Freie Universität Berlin

Prof. R. Rojas

Mustererkennung, WS17/18

Übungsblatt 8

Boyan Hristov, Nedeltscho Petrov

10. Dezember 2017

Link zum Git Repository: https://github.com/BoyanH/FU-MachineLearning-17-18/tree/master/Solutions/Homework8

Principal Component Analysis

Analyse anhand Klassifizierung von Ziffern mit Hilfe von PCA

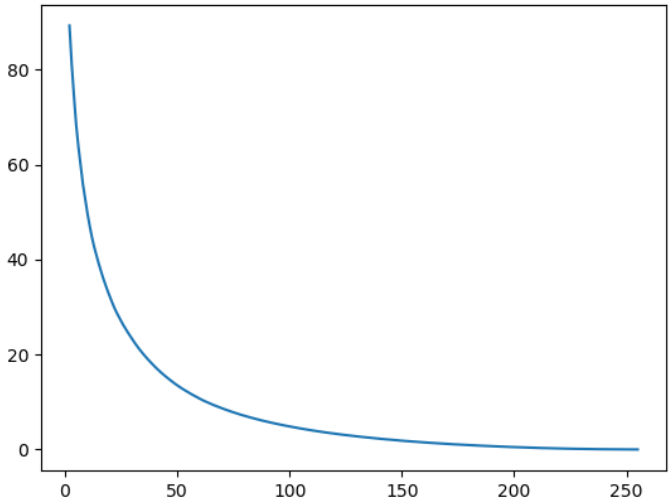
Wir haben unsere PCA Implementierung für die Klassifizierung von Digits angewandt. Wir haben die folgende Ergebnisse bekommen:

```
Score LDA without PCA: 0.9168458781362007
Score LDA with PCA, k=230: 0.9186379928315412
Score LDA with PCA, k=30: 0.9010752688172043
```

Wir wollten mal schauen, was passiert, wenn man einen nicht binären Klassifikator anwendet zusammen mit PCA. Die Ergebnisse waren ziemlich gut. Natürlich bekommt man nicht viel bessere Ergbenisse mit PCA, aber wenn man auch mit etwas schlechtere zufrieden ist, kann man das notwendige Speicherplatz deutlich reduzieren (in unserem Fall um Faktor 8.5).

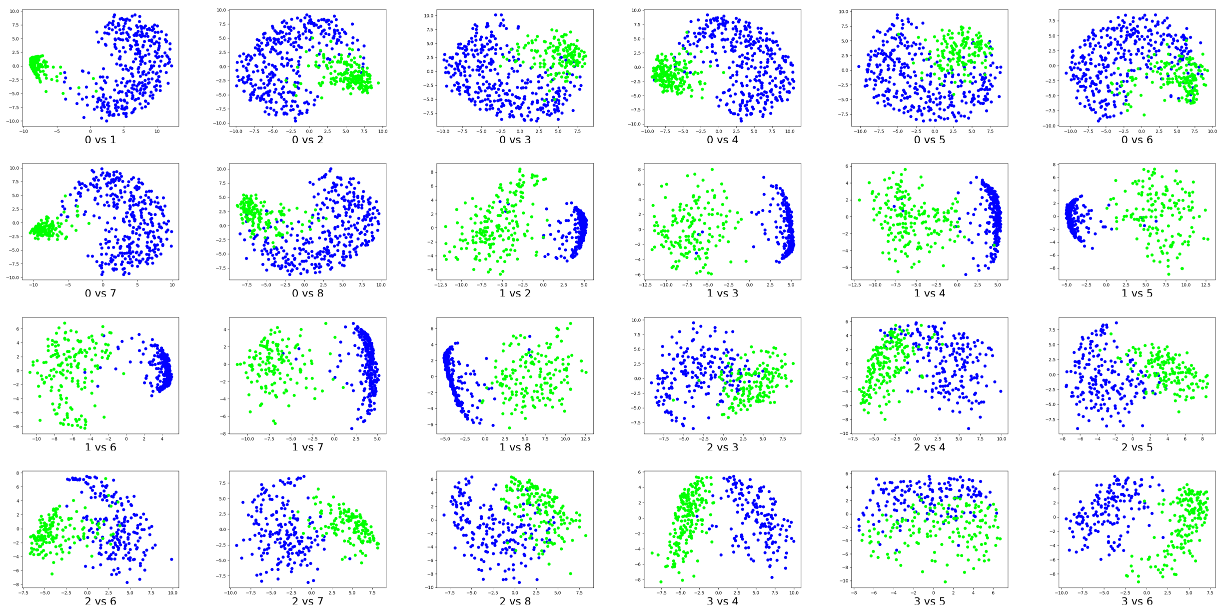
Dabei haben wir LDA von scikit-learn genommen. Wie man sieht, es gibt einige korrelierende Komponenten. Wenn man diese entfernt, kriegt man etwas bessere Ergebnisse. Wichtiger ist, dass man auch ziemlich gute Ergebnisse mit nur 30 Komponenten bekommen kann.

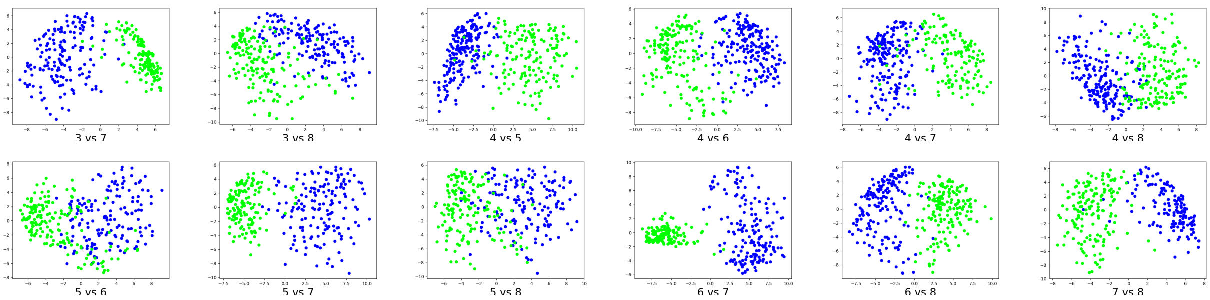
Wie viel Varianz man verlier, wenn man den Datensatz in k Dimensionen reduziert (k auf der x-Achse)



Auf dem Plot ist kein echten Ëlbogenßu sehen oder erst bei k=80, wobei das initiale Datensatz nur 256 Dimensionen hat.

2D Visualisierung von den einzelnen Cluster bei dem Zifferklassifizierungsproblem



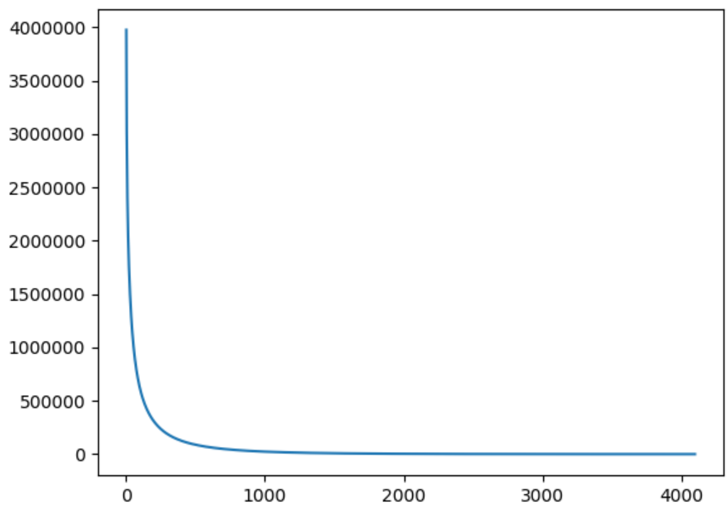


Wie man sieht, sind einige Ziffer linear von einander separierbar (z.B. 1 und 3) und andere gar schwer von einander unterscheidbar (in 2D Raum, 2 und 8). Man sieht, dass es auch einige gibt, die in den meisten Fällen gut unterscheidbar sind, aber haben auch einige Exemplare, die ganz ähnlich sind (z. B. 3 und 5, wenn man das obere Teil von je Ziffer nicht gut geschrieben hat ist das untere ja gleich).

Hier wurde aus dem ganzen Datensatz nur das jeweilige Paar genommen und die Dimensionen davon wurden reduziert. Wenn man aber PCA auf dem ganzen Datensatz anwendet, kriegt man nicht so gute Ergebnisse. Deswegen wurden wir vermuten, das man mit deutlich bessere Ergebnisse bekommen würde, wenn man ein binäres Klassifikator benutzt und damit zusammen PCA anwendet.

Eigenfaces

Wir haben PCA auf dem Datensatz von menschlichen Gescihten angewandt. Die dabei berechnete Hauptkomponenten (die Eigenvektoren der Kovarianzmatrix) nennt man Ëigenfaces”. Wie man auf dem nächsten Plot sieht, ist das Ëlbogen”, also das Punkt, ab dem man nicht viel Varianz mehr gewinnt bei weiteren Hauptkomponenten, erst bei Circa 320 Dimensionen. Die verlorene Varianz hört aber schon bei circa 30 echt drastisch zu sinken, deswegen haben wir 30 Hauptkomponenten genommen.



Hauptkomponenten des Gesichtsdatensatzes (a.k.a Eigenfaces)

Erklärung der PCA Implementierung

Fit Methode

Diese Methode berechnet unsere Hauptkomponenten und damit auch die Transformationsmatritze, mit der man ein Datensatz von n auf k Dimensionen reduziert. Hier wurde PCA genaus so wie in der Vorlesung auf eine höhe Abstraktionsebene implementiert, interessanter sind also eigentlich die Hilfsfunktionen.

```
    def fit(self, X):
        sorted_k_eig_vectors = PCA.get_sorted_eig_vec(X, self.k)
        self.principal_components = sorted_k_eig_vectors
        self.transformation_matrix = sorted_k_eig_vectors.T
```

Hilfsfunktionen

Wir berechnen die Kovarianzmatrix mit np.cov. Hier ist wichtig entweder das tronsponierte Datensatz zu benutzen, oder rowvar auf False zu setzen, damit man die Die Verhältnisse der Features im Datensatz und nicht der Samples.

Nachher können wir mit Hilfe von np.linalg.eigh die Eigenvektoren und iher entsprechende Eigenwerte berechnen. Hier ist ganz wichtig, dass die Eigenvektoren die Spalten der zurückgegebener Matrix sind. Das haben wir initial nicht verstanden und ziemlich viel Zeit deswegen beim Debugging verbracht.

Danach kann man leicht mit argsort() die Indizes bekommen, die die Eigenwerte sortieren und damit die Eigenvektoren sortieren und zurückgeben. Da die Eigenvektoren die Spalten sind, muss man hier aufpassen.

Die sortierte Eigenwerte brauchen wir für die Berechnung der verlorenen Kovarianz bei k Dimensionen.

```python
@staticmethod
    def get_eig_values_sort_args(eig_values, k=None):
        if k is None:
            k = len(eig_values)

        # reverse sorted, first k components
        return eig_values.argsort()[::-1][:k]

    @staticmethod
    def get_eig_vec_and_val(X):
        covariance_matrix = np.cov(X, rowvar=False)
        # the covariance matrix is always symmetric, we don't need to bother any further
        # therefore, we can also use eigh instead of eig
        return np.linalg.eigh(covariance_matrix)

    @staticmethod
    def get_sorted_eig_vec(X, k=None):
        eig_values, eig_vectors = PCA.get_eig_vec_and_val(X)
        sort_args = PCA.get_eig_values_sort_args(eig_values, k)

        return eig_vectors[:,sort_args].T

    @staticmethod
    def get_sorted_eig_values(X, k=None):
        eig_values, eig_vectors = PCA.get_eig_vec_and_val(X)
        # assert(np.all(eig_values >= 0))  # yep, all good
        sort_args = PCA.get_eig_values_sort_args(eig_values, k)

        return eig_values[sort_args]
```

Plotting der verlorenen Varianz bei k Dimensionen

Die Differenz von der Summe aller Eigenwerte mit der Summe von den benutzten (diese, der benutzten Hauptkomponenten) ist eine gute Möglichkeit zu plotten wie viel von der Varianz verloren wurde bei k Dimension. Ein Idealwert wäre also 0, wir wollen aber sehen, ab wie viele Dimensionen der Wert deutlich langsamer senkt.

```python
@staticmethod
    def plot_variance_for_k(X, save_plot_name=None):
        sorted_eig_values = PCA.get_sorted_eig_values(X)
        total_variance = sorted_eig_values.sum()
        ks = np.arange(2, len(X[0]), 1)

        variance_diffs = np.vectorize(lambda k: abs(total_variance - sorted_eig_values[:k].sum()))(ks)
        plt.plot(ks, variance_diffs)

        if save_plot_name is not None:
            plt.savefig(save_plot_name + '.png')
        else:
            plt.show()
```

Transform bzw. Fit-Transform Methoden

Wie unten erklärt, wenn man je x aus dem Datensatz mit je Hauptkomponente multipliziert bekommt man x in den reduzierten Raum.

Bsp: xi * (e1, e2, ..., ek) = xi representiert durch die k Hauptkomponenten

Wir speichern bei der Fit-Methode die Transformationsmatritze und nutzen diese bei Transform. So können wir dan weitere Daten erstmal transformieren, bevor wir Vorhersagen machen.

```
    def transform(self, X):
        '''
        (x11 x12 ... x1n )              (e11 e12 ... e1n)T       (x1 projected in subspace)
        (... ... ... ... )      X       (... ... ... ...)    =   (...                      )
        (xm1 xm2 ... xmn )              (ek1 ek2 ... ekn)        (xm projected in subspace)

        self.transformation_matrix is the already transponded matrix where each row is an eigenvector

        :param X: data set, each row represents a sample
        :return: X in the space defined by the first k eigenvectors of the fit data set
        '''

        return X.dot(self.transformation_matrix)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)
```

Vollständiges Code (auch für Plotting, Eigenfaces, Helpers usw.)

PCA.py

```
from Classifier import Classifier
import numpy as np
from matplotlib import pyplot as plt


class PCA(Classifier):
    def __init__(self, k):
        self.transformation_matrix = None
        self.k = k
        self.X_mean = None
        self.principal_components = None

    @staticmethod
    def get_eig_values_sort_args(eig_values, k=None):
        if k is None:
            k = len(eig_values)

        # reverse sorted, first k components
        return eig_values.argsort()[::-1][:k]

    @staticmethod
    def get_eig_vec_and_val(X):
        covariance_matrix = np.cov(X, rowvar=False)
        # the covariance matrix is always symmetric, we don't need to bother any further
        # therefore, we can also use eigh instead of eig
        return np.linalg.eigh(covariance_matrix)

    @staticmethod
    def get_sorted_eig_vec(X, k=None):
        eig_values, eig_vectors = PCA.get_eig_vec_and_val(X)
        sort_args = PCA.get_eig_values_sort_args(eig_values, k)

        return eig_vectors[:,sort_args].T

    @staticmethod
    def get_sorted_eig_values(X, k=None):
```

```
        eig_values, eig_vectors = PCA.get_eig_vec_and_val(X)
        # assert(np.all(eig_values >= 0))  # yep, all good
        sort_args = PCA.get_eig_values_sort_args(eig_values, k)

        return eig_values[sort_args]

    def fit(self, X):
        '''
        Finds the largest k eigenvalues and their corresponding eigenvectors from the
        covariance matrix of the dataset and saves the transformation matrix
        which can be used to project a data set from it's original space to
        the space defined by those eigenvectors and their corresponding eigenvalues
        '''

        sorted_k_eig_vectors = PCA.get_sorted_eig_vec(X, self.k)
        self.principal_components = sorted_k_eig_vectors
        self.transformation_matrix = sorted_k_eig_vectors.T

    def transform(self, X):
        '''
        (x11 x12 ... x1n )               (e11 e12 ... e1n)T       (x1 projected in subspace)
        (... ... ... ... )       X       (... ... ... ...)    =   (...                     )
        (xm1 xm2 ... xmn )               (ek1 ek2 ... ekn)        (xm projected in subspace)

        self.transformation_matrix is the already transponded matrix where each row is an eigenvector

        :param X: data set, each row represents a sample
        :return: X in the space defined by the first k eigenvectors of the fit data set
        '''

        return X.dot(self.transformation_matrix)

    def fit_transform(self, X):
        self.fit(X)
        return self.transform(X)

    @staticmethod
    def plot_variance_for_k(X, save_plot_name=None):
        sorted_eig_values = PCA.get_sorted_eig_values(X)
        total_variance = sorted_eig_values.sum()
        ks = np.arange(2, len(X[0]), 1)

        variance_diffs = np.vectorize(lambda k: abs(total_variance - sorted_eig_values[:k].sum()))(ks)

        plt.plot(ks, variance_diffs)

        if save_plot_name is not None:
            plt.savefig(save_plot_name + '.png')
        else:
            plt.show()
```

Helpers.py

```
import numpy as np
import math
import os
from matplotlib import pyplot as plt


def plot_multiple_images(title, images, data_set_dimensions, cols=5, fig_name=None):
    rows = math.ceil(len(images) / cols)
    plt.figure(figsize=(8,8))
    plt.suptitle(title, size=20)

    for i, component in enumerate(images):
        plt.subplot(rows, cols, i + 1)
```

```
        component += abs(component.min())
        component *= (1.0 / component.max())
        image = np.reshape(component, (data_set_dimensions, data_set_dimensions))
        plt.imshow(image, cmap=plt.cm.binary)
        plt.xticks(())
        plt.yticks(())

    if fig_name is not None:
        file_name = os.path.join(os.path.dirname(__file__), './{}'.format(fig_name))
        plt.savefig(file_name)
    else:
        plt.show()
```

