Assignment_01

April 14, 2021

1 Assignment 02

1.1 Quadratic Forms (3 points)

Let's take a look at how different matrices produce different "distance landscapes". In the cell below, create a function called `qform()` that takes as input a matrix and two vectors (in that order!), then returns the resulting quadratic form. Make it as general as possible (it should also work for 5-dimensional matrices and vectors, for example).

[ ]:
```
import numpy as np

v1 = np.array([1.0, 1.0, 1.0])
v2 = np.array([0.0, 2.0, 7.0])

A  = np.array([[1.0, 0.0, 0.0], \
               [0.0, 1.0, 0.0], \
               [0.0, 0.0, 1.0]])

# YOUR CODE HERE
raise NotImplementedError()
```

[ ]:
```
A9, v10, v11 = np.eye(3), np.random.rand(3), np.random.rand(3)
assert qform(A, v10, v11) == v10.T @ v11
assert qform(A9, v10, v10) == v10.T @ v10
```

As mentioned in the lecture when you have a notion of *length*, you automatically have a notion of *distance* by simply measuring the length of the difference of two vectors $||\Delta\mathbf{v}|| = ||\Delta\mathbf{v}_2 - \mathbf{v}_1||$, or, in this case, applying the quadratic form to the difference of two vectors.

Let's see what the qform function does to a variety of 2d vector differences by plotting the resulting value for difference vectors that exist around the origin. For example, if the difference of two vectors is $[1, 1]$, the resulting qform result would show up in the plot at $x = 1, y = 1$.

In the following code, apply your `qform()` function from above to an array of vectors called `vecs`, that is created below:

[ ]:
```
# this makes the resulting image a bit more interactive and rotatable in Jupyter
%matplotlib notebook

# from matplotlib, we only need the "pyplot" class
```

```python
import matplotlib.pyplot as plt
# we need the "axes3d" class for 3d plots, so in 2d, this can be omitted
from mpl_toolkits.mplot3d import axes3d


## define your matrix for the quadratic form here
A = np.array([[1.0, 0.0], \
              [0.0, 1.0]])


# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))


# this is an "axis" object, which you can use to plot.
# the "111" means, that this will only contain a single image
# later, we will use subplots with more figures, where the
# first 2 numbers indicate the number of plots per dimension,
# like "224" would indicate 2 by 2 plots. The last number gives
# the total number of plots.
# "projection='3d'" makes this "axis"-object 3d-capable
ax = fig.add_subplot(111, projection='3d')


# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)


# to create coordinate matrices out of this space (similar to matlab):
x, y = np.meshgrid(mesh_points, mesh_points)


# we need the vectors of these coordinates for qform to act on
vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T


# this will plot all our vectors, to see we've done it correctly
ax.scatter(*vecs.T, color='darkorange', s=0.2)


# apply qform function to the vectors
## instructions: create a variable "q" here, which will contain the results
## of the qform, applied on "vecs". Make it a numpy array. It should have
## shape (400,)
# YOUR CODE HERE
raise NotImplementedError()
```

```
print(q.shape)


# the plot functions expect matrices, so we have to reshape the results
# to fit the shape of the coordinate arrays x and y
q = q.reshape(20,20)


# this will plot the surface of the action on the vectors
ax.plot_surface(x, y, q, cmap='viridis')


# with this, you can include a projection plot, where "zdir" gives the direction
ax.contourf(x, y, q, zdir='z', offset=0, cmap='viridis')


# you might need to adjust the axes limits
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim( 0, 8)


# this displays the plot on the screen
plt.show()
```

Feel free to try out a few different matrices and see how that changes the landscape. See for example, what the matrix [[1, 0], [0, -1]] does and how that affects the landscape, or skew it by introducing non-diagonal elements.

[ ]:
```

```

1.2 Singular Matrices (3 points)

We can calculate the pseudoinverse even of singular and non-square matrices. Take for example `B` in the following cell and calculate its pseudo-inverse `B_plus`:

[ ]:
```
import numpy as np

B = np.array([[1.0, 0.5, 1.0], \
              [0.9, 1.0, 0.0], \
              [0.0, 0.0, 0.0]])

print(np.linalg.det(B))

# YOUR CODE HERE
raise NotImplementedError()
```

Check that it is indeed the pseudo-inverse here:

```
[ ]: print(B @ B_plus)
```

```
[ ]: assert np.allclose(B @ B_plus, np.diag([1, 1, 1])) or \
            np.allclose(B @ B_plus, np.diag([1, 1, 0])) or \
            np.allclose(B @ B_plus, np.diag([1, 0, 0]))
```

Now let's explore what the action of the singular matrix and its pseudo-inverse on vectors looks like in 3D. Complete the following code:

```
[ ]: %matplotlib notebook

     import matplotlib.patches as patches
     import matplotlib.pyplot as plt
     from mpl_toolkits.mplot3d import axes3d
     from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection

     # this function helps building the vectors for plotting a parallelepiped
     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]]


     # plot setup, this time we will create two plots
     fig = plt.figure(figsize=(8,4))
     ax1 = fig.add_subplot(121, projection='3d')
     ax1.set_xlim(-1.0, 2.0)
     ax1.set_ylim(-1.0, 2.0)
     ax1.set_title(label="B")
     ax2 = fig.add_subplot(122, projection='3d')
     ax2.set_xlim(-1.0, 2.0)
     ax2.set_ylim(-1.0, 2.0)
     ax2.set_title(label="B_plus B")


     # create a matrix of the basis vectors
     E_vecs = np.eye(3)

     ## instructions: create two matrices of vectors similar to "E_vecs",
     ## called B_vecs and B_plus_vecs, which contain the resulting vectors
     ## after applying B, and after applying B_plus*B respectivly
     # YOUR CODE HERE
```

```python
raise NotImplementedError()


# create the parallelpipeds
E_rhombus = create_parallelepiped_3d(*E_vecs)
B_rhombus = create_parallelepiped_3d(*B_vecs)
B_plus_rhombus = create_parallelepiped_3d(*B_plus_vecs)


# plot original square
ax1.add_collection3d(Poly3DCollection(E_rhombus,
     facecolors='lightgray', linewidths=1, edgecolors='darkgray', alpha=.25))
# apply B
ax1.add_collection3d(Poly3DCollection(B_rhombus,
     facecolors='darkorange', linewidths=1, edgecolors='red', alpha=.25))


# plot B_vecs
ax2.add_collection3d(Poly3DCollection(B_rhombus,
     facecolors='lightgray', linewidths=1, edgecolors='darkgray', alpha=.25))
# apply B_plus
ax2.add_collection3d(Poly3DCollection(B_plus_rhombus,
     facecolors='darkorange', linewidths=1, edgecolors='red', alpha=.25))


plt.tight_layout()
plt.show()
```

If everything worked, you should see that a 3d shape gets reduced to a 2d shape by a singular matrix and hence, that in general, information is lost and cannot be reconstructed even with the pseudoinverse. It's a projection of the original shape onto some 2d subspace of $\mathbb{R}^3$. You will need to rotate the images to see it.

[ ]:

```python

```

1.3 Non-square matrices (1 point)

Non-square matrices can also introduce or destroy geometric information in linear systems. See what the non-square matrix C does to a vector below and how the pseudo-inverse retrieves the original vector.

Calculate the pseudo-inverse `C_plus` of the non-square matrix `C` below.

[ ]:

```python
# C is 3x2
C = np.random.rand(3,2)
```

```python
# YOUR CODE HERE
raise NotImplementedError()

v = np.array([1,1])

# see how the vector v gets transformed into a higher-dimensional space
print(C @ v)

# since no information was destroyed, the pseudo-inverse of C returns the␣
↪original vector
print(C_plus @ C @ v)
```

You see how the non-square matrix `C` transforms the 2d vector into 3d, adding some information, that is destroyed again by `C_plus`. We will later make use of this property for example in the kernel-trick, or in neural networks in general.

[ ]:
```python

```

1.4 Pandas (3 points)

Someone provided you with some data in comma-separated form. In the cell below, use `pandas` to load the file `data.csv` as the variable `dataset`. If it worked, you should get an overview of the table.

[ ]:
```python
import pandas as pd

# create a variable called "dataset" to load the data from the csv-file
# YOUR CODE HERE
raise NotImplementedError()

# don't change the next line, it will print a summary table
dataset
```

[ ]:
```python
assert str(type(dataset)) == "<class 'pandas.core.frame.DataFrame'>"
```

As you can see, there are a few missing data points indicated by `NaN`. We can see this in the standard plot:

[ ]:
```python
dataset.plot()
```

These missing data points pose a problem for many ways in which further processing may happen. Get rid of all rows that contain a `NaN`in the following cell, and save the result in a variable called `dataset_nonan`:

[ ]:
```python
# create a variable called "dataset_nonan" to save the cleaned dataset
# YOUR CODE HERE
raise NotImplementedError()
```

```
# don't change the next line, it will print a summary table
dataset_nonan
```

```
[ ]: assert dataset_nonan.size - dataset_nonan.count().sum() == 0
```

To check how many NaNs there are in a dataset, you can compare the total number of elements `dataset.size` to the number of elements that aren't `NaN` with `dataset.count().sum()`:

```
[ ]: print("\t### Original dataset:###\n")
print("Number of elements in dataset: \t\t", dataset.size)
print("Number of values in dataset: \t\t", dataset.count().sum())
print("Number of NaNs = elements - values: \t", dataset.size - dataset.count().
 ↪sum())

print("\n\n\t### Non-nan dataset:###\n")
print("Number of elements in dataset_nonan: \t", dataset_nonan.size)
print("Number of values in dataset_nonan: \t", dataset_nonan.count().sum())
print("Number of NaNs = elements - values: \t", dataset_nonan.size -␣
 ↪dataset_nonan.count().sum())

dataset_nonan.plot()
```

The plot now doesn't have any gaps like before. Let's plot one column against another:

```
[ ]: from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

def plot_scatter(labels):
    if labels:
        dataset_nonan.plot.scatter(x='petal width (cm)', y='petal length (cm)',␣
 ↪c='target', cmap='viridis')
    else:
        dataset_nonan.plot.scatter(x='petal width (cm)', y='petal length (cm)',␣
 ↪cmap='viridis')

interact(plot_scatter, labels=False)
```

The plot seems to indicate clustering and a linear trend. We'll see plots like these again later in the course. If you check the `labels` box, the plot will color the data points according to the column `target`, which contains the *labels* of the data. We'll come back to what this means later.

In the next cell, save your `dataset_nonan` dataframe as a csv-file called `dataset_nonan.csv`:

```
[ ]: # You don't have to provide any extra arguments to the function here
# YOUR CODE HERE
raise NotImplementedError()
```

```
[ ]:
```