produce sets for different P values and train the student perceptron

Now we generate the training sets. We will do this for a set of different P values: 1, 10, 20, 50, 100, 150, 200. The goal is to understand whether we get an improvement in the learning process whenever we have a bigger training set size.

For each value of P, we do the following:

- set up a random vector of weights
- generate a random 0/1 matrix of size $P \times N$
- get the true values σ_T^μ for each μ-th vector by passing the ξ^μ vector to the perfect perceptron (now this will be the teacher perceptron)
- iterate over variable t, that we max-bounded at 10000 in order to avoid cycles that are too long, looking for the situation in which the training error $\epsilon_t = 0$
- At each time t, iterate over the μ-th data point and check whether the given weights (initalised at random) gives us the same results wrt those given by the teacher perceptron. If not, we *correct* the weights vector with a term given by the learning rule above.

What we would expect is that with bigger training sets the generalisation works better, and students are better able to reproduce the behaviour of the teacher since they are *exposed* more often to what is right and what is wrong.

```
Training for P = 1 converged in time 0
Training for P = 10 converged in time 41
Training for P = 20 converged in time 48
Training for P = 50 converged in time 121
Training for P = 100 converged in time 311
Training for P = 150 converged in time 1129
Training for P = 200 converged in time 554
```

```
In ascending order of P weight vectors are
```

```
[array([-0.08, -0.22, -2.93, -2.8 , -0.7 , -0.46,  0.35,  0.09,  1.28,
         1.13,  1.32, -1.04, -0.21,  0.07, -2.26,  0.91, -0.53, -1.07,
        -1.41,  0.09]),
 array([ 15.68,  -3.96,  11.1 ,  -5.18,  21.22,  -1.61,  13.21,   1.99,
         -5.64, -11.86, -14.87, -10.6 , -10.97,   3.88,   1.3 ,  -6.78,
```

```
         0.17,   2.65,   4.71,  22.77]),
 array([ 2.408e+01,  4.103e+01,  1.027e+01, -2.096e+01, -3.290e+00,
        -3.020e+00,  9.640e+00, -4.190e+00, -1.727e+01, -9.060e+00,
        -1.380e+00, -9.570e+00, -7.710e+00, -1.109e+01,  1.509e+01,
        -3.950e+00, -1.363e+01,  1.093e+01,  1.660e+00,  3.000e-02]),
 array([ 78.7 ,  58.06,  14.15,   2.78,  -2.75, -18.28,  -3.92, -18.21,
         19.32,  -5.02, -78.63, -35.37,   5.87, -12.54,  31.73,  -6.19,
        -20.53, -23.2 ,  10.65, -13.37]),
 array([120.18,  76.54,  34.53,  -7.94,  -2.62, -24.73,   9.09,  -8.6 ,
         -5.81,  11.55, -84.66, -48.55, -67.43, -51.92,   1.19,  32.64,
         35.47, -13.91,  -6.69,  -8.91]),
 array([ 2.9090e+02,  1.6872e+02,  8.0000e+01,  1.8180e+01,  1.8000e-01,
        -1.1140e+01,  2.3830e+01, -4.3020e+01, -2.0530e+01,  1.9420e+01,
        -3.0778e+02, -1.3447e+02, -9.2360e+01, -1.9510e+01,  9.4000e-01,
        -2.2580e+01,  2.5200e+01,  8.4700e+00, -1.0830e+01,  4.7980e+01]),
 array([ 239.3 ,   51.58,   62.33,   -6.69,  -10.87,   21.38,   13.33,
          -8.3 ,   -4.25,   15.94, -222.89, -122.55,  -71.7 ,  -68.51,
           7.23,  -15.38,   16.15,   13.16,   20.77,    9.68])]
```

Step 5: Evaluate single instance test error for 1000 elements

Now that we have trained the student perceptrons, we want to test them against out-of-sample data. We show now the results for a single instance of the $P_{test} = 1000$ case.

