Least squares, SVD and PCA

In this practice we will study two applications of linear algebra. First we are going to use least squares to "solve" an overdetermined system of equations, arising in a parameter estimation problem. The second part of the practice is about Principal Component Analysis, a linear method of dimensionality reduction. We will see how this can be applied to face recognition.

1 Linear Regression

Suppose we have m samples $x_i, y \in \mathbb{R}$, $i = 1, \ldots, m$, as shown in Figure 1. Assume that we know that there is a functional dependence between x and y: $y = f(x)$. In general we do not know f, but we know that it belongs to a certain class of functions (for example, we know that the dependence between x and y should be logarithmic, or polynomial). In order to approximate f, we will use a *regression* technique.

In this Assignment, we will use linear regression to fit a polynomial of degree 3 to the data, given by:

$$\hat{f}(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3.$$

Observe that $\hat{f}$ is not a linear function of x, but it is linear on the coefficients w_i. The coefficients are unknowns we have to determine. We will do so by

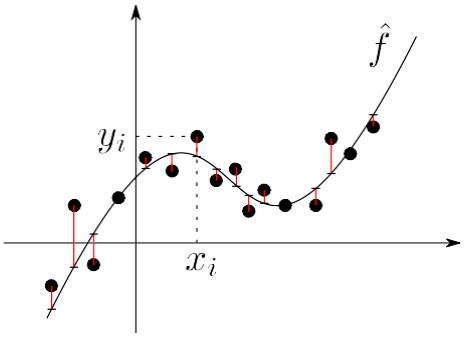

Figure 1: The problem of regression. We want to fit a function $\hat{f}$ to some data x_i, y_i, $i = 1, \ldots, m$. The function should be chosen to minimize the error between $f(x_i)$ and y_i.

minimizing the sum of squared errors, between the predicted value $\hat{f}(x)$ and the measured value, y:

$$J(w_0, w_1, w_2, w_3) = \sum_{i=1}^{m} |y_i - \hat{f}(x_i)|^2 = \sum_{i=1}^{m} (y_i - (w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i)^3)^2$$

We can express J in matrix notation as

$$J(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{\Phi w}\|^2 \tag{1}$$

where $\boldsymbol{y} = [y_1, y_2, \ldots, y_m]^T \in \mathbb{R}^m$, $\boldsymbol{w} = [w_0, w_1, w_2, w_3]^T \in \mathbb{R}^n$ ($n = 4$) and

$$\boldsymbol{\Phi} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & x_m^3 \end{bmatrix}$$

is an $m \times n$ matrix, called the *design matrix*. In general we will have more measurements than parameters: $m >> n$.

We are going to compute the minimum of J using two methods: the normal equations and the SVD.

1.1 Computing the minimum via the normal equations

We seek for the coefficient vector $\boldsymbol{w}^*$ which minimizes J:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} J(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \|\boldsymbol{\Phi w} - \boldsymbol{y}\|^2$$

As you saw in class, if $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ is invertible, the optimal coefficient vector $\boldsymbol{w}^*$ can be computed from the following equation:

$$\boldsymbol{w}^* = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}. \tag{2}$$

These are the *normal equations*. Recall that $\Phi \boldsymbol{w}^*$ is the projection of $\boldsymbol{y}$ over Im$\boldsymbol{\Phi}$.

1. *Complete the Python function* `polyfit_inv_normal_eq` *for computing* $\boldsymbol{w}^*$. *Follow the comments provided in the code.*

2. *Complete the Python function* `polyfit_main`*, to verify that the* $\boldsymbol{w}^*$ *returned by* `polyfit_inv_normal_eq` *satisfies that the residue* $r = \boldsymbol{y} - \boldsymbol{\Phi w}^*$ *is orthogonal to* Im$\boldsymbol{\Phi}$, i.e. $r \perp$ Im$\boldsymbol{\Phi}$. *Hint:* Im$\boldsymbol{\Phi}$ *is the space generated by the columns of* $\boldsymbol{\Phi}$.

1.2 Minimization with the SVD

Note that, for solving the normal equations using Eq. (2) the matrix $\boldsymbol{\Phi}^T \boldsymbol{\Phi}$ needs to be invertible. Let us now derive a different way to compute $\boldsymbol{w}^*$ based on the Singular Value Decomposition (SVD), which can be extended even to the case in which $\boldsymbol{\Phi}^T \boldsymbol{\Phi}$ is not invertible.

Using the SVD decomposition, we can express

$$\boldsymbol{\Phi} = USV^T$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices (a square matrix $A \in \mathbb{R}^{m \times m}$ is orthogonal when $A^T A = AA^T = I_m$), and $S \in \mathbb{R}^{m \times n}$ is a rectangular diagonal matrix. The diagonal values of S, $s_{ii} = \sigma_i > 0$, for $i = 1, \ldots, q$ are the singular values. The number of singular values q is the rank of $\boldsymbol{\Phi}$, thus, $q \leq \min\{m, n\}$.

We will use the SVD to compute the *pseudo-inverse* of $\boldsymbol{\Phi}$, $\boldsymbol{\Phi}^\dagger$. Let us first recall what is the pseudo-inverse of a matrix, and how it is computed.

