PCA

April 1, 2021

If you plan to run the assignment locally: You can download the assignments and run them locally, but please be aware that as much as we would like our code to be universal, computer platform differences may lead to incorrectly reported errors even on correct solutions. Therefore, we encourage you to validate your solution in Coursera whenever this may be happening. If you decide to run the assignment locally, please: 1. Try to download the necessary data files from your home directory one at a time, 2. Don't update anything other than this Jupyter notebook back to Coursera's servers, and 3. Make sure this notebook maintains its original name after you upload it back to Coursera.

Note: You need to submit the assignment to be graded, and passing the validation button's test does not grade the assignment. The validation button's functionality is exactly the same as running all cells.

[1]:
```
%matplotlib inline
%load_ext autoreload
%autoreload 2

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import pandas as pd
import time
import os
from sklearn.decomposition import TruncatedSVD

from aml_utils import test_case_checker, perform_computation
```

0.1 Attention:

This assignment is computationally heavy, and inefficient implementations may not pass the autograding even if they technically produce the correct results. To avoid this, make sure you read and understand all the instructions before starting to implement the tasks. Failure to follow the instructions closely will most likely cause timeouts.

It is **your responsibility** to make sure your implementation is not only **correct**, but also as **efficient** as possible. If you follow all the instructions provided, you should be able to have all the cells evaluated in under 10 minutes.

1 *Assignment Summary

CIFAR-10 is a dataset of 32x32 images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is often used to evaluate machine learning algorithms. You can download this dataset from https://www.cs.toronto.edu/~kriz/cifar.html.

- For each category, compute the mean image and the first 20 principal components. Plot the error resulting from representing the images of each category using the first 20 principal components against the category.
- Compute the distances between mean images for each pair of classes. Use principal coordinate analysis to make a 2D map of the means of each categories. For this exercise, compute distances by thinking of the images as vectors.
- Here is another measure of the similarity of two classes. For class A and class B, define E(A | B) to be the average error obtained by representing all the images of class A using the mean of class A and the first 20 principal components of class B. Now define the similarity between classes to be (1/2)(E(A | B) + E(B | A)). If A and B are very similar, then this error should be small, because A's principal components should be good at representing B. But if they are very different, then A's principal components should represent B poorly. In turn, the similarity measure should be big. Use principal coordinate analysis to make a 2D map of the classes. Compare this map to the map in the previous exercise? are they different? why?

References:

- Textbook section 5.1 https://link.springer.com/chapter/10.1007/978-3-030-18114-7_5

2 0. Data

2.1 0.1 Description

CIFAR-10 is a dataset of 32x32 images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is often used to evaluate machine learning algorithms. You can download this dataset from https://www.cs.toronto.edu/~kriz/cifar.html.

2.2 0.2 Information Summary

- **Input/Output**: This data has a set of 32 pixel rows, 32 pixel columns, and 3 color channels. Therefore, each single image, is vectorized, will consist of $32 \times 32 \times 3$ elements (i.e., each image has 3072 dimensions). There are a total of 60000 samples labelled from 10 class. The data set is balanced with each class having exactly 6000 samples.

- **Missing Data**: There is no missing data.

- **Final Goal**: We want to understand the data using multi-dimensional scaling methods.

2.3 0.3 Loading The Data

If you are curious how the original data was obtained, we used the torchvision API to download and pre-process it. The ready-to-use data is stored in numpy format for easier access.

[2]:
```
if os.path.exists('../PCA-lib/cifar10.npz'):
    np_file = np.load('../PCA-lib/cifar10.npz')
    train_images_raw = np_file['train_images_raw']
    train_labels = np_file['train_labels']
    eval_images_raw = np_file['eval_images_raw']
    eval_labels = np_file['eval_labels']

else:
    import torchvision
    download_ = not os.path.exists('../PCA-lib/cifar10/')
    data_train = torchvision.datasets.CIFAR10('../PCA-lib/cifar10', train=True,␣
 ↪transform=None, target_transform=None, download=download_)
    data_eval = torchvision.datasets.CIFAR10('../PCA-lib/cifar10', train=False,␣
 ↪transform=None, target_transform=None, download=download_)
    train_images_raw = data_train.data
    train_labels = np.array(data_train.targets)
    eval_images_raw = data_eval.data
    eval_labels = np.array(data_eval.targets)
    np.savez('../PCA-lib/cifar10.npz', train_images_raw=train_images_raw,␣
 ↪train_labels=train_labels,
             eval_images_raw=eval_images_raw, eval_labels=eval_labels)
```

[3]:
```
class_to_idx = {'airplane': 0,
                'automobile': 1,
                'bird': 2,
                'cat': 3,
                'deer': 4,
                'dog': 5,
                'frog': 6,
                'horse': 7,
                'ship': 8,
                'truck': 9}
```

[4]:
```
images_raw = np.concatenate([train_images_raw, eval_images_raw], axis=0)
labels = np.concatenate([train_labels, eval_labels], axis=0)
images_raw.shape, labels.shape
```

[4]:
```
((60000, 32, 32, 3), (60000,))
```

3 1. Principal Component Analysis

0. Let's say we have Data Matrix X with N rows (i.e., data points) and d columns (i.e., features).

$$X = [\cdots]_{N\times d}$$

1. Let's perform SVD on the X.

$$X = U_x S_x V_x^T$$

Let's assume $N > d$ (We have 6000 data points per class, which is more than the 3072 dimenstions).

