Prediction of Forest Cover-Type with Adaboost

Department of Economics,Management and
Quantitative Methods
Data Science and Economics

Machine Learning Project

Murat Aydın

September 2021

Abstract

This paper presents a scratch implementation of Adaboost algorithm using decision stumps with its accuracy metric being zero-one loss. The objective of this work is to predict the multi-class forest cover-type of the forest dataset provided on Kaggle. Implementation is done using Python language. For multi-class classification, we used One-vs-all encoding and to tune the hyper-parameter of Adaboost rounds, we applied 5-fold external cross validation. Seven different classifiers(number of classes in the dataset) are trained for each class for the multi-class prediction. Another 5-fold Cross-validation is applied to determine the size of test and training set. The highest test accuracy, around %73.5 , is achieved when 350 Adaboost rounds are used with training size being %90.

Introduction

Ensemble learning is a meta approach to Machine Learning problems that train multiple learners and combine them for the final prediction. An ensemble contains a number of predictors which are usually called base-learners. Base-learners are generally generated from the training data by a base learning algorithm like Decision Stumps, Support Vector Machines, Neural Networks. The idea is similar to that of regularization where we try to prevent over-fitting by injecting bias with a stability term. By using weak-learners(not so accurate meaning any classifier that will do at least slightly better than random coin-flipping) which results in some bias, we create a strong(accurate) classifier that is a linear combination of weak-learners.

In this project, we will use one of the rewarded techniques of ensemble methods which is **Adaboost or Adaptive Boosting**. Adaboost is called adaptive as the weights are re-assigned to each instance, with higher weights assigned to incorrectly classified instances so that they get **boosted.** As weak-learners, we will use Decision stumps which are simply one-noded Decision Trees that have only one split rule. The precision of Decision stumps are generally slightly better than random coin-flipping. By bringing together many weak-learners, we aim to create a final strong classifier that gets fed by those weak base learners.

The structure of the paper will be as follows: we will first give a theoretical background of the techniques we use in the implementation.We then will introduce the functions used for the scratch implementation along with the an example showing the usage of functions, data acquisition and analysis and a comparison with standard python libraries.

Teoretical background

Adaboost

Assume we have a training set: $S = (x_1, y_1), ..., (x_m, y_m)$ and some generic learning algorithm A and from this algorithm A,we want a smarter way to generate $h_1, ..., h_T$, smarter than random subsampling of S. We assume that $h_1, ..., h_t$ belong to some family H of **base-classifiers.**

A typical choice for H is the **decision stumps** which we are using in this project. These are classifiers of the form $h_{i,\tau} : R^d \to \{-1, 1\}$ defined by $h_{i,\tau}(x) = \pm sgn(x_i - \tau)$ where $i = 1, ..., d$ and $\tau \in R$

The first thing is that we do not do a voting like in the other ensemble methods, we do a sort of weighted majority. We assume that our $classifier$:

$$h_i : X \to \{-1, 1\}$$

After learning this h_i, we will learn $F = \sum_{i=1}^{T} w_i h_i$ where $w_i \in R$, are some real weights and then we predict with the $sgn(f)$. What we want to prove at the end is something of the form:

$L_s(L) \leq e^{-\gamma^{2T}}$

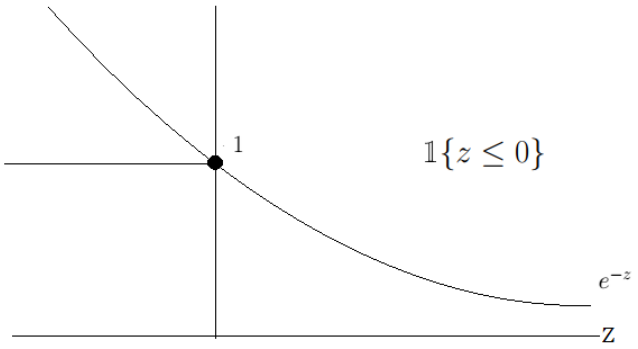
Let's build the algorithm from the proof:

We start from $L_s(f)$ and we want to determine W_i and h_i. It is a binary classification so we can write $L_s(f)$:

$L_s(f) = \frac{1}{m} \sum_{t=1}^{m} \mathbb{1}\{y_t f(x_t) \leq 0\}$, $y_t = \{-1, 1\}$

We make a mistake if $sgnf(x_t)$ disagrees with $sgn(y_t)$, the sign of the label. If $y_t = 1$ and $f(x_t) = 1$ which results in $1 \leq 0$ that is not true. If $y_t = -1$ and $f(x_t) = -1$ which results in $1 \leq 0$ that is not true. However, a disagreement where for example $y_t = -1$ and $f(x_t) = 1$ will satisfy that condition. So this is actually counting the number of classification errors of f over our training set S.

What we are doing is actually of the following form:



We can now have a convex upper-bound to that which is an exponential function. So now, we can replace it:

$L_s(f) \leq \frac{1}{m} \sum_{t=1}^{m} e^{-y_t f(x_t)}$

we can further up the definition

$= \frac{1}{m} \sum_{t=1}^{m} e^{-\sum_{t=1}^{T} y_t w_i h_i(x_t)}$

So now, we can introduce some functions for short-cut:

$L_1, ..., L_T$, these are defined by $L_i(t) = y_t h_i(x_t) \in \{-1, 1\}$ and $L_i(t) = 1 \leftrightarrow h_i(x_t) = y_t$

This is basically the indicator function saying that the i_{th} predictor does well on the t_{th} example in the training set. We now take a random variable Z uniform over the indices of the training set:

Z is uniform over $\{1, ..., m\}$ then $L_i(z)$ is a function of a random variable which we can write as:

$L_s(f) = \frac{1}{m} \sum_{t=1}^{m} e^{-\sum_{t=1}^{T} w_i L_i(t)}$

We can write Z as an expectation. So the probability that Z equals any of the indices above is $\frac{1}{m}$.

$$\mathbb{E}[e^{-\sum_{i=1}^{T} w_i L_i(z)}] = \mathbb{E}[\prod_{i=1}^{T} e^{-w_i L_i(z)}] \tag{1}$$

If the $L_1, ..., L_T$ were independent (i.e., $h_1, ..., h_T$ have independent errors), then we could pull out the product out of expectation.Assuming that (1) holds, which we verify later, we can proceed as follows:

$\mathbb{E}[\prod_{i=1}^{T} e^{-w_i L_i(z)}] = \mathbb{E} \prod_{i=1}^{T} [e^{-w_i L_i(z)}]$

Function of independent random variables are also independent and therefore we could pull out the product from the expectation thanks to the independence property but they are not independent! We will do small trick here and assume pulling out the product is correct but we should put an index there:

$$= \prod_{i=1}^{T} \mathbb{E}_i[e^{-w_i L_i(z)}] \tag{2}$$

What does this mean? It means essentially the expectation with respect to some distribution P_i over the indices in the training set $\{1, ..., m\}$ yet to define. We assume that this distribution P_i exists such that the equation (2) is true and later we will show how to construct it. We assume this holds even if L_i is not independent. Equation (2) gives us a lot! We isolated the contribution of the i_{th} predictor in the overall training error.

We can now proceed as follows:

$L_s(f) \leq \prod_{i=1}^{T} \mathbb{E}_i[e^{-w_i L_i(z)}]$, recall that $L_i(z) \in \{-1, 1\}$

$= \prod_{i=1}^{T} (e^{-w_i} P_i(L_i(z) = 1) + e^{w_i} P_i(L_i(z) = -1))$ and P_i is a probability distribution which is to be defined.

$$= \prod_{i=1}^{T} (e^{-w_i}(1 - \epsilon_i) + e^{w_i}\epsilon_i \tag{3}$$

where $\epsilon_i = P_i(L_i(z) = -1) = \sum_{T=1}^{m} \mathbb{1}\{L_i(t) = -1\} P_i(t)$ and we simply sum the probabilities of elementary events where the random variable takes the value corresponding to the event.

We upper bounded the training error with (3), so we now can learn what is w_i in order to minimize that expression. We can study it as a function of w_i and it turns out that it is minimized at:

$$w_i = \frac{1}{2} ln \frac{(1 - \epsilon_i)}{\epsilon_i} \tag{4}$$

Note that above expression is only defined for $0 < \epsilon_i < 1$. As we will see, $P(t) > 0$ for all $t \in \{1, ..., m\}$. Hence, $\epsilon_i \in \{0, 1\}$ implies either h_i or $-h_i$ has zero training error on S. If this happens, then we can throw away all h_j for $j \neq i$ and avoid using boosting altogether.

