Logistic Regression for Gene Expression Data

In this exercise, we use logistic regression to predict biological characteristics ("phenotypes") from gene expression data. In doing this exercise, you will learn to:

- Handle missing data
- Perform multi-class logistic classification
- Create a confusion matrix (a table of the fraction of samples of class i being confused with class j)
- Use L1-regularization for improved estimation in the case of sparse weights

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 [178]:
```
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 [179]:
```
# TODO
df = pd.read_excel(io = "https://archive.ics.uci.edu/ml/machine-learning-databases/00342/Data_Cortex_Nucl
df.head()
```

Out[179]:

MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAMKII_N	pCREB_N	pELK_N	...	pCFOS_
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373744	0.232224	1.750936	...	0.1083
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292150	0.226972	1.596377	...	0.1043
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283337	0.230247	1.561316	...	0.1062
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152301	0.207004	1.595086	...	0.1112
309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134014	0.192158	1.504230	...	0.1106

5 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 [180]:
```
# TODO
df1 = df.fillna(df.mean())
df1.head()
```

Out[180]:

MouseID	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAMKII_N	pCREB_N	pELK_N	...	pCFOS_
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373744	0.232224	1.750936	...	0.1083
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292150	0.226972	1.596377	...	0.1043
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283337	0.230247	1.561316	...	0.1062
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152301	0.207004	1.595086	...	0.1112
309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134014	0.192158	1.504230	...	0.1106

5 rows × 81 columns

In [181]:
```
df1.isnull().values.any() # check to make sure there is no more missing data
```

Out[181]: False

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 [183]:
```
# TODO
df1["Genotype"].value_counts() # two types of categories
```

Out[183]:
```
Control    570
Ts65Dn     510
Name: Genotype, dtype: int64
```

In [184]:
```
df1["Genotype"] = np.where(df1["Genotype"].str.contains("Control"), 0, 1)
```

In [185]:
```
df1["Genotype"].value_counts() # ensure that categories have been converted to numbers
```

Out[185]:
```
0    570
1    510
Name: Genotype, dtype: int64
```

For the predictors, we will use all but the last four columns of the dataframes. First, get the names of these genes with the command:

```
xnames = df1.columns[:-4]
```

Then, get the data values corresponding to these columns and store the values in a matrix `X`. This matrix will have the expression levels for the 77 genes. Standardize the data matrix and call the standardized matrix `Xs`.

In [186]:
```
# Get names of the genes
xnames=df1.columns[:-4]

# TODO
X = df1.as_matrix(columns = xnames)
```

In [187]:
```
X.shape # 1080 observations, 77 genes
Xs =  preprocessing.scale(X = X) # scaled data matrix X
```

In [188]:
```
Xs.mean() # ensuring that it has been scaled to mean being negligibly 0
```

Out[188]:
```
-3.0417654392808523e-17
```

In [189]:
```
Xs.std() # ensuring that it has been scaled to unit variance
```

Out[189]:
```
1.0
```

Create a `LogisticRegression` object `logreg` and `fit` the training data. You can leave the arguments blank or specify `C=1`; those are equivalent. `C` is an inverse L2 regularization constant. Changing it changes the results.

In [190]:
```
# TODO
logreg = linear_model.LogisticRegression()
y = df1["Genotype"].as_matrix() # create training data's y values
logreg.fit(Xs, y)
```

Out[190]:
```
LogisticRegression(C=1.0, 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. Below, we will properly measure the accuracy on cross-validation data.

In [191]:
```
# TODO
yhat = logreg.predict(Xs)
np.mean(yhat == y) * 100 # The accuracy is 98.5%, which makes sense, since our "testing" data is the trai
```

