Machine Learning A

2023-2024

Home Assignment 1

Yevgeny Seldin and Frederik L. Johansen
Department of Computer Science
University of Copenhagen

The deadline for this assignment is **7 September 2023, 18:00**. You must submit your *individual* solution electronically via the Absalon home page.

A solution consists of:

- A PDF file with detailed answers to the questions, which may include graphs and tables if needed. Do *not* include your full source code in the PDF file, only selected lines if you are asked to do so.
- A .zip file with all your solution source code with comments about the major steps involved in each question (see below). Source code must be submitted in the original file format, not as PDF. The programming language of the course is Python.
- IMPORTANT: Do NOT zip the PDF file, since zipped files cannot be opened in speed grader. Zipped PDF submissions will not be graded.
- Your PDF report should be self-sufficient. I.e., it should be possible to grade it without opening the .zip file. We do not guarantee opening the .zip file when grading.
- Your code should be structured such that there is one main file (or one main file per question) that we can run to reproduce all the results presented in your report. This main file can, if you like, call other files with functions, classes, etc.
- Handwritten solutions will not be accepted, please use the provided latex template to write your report.

1 Make Your Own (10 points)

Imagine that you would like to write a learning algorithm that would predict the final grade of a student in the Machine Learning course based on their profile, for example, their grades in prior courses, their study program, etc. Such an algorithm would have been extremely useful: we could save significant time on grading and predict the final grade when the student just signs up for the course. We expect that the students would also appreciate such service and avoid all the worries about their grades. Anyhow, if you were to make such an algorithm,

1. What profile information would you collect and what would be the sample space $\mathcal{X}$?
2. What would be the label space $\mathcal{Y}$?
3. How would you define the loss function $\ell(y', y)$?
4. Assuming that you want to apply K-Nearest-Neighbors, how would you define the distance measure $d(x, x')$?
5. How would you evaluate the performance of your algorithm? (In terms of the loss function you have defined earlier.)
6. Assuming that you have achieved excellent performance and decided to deploy the algorithm, would you expect any issues coming up? How could you alleviate them?

There is no single right answer to the question. The main purpose is to help you digest the definitions we are working with. Your answer should be short, no more than 2-3 sentences for each bullet point. For example, it is sufficient to mention 2-3 items for the profile information, you should not make a page-long list.

2 Digits Classification with K Nearest Neighbors (40 points)

In this question you will implement and apply the K Nearest Neighbors learning algorithm to classify handwritten digits. You should make your own implementation (rather than use libraries), but it is allowed to use library functions for vector and matrix operations.

Preparation

- Download `MNIST-5-6-Subset.zip` file from Absalon.
  The file contains:
  - `MNIST-5-6-Subset.txt`
  - `MNIST-5-6-Subset-Labels.txt`
  - `MNIST-5-6-Subset-Light-Corruption.txt`
  - `MNIST-5-6-Subset-Moderate-Corruption.txt`
  - `MNIST-5-6-Subset-Heavy-Corruption.txt`
- `MNIST-5-6-Subset.txt` is a space-separated file of real numbers (written as text).[1] It contains a 784×1877 matrix, written column-by-column (the first 784 numbers in the file correspond to the first column; the next 784 numbers are the second column, and so on).
  - Each column in the matrix is a 28×28 grayscale image of a digit, stored column-by-column (the first 28 out of 784 values correspond to the first column of the 28×28 image, the next 28 values correspond to the second column, and so on). Below the question is a Python script that serves as an illustration of one way to load and visualize the data.
- `MNIST-5-6-Subset-Labels.txt` is a space-separated file of 1877 integers. The numbers label the images in `MNIST-5-6-Subset.txt` file: the first number ("5") is the number drawn in the image corresponding to the first column; the second number corresponds to the second column, and so on.
- `Light-Corruption`, `Moderate-Corruption`, and `Heavy-Corruption` are corrupted versions of the digits in `MNIST-5-6-Subset.txt`, the order is preserved. It is a good idea to visualize the corrupted images to get some feeling of the corruption.

Detailed Instructions We pursue several goals in this question:

- Get your hands on implementation of K-NN.
- Explore fluctuations of the validation error as a function of the size of a validation set. (**Task#1**)
- Explore the impact of data corruption on the optimal value of K. (**Task#2**)

IMPORTANT: Please, remember to include axis labels, legends and appropriate titles in your plots!

[1]It is a subset of digits '5' and '6' from the famous MNIST dataset (LeCun et al.).

