In [1]:
```
import numpy
import random
from sklearn import linear_model
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
%matplotlib inline
#from sklearn.metrics import balanced_accuracy_score
# from urllib.request import urlopen
# import scipy.optimize
import warnings
warnings.filterwarnings('ignore')
```

Question 1

In [2]:
```
# From https://archive.ics.uci.edu/ml/datasets/Polish+companies+bankruptcy+dat
a
with open("../code_examples/data/polish_companies/5year.arff", 'r') as f:

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

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

In [3]:
```
# Balanced Error Rate function
def balanced_error_rate(pred, labels):
    TP_ = numpy.logical_and(pred, labels)
    FP_ = numpy.logical_and(pred, numpy.logical_not(labels))
    TN_ = numpy.logical_and(numpy.logical_not(pred), numpy.logical_not(labels
))
    FN_ = numpy.logical_and(numpy.logical_not(pred), labels)

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

    acc = (TP + TN) / (TP + FP + TN + FN)
    BER = 1 - 0.5 * (TP / (TP + FN) + TN / (TN + FP))
    return acc, BER
```

In [4]:
```
# Data setup
X = [d[:-1] for d in dataset]
y = [d[-1] for d in dataset]

# Fit model
mod = linear_model.LogisticRegression(C=1.0)
mod.fit(X,y)

pred = mod.predict(X)
correct = pred == y
#acc = sum(correct) / len(correct)
acc , BER = balanced_error_rate(y, pred)

print('Accuracy:', acc)
print('BER:', BER)
```

```
Accuracy: 0.9663477400197954
BER: 0.26636363636363636
```

Question 3

In [5]:
```
def shuffle_data(X, y):
    Xy = list(zip(X,y))
    random.shuffle(Xy)
    X, y = zip(*Xy)
    return list(X), list(y)

def split_data(X, y, percentages = (50,25,25)):
    # percentages = (train, val, test) (percentages)
    X_split = len(X) // 100 #1 percent of X
    y_split = len(y) // 100 #1 percent of X
    (train, val, test) = percentages
    X_train, X_val, X_test = X[: X_split*train], X[X_split*train : X_split*(tr
ain + val)],\
                             X[X_split*(train + val): ]
    y_train, y_val, y_test = y[: y_split*train], y[y_split*train : y_split*(tr
ain + val)],\
                             y[y_split*(train + val): ]
    return X_train, X_val, X_test, y_train, y_val, y_test
```

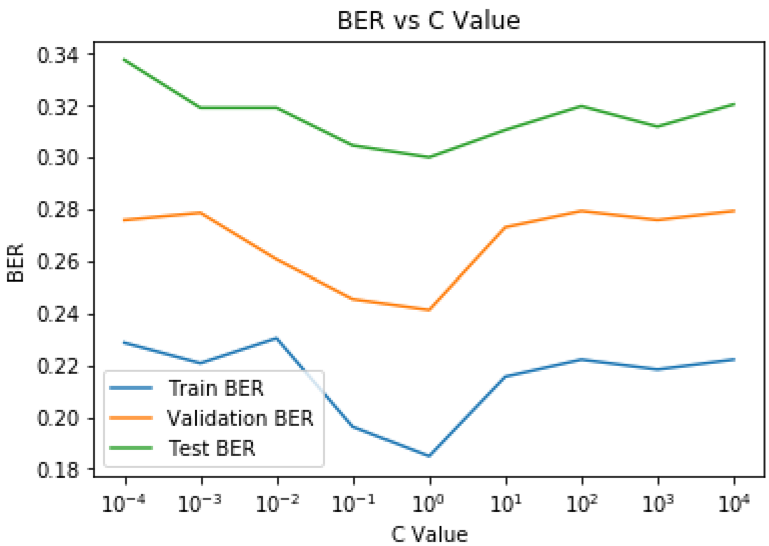
In [6]:
```
X, y = shuffle_data(X, y)
X_train, X_val, X_test, y_train, y_val, y_test =  split_data(X, y, (50,25,25))

# Fit model
mod_train = linear_model.LogisticRegression(C = 1.0, class_weight = 'balanced'
)
mod_train.fit(X_train,y_train)

pred_train = mod_train.predict(X_train)
pred_val   = mod_train.predict(X_val)
pred_test  = mod_train.predict(X_test)

acc_train, BER_train =  balanced_error_rate(pred_train, y_train)
acc_val,   BER_val   =  balanced_error_rate(pred_val, y_val)
acc_test,  BER_test  =  balanced_error_rate(pred_test, y_test)
print('Train accuracy:       {0:.3f}    Train BER:      {1:.3f}'.format(acc_tr
ain, BER_train))
print('Validation accuracy:  {0:.3f}    Validation BER: {1:.3f}'.format(acc_va
l, BER_val))
print('Test accuracy:        {0:.3f}    Test BER:       {1:.3f}'.format(acc_te
st, BER_test))
```

```
Train accuracy:       0.831    Train BER:      0.168
Validation accuracy:  0.809    Validation BER: 0.227
Test accuracy:        0.819    Test BER:       0.234
```

Question 4

In [7]:
```
# Data already shuffled above

C_values   = [10 ** i for i in range(-4,5)]
accuracies = []
BERs       = []

# Regularization pipepline:
for C_cur in C_values:
    # print(C_cur)
    mod = linear_model.LogisticRegression(C = C_cur, class_weight = 'balanced'
)
    mod.fit(X_train, y_train)

    pred_train = mod.predict(X_train)
    pred_val   = mod.predict(X_val)
    pred_test  = mod.predict(X_test)

    acc_train, BER_train =  balanced_error_rate(pred_train, y_train)
    acc_val,   BER_val   =  balanced_error_rate(pred_val, y_val)
    acc_test,  BER_test  =  balanced_error_rate(pred_test, y_test)

    accuracies.append([acc_train, acc_val, acc_test])
    BERs.append([BER_train, BER_val, BER_test])
```

In [332]:
```
plt.plot(C_values,[BER[0] for BER in BERs], label='Train BER')
plt.plot(C_values,[BER[1] for BER in BERs], label='Validation BER')
plt.plot(C_values,[BER[2] for BER in BERs], label='Test BER')
plt.ylabel('BER')
plt.xlabel('C Value'), plt.xscale('log'), plt.xticks(C_values)
plt.title('BER vs C Value')
plt.legend()
plt.show()
```



Q4 ANSWER

Consider I didn't have access to the test set (which is usually how things work). I would then choose C = 1. Because that is the number for which I get the lowest Balanced Error Rate from the validation set. We shouldn't be concerned with how C value effects the training set BER, as we need unseen data (validation set) to fine tune our parameters. (Here the parameter to fine tune is regularization parameter C.)

In this question I also have access to the test set. From the performance on the test set, I see that C = 1 is best for low test set BER. So for this particular shuffling & split of the data, and considering the test set accuracy, C = 1 was the best value indeed.

In [ ]:
```
# plt.plot(C_values,[accuracy[0] for accuracy in accuracies], label='Train')
# plt.plot(C_values,[accuracy[1] for accuracy in accuracies], label='Validatio
n')
# plt.plot(C_values,[accuracy[2] for accuracy in accuracies], label='Test')
# plt.ylabel('Accuracy')
# plt.xlabel('C Value'), plt.xscale('log'), plt.xticks(C_values)
# plt.title('Accuracy vs C Value')
# plt.legend()
# plt.show()
```

Question 6

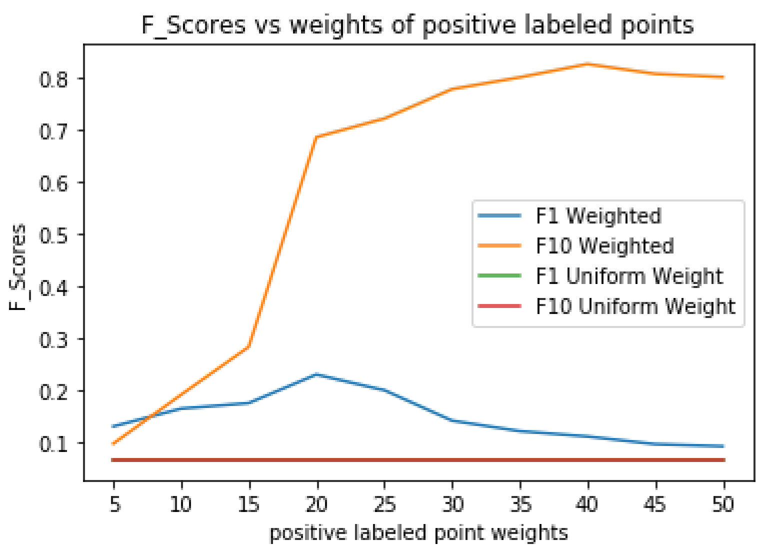
In [306]:
```
def F_Score(pred, labels, Beta = 1):
    retrieved = sum(pred) # number of positivie predictions
    relevant = sum(labels) # number of positive labels
    intersection = sum([y and p for y,p in zip(labels,pred)])
    precision = intersection / retrieved # (retrieved intersection relevant) / 
retrieved
    recall    = intersection / relevant     # (retrieved intersection relevan
t) / relevant
    F = (1 + Beta ** 2) * (precision * recall) / ((Beta ** 2) * precision + re
call)
    return F
```

In [307]:
```
weights = [1.0] * len(y_train)
mod = linear_model.LogisticRegression(C = 1, solver='lbfgs')
mod.fit(X_train, y_train, sample_weight=weights)

pred_test  = mod.predict(X_test)

F1 = F_Score(pred_test, y_test, Beta = 1)
F10 = F_Score(pred_test, y_test, Beta = 10)

positive_count = [point == True for point in y_train]
negative_count = [point == False for point in y_train]
print('Positives:', sum(positive_count), '\nNegatives:', sum(negative_count))
print('Uniform weight, F1 Score: ',  F1)
print('Uniform weight, F10 Score: ', F10)
```

```
Positives: 50
Negatives: 1450
Uniform weight, F1 Score:  0.06451612903225806
Uniform weight, F10 Score:  0.037352071005917156
```

In [300]:
```
positive_weights = range(5,51,5)
negative_weight = 1
F_Scores = []
for positive_weight in positive_weights:
    # A new weight vector is created in each iteration
    weights = [positive_weight if i == True else negative_weight for i in y_tr
ain]
    # New model is trained according to the new weight vector
    mod = linear_model.LogisticRegression(C = 1, solver='lbfgs')
    mod.fit(X_train, y_train, sample_weight = weights)

    pred_test  = mod.predict(X_test)

    F1 = F_Score(pred_test, y_test, Beta = 1)
    F10 = F_Score(pred_test, y_test, Beta = 10)
    F_Scores.append((F1, F10))
```

In [331]:
```
plt.plot(positive_weights,[F_Score[0] for F_Score in F_Scores], label='F1 Weighted')
plt.plot(positive_weights,[F_Score[1] for F_Score in F_Scores], label='F10 Weighted')
plt.plot(positive_weights, [F1] * len(positive_weights), label='F1 Uniform Weight')
plt.plot(positive_weights, [F1] * len(positive_weights), label='F10 Uniform Weight')
plt.ylabel('F_Scores')
plt.xlabel('positive labeled point weights'), plt.xticks(positive_weights)
plt.title('F_Scores vs weights of positive labeled points')
plt.legend()
plt.show()
```



Q6 ANSWER

For this question, I decided to give larger weights to positively labeled data points. This is because there are very few of them, and to balance this they benefit from higher weights (in the model). These higher weights make the classifier focus more on the True Positive Rate. With this idea, I kept the negative label weights fixed at 1, and varied the weight of positive labeled data points. From the plot above we can see that the following weight vectors gave best results (for F1 and F10 seperately):

For F1:
positive labels - weight 5
negative labels - weight 1

For F10:
positive labels - weight 40
negative labels - weight 1

This is within my expectations. For the F1 score, precision and recall has equal contribution. Moderate positive weights (5 for positives) favor recall (and True Positive Rate) more than precision, but not by a large margin. F10 has a high Beta value (B = 10), which gives much more weight to recall. Very high weights (40) make us predict more of the positively labeled points correctly, increasing recall and therefore F10 Score. It reduces the precision, but this doesn't effect the F10 score much.

Question 7

In [328]:
```
pca = PCA(len(X_train[1]))
pca.fit(X_train)
print('There are', pca.n_components_, 'PCA Components')
print('First PCA Component:\n', pca.components_[0])
# mean_axis0 = numpy.mean(numpy.array(X_train), axis = 0)
# print((pca.components_[0] - mean_axis0))
```