Substituting (4) in (3), we get:

$L_s(f) = \prod_{i=1}^{T} \left(\sqrt{\epsilon_i(1 - \epsilon_i)} + \sqrt{\epsilon_i(1 - \epsilon_i)} \right)$

Proceeding:

$$= \prod_{t=i}^{T} \sqrt{(4\epsilon_i(1 - \epsilon_i)} \text{ which results in } \epsilon_i = \sum_{t=1}^{m} \mathbb{1}\{(x_t) \neq y_t\} P_i(t) \tag{5}$$

So we see that ϵ_i is the training error of h_i with respect to the distribution P_i which is not necessarily uniform. Therefore this is the notion of weighted training error according to the distribution P_i.

Note also that $w_i = 0$ if and only if $\epsilon = \frac{1}{2}$, meaning that the weight(according to P_i) of the training points where the error of h_i is exactly $\frac{1}{2}$. Because such a h_i does not affect the value of f since($\epsilon_i = \frac{1}{2}$ implies $w_i = 0$) without loss of generality we may also assume that $\epsilon_i \neq \frac{1}{2}$.

So now we set:

$\gamma_i = \frac{1}{2} - \epsilon_i$ and note that $\epsilon_i \neq \frac{1}{2}$ implies $\gamma_i \neq \frac{1}{2}$. . Substituting γ in equation (5) yields

$\leq \prod_{i=1}^{T} \sqrt{1 - 4\gamma_i^2}$

Using the inequality $1 + x \leq e^x$ which always holds:

$\leq \prod_{i=1}^{T} \sqrt{e^{-4\gamma_i^2}} = \prod_{i=1}^{T} e^{-2\gamma_i^2} = e^{-2\sum_{i=1}^{T} \gamma_i^2}$ which eventually equals to:

$$L_s(f) \leq e^{-2T\gamma^2} \tag{6}$$

If we assume that $|\gamma_i| > |\gamma| > 0$, we get the upper bounded equation in (6).

ϵ_i is the weighted training error of h_i and γ_i is telling us how this ϵ_i is away from random coin-flipping. If h_i makes %50 mistakes weighted by P_i on our traning set, then the γ_i will be zero and it does not contribute to equation (6) but if these absolute values $|\gamma_i|$ are bounded away from 0 which means that ϵ_i is bounded away from a half, then putting our h_i in the committee will drive the training error to 0 exponentially in T. That's why this bound provides a pretty strong control on the bias. Using the observation that $L_s(f) = 0$ if and only if $L_s(f) < \frac{1}{m}$, we conclude that a number :

$$T > \frac{lnm}{2\gamma^2}$$

of boosting rounds is sufficient to bring the training error of f down to zero.

This is very powerful and it rests on an assumption. We are computing some h_i and we are weighting them according to P_i. How do we get P_i and from P_i the recipe to construct the h_i?

We still have to prove that $P_1, ..., P_T$ exist such that the expectation:

$\mathbb{E}[\prod_{i=1}^{T} e^{-w_i L_i(z)}] = \prod_{i=1}^{T} \mathbb{E}_i[e^{-w_i L_i(z)}]$

We define $P_i = \frac{1}{m}, t = 1, ..., m$ and now we observe that $\mathbb{E}_1 = \mathbb{E}$, so the left-hand side expectation above is exactly the expectation over the uniform distribution because it was originally written when Z was uniformly distributed. While in the right-hand side equation, Z, is distributed according $\mathbb{E}_i$. So when P_i is uniform $\mathbb{E}_1 = \mathbb{E}$.

We now make some assumptions, we recursively define $P_i+1(t) = \frac{P_i(t)e^{-w_i L_i(t}}{\mathbb{E}[e^{-w_i L_i(z)}]}$

Here we know how to calculate the basis of our recursive definition that P_i and the denominator is defined as :

$\leq \sum_{S=1}^{m} e^{-w_i L_i(s)} P_i(s)$

So now, $P_i + 1$ is a distribution. The denominator is the normalization constant to make the probability sum to 1. The top is the probabilities because the exponential never go to negative. So they are probability distributions over our indices of the training set. Solving the division above to prove the equation (2):

$e^{-w_i L_i(t)} = \mathbb{E}_i[e^{-w_i L_i(z)}] \frac{P_i+1(t)}{P_i(t)}$

Proceeding further:
$\mathbb{E}_1[\prod_{i=1}^{T} e^{-w_i L_i(z)}] = \frac{1}{m} \sum_{t=1}^{m} \left(\prod_{i=1}^{T} \mathbb{E}_1[e^{-w_i L_i(z)}] \frac{P_i+1(t)}{P_i(t)} \right)$

sum of products of probabilities go as:

$\frac{1}{m} \sum_{t=1}^{m} \left(\prod_{i=1}^{T} \mathbb{E}_1[e^{-w_i L_i(z)}] \right) \left(\prod_{i=1}^{T} \frac{P_i+1(t)}{P_i(t)} \right)$

By definiton $P_i(t) = \frac{1}{m}$ and so it cancels out with $\frac{1}{m}$ and what we are left with is the probability distribution which amounts to 1.

$$= \prod_{i=1}^{T} \mathbb{E}\left[e^{-w_i L_i(z)}\right] \tag{7}$$

These probabilities have a simple interpretation when one studies how $P_i + 1$ depends on P_i. Fix P_i and suppose $\epsilon_i < \frac{1}{2}$ which means $w_i > 0$ and each $P_i+1(t)$ is obtained multiplying $P_i(t)$ for the quantity $e^{-w_i L_i(t)}$, which is bigger than 1 if and only if h_i makes a mistake on (x_t, y_t). Intuitively, the boosting process concentrates the weight on the training examples that are misclassified by the previous classifiers. A similar argument applies to the case when $\epsilon > \frac{1}{2}$.

Cross Validation and Hyperparameter Tuning

In order to analyse a learning problem, we must define a mathematical model of how examples are generated. In statistical learning, a learning problem is defined by (D, L) where D is the distribution and L is the loss function. The performance of a predictor h:

$h : X \to Y$, where X is the data domain and Y is the label set.

h is our predictor that maps the point(X) to the label set(Y). So h is our solution we came up to the learning problem (D, L). We want to define how good a predictor is because we want to discriminate the good ones from the bad ones.

$L\left(Y, h(x)\right) \to$ Loss function coming from the predictor h worked on $h(x)$ with label Y.

$$L_D(h) = \mathbb{E}[L\left(Y, h(x)\right)] \tag{8}$$

This is the **Statistical risk:** risk of a predictor h. Given a certain learning problem, it is the expected loss of the predictor on an example X drawn from an unknown distribution. This is basically the expected value of the loss function on a random example (X, Y) drawn from D.

In practice, learning problems are often specified up to one or more hyperparameters. Some algorithms have parameters some of which are set by the algorithm itself and some others are directly given by the user. A typical example is the K in K-NN where the user has to specify the number of nearest neighbors to be included in the committee. For example in Neural Networks, "the weights" are trainable parameters while the number of epochs and the batch size are hyperparameters. When we have an algorithm with a hyperparameter, we are actually not given a single algorithm but a family of algorithms since we have different algorithms for each hyperparameter given, for example K-NN with K=10 and K=20 will probably give different results.

Let $\{A_\theta : \theta \in \Theta\}$ be such a family of algorithms, where Θ is the set of all possible hyperparameter values. We now fix a learning problem (D, L). Let $A_\theta(s)$ be our predictor and let $\mathbb{E}[L_D(A_\theta)]$ be the average risk of the predictor output by A_θ where the expectation is with respect to the random draw of the traning set S of a given fixed size. So intuitively, $\mathbb{E}[L_D(A_\theta)]$ measures the performance of A_θ on a typical training set of that size.

We assume that we fixed $\theta \in \Theta$ and we ask how good is A_θ with respect to $S \sim D^m$

The problem here is not to evaluate a predictor. We know how to evaluate a predictor. We just take a test set and we evaluate it over that. Here, we want to evaluate an algorithm. We cannot evaluate an algorithm over a specific test set because if we do that, we would be evaluating a predictor. We want to evaluate the algorithm abstracting specific training set. We want to see how good the typical predictors output by this algorithm with respect to the choice of hyperparameter. So, this difference is crucial. An algorithm outputs a potentially different predictor for each training set in input, so we need to do something more complicated. To do that, we use;

External Cross Validation

We know that we cannot focus on a single training set because otherwise we will be evaluating a deterministic predictor, not the algorithm. To evaluate the algorithm we use a technique called **K-Fold(external)** cross-validation.

