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 [105]:
```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import linear_model, preprocessing
```

In [106]:
```
import sys
sys.version
sys.version_info
```

Out[106]:
```
sys.version_info(major=3, minor=6, micro=1, releaselevel='final', seria
l=0)
```

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 [107]:
```
# TODO
# df = ...
link = "./Data_Cortex_Nuclear.xls"
df = pd.read_excel(link, index_col=0)

df.head()
```

Out[107]:

	DYRK1A_N	ITSN1_N	BDNF_N	NR1_N	NR2A_N	pAKT_N	pBRAF_N	pCAM
MouseID								
309_1	0.503644	0.747193	0.430175	2.816329	5.990152	0.218830	0.177565	2.373
309_2	0.514617	0.689064	0.411770	2.789514	5.685038	0.211636	0.172817	2.292
309_3	0.509183	0.730247	0.418309	2.687201	5.622059	0.209011	0.175722	2.283
309_4	0.442107	0.617076	0.358626	2.466947	4.979503	0.222886	0.176463	2.152
309_5	0.434940	0.617430	0.358802	2.365785	4.718679	0.213106	0.173627	2.134

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 [108]:
```
# TODO
# df1 = ...
print("num null df:", df.isnull().sum().sum())
# print("num null df1:", df1.isnull().sum().sum())
df1 = df.fillna(df.mean())
print("Mean^2 Same within:",((df1.mean() - df.mean())**2).sum())
print("num null df:", df.isnull().sum().sum())
print("num null df1:", df1.isnull().sum().sum())
```

```
num null df: 1396
Mean^2 Same within: 3.73630409211e-30
num null df: 1396
num null df1: 0
```

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 [109]:
```
# TODO
# y = ...

yraw = df1['Genotype'].values
negVal, posVal = np.unique(yraw)
y = (yraw == posVal).astype(int)

# y1 = (df1['Genotype'].values == np.unique(yraw)[1]).astype(int)
```

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 [110]:
```
# TODO
# Xs = ...

X = np.array(df1[ df1.columns[:-4] ])
Xs = (X - np.mean(X, axis=0))/np.std(X, axis=0)
```

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

In [111]:
```
# TODO
logreg = linear_model.LogisticRegression()
logreg.fit(Xs, y)
```

Out[111]:
```
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=0.00
01,
          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 [112]:
```
# TODO
yhat = logreg.predict(Xs)
print("The accuracy proportion is {:.3f}.".format(np.sum(y ==
yhat)/len(y)))
```

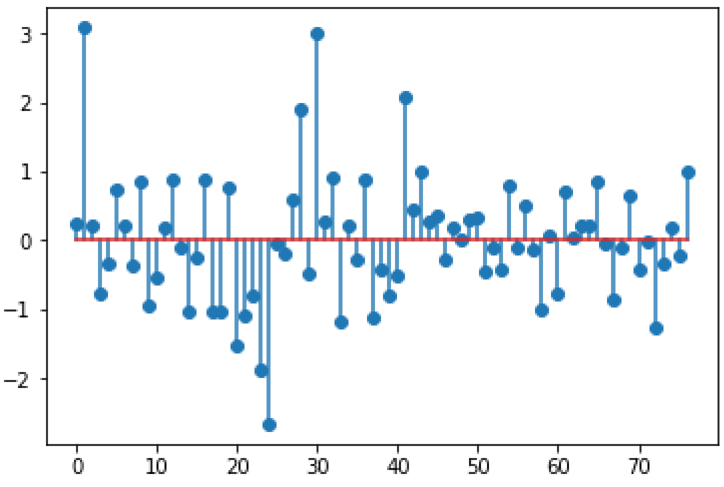
```
The accuracy proportion is 0.985.
```

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 [113]: # TODO
          logreg.intercept_
          W = np.squeeze(logreg.coef_)
          plt.stem(W)
```

```
Out[113]: <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 [114]: # TODO
          W_sort_idx = (-abs(W)).argsort()
          print("\n".join(df1.columns[W_sort_idx[:2]]))
```

```
ITSN1_N
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 [123]:
```
import sklearn
from sklearn import model_selection
from sklearn import metrics
# TODO
nfold = 10

kf = model_selection.KFold(n_splits=nfold,shuffle=True)

prfa = np.zeros((nfold, 4))

print("pre\trecall\tf_1\terr")
for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind
    X_cv_tr = Xs[Itr]
    X_cv_ts = Xs[Its]

    y_cv_tr = y[Itr]
    y_cv_ts = y[Its]
    logreg.fit(X_cv_tr, y_cv_tr)

    y_hat = logreg.predict(X_cv_ts)

#     print(("{:.3f}\t"*3).format(
#         *metrics.precision_recall_fscore_support(y[Its], y_hat, averag
e='binary')) )

    prfa[isplit, :] = metrics.precision_recall_fscore_support(
            y[Its], y_hat, average='binary') #[:3]

    prfa[isplit, 3] = 1-np.mean(y[Its]==y_hat)

    print(("{:.3f}\t"*4).format(*prfa[isplit, :]))

print(("-"*29+"\n"+"{:.3f}\t"*4).format(*np.mean(prfa, axis=0)))
```

```
pre     recall  f_1     err
0.961   1.000   0.980   0.019
0.980   1.000   0.990   0.009
0.957   0.918   0.938   0.056
0.979   1.000   0.989   0.009
0.962   0.980   0.971   0.028
1.000   0.979   0.989   0.009
1.000   0.903   0.949   0.056
0.980   0.960   0.970   0.028
0.945   0.963   0.954   0.046
1.000   0.923   0.960   0.037
-----------------------------
0.976   0.963   0.969   0.030
```

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 [124]:
```
# TODO
# y = ...
yraw = df1['class']
le = preprocessing.LabelEncoder()
le.fit(yraw)
y = le.transform(yraw)

# with unique
y = np.zeros_like(yraw)
yraw = df1['class']
col_names = np.unique(yraw)
nam2num = dict([*zip(col_names, np.arange(8))])

y = np.array([*map((lambda x : nam2num[x]), yraw)])
```

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

In [125]:
```
# TODO
logreg = linear_model.LogisticRegression(C=1e5)
logreg.fit(Xs, y)
```

Out[125]:
```
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 on the training data.

In [126]:
```
# TODO
print("Acc: {:.3f}%".format(100*np.mean(logreg.predict(Xs) == y)))
```

```
Acc: 100.000%
```

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

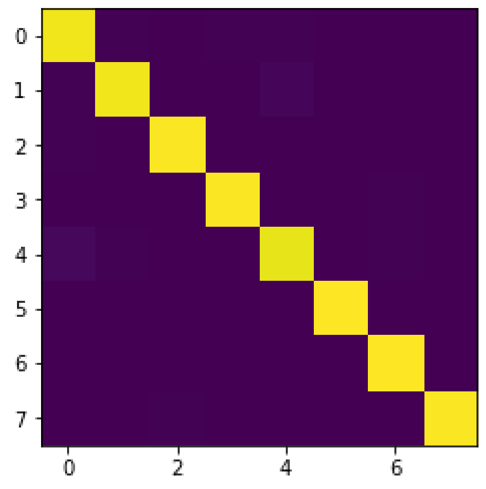
```
print(np.array_str(C, precision=4, suppress_small=True))
```

to create a nicely formatted print. Also print the overall mean and SE of the test error rate across the folds.

In [127]:
```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import KFold

# TODO
nfold = 10

kf = model_selection.KFold(n_splits=nfold,shuffle=True)

logreg = linear_model.LogisticRegression()

C = np.zeros((len(col_names),len(col_names)))

for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind

    X_cv_tr = Xs[Itr]
    X_cv_ts = Xs[Its]

    y_cv_tr = y[Itr]
    y_cv_ts = y[Its]


    logreg.fit(X_cv_tr, y_cv_tr)

    y_hat = logreg.predict(X_cv_ts)

    C += confusion_matrix(y_hat, y_cv_ts)

C = C / np.sum(C, axis=1)
# print(np.array_str(C, precision=4, suppress_small=True))

# errMean = 1-np.sum(C*np.eye(8))/8
# errStd = np.std(C)

acc = np.diagonal(C)
mu = np.mean(acc)
sigma = np.std(acc)
print("The mean and SE of the test error are {:.3f} and {:.3f} respectiv
ely."
      .format(mu, sigma/np.sqrt(nfold-1)))
```

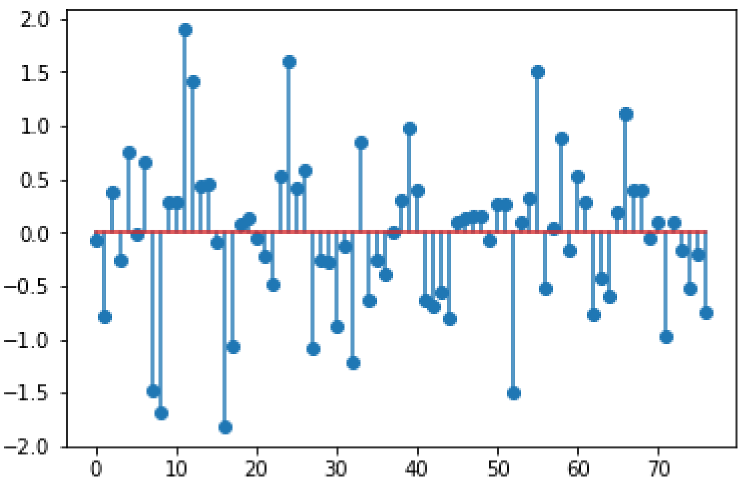
```
The mean and SE of the test error are 0.987 and 0.004 respectively.
```

```
In [128]: print("\n".join([("{:.2f}\t"*row.shape[0]).format(*row) for row in C]))
          plt.imshow(C)

          0.98    0.01    0.00    0.01    0.01    0.00    0.00    0.00
          0.01    0.98    0.00    0.00    0.01    0.00    0.00    0.00
          0.01    0.00    0.99    0.00    0.00    0.00    0.00    0.00
          0.00    0.00    0.00    0.99    0.00    0.00    0.01    0.00
          0.02    0.01    0.00    0.00    0.96    0.00    0.01    0.00
          0.00    0.00    0.00    0.00    0.00    1.00    0.00    0.00
          0.00    0.00    0.00    0.00    0.00    0.00    1.00    0.00
          0.00    0.00    0.01    0.00    0.00    0.00    0.00    0.99

Out[128]: <matplotlib.image.AxesImage at 0x116c64c18>
```



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 [129]: # TODO
          logreg.fit(Xs, y)
          W = logreg.coef_
          plt.stem(W[0,:])

Out[129]: <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`.

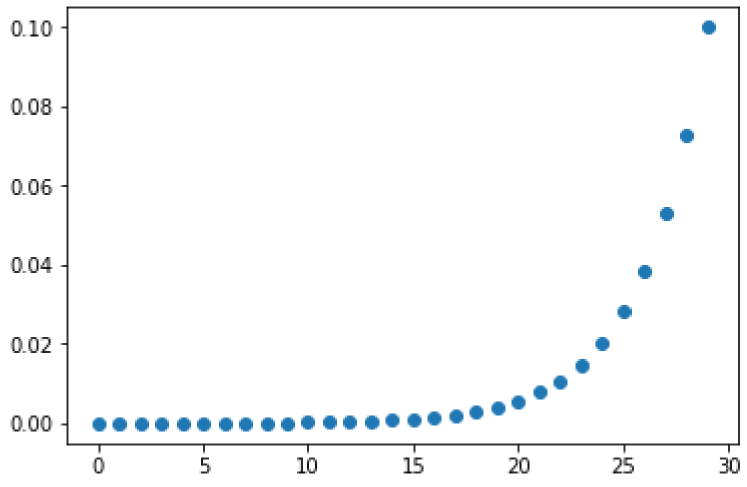
Using the model selection strategies from the prostate cancer analysis demo (../model_sel/prostate.ipynb), use K-fold cross validation to select an appropriate inverse regularization strength.

- 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 [130]:
```
logreg = linear_model.LogisticRegression()
print(logreg)
Cs = np.logspace(-5, -1, 30)
plt.plot(Cs, 'o')
```

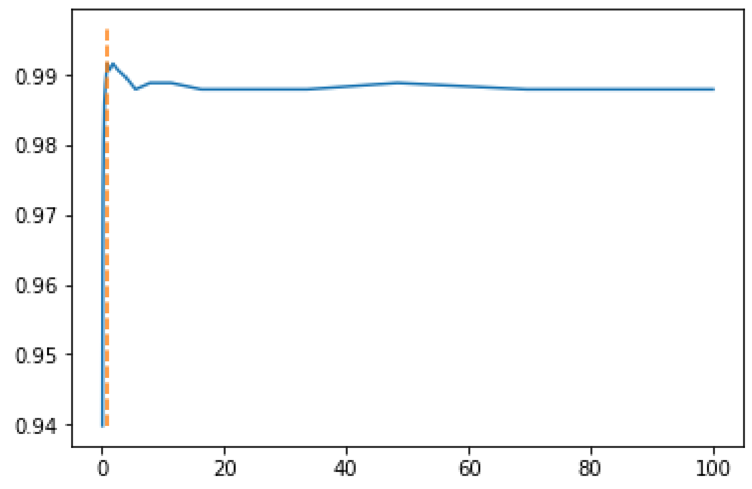
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=
True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=
1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.00
01,
          verbose=0, warm_start=False)
```

Out[130]:
```
[<matplotlib.lines.Line2D at 0x116dc0e80>]
```



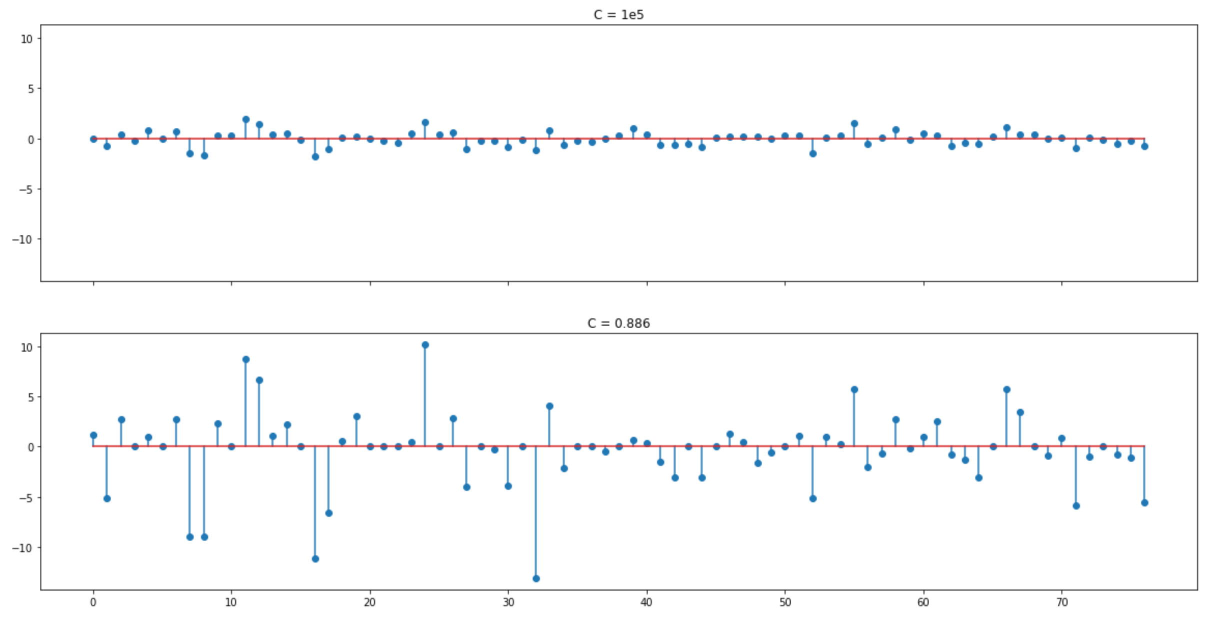
In [131]:
```
# TODO
nfold = 10

kf = model_selection.KFold(n_splits=nfold,shuffle=True)



Cs = np.logspace(-1, 2, 20)
acc = np.zeros((nfold, len(Cs)))

for isplit, Ind in enumerate(kf.split(Xs)):
    Itr, Its = Ind

    X_cv_tr = Xs[Itr]
    X_cv_ts = Xs[Its]

    y_cv_tr = y[Itr]
    y_cv_ts = y[Its]

    for ic, c in enumerate(Cs):
        logreg = linear_model.LogisticRegression(C=c, penalty='l1')
        logreg.fit(X_cv_tr, y_cv_tr)

        y_hat = logreg.predict(X_cv_ts)

        acc[isplit, ic] = np.mean(y_hat == y_cv_ts)
#         print(np.mean(y_hat == y_cv_ts))

#     print(np.array([*zip(Cs, acc[isplit, :])]).shape)
#     plt.figure()
#     plt.plot([*zip(Cs, acc)])

#     plt.plot(Cs, acc, label=isplit)
#     plt.legend()
accMean = np.mean(acc, axis=0)
```

```
In [148]: accMean = np.mean(acc, axis=0)
          # print(acc.shape)
          # print(accMean.shape)

          srtIdx = (-accMean).argsort()
          accMean[srtIdx]
          Cs[srtIdx[:10]]

          optC = Cs[srtIdx[0]]
          print("OptimalC is {:.5f}".format(optC))

          plt.plot(Cs, accMean)
          plt.plot([optC, optC], [np.min(accMean),np.max(accMean)*1.005], '--')

          OptimalC is 0.88587

Out[148]: [<matplotlib.lines.Line2D at 0x11bc6e2b0>]
```

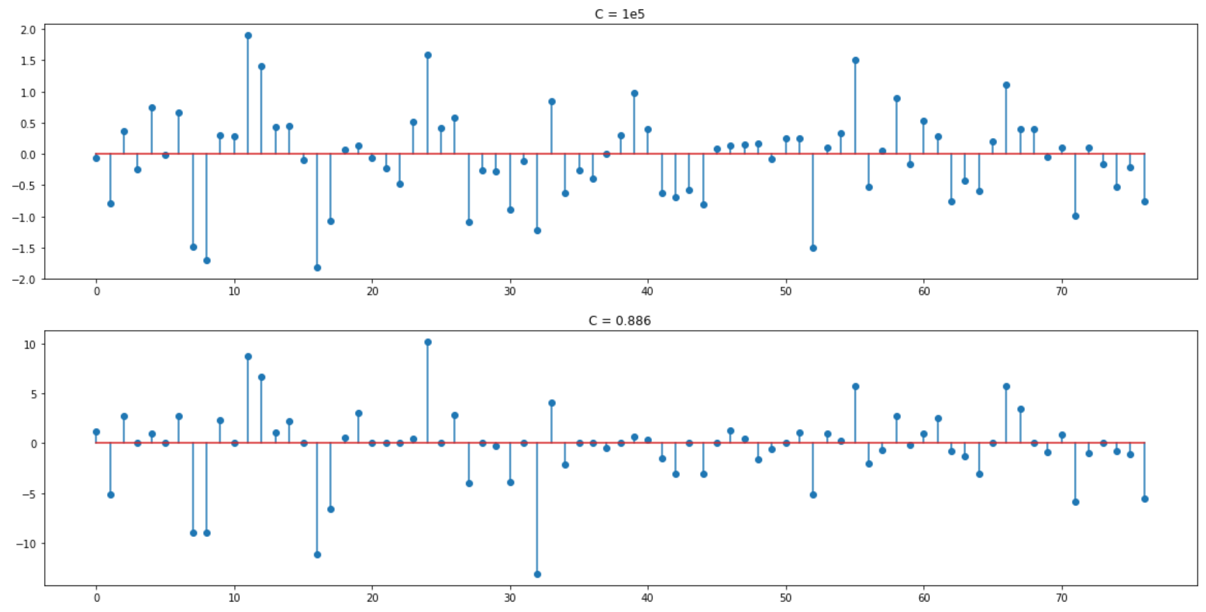


For the optimal `C`, fit the model on the entire training data with l1 regularization. Find the resulting weight matrix, `W_l1`. Plot the first row of this weight matrix and compare it to the first row of the weight matrix without the regularization. You should see that, with l1-regularization, the weight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

```
In [146]: f, axarr = plt.subplots(2, sharex=True, sharey=True, figsize=(20,10))
          axarr[0].stem(W[0,:])
          axarr[0].set_title("C = 1e5")
          axarr[1].stem(Wl1[0,:])
          axarr[1].set_title("C = {:.3f}".format(optC))

Out[146]: <matplotlib.text.Text at 0x11b2c6320>
```



```
In [147]: # TODO
          Wl1 = logreg.coef_

          f, axarr = plt.subplots(2, figsize=(20,10))
          axarr[0].stem(W[0,:])
          axarr[0].set_title("C = 1e5")
          axarr[1].stem(Wl1[0,:])
          axarr[1].set_title("C = {:.3f}".format(optC))

Out[147]: <matplotlib.text.Text at 0x11b95d518>
```



```
In [ ]:
```