In [1]:
```
import findspark
findspark.init()

from pyspark import SparkContext
sc = SparkContext(master="local[4]")

from pyspark.mllib.linalg import Vectors
from pyspark.mllib.regression import LabeledPoint

from string import split,strip

from pyspark.mllib.tree import GradientBoostedTrees, GradientBoostedTreesModel
from pyspark.mllib.tree import RandomForest, RandomForestModel
from pyspark.mllib.util import MLUtils
```

Cover Type

Classify geographical locations according to their predicted tree cover:

- **URL:** http://archive.ics.uci.edu/ml/datasets/Covertype (http://archive.ics.uci.edu/ml/datasets/Covertype)
- **Abstract:** Forest CoverType dataset
- **Data Set Description:** http://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.info (http://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.info)

In [2]:
```
#define a dictionary of cover types
CoverTypes={1.0: 'Spruce/Fir',
            2.0: 'Lodgepole Pine',
            3.0: 'Ponderosa Pine',
            4.0: 'Cottonwood/Willow',
            5.0: 'Aspen',
            6.0: 'Douglas-fir',
            7.0: 'Krummholz' }
print 'Tree Cover Types:'
CoverTypes
```

```
Tree Cover Types:
```

Out[2]:
```
{1.0: 'Spruce/Fir',
 2.0: 'Lodgepole Pine',
 3.0: 'Ponderosa Pine',
 4.0: 'Cottonwood/Willow',
 5.0: 'Aspen',
 6.0: 'Douglas-fir',
 7.0: 'Krummholz'}
```

In [3]:
```
# creating a directory called covtype, download and decompress covtype.data.gz into it

from os.path import exists
if not exists('covtype'):
    print "creating directory covtype"
    !mkdir covtype
%cd covtype
if not exists('covtype.data'):
    if not exists('covtype.data.gz'):
        print 'downloading covtype.data.gz'
        !curl -O http://archive.ics.uci.edu/ml/machine-learning-databases/covtype/covtype.data.gz
    print 'decompressing covtype.data.gz'
    !gunzip -f covtype.data.gz
```

```
!ls -l
%cd ..
```

```
/home/kophy/Github/CSE255-DSE230/Classes/HW-7/covtype
total 73408
-rw-rw-r-- 1 kophy kophy 75169317 Jun  2 20:44 covtype.data
/home/kophy/Github/CSE255-DSE230/Classes/HW-7
```

In [4]:

```
# Define the feature names
cols_txt="""
Elevation, Aspect, Slope, Horizontal_Distance_To_Hydrology,
Vertical_Distance_To_Hydrology, Horizontal_Distance_To_Roadways,
Hillshade_9am, Hillshade_Noon, Hillshade_3pm,
Horizontal_Distance_To_Fire_Points, Wilderness_Area (4 binarycolumns),
Soil_Type (40 binary columns), Cover_Type
"""
```

In [5]:

```
# Break up features that are made out of several binary features.
from string import split,strip
cols=[strip(a) for a in split(cols_txt,',')]
colDict={a:[a] for a in cols}
colDict['Soil_Type (40 binary columns)'] = ['ST_'+str(i) for i in range(40)]
colDict['Wilderness_Area (4 binarycolumns)'] = ['WA_'+str(i) for i in range(4)]
Columns=[]
for item in cols:
    Columns=Columns+colDict[item]
print Columns
```

```
['Elevation', 'Aspect', 'Slope', 'Horizontal_Distance_To_Hydrology', 'Vertical_Distance_To_Hydrolo
gy', 'Horizontal_Distance_To_Roadways', 'Hillshade_9am', 'Hillshade_Noon', 'Hillshade_3pm', 'Horiz
ontal_Distance_To_Fire_Points', 'WA_0', 'WA_1', 'WA_2', 'WA_3', 'ST_0', 'ST_1', 'ST_2', 'ST_3', 'S
T_4', 'ST_5', 'ST_6', 'ST_7', 'ST_8', 'ST_9', 'ST_10', 'ST_11', 'ST_12', 'ST_13', 'ST_14', 'ST_15'
, 'ST_16', 'ST_17', 'ST_18', 'ST_19', 'ST_20', 'ST_21', 'ST_22', 'ST_23', 'ST_24', 'ST_25', 'ST_26
', 'ST_27', 'ST_28', 'ST_29', 'ST_30', 'ST_31', 'ST_32', 'ST_33', 'ST_34', 'ST_35', 'ST_36', 'ST_3
7', 'ST_38', 'ST_39', 'Cover_Type']
```

