lab04_gene_partial

March 4, 2018

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 binary classification, and evaluating performance using various metrics * Perform multi-class logistic classification, and evaluating performance using accuracy and confusion matrix * Use L1-regularization to promote sparse weights for improved estimation (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.

```
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
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 = pd.read_excel('https://archive.ics.uci.edu/ml/machine-learning-databases/00342/Da
        df.head()
```

Out[2]:

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	

MouseID	pCAMKII_N	pCREB_N	pELK_N	...	pCFOS_N	SYP_N	H3AcK18_N	\
				...				
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	

MouseID	EGR1_N	H3MeK4_N	CaNA_N	Genotype	Treatment	Behavior	class
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.

```
In [3]: # 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.

```
In [4]: # TODO
        y = np.unique(df1['Genotype'].values, return_inverse = True)[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 [5]: # TODO
        Xs = preprocessing.scale(np.array(df1[df1.columns[:-4]]))
```

Create a `LogisticRegression` object `logreg` and `fit` the training data. Use `C = 1e5`.

```
In [6]: # TODO
        logreg = linear_model.LogisticRegression(C=1e5)
        logreg.fit(Xs,y)
```

```
Out[6]: LogisticRegression(C=100000.0, class_weight=None, dual=False,
                  fit_intercept=True, 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)
```

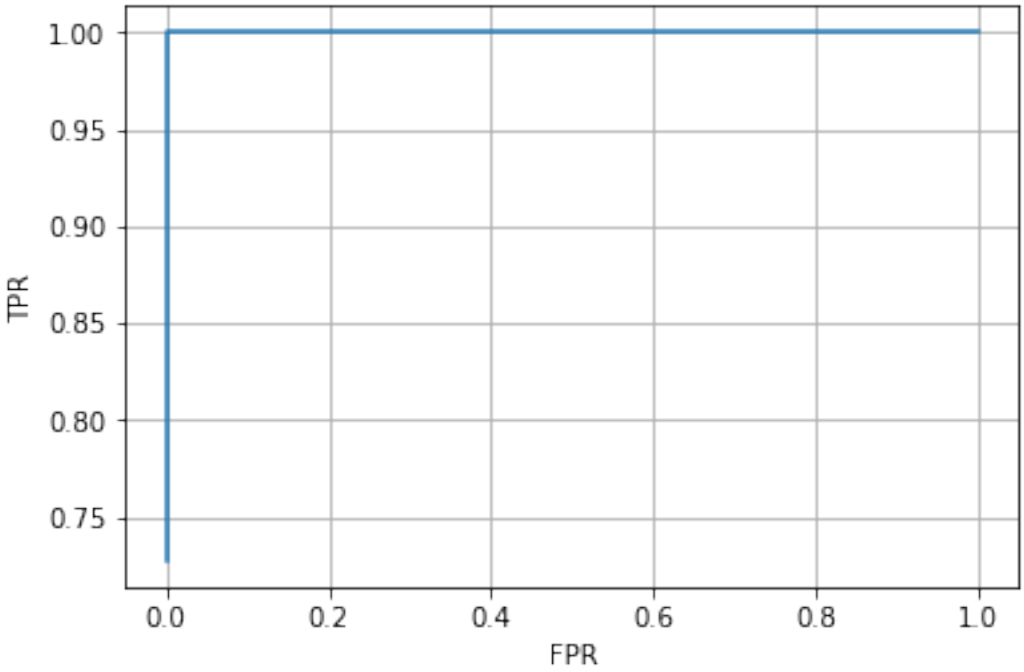
Measure the accuracy of the classifer. That is, use the `logreg.predict` function to predict labels `yhat` and measure the fraction of time that the predictions match the true labels. Also, plot the ROC curve, and measure the AUC. Later, we will properly measure the accuracy and AUC on cross-validation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

