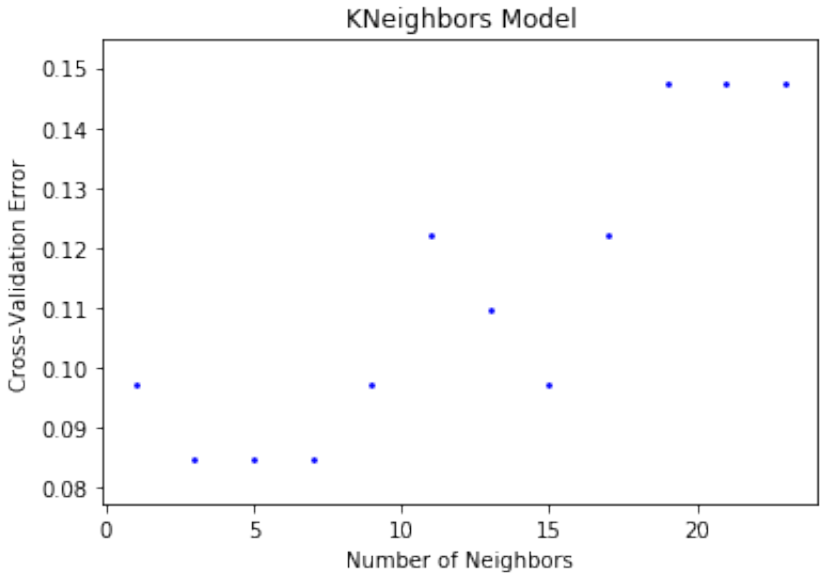
In [37]:
```
# Import required packages here (after they are installed)
import numpy as np
import matplotlib.pyplot as mp
from pylab import show

from sklearn.model_selection import cross_val_score
import math

# Import model packages
from sklearn.neighbors import KNeighborsClassifier
from sklearn.decomposition import KernelPCA
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier


# Load data. csv file should be in the same folder as the notebook for
this to work, otherwise
# give data path
data = np.loadtxt("covid-gdpMerged.csv", skiprows = 1, usecols = (2,3,
4,6,7,8,10,11), delimiter = ",")
```

MODELS FOR AVERAGE DEATHS PER MILLION

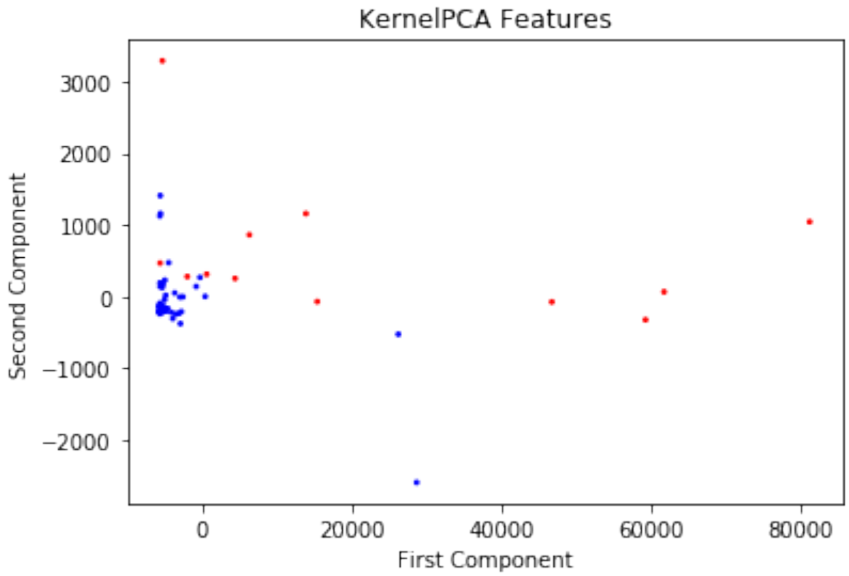
In [38]:
```
#shuffle the data and select training and test data
# np.random.seed(100)
# np.random.shuffle(data)

features = []
labels = []

for row in data:
    features.append( row[:6])
    labels.append( row[6])



#Select the proportion of data to use for training.
#Notice that we have set aside 80% of the data for testing

TRAINING_PORTION = 0.4

numTrain = int(len(features)* TRAINING_PORTION)

trainFeatures = features[:numTrain] # size = 81
testFeatures = features[numTrain:]  # size = 122
trainLabels = labels[:numTrain]
testLabels = labels[numTrain:]
```

In [39]:
```
#KNeighbors Classifier

# Run the KNeigbors Classifier on the COVID dataset
# using models of different k-nearest neighbor sizes
# to find the optimal KNeighbors model for the dataset

kneighborsNumNeighbors = range( 1, 25, 2)
kneighborsCrossVals = []


for numNeighbors in kneighborsNumNeighbors:
    model = KNeighborsClassifier( n_neighbors = numNeighbors)
    kneighborsCrossVals.append( 1 - np.average( cross_val_score( model
, trainFeatures, trainLabels, cv = 10)))




#plot the points
mp.scatter( kneighborsNumNeighbors, kneighborsCrossVals, s=3, c="b")

#specify the axes
# mp.xlim(0,50)
mp.xlabel("Number of Neighbors")
# mp.ylim(0,0.05)
mp.ylabel("Cross-Validation Error")

#label the figure
mp.title("KNeighbors Model")

#display the current graph
show()

optimalNeighbors = kneighborsNumNeighbors[ np.argmin( kneighborsCrossV
als)]
print( "Optimal KNeighbors model a value of", optimalNeighbors, "for n
_neighbors with cross validation error of", np.min( kneighborsCrossVal
s))
```

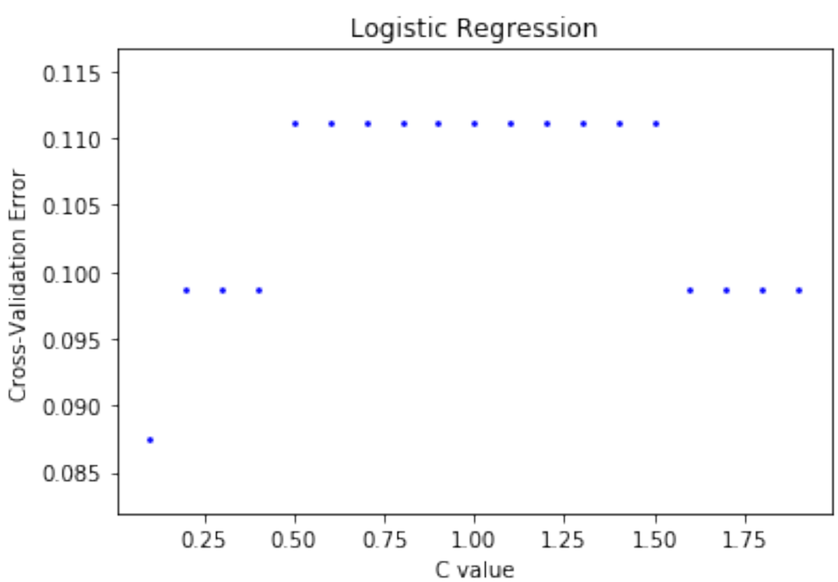



```
Optimal KNeighbors model a value of 3 for n_neighbors with cross val
idation error of 0.08472222222222214
```

In [40]:
```
#KernelPCA

# Run the KernalPCA on the COVID dataset and visualize
# the results to observe whether there are certain
# features that nicely separate the data

model = KernelPCA( n_components=2, kernel='poly', degree=1)
newFeatures = model.fit_transform( trainFeatures, trainLabels)

X = []
Y = []
kpcaTrain = []

colors = []

for index in range( len( newFeatures)):
    X.append( newFeatures[ index][ 0])
    Y.append( newFeatures[ index][ 1])
    kpcaTrain.append( [ newFeatures[ index][ 0], newFeatures[ index][1
]])

    if ( trainLabels[ index] == 0):
        colors.append("b")
    else:
        colors.append("r")


#plot the data points
mp.scatter(X, Y, s = 3, c = colors)

#specify the axes
# mp.xlim(-1,1)
mp.xlabel("First Component")
# mp.ylim(-1,1)
mp.ylabel("Second Component")
#label the figure
mp.title("KernelPCA Features")
#display the current graph
show()
```

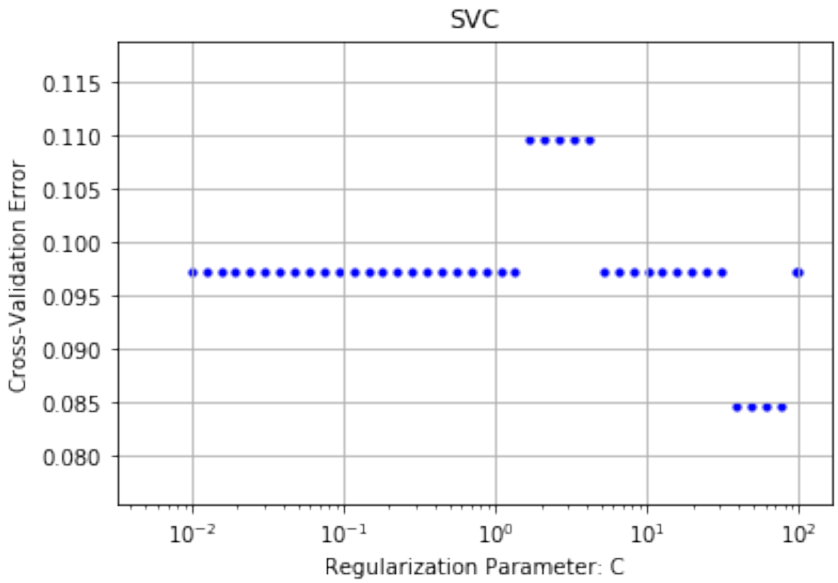

KernelPCA Features
Second Component
First Component
3000
2000
1000
0
−1000
−2000
0
20000
40000
60000
80000

In [41]:
```
#LogisticRegression

# Run the LogisticRegression model on the KernelPCA extracted features
# from the COVID dataset using different c values to find the optimal
# LogisticRegression model for the dataset

logisticRegressionStrength = []
for i in range( 1, 20):
    logisticRegressionStrength.append( i/10)

logisticRegressionCrossVals = []

for i in logisticRegressionStrength:
    model = LogisticRegression( C = i, solver = "liblinear")
    logisticRegressionCrossVals.append( np.average( 1 - cross_val_scor
e( model, kpcaTrain, trainLabels, cv = 10)))



#plot the data points
mp.scatter( logisticRegressionStrength, logisticRegressionCrossVals, s
= 3, c = "blue")

#specify the axes
# mp.xlim(-1,1)
mp.xlabel("C value")
# mp.ylim(-1,1)
mp.ylabel("Cross-Validation Error")
#label the figure
mp.title("Logistic Regression")
#display the current graph
show()
```

