258_HW2

October 29, 2019

0.1 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).

[1]:
```python
import numpy
import urllib
import scipy.optimize
import random
from sklearn import svm
from sklearn import linear_model
numpy.warnings.filterwarnings('ignore')
```

[2]:
```python
f = open("5year.arff", 'r')
```

[3]:
```python
while not '@data' in f.readline():
    pass
```

[4]:
```python
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)
```

[5]:
```python
len(dataset)
```

[5]: 3031

[6]:
```python
X = [value[:-1] for value in dataset]
y = [value[-1] for value in dataset]
```

[7]:
```python
model = linear_model.LogisticRegression(C=1.0)
model.fit(X, y)
```

[7]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

[8]:
```
predictions = model.predict(X)
correctPredictions = predictions == y
sum(correctPredictions) / len(correctPredictions)
```

[8]:
```
0.9663477400197954
```

[9]:
```
TP_ = numpy.logical_and(predictions, y)
FP_ = numpy.logical_and(predictions, numpy.logical_not(y))
TN_ = numpy.logical_and(numpy.logical_not(predictions), numpy.logical_not(y))
FN_ = numpy.logical_and(numpy.logical_not(predictions), y)

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

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

```
0.48580623782459387
```

0.1.1 Accuracy is 0.9663477400197954 BER is 0.48580623782459387

0.2 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).

[10]:
```
# Train/validation/test splits

# Shuffle the data
Xy = list(zip(X,y))
random.shuffle(Xy)

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

N = len(y)

Ntrain = int(round(0.5*N))
```

```python
print(Ntrain)
Nvalid = int(round(0.25*N))
Ntest = int(round(0.25*N))

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

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

mod = linear_model.LogisticRegression(C=1.0, class_weight='balanced')
mod.fit(Xtrain, ytrain)
```

```
1516
```

[10]:
```
LogisticRegression(C=1.0, class_weight='balanced', dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=100, multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

[11]:
```python
#test
pred1 = mod.predict(Xtest)
correct1 = pred1 == ytest

TP_1 = numpy.logical_and(pred1, ytest)
FP_1 = numpy.logical_and(pred1, numpy.logical_not(ytest))
TN_1 = numpy.logical_and(numpy.logical_not(pred1), numpy.logical_not(ytest))
FN_1 = numpy.logical_and(numpy.logical_not(pred1), ytest)

TP1 = sum(TP_1)
FP1 = sum(FP_1)
TN1 = sum(TN_1)
FN1 = sum(FN_1)

# accuracy
print(sum(correct1) / len(correct1))

# BER
print(1 - 0.5 * (TP1 / (TP1 + FN1) + TN1 / (TN1 + FP1)))
```

```
0.7437252311756936
0.2996159107650216
```

[12]:
```python
#training
pred2 = mod.predict(Xtrain)
correct2 = pred2 == ytrain

TP_2 = numpy.logical_and(pred2, ytrain)
FP_2 = numpy.logical_and(pred2, numpy.logical_not(ytrain))
TN_2 = numpy.logical_and(numpy.logical_not(pred2), numpy.logical_not(ytrain))
FN_2 = numpy.logical_and(numpy.logical_not(pred2), ytrain)

TP2 = sum(TP_2)
FP2 = sum(FP_2)
TN2 = sum(TN_2)
FN2 = sum(FN_2)

# accuracy
print(sum(correct2) / len(correct2))

# BER
print(1 - 0.5 * (TP2 / (TP2 + FN2) + TN2 / (TN2 + FP2)))
```

```
0.7236147757255936
0.2754801579334806
```

[13]:
```python
#validation
pred3 = mod.predict(Xvalid)
correct3 = pred3 == yvalid

TP_3 = numpy.logical_and(pred3, yvalid)
FP_3 = numpy.logical_and(pred3, numpy.logical_not(yvalid))
TN_3 = numpy.logical_and(numpy.logical_not(pred3), numpy.logical_not(yvalid))
FN_3 = numpy.logical_and(numpy.logical_not(pred3), yvalid)

TP3 = sum(TP_3)
FP3 = sum(FP_3)
TN3 = sum(TN_3)
FN3 = sum(FN_3)

# accuracy
print(sum(correct3) / len(correct3))

# BER
print(1 - 0.5 * (TP3 / (TP3 + FN3) + TN3 / (TN3 + FP3)))
```

```
0.6992084432717678
0.23279672578444743
```

0.2.1 Training accuracy is 0.7236147757255936 Training BER is 0.2754801579334806 Validation accuracy is 0.6992084432717678 Validation BER is 0.23279672578444743 Test accuracy is 0.7437252311756936 Test BER is 0.2996159107650216

0.3 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)?