The pseudo-inverse

We will define define pseudo-inverse in two steps: first we are going to define the pseudo-inverse of a diagonal matrix. Then we are going to extend this definition to any matrix.

Pseudo-inverse of a diagonal matrix. Let S be an $m \times n$ rectangular diagonal matrix. The pseudo-inverse of S, $S^\dagger$, is also a diagonal matrix, but with dimensions $n \times m$. The diagonal elements of $S^\dagger$ are computed as follows

$$s^\dagger_{ii} = \begin{cases} s_{ii}^{-1} & \text{if } s_{ii} \neq 0 \\ 0 & \text{if } s_{ii} = 0, \end{cases}$$

for $i = 1, \ldots, \min(m, n)$.

Pseudo-inverse of any matrix. Let A be an $m \times n$ matrix. We define its pseudo-inverse based on the SVD decomposition and the previous definition of the pseudo-inverse of a diagonal matrix. Using the SVD decomposition we can express $A = USV^T$. We then define

$$A^\dagger = V S^\dagger U^T.$$

Note that S is a diagonal matrix with the singular values on the diagonal, so we already know how to compute its pseudo-inverse.

Note. The pseudo-inverse is a generalization of the inverse which applies to any matrix (recall that only some square matrices are invertible). If a matrix A is invertible, we have that $S^\dagger = S^{-1}$. The pseudo-inverse has many interesting properties. In the following we will make use of one of them.

3. *Generate a* 5×3 *random matrix, A. Compute its SVD* $A = USV^T$ *using the command* `svd`*. Verify that the U and V are orthogonal matrices and S is diagonal. Verify that* $A = USV^T$*.*

4. *Compute the pseudo-inverse of S,* $S^\dagger$*. Use it to compute the pseudo-inverse of A,* $A^\dagger$*. Verify that* $A^\dagger A$ *is the identity matrix of size* 3*. What happens with* $AA^\dagger$*?*

Solving least squares with the pseudo-inverse

It turns out that we can compute the solution $\boldsymbol{w}^*$ to the least squares problem using the pseudo-inverse of the design matrix $\boldsymbol{\Phi}$ as follows:

$$\boldsymbol{w}^* = \boldsymbol{\Phi}^\dagger \boldsymbol{y} = V S^\dagger U^T \boldsymbol{y}.$$

5. *Complete the Python function* `polyfit_svd_normal_eq` *for computing* $\boldsymbol{w}^*$*. Follow the comments provided in the code.*

6. *Compare with the solution obtained by inverting the normal equations. Compare also with the solution provided by the Python command* `pinv`*.*

2 Principal Components Analysis

In this section we are going to explore the application of Principal Components Analysis (PCA) for *dimensionality reduction*. Imagine we have a data set formed by several (say m) points in $\mathbb{R}^n$, $X = \{x_1, \ldots, x_m\}$. For example, in Figure 2 we show a set of points in $\mathbb{R}^2$. Another example, in the following section, the "points" will be m images of faces. Since the points are in $\mathbb{R}^n$ we are using n coefficients to represent them. In many cases, we would be interested in finding a representation of the dataset, which allows to encode each point x_i using $p << n$ coefficients.

Let us assume for simplicity that the points are centered at the origin (*i.e.* the mean or barycenter is at the origin: $\sum_i x_i = 0$). The idea behind PCA is to approximate the point set by projecting it on a p dimensional subspace V_p, with $p \leq n$.

To do that, we will build an orthonormal basis $\mathrm{V}_n = \{v_1, v_2, \ldots, v_n\}$ of $\mathbb{R}^n$, specially design to adapt to the dataset (these are the red vectors v_1, v_2 in Figure 2). Any point x_i from the data set can be expressed by its n coordinates on the basis: $[\langle x_i, v_1\rangle, \ldots, \langle x_i, v_n\rangle]^T \in \mathbb{R}^n$. This means that we can recover x_i as

$$x_i = \sum_{j=1}^{n} \langle x_i, v_j\rangle v_j.$$

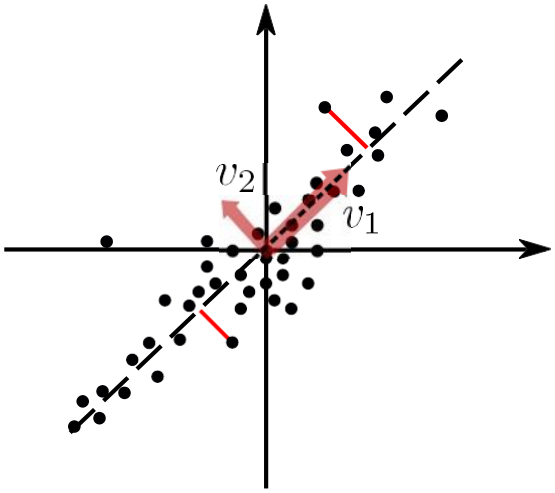

Figure 2: PCA of a two dimensional point set. The principal directions v_1, v_2 form an orthogonal basis of $\mathbb{R}^2$. Note that v_1 "aligns" with the point set: most of the variation of the points is along the direction v_1. These directions are chosen to minimize the mean squared projection error. In the Figure we show the projection errors over the principal direction v_1 for two points (red lines).