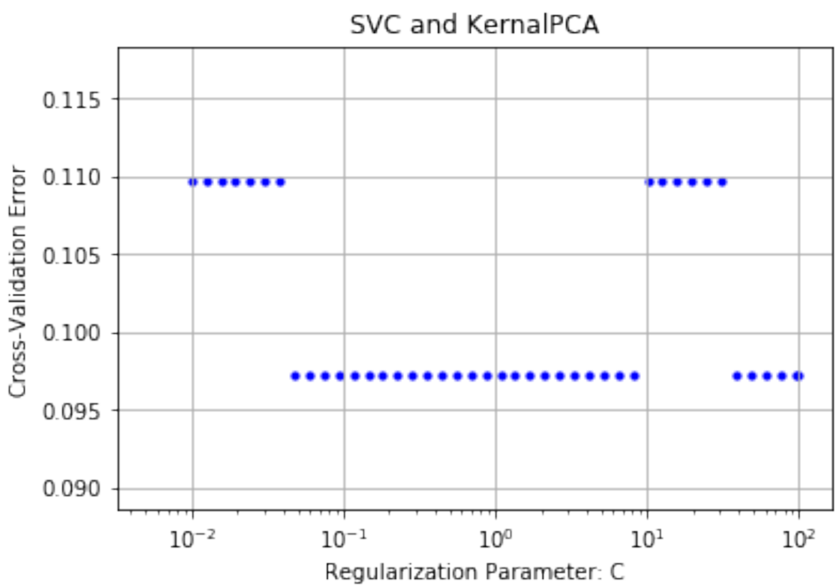


In [42]:
```
#SVM

# Run the SVC model on the features extracted from the KernelPCA
# and use cross_val_score to determine how effectively the data
# was separated in the kpcaTrain model

svcRegularization = []
for i in range( 1, 43):
    svcRegularization.append( np.power( 1.25, i) / 125)
svcRegularization.append( 100)

svcCrossVals = []
svcCrossValsKPCA = []

for i in svcRegularization:
    model = SVC( kernel = "poly", C = i)
    svcCrossVals.append( np.average( 1 - cross_val_score( model, train
Features, trainLabels, cv = 10)))
    svcCrossValsKPCA.append( np.average( 1 - cross_val_score( model, k
pcaTrain, trainLabels, cv = 10)))


#plot the points
mp.scatter( svcRegularization, svcCrossVals, s=10, c="b")

#specify the axes
mp.xlabel("Regularization Parameter: C")
mp.ylabel("Cross-Validation Error")

mp.xscale("log")
mp.grid()

#label the figure
mp.title("SVC")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( svcCrossVals), "at c
=", svcRegularization[ np.argmin( svcCrossVals)])

#plot the points
mp.scatter( svcRegularization, svcCrossValsKPCA, s=10, c="b")

#specify the axes
mp.xlabel("Regularization Parameter: C")
mp.ylabel("Cross-Validation Error")

mp.xscale("log")
```

```
mp.grid()

#label the figure
mp.title("SVC and KernalPCA")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( svcCrossValsKPCA), "
at c =", svcRegularization[ np.argmin( svcCrossValsKPCA)])
```

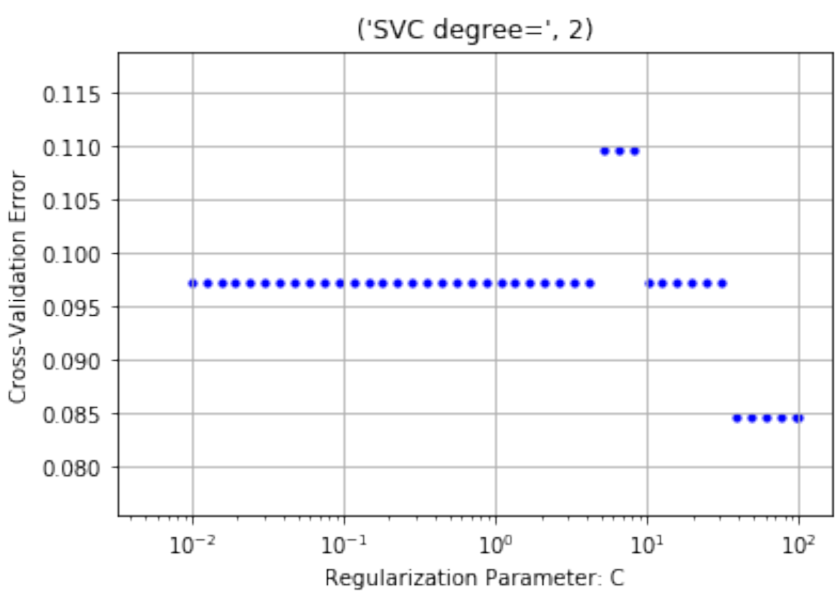



```
Lowest Cross-Validation Error = 0.08472222222222223 at c = 38.518598
887744716
```

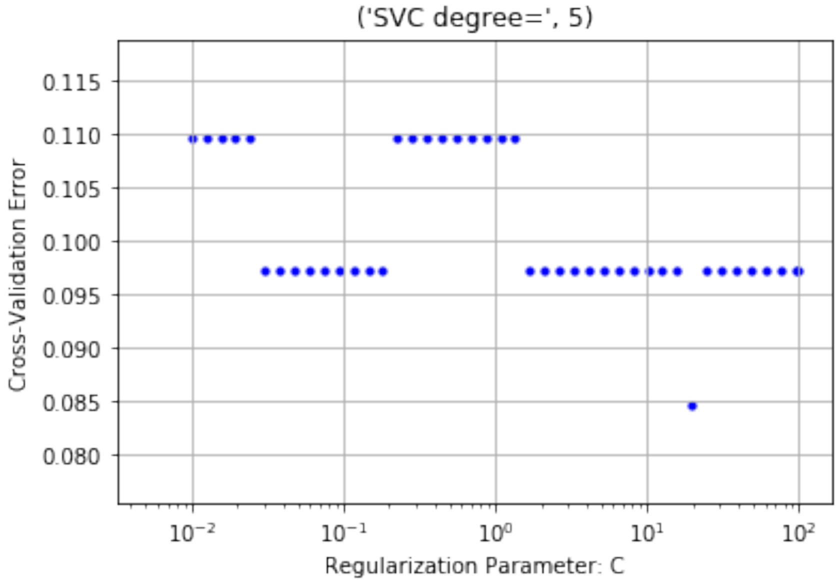



```
Lowest Cross-Validation Error = 0.09722222222222222 at c = 0.0476837
158203125
```

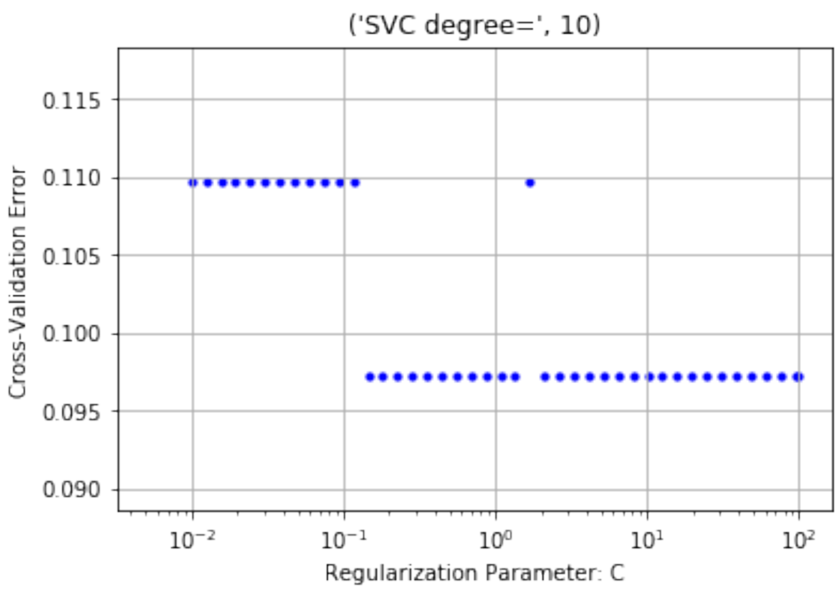
In [43]:
```
cross_val_scores = [[],[],[],[]]

degrees = [2, 5, 10, 20]

for x in range( len( degrees)):
    for i in svcRegularization:
        model = SVC( C = i, kernel = "poly", degree = degrees[ x])
        cross_val_scores[ x].append( np.average(1 - cross_val_score( m
odel, trainFeatures, trainLabels, cv = 10)))

for i in range( len( degrees)):
    #plot the points
    mp.scatter( svcRegularization, cross_val_scores[ i], s=10, c="b")

    #specify the axes
    mp.xlabel("Regularization Parameter: C")
    mp.ylabel("Cross-Validation Error")

    mp.xscale("log")
    mp.grid()

    #label the figure
    title = "SVC degree=", degrees[ i]
    mp.title(title)

    #display the current graph
    show()
    print( "Minimal Cross-Validation Error with kernel degree =", degr
ees[ i], "is at c =", svcRegularization[ np.argmin( cross_val_scores[
i])])
    print( "Cross-Validation Error =", np.min( cross_val_scores[ i]))
```



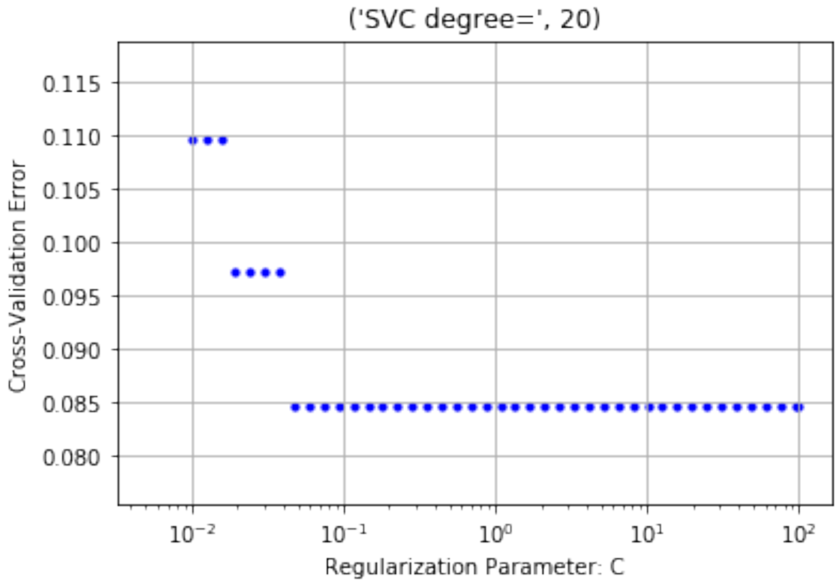
```
Minimal Cross-Validation Error with kernel degree = 2 is at c = 38.5
18598887744716
Cross-Validation Error = 0.08472222222222223
```




```
Minimal Cross-Validation Error with kernel degree = 5 is at c = 19.7
21522630525296
Cross-Validation Error = 0.08472222222222223
```

