Machine Learning HW2

Gaumart Siméon . 0845209

Problem 1:

a)

In this problem, we have to do a gradient descent.
So we set on the python notebook all the variables and constants defined in the subject, knowing that

$$\nabla F(w^k) = 2X^T X w^k - 2X^T y$$

So while the error is greater than the tolerance, we continue to iterate the gradient descent.

When the while loop stops, we know that $y_{pred}^{oprimal} = Xw^k$ where w^k is the last one calculated in the loop. Then we can calculate sse.

So with $w^0 = \begin{matrix}1\\1\\1\end{matrix}$ and $\eta = 1\text{e-}10$ we found that
sse = 114055.7613744898 and k = 17267

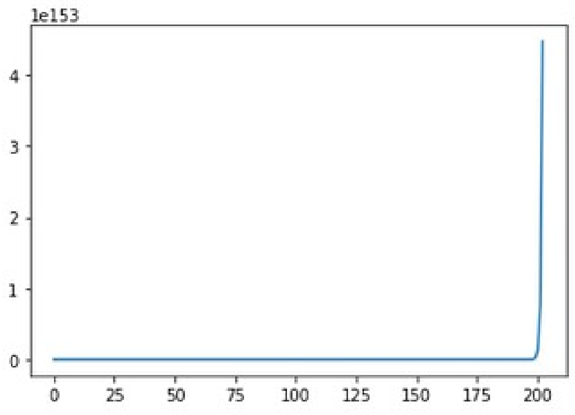
Furthermore the algorithm converges because the while loop stops.

b)

Now, we do the same descent gradient but with $\eta = 1\text{e-}5$

In the python notebook, we quickly reach an error in the loop at k=400 where a value of a variable is to high.

If we print the error depending on k, we found that this error increase exponentially and reaches around ~5e-153 at k=200 and the plot can't display the rest. But it converges to infinity so it diverges.



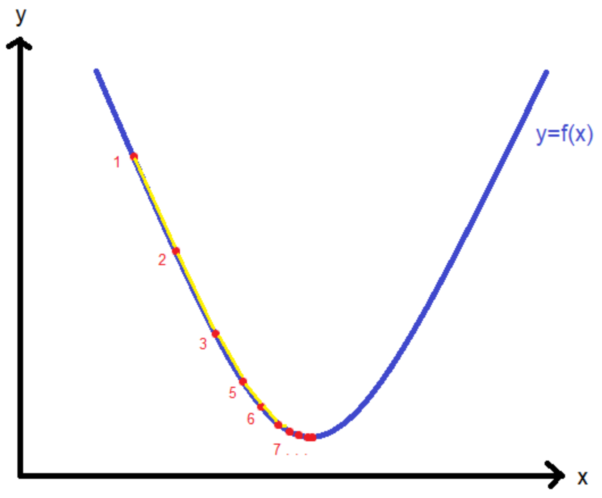
X axis : k
Y axis : error

So the algorithm diverges. So it's useless to calculate sse and k.

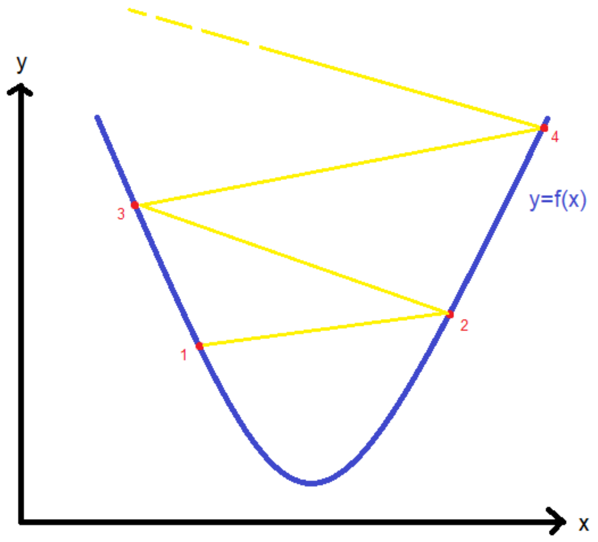
c)

So in the question a) and b) we found that, only with touching η, one algorithm converges and the other diverges.

Indeed, if η is too big, the gradient descent will going in the good direction thanks to the gradient but too far so it will diverge (like in the figure below).