[14]:
```
BERtest = []
BERtrain = []
BERvalid = []
C = []

for i in range(-4, 5):

    mod1 = linear_model.LogisticRegression(C=10**i, class_weight='balanced')
    mod1.fit(Xtrain, ytrain)

    a = i
    C.append(a)
    #test
    pred_test = mod1.predict(Xtest)

    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)

    TPtest = sum(TP_test)
    FPtest = sum(FP_test)
    TNtest = sum(TN_test)
    FNtest = sum(FN_test)

    BER_test = 1 - 0.5 * (TPtest / (TPtest + FNtest) + TNtest / (TNtest +␣
 ↪FPtest))
    BERtest.append(BER_test)

    #training
    pred_train = mod1.predict(Xtrain)

    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)

    TPtrain = sum(TP_train)
    FPtrain = sum(FP_train)
    TNtrain = sum(TN_train)
    FNtrain = sum(FN_train)

    BER_train = 1 - 0.5 * (TPtrain / (TPtrain + FNtrain) + TNtrain / (TNtrain +␣
 ↪FPtrain))
    BERtrain.append(BER_train)

    #validation
    pred_valid = mod1.predict(Xvalid)

    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(yvalid))
    FN_valid = numpy.logical_and(numpy.logical_not(pred_valid), yvalid)

    TPvalid = sum(TP_valid)
    FPvalid = sum(FP_valid)
    TNvalid = sum(TN_valid)
    FNvalid = sum(FN_valid)

    BER_valid = 1 - 0.5 * (TPvalid / (TPvalid + FNvalid) + TNvalid / (TNvalid +␣
 ↪FPvalid))
    BERvalid.append(BER_valid)

print(BERtest)
print(BERtrain)
print(BERvalid)
```

```
[0.30577186151741553, 0.30645585604545933, 0.3098758286856782,
0.3221877301904661, 0.2996159107650216, 0.2961959381248027, 0.3078238451015469,
0.3098758286856782, 0.31808376302220354]
[0.2829886903566887, 0.28571906578330997, 0.2874255504249481,
0.28854313056280534, 0.2754801579334806, 0.27172589172187656,
0.28810814428160336, 0.2810345981395972, 0.2854714582078566]
[0.23893587994542975, 0.2443929058663028, 0.24848567530695775,
0.2787175989085948, 0.23279672578444743, 0.2568894952251023, 0.2664392905866302,
0.2505320600272851, 0.2525784447476125]
```

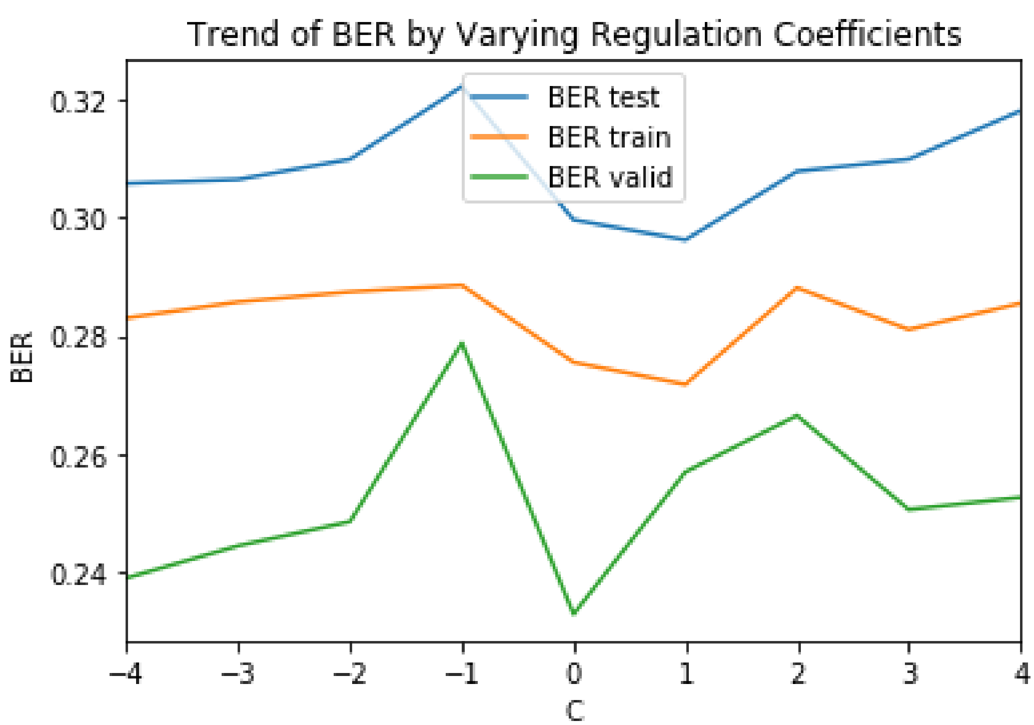
[15]:
```
import pandas as pd
dict = {'C':C, 'BER test':BERtest, 'BER train':BERtrain, 'BER valid': BERvalid}

df = pd.DataFrame(dict)
df.set_index('C', inplace=True)
df
```