Out[191]: 98.518518518518519

Interpreting the weight vector

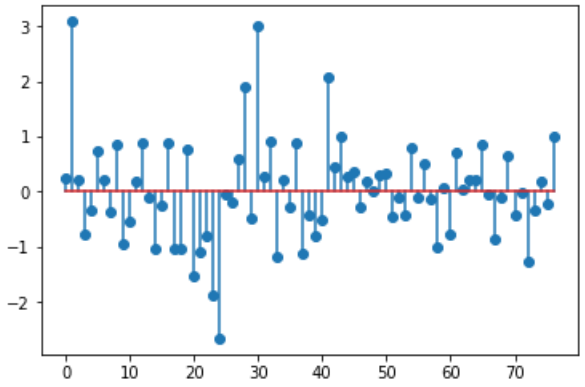
Create a stem plot of the coefficients, `W` in the logistic regression model. You can get the coefficients from `logreg.coef_`, but you will need to reshape this to a 1D array. (Unlike in R or matlab, in Python, an `n`-length row or column vector is considered to be of shape or size `n`. It is a 1-dimensional object, not an `nx1` or `1xn` object. So a 2D array consisting of only 1 row or 1 column is not the same thing as being a 1D array. You can use the `ravel` method where if `u` is `1xn`, `u.ravel()` would return a 1d version of `u`.)

In [192]:
```
# TODO
W = logreg.coef_
W = W.ravel() # convert to 1D
W # values of coefficients
```

Out[192]:
```
array([ 0.25788237,  3.08886725,  0.20267443, -0.78647572, -0.32418999,
        0.74507632,  0.21224994, -0.36626878,  0.85923466, -0.95412431,
       -0.53123443,  0.1889532 ,  0.88912974, -0.11415406, -1.03286982,
       -0.26257207,  0.88237645, -1.0419564 , -1.04696612,  0.77373353,
       -1.53905206, -1.08339142, -0.81271391, -1.88436998, -2.66120895,
       -0.04662753, -0.18580505,  0.58643322,  1.91232686, -0.48261052,
        3.00930717,  0.26481266,  0.90438089, -1.17762606,  0.2017229 ,
       -0.27700545,  0.89465152, -1.12743686, -0.43695293, -0.81224319,
       -0.50092671,  2.07286406,  0.44784534,  1.00574679,  0.28011077,
        0.37450331, -0.28079098,  0.18973779,  0.02318002,  0.31222469,
        0.3359203 , -0.45301172, -0.10554243, -0.42122213,  0.80197349,
       -0.11599938,  0.5109602 , -0.13148415, -1.01969277,  0.07937937,
       -0.77013546,  0.7187128 ,  0.02947349,  0.20564635,  0.20805381,
        0.8470437 , -0.03576105, -0.85109339, -0.09409808,  0.66664633,
       -0.42122213, -0.02432561, -1.25425538, -0.34626058,  0.17505342,
       -0.20960458,  0.990545  ])
```

In [193]:
```
# Plot W
plt.stem(W) # plotting coefficients' magnitude
```

Out[193]: `<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. You could use the command np.argsort with the appropriate arguments.

In [194]:
```
# TODO
mag = np.argsort(a = -abs(W)) # sort the array by decreasing magnitude
df1.columns.values[mag[1]] # APP_N has highest magnitude of W
```

Out[194]: 'APP_N'

In [195]:
```
df1.columns.values[mag[2]] # ERK_N has second highest magnitude of W
```

Out[195]: 'ERK_N'

Cross Validation

The above meaured the accuracy on the training data. It is better to measure the accuracy on the test data. Perform 10-fold cross validation and measure the accuracy. 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 accuracy across all the folds. (If you already did the homework and found precision, f1-score and recall, that is fine. You do not have to redo this slightly easier version. If you want to the latter, you will need to import precision_recall_fscore_support from sklearn.metrics.)

In [196]:
```
# TODO A possible beginning here
from sklearn.model_selection import KFold

# This line is not necesssary if you are just finding accuracy
from sklearn.metrics import precision_recall_fscore_support

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

# find accuracy on each fold
acc = list()

# fit model on training data and use that on testing data
for train, test in kf.split(Xs):

    # split up data into 4 parts
    Xtrain = Xs[train]
    Xtest = Xs[test]
    ytrain = y[train]
    ytest = y[test]

    # create logistic regression model
    logreg = linear_model.LogisticRegression()
    logreg.fit(X = Xtrain, y = ytrain) # fit training model for each fold

    # use training model to predict testing data
    yhat = logreg.predict(Xtest)

    # add to list the accuracy for each fold
    acc.append(np.mean(yhat == ytest) * 100)

# accuracy for each fold
print(acc)

# total accuracy over all folds
print(np.mean(acc)) # this accuracy is just slightly lower than the accuracy on just over the training da
```

```
[97.222222222222214, 96.296296296291, 98.148148148148152, 96.296296296291, 97.222222222222214, 9
7.222222222222214, 99.074074074076, 95.370370370367, 95.370370370367, 97.222222222222214]
96.9444444444
```

Multi-Class Classification

Now use the response variable in `df1['class']`. This data has 8 possible classes from the three binary properties of mice described above. Use the `np.unique` funtion as before to convert this to a vector `y` with values 0 to 7. You can see them by starting below.

In [197]:
```
ystr = df1['class'].values
# TODO
# all categories of variable "class"
df1["class"].value_counts()

# convert categories into numbers
ystr[np.where(ystr == "c-CS-m")] = 0
ystr[np.where(ystr == "c-SC-m")] = 1
ystr[np.where(ystr == "t-SC-m")] = 2
ystr[np.where(ystr == "t-SC-s")] = 3
ystr[np.where(ystr == "t-CS-m")] = 4
ystr[np.where(ystr == "c-CS-s")] = 5
ystr[np.where(ystr == "c-SC-s")] = 6
ystr[np.where(ystr == "t-CS-s")] = 7

# check to ensure they are all numbers now
print(np.unique(ystr))

# make it y
y = ystr
```

```
[0 1 2 3 4 5 6 7]
```

Fit a multi-class logistic model by creating a `LogisticRegression` object, `logreg` and then calling the `logreg.fit` method. Again, use C=1 or leave the arguments blank.

In [198]:
```
# TODO
logreg = linear_model.LogisticRegression()
y = y.astype("int") # convert to int
logreg.fit(X = Xs, y = y)
```

Out[198]:
```
LogisticRegression(C=1.0, 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 on the training data.

In [199]:
```
# TODO
yhat = logreg.predict(X = Xs)
np.mean(yhat == y)*100 # this is a high accuracy rate on the multi-class classification!
```

Out[199]:
```
99.907407407407405
```

Now perform 10-fold cross validation, and measure the confusion matrix `C` 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 `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 [200]:
```
# TODO  POssible beginning
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold

logreg = linear_model.LogisticRegression()

# Initialize the confusion matrix counts
ny = np.max(y)
C = np.zeros((ny+1,ny+1))

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

# find accuracy over each fold
acc = list()

# fit model on training data and use that on testing data
for train, test in kf.split(Xs):

    # split up data into 4 parts
    Xtrain = Xs[train]
    Xtest = Xs[test]
    ytrain = y[train]
    ytest = y[test]

    # create logistic regression model
    logreg = linear_model.LogisticRegression()
    logreg.fit(X = Xtrain, y = ytrain) # fit training model for each fold