```
In [7]: # TODO
        yhat = logreg.predict(Xs)
        yprob = logreg.predict_proba(Xs)
        print("Accuracy of the classifier = %f" % np.mean(yhat == y))
        from sklearn import metrics
        fpr, tpr, thresholds = metrics.roc_curve(y, yprob[:,1])

        plt.plot(fpr, tpr)
        plt.grid()
        plt.xlabel('FPR')
        plt.ylabel('TPR')
        plt.show()

        auc=metrics.roc_auc_score(y, yprob[:,1])
        print("AUC = %f" % auc)
```

```
Accuracy of the classifier = 1.000000
```

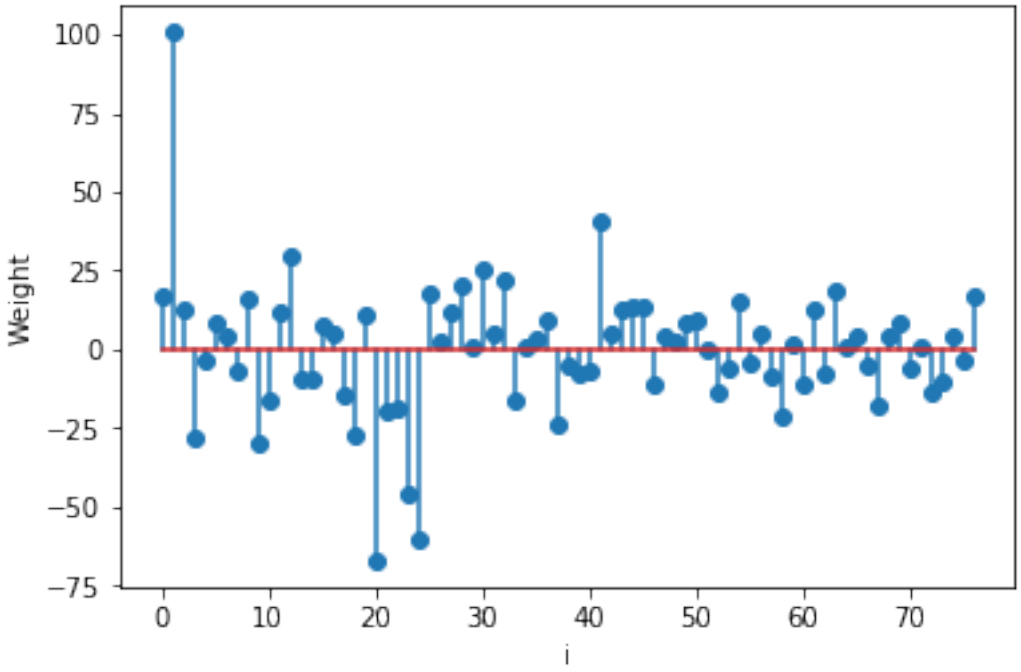



```
AUC = 1.000000
```

1.4 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 [8]: # TODO
        W = logreg.coef_
        plt.stem(logreg.coef_.reshape(-1,1))
        plt.xlabel('i')
        plt.ylabel('Weight')
        plt.show()
```



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.

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

```
In [9]: # TODO
        print('The two components are \'%s\' and \'%s\'.' % tuple(df1.columns[np.argsort(-logr
```

```
The two components are 'ITSN1_N' and 'TIAM1_N'.
```

1.5 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, as well as the AUC. 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 [10]: # TODO
         from sklearn.model_selection import KFold
         from sklearn.metrics import precision_recall_fscore_support

         nfold = 10
         kf = KFold(n_splits=nfold)
```

```
prec = []
rec = []
f1 = []
err = []

for train, test in kf.split(Xs):
    # Get training and test data
    Xtr = Xs[train,:]
    ytr = y[train]
    Xts = Xs[test,:]
    yts = y[test]