By the way SVD works, we should have

$$U_x = [\cdots]_{N\times d}$$

$$S_x = [\cdots]_{d\times d}$$

$$V_x = [\cdots]_{d\times d}$$

and

$$U_x^T U_x = I_{d\times d}$$

$$V_x^T V_x = I_{d\times d}$$

2. The textbook says we need the following decomposition for the covariance matrix Σ:

$$\Sigma \mathcal{U} = \mathcal{U}\Lambda$$

3. We assume that X has mean zero (i.e., we already subtracted the feature averages). If X has N rows (i.e., data items), we have
$$\Sigma = \frac{1}{N} X^T X$$
4. Let's find Σ in terms of U_x, S_x, and V_x

$$\Sigma = \frac{1}{N} X^T X = \frac{1}{N} V_x S_x U_x^T U_x S_x V_x^T = V_x \frac{S_x^2}{N} V_x^T$$

$$\Rightarrow \Sigma V_x = V_x \frac{S_x^2}{N}$$

5. By comparison, we have
$$\mathcal{U} = V_x$$
$$\Lambda = \frac{S_x^2}{N}$$

3.0.1 Considering the above:

1. **There is no need to compute the covariance matrix** Σ and then find its diagonalization; You can easily perform SVD on the data matrix X, and get what you need!
2. In fact, you do not even need the matrices V_x and U_x for computing the mean squared error; You can infer the mean squared error using only the S_x matrix.
    - Numpy's SVD function `np.linalg.svd` has an argument `compute_uv` that turns off returning the U and V matrices for better efficiency. Therefore, you may be able to save some runtime in large data sets if you only care about the mean squared error!

4 Task 1

Write a function `pca_mse` that takes two arguments as input

1. `data_raw`: a numpy array with the shape $(N, \cdots)$, where N is the number of samples, and there may be many excess dimensions denoted by $\cdots$. You will have to reshape this input `data_raw` matrix to obtain a shape of (N, d), where d is the vectorized data's dimension. For example, `data_raw` could have an input shape of `(6000, 50, 50, 3)`. In this case you will have to reshape the input data to have a shape of `(6000, 7500)`.

2. `num_components`: This is the number of PCA components that we want to retain. This variable is denoted by r in the PCA definition in the textbook.

and returns the variable `mse` which is the mean squared error of the PCA projection into the designated number of principal components.

Important Note: Make sure you use `np.linalg.svd` for the SVD operation. Do not use any other matrix factorization function for this question (such as `np.linalg.eig`).

Important Note: Make sure you read and understand the notes from the previous cells before you start implementing. Failing to properly set the arguments for `np.linalg.svd` or trying to find the mean squared error by calculating the reconstruction error may cause extreme delays and timeouts for your implementation.

Hint: If you don't know how to extract the mean squared error of the PCA projection, or don't have a fresh probability and statistics memory, take a look at the Principal Component Analysis chapter of the most recent version of the textbook; the subsection titled "The error in a low-dimensional representation" explains how to find the mean squared error of the PCA projection as a function of the eigenvalues that were dropped.

[5]:
```python
def pca_mse(data_raw, num_components=20):

    # your code here
    temp = np.shape(data_raw)
    N = temp[0]
    x = np.reshape(data_raw,(N,-1)) - np.mean(np.
↪reshape(data_raw,(N,-1)),axis=0)
    U,S,VT = np.linalg.svd(x)
    v = VT.T
    s = np.diag(S)
    h = np.dot(s[:num_components, :num_components], v[:, :num_components].T)
    m = (x - np.dot(U[:, :num_components], h))**2
    mse = np.sum(np.mean(m,axis=0))

    assert np.isscalar(mse)
    return np.float64(mse)
```

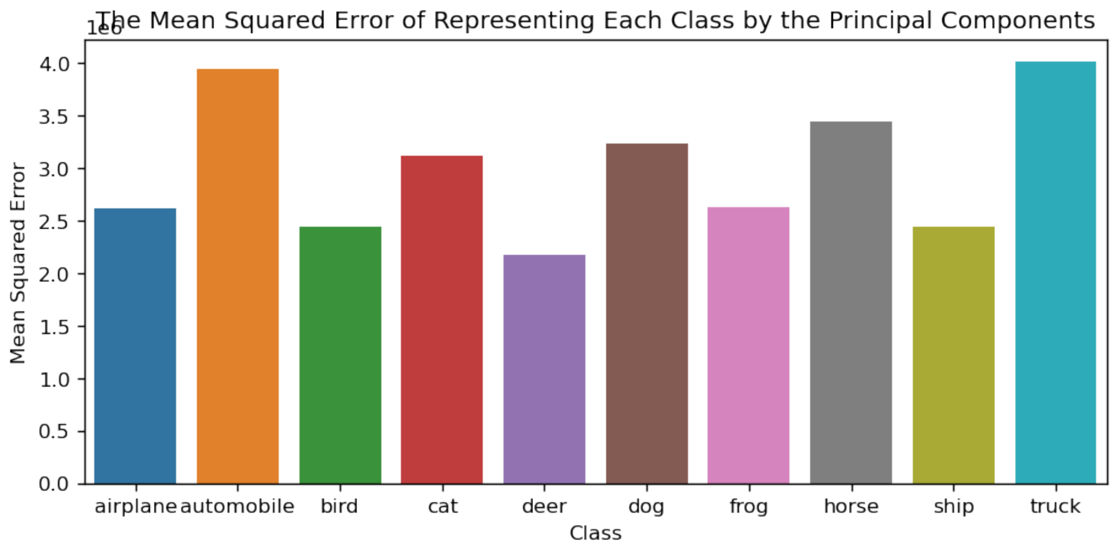
[6]:
```python
# Performing sanity checks on your implementation
some_data = (np.arange(35).reshape(5,7) ** 13) % 20
some_mse = pca_mse(some_data, num_components=2)
assert some_mse.round(3) == 37.903

# Checking against the pre-computed test database
test_results = test_case_checker(pca_mse, task_id=1)
assert test_results['passed'], test_results['message']
```

