In [1]:
```
import csv as csv
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn import svm
from sklearn import cross_validation

# Open up the csv file in to a Python object
data = pd.read_csv('2013MT60597.csv',header = -1)
# data.describe()
```

In [2]:
```
X_train_raw = data.iloc[:,0:25]
Y_train = data.iloc[:,25]


#normalize the data
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train_raw)

from scipy import linalg
a,b,c = linalg.svd(X_train)
print(b)
```

```
[ 59.2730931   58.37051386  57.85374196  57.35637875  57.19679766
  57.06698198  56.27697423  56.16808756  56.11975154  55.40697927
  55.25683585  55.02877665  54.98227379  54.53023193  54.14857773
  54.09110117  53.54706172  53.28792843  52.7684147   52.4185597
  52.16889704  51.74522114  51.33870103  51.17326139  50.44524561]
```

Observations:

- From the above SVD of the design matrix, we can see that all of the singular values are significant
- There is no sudden drop in singular values, which shows that all dimensions hold some significant variance of the data
- Even if we take the first 10 features, we wont get better performance

In [17]:
```
#Create a subset of the database
# choose only those classes whose labels are 2 and 3
B = np.zeros(Y_train.shape[0],dtype=bool)
for i in range(0,Y_train.shape[0]):
    if(Y_train[i] == 2 or Y_train[i] == 3):
        B[i] = True

X_23 = X_train[B]
Y_23 = Y_train[B]
# X_23.iloc[:,0:10]
clf = svm.SVC()
# clf.fit(X_23,Y_23)
score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jobs=4).me
an()
print(score)
C = [0.01,0.1,0.5,1,1.5,2,10]
G = [0.01,0.1,0.3,0.5,0.7,0.9,2]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j])
        score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jo
bs=4).mean()
        scores[i,j] = score
print(scores)
print(np.amax(scores))
```

```
0.977826580901
[[ 0.51029346  0.51029346  0.51029346  0.51029346  0.51029346  0.51029346
   0.51029346]
 [ 0.95880136  0.8859807   0.51029346  0.51029346  0.51029346  0.51029346
   0.51029346]
 [ 0.97301267  0.97462638  0.57056532  0.53566708  0.52140617  0.51029346
   0.51029346]
 [ 0.97303827  0.97777618  0.86867079  0.67369832  0.56424091  0.54045459
   0.53566708]
 [ 0.97462558  0.97618888  0.8781698   0.69912234  0.58331493  0.553129
   0.53566708]
 [ 0.97306388  0.97618888  0.8781698   0.69912234  0.58331493  0.553129
   0.53566708]
 [ 0.97938908  0.97618888  0.8781698   0.69912234  0.58331493  0.553129
   0.53566708]]
0.979389080901
```

Observation

We can see that using the default settings , we get a accuracy of 0.9778. Using a little bit of tuning we saw that we get the best value (still less than the default) of 0.9777. for this C=1,gamma=0.1 We still need to optimize it further.

In [4]:
```
G = [0.001,0.01,0.02,0.03,0.04,0.05,0.1,0.2,0.3]
scores = []
for g in G:
    clf = svm.SVC(C = 1, gamma=g)
    score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jobs=4
).mean()
    scores.append(score)
print(scores)
```

```
[0.95562676011264713, 0.97303827444956459, 0.97303827444956459, 0.9762136776
7537104, 0.97782658090117758, 0.97782658090117758, 0.97777617767537106, 0.95
882776497695832, 0.8686707949308754]
```

Observation

We observe that the maximum accuracy is obtained when C = 1 and Gamma = 0.4 or 0.5 and best accuracy is 0.9778

In the next section, we vary the kernel and do the same optimization

In [5]:
```
#using the default settings
scores = []

clf = svm.SVC(kernel='poly')
score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)
```

```
[0.98092517921146949]
```