Let S be our entire dataset. We partition S in K subsets(folds) $D_1, ..., D_K$ of size $\frac{m}{K}$. Now let $S^{(k)} \equiv \frac{S}{D_k}$. We call D_k the **testing part** of the k-th fold while $S^{(k)}$ is the **training part**.

For example, if we partition $S = \{(x_1, y_1), ..., (x_{20}, y_{20}\}$ in $K = 4$ subsets, we have:
$D_1 = \{(x_1, y_1), ..., (x_5, y_5)\}, D_2 = \{(x_6, y_6), ..., (x_{10}, y_{10})\}$

$D_3 = \{(x_{11}, y_{11}), ..., (x_{15}, y_{15})\}, D_4 = \{(x_{16}, y_{16}), ..., (x_{20}, y_{20})\}$

then $S^2 = \{(x_1, y_1), ..., (x_5, y_5), (x_{11}, y_{11}), ..., (x_{20}, y_{20})\}$ where D_2 is removed from S.

The K-fold CV estimate of $\mathbb{E}[L_D(A)]$ on S, denoted by $L_s^{cv}(A)$ is then computed as follows:

$$\hat{l}_{D_k}(h_k) = \frac{K}{M} \sum_{(x,y) \in D_k} l(y, h_k(x)) \tag{9}$$

Finally, we compute the CV estimate by averaging these errors:

$$\hat{l}_s^{(cv)}(A) = \frac{1}{K} \sum_{k=1}^{K} l_{D_k}(h_k) \tag{10}$$

One vs Rest Classifier

When we have a multi-classification problem, we have a few techniques to apply one which is **OVR** or **One vs rest encoding/classifier.** This strategy consists in fitting one classifier per class. For each class, we have a classifier that is trained against the other classes. The i_{th} classifier distinguishes the $class_i$ from the rest. For example, if we have three classes i, j, k then we should have three classifiers for each class separating them from the rest as $(i_{th}\ vs\ j, k), (j_{th}\ vs\ i, k), (k_{th}\ vs\ i, j)$. Suppose $h_1, ..., h_k : X \to R$ are our binary classifiers that outputs classes in the form of $\{1, -1\}$, then the final prediction is:

$$h(x) = \underset{i \in 1...k}{\operatorname{argmax}}\, h_i(x)$$

This method is easily interpretable. Since each class is represented by only one binary classifier, we can inspect that classifier to gain knowledge about that class. One disadvantage of this method is that we will have imbalanced dataset.

In the case of multi-class Adaboost, for each class we will be given a what we call **alpha** value that is given in equation (4):

$$w_i = \tfrac{1}{2} ln \tfrac{(1-\epsilon_i)}{\epsilon_i}$$

We will have k alpha values for each data point where $k = number\, of\, classes$. The largest alpha value summed over the rounds by being multiplied with the prediction of that round for each data point will give the final prediction for that data point:

$$H(x) = \underset{y \in 1...k}{\operatorname{argmax}} \sum_{t: h_t(x) = y} \alpha_t$$

As a simple example, assume that we have three data points and that we had three adaboost rounds whose predictions and alpha values for each data point are given as:

round1-alpha	round2-alpha	round3-alpha
0.42	0.65	0.38
round1-prediction	round2-prediction	round3-prediction
1	-1	-1
1	1	1
-1	-1	1

then the first prediction is 0.42 X 1 + 0.65 X -1 + 0.38 X -1 = -0.61. We then apply the *sgn* function which will return a -1 for this data point and that will be our prediction. This is the binary case. In the multi-class case, we will have a $m\, x\, n$ matrix where **m** is the number of data points and **n** is the number

of classes composed of alpha values like -0.61. There will be k number of alpha values for each data point where k = $number of classes$.

Now assume that we have three classes, ran three adaboost rounds for three data points. For this, we need to train three different classifiers and assume we used one-vs-rest encoding, we will then have three alpha values for each data points coming from corresponding adaboost rounds:

alpha values	predictions
0.42	1
0.65	-1
0.38	-1

The first classifier will return an alpha value of -0.61 for this data point. Assume that the second classifier returned -0.149 and the third returned -1.45. Since we have three data points, we repeat this process three times which results in the following alpha $3x3$ matrix:

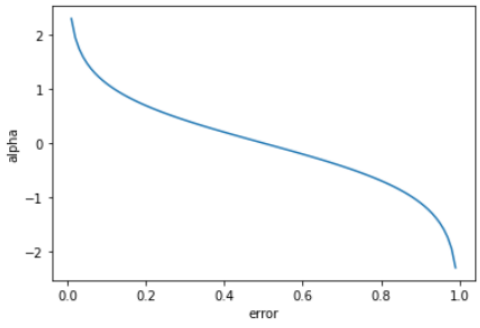
Alpha values		
-.61	-0.149	-1.45
1.45	0.69	0.172
-0.149	-.61	0.172

and the class prediction of the first point will be 2, second point will be 1 and the third point will be 3.

```
epsilon=np.arange(0.01,1,0.01)
alpha=1/2*np.log((1-epsilon)/epsilon)

plt.plot(epsilon, alpha)
plt.xlabel('error')
plt.ylabel('alpha')
```

```
Text(0, 0.5, 'alpha')
```



The relationship between epsilon(weighted training error) and alpha is clearly telling us why we are using the highest alpha value to classify the data points in

the multi-class case. As the plot suggests, the highest alpha implies lowest epsilon which means that the algorithm is highly confident in classifying that point so that it assigns a high alpha value. Since the algorithm returns n alpha values for a dataset composed of n classes, we pick n_{th} class for a given data point for which the algorithm has the highest confidence to classify it.

FUNCTIONS

In the following, the main python functions for Adaboost, Decision Stump and Cross-Validation are introduced.

In [1]:
```
import numpy as np
import pandas as pd
import random
from matplotlib.pyplot import figure
import matplotlib.pyplot as plt
import seaborn as sns
```

- numpy for performing linear algebra on vectors and matrices
- pandas for manipulating and analysing the data
- matplotlib and seaborn for plotting the results

In [2]:
```
def sign(y):
    x=[]
    for i in y:
        if i>0:
            x.append(1)
        elif i<0:
            x.append(-1)
        else:
            x.append(0)
    return(np.mat(x).T)
```

Sign function simply implements the numpy sgn.

In [3]:
```
def train_test_split(df,test_size,label_col,random_state=50):
    import random
    import pandas as pd

    random.seed(random_state)

    label_col = str(label_col)
    dat_len  = len(df)
    X= df.drop(columns=label_col)
    Y = data[label_col]



    if isinstance(test_size,float):
        test_size = round(test_size*dat_len)

    indices = list(data.index)
    test_indices = random.sample(population=indices, k=test_size)

    X_train = X.drop(test_indices)
    x_test  = X.loc[test_indices]

    Y_train= Y.drop(test_indices)
    y_test = Y.loc[test_indices]


    return X_train.sample(frac=1,random_state=random_state),x_test,
    Y_train.sample(frac=1,random_state=random_state),y_test
```

This function implements train and test splitting based on the required parameters. It takes the whole dataset with the name of label column as a string, it returns randomly shuffled test and train set.

Decision Stump

In [4]:
```
def Decision_stump(data, label_set,P_i):
    data = np.mat(data)
    row_len,col_len = np.shape(data)
    label_set = np.mat(label_set).T
    predictions = np.ones((np.shape(data)[0],1))
    #search_size = search_size
    lowest_err = float('inf')
    decision_stump = {}
    for col in range(col_len):
        search_size= np.unique(data[:,col],axis=0)
        if len(search_size) > 100:
            search_size = round(np.sqrt(len(search_size)))
            #search_size= len(np.unique(data[:,col],axis=0))
            #search_size=round(search_size*0.6)
            col_min = data[:,col].min()
            col_max = data[:,col].max()
            stepSize = (col_max - col_min) / search_size
            for j in range(int(search_size)):
                    for theta_inq in ['lower', 'greater']:
                        theta = (col_min + float(j) * stepSize)
                        if theta_inq =='lower':
                            predictions[data[:,col] <= theta ]= -1
                            est_mat = predictions.copy()
                            errors = np.mat(np.ones((row_len,1)))
                            errors[predictions == label_set] = 0
                            weighted_Error = P_i.T * errors
                            predictions = np.ones((np.shape(data)[0],1))

                            if weighted_Error < lowest_err:
                                lowest_err = weighted_Error
                                est_matrix = est_mat.copy()
                                decision_stump['dimen'] = col
                                #decision_stump['theta'] = round(theta,2)
                                decision_stump['theta'] = theta
                                decision_stump['theta_inq'] = theta_inq
                    else:

                        predictions[data[:,col]> theta ]= -1
                        est_mat = predictions.copy()
                        errors = np.mat(np.ones((row_len,1)))
                        errors[predictions == label_set] = 0
                        weighted_Error = P_i.T * errors
                        predictions = np.ones((np.shape(data)[0],1))

                        if weighted_Error < lowest_err:
                            lowest_err = weighted_Error
                            est_matrix = est_mat.copy()
                            decision_stump['dimen'] = col
                            #decision_stump['theta'] = round(theta,2)
                            decision_stump['theta'] = theta
                            decision_stump['theta_inq'] = theta_inq
```

```
    else:
        for theta in search_size:
            for theta_inq in ['lower', 'greater']:
                if theta_inq =='lower':
                    predictions[data[:,col] <= theta ]= -1
                    est_mat = predictions.copy()
                    errors = np.mat(np.ones((row_len,1)))
                    errors[predictions == label_set] = 0
                    weighted_Error = P_i.T * errors
                    predictions = np.ones((np.shape(data)[0],1))

                    if weighted_Error < lowest_err:
                        lowest_err = weighted_Error
                        est_matrix = est_mat.copy()
                        decision_stump['dimen'] = col
                        #decision_stump['theta'] = round(theta,2)
                        decision_stump['theta'] = theta
                        decision_stump['theta_inq'] = theta_inq
                else:

                    predictions[data[:,col]> theta ]= -1
                    est_mat = predictions.copy()
                    errors = np.mat(np.ones((row_len,1)))
                    errors[predictions == label_set] = 0
                    weighted_Error = P_i.T * errors
                    predictions = np.ones((np.shape(data)[0],1))

                    if weighted_Error < lowest_err:
                        lowest_err = weighted_Error
                        est_matrix = est_mat.copy()
                        decision_stump['dimen'] = col
                        #decision_stump['theta'] = round(theta,2)
                        decision_stump['theta'] = theta
                        decision_stump['theta_inq'] = theta_inq

return decision_stump,lowest_err,est_matrix
```

Decision Stump function is looking for the best split having the lowest weighted error on the training set. It has three arguments **the data, label set and probability distribution** . It returns a dictionary composed of best split, best split value and split feature with the lowest error and the estimated prediction matrix resulting in these values. In order to reduce the time complexity of the algorithm, the decision stump function uses all the unique values as thresholds if they are lower than 100 in a given feature, otherwise it uses the square root of the total number of unique values in a given feature to create a step-size to decide the how big the jumps should be between thresholds. What we call probability distribution, P_i, is the weights that is firstly given by the length of the dataset to be updated by the adaboost algorithm.

In [5]:
```
def Adaboost(data,label_set,T_rounds):

    data = np.mat(data)
    models = []
    row_len = data.shape[0]
    P_i = np.mat(np.ones(shape=(row_len, 1)) / row_len)
    preds = np.mat(np.zeros(shape=(row_len, 1)))
    for i in range(T_rounds):
        decision_stump, epsilon, estimation = Decision_stump(data, label_set, P_i)
        alpha = float(0.5 * np.log((1 - epsilon) / max(epsilon, 1e-16)))
        decision_stump['alpha'] = alpha
        models.append(decision_stump)
        P_i = np.multiply(P_i, np.exp(np.multiply(-1 * alpha * np.mat(label_set).T, estimation)))
        P_i =P_i / P_i.sum()   #normalize
        preds += alpha * estimation  #prediction
        totalerror = np.multiply(sign(preds) != np.mat(label_set).T, np.ones(shape=(row_len, 1)))
        errorRate = totalerror.sum() / row_len

        if errorRate == 0.0:
            break
    return models, preds
```

There is a two-way relationship between Adaboost and Decision-Stump functions. Decision-Stump feeds adaboost with the best matrix resulting in the lowest error using the probability distribution, then adaboost takes this lowest error, **epsilon**, calculates **alpha** values. It then uses the best estimated matrix coming from Decision Stump and the alpha value to update the probability distribution to again feed the Decision Stump. This function returns best models and predictions.

In [6]:
```
def Adaboost_Predict(data,models):
    data = np.mat(data)
    row_len = data.shape[0]
    preds = np.mat(np.zeros(shape =(row_len, 1)))
    for model in models:
        if model['theta_inq']=='lower':
            res = np.ones((row_len,1))
            res[data[:,model['dimen']] <= model['theta']] = -1
            preds += res*model['alpha']
        else:
            res = np.ones((row_len,1))
            res[data[:,model['dimen']]  > model['theta']] = -1
            preds += res*model['alpha']

    return(preds)
```

Adaboost predict function takes the test set with the best models returned by Adaboost function to make predictions on the test set.

In [7]:
```
def Adaboost_Multi_CV(data,k_folds,T_rounds,label_col,random_state):

    random.seed(random_state)

    X_train,x_test,Y_train,y_test = train_test_split(data,test_size=0.2,label_col=label_col,random_state=random_state)
    indices = np.array_split(list(X_train.index),k_folds)

    Cv_test_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))
    Cv_train_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))

    for T in T_rounds:
        for k in range(k_folds):
            train_preds= np.mat(np.ones(shape=(np.shape(X_train.drop(indices[k]))[0],len(np.unique(data[label_col])))))
            test_preds = np.mat(np.ones(shape=(np.shape(indices[k])[0],len(np.unique(data[label_col])))))
            for classes in range(len(np.unique(data[label_col]))):
                model,pred = Adaboost(X_train.drop(indices[k]),np.where(Y_train.drop(indices[k])==classes+1,1,-1),T)
                train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
                test_est = Adaboost_Predict(X_train.loc[indices[k]],model)
                test_preds[:,classes] = np.multiply(test_preds[:,classes],test_est)

            train_predictions = np.argmax(train_preds,axis=1)+1
            training_error =np.where(train_predictions!=np.mat(Y_train.drop(indices[k])).T,1,0).sum()
            Cv_train_acc[T_rounds.index(T),k]=1-(training_error/len(train_predictions))
            test_prediction = np.argmax(test_preds,axis=1)+1
            test_error = np.where(test_prediction!=np.mat(Y_train.loc[indices[k]]).T,1,0).sum()
            Cv_test_acc[T_rounds.index(T),k]=1-(test_error/len(test_prediction))
    return(Cv_train_acc,Cv_test_acc)
```

This function implements multi-class external cross validation. As arguments, it takes the whole dataset, number of k-folds, T-rounds, label-column as a string. It has an internal train-test-split function so we do not have to split the dataset before feeding it.

In [8]:
```
def learning_curve(data,k_folds,T,test_size,label_col,random_state):

    if isinstance(test_size,int):
                raise TypeError('Test_size should be a range of numbers')

    random.seed(random_state)

    Cv_test_acc = np.mat(np.ones(shape=(len(test_size),k_folds)))
    Cv_train_acc = np.mat(np.ones(shape=(len(test_size),k_folds)))
    train_size =[]
    for value in test_size:
        X_train,x_test,Y_train,y_test = train_test_split(data,test_size=value,label_col=label_col,random_state=random_state)
        indices = np.array_split(list(X_train.index),k_folds)
        train_size.append(value*len(data))
        for k in range(k_folds):
            train_preds= np.mat(np.ones(shape=(np.shape(X_train.drop(indices[k]))[0],len(np.unique(data[label_col])))))
            test_preds = np.mat(np.ones(shape=(np.shape(indices[k])[0],len(np.unique(data[label_col])))))
            for classes in range(len(np.unique(data[label_col]))):
                model,pred = Adaboost(X_train.drop(indices[k]),np.where(Y_train.drop(indices[k])==classes+1,1,-1),T)
                train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
                test_est = Adaboost_Predict(X_train.loc[indices[k]],model)
                test_preds[:,classes] = np.multiply(test_preds[:,classes],test_est)
            train_predictions = np.argmax(train_preds,axis=1)+1
            training_error =np.where(train_predictions!=np.mat(Y_train.drop(indices[k])).T,1,0).sum()
            Cv_train_acc[test_size.index(value),k]=1-(training_error/len(train_predictions))
            test_prediction = np.argmax(test_preds,axis=1)+1
            test_error = np.where(test_prediction!=np.mat(Y_train.loc[indices[k]]).T,1,0).sum()
            Cv_test_acc[test_size.index(value),k]=1-(test_error/len(test_prediction))


    return(Cv_train_acc,Cv_test_acc,train_size)
```

This function determines cross-validated training and test scores for different training set sizes. It has an internal train-test-split function so we do not have to split the dataset before feeding it.

