EEE 485
Statistical Learning and Data Analytics

First Report

Section 01

Muhammed Can Küçükaslan 21901779

Muhammet Hikmet Şimşir 21802880

Contents

Dataset Description

The dataset has 8 features and 10 categoric outputs. We aim to predict the cellular localization sites of proteins in a yeast cell. Features correspond to some scores which are derived from some signal sequence detection methods and some analysis routines. All features are in the range $[0, 1]$. The dataset does not have any missing values. The dataset has an unbalanced data. The distribution of outputs is the following:

Localization Site	No. of instances
CYT (cytosolic or cytoskeletal)	463
NUC (nuclear)	429
MIT (mitochondrial)	244
ME3 (membrane protein, no N-terminal signal)	163
ME2 (membrane protein, uncleaved signal)	51
ME1 (membrane protein, cleaved signal)	44
EXC (extracellular)	37
VAC (vacuolar)	30
POX (peroxisomal)	20
ERL (endoplasmic reticulum lumen)	5

In total, we have 1484 data instances.

Using some external libraries, we made PCA to detect some possible uncorrelated data and observed the change in the predictions of the models. We observed that after reducing the dimension from 8 to 7, the accuracy of the k-NN model has reduced from around 57% to 52%. Therefore, we decided not to implement and use PCA on our data.

Logistic Regression

Review of the Method & Reason to Choose the Model

Since the data we have has 8 continuous features and 10 categorical output, we first attempt to solve the classification problem we have by implementing a Logistic Regression Algorithm. Logistic Regression is a machine learning algorithm commonly used to predict binary output. To achieve this, logistic function is used to map the continuous data to the interval $[0,\ 1]$. After mapping, a threshold value is used to distinguish the case '0' and '1'. If the result of logistic functions is greater than the threshold, we classify this data as '1'. And, '0' for the other case. To achieve this mapping, we need weights (β values) to be determined. These values will be determined by gradient descent algorithm. Gradient descent algorithm, assumes an initial value to β and tries to reach the local, preferably global, minimum of the loss function, when loss function is considered as a function of β. This will be achieved by going in the opposite direction to the gradient function. η (step size) will determine the length of the movement. We start with a high value of gradient and decrease the value of gradient at each iteration of the algorithm. At each iteration, beta value will be updated by the following equality:

$$\beta = \beta - \eta \frac{1}{n} \sum_{i=1}^{n} (p^i - y^i) x^i$$

where n is the number of training data, and p^i is defined as

$$p^i = \text{sig}(\beta^T x^i)$$

Challenges

One challenge we experience is having 10 different categorical outputs. We solve this issue by training different logistic regression models for all outputs, making predictions for all of them, and choosing the output with highest probability.

Simulation Setup

Total data instances is 1485. We use 1200 instances to train the data and remaining 284 instances to test the data. We set η to 1, and decrement it at each iteration by the rule

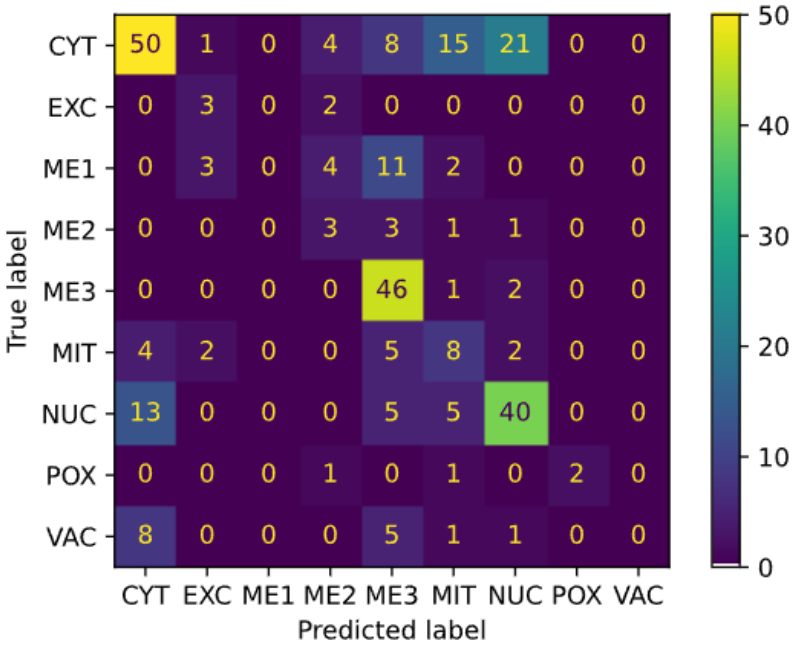
$$\eta = \eta * 0.99$$

so that we achieve the minimum faster.