    # Fit a model
    logreg.fit(Xtr, ytr)
    yhat = logreg.predict(Xts)

    # Measure performance
    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)

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

# 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)
err_se = np.std(err)/np.sqrt(nfold-1)

print('Precision =  %0.4f, SE = %0.4f' % (precm,prec_se))
print('Recall =     %0.4f, SE = %0.4f' % (recm, rec_se))
print('f1 =         %0.4f, SE = %0.4f' % (f1m, f1_se))
print('Error rate = %0.4f, SE = %0.4f' % (errm, err_se))
```

```
/usr/local/lib/python3.6/site-packages/sklearn/metrics/classification.py:1137: UndefinedMetricW
  'recall', 'true', average, warn_for)


Precision =  0.4689, SE = 0.1590
Recall =     0.3378, SE = 0.1211
f1 =         0.3804, SE = 0.1308
```

```
Error rate = 0.3565, SE = 0.0576
```

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.

```
In [11]: # 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. In general, you could either use the 'one over rest (ovr)' option or the 'multinomial' option. In this exercise use the default 'ovr' and `C=1`. As an optional exercise, you could also compare the results obtained with these two options.

```
In [12]: # TODO
         logreg = linear_model.LogisticRegression(multi_class='ovr', C=1)
         logreg.fit(Xs, y)

Out[12]: LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                   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 [13]: # TODO
         yhat = logreg.predict(Xs)
         print("Accuracy on training data = %f" % np.mean(yhat == y))

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 accuracy across the folds.

```
In [14]: from sklearn.metrics import confusion_matrix
         from sklearn.model_selection import KFold

         # TODO
         C = np.zeros((8, 8))
```

```
        nfold = 10
        kf = KFold(n_splits=nfold, shuffle=True)

        for train, test in kf.split(Xs):
            Xtr = Xs[train,:]
            ytr = y[train]
            Xts = Xs[test,:]
            yts = y[test]

            logreg.fit(Xtr, ytr)
            yhat = logreg.predict(Xts)

            C += confusion_matrix(yts, yhat)

            RSS = np.mean((yhat-yts)**2)/np.std(yts)**2
            print("Test error rate for fold %02d = %.4f" % (11-nfold, RSS))
            nfold -= 1

        new_matrix = preprocessing.normalize(C, axis=1)

        print("\nConfusion Matrix: \n", np.array_str(new_matrix, precision=4, suppress_small=
```

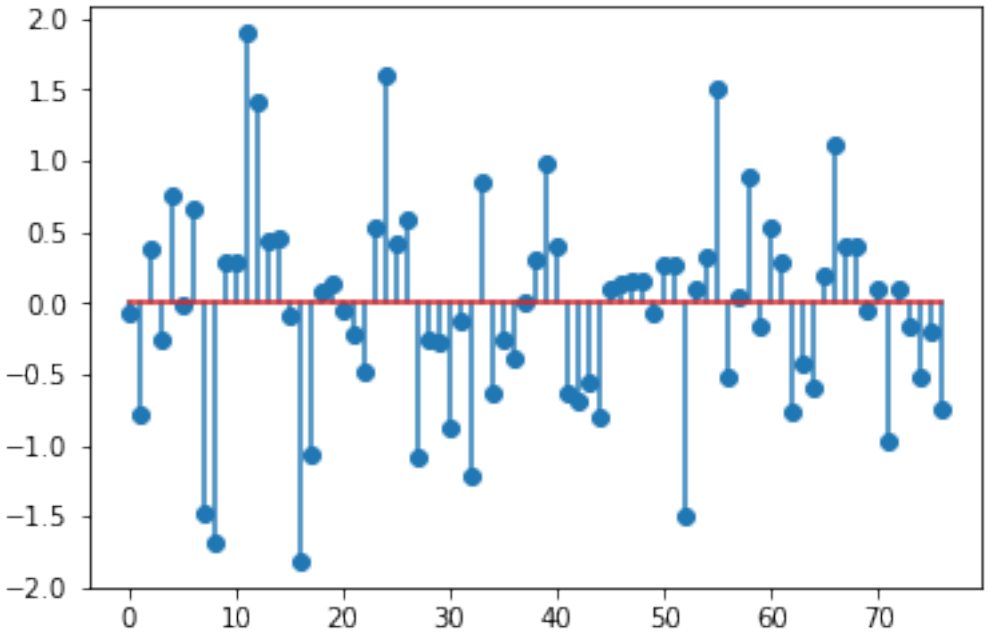
```
Test error rate for fold 01 = 0.0349
Test error rate for fold 02 = 0.0190
Test error rate for fold 03 = 0.0542
Test error rate for fold 04 = 0.0856
Test error rate for fold 05 = 0.0346
Test error rate for fold 06 = 0.0343
Test error rate for fold 07 = 0.0326
Test error rate for fold 08 = 0.0000
Test error rate for fold 09 = 0.0000
Test error rate for fold 10 = 0.0000