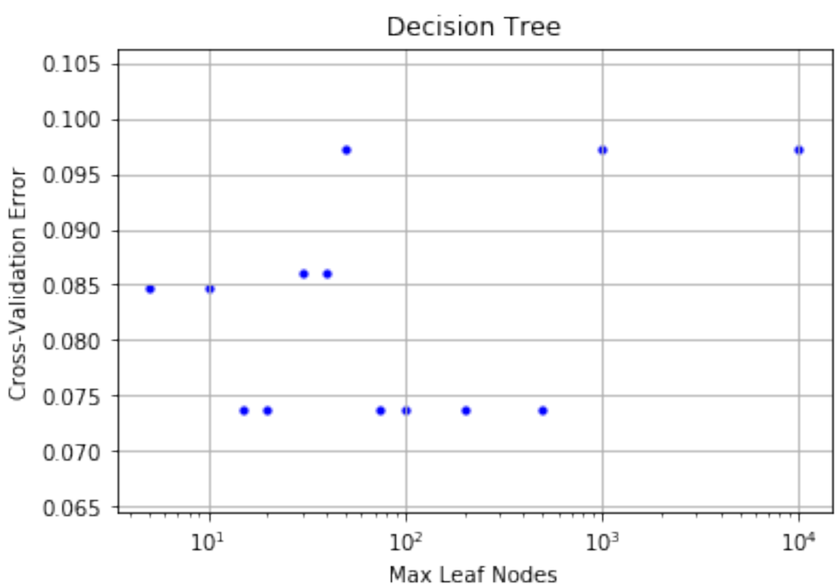



```
Minimal Cross-Validation Error with kernel degree = 10 is at c = 0.1
4551915228366852
Cross-Validation Error = 0.09722222222222222
```




```
Minimal Cross-Validation Error with kernel degree = 20 is at c = 0.0
476837158203125
Cross-Validation Error = 0.08472222222222223
```

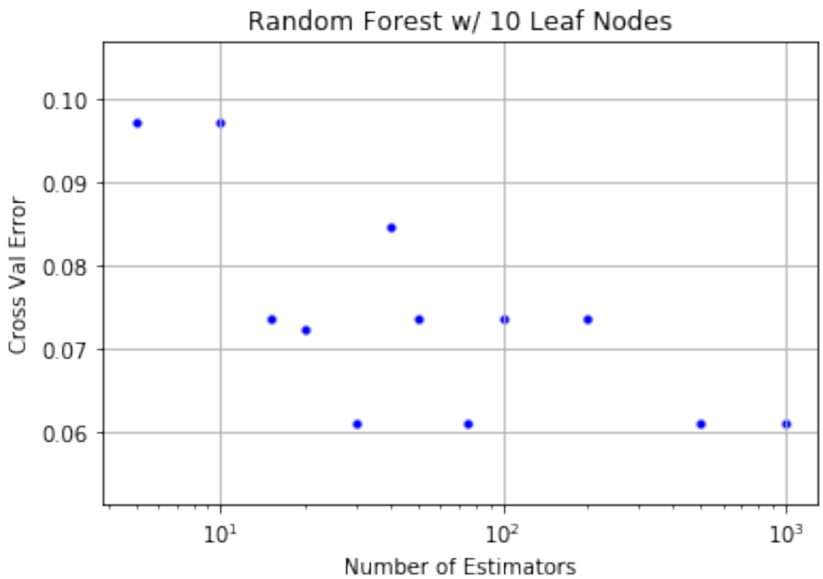
In [44]:
```
maxLeafNodes = [5,10,15,20,30,40,50,75,100,200,500,1000,10000]
decisionTreeCrossVals = []

for i in maxLeafNodes:
    model = DecisionTreeClassifier(criterion = "entropy", max_leaf_nod
es = i)
    decisionTreeCrossVals.append( np.average( 1 - cross_val_score( mod
el, kpcaTrain, trainLabels, cv = 10)))

#plot the points
mp.scatter(maxLeafNodes,decisionTreeCrossVals,s=10,c="b")

#specify the axes
mp.xlabel("Max Leaf Nodes")
mp.ylabel("Cross-Validation Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Decision Tree")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( decisionTreeCrossVal
s), "at max_leaf_nodes =", maxLeafNodes[ np.argmin( decisionTreeCrossV
als)])
```

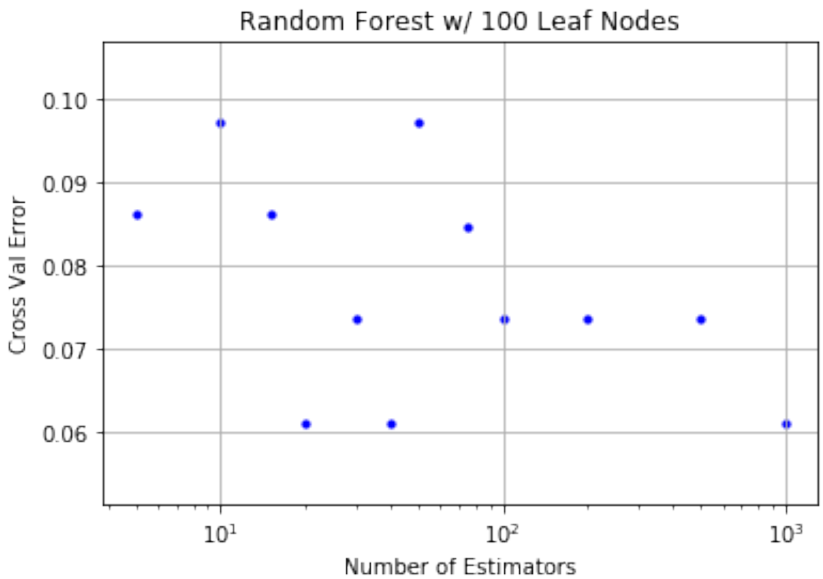


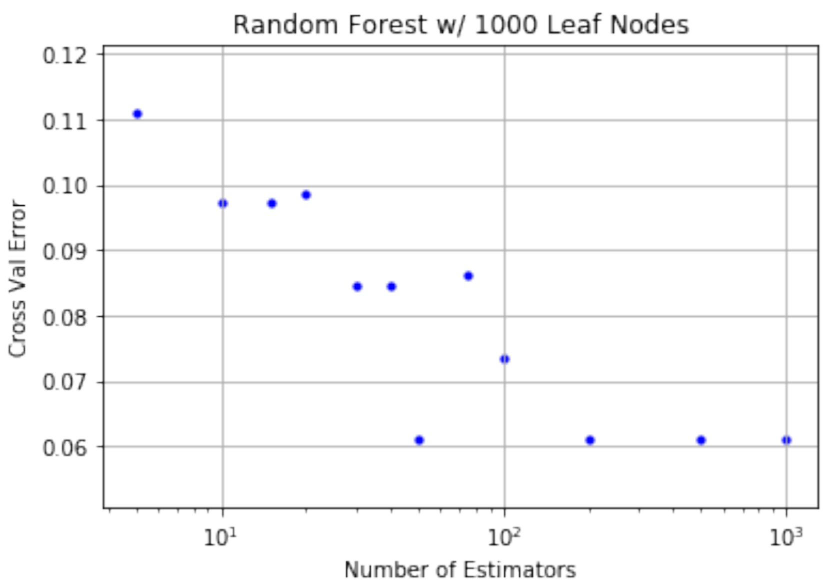

```
Lowest Cross-Validation Error = 0.07361111111111111 at max_leaf_node
s = 15
```

In [45]:
```
from sklearn.ensemble import RandomForestClassifier

maxLeafNodes = [10, 100, 1000]
numEstimators = [5,10,15,20,30,40,50,75,100,200,500,1000]

errors = [ [], [], [] ]

i = 0
for nNodes in maxLeafNodes:
    for nEstimators in numEstimators:
        model = RandomForestClassifier( n_estimators = nEstimators, ma
x_leaf_nodes = nNodes, n_jobs = -1)
        errors[i].append( np.average( 1 - cross_val_score( model, kpca
Train, trainLabels, cv = 10)))
    i+=1


#plot the points
mp.scatter(numEstimators,errors[0],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 10 Leaf Nodes")

#display the current graph
show()


#plot the points
mp.scatter(numEstimators,errors[1],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 100 Leaf Nodes")

#display the current graph
show()
```

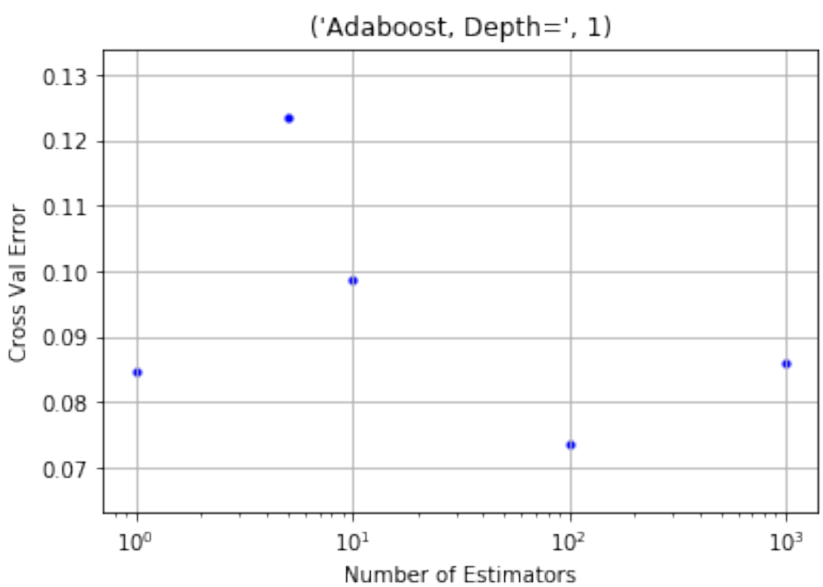
```
#plot the points
mp.scatter(numEstimators,errors[2],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 1000 Leaf Nodes")

#display the current graph
show()
```

