* Prerequisites

In this assignment you will implement the Naive Bayes Classifier. Before starting this assignment, make sure you understand the concepts discussed in the videos in Week 2 about Naive Bayes. You can also find it useful to read Chapter 1 of the textbook.

Also, make sure that you are familiar with the `numpy.ndarray` class of python's `numpy` library and that you are able to answer the following questions:

Let's assume `a` is a numpy array.

- What is an array's shape (e.g., what is the meaning of `a.shape`)?
- What is numpy's reshaping operation? How much computational over-head would it induce?
- What is numpy's transpose operation, and how it is different from reshaping? Does it cause computation overhead?
- What is the meaning of the commands `a.reshape(-1, 1)` and `a.reshape(-1)`?
- Would happens to the variable `a` after we call `b = a.reshape(-1)`? Does any of the attributes of `a` change?
- How do assignments in python and numpy work in general?
  - Does the `b=a` statement use copying by value? Or is it copying by reference?
  - Would the answer to the previous question change depending on whether `a` is a numpy array or a scalar value?

You can answer all of these questions by

```
1. Reading numpy's documentation from https://numpy.org/doc/stable/.
2. Making trials using dummy variables.
```

*Assignment Summary

The UC Irvine machine learning data repository hosts a famous dataset, the Pima Indians dataset, on whether a patient has diabetes originally owned by the National Institute of Diabetes and Digestive and Kidney Diseases and donated by Vincent Sigillito. You can find it at https://www.kaggle.com/uciml/pima-indians-diabetes-database/data. This data has a set of attributes of patients, and a categorical variable telling whether the patient is diabetic or not. For several attributes in this data set, a value of 0 may indicate a missing value of the variable. It has a total of 768 data-points.

- **Part 1-A)** First, you will build a simple naive Bayes classifier to classify this data set. We will use 20% of the data for evaluation and the other 80% for training.

  You should use a normal distribution to model each of the class-conditional distributions.
  (**Class conditional probability** is the probability of each attribute value for an attribute, for each outcome value.
  https://www.sciencedirect.com/topics/mathematics/conditional-probability)
  (The **posterior probability** is calculated by updating the prior probability using Bayes' theorem.)

  Report the accuracy of the classifier on the 20% evaluation data, where accuracy is the number of correct predictions as a fraction of total predictions.

- **Part 1-B)** Next, you will adjust your code so that, for attributes 3 (Diastolic blood pressure), 4 (Triceps skin fold thickness), 6 (Body mass index), and 8 (Age), it **regards a value of 0 as a missing value** when estimating the class-conditional distributions, and the posterior.

  Report the accuracy of the classifier on the 20% that was held out for evaluation.

- **Part 1-C)** Last, you will have some experience with SVMLight, an off-the-shelf implementation of Support Vector Machines or SVMs. For now, you don't need to understand much about SVM's, we will explore them in more depth in the following exercises. You will install SVMLight, which you can find at http://svmlight.joachims.org, to train and evaluate an SVM to classify this data.

  You should NOT substitute NA values for zeros for attributes 3, 4, 6, and 8.

  Report the accuracy of the classifier on the held out 20%

0. Data

0.1 Description

The UC Irvine's Machine Learning Data Repository Department hosts a Kaggle Competition with famous collection of data on whether a patient has diabetes (the Pima Indians dataset), originally owned by the National Institute of Diabetes and Digestive and Kidney Diseases and donated by Vincent Sigillito.

You can find this data at https://www.kaggle.com/uciml/pima-indians-diabetes-database/data. The Kaggle website offers valuable visualizations of the original data dimensions in its dashboard. It is quite insightful to take the time and make sense of the data using their dashboard before applying any method to the data.

0.2 Information Summary

- **Input/Output**: This data has a set of attributes of patients, and a categorical variable telling whether the patient is diabetic or not.

- **Missing Data**: For **several** attributes in this data set, a value of 0 may indicate a missing value of the variable.

- **Final Goal**: We want to build a classifier that can predict whether a patient has diabetes or not. To do this, we will train multiple kinds of models, and will be **handing the missing data with different approaches for each method** (i.e., some methods will ignore their existence, while others may do something about the missing data).

0.3 Loading

In [36]:

```python
%matplotlib inline
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from aml_utils import test_case_checker
```

In [37]:

```python
df = pd.read_csv('../BasicClassification-lib/diabetes.csv')
df.head()
```

Out[37]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

0.1 Splitting The Data

First, we will shuffle the data completely, and forget about the order in the original csv file.

- The training and evaluation dataframes will be named `train_df` and `eval_df`, respectively.

- We will also create the 2-d numpy array `train_features` whose number of rows is the number of training samples, and the number of columns is 8 (i.e., the number of features). We will define `eval_features` in a similar fashion

- We would also create the 1-d numpy arrays `train_labels` and `eval_labels` which contain the training and evaluation labels, respectively.

In [38]:

```python
# Let's generate the split ourselves.
np_random = np.random.RandomState(seed=12345) #??
rand_unifs = np_random.uniform(0,1,size=df.shape[0]) #produce N random numbers from Un
division_thresh = np.percentile(rand_unifs, 80) # return 80th %ile of the random numbe
train_indicator = rand_unifs < division_thresh # train_indicator = 1 if random number
eval_indicator = rand_unifs >= division_thresh # eval_indicator = 1 if random number
```

In [39]:

```python
train_df = df[train_indicator].reset_index(drop=True) #use training obs indicator to
train_features = train_df.loc[:, train_df.columns != 'Outcome'].values #training feat
train_labels = train_df['Outcome'].values # outcome column is the labels vector
train_df.head()
```

Out[39]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	1	85	66	29	0	26.6	0.351	31	0
1	8	183	64	0	0	23.3	0.672	32	1
2	1	89	66	23	94	28.1	0.167	21	0
3	0	137	40	35	168	43.1	2.288	33	1
4	5	116	74	0	0	25.6	0.201	30	0

In [40]:

```python
#similarly to above create features df and outcome vector for evaluation/testing
eval_df = df[eval_indicator].reset_index(drop=True)
eval_features = eval_df.loc[:, eval_df.columns != 'Outcome'].values
eval_labels = eval_df['Outcome'].values
eval_df.head()
```

Out[40]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	3	78	50	32	88	31.0	0.248	26	1
2	10	168	74	0	0	38.0	0.537	34	1
3	0	118	84	47	230	45.8	0.551	31	1
4	7	107	74	0	0	29.6	0.254	31	1

In [41]:

```python
train_features.shape, train_labels.shape, eval_features.shape, eval_labels.shape
```

Out[41]:

```
((614, 8), (614,), (154, 8), (154,))
```

There are 614 obs for training, 154 for testing and 8 features

0.2 Pre-processing The Data

Some of the columns exhibit missing values. We will use a Naive Bayes Classifier later that will treat such missing values in a special way. To be specific, for attribute 3 (Diastolic blood pressure), attribute 4 (Triceps skin fold thickness), attribute 6 (Body mass index), and attribute 8 (Age), we should regard a value of 0 as a missing value.

Therefore, we will be creating the `train_featues_with_nans` and `eval_features_with_nans` numpy arrays to be just like their `train_features` and `eval_features` counter-parts, but with the zero-values in such columns replaced with nans.

In [42]:

```python
train_df_with_nans = train_df.copy(deep=True)
eval_df_with_nans = eval_df.copy(deep=True)
for col_with_nans in ['BloodPressure', 'SkinThickness', 'BMI', 'Age']:
    train_df_with_nans[col_with_nans] = train_df_with_nans[col_with_nans].replace(0, n
    eval_df_with_nans[col_with_nans] = eval_df_with_nans[col_with_nans].replace(0, np
train_features_with_nans = train_df_with_nans.loc[:, train_df_with_nans.columns != 'Ou
eval_features_with_nans = eval_df_with_nans.loc[:, eval_df_with_nans.columns != 'Outco
```

pandas.DataFrame.loc: Access a group of rows and columns by label(s) or a boolean array.

pandas.DataFrame.values property: DataFrame.values Return a Numpy representation of the DataFrame, ie numpy.ndarray where only the values in the dataFrame will be returned, the axes labels will be removed. Warning: We recommend using DataFrame.to_numpy() instead.

In [43]:

```python
print('Here are the training rows with at least one missing values.')
print('')
print('You can see that such incomplete data points constitute a substantial part of t
print('')
#Show only the rows with missing values:
nan_training_data = train_df_with_nans[train_df_with_nans.isna().any(axis=1)]
nan_training_data
#len(nan_training_data)/len(train_df_with_nans)
#30%
```

```
Here are the training rows with at least one missing values.

You can see that such incomplete data points constitute a substantial part of the dat
a.
```

Out[43]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outco
1	8	183	64.0	NaN	0	23.3	0.672	32	
4	5	116	74.0	NaN	0	25.6	0.201	30	
5	10	115	NaN	NaN	0	35.3	0.134	29	
7	8	125	96.0	NaN	0	NaN	0.232	54	
8	4	110	92.0	NaN	0	37.6	0.191	30	
...	...	...	...	...	...	...	...	...	
598	6	162	62.0	NaN	0	24.3	0.178	50	
599	4	136	70.0	NaN	0	31.2	1.182	22	
605	1	106	76.0	NaN	0	37.5	0.197	26	
606	6	190	92.0	NaN	0	35.5	0.278	66	
612	1	126	60.0	NaN	0	30.1	0.349	47	

186 rows × 9 columns

df.isna(): Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, get mapped to True values. **pandas.DataFrame.any**
DataFrame.any(axis=0) Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

axis{0 or 'index', 1 or 'columns', None}

- 0 / 'index' : reduce/eliminate the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce/eliminate the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

1. Part 1 (Building a simple Naive Bayes Classifier)

Consider a single sample (or observation) $(\mathbf{x}, y)$, where the feature vector is denoted with $\mathbf{x}$, and the label is denoted with y. We will also denote the j^{th} feature of $\mathbf{x}$ with $x^{(j)}$.

According to the textbook, the Naive Bayes Classifier uses the following decision rule:

"Choose y such that

$$\left[\log p(y) + \sum_j \log p(x^{(j)}|y)\right]$$

is the largest"

However, we first need to define the probabilistic models (ie: the PDFs) of the prior $p(y)$ and the class-conditional feature distributions $p(x^{(j)}|y)$ using the training data.

- **Modelling the prior** $p(y)$: We fit a Bernoulli distribution to the `Outcome` variable of `train_df`.
- **Modelling the class-conditional feature distributions** $p(x^{(j)}|y)$: We fit Gaussian distributions, and infer the Gaussian mean and variance parameters from `train_df`.

Task 1

Write a function `log_prior` that takes a numpy array `train_labels` as input, and outputs the following vector as a column numpy array (i.e., with shape $(2, 1)$).

$$\log p_y = \begin{bmatrix} \log p(y=0) \\ \log p(y=1) \end{bmatrix}$$

Try and avoid the utilization of loops as much as possible. No loops are necessary.

Hint: Make sure all the array shapes are what you need and expect. You can reshape any numpy array without any tangible computational over-head.
P_y **is log of the prior probabiliy.**

In [44]:

```python
    log_py = np.array( [1-sum(train_labels)/len(train_labels), sum(train_labels)/len(t
    #log_py.shape
    #sum(train_labels)/len(train_labels)
    log_py
```

Out[44]:

```
array([[0.65960912],
       [0.34039088]])
```

In [45]:

```python
def log_prior(train_labels):

    log_py = np.array( [np.log(1-sum(train_labels)/len(train_labels)), np.log(sum(tra
    assert log_py.shape == (2,1)

    return log_py
```

In [46]:

```python
# Performing sanity checks on your implementation
some_labels = np.array([0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1])
some_log_py = log_prior(some_labels)
assert np.array_equal(some_log_py.round(3), np.array([[-0.916], [-0.511]]))

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

In [47]:

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

In [48]:

```python
log_py = log_prior(train_labels)
log_py
```

Out[48]:

```
array([[-0.41610786],
       [-1.07766068]])
```

Task 2

Write a function `cc_mean_ignore_missing` that takes the numpy arrays `train_features` and `train_labels` as input, and outputs the following matrix with the shape $(8, 2)$, where 8 is the number of features.

$$\mu_y = \begin{bmatrix} \mathbb{E}[x^{(0)}|y=0] & \mathbb{E}[x^{(0)}|y=1] \\ \mathbb{E}[x^{(1)}|y=0] & \mathbb{E}[x^{(1)}|y=1] \\ \cdots & \cdots \\ \mathbb{E}[x^{(7)}|y=0] & \mathbb{E}[x^{(7)}|y=1] \end{bmatrix}$$

Some points regarding this task:

- The `train_features` numpy array has a shape of `(N,8)` where `N` is the number of training data points, and 8 is the number of the features.

- The `train_labels` numpy array has a shape of `(N,)`.

- **You can assume that `train_features` has no missing elements in this task.**

- Try and avoid the utilization of loops as much as possible. No loops are necessary.

Thought process notes:

- need train features where train_labels are 0 vs. 1
- need sum $X^{(i)}$ where y = 0 and count $X^{(i)}$, where y = 1

In [49]:

```python
def cc_mean_ignore_missing(train_features, train_labels):
    N, d = train_features.shape

    E_yeq1 = np.mean(train_features[(train_labels == 1)], axis = 0)
    E_yeq0 = np.mean(train_features[(train_labels == 0)], axis = 0)

    #combine into list, convert to array, transpose
    mu_y = np.array([E_yeq0, E_yeq1]).T
    assert mu_y.shape == (d, 2)
    return mu_y
```

In [50]:

```python
# Performing sanity checks on your implementation
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

some_mu_y = cc_mean_ignore_missing(some_feats, some_labels)

assert np.array_equal(some_mu_y.round(2), np.array([[  2.33,   4.  ],
                                                    [ 96.67, 160.  ],
                                                    [ 68.67,  52.  ],
                                                    [ 17.33,  17.5 ],
                                                    [ 31.33,  84.  ],
                                                    [ 26.77,  33.2 ],
                                                    [  0.27,   1.5 ],
                                                    [ 27.33,  32.5 ]]))

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

In [51]:

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

In [52]:

```python
mu_y = cc_mean_ignore_missing(train_features, train_labels)
mu_y
```

Out[52]:

```
array([[  3.48641975,   4.91866029],
       [109.99753086, 142.30143541],
       [ 68.77037037,  70.66028708],
       [ 19.51358025,  21.97129187],
       [ 66.25679012, 100.55980861],
       [ 30.31703704,  35.1492823 ],
       [  0.42825926,   0.55279904],
       [ 31.57283951,  37.39712919]])
```

Task 3

Write a function `cc_std_ignore_missing` that takes the numpy arrays `train_features` and `train_labels` as input, and outputs the following matrix with the shape $(8, 2)$, where 8 is the number of features.

$$\sigma_y = \begin{bmatrix} \text{std}[x^{(0)}|y=0] & \text{std}[x^{(0)}|y=1] \\ \text{std}[x^{(1)}|y=0] & \text{std}[x^{(1)}|y=1] \\ \cdots & \cdots \\ \text{std}[x^{(7)}|y=0] & \text{std}[x^{(7)}|y=1] \end{bmatrix}$$

Some points regarding this task:

- The `train_features` numpy array has a shape of `(N,8)` where `N` is the number of training data points, and 8 is the number of the features.

- The `train_labels` numpy array has a shape of `(N,)`.

- **You can assume that `train_features` has no missing elements in this task.**

- Try and avoid the utilization of loops as much as possible. No loops are necessary.

In [53]:

```python
def cc_std_ignore_missing(train_features, train_labels):
    N, d = train_features.shape

    std_yeq1 = np.std(train_features[(train_labels == 1)], axis = 0)
    std_yeq0 = np.std(train_features[(train_labels == 0)], axis = 0)
    #combine into list, convert to array, transpose
    sigma_y = np.array([std_yeq0, std_yeq1]).T
    assert sigma_y.shape == (d, 2)

    return sigma_y
```

In [54]:

```python
# Performing sanity checks on your implementation
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

some_std_y = cc_std_ignore_missing(some_feats, some_labels)

assert np.array_equal(some_std_y.round(3), np.array([[ 1.886,  4.   ],
                                                     [13.768, 23.   ],
                                                     [ 3.771, 12.   ],
                                                     [12.499, 17.5  ],
                                                     [44.312, 84.   ],
                                                     [ 1.027,  9.9  ],
                                                     [ 0.094,  0.8  ],
                                                     [ 4.497,  0.5  ]]))

# Checking against the pre-computed test database
test_results = test_case_checker(cc_std_ignore_missing, task_id=3)
assert test_results['passed'], test_results['message']
```

In [55]:

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

In [56]:

```python
sigma_y = cc_std_ignore_missing(train_features, train_labels)
sigma_y
```

Out[56]:

```
array([[  3.1155426 ,   3.75417931],
       [ 25.96811899,  32.50910874],
       [ 18.07540068,  21.69568568],
       [ 15.02320635,  17.21685884],
       [ 95.63339586, 139.24364214],
       [  7.50030986,   6.6625219 ],
       [  0.29438217,   0.37201494],
       [ 11.67577435,  11.01543899]])
```

Task 4

Write a function `log_prob` that takes the numpy arrays `train_features`, μ_y, σ_y, and $\log p_y$ as input, and outputs the following matrix with the shape $(N, 2)$

IN:

$$\log p_y = \begin{bmatrix} \log p(y=0) \\ \log p(y=1) \end{bmatrix}$$

$$\mu_y = \begin{bmatrix} \mathbb{E}[x^{(0)}|y=0] & \mathbb{E}[x^{(0)}|y=1] \\ \mathbb{E}[x^{(1)}|y=0] & \mathbb{E}[x^{(1)}|y=1] \\ \cdots & \cdots \\ \mathbb{E}[x^{(7)}|y=0] & \mathbb{E}[x^{(7)}|y=1] \end{bmatrix}$$

$$\sigma_y = \begin{bmatrix} \text{std}[x^{(0)}|y=0] & \text{std}[x^{(0)}|y=1] \\ \text{std}[x^{(1)}|y=0] & \text{std}[x^{(1)}|y=1] \\ \cdots & \cdots \\ \text{std}[x^{(7)}|y=0] & \text{std}[x^{(7)}|y=1] \end{bmatrix}$$

OUT:

$$\log p_{x,y} = \begin{bmatrix} \left[\log p(y=0) + \sum_{j=0}^{7} \log p(x_1^{(j)}|y=0)\right] & \left[\log p(y=1) + \sum_{j=0}^{7} \log p(x_1^{(j)}|y=1)\right] \\ \left[\log p(y=0) + \sum_{j=0}^{7} \log p(x_2^{(j)}|y=0)\right] & \left[\log p(y=1) + \sum_{j=0}^{7} \log p(x_2^{(j)}|y=1)\right] \\ \cdots & \cdots \\ \left[\log p(y=0) + \sum_{j=0}^{7} \log p(x_N^{(j)}|y=0)\right] & \left[\log p(y=1) + \sum_{j=0}^{7} \log p(x_N^{(j)}|y=1)\right] \end{bmatrix}$$

where

- N is the number of training data points.
- x_i is the i^{th} training data point.

Try and avoid the utilization of loops as much as possible. No loops are necessary.

Hint: Remember that we are modelling $p(x_i^{(j)}|y)$ with a Gaussian whose parameters are defined inside μ_y and σ_y. Write the Gaussian PDF expression and take its natural log **on paper**, then implement it.

Important Note: Do not use third-party and non-standard implementations for computing $\log p(x_i^{(j)}|y)$. Using functions that find the Gaussian PDF, and then taking their log is **numerically unstable**; the Gaussian PDF values can easily become extremely small numbers that cannot be represented using floating point standards and thus would be stored as zero. Taking the log of a zero value will throw an error. On the other hand, it is unnecessary to compute and store $p(x_i^{(j)}|y)$ in order to find $\log p(x_i^{(j)}|y)$; you can write $\log p(x_i^{(j)}|y)$ as a direct function of μ_y, σ_y and the features. This latter approach is numerically stable, and can be applied when the PDF values are much smaller than could be stored using the common standards.

THOUGHT PROCESS
Want:

$$y = argmax_{y \in Y}\left[\sum_i [\log[\frac{1}{\sigma_y\sqrt{2\pi}}] - \frac{1}{2}\frac{(x^{(i)} - \mu_y)^2}{\sigma_y^2}] + \log[P(y)]\right]$$

, where
train_features: (N, 8)
μ_y: (8, 2)
σ_y (8, 2)

Part 1. $\log[P(y)]$ Shape: (2, 1)

Part 2. $\log(\frac{1}{\sigma_y\sqrt{2\pi}})$: Compute using sigma_y. Straight forward. Shape: (8, 2)

Part 3. $-\frac{(x^{(i)} - \mu_y)^2}{2\sigma_y^2}$ Two (5, 8)s

The three variables that need to be used, train_features, mu_y, and sigma_y, have different shapes; (5, 8), (8, 2), and (8, 2), respectively. So you cannot just do train_features - mu_y. mu_y has both y == 0 and y == 1 component. The final answer also has both y == 0 component and y == 1 component. So, take y == 0 component from mu_y to do the math and do another with y == 1 component. The same logic applies to sigma_y during the computation. Shape of the result for one component: (5, 8), same as that of train_features. There are two of these.

Part 4. The final math is: for each component of y, $\sum(Part2 + Part3) + Part1$. Reminder, both Part 1 and Part 2 have y == 0 and y == 1 component.

Part 5. Finally, I put together the final answer by arranging and transposing the two components. i.e. log_p_x_y = np.transpose(np.array([log_p_x_y_0, log_p_x_y_1])). Shape: (5, 2).

In [57]:

```python
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

some_mu_y = cc_mean_ignore_missing(some_feats, some_labels)
some_std_y = cc_std_ignore_missing(some_feats, some_labels)
some_log_py = log_prior(some_labels)
N, d = train_features.shape
#part 2
p2 = np.log((sigma_y * np.sqrt(2*np.pi))**-1) #(8,2)
p2_0 = p2[:,0]
p2_1 = p2[:,1]

#part 3
# -1 * (((8,5) - (8,1))/(8,1))
p3_0 = -1*((some_feats.T - mu_y[:, 0].reshape(d,1))**2  / (2 * some_std_y[:, 0].reshap
p3_1 = -1*((some_feats.T - mu_y[:, 1].reshape(d,1))**2  / (2 * some_std_y[:, 1].reshap

p1_0 = some_log_py[0,0] #scalar
p1_1 = some_log_py[1,0] #scalar

#result should be (N,2)
# part 2: (8,) (d,)
# part 3: (5,8) (N,d)
#                     d,N                 d,1
log_p_x_y = np.array([np.sum(p3_0 + p2_0.reshape(8,1), axis = 0) + p1_0, np.sum(p3_1 +
log_p_x_y.shape
```

Out[57]:

```
(5, 2)
```

In [58]:

```python
def log_prob(train_features, mu_y, sigma_y, log_py):
    N, d = train_features.shape

    #part 2
    p2 = np.log((sigma_y * np.sqrt(2*np.pi))**-1) #(8,2)
    p2_0 = p2[:,0]
    p2_1 = p2[:,1]

    #part 3
    # -1 * (((8,5) - (8,1))/(8,1))
    p3_0 = -1*((some_feats.T - mu_y[:, 0].reshape(d,1))**2  / (2 * some_std_y[:, 0].re
    p3_1 = -1*((some_feats.T - mu_y[:, 1].reshape(d,1))**2  / (2 * some_std_y[:, 1].re

    p1_0 = some_log_py[0,0] #scalar
    p1_1 = some_log_py[1,0] #scalar

    #result should be (N,2)
    # part 2: (8,) (d,)
    # part 3: (5,8) (N,d)
    #                     d,N                 d,1
    log_p_x_y = np.array([np.sum(p3_0 + p2_0.reshape(8,1), axis = 0) + p1_0, np.sum(p

    assert log_p_x_y.shape == (N,2)
    return log_p_x_y
```

In [59]:

```python
# Performing sanity checks on your implementation
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

some_mu_y = cc_mean_ignore_missing(some_feats, some_labels)
some_std_y = cc_std_ignore_missing(some_feats, some_labels)
some_log_py = log_prior(some_labels)

some_log_p_x_y = log_prob(some_feats, some_mu_y, some_std_y, some_log_py)

assert np.array_equal(some_log_p_x_y.round(3), np.array([[ -20.822,  -36.606],
                                                         [ -60.879,  -27.944],
                                                         [ -21.774, -295.68 ],
                                                         [-417.359,  -27.944],
                                                         [ -23.2  ,  -42.6  ]]))

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

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/tmp/ipykernel_60/800464219.py in <module>
     21 # Checking against the pre-computed test database
     22 test_results = test_case_checker(log_prob, task_id=4)
---> 23 assert test_results['passed'], test_results['message']

AssertionError:
*******************************************************    Error in Task 4    *********
*********************************************
-------------------------------------------------------       Summary        ---------
----------------------------------------------
Your code crashed during the evaluation of a test case argument. The rest of this mess
age gives you the following material:
  1. The exception traceback detailing how the error occured.
  2. The specific test case arguments that caused the error.
  3. A code snippet that can conviniently reproduce the error.
    -> You can copy and paste the code snippet into a new cell, and run the cell to r
eproduce the error.

-------------------------------------------------------  Exception Traceback  --------
----------------------------------------------
AssertionError Traceback (most recent call last)
/Jupyter/Notebook/Student/Task/Implementation/Cells)
     21     log_p_x_y = np.array([np.sum(p3_0 + p2_0.reshape(8,1), axis = 0) + p1_0, n
p.sum(p3_1 + p2_1.reshape(8,1), axis = 0) + p1_1]).T #(5,2) = (N,2)
     22
---> 23     assert log_p_x_y.shape == (N,2)
     24     return log_p_x_y

AssertionError:

-------------------------------------------------------  Test Case Arguments  --------
----------------------------------------------
log_py = np.array([[-0.40546511],
       [-1.09861229]])
mu_y = np.array([[-1.34127486, -0.56024191],
       [-3.24418424,  5.6586197 ],
       [-0.92487962, -1.19573902],
       [ 6.53068018,  4.08709238],
       [-1.18911339,  5.30706393],
       [-8.4565552 ,  7.66223767],
       [ 0.99217652, -2.82666943],
       [-7.60190338,  0.38475557]])
sigma_y = np.array([[6.87450118, 8.93814137],
       [5.51200317, 2.61487541],
       [5.11012132, 5.54466   ],
       [1.63257473, 3.17213761],
       [6.33579734, 2.66937004],
       [1.39194346, 0.602003  ],
       [7.54738978, 1.67183637],
       [2.2723157 , 9.35673231]])
train_features = np.array([[-9.49838327,  3.04374429, -6.74039902,  0.91495477,  7.976
43398,
         7.06023467, -4.4985058 ,  9.74148788],
       [ 8.37789946,  8.27349512,  4.34892098,  7.25922999,  2.63769389,
         8.26424067, -1.15483306, -8.97197674],
       [-4.50834093, -9.80100556, -8.52155491,  8.72995625, -9.47867952,
        -9.16549087,  7.16644183, -7.93507627],
       [-8.8655367 , -6.38517908,  5.86102575,  5.32905634, -3.64497922,
        -6.17479696, -7.66996382, -3.81338443],
       [ 9.71569349,  4.89464348, -0.09130156,  4.63850721,  7.93120011,
        -8.60888033,  9.70616698, -9.66130399],
       [-1.70691529, -1.68519579, -0.94768778,  7.42520094,  0.43600506,
        -9.87705265, -5.23393889, -8.99784883]])

------------------------------------------------------------ Reproducing Code Snippet ------
----------------------------------------------
### You can copy the following auto-generated snippet into a new cell to reproduce the
 issue.
### Use the + button on the top left of the screen to insert a new cell below.

####### Test Arguments #######
from copy import deepcopy
failed_arguments = deepcopy(test_results['test_kwargs'])
log_py = failed_arguments['log_py']
mu_y = failed_arguments['mu_y']
sigma_y = failed_arguments['sigma_y']
train_features = failed_arguments['train_features']

####### Your Code Body #######
N, d = train_features.shape

p2 = np.log((sigma_y * np.sqrt(2*np.pi))**-1) #(8,2)
p2_0 = p2[:,0]
p2_1 = p2[:,1]

p3_0 = -1*((some_feats.T - mu_y[:, 0].reshape(d,1))**2  / (2 * some_std_y[:, 0].reshap
e(d,1)**2) ) #(N,8)
p3_1 = -1*((some_feats.T - mu_y[:, 1].reshape(d,1))**2  / (2 * some_std_y[:, 1].reshap
e(d,1)**2) ) #(N,8)

p1_0 = some_log_py[0,0] #scalar
p1_1 = some_log_py[1,0] #scalar

log_p_x_y = np.array([np.sum(p3_0 + p2_0.reshape(8,1), axis = 0) + p1_0, np.sum(p3_1 +
 p2_1.reshape(8,1), axis = 0) + p1_1]).T #(5,2) = (N,2)

assert log_p_x_y.shape == (N,2)
returned_var = log_p_x_y # returned variable

##### Checking Solutions ######
my_solution = returned_var # Your Solution
correct_sol = test_results['correct_sol'] # The Reference Solution
if isinstance(correct_sol, np.ndarray):
    assert isinstance(my_solution, np.ndarray)
    assert my_solution.dtype is correct_sol.dtype
    assert my_solution.shape == correct_sol.shape
    assert np.allclose(my_solution, correct_sol)
    print('If you passed the above assertions, it probably means that you have fixed t
he issue! Well Done!')
    print('Now you have to do 3 things:')
    print('  1) Carefully copy the fixed code body back to the log_prob function.')
    print('  2) If you copied any "returned_var = " lines, convert them back to return
 statements.')
    print('  3) Carefully remove this cell (i.e., the cell you inserted and modified)
 once you are done.')
```

In [20]:

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

In [ ]:

```python
log_p_x_y = log_prob(train_features, mu_y, sigma_y, log_py)
log_p_x_y
```

1.1. Writing the Simple Naive Bayes Classifier

In [ ]:

```python
class NBClassifier():
    def __init__(self, train_features, train_labels):
        self.train_features = train_features
        self.train_labels = train_labels
        self.log_py = log_prior(train_labels)
        self.mu_y = self.get_cc_means()
        self.sigma_y = self.get_cc_std()

    def get_cc_means(self):
        mu_y = cc_mean_ignore_missing(self.train_features, self.train_labels)
        return mu_y

    def get_cc_std(self):
        sigma_y = cc_std_ignore_missing(self.train_features, self.train_labels)
        return sigma_y

    def predict(self, features):
        log_p_x_y = log_prob(features, self.mu_y, self.sigma_y, self.log_py)
        return log_p_x_y.argmax(axis=1)
```

In [ ]:

```python
diabetes_classifier = NBClassifier(train_features, train_labels)
train_pred = diabetes_classifier.predict(train_features)
eval_pred = diabetes_classifier.predict(eval_features)
```

In [ ]:

```python
train_acc = (train_pred==train_labels).mean()
eval_acc = (eval_pred==eval_labels).mean()
print(f'The training data accuracy of your trained model is {train_acc}')
print(f'The evaluation data accuracy of your trained model is {eval_acc}')
```

1.2 Running an off-the-shelf implementation of Naive-Bayes For Comparison

In [22]:

```python
from sklearn.naive_bayes import GaussianNB
```

```python
gnb = GaussianNB().fit(train_features, train_labels)
train_pred_sk = gnb.predict(train_features)
eval_pred_sk = gnb.predict(eval_features)
print(f'The training data accuracy of your trained model is {(train_pred_sk == train_
print(f'The evaluation data accuracy of your trained model is {(eval_pred_sk == eval_
```

```
The training data accuracy of your trained model is 0.7671009771986971
The evaluation data accuracy of your trained model is 0.7532467532467533
```

Part 2 (Building a Naive Bayes Classifier Considering Missing Entries)

In this part, we will modify some of the parameter inference functions of the Naive Bayes classifier to make it able to ignore the NaN entries when inferring the Gaussian mean and stds.

Task 5

Write a function `cc_mean_consider_missing` that

- has exactly the same input and output types as the `cc_mean_ignore_missing` function,
- and has similar functionality to `cc_mean_ignore_missing` except that it can handle and ignore the NaN entries when computing the class conditional means.

You can borrow most of the code from your `cc_mean_ignore_missing` implementation, but you should make it compatible with the existence of NaN values in the features.

Try and avoid the utilization of loops as much as possible. No loops are necessary.

- **Hint**: You may find the `np.nanmean` function useful.

```python
def cc_mean_consider_missing(train_features_with_nans, train_labels):
    N, d = train_features_with_nans.shape

    # your code here
    raise NotImplementedError

    assert not np.isnan(mu_y).any()
    assert mu_y.shape == (d, 2)
    return mu_y
```

```python
# Performing sanity checks on your implementation
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

for i,j in [(0,0), (1,1), (2,3), (3,4), (4, 2)]:
    some_feats[i,j] = np.nan

some_mu_y = cc_mean_consider_missing(some_feats, some_labels)

assert np.array_equal(some_mu_y.round(2), np.array([[  3.  ,   4.  ],
                                                    [ 96.67, 137.  ],
                                                    [ 66.  ,  52.  ],
                                                    [ 14.5 ,  17.5 ],
                                                    [ 31.33,   0.  ],
                                                    [ 26.77,  33.2 ],
                                                    [  0.27,   1.5 ],
                                                    [ 27.33,  32.5 ]]))

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

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```python
mu_y = cc_mean_consider_missing(train_features_with_nans, train_labels)
mu_y
```

Task 6

Write a function `cc_std_consider_missing` that

- has exactly the same input and output types as the `cc_std_ignore_missing` function,
- and has similar functionality to `cc_std_ignore_missing` except that it can handle and ignore the NaN entries when computing the class conditional means.

You can borrow most of the code from your `cc_std_ignore_missing` implementation, but you should make it compatible with the existence of NaN values in the features.

Try and avoid the utilization of loops as much as possible. No loops are necessary.

- **Hint**: You may find the `np.nanstd` function useful.

```python
def cc_std_consider_missing(train_features_with_nans, train_labels):
    N, d = train_features_with_nans.shape

    # your code here
    raise NotImplementedError

    assert not np.isnan(sigma_y).any()
    assert sigma_y.shape == (d, 2)
    return sigma_y
```

```python
# Performing sanity checks on your implementation
some_feats = np.array([[  1. ,  85. ,  66. ,  29. ,   0. ,  26.6,   0.4,  31. ],
                       [  8. , 183. ,  64. ,   0. ,   0. ,  23.3,   0.7,  32. ],
                       [  1. ,  89. ,  66. ,  23. ,  94. ,  28.1,   0.2,  21. ],
                       [  0. , 137. ,  40. ,  35. , 168. ,  43.1,   2.3,  33. ],
                       [  5. , 116. ,  74. ,   0. ,   0. ,  25.6,   0.2,  30. ]])
some_labels = np.array([0, 1, 0, 1, 0])

for i,j in [(0,0), (1,1), (2,3), (3,4), (4, 2)]:
    some_feats[i,j] = np.nan

some_std_y = cc_std_consider_missing(some_feats, some_labels)

assert np.array_equal(some_std_y.round(2), np.array([[ 2.  ,  4.  ],
                                                     [13.77,  0.  ],
                                                     [ 0.  , 12.  ],
                                                     [14.5 , 17.5 ],
                                                     [44.31,  0.  ],
                                                     [ 1.03,  9.9 ],
                                                     [ 0.09,  0.8 ],
                                                     [ 4.5 ,  0.5 ]]))

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

```python
# This cell is left empty as a seperator. You can leave this cell as it is, and you s
```

```python
sigma_y = cc_std_consider_missing(train_features_with_nans, train_labels)
sigma_y
```

2.1. Writing the Naive Bayes Classifier With Missing Data Handling

```python
class NBClassifierWithMissing(NBClassifier):
    def get_cc_means(self):
        mu_y = cc_mean_consider_missing(self.train_features, self.train_labels)
        return mu_y

    def get_cc_std(self):
        sigma_y = cc_std_consider_missing(self.train_features, self.train_labels)
        return sigma_y

    def predict(self, features):
        preds = []
        for feature in features:
            is_num = np.logical_not(np.isnan(feature))
            mu_y_not_nan = self.mu_y[is_num,:]
            std_y_not_nan = self.sigma_y[is_num,:]
            feats_not_nan = feature[is_num].reshape(1,-1)
            log_p_x_y = log_prob(feats_not_nan, mu_y_not_nan, std_y_not_nan, self.log_
            preds.append(log_p_x_y.argmax(axis=1).item())

        return np.array(preds)
```

```python
diabetes_classifier_nans = NBClassifierWithMissing(train_features_with_nans, train_lab
train_pred = diabetes_classifier_nans.predict(train_features_with_nans)
eval_pred = diabetes_classifier_nans.predict(eval_features_with_nans)
```

```python
train_acc = (train_pred==train_labels).mean()
eval_acc = (eval_pred==eval_labels).mean()
print(f'The training data accuracy of your trained model is {train_acc}')
print(f'The evaluation data accuracy of your trained model is {eval_acc}')
```

3. Running SVMlight

In this section, we are going to investigate the support vector machine classification method. We will become familiar with this classification method in week 3. However, in this section, we are just going to observe how this method performs to set the stage for the third week.

`SVMlight` (http://svmlight.joachims.org/) is a famous implementation of the SVM classifier.

`SVMLight` can be called from a shell terminal, and there is no nice wrapper for it in python3. Therefore:

1. We have to export the training data to a special format called `svmlight/libsvm`. This can be done using scikit-learn.
2. We have to run the `svm_learn` program to learn the model and then store it.
3. We have to import the model back to python.

3.1 Exporting the training data to libsvm format

```python
from sklearn.datasets import dump_svmlight_file
dump_svmlight_file(train_features, 2*train_labels-1, 'training_feats.data',
                   zero_based=False, comment=None, query_id=None, multilabel=False)
```

3.2 Training `SVMlight`

```python
!chmod +x ../BasicClassification-lib/svmlight/svm_learn
from subprocess import Popen, PIPE
process = Popen(["../BasicClassification-lib/svmlight/svm_learn", "./training_feats.da
stdout, stderr = process.communicate()
print(stdout.decode("utf-8"))
```

3.3 Importing the SVM Model

```python
from svm2weight import get_svmlight_weights
svm_weights, thresh = get_svmlight_weights('svm_model.txt', printOutput=False)

def svmlight_classifier(train_features):
    return (train_features @ svm_weights - thresh).reshape(-1) >= 0.
```

```python
train_pred = svmlight_classifier(train_features)
eval_pred = svmlight_classifier(eval_features)
```

```python
train_acc = (train_pred==train_labels).mean()
eval_acc = (eval_pred==eval_labels).mean()
print(f'The training data accuracy of your trained model is {train_acc}')
print(f'The evaluation data accuracy of your trained model is {eval_acc}')
```

```python
# Cleaning up after our work is done
!rm -rf svm_model.txt training_feats.data
```