In [6]:
```
#Using some custom settings to optimize further
C = [0.1,0.5,1,3,10]
G = [0.001,0.01,0.02,0.03,0.04,0.05,1]
scores = []
for c in C:
    for g in G:
        clf = svm.SVC(C = c, gamma=g,kernel='poly')
        score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jo
bs=4).mean()
        scores.append(score)
print(scores)
print(scores.index(max(scores)))
print(scores[scores.index(max(scores))])
```

```
[0.5102934587813619, 0.5102934587813619, 0.5102934587813619, 0.9557027649769
5839, 0.96835157450076781, 0.97785138248847936, 0.97460077444956461, 0.51029
34587813619, 0.5102934587813619, 0.95887736815156155, 0.97785138248847936, 0
.98095078084997456, 0.98092517921146949, 0.97460077444956461, 0.510293458781
3619, 0.54509168586789547, 0.97311427931387606, 0.98095078084997456, 0.98092
517921146949, 0.97618807603686641, 0.97460077444956461, 0.5102934587813619,
0.95731486815156153, 0.98095078084997456, 0.97933787762416791, 0.97460077444
956461, 0.97460077444956461, 0.97460077444956461, 0.5102934587813619, 0.9778
5138248847936, 0.97933787762416791, 0.97460077444956461, 0.97460077444956461
, 0.97460077444956461, 0.97460077444956461]
11
0.98095078085
```

Observation

We see that the best result is obtained when C = 0.5 and gamma = 0.04. This is a maxima and we decrease our accuracy by moving in any direction. The best result is 0.98095

Result: polynomial kernel gives a better performance than rbf kernel

In [7]:
```
#using the default settings
scores = []

clf = svm.SVC(kernel='linear')
score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)
```

```
[0.95557795698924719]
```

In [8]:
```
C = [0.001,0.003,0.01,0.03,0.1,0.5,1,3,10]
scores = []
for c in C:
    clf = svm.SVC(C = c,kernel='linear')
    score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jobs=4
).mean()
    scores.append(score)
print(scores)
print(scores.index(max(scores)))
print(scores[scores.index(max(scores))])
```

```
[0.9508640552995391, 0.95721406169994872, 0.96822596646185366, 0.96668906810
035826, 0.97142697132616473, 0.96029025857654882, 0.95557795698924719, 0.957
14045698924721, 0.95719086021505362]
4
0.971426971326
```

Observation

- We see that the best performance we can extract by using a linear kernel is 0.9714, which is worse than both polynomial kernel and rbf kernel.
- Also from here we get an intuition that maybe higher order polynomials may perform better. We check that in the next section

