MARTIN KIM (30841222416)

BIOENG 145

HW #1

02/02/2021

1.1 RANK AND EIGENVECTORS

A. $A = \begin{bmatrix} 6 & 4 \\ 4 & 6 \end{bmatrix} \Rightarrow \det(A-\lambda I) = (6-\lambda)^2 - 16 = \lambda^2 - 12\lambda + 36 - 16 = \lambda^2 - 12\lambda + 20 = (\lambda-10)(\lambda-2) \Rightarrow \lambda_1 = 2,\ \lambda_2 = 10$

$E_1 \sim \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \Rightarrow v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad E_2 \sim \begin{bmatrix} -4 & 4 \\ 4 & -4 \end{bmatrix} \sim \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \Rightarrow v_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$B = \begin{bmatrix} 6 & 9 \\ 4 & 6 \end{bmatrix} \Rightarrow \det(B-\lambda I) = (6-\lambda)^2 - 36 = \lambda^2 - 12\lambda = \lambda(\lambda-12) \Rightarrow \lambda_1 = 0,\ \lambda_2 = 12$

$E_1 \sim \begin{bmatrix} 6 & 9 \\ 4 & 6 \end{bmatrix} \sim \begin{bmatrix} 2 & 3 \\ 0 & 0 \end{bmatrix} \Rightarrow v_1 = \begin{bmatrix} -3 \\ 2 \end{bmatrix} \quad E_2 \sim \begin{bmatrix} -6 & 9 \\ 4 & -6 \end{bmatrix} \Rightarrow v_2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$

B. $\text{Nul}(A) = \text{Nul}\begin{bmatrix} 6 & 4 \\ 4 & 6 \end{bmatrix} \sim \begin{bmatrix} 1 & 2/3 \\ 4 & 6 \end{bmatrix} \sim \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \text{Rank}(A) = 2,\ \dim \text{Nul}(A) = 0$

$\text{Nul}(B) = \text{Nul}\begin{bmatrix} 6 & 9 \\ 4 & 6 \end{bmatrix} \sim \begin{bmatrix} 1 & 3/2 \\ 0 & 0 \end{bmatrix} \Rightarrow \text{span}\left\{\begin{bmatrix} -3 \\ 2 \end{bmatrix}\right\} = \text{Nul}(B) \Rightarrow \text{Rank}(B) = 1,\ \dim \text{Nul}(B) = 1$

1.2 SYMMETRIC MATRICES

According to the spectral theorem, $A = Q\Lambda Q^T = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix}^T$, where (λ_i, v_i) are eigenvalue – eigenvector pairs of A

$$\Rightarrow A = Q\Lambda Q^T = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \begin{bmatrix} - & v_1 & - \\ & \vdots & \\ - & v_n & - \end{bmatrix} = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix} \begin{bmatrix} \lambda_1 v_{11} & \cdots & \lambda_1 v_{1n} \\ & & \\ \lambda_n v_{n1} & \cdots & \lambda_n v_{nn} \end{bmatrix} = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix} \begin{bmatrix} - & \lambda_1 v_1 & - \\ & \vdots & \\ - & \lambda_n v_n & - \end{bmatrix}$$

$$= \begin{bmatrix} \lambda_1 v_{11} v_{11} + \ldots + \lambda_n v_{n1} v_{n1} & \cdots & \lambda_1 v_{11} v_{1n} + \ldots + \lambda_n v_{n1} v_{nn} \\ \vdots & & \\ \lambda_1 v_{1n} v_{11} + \ldots + \lambda_n v_{nn} v_{n1} & \cdots & \lambda_1 v_{1n} v_{1n} + \ldots + \lambda_n v_{nn} v_{nn} \end{bmatrix} = \begin{bmatrix} | \\ \lambda_1 v_1 \\ | \end{bmatrix} \begin{bmatrix} - & v_1 & - \end{bmatrix} + \ldots + \begin{bmatrix} | \\ \lambda_n v_n \\ | \end{bmatrix} \begin{bmatrix} - & v_n & - \end{bmatrix} = \sum_{i=1}^{n} \lambda_i v_i v_i^T$$

2.1

A. $Pr\{C, A\} = \frac{23}{80} \times \frac{27}{60} = 0.2484$, $Pr\{T, C\} = \frac{23}{80} \times \frac{23}{60} = 0.2484$

B. i. $P(C \text{ pA}) = \frac{20}{256} = P(A \text{ second} \mid C \text{ first}) P(C) = \frac{20}{20+15+8+17} P(C)$ $\Rightarrow P(C) = 0.234$

$P(C|A) = \frac{P(A|C)P(C)}{P(A)} \Rightarrow P(A) = \frac{P(A|C)P(C)}{P(C|A)} = \frac{\left(\frac{20}{60}\right)(0.234)}{\frac{16}{68}} = 0.332$

ii. $P(A|T) = \frac{16}{16+15+20+17} = \frac{16}{68} = 0.235$, $P(C|T) = \frac{15}{68} = 0.221$, $P(G|T) = \frac{20}{68} = 0.294$, $P(T|T) = \frac{17}{68} = 0.250$

iii. $P(A \text{ first} \mid G \text{ second}) = \frac{17}{17+8+15+20} = 0.283$, $P(C|G) = \frac{8}{60} = 0.133$, $P(G|G) = \frac{15}{60} = 0.25$, $P(T|G) = \frac{20}{60} = 0.333$

iv. Some nucleotide sequences may be more prone to mutations, leading to those sequences appearing less frequently in a genome.

2.2

A.

	1	2	3	4	5	6
fair	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$
weighted	0	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{12}$	$\frac{1}{6}$

B. $P(6 \mid \text{fair}) = 1/6$, $P(6 \mid \text{weighted}) = 1/3$

C. $P(\text{fair} \mid 6) = (1/12)(1/12 + 1/6) = 1/3$, $P(\text{weighted} \mid \text{six}) = 2/3$

3.1

A. $\hat{\theta} = \arg\max_\theta \mathcal{L}(\theta, D) = \arg\max_\theta P(y_1, \dots, y_n \mid h_\theta, x_1, \dots, x_n) = \arg\max_\theta \prod_{i=1}^{n} P(y_i \mid h_\theta, x_i)$

B. Since we know that $z_i \sim N(0, \sigma^2)$, $h_\theta(x_i)$ is fixed for each i, and $y_i = h_\theta(x_i) + z_i$, then that means that the conditional probability distribution of y_i given h_θ and x_i is just a Gaussian with $\mu = h_\theta(x_i)$ and σ^2.

C. Using (A) and (B), we see that $\hat{\theta} = \arg\max_\theta \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - h_\theta(x_i)}{\sigma}\right)^2}$

We can get the same optimizer by taking the log of everything, since they are monotonic functions:

$$\Rightarrow \hat{\theta} = \arg\max_\theta \log\left[\prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - h_\theta(x_i)}{\sigma}\right)^2}\right] = \arg\max_\theta \sum_{i=1}^{n} \log\left[\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - h_\theta(x_i)}{\sigma}\right)^2}\right] = \arg\max_\theta \sum_{i=1}^{n} \log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \log\left(e^{-\frac{1}{2}\left(\frac{y_i - h_\theta(x_i)}{\sigma}\right)^2}\right)$$

$$= \arg\max_\theta \sum_{i=1}^{n} \log\left(e^{-\frac{1}{2}\left(\frac{y_i - h_\theta(x_i)}{\sigma}\right)^2}\right) = \arg\max_\theta \sum_{i=1}^{n} -\frac{1}{2\sigma^2}(y_i - h_\theta(x_i))^2 = \arg\max_\theta \sum_{i=1}^{n} -(y_i - h_\theta(x_i))^2 = \arg\min_\theta \sum_{i=1}^{n} (y_i - h_\theta(x_i))^2$$

Assignment 1

```
#Install if needed
#!pip install numpy
#!pip install pandas
#!pip install matplotlib
#!pip install scipy
```

0. Short Python Practice

Task: Object Oriented Programming: Create a class named Dog

- Each instance of the class should have a variable, name (should be set on creating one)
- The class has two main methods, `action` and `age`
- `action` is an instance method taking in an *optional* parameter, quietly. Calling action prints "WOOF WOOF WOOF " if nothing is passed in, "woof " if quietly is passed in as True. (Use only one hardcoded string in the method)
- `age` is a class method that takes in a list of ages in human years and returns it in dog years (Use list comprehension)

```
import numpy as np

class Dog:
    def __init__(self, name):
        self.name = name
        self.bark = "WOOF"

    def action(self, quietly=False):
        if quietly:
            print(self.bark.lower())
        else:
            print("{0} {0} {0}".format(self.bark))

    def age(self, humans):
        return np.array(humans)*7


human_ages = [3, 4, 7, 4, 10, 6]
```

1. Create an instance of this class, and print its name
2. Call action with both possible options for the parameter
3. Call age on the provided array above

```
my_dog = Dog("doggo")
print(my_dog.name)
my_dog.action()
my_dog.action(quietly=True)
my_dog.age(human_ages)
```

```
doggo
WOOF WOOF WOOF
woof
array([21, 28, 49, 28, 70, 42])
```

1. Using Numpy

Numpy is a commonly used library in Python for handling data, especially helpful for handling arrays/multi-dimensional arrays. It's particularly important for helping bridge the ineffiencies of Python as a high level language with the improved performance of handling data structures in low level languages like C.

```
import numpy as np
```

```
np.random.seed(0) #We set a seed equal to 0 to prevent system-introduced randomness
```

```
### Creating a baseline example of a random protein expression dataset

NUM_FEATURES = 2 #number of proteins we are measuring
NUM_SAMPLES = 100 #number of samples we are observing, each with `NUM_FEATURES` number o

# Defining the difference between old and young population
young_mean, young_std = 1, 1
old_mean, old_std = 0, 1.5

young = young_mean + np.random.randn(NUM_SAMPLES, NUM_FEATURES) * young_std
old = old_mean + np.random.randn(NUM_SAMPLES, NUM_FEATURES) * old_std
```

1. What does the function `np.random.randn(a, b)` do?

HINT: You can find the output dimensions by calling `{YOUR_NUMPY_ARRAY}.shape`, e.g. `young.shape`.

`Returns samples from the standard normal distribution in an axb array (b samples with a features each).`

2. What is the mean and standard deviation of young? What about old?

HINT: Look up the numpy documentation, and keep in mind we are operating on random data

`According to the setup above, these are the supposed values:`

`Young: mean = 1, standard deviation = 1`

`Old: mean = 0, standard deviation = 1.5`

`However, we see that our samples have slightly different values:`

```
print("Young values:")
for i in range(NUM_FEATURES):
    print("Mean of feature {0} is {1}".format(i+1, np.mean(young[:,i])))
    print("Standard deviation of feature {0} is {1}".format(i+1, np.std(young[:,i])))

print("\nOld values:")
for i in range(NUM_FEATURES):
    print("Mean of feature {0} is {1}".format(i+1, np.mean(old[:,i])))
    print("Standard deviation of feature {0} is {1}".format(i+1, np.std(old[:,i])))
```

```
Young values:
Mean of feature 1 is 0.8649775389709009
Standard deviation of feature 1 is 0.9719379589858951
Mean of feature 2 is 0.9962709136106231
Standard deviation of feature 2 is 0.9896232289570579

Old values:
Mean of feature 1 is -0.04729283743523959
Standard deviation of feature 1 is 1.6097624807139423
Mean of feature 2 is 0.1032590236791227
Standard deviation of feature 2 is 1.3598251782984
```

2. Using Pandas and Matplotlib

"In computer programming, pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license."

https://pandas.pydata.org/docs/ https://en.wikipedia.org/wiki/Pandas_(software)

```
NUM_FEATURES = 100 #number of proteins we are measuring
PROTEIN_NAMES = ['protein_'+str(i) for i in range(NUM_FEATURES)] #random string as prote
NUM_SAMPLES = 1000 #number of samples we are observing, each with `NUM_FEATURES` number

# Defining the difference between old and young population
young_mean, young_std = 2, 1
old_mean, old_std = 0, 2

young = young_mean + np.random.randn(NUM_SAMPLES, NUM_FEATURES) * young_std
old = old_mean + np.random.randn(NUM_SAMPLES, NUM_FEATURES) * old_std
```

3. What is the new mean and standard deviation of young and old respectively (show the numpy function to find this)?

```
According to the setup above, these are the supposed values:

Young: mean = 2, standard deviation = 1

Old: mean = 0, standard deviation = 2

However, we see that our samples have slightly different values:
```

```
print("Young values:")
for i in range(NUM_FEATURES):
    print("Mean of feature {0} is {1}".format(i+1, np.mean(young[:,i])))
    print("Standard deviation of feature {0} is {1}".format(i+1, np.std(young[:,i])))

print("\nOld values:")
for i in range(NUM_FEATURES):
    print("Mean of feature {0} is {1}".format(i+1, np.mean(old[:,i])))
    print("Standard deviation of feature {0} is {1}".format(i+1, np.std(old[:,i])))
```

```
Mean of feature 35 is -0.05784779377890795
Standard deviation of feature 35 is 1.947297859943345
Mean of feature 36 is 0.01903947650041265
Standard deviation of feature 36 is 1.9877380552465769
Mean of feature 37 is 0.03822627965590059
Standard deviation of feature 37 is 2.0171127245569975
Mean of feature 38 is -0.08737358925574291
Standard deviation of feature 38 is 1.9950986885449107
```

Mean of feature 39 is -0.10152911301066664
Standard deviation of feature 39 is 1.99054523844502
Mean of feature 40 is 0.05441597798354809
Standard deviation of feature 40 is 1.9759196660189355
Mean of feature 41 is -0.027921636132283382
Standard deviation of feature 41 is 1.9574281298752936
Mean of feature 42 is 0.09453405010186534
Standard deviation of feature 42 is 1.983592624524083
Mean of feature 43 is -0.09086284527703659
Standard deviation of feature 43 is 1.993858736974707
Mean of feature 44 is 0.09009651477371125
Standard deviation of feature 44 is 1.982769557065519
Mean of feature 45 is 0.031853655568407524
Standard deviation of feature 45 is 1.976926192315213
Mean of feature 46 is 0.11592327905841389
Standard deviation of feature 46 is 1.9859274068324049
Mean of feature 47 is -0.001125191681283308
Standard deviation of feature 47 is 2.0376942266411486
Mean of feature 48 is 0.0626400371078718
Standard deviation of feature 48 is 1.9742128057877883
Mean of feature 49 is 0.03925630880126949
Standard deviation of feature 49 is 1.9993889816802393
Mean of feature 50 is 0.2089966837562494
Standard deviation of feature 50 is 2.037341616232548
Mean of feature 51 is 0.05509025170911517
Standard deviation of feature 51 is 1.920420718569035
Mean of feature 52 is 0.02602426926189442
Standard deviation of feature 52 is 1.9690173545912073
Mean of feature 53 is -0.016120691125340158
Standard deviation of feature 53 is 2.022391980910943
Mean of feature 54 is 0.10658634233107861
Standard deviation of feature 54 is 2.0801687355806533
Mean of feature 55 is 0.0634696178597126
Standard deviation of feature 55 is 1.986460165317464
Mean of feature 56 is -0.03891228013590775
Standard deviation of feature 56 is 2.0308709889374787
Mean of feature 57 is 0.028090904431489917
Standard deviation of feature 57 is 1.9947010527921647
Mean of feature 58 is -0.0632047248919879
Standard deviation of feature 58 is 1.9983927374643164
Mean of feature 59 is 0.06357992566156395
Standard deviation of feature 59 is 1.932658660006119
Mean of feature 60 is 0.04618097403288009
Standard deviation of feature 60 is 2.023913258582126
Mean of feature 61 is -0.033580526616835644
Standard deviation of feature 61 is 2.0057024002303696
Mean of feature 62 is 0.038969050207740295
Standard deviation of feature 62 is 2.025142518833139
Mean of feature 63 is -0.02753874830231841
Standard deviation of feature 63 is 2.018002828544695
Mean of feature 64 is -0.054122984856931095
Standard deviation of feature 64 is 2.016583038375476

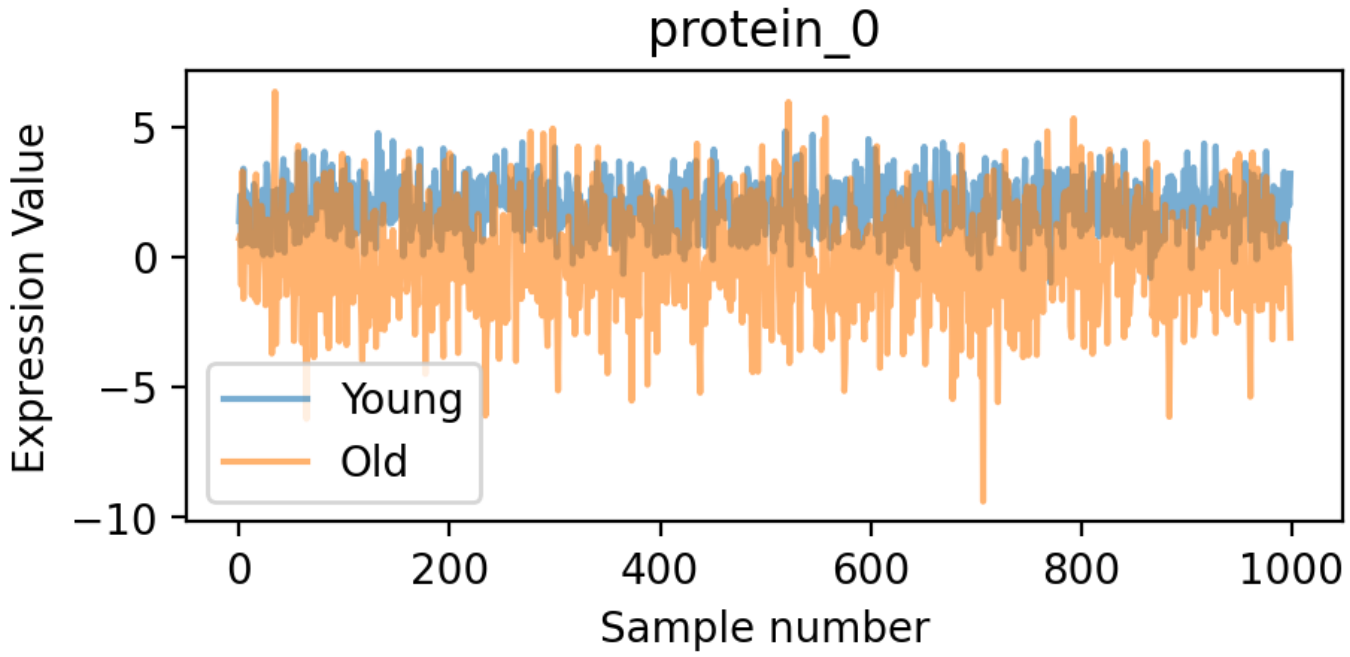
```
### Visualizing our toy dataset
import pandas as pd
import matplotlib.pyplot as plt


young_df, old_df = pd.DataFrame(young), pd.DataFrame(old)
young_df.columns = old_df.columns = list(PROTEIN_NAMES)
young_df.head()
```

	protein_0	protein_1	protein_2	protein_3	protein_4	protein_5	protein_6
0	1.401346	0.884103	2.766663	2.356293	0.231462	2.355482	2.814520
1	2.382732	1.965758	3.096347	1.765784	1.652549	1.418732	0.367365
2	0.449571	2.417319	1.055632	2.238103	0.594037	1.409942	1.889511
3	0.555060	0.789457	1.211331	3.094638	2.234822	4.132153	2.936446
4	3.411172	2.785804	1.942530	1.608783	2.940918	2.405204	2.498052

5 rows × 100 columns

```
### Sample way to quickly visualize the expression in 1 protein
plt.figure(figsize=(5,2), dpi=200)
plt.plot(young_df['protein_0'], label="Young", alpha=0.6)
plt.plot(old_df['protein_0'], label="Old", alpha=0.6)
plt.xlabel("Sample number")
plt.ylabel("Expression Value")
plt.legend()
plt.title("protein_0")
```

```
Text(0.5, 1.0, 'protein_0')
```



4. Fill in the code below to visualize all 100 protein expressions. You may need to look up some syntax in matplotlib.

As you can image, as the number of proteins and number of samples increases, it is computationally and practically infeasible to plot the expression profiles of every protein.

HINT: Look at `plt.subplots(...)`

```
fig, ax = plt.subplots(10,10,figsize=(15,15), sharex=True, sharey=True)
for i, protein in enumerate(PROTEIN_NAMES):
```

```
    ax[i//10,i%10].plot(young_df[protein], label="Young", alpha=0.6)
    ax[i//10,i%10].plot(old_df[protein], label="Old", alpha=0.6)
    ax[i//10,i%10].set_title(protein)

fig.text(0.5, 0.04, 'Sample number', ha='center', va='center')
fig.text(0.06, 0.5, 'Expression Value', ha='center', va='center', rotation='vertical')
```

```
Text(0.06, 0.5, 'Expression Value')
```

Joins in pandas and stacking in numpy

```
#one-hot encoding (https://en.wikipedia.org/wiki/One-hot)
young_df['label'] = 1
old_df['label'] = 0

#This is equivalent to a vertical stack of the design matrix.
sample_df = pd.concat([young_df, old_df])
sample_df
```

	protein_0	protein_1	protein_2	protein_3	protein_4	protein_5	protein_(
0	1.401346	0.884103	2.766663	2.356293	0.231462	2.355482	2.81452(
1	2.382732	1.965758	3.096347	1.765784	1.652549	1.418732	0.36736!
2	0.449571	2.417319	1.055632	2.238103	0.594037	1.409942	1.88951'
3	0.555060	0.789457	1.211331	3.094638	2.234822	4.132153	2.93644(
4	3.411172	2.785804	1.942530	1.608783	2.940918	2.405204	2.49805!
...	...	...	...	...	...	...	..
995	-0.046359	-1.828776	0.810152	2.687185	-2.531080	1.386326	-4.49971:
996	0.079020	0.676755	-1.684366	-0.099265	-2.460491	-1.977585	1.20414!
997	0.398352	-2.199046	-1.078015	-2.393127	0.572875	-0.079467	2.97687(
998	-0.740328	1.120627	1.135134	-2.704248	-3.209807	-0.214572	-1.13646!
999	-3.094272	-0.701402	2.593760	-1.053016	0.649605	-0.273949	-0.04850!

2000 rows × 101 columns

5. Can you do the same in numpy? Don't encode the label of the dataset when doing this step.

```
Insert answer here
```

```
pd.DataFrame(np.vstack((young_df, old_df)))
```

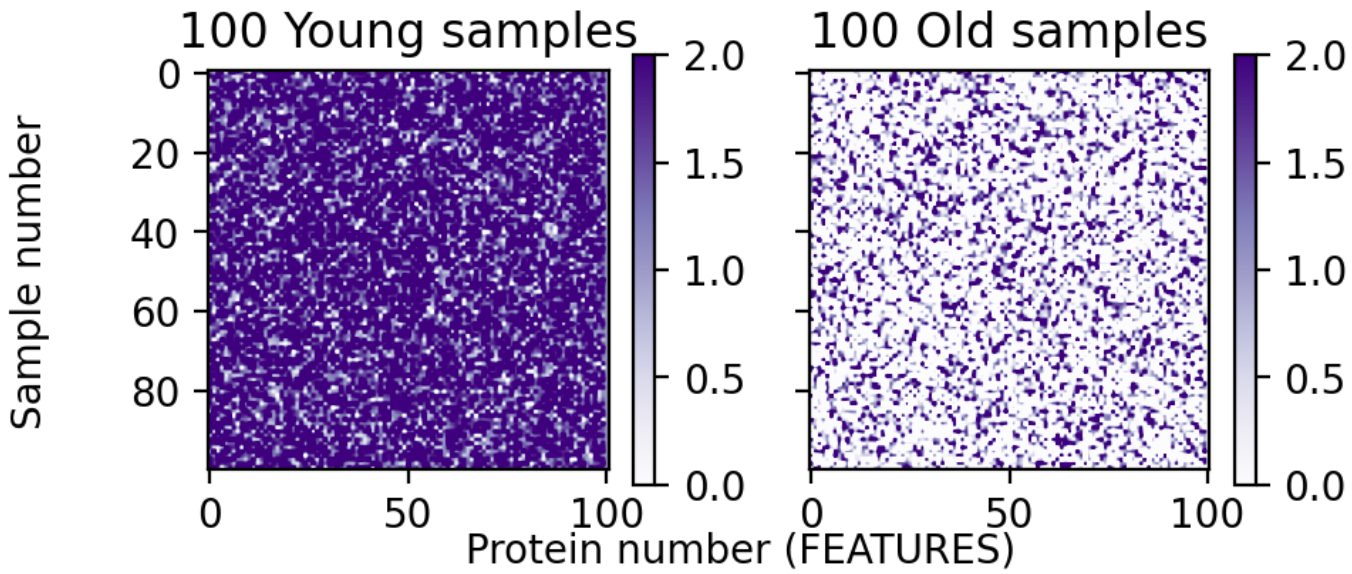
	0	1	2	3	4	5	6	7
0	1.401346	0.884103	2.766663	2.356293	0.231462	2.355482	2.814520	2.058926
1	2.382732	1.965758	3.096347	1.765784	1.652549	1.418732	0.367365	0.432232
2	0.449571	2.417319	1.055632	2.238103	0.594037	1.409942	1.889511	0.339300
3	0.555060	0.789457	1.211331	3.094638	2.234822	4.132153	2.936446	1.964905
4	3.411172	2.785804	1.942530	1.608783	2.940918	2.405204	2.498052	1.973808
...	...	...	...	...	...	...	...	...
1995	-0.046359	-1.828776	0.810152	2.687185	-2.531080	1.386326	-4.499713	-0.835314
1996	0.079020	0.676755	-1.684366	-0.099265	-2.460491	-1.977585	1.204149	-0.319097
1997	0.398352	-2.199046	-1.078015	-2.393127	0.572875	-0.079467	2.976876	0.556308
1998	-0.740328	1.120627	1.135134	-2.704248	-3.209807	-0.214572	-1.136468	1.762052
1999	-3.094272	-0.701402	2.593760	-1.053016	0.649605	-0.273949	-0.048508	-0.814304

2000 rows × 101 columns

Quickly visualizing this data as an image...

```
fig, ax = plt.subplots(1,2,figsize=(5,2), dpi=200, sharex=True, sharey=True)
yimg = ax[0].imshow(sample_df[:100], vmin=0, vmax=2, cmap='Purples')
ax[0].set_title("100 Young samples")
fig.colorbar(yimg, ax=ax[0])

oimg = ax[1].imshow(sample_df[1000:1100], vmin=0, vmax=2, cmap='Purples')
ax[1].set_title("100 Old samples")
fig.colorbar(oimg)

fig.text(0.5, 0.01, 'Protein number (FEATURES)', ha='center', va='center')
fig.text(0.0001, 0.5, 'Sample number', ha='center', va='center', rotation='vertical')
```

```
Text(0.0001, 0.5, 'Sample number')
```



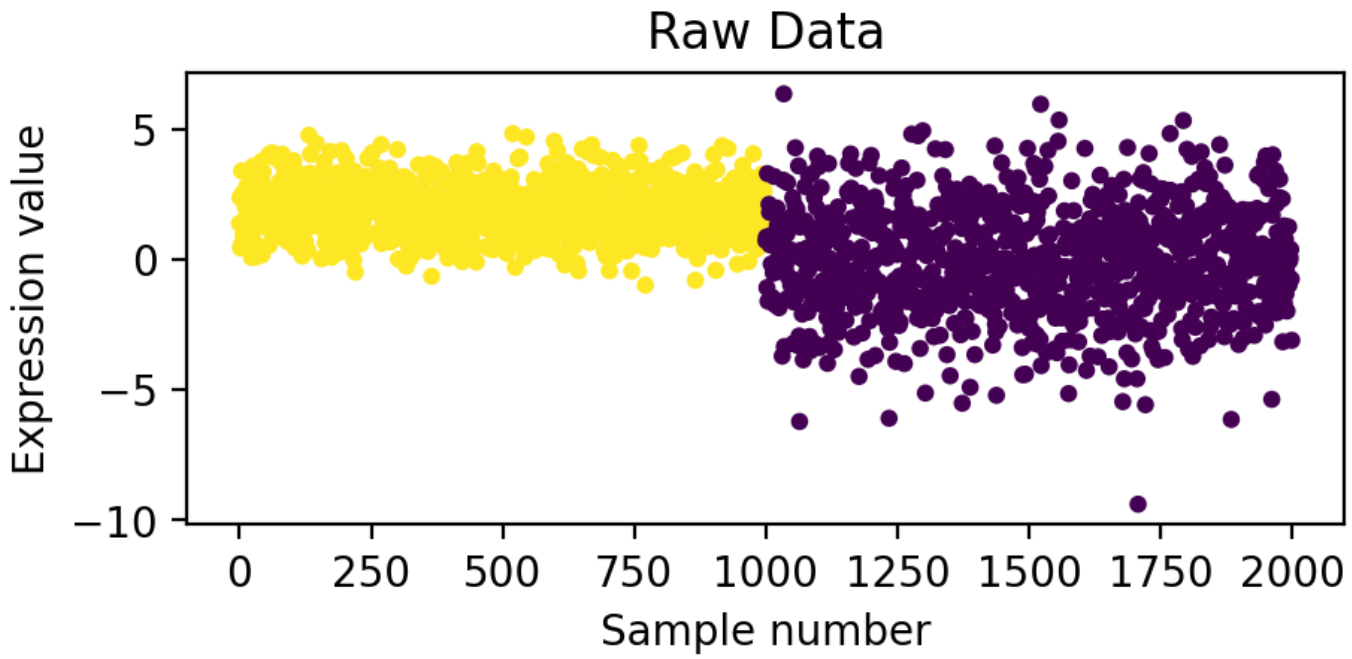
3. Maximum Liklihood Estimation

Let's use our toy data to estimate the parameters for a multivariate normal distribution. Then, from our estimated parameters, we will try to predict whether new data originates from an old tissue sample or young tissue sample.

First, for simplicity, we will consider the case for just 1 protein (univariate normal distribution).

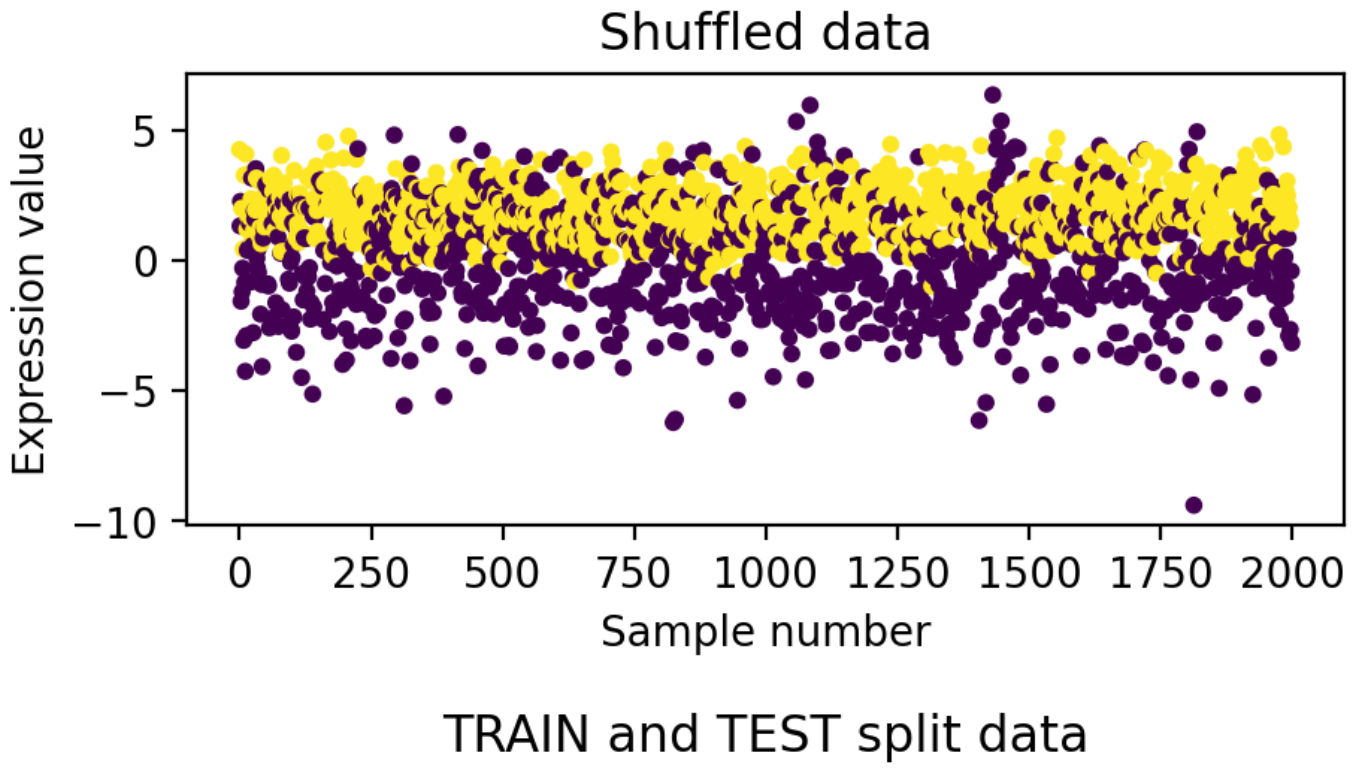
```
X, y = sample_df['protein_0'].values, sample_df['label'].values
### This is equivalent to:
#X, y = sample[:,0], [1] * 1000 + [0] * 1000

print(X.shape, y.shape)
plt.figure(figsize=(5,2), dpi=200)
plt.scatter(range(len(X)), X, c=y, s=10)
plt.title("Raw Data")
plt.xlabel("Sample number")
plt.ylabel("Expression value")
```

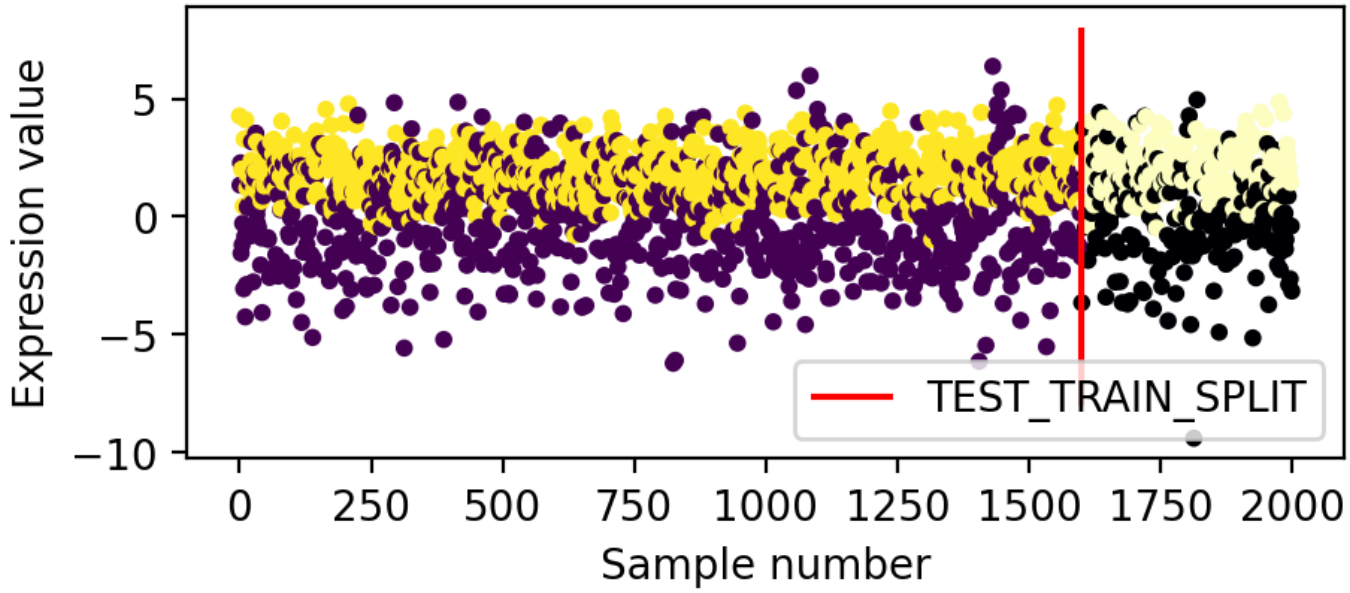
```
(2000,) (2000,)
Text(0, 0.5, 'Expression value')
```




```
### Prepare the training vs testing data, and shuffle.
# It is not essential to understand what is happening in the following steps, but this i
# yet simple machine learning pre-processing workflow.

TRAIN_TEST_SPLIT = 0.8 #80% of the data will be used for training, 20% for testing

def preprocessData(X, y, oneDimDatasetPlot=False):
    #Shuffling the entire dataset
    randomPermutationOrder = np.random.permutation(len(X))
```

```
    X_shuffled, y_shuffled = X[randomPermutationOrder], y[randomPermutationOrder]

    if oneDimDatasetPlot:
        plt.figure(figsize=(5,2), dpi=200)
        plt.scatter(range(len(X)), X_shuffled, c=y_shuffled, s=10)
        plt.title("Shuffled data")
        plt.xlabel("Sample number")
        plt.ylabel("Expression value")
        plt.show()

    #Splitting the dataset into TRAIN vs TEST
    # it is important to cast the indices as integers to avoid floating point indexing
    n = len(X_shuffled)
    X_train, y_train = X_shuffled[:int(n*TRAIN_TEST_SPLIT)], y_shuffled[:int(n*TRAIN_TES
    X_test, y_test = X_shuffled[int(n*TRAIN_TEST_SPLIT):], y_shuffled[int(n*TRAIN_TEST_S

    if oneDimDatasetPlot:
        plt.figure(figsize=(5,2), dpi=200)
        plt.scatter(range(int(n*TRAIN_TEST_SPLIT)), X_train, c=y_train, s=10)
        plt.scatter(range(int(n*TRAIN_TEST_SPLIT), len(X)), X_test, c=y_test, cmap='magr
        plt.vlines(int(n*TRAIN_TEST_SPLIT), -8, 8, label="TEST_TRAIN_SPLIT", color="red'
        plt.legend()
        plt.title("TRAIN and TEST split data")
        plt.xlabel("Sample number")
        plt.ylabel("Expression value")
        plt.show()

    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test = preprocessData(X, y, oneDimDatasetPlot=True)
```






```
### MLE Estimation
### Show in lab how to derive MLE for normal distribution
import scipy
from scipy.stats import multivariate_normal

X_train_young = X_train[y_train == 1]
X_train_old = X_train[y_train == 0]
```

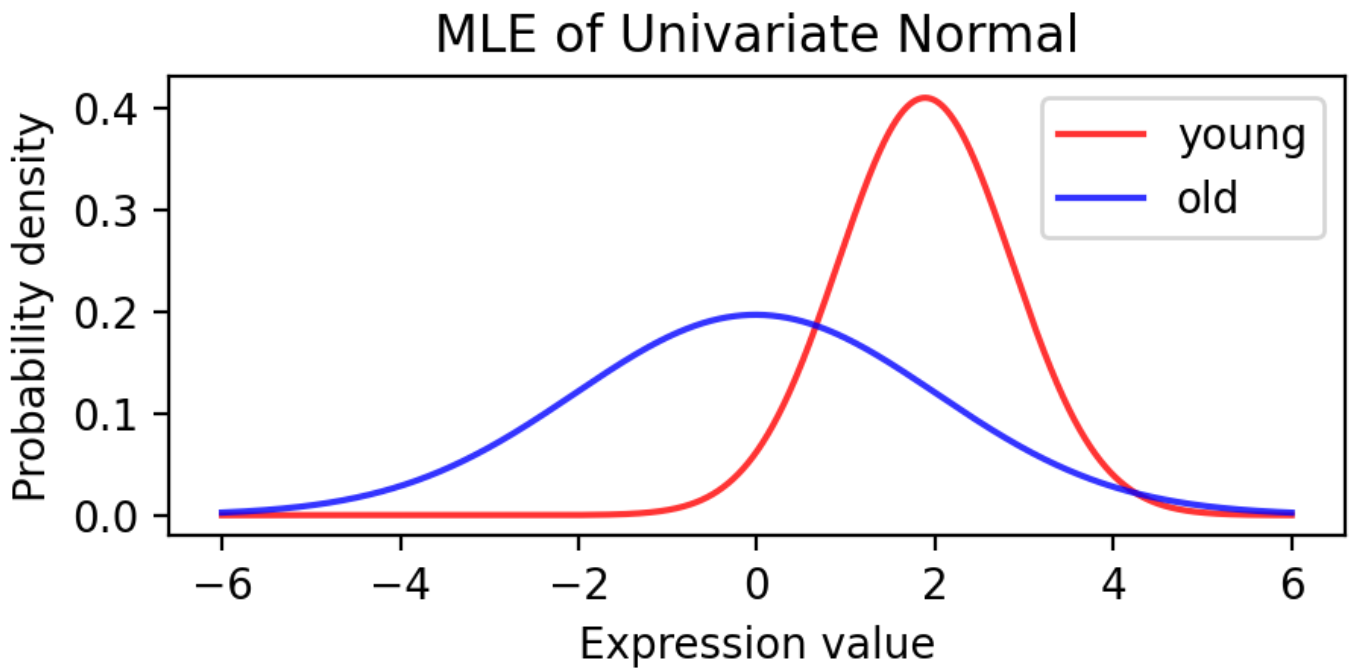
6. Perform the MLE for old and young training samples, and plot the results

HINT: `scipy.stats.multivariate_normal(...).pdf` can be used to generate the probability density function over a range of input values.

```
Insert answer here
```

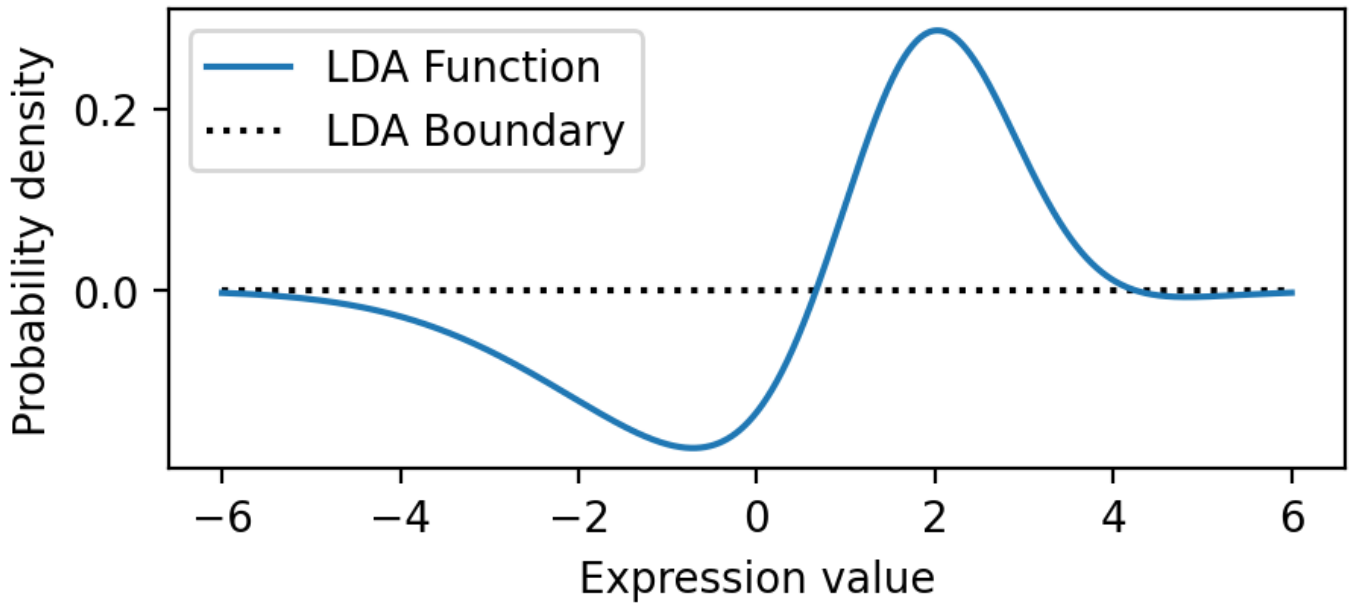
```
youngDist = multivariate_normal(mean=np.mean(X_train_young), cov=np.cov(X_train_young))
oldDist = multivariate_normal(mean=np.mean(X_train_old), cov=np.cov(X_train_old))


linspace = np.linspace(-6, 6, 5000)
plt.figure(figsize=(5,2), dpi=200)
plt.plot(linspace, youngDist.pdf(linspace), alpha=0.8, color="red", label="young")
plt.plot(linspace, oldDist.pdf(linspace), alpha=0.8, color="blue", label="old")
plt.title("MLE of Univariate Normal")
plt.xlabel("Expression value")
plt.ylabel("Probability density")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f4861aa4dd8>
```



```
### Simple LDA (Linear Discriminant Analysis) implementation.
# Areas where the probability is greater than 0 implies it belongs to the young populat
# Areas where the probability is less than 0 implies it belongs to the old population

plt.figure(figsize=(5,2), dpi=200)
plt.plot(linspace, youngDist.pdf(linspace) - oldDist.pdf(linspace), label="LDA Function"
plt.hlines(0, -6, 6, color="black", label="LDA Boundary", linestyles='dotted')
plt.xlabel("Expression value")
plt.ylabel("Probability density")
plt.legend()
```

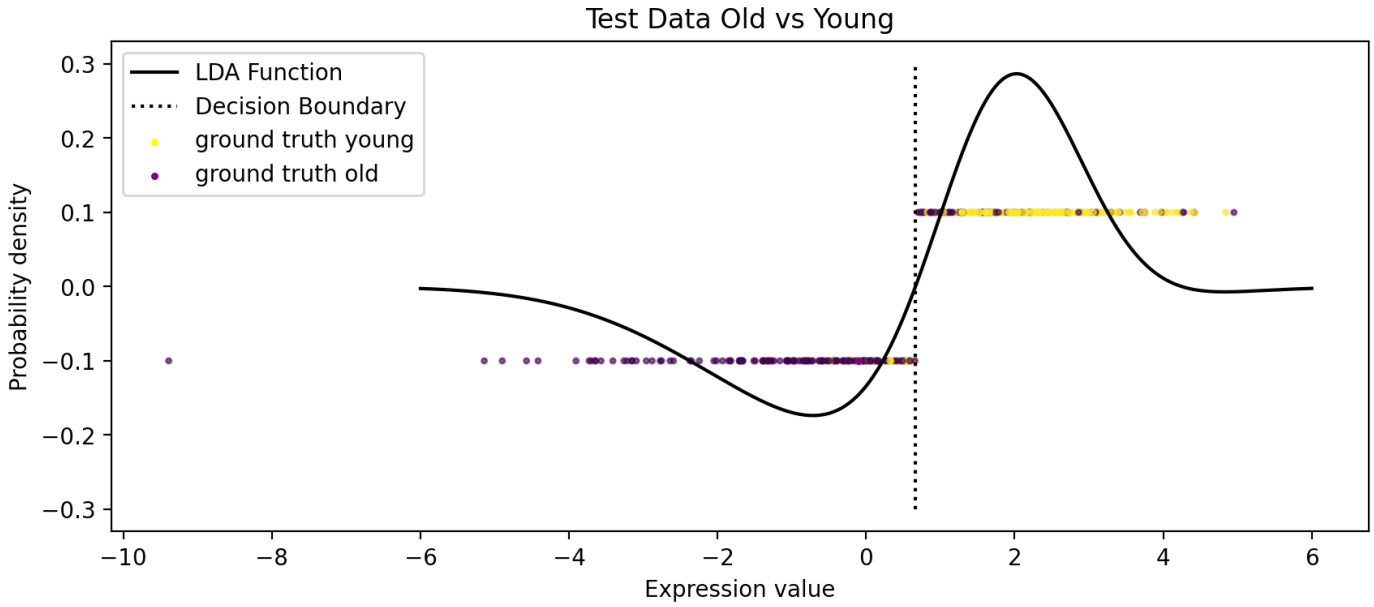
```
<matplotlib.legend.Legend at 0x7f4861a8eda0>
```




```
### Now to evaluate how good (we'll just stick with accuracy here) our LDA function is
# We need to find the point at around 1 where the function flips from negative to posit

# For simplicity (a lot of shortcuts are used here, DO NOT use this in practice)
# In practice, use sci-kit learn's LDA function or implement your own log-liklihood func
diff = youngDist.pdf(linspace) - oldDist.pdf(linspace)
sort = np.argsort(diff[2000:3000])
idx = np.searchsorted(diff[2000:3000][sort], 0)
boundary = linspace[2000+sort[idx]]
print("LDA boundary is located at protein value: ", boundary)
plt.figure(figsize=(10,4), dpi=200)
plt.plot(linspace, youngDist.pdf(linspace) - oldDist.pdf(linspace), label="LDA Function"
```

```
plt.vlines(boundary,ymin=-0.3, ymax=0.3, linestyles='dotted', color="black", label="Dec
#Classifying young
pred_young_vals, ground_truth_young_vals = X_test[X_test >= boundary], y_test[X_test >=
plt.scatter(pred_young_vals, [0.1]*len(pred_young_vals), s=5, c=ground_truth_young_vals,

#Classifying old
pred_old_vals, ground_truth_old_vals = X_test[X_test < boundary], y_test[X_test < bounda
plt.scatter(pred_old_vals, [-0.1]*len(pred_old_vals), s=5, c=ground_truth_old_vals, alph

#kludge to label legend
plt.scatter(-.1,-0.1,color="yellow",label="ground truth young",s=5)
plt.scatter(-.1,-0.1,color="purple",label="ground truth old",s=5)

plt.title("Test Data Old vs Young")
plt.xlabel("Expression value")
plt.ylabel("Probability density")
plt.legend()
plt.show()

#Computing Test accuracy
young_acc = len(ground_truth_young_vals[ground_truth_young_vals == 1]) / len(ground_trut
old_acc = len(ground_truth_old_vals[ground_truth_old_vals == 0]) / len(ground_truth_old_
print("Young Test Accuracy", young_acc)
print("Old Test Accuracy", old_acc)
```

```
LDA boundary is located at protein value:  0.6661332266453295
```




```
Young Test Accuracy 0.7368421052631579
Old Test Accuracy 0.8300653594771242
```

LDA:

$$\delta_k(\mathbf{x}) = \mathbf{x}^T \mathbf{\Sigma}^{-1} \mu_k - \frac{1}{2} \mu_k^T \mathbf{\Sigma}^{-1} \mu_k + \log \pi_k \ ,$$

QDA:

$$\delta_k(\mathbf{x}) = -\frac{1}{2} \log |\mathbf{\Sigma}_k| - \frac{1}{2} (\mathbf{x} - \mu_k)^T \mathbf{\Sigma}_k^{-1} (\mathbf{x} - \mu_k) + \log \pi_k \ .$$