Codebook Final project hard-coded PCA

Fabio Setti

2023-06-07

Machine learning class final project. The PCA code implements regularized PCA with gradient descent as the parameter estimation method (PCA algorithm only utilizes no python libraries aside from numpy). DISCLAIMER: I am not a computer science student (self-thought), so I realize that much of the code is probably not as optimized as it could be.

Important: all of the functions and code referenced below can be found in the *python code* folder at https://github.com/quinix45/Hard-coded-regularized-PCA/tree/main/python%20code

Replicate Project Results

To replicate the results presented in the final project, follow these steps:

Step 1: Run the `Functions_Project_PCA.py` file to define the 5 functions used in the project. The uses, arguments, and outputs of the functions are elaborated upon in the *Functions References* section.

Step 2: Get results from Table 1 by running the `Results_Table 1.py` file. The file should take approximately from 1 minute to 1 minute and 30 seconds to run. The results of Table 1 will be saved as 2D numpy array called `Table_1` and printed. (NOTE: no random state was set for Table 1, so the results may differ slightly depending on the training split)

Step 3: Get results from Table 2 by running the `Results_Table 2.py` file. The file should take approximately from 1 minute to 1 minute and 30 seconds to run. The results of Table 1 will be saved as 2D numpy array called `Table_2` and printed.

More detailed notes for each line of code are present in the respective `py` files.

Functions References

More detailed comments about functions are provided here.

Function 1: `Standardizer(...)`

`standardizer(data)`

```python
def standardizer(data):

    d = data

    # calculate feature means
    means = sum(data)/data.shape[0]
```

```
    # calculate feature Standard deviations (uses population sigma)
    sds = np.sqrt((sum((data-(sum(data)/ data.shape[0]))**2))/(data.shape[0]))

    # return (x_i - mu_x)/sigma_x
    return (d - means)/(sds)
```

Description

Returns input data in standardized (normalized) form with $\mu = 0$ and $\sigma = 1$.

Arguments

- `data`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points.

Output

The function returns a single object:

- Returns 2D numpy array where all features (output[:, $feature_i$]) have mean, $\mu = 0$, and standard deviation, $\sigma = 1$.

Example

```
import numpy as np

# load data ans store it as X
from sklearn import datasets

X =  datasets.load_breast_cancer().data


norm_X = standardizer(X)

# check that sum of data points is 0 for each feature
np.round(sum(norm_X), 3)
```

```
## array([-0., -0., -0., -0.,  0., -0., -0.,  0., -0., -0., -0., -0.,  0.,
##        -0., -0., -0.,  0., -0., -0., -0., -0.,  0., -0.,  0., -0., -0.,
##         0., -0., -0.,  0.])
```

Function 2: `SSE(...)`

`SSE(dat1, dat2)`

```
def SSE (dat1, dat2):

   # return Sum of squared error between two arrays
   return sum(sum(((dat1) - (dat2))**2))
```

Description

Returns the Sum of squared error ($\Sigma(X_1 - X_2)^2$) between two matrices/data sets with the same dimensions.

Arguments

- `dat1`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points.
- `dat2`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points.

Output

The function returns a single object:

- Scalar value of SSE between 2 input2D numpy arrays.

Example

```
import numpy as np
# load data ans store it as X
from sklearn import datasets

X =  datasets.load_breast_cancer().data

# split data evenly

X1 = X[:, 0:int(X.shape[1]/2)]
X2 = X[:, int(X.shape[1]/2):int(X.shape[1])]

# calculate SSE between 2 parts of the data (not normalized)

SSE(X1, X2)
```

```
## 954840511.9709946
```

Function 3: `compute_u(...)`

```
def compute_u(dat, iter, lam):

        x = standardizer(dat)

        def descent (x, u, ind1, ind2):
                id = np.ones(x.shape[1])*-1
                # implementation of the gradient descent of reconstruction error formula
                id[ind1] = 2
                return 2*np.dot(u*id, x[ind2]) * (x[ind2,ind1] - u[ind1]*(np.dot(u,
                ↪  x[ind2]))) + 2*lam*u[ind1]

        # starting value of u and learning rate
        u = np.ones(dat.shape[1])
        u_prev = np.zeros(dat.shape[1])
        epsilon = .1/dat.shape[0]

        # find components
        for j in range(iter+1):
                        for n in range(dat.shape[0]):
                                for i in range(dat.shape[1]):
                                        u_prev[i] = u[i]
                                        u[i] = u[i] - descent(x, u, i, n)*epsilon
↪


        Us = np.zeros((x.shape[0], x.shape[1])) + u
        # reconstructed data
        rec_dat = np.dot(x,u)*Us.T
        # difference between input data and reconstructed data
        dif_dat = x.T - rec_dat

        return u, rec_dat, dif_dat
```

Description

Finds vector u^q (principal component) that minimizes project report's equation 1 with L_2 regularization, $RE = \sum_i \sum_j (x_i^j - (u^{qT} x_i) u^q)^2 + \lambda (u^q)^2$. Note that the data is normalized inside the function through the `standardizer()` function and can be inputted without pre-processing.

Arguments

- `dat`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points.
- `iter`: Maximum number of iterations to perform. At each iteration, each element of u^q is updated a number of times equal to the data points in the input data.
- `lam`: Value of the λ parameter for L_2 regularization.

Output

The function returns 3 objects:

1. 1D numpy array containing Principal component u^q that minimizes $\sum_i \sum_j (xi^j - (u^{qT}x^i)u^q)^2 + \lambda(u^q)^2$ given input data.
2. 2D numpy array representing reconstructed data $\tilde{X} = (u^{qT}x^i)u^q$. Note that the dimensions of this object are the transposed dimensions of the original data.
3. 2D numpy array representing the difference between the input data set and the reconstructed data, $X - \tilde{X}$. Note that the dimensions of this object are the transposed dimensions of the original data.

Example

```
import numpy as np
# load data ans store it as X
from sklearn import datasets

X =  datasets.load_breast_cancer().data

# get 1st component, recreated data, and the difference between original and recreated
↪  data

u, rec_data, dif_data = compute_u(X, 5, lam = 0)

print(X.shape)
```

```
## (569, 30)
```

```
print(u.shape)

# data is return as the transpose to the original data
```

```
## (30,)
```

```
print(rec_data.shape)
```

```
## (30, 569)
```

```
print(dif_data.shape)
```

```
## (30, 569)
```

Function 4: `optimal_component(...)`

```
def optimal_component(dat, max_iter = 50, max_comp = 10, SSE_ratio = .05, lam = 0):

   # give an arbitrary SSE value to start loop
   # starting data
   x = dat
   x_org = standardizer(x)
   rec_data_tot = np.zeros((x.shape[1],x.shape[0]))
   # empty array of components
   components = np.array([])
   #empty array of SSEs
   SSEs = np.array([])

  # loop compute_u to find components up to max_comp value
   for i in range(max_comp):

       u1, rec_data, new_x = compute_u(x, max_iter, lam)
       # recompute recreated data for each iteration
       rec_data_tot =  rec_data_tot + rec_data
       # calculate SSE with new component
       new_SSE = SSE(x_org.T, rec_data_tot)
       # calculate SSE without new component
       prev_SSE = SSE(x_org.T, rec_data_tot - rec_data)

       # decide if continue extracting components if delta RE ratios < 1 - SSE_ratio
       ↪  value (.5 default)
       if (new_SSE/prev_SSE) < 1 - SSE_ratio:

        # append delaRE to SSEs
         SSEs = np.append(SSEs, (new_SSE/prev_SSE))
        # append compnent to components
         components = np.append(components, u1)
        # redefine x to feed back to compute_u
         x = new_x.T
        # reshape components if max_comp iter is reached
         if i == max_comp - 1:
          components = components.reshape(i+1, len(u1))
        # break loop if delta RE ratios >= 1 - SSE_ratio
       else:
          components = components.reshape(i, len(u1))
          x_org = x
          break

   return components, (1 - SSEs)
```

Description

Extracts either a given number of principal components from input data or optimal number of principal components based on specified value of final project's equation 10, $\Delta RE = 1 - \frac{RE_{u^{q+1}}}{RE_{u^q}}$. This function iterates over function 4, `compute_u()`, to find multiple principal components.

Arguments

- `dat`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points.
- `max_iter`: Maximum number of iterations to perform. At each iteration, each element of u^q is updated a number of times equal to the data points in the input data. Default `max_iter` = 50.
- `max_comp`: The maximum number of components to be extracted if the specified `SSE_ratio` is not reached (see `SSE_ratio` argument). default `max_comp` = 10.
- `SSE_ratio`: The stopping rule for component extraction. When the improvement of reconstruction error (see *Optimal Number of Components* in project report) is lower than the specified value, components extraction is halted and the previously extracted components are return. Default value is `SSE_ratio` = .05.
- `lam`: Value of the λ parameter for L_2 regularization. Default value is `lam` = 0, meaning that no regularization happens by defaults.

Output

The function returns 2 objects:

1. 2D numpy array containing the extracted principal components, where dimension 0 represents different components and dimension 1 represent the feature weights for each component.

2. 1D numpy array containing all the ΔREs. The length of this array will equal the number of extracted components.

Example

```
import numpy as np
# load data ans store it as X
from sklearn import datasets

X =  datasets.load_breast_cancer().data

# get 1st component, recreated data, and the diff between original and recreated data

components, delta_REs = optimal_component(X,
                                          max_iter = 10,
                                          max_comp = 6,
                                          SSE_ratio = .05,
                                          lam = .5)

#shape of components 2D array
print(components.shape)

# Delta REs
```

```
## (4, 30)
```

```
print(delta_REs)
```

```
## [0.43127632 0.2796304  0.15190861 0.06599077]
```

Here, for example, the function stopped before extracting the 5_{th} component as the threshold $\Delta RE = .05$ was reached at the 5_{th} component.

Function 5: `comp_feature(...)`

```
def comp_feature(org_data, components, features):

  x = org_data
  u = components
  comp_features = []

  for i in range(features):
    # append to u
    comp_features = np.append(comp_features, np.dot(x, u[i]))
    # calculate remaining data
    Us = np.zeros((x.shape[0], x.shape[1])) + u[i]
    rec_dat = np.dot(x,u[i])*Us.T
    dif_dat = x.T - rec_dat
    x = dif_dat.T

  return comp_features.reshape(features,x.shape[0]).T
```

Description

Function to reconstruct features based on component weights. Used specifically to conveniently get results for Table 2 in the project report.

Arguments

- `org_dat`: 2D numpy array where dimension 0 represents data points and dimension 1 represents features of data points. The original data from which components were extracted.
- `components`: components output from function 4, `optimal_component()`.
- `features`: number of features to be reconstructed. It can have a maximum length of `components.shape[0]`, meaning that only features up to the number of total extracted components can be recreated.

Output

- 2D array with dimension 0 = `org_dat.shape[0]`, and dimension 1 equals to `features` argument (total number of different principal components).

Example

see `Results_Table2.py` file for how the function is used.