Technische Universität Berlin
Institut für Softwaretechnik
und theoretische Informatik

Wintersemester 2019/2020
Prof. Dr. Klaus-Robert Müller

Machine Learning I - Homework III

Jacky 391049, Viktor 392636, Duc 395220, Laura 391342, Laura 392032

---

1. Let $(x_k)_{k=1}^n \subset \mathbb{R}^d$ be a data set of n samples. We consider the objective (??) function

$$J(\theta) = \sum_{k=1}^{n} \|\theta - x_k\|^2$$

to be minimized with respect to the parameter $\theta \in \mathbb{R}^d$. It can shown that in absence of constraints for θ, the θ^* that minimizes this objective is given by the empirical mean $\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$.

(a) Using the method of Lagrange multipliers, find the parameter θ that minimizes $J(\theta)$ subject to the constraint $\theta^T b = 0$, where $b \in \mathbb{R}^d$. Give a geometrical interpretation to your solution.

We define the Lagrangian (use $y := \theta$ for convenience)

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

and compute its gradient:

$$\nabla\mathcal{L}(y, \lambda) = \begin{pmatrix} \sum_{k=1}^{n} 2(y - x_k) - \lambda b \\ y^T b \end{pmatrix}.$$

Setting this equal to zero yields

$$\sum_{k=1}^{n} 2(y - x_k) = \lambda b \iff 2ny - 2\sum_{k=1}^{n} x_k = \lambda b \implies y = \frac{1}{n}\sum_{k=1}^{n} x_k + \frac{\lambda}{2n} \cdot b$$

Calculating $y^T b$ yields $\frac{1}{n}\sum_{k=1}^{n} x_k^T b + \frac{\lambda}{2n} b^T b$, setting zero yields $\lambda = -\frac{2}{\|b\|^2}\sum_{k=1}^{n} x_k^T b$. Plugging this in yields

$$y = \frac{1}{n}\sum_{k=1}^{n} x_k - \frac{1}{\|b\|^2 n}\sum_{k=1}^{n} x_k^T b \cdot b$$

This solution to the minimization problem is the projection of the minimum of J on the hyperplane corresponding to b

(b) Using the same method, find the parameter θ that minimizes $J(\theta)$ subject to $\|\theta - c\|^2 = 1$, where $c \in \mathbb{R}^d$. Give a geometrical interpretation to your solution.

Again we will use LAGRANGE multipliers to minimize $J(\theta)$ subject to the constrain $||\theta - c||^2 = 1$. The LAGRANGIAN in this case is (setting $\theta = y$)

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

Then we obtain

$$\nabla\mathcal{L}(y, \lambda) = \begin{pmatrix} 2(ny - \sum_{k=1}^{n} x_k - \lambda(y - c)) \\ ||y - c||^2 - 1 \end{pmatrix}$$

Setting this zero yields:

$$\begin{aligned} 0 &= 2(ny - \sum_{k=1}^{n} x_k - \lambda(y - c)) \\ \Leftrightarrow 0 &= ny - \sum_{k=1}^{n} x_k - \lambda(y - c) \\ \Leftrightarrow y &= \frac{\sum_{k=1}^{n} x_k - \lambda c}{(n - \lambda)} \end{aligned}$$

We further observe

$$1 = ||y - c|| = ||\frac{\sum_{k=1}^{n} x_k - \lambda c}{(n - \lambda)} - c|| = ||\frac{\sum_{k=1}^{n} x_k - \lambda c - (n - \lambda)c}{(n - \lambda)}|| = \frac{1}{n - \lambda}||\sum_{k=1}^{n} x_k - cn||$$

And therefore $\lambda = -||\sum_{k=1}^{n} x_k - cn|| + n$ and $y = \frac{\sum_{k=1}^{n} x_k - \lambda c}{(||\sum_{k=1}^{n} x_k - cn||)}$. This solution is the projection of the minimizer of J on the closed unit ball around c.

2. We consider a data set $(x_k)_{k=1}^{n} \subset \mathbb{R}^d$. The empirical mean m and the scatter matrix S are given by

$$m = \frac{1}{n}\sum_{k=1}^{n} x_k \quad \text{and} \quad S = \sum_{k=1}^{n}(x_k - m)(x_k - m)^T.$$

Let λ_1 be the largest eigenvalue of the matrix S. It quantifies the amount of variation in the data on the first principal component. Because computation of the full scatter matrix and respective eigenvalues can be slow, it can be useful to relate them to the diagonal elements of the scatter matrix $\{S_{ii}\}$ than can be computed in linear time.