EXAMPLE

In the following, we will present an example to show the usage of above functions.

In [9]:
```
from sklearn.datasets import make_classification
X,y = make_classification(n_samples=1000, n_features=3, n_informative=3,
                          n_redundant=0, n_repeated=0, n_classes=3,class_sep=1.1,
                          flip_y=0, random_state=50)

#manipulate the dataset so that it has the proper form for the usage of above functions.

data= np.column_stack((X,y))
data = pd.DataFrame.from_records(data)
data.columns = data.columns.astype(str)
data = data.rename(columns={'3':'decision'})
data['decision'] = data['decision']+1
data
```

Out[9]:

	0	1	2	decision
0	0.296113	-0.353656	-0.581267	1.0
1	1.629059	0.653927	-1.874410	3.0
2	-0.342091	1.752425	1.344634	3.0
3	0.651383	-1.728245	-1.267767	1.0
4	1.580371	1.737306	-1.313136	3.0
...	...	...	...	...
995	1.673624	0.691964	-2.277399	3.0
996	1.846340	0.756485	-2.262380	3.0
997	-0.628881	-0.232976	3.399956	2.0
998	-1.636583	1.522925	-0.193600	1.0
999	0.783471	1.974829	0.107448	3.0

1000 rows × 4 columns

We have a thousand randomly generated observations with three different classes. Now, using Adaboost we will try to predict those classes. We first call our Adaboost-multi-class-Cv function to decide about optimal number of boosting rounds. For the number of boosting, we will try five values ranging from 10 to 50. To tune these parameters, we use 5-fold external cross validation. The result is the following:

```
In [10]: train,test = Adaboost_Multi_CV(data,k_folds=5,T_rounds=[10,20,30,40,50],
                                        label_col='decision',random_state=50)

In [11]: np.mean(test,axis=1)

Out[11]: matrix([[0.7525 ],
                 [0.76625],
                 [0.77125],
                 [0.775  ],
                 [0.77625]])

In [12]: np.mean(train,axis=1)

Out[12]: matrix([[0.77125 ],
                 [0.79    ],
                 [0.796875],
                 [0.798125],
                 [0.80125 ]])
```

The highest test and training accuracy comes with 50 adaboost rounds.

We now call the learning-curve function to see how the algorithm learns with respect to the number of data points given to it. We will again apply 5-fold cross validation with 0.1,0.2,0.3,0.6 and 0.8 test size and try to understand what is the best option.

```
In [14]: train_acc,test_acc,test_size = learning_curve(data,k_folds=5,T=50,
                                        test_size=[0.1,0.2,0.3,0.7,0.8],label_col='decision',random_state=50)

In [15]: np.mean(train_acc,axis=1)

Out[15]: matrix([[0.79527778],
                 [0.80125   ],
                 [0.79142857],
                 [0.82666667],
                 [0.86      ]])

In [16]: np.mean(test_acc,axis=1)

Out[16]: matrix([[0.76111111],
                 [0.77625   ],
                 [0.75857143],
                 [0.76333333],
                 [0.735     ]])

In [17]: test_size

Out[17]: [100.0, 200.0, 300.0, 700.0, 800.0]
```

The highest training accuracy is reached when we feed the adaboost with only %20 of the dataset to train and %80 to test. While the highest test accuracy comes when we use only %20 of the whole dataset as testing size. Using these findings, we get:

```
In [17]: X_train,x_test,Y_train,y_test = train_test_split(data,test_size=0.2,label_col='decision',random_state=50)
         train_preds= np.mat(np.ones(shape=(np.shape(X_train)[0],len(np.unique(data['decision'])))))
         test_preds =  np.mat(np.ones(shape=(np.shape(x_test)[0],len(np.unique(data['decision'])))))
         for classes in range(len(np.unique(data['decision']))):
             model,pred = Adaboost(X_train,np.mat(np.where(Y_train==int(classes)+1,1,-1)),50)
             train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
             test_est = Adaboost_Predict(x_test,model)
             test_preds[:,classes]=np.multiply(test_preds[:,classes],test_est)

         train_predictions = np.argmax(train_preds,axis=1)+1 # because indexing starts from 0
         training_error = np.where(train_predictions != np.mat(Y_train).T,1,0).sum()
         test_prediction = np.argmax(test_preds,axis=1)+1
         test_error = np.where(test_prediction != np.mat(y_test).T,1,0).sum()
         print('test accuracy:',1-(test_error/len(x_test)))
         print('training accuracy:',1-(training_error/len(X_train)))

         test accuracy: 0.735
         training accuracy: 0.8025

In [18]: np.mean(test,axis=1)-np.std(test,axis=1)

Out[18]: matrix([[0.73455912],
                 [0.74368066],
                 [0.74868066],
                 [0.74622284],
                 [0.74932418]])
```

A testing score around %73.5 that is inside the standard errors. So we got a reasonable result.

```
In [19]: model

Out[19]: [{'dimen': 0,
           'theta': 0.37016586976675825,
           'theta_inq': 'lower',
           'alpha': 0.6438274730675708},
          {'dimen': 2,
           'theta': -1.828382216597317,
           'theta_inq': 'greater',
           'alpha': 0.39475846201525366},
          {'dimen': 2,
           'theta': -0.6494302638476057,
           'theta_inq': 'lower',
           'alpha': 0.36315399598851705},
          {'dimen': 0,
           'theta': -0.11624070648315232,
           'theta_inq': 'lower',
           'alpha': 0.34666318001498214},
          {'dimen': 2,
           'theta': -1.828382216597317,
           'theta_inq': 'greater',
           'alpha': 0.2307525348514036}
```

Here are the models used for the prediction. These models include best split with the feature index, thresholds and alpha values. Adaboost-predict uses them to do predictions.

Data Acquisition and Analysis

```
In [23]: data = pd.read_csv('forest-cover-type.csv')
         data = data.drop(columns=['Id'])
         forest = data.copy()
```

```
In [24]: forest.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15120 entries, 0 to 15119
Data columns (total 55 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   Elevation                           15120 non-null  int64
 1   Aspect                              15120 non-null  int64
 2   Slope                               15120 non-null  int64
 3   Horizontal_Distance_To_Hydrology    15120 non-null  int64
 4   Vertical_Distance_To_Hydrology      15120 non-null  int64
 5   Horizontal_Distance_To_Roadways     15120 non-null  int64
 6   Hillshade_9am                       15120 non-null  int64
 7   Hillshade_Noon                      15120 non-null  int64
 8   Hillshade_3pm                       15120 non-null  int64
 9   Horizontal_Distance_To_Fire_Points  15120 non-null  int64
 10  Wilderness_Area1                    15120 non-null  int64
 11  Wilderness_Area2                    15120 non-null  int64
 12  Wilderness_Area3                    15120 non-null  int64
 13  Wilderness_Area4                    15120 non-null  int64
 14  Soil_Type1                          15120 non-null  int64
 15  Soil_Type2                          15120 non-null  int64
 16  Soil_Type3                          15120 non-null  int64
 17  Soil_Type4                          15120 non-null  int64
 18  Soil_Type5                          15120 non-null  int64
 19  Soil_Type6                          15120 non-null  int64
 20  Soil_Type7                          15120 non-null  int64
 21  Soil_Type8                          15120 non-null  int64
 22  Soil_Type9                          15120 non-null  int64
 23  Soil_Type10                         15120 non-null  int64
 24  Soil_Type11                         15120 non-null  int64
 25  Soil_Type12                         15120 non-null  int64
 26  Soil_Type13                         15120 non-null  int64
 27  Soil_Type14                         15120 non-null  int64
 28  Soil_Type15                         15120 non-null  int64
 29  Soil_Type16  15120 non-null  int64
 30  Soil_Type17  15120 non-null  int64
 31  Soil_Type18  15120 non-null  int64
 32  Soil_Type19  15120 non-null  int64
 33  Soil_Type20  15120 non-null  int64
 34  Soil_Type21  15120 non-null  int64
 35  Soil_Type22  15120 non-null  int64
 36  Soil_Type23  15120 non-null  int64
 37  Soil_Type24  15120 non-null  int64
 38  Soil_Type25  15120 non-null  int64
 39  Soil_Type26  15120 non-null  int64
 40  Soil_Type27  15120 non-null  int64
 41  Soil_Type28  15120 non-null  int64
 42  Soil_Type29  15120 non-null  int64
 43  Soil_Type30  15120 non-null  int64
 44  Soil_Type31  15120 non-null  int64
 45  Soil_Type32  15120 non-null  int64
 46  Soil_Type33  15120 non-null  int64
 47  Soil_Type34  15120 non-null  int64
 48  Soil_Type35  15120 non-null  int64
 49  Soil_Type36  15120 non-null  int64
 50  Soil_Type37  15120 non-null  int64
 51  Soil_Type38  15120 non-null  int64
 52  Soil_Type39  15120 non-null  int64
 53  Soil_Type40  15120 non-null  int64
 54  Cover_Type   15120 non-null  int64
dtypes: int64(55)
memory usage: 6.3 MB
```

