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 (./breast_cancer.ipynb), 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)

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 (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.

Load the Data

We begin by loading the standard modules.

In [1]:

```
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 (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.

In [2]:

```
# TODO
# df = ...


df = pd.read_excel('https://archive.ics.uci.edu/ml/machine-learning-databases/00342/Dat
a_Cortex_Nuclear.xls',
                   sheetname="Hoja1" , index_col = 0
                   );


#df = df.dropna()
print(df.shape);
df.head(4)
```

```
(1080, 81)
```

Out[2]:

	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	p
MouseID								
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2

4 rows × 81 columns

This data has missing values. The site:

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

In [3]:

```
# TODO

df1 =  df.copy()
#print(df1.columns,df1.columns.shape)

cols = df1.columns[:77];
#print(cols.shape);
print(df1[:][cols].shape)

df1[cols].fillna(df1.mean())
#df1[:][cols].apply(lambda x: x.fillna(x.mean()),axis=0)

for i in cols:
    df1[:][i] = df1[:][i].fillna((df1[i].mean()))



print(df1[29:33]['H3MeK4_N'])
df1.head(3)
```

```
(1080, 77)
MouseID
311_15    0.208286
320_1     0.205440
320_2     0.205440
320_3     0.205440
Name: H3MeK4_N, dtype: float64
```

Out[3]:

	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	p
MouseID								
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2

3 rows × 81 columns

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.

In [22]:

```
# TODO

print(df1['Genotype'].values)


u, y0 = np.unique( df1['Genotype'].values,return_inverse=True)

print(u,y0)
```

```
['Control' 'Control' 'Control' ..., 'Ts65Dn' 'Ts65Dn' 'Ts65Dn']
['Control' 'Ts65Dn'] [0 0 0 ..., 1 1 1]
```

As predictors, get all but the last four columns of the dataframes. Standardize the data matrix and call the standardized matrix `Xs`. The predictors are the expression levels of the 77 genes.

In [26]:

```
# TODO

X = np.array(df1.iloc[:,0:77])  # 81-4=77
Xs = preprocessing.scale(X)
print(Xs.shape,Xs[:1])
```

```
(1080, 77) [[ 0.31271112  0.5179336   2.2536689   1.49736237  2.30436515 -
0.34501948
  -0.15860115 -0.89990166  0.60411506  0.69147472  0.41199561 -0.13730652
   1.62561988  0.47515331  1.66663759 -0.64615366  1.16830859  1.62908876
  -0.0187583   1.39255039  0.1748746   0.11963335 -0.05878581  2.08208893
   1.85645802  1.49253698  0.67622532  1.14575898  1.000709   -0.09153425
   0.80415841  2.0631779  -0.61819788  0.09301054 -0.89734285  0.5412031
  -0.08236006  1.27633505  0.2380205   0.59811718  0.43945271  0.95675213
  -0.6434762   0.0365529  -0.58110866  1.62689083 -0.45575874  0.06064509
  -0.54318618  0.39175716 -0.25031145 -0.23615435  0.50723226 -1.06627553
  -0.76263608 -0.18598559 -1.23661012 -0.42323847  0.17759001  0.60267851
  -1.1752248  -1.4578892   0.2216074  -0.86661939 -2.13707513 -1.11983482
  -0.18256643 -1.305403   -1.33322422  0.         -1.06627553 -0.98737084
  -0.2857443  -1.01161458 -1.41662394 -1.60789061  1.06590091]]
```

Create a `LogisticRegression` object `logreg` and `fit` the training data.

In [27]:

```
# TODO

logreg = linear_model.LogisticRegression()
logreg.fit(Xs, y0)
```

Out[27]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Measure the accuracy of the classifer. That is, use the `logreg.predict` function to predict labels `yhat` and measure the fraction of time that the predictions match the true labels. Below, we will properly measure the accuracy on cross-validation data.

In [35]:

```
# TODO
from sklearn.metrics import precision_recall_fscore_support

yhat = logreg.predict(Xs)
acc = np.mean(yhat == y0)
preci,reci,f1i, _= precision_recall_fscore_support(y0,yhat,average='binary')

print(preci,reci,f1i)
print("Accuracy on training data = %f" % acc)
```

```
0.990079365079 0.978431372549 0.984220907298
Accuracy on training data = 0.985185
```

Interpreting the weight vector

Create a stem plot of the coefficients, `W` in the logistic regression model. You can get the coefficients from `logreg.coef_`, but you will need to reshape this to a 1D array.

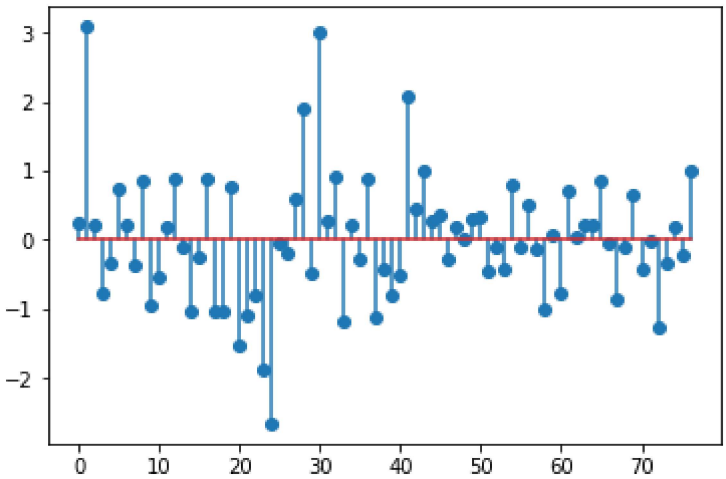
In [44]:

```
# TODO

W = logreg.coef_.reshape(-1)
plt.stem(W)
#print(W)
```

Out[44]:

```
<Container object of 3 artists>
```



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. Although, we do not discuss it in this class, there are ways to force the logistic regression to return a sparse vector `W`.

Find the names of the genes for two components `i` where the magnitude of `W[i]` is largest.

In [45]:

```
# TODO

W_max = np.argsort(-abs(W))[:2]
print(W_max)
Gene_max = np.array(df1.columns[W_max])
print("Two genes with largest magnitude of W are %s and %s." %
(Gene_max[0],Gene_max[1]))
```

```
[ 1 30]
Two genes with largest magnitude of W are ITSN1_N and APP_N.
```

Cross Validation

The above meaured the accuracy on the training data. It is more accurate to measure the accuracy on the test data. 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.

In [50]:

```
# TODO

from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support

nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
prec = []
rec = []
f1 = []
err = []

for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind
    Xtr = Xs[Itr]
    ytr = y0[Itr]
    Xts = Xs[Its]
    yts = y0[Its]
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)
    preci,reci,f1i,_= precision_recall_fscore_support(yts,yhat,average='binary')
    prec.append(preci)
    rec.append(reci)
    f1.append(f1i)
    erri = np.mean(yhat != yts)
    err.append(erri)

precm = np.mean(prec)
recm = np.mean(rec)
f1m = np.mean(f1)
errm= np.mean(err)
print('Precision =  {0:.4f}'.format(precm))
print('Recall =     {0:.4f}'.format(recm))
print('f1 =         {0:.4f}'.format(f1m))
print('Error rate =  {0:.4f}'.format(errm))
```

```
Precision =  0.9802
Recall =     0.9611
f1 =         0.9705
Error rate =  0.0259
```

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.

In [54]:

```
# TODO
# y = ...

uClass , y = np.unique(df1['class'].values,return_inverse=True)
print(uClass,y)
```

```
['c-CS-m' 'c-CS-s' 'c-SC-m' 'c-SC-s' 't-CS-m' 't-CS-s' 't-SC-m' 't-SC-s']
 [0 0 0 ..., 7 7 7]
```

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

In [55]:

```
# TODO

logreg = linear_model.LogisticRegression()
logreg.fit(Xs, y)
```

Out[55]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tru
e,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

Measure the accuracy on the training data.

In [58]:

```
# TODO

yhat = logreg.predict(Xs)
acc = np.mean(yhat == y)

print("Accuracy on training data = %f" % acc)
```

```
Accuracy on training data = 0.999074
```

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 error rate across the folds.

In [71]:

```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold

# TODO

nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
err = []
cm = np.zeros([8,8])
for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind
    Xtr = Xs[Itr]
    ytr = y[Itr]
    Xts = Xs[Its]
    yts = y[Its]
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)
    cm += confusion_matrix(yts,yhat)
    erri = np.mean(yhat != yts)
    err.append(erri)
C = cm.astype('float')/cm.sum(axis=1)
print(np.array_str(C, precision=4, suppress_small=True))

errm = np.mean(err)
err_se = np.std(err)/np.sqrt(nfold-1)
print('Error rate={0:.4f}, SE={1:.4f}'.format(errm,err_se))
```

```
[[ 0.9867  0.0074  0.      0.      0.0074  0.      0.      0.    ]
 [ 0.      0.9926  0.      0.      0.0074  0.      0.      0.    ]
 [ 0.      0.      1.      0.      0.      0.      0.      0.    ]
 [ 0.      0.      0.      0.9926  0.0074  0.      0.      0.    ]
 [ 0.0067  0.0074  0.      0.      0.9778  0.      0.      0.0074]
 [ 0.      0.      0.      0.      0.      1.      0.      0.    ]
 [ 0.      0.      0.      0.      0.0074  0.      0.9926  0.    ]
 [ 0.      0.      0.      0.      0.      0.      0.      1.    ]]
Error rate=0.0074, SE=0.0027
```

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.

In [61]:

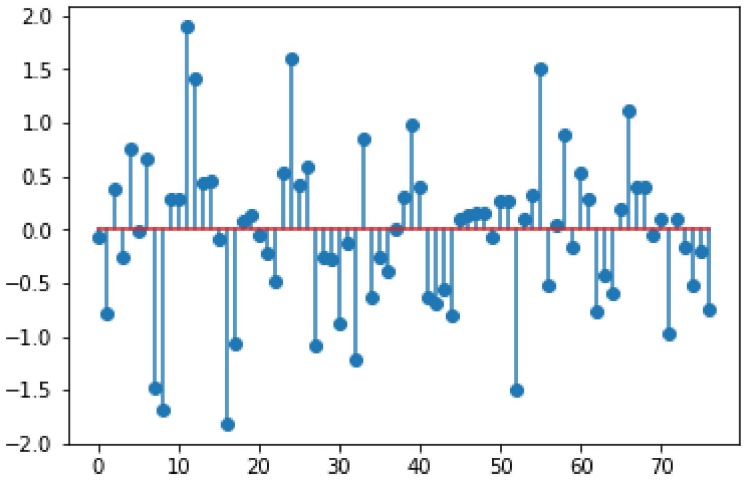
```
# TODO


logreg.fit(Xs, y)
W = logreg.coef_
print(W.shape)
plt.stem(W[0])
```

```
(8, 77)
```

Out[61]:

```
<Container object of 3 artists>
```



L1-Regularization

Graduate students only complete this section.

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 (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) 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 prostate cancer analysis demo (../model_sel/prostate.ipynb), 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.

In [62]:

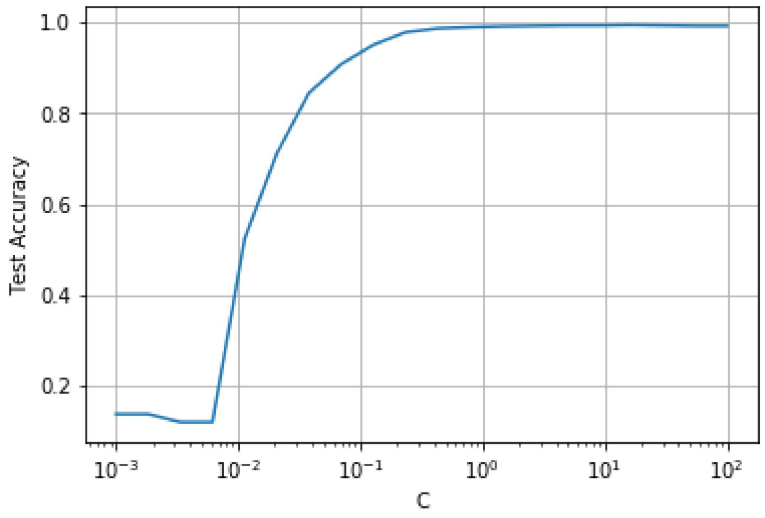
```
# TODO

nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
acc = np.zeros((20,nfold))
err = np.zeros((20,nfold))
d = np.logspace(-3,2,20)
for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind
    Xtr = Xs[Itr]
    ytr = y[Itr]
    Xts = Xs[Its]
    yts = y[Its]
    for i in range(20):
        logreg = linear_model.LogisticRegression(penalty='l1', C=d[i])
        logreg.fit(Xtr, ytr)
        yhat = logreg.predict(Xts)
        acc[i,isplit] = np.mean(yhat == yts)
        err[i,isplit] = np.mean(yhat != yts)
accm = np.mean(acc,axis=1)
errm = np.mean(err,axis=1)
err_se = np.std(err,axis=1)/np.sqrt(nfold-1)
```

In [70]:

```
print(accm)
plt.semilogx(d, accm)
plt.xlabel('C')
plt.ylabel('Test Accuracy')
plt.grid()
plt.show()
```

```
[ 0.13888889  0.13888889  0.1212963   0.1212963   0.525       0.71203704
  0.84537037  0.90833333  0.95092593  0.9787037   0.98703704  0.98981481
  0.99166667  0.99259259  0.99351852  0.99351852  0.99444444  0.99351852
  0.99259259  0.99259259]
```



Compared to the accuracy achieved without regularization, the accuracy with L1 regularization gets lower when regularization strength increases.

For the optimal `C`, fit the model on the entire training data with l1 regularization. Find the resulting weight matrix, `W_l1`. Plot the first row of this weight matrix and compare it to the first row of the weight matrix without the regularization. You should see that, with l1-regularization, the weight matrix is much more sparse and hence the roles of particular genes are more clearly visible.

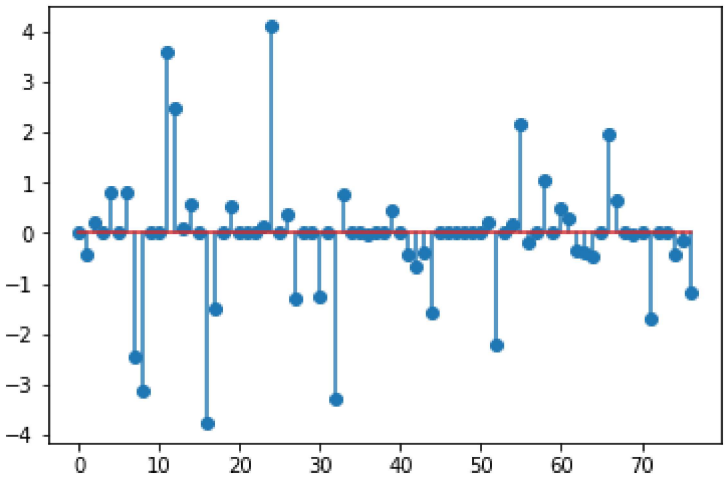
In [67]:

```
# TODO

# Use the one standard error rule

imin = np.argmin(errm)
err_tgt = errm[imin] + err_se[imin]
I = np.where(errm < err_tgt)[0]
iopt = I[0]
c_opt = d[iopt]
print('The optimal C is {0:.4f}.'.format(c_opt))
logreg = linear_model.LogisticRegression(penalty='l1', C=c_opt)
logreg.fit(Xs, y)
W_l1 = logreg.coef_
plt.stem(W_l1[0])
print('The weight matrix is much more sparse.')
```

```
The optimal C is 1.4384.
The weight matrix is much more sparse.
```



The weight matrix is much more sparse and hence the roles of particular genes are more clearly visible.