(a) Show that $\sum_{k=1}^{d} S_{ii}$ is an upper bound to the eigenvalue λ_1.

Answer: As S is symmetric, there exists an eigendecomposition of $S = Q\Lambda Q^T$, where Λ is a diagonal matrix containing the eigenvalues and Q is orthogonal.

The expression $\sum_{k=1}^{d} S_{ii}$ is also called the trace of the matrix, $\text{tr}(S)$. Similar matrices (A and B are similar if there exists a $P \in \text{GL} : A = P^{-1}BP$) have the same trace, i.e. $\text{tr}(\Lambda) = \text{tr}(S)$. It is well know that $\text{tr}(A) = \sum_{k=1}^{n} n_k\lambda_k$ for any matrix A, where λ_k are the eigenvalues of A with algebraic multiplicities $n_k \in \mathbb{N}_{\geq 1}$. Therefore,

$$\lambda_1 \leq n_1\lambda_1 \leq \text{tr}(\Lambda) = \text{tr}(S).$$

(b) State the conditions on the data for which the upper bound is tight.

Answer: If λ_1 is the only non-zero eigenvalue with algebraic multiplicity 1, the answer for (a) immediately yields that the bound is tight. Examples for such matrices $A \in \mathbb{R}^{n\times n}$ must have the characteristic polynomial $p_A(\lambda) = \lambda^{n-1}(\lambda - \lambda_1)$ for some $n \in \mathbb{N}_{\geq 1}$. All such matrices represent projections into a one-dimensional subspace.

(c) Show that $\max_{i=1}^{d} S_{ii}$ is a lower bound to λ_1.

Proof. It is well known that $\lambda_1 = \max_{||x||=1} xS^T x$. Using this fact one obtains

$$\lambda_1 = \max_{||x||=1} xS^T x \geq e_i S^T e_i = S_{ii} \quad \forall i \in \{1, ..., d\}$$

where e_i is the i-th unit vector. Thus, it follows immediately that $\max_{i=1}^{d} S_{ii} \leq \lambda_i$. □

(d) State the conditions on the data for which the lower bound is tight.

Answer: Equality holds whenever there exists at least one $i \in \{1, ..., d\}$ such that e_i is an associated eigenvector for the largest eigenwert. To see this, assume e_i is an eigenvector for the largest eigenwert of the scatter matrix S, i.e.

$$Se_i = \lambda_1 e_i = S_{ii} e_i.$$

Therefore, $\lambda_1 = \max_{i=1}^{d} S_{ii}$.

3. When performing principal component analysis, computing the full eigendecomposition of the scatter matrix S is typically slow, and we are often only interested in the few first principal components. An efficient procedure to find the first eigenvector is the power iteration method, which starts with a random vector $w \in \mathbb{R}^d$, and iterative applies the parameter update $w \leftarrow \frac{Sw}{\|Sw\|}$ until some convergence criterion is met.

(a) Show that application of the power iteration method is equivalent to defining the unconstrained objective

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

and performing the gradient ascent $v \leftarrow v + \gamma \frac{\partial I}{\partial v}$, where $v = S^{0.5} w$ is a reparametrization of v, for some learning γ. We assume that the matrix S is invertible.

Answer: First we substitute $v = S^{0.5} w$ in J and use that S is symmetric:

$$J(w) = ||S^{0.5}(S^{0.5}w)|| - w^T S^{0.5} S^{0.5} w \Rightarrow J(v) = ||S^{0.5}v|| - \frac{1}{2} v^T v$$

Then we obtain

$$\frac{\partial J}{\partial v} = \frac{\partial}{\partial v}(||S^{0.5}v|| - \frac{1}{2} v^T v) = \frac{Sv}{||S^{0.5}v||} - v$$

Choosing the learning rate $\gamma = 1$ yields

$$v \leftarrow v + \gamma(\frac{Sv}{||S^{0.5}v||} - v)$$

$$S^{0.5}w \leftarrow S^{0.5}w + (\frac{SS^{0.5}w}{||S^{0.5}S^{0.5}w||} - S^{0.5}w)$$

Since S is invertible we obtain

$$w \leftarrow w + (\frac{Sw}{||S^{0.5}S^{0.5}w||} - w) = \frac{Sw}{||Sw||}$$

We observe that this is equivalent to the power iteration

(b) Show that a necessary condition for w to maximize the objective $J(w)$ is to be a unit vector (i.e. $\|w\| = 1$).

Proof. First, the gradient of J is given by

$$\nabla J(w) = \frac{SSw}{||Sw||} - Sw.$$

Setting the gradient of J to zero yields

$$Sw = \frac{1}{||Sw||} SSw \implies w = \frac{1}{||Sw||} Sw,$$

which shows that w must be a unit vector. Note that S^{-1} exists, otherwise the argument does not work. □

sheet03

November 4, 2019

1 Team 42

Follow us on instagram.com/official_team42!

---

2 Principal Component Analysis

2.1 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:

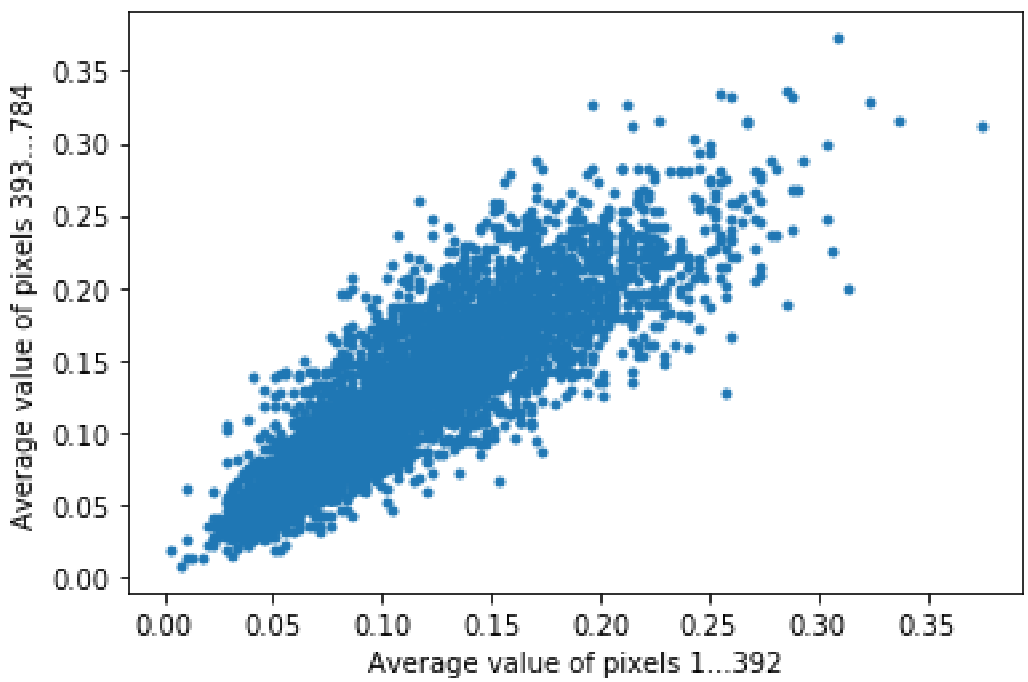
- **`utils.load()`** load data from the file `characters.csv` and stores them in a data matrix of size 4631×784. (The data is a subset of the original dataset available here: http://lvdmaaten.github.io/publications/misc/characters.zip)
- **`utils.scatterplot(...)`** produces a scatter plot from a two-dimensional 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.