Our dataset contains 15120 observations of seven different cover-types.

- Elevation : Elevation in meters
- Aspect : Aspect in degrees
- Slope : Slope in degrees
- Horizontal-Distance-To-Hydrology: Horz Dist to nearest surface water features
- Vertical-Distance-To-Hydrology - Vert Dist to nearest surface water features
- Horizontal-Distance-To-Roadways - Horz Dist to nearest roadway
- Hillshade-9am (0 to 255 index) - Hillshade index at 9am, summer solstice
- Hillshade-Noon (0 to 255 index) - Hillshade index at noon, summer solstice
- Hillshade-3pm (0 to 255 index) - Hillshade index at 3pm, summer solstice
- Horizontal-Distance-To-Fire-Points - Horz Dist to nearest wildfire ignition points
- Wilderness-Area (4 binary columns, 0 = absence or 1 = presence)- Wilderness area designation

• Soil-Type (40 binary columns, 0 = absence or 1 = presence) - Soil Type designation

• Cover-Type (7 types, integers 1 to 7) - Forest Cover Type designation

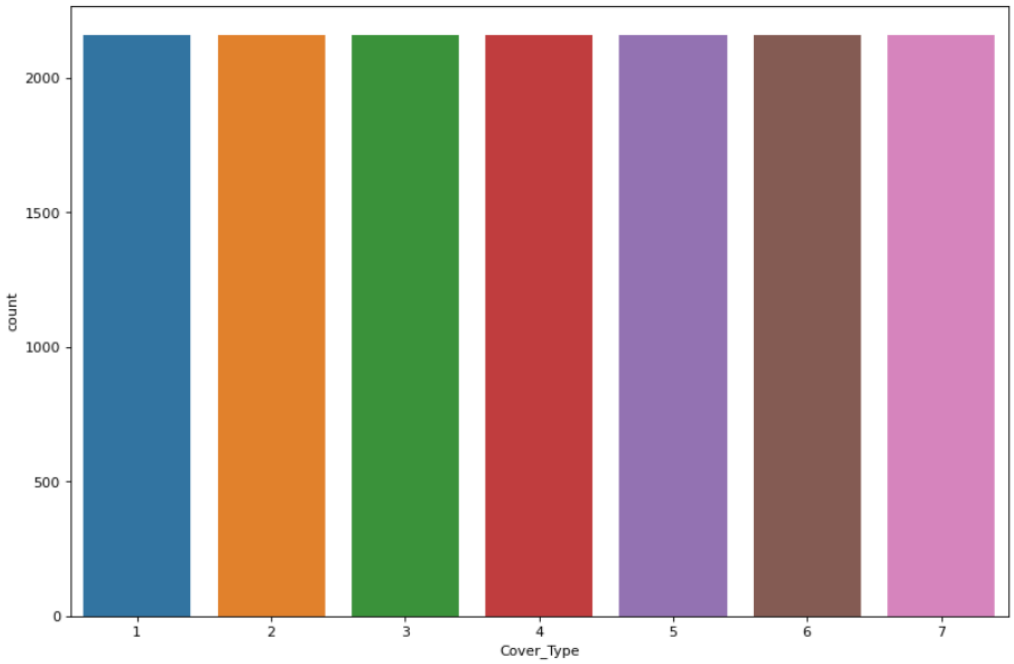
```
In [25]: forest.isna().sum()

Out[25]: Elevation                           0
         Aspect                              0
         Slope                               0
         Horizontal_Distance_To_Hydrology    0
         Vertical_Distance_To_Hydrology      0
         Horizontal_Distance_To_Roadways     0
         Hillshade_9am                       0
         Hillshade_Noon                      0
         Hillshade_3pm                       0
         Horizontal_Distance_To_Fire_Points  0
         Wilderness_Area1                    0
         Wilderness_Area2                    0
         Wilderness_Area3                    0
         Wilderness_Area4                    0
         Soil_Type1                          0
         Soil_Type2                          0
         Soil_Type3                          0
         Soil_Type4                          0
         Soil_Type5                          0
         Soil_Type6                          0
         Soil_Type7                          0
         Soil_Type8                          0    Soil_Type25     0
         Soil_Type9                          0    Soil_Type26     0
         Soil_Type10                         0    Soil_Type27     0
         Soil_Type11                         0    Soil_Type28     0
         Soil_Type12                         0    Soil_Type29     0
         Soil_Type13                         0    Soil_Type30     0
         Soil_Type14                         0    Soil_Type31     0
         Soil_Type15                         0    Soil_Type32     0
         Soil_Type16                         0    Soil_Type33     0
         Soil_Type17                         0    Soil_Type34     0
         Soil_Type18                         0    Soil_Type35     0
         Soil_Type19                         0    Soil_Type36     0
         Soil_Type20                         0    Soil_Type37     0
         Soil_Type21                         0    Soil_Type38     0
         Soil_Type22                         0    Soil_Type39     0
         Soil_Type23                         0    Soil_Type40     0
         Soil_Type24                         0    Cover_Type      0
```

Our dataset does not contain any null values.

```
In [28]: figure(figsize=(12, 8), dpi=80)
         sns.countplot(x="Cover_Type", data=forest)
```

```
Out[28]: <AxesSubplot:xlabel='Cover_Type', ylabel='count'>
```



We have a perfectly balanced dataset. However, since we will be using one-vs-rest classifier for the multi-class classification, this will not help us. We now call our learning-curve function with 5-fold cross validation to see how the algorithm behaves with respect to the training or test size provided. Recall that it has an internal train-test-split function, so we do not have to split it before.

In [131]:
```
train_acc,test_acc,test_size = learning_curve(forest,k_folds=5,T=30,
                               test_size=[0.1,0.2,0.3,0.7,0.8,0.9],label_col='Cover_Type',random_state=50)
```

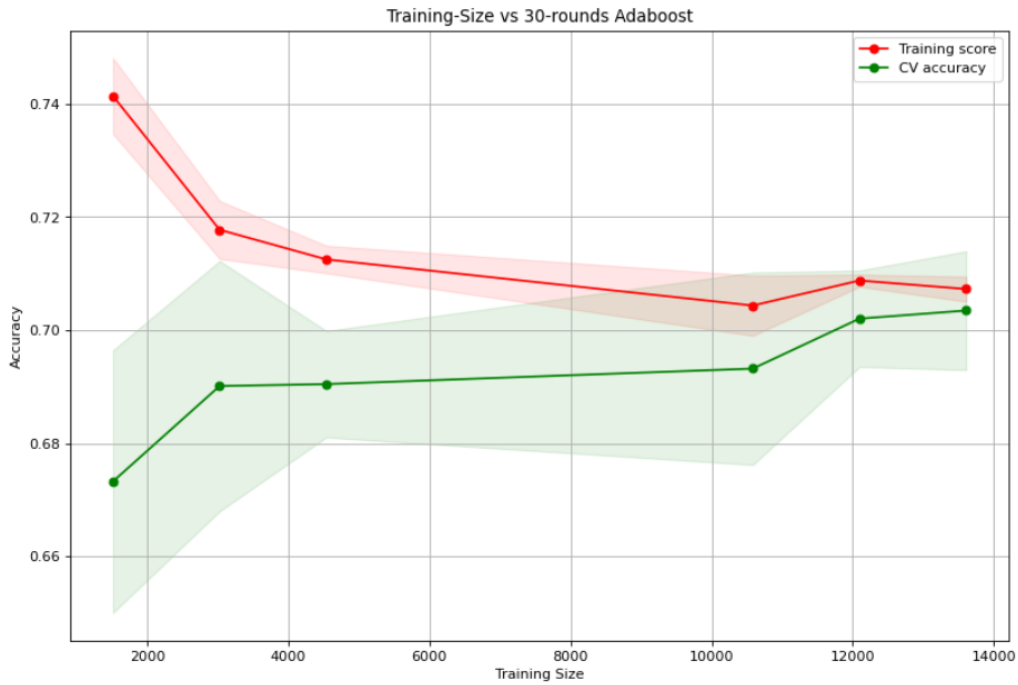
```
Cv for 0.1 Test size is completed
Cv for 0.2 Test size is completed
Cv for 0.3 Test size is completed
Cv for 0.7 Test size is completed
Cv for 0.8 Test size is completed
Cv for 0.9 Test size is completed
```

In [132]:
```
np.mean(test_acc,axis=1)
```

Out[132]:
```
matrix([[0.70348405],
        [0.70205054],
        [0.69321774],
        [0.69047642],
        [0.69013683],
        [0.67326951]])
```

In [135]:
```
test_size=[0.1,0.2,0.3,0.7,0.8,0.9]
train_acc_mean = np.mean(np.array(train_acc),axis=1)
train_acc_std = np.std(np.array(train_acc),axis=1)
test_acc_mean = np.mean(np.array(test_acc),axis=1)
test_acc_std = np.std(np.array(test_acc),axis=1)
training_size = np.flip(np.array([int(i*len(forest)) for i in test_size]))
plt.figure(figsize=(12, 8), dpi=80)
plt.title('Training-Size vs 30-rounds Adaboost')
plt.grid()
plt.fill_between(training_size, train_acc_mean - train_acc_std,
                 train_acc_mean + train_acc_std, alpha=0.1,
                 color="r")
plt.fill_between(training_size, test_acc_mean - test_acc_std,
                 test_acc_mean + test_acc_std, alpha=0.1, color="g")
plt.plot(training_size, train_acc_mean, 'o-', color="r",
         label="Training score")
plt.plot(training_size, test_acc_mean, 'o-', color="g",
         label="CV accuracy")
plt.legend()
plt.xlabel('Training Size')
plt.ylabel('Accuracy')
```

```
Out[135]: Text(0, 0.5, 'Accuracy')
```



Here is the signature of over-fitting. As shown, when few data points are provided for training, there is an over-fitting where training score is high and test score is low, relatively. There is a significant gap between them compared to the situation where we provide more data points for training. Our aim is to get the red curve down and green curve up which we succeed by increasing the training-size.

We now call Adaboost-multiclass-Cv function to find the optimal number of T rounds for our dataset

In [26]:
```
cv_train,cv_test = Adaboost_Multi_CV(forest,k_folds=5,T_rounds=[30,60,90,130,180,270,350,450],
                                     label_col='Cover_Type',random_state=50)
```

```
Cv for 30 rounds is completed
Cv for 60 rounds is completed
Cv for 90 rounds is completed
Cv for 130 rounds is completed
Cv for 180 rounds is completed
Cv for 270 rounds is completed
Cv for 350 rounds is completed
Cv for 450 rounds is completed
```

In [64]:
```
np.mean(cv_test,axis=1)
```

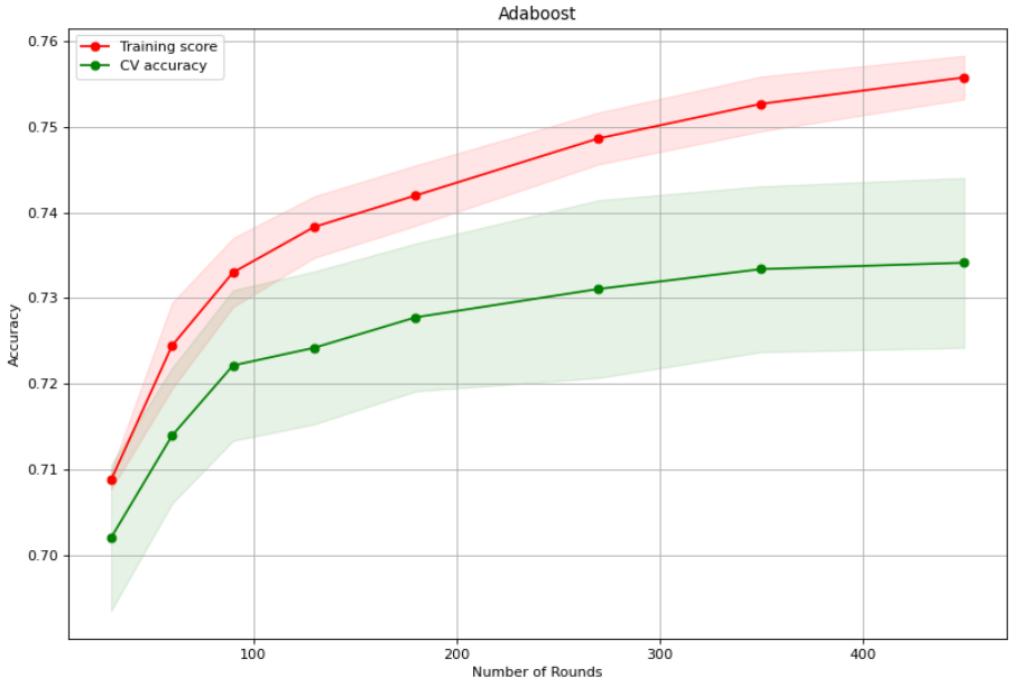
Out[64]:
```
matrix([[0.70205054],
        [0.71395563],
        [0.72213998],
        [0.72420692],
        [0.72776142],
        [0.7310682 ],
        [0.7333831 ],
        [0.73412724]])
```

In [33]:
```
np.mean(cv_train,axis=1)
```

Out[33]:
```
matrix([[0.70878801],
        [0.72445434],
        [0.73299026],
        [0.73830187],
        [0.74198084],
        [0.74863593],
        [0.75266622],
        [0.75574574]])
```

In [63]:
```
train_score_mean = np.mean(np.array(cv_train),axis=1)
train_score_std = np.std(np.array(cv_train),axis=1)
test_score_mean = np.mean(np.array(cv_test),axis=1)
test_score_std = np.std(np.array(cv_test),axis=1)
T_rounds = np.array([30,60,90,130,180,270,350,450])
plt.figure(figsize=(12, 8), dpi=80)
plt.title('Adaboost vs number of rounds')
plt.grid()
plt.fill_between(T_rounds, train_score_mean - train_score_std,
                 train_score_mean + train_score_std, alpha=0.1,
                 color="r")
plt.fill_between(T_rounds, test_score_mean - test_score_std,
                 test_score_mean + test_score_std, alpha=0.1, color="g")
plt.plot(T_rounds, train_score_mean, 'o-', color="r",
         label="Training score")
plt.plot(T_rounds, test_score_mean, 'o-', color="g",
         label="CV accuracy")
plt.legend()
plt.xlabel('Number of Rounds')
plt.ylabel('Accuracy')
```

Out[63]:
```
Text(0, 0.5, 'Accuracy')
```



Here is the training and cv accuracy with their standard errors. There is a nice increase in the test accuracy as we increase number of rounds, however, after 350 Adaboost rounds there is a negligible increase which has a very high computational cost compared to what it makes us gain.