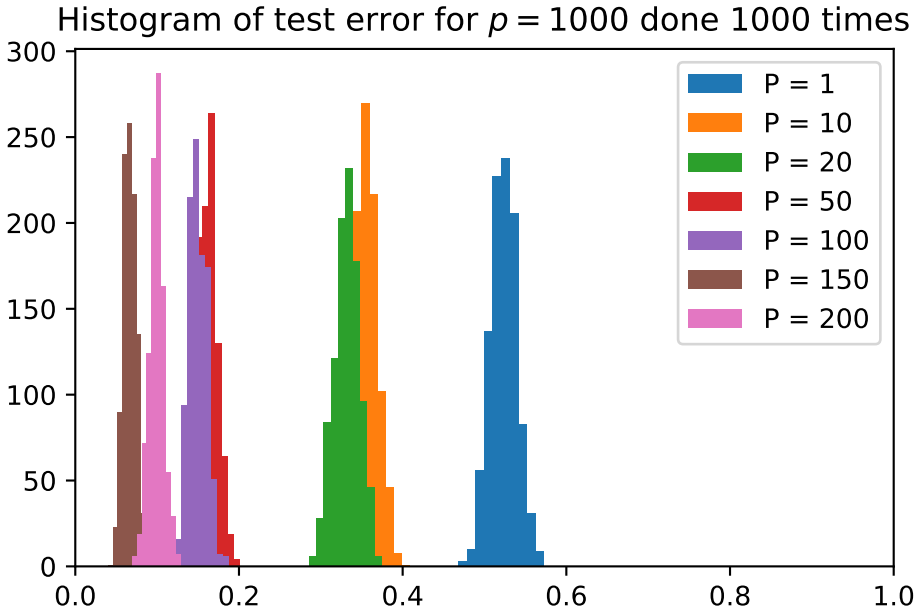
```
On a single instance:
- the test error for P = 1 is 0.51
- the test error for P = 10 is 0.335
- the test error for P = 20 is 0.356
- the test error for P = 50 is 0.164
- the test error for P = 100 is 0.145
- the test error for P = 150 is 0.068
- the test error for P = 200 is 0.106
```

We can clearly see that there is a particular behaviour as P increases, in particular we see that the training error is decreasing. What happens if we do this many times? How will the training error be distributed over, say, a 1000 instantiations of the test?

Step 6: Do this 1000 times and evaluate the average test error

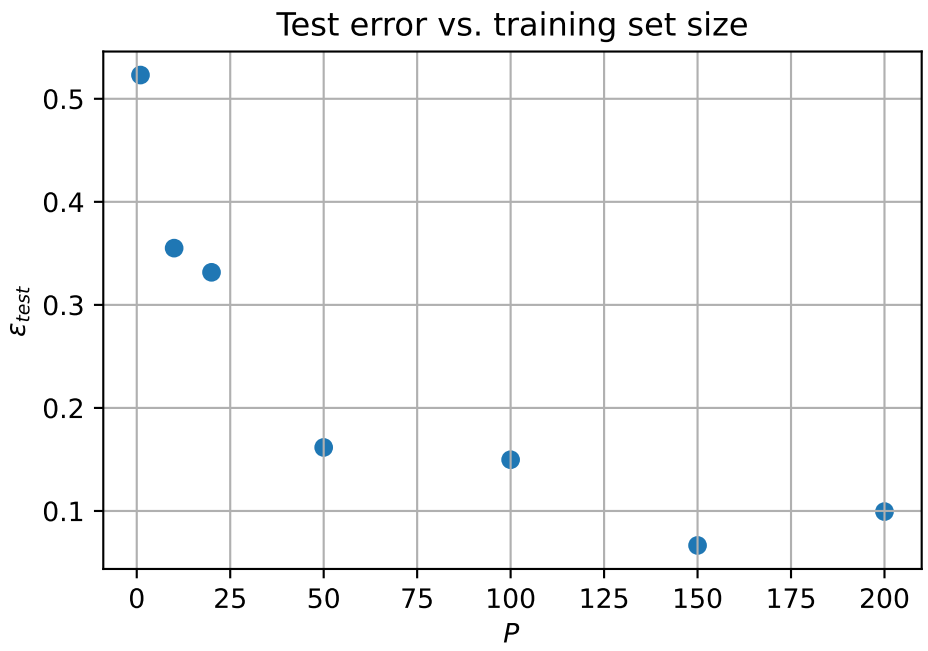
We can plot histograms of the result and see the distributions of test errors over a set of different P values. This result doesn't come as totally unexpected: the bigger the training set,

the lesser the errors we make. We could also see a hint of this by seeing that array elements of higher P weight vectors were more and more similar to those of the perfect perceptron.



