Machine Learning 1 WS18/19

Submission for Exercise Sheet 3

Hendrik Makait 384968
Michael Hoppe 362514
Wai Tang Victor Chan 406094
Rudi Poepsel Lemaitre 373017
Jonas Piotrowski 399334
Aki Saksala 399293

Machine Learning I Ex3

1) $x_1, \dots, x_n \in \mathbb{R}^d$

$J(\theta) = \sum_{k=1}^{n} \|\theta - x_k\|^2$ to be minimized with respect to $\theta \in \mathbb{R}^d$

a) use Lagrange multipliers to find θ that minimizes $J(\theta)$ subject to $\theta^T b = 0$; $b \in \mathbb{R}$

$$\boxed{\min_{\theta} J(\theta) \qquad \text{s.t. } \theta^T b = 0, \ b \in \mathbb{R}^d}$$

(I) Lagrange function:

$$\mathcal{L}(\theta, \lambda) = \sum_{k=1}^{n} (\|\theta - x_k\|^2) + \lambda (\theta^T b)$$
$$= \sum_{k=1}^{n} ((\theta - x_k)^T (\theta - x_k)) + \lambda(\theta^T b)$$

(II) set gradient of $\mathcal{L}$ to zero:

① $\mathcal{L}_\theta'(\theta, \lambda) = \sum_{k=1}^{n} \left(\frac{\partial (\theta - x_k)^T (\theta - x_k)}{\partial \theta} \right) + \frac{\partial(\lambda(\theta^T b))}{\partial \theta} = 0$

$$= \sum_{k=1}^{n} (2\theta^T - x_k^T - x_k) + \lambda b \qquad = 0$$

② $\mathcal{L}_\lambda'(\theta, \lambda) = \theta^T b = 0$

b) use Lagrange multipliers to find θ that minimizes $J(\theta)$ s.t. $\|\theta - c\|^2 = 1$; $c \in \mathbb{R}^d$

$$\boxed{\min_{\theta} J(\theta) \quad \text{s.t.} \quad \underset{c \in \mathbb{R}^d}{\|\theta - c\|^2 = 1}}$$

I. Lagrange function:

$$\mathcal{L}(\theta, \lambda) = \sum_{k=1}^{n} \left(\|\theta - x_k\|^2\right) + \lambda \left(\|\theta - c\|^2 - 1\right)$$

$$= \sum_{k=1}^{n} \left((\theta - x_k)^T (\theta - x_k)\right) + \lambda \left((\theta - c)^T (\theta - c) - 1\right)$$

II. set gradient of $\mathcal{L}$ to zero

① $\mathcal{L}'_\theta(\theta, \lambda) = \sum_{k=1}^{n} \left(2\theta^T - x_k^T - x_k\right) + \lambda($

② $\mathcal{L}'_\lambda(\theta, \lambda) = \|\theta - c\|^2 - 1$

Scanned by CamScanner

Machine Learning I Ex3

2a) Since S is symmetric $\left(\begin{aligned} S^T &= \sum_{k=1}^{n} \left((x_k - m)(x_k - m)^T \right)^T \\ &= \sum_{k=1}^{n} \left((x_k - m)^T \right)^T (x_k - m)^T = S \end{aligned} \right)$,

there exists orthogonal matrix Q and diagonal matrix Σ such that $S = Q \Sigma Q^T$ (by Spectral Theorem)

Then $\sum_{i=1}^{n} S_{ii} = tr(S) = tr(Q\Sigma Q^T) = tr(QQ^T\Sigma) = tr(I\Sigma) = tr(\Sigma) = \sum_{i=1}^{n} \lambda_i$

All $\lambda_i \geq 0$, since for any eigenvector v_i with eigenvalue λ_i,

$$\lambda_i \underbrace{v_i^T v}_{\geq 0} = v_i^T(\lambda_i v_i) = x^T(S v_i) = \underbrace{\sum_{k=1}^{n} \left((x_k - m)^T v_i \right)^T (x_k - m)^T v_i}_{\geq 0} \geq 0$$

Therefore $\sum_{i=1}^{n} S_{ii} = \sum_{i=1}^{n} \lambda_i \geq \lambda_1$

2b) For the bound to be tight, we need $\lambda_2, \ldots, \lambda_n \ll \lambda_1$

That means $S \approx \lambda_1 v_1 v_1^T$

The condition would be that the data are "close" to linear, i.e. lying "mostly" in one direction.