    # use training model to predict testing data
    yhat = logreg.predict(Xtest)

    # add to list the accuracy for each fold
    acc.append(np.mean(yhat == ytest) * 100)

    # create confusion matrix
    C = np.add(C, confusion_matrix(y_true = ytest, y_pred = yhat))

# standard error across all folds
C = preprocessing.normalize(C)

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

# mean error across all folds
print(np.mean(np.diagonal(C)))

# mean standard error across all folds
print(np.std(np.diagonal(C)))
```

```
[[ 0.9998  0.      0.      0.      0.0205  0.0068  0.      0.    ]
 [ 0.      1.      0.      0.0067  0.      0.      0.      0.    ]
 [ 0.      0.      1.      0.      0.      0.      0.      0.    ]
 [ 0.      0.      0.      1.      0.      0.      0.      0.    ]
 [ 0.0076  0.      0.      0.      0.9999  0.0151  0.      0.    ]
 [ 0.0077  0.      0.      0.      0.0154  0.9997  0.      0.0154]
 [ 0.0075  0.      0.      0.      0.      0.      1.      0.    ]
 [ 0.      0.      0.      0.      0.      0.      0.      1.    ]]
0.999913192027
0.000104523824158
```

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 predicting the first class.

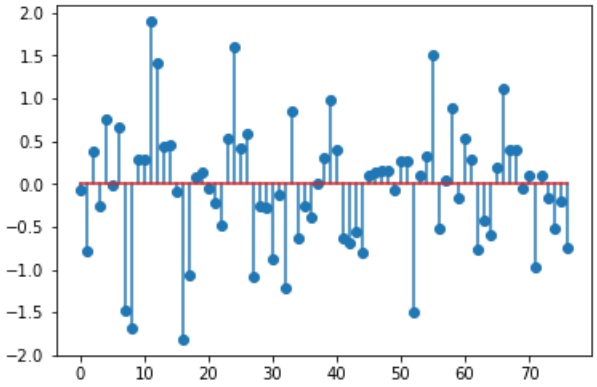
In [201]:
```
# TODO
logreg = linear_model.LogisticRegression() # create model
y = y.astype("int") # convert into int

# fit model
logreg.fit(X = Xs, y = y)

# find coefficients
W = logreg.coef_
W = W[0, ] # first row of matrix

# stem plot
plt.stem(W)
```

Out[201]: `<Container object of 3 artists>`



L1-Regularization

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`.

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
- Use the one SE rule to find the optimal `C`.
- How does the accuracy compare to the accuracy achieved without regularization.

In [205]:
```
# TODO:  Perform the cross validation to compute the accuracy per fold and per C
c = np.logspace(-2, 2, num = 20) # choosing 20 values from range of c
acc = np.zeros(shape = (20, 10)) # accuracy matrix per fold for all 20 values of C

# fit over all training models and fit onto testing data
for isplit, [train, test] in enumerate(kf.split(Xs)):

    # print out each fold
    print("Fold = %d " % isplit)

    # split up data into 4 parts
    Xtest = Xs[test]
    Xtrain = Xs[train]
    ytrain = y[train]
    ytest = y[test]

    # loop through all c values
    for each, i in enumerate(c):

        # fit model with l1 penalty
        logreg = linear_model.LogisticRegression(penalty = "l1", C = i)
        logreg.fit(X = Xtrain, y = ytrain)

        # predict values
        yhat = logreg.predict(Xtest)

        # add to accuracy matrix
        acc[each, isplit] = np.mean(ytest == yhat)

# print out accuracy matrix
print(acc)

# total accuracy over all folds
print(np.mean(acc) * 100)
```

```
Fold = 0
Fold = 1
Fold = 2
Fold = 3
Fold = 4
Fold = 5
Fold = 6
Fold = 7
Fold = 8
Fold = 9
[[ 0.5         0.5         0.48148148  0.48148148  0.50925926  0.49074074
   0.40740741  0.37037037  0.50925926  0.53703704]
 [ 0.63888889  0.68518519  0.62962963  0.68518519  0.67592593  0.62962963
   0.59259259  0.55555556  0.67592593  0.62037037]
 [ 0.74074074  0.75925926  0.75        0.86111111  0.77777778  0.78703704
   0.76851852  0.75925926  0.84259259  0.78703704]
 [ 0.85185185  0.84259259  0.85185185  0.93518519  0.88888889  0.85185185
   0.86111111  0.85185185  0.86111111  0.87037037]
 [ 0.87962963  0.89814815  0.89814815  0.93518519  0.94444444  0.89814815
   0.88888889  0.9537037   0.89814815  0.94444444]
 [ 0.93518519  0.91666667  0.94444444  0.96296296  0.97222222  0.9537037
   0.92592593  0.97222222  0.9537037   0.96296296]
 [ 0.9537037   0.9537037   0.97222222  0.99074074  0.98148148  0.97222222
   0.94444444  0.98148148  0.97222222  0.98148148]
 [ 0.97222222  0.99074074  0.99074074  0.99074074  0.98148148  0.98148148
   0.9537037   0.99074074  0.99074074  0.98148148]
 [ 0.97222222  0.99074074  0.99074074  0.99074074  0.98148148  0.99074074
   0.96296296  1.          0.99074074  0.98148148]
 [ 0.98148148  0.99074074  0.99074074  1.          0.99074074  1.
   0.96296296  1.          0.99074074  0.98148148]
 [ 0.99074074  0.99074074  1.          1.          0.98148148  1.
   0.96296296  1.          0.99074074  0.99074074]
 [ 1.          0.99074074  1.          1.          0.98148148  1.
   0.96296296  1.          0.99074074  0.99074074]
 [ 1.          0.99074074  1.          1.          0.98148148  1.
   0.96296296  1.          0.99074074  0.99074074]
```

```
 [ 1.          0.99074074  1.          1.          0.98148148  1.
   0.96296296  1.          0.99074074  0.99074074]
 [ 1.          0.99074074  1.          1.          0.98148148  1.
   0.96296296  0.99074074  0.99074074  0.99074074]
 [ 1.          0.98148148  1.          1.          0.97222222  1.
   0.96296296  0.99074074  0.99074074  0.99074074]
 [ 1.          0.98148148  1.          1.          0.97222222  1.
   0.96296296  0.99074074  0.99074074  0.99074074]
 [ 1.          0.98148148  0.99074074  1.          0.97222222  1.
   0.96296296  0.99074074  0.99074074  0.99074074]
 [ 1.          0.98148148  0.99074074  1.          0.97222222  1.
   0.97222222  0.99074074  0.99074074  0.99074074]
 [ 1.          0.98148148  0.99074074  1.          0.97222222  1.
   0.98148148  0.99074074  1.          0.99074074]]
92.3009259259
```

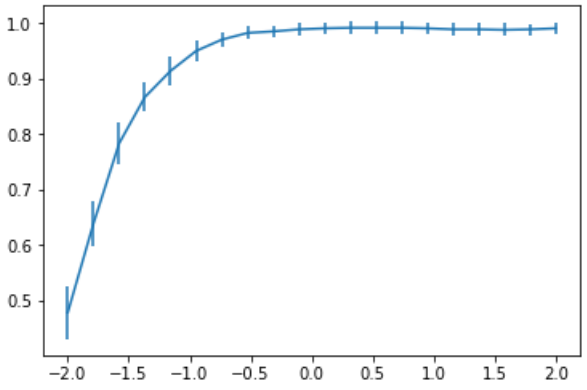
In [206]:
```
# TODO:  Compute the mean and one SE accuracy and plot with errorbars.
c_avg = acc.mean(axis = 1)
c_sd = acc.std(axis = 1)

# print values
print(c_avg)
print(c_sd)

# plot error bars
plt.errorbar(np.log10(c), c_avg, yerr = c_sd)
```

```
[ 0.4787037   0.63888889  0.78333333  0.86666667  0.91388889  0.95
  0.97037037  0.98240741  0.98518519  0.98888889  0.99074074  0.99166667
  0.99166667  0.99166667  0.99074074  0.98888889  0.98888889  0.98796296
  0.98888889  0.99074074]
[ 0.04812143  0.04078281  0.03731378  0.02592593  0.02587628  0.01814437
  0.01422434  0.01130237  0.01031067  0.01079806  0.0109557   0.01130237
  0.01130237  0.01130237  0.0109557   0.0122838   0.0122838   0.01174868
  0.00997253  0.00925926]
```

Out[206]: <Container object of 3 artists>



In [208]:
```
# TODO:  Compute the optimal C with the one SE rule
i_max = np.argmax(c_avg)
c_opt = c[i_max] # c val with highest accuracy
se_opt = c_sd[i_max]
thresh = c_avg[i_max] - se_opt

# which c_avg are in thresh?
ind = [i for i in range(len(c_avg)) if c_avg[i] > thresh and c_avg[i] < c_avg[i_max]]
c_vals = [c[i] for i in ind]
c_se_opt = c[np.argmax(c_vals)] # find max

# optimal c
print(c_se_opt)
```

```
0.784759970351
```

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 [209]:
```
# TODO
# fit model with l1 penalty
logreg = linear_model.LogisticRegression(penalty = "l1", C = c_se_opt)
logreg.fit(X = Xs,y = y)

# find weights
W_l1 = logreg.coef_
W_l1 = W_l1[0, ] # first row of weight matrix

# stemplot
plt.stem(W_l1)
```

Out[209]: `<Container object of 3 artists>`