```
There are 65 PCA Components
First PCA Component:
 [-5.29202960e-19  1.33043608e-08  3.01828418e-07  1.47274093e-06
  6.76386291e-06  8.40490801e-04 -1.30568738e-06  2.06682343e-06
  7.23377455e-06 -8.73897166e-07 -8.40140858e-08  3.44607559e-07
  1.94236269e-06  4.28742351e-07  2.06682343e-06 -9.74057737e-04
  1.78806294e-06  8.02272170e-06  2.06682343e-06  4.87273604e-07
  5.87299510e-05 -1.66296270e-05  3.02906685e-07  4.24600753e-07
  6.64388756e-07  8.11565274e-07  1.61071258e-06 -4.65364645e-05
  2.64556517e-06  4.23545973e-06 -1.75026675e-06  4.80585438e-07
 -7.74892407e-04  7.59745909e-06 -1.09617285e-06  2.56312314e-07
 -1.02373003e-06  9.63870105e-03 -5.77738247e-07  2.90954686e-07
  2.90017358e-06  9.27405978e-07  3.44823396e-07  1.04975126e-04
  4.62435386e-05 -2.65176695e-06  4.83216113e-06 -4.20247676e-04
  3.53274872e-07  4.01931850e-07  5.95834553e-06 -1.05091716e-06
 -2.09917314e-06  4.66554027e-06  2.71958154e-06  9.99952223e-01
  2.64713473e-07 -1.33709692e-07 -3.72307529e-07  1.71654246e-06
 -2.21438239e-04 -1.64828132e-05 -3.80869631e-04  9.33344875e-06
 -1.53802351e-05]
```

Question 8

In [170]:
```
X_pca_train = numpy.matmul(X_train, pca.components_.T)
X_pca_val   = numpy.matmul(X_val,   pca.components_.T)
X_pca_test  = numpy.matmul(X_test,  pca.components_.T)

N_Components = range(5,31,5)
BERs_PCA = []

for N in N_Components:
    # Extract first N components
    pca_train, pca_val, pca_test = X_pca_train[:,:N], X_pca_val[:,:N], X_pca_t
est[:,:N]
    mod = linear_model.LogisticRegression(C = 1, class_weight = 'balanced')
    mod.fit(pca_train, y_train)

    pred_val   = mod.predict(pca_val)
    pred_test  = mod.predict(pca_test)

    _,  BER_val   =  balanced_error_rate(pred_val, y_val)
    _,  BER_test  =  balanced_error_rate(pred_test, y_test)

    BERs_PCA.append((BER_val, BER_test))
```

In [329]:
```
plt.plot(N_Components,[BER[0] for BER in BERs_PCA], label='Validation BER')
plt.plot(N_Components,[BER[1] for BER in BERs_PCA], label='Test BER')
plt.ylabel('BER')
plt.xlabel('using first N pca comp.'), plt.xticks(N_Components)
plt.title('using first N pca comp. vs BER')
plt.legend()
plt.show()
```