Parser.py

```
import csv
import numpy as np
import pandas as pd
import os
from sklearn.model_selection import train_test_split


def parse_data(file_name, sep=' '):
    file_name = os.path.join(os.path.dirname(__file__), './Dataset/{}'.format(file_name))
    return pd.read_csv(file_name, header=None, na_filter=True, sep=sep).as_matrix()


def get_points_and_labels_from_data(data, label_idx=0):
    points = (np.array(data[:, 1:], dtype=np.float64) if label_idx == 0 else
              np.array(data[:, :label_idx], dtype=np.float64))
    labels = np.array(data[:,label_idx])

    return points, labels


def get_data_set(file_name, sep=' ', label_idx=0, seed=4):
    data = parse_data(file_name, sep)
    X, y = get_points_and_labels_from_data(data, label_idx)
    # for determined results we use a seed for random_state, so that data is always split
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, test_size=0.3,
                                                        random_state=seed)

    return X_train, X_test, y_train, y_test


def extract_classes_from_data_set(X, y, classes):
    labels_are_numbers = False

    try:
        a = int(y[0])
        labels_are_numbers = True
    except:
        pass

    is_from_classes = np.vectorize(lambda y: (int(y) if labels_are_numbers else y) in classes)
    filter_arr = is_from_classes(y)
    return X[filter_arr], y[filter_arr]
```

digits_classification.py

```
import numpy as np
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from PCA import PCA
from Parser import get_data_set
```

```python
X_train, X_test, y_train, y_test = get_data_set('digits.data')

pca_230 = PCA(230)
train_transformed_230 = pca_230.fit_transform(X_train)
test_transformed_230 = pca_230.transform(X_test)

pca_30 = PCA(30)
train_transformed_30 = pca_30.fit_transform(X_train)
test_transformed_30 = pca_30.transform(X_test)

PCA.plot_variance_for_k(X_train)

prediction_non_transformed = LDA().fit(X_train, y_train).predict(X_test)
score_non_transformed = np.mean(prediction_non_transformed == y_test)
print('Score LDA without PCA: {}'.format(score_non_transformed))

prediction_transformed_230 = LDA().fit(train_transformed_230, y_train).predict(
    test_transformed_230)
score_transformed_230 = np.mean(prediction_transformed_230 == y_test)
print('Score LDA with PCA, k={}: {}'.format(230, score_transformed_230))

prediction_transformed_30 = LDA().fit(train_transformed_30, y_train).predict(
    test_transformed_30)
score_transformed_30 = np.mean(prediction_transformed_30 == y_test)
print('Score LDA with PCA, k={}: {}'.format(30, score_transformed_30))
```

demo_digits_PCA.py

```python
from matplotlib import pyplot as plt
import numpy as np
from sklearn.decomposition import PCA as PCA_sklearn
from PCA import PCA
from Parser import parse_data, get_points_and_labels_from_data,
    extract_classes_from_data_set

data = parse_data('digits_test.data')
X, y = get_points_and_labels_from_data(data)

for a in range(10):
    for b in range(a+1, 10, 1):
        X_extracted, y_extracted = extract_classes_from_data_set(X, y, [a,b])
        X_transformed = PCA(2).fit_transform(X_extracted)
        X_a_idx = y_extracted.astype(int) == a
        X_a = X_transformed[X_a_idx]
        X_b = X_transformed[np.invert(X_a_idx)]

        plt.scatter(X_a[:,0], X_a[:,1], c='#0000FF')
        plt.scatter(X_b[:,0], X_b[:,1], c='#00FF00')
        plt.xlabel('{} vs {}'.format(a, b), fontsize=24)
        plt.savefig('figs/{}vs{}.png'.format(a,b))
        plt.clf()
```

eigenfaces.py

```python
from PCA import PCA
from Helpers import plot_multiple_images
import numpy as np
import math
import glob
import os
import cv2


def read_pgm(file_name):
    return cv2.imread(file_name, -1)

data_folder = os.path.abspath(os.path.join(os.path.dirname(__file__), './Dataset'))
```

```python
files = glob.glob(data_folder + '/lfwcrop_grey/faces/*.pgm')
data_set = []

for file in files:
    image = read_pgm(file)
    data_set.append(image.flatten())

data_set = np.array(data_set)

# drops quickly until about k=320
# but really quickly until about 30, plus we can't really submit 320 eigenfaces for this
    homework...
# PCA.plot_variance_for_k(data_set, save_plot_name='eigenfaces_variance_for_k')


pca_eigenfaces = PCA(30)
pca_eigenfaces.fit(data_set)
data_set_dimensions = int(math.sqrt(len(data_set[0])))
principal_components = pca_eigenfaces.principal_components

plot_multiple_images('Eigenfaces', principal_components, data_set_dimensions, fig_name='
    eigenfaces.png')
```