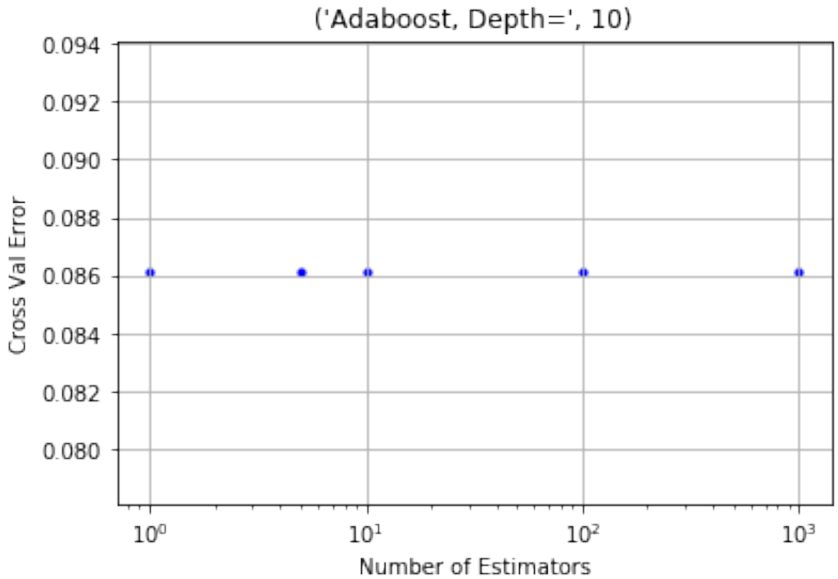


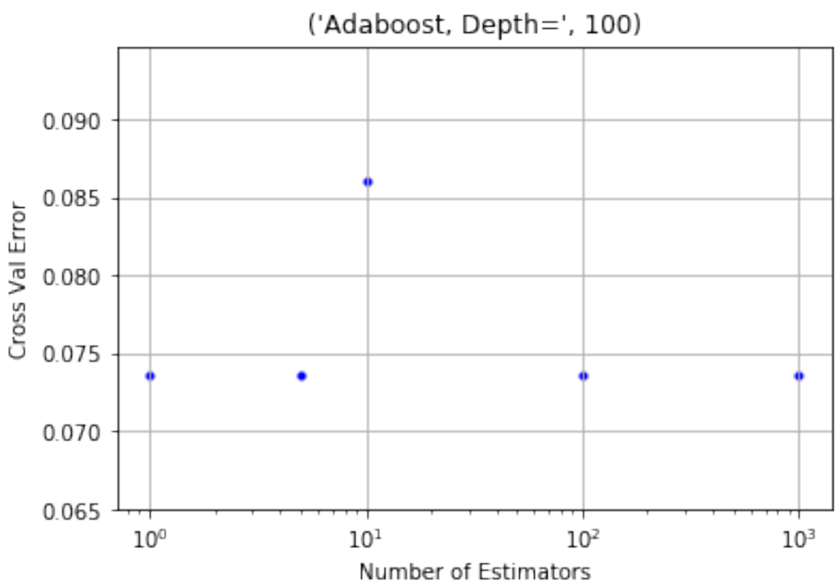

Random Forest w/ 100 Leaf Nodes
Cross Val Error
0.10
0.09
0.08
0.07
0.06
10¹
10²
10³
Number of Estimators


Random Forest w/ 1000 Leaf Nodes
Cross Val Error
0.12
0.11
0.10
0.09
0.08
0.07
0.06
10¹
10²
10³
Number of Estimators

In [46]:
```
numEstimators = [1,5,10,100,1000]
maxDepths = [1, 10, 100]

for maxDepth in maxDepths:
    errors = []
    for i in numEstimators:
        model = AdaBoostClassifier( base_estimator = DecisionTreeClass
ifier( max_depth = maxDepth), n_estimators = i)
        errors.append( np.average( 1 - cross_val_score( model, kpcaTra
in, trainLabels, cv = 10)))

    #plot the points
    mp.scatter(numEstimators,errors,s=10,c="b")

    #specify the axes
    mp.xlabel("Number of Estimators")
    mp.ylabel("Cross Val Error")

    mp.grid()
    mp.xscale("log")

    #label the figure
    title = "Adaboost, Depth=", maxDepth
    mp.title(title)

    #display the current graph
    show()
```




('Adaboost, Depth=', 10)
0.094
0.092
0.090
0.088
0.086
0.084
0.082
0.080
Cross Val Error
10^0
10^1
10^2
10^3
Number of Estimators


('Adaboost, Depth=', 100)
0.090
0.085
0.080
0.075
0.070
0.065
Cross Val Error
10^0
10^1
10^2
10^3
Number of Estimators

In [47]:
```
# Build simpleTest to test the optimal models

model = KernelPCA( n_components=2, kernel='poly', degree=1)
newFeatures = model.fit_transform( testFeatures, testLabels)

X = []
Y = []
simpleTest = []

colors = []

for index in range( len( newFeatures)):
    X.append( newFeatures[ index][ 0])
    Y.append( newFeatures[ index][ 1])
    simpleTest.append( [ newFeatures[ index][ 0], newFeatures[ index][
1]])
```

In [48]:
```
kNeighborsModel = KNeighborsClassifier( n_neighbors = optimalNeighbors
)
logisticRegressionModel = LogisticRegression( C = 0.1, solver = "libli
near")
svmModel = SVC( kernel = "poly", C = svcRegularization[ np.argmin( svc
CrossValsKPCA)])
decisionTreeModel = DecisionTreeClassifier(criterion = "entropy", max_
leaf_nodes = 15)
randomForestModel = RandomForestClassifier( n_estimators = 20, max_lea
f_nodes = 100, n_jobs = -1)
adaBoostModel = model = AdaBoostClassifier( base_estimator = DecisionT
reeClassifier( max_depth = 10), n_estimators = 10)

kNeighborsModel.fit( kpcaTrain, trainLabels)
logisticRegressionModel.fit( kpcaTrain, trainLabels)
svmModel.fit( kpcaTrain, trainLabels)
decisionTreeModel.fit( kpcaTrain, trainLabels)
randomForestModel.fit( kpcaTrain, trainLabels)
adaBoostModel.fit( kpcaTrain, trainLabels)

models = [ "K-Neighbors", "Logistic Regression", "Support Vector Machi
ne", "Decision Tree", "Random Forest", "AdaBoost"]
errors = []
errors.append( 1 - kNeighborsModel.score( simpleTest, testLabels))
errors.append( 1 - logisticRegressionModel.score( simpleTest, testLabe
ls))
errors.append( 1 - svmModel.score( simpleTest, testLabels))
errors.append( 1 - decisionTreeModel.score( simpleTest, testLabels))
errors.append( 1 - randomForestModel.score( simpleTest, testLabels))
errors.append( 1 - adaBoostModel.score( simpleTest, testLabels))
```

In [49]:
```
confidences = [0.75, 0.95, 0.99]

for i in range( len( confidences)):
    print( "Markov Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", 1 - errors[ j] / ( 1 - conf
idences[ i]))
    print()
print()


for i in range( len( confidences)):
    print( "Chebyshev Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", errors[ j], "+-", math.sqrt
( 1/( 4*len( testLabels)*( 1 - confidences[ i]))))
    print()
print()


for i in range( len( confidences)):
    print( "Hoeffding Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", errors[ j], "+-", math.sqrt
( math.log( 1 - confidences[ i])/( -2*len( testLabels))))
    print()
print()
```

```
Markov Bound at 75.0 %
         K-Neighbors Bound: 0.540983606557377
         Logistic Regression Bound: 0.6065573770491803
         Support Vector Machine Bound: 0.540983606557377
         Decision Tree Bound: 0.6065573770491803
         Random Forest Bound: 0.639344262295082
         AdaBoost Bound: 0.6065573770491803

Markov Bound at 95.0 %
         K-Neighbors Bound: -1.2950819672131129
         Logistic Regression Bound: -0.9672131147540965
         Support Vector Machine Bound: -1.2950819672131129
         Decision Tree Bound: -0.9672131147540965
         Random Forest Bound: -0.8032786885245884
         AdaBoost Bound: -0.9672131147540965

Markov Bound at 99.0 %
         K-Neighbors Bound: -10.475409836065566
```

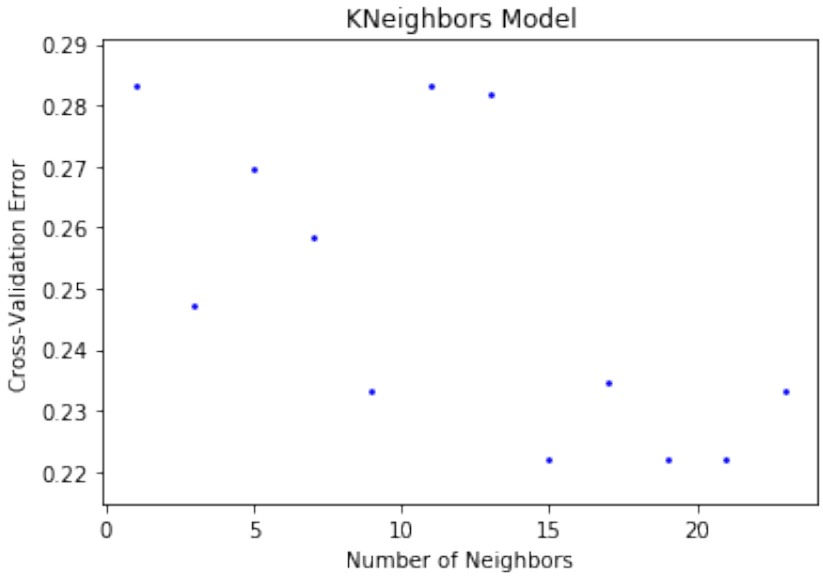
```
        Logistic Regression Bound: -8.836065573770483
        Support Vector Machine Bound: -10.475409836065566
        Decision Tree Bound: -8.836065573770483
        Random Forest Bound: -8.016393442622942
        AdaBoost Bound: -8.836065573770483


Chebyshev Bound at 75.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.090535746042518
53
        Logistic Regression Bound: 0.09836065573770492 +- 0.0905357
4604251853
        Support Vector Machine Bound: 0.11475409836065575 +- 0.0905
3574604251853
        Decision Tree Bound: 0.09836065573770492 +- 0.0905357460425
1853
        Random Forest Bound: 0.0901639344262295 +- 0.09053574604251
853
        AdaBoost Bound: 0.09836065573770492 +- 0.09053574604251853

Chebyshev Bound at 95.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.202444082544728
93
        Logistic Regression Bound: 0.09836065573770492 +- 0.2024440
8254472893
        Support Vector Machine Bound: 0.11475409836065575 +- 0.2024
4408254472893
        Decision Tree Bound: 0.09836065573770492 +- 0.2024440825447
2893
        Random Forest Bound: 0.0901639344262295 +- 0.20244408254472
893
        AdaBoost Bound: 0.09836065573770492 +- 0.20244408254472893

Chebyshev Bound at 99.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.452678730212592
5
        Logistic Regression Bound: 0.09836065573770492 +- 0.4526787
302125925
        Support Vector Machine Bound: 0.11475409836065575 +- 0.4526
787302125925
        Decision Tree Bound: 0.09836065573770492 +- 0.4526787302125
925
        Random Forest Bound: 0.0901639344262295 +- 0.45267873021259
25
        AdaBoost Bound: 0.09836065573770492 +- 0.4526787302125925


Hoeffding Bound at 75.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.075375952842301
09
```

```
        Logistic Regression Bound: 0.09836065573770492 +- 0.0753759
5284230109
        Support Vector Machine Bound: 0.11475409836065575 +- 0.0753
7595284230109
        Decision Tree Bound: 0.09836065573770492 +- 0.0753759528423
0109
        Random Forest Bound: 0.0901639344262295 +- 0.07537595284230
109
        AdaBoost Bound: 0.09836065573770492 +- 0.07537595284230109

Hoeffding Bound at 95.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.110804292719448
99
        Logistic Regression Bound: 0.09836065573770492 +- 0.1108042
9271944899
        Support Vector Machine Bound: 0.11475409836065575 +- 0.1108
0429271944899
        Decision Tree Bound: 0.09836065573770492 +- 0.1108042927194
4899
        Random Forest Bound: 0.0901639344262295 +- 0.11080429271944
899
        AdaBoost Bound: 0.09836065573770492 +- 0.11080429271944899

Hoeffding Bound at 99.0 %
        K-Neighbors Bound: 0.11475409836065575 +- 0.137381397224041
52
        Logistic Regression Bound: 0.09836065573770492 +- 0.1373813
9722404152
        Support Vector Machine Bound: 0.11475409836065575 +- 0.1373
8139722404152
        Decision Tree Bound: 0.09836065573770492 +- 0.1373813972240
4152
        Random Forest Bound: 0.0901639344262295 +- 0.13738139722404
152
        AdaBoost Bound: 0.09836065573770492 +- 0.13738139722404152
```