2c) Take any $i = 1, \dots, n$, denote $e_i = (0, \dots 0, \overset{\overset{\text{ith entry}}{\downarrow}}{1}, 0 \dots 0)^T$

Then $e_i = \sum_{j=1}^{n} t_j v_j$, as linear combination of unit eigenvectors of S, where each $t_j \in \mathbb{R}$

Then
$$
\begin{aligned}
S_{ii} &= e_i^T S e_i \\
&= \left(\sum_{j=1}^{n} t_j v_j^T\right) S \left(\sum_{j=1}^{n} t_j v_j\right) \\
&= \left(\sum_{j=1}^{n} t_j v_j^T\right)\left(\sum_{j=1}^{n} t_j \lambda_j v_j\right) \\
&= \sum_{j=1}^{n} \lambda_j t_j^2 \underbrace{v_j^T v_j}_{=1} \qquad \left(\text{since } v_i^T v_j = 0 \text{ for } i \neq j\right) \\
&= \sum_{j=1}^{n} \lambda_j t_j^2 \\
&\leq \lambda_1 \underbrace{\sum_{j=1}^{n} t_j^2}_{=\|e_i\|^2 = 1} \\
&= \lambda_1
\end{aligned}
$$

Hence $\lambda_1 \geq S_{ii}$ for all $i = 1, \dots, n$ $\Rightarrow$ $\lambda_1 \geq \max_{i=1}^{d} S_{ii}$

2d) This bound is tight when the dominant principal axis the data is "close" to parallel to one of the coordinate axis.

Exercise 3 Q3

a) S is invertible and symetric $\Rightarrow$ $S^{1/2}$ invertible and symetric

$\Rightarrow J(w) = \|Sw\| - \frac{1}{2} w^T S w = \| S^{1/2} S^{1/2} w\| - \frac{1}{2} (S^{1/2} w)^T (S^{1/2} w)$

$= \|Sw\| - \frac{1}{2} w^T S w = \| S^{1/2} v\| - \frac{1}{2} v^T v$

$$\nabla_v J(w) = \frac{S^{1/2 T} S^{1/2} v}{\| S^{1/2} v\|} - v = \frac{Sv}{\|S^{1/2} v\|} - v$$

Gradient ascend

$$v \leftarrow v + \gamma \left(\frac{Sv}{\|S^{1/2} v\|} - v \right)$$

$$\Leftrightarrow \quad S^{1/2} w \leftarrow S^{1/2} w \gamma \left(\frac{S^{3/2} w}{\|Sw\|} - S^{1/2} w \right)$$

$$w \leftarrow w \gamma \left(\frac{Sw}{\|Sw\|} - w \right) = \frac{Sw}{\|Sw\|} \quad \text{if } \gamma = 1$$

b) $$\nabla_w J(w) = \frac{S^T S w}{\|Sw\|} - Sw = 0$$

$$\Leftrightarrow Sw = \frac{S^T S w}{\|Sw\|} = \frac{SSw}{\|Sw\|}$$

$$w = S^{-1} \cdot \frac{SSw}{\|Sw\|} = \frac{Sw}{\|Sw\|}$$

$\Rightarrow$ w has to be a unit vector

Principal Component Analysis

Introduction

In this exercise, you will experiment with two different techniques to compute the principal components of a dataset:

- **Basic PCA**: The standard technique based on singular value decomposition.
- **Iterative PCA**: A technique that progressively optimizes the PCA objective function.

Principal component analysis is applied here to modeling handwritten characters data (characters "O" and "I") using the dataset introduced in the paper "L.J.P. van der Maaten. 2009. A New Benchmark Dataset for Handwritten Character Recognition". The dataset consists of black and white images of 28×28 pixels, each representing a handwritten character. For the purpose of the PCA analysis, these images are interpreted as 784-dimensional vectors with values between 0 and 1. Three methods are provided for your convenience and are available in the module `utils` that is included in the zip archive. The methods are the following:

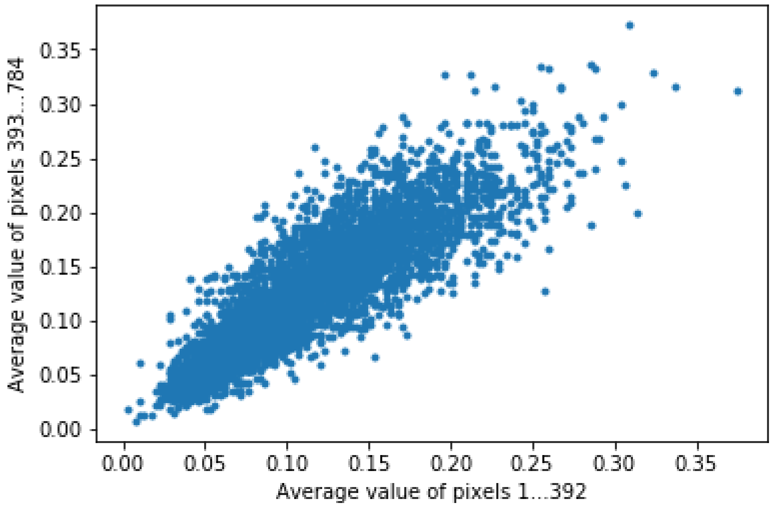
- **`utils.load()`** load data from the file `characters.csv` and stores them in a data matrix of size 4631×784. (The data is a subset of the original dataset available here: [http://lvdmaaten.github.io/publications/misc/characters.zip (http://lvdmaaten.github.io/publications/misc/characters.zip)](http://lvdmaaten.github.io/publications/misc/characters.zip))
- **`utils.scatterplot(...)`** produces a scatter plot from a two-dimensional data set. Each point in the scatter plot represents one handwritten character. This method provides a convenient way to produce two-dimensional PCA plots.
- **`utils.render(...)`** takes a matrix of size $n \times 784$ as input, interprets it as n images of size 28×28, and renders these images in the IPython notebook.

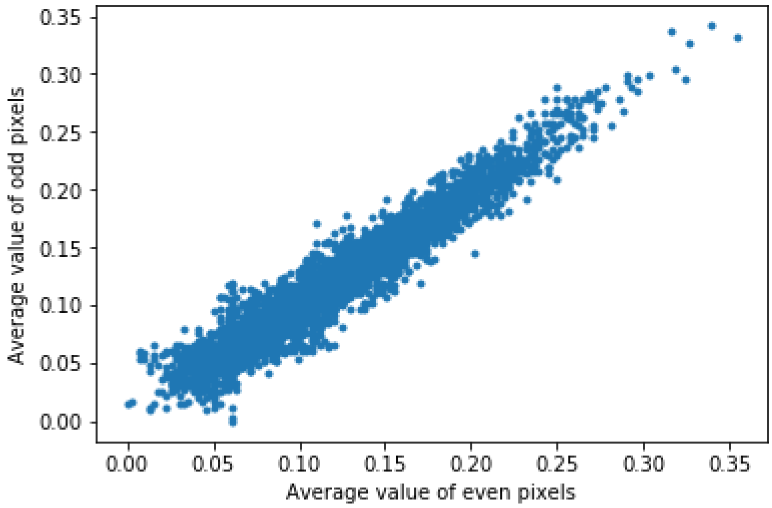
A demo code that makes use of these methods is given below. It performs basic data analysis, for example, plotting simple statistics for each data point in the dataset, or rendering a few examples randomly selected from the dataset.

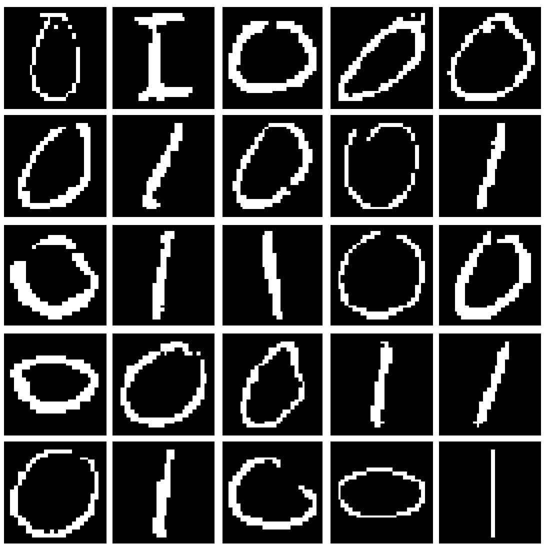
In [111]:

```
import utils,numpy
%matplotlib inline

# Load the characters "O" and "I" from the handwritten characters dataset
X = utils.load()

print('dataset size: %s'%str(X.shape))

# Plot some statistics of the data using the scatterplot function
utils.scatterplot(X[:,:392].mean(axis=1),X[:,392:].mean(axis=1),
                  xlabel='Average value of pixels 1...392',
                  ylabel='Average value of pixels 393...784')
utils.scatterplot(X[:,::2].mean(axis=1),X[:,1::2].mean(axis=1),
                  xlabel='Average value of even pixels',
                  ylabel='Average value of odd pixels')

# Render some randomly selected examples
R=numpy.random.randint(0,len(X),[25])
utils.render(X[R])
```

```
dataset size: (4631, 784)
```







The preliminary data analysis above does not reveal particularly interesting structure in the data. For example scatter plots fail to let appear the two types of characters present in the dataset ("O" and "I"). Therefore, we would like to gain more insight on the dataset by performing a more sophisticated analysis based on PCA.

PCA with Singular Value Decomposition (15 P)

As shown during the lecture, principal components can be found by solving the eigenvalue problem

$$\boldsymbol{S}\boldsymbol{w} = \lambda\boldsymbol{w}.$$

While we could eigendecompose the scatter matrix to find the desired eigenvalues and eigenvectors (for example, by using the function `numpy.linalg.eigh`), we usually prefer to recover principal components directly from singular value decomposition

$$\boldsymbol{X} = \boldsymbol{U}\,\boldsymbol{\Sigma}\,\boldsymbol{V}^\top,$$

where the principal components and projection of data onto these components can also be retrieved from the matrices $\boldsymbol{U}$, $\boldsymbol{\Sigma}$ and $\boldsymbol{V}$.

Tasks:

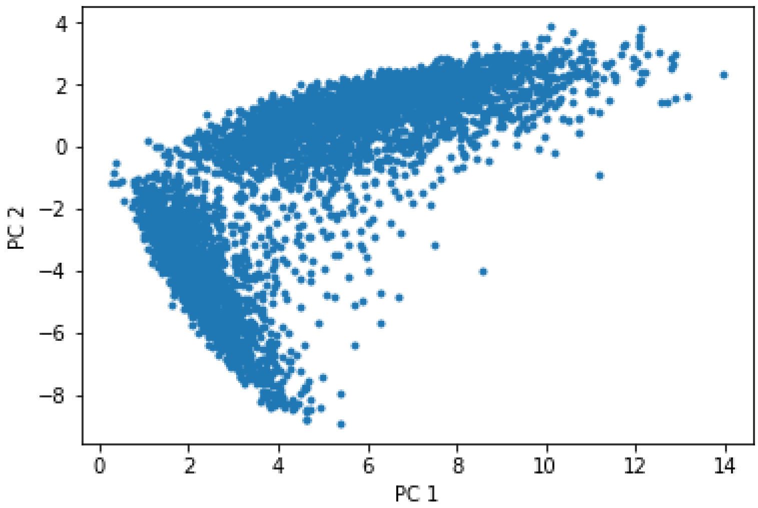
- **Compute the principal components of the data using the function `numpy.linalg.svd` .**
- **Measure the computational time required to find the principal components. Use the function `time.time()` for that purpose. Do *not* include in your estimate the computation overhead caused by loading the data, plotting and rendering.**
- **Plot the projection of the dataset on the first two principal components using the function `utils.scatterplot` .**
- **Visualize the 25 leading principal components using the function `utils.render` .**

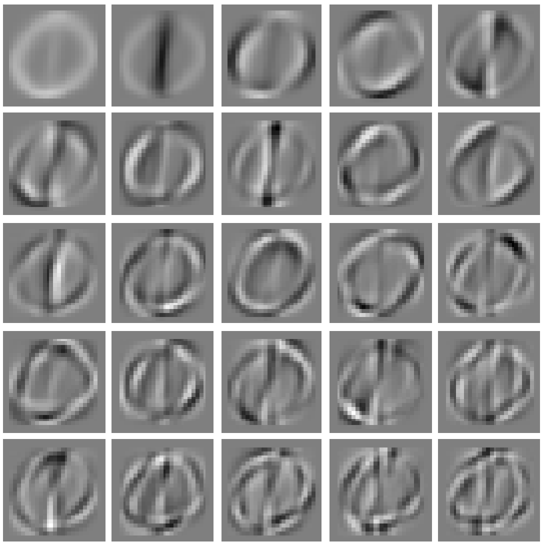
Note that if the algorithm runs for more than 1 minute, you might be doing something wrong.

In [112]:

```
import time

t_start = time.time()
U, S, Vt = numpy.linalg.svd(X, full_matrices=False)
t_end = time.time()
print("Computation time: {} seconds".format(t_end - t_start))
# extract projection of data onto principal components by
# multiplying U and the diagonal matrix S
projection = numpy.matmul(U, numpy.diag(S)).transpose()
utils.scatterplot(projection[0], projection[1], xlabel="PC 1", ylabel="PC 2")
# the principal components are the rows of Vt
utils.render(Vt[:25])
```

```
Computation time: 0.43018126487731934 seconds
```





Iterative PCA (15 P)

The objective that PCA optimizes is given by

$$J(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{S} \boldsymbol{w}$$

subject to

$$\boldsymbol{w}^\top \boldsymbol{w} = 1.$$

The power iteration algorithm maximizes this objective using an iterative procedure. It starts with an initial weight vector $\boldsymbol{w}$, and iteratively applies the update rule

$$\boldsymbol{w} \leftarrow \frac{\boldsymbol{S}\boldsymbol{w}}{\|\boldsymbol{S}\boldsymbol{w}\|}$$

Tasks:

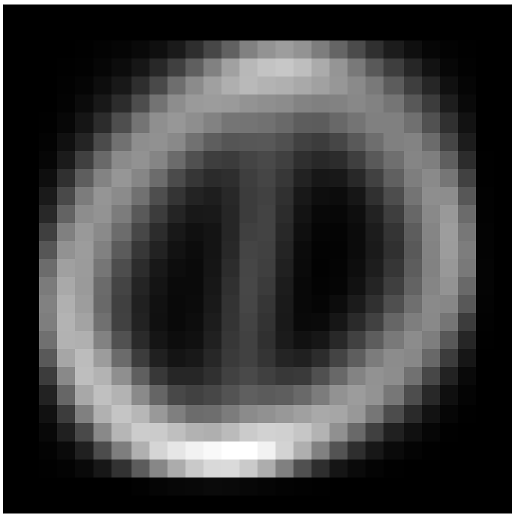
- **Implement the iterative procedure. Use as a stopping criterion the value of $J(\boldsymbol{w})$ between two iterations increasing by less than 0.01.**
- **Print the value of the objective function $J(\boldsymbol{w})$ at each iteration.**
- **Measure the time taken to find the principal component.**
- **Visualize the the eigenvector $\boldsymbol{w}$ obtained after convergence using the function `utils.render`.**

Note that if the algorithm runs for more than 1 minute, you might be doing something wrong.

In [113]:

```
import math

X = utils.load().transpose()
t_start = time.time()
S = numpy.matmul(X, X.transpose())
n = math.sqrt(1 / X.shape[0])
w = numpy.full(X.shape[0], n)
obj_a = numpy.matmul((numpy.matmul(w.transpose(), S)), w)
print("Value of J(w): {}".format(obj_a))

Sw = numpy.matmul(S, w)
Sw_norm = numpy.linalg.norm(Sw)
w = Sw / Sw_norm
obj_b = numpy.matmul((numpy.matmul(w.transpose(), S)), w)
print("Value of J(w): {}".format(obj_a))
while (obj_b - obj_a) > 0.01:
    obj_a = obj_b
    Sw = numpy.matmul(S, w)
    Sw_norm = numpy.linalg.norm(Sw)
    w = Sw / Sw_norm
    obj_b = numpy.matmul((numpy.matmul(w.transpose(), S)), w)
    print("Value of J(w): {}".format(obj_a))

t_end = time.time()
print("\nComputation time: {} seconds".format(t_end - t_start))
utils.render(numpy.array([w]))
```

```
Value of J(w): 63966.61224489796
Value of J(w): 63966.61224489796
Value of J(w): 128488.24282898365
Value of J(w): 128924.4071584183
Value of J(w): 128970.96817624214
Value of J(w): 128977.0264139759
Value of J(w): 128977.82326230484
Value of J(w): 128977.92815396568
Value of J(w): 128977.94196206174

Computation time: 0.023209095001220703 seconds
```



In [ ]:

```

```