In [6]:

```
# Have a look at the first two lines of the data file
!head -2 covtype/covtype.data
```

```
2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,1,0,0,0,0,0,0,0,0,0,0,0,0,5
2590,56,2,212,-6,390,220,235,151,6225,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,5
```

In [7]:

```
# Read the file into an RDD
# If doing this on a real cluster, you need the file to be available on all nodes, ideally in HDFS
.
path='covtype/covtype.data'
inputRDD=sc.textFile(path).cache()
inputRDD.first()
```

Out[7]:

```
u'2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,5'
```

In [8]:

```
# Transform the text RDD into an RDD of LabeledPoints
Data=inputRDD.map(lambda line: [float(strip(x)) for x in line.split(',')]).map(lambda x: (x[-1], x
[:-1]))
Data.first()
```

Out[8]:

```
(5.0,
 [2596.0,
  51.0,
  3.0,
  258.0,
  0.0,
  510.0,
  221.0,
  232.0,
  148.0,
  6279.0,
  1.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
```

```
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  1.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0,
  0.0])
```

In [9]:
```python
# count the number of examples of each type
total=Data.cache().count()
print 'total data size=',total
counts = Data.map(lambda (k,v): (k,1)).reduceByKey(lambda v1, v2: v1 + v2).collect()
counts.sort(key = lambda x:x[1],reverse = True)
print '              type (label):   percent of total'
print '---------------------------------------------------------'
print '\n'.join(['%20s (%3.1f):\t%4.2f'%(CoverTypes[a[0]],a[0],100.0*a[1]/float(total)) for a in counts])
```

```
total data size= 581012
              type (label):   percent of total
---------------------------------------------------------
      Lodgepole Pine (2.0):     48.76
          Spruce/Fir (1.0):     36.46
      Ponderosa Pine (3.0):     6.15
           Krummholz (7.0):     3.53
         Douglas-fir (6.0):     2.99
               Aspen (5.0):     1.63
   Cottonwood/Willow (4.0):     0.47
```

Making the problem binary

The implementation of BoostedGradientTrees in MLLib supports only binary problems. the `CovTYpe` problem has 7 classes. To make the problem binary we choose the `Lodgepole Pine` (label = 2.0). We therefor transform the dataset to a new dataset where the label is `1.0` is the class is `Lodgepole Pine` and is `0.0` otherwise.

In [10]:
```python
Label=2.0

def getBinaryLabel(label):
    return 1.0 if label == Label else 0.0

Data=inputRDD.map(lambda line: [float(x) for x in line.split(',')])\
    .map(lambda V:LabeledPoint(getBinaryLabel(V[-1]), V[:-1]))
```

Reducing data size

In order to see the effects of overfitting more clearly, we reduce the size of the data by a factor of 10

In [11]:
```
Data1=Data.sample(False,0.1).cache()
(trainingData,testData)=Data1.randomSplit([0.7,0.3], seed=255)

print 'Sizes: Data1=%d, trainingData=%d, testData=%d'%(Data1.count(),trainingData.cache().count(),
testData.cache().count())
```

```
Sizes: Data1=57888, trainingData=40740, testData=17148
```

In [12]:
```
counts=testData.map(lambda lp:(lp.label,1)).reduceByKey(lambda x,y:x+y).collect()
counts.sort(key=lambda x:x[1],reverse=True)
counts
```

Out[12]:
```
[(0.0, 8764), (1.0, 8384)]
```

Gradient Boosted Trees

- Following this example (http://spark.apache.org/docs/latest/mllib-ensembles.html#gradient-boosted-trees-gbts) from the mllib documentation
- pyspark.mllib.tree.GradientBoostedTrees documentation (http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.tree.GradientBoostedTrees)

Main classes and methods

- `GradientBoostedTrees` is the class that implements the learning trainClassifier,
  - It's main method is `trainClassifier(trainingData)` which takes as input a training set and generates an instance of `GradientBoostedTreesModel`
  - The main parameter from train Classifier are:
    - **data** – Training dataset: RDD of LabeledPoint. Labels should take values {0, 1}.
    - categoricalFeaturesInfo – Map storing arity of categorical features. E.g., an entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
    - **loss** – Loss function used for minimization during gradient boosting. Supported: {"logLoss" (default), "leastSquaresError", "leastAbsoluteError"}.
    - **numIterations** – Number of iterations of boosting. (default: 100)
    - **learningRate** – Learning rate for shrinking the contribution of each estimator. The learning rate should be between in the interval (0, 1]. (default: 0.1)
    - **maxDepth** – Maximum depth of the tree. E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. (default: 3)
    - **maxBins** – maximum number of bins used for splitting features (default: 32) DecisionTree requires maxBins >= max categories

- `GradientBoostedTreesModel` represents the output of the boosting process: a linear combination of classification trees. The methods supported by this class are:
  - `save(sc, path)` : save the tree to a given filename, sc is the Spark Context.
  - `load(sc,path)` : The counterpart to save - load classifier from file.
  - `predict(X)` : predict on a single datapoint (the `.features` field of a `LabeledPont`) or an RDD of datapoints.
  - `toDebugString()` : print the classifier in a human readable format.