Simulation Results

This configuration gives 0.5352112676056338 accuracy for 284 instances. It took approximately 2 seconds for preprocessing the data and 1 minute and 50 seconds to train 1200 data instances, and less than 0.1 seconds to test the around 300 data instances. Execution time of the algorithms depends on the initial value of the β, which is random. Also, number of data instance used is a paramater. The most dramatic change occurs with η value. If take η as constant, for example 0.001, the speed of convergence of the gradient increases dramatically. Even after 1000 iteration, we observe a relatively

big gradient. The algorithm stops because max. number of iteration is achieved. Thus, it is essentially important to decrease the η value at each iteration to make the gradient converge to 0. Confusion matrix is the following:



We observe that our model is not capable of predicting the outputs ME1 and VAC correctly. We have 0% accuracy for them. Also, the model has relatively more instances where it predicts MIT and NUC when the true label is CYT. A similar case happens when we predict CYT and the true label is NUC. One can have better understanding of the flaws of the model by examining the following precision table:

	precision	recall	f1-score	support
CYT	0.67	0.51	0.57	99
EXC	0.33	0.60	0.43	5
ME1	1.00	0.00	0.00	20
ME2	0.21	0.38	0.27	8
ME3	0.55	0.94	0.70	49
MIT	0.24	0.38	0.29	21
NUC	0.60	0.63	0.62	63
POX	1.00	0.50	0.67	4
VAC	1.00	0.00	0.00	15
accuracy			0.54	284
macro avg	0.62	0.44	0.39	284
weighted avg	0.63	0.54	0.50	284

K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a simple yet very powerful classification algorithm. The main idea of the KNN is to locate a point on p-dimensional space (where p is the number of parameters) and determine the nearest K points. Then it will identify the most frequent class among those K points, which will be the prediction.

There are two main decisions that must be made with consideration. First one of them is what value should K be, i.e. how to determine optimal value for K. The second one is how we should measure the distance.

Optimal value of K can be found by assigning different values to K and comparing the results. In our manual experiments, with the *Manhattan metric*, we got the highest accuracy rate when K was set to 25. It gave approximately 59.6% correct classification among ten classes, which is much higher than base case probability 10%. Setting K to 11 gave 0.559%, setting K to 13 and 15 gave 57.23%, setting K to 17 gave 57.9%, setting K to 19 gave 56.9%, K=23 gave 57.5%, K=25 gave 59.6%, K=27 gave 58.2%, K=29 gave 56.6%.

However, it is important to emphasize that, for no test input, the model predicted the VAC. There were 15 of VAC in test data but none were predicted correctly, as seen in the figure below. It is also important to notice that there were 30 VAC in the whole dataset, half of them were in the training set (which contains 80% of the dataset). So another important issue here is randomly splitting the training and test data. Beside the split of train and test data set, this error can be attributed to data imbalance between the classes. We will try to overcome these problems in the remaining part of the project.

On the distance metric, we have tested *euclidean, manhattan,* and *chebyshev's* distances. Among them the most successful metric was the *Manhattan metric,* the *euclidean metric* performed slightly worse, and the *chebyshev's metric* performed worse.

Finally, we can use k-fold cross validation to further improve the parameter tuning or selecting the optimal metric for KNN. The cross validation can also be used for other machine learning methods mentioned.

It took approximately 2 seconds for preprocessing the data and 5.8 seconds to test approximately 300 data instances for 25 nearest neighbors, according to manhattan metric, among approximately 1200 training data instances.

