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 [2]:

```
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 [3]:

```
df = pd.read_excel('https://archive.ics.uci.edu/ml/machine-learning-databases/00
342/Data_Cortex_Nuclear.xls'
                  ,index_col='MouseID'   )




df.head(10)
```

```
Out[3]:
```

	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pC
MouseID								
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.3
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.2
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.2
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.1
309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.1
309_6	0.447506	0.628176	0.367388	2.385939	4.807635	0.218578	0.176233	2.1
309_7	0.428033	0.573696	0.342709	2.334224	4.473130	0.225173	0.184004	2.0
309_8	0.416923	0.564036	0.327703	2.260135	4.268735	0.214834	0.179668	2.0
309_9	0.386311	0.538428	0.317720	2.125725	4.063950	0.207222	0.167778	1.8
309_10	0.380827	0.499294	0.362462	2.096266	3.598587	0.227649	0.188093	1.7

10 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 [4]:

```
df1 = df.fillna(df.mean())
```

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 [5]:

```
print(df1['Genotype'].values)
rows, cols = df1.shape
y = np.zeros(rows);

yy = np.array(df1['Genotype'])
for k in range(rows):
    y[k] = int( yy[k] == 'Control')# converts the string 'control' to 1.0 and 'T
s65Dn' to 0.0

print(y)
```

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

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 [6]:

```
Xs = df1.as_matrix()

#Xs.shape
Xtr = Xs[:,:77] #without the last 4 cols
#Xtr = preprocessing.scale(Xtr)
Xtr.shape
```

Out[6]:

```
(1080, 77)
```

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

In [7]:

```
logreg = linear_model.LogisticRegression(C=1e5)
logreg.fit(Xtr, y)
```

Out[7]:

```
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=No
ne,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=Fals
e)
```

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 [8]:

```
yhat = logreg.predict(Xtr)
#measure the fraction of time that the predictions match the true labels
acc = np.mean(yhat == y)
print("Accuracy on training data = %f" % acc)
```

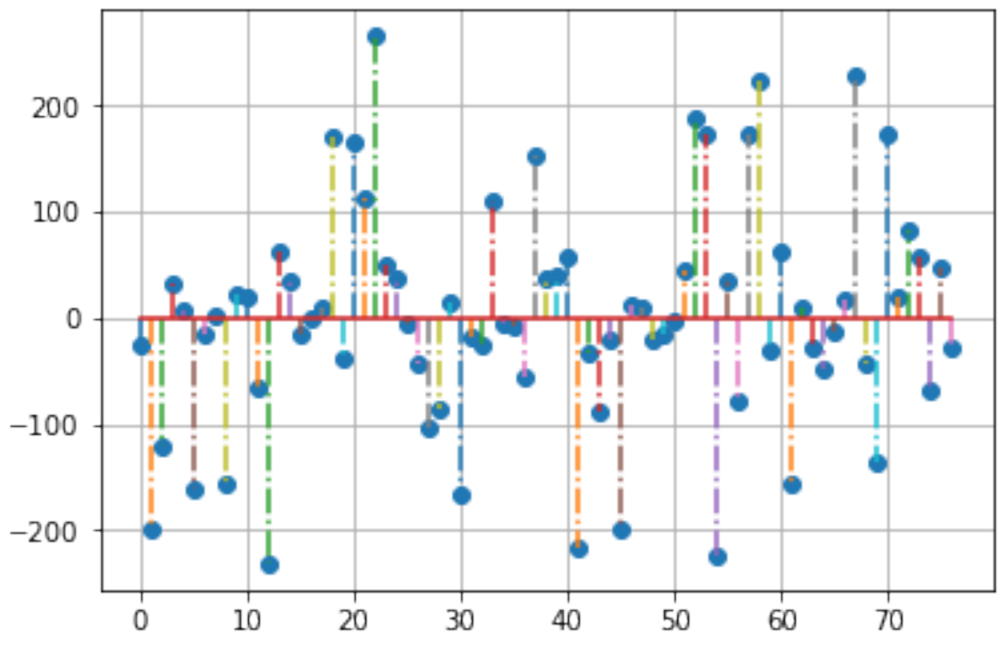
```
Accuracy on training data = 1.000000
```

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 [9]:

```
#data = {'feature': 'names', 'slope': np.squeeze(logreg.coef_)}
#dfslope = pd.DataFrame(data=data)
#dfslope
w = np.zeros(77)
w = logreg.coef_[0]

#plot stem
x = range (77)
plt.stem(x,w,'-.')
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. 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 [10]:

```
#print(logreg.coef_[0])
#print(abs(w))
#print(df1.columns.values)#hear are all the names of the genes -the df col names
colnames = np.array(df1.columns.values) # I put them in an array
# I then find the largest magnitude of w and record its index
#print(abs(w).argmax(axis=0))
index = abs(w).argmax(axis=0)
#the index of the w should be the col of the certain gene
#print(w[index])
#I then print the colnames at that index
print("this names of the genes for the first components i where the magnitude of
W[i] is largest:", colnames[index])

#####the second largest magnitude

#so I just drop the highest on from the last thing, by forcing it to 0
newW= np.array(w)
#print(newW)
newW[index] = 0.0
#print(newW)
index2 = abs(newW).argmax(axis=0)
#print(index2)
print("this names of the genes for the second components i where the magnitude o
f W[i] is largest:", colnames[index2])
```

```
this names of the genes for the first components i where the magnitu
de of W[i] is largest: CREB_N
this names of the genes for the second components i where the magnit
ude of W[i] is largest: PKCA_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, f1-score, and error rate across all the folds.

In [11]:

```
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support
nfold = 10 #10-fold cross validation

############Note, that in performing the cross-validation,
#you will want to randomly permute the test
#and training sets using the shuffle option.#################
kf = KFold(n_splits=nfold,shuffle=True)#shuffle=True
prec = []
```

```
rec = []

f1 = []
acc = []
for train, test in kf.split(Xtr):
    # Get training and test data
    X_tr = Xtr[train,:]
    y_tr = y[train]
    X_ts = Xtr[test,:]
    y_ts = y[test]

    # Fit a model
    logreg.fit(X_tr, y_tr)
    y_hat = logreg.predict(X_ts)

    # Measure performance
    preci,reci,f1i,_= precision_recall_fscore_support(y_ts,y_hat,average='binary
')
    prec.append(preci)
    rec.append(reci)
    f1.append(f1i)
    acci = np.mean(y_hat == y_ts)
    acc.append(acci)

# 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 the mean precision
print('Precision = {0:.4f}, SE={1:.4f}'.format(precm,prec_se))
#print the recall
print('Recall =    {0:.4f}, SE={1:.4f}'.format(recm, rec_se))
#print the f1-score
print('f1 =        {0:.4f}, SE={1:.4f}'.format(f1m, f1_se))
#print the error rate across all the folds
print('Accuracy =  {0:.4f}, SE={1:.4f}'.format(accm, acc_se))
```

```
Precision = 0.9638, SE=0.0093
Recall =    0.9520, SE=0.0082
f1 =        0.9574, SE=0.0055
Accuracy =  0.9556, SE=0.0053
```

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 [12]:

```
print(df1['class'].values)
rows, cols = df1.shape
y0 = np.zeros(rows);

yy0 = np.array(df1['class'])
print(np.unique(yy0))
for k in range(rows):
    if yy0[k] == 'c-CS-m':
        y0[k]= 0
    elif yy0[k] == 'c-CS-s':
        y0[k] = 1
    elif yy0[k] == 'c-SC-m':
        y0[k] = 2
    elif yy0[k] == 'c-SC-s':
        y0[k] = 3
    elif yy0[k] == 't-CS-m':
        y0[k] = 4
    elif yy0[k] == 't-CS-s':
        y0[k] = 5
    elif yy0[k] == 't-SC-m':
        y0[k] = 6
    else:
        y0[k]=7


print(y0)
```

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

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

In [13]:

```
log_reg2 = linear_model.LogisticRegression(C=1e5)
log_reg2.fit(Xtr, y0)

