Homework 5 - Naive Bayes

Make sure you have downloaded:

- heart_processed.csv

This homework will ask you to implement naive bayes using a custom likelihood and then comparing it against sklearn's Gaussian naive Bayes.

The execution is slightly different from lecture and section.

- It is more streamlined to take adavantage of vector multiplications and numpy functions, which has its own benefits if we want to scale up our naive bayes prediction to higher dimensions.
- However, you may need to familiarize yourself with the "dictionary" data structure.

Before attempting this homework, make sure you understand the broad strokes of naive Bayes. This will make your coding and debugging much smoother.

0 Data

Load `heart_processed.csv` from the Heart Failure Clinical Records Dataset It contains various predictors (which are in log-scale) for predicting the event of death `DEATH_EVENT`.

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Before submitting your homework, remember to set:

- random_state = 0

In [ ]:
```
dataset = pd.read_csv("heart_processed_log.csv", index_col=0)
X = dataset.drop("DEATH_EVENT", axis=1).values
y = dataset["DEATH_EVENT"].values

# split the data into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar

# print the shapes of the training and testing sets
print('train shapes:')
print('\t X_train ->', X_train.shape)
print('\t y_train ->', y_train.shape)
print('test shapes:')
print('\t X_test ->', X_test.shape)
print('\t y_test ->', y_test.shape)

display(dataset)
```

```
train shapes:
	 X_train -> (209, 6)
	 y_train -> (209,)
test shapes:
	 X_test -> (90, 6)
	 y_test -> (90,)
```

	age	creatinine_phosphokinase	ejection_fraction	platelets	serum_creatinine	se
0	4.317488	6.366470	2.995732	12.487485	0.641854	
1	4.007333	8.969669	3.637586	12.481270	0.095310	
2	4.174387	4.983607	2.995732	11.995352	0.262364	
3	3.912023	4.709530	2.995732	12.254863	0.641854	
4	4.174387	5.075174	2.995732	12.697715	0.993252	
...	...	...	...	...	...	
294	4.127134	4.110874	3.637586	11.951180	0.095310	
295	4.007333	7.506592	3.637586	12.506177	0.182322	
296	3.806662	7.630461	4.094345	13.517105	-0.223144	
297	3.806662	7.788626	3.637586	11.849398	0.336472	
298	3.912023	5.278115	3.806662	12.886641	0.470004	

299 rows × 7 columns

Recall: naive Bayes is choosing the class k, C_k, that maximizes the posterior

$$P(C_k|\,\boldsymbol{x}) = \frac{\pi(C_k)\,\mathcal{L}_{\boldsymbol{x}}(C_k)}{Z}.$$

Hence, we maximize the numerator + assume that all d features x_i are independent ("naive-ness"). So we want to find the k that satisfies

$$\max_k \pi(C_k)\,\mathcal{L}_{\boldsymbol{x}}(C_k) \quad = \quad \max_k \left(\pi(C_k)\prod_{i=1}^{d} p(x_i|C_k)\right).$$

1 Custom Naive Bayes Classifier with KDE

You will create a naive Bayes classifier:

- using the training data

- with KDE to approximate the likelihood
- with bernoulli as the prior

Use only the training data `X_train, y_train` to fit the naive Bayes classifier.

1.1 Prior

1. [2 pt] Compute `prior`, a two element array.
   - prior[0] is the probability of death event 0, $\pi(C_0)$
   - prior[1] is the probability of death event 1, $\pi(C_1)$
   - You should construct the prior probabilities based on frequency of death events from the training data.
   - Tip: Use np.unique() with return_counts.
2. [1 pt] Print `prior`.

In [ ]:
```
unique, counts = np.unique(y_train, return_counts=True)
prior = counts/len(y_train)

print('The prior probabilities are:', prior)
```

```
The prior probabilities are: [0.67464115 0.32535885]
```

1.2 Likelihood (KDE)

1. [2 pt] Define dictionaries `kde0` and `kde1` which fulfill the following:
   - kde0[i] corresponds to the kde object (created by calling `scipy.stats.gaussian_kde`) for feature i when death event is 0. kde1[i] defined likewise.
   - Make sure you index the correct rows of `X_train` when defining kdes.
   - Use bandwidth method 'scott'. (For fun, you can try 'silverman' and see what difference in result you get.)
   - As with all arrays you throw into sklearn or scipy, you may need to take transposes.