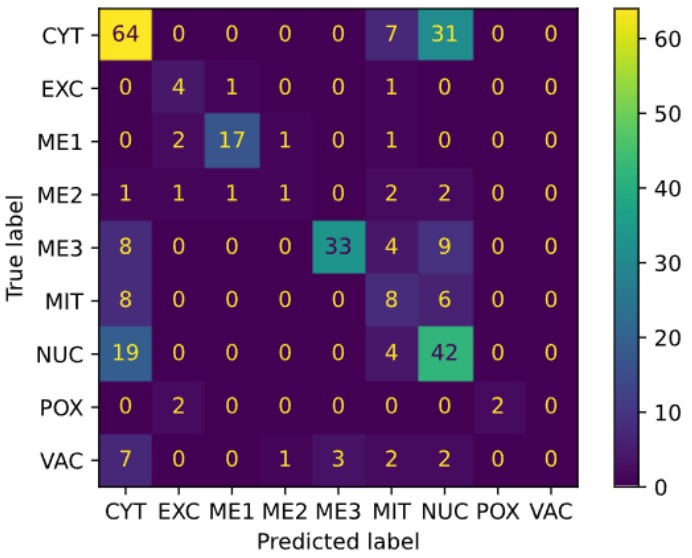


Figure: Confusion Matrix when K=25, with manhattan metric

Naive Bayesian

Review of the Method & Reason to Choose the Model

This model assumes the independence between features and tries to maximize the likelihood function by using Bayes' Rule. Naive Baesian is a simple but powerful model that can give high accuracy rates. For this reason, we are planning to implement this model.

Challenges

We have continuous features but the Naive Bayesian Model requires discrete feature values. To solve this issue we need to map our continuous features to discrete ones. At this point, choosing the correct bin size, the total different number of discrete values for a continuous feature, will be important. We may test prediction results using the *sklearn* libraries for different bin size values. For example, Complement Naive Bayesian gives the following results for test size 0.3:

Bin Size	Accuracy
3	0.5067264573991032
5	0.4304932735426009
10	0.49551569506726456
20	0.5112107623318386

Which Beyesian Implementation?

There are different ways to implement Naive Beysian algorithms. Therefore, before actually implementing it, we test for different implementations using *sklearn* library. We get the following accuracy results for test size 0.3:

Name of the Model	Accuracy
Complement NB	0.5246636771300448
Bernoulli NB	0.3164983164983165
Gaussian NB	0.1531986531986532

It is not surprising that the Complement NB method gives the highest accuracy. This is because this method considers the imbalance in making predictions while others do not. Thus, it is wise to choose implementing Complement NB.

Neural Network Classifier

Neural Network Classifier is the last machine learning model we are planning to implement. More specifically, we want to implement a multilayer perceptron. The main advantage of the Neural Network Classifier is that it practically enables us to model non-linear relations efficiently.

The multilayer perceptron model consists of the input layer, (probably multiple) hidden layers, and the output layer. Each layer also consists of nodes. The values of the nodes of a layer are used with their respective *weights* and *activation functions* to determine the values of nodes in the next layer. Thus, the output of a layer becomes the input of the next layer.

The model is trained by updating the weights according to error rate, after each data input is processed. The updated weights are calculated by a technique called backpropagation that takes advantage of the *chain rule* to simplify calculations.

Deciding on the number of layers, and the number of nodes in each layer requires careful consideration. It is also important to determine an activation function that would fit the relation we want to reveal. We have started implementing it but couldn't complete the interim demos. So we did not actually test for optimal parameters for our dataset. Nevertheless, we used scikit-learn's MLPClassifier as we survey on various machine learning, and we got approximately 60% percent accuracy rate, which was one of the highest results we got. We are planning to include a Multilayered Perceptron Classifier in the final demos.

Gantt Chart

		February 21 2022-February 28 2022	February 28 2022 - April 9 2022	April 11 2022-April 17 2022	April 18 2022-April 25 2022	April 25 2022-May 8 2022
Muhammed Can Küçükaslan						
	Implementing the KNN Algorithm					
	Implementing the Neural Network Algorithm					
	Research PCA and SMOTE					
Muhammet Hikmet Şimşir						
	Implementing the Logistic Regression Algorithm					
	Implementing the Naive Bayesian Algorithm					
	Research for data balancing methods					
	Research Decision Tree					
Together	Searching for project topic and dataset					
	Deciding on ML methods to be used					
	Writing The Proposal					
	Writing The First Report					
	Interim Demo					
	Writing Final Report					
	Final Demo					