Step 7: Plot ϵ vs. P

Now let us clearly state this by plotting the behaviour of the averages of the test error with respect to the progressively increasing size of the training set. As previously stated, it is unexpectedly decreasing.



Step 8: Repeat Steps 3-6 again but with $N = 40$

Also notice that this time P values are 1, 2, 20, 40, 100, 200, 300.

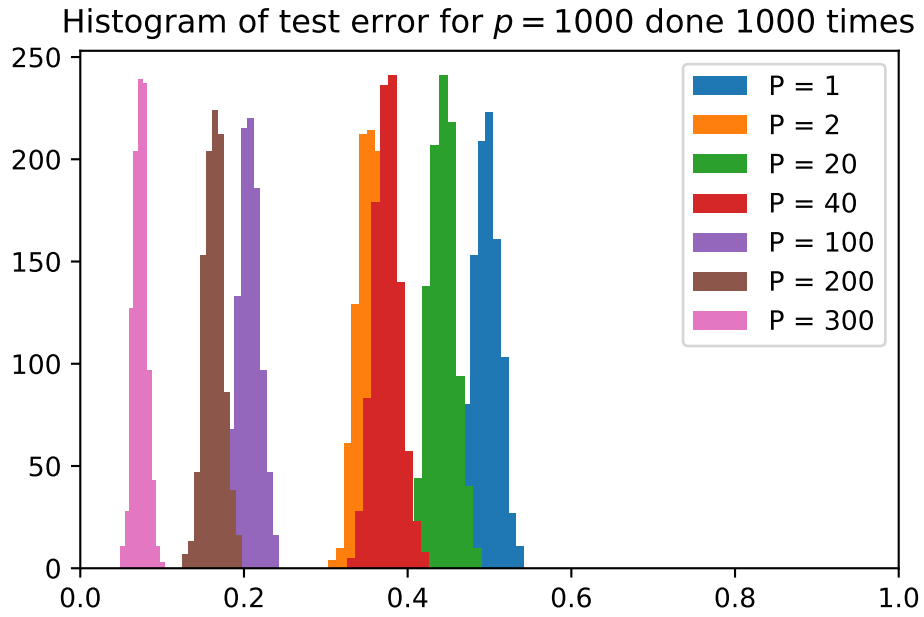
```
Currently N is 40 and the weights of the perfect perceptron are:
[ 5.24288e+05  2.62144e+05  1.31072e+05  6.55360e+04  3.27680e+04
  1.63840e+04  8.19200e+03  4.09600e+03  2.04800e+03  1.02400e+03
  5.12000e+02  2.56000e+02  1.28000e+02  6.40000e+01  3.20000e+01
  1.60000e+01  8.00000e+00  4.00000e+00  2.00000e+00  1.00000e+00
 -5.24288e+05 -2.62144e+05 -1.31072e+05 -6.55360e+04 -3.27680e+04
 -1.63840e+04 -8.19200e+03 -4.09600e+03 -2.04800e+03 -1.02400e+03
 -5.12000e+02 -2.56000e+02 -1.28000e+02 -6.40000e+01 -3.20000e+01
 -1.60000e+01 -8.00000e+00 -4.00000e+00 -2.00000e+00 -1.00000e+00]

Training for P = 1 converged in time 2
Training for P = 2 converged in time 1
Training for P = 20 converged in time 78
Training for P = 40 converged in time 118
Training for P = 100 converged in time 524
Training for P = 200 converged in time 569
Training for P = 300 converged in time 2728
```