Confusion Matrix:
 [[0.9997 0.0069 0.0069 0.     0.0207 0.     0.     0.    ]
 [0.0076 0.9999 0.     0.     0.0076 0.0076 0.     0.    ]
 [0.     0.     1.     0.     0.     0.     0.     0.0067]
 [0.0075 0.     0.     1.     0.     0.     0.     0.    ]
 [0.0076 0.0229 0.     0.     0.9997 0.     0.     0.    ]
 [0.     0.     0.     0.     0.     1.     0.     0.    ]
 [0.     0.     0.0075 0.     0.     0.     1.     0.    ]
 [0.     0.     0.     0.     0.     0.     0.     1.    ]]
```

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 for the first class.

```
In [15]: # TODO
```

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



1.7 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 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, 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 to find a good range. * For each C and each fold, you should compute the classification error rate * For each C and each fold, you should also determine the nubmer of non-zero coefficients for the first class. For this purpse, you can assume coefficient with magnitude <0.01 as zero.

```
In [16]: # TODO
         nfold = 10
         kf = KFold(n_splits=nfold, shuffle=True)

         npen = 20
```

```
C_test = np.logspace(-2,2,npen)
err_rate = np.zeros((npen,nfold))
num_nonzerocoef = np.zeros((npen,nfold))

# Create the logistic regression object
logreg = linear_model.LogisticRegression(penalty='l1', warm_start=True)

for ifold, Ind in enumerate(kf.split(Xs)):
    # Get training and test data
    Itr, Its = Ind
    Xtr = Xs[Itr,:]
    ytr = y[Itr]
    Xts = Xs[Its,:]
    yts = y[Its]

    # Loop over penalty levels
    for ipen, c in enumerate(C_test):

        # Set the penalty level
        logreg.C = c

        # Fit a model on the training data
        logreg.fit(Xtr, ytr)

        # Predict the labels on the test set.
        yhat = logreg.predict(Xts)

        # Measure the accuracy
        err_rate[ipen, ifold] = np.mean(yhat != yts)
        num_nonzerocoef[ipen, ifold]=np.sum(abs(logreg.coef_) > 0.01)
    print("Fold %d" % (ifold+1))
```