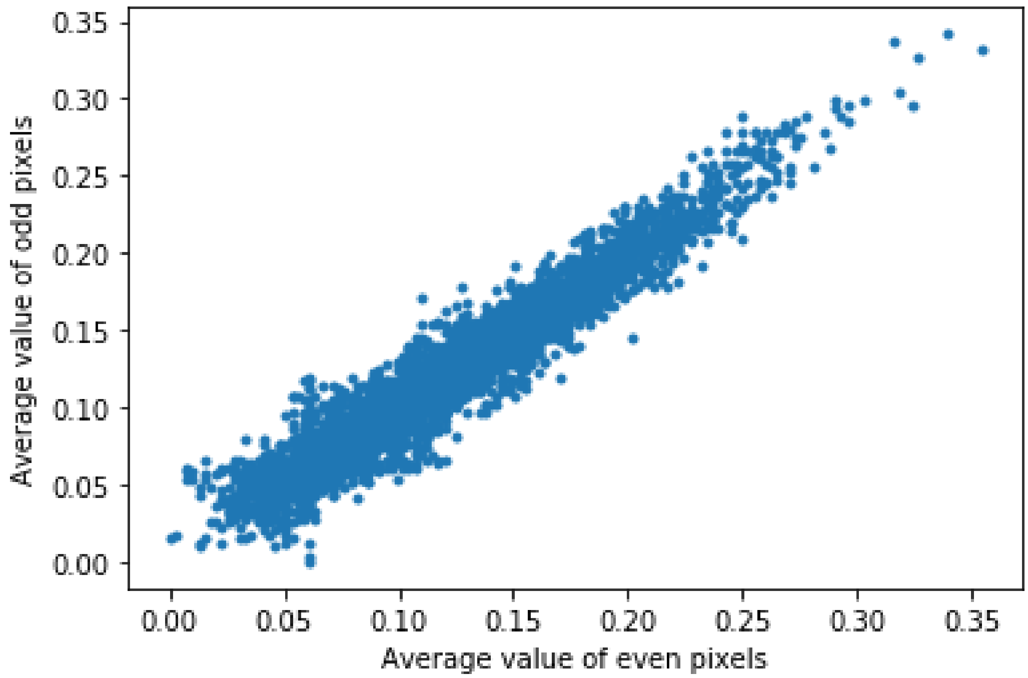
A demo code that makes use of these methods is given below. It performs basic data analysis, for example, plotting simple statistics for each data point in the dataset, or rendering a few examples randomly selected from the dataset.

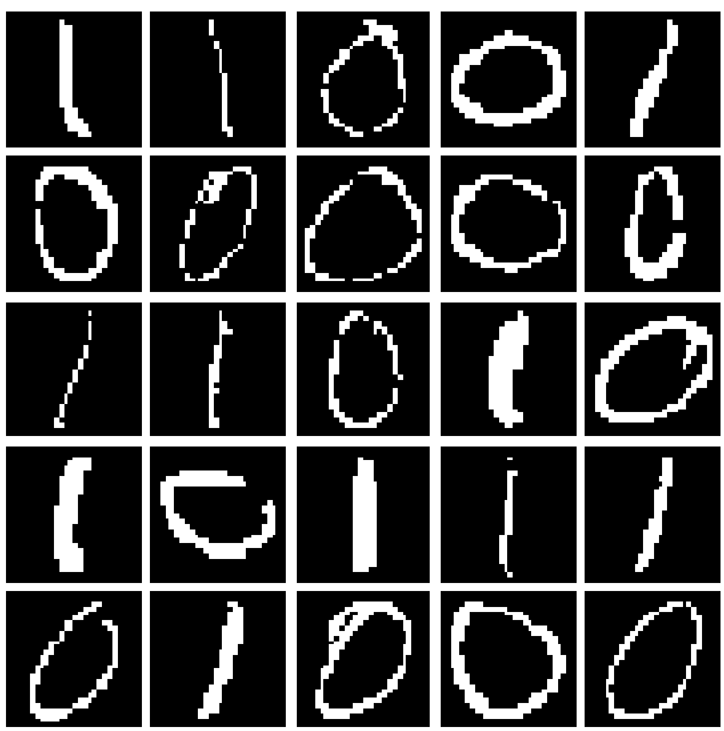
[31]:
```
import utils
import numpy as np
```

```python
%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.

2.2 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:

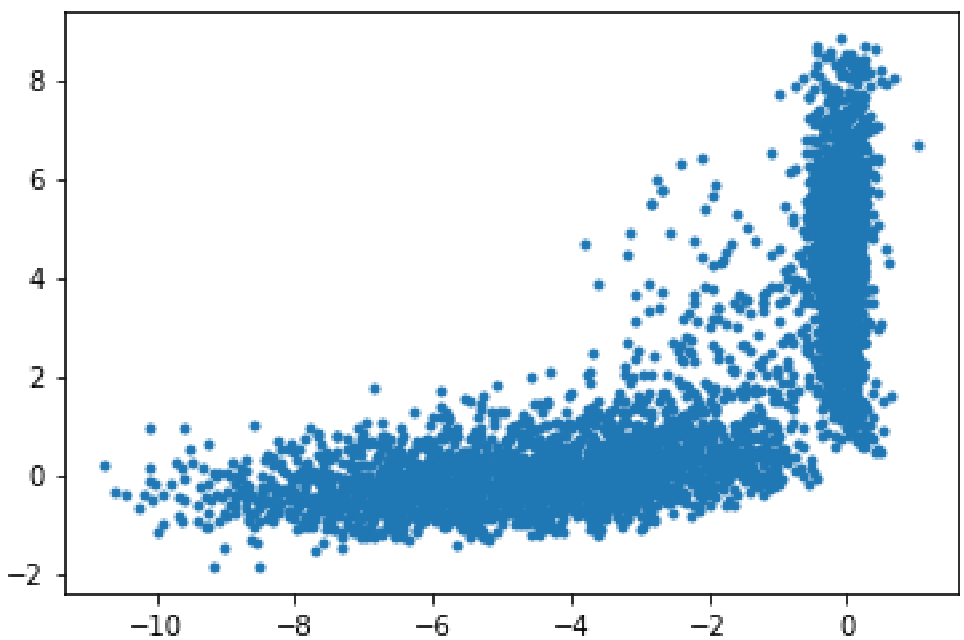
- **Compute the principal components of the data using the function `numpy.linalg.svd`.**
- **Measure the computational time required to find the principal components. Use the function `time.time()` for that purpose. Do *not* include in your estimate the computation overhead caused by loading the data, 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`.**

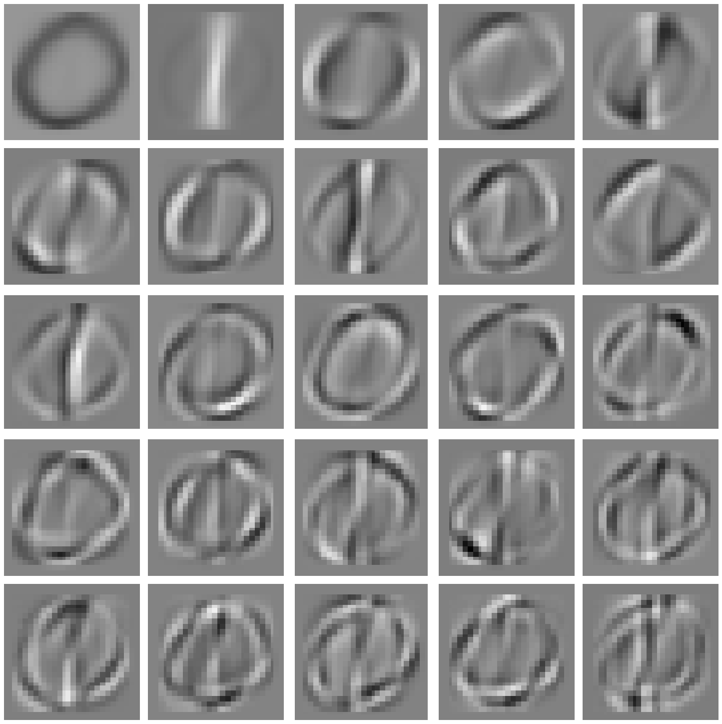
Note that if the algorithm runs for more than 1 minute, you might be doing something wrong.

[49]:
```
import time

# Load data
print("Loading data...")
X = utils.load()

# Centering is done by subtracting the mean of all x_i with each row of X
print("Centering data...")
X_center = X - np.mean(X, axis=1)[:,np.newaxis]

# Measure time
t0 = time.time()

# Compute SVD
print("Computing SVD...")
U, S, VT = numpy.linalg.svd(X_center, full_matrices=False)
S = np.diag(S)
```

```python
V = VT.transpose()
print("Finished computing SVD")
print("U.shape={}, V.shape={}, S.shape={}".format(U.shape, V.shape, S.shape))

# Compute first TWO principal components
print("Computing principal components...")
PC = U.dot(S)
print("PC.shape={}".format(PC.shape))

# Output time
t1 = time.time()
print("Time elapsed: {}".format(t1-t0))

# Plot the projection of the dataset on the first two principal components
utils.scatterplot(PC[:,0],PC[:,1])

# Visualize the 25 leading principal components
utils.render(V[:,:25].transpose())
```

```
Centering data…
Computing SVD…
Finished computing SVD
U.shape=(4631, 784), V.shape=(784, 784), S.shape=(784, 784)
Computing principal components…
PC.shape=(4631, 784)
Time elapsed: 1.152665138244629
```





2.3 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:

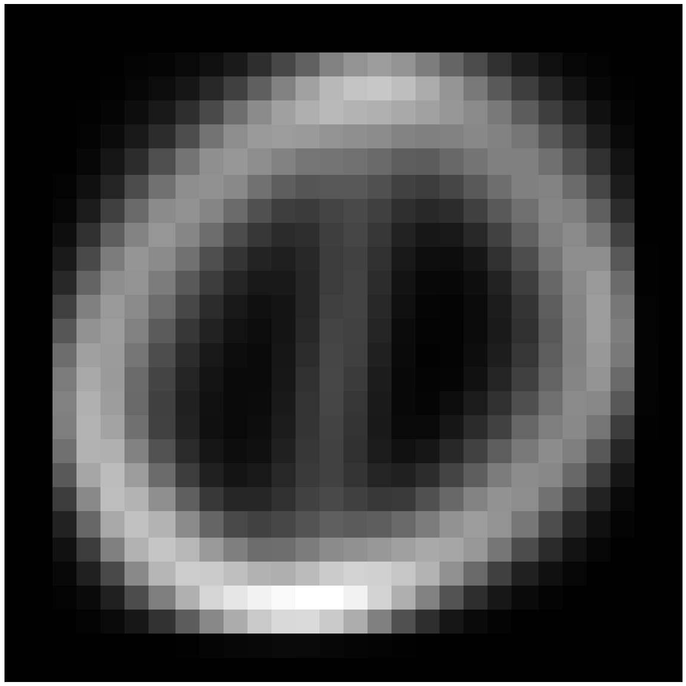
- **Implement the iterative procedure. Use as a stopping criterion the value of $J(\boldsymbol{w})$ between two iterations increasing by less than 0.01.**
- **Print the value of the objective function $J(\boldsymbol{w})$ at each iteration.**
- **Measure the time taken to find the principal component.**

- **Visualize the the eigenvector 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.

[71]:
```
import time

print("Loading data...")
X = utils.load()

print("Compute scatter matrix...")
S = X.transpose().dot(X)

print("Initialize random start vector")
w = np.random.random(size=(S.shape[0],1))

def J(x):
    return x.transpose().dot(S.dot(x))

thres = 0.01

t0 = time.time()
while True:
    w_old = w
    Sw = S.dot(w)
    w = Sw / np.linalg.norm(Sw)
    Jw = J(w)
    print("Objective function: {}".format(Jw))
    if np.linalg.norm(Jw - J(w_old)) <= thres:
        break
t1 = time.time()
print("Time elapsed: {}".format(t1-t0))

# Visualize the the eigenvector w obtained after convergence
utils.render(w.transpose())
```

```
Loading data…
Compute scatter matrix…
Initialize random start vector
Objective function: [[128531.68858079]]
Objective function: [[128929.95023765]]
Objective function: [[128971.70950765]]
Objective function: [[128977.12431733]]
Objective function: [[128977.83615764]]
Objective function: [[128977.9298517]]
Objective function: [[128977.94218556]]
Objective function: [[128977.94380921]]
Time elapsed: 0.02971792221069336
```



[ ]:

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.

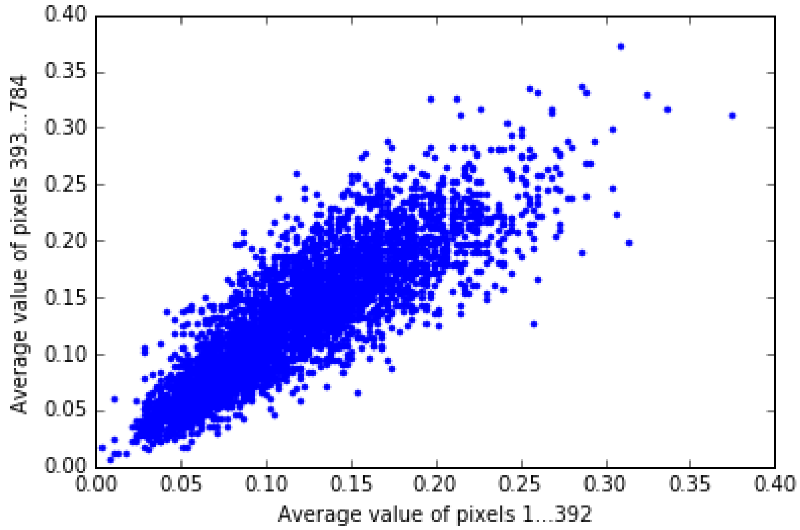
Principal component analysis is applied here to modeling handwritten characters data (characters "O" and "I") using the dataset introduced in the paper "L.J.P. van der Maaten. 2009. A New Benchmark Dataset for Handwritten Character Recognition". The dataset consists of black and white images of 28×28 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:

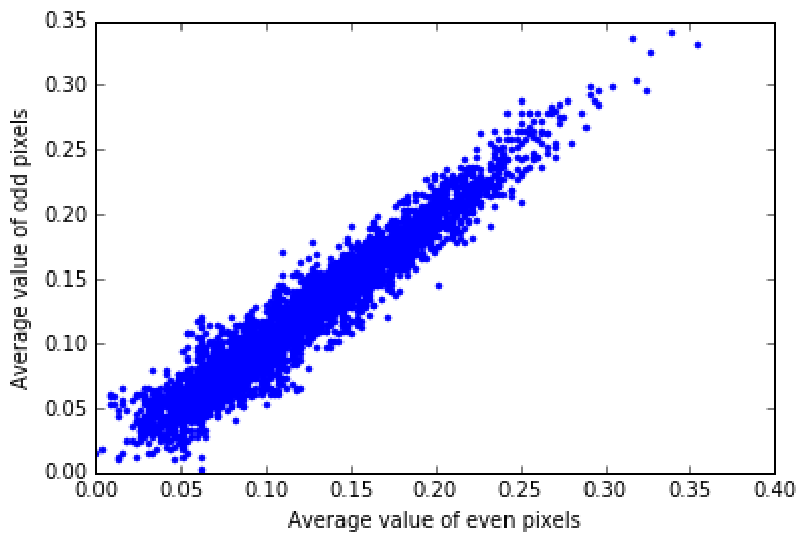
- `utils.load()` load data from the file `characters.csv` and stores them in a data matrix of size 4631×784. (The data is a subset of the original dataset available here: http://lvdmaaten.github.io/publications/misc/characters.zip)
- `utils.scatterplot(...)` produces a scatter plot from a two-dimensional 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.

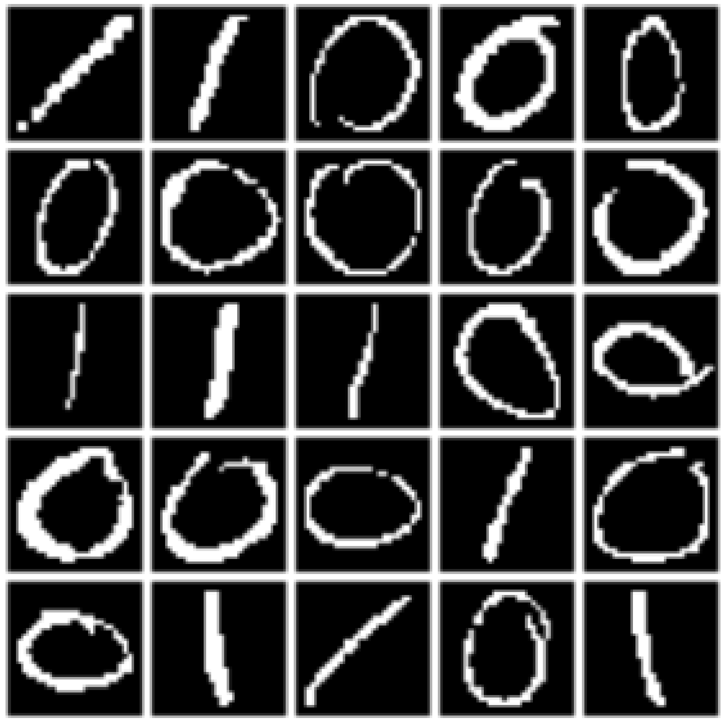
A demo code that makes use of these methods is given below. It performs basic data analysis, for example, plotting simple statistics for each data point in the dataset, or rendering a few examples randomly selected from the dataset.

In [1]:
```python
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)
```


0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Average value of pixels 393...784
Average value of pixels 1...392


0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Average value of odd pixels
Average value of even pixels



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:

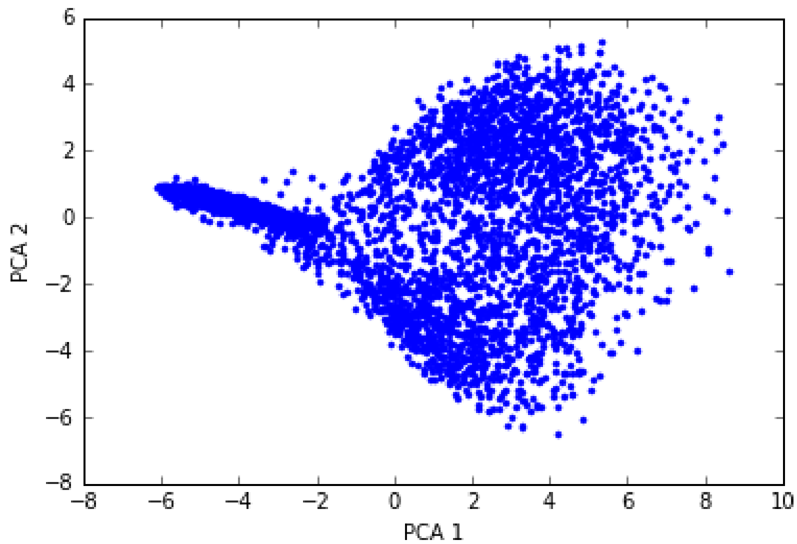
- **Compute the principal components of the data using the function** `numpy.linalg.svd`**.**
- **Measure the computational time required to find the principal components. Use the function** `time.time()` **for that purpose. Do not include in your estimate the computation overhead caused by loading the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

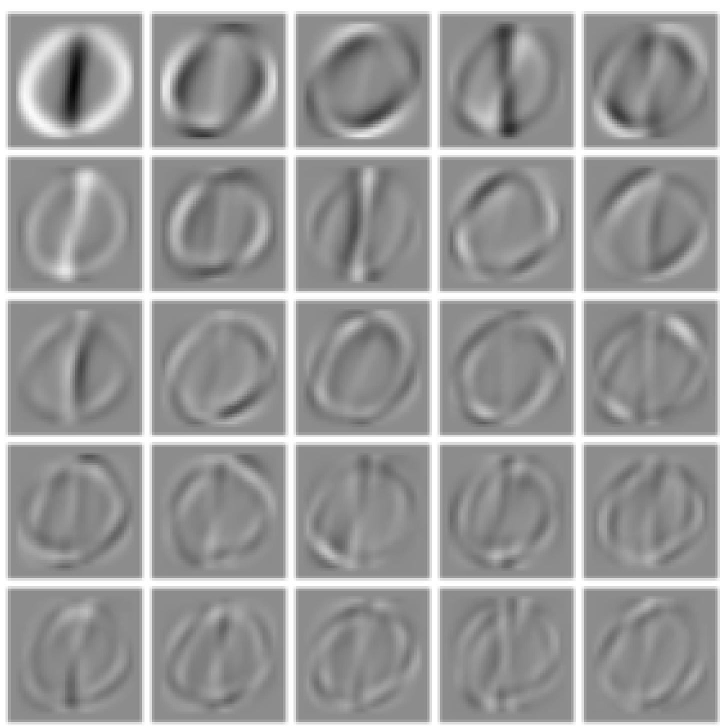
- **Visualize the 25 leading principal components using the function** `utils.render`**.**

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

In [2]:
```
### REPLACE BY YOUR CODE
import solutions; solutions.basic()
###
```

```
Time: 8.869 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 $J(w)$ between two iterations increasing by less than 0.01.**
- **Print the value of the objective function $J(w)$ at each iteration.**
- **Measure the time taken to find the principal component.**
- **Visualize the the eigenvector 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 [3]:
```
### REPLACE BY YOUR CODE
import solutions; solutions.iterative()
###
```

```
iteration  0   J(w) =    381.129
iteration  1   J(w) =  37876.300
iteration  2   J(w) =  59212.799
iteration  3   J(w) =  60549.709
iteration  4   J(w) =  60618.765
iteration  5   J(w) =  60622.761
iteration  6   J(w) =  60623.022
iteration  7   J(w) =  60623.041
iteration  8   J(w) =  60623.042
stopping criterion satisfied
Time: 0.935 seconds
```