COMP.4290/5510 Take-home Final Spring 2022

**Take-home Final Exam: Due on 5/2 (M) Midnight. No extension on the final exam.**

1. Suppose a dishonest dealer has two coins, one fair and the other biased. The biased coin has heads with probability of ¼ in each coin toss. Assume that the dealer never switches the coins. Which coin is more likely to have been used when coin tosses produced HTTTHHHTTTTHTHHTT ?

Number of heads = 7

Number of tails = 10

In this space, P(H) = 41.1%

P(T) = 59.9%

Our fair coin has a fair chance of heads or tails while the biased coin is tails favored.

P(X) = P(X|Fake) \* P(Fake) + P(X|Fair) \* P(Fair)

P(X) = (3.43710e-6 \* 0.5) + (7.629e-6 \* 0.5) = 5.533e-6

P(Fake|X) = (3.43710e-6 \* 0.5) / 5.533e-6 = 0.3106

P(Fair|X) = (7.629e-6 \* 0.5) / 5.533e-6 = 0.6894

This shows that there’s a 68% chance that the fair coin is used rather then the biased coin.

1. A two-state HMM is constructed from the measurements shown below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | G | T | C | Mean length |
| P | 2/5 | 2/5 | 1/10 | 1/10 | 10 |
| Y | 3/10 | 3/10 | 1/5 | 1/5 | 10 |

The mean length indicates the average time that the HMM stays in the state. Decode the most likely sequence of states (P/Y) for sequence GGCT.

Both P and Y have the same average time in state. We can see this by adding measurements together for the given sequence.

P = 4 + 4 + 1 + 1 = 10

Y = 3 + 3 + 2 + 2 = 10

While both P and Y have the same average time in their states, Y is the most likely sequence of states.

1. Sequencing: Velvet

Velvet uses de Bruijn graph to do DNA/RNA sequencing. Check out the source code in <http://www.cs.uml.edu/~kim/580/velvet.tgz>.

Describe how the following are implemented in Velvet in details. Include snippets of code, if necessary;

* how reads are stored in memory

The storage of reads depends on the length, type of pairing, and their respective library. There’s no definition in the program of what’s long or short. While reads over 200 bp are considered long, without the correct memory constraints, it can come up short as well. It’s recommended to have at least 12 gb of memory to store reads. longer reads are stored in a more complex data structure then simpler ones. This means the path of longer reads is reconstructed to be stored properly. The amount you can store also depends on the number of reads analyzed simultaneously. The more reads being “read” the more memory required.

switch (seqWriteInfo->m\_hostBuffersInUse) {

    case 1: // buf[0] is being written

        break;

    case 2: // buf[0] and buf[1] are being written, write buf[0] to disk

        if (fseek(seqWriteInfo->m\_pFile, seqWriteInfo->m\_hostBufferFilePos[0], SEEK\_SET) < 0) {

        velvetLog("Unable to seek in CnyUnifiedSeq\n");

        exit(1);

        }

        if (fwrite(seqWriteInfo->m\_pWriteBuffer[0], WRITE\_BUF\_SIZE, 1, seqWriteInfo->m\_pFile) != 1) {

        velvetLog("Unable to write CnyUnifiedSeq\n");

        exit(1);

        }

* how de Bruijn graph is constructed in the program

Each node is represented by a single rectangle, representing a series of overlapping kmers. The sequence of the final nucleotides is the sequence of nodes. The last kmers from these final nucleotides overlap with the starting position of the first nucleotides. Another structure in the graph is unions formed from a node N and its corresponding twin, forming a block that has the changes of node N applied to the twin.

* and how contigs are generated from the graph.

Contigs are produced as a byproduct of the reads, which can be seen on the graph. The length of the contig is directly correlated to the length of the read. The larger our K value the longer the contig becomes. If we wanted to create longer contigs with less reliance on K we would use pair ended reads, as being pair ended tells the algorithm where to read to. Contigs can also be created via error correction from the graphs themselves in their post analysis phase. Contigs can be generated into a graph to patch gaps in a graph that has a larger K value.

// If too long

    if (totalLength >= cutoffLength)

        return false;

    // If isolated snippet:

    if (currentIndex == 0)

        return true;

    // Joined tips

    if (simplePreArcCount\_pg(-currentIndex, preGraph) < 2)

        return false;

1. Data Analysis

Expression levels of 77 proteins in cerebral cortex of 8 classes of control and trisomic mice (Down syndrome) are available in

<https://archive.ics.uci.edu/ml/datasets/Mice+Protein+Expression>

A comparison of protein profiles between t-CS-m (Rescued Leanring) and c-CS-s (Normal Learning) identifies how well, or poorly, profiles in rescued learning resemble those in normal learning in control mice. In the data, delete columns with some blank values. You are free to use packages for data analyses in scikit-learn.org or biopython.org.

1. Do a scatter plot of data with respect to two Principal Components from PCA. Which three proteins have most influence to the classifications and how the regions of data points are separated (or not separated).

GSK3B , CDK5, and KAPK2 have the most influence on the classifications of the proteins. The regions of datapoints are separated with a lower and higher range for each of the proteins. The different components for PCA will produce different results for the scatterplots. Datapoints for each protein have a high and a low region that belong to each. Below is an example scatterplot using the first three classes starting from 0, which includes t-CS-m and c-CS-s regions corresponding 0 lower and 1 upper for t-CS-m and 2 as the lower region and 3 as the upper region for c-CS-s .

Chart, scatter chart

Description automatically generated

1. Use SVM (Support Vector Machine) package of your choice to do the classification and discuss the results in comparison to what you discovered in part a.

When using the SVM, the three proteins are correctly classified into their corresponding regions. The overall score of the SVM for training is 85% and the actual score on the test data is around 83%. It seems that the SVM produces results that are not scaled as much as those in the PCA analysis meaning that the SVM gives a more accurate representation of t-CS-m and c-CS-s. While there’s only classification being done with the scatterplot, the percent accuracy for our scatterplot proved to be much lower, ending around 70%.

In [ ]:

import pandas as pd

import numpy as np

import scipy.stats as stats

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

In [ ]:

df = pd.read\_csv('Data\_Cortex\_Nuclear.csv')

df

Out[ ]:

|  | **MouseID** | **DYRK1A\_N** | **ITSN1\_N** | **BDNF\_N** | **NR1\_N** | **NR2A\_N** | **pAKT\_N** | **pBRAF\_N** | **pCAMKII\_N** | **pCREB\_N** | **...** | **pCFOS\_N** | **SYP\_N** | **H3AcK18\_N** | **EGR1\_N** | **H3MeK4\_N** | **CaNA\_N** | **Genotype** | **Treatment** | **Behavior** | **class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 309\_1 | 0.503644 | 0.747193 | 0.430175 | 2.816329 | 5.990152 | 0.218830 | 0.177565 | 2.373744 | 0.232224 | ... | 0.108336 | 0.427099 | 0.114783 | 0.131790 | 0.128186 | 1.675652 | Control | Memantine | C/S | c-CS-m |
| **1** | 309\_2 | 0.514617 | 0.689064 | 0.411770 | 2.789514 | 5.685038 | 0.211636 | 0.172817 | 2.292150 | 0.226972 | ... | 0.104315 | 0.441581 | 0.111974 | 0.135103 | 0.131119 | 1.743610 | Control | Memantine | C/S | c-CS-m |
| **2** | 309\_3 | 0.509183 | 0.730247 | 0.418309 | 2.687201 | 5.622059 | 0.209011 | 0.175722 | 2.283337 | 0.230247 | ... | 0.106219 | 0.435777 | 0.111883 | 0.133362 | 0.127431 | 1.926427 | Control | Memantine | C/S | c-CS-m |
| **3** | 309\_4 | 0.442107 | 0.617076 | 0.358626 | 2.466947 | 4.979503 | 0.222886 | 0.176463 | 2.152301 | 0.207004 | ... | 0.111262 | 0.391691 | 0.130405 | 0.147444 | 0.146901 | 1.700563 | Control | Memantine | C/S | c-CS-m |
| **4** | 309\_5 | 0.434940 | 0.617430 | 0.358802 | 2.365785 | 4.718679 | 0.213106 | 0.173627 | 2.134014 | 0.192158 | ... | 0.110694 | 0.434154 | 0.118481 | 0.140314 | 0.148380 | 1.839730 | Control | Memantine | C/S | c-CS-m |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **1075** | J3295\_11 | 0.254860 | 0.463591 | 0.254860 | 2.092082 | 2.600035 | 0.211736 | 0.171262 | 2.483740 | 0.207317 | ... | 0.183324 | 0.374088 | 0.318782 | 0.204660 | 0.328327 | 1.364823 | Ts65Dn | Saline | S/C | t-SC-s |
| **1076** | J3295\_12 | 0.272198 | 0.474163 | 0.251638 | 2.161390 | 2.801492 | 0.251274 | 0.182496 | 2.512737 | 0.216339 | ... | 0.175674 | 0.375259 | 0.325639 | 0.200415 | 0.293435 | 1.364478 | Ts65Dn | Saline | S/C | t-SC-s |
| **1077** | J3295\_13 | 0.228700 | 0.395179 | 0.234118 | 1.733184 | 2.220852 | 0.220665 | 0.161435 | 1.989723 | 0.185164 | ... | 0.158296 | 0.422121 | 0.321306 | 0.229193 | 0.355213 | 1.430825 | Ts65Dn | Saline | S/C | t-SC-s |
| **1078** | J3295\_14 | 0.221242 | 0.412894 | 0.243974 | 1.876347 | 2.384088 | 0.208897 | 0.173623 | 2.086028 | 0.192044 | ... | 0.196296 | 0.397676 | 0.335936 | 0.251317 | 0.365353 | 1.404031 | Ts65Dn | Saline | S/C | t-SC-s |
| **1079** | J3295\_15 | 0.302626 | 0.461059 | 0.256564 | 2.092790 | 2.594348 | 0.251001 | 0.191811 | 2.361816 | 0.223632 | ... | 0.187556 | 0.420347 | 0.335062 | 0.252995 | 0.365278 | 1.370999 | Ts65Dn | Saline | S/C | t-SC-s |