In ascending order of P weight vectors are

```
[array([ 0.68, -1.29,  0.55,  1.12, -0.53, -0.47,  0.33,  1.08,  1.33,
        -0.34, -0.45, -0.03,  1.3 , -0.52,  0.48,  0.63, -0.09,  2.02,
        -0.13, -0.8 , -0.27,  0.12,  1.25,  1.47, -2.59,  0.28,  0.05,
         0.31,  0.78, -3.58, -0.97, -0.14, -1.06, -0.52,  0.17,  0.83,
        -0.04,  1.28,  0.57, -1.07]),
 array([ 1.96, -0.32,  0.01,  1.52,  2.6 ,  0.85, -0.85, -1.93,  0.26,
         0.85, -1.16, -0.38, -1.34,  1.1 ,  0.15, -1.89,  2.5 ,  0.19,
        -1.87,  1.44, -4.31, -1.  ,  0.38,  0.72,  1.56, -0.83, -1.68,
         0.3 , -0.43,  3.67,  0.29,  0.51,  0.36,  0.85, -0.14,  1.36,
        -1.11, -0.59, -1.81,  0.19]),
 array([ 24.74, -15.3 ,  19.4 ,  10.68,   1.49,   4.16, -13.99,   9.8 ,
         -0.05,   4.81,   2.15,  -2.63, -23.94,  -3.09,   2.27,  12.27,
         -6.18,   5.25,  17.5 , -15.02,  -7.25,   5.46,   0.45,   1.57,
         12.68, -27.06,   7.  , -17.56, -11.59,  28.68,  -3.38,   4.56,
        -16.99, -27.72,  20.66,  -0.05,  10.79,  30.14,   9.19, -23.52]),
 array([ 41.49,  48.64,   8.5 ,  15.02, -41.58,  15.94,  -1.6 ,  14.69,
         -9.52, -20.04,  -6.66, -21.62,  -0.17,  19.06, -11.23,  -1.11,
         -5.19,  17.15,  12.7 , -14.38, -30.53,   3.21,  41.72,  -0.55,
         11.65,   7.93, -20.33,  35.87, -14.81,   2.49,  14.82,  10.85,
         23.97,  -7.91,  -1.05, -19.51, -33.07, -39.16,  -0.06, -40.58]),
 array([ 149.31,  104.73,    8.6 ,   17.27,   34.02,   -6.8 ,   13.88,
          -0.62,   44.5 ,  -61.67,  -29.09,  -16.02,    2.62,   71.56,
           9.92,  -55.57,   50.84,    7.7 ,  -12.5 ,  -40.92, -156.16,
        -116.26,  -58.48,    1.19,   20.82,    8.27,   -0.49,    2.89,
          37.94,  -57.15,  -50.33,   -3.1 ,  -42.25,  -20.87,   31.31,
          46.1 ,   72.25,  -26.22,    2.78,   15.04]),
 array([ 241.97,  139.97,  110.87,   19.  ,  -41.79,  -16.86,   -1.93,
         -10.92,   23.64,  -29.76,   34.04,  -54.01,  -22.27,   26.44,
          18.28,   -9.35,  -49.61,  -51.76,  -11.87,  -10.86, -203.61,
        -103.79,  -21.  ,    8.42,   -2.88,   14.48,   40.87,  -45.84,
          11.17,   37.35,  -67.08,    8.82,  -13.28,  -18.74,   64.82,
           3.53,   -1.35,   19.26,   52.96,  -60.24]),
 array([ 396.22,  199.4 ,  123.69,   22.46,   13.74,   12.39,  -16.08,
          24.93,  -19.42,    5.73,   17.15,  -35.5 ,   38.61,  -11.02,
         -19.13,  -35.83,  -19.14,    6.96,  -19.45,    6.84, -371.31,
        -226.35, -125.29,  -40.18,   19.31,   20.68,   14.3 ,  -64.99,
           2.69,   42.88,   -9.81,  -11.  ,   18.45,   17.99,    2.65,
         -22.05,    5.76,   27.3 ,   -7.62,  -15.71])]
```

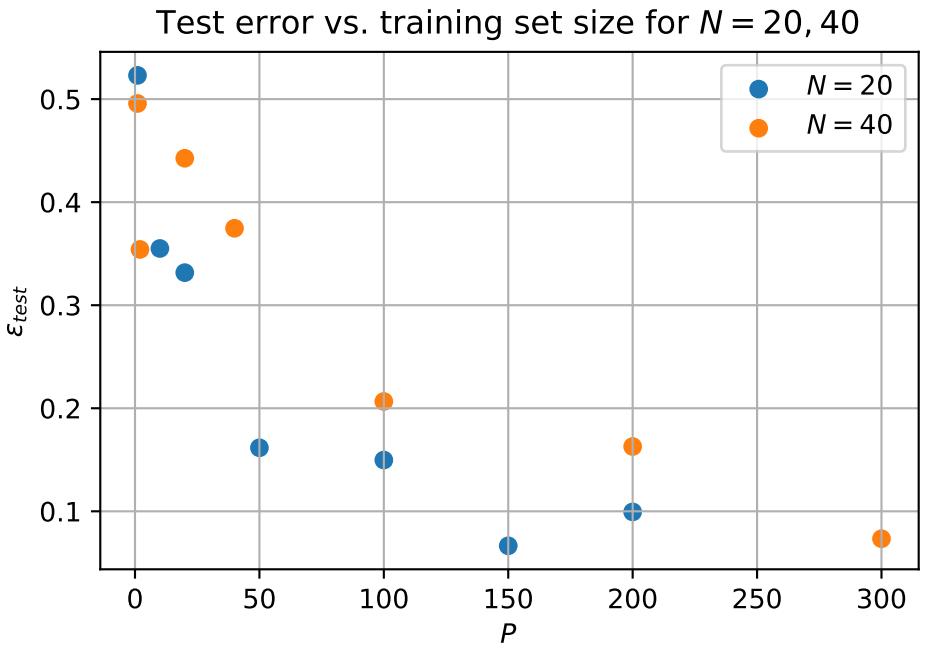
On a single instance:

```
- the test error for P = 1 is 0.473
- the test error for P = 2 is 0.357
- the test error for P = 20 is 0.461
- the test error for P = 40 is 0.384
- the test error for P = 100 is 0.215
- the test error for P = 200 is 0.176
- the test error for P = 300 is 0.078
```