η not too large : the gradient descent converge

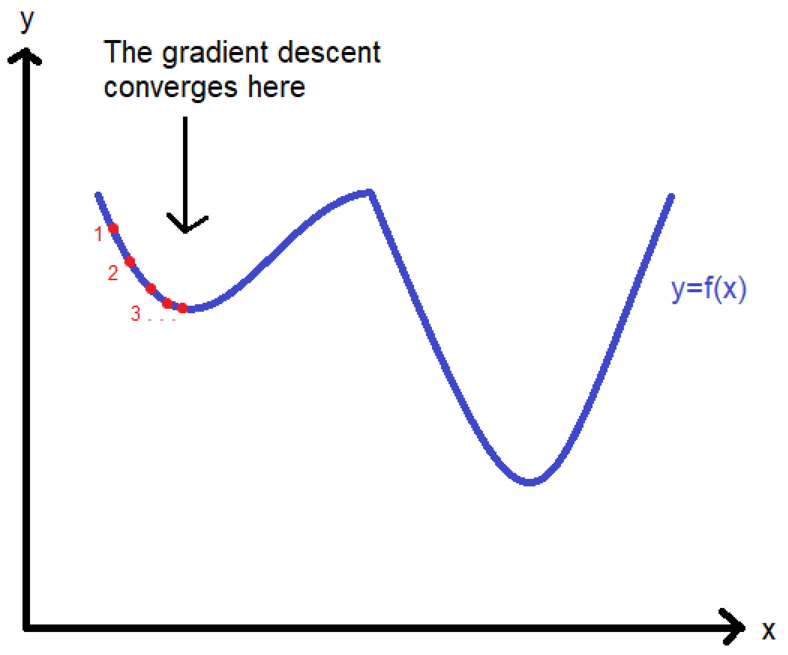


η too large : the gradient descent diverge

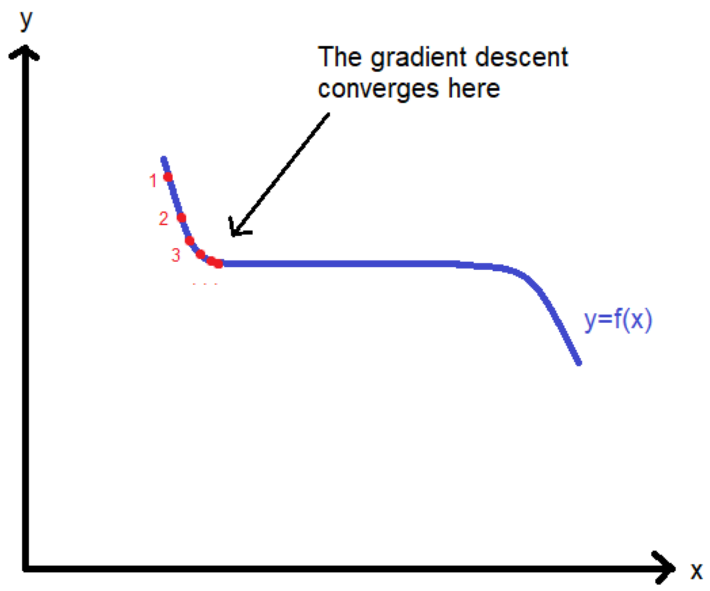
d)

We found that the sse of problem 2 d) in HW1 is sse_hw1 = 36753.355134970734
So sse_hw1 is lower than the sse of question a), so the result is better.

We can explain this because the gradient descent is able to find only a local minimum and not the global minimum.



Furthermore, the gradient descent will converges too with a plateau :



Or when the slope is too low, because we use a tolerance to stop the algorithm, so the error can be greater than 0 (so the gradient can be greater than 0).

To reach the same result found in the problem 2 d) of HW1, we should choose a better starting point w^0 .

Problem 2:

a)

Firstly, we use a package mnist to import the training data, training labels, testing data and testing labels. The training data and testing data are in 3D matrix so we reshape it to have 2D matrix that fits with the sklearn function we will use. So we get a 60000x784 matrix from a 60000x24x24 one for training data, and a 10000x784 matrix from 10000x24x24 for testing data.

To do the training and the test we use the sklearn.neighbors library and we calculate the CPU time for each training and test as complexity of the algorithms.

For 1NN training we got a CPU times of 1 min 28 sec.
For 1NN testing we got a CPU times of 21 min 12 sec and a score of 96,91% (it means 96,91% of the predictions are true).
For 5NN training we got a CPU times of 1 min 28 sec.
For 5NN testing we got a CPU times of 22 min 11 sec and a score of 96,88%.

The results explanations are in the **d)** part.

(because my computer have low performances, I closed and restarted jupyter notebook for each pairs training testing with running only the interesting blocks to clear the RAM which could be quickly saturated, to have relevant CPU times for each pairs).

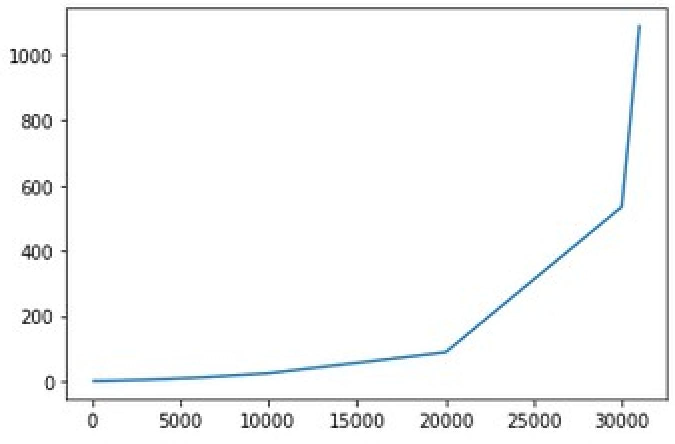
b)

To do the training and the test we use the sklearn.svm library.