[7]:
```python
#Task 1 Test Cell
```

[8]:
```python
if perform_computation:
    class_names = []
    class_mses = []
    for cls_name, cls_label in class_to_idx.items():
        data_raw = images_raw[labels == cls_label,:,:,:]
        start_time = time.time()
        print(f'Processing Class {cls_name}', end='')
        cls_mse = pca_mse(data_raw, num_components=20)
        print(f' (The SVD operation took %.3f seconds)' % (time.
 ↪time()-start_time))
        class_names.append(cls_name)
        class_mses.append(cls_mse)
```

```
Processing Class airplane (The SVD operation took 16.170 seconds)
Processing Class automobile (The SVD operation took 16.174 seconds)
Processing Class bird (The SVD operation took 16.201 seconds)
Processing Class cat (The SVD operation took 16.280 seconds)
Processing Class deer (The SVD operation took 16.290 seconds)
Processing Class dog (The SVD operation took 16.168 seconds)
Processing Class frog (The SVD operation took 16.059 seconds)
Processing Class horse (The SVD operation took 16.118 seconds)
Processing Class ship (The SVD operation took 16.149 seconds)
Processing Class truck (The SVD operation took 16.068 seconds)
```

[9]:
```python
if perform_computation:
    fig, ax = plt.subplots(figsize=(9,4.), dpi=120)
    sns.barplot(class_names, class_mses, ax=ax)
    ax.set_title('The Mean Squared Error of Representing Each Class by the␣
 ↪Principal Components')
    ax.set_xlabel('Class')
    _ = ax.set_ylabel('Mean Squared Error')
```



5 2. Principal Coordinate Analysis

```
[10]: class_mean_list = []
      for cls_label in sorted(class_to_idx.values()):
          data_raw = images_raw[labels == cls_label,:,:,:]
          class_mean = np.mean(data_raw, axis=0).reshape(1,-1)
          class_mean_list.append(class_mean)
      class_means = np.concatenate(class_mean_list, axis=0)
```

6 Task 2

Write a function `mean_image_squared_distances` that takes the matrix `class_means` as an input and return the `SquaredDistances` matrix as output.

`class_means` is a numpy array like a traditional data matrix; it has a shape of (N, d) where there are N individual data-points where each is stored in a single d dimensional row. (N, d) could be anything, so do not make assumptions about it.

Your job is to produce the numpy array `SquaredDistances` whose i^{th} row and j^{th} column is the **squared** Euclidean distance between the i^{th} row of `class_means` and j^{th} row of `class_means`. Obviously * The diagonal elements should be zero. * The `SquaredDistances` should be symmetric.

```
[15]: def mean_image_squared_distances(class_means):

          # your code here
          N = np.shape(class_means)[0]
          SquaredDistances = np.zeros((N,N))
          for i in range(N):
              for j in range(N):
                  SquaredDistances[i,j] = np.sum(np.square(class_means[i,:
       ↪]-class_means[j,:]))

          return SquaredDistances
```

```
[16]: #Performing sanity checks
      some_data = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
      some_dist = mean_image_squared_distances(some_data)
      assert np.array_equal(some_dist.round(3), np.array([[ 0.   ,  4.551, 18.204,  8.
       ↪306, 14.041],
                                                          [ 4.551,  0.   , 12.714,  3.
       ↪918, 12.551],
                                                          [18.204, 12.714,  0.   ,  8.
       ↪633,  8.735],
                                                          [ 8.306,  3.918,  8.633,  0.
      ↪    ,  7.49 ],
                                                          [14.041, 12.551,  8.735,  7.
      ↪49 ,  0.    ]]))

      # Checking against the pre-computed test database
      test_results = test_case_checker(mean_image_squared_distances, task_id=2)
      assert test_results['passed'], test_results['message']
```

[17]:
```
# Task 2 Test Cell
```

7 Task 3

Read and implement the Principal Coordinate Analysis procedure from your textbook by writing the function `PCoA` which takes the following arguments: 1. `SquaredDistances`: A numpy array which is square in shape, symmetric, and is the square of a distance matrix of some unknown set of points. The output of the `mean_image_squared_distances` function you wrote previously will be fed as this argument.

2. `r`: This is the dimension of the visualization space, and corresponds to the same r variable in the textbook procedure.

Things to keep in mind: 1. There is an erratum in the textbook's description of the PCoA procedure. There is a missing negative sign when computing the matrix $\mathcal{W}$; the correct definition of $\mathcal{W}$ is $\mathcal{W} := -\frac{1}{2}\mathcal{A}\mathcal{D}^{(2)}\mathcal{A}^T$. 2. It is **vital** to make sure that eigenvalues are sorted as the textbook mentioned, and the eigenvectors are also ordered accordingly. Some decomposition functions such as `numpy`'s `np.linalg.eig` do not guarantee to return the eigenvalues and eigenvectors in any sorted way, and `np.linalg.eigh` guarantees to return them in ascending order; you will have to make sure they are sorted as the textbook says.

Note: You should only use `np.linalg.eigh` for matrix factorization in this question since we're dealing with a symmetric matrix; do not use `np.linalg.eig`, `np.linalg.svd`, or any other matrix decomposition function in this question.