In [16]:
```
from time import time
errors={}
for depth in [1, 3, 6, 10]:
    start=time()
    model=GradientBoostedTrees.trainClassifier(trainingData, categoricalFeaturesInfo={},
                                               maxDepth=depth, numIterations=10)
    #print model.toDebugString()
    errors[depth]={}
    dataSets={'train':trainingData,'test':testData}
    for name in dataSets.keys():  # Calculate errors on train and test sets
        data=dataSets[name]
        Predicted=model.predict(data.map(lambda x: x.features))
        LabelsAndPredictions=data.map(lambda x: x.label).zip(Predicted)
        Err = LabelsAndPredictions.filter(lambda (v,p):v != p).count()/float(data.count())
        errors[depth][name] = Err
    print depth,errors[depth],int(time()-start),'seconds'
print errors
```

```
1 {'test': 0.26831117331467225, 'train': 0.2721158566519391} 10 seconds
```

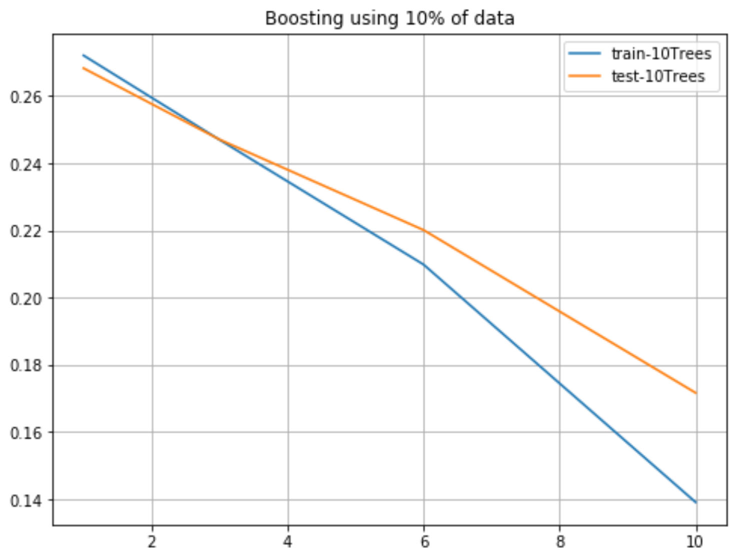
```
3 {'test': 0.24708420807091205, 'train': 0.24700540009818361} 10 seconds
6 {'test': 0.2201422906461395, 'train': 0.20986745213549338} 13 seconds
10 {'test': 0.17168182878469793, 'train': 0.13912616593028965} 23 seconds
{1: {'test': 0.26831117331467225, 'train': 0.2721158566519391}, 10: {'test': 0.17168182878469793,
'train': 0.13912616593028965}, 3: {'test': 0.24708420807091205, 'train': 0.24700540009818361}, 6:
{'test': 0.2201422906461395, 'train': 0.20986745213549338}}
```

In [17]:
```
B10 = errors
```

In [18]:
```
# Plot Train/test accuracy vs Depth of trees graph
%pylab inline
from plot_utils import *
make_figure([B10],['10Trees'],Title='Boosting using 10% of data')
```

```
Populating the interactive namespace from numpy and matplotlib

/usr/local/lib/python2.7/dist-packages/IPython/core/magics/pylab.py:161: UserWarning: pylab import
has clobbered these variables: ['e', 'split']
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```



Random Forests

- Following this example (http://spark.apache.org/docs/latest/mllib-ensembles.html#classification) from the mllib documentation
- pyspark.mllib.trees.RandomForest documentation (http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.tree.RandomForest)

trainClassifier(data, numClasses, categoricalFeaturesInfo, numTrees, featureSubsetStrategy='auto', impurity='gini', maxDepth=4, maxBins=32, seed=None)
Method to train a decision tree model for binary or multiclass classification.

Parameters:

- *data* – Training dataset: RDD of LabeledPoint. Labels should take values {0, 1, ..., numClasses-1}.
- *numClasses* – number of classes for classification.
- *categoricalFeaturesInfo* – Map storing arity of categorical features. E.g., an entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- *numTrees* – Number of trees in the random forest.
- *featureSubsetStrategy* – Number of features to consider for splits at each node. Supported: “auto” (default), “all”, “sqrt”, “log2”, “onethird”. If “auto” is set, this parameter is set based on numTrees: if numTrees == 1, set to “all”; if numTrees > 1 (forest) set to “sqrt”.
- *impurity* – Criterion used for information gain calculation. Supported values: “gini” (recommended) or “entropy”.
- *maxDepth* – Maximum depth of the tree. E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. (default: 4)
- *maxBins* – maximum number of bins used for splitting features (default: 32)
- *seed* – Random seed for bootstrapping and choosing feature subsets.

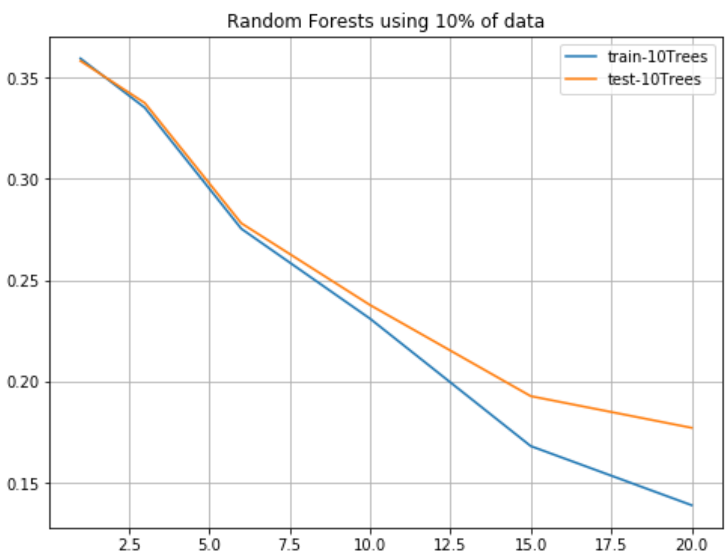
Returns:
RandomForestModel that can be used for prediction

In [20]:
```
from time import time
errors={}
```

```
for depth in [1,3,6,10,15,20]:
    start=time()
    model = RandomForest.trainClassifier(trainingData, numClasses=2, categoricalFeaturesInfo={},
                                         numTrees=10, featureSubsetStrategy="auto",
                                         impurity="gini", maxDepth=depth, maxBins=32)
    #print model.toDebugString()
    errors[depth]={}
    dataSets={'train':trainingData,'test':testData}
    for name in dataSets.keys():  # Calculate errors on train and test sets
        data=dataSets[name]
        Predicted=model.predict(data.map(lambda x: x.features))
        LabelsAndPredictions=data.map(lambda x: x.label).zip(Predicted)
        Err = LabelsAndPredictions.filter(lambda (v,p):v != p).count()/float(data.count())
        errors[depth][name]=Err
    print depth,errors[depth],int(time()-start),'seconds'
print errors
```

```
1 {'test': 0.35823419640774434, 'train': 0.35945017182130584} 2 seconds
3 {'test': 0.3374737578726382, 'train': 0.33512518409425623} 2 seconds
6 {'test': 0.27810823419640773, 'train': 0.2753804614629357} 3 seconds
10 {'test': 0.23775367389783064, 'train': 0.23102601865488465} 5 seconds
15 {'test': 0.19273384651271286, 'train': 0.16806578301423664} 8 seconds
20 {'test': 0.17710520177280148, 'train': 0.1389052528227786} 15 seconds
{1: {'test': 0.35823419640774434, 'train': 0.35945017182130584}, 3: {'test': 0.3374737578726382, '
train': 0.33512518409425623}, 6: {'test': 0.27810823419640773, 'train': 0.2753804614629357}, 10: {
'test': 0.23775367389783064, 'train': 0.23102601865488465}, 15: {'test': 0.19273384651271286, 'tra
in': 0.16806578301423664}, 20: {'test': 0.17710520177280148, 'train': 0.1389052528227786}}
```

In [21]:
```
RF_10trees = errors
# Plot Train/test accuracy vs Depth of trees graph
make_figure([RF_10trees],['10Trees'],Title='Random Forests using 10% of data')
```



Now plot B10 and RF_10trees performance curves in the same graph

In [23]:
```
make_figure([B10,RF_10trees],['B_10Trees','RF_10Trees'],Title='Boosting and Random Forests using 1
0% of data')
```