

Large-Scale Data Analysis

Homework 1

Lucian Tirca (hrn947@alumni.ku.dk)
Martin Metaksov (pqh518@alumni.ku.dk)
Pantelis Kouris (gzx135@alumni.ku.dk)

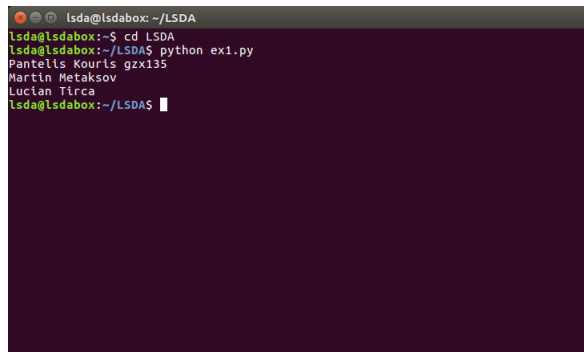
May 1, 2017

1

For this exercise we made use of a virtual machine. We installed the virtual machine as suggested in the notes from the first practical session.

1.2

For this task, we created a python program that outputs our names and student IDs. The screenshot from the virtual machine showing the output of this Python program can be seen below.



```
lsda@lsdabox: ~/LSDA
lsda@lsdabox:~$ cd LSDA
lsda@lsdabox:~/LSDA$ python ex1.py
Pantelis Kouris gzx135
Martin Metaksov
Lucian Tirca
lsda@lsdabox:~/LSDA$
```

Figure 1

Screenshot from virtual machine showing the output of Python program for task 1.2

1.3

For this task, we created a new Jupyter notebook that outputs our names and student IDs. The screenshot from the virtual machine showing the output of the Jupyter notebook can be seen below.

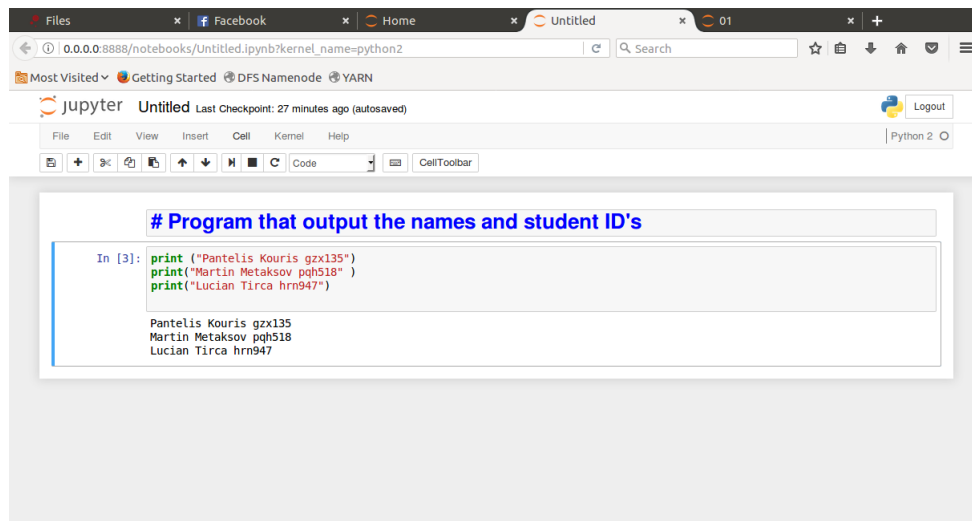


Figure 2
Screenshot from virtual machine showing the output of Jupyter notebook for task 1.3

2

2.1

An example of a large-scale data analysis related domain/application we could think of is self-driving cars. The problem with this application is mainly focused around gathering a huge amount of data through a large number of sensors (cameras, detectors, etc.) and being able to process the data as quickly as possible to make crucial decisions in time. We have been inspired to write about this application by a recent video uploaded by Tesla, where viewers can clearly see the amount of images being processed every second only by 3 of the cars sensors (medium-range camera, left rear camera and right rear camera)¹. While only the cameras can produce a large amount of decision-making information, there is a lot more needed to make the self-driving car a reality (e.g. it is necessary to take more cautious actions when the road is slippery, which requires a sort of road condition sensing).

2.2

For this question, we assume that the data obtained from the different sensors comes in various sizes and is prioritized in many ways when being processed. While there can be many data analysis tasks involved in this problem, one that rises our attention is namely image processing and more precisely object detection. For the first, it is crucial for the application to be able to correctly identify road markings and other participants in the traffic - people, cars, bikes and many more. They all come in different shapes, colors and sizes, which produces an unique

¹Tesla self-driving car video: https://www.tesla.com/en_CA/videos/autopilot-self-driving-hardware-neighborhood-long

situation in every image. Supposing we can combine data from all 3 cameras in a single prediction model, an analysis task involves performing object recognition on each image, fitting the data to a model, and finally making a decision. Of course this needs to happen as quickly as possible.