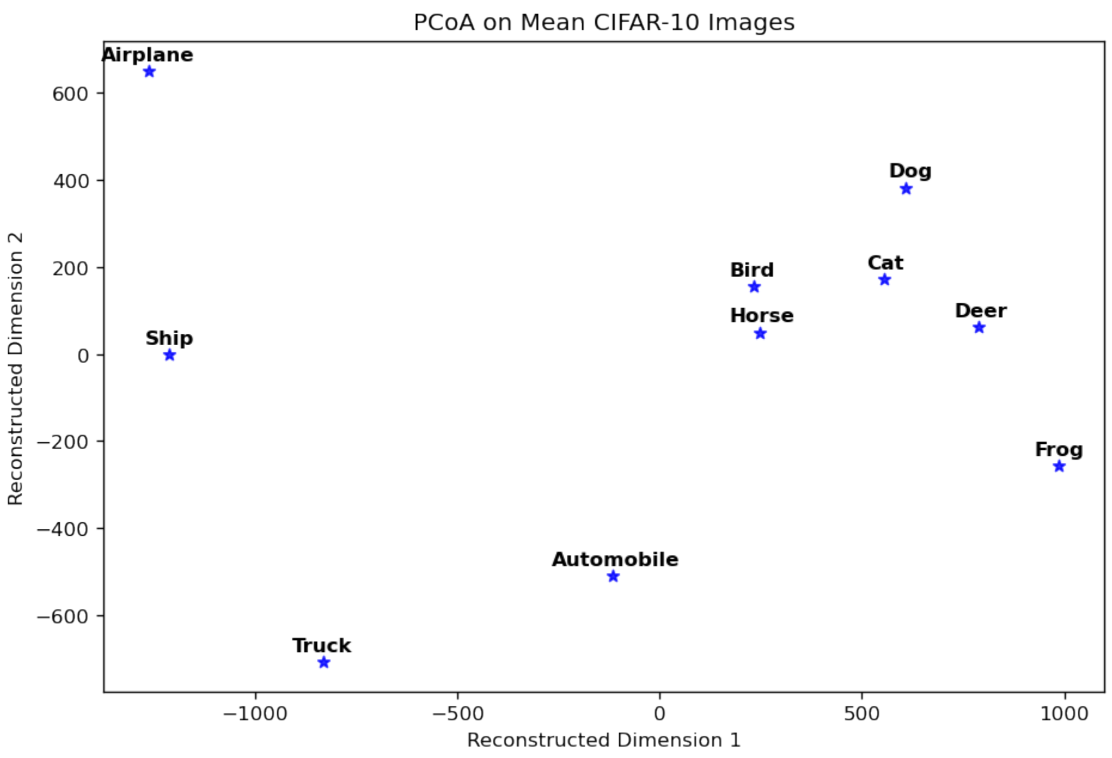
[18]:
```
def PCoA(SquaredDistances, r=2):
    assert SquaredDistances.shape[0] == SquaredDistances.shape[1]
    num_points = SquaredDistances.shape[0]

    # your code here
    ones = np.ones((num_points,1))
    a = np.identity(num_points) - np.matmul(ones,ones.T)/num_points
    w = -np.matmul(np.matmul(a,SquaredDistances),a.T)/2
    eigval, eigvec = np.linalg.eigh(w)
    e1 = np.flip(eigval,axis=0)
    e2 = np.flip(eigvec,axis=1)
    s = np.sqrt(np.diag(e1[:r]))
    VT = np.matmul(e2[:, :r],s)
    assert VT.shape[0] == num_points
    assert VT.shape[1] == r
    return VT
```

[19]:
```
some_data = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
some_dist = mean_image_squared_distances(some_data)
some_pcoa = PCoA(some_dist, r=2)
assert np.array_equal(some_pcoa.round(3), np.array([[-1.974,  0.421],
                                                     [-1.285, -0.646],
                                                     [ 1.98 , -1.137],
                                                     [-0.091, -0.266],
                                                     [ 1.369,  1.628]]))

# Checking against the pre-computed test database
test_results = test_case_checker(lambda *args, **kwargs: PCoA(*args, **kwargs).
 ↪astype(np.complex128), task_id=3)
assert test_results['passed'], test_results['message']
```

[20]:
```
#Task 3 Test Cell
```

[21]:
```
if perform_computation:
    SquaredDistances = mean_image_squared_distances(class_means)
    VT = PCoA(SquaredDistances, r=2)
```

[22]:
```
if perform_computation:
    class_names_list = sorted(list(class_to_idx.keys()))
    fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
    x_components = VT[:,0]
    y_components = VT[:,1]
    sns.regplot(x=x_components, y=y_components, fit_reg=False, marker="*",␣
 ↪color="Blue", ax=ax)
    for class_idx in range(VT.shape[0]):
        num_letters = len(class_names_list[class_idx])
        ax.text(x_components[class_idx]-num_letters*15,␣
 ↪y_components[class_idx]+25,
                class_names_list[class_idx].capitalize(),
                horizontalalignment='left', size='medium', color='black',␣
 ↪weight='semibold')
    ax.set_xlabel('Reconstructed Dimension 1')
    ax.set_ylabel('Reconstructed Dimension 2')
    _ = ax.set_title('PCoA on Mean CIFAR-10 Images')
```



8 3. Generalized PCoA with Non-Metric Similarities

9 Task 4

Write a function `principal_components_precise_svd` that returns the principal components of a data matrix and takes the following arguments as input

1. `data_raw`: a numpy array with the shape $(N, \cdots)$, where N is the number of samples, and there may be many excess dimensions denoted by $\cdots$. You will have to reshape this input `data_raw` matrix to obtain a shape of (N, d), where d is the vectorized data's dimension. For example, `data_raw` could have an input shape of `(6000, 50, 50, 3)`. In this case you will have to reshape the input data to have a shape of `(6000, 7500)`.
2. `num_components`: This is the number of PCA components that we want to retain. This variable is denoted by r in the PCA definition in the textbook.

and returns the variable `V_x` which is a numpy array with the shape `(d, num_components)`. The columns are the unitay principal components sorted descendingly with respect to the eigenvalues.