MODELS FOR MEDIAN DEATHS PER MILLION

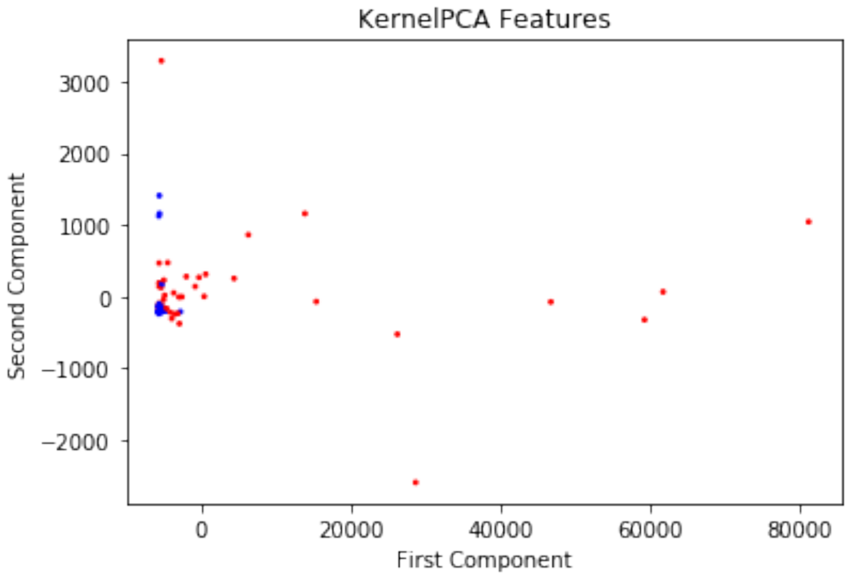
In [50]:
```
#shuffle the data and select training and test data
# np.random.seed(100)
# np.random.shuffle(data)

features = []
labels = []

for row in data:
    features.append( row[:6])
    labels.append( row[7])



#Select the proportion of data to use for training.
#Notice that we have set aside 80% of the data for testing

TRAINING_PORTION = 0.4

numTrain = int(len(features)* TRAINING_PORTION)

trainFeatures = features[:numTrain] # size = 81
testFeatures = features[numTrain:]  # size = 122
trainLabels = labels[:numTrain]
testLabels = labels[numTrain:]
```

In [51]:
```
#KNeighbors Classifier

# Run the KNeigbors Classifier on the COVID dataset
# using models of different k-nearest neighbor sizes
# to find the optimal KNeighbors model for the dataset

kneighborsNumNeighbors = range( 1, 25, 2)
kneighborsCrossVals = []


for numNeighbors in kneighborsNumNeighbors:
    model = KNeighborsClassifier( n_neighbors = numNeighbors)
    kneighborsCrossVals.append( 1 - np.average( cross_val_score( model
, trainFeatures, trainLabels, cv = 10)))



#plot the points
mp.scatter( kneighborsNumNeighbors, kneighborsCrossVals, s=3, c="b")

#specify the axes
# mp.xlim(0,50)
mp.xlabel("Number of Neighbors")
# mp.ylim(0,0.05)
mp.ylabel("Cross-Validation Error")

#label the figure
mp.title("KNeighbors Model")

#display the current graph
show()

optimalNeighbors = kneighborsNumNeighbors[ np.argmin( kneighborsCrossV
als)]
print( "Optimal KNeighbors model a value of", optimalNeighbors, "for n
_neighbors with cross validation error of", np.min( kneighborsCrossVal
s))
```



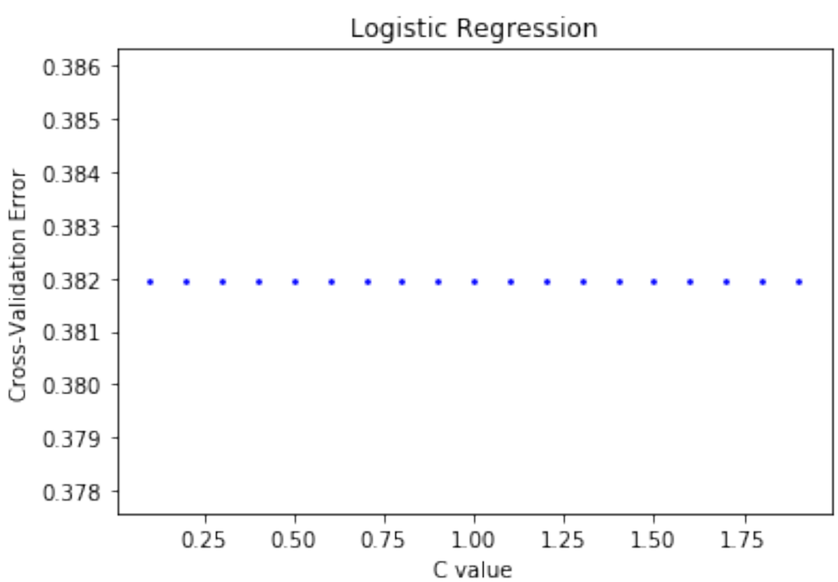

```
Optimal KNeighbors model a value of 15 for n_neighbors with cross va
lidation error of 0.2222222222222222
```

In [52]:
```
#KernelPCA

# Run the KernalPCA on the COVID dataset and visualize
# the results to observe whether there are certain
# features that nicely separate the data

model = KernelPCA( n_components=2, kernel='poly', degree=1)
newFeatures = model.fit_transform( trainFeatures, trainLabels)

X = []
Y = []
kpcaTrain = []

colors = []

for index in range( len( newFeatures)):
    X.append( newFeatures[ index][ 0])
    Y.append( newFeatures[ index][ 1])
    kpcaTrain.append( [ newFeatures[ index][ 0], newFeatures[ index][1
]])

    if ( trainLabels[ index] == 0):
        colors.append("b")
    else:
        colors.append("r")


#plot the data points
mp.scatter(X, Y, s = 3, c = colors)

#specify the axes
# mp.xlim(-1,1)
mp.xlabel("First Component")
# mp.ylim(-1,1)
mp.ylabel("Second Component")
#label the figure
mp.title("KernelPCA Features")
#display the current graph
show()
```


KernelPCA Features
3000
2000
1000
0
−1000
−2000
Second Component
0
20000
40000
60000
80000
First Component

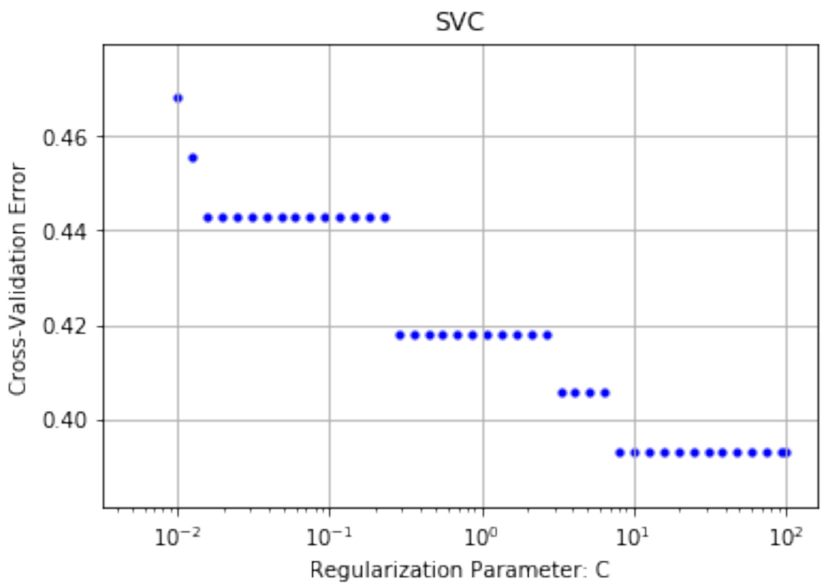
In [53]:

```
#LogisticRegression

# Run the LogisticRegression model on the KernelPCA extracted features
# from the COVID dataset using different c values to find the optimal
# LogisticRegression model for the dataset

logisticRegressionStrength = []
for i in range( 1, 20):
    logisticRegressionStrength.append( i/10)

logisticRegressionCrossVals = []

for i in logisticRegressionStrength:
    model = LogisticRegression( C = i, solver = "liblinear")
    logisticRegressionCrossVals.append( np.average( 1 - cross_val_scor
e( model, kpcaTrain, trainLabels, cv = 10)))



#plot the data points
mp.scatter( logisticRegressionStrength, logisticRegressionCrossVals, s
= 3, c = "blue")

#specify the axes
# mp.xlim(-1,1)
mp.xlabel("C value")
# mp.ylim(-1,1)
mp.ylabel("Cross-Validation Error")
#label the figure
mp.title("Logistic Regression")
#display the current graph
show()
```

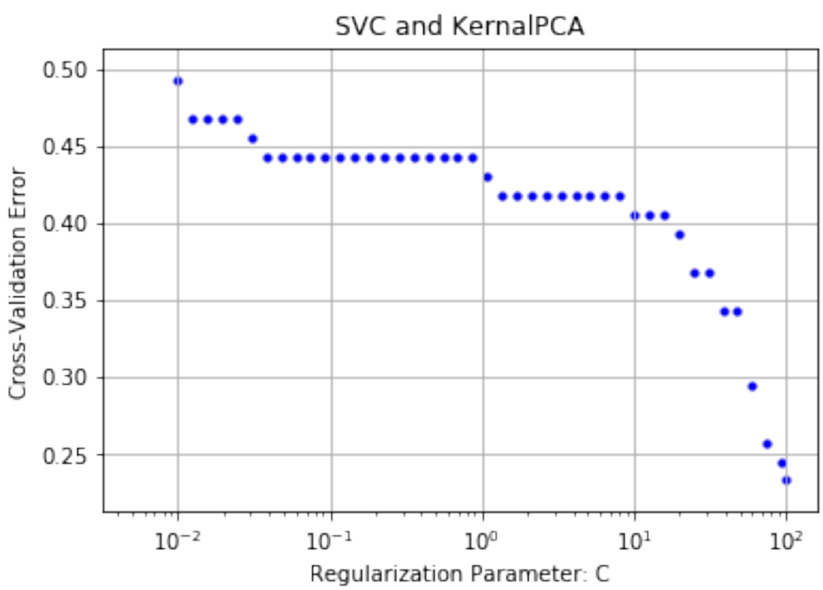


In [54]:
```
#SVM

# Run the SVC model on the features extracted from the KernelPCA
# and use cross_val_score to determine how effectively the data
# was separated in the kpcaTrain model

svcRegularization = []
for i in range( 1, 43):
    svcRegularization.append( np.power( 1.25, i) / 125)
svcRegularization.append( 100)

svcCrossVals = []
svcCrossValsKPCA = []

for i in svcRegularization:
    model = SVC( kernel = "poly", C = i)
    svcCrossVals.append( np.average( 1 - cross_val_score( model, train
Features, trainLabels, cv = 10)))
    svcCrossValsKPCA.append( np.average( 1 - cross_val_score( model, k
pcaTrain, trainLabels, cv = 10)))


#plot the points
mp.scatter( svcRegularization, svcCrossVals, s=10, c="b")

#specify the axes
mp.xlabel("Regularization Parameter: C")
mp.ylabel("Cross-Validation Error")

mp.xscale("log")
mp.grid()

#label the figure
mp.title("SVC")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( svcCrossVals), "at c
=", svcRegularization[ np.argmin( svcCrossVals)])

#plot the points
mp.scatter( svcRegularization, svcCrossValsKPCA, s=10, c="b")

#specify the axes
mp.xlabel("Regularization Parameter: C")
mp.ylabel("Cross-Validation Error")

mp.xscale("log")
```

```
mp.grid()

#label the figure
mp.title("SVC and KernalPCA")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( svcCrossValsKPCA), " at c =", svcRegularization[ np.argmin( svcCrossValsKPCA)])
```

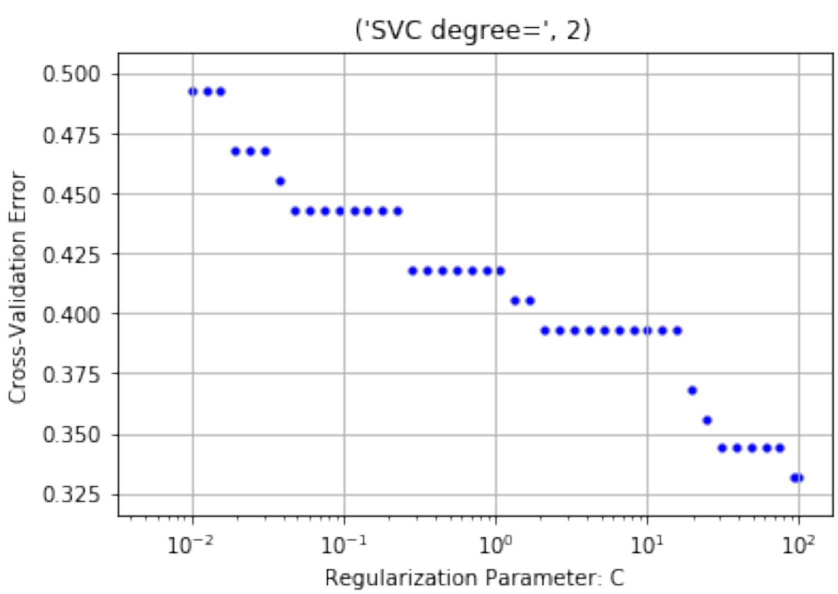



```
Lowest Cross-Validation Error = 0.39305555555555555 at c = 8.0779356
6946316
```

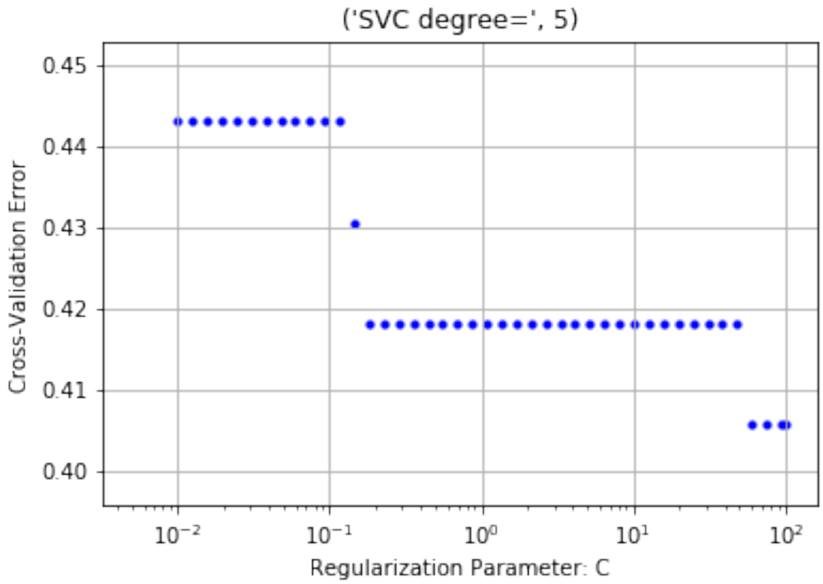



```
Lowest Cross-Validation Error = 0.23333333333333334 at c = 100
```

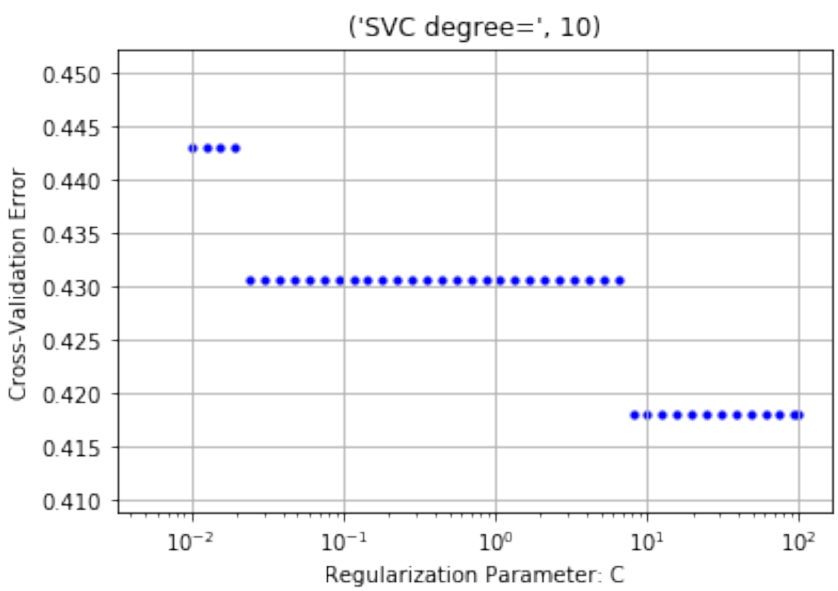
In [55]:
```
cross_val_scores = [[],[],[],[]]

degrees = [2, 5, 10, 20]

for x in range( len( degrees)):
    for i in svcRegularization:
        model = SVC( C = i, kernel = "poly", degree = degrees[ x])
        cross_val_scores[ x].append( np.average(1 - cross_val_score( m
odel, trainFeatures, trainLabels, cv = 10)))

for i in range( len( degrees)):
    #plot the points
    mp.scatter( svcRegularization, cross_val_scores[ i], s=10, c="b")

    #specify the axes
    mp.xlabel("Regularization Parameter: C")
    mp.ylabel("Cross-Validation Error")

    mp.xscale("log")
    mp.grid()

    #label the figure
    title = "SVC degree=", degrees[ i]
    mp.title(title)

    #display the current graph
    show()
    print( "Minimal Cross-Validation Error with kernel degree =", degr
ees[ i], "is at c =", svcRegularization[ np.argmin( cross_val_scores[
i])])
    print( "Cross-Validation Error =", np.min( cross_val_scores[ i]))
```



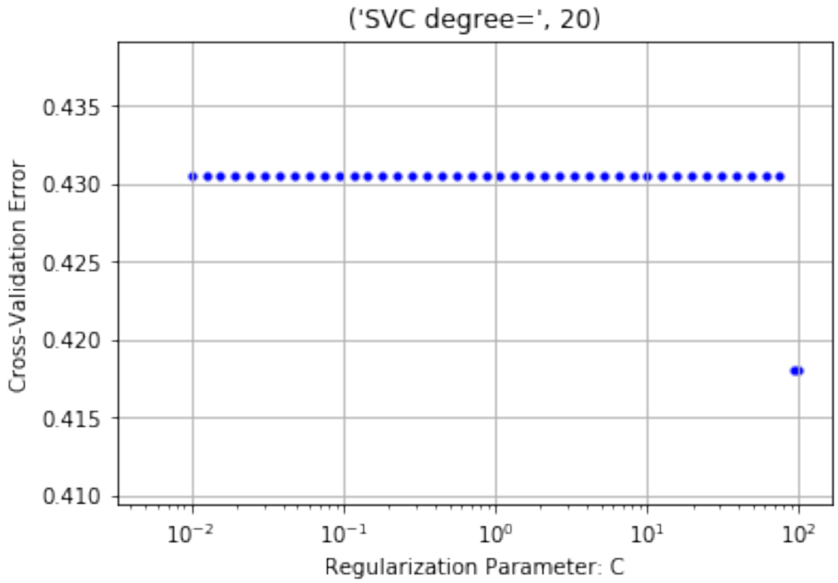
```
Minimal Cross-Validation Error with kernel degree = 2 is at c = 94.0
3954806578301
Cross-Validation Error = 0.3319444444444445
```




```
Minimal Cross-Validation Error with kernel degree = 5 is at c = 60.1
85310762101125
Cross-Validation Error = 0.40555555555555556
```

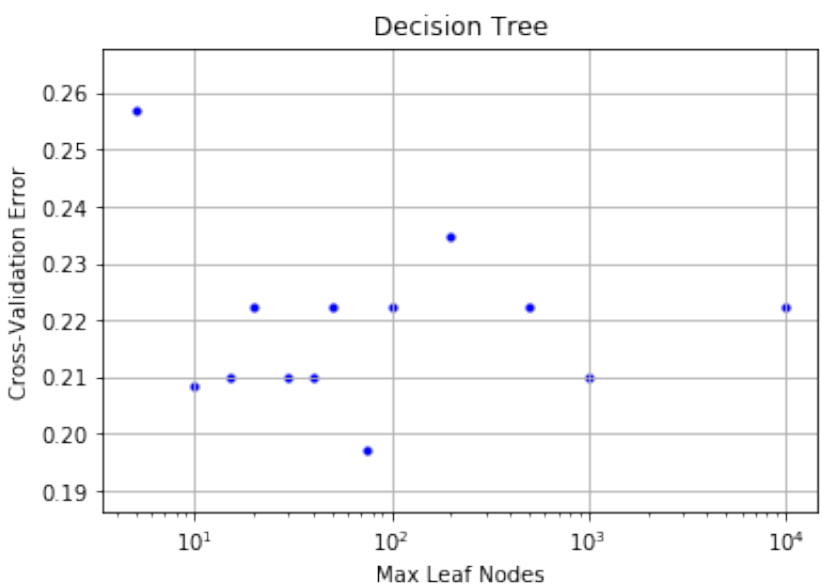



```
Minimal Cross-Validation Error with kernel degree = 10 is at c = 8.0
7793566946316
Cross-Validation Error = 0.4180555555555555
```




```
Minimal Cross-Validation Error with kernel degree = 20 is at c = 94.
03954806578301
Cross-Validation Error = 0.4180555555555555
```

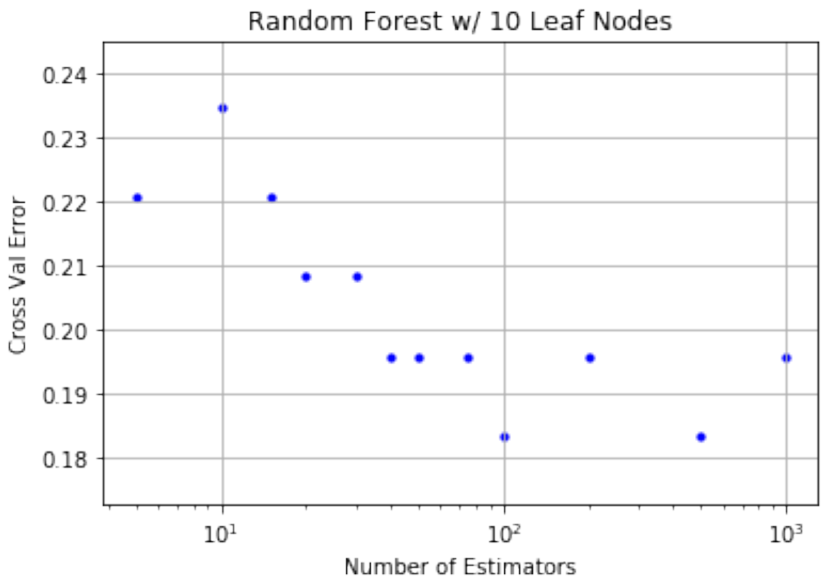
In [56]:
```
maxLeafNodes = [5,10,15,20,30,40,50,75,100,200,500,1000,10000]
decisionTreeCrossVals = []

for i in maxLeafNodes:
    model = DecisionTreeClassifier(criterion = "entropy", max_leaf_nod
es = i)
    decisionTreeCrossVals.append( np.average( 1 - cross_val_score( mod
el, kpcaTrain, trainLabels, cv = 10)))

#plot the points
mp.scatter(maxLeafNodes,decisionTreeCrossVals,s=10,c="b")

#specify the axes
mp.xlabel("Max Leaf Nodes")
mp.ylabel("Cross-Validation Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Decision Tree")

#display the current graph
show()

print( "Lowest Cross-Validation Error =", np.min( decisionTreeCrossVal
s), "at max_leaf_nodes =", maxLeafNodes[ np.argmin( decisionTreeCrossV
als)])
```

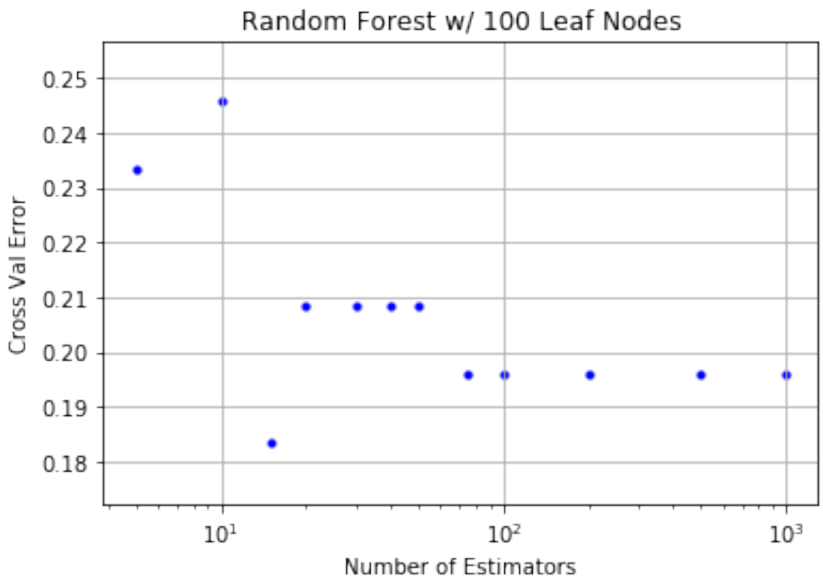


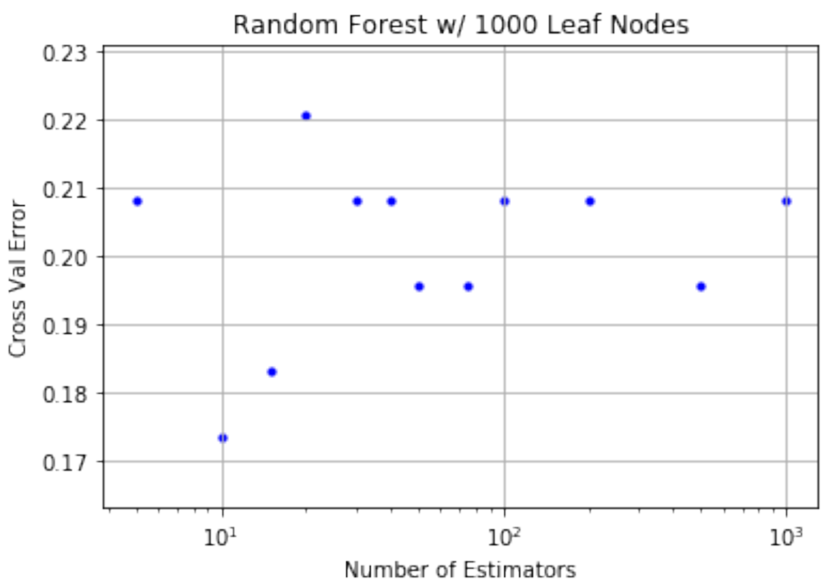

```
Lowest Cross-Validation Error = 0.19722222222222224 at max_leaf_node
s = 75
```

In [57]:
```
from sklearn.ensemble import RandomForestClassifier

maxLeafNodes = [10, 100, 1000]
numEstimators = [5,10,15,20,30,40,50,75,100,200,500,1000]

errors = [ [], [], [] ]

i = 0
for nNodes in maxLeafNodes:
    for nEstimators in numEstimators:
        model = RandomForestClassifier( n_estimators = nEstimators, ma
x_leaf_nodes = nNodes, n_jobs = -1)
        errors[i].append( np.average( 1 - cross_val_score( model, kpca
Train, trainLabels, cv = 10)))
    i+=1


#plot the points
mp.scatter(numEstimators,errors[0],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 10 Leaf Nodes")

#display the current graph
show()


#plot the points
mp.scatter(numEstimators,errors[1],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 100 Leaf Nodes")

#display the current graph
show()
```

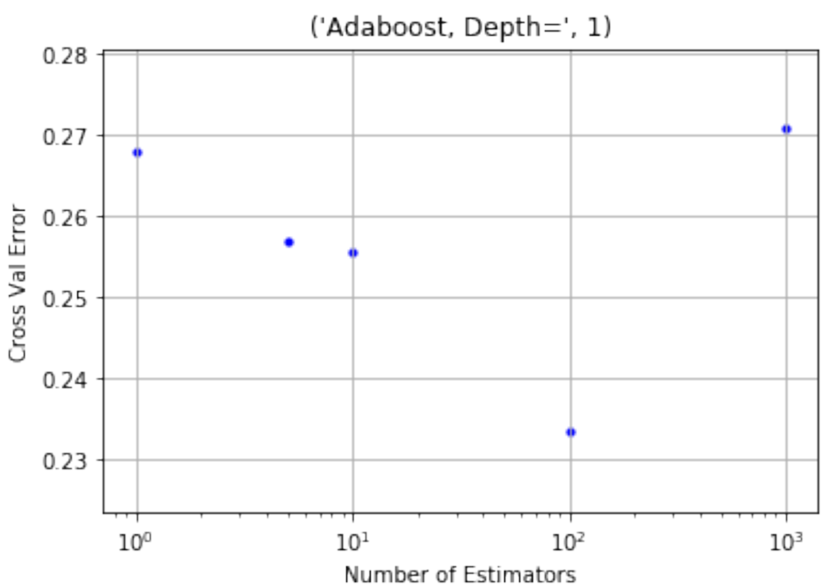
```
#plot the points
mp.scatter(numEstimators,errors[2],s=10,c="b")

#specify the axes
mp.xlabel("Number of Estimators")
mp.ylabel("Cross Val Error")

mp.grid()
mp.xscale("log")

#label the figure
mp.title("Random Forest w/ 1000 Leaf Nodes")

#display the current graph
show()
```

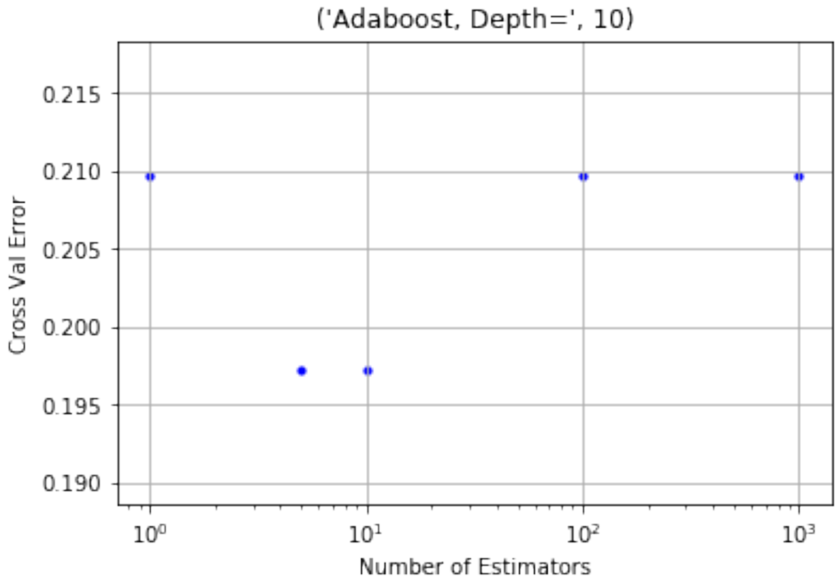


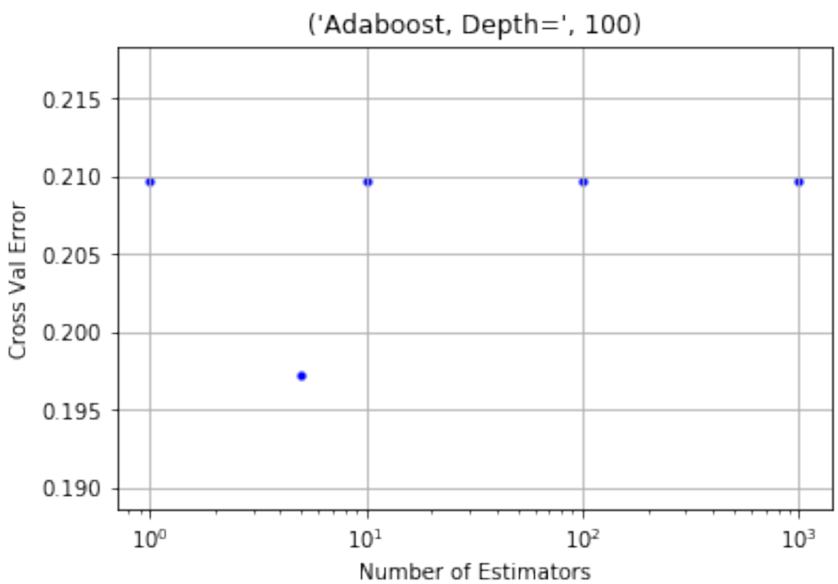

Random Forest w/ 100 Leaf Nodes
Cross Val Error
0.25
0.24
0.23
0.22
0.21
0.20
0.19
0.18
10^1
10^2
10^3
Number of Estimators


Random Forest w/ 1000 Leaf Nodes
Cross Val Error
0.23
0.22
0.21
0.20
0.19
0.18
0.17
10^1
10^2
10^3
Number of Estimators

In [58]:
```
numEstimators = [1,5,10,100,1000]
maxDepths = [1, 10, 100]

for maxDepth in maxDepths:
    errors = []
    for i in numEstimators:
        model = AdaBoostClassifier( base_estimator = DecisionTreeClass
ifier( max_depth = maxDepth), n_estimators = i)
        errors.append( np.average( 1 - cross_val_score( model, kpcaTra
in, trainLabels, cv = 10)))

    #plot the points
    mp.scatter(numEstimators,errors,s=10,c="b")

    #specify the axes
    mp.xlabel("Number of Estimators")
    mp.ylabel("Cross Val Error")

    mp.grid()
    mp.xscale("log")

    #label the figure
    title = "Adaboost, Depth=", maxDepth
    mp.title(title)

    #display the current graph
    show()
```




('Adaboost, Depth=', 10)
0.215
0.210
0.205
0.200
0.195
0.190
Cross Val Error
10^0
10^1
10^2
10^3
Number of Estimators


('Adaboost, Depth=', 100)
0.215
0.210
0.205
0.200
0.195
0.190
Cross Val Error
10^0
10^1
10^2
10^3
Number of Estimators

In [59]:
```
# Build simpleTest to test the optimal models

model = KernelPCA( n_components=2, kernel='poly', degree=1)
newFeatures = model.fit_transform( testFeatures, testLabels)

X = []
Y = []
simpleTest = []

colors = []

for index in range( len( newFeatures)):
    X.append( newFeatures[ index][ 0])
    Y.append( newFeatures[ index][ 1])
    simpleTest.append( [ newFeatures[ index][ 0], newFeatures[ index][
1]])
```

In [60]:
```
kNeighborsModel = KNeighborsClassifier( n_neighbors = optimalNeighbors
)
logisticRegressionModel = LogisticRegression( C = 0.1, solver = "libli
near")
svmModel = SVC( kernel = "poly", C = svcRegularization[ np.argmin( svc
CrossValsKPCA)])
decisionTreeModel = DecisionTreeClassifier(criterion = "entropy", max_
leaf_nodes = 75)
randomForestModel = RandomForestClassifier( n_estimators = 10, max_lea
f_nodes = 1000, n_jobs = -1)
adaBoostModel = model = AdaBoostClassifier( base_estimator = DecisionT
reeClassifier( max_depth = 10), n_estimators = 10)

kNeighborsModel.fit( kpcaTrain, trainLabels)
logisticRegressionModel.fit( kpcaTrain, trainLabels)
svmModel.fit( kpcaTrain, trainLabels)
decisionTreeModel.fit( kpcaTrain, trainLabels)
randomForestModel.fit( kpcaTrain, trainLabels)
adaBoostModel.fit( kpcaTrain, trainLabels)

models = [ "K-Neighbors", "Logistic Regression", "Support Vector Machi
ne", "Decision Tree", "Random Forest", "AdaBoost"]
errors = []
errors.append( 1 - kNeighborsModel.score( simpleTest, testLabels))
errors.append( 1 - logisticRegressionModel.score( simpleTest, testLabe
ls))
errors.append( 1 - svmModel.score( simpleTest, testLabels))
errors.append( 1 - decisionTreeModel.score( simpleTest, testLabels))
errors.append( 1 - randomForestModel.score( simpleTest, testLabels))
errors.append( 1 - adaBoostModel.score( simpleTest, testLabels))
```

In [61]:
```
confidences = [0.75, 0.95, 0.99]

for i in range( len( confidences)):
    print( "Markov Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", 1 - errors[ j] / ( 1 - conf
idences[ i]))
    print()
print()


for i in range( len( confidences)):
    print( "Chebyshev Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", errors[ j], "+-", math.sqrt
( 1/( 4*len( testLabels)*( 1 - confidences[ i]))))
    print()
print()


for i in range( len( confidences)):
    print( "Hoeffding Bound at", confidences[ i] * 100, "%")

    for j in range( len( models)):
        print( "\t", models[ j], "Bound:", errors[ j], "+-", math.sqrt
( math.log( 1 - confidences[ i])/( -2*len( testLabels))))
    print()
print()
```

```
Markov Bound at 75.0 %
         K-Neighbors Bound: -0.14754098360655732
         Logistic Regression Bound: -0.540983606557377
         Support Vector Machine Bound: -0.3114754098360657
         Decision Tree Bound: -0.5737704918032787
         Random Forest Bound: -0.21311475409836067
         AdaBoost Bound: -0.540983606557377

Markov Bound at 95.0 %
         K-Neighbors Bound: -4.737704918032781
         Logistic Regression Bound: -6.704918032786878
         Support Vector Machine Bound: -5.557377049180323
         Decision Tree Bound: -6.868852459016386
         Random Forest Bound: -5.065573770491798
         AdaBoost Bound: -6.704918032786878

Markov Bound at 99.0 %
         K-Neighbors Bound: -27.688524590163908
```

```
        Logistic Regression Bound: -37.52459016393439
        Support Vector Machine Bound: -31.786885245901615
        Decision Tree Bound: -38.34426229508193
        Random Forest Bound: -29.32786885245899
        AdaBoost Bound: -37.52459016393439


Chebyshev Bound at 75.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.090535746042518
53
        Logistic Regression Bound: 0.38524590163934425 +- 0.0905357
4604251853
        Support Vector Machine Bound: 0.3278688524590164 +- 0.09053
574604251853
        Decision Tree Bound: 0.39344262295081966 +- 0.0905357460425
1853
        Random Forest Bound: 0.30327868852459017 +- 0.0905357460425
1853
        AdaBoost Bound: 0.38524590163934425 +- 0.09053574604251853

Chebyshev Bound at 95.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.202444082544728
93
        Logistic Regression Bound: 0.38524590163934425 +- 0.2024440
8254472893
        Support Vector Machine Bound: 0.3278688524590164 +- 0.20244
408254472893
        Decision Tree Bound: 0.39344262295081966 +- 0.2024440825447
2893
        Random Forest Bound: 0.30327868852459017 +- 0.2024440825447
2893
        AdaBoost Bound: 0.38524590163934425 +- 0.20244408254472893

Chebyshev Bound at 99.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.452678730212592
5
        Logistic Regression Bound: 0.38524590163934425 +- 0.4526787
302125925
        Support Vector Machine Bound: 0.3278688524590164 +- 0.45267
87302125925
        Decision Tree Bound: 0.39344262295081966 +- 0.4526787302125
925
        Random Forest Bound: 0.30327868852459017 +- 0.4526787302125
925
        AdaBoost Bound: 0.38524590163934425 +- 0.4526787302125925


Hoeffding Bound at 75.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.075375952842301
09
```

```
        Logistic Regression Bound: 0.38524590163934425 +- 0.0753759
5284230109
        Support Vector Machine Bound: 0.3278688524590164 +- 0.07537
595284230109
        Decision Tree Bound: 0.39344262295081966 +- 0.0753759528423
0109
        Random Forest Bound: 0.30327868852459017 +- 0.0753759528423
0109
        AdaBoost Bound: 0.38524590163934425 +- 0.07537595284230109

Hoeffding Bound at 95.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.110804292719448
99
        Logistic Regression Bound: 0.38524590163934425 +- 0.1108042
9271944899
        Support Vector Machine Bound: 0.3278688524590164 +- 0.11080
429271944899
        Decision Tree Bound: 0.39344262295081966 +- 0.1108042927194
4899
        Random Forest Bound: 0.30327868852459017 +- 0.1108042927194
4899
        AdaBoost Bound: 0.38524590163934425 +- 0.11080429271944899

Hoeffding Bound at 99.0 %
        K-Neighbors Bound: 0.28688524590163933 +- 0.137381397224041
52
        Logistic Regression Bound: 0.38524590163934425 +- 0.1373813
9722404152
        Support Vector Machine Bound: 0.3278688524590164 +- 0.13738
139722404152
        Decision Tree Bound: 0.39344262295081966 +- 0.1373813972240
4152
        Random Forest Bound: 0.30327868852459017 +- 0.1373813972240
4152
        AdaBoost Bound: 0.38524590163934425 +- 0.13738139722404152
```

In [ ]: