ece5300sp21 / **ece5300** Private

ECE 5300 Sp21

4 stars 0 forks

Star Watch

Code Issues Pull requests Actions Projects Wiki Security Insights

master

ece5300 / Demos / **demo05b_breast_cancer.ipynb**

schniter minor updates to lect05

1 contributor

Raw Blame

1365 lines (1365 sloc) | 179 KB

Demo: Breast Cancer Diagnosis via Logistic Regression

In this demo, we will see how to visualize training data for classification, plot the logistic function and perform logistic regression. As an example, we will use the widely-used breast cancer data set. This data set is described here:

https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin (https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin)

Each sample is a collection of features that were manually recorded by a physician upon inspecting a sample of cells from fine needle aspiration. The goal is to detect if the cells are benign or malignant.

We first load the packages as usual.

In [1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import datasets, linear_model, preprocessing
%matplotlib inline
```

Loading and Visualizing the Data

Next, we load the data. It is important to remove the missing values.

In [2]:
```
names = ['id','thick','size_unif','shape_unif','marg','cell_size','bare',
         'chrom','normal','mit','class']
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-database
s/' +
                 'breast-cancer-wisconsin/breast-cancer-wisconsin.data',
                 names=names,na_values='?',header=None)
df = df.dropna()
df.head(6)
```

Out[2]:

	id	thick	size_unif	shape_unif	marg	cell_size	bare	chrom	normal	n
0	1000025	5	1	1	1	2	1.0	3	1	1
1	1002945	5	4	4	5	7	10.0	3	2	1
2	1015425	3	1	1	1	2	2.0	3	1	1
3	1016277	6	8	8	1	3	4.0	3	7	1
4	1017023	4	1	1	3	2	1.0	3	1	1
5	1017122	8	10	10	8	7	10.0	9	7	1

The first attribute, `id`, will not be used for prediction. The last attribute, `class`, is the target variable

and takes on the values {2,4}. We will convert this to {0,1}, which is more standard. Thus, there are 9 features in this dataset.

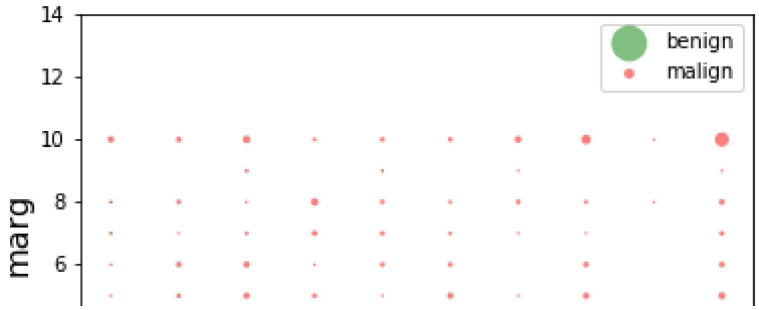
In [3]:
```
# Extract the targets.  Convert to a zero-one indicator
yraw = np.array(df['class'])
BEN_VAL = 2   # value in the 'class' label for benign samples
MAL_VAL = 4   # value in the 'class' label for malignant samples
y = (yraw == MAL_VAL).astype(int)
Iben = (y==0) # indices of benign samples
Imal = (y==1) # indices of malignant samples
```

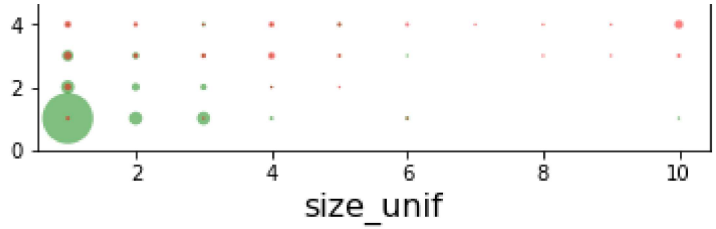
First, let's try using only two of the features, so that we can visualize what is going on.

In [4]:
```
# Choose two features
xnames =['size_unif','marg'] #size_unif, marg
X = np.array(df[xnames])
```

We now plot the target `y0` versus the two extracted features, which we'll call `x1` and `x2`. Since the features each take on values in {1,2,...,10}, we plot over 2D grid of 100 points. At each (`x1`,`x2`), we plot a red circle with a radius proportional to the number of malignant samples, and a green circle with a radius proportional to the number of benign samples. Since we will re-use this code, we define it as a function.

In [5]:
```
def plot_cnt(X,y):

    # Compute the bin edges for the 2d histogram
    x1val = np.array(list(set(X[:,0]))).astype(float)
    x2val = np.array(list(set(X[:,1]))).astype(float)
    x1, x2 = np.meshgrid(x1val,x2val)
    x1e= np.hstack((x1val,np.max(x1val)+1))
    x2e= np.hstack((x2val,np.max(x2val)+1))

    # Make a plot for each class
    yval = list(set(y))
    color = ['g','r']
    for i in range(len(yval)):
        I = np.where(y==yval[i])[0]
        cnt, x1e, x2e = np.histogram2d(X[I,0],X[I,1],[x1e,x2e])
        x1, x2 = np.meshgrid(x1val,x2val)
        plt.scatter(x1.ravel(), x2.ravel(), s=2*cnt.ravel(),alpha=0.5,
                    c=color[i],edgecolors='none')
    plt.ylim([0,14])
    plt.legend(['benign','malign'], loc='upper right')
    plt.xlabel(xnames[0], fontsize=16)
    plt.ylabel(xnames[1], fontsize=16)
    return plt

plot_cnt(X,y);
```





The above plot gives some intuition about how these two features could be used to classify whether a given sample was bening or malignant. Roughly speaking, we might predict that the sample is benign if both `marg` and `size_unif` are sufficiently small.

Least-Squares Linear Regression

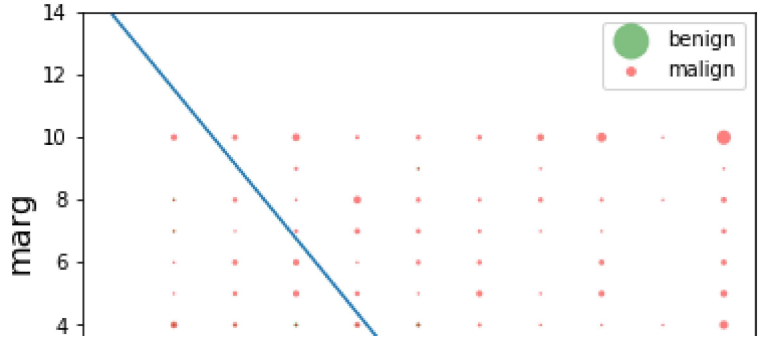
As a first attempt in designing a classifier, let's apply the least-squares (LS) linear regression method that we studied in earlier units. That is, let's treat the target `y` as a real number and try to predict it from `x0` and `x1`.

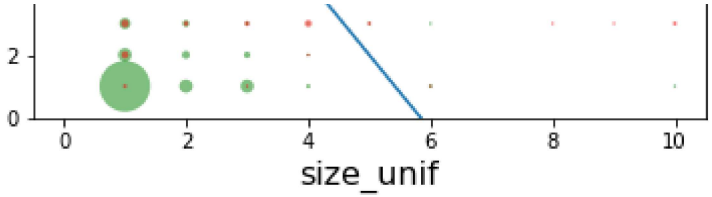
In [6]:
```
xnames =['size_unif','marg']
X = np.array(df[xnames])
from sklearn import datasets, linear_model
regr = linear_model.LinearRegression()
regr.fit(X, y)
print('regr.intercept=',regr.intercept_)
print('regr.coef=',regr.coef_)
```

```
regr.intercept= -0.08420066620061456
regr.coef= [0.10007944 0.04197521]
```

Because our target `y` takes on the value 0 for benign and 1 for malignant, we could assign a predicted value of less than 0.5 as benign and a predicted value of more than 0.5 as malignant. This rule partitions the (x1,x2) space linearly into two halves, as illustrated below. The linear separation occurs because $b + w_1 x_2 + w_2 x_2 < 0.5$ can be rewritten as $x_2 < (0.5 - b) / w_2 + (-w_1 / w_2) x_1$, which shows that the decision boundary is a line.

In [7]:
```
b=regr.intercept_
w1=regr.coef_[0]
w2=regr.coef_[1]

x1=np.zeros([11,1])
x2=np.zeros([11,1])
for i in range(11):
    x1[i]=i;
    x2[i]=(0.5-b-w1*i)/w2

plt = plot_cnt(X,y)
plt.plot(x1, x2);
```





Let's assess the accuracy of the 2-feature linear prediction as the percentage of correct classifications on the training data:

In [8]:
```
yhat = regr.predict(X)
yhati = (yhat >=0.5).astype(int)
acc = np.mean(yhati == y)
print("Accuracy on training data using two features = %f" % acc)
```

```
Accuracy on training data using two features = 0.922401
```

And now let's repeat the linear-prediction procedure using all 9 features. We first standardize the `X` values so that the slope values are interpretable.

In [9]:
```
xnames = ['thick','size_unif','shape_unif','marg','cell_size','bare',
          'chrom','normal','mit']
X = np.array(df[xnames])

regr.fit(X,y)
yhat=regr.predict(X)

yhati= (yhat >=0.5).astype(int)
acc = np.mean(yhati == y)
print("Accuracy on training data using 9 features = %f" % acc)
```

```
Accuracy on training data using 9 features = 0.960469
```

Logistic Regression

In logistic regression, we must fit the weights b, w_1, w_2, ..., w_d in the model

$$P(y = 1|x) = f(z), \quad z = b + w_1 x_1 + \cdots + w_d x_d,$$

The `sklearn` module provides a `LogisticRegression` object that does this.

In fact, this object includes L2 regularization by default, with parameter `C` controlling the inverse regularization strength. So, let's set `C` large to suppress the effects of L2 regularization.

In [10]:
```
logreg = linear_model.LogisticRegression(C=1e5)
```

For visualization, let's first try logisitic regression with only two features. We would normally standardize the features `X` since we are using regularization, but we will avoid doing so in order to plot the decision boundary in the raw-feature space (to compare to the linear-regression case).

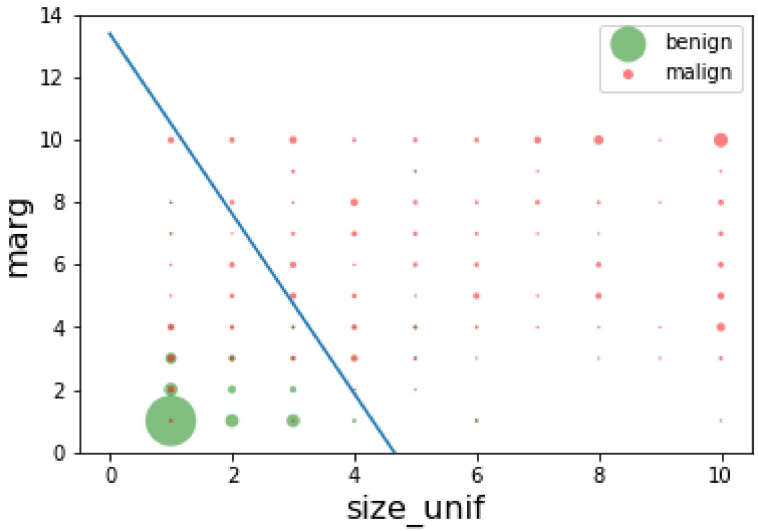
In [11]:
```
xnames =['size_unif','marg']
X = np.array(df[xnames])
logreg.fit(X, y)
yhat = logreg.predict(X)
acc = np.mean(yhat == y)
print("Accuracy on training data = %f" % acc)
```

```
Accuracy on training data = 0.941435
```

Accuracy on training data = 0.941435

We see that the accuracy of logistic regression is higher than with linear prediction, as used earlier. Next, let's plot the decision boundary for the logistic model

In [12]:
```
b = logreg.intercept_
w = np.array(logreg.coef_).T
w1 = w[0]
w2 = w[1]
x1=np.zeros([11,1])
x2=np.zeros([11,1])
for i in range(11):
    x1[i]=i;
    x2[i]=(0.5-b-w1*i)/w2

plt = plot_cnt(X,y)
plt.plot(x1, x2);
```



Relative to linear regression, the decision boundary has moved southwest and appears to better match the data.

Now let's try logistic regression with all 9 features. We first standardize the `X` values because an L2 penalty is used by default.

In [13]:
```
xnames = ['thick','size_unif','shape_unif','marg','cell_size','bare',
          'chrom','normal','mit']
X = np.array(df[xnames])
Xs = preprocessing.scale(X)
logreg.fit(Xs, y)
```

Out[13]:
```
LogisticRegression(C=100000.0)
```

We can next plot the accuracy on the training data. We see we get an accuracy better than with linear regression, as used earlier.

In [14]:
```
yhat = logreg.predict(Xs)
acc = np.mean(yhat == y)
print("Accuracy on training data = %f" % acc)
```

```
Accuracy on training data = 0.969253
```

It is also useful to print the weights for each feature. We can use the `pandas` package to make a

It is also useful to print the weights for each feature. We can use the `pandas` package to make a table.

In [15]:
```
W=logreg.coef_
data = {'feature': xnames, 'slope': np.squeeze(W)}
dfslope = pd.DataFrame(data=data)
dfslope
```

Out[15]:

	feature	slope
0	thick	1.508053
1	size_unif	-0.019233
2	shape_unif	0.963721
3	marg	0.946439
4	cell_size	0.214663
5	bare	1.394654
6	chrom	1.094675
7	normal	0.649846
8	mit	0.926017

Cross-Validation

The above code measured the classification performance on the training data. However, we should really measure performance on test data.

So, we now perform 10-fold cross-validation and print average precision, recall, f1-score, and accuracy.

In [16]:
```
from sklearn.model_selection import KFold
from sklearn.metrics import precision_recall_fscore_support

nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
prec = []
rec = []
f1 = []
acc = []
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='bina
ry')
    prec.append(preci)
    rec.append(reci)
```

```
    f1.append(f1i)
    acci = np.mean(yhat == yts)
    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,ddof=1)/np.sqrt(nfold)
rec_se = np.std(rec,ddof=1)/np.sqrt(nfold)
f1_se = np.std(f1,ddof=1)/np.sqrt(nfold)
acc_se = np.std(acc,ddof=1)/np.sqrt(nfold)

print('Precision = {0:.4f}, SE={1:.4f}'.format(precm,prec_se))
print('Recall =    {0:.4f}, SE={1:.4f}'.format(recm, rec_se))
print('f1 =        {0:.4f}, SE={1:.4f}'.format(f1m, f1_se))
print('Accuracy =  {0:.4f}, SE={1:.4f}'.format(accm, acc_se))
```

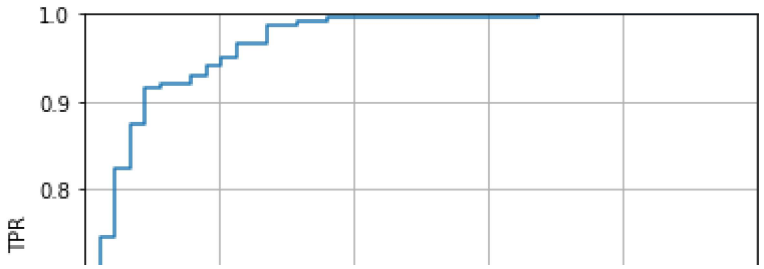
```
Precision = 0.9556, SE=0.0139
Recall =    0.9470, SE=0.0104
f1 =        0.9502, SE=0.0058
Accuracy =  0.9648, SE=0.0045
```

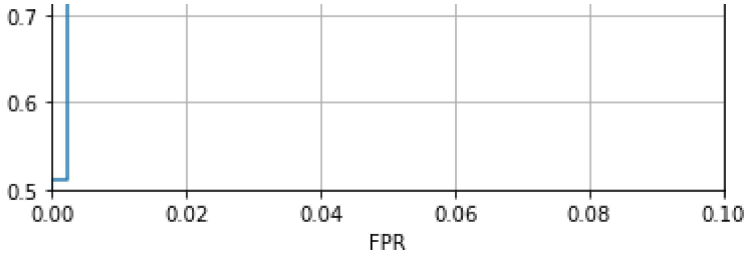
ROC curve

As discussed above, the logistic classifier outputs a *soft* classification $P(y=1|x)$. A simple idea is to select the class label $\hat{y}=1$ whenever $P(y=1|x)>0.5$. However, one could also set $\hat{y}=1$ whenever $P(y=1|x)>t$ for some other threshold t. Using a higher threshold would select $\hat{y}=1$ less often, which would result in fewer *false alarms* but more *missed detections* (i.e., reduced *sensitivity*). Likewise, a lower threshold value would select $\hat{y}=1$ more often, which would result in fewer *missed detections* (i.e., increased *sensitivity*) but more *false alarms*. The ROC curve helps to visualize this tradeoff by graphing the *true positive rate* vs. the *false positive rate* as the threshold t is varied.

The ROC curve can be plotted using the `sklearn` package as follows.

In [17]:
```
from sklearn import metrics
yprob = logreg.predict_proba(Xs)
fpr, tpr, thresholds = metrics.roc_curve(y,yprob[:,1])
thresholds[0] = 1

plt.plot(fpr,tpr)
plt.grid()
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.ylim([0.5,1])
plt.xlim([0,0.1]);
```

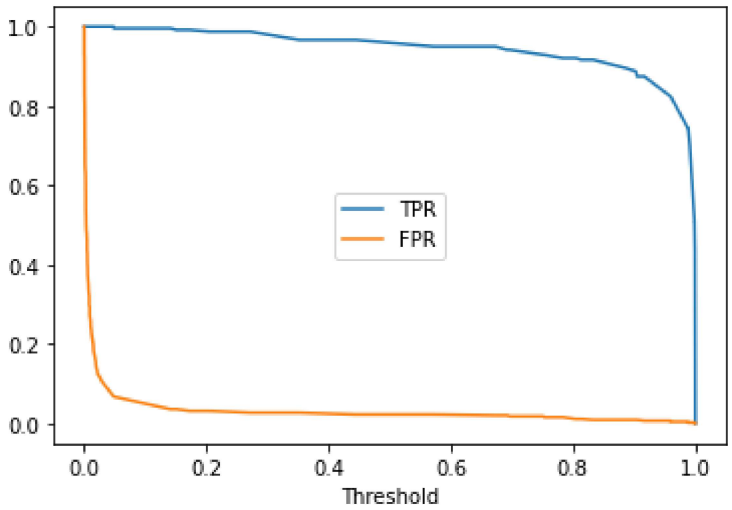




Now let's see how the TPR and FPR vary as a function of the threshold.

In [18]:
```
plt.plot(thresholds,tpr, thresholds,fpr)
plt.legend(['TPR','FPR'])
plt.xlabel('Threshold')
```

Out[18]: Text(0.5, 0, 'Threshold')



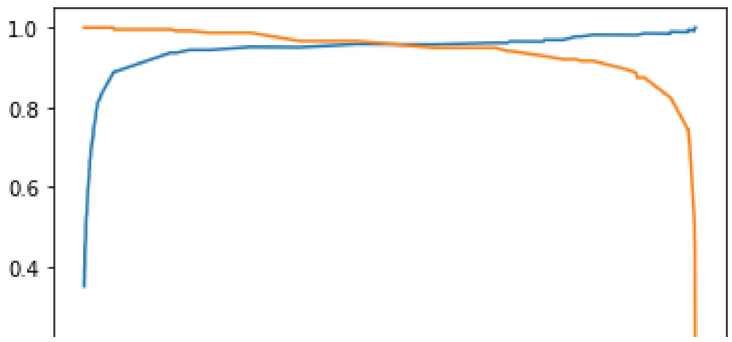
Similarly, we can see how the precision and recall vary as a function of the threshold.

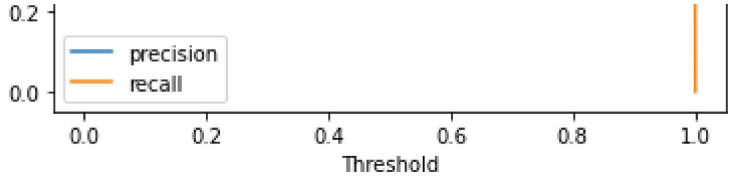
In [19]:
```
py1 = np.mean(y==1)
py0 = np.mean(y==0)
pyhat1 = fpr*py0 + tpr*py1
precision = tpr*py1/pyhat1
recall = tpr
```

```
<ipython-input-19-bdef480f1716>:4: RuntimeWarning: invalid value encountered
in true_divide
  precision = tpr*py1/pyhat1
```

In [20]:
```
plt.plot(thresholds,precision, thresholds,recall)
plt.legend(['precision','recall'])
plt.xlabel('Threshold')
```

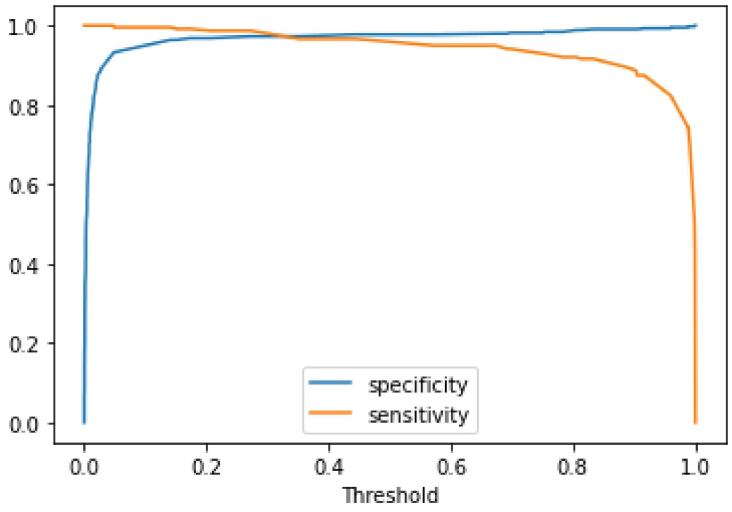
Out[20]: Text(0.5, 0, 'Threshold')





In [21]:
```
sensitivity = tpr
specificity = 1-fpr
plt.plot(thresholds,specificity, thresholds,sensitivity)
plt.legend(['specificity','sensitivity'])
plt.xlabel('Threshold')
```

Out[21]: Text(0.5, 0, 'Threshold')



And we can see how sensitivity and specificity vary as a function of the threshold.

To measure the goodness-of-classification in a threshold-*indepedent* manner, we can use the *area under the curve* (AUC). A higher AUC means that, for a given FPR, the PPR is higher. In practice, one should evalute UAC over different cross validation folds and report the mean AUC.

In [22]:
```
auc=metrics.roc_auc_score(y,yprob[:,1])
print("AUC=%f" % auc)
```

```
AUC=0.996315
```

L1 regularization

As with linear regression, we can add an L1 penalty in logistic regression to encourage fewer non-zero weight coefficients. This is particularly important when there are many features and not enough training samples. (In the current dataset, we have a sufficient number of samples for the number of features, and so we do not expect to see too much benefit from L1 regression.) In the classification context, the term "LASSO" is often used to mean L1-regularized logistic regression, even though it was originally proposed as L1-regularized linear regression.

The `LogisticRegression` method in sklearn uses L2 regularization by default, but it allows the user to select L1 regularization instead. In either case, the user controls the *inverse* regularization strength using the parameter C. So, as C gets smaller with L1 regularization, there should be fewer non-zero weights. To properly choose C, one should use cross validation.

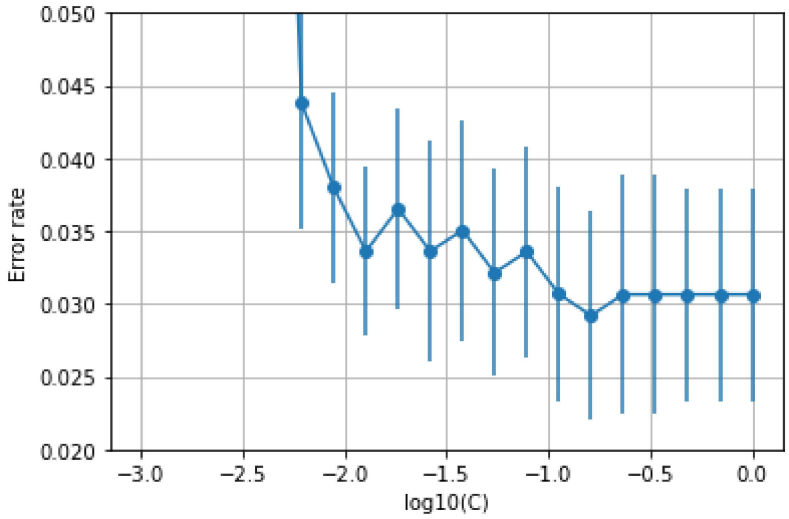
Below, we use 10-fold cross-validation to determine the value of C that yields the minimum error-

rate (i.e., 1-accuracy). We also count the number of non-zero coefficients.

In [23]:
```
npen = 20
C_test = np.logspace(-3,0,npen)

# Create the cross-validation object and error rate matrix
nfold = 10
kf = KFold(n_splits=nfold,shuffle=True)
err_rate = np.zeros((npen,nfold))
num_nonzerocoef = np.zeros((npen,nfold))
# Create the logistic regression object
logreg = linear_model.LogisticRegression(penalty='l1', solver='liblinear', w
arm_start=True)

# Loop over the folds in the cross-validation
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_)>0.001)
    print("Fold %d" % ifold)

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,ddof=1)/np.sqrt(nfold)
plt.errorbar(np.log10(C_test), err_mean, marker='o',yerr=err_se)
plt.ylim([0.02,0.05])
plt.grid()
plt.xlabel('log10(C)')
plt.ylabel('Error rate')

imin = np.argmin(err_mean)

print("The minimum test error rate = %12.4e, SE=%12.4e" % (err_mean[imin],
err_se[imin]))
print("The C value corresponding to minimum error = %12.4e" % (C_test[imin
]))
```

```
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4
```

```
Fold 5
Fold 6
Fold 7
Fold 8
Fold 9
The minimum test error rate =   2.9199e-02, SE=  7.1850e-03
The C value corresponding to minimum error =   1.6238e-01
```



Now let us choose C using one standard error rule. Note that because smaller C implies a simpler model, we want to find the smallest C that satisfies the error target. We also find the corresponding test accuracy with this choice of C.

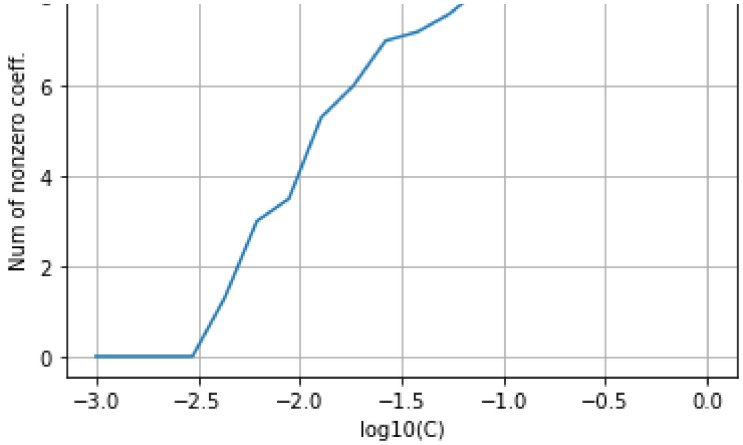
In [24]:
```
err_tgt = err_mean[imin] + err_se[imin]
iopt = np.where(err_mean < err_tgt)[0][0]
C_opt = C_test[iopt]

print("One-standard-error-rule C=%12.4e" % C_opt)
print("The test error rate = %12.4e, SE=%12.4e" % (err_mean[iopt], err_se[
iopt]))

print('Accuracy =  {0:.4f}, SE={1:.4f}'.format(1-err_mean[iopt], err_se[io
pt]))
```

```
One-standard-error-rule C=  1.2743e-02
The test error rate =   3.3610e-02, SE=  5.7556e-03
Accuracy =  0.9664, SE=0.0058
```

Now we plot the number of non-zero coefficients versus the value of C. And we print the number of non-zero coefficients corresponding to C_opt (averaged over the folds).

In [25]:
```
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_non
zerocoef_mean[iopt])
```

```
The number of non-zero coefficients for the optimal C = 5.300000
```



For the optimal C, we now fit the model on the entire training data with L1 regularization and plot the resulting weights, `W_L1`. We can compare these weights to the weights fit without regularization. We can see that, with L1-regularization, the weight vector is more sparse.

In [26]:
```
logreg = linear_model.LogisticRegression(C=C_opt,penalty='l1',solver='liblinear')
logreg.C= C_opt
logreg.fit(Xs,y)
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')
```

Out[26]: `Text(0.5, 1.0, 'L1-regularization')`