2.3

From the video, we could observe that each sensor gives approx. 10 images per second, that is 30 images per second from the 3 cameras. Supposing an image takes 1 MB space (optimistically), this is already 30 MB per second. Scaling it up, this makes $30 * 60 = 1800$ MB per minute, 108 GB per hour, 2.6 TB per day, 78 TB per month and 926 TB per year per car. This is huge amount of data to process quickly (as this data needs to be processed in the limits of a few milliseconds max. - in order for the predictions to be at least as good as the ones of a human driving) and obviously enormous amounts to store in a single car. Despite that, we believe that not all of the data needs to be persistently stored, as it becomes irrelevant after a decision has been taken - each situation is different from the other.

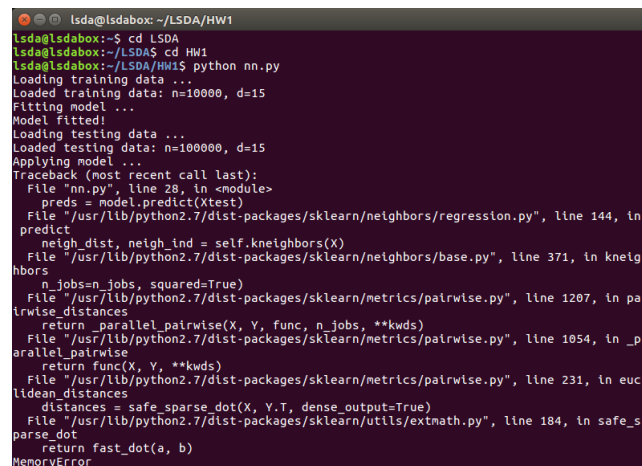
2.4

As mentioned in 2.3, this task indeed produces a computationally challenging problem and a compute time bottleneck, as a large amount of data must be processed very quickly by a mobile computer (given that one cannot fit a huge server machine in a Tesla car). Depending on the needs of the application, disk space can also become a bottleneck, in case the images need to be stored persistently - despite that, there are many smart ways for this to be improved, as images may not need to be stored, or at least fewer images may need to be stored.

3

3.1

When we tried to run nn.py we saw the following output in the terminal.



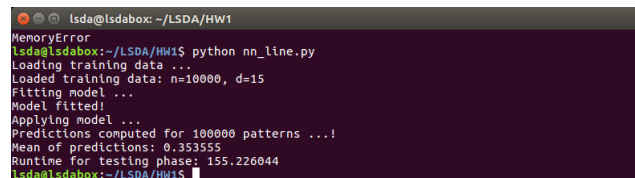
```
lsda@lsdabox: ~/LSDA/HW1
lsda@lsdabox:~$ cd LSDA
lsda@lsdabox:~/LSDA$ cd HW1
lsda@lsdabox:~/LSDA/HW1$ python nn.py
Loading training data ...
Loaded training data: n=10000, d=15
Fitting model ...
Model fitted!
Loading testing data ...
Loaded testing data: n=100000, d=15
Applying model ...
Traceback (most recent call last):
  File "nn.py", line 28, in <module>
    preds = model.predict(Xtest)
  File "/usr/lib/python2.7/dist-packages/sklearn/neighbors/regression.py", line 144, in predict
    neigh_dist, neigh_ind = self.kneighbors(X)
  File "/usr/lib/python2.7/dist-packages/sklearn/neighbors/base.py", line 371, in kneig
hbors
    n_jobs=n_jobs, squared=True)
  File "/usr/lib/python2.7/dist-packages/sklearn/metrics/pairwise.py", line 1207, in pa
irwise_distances
    return _parallel_pairwise(X, Y, func, n_jobs, **kwargs)
  File "/usr/lib/python2.7/dist-packages/sklearn/metrics/pairwise.py", line 1054, in _p
arallel_pairwise
    return func(X, Y, **kwargs)
  File "/usr/lib/python2.7/dist-packages/sklearn/metrics/pairwise.py", line 231, in euc
clidean_distances
    distances = safe_sparse_dot(X, Y.T, dense_output=True)
  File "/usr/lib/python2.7/dist-packages/sklearn/utils/extmath.py", line 184, in safe_s
parse_dot
    return fast_dot(a, b)
MemoryError
```

Figure 3
Screenshot from virtual machine showing the output of nn.py

The problem here is that the test.csv is a large file containing 100.000 lines. When the program tries to calculate the predictions on the test data, runs out of memory since in the virtual machine, we only use 2GB of memory. As we saw in our attempts to solve the next exercise, we are able to load the whole test data, but apparently we run out of memory during the calculations of the predictions.