Task #1 In order to explore fluctuations of the validation error as a function of the size of the validation set, we use the following construction:

- Implement a Python function `knn(training_points, training_labels, test_point, test_label)` that takes as input a $d \times m$ matrix of training points `training_points`, where m is the number of training points and d is the dimension of each point ($d = 784$ in the case of digits), a vector `training_labels` of the corresponding m training labels, a d-dimensional vector `text_point` representing one test point, and its label `test_label` $\in \{-1, 1\}$ (you will need to convert the labels from $\{5, 6\}$ to $\{-1, 1\}$). The function should return a binary vector of length m, where each element represents the error of K-NN for the corresponding value of K for $K \in \{1, \dots, m\}$. **Include a printout of your implementation of the function in the report.** (Only this function, not all of your code, the complete code should be included in the `.zip` file.) Ideally, the function should have no for-loops, check the practical advice at the end of the question.

- Use the first m digits for training the K-NN model. Take $m = 50$.

- Consider five validation sets, where for $i \in \{1, \dots, 5\}$ the set i consists of digits $m + (i \times n) + 1, \dots, m + (i + 1) \times n$, and where n is the size of each of the five validation sets (we will specify n in a moment). The data split is visualized below.

Training data (m points)	Validation set #1 (n points)	Validation set #2 (n points)	Validation set #3 (n points)	Validation set #4 (n points)	Validation set #5 (n points)

- Calculate the validation error for each of the sets as a function of K, for $K \in \{1, \dots, m\}$. Plot the validation error for each of the five validation sets as a function of K in the same figure (you will get five lines in the figure).

- Execute the experiment above with $n \in \{10, 20, 40, 80\}$. You will get four figures for the four values of n, with five lines in each figure. **Include these four figures in your report.**

- Create a figure where for each $n \in \{10, 20, 40, 80\}$ you plot the variance of the validation error over the five validation sets, as a function of K. You will get four lines in this figure, one for each n. **Include this figure in your report.** (Clarification, in case you got confused: fix n and K, then you have five numbers corresponding to validation errors on the five validation sets. You should compute the variance of these five values. Now keep n fixed, take $K \in \{1, \dots, m\}$, and compute the variance as a function of K, i.e., compute it for each K separately. This gives you one line. And then each $n \in \{10, 20, 40, 80\}$ gives you a line, so you get four lines.)

- What can you say about fluctuations of the validation error as a function of n? **Answer in the report.**
- What can you say about the prediction accuracy of K-NN as a function of K? **Answer in the report.**
- A high-level comment: a more common way of visualizing variation of outcomes of experiment repetitions is to plot the mean and error bars, but this form of visualization makes it too easy for humans to ignore the error bars and concentrate just on the mean, see the excellent book of Kahneman (2011). The visualization you are asked to provide in this question makes it hard to ignore the variation.

Task #2 In order to explore the influence of corruptions on the performance of K-NN and on the optimal value of K, we use this construction:

- Take the uncorrupted set, take m as before and $n = 80$, and construct training and validation sets as above. Plot five lines for the five validation sets, as a function of K, for $K \in \{1, \ldots, m\}$. **Include this figure in your report.**
- Repeat the experiment with the `Light-Corruption` set (both training and test images should be taken from the lightly corrupted set), then with the `Moderate-Corruption` set, and then with the `Heavy-Corruption` set. **Include one figure for each of the corrupted sets in your report.**
- Discuss how corruption magnitude influences the prediction accuracy of K-NN and the optimal value of K. **Answer in the report.**

Practical Details and Some Practical Advice

- Use square Euclidean distance to calculate the distance between the images. If $\mathbf{x}_1$ and $\mathbf{x}_2$ are two 784-long vectors representing two images, then the square distance is $\|\mathbf{x}_1 - \mathbf{x}_2\|^2 = (\mathbf{x}_1 - \mathbf{x}_2)^T(\mathbf{x}_1 - \mathbf{x}_2)$. Explanation: sorting points by square Euclidean distance to a given point is equivalent to sorting them by Euclidean distance, but you save computation of the square root.
- If you work with an interpreted programming language, such as Python, do your best to use vector operations and avoid for-loops as much as you can. This will make your code orders of magnitude faster.
- Assume that $\mathbf{X} = \left(\begin{pmatrix} | \\ \mathbf{x}_1 \\ | \end{pmatrix}, \cdots, \begin{pmatrix} | \\ \mathbf{x}_n \\ | \end{pmatrix} \right)$ is a matrix holding data vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and you want to calculate distances between all these points

and a test point $\mathbf{x}$. *Do your best to avoid a for-loop!* One way of doing so is to create another matrix $\mathbf{X}' = \left(\begin{pmatrix} | \\ \mathbf{x} \\ | \end{pmatrix}, \cdots, \begin{pmatrix} | \\ \mathbf{x} \\ | \end{pmatrix} \right)$ and calculate all n distances in one shot using matrix and vector operations.

- Note that for a single data point you can compute the output of K-NN for all K in one shot using vector operations. No need in for-loops!
- You may find the following functions useful:
  - Built-in sorting functions for sorting the distances.
  - Built-in functions for computing a cumulative sum of elements of a vector `v` (for computing the predictions of K-NN for all K at once).
- It may be a good idea to debug your code with a small subset of the data.

Optional, not for submission: You are very welcome to experiment further with the data.

References

Daniel Kahneman. *Thinking, fast and slow.* Farrar, Straus and Giroux, New York, 2011.

Y. LeCun, C. Cortes, and C. J. C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

Appendix: Python Tips

Before you begin, ensure you have the following installed:

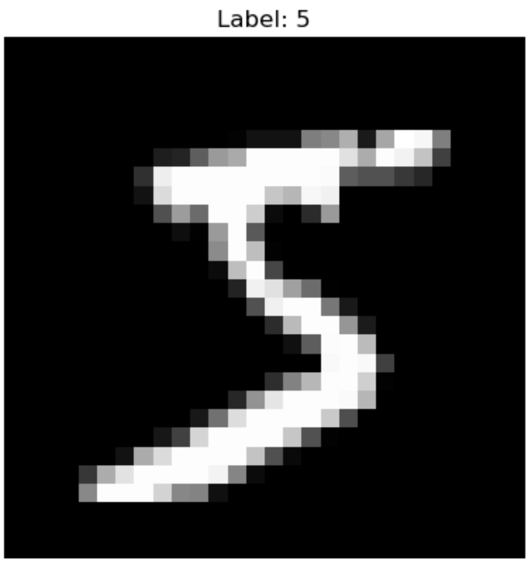
- Python (recommended version 3.x)
- NumPy library
- Matplotlib library

Loading and Visualizing MNIST-5-6-Subset Data

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

```
# Load the data from MNIST-5-6-Subset.txt
# Change the path as needed
data_file_path = "MNIST-5-6-Subset/MNIST-5-6-Subset.txt"
data_matrix = np.loadtxt(data_file_path).reshape(1877, 784)

# Load the labels from MNIST-5-6-Labels.txt
# Change the path as needed
labels_file_path = "MNIST-5-6-Subset/MNIST-5-6-Subset-Labels.txt"
labels = np.loadtxt(labels_file_path)
```

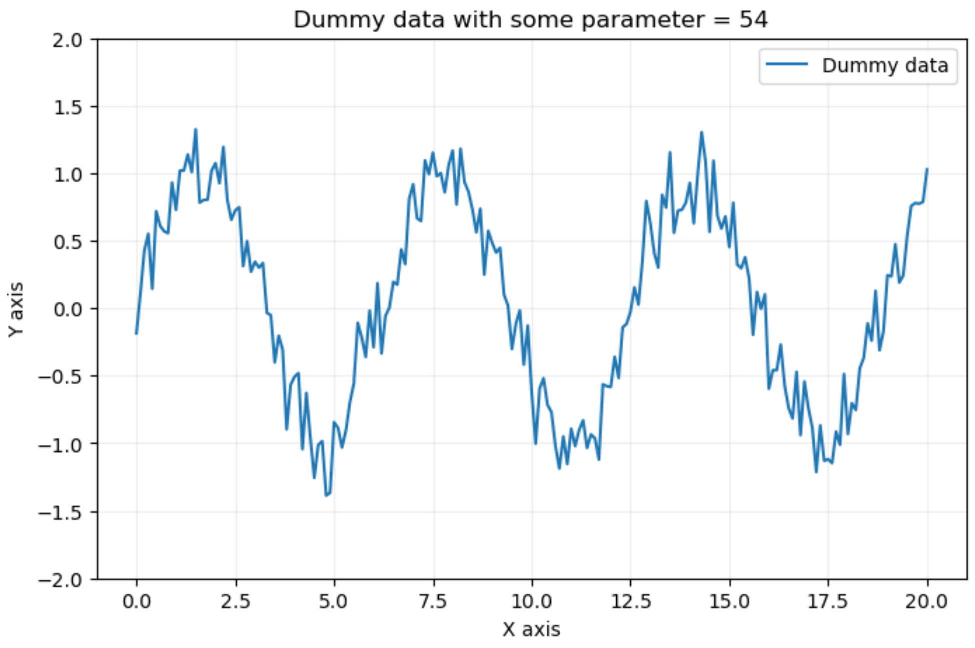
```
# Assuming you want to visualize the first image
# Change the index as needed
image_index = 0
image_data = data_matrix[image_index]
selected_label = int(labels[image_index])

# Visualize the image using Matplotlib
# We transpose the image to make the number look upright.
plt.imshow(image_data.reshape(28,28).transpose(1,0), cmap='gray')
plt.title(f"Label: {selected_label}")
plt.axis('off')  # Turn off axis
plt.show()
```



Setting up a figure with axis labels, legend and title

```
# Dummy data, x and y
x = np.arange(0, 20.1, 0.1)
y = np.sin(x) + np.random.normal(0, 0.2, len(x))
some_parameter = 54

# Initialise figure (fig) and axis (ax)
fig, ax = plt.subplots(figsize=(8,5))

# Plot in axis, add label to data
ax.plot(x, y, label='Dummy data') # (*)

# Set labels and title
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_title(f'Dummy data with some parameter =␣
 ↪{some_parameter}')

# Add grid
ax.grid(alpha=0.2)

# Set axes limits
```

```
ax.set_ylim(-2,2)

# Add legend (remember to label the data as shown above (*))
ax.legend()

# Show plot
plt.show()

# Save plot to some local path
fig.savefig('validation_err.png')
```



Using vector operations with Numpy

```
# Say we have a data matrix with dimension (50, 10)
data_matrix = np.random.rand(50, 10)
print('data_matrix shape:', data_matrix.shape)

# .. and we want to subtract from all of its columns a vector of␣
↪dimension (10)
some_vector = np.random.rand(10)
```

```
print('some_vector shape:', some_vector.shape)

# Instead of looping through the data matrix and subtracting␣
 ↪like so,
result_loop = np.zeros_like(data_matrix)
for i,column in enumerate(data_matrix):
    result_loop[i] = column - some_vector
print('result_loop shape:', result_loop.shape)

# We can use vector operations to greatly improve the speed, at␣
 ↪which we achieve the same result. The essential action␣
 ↪involves expanding the dimensions of "some_vector", aligning␣
 ↪it with the dimensions of the "data_matrix." np.newaxis␣
 ↪accomplishes this by encapsulating the original data with ":
 ↪", while simultaneously creating a new dimension.
some_vector_new = some_vector[np.newaxis, :]
print('some_vector shape after expansion:', some_vector_new.
 ↪shape)

# Now we can subtract some_vector simply like this
result_vector = data_matrix - some_vector_new
print('result_vector shape:', result_vector.shape)

# Assert that the two results are equal
print('result_loop == result_vector:', np.all(result_loop ==␣
 ↪result_vector))

# We can easily check how large of a speedup we achieve
# by using the time package
from time import time
loop_time = []
vector_time = []

for _ in range(250):

    # For loop
    t = time()
    for i,column in enumerate(data_matrix):
        result_loop[i] = column - some_vector
    loop_time.append(t - time())

    # Vector operation
```

```python
    t = time()
    result_vector = data_matrix - some_vector[np.newaxis, :]
    vector_time.append(t - time())

print(f'Speed up: {(np.mean(loop_time) / np.mean(vector_time)):1.
 ↪3f}')
```

```
data_matrix shape: (50, 10)
some_vector shape: (10,)
result_loop shape: (50, 10)
some_vector shape after expansion: (1, 10)
result_vector shape: (50, 10)
result_loop == result_vector: True
Speed up: 21.828
```

Other useful Numpy functions: `cumsum`, `sort` and `argsort`

```python
# Creating an example array
data = np.array([5, 2, 8, 1, 6])

# 1)
# Calculating cumulative sum using cumsum
cumulative_sum = np.cumsum(data)
print("Original data:", data)
print("Cumulative sum:", cumulative_sum)
# Documentation for np.cumsum: https://numpy.org/doc/stable/
 ↪reference/generated/numpy.cumsum.html

# 2)
# Sorting the array using sort
sorted_data = np.sort(data)
print("\nOriginal data:", data)
print("Sorted data:", sorted_data)
# Documentation for np.sort: https://numpy.org/doc/stable/
 ↪reference/generated/numpy.sort.html

# 3)
# Getting indices that would sort the array using argsort
sorted_indices = np.argsort(data)
print("\nOriginal data:", data)
print("Sorted indices:", sorted_indices)
```

```
# Documentation for np.argsort: https://numpy.org/doc/stable/
 ↪reference/generated/numpy.argsort.html

# 4)
# Accessing elements in sorted order using sorted indices
sorted_data_using_indices = data[sorted_indices]
print("\nOriginal data:", data)
print("Sorted data using indices:", sorted_data_using_indices)
```

```
1)
Original data: [5 2 8 1 6]
Cumulative sum: [ 5  7 15 16 22]

2)
Original data: [5 2 8 1 6]
Sorted data: [1 2 5 6 8]

3)
Original data: [5 2 8 1 6]
Sorted indices: [3 1 0 4 2]

4)
Original data: [5 2 8 1 6]
Sorted data using indices: [1 2 5 6 8]
```