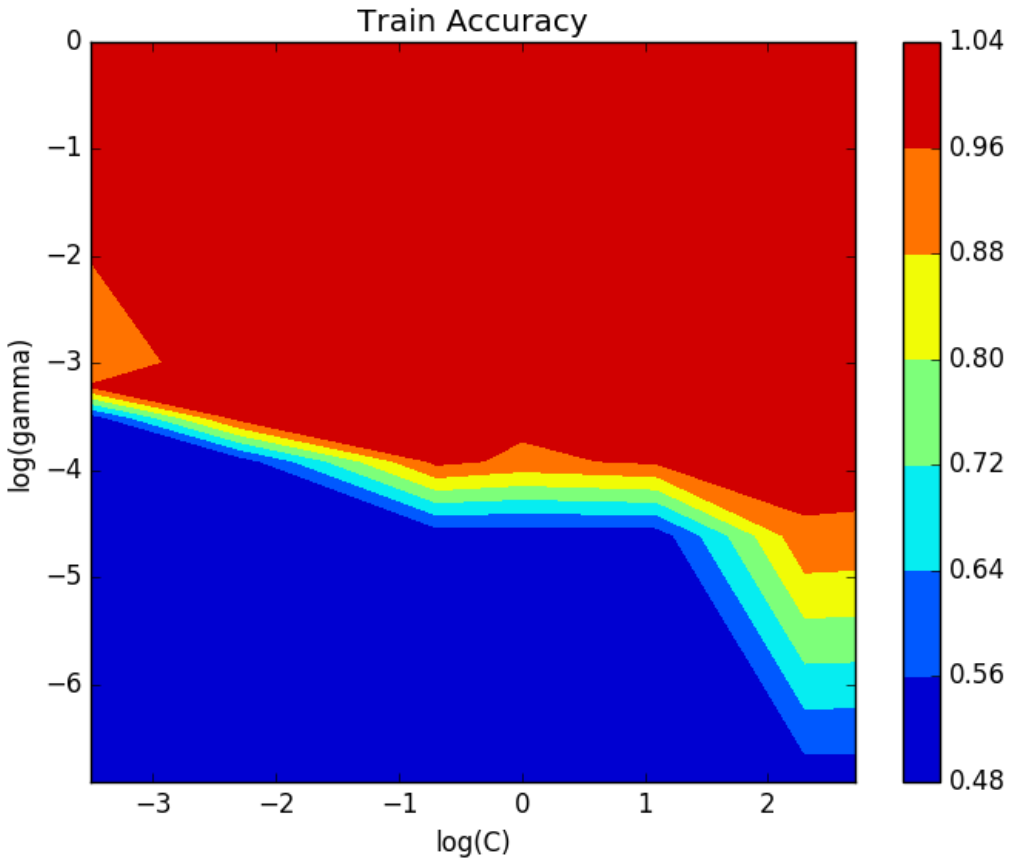
In [23]:
```
# Increase the degree of the polynomial kernel
#Using some custom settings to optimize further
C = [0.03,0.1,0.5,1,3,10,15]
G = [0.001,0.01,0.02,0.03,0.04,0.05,1]
scores = np.zeros([7,7])
i = 0
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j],kernel='poly', degree=4)
        score = cross_validation.cross_val_score(clf, X_23,Y_23, cv=10, n_jo
bs=4).mean()
#         score = clf.fit(X_23,Y_23).score(X_23,Y_23)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
```

```
[[ 0.51029346  0.51029346  0.51029346  0.51029346  0.86839798  0.91606903
   0.96515217]
 [ 0.51029346  0.51029346  0.51029346  0.91599382  0.93032994  0.95255376
   0.96515217]
 [ 0.51029346  0.51029346  0.90961902  0.94620376  0.96361527  0.97467678
   0.96515217]
 [ 0.51029346  0.51029346  0.91765553  0.95572837  0.97308948  0.96986367
   0.96515217]
 [ 0.51029346  0.51029346  0.95094086  0.97308948  0.97147657  0.96512737
   0.96515217]
 [ 0.51029346  0.94781506  0.96674027  0.97147657  0.96671467  0.96515217
   0.96515217]
 [ 0.51029346  0.91455613  0.97150218  0.96668907  0.96671467  0.96515217
   0.96515217]]
0.974676779314
```

In [12]:
```
Cvalues = np.repeat(C,7)
Gvalues = np.tile(G,7)

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

Observation

We see that the polynomial kernel decreases in accuracy when we increase the degree of the kernel.

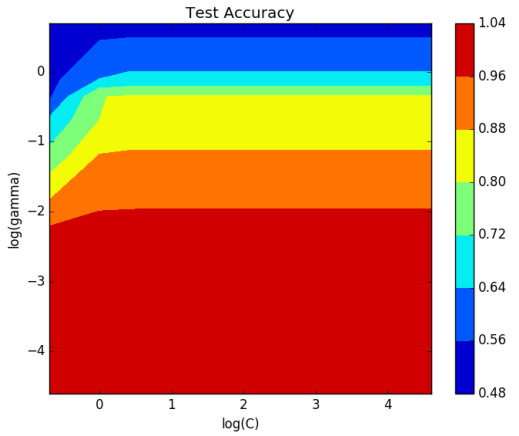


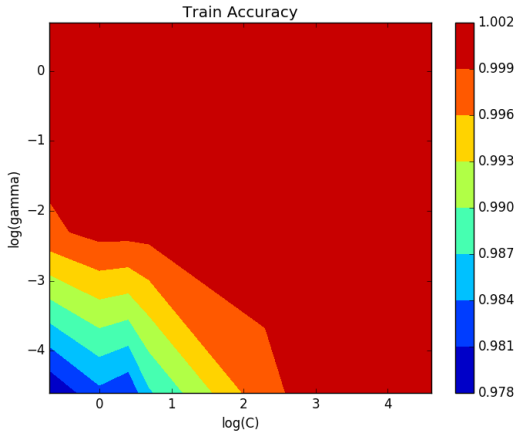
In [36]:
```
# Now do the same thing for two other classes
# This time take the classes to be 5 and 6

B = np.zeros(Y_train.shape[0],dtype=bool)
for i in range(0,Y_train.shape[0]):
    if(Y_train[i] == 5 or Y_train[i] == 6):
        B[i] = True

X_56 = X_train[B]
Y_56 = Y_train[B]
# X_23.iloc[:,0:10]
clf = svm.SVC()
# clf.fit(X_23,Y_23)
score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jobs=4).me
an()
print(score)
C = [0.5,1,1.5,2,10,20,100]
G = [0.01,0.1,0.3,0.5,0.7,0.9,2]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j])
#         score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_
jobs=4).mean()
        score = clf.fit(X_56,Y_56).score(X_56,Y_56)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
Cvalues = np.repeat(C,7)
Gvalues = np.tile(G,7)

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

```
0.988218390805
[[ 0.97815126  0.99831933  1.          1.          1.          1.          1
.        ]
 [ 0.98319328  1.          1.          1.          1.          1.          1
.        ]
 [ 0.98151261  1.          1.          1.          1.          1.          1
.        ]
 [ 0.98655462  1.          1.          1.          1.          1.          1
.        ]
 [ 0.99831933  1.          1.          1.          1.          1.          1
.        ]
 [ 1.          1.          1.          1.          1.          1.          1
.        ]
 [ 1.          1.          1.          1.          1.          1.          1
.        ]]
1.0
```

Observation

We can see that using the default settings , we get a accuracy of 0.9882. Using a little bit of tuning we saw that we get the best value of 0.9916. We see that it is in somewhere in the middle of the matrix. So it is the optimal and we have obtained the optimal value.





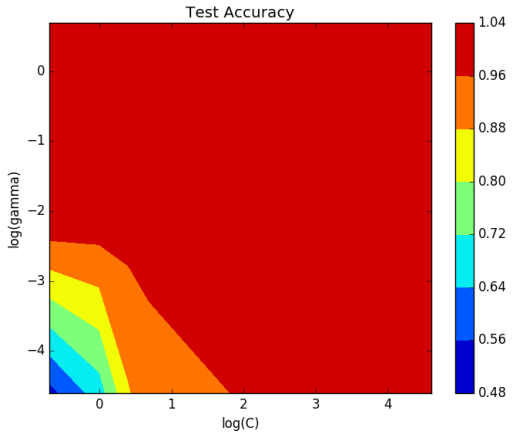
In the next section, we vary the kernel and do the same optimization

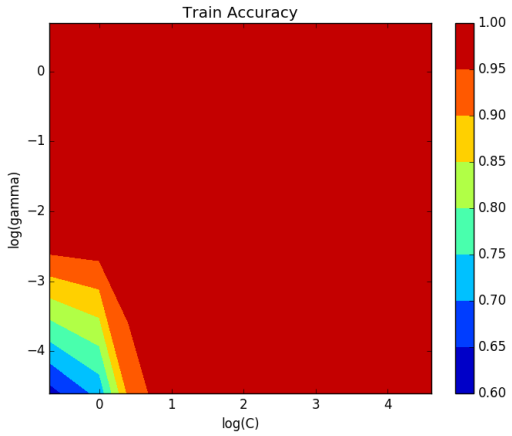
In [38]:
```
#using the default settings
scores = []

clf = svm.SVC(kernel='poly')
score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.01,0.1,0.3,0.5,0.7,0.9,2]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly')
#         score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_
jobs=4).mean()
        score = clf.fit(X_56,Y_56).score(X_56,Y_56)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
Cvalues = np.repeat(C,7)
Gvalues = np.tile(G,7)

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

```
[0.98155269822715741]
[[ 0.62857143  1.          1.          1.          1.          1.          1
.        ]
 [ 0.71596639  1.          1.          1.          1.          1.          1
.        ]
 [ 0.91092437  1.          1.          1.          1.          1.          1
.        ]
 [ 0.9512605   1.          1.          1.          1.          1.          1
.        ]
 [ 0.98991597  1.          1.          1.          1.          1.          1
.        ]
 [ 0.99831933  1.          1.          1.          1.          1.          1
.        ]
 [ 1.          1.          1.          1.          1.          1.          1
.        ]]
1.0
```

Observations

We see that in this case , the best performance obtained is 0.984 which is better than the default settings for the kernel.





Result : For this case, we see that polynomial kernel gives a worse performance than rbf kernel

In [26]:
```
#using the linear kernel
C = [0.5,1,1.5,2,10,20,100]
G = [0.01,0.1,0.3,0.5,0.7,0.9,2]
scores = np.zeros([7,1])
for i in range(0,7):
    clf = svm.SVC(C = C[i], kernel='linear')
    score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jobs=4
).mean()
    scores[i] = score
print(scores)
print(np.amax(scores))
```

```
[[ 0.95635106]
 [ 0.9530752 ]
 [ 0.9546844 ]
 [ 0.95635106]
 [ 0.96137931]
 [ 0.96143678]
 [ 0.95477011]]
0.961436781609
```

Observation

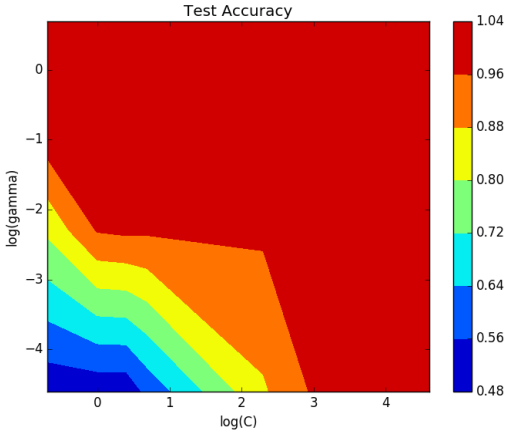
- We see that the best performance we can extract by using a linear kernel is 0.9614, which is worse than both polynomial kernel and rbf kernel.
- Also from here we get an intuition that maybe higher order polynomials may perform better. We check that in the next section

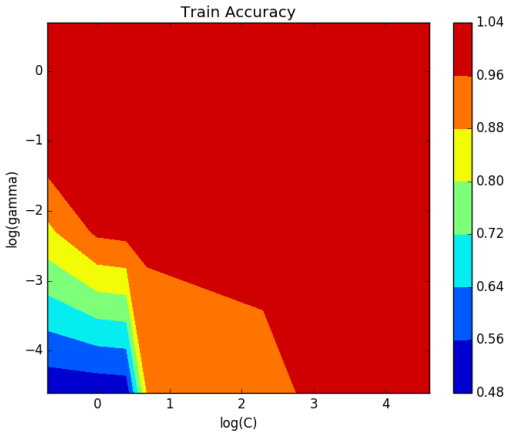
In [42]:
```
#Changing the degree of the polynomial kernel
scores = []

clf = svm.SVC(kernel='poly', degree = 4)
score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.01,0.02,0.04,0.05,0.1,0.3,1]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly', degree=4)
#         score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_
jobs=4).mean()
        score = clf.fit(X_56,Y_56).score(X_56,Y_56)
        scores[i,j] = score
print(scores)
print(np.amax(scores))

