lab_gene_partial

October 20, 2021

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.

[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

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.

[2]:
```
# TODO 1
df = pd.read_excel('https://archive.ics.uci.edu/ml/machine-learning-databases/
 ↪00342/Data_Cortex_Nuclear.xls',
                   index_col = 0)
df.head()
```

[2]:

MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAMKII_N	pCREB_N	pELK_N	…	pCFOS_N	SYP_N	H3AcK18_N	EGR1_N	H3MeK4_N	CaNA_N	Genotype	Treatment	Behavior	class
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373744	0.232224	1.750936	…	0.108336	0.427099	0.114783	0.131790	0.128186	1.675652	Control	Memantine	C/S	c-CS-m
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292150	0.226972	1.596377	…	0.104315	0.441581	0.111974	0.135103	0.131119	1.743610	Control	Memantine	C/S	c-CS-m
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283337	0.230247	1.561316	…	0.106219	0.435777	0.111883	0.133362	0.127431	1.926427	Control	Memantine	C/S	c-CS-m
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152301	0.207004	1.595086	…	0.111262	0.391691	0.130405	0.147444	0.146901	1.700563	Control	Memantine	C/S	c-CS-m
309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134014	0.192158	1.504230	…	0.110694	0.434154	0.118481	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.

[3]:
```
# TODO 2
df1 = df.fillna(df.mean())
df1.head()
```

[3]:

MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N

```
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]
```

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.

[4]:
```
# TODO 3
y = np.unique(df1['Genotype'].values,return_inverse=True)[1]
y
```

[4]:
```
array([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`.

[5]:
```
# TODO 4
xnames = df1.columns[0:-4]
X = df1.loc[:,xnames]
X.head()
```

[5]:
```
         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  …      SHH_N     BAD_N    BCL2_N  \
MouseID                                 …
309_1     2.373744  0.232224  1.750936  …  0.188852  0.122652  0.134762
309_2     2.292150  0.226972  1.596377  …  0.200404  0.116682  0.134762
309_3     2.283337  0.230247  1.561316  …  0.193685  0.118508  0.134762
309_4     2.152301  0.207004  1.595086  …  0.192112  0.132781  0.134762
309_5     2.134014  0.192158  1.504230  …  0.205604  0.129954  0.134762

            pS6_N   pCFOS_N     SYP_N  H3AcK18_N    EGR1_N  H3MeK4_N    CaNA_N
MouseID
309_1    0.106305  0.108336  0.427099   0.114783  0.131790  0.128186  1.675652
309_2    0.106592  0.104315  0.441581   0.111974  0.135103  0.131119  1.743610
309_3    0.108303  0.106219  0.435777   0.111883  0.133362  0.127431  1.926427
309_4    0.103184  0.111262  0.391691   0.130405  0.147444  0.146901  1.700563
309_5    0.104784  0.110694  0.434154   0.118481  0.140314  0.148380  1.839730

[5 rows x 77 columns]
```

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

[6]:
```
from sklearn.model_selection import train_test_split

# Use : shuffle=True, random_state=3 so we all can have same split.
# TODO 5:
Xtr, Xts, ytr, yts = train_test_split(X,y,test_size = 0.30, shuffle=True,␣
 ↪random_state = 3)
```

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

[7]:
```
from sklearn.preprocessing import StandardScaler

# TODO 6
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`.

[8]:
```
# TODO 7
logreg = linear_model.LogisticRegression(C=1e5, solver='liblinear')
logreg.fit(Xtr1,ytr)
```

[8]:
```
LogisticRegression(C=100000.0, solver='liblinear')
```

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

[9]:
```
# TODO 8
yhat = logreg.predict(Xts1)
acc = np.mean(yhat==yts)
acc
```

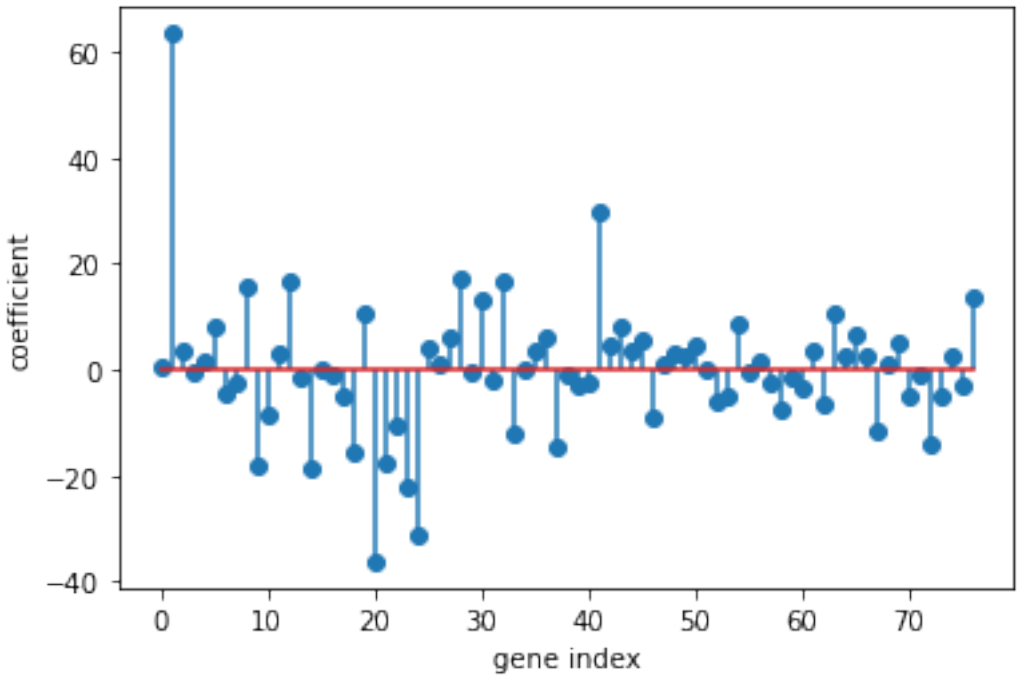
[9]:
```
0.9475308641975309
```

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.

[10]:
```
# TODO 9
W = np.squeeze(logreg.coef_)
plt.stem(W,use_line_collection=True)
plt.xlabel('gene index')
plt.ylabel('coefficient')
```

[10]:
```
Text(0, 0.5, 'coefficient')
```



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.

[11]:
```
# TODO 10
np.argsort(-np.abs(W))
```

[11]:
```
array([ 1, 20, 24, 41, 23, 14,  9, 28, 21, 12, 32,  8, 18, 37, 72, 76, 30,
       33, 67, 19, 22, 63, 46, 54, 10, 43,  5, 58, 65, 62, 52, 36, 27, 45,
       69, 73, 17, 70, 53,  6, 42, 50, 25,  2, 35, 61, 60, 44, 11, 39, 48,
       49, 75, 66, 40,  7, 64, 74, 57, 31, 56,  4, 59, 13, 68, 38, 26, 16,
       47, 71,  0, 55, 29,  3, 51, 34, 15])
```

[12]:
```
largest = xnames[np.argsort(-np.abs(W))[0]]
second = xnames[np.argsort(-np.abs(W))[1]]
print('The name of genes where W[i] is largest and second largest:
 ↪'+'\n'+str(largest)+', '+str(second))
```

```
The name of genes where W[i] is largest and second largest:
ITSN1_N, BRAF_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.

[13]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)

# TODO 11
precision = np.zeros(nfold)
recall = np.zeros(nfold)
f1 = np.zeros(nfold)
error_rate = np.zeros(nfold)


for i,I in enumerate(kf.split(X)):
    train,test = I
    Xtr = X.iloc[train,:]
    ytr = y[train]
    Xts = X.iloc[test,:]
    yts = y[test]

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

```
    logreg.fit(Xtr1,ytr)
    yhat = logreg.predict(Xts1)
    error_rate[i] = 1-np.mean(yhat==yts)
    precision[i],recall[i],f1[i],_ =␣
↪precision_recall_fscore_support(yts,yhat,average='binary')


mean_precision = np.mean(precision)
mean_recall = np.mean(recall)
mean_f1 = np.mean(f1)
mean_error_rate = np.mean(error_rate)

print('The mean precision = '+str(mean_precision))
print('The mean recall = '+str(mean_recall))
print('The mean f1 = '+str(mean_f1))
print('The mean error rate = '+str(mean_error_rate))
```

```
The mean precision = 0.9573622452088838
The mean recall = 0.9507563002897086
The mean f1 = 0.9536597685772715
The mean error rate = 0.043518518518518526
```

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.

[14]:
```
# TODO 12
y = np.unique(df1['class'].values, return_inverse=True)[1]
y
```

[14]:
```
array([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.

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.

[15]:
```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold
```

```
# TODO 13
nfold = 10
kf = KFold(n_splits = nfold, shuffle = True)
num_class = len(np.unique(df1['class']))
C = np.zeros((num_class,num_class))
acc = np.zeros(nfold)
logreg = linear_model.LogisticRegression(C=1e5, solver='liblinear')

for i,I in enumerate(kf.split(X)):
    train,test = I
    Xtr = X.iloc[train,:]
    ytr = y[train]
    Xts = X.iloc[test,:]
    yts = y[test]

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

    logreg.fit(Xtr1,ytr)
    yhat = logreg.predict(Xts1)
    acc[i] = np.mean(yhat==yts)
    current_c = confusion_matrix(yts,yhat)
    C += current_c

C = C/(np.sum(C,axis=1)[:,None])

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

acc_mean = np.mean(acc)
acc_se = np.std(acc)*np.sqrt(nfold)/np.sqrt(nfold-1)
print('\nMean Accuracy = '+str(acc_mean))
print('SE Accuracy = '+str(acc_se))
```

```
[[0.9933 0.0067 0.     0.     0.     0.     0.     0.    ]
 [0.0148 0.9778 0.     0.     0.0074 0.     0.     0.    ]
 [0.     0.0067 0.9867 0.     0.0067 0.     0.     0.    ]
 [0.0074 0.     0.     0.9926 0.     0.     0.     0.    ]
 [0.     0.0148 0.     0.     0.9852 0.     0.     0.    ]
 [0.0095 0.     0.     0.     0.     0.9905 0.     0.    ]
 [0.     0.     0.     0.0074 0.     0.     0.9926 0.    ]
 [0.     0.     0.     0.     0.     0.     0.     1.    ]]

Mean Accuracy = 0.9898148148148149
SE Accuracy = 0.008107361442323257
```

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.

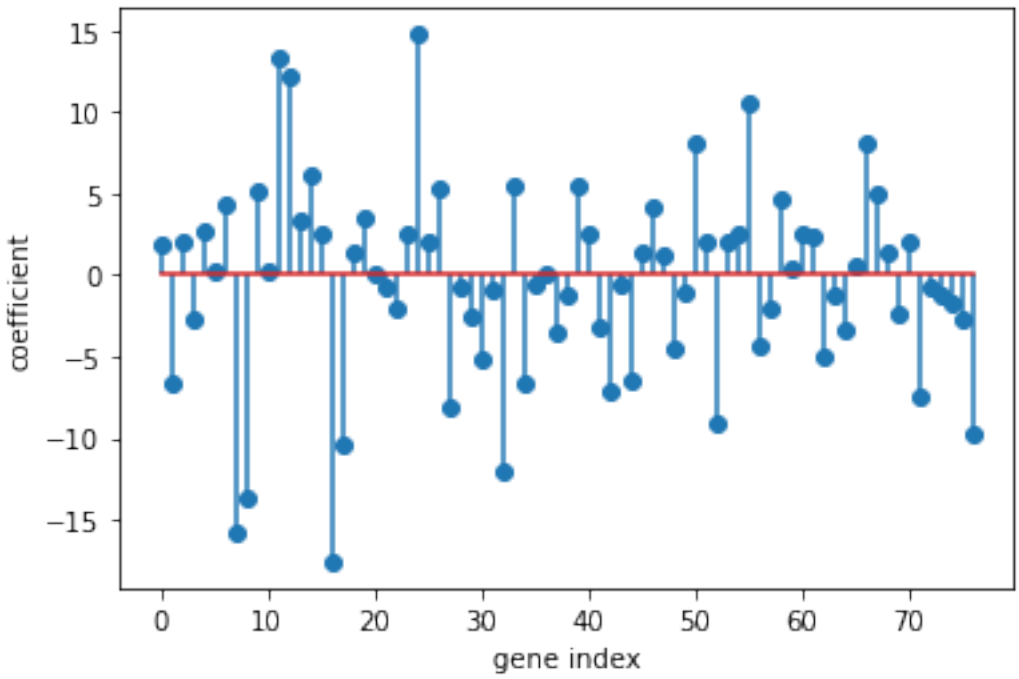
[16]:
```
# TODO 14
# I used X1 & y to fit the logreg since I used kf.split(X)
scal = StandardScaler()
X1 = scal.fit_transform(X)
logreg.fit(X1,y)
W = logreg.coef_
```

[17]:
```
np.shape(W)
```

[17]:
```
(8, 77)
```

[18]:
```
w = W[0,:]
plt.stem(w,use_line_collection=True)
plt.xlabel('gene index')
plt.ylabel('coefficient')
```

[18]:
```
Text(0, 0.5, 'coefficient')
```



[19]:
```
coeff_sort_ls_without = np.flipud(np.sort(np.abs(w)))
```

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`.

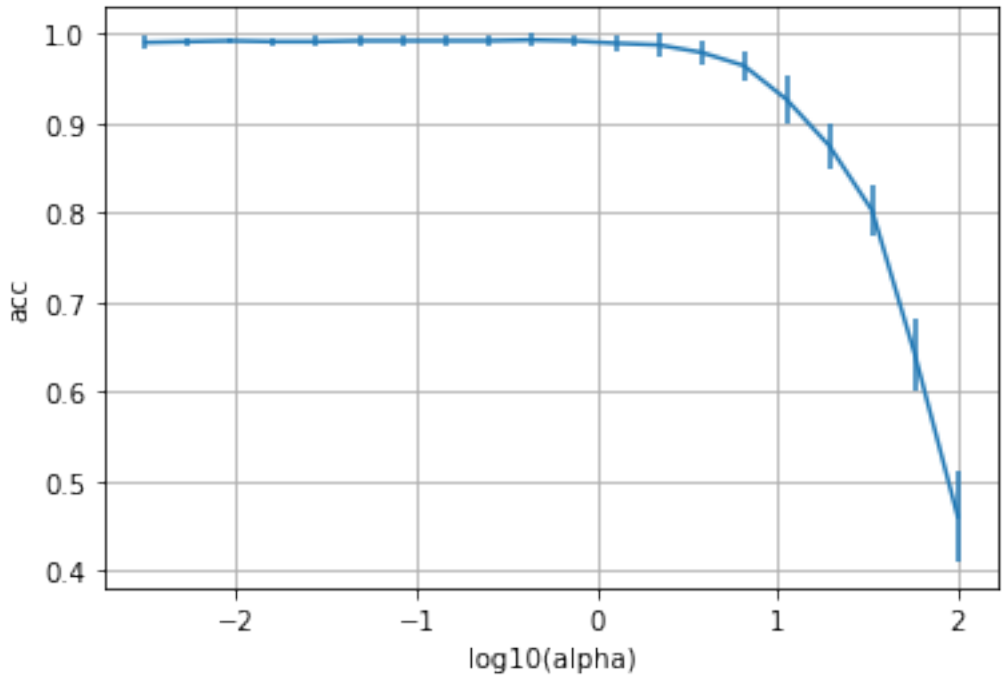
Using the model selection strategies from the housing demo, use K-fold cross validation to select an appropriate inverse regularization strength.
* Use 10-fold cross validation * You should select around 20 values of `C`. It is up to you find a good range. * Make appropriate plots and print out to display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

[20]:
```
# TODO 15
nfold = 10
kf = KFold(n_splits = nfold, shuffle = True)
alpha = np.logspace(-2.5,2,20)
nalpha = len(alpha)
acc = np.zeros((nalpha,nfold))

for i, I in enumerate(kf.split(X)):
    train,test = I
    Xtr = X.iloc[train,:]
    ytr = y[train]
    Xts = X.iloc[test,:]
    yts = y[test]

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

    for i_alpha, al in enumerate(alpha):
        lasso = linear_model.LogisticRegression(C=(1/al), solver='liblinear',␣
 ↪penalty = 'l1')
        lasso.fit(Xtr1,ytr)
        yhat = lasso.predict(Xts1)
        acc[i_alpha,i] = np.mean(yts==yhat)
    print('fold# '+str(i)+' done')

acc_mean = np.mean(acc,axis = 1)
acc_se = np.std(acc,axis=1)*np.sqrt(nfold)/np.sqrt(nfold-1)
```

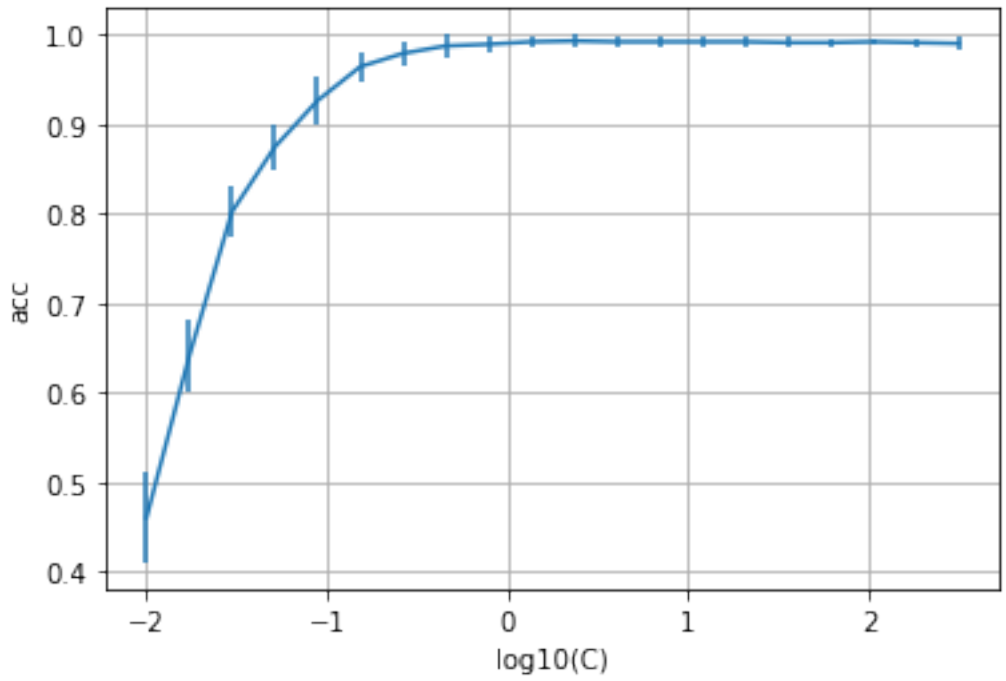
```
fold# 0 done
fold# 1 done
fold# 2 done
fold# 3 done
fold# 4 done
fold# 5 done
```

```
fold# 6 done
fold# 7 done
fold# 8 done
fold# 9 done
```

[22]:
```
plt.errorbar(np.log10(alpha), acc_mean, yerr=acc_se)
plt.xlabel('log10(alpha)')
plt.ylabel('acc')
plt.grid()
```



[23]:
```
plt.errorbar(np.log10(1/alpha), acc_mean, yerr=acc_se)
plt.xlabel('log10(C)')
plt.ylabel('acc')
plt.grid()
```



[24]:
```
opt_index = np.argmax(acc_mean)
opt_alpha = alpha[opt_index]
opt_acc = acc_mean[opt_index]
print('using normal rule:\n')
print('optimal index = '+str(opt_index))
print('optimal alpha = '+str(opt_alpha))
print('optimal C = '+str(1/opt_alpha))
print('optimal accuracy = '+str(opt_acc))
```

```
using normal rule:

optimal index = 9
optimal alpha = 0.42813323987193913
optimal C = 2.3357214690901236
optimal accuracy = 0.9925925925925926
```

[25]:
```
print('using 1-SE rule:\n')
acc_target = acc_mean[opt_index]-acc_se[opt_index]
I = np.where(acc_mean >= acc_target)[0]
iopt = I[0]
opt_alpha = alpha[iopt]
opt_acc = acc_mean[iopt]
print('optimal index = '+str(iopt))
print('optimal alpha = '+str(opt_alpha))
print('optimal C = '+str(1/opt_alpha))
```

```
print('optimal accuracy = '+str(opt_acc))
```

```
using 1-SE rule:

optimal index = 0
optimal alpha = 0.0031622776601683794
optimal C = 316.2277660168379
optimal accuracy = 0.9898148148148147
```

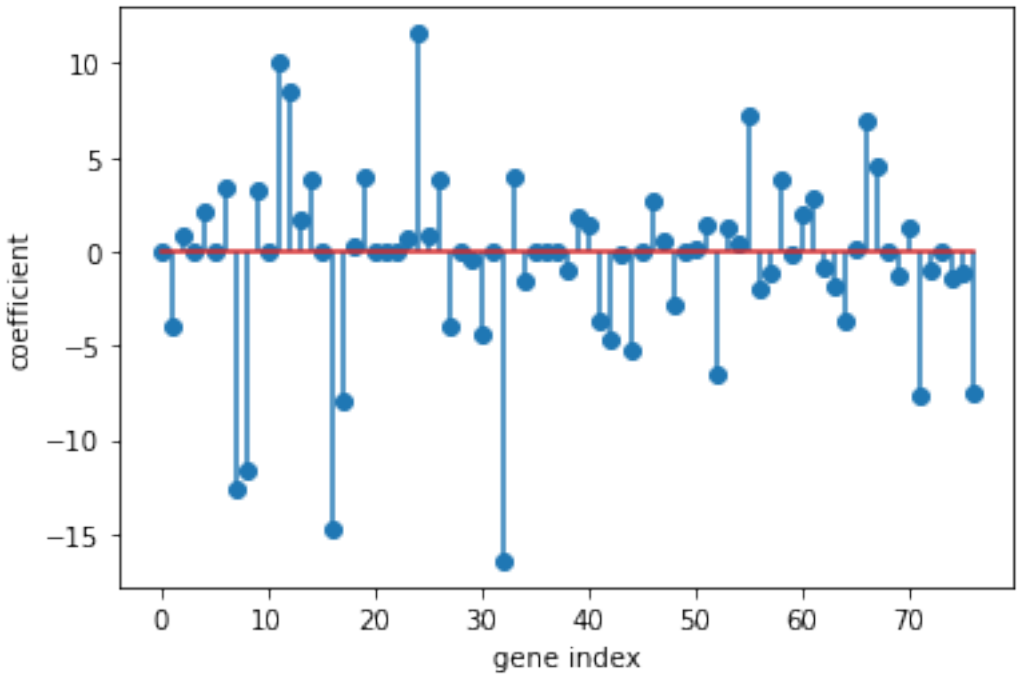
[26]:
```
optimal_lasso = linear_model.LogisticRegression(C=(1/opt_alpha),␣
 ↪solver='liblinear', penalty = 'l1')
# fit with the entire dataset
scal = StandardScaler()
X1 = scal.fit_transform(X)
optimal_lasso.fit(X1,y)
W = optimal_lasso.coef_
```

[27]:
```
np.shape(W)
```

[27]:
```
(8, 77)
```

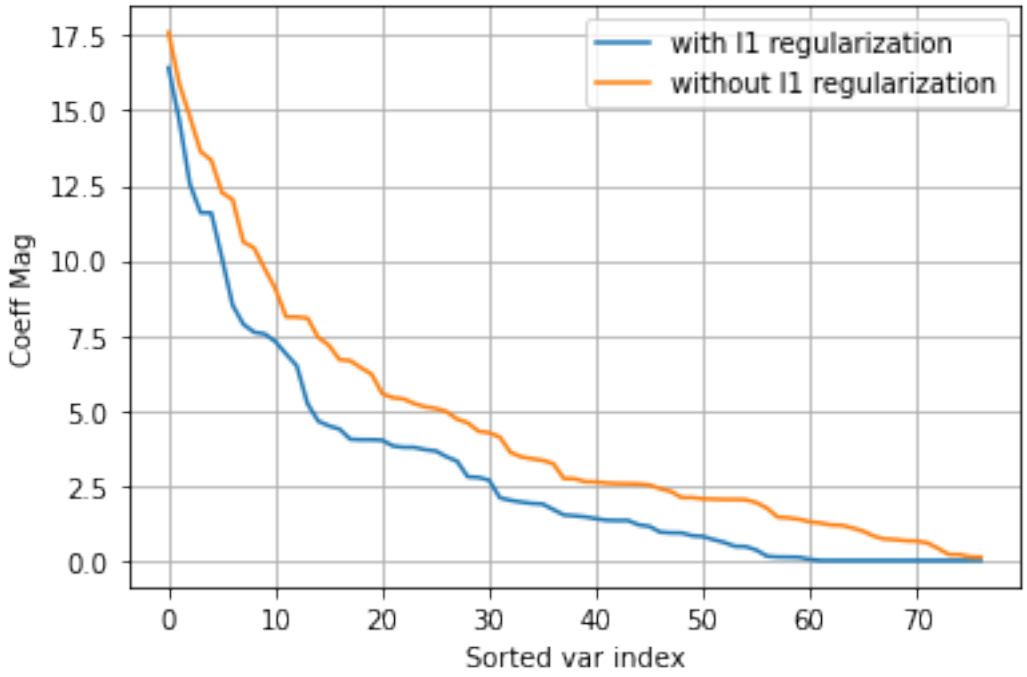
[28]:
```
w = W[0,:]
plt.stem(w,use_line_collection=True)
plt.xlabel('gene index')
plt.ylabel('coefficient')
```

[28]:
```
Text(0, 0.5, 'coefficient')
```



[29]:
```
coeff_sort_ls = np.flipud(np.sort(np.abs(w)))
plt.plot(coeff_sort_ls,label='with l1 regularization')
plt.plot(coeff_sort_ls_without,label = 'without l1 regularization')
plt.xlabel('Sorted var index')
plt.ylabel('Coeff Mag')
plt.grid()
plt.legend()
```

[29]:
```
<matplotlib.legend.Legend at 0x7ffc4855ea60>
```



The optimal accuracy with l1 penalty is greater than the accuracy without the l1 penalty by a small amount using the normal rule.

But the optimal accuracy with l1 penalty is almost the same with the accuracy without the l1 penalty using the 1-se rule.

From the stem plot, we can observe variable sparsity for l1 penalty. We can also deduce the same results from the abs coefficient magnitude vs. sorted index plot above.

[ ]: