sheet03-programming

November 5, 2025

[35]:
```
print("Group 49")
```

```
Group 49
```

[36]:
```
import numpy,sklearn,sklearn.datasets,utils
%matplotlib inline
```

1 Principal Component Analysis

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

- **Singular Value Decomposition (SVD)**
- **Power Iteration**: A technique that iteratively optimizes the PCA objective.

We consider a random subset of the Labeled Faces in the Wild (LFW) dataset, readily accessible from sklearn, and we apply some basic preprocessing to discount strong variations of luminosity and contrast.

[37]:
```
D = sklearn.datasets.fetch_lfw_people(resize=0.5)['images']
D = D[numpy.random.mtrand.RandomState(1).permutation(len(D))[:2000]]*1.0
D = D - D.mean(axis=(1,2),keepdims=True)
D = D / D.std(axis=(1,2),keepdims=True)
print(D.shape)
```

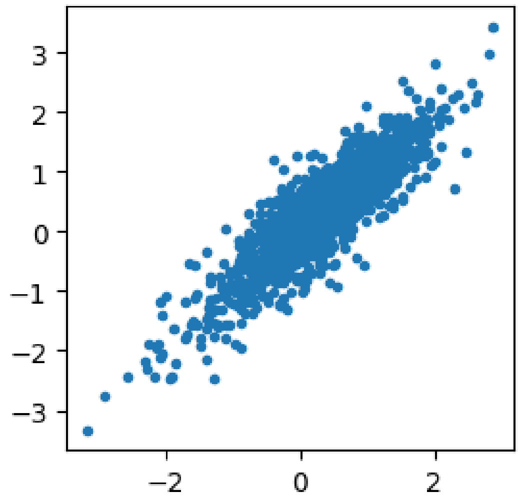
```
(2000, 62, 47)
```

Two functions are provided for your convenience and are available in `utils.py` that is included in the zip archive. The functions are the following:

- **`utils.scatterplot`** produces a scatter plot from a two-dimensional data set.
- **`utils.render`** takes an array of data points or objects of similar shape, and renders them in the IPython notebook.

Some demo code that makes use of these functions is given below.

[38]:
```
utils.scatterplot(D[:,32,20],D[:,32,21]) # Plot relation between adjacent pixels
utils.render(D[:30],15,2,vmax=5)         # Display first 30 examples in the data
```



1.1 PCA with Singular Value Decomposition (15 P)

Principal components can be found computing a singular value decomposition. Specifically, we assume a matrix $\bar{X}$ whose columns contain the data points represented as vectors, and where the data points have been centered (i.e. we have substracted to each of them the mean of the dataset). The matrix $\bar{X}$ is of size $d \times N$ where d is the number of input features and N is the number of data points. This matrix, more specifically, the rescaled matrix $Z = \frac{1}{\sqrt{N}}\bar{X}$ is then decomposed using singular value decomposition:

$$U\Lambda V = Z$$

The k principal components can then be found in the first k columns of the matrix U.

Tasks:

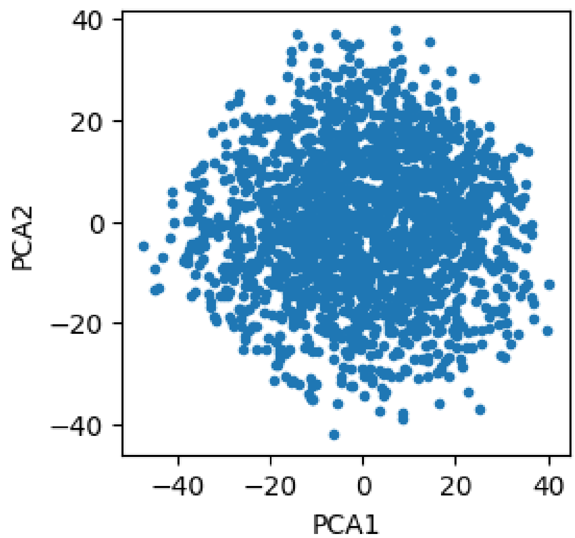
- **Compute the principal components of the data using the function `numpy.linalg.svd`.**
- **Measure the computational time required to find the principal components. Use the function `time.time()` for that purpose. Do *not* include in your estimate the computation overhead caused by loading the data, plotting and rendering.**
- **Plot the projection of the dataset on the first two principal components using the function `utils.scatterplot`.**
- **Visualize the 60 leading principal components using the function `utils.render`.**

Note that if the algorithm runs for more than 3 minutes, there may be some error in your implementation.

[39]:
```python
import time

N, height, width = D.shape
# Number of input features
d = height * width

# (N, height, width) -> (N x d)
# Each row is now an image
X_flat = D.reshape(N, d)

dataset_mean = X_flat.mean(axis=0)
# center data points
X_centered = X_flat - dataset_mean
# Transpose to get (N x d) -> (d x N)
X_bar = X_centered.T

Z = (1.0 / numpy.sqrt(N)) * X_bar

start_time = time.time()

principal_components, singl_values, Vh = numpy.linalg.svd(Z, full_matrices=True)

time_to_find_principal_components = time.time() - start_time
print(f"Time: {time_to_find_principal_components:.3f} seconds")

k = 2
singl_values_k = singl_values[:k]
Vh_k = Vh[:k, :]

# (2, 2) @ (2, 2000) -> (2, 2000) and normalize with sqrt(N)
projected_coordinates = numpy.diag(singl_values_k) @ (Vh_k) * numpy.sqrt(N)

utils.scatterplot(projected_coordinates[0, :], projected_coordinates[1, :],␣
  ↪xlabel='PCA1', ylabel='PCA2')

# Visualize the 60 leading principal components using the function `utils.
  ↪render`
k = 60

# Get the first 60 PCs (columns of U)
leading_principal_components = principal_components[:, :k]

# Reshape for rendering: (d, k) -> (k, d) -> (k, h, w)
```

```
leading_principal_components_images = leading_principal_components.T.
↪reshape((k, height, width))

utils.render(leading_principal_components_images, int(k / 4), 4, vmax=0.05)
```

```
Time: 2.398 seconds
```



When looking at the scatter plot, we observe that much more variance is expressed in the first two principal components than in individual dimensions as it was plotted before. When looking at the principal components themselves which we render as images, we can see that the first principal components correspond to low-frequency filters that select for coarse features, and the following principal components capture progressively higher-frequency information and are also becoming more noisy.

1.2 PCA with Power Iteration (15 P)

The first PCA algorithm based on singular value decomposition is quite expensive to compute. Instead, the power iteration algorithm looks only for the first component and finds it using an iterative procedure. It starts with an initial weight vector $w \in \mathbb{R}^d$, and repeatedly applies the update rule

$$w \leftarrow Sw \big/ \|Sw\|.$$

where S is the covariance matrix defined as $S = \frac{1}{N}\bar{X}\bar{X}^\top$. Like for standard PCA, the objective that iterative PCA optimizes is $J(w) = w^\top Sw$ subject to the unit norm constraint for w. We can therefore keep track of the progress of the algorithm after each iteration.

Tasks:

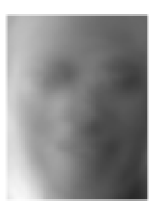
- **Implement the power iteration algorithm. Use as a stopping criterion the value of $J(w)$ between two iterations increasing by less than 0.01.**
- **Print the value of the objective function $J(w)$ at each iteration.**
- **Measure the time taken to find the principal component.**
- **Visualize the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

Note that if the algorithm runs for more than 1 minute, there may be some error in your implementation.

[40]:
```
start_time = time.time()

SEED = 123456
w = numpy.random.mtrand.RandomState(SEED).rand(d)
w = w / numpy.linalg.norm(w) # Normalize to unit norm

tolerance = 0.01
max_iterations = 1000
J_w_prev = numpy.inf * -1
iteration = 0

while iteration < max_iterations:
    X_bar_transpose = X_bar.T @ w
    # S_w = (1/N) * X_bar @ X_bar_transpose (d x N) @ (N x 1) -> (d x 1)
    S_w = (1.0 / N) * X_bar @ X_bar_transpose

    # J(w) = w^T @ S_w
    J_w = w.T @ S_w

    print(f"iteration {iteration:2} J(w) =  {J_w:8.3f}")

    # Check stopping criterion: J(w) increasing by less than 0.01
    J_w_increase = J_w - J_w_prev
    if iteration > 0 and J_w_increase < tolerance:
```

```python
        print(f"stopping criterion satisfied")
        break

    # Update rule: w <- S*w / ||S*w||
    w = S_w / numpy.linalg.norm(S_w)

    # Update for next iteration
    J_w_prev = J_w
    iteration += 1

end_time = time.time()
time_taken = end_time - start_time

print(f"Time: {time_taken:.3f} seconds")

utils.render(w, 1, 1, vmax=0.05)

###
```

```
iteration  0 J(w) =     0.197
iteration  1 J(w) =   107.610
iteration  2 J(w) =   165.715
iteration  3 J(w) =   196.344
iteration  4 J(w) =   220.822
iteration  5 J(w) =   240.139
iteration  6 J(w) =   252.550
iteration  7 J(w) =   259.336
iteration  8 J(w) =   262.783
iteration  9 J(w) =   264.514
iteration 10 J(w) =   265.399
iteration 11 J(w) =   265.865
iteration 12 J(w) =   266.117
iteration 13 J(w) =   266.257
iteration 14 J(w) =   266.335
iteration 15 J(w) =   266.380
iteration 16 J(w) =   266.406
iteration 17 J(w) =   266.421
iteration 18 J(w) =   266.430
stopping criterion satisfied
Time: 0.337 seconds
```



We observe that the computation time has decreased significantly. The difference of performance becomes larger as the number of dimensions and data points increases. We can observe that the principal component is the same (sometimes up to a sign flip) as the one obtained by the SVD algorithm.