Important Note: **Do not** try to recover the covariance matrix Σ and then find its eigenvalues. This can prove to be both inefficient and unnecessary. As the theoretical review before the first task concluded, **There is no need to compute the covariance matrix** Σ. Instead, all you need to do is to find the SVD of the data matrix, and extract the principal components from it.

Important Note: Do not use any matrix factorization function other than `np.linalg.svd` for this task; incorporating any other matrix factorization function (such as `np.linalg.eig`) may not be compatible with the results we expect and may even be inefficient.

[23]:
```
def principal_components_precise_svd(data_raw, num_components=20):
    # your code here
    N = np.shape(data_raw)[0]
    dr = data_raw.reshape(N,-1)
    U,S,VT = np.linalg.svd(dr - np.mean(dr,axis=0))
    V_x = VT.T[:, :num_components]

    assert V_x.ndim==2
    # Don't mind/change the following lines.
    # This is for mitigating the ambiguity up to -/+1 factor in PCs.
    # (i.e., if x is a unitary PC, then -x is also a unitary PC).
    # We multiply each column by the sign of the largest element (in absolute␣
↪value) of that column
    sign_unambiguity = np.sign(V_x[np.abs(V_x).argmax(axis=0), np.arange(V_x.
↪shape[1])]).reshape(1,-1)
    V_x *= sign_unambiguity
    return V_x
```

[24]:
```
some_data = (np.arange(35).reshape(5,7) ** 13) % 20
some_pcs = principal_components_precise_svd(some_data, num_components=2)
assert np.array_equal(some_pcs.round(3), np.array([[-0.123, -0.114],
                                                    [-0.43 ,  0.119],
                                                    [-0.021,  0.41 ],
                                                    [-0.603, -0.164],
                                                    [ 0.084,  0.491],
                                                    [-0.223,  0.724],
                                                    [ 0.616,  0.109]]))

# Checking against the pre-computed test database
test_results = test_case_checker(principal_components_precise_svd, task_id=4)
assert test_results['passed'], test_results['message']
```

[25]:
```
#Task 4 Test Cell
```

The following cell will run your `principal_components_precise_svd` function on a single class of data, and provide you with some running time estimate.

[26]:
```
if perform_computation:
    first_class_features = images_raw[labels == 0, :, :, :]

    starting_time = time.time()
    first_class_pcs = principal_components_precise_svd(first_class_features,␣
↪num_components=20)
    end_time = time.time()
    print('Finding the principal components on a single class took %.3f seconds.
↪'%(end_time-starting_time))
```

```
Finding the principal components on a single class took 15.999 seconds.
```

Although, this performance is extremely hardware-dependent, it's certainly not negligible. Keep in mind that we will have to run this function about 100 times on data of the same size to construct a similarity matrix in later tasks; any speedup may very well be appreciated.

Most of the computation time in the previous task was spent on the SVD factorization. Essentially, we found all the singular values and directions, ignored most of them, and only kept the top 20. This can be a very good place to start saving on computation; if only there was an SVD variant which you could tell in advance that you're only interested in the top 20 components, so that it wouldn't waste your time computing non-important singular values and directions…

Spoiler Alert: Such an efficient SVD variant exists, and sometimes is referred to as the "Truncated SVD" in application. Next task will be a redo of the previous task using this fast factroization.

10 Task 5

Similar to `principal_components_precise_svd`, write a function `principal_components` that uses `scikit-learn`'s `TruncatedSVD` decomposition instead of the precise `np.linalg.svd` decomposition that was used in the previous task. As in the previous task, `principal_components` should return the principal components of a data matrix and take the following arguments as input

1. `data_raw`: a numpy array with the shape $(N, \cdots)$, where N is the number of samples, and there may be many excess dimensions denoted by $\cdots$. You will have to reshape this input `data_raw` matrix to obtain a shape of (N, d), where d is the vectorized data's dimension. For example, `data_raw` could have an input shape of `(6000, 50, 50, 3)`. In this case you will have to reshape the input data to have a shape of `(6000, 7500)`.
2. `num_components`: This is the number of PCA components that we want to retain. This variable is denoted by r in the PCA definition in the textbook.

`principal_components` should return the variable `V_x` which is a numpy array with the shape `(d, num_components)`. The columns are the unitay principal components sorted descendingly with respect to the eigenvalues.

Important Note: You should only use `scikit-learn`'s `TruncatedSVD` decomposition for this task. You can read about this function at https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html.

- You must use the `randomized` algorithm implementation as it is more efficient.
- Since this heuristic is stochastic, you must provide `random_state=12345` as an input argument to this object's constructor for reproducibility.
- Use exactly 5 iterations for this heuristic (i.e., specify `n_iter` to be exactly 5).