[15]:

C	BER test	BER train	BER valid
-4	0.305772	0.282989	0.238936
-3	0.306456	0.285719	0.244393
-2	0.309876	0.287426	0.248486
-1	0.322188	0.288543	0.278718
0	0.299616	0.275480	0.232797
1	0.296196	0.271726	0.256889
2	0.307824	0.288108	0.266439
3	0.309876	0.281035	0.250532
4	0.318084	0.285471	0.252578

[17]:
```
from matplotlib import pyplot as plt
df.plot()
plt.xlabel('C')
plt.ylabel('BER')
plt.title('Trend of BER by Varying Regulation Coefficients')
```

[17]: Text(0.5, 1.0, 'Trend of BER by Varying Regulation Coefficients')



0.3.1 Based on the plot, I would like to select C = 10 **0 (C = 0) because the BER test value is lowest under this classifier.

0.4 6 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 F score (on the test set) of your (unweighted) classifier, for = 1 and = 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).1

[18]:
```
weights = [1.0] * len(ytrain)
mod2 = linear_model.LogisticRegression(C=1, solver='lbfgs')
mod2.fit(Xtrain, ytrain, sample_weight=weights)
```

[18]:
```
LogisticRegression(C=1, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

[19]:
```
pred = mod2.predict(Xtest)
retrieved = sum(pred)
relevant = sum(ytest)
intersection = sum([y and p for y,p in zip(ytest,pred)])
```

[20]:
```
precision = intersection / retrieved
recall = intersection / relevant

F1 = 2*(precision*recall)/(precision+recall)
F10 = 101*(precision*recall)/(100*precision + recall)
print(F1)
print(F10)
```

```
0.06896551724137931
0.03880138301959278
```

[31]:

```
f1 = []
f10 = []

for i in range(1,25):
    weights = []
    for a in range(len(ytrain)):
        if ytrain[a] == False:
            weights.append(1)
        else:
            weights.append(i)

    mod3 = linear_model.LogisticRegression(C=1, solver='lbfgs')
    mod3.fit(Xtrain, ytrain, sample_weight=weights)
    predd = mod3.predict(Xtest)
    retrieved = sum(predd)
    relevant = sum(ytest)
    intersection = sum([y and p for y,p in zip(ytest,predd)])

    precision = intersection / retrieved
    recall = intersection / relevant

    F1_ = 2*(precision*recall)/(precision+recall)
    f1.append(F1_)
    F10_ = 101*(precision*recall)/(100*precision + recall)
    f10.append(F10_)
```

[32]:

```
W = range(1, 25)

dict1 = {'Weight': W, 'F1':f1, 'F10':f10}

df1 = pd.DataFrame(dict1)
df1.set_index('Weight', inplace=True)
df1
```

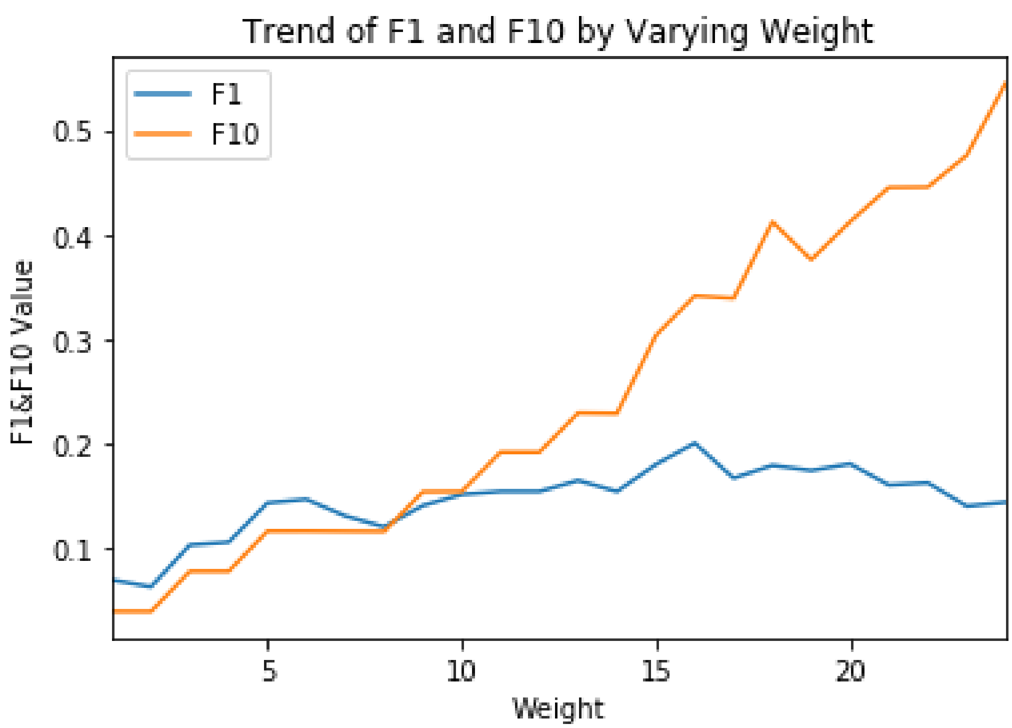
[32]:

Weight	F1	F10
1	0.068966	0.038801
2	0.062500	0.038757
3	0.102564	0.077306
4	0.105263	0.077335
5	0.142857	0.115826
6	0.146341	0.115870
7	0.130435	0.115649
8	0.120000	0.115473
9	0.140351	0.153554
10	0.150943	0.153788
11	0.153846	0.191360

```
12        0.153846  0.191360
13        0.164384  0.228938
14        0.153846  0.228507
15        0.179775  0.303417
16        0.200000  0.341216
17        0.166667  0.338926
18        0.178862  0.411939
19        0.173913  0.375604
20        0.180328  0.412092
21        0.160000  0.444934
22        0.162162  0.445261
23        0.139785  0.475725
24        0.143541  0.544377
```

[33]:
```
df1.plot()
plt.xlabel('Weight')
plt.ylabel('F1&F10 Value')
plt.title('Trend of F1 and F10 by Varying Weight')
```

[33]:
```
Text(0.5, 1.0, 'Trend of F1 and F10 by Varying Weight')
```



0.4.1 Based on the dataframe and plot above, we can see that many weight vectors can yield better performance in terms of the F1 and F10 scores. For example, weight vector 15 can yield higher F1 and F10 scores than weight vector 1.

0.5 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).

[34]:
```
from sklearn.decomposition import PCA # PCA library
```

[39]:
```
pca = PCA(n_components= len(Xtrain[0]))
pca.fit(Xtrain)
pca.components_[0]
```

```
65
```

[39]:
```
array([-5.41304419e-19,  1.50900971e-07, -7.77884215e-07,  8.57420252e-07,
        3.22071801e-06,  1.61739354e-03,  8.33542272e-07,  1.69501672e-07,
        4.44314754e-06, -4.28888028e-07,  7.36979300e-07,  1.27063148e-07,
        9.95748266e-07, -4.30783114e-06,  1.69200566e-07,  3.90380312e-03,
        8.35923061e-07,  4.74823620e-06,  5.85303991e-08,  4.49245282e-07,
        1.55180148e-05,  8.35214050e-08,  1.11816196e-07,  4.24514039e-07,
        6.70961850e-07,  9.33632802e-07,  7.38620221e-07, -6.67083412e-06,
        1.59472614e-06,  3.33041547e-06, -1.80798708e-06,  4.30036098e-07,
       -6.45902462e-04,  3.27101197e-06, -1.86385550e-06,  1.01338127e-07,
       -5.57053689e-07,  9.39777249e-04,  6.33228487e-07,  1.83515657e-07,
        1.45861838e-06, -3.32721845e-06,  3.01961713e-07,  1.00827924e-05,
       -5.43672719e-06, -2.65290128e-06,  2.43488623e-06, -3.99708113e-04,
        1.47653902e-07,  4.30944453e-07,  3.06027163e-06, -6.22562613e-07,
       -1.73238817e-06, -3.44046575e-06,  1.63774714e-06,  9.99990304e-01,
        1.93126575e-07, -3.95293751e-07, -2.52890204e-07,  6.61005811e-07,
       -1.42997276e-04, -2.00184047e-06, -2.36337201e-04,  4.38508837e-06,
       -7.53766759e-06])
```

0.6 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).

[36]:
```
BERvalid1 = []
BERtest1 = []

for i in range(5, 35, 5):
```

```
    pca = PCA(n_components= i)
    pca.fit(Xtrain)
    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.fit(Xpca_train, ytrain)
    pred_valid1 = mod.predict(Xpca_valid)

    TP_valid1 = numpy.logical_and(pred_valid1, yvalid)
    FP_valid1 = numpy.logical_and(pred_valid1, numpy.logical_not(yvalid))
    TN_valid1 = numpy.logical_and(numpy.logical_not(pred_valid1), numpy.
 ↪logical_not(yvalid))
    FN_valid1 = numpy.logical_and(numpy.logical_not(pred_valid1), yvalid)

    TPvalid1 = sum(TP_valid1)
    FPvalid1 = sum(FP_valid1)
    TNvalid1 = sum(TN_valid1)
    FNvalid1 = sum(FN_valid1)

    BER_valid1 = 1 - 0.5 * (TPvalid1 / (TPvalid1 + FNvalid1) + TNvalid1 /␣
 ↪(TNvalid1 + FPvalid1))
    BERvalid1.append(BER_valid1)

    pred_test1 = mod.predict(Xpca_test)

    TP_test1 = numpy.logical_and(pred_test1, ytest)
    FP_test1 = numpy.logical_and(pred_test1, numpy.logical_not(ytest))
    TN_test1 = numpy.logical_and(numpy.logical_not(pred_test1), numpy.
 ↪logical_not(ytest))
    FN_test1 = numpy.logical_and(numpy.logical_not(pred_test1), ytest)

    TPtest1 = sum(TP_test1)
    FPtest1 = sum(FP_test1)
    TNtest1 = sum(TN_test1)
    FNtest1 = sum(FN_test1)

    BER_test1 = 1 - 0.5 * (TPtest1 / (TPtest1 + FNtest1) + TNtest1 / (TNtest1 +␣
 ↪FPtest1))
    BERtest1.append(BER_test1)

print(BERvalid1)
print(BERtest1)
```

```
[0.404174624829468, 0.26098226466575714, 0.13186903137789907,
0.24030013642564807, 0.2512141882673943, 0.2416643929058664]
[0.3956908344733242, 0.3662001473218983, 0.2976428496264337,
```

```
0.33252657055666623, 0.31739976849415974, 0.3091918341576344]
```

[37]:
```python
N = range(5, 35, 5)

dict1 = {'N': N, 'BER Valid':BERvalid1, 'BER Test':BERtest1}

df1 = pd.DataFrame(dict1)
df1.set_index('N', inplace=True)
df1
```

[37]:

N	BER Valid	BER Test
5	0.404175	0.395691
10	0.260982	0.366200
15	0.131869	0.297643
20	0.240300	0.332527
25	0.251214	0.317400
30	0.241664	0.309192

[38]:
```python
df1.plot()
plt.xlabel('N')
plt.ylabel('BER')
plt.title('Trend of BER by Varying the number of Components')
```

[38]:
```
Text(0.5, 1.0, 'Trend of BER by Varying the number of Components')
```