Appendices

KNN Implementation in Python

```
# %% [markdown]
# # KNN Algorithm for Yeast Data
#
# %%
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd
from math import floor, ceil, sqrt
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay



# %%
def the_train_test_split(X, test_ratio = 0.2):
    if(test_ratio >= 1 or test_ratio <0):
        test_ratio = 0.2
    row, _ = X.shape
    train_count = floor(row * (1-test_ratio))
    train = X[:train_count]
    test = X[train_count:]
    return train, test

# %%
def euclidean_distance(x,y):
    return sqrt(sum(np.square(x-y)))

def manhattan_distance(x,y):
    return sum(abs(x - y))

def chebyshev_distance(x,y):
    return max(abs(x - y))

def minkowski_distance(x,y):
    return -1;

def get_distance(x, y, algorithm ="euclidean"):
```

```
    if(algorithm == "euclidean"):
        return euclidean_distance(x,y)
    elif(algorithm == "manhattan"):
        return manhattan_distance(x,y)
    elif(algorithm == "chebyshev"):
        return chebyshev_distance(x,y)
    else:
        #print("The algorithm ", algorithm, " couldn't be recognized.\n",
"\"euclidean\" algorithm is used instead")
        return euclidean_distance(x,y)

# %%
class K_Neigbours_Classifier():
    def __init__(self, neigbour_count = 7, algorithm = "euclidean"):
        self.alg = algorithm
        self.n_count = neigbour_count

    def fit(self, train_input, train_output):
        self.train_in = train_input
        self.train_out = train_output
        #
        pd.unique(self.train_out) # since it is array of arrays sized 1
        self.categories = pd.unique(self.train_out.ravel())

    def predict(self, single):
        # calculate the distances
        distances = np.apply_along_axis(get_distance, 1, self.train_in,
y=single, algorithm=self.alg)
        #print(distances)
        nearest_indices = np.argpartition(distances,
self.n_count)[:self.n_count]
        #print(nearest_indices)
        category_dict = dict.fromkeys(self.categories, 0)
        nearest_keys = self.train_out[nearest_indices]
        for neigbour_key in nearest_keys:
            category_dict[neigbour_key] = 1 + category_dict[neigbour_key]
        the_key_with_max = max(category_dict, key=category_dict.get)
        #print("We predict this one to be: ", the_key_with_max)
        return the_key_with_max

# %%
def measure(X_train, Y_train, X_test, Y_test  ):
```

```
    knc = K_Neigbours_Classifier(neigbour_count=25, algorithm="manhattan")
    knc.fit(X_train, Y_train[:,0]) # we know that y_train is 1 dimensional
    correct_pred = 0
    incorrect_pred = 0
    correct_pred_dict = dict.fromkeys(cat,0)
    failed_to_pred_dict = dict.fromkeys(cat,0)
    assumed_to_pred_dict = dict.fromkeys(cat,0)

    predictions = [] #= np.empty(Y_test.size,  dtype="S3")
    for i in range (Y_test.size):
        correct_key = Y_test[i][0]
        predicted_key =knc.predict(X_test[i])
        predictions.append(predicted_key)
        if(  predicted_key== correct_key):
            correct_pred = 1 + correct_pred
            correct_pred_dict[correct_key] = 1 +
correct_pred_dict[correct_key]

        else:
            incorrect_pred = 1 + incorrect_pred
            failed_to_pred_dict[correct_key] = 1 +
failed_to_pred_dict[correct_key]
            assumed_to_pred_dict[predicted_key] = 1 +
assumed_to_pred_dict[predicted_key]

    print("Accuracy: ", correct_pred/(correct_pred + incorrect_pred) )
    print("Number of correct predictions: ", correct_pred)
    print("Number of incorrect predictions: ", incorrect_pred)
    print("correct predict(ion) count:\n", correct_pred_dict)
    print("failed_to predict(ion) count:\n", failed_to_pred_dict)
    print("assumed_to predict(ion) count:\n", assumed_to_pred_dict)


    print("\n                    Classification Report                    \n",
    classification_report(Y_test, predictions, zero_division=1)) # ignores
zero division warning

    ConfusionMatrixDisplay.from_predictions(Y_test, predictions)
    plt.show()


# %% [markdown]
```

```
# ## Read the data

# %%
file_name = "yeast.csv"
md = pd.read_csv(file_name)

# md.dropna(inplace = True)
# md.replace('unknown', 0, inplace = True)
md.head()


# %% [markdown]
# ## Prepare the data
# * Separate the input and output variables
# * Seperate the data into training and test sets
# * Normalize the data
#

# %%
test_ratio = 0.2
X = md.values[:,1:9]
Y = md.values[:,9:]
cat = pd.unique(Y[:,0])

# normalize X:
for i in range(X.shape[1]):
    X[:,i] = (X[:,i] - X[:,i].mean())/X[:,i].std()


# %%
X_train, X_test = the_train_test_split(X, test_ratio = test_ratio)
Y_train, Y_test = the_train_test_split(Y, test_ratio = test_ratio)

# %%
measure(X_train, Y_train, X_test, Y_test)


# %% [markdown]
# ### PCA pretest VIA Sci-Kit LEARN
#

# %%
```

```
# Only for test purposes:
#  from sklearn.decomposition import PCA
# pca = PCA(n_components=7)
# X = pca.fit_transform(X)
#
# X_train, X_test = the_train_test_split(X, test_ratio = test_ratio)
# Y_train, Y_test = the_train_test_split(Y, test_ratio = test_ratio)
#
# measure(X_train, Y_train, X_test, Y_test)
#
```