3.2

For this task we processed the test.csv line by line. For each line we extracted the relevant pattern, applied the model to get the prediction and we appended the prediction to the preds list. Although it was possible to read all the lines of the test.csv at once, we chose to read each line separately, to make use of as less memory as we can because if we had bigger test data, we would not be able to read all lines at once. Below you can see the output of the program in the virtual machine.

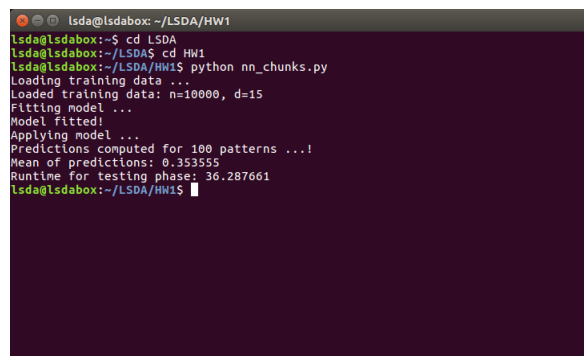


```
lsda@lsdabox: ~/LSDA/HW1
MemoryError
lsda@lsdabox:~/LSDA/HW1$ python nn_line.py
Loading training data ...
Loaded training data: n=10000, d=15
Fitting model ...
Model fitted!
Applying model ...
Predictions computed for 100000 patterns ...!
Mean of predictions: 0.353555
Runtime for testing phase: 155.226044
lsda@lsdabox:~/LSDA/HW1$
```

Figure 4
Screenshot from virtual machine showing the output of *nn_line.py*

3.3

For this task we processed the test.csv in chunks of 1000 lines each time. The mean of predictions in this case is 0.353555(the mean of predictions remains the same in every exercise below).



```
lsda@lsdabox: ~/LSDA/HW1
lsda@lsdabox:~$ cd LSDA
lsda@lsdabox:~/LSDA$ cd HW1
lsda@lsdabox:~/LSDA/HW1$ python nn_chunks.py
Loading training data ...
Loaded training data: n=10000, d=15
Fitting model ...
Model fitted!
Applying model ...
Predictions computed for 100 patterns ...!
Mean of predictions: 0.353555
Runtime for testing phase: 36.287661
lsda@lsdabox:~/LSDA/HW1$
```

Figure 5
Screenshot from virtual machine showing the output of *nn_chunks.py*

The runtime of *nn_chunks* is 36.28 seconds which is significantly smaller than the runtime of *nn_line* which is 155.22 seconds. In *nn_line* we have more disk writes and that's why we can observe this difference in the computational time.

3.4

a)

The size of the test.csv.h5 is 6.4MB whereas the size of the test.csv is 40MB. The HDF5 file is a container of the datasets X and Y, which are stored using a lossless compression filter called LZ4 and that's why the size of test.csv.h5 is much smaller than the test.csv

b)

The first benefit of the HD5F format is that the datasets in a HD5F file are self describing which allows us to extract metadata without needing an additional metadata document. Another benefit is that different types of datasets can be contained in an HD5F file. In addition, as mentioned before HD5F is a compressed file which means that it supports large, heterogeneous and complex datasets. Finally the HDF5 is supported by many programming languages such as R, Python and JAVA.

c)

The maximum percentage of memory usage observed during the execution of *nn_h5.py* using *htop* is 42.4%.

4

$$c^* = (K + \lambda I)^{-1}y = (Q\Lambda Q^T + \lambda Q Q^T)^{-1}y = (Q(\Lambda Q^T + \lambda Q^T))^{-1}y = (Q(\Lambda + \lambda I)Q^T)^{-1}y = (Q^T)^{-1}(\Lambda + \lambda I)^{-1}Q^{-1}y = Q(\Lambda + \lambda I)^{-1}Q^T y$$

We are now looking for a λ using grid search, and we want to see how much we need to "spend" for it. We can assume that for a given kernel K we have already pre-computed the eigenvalue decomposition once, which takes $O(n^3)$ time. This gives us Q and Λ which will be constant along the rest of the operation. Therefore, this time is negligible. An additional speed-up can be done by computing the vector $z = Q^T y$

By explicitating what this equation means in terms of Λ and λ we get:

$$Q \cdot \begin{bmatrix} \frac{1}{\lambda_1 + \lambda} & 0 & \dots & 0 \\ 0 & \frac{1}{\lambda_2 + \lambda} & \dots & 0 \\ 0 & 0 & \dots & \frac{1}{\lambda_n + \lambda} \end{bmatrix} \cdot z \text{ Where } \lambda_1 \dots \lambda_n \text{ are the eigenvalue components of } \Lambda$$

It takes $O(n)$ to compute $t = (\Lambda + \lambda I)^{-1}z$ since we are multiplying with a diagonal matrix. Therefore, the more costly computation is $Q \cdot t$, which takes $O(n^2)$ time and is therefore the main cost of the grid search operation.