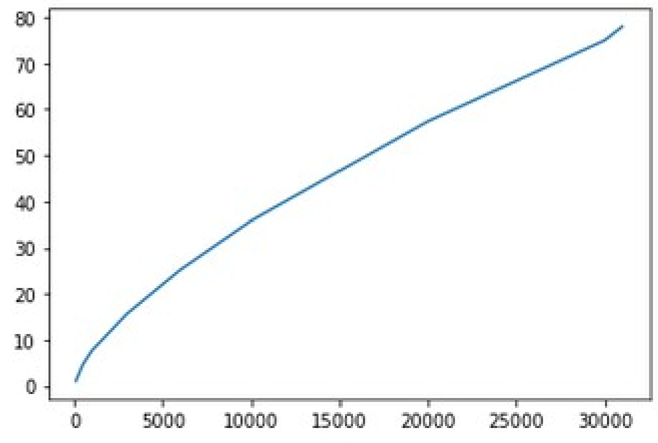
Because my computer have low performances, the training test with 60000 images take more than 12 hours and I was not able to finish it.
So to estimate the CPU times of training, testing and the score, I did the training with different data size (from 32000 training images, my computer can't run it), and with keeping the same data set for testing of 10000 images:

number of training images	training CPU times (s)	testing CPU times (s)	prediction score (%)
100	0,0237	1,06	69,3
500	0,342	4,68	85,14
1000	0,881	7,63	87,58
3000	3,68	15,7	90,13
6000	10,4	25,2	91
10000	23,6	35,8	91,23
20000	88	57,4	91,43
30000	535	75	91,45
31000	1087	78	91,28

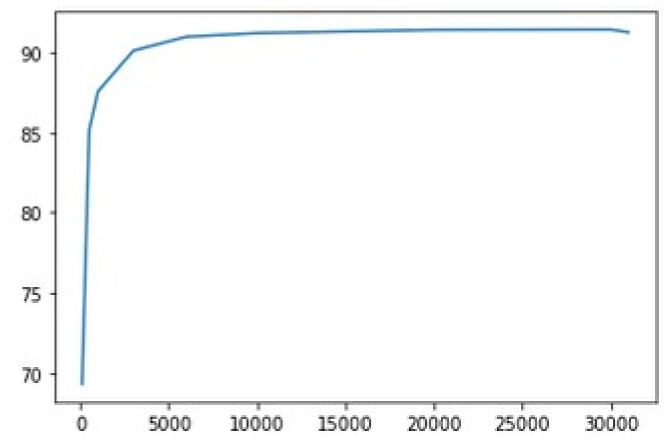
Thanks to this table, we can draw the curve of training CPU times, testing CPU times and prediction score depending on the number of training images:



X axis : number of training images
Y axis : training CPU times (s)



X axis : number of training images
Y axis : testing CPU times (s)



X axis : number of training images
Y axis : prediction score (%)

According to the first curve, it’s hard to predict the value of training CPU times for 60000 training images but we can say that it increases exponentially and it would be much higher than the one for KNN algorithms.

According to the second curve, we can see that the testing CPU times depending on the number of training images is almost linear. So for 60000 training images, the testing CPU times would be around 150 seconds (2min 30sec).

According to the last curve, we can see that the prediction score converges. So we can predict that that the prediction score would be around 91,5% for 60000 training images.

c)

To do the training and the test we use the sklearn.neighbors library and we calculate the CPU time for each training and test as complexity of the algorithms.
(but I don’t understand how it works because the algorithm KernelDensity just fit with one array and the score returns the total log density but do not predict. So I will only talk about the CPU times)

For the training we got a CPU times of 1min 28sec.
For the testing we got a CPU times of 22min 20sec.

d)

For KNN, the CPU times for training is pretty short because the model just needs to take into consideration the data, but there is no processing performed on these data.
However the CPU times for testing is much longer because the algorithm have to find the K nearest neighbors for each data and the algorithm to find the nearest neighbors is more complex (nlogn complexity for the fastest algorithm with n the number of data).
Furthermore testing for 5NN is longer than testing for 1NN because for 5NN we have to find the 5 nearest neighbors instead of one for 1NN.

For SVM, the CPU times is much longer than for KNN for training because the model needs to separate the data with calculating the border which maximize the distance with the closest points, so there is a complex processing performed during the training.
However the CPU times for testing is shorter because the model just needs to see in which side of the border the data is, so there is a process performed but much less complex.

For KDE, the CPU times for training is short because there is no complex processing performed, the model just needs to take into consideration the data. But for testing, the CPU times is much longer because the algorithm needs to calculate the density probability for each data.