If only keep the first p coefficients $[\langle x_i, v_1\rangle, \ldots, \langle x_i, v_p\rangle]^T$, we recover the projection of x_i over V_p, the subspace spanned by the first p vectors in the basis:

$$P_{V_p}(x_i) = \sum_{j=1}^{p} \langle x_i, v_j\rangle v_j.$$

What we want to find is a basis that best approximates the dataset. Meaning that for each $p \leq n$, V_p is the p-dimensional vector space minimizing the mean squared projection error

$$\frac{1}{m} \sum_{i=1}^{m} \|P_{V_p}(x_i) - x_i\|^2.$$

The resulting vectors $v_1, \ldots, v_n$ are called the *principal directions* of the set of points $X = \{x_1, \ldots, x_m\}$. For any $x \in \mathbb{R}^n$, the coefficients $\langle x, v_1\rangle, \ldots, \langle x, v_n\rangle$ are called the *principal components* of x.

Computation and Properties of the Principal Components

Let us consider a data set $X = \{x_1, \ldots, x_m\} \subset \mathbb{R}^n$. Let $\boldsymbol{X}$ be the data matrix: the rows of $\boldsymbol{X}$ are the vectors $x_i \in X$.

- The principal directions of X form an orthonormal basis $v_1, \ldots, v_n$ of $\mathbb{R}^n$ given by the eigenvectors of the empirical covariance matrix $\frac{1}{m-1}\boldsymbol{X}^T\boldsymbol{X}$.

  Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$ be the corresponding eigenvalues.

- For $p = 1, \ldots, n$, the subspace V_p spanned by the first p principal directions is the p-dimensional subspace which minimizes the mean squared projection error:

$$\frac{1}{m}\sum_{i=1}^{m} \|x_i - P_{V_p}(x_i)\|^2 = \sum_{j=p+1}^{n} \lambda_j$$

- Equivalently V_p is also the p-dimensional subspace that maximizes the projection variance:

$$\frac{1}{m}\sum_{i=1}^{m} \|P_{V_p}(x_i)\|^2 = \sum_{j=1}^{p} \lambda_j$$

1. *Complete the Python function* `pca_prin_dir` *for computing the p first principal directions via the eigenvectors of* $\boldsymbol{X}^T\boldsymbol{X}$*. Follow the comments in the code.*

2. *Complete the Python functions* `pca_prin_comp` *for computing the p first principal components* $[\langle x, v_i \rangle]_{i=1,\ldots,p}$ *of a point* x*.*

3. *Complete the Python functions* `pca_reconstruct` *for reconstructing a point* x *from its principal components.*

4. *For the* flat ellipsoid *dataset (provided together with the code) run the* `pca_ellipsoid_test`*. Four figures will open. Read the code and explain what each figure is showing. Answer the questions asked in the code.*

2.1 Eigenfaces: application of PCA for face recognition

Figure 3: Some of the images used for the face recognition application

We are going to apply PCA to reduce the dimensionality in a face recognition application. For this we are going to use the functions you completed. The objective is to recognize a person among a reduced set of people, given a frontal picture of his face. The data we have is a data base of m images of faces from different persons, in different positions, expressions and illumination conditions. The images are in gray scale, of size $n = 211 \times 219$. We consider them as vectors in $\mathbb{R}^{48319}$, a high dimensional space.

We will apply PCA to reduce the dimensionality of the space to p (we will try different values for p). We will proceed as follows.

2.1.1 Off-line stage: learning the principal directions.

We consider now that $\mathrm{X} = \{x_1, \ldots, x_m\}$ are the face images in vector form.

1. Compute the first p eigenvectors and eigenvalues of $\frac{1}{m-1}\boldsymbol{X}^T\boldsymbol{X}$. Let $\boldsymbol{V}_p$ be the matrix whose columns are these eigenvectors (the principal directions).

2. For each $x_i \in \mathrm{X}$, compute its p first principal components: This can be done with the following matrix multiplication:

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{V}_p,$$

   where the ith row of $\boldsymbol{Z}$, denoted by z_i, are the principal components of the face x_i. z_i is the low dimensional representative of x_i.

2.1.2 On-line stage: recognition with principal components.

The objective now is to recognize a new face x. Again, x is a grayscale image of the face, considered as a vector. To simplify the algorithm, we assume that x is an image from a person of the dataset. Our recognizer is a very simple nearest neighbor classifier.

1. Compute the principal components of x,

$$z = [\langle x, v_1\rangle, \langle x, v_2\rangle, \ldots, \langle x, v_p\rangle].$$

2. Look in the database for the face with nearest principal components:

$$j^* = \arg\min_j \|z - z_j\|^2.$$

5. *The code for this part is provided. Run the provided functions and Read their code and comments and explain what each of these functions do, and the figures that open. In step 6 try changing the number of principal components used and comment how this affects to the number of classification errors.*