Lab6: Gene Expression Data$_dht258$

October 6, 2020

1 Lab: Logistic Regression for Gene Expression Data

In this lab, we use logistic regression to predict biological characteristics ("phenotypes") from gene expression data. In addition to the concepts in breast cancer demo, you will learn to: * Handle missing data * Perform multi-class logistic classification * Create a confusion matrix * Use L1-regularization for improved estimation in the case of sparse weights (Grad students only)

1.1 Background

Genes are the basic unit in the DNA and encode blueprints for proteins. When proteins are synthesized from a gene, the gene is said to "express". Micro-arrays are devices that measure the expression levels of large numbers of genes in parallel. By finding correlations between expression levels and phenotypes, scientists can identify possible genetic markers for biological characteristics.

The data in this lab comes from:

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

In this data, mice were characterized by three properties: * Whether they had down's syndrome (trisomy) or not * Whether they were stimulated to learn or not * Whether they had a drug memantine or a saline control solution.

With these three choices, there are 8 possible classes for each mouse. For each mouse, the expression levels were measured across 77 genes. We will see if the characteristics can be predicted from the gene expression levels. This classification could reveal which genes are potentially involved in Down's syndrome and if drugs and learning have any noticeable effects.

1.2 Load the Data

We begin by loading the standard modules.

[20]:

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import linear_model, preprocessing
```

Use the `pd.read_excel` command to read the data from

https://archive.ics.uci.edu/ml/machine-learning-databases/00342/Data_Cortex_Nuclear.xls into a dataframe df. Use the index_col option to specify that column 0 is the index. Use the df.head() to print the first few rows.

```
[21]: # TODO
      df = pd.read_excel("https://archive.ics.uci.edu/ml/machine-learning-databases/
       ↪00342/Data_Cortex_Nuclear.xls",na_values='?',index_col=0)
      df.head()
```

```
[21]:          DYRK1A_N   ITSN1_N    BDNF_N     NR1_N    NR2A_N    pAKT_N   pBRAF_N  \
      MouseID
      309_1    0.503644  0.747193  0.430175  2.816329  5.990152  0.218830  0.177565
      309_2    0.514617  0.689064  0.411770  2.789514  5.685038  0.211636  0.172817
      309_3    0.509183  0.730247  0.418309  2.687201  5.622059  0.209011  0.175722
      309_4    0.442107  0.617076  0.358626  2.466947  4.979503  0.222886  0.176463
      309_5    0.434940  0.617430  0.358802  2.365785  4.718679  0.213106  0.173627

               pCAMKII_N   pCREB_N    pELK_N  ...   pCFOS_N     SYP_N  H3AcK18_N  \
      MouseID                                 ...
      309_1     2.373744  0.232224  1.750936  ...  0.108336  0.427099   0.114783
      309_2     2.292150  0.226972  1.596377  ...  0.104315  0.441581   0.111974
      309_3     2.283337  0.230247  1.561316  ...  0.106219  0.435777   0.111883
      309_4     2.152301  0.207004  1.595086  ...  0.111262  0.391691   0.130405
      309_5     2.134014  0.192158  1.504230  ...  0.110694  0.434154   0.118481

                 EGR1_N  H3MeK4_N    CaNA_N  Genotype  Treatment  Behavior   class
      MouseID
      309_1    0.131790  0.128186  1.675652   Control  Memantine       C/S  c-CS-m
      309_2    0.135103  0.131119  1.743610   Control  Memantine       C/S  c-CS-m
      309_3    0.133362  0.127431  1.926427   Control  Memantine       C/S  c-CS-m
      309_4    0.147444  0.146901  1.700563   Control  Memantine       C/S  c-CS-m
      309_5    0.140314  0.148380  1.839730   Control  Memantine       C/S  c-CS-m

      [5 rows x 81 columns]
```

This data has missing values. The site:

http://pandas.pydata.org/pandas-docs/stable/missing_data.html

has an excellent summary of methods to deal with missing values. Following the techniques there, create a new data frame df1 where the missing values in each column are filled with the mean values from the non-missing values.

```
[22]: # TODO
      df1 = df.fillna(df.mean())
```

1.3 Binary Classification for Down's Syndrome

We will first predict the binary class label in df1['Genotype'] which indicates if the mouse has Down's syndrome or not. Get the string values in df1['Genotype'].values and convert this to a numeric vector y with 0 or 1. You may wish to use the np.unique command with the return_inverse=True option.

[30]:

```
# TODO
y_values = df1['Genotype'].values
y= (y_values == 'Control').astype(int)
y = np.unique(y_values, return_inverse=True)[1]
print(y)
```

```
[0 0 0 ... 1 1 1]
```

As predictors, get all but the last four columns of the dataframes. Store the data matrix into X and the names of the columns in xnames.

[31]:

```
# TODO
xnames=df1.columns[:-4]
X=np.array(df1[xnames].values)
```

Split the data into training and test with 30% allocated for test. You can use the train

[32]:

```
from sklearn.model_selection import train_test_split

# TODO:
Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.3)
```

Scale the data with the StandardScaler. Store the scaled values in Xtr1 and Xts1.

[33]:

```
from sklearn.preprocessing import StandardScaler

# TODO
scal = StandardScaler()
Xtr1 = scal.fit_transform(Xtr)
Xts1 = scal.transform(Xts)
```

Create a LogisticRegression object logreg and fit on the scaled training data. Set the regularization level to C=1e5 and use the optimizer solver=liblinear.

[34]:

```
# TODO
#logreg = linear_model.LogisticRegression(C=1e5)
#logreg.fit(Xs, y)
logreg= linear_model.LogisticRegression(C=1e5, solver='liblinear')
logreg.fit(Xtr1,ytr)
```

[34]:

```
LogisticRegression(C=100000.0, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

Measure the accuracy of the classifer on test data. You should get around 94%.

[35]:

```
# TODO
yhat = logreg.predict(Xts1)
acc = np.mean(yhat == yts)
print("Accuracy = %f " % acc)
```

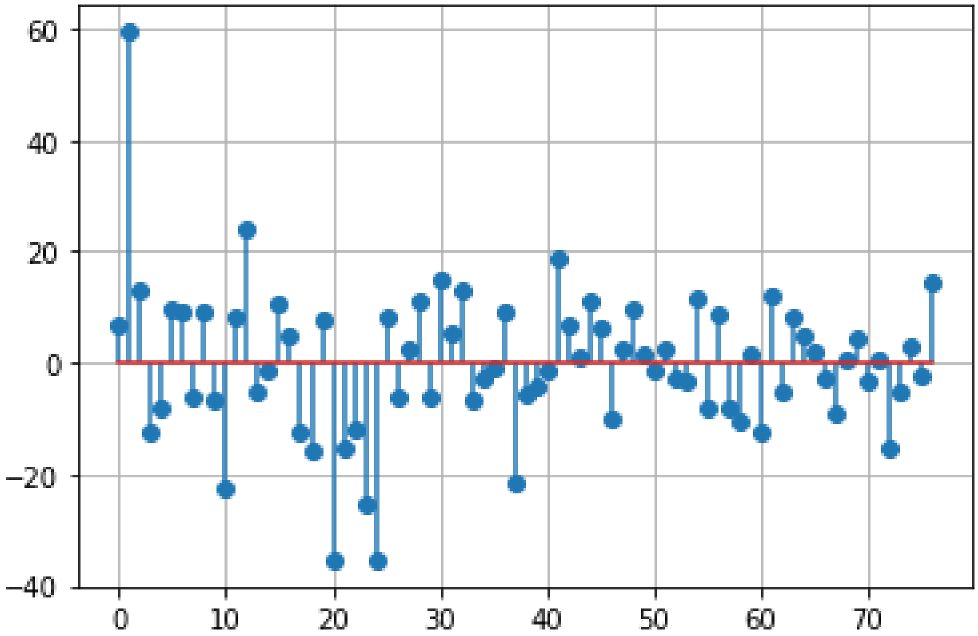
```
Accuracy = 0.935185
```

1.4 Interpreting the weight vector

Create a stem plot of the coefficients, `W` in the logistic regression model. Jse the `plt.stem()` function with the `use_line_collection=True` option. You can get the coefficients from `logreg.coef_`, but you will need to reshape this to a 1D array.

[36]:

```
# TODO
W = logreg.coef_
W = W.flatten()
plt.stem(W,use_line_collection=True)
plt.grid()
```



You should see that `W[i]` is very large for a few components `i`. These are the genes that are likely to be most involved in Down's Syndrome. Below we will use L1 regression to enforce sparsity. Find the names of the genes for two components `i` where the magnitude of `W[i]` is largest.

[37]:

```
# TODO
print(W)
maximum_1,maximum_2 = np.absolute(W).argsort()[-2:]
```

```
print('Largest magnitude genes:', df.columns[maximum_1], ' and ', df.
 ↪columns[maximum_2])
```

```
[  6.58331491  59.43169674  12.9336585  -12.24907466  -8.27102776
   9.66284805   9.40036586  -6.08553036   9.13752802  -6.49642516
 -22.31278863   8.04415401  24.07822996  -5.40732567  -1.47356327
  10.75343951   4.99553913 -12.50968512 -15.54113493   7.66273457
 -35.1529924  -15.29099753 -11.69453872 -25.17089004 -35.34758272
   8.41639037  -5.97717393   2.62705583  11.31512493  -6.0134401
  14.75726437   5.13453293  13.05805776  -6.69170161  -2.57508094
  -0.98418219   9.28917195 -21.4611864   -5.64129      -4.02386875
  -1.13654908  18.68242906   6.56915949   1.07985927  11.11310665
   6.21288526 -10.10431533   2.47403455   9.7098909    1.46378468
  -1.29932527   2.36819916  -2.87026214  -3.11771326  11.82174489
  -8.29140659   8.95861337  -8.26146922 -10.54556043   1.54099812
 -12.19053199  12.14765026  -5.22016774   8.41591743   4.75552589
   2.22643574  -3.00409416  -8.81570721   0.52535126   4.37004234
  -3.11771326   0.38342617 -15.06687631  -5.34693871   2.76145409
  -2.21324686  14.48052741]
Largest magnitude genes: ERK_N  and  ITSN1_N
```

1.5 Cross Validation

To obtain a slightly more accurate result, now perform 10-fold cross validation and measure the average precision, recall and f1-score. Note, that in performing the cross-validation, you will want to randomly permute the test and training sets using the `shuffle` option. In this data set, all the samples from each class are bunched together, so shuffling is essential. Print the mean precision, recall and f1-score and error rate across all the folds.

[44]:

```
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)


acc = np.zeros(nfold)
prec=np.zeros(nfold)
rec=np.zeros(nfold)
f1=np.zeros(nfold)
for i, I in enumerate(kf.split(X)) :

    # Get training and test data
    train, test = I
    Xtr = X[train,:]
    ytr = y[train]
    Xts = X[test,:]
    yts = y[test]
    # Scale the data
    scal = StandardScaler()
    Xtr1 = scal.fit_transform(Xtr)
    Xts1 = scal.transform(Xts)

    # Fit a model
    logreg.fit(Xtr1, ytr)
    # Predict on test samples and measure accuracy
    yhat = logreg.predict(Xts1)
    acc[i] = np.mean(yhat == yts)
    # Measure other performance metrics
    prec[i],rec[i],f1[i],_ =␣
 ↪precision_recall_fscore_support(yts,yhat,average='binary')

# Take average values of the metrics
precm = np.mean(prec)
recm = np.mean(rec)
f1m = np.mean(f1)
accm= np.mean(acc)

# Compute the standard errors
prec_se = np.std(prec)/np.sqrt(nfold-1)
rec_se = np.std(rec)/np.sqrt(nfold-1)
f1_se = np.std(f1)/np.sqrt(nfold-1)
acc_se = np.std(acc)/np.sqrt(nfold-1)
print('Precision = {0:.4f}, SE={1:.4f}'.format(precm,prec_se))
print('Recall = {0:.4f}, SE={1:.4f}'.format(recm, rec_se))
print('f1 = {0:.4f}, SE={1:.4f}'.format(f1m, f1_se))
print('Accuracy = {0:.4f}, SE={1:.4f}'.format(accm, acc_se))
```

```
Precision = 0.9591, SE=0.0076
Recall = 0.9595, SE=0.0064
f1 = 0.9590, SE=0.0041
Accuracy = 0.9611, SE=0.0043
```

1.6 Multi-Class Classification

Now use the response variable in `df1['class']`. This has 8 possible classes. Use the `np.unique` funtion as before to convert this to a vector `y` with values 0 to 7.

[45]:

```
# TODO
y = np.unique(df1['class'].values,return_inverse=True)[1]
```

Fit a multi-class logistic model by creating a `LogisticRegression` object, `logreg` and then calling the `logreg.fit` method.

```
[46]: Xtr, Xts, ytr, yts = train_test_split(X, y, test_size=0.3)
      scal = StandardScaler()
      Xtr1 = scal.fit_transform(Xtr)
      Xts1 = scal.transform(Xts)
      logreg1 = linear_model.
       ↪LogisticRegression(C=1e5,solver='liblinear',multi_class='ovr')
      logreg1.fit(Xtr1,ytr)
```

```
[46]: LogisticRegression(C=100000.0, class_weight=None, dual=False,
                         fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                         max_iter=100, multi_class='ovr', n_jobs=None, penalty='l2',
                         random_state=None, solver='liblinear', tol=0.0001, verbose=0,
                         warm_start=False)
```

Now perform 10-fold cross validation, and measure the confusion matrix `C` on the test data in each fold. You can use the `confustion_matrix` method in the `sklearn` package. Add the confusion matrix counts across all folds and then normalize the rows of the confusion matrix so that they sum to one. Thus, each element `C[i,j]` will represent the fraction of samples where `yhat==j` given `ytrue==i`. Print the confusion matrix. You can use the command

```
print(np.array_str(C, precision=4, suppress_small=True))
```

to create a nicely formatted print. Also print the overall mean and SE of the test accuracy across the folds.

```
[47]: from sklearn.metrics import confusion_matrix
      from sklearn.model_selection import KFold

      # TODO
      nfold = 10
      kf = KFold(n_splits=nfold,shuffle=True)

      acc = np.zeros(nfold)

      for i, I in enumerate(kf.split(X)) :
              # Get training and test data
              train, test = I
              Xtr = X[train,:]
              ytr = y[train]
              Xts = X[test,:]
              yts = y[test]
              # Scale the data
              scal = StandardScaler()
              Xtr1 = scal.fit_transform(Xtr)
              Xts1 = scal.transform(Xts)
              # Fit a model
              logreg1.fit(Xtr1, ytr)
              # Predict on test samples and measure accuracy
```

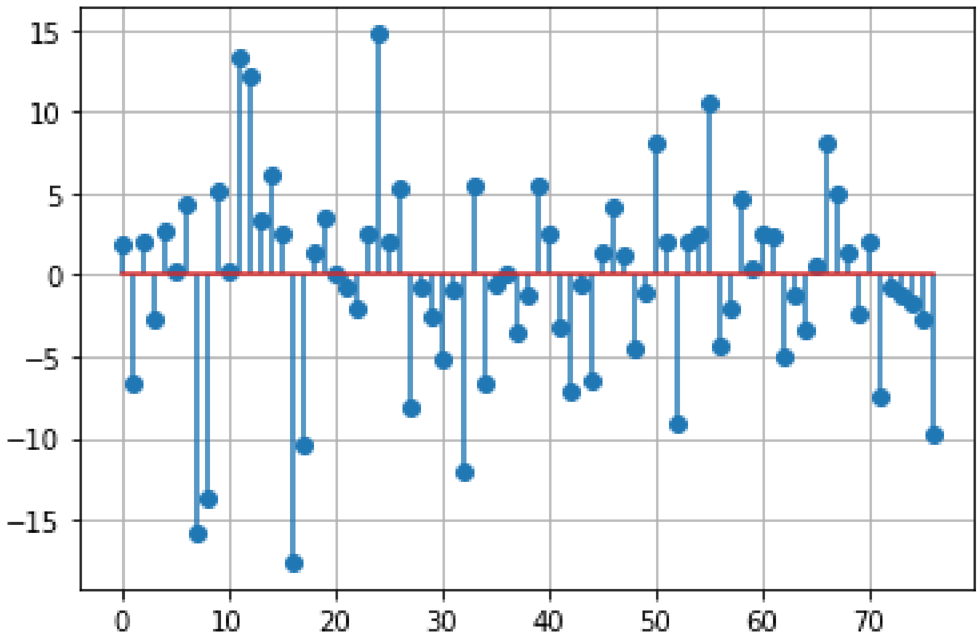
```
        yhat = logreg1.predict(Xts1)
        acc[i] = np.mean(yhat == yts)
        Conf = confusion_matrix(yts, yhat)

# Take average values of the metrics
accm= np.mean(acc)

# Compute the standard errors
acc_se = np.std(acc)/np.sqrt(nfold-1)
print(np.array_str(Conf, precision=4, suppress_small=True))
print('Accuracy = {0:.4f}, SE={1:.4f}'.format(accm, acc_se))
```

```
[[12  0  0  0  0  0  0  0]
 [ 1 15  0  0  0  0  0  0]
 [ 0  0 17  0  0  0  0  0]
 [ 0  0  0 14  0  0  0  0]
 [ 0  0  0  0 15  0  0  0]
 [ 0  0  0  0  0  8  0  0]
 [ 0  0  0  0  1  0 11  0]
 [ 0  0  0  0  0  0  0 14]]
Accuracy = 0.9889, SE=0.0023
```

Re-run the logistic regression on the entire training data and get the weight coefficients. This should be a 8 x 77 matrix. Create a stem plot of the first row of this matrix to see the coefficients on each of the genes.

[49]:

```
# TODO
scal = StandardScaler()
Xs = scal.fit_transform(X)
logreg2 = linear_model.
 ↪LogisticRegression(C=1e5,solver='liblinear',multi_class='ovr')
logreg2.fit(Xs,y)
W = logreg2.coef_
plt.stem(W[0,:],use_line_collection=True)
plt.grid()
```



1.7 L1-Regularization

This section is bonus.

In most genetic problems, only a limited number of the tested genes are likely influence any particular attribute. Hence, we would expect that the weight coefficients in the logistic regression model should be sparse. That is, they should be zero on any gene that plays no role in the particular attribute of interest. Genetic analysis commonly imposes sparsity by adding an l1-penalty term. Read the `sklearn` documentation on the `LogisticRegression` class to see how to set the l1-penalty and the inverse regularization strength, `C`.

Using the model selection strategies from the housing demo, use K-fold cross validation to select an appropriate inverse regularization strength.
* Use 10-fold cross validation * You should select around 20 values of `C`. It is up to you find a good range. * Make appropriate plots and print out to display your results * How does the accuracy compare to the accuracy achieved without regularization.

[50]:

```
# TODO
nfold=10
kf = KFold(n_splits=10,shuffle=True)
C = np.logspace(-1,2,20)
err = np.zeros((20,nfold))
for i,c in enumerate(C):
    j=0
    for train,test in kf.split(Xs):
        Xtr = Xs[train,:]
        ytr = y[train]
        Xts = Xs[test,:]
        yts = y[test]
        logreg3 = linear_model.
 ↪LogisticRegression(penalty='l1',C=c,solver='liblinear',multi_class='ovr')
        logreg3.fit(Xtr,ytr)
        yhat = logreg3.predict(Xts)

        err[i,j] = np.mean(yhat != yts)
        j=j+1
mean_err = np.mean(err,axis=1)
std_err = np.std(err,axis=1)
```

[52]:

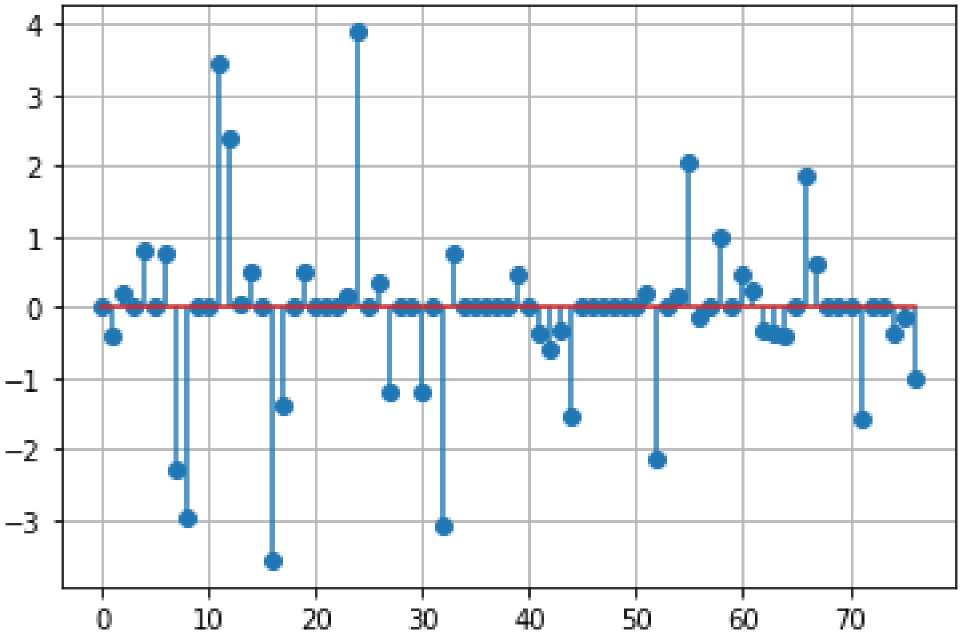
```
min = np.argmin(mean_err)
for i in range(min):
    if mean_err[i]<=mean_err[min]+std_err[min]:
        Cfin = C[i]
        break
print('Optimal C = {0:.4f}'.format(Cfin))
```

```
Optimal C = 1.2743
```

[54]:

```
# TODO
logreg4 = linear_model.
 ↪LogisticRegression(penalty='l1',C=Cfin,solver='liblinear',multi_class='ovr')
logreg4.fit(Xs,y)
yhat= logreg4.predict(Xs)
W1 = logreg4.coef_

plt.stem(W1[0,:],use_line_collection=True)
plt.grid()

accuracy = np.mean(y==yhat)
print('Accuracy = {0:.4f}'.format(accuracy))
```

```
Accuracy = 1.0000
```


4
3
2
1
0
−1
−2
−3
0
10
20
30
40
50
60
70