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

```
[0.97985680888369375]
[[ 0.50084034  0.85882353  0.99663866  1.          1.          1.          1
.        ]
 [ 0.50084034  0.97478992  1.          1.          1.          1.          1
.        ]
 [ 0.50756303  0.98655462  1.          1.          1.          1.          1
.        ]
 [ 0.88235294  0.98151261  1.          1.          1.          1.          1
.        ]
 [ 0.9210084   0.99663866  1.          1.          1.          1.          1
.        ]
 [ 0.97983193  1.          1.          1.          1.          1.          1
.        ]
 [ 0.99495798  1.          1.          1.          1.          1.          1
.        ]]
1.0
```

Observation

We see that by increasing the degree of the polynomial, we have increased the (best) accuracy of the model, so we can conclude that increasing the degree is increasing the accuracy in this case. We check for higher degree in the next section.





In [30]:
```
#Changing the degree of the polynomial kernel
scores = []

clf = svm.SVC(kernel='poly', degree = 5)
score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.01,0.02,0.04,0.05,0.1,0.3,1]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly', degree=5)
        score = cross_validation.cross_val_score(clf, X_56,Y_56, cv=10, n_jo
bs=4).mean()
        #         score = clf.fit(X_23,Y_23).score(X_23,Y_23)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
```

```
[0.97307519968829137]
[[ 0.50084746  0.58147769  0.96293201  0.97979934  0.98324761  0.98324761
   0.98324761]
 [ 0.50084746  0.77641535  0.9730752   0.98155172  0.98324761  0.98324761
   0.98324761]
 [ 0.50084746  0.8588535   0.97813267  0.98152348  0.98324761  0.98324761
   0.98324761]
 [ 0.50084746  0.88899669  0.98319014  0.98152348  0.98324761  0.98324761
   0.98324761]
 [ 0.52101987  0.95275959  0.97985681  0.98324761  0.98324761  0.98324761
   0.98324761]
 [ 0.62354568  0.96798948  0.98324761  0.98324761  0.98324761  0.98324761
   0.98324761]
 [ 0.92086402  0.98155172  0.98324761  0.98324761  0.98324761  0.98324761
   0.98324761]]
0.983247613481
```

Observations

We see that for degree 5, the accuracy is less than degree 4. So we can conclude that the best accuracy is obtained for polynomial kernel of degree 4.

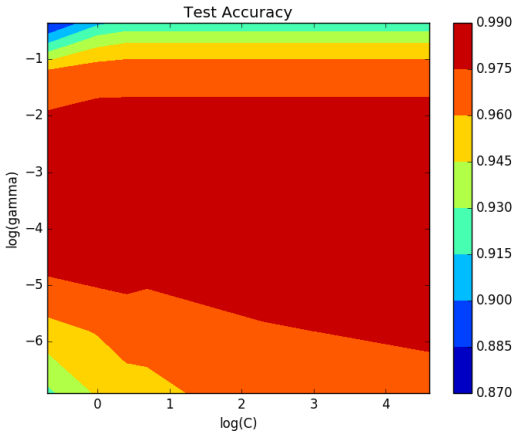
Now we repeat the same analysis for classes 1 and 8

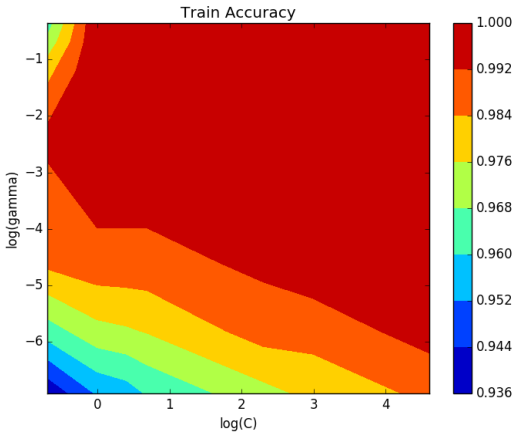
In [45]:
```
B = np.zeros(Y_train.shape[0],dtype=bool)
for i in range(0,Y_train.shape[0]):
    if(Y_train[i] == 1 or Y_train[i] == 8):
        B[i] = True

X_18 = X_train[B]
Y_18 = Y_train[B]
# X_23.iloc[:,0:10]
clf = svm.SVC()
# clf.fit(X_23,Y_23)
score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jobs=4).me
an()
print(score)
C = [0.5,1,1.5,2,10,20,100]
G = [0.001,0.003,0.01,0.1,0.3,0.5,0.7]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j])
#         score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_
jobs=4).mean()
        score = clf.fit(X_18,Y_18).score(X_18,Y_18)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
Cvalues = np.repeat(C,7)
Gvalues = np.tile(G,7)

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