```
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Fold 6
Fold 7
Fold 8
Fold 9
Fold 10
```

Now compute the mean and standard error on the error rate for each `C` and plot the results (Use `errorbar()` method). Also determine and print the minimum test error rate and corresponding C value.

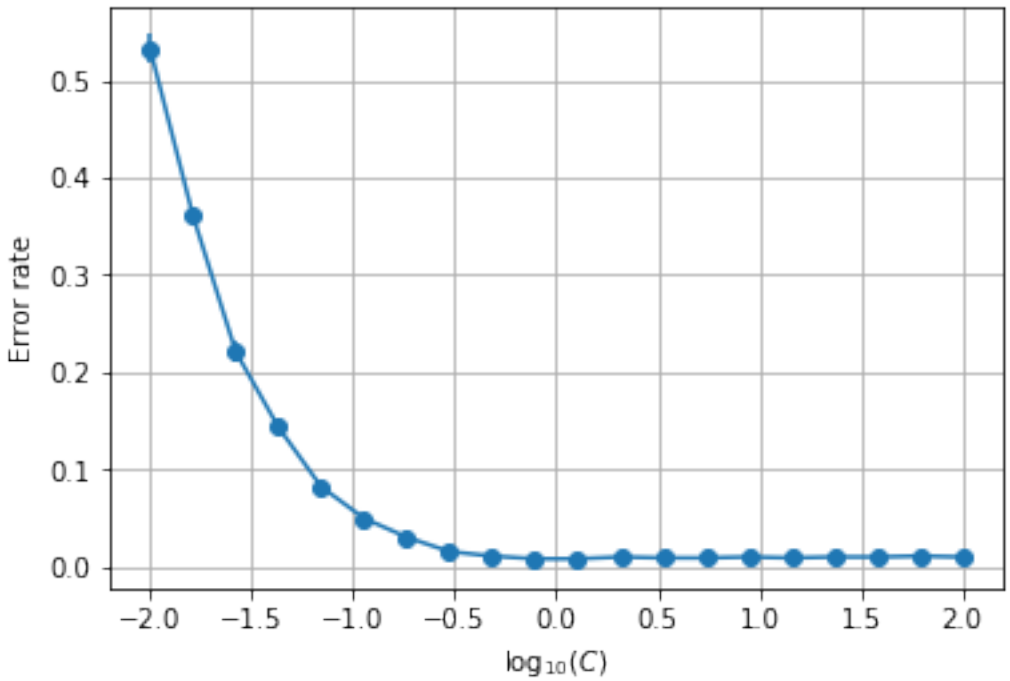
```
In [17]: # TODO
```

```
        err_mean = np.mean(err_rate, axis=1)
        num_nonzerocoef_mean = np.mean(num_nonzerocoef, axis=1)
        err_se = np.std(err_rate, axis=1) / np.sqrt(nfold-1)
        plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
        plt.grid()
        plt.xlabel(r'$\log_{10}(C)$')
        plt.ylabel('Error rate')

        imin = np.argmin(err_mean)

        print("The minimum test error rate = %0.4e, SE=%0.4e" % (err_mean[imin], err_se[imin])
        print("The C value corresponding to minimum error = %0.4e" % (C_test[imin]))
```

```
The minimum test error rate = 8.3333e-03, SE=2.9117e-03
The C value corresponding to minimum error = 7.8476e-01
```



We see that the minimum error rate is significantly below the classifier that did not use the l1-penalty. Use the one-standard error rule to determine the optimal `C` and the corresponding test error rate. Note that because `C` is inversely proportional to the regularization strength, you want to select a `C` as *small* as possible while meeting the error target!

In [18]:
```
        # TODO
        err_tgt = err_mean[imin] + err_se[imin]
        iopt = np.where(err_mean < err_tgt)[0][0]
        C_opt = C_test[iopt]
```

```
        print("Optimal C = %0.4e" % C_opt)
        print("The test error rate = %0.4e, SE=%0.4e" % (err_mean[iopt], err_se[iopt]))
        print('Accuracy = %0.4f, SE=%0.4f' % (1-err_mean[iopt], err_se[iopt]))
```

```
Optimal C = 4.8329e-01
The test error rate = 1.1111e-02, SE=2.3097e-03
Accuracy = 0.9889, SE=0.0023
```

Question: How does the test error rate compare with the classifier that did not use the l1-penalty? Explain why.

Type Answer Here:

The error rate with L1-penalty is a bit smaller than the classifier that did not use the L1-penalty. I think the reason is that L1-penalty shrink some of the features to be zero, which means these features play no role on the prediction. So the prediction will focus on the true related features. Thus, it leads to a smaller error rate.

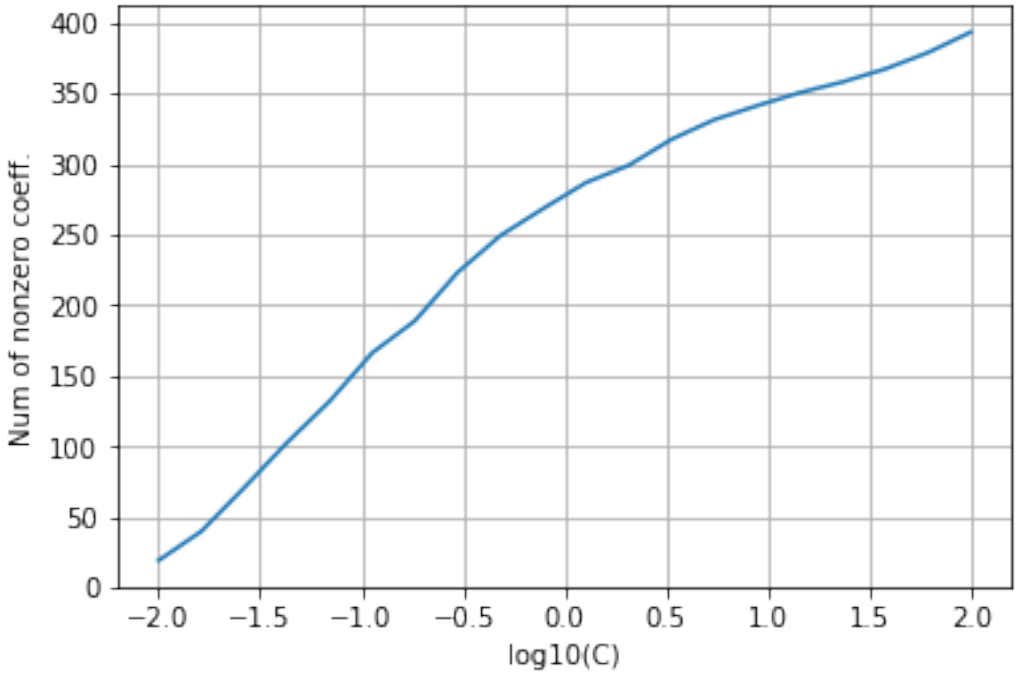
Now plot the nubmer of non-zero coefficients for the first class for different C values. Also determine and print the number of non-zero coefficients corresponding to C_opt.

```
In [19]: # TODO
        num_nonzerocoef_mean = np.mean(num_nonzerocoef, axis=1)
        plt.plot(np.log10(C_test),num_nonzerocoef_mean)

        plt.grid()
        plt.xlabel('log10(C)')
        plt.ylabel('Num of nonzero coeff.')

        print("The number of non-zero coefficients for the optimal C = %f" % num_nonzerocoef_
```

