BasicClassification

March 24, 2021

Warning: Make sure this file is named BasicClassification.ipynb on Coursera or the submit button will not work.

1 * 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.
```

2 *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.

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%

3 0. Data

3.1 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.

3.2 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).

3.3 0.3 Loading

[1]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from utils import test_case_checker
```

[2]:
```
df = pd.read_csv('diabetes.csv')
df.head()
```

[2]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

3.4 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.

[3]:
```
# Let's generate the split ourselves.
np_random = np.random.RandomState(seed=12345)
rand_unifs = np_random.uniform(0,1,size=df.shape[0])
division_thresh = np.percentile(rand_unifs, 80)
train_indicator = rand_unifs < division_thresh
eval_indicator = rand_unifs >= division_thresh
```

[4]:
```
train_df = df[train_indicator].reset_index(drop=True)
train_features = train_df.loc[:, train_df.columns != 'Outcome'].values
train_labels = train_df['Outcome'].values
train_df.head()
```

[4]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            1       85             66             29        0  26.6
1            8      183             64              0        0  23.3
```

```
2            1      89             66             23      94  28.1
3            0     137             40             35     168  43.1
4            5     116             74              0       0  25.6

   DiabetesPedigreeFunction  Age  Outcome
0                     0.351   31        0
1                     0.672   32        1
2                     0.167   21        0
3                     2.288   33        1
4                     0.201   30        0
```

[5]:
```
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()
```

[5]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            3       78             50             32       88  31.0
2           10      168             74              0        0  38.0
3            0      118             84             47      230  45.8
4            7      107             74              0        0  29.6

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.248   26        1
2                     0.537   34        1
3                     0.551   31        1
4                     0.254   31        1
```

[6]:
```
train_features.shape, train_labels.shape, eval_features.shape, eval_labels.shape
```

[6]:
```
((614, 8), (614,), (154, 8), (154,))
```

3.5 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.

[7]:
```
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, np.nan)
    eval_df_with_nans[col_with_nans] = eval_df_with_nans[col_with_nans].
 ↪replace(0, np.nan)
train_features_with_nans = train_df_with_nans.loc[:, train_df_with_nans.columns␣
 ↪!= 'Outcome'].values
eval_features_with_nans = eval_df_with_nans.loc[:, eval_df_with_nans.columns !=␣
 ↪'Outcome'].values
```

[9]:
```
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 the data.')
print('')
nan_training_data = train_df_with_nans[train_df_with_nans.isna().any(axis=1)]
nan_training_data
```

```
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 data.
```

[9]:
```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
1              8      183           64.0            NaN        0  23.3
4              5      116           74.0            NaN        0  25.6
5             10      115            NaN            NaN        0  35.3
7              8      125           96.0            NaN        0   NaN
8              4      110           92.0            NaN        0  37.6
..           …        …            …              …      …    …
598            6      162           62.0            NaN        0  24.3
599            4      136           70.0            NaN        0  31.2
605            1      106           76.0            NaN        0  37.5
606            6      190           92.0            NaN        0  35.5
612            1      126           60.0            NaN        0  30.1

     DiabetesPedigreeFunction  Age  Outcome
1                       0.672   32        1
4                       0.201   30        0
5                       0.134   29        0
7                       0.232   54        1
8                       0.191   30        0
..                        …  …      …
598                     0.178   50        1
599                     1.182   22        1
605                     0.197   26        0
606                     0.278   66        1
```

```
612                     0.349   47        1

[186 rows x 9 columns]
```

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

Consider a single sample $(\mathbf{x}, y)$, 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 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`.

5 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.

[87]:
```python
def log_prior(train_labels):

    # your code here
    num_x = 0
    num_y = 0
    for label in train_labels:
        if label == 0:
            num_x += 1
        else:
            num_y += 1
    log_py = np.log([[num_x/len(train_labels)],[num_y/len(train_labels)]])
    return log_py
```

[88]:
```
# 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']
```

[89]:
```
# The following are hints to make your life easier duing debugging if you␣
 ↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
 ↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
 ↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
 ↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
 ↪solution.
```

[90]:
```
log_py = log_prior(train_labels)
log_py
```

[90]:
```
array([[-0.41610786],
       [-1.07766068]])
```

6 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.

[83]:
```
def cc_mean_ignore_missing(train_features, train_labels):
    N, d = train_features.shape

    # your code here
    pos = np.sum(train_labels)
    neg = N - pos
    train_labels = train_labels.reshape(-1,1)

    positive = train_features * train_labels
    train = (train_labels == 0)
    pos_average = np.sum(positive, axis=0) / pos
    pos_average = pos_average.reshape(-1,1)
    negative = train_features * train
    neg_average = np.sum(negative, axis=0) / neg
    neg_average = neg_average.reshape(-1,1)
    mu_y = np.hstack((neg_average, pos_average))

    return mu_y
```

[84]:
```
# 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']
```

[85]:
```
# The following are hints to make your life easier duing debugging if you␣
 ↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
 ↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
 ↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
 ↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
 ↪solution.
```

[86]:
```
mu_y = cc_mean_ignore_missing(train_features, train_labels)
mu_y
```

[86]:
```
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]])
```

7 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.

[79]:
```python
def cc_std_ignore_missing(train_features, train_labels):
    N, d = train_features.shape

    # your code here
    positive_rows = train_labels == 1
    negative_rows = train_labels == 0
    positive = train_features[positive_rows,:]
    negative = train_features[negative_rows,:]

    pos = np.std(positive, axis = 0)
    neg = np.std(negative, axis = 0)
    sigma_y = np.column_stack((neg, pos))

    return sigma_y
```

[80]:
```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']
```

[81]:
```
# The following are hints to make your life easier duing debugging if you␣
↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
↪solution.
```

[82]:
```
sigma_y = cc_std_ignore_missing(train_features, train_labels)
sigma_y
```

[82]:
```
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]])
```

8 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 $(N, 2)$

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

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.

[75]:
```
def log_prob(train_features, mu_y, sigma_y, log_py):
    N, d = train_features.shape

    # your code here
    first = np.sum(np.log(1 / (sigma_y.T * (np.sqrt(2 * np.pi)))), axis = 1)
    first_neg = np.power(train_features - mu_y.T[0,:],2) / (2* np.power(␣
↪sigma_y.T[0,:],2))
    log_neg = np.sum(first_neg, axis=1)
    log_neg -= first[0] + log_py[0]
    first_pos = np.power(train_features - mu_y.T[1,:],2) / (2* np.power(␣
↪sigma_y.T[1,:],2))
    log_pos = np.sum(first_pos, axis=1)
    log_pos -= first[1] + log_py[1]
    log_p_x_y = -np.column_stack((log_neg, log_pos))
    return log_p_x_y
```

[76]:
```
# 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']
```

[77]:
```
# The following are hints to make your life easier duing debugging if you␣
↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
↪solution.
```

[78]:
```
log_p_x_y = log_prob(train_features, mu_y, sigma_y, log_py)
log_p_x_y
```

[78]:
```
array([[-26.84275126, -31.03978768],
       [-33.43017232, -32.45548495],
       [-27.22410142, -31.75514973],
       …,
       [-26.33846452, -29.27426699],
       [-29.11688534, -31.15736336],
       [-26.72282878, -30.71860316]])
```

8.1 1.1. Writing the Simple Naive Bayes Classifier

[35]:
```
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, mu_y, sigma_y, log_py)
        return log_p_x_y.argmax(axis=1)
```

[36]:
```
diabetes_classifier = NBClassifier(train_features, train_labels)
train_pred = diabetes_classifier.predict(train_features)
eval_pred = diabetes_classifier.predict(eval_features)
```

[37]:
```
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}')
```

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

8.2 1.2 Running an off-the-shelf implementation of Naive-Bayes For Comparison

[38]:
```
from sklearn.naive_bayes import GaussianNB
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_labels).mean()}')
```

```
print(f'The evaluation data accuracy of your trained model is {(eval_pred_sk ==␣
↪eval_labels).mean()}')
```

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

9 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.

10 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.

[67]:
```
def cc_mean_consider_missing(train_features_with_nans, train_labels):
    N, d = train_features_with_nans.shape

    # your code here
    train_labels = train_labels.T
    train_labels_p = train_labels == 1
    train_labels_n =  train_labels == 0
    positive = train_features_with_nans[train_labels_p, :]
    p_mean = np.nanmean(positive, axis=0)
    p_mean = p_mean.reshape(-1,1)
    negative = train_features_with_nans[train_labels_n, :]
    n_mean = np.nanmean(negative, axis=0)
    n_mean = n_mean.reshape(-1,1)
    mu_y = np.hstack((n_mean, p_mean))

    return mu_y
```

[68]:
```
# 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']
```

[69]:
```
# The following are hints to make your life easier duing debugging if you␣
 ↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
 ↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
 ↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
 ↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
 ↪solution.
```

[70]:
```
mu_y = cc_mean_consider_missing(train_features_with_nans, train_labels)
mu_y
```

```
[70]: array([[  3.48641975,   4.91866029],
       [109.99753086, 142.30143541],
       [ 71.41538462,  75.34693878],
       [ 27.53658537,  32.11188811],
       [ 66.25679012, 100.55980861],
       [ 30.85025126,  35.31826923],
       [  0.42825926,   0.55279904],
       [ 31.57283951,  37.39712919]])
```

11 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.

```
[108]: def cc_std_consider_missing(train_features_with_nans, train_labels):
           N, d = train_features_with_nans.shape

           # your code here
           rows_p = train_labels == 1
           rows_n = train_labels == 0
           positive = train_features_with_nans[rows_p,:]
           negative = train_features_with_nans[rows_n,:]

           pos = np.nanstd(positive, axis = 0)
           neg = np.nanstd(negative, axis = 0)
           sigma_y = np.column_stack((neg, pos))
           return sigma_y
```

```
[109]: # 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']
```

[110]:
```
# The following are hints to make your life easier duing debugging if you␣
↪failed the pre-computed tests.
#
#   When an error is raised in checking against the pre-computed test database:
#
#     0. test_results will be a python dictionary, with the bug information␣
↪stored in it. Don't be afraid to look into it!
#
#     1. You can access the failed test arguments by reading␣
↪test_results['test_kwargs']. test_results['test_kwargs'] will be
#        another python dictionary with its keys being the argument names and␣
↪the values being the argument values.
#
#     2. test_results['correct_sol'] will contain the correct solution.
#
#     3. test_results['stu_sol'] will contain your implementation's returned␣
↪solution.
```

[111]:
```
sigma_y = cc_std_consider_missing(train_features_with_nans, train_labels)
sigma_y
```

[111]:
```
array([[  3.1155426 ,   3.75417931],
       [ 25.96811899,  32.50910874],
       [ 12.26342359,  12.1982786 ],
       [  9.87753687,  10.37284304],
       [ 95.63339586, 139.24364214],
       [  6.38703834,   6.21564813],
       [  0.29438217,   0.37201494],
       [ 11.67577435,  11.01543899]])
```

11.1 2.1. Writing the Naive Bayes Classifier With Missing Data Handling

```
[105]: 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
```

```
[106]: diabetes_classifier_nans = NBClassifierWithMissing(train_features_with_nans,␣
        ↪train_labels)
       train_pred = diabetes_classifier_nans.predict(train_features_with_nans)
       eval_pred = diabetes_classifier_nans.predict(eval_features_with_nans)
```

```
[112]: 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}')
```

```
The training data accuracy of your trained model is 0.7231270358306189
The evaluation data accuracy of your trained model is 0.7012987012987013
```

12 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.

12.1 3.1 Exporting the training data to libsvm format

```
[113]: 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)
```

12.2 3.2 Training SVMlight

[114]:
```
!chmod +x ./svmlight/svm_learn
from subprocess import Popen, PIPE
process = Popen(["./svmlight/svm_learn", "./training_feats.data", "svm_model.
 ↪txt"], stdout=PIPE, stderr=PIPE)
stdout, stderr = process.communicate()
print(stdout.decode("utf-8"))
```

```
chmod: changing permissions of './svmlight/svm_learn': Operation not permitted
Scanning examples…done
Reading examples into memory…100..200..300..400..500..600..OK. (614 examples
read)
Setting default regularization parameter C=0.0000
Optimizing…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…
…done. (1781 iterations)
Optimization finished (141 misclassified, maxdiff=0.00099).
Runtime in cpu-seconds: 0.19
Number of SV: 375 (including 369 at upper bound)
L1 loss: loss=335.23204
Norm of weight vector: |w|=0.03179
Norm of longest example vector: |x|=871.75350
Estimated VCdim of classifier: VCdim<=769.24695
Computing XiAlpha-estimates…done
Runtime for XiAlpha-estimates in cpu-seconds: 0.00
XiAlpha-estimate of the error: error<=60.75% (rho=1.00,depth=0)
XiAlpha-estimate of the recall: recall=>10.53% (rho=1.00,depth=0)
```

```
XiAlpha-estimate of the precision: precision=>10.58% (rho=1.00,depth=0)
Number of kernel evaluations: 71356
Writing model file…done
```

12.3 3.3 Importing the SVM Model

[115]:
```
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.
```

[116]:
```
train_pred = svmlight_classifier(train_features)
eval_pred = svmlight_classifier(eval_features)
```

[117]:
```
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}')
```

```
The training data accuracy of your trained model is 0.7703583061889251
The evaluation data accuracy of your trained model is 0.7402597402597403
```

[ ]:
```

```