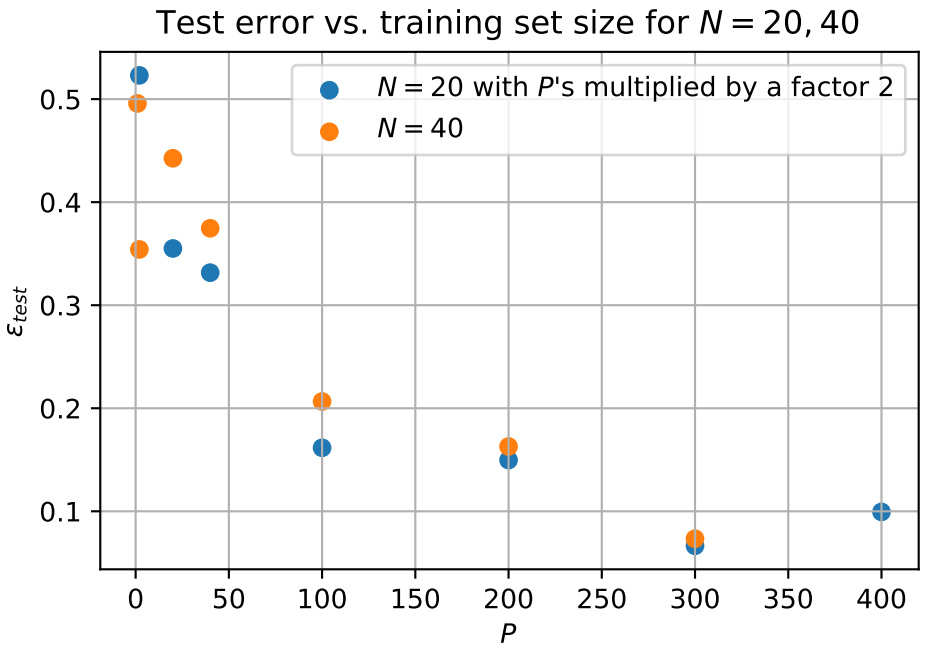
Step 9: Plot ϵ vs. P for $N = 40, 20$

Let us compare the two plots.