```
The number of non-zero coefficients for the optimal C = 249.300000
```

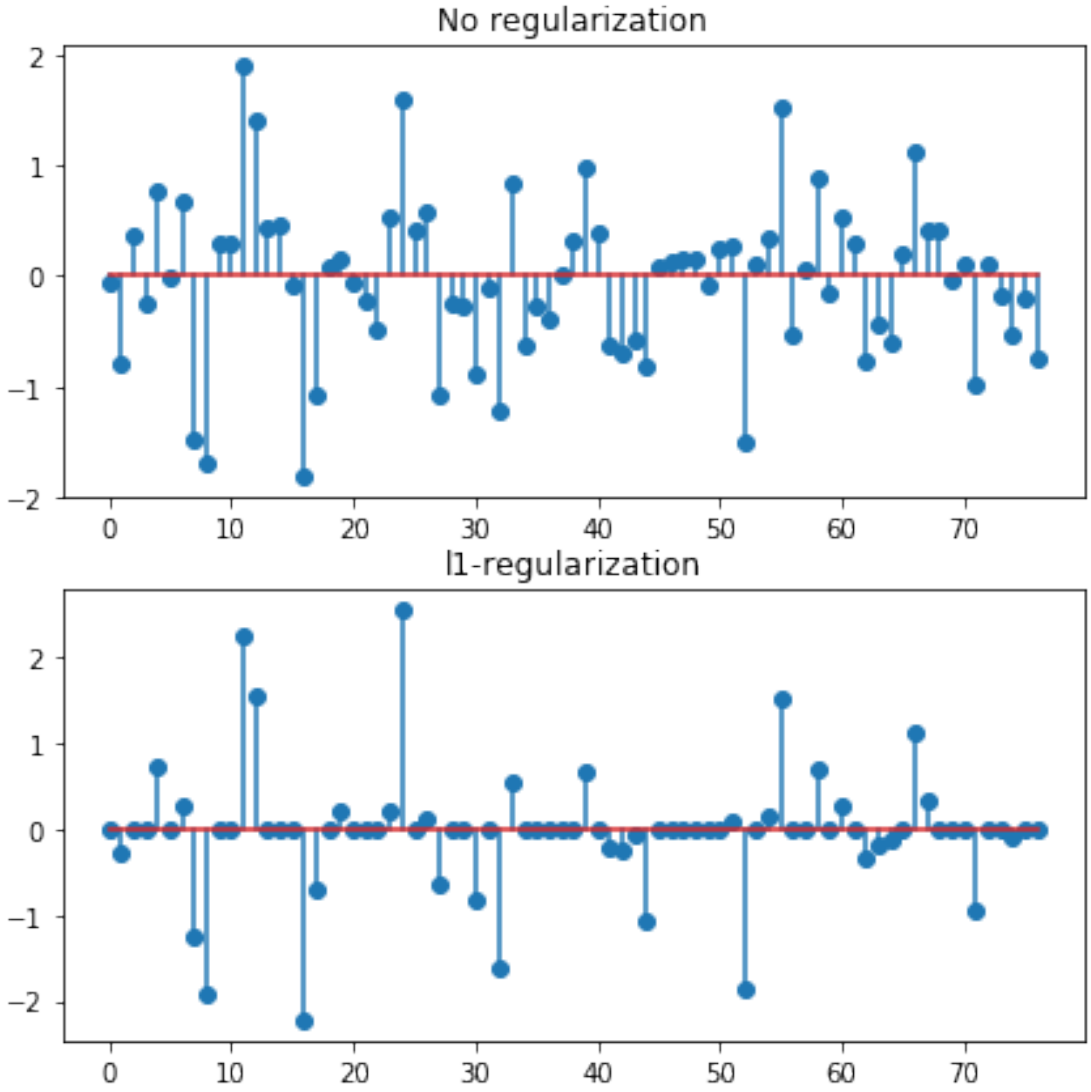


For the optimal `C`, fit the model on the entire training data with l1 regularization. Find the resulting weight matrix, `W_l1`. Plot the first row of this weight matrix and compare it to the first row of the weight matrix without the regularization. You should see that, with l1-regularization, the weight matrix is much more sparse and hence the roles of particular genes are more clearly visible. Please also compare the accuracy for the training data using optimal `C` with the previous results not using LASSO regularization. Do you expect the accuracy to improve?

```
In [20]: # TODO
         logreg = linear_model.LogisticRegression(C=C_opt,penalty='l1')
         logreg.C= C_opt
         logreg.fit(Xs,y)
         yhat = logreg.predict(Xs)
         acc = np.mean(yhat == y)
         print('Accuracy on the training data is {0:f}'.format(acc))
         W_l1 = logreg.coef_

         plt.figure(figsize=(7,7))
         plt.subplot(2,1,1)
         plt.stem(W[0,:])
         plt.title('No regularization')
         plt.subplot(2,1,2)
         plt.stem(W_l1[0,:])
         plt.title('l1-regularization')
         plt.show()
```

```
Accuracy on the training data is 0.997222
```


No regularization
2
1
0
−1
−2
0
10
20
30
40
50
60
70
l1-regularization
2
1
0
−1
−2
0
10
20
30
40
50
60
70