Homework 2: CSE 258

By Amogha Sekhar, A53301791

Basic Preprocessing

In [1]:

```
f = open("5year.arff", 'r')
```

In [2]:

```
while not '@data' in f.readline():
    pass
```

In [3]:

```
dataset = []
for l in f:
    if '?' in l: # Missing entry
        continue
    l = l.split(',')
    values = [1] + [float(x) for x in l]
    values[-1] = values[-1] > 0 # Convert to bool
    dataset.append(values)
```

In [195]:

```
len(dataset[0])
```

Out[195]:

```
66
```

Question 1: Download and parse the bankruptcy data. We'll use the 5year.arff file. Code to read the data is available in the stub. Train a logistic regressor (e.g. sklearn.linear model.LogisticRegression) with regularization coefficient C = 1.0. Report the accuracy and Balanced Error Rate (BER) of your classifier (1 mark).

In [5]:

```
X = [d[:-1] for d in dataset]
y = [d[-1] for d in dataset]
```

In [6]:

```
from sklearn.linear_model import LogisticRegression
from sklearn import metrics

mod = LogisticRegression(C=1.0)
mod.fit(X,y)
```

Out[6]:

```
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=0.0
001,
          verbose=0, warm_start=False)
```

In [7]:

```
pred = mod.predict(X)
```

In [8]:

```
import numpy
TP_ = numpy.logical_and(pred, y)
FP_ = numpy.logical_and(pred, numpy.logical_not(y))
TN_ = numpy.logical_and(numpy.logical_not(pred), numpy.logical_not(y))
FN_ = numpy.logical_and(numpy.logical_not(pred), y)
```

In [9]:

```
TP = sum(TP_)
FP = sum(FP_)
TN = sum(TN_)
FN = sum(FN_)
```

In [10]:

```
#Accuracy
(TP + TN) / (TP + FP + TN + FN)
```

Out[10]:

```
0.9663477400197954
```

In [11]:

```
# BER
1 - 0.5*(TP / (TP + FN) + TN / (TN + FP))
```

Out[11]:

```
0.48107498376612512
```

Answer for Question 1

The Accuracy of the model is 96.63477400197954%.

The Balanced Error Rate(BER) of the model is 48.107498376612512%.

Question 3: Shuffle the data, and split it into training, validation, and test splits, with a 50/25/25% ratio. Using the class weight='balanced' option, and training on the training set, report the training/validation/test accuracy and BER (1 mark).

In [37]:

```
Xy = list(zip(X,y))
```

In [38]:

```
import random
random.shuffle(Xy)
```

In [39]:

```
X = [d[0] for d in Xy]
y = [d[1] for d in Xy]
```

In [40]:

```
N = len(y)
N
```

Out[40]:

```
3031
```

In [41]:

```
Ntrain= 1515
Nvalid= 758
Ntest= 758
```

In [42]:

```
Xtrain = X[:Ntrain]
Xvalid = X[Ntrain:Ntrain+Nvalid]
Xtest = X[Ntrain+Nvalid:]
```

In [43]:

```
ytrain = y[:Ntrain]
yvalid = y[Ntrain:Ntrain+Nvalid]
ytest = y[Ntrain+Nvalid:]
```

In [44]:

```
mod = LogisticRegression(C=1.0, class_weight = 'balanced')
```

In [45]:

```
mod.fit(Xtrain,ytrain)
```

Out[45]:

```
LogisticRegression(C=1.0, class_weight='balanced', dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=Non
e,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

In [46]:

```
pred_train= mod.predict(Xtrain)
pred_valid= mod.predict(Xvalid)
pred_test= mod.predict(Xtest)
```

In [47]:

```
#Accuracy and BER for training set

TP__train = numpy.logical_and(pred_train, ytrain)
FP__train = numpy.logical_and(pred_train, numpy.logical_not(ytrain))
TN__train= numpy.logical_and(numpy.logical_not(pred_train), numpy.logical_not(ytrain
FN__train = numpy.logical_and(numpy.logical_not(pred_train), ytrain)

TP_train = sum(TP__train)
FP_train = sum(FP__train)
TN_train = sum(TN__train)
FN_train = sum(FN__train)
```

In [48]:

```
#Accuracy for training set

(TP_train + TN_train) / (TP_train + FP_train + TN_train + FN_train)
```

Out[48]:

```
0.76303630363036301
```

In [49]:

```
#BER for training set

1 - 0.5*(TP_train / (TP_train + FN_train) + TN_train / (TN_train + FP_train))
```

Out[49]:

```
0.21203133318123046
```

In [50]:

```
#Accuracy and BER for validation set

TP__valid = numpy.logical_and(pred_valid, yvalid)
FP__valid = numpy.logical_and(pred_valid, numpy.logical_not(yvalid))
TN__valid = numpy.logical_and(numpy.logical_not(pred_valid), numpy.logical_not(yval:
FN__valid = numpy.logical_and(numpy.logical_not(pred_valid), yvalid)

TP_valid = sum(TP__valid)
FP_valid = sum(FP__valid)
TN_valid = sum(TN__valid)
FN_valid = sum(FN__valid)
```

In [51]:

```
#Accuracy for validation set

(TP_valid + TN_valid) / (TP_valid + FP_valid + TN_valid + FN_valid)
```

Out[51]:

0.76912928759894461

In [52]:

```
#BER for validation set

1 - 0.5*(TP_valid / (TP_valid + FN_valid) + TN_valid / (TN_valid + FP_valid))
```

Out[52]:

0.21574697173620461

In [53]:

```
#Accuracy and BER for test set

TP__test = numpy.logical_and(pred_test, ytest)
FP__test = numpy.logical_and(pred_test, numpy.logical_not(ytest))
TN__test = numpy.logical_and(numpy.logical_not(pred_test), numpy.logical_not(ytest)
FN__test = numpy.logical_and(numpy.logical_not(pred_test), ytest)

TP_test = sum(TP__test)
FP_test = sum(FP__test)
TN_test = sum(TN__test)
FN_test = sum(FN__test)
```

In [54]:

```
#Accuracy for test set

(TP_test + TN_test) / (TP_test + FP_test + TN_test + FN_test)
```

Out[54]:

0.79287598944591031

In [55]:

```
#BER for test set

1 - 0.5*(TP_test / (TP_test + FN_test) + TN_test / (TN_test + FP_test))
```

Out[55]:

```
0.1371995820271682
```

In [56]:

```
pred= mod.predict(X)
```

In [57]:

```
TP_ = numpy.logical_and(pred, y)
FP_ = numpy.logical_and(pred, numpy.logical_not(y))
TN_ = numpy.logical_and(numpy.logical_not(pred), numpy.logical_not(y))
FN_ = numpy.logical_and(numpy.logical_not(pred), y)
```

In [58]:

```
TP = sum(TP_)
FP = sum(FP_)
TN = sum(TN_)
FN = sum(FN_)
```

In [59]:

```
#Accuracy
(TP + TN) / (TP + FP + TN + FN)
```

Out[59]:

```
0.7720224348398678
```

In [60]:

```
#BER
1 - 0.5*(TP / (TP + FN) + TN / (TN + FP))
```

Out[60]:

```
0.18892715843592467
```

Answer for Question 3:

Set	Accuracy	Balanced Error Rate
Training	76.303630363036301%	21.203133318123046%
Validation	76.912928759894461%	21.574697173620461%
Test	79.287598944591031%	13.71995820271682%
Entire dataset	77.202243483998678%	18.892715843592467%

Question 4: Implement a complete regularization pipeline with the balanced classifier. Consider values of C in the range {10–4, 10–3, . . . , 103, 104}. Report

(or plot) the train, validation, and test BER for each value of C. Based on these values, which classifier would you select (in terms of generalization performance) and why (1 mark)?

In [72]:

```
values= [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000]
train_BER= []
test_BER= []
valid_BER= []

for x in values:
    mod = LogisticRegression(C= x, class_weight = 'balanced')
    mod.fit(Xtrain,ytrain)
    pred_train= mod.predict(Xtrain)
    pred_valid= mod.predict(Xvalid)
    pred_test= mod.predict(Xtest)

    #For training set
    TP__train = numpy.logical_and(pred_train, ytrain)
    FP__train = numpy.logical_and(pred_train, numpy.logical_not(ytrain))
    TN__train= numpy.logical_and(numpy.logical_not(pred_train), numpy.logical_not(yt
    FN__train = numpy.logical_and(numpy.logical_not(pred_train), ytrain)

    TP_train = sum(TP__train)
    FP_train = sum(FP__train)
    TN_train = sum(TN__train)
    FN_train = sum(FN__train)

    tr= 1 - 0.5*(TP_train / (TP_train + FN_train) + TN_train / (TN_train + FP_train)
    train_BER.append(tr)

    #For validation set
    TP__valid = numpy.logical_and(pred_valid, yvalid)
    FP__valid = numpy.logical_and(pred_valid, numpy.logical_not(yvalid))
    TN__valid = numpy.logical_and(numpy.logical_not(pred_valid), numpy.logical_not(y
    FN__valid = numpy.logical_and(numpy.logical_not(pred_valid), yvalid)

    TP_valid = sum(TP__valid)
    FP_valid = sum(FP__valid)
    TN_valid = sum(TN__valid)
    FN_valid = sum(FN__valid)

    v= 1 - 0.5*(TP_valid / (TP_valid + FN_valid) + TN_valid / (TN_valid + FP_valid))
    valid_BER.append(v)

    #For test set

    TP__test = numpy.logical_and(pred_test, ytest)
    FP__test = numpy.logical_and(pred_test, numpy.logical_not(ytest))
    TN__test = numpy.logical_and(numpy.logical_not(pred_test), numpy.logical_not(yte
    FN__test = numpy.logical_and(numpy.logical_not(pred_test), ytest)

    TP_test = sum(TP__test)
    FP_test = sum(FP__test)
    TN_test = sum(TN__test)
    FN_test = sum(FN__test)

    tst= 1 - 0.5*(TP_test / (TP_test + FN_test) + TN_test / (TN_test + FP_test))
    test_BER.append(tst)


print(train_BER)
print(valid_BER)
print(test_BER)
```

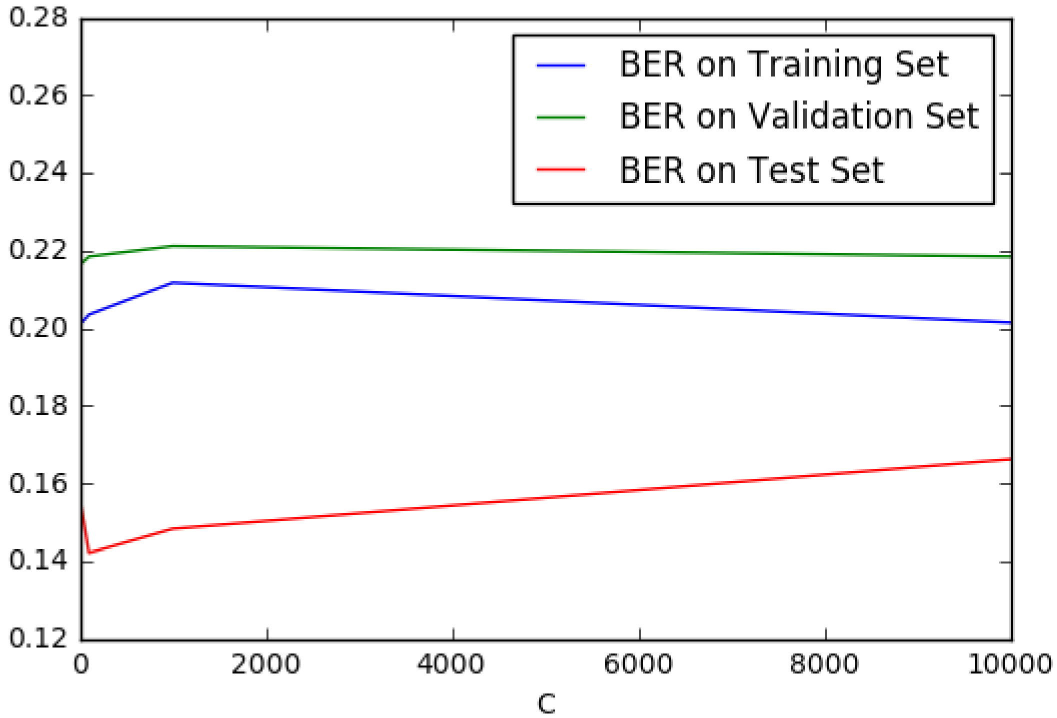
```
[0.2774735721347632, 0.18872157578523074, 0.20242984257357977, 0.20174
537987679675, 0.21203133318123046, 0.20106091718001373, 0.203456536618
75418, 0.21168910183283907, 0.20140314852840513]
[0.24333781965006729, 0.21776581426648711, 0.21776581426648711, 0.2184
3876177658139, 0.21574697173620461, 0.21641991924629878, 0.21843876177
658139, 0.22113055181695818, 0.21843876177658139]
[0.1751306165099269, 0.1902194357366771, 0.15304075235109726, 0.179205
85161964464, 0.1371995820271682, 0.15510971786833849, 0.14202716823406
48, 0.14821316614420066, 0.16612330198537095]
```

In [73]:

```python
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(values, train_BER)
plt.plot(values, valid_BER)
plt.plot(values, test_BER)
plt.xlim(0, 10000)
plt.legend(['BER on Training Set','BER on Validation Set', 'BER on Test Set'], loc=(
plt.xlabel('C')
```

Out[73]:

<matplotlib.text.Text at 0x11a808d30>



Answer for Question 4

In terms of generalization performance, the best C will be that value of C which has the lowest BER on the validation set. The test set only serves for reporting purpose (on how good our strategy on selecting the model, how it performs on unseen data, etc.) so we can't use it when we decide the model.

The lowest BER is 0.21574697173620461 for C= 1.

C	Train_BER	Valid_BER	Test_BER
0.0001	0.2774735721347632	0.24333781965006729	0.1751306165099269
0.001	0.18872157578523074	0.21776581426648711	0.1902194357366771
0.01	0.20242984257357977	0.21776581426648711	0.15304075235109726
0.1	0.20174537987679675	0.21843876177658139	0.17920585161964464
1	0.21203133318123046	0.21574697173620461	0.1371995820271682
10	0.20106091718001373	0.21641991924629878	0.15510971786833849
100	0.20345653661875418	0.21843876177658139	0.1420271682340648
1000	0.21168910183283907	0.22113055181695818	0.14821316614420066
10000	0.20140314852840513	0.21843876177658139	0.16612330198537095

Question 6: (CSE258 only) The sample weight option allows you to manually build a balanced (or imbalanced) classifier by assigning different weights to each datapoint (i.e., each label y in the training set). For example, we would assign equal weight to all samples by fitting: weights = [1.0] * len(ytrain) mod = linear_model.LogisticRegression(C=1, solver='lbfgs') mod.fit(Xtrain, ytrain, sample_weight=weights) (note that you should use the lbfgs solver option, and need not set class weight='balanced' in this case). Assigning larger weights to (e.g.) positive samples would encourage the logistic regressor to optimize for the True Positive Rate. Using the above code, compute the Fβ score (on the test set) of your (unweighted) classifier, for β = 1 and β = 10. Following this, identify weight vectors that yield better performance (compared to the unweighted vector) in terms of the F1 and F10 scores (2 marks).

In [108]:

```
from sklearn import metrics
values= [1.0, 10.0]
fb_score=[]

for b in values:
    weights = [1] * len(ytrain)
    mod = LogisticRegression(C=1, solver= "lbfgs")
    mod.fit(Xtrain, ytrain, sample_weight=weights)
    pred_test= mod.predict(Xtest)
    fb_score.append(metrics.fbeta_score(ytest, pred_test, beta= b))

print(fb_score)
```

```
[0.16666666666666669, 0.091734786557674836]
```

In [109]:

```
ytrain
```

```
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
```

In [110]:

```
count_positive= 0
count_negative= 0
for i in range(len(ytrain)):
    if ytrain[i]== True:
        count_positive+= 1
    else:
        count_negative+= 1
print(count_positive)
print(count_negative)
```

```
54
1461
```

In [189]:

```
weights= []

for i in range(len(ytrain)):
    if ytrain[i]== True:
        weights.append(8)
    else:
        weights.append(1)


mod = LogisticRegression(C=1, solver= "lbfgs")
mod.fit(Xtrain, ytrain, sample_weight=weights)
pred_test= mod.predict(Xtest)
fb_score= metrics.fbeta_score(ytest, pred_test, beta= 1)

print(fb_score)
```

```
0.313725490196
```

In [211]:

```
weights= []

for i in range(len(ytrain)):
    if ytrain[i]== True:
        weights.append(1000)
    else:
        weights.append(1)


mod = LogisticRegression(C=1, solver= "lbfgs")
mod.fit(Xtrain, ytrain, sample_weight=weights)
pred_test= mod.predict(Xtest)
fb_score= metrics.fbeta_score(ytest, pred_test, beta= 10)

print(fb_score)
```

```
0.833875406555
```

Answer for Question 6:

The f-beta score on the unweighted classifier for beta= 1 is 0.16666666666666669. The f-beta score on the unweighted classifier for beta= 10 is 0.091734786557674836.

We know that, assigning larger weights to positive samples would encourage the logistic regressor to optimize to the True Positive Rate. Following this logic,

for beta=1: assigning weight of 8 for the positive examples, and 1 for the negative examples, we get f-beta score of 0.313725490196 which is better than that for the unweighted classifier (0.16666666666666669)

for beta=10: assigning weight of 1000 for the positive examples, and 1 for the negative examples, we get f-beta score of 0.833875406555 which is better than that for the unweighted classifier (0.091734786557674836)

Question 7: Following the stub code, compute the PCA basis on the training set. Report the first PCA component (i.e., pca.components [0]) (1 mark).

In [196]:

```
from sklearn.decomposition import PCA

print(len(dataset[0])) #to find out number of components

pca = PCA(n_components=65)
pca.fit(Xtrain)
print(pca.components_[0])
```

```
66
[  3.59696951e-18   3.52972178e-08  -3.48754783e-07  -1.08197148e-06
  -4.83641153e-06  -2.31314913e-03   5.14740627e-07  -1.56308498e-06
  -4.88675683e-06   8.34432964e-07   1.60599751e-07  -2.18245754e-07
  -9.62923023e-07   5.64962886e-06  -1.56270810e-06  -5.17499401e-03
  -8.48169067e-07  -5.34032476e-06  -1.42418580e-06  -3.06995711e-07
  -4.21458395e-05   1.38252766e-05  -1.95742060e-07  -2.62617608e-07
  -9.64572692e-07  -6.34467301e-07  -7.68680897e-07   3.17341546e-05
  -2.79259317e-05  -4.04360878e-06   1.41584907e-06  -2.92388744e-07
   2.80150264e-04  -4.04649778e-06   2.43894161e-06  -1.63276908e-07
   5.59931636e-07  -6.19268719e-03   5.54495449e-07  -1.77014323e-07
  -2.05706327e-06   2.46069214e-06  -1.09321001e-07  -9.13233369e-05
  -4.91752089e-05   1.87498983e-06  -3.60175192e-06  -6.27252940e-05
  -2.34902109e-07  -2.70643512e-07  -4.30735106e-06   6.85543295e-07
   7.50919665e-07   5.73274953e-06  -2.78127038e-05  -9.99964658e-01
  -2.13350088e-07   5.49764986e-07   3.05677306e-07  -4.56649501e-07
   1.78425420e-04   1.34440443e-05   2.69309744e-04  -5.13171122e-06
  -3.64571765e-07]
```

Answer for Question 7:

The first PCA component is shown above.

Question 8: Next we'll train a model using a low-dimensional feature vector. By representing the data in the above basis, i.e.: Xpca*train = numpy.matmul(Xtrain, pca.components*.T) Xpca*valid = numpy.matmul(Xvalid, pca.components*.T) Xpca*test = numpy.matmul(Xtest, pca.components*.T) compute the validation and test BER of a model that uses just the first N components (i.e., dimensions) for N = 5, 10, . . . , 25, 30. Again use class weight='balanced' and C = 1.0 (2 marks).

In [174]:

```
N= [5, 10, 15, 20, 25, 30]
valid_BER= []
test_BER= []

for i in N:
    pca = PCA(n_components= i)
    pca.fit(Xtrain)
    #print(pca.components_)
    Xpca_train = numpy.matmul(Xtrain, pca.components_.T)
    Xpca_valid = numpy.matmul(Xvalid, pca.components_.T)
    Xpca_test = numpy.matmul(Xtest, pca.components_.T)

    mod = LogisticRegression(C=1.0, class_weight = 'balanced')
    mod.fit(Xpca_train,ytrain)
    pred_valid= mod.predict(Xpca_valid)
    pred_test= mod.predict(Xpca_test)

    #For validation BER
    TP__valid = numpy.logical_and(pred_valid, yvalid)
    FP__valid = numpy.logical_and(pred_valid, numpy.logical_not(yvalid))
    TN__valid = numpy.logical_and(numpy.logical_not(pred_valid), numpy.logical_not(y
    FN__valid = numpy.logical_and(numpy.logical_not(pred_valid), yvalid)

    TP_valid = sum(TP__valid)
    FP_valid = sum(FP__valid)
    TN_valid = sum(TN__valid)
    FN_valid = sum(FN__valid)

    v= 1 - 0.5*(TP_valid / (TP_valid + FN_valid) + TN_valid / (TN_valid + FP_valid)
    valid_BER.append(v)

    #For test BER
    TP__test = numpy.logical_and(pred_test, ytest)
    FP__test = numpy.logical_and(pred_test, numpy.logical_not(ytest))
    TN__test = numpy.logical_and(numpy.logical_not(pred_test), numpy.logical_not(yte
    FN__test = numpy.logical_and(numpy.logical_not(pred_test), ytest)

    TP_test = sum(TP__test)
    FP_test = sum(FP__test)
    TN_test = sum(TN__test)
    FN_test = sum(FN__test)

    tst= 1 - 0.5*(TP_test / (TP_test + FN_test) + TN_test / (TN_test + FP_test))
    test_BER.append(tst)

print(valid_BER)
print(test_BER)
```

```
[0.36379542395693132, 0.31269627635711084, 0.30026917900403771, 0.2712
8757290264693, 0.25948855989232844, 0.23660834454912516]
[0.36539184952978054, 0.24881922675026125, 0.26873563218390806, 0.1875
026123301986, 0.19028213166144203, 0.1833646812957157]
```

Answer for Question 8:

N	Validation Set BER	Test Set BER
5	0.36379542395693132	0.36539184952978054

N	Validation Set BER	Test Set BER
10	0.31269627635711084	0.24881922675026125
15	0.30026917900403771	0.26873563218390806
20	0.27128757290264693	0.1875026123301986
25	0.25948855989232844	0.19028213166144203
30	0.23660834454912516	0.1833646812957157

This is true as N increases, the Balanced Error Rate on the Validation and Test Set decreases as we now have more information to make a more accurate prediction.

In [ ]:

```

```