1080 rows × 82 columns

In [ ]:

df = df.dropna(how='any', thresh=75)

df = df.fillna(df.mean())

In [ ]:

df.describe()

Out[ ]:

|  | **DYRK1A\_N** | **ITSN1\_N** | **BDNF\_N** | **NR1\_N** | **NR2A\_N** | **pAKT\_N** | **pBRAF\_N** | **pCAMKII\_N** | **pCREB\_N** | **pELK\_N** | **...** | **SHH\_N** | **BAD\_N** | **BCL2\_N** | **pS6\_N** | **pCFOS\_N** | **SYP\_N** | **H3AcK18\_N** | **EGR1\_N** | **H3MeK4\_N** | **CaNA\_N** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | ... | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 | 1077.000000 |
| **mean** | 0.425810 | 0.617102 | 0.319088 | 2.297269 | 3.843934 | 0.233168 | 0.181846 | 3.537109 | 0.212574 | 1.428682 | ... | 0.226754 | 0.157834 | 0.134757 | 0.121528 | 0.131114 | 0.446149 | 0.169620 | 0.183135 | 0.205414 | 1.337442 |
| **std** | 0.249362 | 0.251640 | 0.049383 | 0.347293 | 0.933100 | 0.041634 | 0.027042 | 1.295169 | 0.032587 | 0.466904 | ... | 0.028991 | 0.026469 | 0.023551 | 0.014295 | 0.023020 | 0.066507 | 0.054294 | 0.036302 | 0.048128 | 0.317499 |
| **min** | 0.145327 | 0.245359 | 0.115181 | 1.330831 | 1.737540 | 0.063236 | 0.064043 | 1.343998 | 0.112812 | 0.429032 | ... | 0.155869 | 0.088305 | 0.080657 | 0.067254 | 0.085419 | 0.258626 | 0.079691 | 0.105537 | 0.101787 | 0.586479 |
| **25%** | 0.288121 | 0.473361 | 0.287444 | 2.057411 | 3.155678 | 0.205755 | 0.164595 | 2.479834 | 0.190823 | 1.203665 | ... | 0.206505 | 0.141013 | 0.119972 | 0.110793 | 0.114398 | 0.398036 | 0.133888 | 0.159156 | 0.174167 | 1.081231 |
| **50%** | 0.366378 | 0.565782 | 0.316564 | 2.296546 | 3.760855 | 0.231177 | 0.182302 | 3.326520 | 0.210594 | 1.355846 | ... | 0.224074 | 0.157834 | 0.134757 | 0.121627 | 0.128580 | 0.448585 | 0.169620 | 0.183135 | 0.205414 | 1.316591 |
| **75%** | 0.487711 | 0.698032 | 0.348197 | 2.528481 | 4.440011 | 0.257261 | 0.197418 | 4.481940 | 0.234595 | 1.561316 | ... | 0.241687 | 0.167400 | 0.139290 | 0.131989 | 0.142504 | 0.490805 | 0.187594 | 0.196044 | 0.219237 | 1.585916 |
| **max** | 2.516367 | 2.602662 | 0.497160 | 3.757641 | 8.482553 | 0.539050 | 0.317066 | 7.464070 | 0.306247 | 6.113347 | ... | 0.358289 | 0.282016 | 0.261506 | 0.158748 | 0.256529 | 0.759588 | 0.479763 | 0.360692 | 0.413903 | 2.129791 |

8 rows × 77 columns

In [ ]:

df['class'].value\_counts().plot(kind='bar')

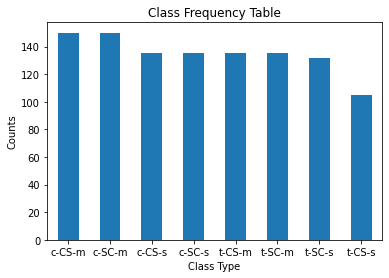
plt.title('Class Frequency Table')

plt.xlabel('Class Type')

plt.xticks(rotation='horizontal')

plt.ylabel('Counts')

plt.show()



In [ ]:

# Drops Null values

mouse\_new = df[df.columns[df.isnull().sum() < 5]].dropna().drop(['MouseID', 'Genotype', 'Treatment', 'Behavior'], axis=1)

mouse\_new.shape

Out[ ]:

(1077, 78)

In [ ]:

df = mouse\_new[(mouse\_new['class'] == 'c-CS-s') | (mouse\_new['class'] == 't-CS-m')].copy()

df.loc[:, 'class'].replace({'c-CS-s': 1, 't-CS-m': 0}, inplace=True)

X = df.drop(['class'], axis=1)

y = df['class']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 42)

svm = SVC(gamma='scale')

svm.fit(X\_train, y\_train)

print(svm.score(X\_train, y\_train))

print(svm.score(X\_test, y\_test))

0.8611111111111112

0.8333333333333334