Important Note: **Do not** try to recover the covariance matrix Σ and then find its eigenvalues. This can prove to be both inefficient and unnecessary. As the theoretical review before the first task concluded, **There is no need to compute the covariance matrix** Σ. Instead, all you need to do is to find the SVD of the data matrix, and extract the principal components from it.

[27]:
```
def principal_components(data_raw, num_components=20):

    # your code here
    N = np.shape(data_raw)[0]
    dr = data_raw.reshape(N,-1)
    dr = dr - np.mean(dr, axis=0)
    s = TruncatedSVD(n_components=num_components,n_iter=5,random_state=12345)
    s.fit_transform(dr)
    V_x = s.components_.T

    assert V_x.ndim==2
    # Don't mind/change the following lines.
    # This is for mitigating the ambiguity up to -/+1 factor in PCs.
    # (i.e., if x is a unitary PC, then -x is also a unitary PC).
    # We multiply each column by the sign of the largest element (in absolute␣
 ↪value) of that column
    sign_unambiguity = np.sign(V_x[np.abs(V_x).argmax(axis=0), np.arange(V_x.
 ↪shape[1])]).reshape(1,-1)
    V_x *= sign_unambiguity
    return V_x
```

[28]:
```
some_data = (np.arange(35).reshape(5,7) ** 13) % 20
some_pcs = principal_components(some_data, num_components=2)
assert np.array_equal(some_pcs.round(3), np.array([[-0.123, -0.114],
                                                   [-0.43 ,  0.119],
                                                   [-0.021,  0.41 ],
                                                   [-0.603, -0.164],
                                                   [ 0.084,  0.491],
                                                   [-0.223,  0.724],
                                                   [ 0.616,  0.109]]))

# Checking against the pre-computed test database
test_results = test_case_checker(principal_components, task_id=5)
assert test_results['passed'], test_results['message']
```

[29]:
```
#Task 5 Test Cell
```

[30]:
```
if perform_computation:
    first_class_features = images_raw[labels == 0, :, :, :]

    starting_time = time.time()
    first_class_pcs = principal_components(first_class_features,␣
 ↪num_components=20)
    end_time = time.time()
    print('Finding the principal components on a single class took %.3f seconds.
↪'%(end_time-starting_time))
```

```
Finding the principal components on a single class took 0.666 seconds.
```

Using this `principal_components` function, and the `images_raw` array, you could reconstruct an arbitrary image using a small number of components, see the effect of the number of components on the reconstructed image's quality, and share your results on Piazza!

11 Task 6

Write the function `E_A_given_B` that computes the $E[A|B]$ and takes the two matrices `class_A_data` and `class_B_data`.

1. `class_A_data` is a numpy arrays with the shape $(N, \cdots)$, where N is the number of samples, and there may be many excess dimensions denoted by $\cdots$. You will have to reshape this input matrix to obtain a shape of (N, d), where d is the vectorized data's dimension.
2. `class_B_data` has the same data structure as `class_A_data`.

To compute $E[A|B]$: 1. First, do whatever reshaping you have to do. 2. Subtract Class A's mean from its data 3. Use the `principal_components` function you wrote before to extract the 20 principal components of `class_B_data`. 4. Project Class A's data onto the mentioned principal components and get back to the original space. 5. Compute Class A's residuals (i.e., the difference between the original and the projection). 5. Find the squared residual sizes **for each sample**, and then return their mean as the `E_A_cond_B` scalar. In other words, square class A's residuals, sum them over each sample (which should reduce the squared residual matrix to only N elements), and then report the mean of them as `E_A_cond_B`.

[31]:
```
def E_A_given_B(class_A_data, class_B_data):

    # your code here
    N_a = np.shape(class_A_data)[0]
    N_b = np.shape(class_B_data)[0]
    va = np.reshape(class_A_data,(N_a,-1))
    vb = np.reshape(class_B_data,(N_b,-1))
    va = va - np.mean(va,axis=0)
    vb = principal_components(vb, num_components=20)
    vb_t = vb.T
    temp1 = np.matmul(np.matmul(va,vb),vb_t)
    temp2 = np.sum(np.square(va-temp1),axis=1)
    E_A_cond_B = np.mean(temp2)

    return E_A_cond_B
```

[32]:
```
some_data = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
some_data = np.repeat(some_data, 8, axis=1)
some_E = E_A_given_B(some_data, (some_data**1.02))
```

```
assert some_E.round(3)==0.001

# Checking against the pre-computed test database
test_results = test_case_checker(E_A_given_B, task_id=6)
assert test_results['passed'], test_results['message']
```

[33]:
```
#Task 6 Test Cell
```

[34]:
```
if perform_computation:
    num_classes = class_means.shape[0]
    SimilarityMatrix = np.zeros((num_classes, num_classes))
    for row in range(num_classes):
        print(f'Row {row}', end='')
        row_st_time = time.time()
        for col in range(row+1):
            class_A_data = images_raw[labels == row, :, :, :]
            class_B_data = images_raw[labels == col, :, :, :]
            E_A_cond_B = E_A_given_B(class_A_data, class_B_data)
            E_B_cond_A = E_A_given_B(class_B_data, class_A_data)
            SimilarityMatrix[col, row] = (E_A_cond_B + E_B_cond_A)/2.
            SimilarityMatrix[row, col] = (E_A_cond_B + E_B_cond_A)/2.
        print(f' (This row took %.3f seconds to finish)'%(time.time() -␣
 ↪row_st_time))
```

```
Row 0 (This row took 1.907 seconds to finish)
Row 1 (This row took 3.813 seconds to finish)
Row 2 (This row took 5.685 seconds to finish)
Row 3 (This row took 7.710 seconds to finish)
Row 4 (This row took 9.644 seconds to finish)
Row 5 (This row took 11.594 seconds to finish)
Row 6 (This row took 13.635 seconds to finish)
Row 7 (This row took 15.349 seconds to finish)
Row 8 (This row took 17.119 seconds to finish)
Row 9 (This row took 19.009 seconds to finish)
```

If you apply any general `SimilarityMatrix` variable to the previously defined `PCoA` function, you may get `NaN` entries due to the fact that they may not generally be a metric distance matrix (i.e., having non-zero diagonal elements and the triangle inequality not alway holding).

This issue can be best seen when having a similarity measure that is extremely uneven (i.e., when the small entries are extremely small and the large entries are extremely large). This will make it difficult for the triangle inequality to hold. It is a good idea to ammend the PCoA in a way that can deal with such non-metric similarity measures.

[35]:
```
VT = None
if perform_computation:
    VT = PCoA(SimilarityMatrix**40, r=10)
VT
```

```
[35]: array([[-1.19122359e+131, -1.01273169e+131, -6.64858390e+129,
         -8.41498165e+128,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.89955534e+132, -2.69508986e+131, -2.99388279e+126,
         -2.89402188e+125,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.22403442e+131, -5.41618003e+130,  1.22455240e+129,
          1.03543307e+129,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [ 1.13544520e+131,  2.36886946e+130,  1.23040415e+129,
         -1.24137369e+129,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.29411460e+131, -5.17027543e+130,  2.17747099e+129,
          2.70985016e+129,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [ 1.16044578e+132,  2.11801847e+131, -8.28870879e+128,
          2.11454136e+128,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.23537602e+131, -5.69470418e+130, -1.66781093e+129,
          6.27233461e+128,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [ 1.44463768e+132, -4.45826856e+131,  5.30470459e+128,
          1.67460728e+127,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.29378744e+131, -5.39706922e+130,  3.97620683e+129,
         -2.52086520e+129,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan],
        [-1.95219037e+131,  7.97900759e+131,  9.15476863e+126,
          3.30955791e+126,  2.82100788e+123,             nan,
                      nan,             nan,             nan,
                      nan]])
```

12 Task 7

Write the function `Lingoes_PreProcessing` that does some pre-processing to the `SimilarityMatrix` to make it have the Euclidean property and the triangles to close.

Here is a very brief and to the point description from the r documentation page (https://www.rdocumentation.org/packages/ape/versions/5.2/topics/pcoa).

"In the Lingoes (1971) procedure, a constant c1, equal to twice absolute value of the largest negative eigenvalue of the original principal coordinate analysis, is added to each original squared distance in the distance matrix, except the diagonal values. A newe principal coordinate analysis, performed on the modified distances, has at most (n-2) positive eigenvalues, at least 2 null eigenvalues, and no negative eigenvalue."

If you're interested, you can read more about correction for negative eigenvalues in http://biol09.biol.umontreal.ca/PLcourses/Ordination_sections_1.3+1.4_PCoA_Eng.pdf.

The function `Lingoes_PreProcessing` takes the numpy array `SimilarityMatrix` as input, and returns `ProcessedSimilarityMatrix` based on the following condition: 1. If all eigenvalues computed during PCoA are non-negative, then `ProcessedSimilarityMatrix` should be the same as the `SimilarityMatrix`. 2. Otherwise, follow the instructions to perform the Lingoes correction on the `SimilarityMatrix` and return `ProcessedSimilarityMatrix`.

In other words, this is what you're supposed to do: 1. Perform the PCoA analysis on `SimilarityMatrix` right up to the point where you find the eigenvalues. Do not go any further. More precisely, you should only find the eigenvalues of the matrix $\mathcal{W}$ corresponding to `SimilarityMatrix` in the PCoA analysis. 2. Find the minimum eigenvalue and call it $\lambda_{\min}$. 3. If $\lambda_{\min} \geq 0$, then stop and return `SimilarityMatrix` as it was without any change. 4. If $\lambda_{\min} < 0$, then add $2|\lambda_{\min}|$ to all the non-diagonal elements of `SimilarityMatrix` and return the resulting matrix.

Important Note: **Do not call the `PCoA` function on `SimilarityMatrix`**. You should not call the whole `PCoA` function on `SimilarityMatrix`, as you do not care about the output reconstructions of `PCoA`. Instead, you need the eigenvalues for further processing, which are not returned by the `PCoA` function.

Note: You do not need a `for` loop for adding a scalar to the non-diagonal elements of a matrix; you can add the scalar to all the elements of the matrix, and then subtract it from the same scalar multiple of the identity matrix (i.e., using a function like `np.eye` for instance).

[36]:
```python
def Lingoes_PreProcessing(SimilarityMatrix):

    assert SimilarityMatrix.shape[0] == SimilarityMatrix.shape[1]
    num_points = SimilarityMatrix.shape[0]

    # your code here
    ones = np.ones((num_points,1))
    a = np.identity(num_points) - np.matmul(ones,ones.T)/num_points
    w = -np.matmul(np.matmul(a,SimilarityMatrix),a.T)/2
    eigval, eigvec = np.linalg.eigh(w)
    if (eigval[0] < 0):
        temp = np.abs(eigval[0])
        ProcessedSimilarityMatrix = SimilarityMatrix + 2*temp
        ProcessedSimilarityMatrix -= 2*temp*np.identity(num_points)
    else:
        ProcessedSimilarityMatrix = SimilarityMatrix
    return ProcessedSimilarityMatrix
```

[37]:
```
some_data = ((np.arange(35).reshape(5,7) ** 13) % 20) / 7.
some_dist = mean_image_squared_distances(some_data)**5.
some_lingoes = Lingoes_PreProcessing(some_dist)
assert np.array_equal(some_lingoes.round(1), np.array([[      0. ,   898987.1,␣
 ↪2896177.9,  936570.7, 1442744.7],
                                                        [ 898987.1,       0. ,␣
 ↪1229280.9,  897958.5, 1208489.7],
                                                        [2896177.9, 1229280.9,  ␣
 ↪     0. ,  944977.4,  947878.7],
                                                        [ 936570.7,  897958.5, ␣
 ↪944977.4,       0. ,  920604.3],
                                                        [1442744.7, 1208489.7, ␣
 ↪947878.7,  920604.3,       0. ]]))

# Checking against the pre-computed test database
test_results = test_case_checker(Lingoes_PreProcessing, task_id=7)
assert test_results['passed'], test_results['message']
```

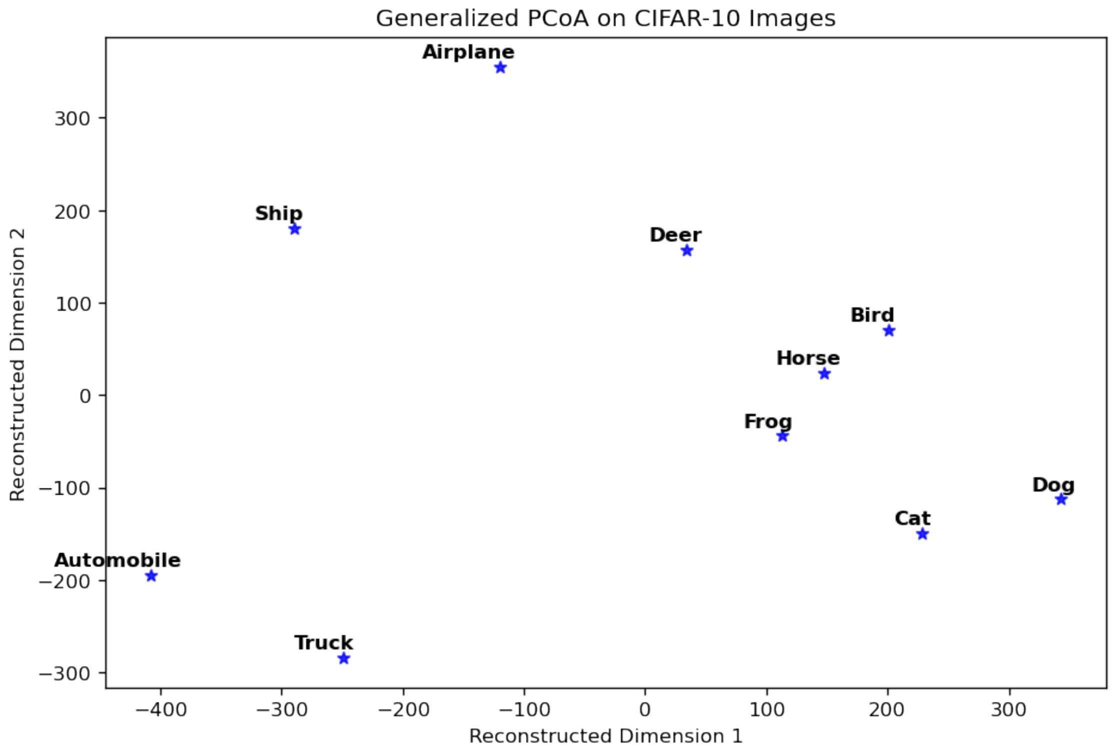
[38]:
```
# Task 7 Test Cell
```

[39]:
```
def PCoA_lingoes(SimilarityMatrix, r=2):
    ProcessedSimilarityMatrix = Lingoes_PreProcessing(SimilarityMatrix)
    return PCoA(ProcessedSimilarityMatrix, r=r)
```

[40]:
```
VT = None
if perform_computation:
    VT = PCoA_lingoes(SimilarityMatrix, r=2)
VT
```

[40]:
```
array([[-119.98786571,  355.24236881],
       [-407.7649715 , -194.47481966],
       [ 200.74413716,   69.91510988],
       [ 228.74930337, -150.11666377],
       [  34.48668888,  156.6042217 ],
       [ 343.06053853, -112.88238769],
       [ 112.75711242,  -44.2970879 ],
       [ 147.41580395,   23.49865708],
       [-290.00131424,  180.85368172],
       [-249.45943287, -284.34308017]])
```

[41]:
```
if perform_computation:
    class_names_list = sorted(list(class_to_idx.keys()))
    fig, ax = plt.subplots(figsize=(9,6.), dpi=120)
    x_components = VT[:,0]
    y_components = VT[:,1]
    sns.regplot(x=x_components, y=y_components, fit_reg=False, marker="*",␣
↪color="Blue", ax=ax)
    for class_idx in range(VT.shape[0]):
        num_letters = len(class_names_list[class_idx])
        ax.text(x_components[class_idx]-num_letters*8,␣
↪y_components[class_idx]+10,
                class_names_list[class_idx].capitalize(),
                horizontalalignment='left', size='medium', color='black',␣
↪weight='semibold')
    ax.set_xlabel('Reconstructed Dimension 1')
    ax.set_ylabel('Reconstructed Dimension 2')
    _ = ax.set_title('Generalized PCoA on CIFAR-10 Images')
```



```
[ ]:
```