Step 10: Find a way to represent the data

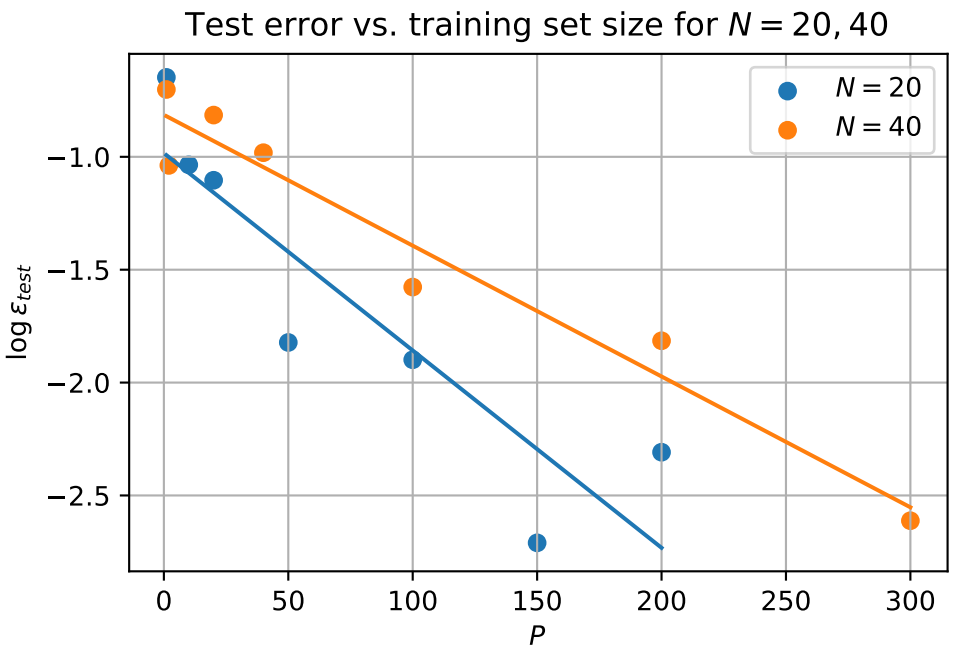
In order to superimpose the two plots we may play with a rescaling of the x-axis, for example:



In this case we used a factor 2. This means that for double the input vector size, the P values required to reach the same amount of test error convergence are the double. This is what we expect from the fact that (see, for example, Engel & Van den Broek p. 12):

$$P = \alpha N$$

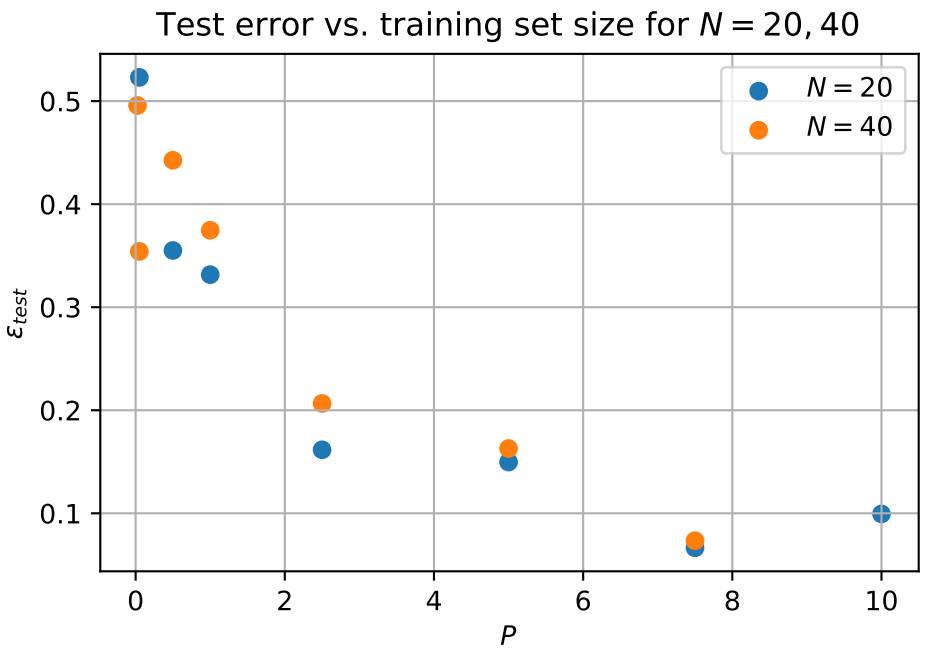
Another way to see this may be the following: as we hinted in the plot below, we may make the hypothesis that the reduction of error with increasing P is an **exponential decay**, and compare the slopes for different sizes N of the input vector. To attain this we might fit the log curves in order to linearise the exponential behaviour and compare the slopes.




```
The slopes are -0.008740877040049493 -0.00579455518905481
The inverse of the slopes (decay time in terms of P) are 114.4049956792823 172.57580044951078
```

As the size N of input vectors increases, the error decay is reached for higher values of the training set size P. In particular for double the size of the input vector, the decay time nearly doubles.

Yet another way (suggested by the teacher) to rescale is to divide the x axis by N, in order to have values that do not depend on N anymore. This yields the same behaviour seen in the first example.


Test error vs. training set size for $N = 20, 40$
$N = 20$
$N = 40$
ε_{test}
P
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10