Logistic Regression Implementation in Python

```
import pandas as pd
import numpy as np
def normalize(X):
    for i in range(X.shape[1]):
        X[:,i] = (X[:,i] - X[:,i].mean())/X[:,i].std()
    return X
k = 10
p = 8
def get_data():
    """
    Returns the data from the xlsx file
    """
    file_name = 'yeast.csv'
    df = pd.read_csv(file_name, index_col=0, header=None)
    # print(df.head()) # print the first 5 rows
    return df
def p(i):
    return 1 / (1 + np.exp(-i))

md = get_data()
md.dropna(inplace = True)
md.replace('?', 0, inplace = True)
X = md.iloc[:, 0:8].values.reshape(-1, 8)
X = normalize(X)

M = X.shape[0]
Y = md.iloc[:, 8:9].values.reshape(-1, 1)
outputs = ['CYT', 'NUC', 'MIT', 'ME3', 'ME2', 'ME1', 'EXC', 'VAC', 'POX',
'ERL']

y = np.zeros((len(Y), 10))
for i in range(len(Y)):
        for j in range(10):
            if outputs[j] == Y[i]:
                  y[i][j] = 1
            else:
                  y[i][j] = 0
eta = 1
beta = np.zeros((10, 8))
```

```
for i in range(10):
    beta[i] = np.random.rand(1, 8)

gradient = np.zeros((10, 8))
trainNo = 1000

n = 0
while (True):
    for i in range(trainNo):
        tmp = p(X[i] @ beta.T)
        tmp = tmp - y[i]
        for t in range(k):
            gradient[t] = gradient[t] + tmp[t] * X[i]
    gradient = gradient / trainNo

    beta = beta - eta * gradient
    # print('g: ', gradient, 'norm', np.linalg.norm(gradient))

    n = n + 1
    if n == trainNo or np.linalg.norm(gradient) < 0.0097:
        break
    eta = eta * 0.99

# print('beta: ', beta)
# one or not classification
y_pred = np.zeros(M - trainNo)
estimates = np.zeros(10)

for i in range(M - trainNo):
    for t in range(10):
        estimates[t] = p(np.dot(X[trainNo + i], beta[t]))
    y_pred[i] = np.argmax(estimates)
    # print(estimates)
    # print(x_pred[i])
# print(y_pred)
# print(type(y_pred[0]))
# for i in range(M - trainNo):
#     print(outputs[int(y_pred[i])], end=' ')
# print()

correct = 0
for i in range(M - trainNo):
```

```
    if np.where(y[trainNo + i] == 1) == y_pred[i]:
        correct = correct + 1
print("accuracy: %1.5f test size %5d" %(correct / (M - trainNo), (M -
trainNo)))
```