NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU, Institutt for Matematiske fag

TMA4320 vår 2023 - Industriell Matematikk-Prosjekt

Prosjektansvarlig: Martin Ludvigsen

February 19, 2023

Dictionary learning for classification problems

Praktisk informasjon

Prosjektperiode: 20. Februar - 6. Mars
Språk: Engelsk eller Norsk
Format: Jupyter Notebook.
Vurderingsansvarlig: Martin Ludvigsen.
Mail: `martilud@ntnu.no`. (Bruk heller Mattelab enn å sende meg mail, med mindre det er personsensitive ting!)

Noen generelle retningslinjer for rapporten:

- Prosjektet kommer med en notebook som dere kan bruke for å starte, og den viser dere spesielt hvordan dere laster inn og håndterer data. Vi kaller denne *handed-out code* i denne prosjektbeskrivelsen.
- Forsøk å ha tydelige overskrifter slik at det er enkelt å se hvor hvilke oppgaver er svart.
- Prøv å svare konkret, og gjør det tydelig hva svaret deres er. Spesielt på oppgaver hvor dere får konkrete spørsmål kan dere for eksempel svare med to linjer under svaret. Et kort og konsist svar er alltid den beste måten å svare på en oppgave.
- Alltid forklar hva neste kodecelle gjør, og forklar resultatene fra en kodecelle etter kodecellen. Unngå å ha "hengende" kodeceller som ikke er forklart. Unngå også å skrive lange tekster, for å så inkludere mange kodeceller. Vev kode og tekst sammen.
- Dere trenger ikke å legge til figurer som ikke er generert av kode i jupyter notebooken deres, men om dere gjør det må dere huske på å legge disse til i den endelige besvarelsen deres.
- Kode burde være for det meste selvforklarende, men det er god kodepraksis å forklare hva en linje/flere linjer kode gjør med kommentarer, samt inkludere såkalte docstrings i funksjoner.
- Sørg for at rapporten er sammenhengende, ryddig og lesbar. Helhetsinntrykk av den leverte .ipynb notebooken teller også på karakteren.
- Prosjektet er vanskelig og tidkrevende, og det er ikke sikkert dere klarer å gjennomføre hele prosjektet. Dere kan fortsatt få god/beste karakter selv om dere ikke har gjennomført hele prosjektet, men sørg for at dere svarer godt på de oppgavene dere får til å svare på. Det er bedre å gjøre halve prosjektet godt enn å gjøre hele prosjektet halvveis.

1 Introduction

Machine learning is a field of artificial intelligence that involves building algorithms and models that can learn from and make predictions on data. In other words, it involves training a computer to perform a task without explicitly programming it to do so. Machine learning has become hugely important in the last decades with the invention of novel algorithms and the access to data and computation resources.

In this project we will be concerned with a subfield of machine learning called dictionary learning, and we will apply this to a classification task on the famous MNIST dataset.

2 Dictionary Learning

Dictionary learning is a technique used to extract a set of (linear) basis functions, or *dictionary*, from a large dataset. These basis functions can be used to represent the data in a more compact and efficient way. The goal of dictionary learning is to find a dictionary that can represent the data with a small number of basis vectors, while still maintaining a high level of accuracy in the representation. Mathematically, a dictionary is an $m \times d$ matrix W containing d basis vectors. A size m vector b can be represented as a linear combination of the basis vectors

$$b \approx Wh, \tag{1}$$

where h is a size d vector that is usually called a *latent variable*. As an example, a vector b can represent an image, and W contains relevant "basis-images" which should contain important features of the image, and h is just the weight of the different basis vectors.

Dictionary learning has close links to matrix factorization. This approach can be seen as a way of decomposing a given matrix into the product of two or more smaller matrices, with the goal of finding a dictionary matrix that can represent the data matrix with a small number of basis vectors. For example, if we store a dataset column-wise in an $m \times n$ matrix A (n datapoints of size m), we can factorize this matrix as

$$A \approx WH, \tag{2}$$

where W is a $m \times d$ matrix and H is a $d \times n$ matrix, where d is usually much smaller than m and n. In this case, each column of H is the corresponding latent variable of a column in A. [1]

An illistration of equation (2) is shown in figure 1

Dictionary learning has many applications for real-world problems that concern large amounts of data. Some of what the matrix A contains, as well as what the learnt dictionary W represents is shown in table 1.

[1] In most machine learning applications, data is usually stored row-wise instead of column-wise. The reason for this is that in most programming languages the first index of an array corresponds to a row, so when storing row-wise we can access the i-th datapoint as `A[i]`. When storing column-wise, we instead access this in NumPy as `A[:,i]`.

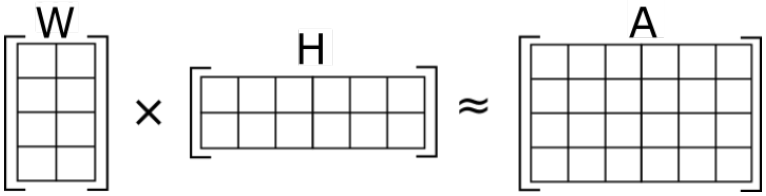

Figure 1: An illustration of dictionary learning as matrix factorization. The dictionary W here consists of $d = 2$ basis vectors, which is used to represent a dataset of $n = 6$ vectors of size $m = 4$. In applications where dictionary learning is used m and n are usually in the thousands and even millions. Source: https://en.wikipedia.org/wiki/Non-negative_matrix_factorization

Application	Entires of A	Columns of A	Rows of A	Columns of W
Movie ratings	Movie rating	Movie rater	Movie	Genres
Cancer screening	Cancer risk	Patient	Time	Risk profiles
Cost analysis	Prices	Seller	Product	Cost trends
Spotify	Time listening	User	Song	Daily mix
NLP	Word usage	User	Word	Type of text
Imaging	Pixel values	Images	Pixels	Image features

Table 1: Examples of applications of dictionary learning.

2.1 Matrix factorization

This subsection will be about some factorization techniques, particularly the SVD, which we will use in this project.

2.1.1 Eigenvalue decomposition

An important tool from linear algebra is *matrix diagonalization*. A square, diagonalizable $n \times n$ matrix A can be written as

$$A = PDP^{-1}, \tag{3}$$

where D is a diagonal matrix $n \times n$ matrix and P is an $n \times n$ invertible matrix. This is also called the eigendecomposition of the matrix A, because it can be calculated by selecting D as the eigenvalues of A and the columns of P containing the eigenvectors of A. In the particular case where the eigenvectors of A form an orthonormal basis, such that $P^T P = I$, matrix diagonalization can be written in the simpler form

$$A = PDP^T. \tag{4}$$

We can also write a diagonalizable matrix as the product of two matrices

$$A = WH, \tag{5}$$

with for example $W = P$ and $H = DP^{-1}$, which links us back to dictionary learning and matrix factorization. The columns of W, which correspond to interesting features

of A, are the eigenvectors of A. The eigenvectors that correspond to large eigenvalues carry more information about the matrix.

While diagonalization is useful, it has some weaknesses. This decomposition only exists if A is square and diagonalizable.

2.1.2 Singular Value Decomposition

There exists a generalization of matrix diagonalization called the *Singular Value Decomposition* (SVD). SVD is incredibly useful and finds its use in most fields imaginable: quantum physics, statistics, numerics and machine learning. It can (amongst other things) be used to learn dictionaries from data.

Any $m \times n$ matrix A has a unique SVD decomposition

$$A = U\Sigma V^T, \tag{6}$$

where U is an orthogonal ($U^TU = UU^T = I$) $m \times m$ matrix, V is an orthogonal $n \times n$ matrix and Σ is a rectangular, diagonal $m \times n$ matrix containing the so-called singular values of A. The singular values are usually ordered from largest to smallest, so that the largest singular value appears in the upper left corner of Σ. An illustration is shown in the upper line of figure 2. Because this is a generalization of eigenvalue decomposition, you can safely think of the singular vectors as eigenvectors and the singular values as eigenvalues.

Similarily to equation (5), this implies that any matrix A can be decomposed into two matrices $A = WH$, with for example $W = U$, $H = \Sigma V^T$.

When we introduced dictionary learning we talked about decomposing a $m \times n$ matrix A into a small amount of basis vectors. The basis vectors stored in W often contain redundant information, and not all columns are equally important for reconstructing A. For a square matrix A, it is possible that the columns are linearly dependent, in which case some of its eigenvalues are 0. These linearly dependant columns are essentially redundant, and do not represent any new (useful) information. The same thing holds for the SVD. This leads us to the notion of reduced SVDs.

2.1.3 Reduced SVD

Reduced SVD is a variant of SVD where we remove (or never compute) the least important parts of the SVD. Denote U_d the $m \times d$ matrix obtained by removing the $m - d$ last columns of U. Similarily, denote V_d^T the $d \times n$ matrix obtained by removing the last $n - d$ columns of V_d (rows of V_d^T) and Σ_d the $d \times d$ matrix obtained by similarly removing rows and columns from Σ. A reduced SVD is then a factorization on the form

$$A \approx U_d\Sigma_d V_d^T. \tag{7}$$

There are two important results here. Firstly, if A has only d non-zero singular values, we have equality $A = U_d\Sigma_d V_d^T$. In other words, we can safely remove redundant information and still achieve a perfect reconstruction. This produces a more compact representation

of A. Note that U_d has orthogonal columns, which means that $U_d^T U_d = I_d$, where I_d is a $d \times d$ identity matrix, but $U_d U_d^T \neq I_m$.

This is not even the final form of reduced SVDs, and we can go even further beyond. It is feasible to remove columns and rows corresponding to non-zero singular values. This means we are discarding "useful" information. The main point is that some components are more important for reconstructing A than others, and the most important components are the ones that correspond to large singular values. This is called *truncated SVD*. An illustration of the SVD, reduced SVD and truncated SVD is shown in figure 2.

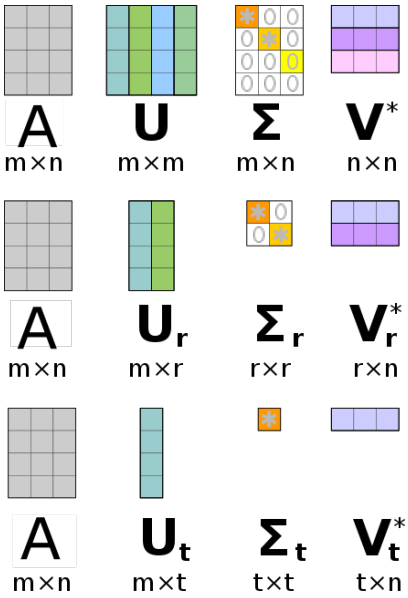

Figure 2: Top: Full SVD of matrix A. Middle: Reduced SVD of A obtained by removing redundant rows and colums. Bottom: Truncated SVD obtained by removing further rows and columns. Note that the reduced SVD perfectly reconstructs A, while the truncated SVD is in this case the best rank 1 approximation to A. Source: https://en.wikipedia.org/wiki/Singular_value_decomposition#Reduced_SVDs.

It is possible to show that the best low-rank approximation to a $m \times n$ matrix A is the truncated SVD. In other words, the solution to

$$\min_{\hat{A} \in \mathbb{R}^{m \times n}} \|A - \hat{A}\|_F, \quad \text{so that } \operatorname{rank}(\hat{A}) = d, \tag{8}$$

is the truncated SVD $\hat{A} = U_d \Sigma_d V_d^T$. $\|.\|_F$ here refers to the Frobenius norm, which is one natural way of talking about distance between matrices. This becomes very important when the dimensions m and n are large, making it hard to extract information from A, and the full SVD costly (and redundant) to compute.

The main takeaway from all of this is that a dictionary that represents data well is the left singular vectors $W = U_d$, and we also get the weights for the data as $H = \Sigma_d V_d^T$.

2.1.4 Calculating SVDs

Throughout this project, we will not really be concerned with computing the SVD, but we will use the numpy implementation `np.linalg.svd`. You should always pass the argument `full_matrices = False` to ensure that no redundant information is calculated. One important thing to note is that calculating the SVD is *expensive*, and scales poorly for large datasets. In some cases, the SVD can take hours or even days to compute, in which case you are most likely better off using other, faster methods. Directly calculating the truncated SVD, that is, only calculating the first d vectors and singular values is the best way to do this. We will always just compute the full SVD and then create the reduced and truncated SVD by removing columns and rows. One of the nice things about calculating the full SVD is that we can select an appropriate d after we have calculated the SVD by simply removing columns and rows, depending on what the application needs.

What we have discussed so far is what is called *training* in machine learning, that is, we use data to learn some structure of the data. The next phase of machine learning is what is called *testing*, where we try to use what we have learnt to make inferences or predictions. The testing problem we are initially concerned with is: When we have found the dictionary W, and we observe a new datapoint b, how can we describe this datapoint as an element of the dictionary W? That is, how do we find h so that

$$b \approx Wh? \tag{9}$$

2.1.5 Orthogonal dictionaries and projections

The most natural way of representing b from the basis-vectors of the dictionary is by selecting the representation that is closest to b, which corresponds to solving the Least Squares projection problem

$$h^* = \arg\min_{h \in \mathbb{R}^d} \|Wh - b\|_2^2. \tag{10}$$

A solution to this problem satisfies the normal equations

$$W^T W h^* = W^T b. \tag{11}$$

If we select the dictionary as the d first left singular vectors $W = U_d$, then W has orthogonal columns and

$$h^* = W^T b. \tag{12}$$

Thus, when we represent b as a linear combination of basis vectors stored in W, we are projecting b onto this linear space, which can be done easily when W is orthogonal as

$$P_W(b) = Wh^* = WW^T b. \tag{13}$$

Note that this is the same as the form you have most likely seen from Matte 3 [2]:

$$\begin{aligned} P_W(b) &= P_{w_1}(b) + \ldots + P_{w_d}(b) \\ &= \langle w_1, b\rangle w_1 + \ldots \langle w_d, b\rangle w_d. \end{aligned}$$

If we want to project every column of a $m \times l$ matrix B onto the basis stored in W we can simply do this as

$$P_W(B) = \begin{bmatrix} P_W(b_1) \ldots P_W(b_l) \end{bmatrix} = WW^T B, \tag{14}$$

which is easy to implement[3].

We denote the distance from a vector b to the space spanned by W as

$$D_W(b) = \|b - P_W(b)\|_2 = \|(I - WW^T)b\|_2 = \min_{h \in \mathbb{R}^d} \|b - Wh\|_2. \tag{15}$$

When we project many vectors onto a space in this way we are interested in measuring the distance between each projected vector and the space. We define this columnwise as

$$D_W(B) = \begin{bmatrix} D_W(b_1) & \ldots & D_W(b_l) \end{bmatrix}. \tag{16}$$

Important implementation detail: Whenever one calculates matrix-vector products with more than 2 matrices/vectors, the order of the computation can vastly influence computation speed and storage space required. The matrix WW^T is an $m \times m$ matrix with rank d where $d \ll m$, so it contains $m - d$ "redundant" vectors that are wasteful to calculate and store. The correct way to calculate $P_W(B)$ is to first calculate $W^T B$, then $W(W^T B)$.

2.2 Exemplar-based Non-negative Matrix Factorization

We will now introduce a different way of learning dictionaries. A common alternative to SVD is what is called *Non-Negative Matrix Factorization* (NMF). The main point of NMF is to decompose a non-negative matrix (a matrix where each entry is larger or equal to 0) $A \approx W_+ H_+$ where W_+ and H_+ are also non-negative. Thus each column of A is a non-negative linear combination of non-negative basis vectors.

We will take an approach called *Exemplar-based NMF* (ENMF). Exemplar-based dictionaries can be made using a very simple concept: Given data stored in the matrix A, we can select the basis-vectors W_+ as columns from A. We will select these vectors at random. An illustration of this is shown in figure 3 The main benefit of this approach is that we do not have to do any training, compared to SVD where we have to calculate the SVD, which can be computationally infeasible for large problems.

[2] Equation 9.12 from https://www.math.ntnu.no/emner/TMA4110/2022h/Forelesningsnotater/9-projeksjon.pdf

[3] Our notation here suggests that we are projecting a $m \times l$ matrix B onto a $m \times d$ matrix W, which is not the case as this is not well-defined. What we are doing is projecting each column of B onto the columns of W, and we are overloading the meaning of P_W

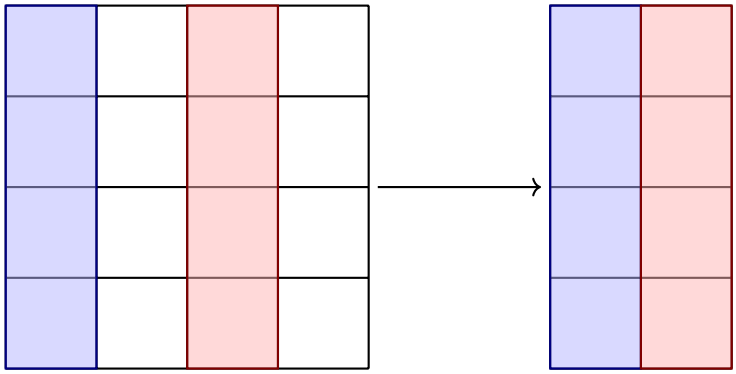

Figure 3: Training of ENMF. We sample $d = 2$ random columns from the data matrix A, in this case column 1 and 3, and use these as basis vectors for the non-negative dictionary W_+.

When training is completed, we can use W_+ as a dictionary that represents data stored in A. One reason we might want to use NMF over SVD is because many signals, like images, are non-negative. It therefore makes sense to also look for non-negative basis vectors.

It is worth noting that non-negative linear algebra deviates slightly from linear algebra. In standard linear algebra we know that d linearly independent vectors span a linear subspace in $\mathbb{R}^m$. This is no longer true for non-negative linear combinations. Non-negative linear combinations span what is called a convex cone, illustrated in figure 4.

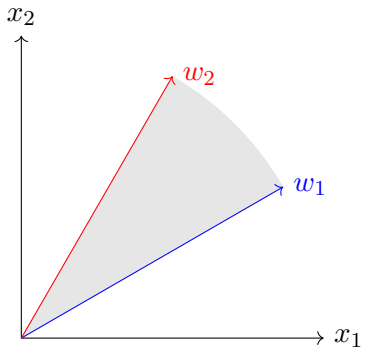

Figure 4: A convex cone spanned by non-negative linear combinations of the two basis vectors w_1 and w_2. There is usually a large amount of the space we cannot reach by non-negative linear combination.

2.2.1 Non-negative projections

After W_+ is trained, we still need to find the non-negative weights H_+ so that $A \approx W_+H_+$ in the case where we want to calculate the matrix factorization. We also need to select H_+ if we observe a new dataset B and want to project this onto W_+, that is, the testing problem. Similarily to what we did for SVD, we will do it by solving a Non-Negative Least Squares projection problem

$$H^* = \arg\min_{H_+ \in \mathbb{R}^{d\times n}} \|A - W_+H_+\|_F^2, \quad \text{so that } H_+ \text{ is non-negative.} \tag{17}$$

This does not have a closed form in comparison like what we had for orthogonal dictionaries, and we therefore need to implement a solution to this problem numerically.

Using the notation we used for the orthogonal dictionaries we can define the non-negative projection of A as

$$P^+_{W_+}(A) = W_+H^*, \tag{18}$$

where H^* is given in equation (17). We use the $+$ superscript here to differentiate non-negative projection from the orthogonal projection in equation (13).

Similarily we have the distances

$$D^+_W(A) = \begin{bmatrix} D^+_W(a_1) & \dots & D^+_W(a_n) \end{bmatrix}. \tag{19}$$

2.2.2 Numerical algorithm for Non-negative projections

One way of finding H^* this is using the relatively simple multiplicative update

$$H_{k+1} \leftarrow H_k \odot (W_+^T A) \oslash (W_+^T W_+ H_k + \delta), \tag{20}$$

where the subscript k denotes the iteration, and the $\delta > 0$ in the denominator of the update is a safe-division factor.

Here $\odot$ denotes elementwise product and $\oslash$ denotes elementwise division, for example

$$A \odot B = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \odot \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1b_1 & a_2b_2 \\ a_3b_3 & a_4b_4 \end{bmatrix}, \quad A \oslash B = \begin{bmatrix} a_1/b_1 & a_2/b_2 \\ a_3/b_3 & a_4/b_4 \end{bmatrix}. \tag{21}$$

This is also called *Hadamard multiplication* and *Hadamard division*. It is important to note that the elementwise product $A \odot B$ is NOT the same as the matrix-matrix product AB. The main difficulty in implementing this is the difference between matrix products, which can be calculated as `np.dot(A,B)` [4], and the Hamamard product, which can be calculated as `A * B` [5]. Similarily the Hadamard division can be calculated as `A / B` [6].

There exists convergence criterion for the update (20), but we will instead just run the algorithm for a certain amount of iterations chosen a priori. It is worth noting that this is an *iterative method*, while the projection for SVD shown in (13) is a *direct method*.

[4] Or `A @ B` or `A.dot(B)` or `np.matmul(A,B)`

[5] Or `np.multiply(A,B)`

[6] Or `np.divide(A,B)`

One potential issue with this algorithm is that we need an initial estimate for H_+, and this initial estimate needs to be non-negative. We will simply use a random vector for initial guess, and for this purpose you can use the `np.random.uniform` function to sample random numbers between 0 and 1.

Important implementation detail: We only need to calculate $W^T A$ and $W^T W$ a single for a given dictionary W and dataset A. In contrary to the SVD approach, $W^T W$ is a $d \times d$ matrix, and it is most likely more efficient to pre-calculate. You should not recalculate these for each iteration.

3 Task 1 (4 points)

Throughout this task we will investigate the test matrices

$$A_1 = \begin{bmatrix} 1000 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

These matrices represent two datasets, and each column is a datapoint. We will also investigate the vectors

$$b_1 = \begin{bmatrix} 2 & 1 & 0 \end{bmatrix}^T, \quad b_2 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T, b_3 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$$

which can be concatenated into the matrix

$$B = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This matrix represents three new datapoints.

- **a)** Calculate the SVD $A_1 = U\Sigma V^T$ using `np.linalg.svd`, with the argument `full_matrices = False`. Print the individual matrices U, Σ and V^T. Confirm that $A_1 = U\Sigma V^T$ by multiplying the three matrices on the right hand side. Which of the basis vectors in the dictionary $W_1 = U$ is the most important for reconstructing A_1?

  Hint: When the SVD is calculated, Σ is stored in a vector, and you should multiply the matrices together without storing Σ in a diagonal 2-D array. Due to floating point errors, you will most likely not get *exactly* equality $A_1 = U\Sigma V^T$, but it should be extremely close.

- **b)** Calculate the SVD $A_2 = U\Sigma V^T$, and print the individual matrices. Does any of the basis vectors in U carry no relevant information about the matrix A_2? What does this say about the columns of A_2? Use this to confirm that we can safely use a reduced SVD to obtain $A_2 = U_d\Sigma_d V_d^T$ for a well-chosen $d < 3$. We then have a dictionary $W_2 = U_d$.

  You will do this several times this project, so write a function that takes in the SVD matrices U, Σ, V^T, and an integer d and returns the dictionary $W = U_d$ and the weights $H = \Sigma_d V_d^T$. Call this function `truncSVD`.

  Hint: Finding U_d given W can easily be done using slicing in NumPy. For example: `U[:,:d]` is an array that contains all rows and the first d columns of `U`.

- **c)** Implement a function that takes in a dictionary with orthogonal columns and a matrix and calculates the projection of the columns of the matrix onto the dictionary as shown in equation (13). Call this function `orthproj`.

Calculate the projection of the test matrix B onto the dictionaries you found in a) and b) and print them. Implement a function that calculates the columnwise distances from B to W as shown in equation (19) and print them.

Hint 1: Make sure your implementation follows the important implementation detail in section 2.1.5

Hint 2: You can calculate the columnwise distance using one call of `np.linalg.norm` by passing a suitable `axis`. You should observe that $D_{W_1}(B) \approx \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$, as both b_1 and b_3 are clearly in the range of A_1, but b_2 is not.

- **d)** We now want to test the ENMF approach. Implement a function that takes in a non-negative dictionary and a matrix and calculates the non-negative projection as shown in equation (17) using the iterative update shown in (20). Call this function `nnproj`. Initialize H_+ using `np.random.uniform`, and make sure the function can also take in `maxiter` and δ. In your experiments, you can use `maxiter = 50` and $\delta = 10^{-10}$.

  Calculate $P^+_{A_1}(B)$ and $P^+_{A_2}(B)$, print the resulting weights and the projections and confirm that your algorithm works. Implement a function, or reuse the one from c), that calculates the (non-negative) distances from B to the dictionaries and print them.

  Hint: Make sure your implementation follows the important implementation detail in section 2.2.1. You can create a random non-negative $d \times n$ initial matrix for example as

  `np.random.uniform(0,1,(d,n))`.

  You should observe that $D_{A_1}(B) \approx \begin{bmatrix} 0 & 1 & 1/\sqrt{2} \end{bmatrix}$, but with a larger numerical error than the last task. This illustrates the difference between projections onto linear subspaces and projections onto cones, as b_3 is in the span A_1, but not as a non-negative span of A_1.

You will reuse more or less all code in this task throughout the project, so make sure it is written well!

4 Images

We now turn to the area of interest in this project, images. In this project we represent a digital image as a discrete vector. An image has width m_x and height m_y, where m_x and m_y are integers. These integers together are usually called the resolution of the image. At each coordinate exists the smallest discernible image element, usually called a pixel. In this project we will only be concerned with grayscale images, which can be represented as either a $m_x \times m_y$ matrix, or it can be represented as an "unrolled" vector of size $m = m_x m_y$. For this project the latter will be used, but in order to plot images using `plt.imshow` we need to reshape them to matrices. Grayscale images usually have integer pixel values between 0 and 255, but we will rescale this to real numebrs between 0 and 1.

An example of a grayscale image with its pixel values is shown in figure 5

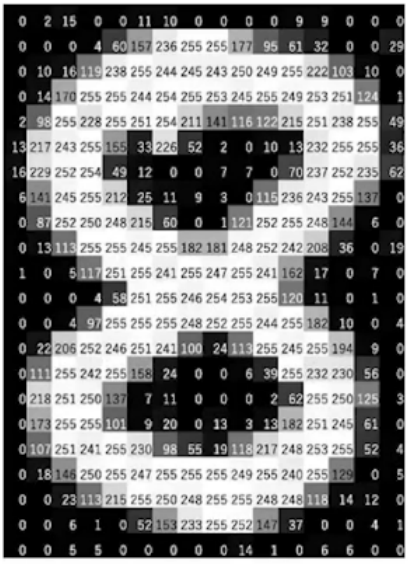

Figure 5: A rectangular grayscale image with its pixel values written in each pixel.

A dataset of n images with m pixels can be represented as a (m, n) Numpy array, where each column represents an image and each row represents a pixel.

4.1 MNIST dataset

The MNIST dataset is a widely used dataset for benchmarking and testing machine learning models. It consists of a total of 70000 handwritten digits from 0 to 9.

Each image is 28 pixels by 28 pixels, for a total of $m = 784$ pixels. The images are centered and have been normalized, meaning that the background is black and the digits are white and have been resized to fit a 20 pixel by 20 pixel box while preserving their aspect ratio.

The MNIST dataset is a popular choice for learning and testing machine learning models because it is relatively small, well-structured, and easy to use, yet still challenging enough to provide a good test of a models capabilities. Some examples of MNIST integers are shown in figure 6.

Figure 6: A selection of example MNIST images.

We will be particularily interested in looking at images of one integer at a time, and try to learn dictionaries that are able to represent the important features of the individual digits. For this reason, we separate the data into a training set which contains $N_{\text{train}} = 5000$ images of each class, and a test set which contains $N_{\text{test}} = 800$ images of each class.

5 Task 2 (3 points)

Throughout the task you will work with real data, which can be computationally demanding when there are large amounts of data. If you want faster results, you can use a smaller amount of data for training and testing. You can for example use $N_{\text{train}} = 1000$ and $N_{\text{test}} = 200$ to achieve more or less the same results, but much faster. For the ENMF approach, use `maxiter` somewhere between 50 and 100.

- **a)** Load the datasets of integers using the handed out code, and plot 16 images of an integer of your choice on a 4×4 grid.

- **b)** Calculate the SVD $A = U\Sigma V^T$, where A is a matrix containing a dataset of the integer you chose in a) columnwise. Here it is important that A should only include data of only one integer. Plot the first $d = 16$ left singular vectors of A, that is, the columns of U_d. Do these basis vectors capture the important features of the dataset? Also plot the singular values from highest to lowest where the y-axis is logarithmic using `plt.semilogy`. Interpret the plot.

  You can try other digits for fun, but only include the plots and discussion of one integer in the report.

  Hint: Singular values are stored from largest to smallest, and a matrix only has non-zero singular values (up to numerical errors) equal to the rank of the matrix. The corresponding singular vectors are also sorted from most important to reconstruct the matrix to least important.

- **c)** Calculate the orthogonal projection $P_W(b)$ where b is one image of the integer you chose in a) (a column of A) using $W = U_d$ with $d = 16, 32, 64, 128$. Plot the results along with the original image. What can you qualitatively say about the images as d increases? Do the same experiment using an image $\tilde{b}$ that is a different integer than what W is trained on. For example, if W is trained on zero digits, $\tilde{b}$ can be a vector containing an image of a one digit. What do you observe?

  Hint: You should calculate the full SVD only once.

- **d)** We now want to investigate quantitatively what happens as d increases. For $d \in [1, 784]$ calculate $\|A - P_W(A)\|_F^2$ for this digit. You should choose a relatively coarse grid for d to avoid unnecessary computation. Plot the results using `plt.semilogy`.

  Do the same but with a dataset containing a different integer than what W was trained on similar to 2c). What do you observe?

  Hint: Use `np.linspace` with a suitable `step` to create a grid. Use a step of for example 20 to test every 20-th d. You should observe that the plot of $\|A - P_W(A)\|_F^2$ against d is almost identical to the plot of the singular values (why?).

- **e)** We now want to try the ENMF approach. Implement code that constructs exemplar-based dictionaries, that is, sample $d = 32$ columns of A randomly and store them in the matrix W_+. Calculate the projections $P^+_{W_+}(A)$ and plot at least

16 of them. You can also plot the result using the SVD approach for comparison. What do you observe?

Hint: Use the function `np.random.choice` with `replace = False` to sample columns. For example `np.random.choice(A.shape[1],d,replace=False)` samples d indexes from 0 to the amount of columns of A without replacement.

- **f)** Repeat **d)** but using the ENMF approach with $d \in [1, 1000]$. You should probably use a coarser grid for d than what you used in **b)** as this approach is much slower. What do you observe? How does this compare to the SVD approach? Do you see any effects of the randomness used in the sampling of the dictionary?

  Hint 1: Use for example `np.logspace(1,3,10, dtype = np.int64)` to generate 10 values of d logarithmically scaled between 1 and $10^3 = 1000$.

  Hint 2: You will most likely see that even for high d, the distances will still be quite large. This is most likely caused by numerical errors of the projection method, as it would need a very large amount of iterations to actually converge.

6 Classification

Classification concerns categorizing observations into classes. Some examples of classification are:

- Determining if an email is spam or not.
- If a patient has a high risk of getting a certain disease or not.
- Classifying if a picture contains a pedestrian or not, which is particularly important for self-driving cars.
- Selecting what movie to reccomend to a certain user on Netflix.
- Given an image containing a hand-written digit, determine what digit it is.

Automatic classification is the process of assigning a given input to one of a predefined set of classes based on statistical and mathematical algorithms. In the past, classification relied heavily on *knowledge-based* methods, where expert knowledge and domain expertise were used to create rules and mathematical models for classification. However, in recent decades, there has been a shift towards *data-driven* methods, where the classification is done using large amounts of data and machine learning algorithms.

This shift towards data-driven methods has been driven by the availability of large amounts of data and the advancement of machine learning techniques. With this availability the belief in data-driven methods has also increased, which again has made serious data collection more common.

One of the key features of automatic classification is that it should happen without human intervention. This means that the classification process is fully automated, with no need for manual input or decision-making by humans. This allows for faster and more efficient classification, and eliminates the potential for human bias or error, but introduces new possibilities for machine bias and error.

Classification is, as most machine learning, done in two phases: First, we learn some mathematical structure from a the training data, in this case we learn dictionaries. Second, we test our method on the test data, which is different from the training data, to test how well our method performs in a realistic problem setting. While the MNIST dataset contains 10 different digits, we will only do classification on a subset of the digits, and we assume that we know which digits are present in the test data.

With both training and testing data we have access to the true labels, that is, we know what digit an image corresponds to. These labels are usually stored in a vector, where the i-th entry corresponds to the label of the i-th data of our test set. When we do testing, the goal is of course to classify without access to the labels, and see how well our predictions correspond to the true labels.

6.1 Score functions

In order to do classification, the only thing we really need is something called a *score function*, which is a function $f_k : \mathbb{R}^m \to \mathbb{R}$ that assigns a "score" that a certain data b

belongs to class k.

$$\text{score}(b, k) = f_k(b). \tag{22}$$

We can then classify the data u to a class k by selecting the class where it scored highest/lowest (in this project: lowest). As an example, b is here an image of a handwritten digit, and k is what digit the image contains. At this point you can ponder: What should a function that yields low values for 0-digits look like? What about a function that yields low values for 1-digits? Can you think of any functions we have defined in the project so far?

6.2 Distance to dictionary as score function

We will choose scoring function using what we found in Task 1 and 2. A natural scoring function is the distance to a dictionary that is trained on class k, that is

$$\text{score}(b, k) = D_{W^{(k)}}(b), \tag{23}$$

where $W^{(k)}$ is a dictionary fitted to data of class k. For the ENMF approach, we instead use the non-negative distances.

Using this idea, we formulate a method for automatic classification. During training, we fit dictionaries $W^{(i)}$ for each class. During testing:

1. Store all testing data in a matrix B.
2. Project the testing data onto all dictionaries $P_{W^{(i)}}(B)$.
3. Calculate the distance to each basis $D_{W^{(i)}}(B)$ (you do not have to store the projections, only the distances).
4. For each column b_i of B, predict the class equal to which dictionary it is closest to, that is, $c(b_i) = \arg\min_k D_{W^{(k)}}(b_i)$, where $c(b_i)$ is the predicted class of data b_i.

For the ENMF approach, we instead use the non-negative projections and distances.

A simple visual interpretation of this method is shown in figure 7 and 8.

One of the main benefits of an approach like this is that we can do the training completely independent of the classes that exist in the dataset, which means that we can easily add more classes if we need to. On the contrary, models trained with the knowledge of what classes we need to classify usually perform better, but lack this flexibility. The fact that we train the dictionaries independently of the classification problem we are trying to solve also means that the dictionaries will not necessarily be good at separating between different classes. For high d, we will expect that any orthogonal dictionary will be "too flexible" and will be able to represent a large variety of signals, which makes them unsuitable for this classification tasks. Contrarily, for small d we would expect the dictionaries to barely be able to represent the relevant signals, and this will lead to misclassifcation.

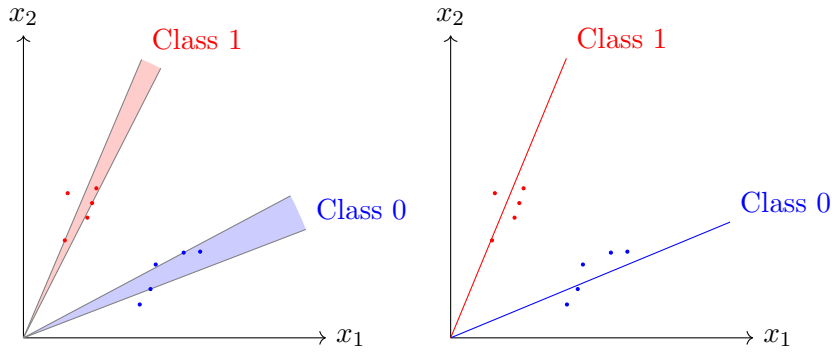

Figure 7: Illustration of training dictionaries to different classes. The small blue and red points are training data for the two different classes. Left: ENMF fitted to data with $d = 2$, which produces a cone that hopefully reconstructs data. Note that the booundaries of the cones passes through randomly selected datapoints. Right: Truncated SVD with $d = 1$, which produces a line (linear subspace) that best reconstructs data.

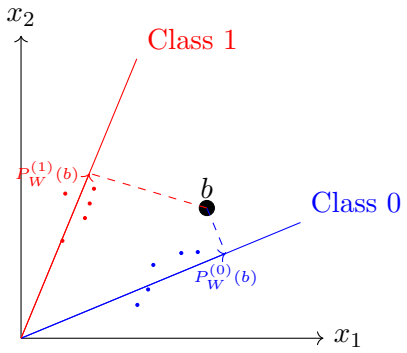

Figure 8: Illustration of testing for the truncated SVD approach. A new datapoint b is projected onto the two spaces spanned by the dictionaries $W^{(0)}$ and $W^{(1)}$, and we classify it as the corresponding dictionary it is closest to. In this case, we classify b as class 0.

6.3 Evaluating classification

When we are done classifying we need to evaluate how well our classification method performed. Perhaps the most common metric used for balanced datasets [7] is *accuracy*, which can be calculated as

$$\text{accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}. \tag{24}$$

Accuracy is useful for determining if a classification method overall performs well.

What if we want to check how many of our predictions were correct for a certain

[7] Balanced datasets are datasets where we have roughly equal amounts of data for all classes. For imbalanced datasets, we need better metrics.

class? For this we can use what is called *recall* for that class, which is defined as

$$\text{recall} = \frac{\text{Number of correct predictions for that class}}{\text{Total number of data for that class}}. \tag{25}$$

This can give a better estimate of how well our method performs for the individual classes. This way, we can see if our method is biased towards some classes.

Both of these metrics have available implementations, so it is up to you if you want to implement them yourself or use an existing implementation. Most likely you will have more control if you implement them yourself.

7 Task 3 (3 points)

- **a)** Implement a function that takes in a test dataset stored in a matrix B, and a list of dictionaries. The function should project the data onto the basises and store the distances to each basis. You should do this for both orthogonal dictionaries and non-negative dictionaries. Extend this function (or write an additional one) which classifies each vector stored in the columns of B to a class and returns a vector with predicted labels based on the method in section 6.2.

  Hint: Depending on how you program this, `np.argmin` might help you.

- **b)** Test your method by generating a test set which contains 3 different digits using handed out code and classifies them using both the SVD dictionaries and the ENMF dictionaries with $d = 32$. Calculate the test accuracy of your classification. Also calculate the recall of the invididual classes. Print your results in a table or any other readable format. Which method performs better? Do some integers have a noticeably different recall? Feel free to try different values for d.

  Hint: When you test this for 3 different digits, you should see accuracy above ~ 0.9, that is, over 90% of images gets classified correctly.

- **c)** For one of the integers in the test set, plot the image that is the "most likely" image containing one of the integers you choose, that is, the one that is closest to its basis for either the SVD or the ENMF approach. Plot this along with its projection onto this basis.

- **d** Select an image of that integer of the same class as c) which was misclassified. What is the main difference between this image and the one you found in c)? Does the misclassified image have any features that may be harder to capture compared to the other?

  Hint: The mislcassified image you found is most likely an *outlier*, that is, a datapoint that strongly deviates from the rest of the data.

- **e)** Generate a new dataset which contains the same digits as in **b)** and at least one more digit.

  Classify the data, calculate the accuracy and recall and print them in a table. What do you observe about the results compared to **b)**? Can you explain the behaviour?

  Hint: Because the test set is created randomly, you have to redo the calculations of the projections for the entire dataset. In practice, you only have to do training for the extra classes you added.

- **f)** We now want to do a larger experiment to see how the results depend on the amount of basis vectors d in our dictionaries. For a large range of d, create dictionaries that all contain d basis vectors, classify and store the accuracy. For example, you can choose $d = 2^i, i = 1, ...10$. Plot the accuracy of the classification

against the amount of basis vectors d for both the SVD and the ENMF approach in the same plot. Can you explain the behaviour for small and large d, and how does this compare to the results from task 2? How does the two methods perform differently? Which d yields the best accuracy for the two methods?

Hint: For this problem it might be smart to set the safe division factor in the non-negative projection algorithm a bit higher, like $\delta = 10^{-2}$, as this might improve results.

- **g)** Write a suitable discussion and conclusion to the entire project, which should show what you have learnt. Include some pros and cons of the different dictionaries we have experimented with this project, and the overall approach we have taken to classification.