In [ ]:
```
from scipy.stats import gaussian_kde
kde0 = {}
kde1 = {}

for i in range(X_train.shape[1]):
    data0 = X_train[y_train == 0, i].T
    data1 = X_train[y_train == 1, i].T

    kde0[i] = gaussian_kde(data0, bw_method='scott')
    kde1[i] = gaussian_kde(data1, bw_method='scott')

# display(kde1) # Use this to check what you made. swap kde0 for kde1 if you
"""
display(kde1)
```

```
feature_index = 1 # which feature/col you want to examine

x_grid = np.linspace(min(X_train[:, feature_index]), max(X_train[:, feature_

kde0_evaluated = kde0[feature_index].evaluate(x_grid)
kde1_evaluated = kde1[feature_index].evaluate(x_grid)

plt.figure(figsize=(10, 6))
plt.plot(x_grid, kde0_evaluated, label='Death Event 0', color='blue')
plt.plot(x_grid, kde1_evaluated, label='Death Event 1', color='red')
plt.title(f'KDE of Feature {feature_index}')
plt.xlabel('Feature Value')
plt.ylabel('Density')
plt.legend()
plt.show()
"""
```

Out[ ]:
```
"\ndisplay(kde1)\n\nfeature_index = 1 # which feature/col you want to exami
ne\n\nx_grid = np.linspace(min(X_train[:, feature_index]), max(X_train[:, f
eature_index]), 1000)\n\nkde0_evaluated = kde0[feature_index].evaluate(x_gr
id)\nkde1_evaluated = kde1[feature_index].evaluate(x_grid)\n\nplt.figure(fi
gsize=(10, 6))\nplt.plot(x_grid, kde0_evaluated, label='Death Event 0', col
or='blue')\nplt.plot(x_grid, kde1_evaluated, label='Death Event 1', color
='red')\nplt.title(f'KDE of Feature {feature_index}')\nplt.xlabel('Feature
Value')\nplt.ylabel('Density')\nplt.legend()\nplt.show()\n"
```

2. [2 pt] Complete the code for `compute_likelihood` function.
   - The objects kde0[i] and kde1[i] have a method .pdf(), which you will use when computing the likelihood.
     - Read the documentation to understand how it works.
   - `likelihood0[j]` is the likelihood of seeing j th data $x_j = \left(x_{j1}, \ldots, x_{jd}\right)$ for death event 0, i.e., $L_{x_j}(C_0) = \prod_{i=1}^{d} p(x_{j_i}|C_0)$
   - `likelihood1[j]` defined likewise.
   - You can loop over the kde objects kde[i] to populate the likelihood arrays.

(Your solution shouldn't be very complicated. A working solutions needs only about 5-10 lines of code.)

In [ ]:
```
def compute_likelihood(x, kde0, kde1):
    # input:    x, a (# data) by (# features) array of test data
    #           kde0 and kde1, dictionaries that will be used to compute the
    # output:   likelihood, a (# data) by (# classes) array.
    #           likelihood[j,k] is the likelihood of data j given class k

    # likelihood0[j] is the likelihood of data j given class 0. Analogously
    likelihood0 = np.ones(x.shape[0])    # TODO
    likelihood1 = np.ones(x.shape[0])    # TODO

    for i in range(x.shape[1]):
        pdf0 = kde0[i].pdf(x[:, i])
        pdf1 = kde1[i].pdf(x[:, i])
        likelihood0 *= pdf0
        likelihood1 *= pdf1

    likelihood = np.vstack((likelihood0, likelihood1)).T

    return likelihood
```

1.3 Posterior

1. [2 pt] Complete the code for `compute_posterior` function.
   - It should include calling the function `compute_likelihood`.

In [ ]:
```
def compute_posterior(x, prior, kde0, kde1):
    # input:    x, a (# data) by (# features) array of test data
    #           prior, a 1 by 2 array
    #           kde0 and kde1, kde dictionaries that will be used to compute
    # output:   posterior, a (# data) by (# classes) array

    likelihood = compute_likelihood(x, kde0, kde1)       # TODO

    unnormalized_posterior = likelihood * prior
    posterior  = unnormalized_posterior / unnormalized_posterior.sum(axis =

    return posterior
```

1.4 Combine prior, likelihood, posterior

Now, we are ready to piece all the code we prepared above.

1. [2 pt] Complete the code for `naive_bayes_predict`.
   - Your code should include calling the `compute_posterior` function.
   - Computing y_pred should be a simple one line of code. You may consider using numpy functions that find the index of the largest entry on every row.
2. [1 pt] Complete the code for `print_success_rates`.

In [ ]:
```
def naive_bayes_predict(x, prior, kde0, kde1):
    # input:    x, a (# data) by (# features) array
    #           prior, a 1 by 2 array
    #           kde0 and kde1, kde dictionaries that will be used to compute
    # output:   y_pred, an array of length (# data)

    posterior = compute_posterior(x, prior, kde0, kde1)
    y_pred    = np.argmax(posterior, axis=1)

    return y_pred

def print_success_rates(y_true,y_pred):
    n_success = np.sum(y_true == y_pred)
    n_total   = len(y_true)
    print("Number of correctly labeled points: %d of %d.  Accuracy: %.2f"
        % (n_success, n_total, n_success/n_total))
```

1.5 Predict

1. [1 pt] Use your custom naive Bayes to:
   - predict *TRAINING*
   - print the results with `print_success_rates`

In [ ]:
```
# TODO predict training data and print

y_pred_train = naive_bayes_predict(X_train, prior, kde0, kde1)

print("Training Data:")
print_success_rates(y_train, y_pred_train)
```

```
Training Data:
Number of correctly labeled points: 171 of 209.  Accuracy: 0.82
```

2. [1 pt] Use your custom naive Bayes to:
   - predict *TEST* data
   - print the results with `print_success_rates`

In [ ]:
```
# TODO predict test data and print

y_pred_test = naive_bayes_predict(X_test, prior, kde0, kde1)

print("Test Data:")
print_success_rates(y_test, y_pred_test)
```

```
Test Data:
Number of correctly labeled points: 67 of 90.  Accuracy: 0.74
```

2. sklearn Gaussian naive Bayes

Let's compare our custom naive Bayes with KDE to the sklearn Gaussian naive Bayes.

2.1 Train

1. [1 pt] Fit `gnb` using training data.

In [ ]:
```
# run sklearn's version - read up on differences if interested
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()

gnb.fit(X_train, y_train)
```

```
Out[ ]: ▾ GaussianNB
        GaussianNB()
```

2.2 Predict

1. [1 pt] Use sklearn naive Bayes to:
   - predict *TRAINING* data
   - print the results with `print_success_rates`

```
In [ ]: # TODO predict training data and print

y_pred_train_gnb = gnb.predict(X_train)

print("Training Data (sklearn GaussianNB):")
print_success_rates(y_train, y_pred_train_gnb)
```

```
Training Data (sklearn GaussianNB):
Number of correctly labeled points: 160 of 209.  Accuracy: 0.77
```

2. [1 pt] Use sklearn naive Bayes to:
   - predict *TEST* data
   - print the results with `print_success_rates`

```
In [ ]: # TODO predict test data and print

y_pred_test_gnb = gnb.predict(X_test)

print("Test Data (sklearn GaussianNB):")
print_success_rates(y_test, y_pred_test_gnb)
```

```
Test Data (sklearn GaussianNB):
Number of correctly labeled points: 68 of 90.  Accuracy: 0.76
```

3. Discussion

3.1 random_state = 0

Using random_state=0 and respond to the following questions.

[2 pt] For **custom NB**, what is the difference between the training and test accuracy? Give an explanation for why it might be so.

Ans: The training accuracy was was 0.82 since it predicted 171/209 points, the test accuracy was 0.74 since it predicted 67/90 points. I expect training accuracy to be higher than test accuracy since the model was fit to be on the training set, so it should do pretty well on that set versus test which was separated to evaluate the model.

3.2 change random_state

Now, experiment with a range of random_state and respond to the following question.

[2 pt] Does your responses to 3.1 change? If so, describe how your responses change and why you changed them.

- (You do not need to artificially adjust your response to 3.1 to fit the any new findings you made after changing random_state)

Ans: With the random_state set to 2, they are a lot closer at 0.78 and 0.77 for train and test respectively. And for random_state of 9, 0.79 vs 0.76. So now, my response doesn't change I still consistently find training accuracy to be higher than test accuracy, just closer so random_state of 0 may just be a "better" split at showing a difference in them.

3.3 Choice of model

[2 pt] Compare **test** accuracy results for **custom NB and sklearn GNB**? Which model would you choose to use, and why?

(There is no one right answer to your choice. A reasonable justification for your choice suffices.)

Ans: I believe the sklearn GNB is a better model to choose than custom NB. Although the custom NB compared to sklearn GNB had a much higher training accuracy rate, 0.82 vs 0.77, it had a lower test accuracy at 0.74 vs 0.76. Not only should I try to pick the model that does out better on the held out test data set (as it was held out for the purposes of evaluation), a starker difference in performance between train and test from the custom NB compared to sklearn GNB could imply some overfitting is happening of the model with the split, which gives further reason to prefer sklearn GNB (which may be better at generalization).

Before submitting your hw, set train test split to random_state=0. Restart kernel and rerun all cells.