#classifier = svm.SVC(kernel='linear', C=0.01)
#y_pred = classifier.fit(X_train, y_train).predict(X_test)
```

Out[13]:

```
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=No
ne,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=Fals
e)
```

Measure the accuracy on the training data.

In [14]:

```
yyhat = log_reg2.predict(Xtr)
#measure the fraction of time that the predictions match the true labels
acc0 = np.mean(yyhat == y0)
print("Accuracy on training data = %f" % acc0)
```

```
Accuracy on training data = 1.000000
```

Now perform 10-fold cross validation, and measure the confusion matrix `cnf_matrix` on the test data in each fold. You can use the `confusion_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 `cnf_matrix[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 [19]:

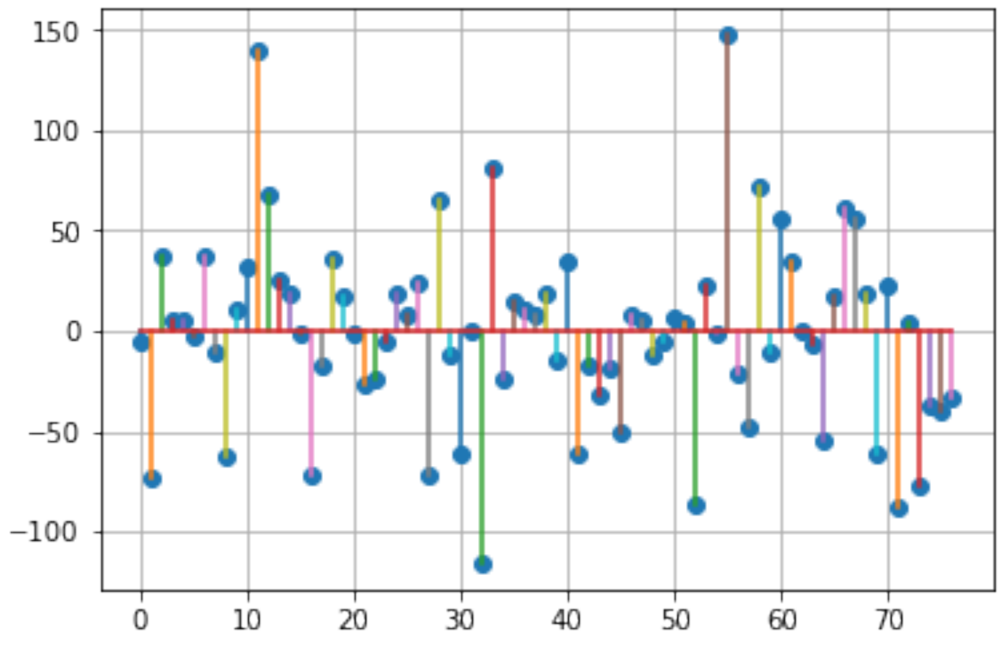
```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold

nfold = 10 #10-fold cross validation
kf2 = KFold(n_splits=nfold, shuffle= True)

#counts = np.zeros(Xs.shape)
#each element cnf_matrix[i,j] will represent the fraction of samples where yhat=
=j given yts2==i
cnf_matrix = np.zeros((8,8))

#for train, test in kf2.split(Xtr):
mse = np.zeros(nfold)
for ifold, ind in enumerate(kf2.split(Xtr)): # need to enumerate the splits to g
et all the
    # Get training and test data
    train,test = ind
    Xtr2 = Xtr[train,:]
    ytr2 = y0[train]
    Xts2 = Xtr[test,:]
    yts2 = y0[test]

    # Fit a model
    log_reg2.fit(Xtr2, ytr2)
    yhat2 = log_reg2.predict(Xts2)

    # Measure performance
    #measure the confusion matrix cnf_matrix on the test data in each fold
    cnf_matrix = cnf_matrix + confusion_matrix(yts2, yhat2) #Add the confusion m
atrix counts across all folds

    mse[ifold] = np.mean((yhat2-yts2)**2) #error rate across the folds

#print(cnf_matrix.shape)
#another matrix on the outside for the error
#Compute the prediction error on the test data

#print(np.array_str(cnf_matrix, precision=4, suppress_small=True))
C_norm = preprocessing.normalize(cnf_matrix)
print(np.array_str(C_norm, precision=4, suppress_small=True))

#print the overall mean and SE of the test error rate across the folds.

print (mse)
```

```
[[ 0.9999  0.0136  0.      0.      0.0068  0.      0.      0.    ]
 [ 0.0403  0.9983  0.      0.      0.0403  0.      0.0081  0.    ]
 [ 0.      0.      1.      0.      0.      0.      0.      0.    ]
 [ 0.0075  0.      0.      1.      0.      0.      0.      0.    ]
 [ 0.0075  0.0075  0.      0.      0.9999  0.      0.      0.    ]
 [ 0.      0.      0.      0.      0.      1.      0.      0.    ]
 [ 0.      0.      0.      0.      0.      0.      1.      0.    ]
 [ 0.      0.      0.      0.      0.      0.      0.      1.    ]]
[ 0.          0.24074074  0.00925926  0.          0.23148148  0.0925
9259
  0.25        0.24074074  0.01851852  0.09259259]
```

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 [20]:

```
log_reg3 = linear_model.LogisticRegression(C=1e5)
log_reg3.fit(Xtr,y0) #weighted coefficients #entire training data and get the we
ight coefficients

yhat = log_reg3.predict(Xtr)
#measure the fraction of time that the predictions match the true labels



l = np.zeros(77)
l = log_reg3.coef_[0]

#plot stem
x = range (77)
plt.stem(x,l,'-')
plt.grid()
```



L1-Regularization

Graduate students must 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`.

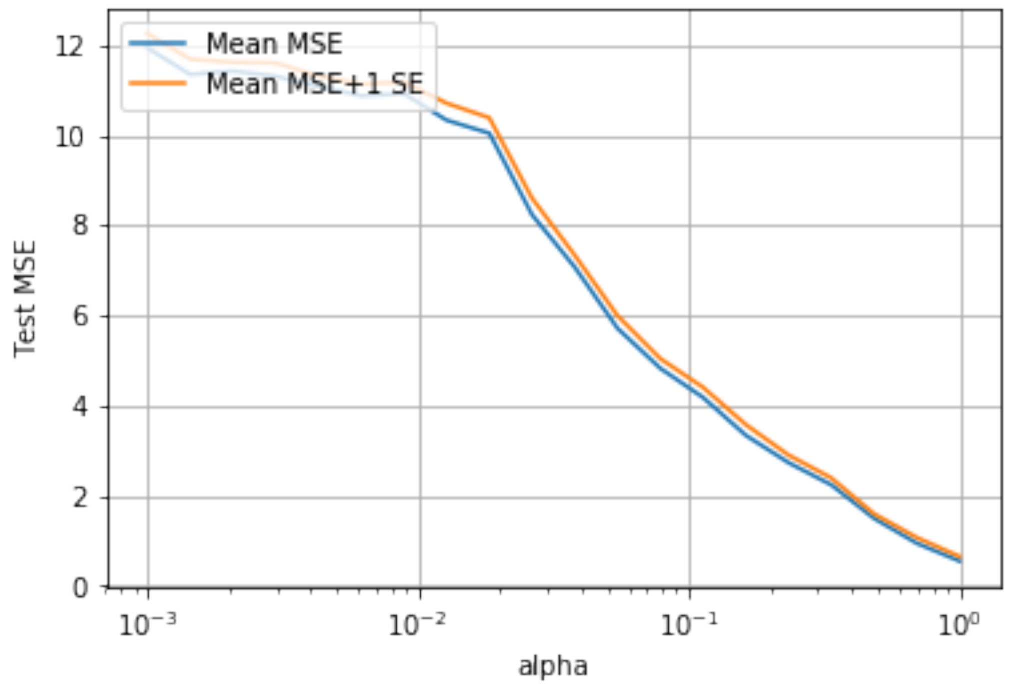
Using the model selection strategies from the prostate cancer analysis demo (../model_sel/prostate.ipynb), use K-fold cross validation to select an appropriate inverse regularization strength.

- Use 10-fold cross validation
- You should select around 20 values of `C`. It is up to you to 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 [45]:

```
# l1-penalty and the inverse regularization strength, C
# Create a k-fold cross validation object Use 10-fold cross validation
nfold = 10

# Regularization values to test
#You should select around 20 values of C. It is up to you to find a good range.

#log_reg4 = linear_model.LogisticRegression(C=20,penalty='l1')

# Regularization values to test
Crange = 20 # lower values of C increase the strength (greater weight)
alphas = np.logspace(-3,0,Crange)

# MSE for each alpha and fold value
mse2 = np.zeros((Crange,nfold))
for ifold, ind in enumerate(kf.split(Xtr)):


    # Get the training data in the split
    Itr,Its = ind
    X_tr_l = Xtr[Itr,:]
    y_tr_l = y0[Itr]
    X_ts_l = Xtr[Its,:]
    y_ts_l = y0[Its]

    #print(alphas)
    for ia, a in enumerate(alphas):
        #print(a)
        #You should select around 20 values of C. It is up to you to find a good
range,
        #my range is from .001-1.0 in log space
        log_reg4 = linear_model.LogisticRegression(C=a,penalty='l1')
        #Inverse of regularization strength; must be a positive float.
        #penalty='l1'
        # Fit the model on the training data
        log_reg4.fit(X_tr_l,y_tr_l)

        # Compute the prediction error on the test data
        y_ts_pred_l = log_reg4.predict(X_ts_l)
        mse2[ia,ifold] = np.mean((y_ts_pred_l-y_ts_l)**2)
        #print("y predict",y_ts_pred_l )
        #print("y",y0 )
        accNew = np.mean(y_ts_pred_l == y_ts_l)
#acc1 = np.mean(y_ts_pred_l == y0)

print("Accuracy on training data = %f" % accNew)

# Compute the mean and standard deviation over the different folds.
mse_mean = np.mean(mse2,axis=1)
mse_std = np.std(mse2,axis=1) / np.sqrt(nfold-1)

# Plot the mean MSE and the mean MSE + 1 std dev
plt.semilogx(alphas, mse_mean)
plt.semilogx(alphas, mse_mean+mse_std)
plt.legend(['Mean MSE', 'Mean MSE+1 SE'],loc='upper left')
plt.xlabel('alpha')
plt.ylabel('Test MSE')
plt.grid()
plt.show()
```

```
Accuracy on training data = 0.925926
```

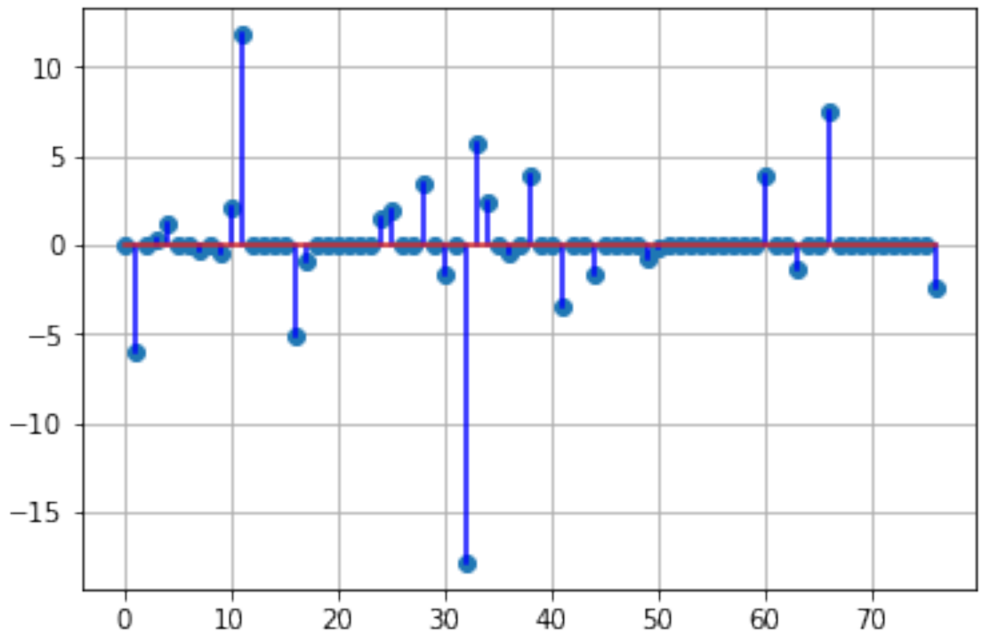


For the optimal `C`, fit the model on the entire training data with l1 regularization. Find the resulting weight matrix, `W_l1`. Plot the first row of this weight matrix and compare it to the first row of the weight matrix without the regularization. You should see that, with l1-regularization, the weight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

In [44]:

```
# Find the minimum MSE and MSE target
imin = np.argmin(mse_mean)
mse_tgt = mse_mean[imin] + mse_std[imin]
alpha_min = alphas[imin]

# Find the least complex model with mse_mean < mse_tgt
I = np.where(mse_mean < mse_tgt)[0]
iopt = I[-1]
alpha_opt = alphas[iopt]
print("Optimal c = %f" % alpha_opt)

log_reg5 = linear_model.LogisticRegression(C=alpha_opt,penalty='l1')# optimal c
is the last alphas_opt

log_reg5.fit(Xtr,y0) #weighted coefficients #entire training data and get the we
ight coefficients

yhat = log_reg5.predict(Xtr)

w_l1 = log_reg5.coef_[0]

#plot stem with l1 regularization
x = range (77)
plt.stem(x,w_l1,'b-')
plt.grid()
```

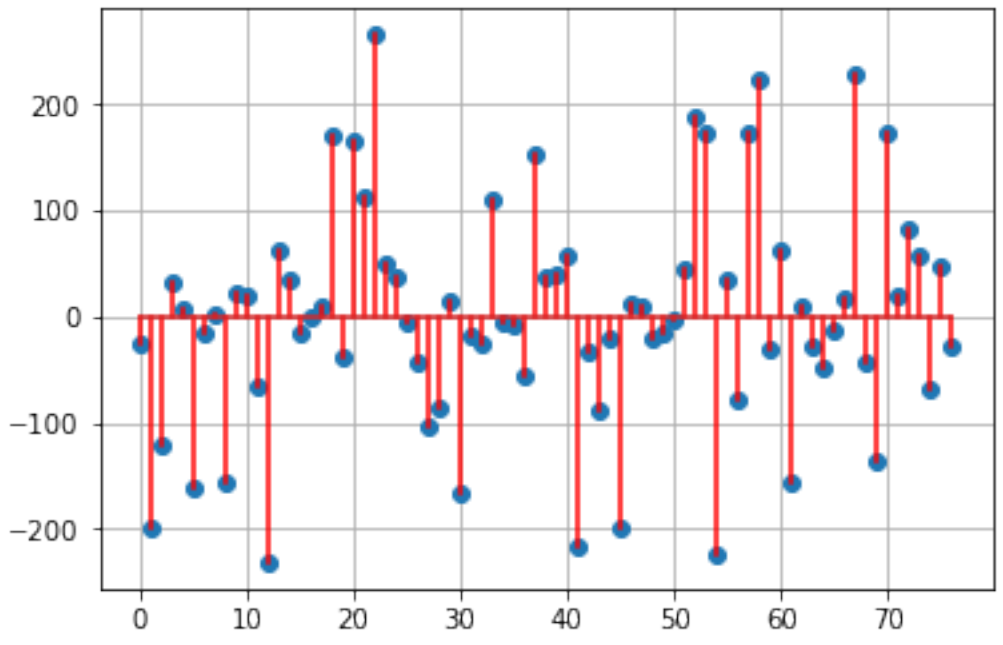
```
Optimal c = 1.000000
```



There are alot of zero elements as you can see above, which is what we expect, a more sparse plot

In [40]:

```
#plot stem with out l1 regularization
x = range (77)
plt.stem(x,w,'r-')
plt.grid()
```



the original is far more dense

In [ ]:

```
```