```
0.987499236874
[[ 0.93730408  0.96394984  0.98589342  0.99373041  0.98119122  0.97021944
   0.96394984]
 [ 0.95297806  0.97335423  0.98902821  1.          1.          1.          1
.        ]
 [ 0.95611285  0.97492163  0.98902821  1.          1.          1.          1
.        ]
 [ 0.96081505  0.97648903  0.98902821  1.          1.          1.          1
.        ]
 [ 0.97335423  0.98746082  0.99373041  1.          1.          1.          1
.        ]
 [ 0.97805643  0.98746082  0.9968652   1.          1.          1.          1
.        ]
 [ 0.98589342  0.99529781  1.          1.          1.          1.          1
.        ]]
1.0
```

Observations

The best accuracy obtained is 0.9874 using the rbf kernel for this case.





Now we change the kernel type.

In [48]:
```
#using the default settings for the polynomial kernel
scores = []

clf = svm.SVC(kernel='poly')
score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.009,0.03,0.05,0.01,0.1,0.3,0.5]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly')
        score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jo
bs=4).mean()
#         score = clf.fit(X_18,Y_18).score(X_18,Y_18)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
Cvalues = np.repeat(C,7)
Gvalues = np.tile(G,7)

plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.reshape
(len(C), len(C))),
             scores)
plt.colorbar()
plt.title('Train Accuracy')
plt.xlabel('log(C)')
plt.ylabel('log(gamma)')
plt.show()
```

```
[0.98898809523809528]
[[ 0.52820055  0.98276213  0.9889881   0.52820055  0.9874256   0.9889881
   0.9889881 ]
 [ 0.52820055  0.98432463  0.9889881   0.52820055  0.9889881   0.9889881
   0.9889881 ]
 [ 0.52820055  0.98588713  0.9889881   0.53916361  0.9889881   0.9889881
   0.9889881 ]
 [ 0.53757631  0.98588713  0.9889881   0.59236607  0.9889881   0.9889881
   0.9889881 ]
 [ 0.97028465  0.9889881   0.9889881   0.97653541  0.9889881   0.9889881
   0.9889881 ]
 [ 0.98432463  0.9874256   0.9889881   0.98276213  0.9889881   0.9889881
   0.9889881 ]
 [ 0.9889881   0.9889881   0.9889881   0.9889881   0.9889881   0.9889881
   0.9889881 ]]
0.988988095238
```

Observation

The best accuracy obtained is 0.9889 which is better than rbf kernel.

Result: Polynomial kernel is better than rbf kernel for this case.

Now we check for linear kernel

In [50]:
```
#using the linear kernel
C = [0.1,0.3,0.5,1,1.5,2,10]

scores = np.zeros([7,1])
for i in range(0,7):
    clf = svm.SVC(C = C[i], kernel='linear')
    score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jobs=4
).mean()
    scores[i] = score
print(scores)
print(np.amax(scores))
```

```
[[ 0.96872215]
 [ 0.97028541]
 [ 0.97499771]
 [ 0.97026061]
 [ 0.97026061]
 [ 0.97028465]
 [ 0.96869811]]
0.974997710623
```

Observation

Linear kernel performs worse than both rbf and polynomial kernel. Now we vary the degree of the polynomial kernel.

In [55]:
```
#using the default settings for the polynomial kernel
scores = []

clf = svm.SVC(kernel='poly', degree = 4)
score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.001,0.005,0.009,0.03,0.05,0.01,0.1]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly', degree=4)
        score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jo
bs=4).mean()
#         score = clf.fit(X_18,Y_18).score(X_18,Y_18)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
# Cvalues = np.repeat(C,7)
# Gvalues = np.tile(G,7)

# plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.resha
pe(len(C), len(C))),
#              scores)
# plt.colorbar()
# plt.title('Test Accuracy')
# plt.xlabel('log(C)')
# plt.ylabel('log(gamma)')
# plt.show()
```

```
[0.99213789682539688]
[[ 0.52820055  0.52820055  0.52820055  0.85264461  0.9921379   0.52820055
   0.98434867]
 [ 0.52820055  0.52820055  0.52820055  0.97340888  0.9921379   0.52820055
   0.98122367]
 [ 0.52820055  0.52820055  0.52820055  0.98437271  0.98901213  0.52820055
   0.98122367]
 [ 0.52820055  0.52820055  0.52820055  0.99059944  0.98901213  0.52820055
   0.98122367]
 [ 0.52820055  0.52820055  0.52820055  0.9905506   0.98434867  0.55327648
   0.98122367]
 [ 0.52820055  0.52820055  0.59082837  0.98744963  0.98122367  0.67394231
   0.98122367]
 [ 0.52820055  0.52820055  0.95934562  0.98122367  0.98122367  0.97965965
   0.98122367]]
0.992137896825
```

Observation

We see that with degree 4 , polynomial kernel performs better than degree 3. We now check for degree 5.

In [56]:
```
#using the default settings for the polynomial kernel
scores = []

clf = svm.SVC(kernel='poly', degree = 5)
score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jobs=4).me
an()
scores.append(score)
print(scores)

C = [0.5,1,1.5,2,10,20,100]
G = [0.001,0.005,0.009,0.03,0.05,0.01,0.1]
scores = np.zeros([7,7])
for i in range(0,7):
    for j in range(0,7):
        clf = svm.SVC(C = C[i], gamma=G[j], kernel='poly', degree=5)
        score = cross_validation.cross_val_score(clf, X_18,Y_18, cv=10, n_jo
bs=4).mean()
#         score = clf.fit(X_18,Y_18).score(X_18,Y_18)
        scores[i,j] = score
print(scores)
print(np.amax(scores))
# Cvalues = np.repeat(C,7)
# Gvalues = np.tile(G,7)

# plt.contourf(np.log(Cvalues.reshape(len(C), len(C))), np.log(Gvalues.resha
pe(len(C), len(C))),
#              scores)
# plt.colorbar()
# plt.title('Test Accuracy')
# plt.xlabel('log(C)')
# plt.ylabel('log(gamma)')
# plt.show()
```

```
[0.94047161172161187]
[[ 0.52820055  0.52820055  0.52820055  0.63011561  0.96088408  0.52820055
   0.98429983]
 [ 0.52820055  0.52820055  0.52820055  0.71938034  0.96557158  0.52820055
   0.98273733]
 [ 0.52820055  0.52820055  0.52820055  0.80249199  0.97025984  0.52820055
   0.98273733]
 [ 0.52820055  0.52820055  0.52820055  0.83696848  0.97338561  0.52820055
   0.98273733]
 [ 0.52820055  0.52820055  0.52820055  0.96400908  0.98117483  0.52820055
   0.98273733]
 [ 0.52820055  0.52820055  0.52820055  0.97333677  0.98429983  0.52820055
   0.98273733]
 [ 0.52820055  0.52820055  0.56421474  0.98117483  0.98273733  0.60811508
   0.98273733]]
0.984299832112
```

Observation

We see that for degree 5 , performance decreases. So we can say that optimal performance is obtained at degree 4.