Based on these findings, we try a final model with 350 adaboost rounds with test-size being %10 of the dataset and we get the following result:

In [96]:
```
models=[]
X_train,x_test,Y_train,y_test = train_test_split(forest,test_size=0.1,label_col='Cover_Type',random_state=50)
train_preds= np.mat(np.ones(shape=(np.shape(X_train)[0],len(np.unique(forest['Cover_Type'])))))
test_preds =  np.mat(np.ones(shape=(np.shape(x_test)[0],len(np.unique(forest['Cover_Type'])))))
for classes in range(len(np.unique(forest['Cover_Type']))):
    model,pred = Adaboost(X_train,np.mat(np.where(Y_train==int(classes)+1,1,-1)),350)
    models.append(model)
    train_preds[:,classes]=np.multiply(train_preds[:,classes],pred)
    test_est = Adaboost_Predict(x_test,model)
    test_preds[:,classes]=np.multiply(test_preds[:,classes],test_est)

train_predictions = np.argmax(train_preds,axis=1)+1 # because indexing starts from 0
training_error = np.where(train_predictions != np.mat(Y_train).T,1,0).sum()
test_prediction = np.argmax(test_preds,axis=1)+1
test_error = np.where(test_prediction != np.mat(y_test).T,1,0).sum()
print('test accuracy:',1-(test_error/len(x_test)))
print('training accuracy:',1-(training_error/len(X_train)))
```

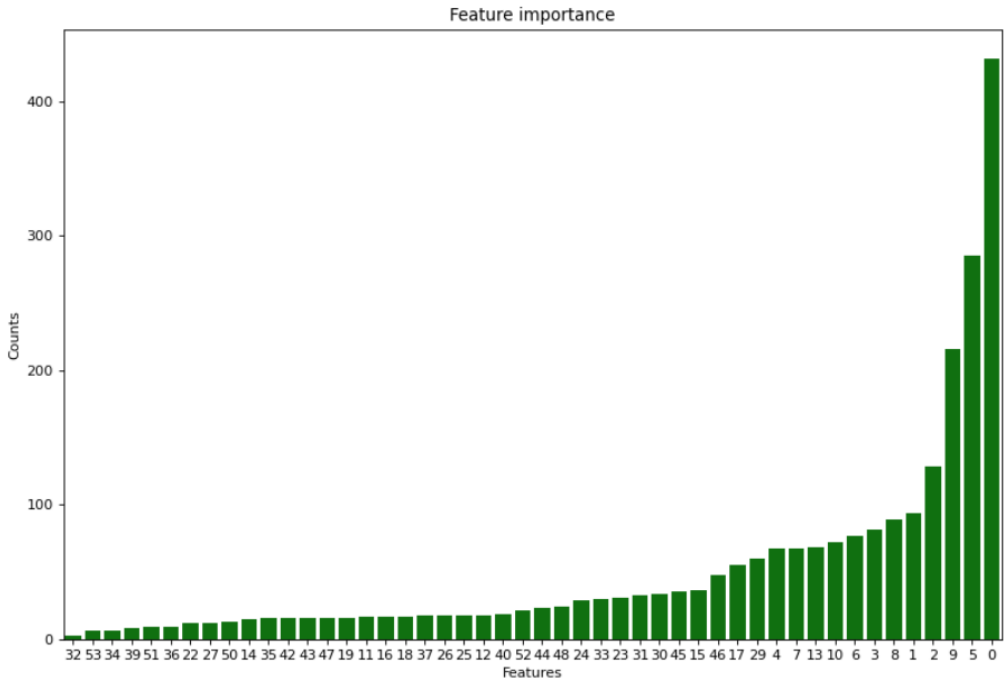
```
test accuracy: 0.7321428571428572
training accuracy: 0.7447089947089947
```

We got a result that is in the support of Cv results.

We can now have a look at the most used features namely the features that have the lowest weighted training error when choosen to be the best stump.

In [97]:
```
features = np.unique([d['dimen'] for model in models for d in model],return_counts=True)[0]
counts = np.unique([d['dimen'] for model in models for d in model],return_counts=True)[1]
counts = pd.DataFrame(counts)
features =pd.DataFrame(features)
frames = [features,counts]
dat_to_plot=pd.concat(frames,axis=1)
dat_to_plot.columns = ['dim','count']
dat_to_plot=dat_to_plot.sort_values(by='count')
figure(figsize=(12, 8), dpi=80)
sns.barplot(x='dim',y='count',data=dat_to_plot, order=dat_to_plot['dim'],color='g')
plt.title("Most used features")
plt.ylabel('Counts')
plt.xlabel('Features')
```

Out[97]:
```
Text(0.5, 0, 'Features')
```



The most used feature by far is the **elevation** feature followed by **horizontal distance to roadways,horizontal distance to fire points and slope**. We can confidently say these features are important in predicting the Cover-Type.

Comparison with Sklearn libraries

When using standard python libraries, we got %74 accuracy with $SAMME.R$ algorithm.

In [115]:
```
from sklearn.metrics import accuracy_score
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import train_test_split, cross_val_score, validation_curve, learning_curve,GridSearchC
from sklearn.multiclass import OneVsRestClassifier
data = pd.read_csv('forest-cover-type.csv')
X=data.drop(columns=['Cover_Type','Id'])
Y=data['Cover_Type']
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.1, random_state=50)


T_rounds = [30,60,90,130,180,270,350,450]
k_folds=5
Cv_test_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))

indices = np.array_split(list(X_train.index),k_folds)
for T in T_rounds:
    for k in  range(k_folds):
        ovo=OneVsRestClassifier(AdaBoostClassifier(n_estimators=T,algorithm='SAMME.R'))
        ovo.fit(X_train.drop(indices[k]),y_train.drop(indices[k]))
        y_pred = ovo.predict(X_train.loc[indices[k]])
        score = accuracy_score(y_train.loc[indices[k]],y_pred)
        Cv_test_acc[T_rounds.index(T),k]=score

np.mean(Cv_test_acc,axis=1)
```

Out[115]:
```
matrix([[0.71443263],
        [0.72913022],
        [0.73434754],
        [0.7381694 ],
        [0.73567069],
        [0.7375072 ],
        [0.73390642],
        [0.73148122]])
```

In [120]:
```
ovo=OneVsRestClassifier(AdaBoostClassifier(n_estimators=130,algorithm='SAMME.R'))
ovo.fit(X_train,y_train)
y_pred = ovo.predict(X_test)
score = accuracy_score(y_test,y_pred)
score
```

Out[120]:
```
0.7473544973544973
```

While with $SAMME$, we got %72 of accuracy.

```
In [121]: T_rounds = [30,60,90,130,180,270,350,450]
          k_folds=5
          Cv_test_acc = np.mat(np.ones(shape=(len(T_rounds),k_folds)))

          indices = np.array_split(list(X_train.index),k_folds)
          for T in T_rounds:
              for k in  range(k_folds):
                  ovo=OneVsRestClassifier(AdaBoostClassifier(n_estimators=T,algorithm='SAMME'))
                  ovo.fit(X_train.drop(indices[k]),y_train.drop(indices[k]))
                  y_pred = ovo.predict(X_train.loc[indices[k]])
                  score = accuracy_score(y_train.loc[indices[k]],y_pred)
                  Cv_test_acc[T_rounds.index(T),k]=score

          np.mean(Cv_test_acc,axis=1)

Out[121]: matrix([[0.67019473],
                  [0.68224569],
                  [0.69598772],
                  [0.70208715],
                  [0.70737867],
                  [0.71413919],
                  [0.7185485 ],
                  [0.71920951]])

In [122]: ovo=OneVsRestClassifier(AdaBoostClassifier(n_estimators=270,algorithm='SAMME'))
          ovo.fit(X_train,y_train)
          y_pred = ovo.predict(X_test)
          score = accuracy_score(y_test,y_pred)
          score

Out[122]: 0.7294973544973545
```

Therefore we can conclude that our results are robust.

Conclusion

In conclusion, when creating decision stump, we did not exhaustively searched for the best threshold since it has a high computational cost. We exhaustively used all the thresholds in the features where number of unique values is lower 100, otherwise we used the square root of the total number of unique values to create a step-size with which we jumped over unique values to create a threshold. This made the process way more faster. We then tried to see what is the optimal value for training and testing size. We saw that it was when %90 of the dataset is used for training. This result is approved with 5-fold cross validation. We then applied another 5-fold cross validation to find out the optimal number of boosting which comes with 350 boosting rounds. We saw that the best feature in predicting Cover-Type was **elevation**. After optimizing our model with hyperparameters, as a sort of robustness test, we compared our result with standard python libraries and got a similar result.