Assignment 07 - due 27th of January 2021

Singular Value Decomposition (5 Points)

Let's see what an SVD does to a 3D vector.

Start by calculating the SVD of `H` and save the resulting matrices in the variables `U` , `S` , and `V` below:

In [ ]:

```python
import numpy as np

# H is 3x3
H = np.array([[1.2, 0.3, 0.1], \
              [0.3, 1.3, 0.3], \
              [0.1, 0.3, 0.8]])

# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:

```python
assert np.allclose(U @ np.diag(S) @ V, H) or np.allclose(U @ S @ V, H)
```

Use these matrices in the code below to calculate a reduced representation of the original matrix. First, you need to calculate the compressed matrix, for which multiple ways exist. Make sure that your calculation of the singular values `S` above is consistent with what you're doing below. Some methods return `S` as a list of singular values, some return it as a matrix directly.

The resulting geometrical transformation will be plotted interactively. Is this what you expected the SVD to do?

In [ ]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
# for custom legend
from matplotlib.lines import Line2D
custom_lines = [Line2D([0], [0], color="lightgray", lw=3),
                Line2D([0], [0], color="steelblue", lw=3),
                Line2D([0], [0], color="darkorange", lw=3)]


# this function helps create parallelepipeds in 3d from 3 vectors
def create_parallelepiped_3d(v1, v2, v3):
    origin = np.array([0.0, 0.0, 0.0])
    return [[origin, v1, v1+v2, v2], \
            [v3, v1+v3, v1+v2+v3, v2+v3], \
            [origin, v1, v1+v3, v3], \
            [v1+v2, v2, v2+v3, v1+v2+v3], \
            [v1, v1+v2, v1+v2+v3, v1+v3], \
            [v3, v2+v3, v2, origin]]


# we encapsulate the plotting into a function, so a slider can be added later
def compress_matrix(H, U,S,V, cutoff=2):
    ## instructions: create a variable called "compressed_matrix" and
    ## save the _reduced_ SVD-representation into it (see SVD lesson)
    # YOUR CODE HERE
    raise NotImplementedError()


    # unit cube basis vectors
    E_vecs = np.eye(3)

    ## instructions: create a variable called "H_vecs" that contains the unit vecto
    ## transformed by the original matrix "H", and another variable called "SVD_vec
    ## that contains the unit vectors transformed by the "compressed_matrix"
    # YOUR CODE HERE
    raise NotImplementedError()


    # create rhombi
    E_rhombus = create_parallelepiped_3d(*E_vecs)
    H_rhombus = create_parallelepiped_3d(*H_vecs)
    SVD_rhombus = create_parallelepiped_3d(*SVD_vecs)


    # plot setup
    fig = plt.figure(figsize=[8,5])

    ax1 = fig.add_subplot(111, projection='3d')
    ax1.set_title(label=r"Compare $H$ and SVD$(H)$")

    # turn off everything to make plot more clear
    fig.patch.set_visible(False)
    ax1.axis('off')
    # add all polygons
    ax1.add_collection3d(Poly3DCollection(E_rhombus,
      facecolors='lightgray', linewidths=1, edgecolors='darkgray', alpha=.25))

    ax1.add_collection3d(Poly3DCollection(H_rhombus,
      facecolors='steelblue', linewidths=1, edgecolors='darkblue', alpha=.25))

    ax1.add_collection3d(Poly3DCollection(SVD_rhombus,
      facecolors='darkorange', linewidths=1, edgecolors='red', alpha=.15))

    # if you try different matrices, you might need to adjust axes limits:
    ax1.set_xlim(-0.1, 2.0)
    ax1.set_ylim(-0.1, 1.6)
    ax1.set_zlim(-0.1, 1.5)

    ax1.legend(custom_lines, [r"Original", r"$H$", r"SVD$(H)$"])

    plt.tight_layout()

interact(compress_matrix,
         H=fixed(H),
         U=fixed(U),
         S=fixed(S),
         V=fixed(V),
         cutoff=widgets.IntSlider(min=0, max=3, value=2, continuous_update=False));
```

In [ ]:

```

```

If everything worked, you should see that SVD-compression causes a geometric projection onto a lower-dimensional subspace and just like in the case with singular and non-square matrices, cannot be reversed without more knowledge about the original vectors.

Generative PCA (7 Points)

As discussed in the lecture, PCA can also be used generatively by applying the inverse transformation to arbitrary latent space vectors. Let's see if we can generate original airfoil designs using PCA.

First, we need to load a dataset. The following data is taken from the UIUC Airfoil Coordinates Database (https://m-selig.ae.illinois.edu/ads/coord_database.html). The full database contains 1550 designs, of which we'll only use the *Drela AG* airfoil designs, named `agXX.dat` in the full dataset. First, we need to load the appropriate data:

In [ ]:

```
%matplotlib inline
import numpy as np
# glob.iglob allows iterating over wildcard filenames
import glob
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# the data is given in columns in *.dat files
# we'll append every sample to this list
raw_data = []

for path in glob.iglob(r'coord_seligFmt/ag*.dat'):
    try:
        # np.loadtxt tries its best to parse data from text files
        # skiprows=1 tells numpy to ignore the first row
        raw_data.append(np.loadtxt(path, dtype=np.float, skiprows=1))
    except:
        # when parsing the file does not work, ignore it instead
        # of stopping the whole process
        pass


fig = plt.figure()

def plot_airfoils(i):
    plt.cla()

    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.03, 0.1])

    fig.patch.set_visible(False)
    plt.axis('off')

    plt.plot(raw_data[i][:,0], raw_data[i][:,1], lw=6, color="black")

interact(plot_airfoils, i=(0, len(raw_data)-1))

# not necessary, but if you'd like to save python objects to file,
# you can use pickle, which serializes objects, to do so:
#
#import pickle
#
#with open('raw_data.pkl', 'wb') as f:
#    pickle.dump(raw_data, f)
#
# and to load the file:
#with open('raw_data.pkl', 'rb') as f:
#    raw_data = pickle.load(f)
```

The instruction above is wrapped in a `try` block, because some of the data contains instructions or words that cannot be parsed easily. In such cases, manual work is necessary to clean up the data. What happens in a `try` block is that the code is executed, but when an error in form of an `Exception` occurs, the `except` block is executed instead of halting the computation completely.

Now we have the raw data in a list, but there is a problem with the raw data:

In [ ]:

```
print([len(coords) for coords in raw_data])
```

Different numbers of coordinates for the airfoil designs are provided, so we cannot directly apply a PCA here. We can *interpolate* the data using cubic splines from `scipy` :

In [ ]:

```
from scipy import interpolate

interpolated_data = []

for d in raw_data:
    try:
        tck,u = interpolate.splprep(d.transpose(), s=0, k=3)
        unew = np.arange(0, 1.01, 0.01)
        interpolated_data.append(interpolate.splev(unew, tck))
    except:
        pass

# now that all samples have the same number of features,
# we can put everything into a numpy array for ease of use
interpolated_data = np.array(interpolated_data)

# not necessary, but saving numpy arrays is as easy as
#np.save("interpolated_data.npy", interpolated_data)
# and loading them:
#interpolated_data = np.load("interpolated_data.npy")

print(interpolated_data.shape)
```

The number of data points is now consistent with 101 features.

Plotting the interpolated data yields smoother airfoil plots, although it hardly makes a difference for the small set we're using here. You can try incorporating all the samples from the UIUC dataset on your own system, where in some cases the interpolation makes a huge difference in smoothing the depictions.

In [ ]:

```
def plot_profile(i):
    fig = plt.figure()

    plt.plot(interpolated_data[i][0], interpolated_data[i][1], lw=6, color="black")

    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.03, 0.1])

    fig.patch.set_visible(False)
    plt.axis('off')

interact(plot_profile, i=(0, len(interpolated_data)-1))
```

Let's perform the PCA now. Check the [scikit-learn documention (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html) to see how it works. You first need to import the `PCA` class, then instantiate it in the variable `af_pca` , with the number of components set to `latent_dim` .

The next step is to "fit" the PCA object to the data and saving the result in a variable called `af_pca_results`.

The data to fit the PCA to has to be 2-dimensional, so we need to reshape the `interpolated_data` array. You can do so with `interpolated_data.reshape(29,202)`, which will effectively create a single feature vector consisting of all 101 x and all 101 y coordinates. You can either do this in-place, or by creating and intermediate variable `interpolated_data_reshaped` or so, that you provide as an argument to the `fit` function.

In [ ]:

```
latent_dim = 10

# keep the latent_dim at 10, and put your solution below this line:
# YOUR CODE HERE
raise NotImplementedError()

fig, axes = plt.subplots((latent_dim-1 ) // 5 + 1, min(latent_dim,5),figsize=(14,9)
fig.patch.set_visible(False)


for i, ax in enumerate(axes.flat):
    ax.scatter(af_pca_results.components_[i].reshape(2,101)[0],
               af_pca_results.components_[i].reshape(2,101)[1],
               lw=1, color="black")
    ax.axis('off')
```

In [ ]:

```
assert af_pca_results.n_components == latent_dim
```

The plots above are the components found by the PCA. The shapes don't mean anything, they are just the coordinates that will be combined to form new airfoil shapes. The original data can be expressed as linear combinations of these:

In [ ]:

```
components = af_pca.transform(interpolated_data.reshape(29,202))
projected = af_pca.inverse_transform(components)

fig, axes = plt.subplots(5,5,figsize=(9,9))
fig.patch.set_visible(False)

offset=0

for i, ax in enumerate(axes.flat):
    ax.plot(projected[i+offset].reshape(2,101)[0], projected[i+offset].reshape(2,10
    ax.set_xlim([-0.1, 1.1])
    ax.set_ylim([-0.03, 0.1])
    ax.axis('off')
```

The PCA should have yielded components which meaningfully convey certain properties of the airfoil designs above. Let's see if this is true:

In [ ]:

```
def plot_generated(**kwargs):
    plt.figure(figsize=(7,7))

    input_vector = [weight for weight in kwargs.values()]

    generated = af_pca.inverse_transform(input_vector)

    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.15, 0.2])

    fig.patch.set_visible(False)
    plt.axis('off')

    plt.plot(generated.reshape(2,101)[0], generated.reshape(2,101)[1], lw=6, color=

base_sample = 18
component_sliders = [widgets.FloatSlider(
        value=components[base_sample][i],
        min=min(components[:,i]),
        max=max(components[:,i]),
        step=(max(components[:,i] - min(components[:,i]))/10),
    ) for i in range(latent_dim)]

kwargs = {'c' + str(i):slider for i,slider in enumerate(component_sliders)}

interact(plot_generated, **kwargs)
```

You should see that the last sliders barely have an influence on the airfoil shape at all, since they represent the components with the lowest variance. The first sliders influence the result the most, since here, the variance is high. You should also see that the the variations induced by manipulating the sliders are meaningful relative to the "training" set.

PCA should also find representations of the data that makes it possible to understand the difference in samples in a much lower-dimensional setting. We can see that this works in the correlation plot of principal components 1 and 2. To do so, save all the x-coordinates of the principal components from the `components` array into a variable called `PC1` , and all the y-coordinates into a variable called `PC2` below:

In [ ]:

```
%matplotlib inline
fig = plt.figure(figsize=(9,9))

# YOUR CODE HERE
raise NotImplementedError()

plt.scatter(PC1, PC2, ec="black")
plt.xlabel(r"PC 1", fontsize=18)
plt.ylabel(r"PC 2", fontsize=18)

for i in range(components.shape[0]):
    plt.annotate(str(i), (components[i,0]+0.002, components[i,1]+0.001))
```

In [ ]:

```
assert PC1.shape == (29,)
assert PC2.shape == (29,)
```

Here, the samples 18, 23, 27, and 28 seem to differ a lot from the rest of the samples regarding the first component. It's clear which property is captured by the first component when you look at the specific samples in the first graph of this exercise, by moving the slider to the respective number.

PCA and t-SNE (10 Points)

We saw a brief introduction to t-SNE in the lecture today, and learned that it scales poorly with the number of features/dimensions of a problem. We also briefly mentioned that dimensionality reduction is a preprocessing step of many algorithms, reducing the number of dimensions and noise in the data. Let's see how that works and how it can help.

We'll use the infamous MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/) here, which is a very "clean" dataset in the sense that most techniques just work on it without many problems. For real data, it won't be so easy. The data will suffice anyway to showcase the power of combining dimensionality reduction with other techniques.

First, we need to load the data:

In [ ]:

```
# if you don't have the data locally available and want to try things
# on your own system, you can get the data via the following code:
#import scikit.datasets as ds
#mX, my = ds.fetch_openml()

import numpy as np

# the full dataset consists of 70000 samples
# that's too much for this little server to handle
# so we'll reduce it to 6000
# here, it's okay to use the first num_samples data points,
# since the array is already shuffled
num_samples = 6000

# rescale the values to the range [0,1] for plotting in grayscale
X = np.load("/data/mX.npy")[:num_samples] / 255.0
# sometimes you need to tell numpy the datatype with .astype(type)
y = np.load("/data/my.npy", allow_pickle=True)[:num_samples].astype(int) # necessar

print(X.shape, y.shape)
```

It makes sense to visualize what the data actually looks like:

In [ ]:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))

for i in range(0,9):
    ax = fig.add_subplot(3,3,i+1, title="Digit: " + str(y[i]))

    ax.imshow(X[i].reshape((28,28)), cmap="gray_r")

    ax.set_xticks([])
    ax.set_yticks([])

plt.tight_layout()
```

So the set consists of handwritten numbers as images of 28×28 pixels, already shuffled randomly. The features in this dataset are the 784 pixel values x_i in the range $x_i \in [0, 1]$, so the dimension of this problem is $\dim(X) = 784$, which is already quite high (in data-heavy companies, dimensions of some problems can reach millions). It makes sense to try to reduce the dimensionality of the problem. Let's see what PCA can do first.

For the cell below, look at the scikit-learn documentation () and find out what you have to import here. You have to instantiate a `PCA` object, referred to by the variable `pca` here, so after the `import` statement you should have something like `pca = ?()` . The only argument you need here for instantiating the object is the number of components `n_components=3` .

The next step is to actually apply the PCA to the dataset `X` . Save the result of doing so in the variable `pca_results` below. The way scikit-learn does this is to first "fit" the PCA to the data, then "transform" it to get the `pca_results` , but there is also a function implemented that does both at once. Hence, this part of the exercise can be done in three lines of code.

In [ ]:

```
# YOUR CODE HERE
raise NotImplementedError()

print("Explained variance per principal component: " + str(pca.explained_variance_r
```

In [ ]:

```
assert pca.n_components == 3
```

The first three components account for roughly $10\%, 7$ and 6% of the variance in the data respectively. These numbers seem low, but let's see what they actually capture visually (we'll leave out the 1D plot for clarity):

In [ ]:

```
%matplotlib notebook

# helper function for future plotting
def plot_dimred_results(results, kind):
    fig = plt.figure(figsize=(12,7))

    #ax1 = fig.add_subplot(131) # next numbers would have to be 132 and 133
    ax2 = fig.add_subplot(121)#, sharey=ax1)
    ax3 = fig.add_subplot(122, projection='3d')

    #ax1.scatter(results[:,0], np.zeros(6000), c=y)
    #ax1.set_title("1D " + kind)

    scatter2 = ax2.scatter(results[:,0], results[:,1], c=y, cmap="tab10")
    ax2.set_title("2D " + kind, fontsize=22)

    scatter3 = ax3.scatter(results[:,0], results[:,1], results[:,2], c=y, cmap="tab
    ax3.set_title("3D " + kind, fontsize=22)

    #legend1 = ax1.legend(*scatter1.legend_elements(), title=r"Digits")
    #ax1.add_artist(legend1)

    legend2 = ax2.legend(*scatter2.legend_elements(), title=r"Digits")
    ax2.add_artist(legend2)

    legend3 = ax3.legend(*scatter3.legend_elements(), title=r"Digits")
    ax3.add_artist(legend3)

    #ax1.set_xlabel("Component 1")
    ax2.set_xlabel(r"Component 1", fontsize=18)
    ax2.set_ylabel(r"Component 2", fontsize=18)
    ax3.set_xlabel(r"Component 1", fontsize=18)
    ax3.set_ylabel(r"Component 2", fontsize=18)
    ax3.set_zlabel(r"Component 3", fontsize=18)

    plt.tight_layout()



plot_dimred_results(pca_results, "PCA")
```

The separation isn't perfect, but we get an impression of which points are close and which are very different in ths principal space. All this *without* using the labels `y` at all (they are only used to color the points here, to see how well the PCA worked)! So this is an *unsupervised technique*. The $9$2 and 4s are still close in this space and looking at how they are written by hand, ring on top, line at the bottom, it's clear why this is the case. They look quite similar and someone not used to our way of writing digits (such as a machine) may easily confuse the too.

Let's see now what t-SNE does to the data and measure the time it takes (this takes well over 70 seconds, if the server is not under full load). The `TSNE` class is already imported. Look at the scikit-learn documentation () to see how it is instantiated, and refer to the object with the variable `tsne` . Set the number of components to find to 3, the perplexity to 50, and the number of iterations to 300. For more info during the fitting, you can also set the verbosity to 1, but this is optional.

Then, "fit" `tsne` to the data `X` and "transform" it, saving the result in a variable called `tsne_results` . This can be done in a single step with the appropriate method from the `TSNE` class.

In [ ]:

```
from sklearn.manifold import TSNE
import time

time_start = time.time()

# create a variable called tsne that regers to the object created
# by calling TSNE, imported at the beginning of this cell
# use 3 components and a perplexity of 50. Iterations should be 300
# YOUR CODE HERE
raise NotImplementedError()

print("t-SNE done! This took: " + str(time.time()-time_start) + " seconds")
```

In [ ]:

```
assert tsne.n_components == 3
```

Let's see which components t-SNE found and compare in 2D and 3D:

In [ ]:

```
%matplotlib notebook

plot_dimred_results(tsne_results, "t-SNE")
```

The separation works much better, especially in 3D the individual classes are spaced out further apart from each other. This is a great way to preprocess data before trying clustering algorithms.

The idea of this exercise was to combine the approaches, for saving time during the t-SNE algorithm, since it scales poorly with the number of features in a dataset. So we can use PCA to reduce the number of dimensions to, say, 50:

In [ ]:

```
# perform a pca here with 50 components, refered to as pca_50
# fit the PCA to the data and transform it, save the result
# in a variable called pca_50_results
# YOUR CODE HERE
raise NotImplementedError()

print("Cumulative explained variance for 50D PCA: " + str(np.sum(pca_50.explained_v
```

In [ ]:

```
assert pca_50.n_components == 50
assert pca_50_results.shape == (6000, 50)
```

Then use t-SNE on the reduced dataset (this takes roughly 20 seconds, if the server is not under full load). Use the same arguments as before for the full t-SNE. Refer to the `TSNE` object with the variable `tsne_pca` and save the results of the transformation in the variable `tsne_pca_results` . The data to fit t-SNE to and transform now is the `pca_50_results` from above:

In [ ]:

```
time_start = time.time()

# YOUR CODE HERE
raise NotImplementedError()

print("t-SNE done! This took " + str(round(time.time()-time_start)) + " seconds")
```

In [ ]:

```
assert tsne_pca.n_components == 3
```

This time, it only took a fraction of the time it took to perform t-SNE on the full dataset. Let's compare all the results:

In [ ]:

```
def plot_comparison(data):
    fig = plt.figure(figsize=(12,7))

    ax1 = fig.add_subplot(231)
    ax2 = fig.add_subplot(232, sharey=ax1)
    ax3 = fig.add_subplot(233, sharey=ax1)

    axes_2d = [ax1, ax2, ax3]

    ax4 = fig.add_subplot(234, projection='3d')
    ax5 = fig.add_subplot(235, projection='3d')
    ax6 = fig.add_subplot(236, projection='3d')

    axes_3d = [ax4, ax5, ax6]

    for i,results in enumerate(data):
        axes_2d[i].scatter(results[:,0], results[:,1], c=y, cmap="tab10")
        axes_2d[i].set_xlabel("component 1")
        axes_2d[i].set_ylabel("component 2")
        axes_3d[i].scatter(results[:,0], results[:,1], results[:,2], c=y, cmap="tab
        axes_3d[i].set_xlabel("component 1")
        axes_3d[i].set_ylabel("component 2")
        axes_3d[i].set_zlabel("component 3")


    plt.tight_layout()


plot_comparison([pca_results, tsne_results, tsne_pca_results])
```

The t-SNE results are quite similar, although the time to compute the rightmost dimensionality reduction was much shorter than for the full analysis. This is how dimensionality reduction can help in reducing noise in systems, as well reducing computation time for large problems. The combination we used above, namely reducing the dimensionality of the problem to 50 dimensions via PCA, is what happens when using the parameter `init="pca"` when instantiating a `TSNE` object from scikit-learn.

Keep in mind that these are probabilistic methods and as such, yield different results everytime they're run. You may have to rerun the t-SNE part once or twice to get good results.