The AI-ML and Data Science Club of IITH

Bagging and Boosting techniques

Contents

Compiled by : Anshul Sangrame

1. What is Bootstrapping?

Let's first start by understanding what is Bootstrapping. This statistical technique consists in generating samples of size B (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement B observations.

2. What is Bagging?

The algorithm is explained in the following steps:

1) Given a dataset D with N training points and a training model
2) Create M bootstrap samples of D i.e $\{\tilde{D}_i\}_{i=1}^{M}$ with same number of training points i.e N.
3) Create M copies of untrained model $\{h_i\}_{i=i}^{M}$ and train each h_i on $\tilde{D}_i$
4) Let y_i be predicted value of mode h_i. Then the final predicted value will be $y = \frac{1}{M}\sum_{i=1}^{M} y_i$

3. How Bagging works?

Now that we know the algorithms, we will try to understand how does it work. But for that, we need some knowledge of bias and variance.

3.1 Bias and variance

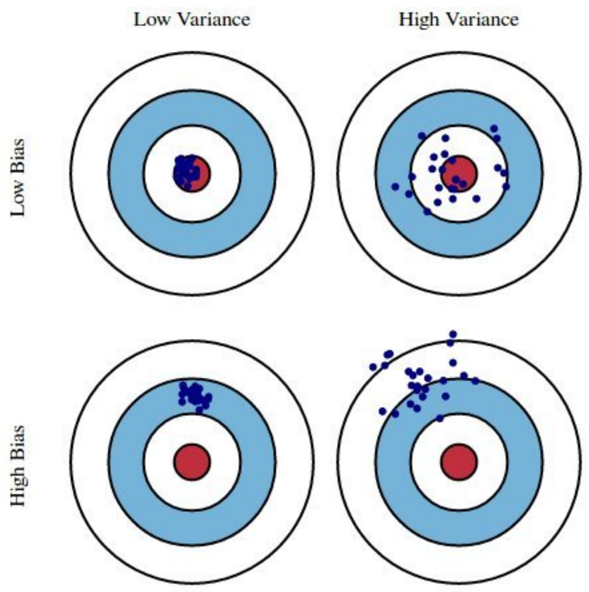
Let $\hat{y}$ be the predicted target value of a random input X and y be the actual target value. y_* be the best-predicted target value of input X that can be made. We can show that,

$$E[(\hat{y} - y)^2] = (y_* - E[\hat{y}])^2 + Var[\hat{y}] + Var[y]$$

We can interpret the following terms in the above equation:

1) $E[(\hat{y} - y)^2]$ is the expected loss of the predicted value. We need to minimize this.
2) $(y_* - E[\hat{y}])^2$ is called the bias. It indicates how close is the predicted value to the best prediction that can be achieved. Higher bias corresponds to underfitting.
3) $Var[\hat{y}]$ is called the variance. It indicates the amount of variability in the predictions (a higher value corresponds to overfitting).
4) $Var[y]$ is called the Bayes error and is the inherent unpredictability of the targets. We cannon reduce this.

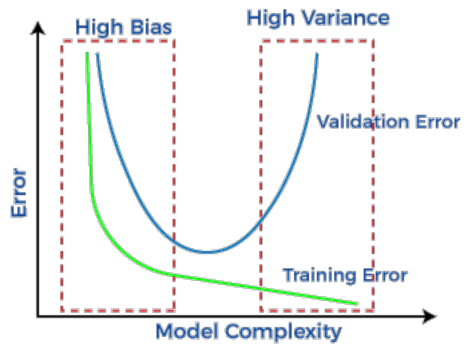
For a better understanding refer to the image below:



Other useful observations related to bias and variance are:

1) High bias indicates that the model is less complex and is not trained sufficiently(i.e. training error is high) which indicates underfitting.
2) High variance indicates that the model is more complex and is over-trained (i.e. training error is low) which indicates overfitting.

The above two points are portrayed in the following image:



3.2 How Bagging reduces variance?

In the previous section, we saw different terms in the expected loss function. Now we will try to understand how each term is affected in bagging.

1) Bayes error: Unchanged, since we have no control over it.
2) Bias: Unchanged

$$\begin{aligned} E[\hat{y}] &= E[\frac{1}{M}\sum_{i=1}^{M}\hat{y}_i] \\ &= \frac{1}{M}\sum_{i=1}^{M}E[\hat{y}_i] \\ &= E[\hat{y}_1] \end{aligned}$$

3) Variance: Reduced, since we're averaging over independent samples

$$\begin{aligned} Var[\hat{y}] &= Var[\frac{1}{M}\sum_{i=1}^{M}\hat{y}_i] \\ &= \frac{1}{M^2}\sum_{i=1}^{M}Var[\hat{y}_i] \\ &= \frac{1}{M}Var[\hat{y}_1] \end{aligned}$$

3.3 Effect of Correlation

Till now we assumed that the data are independent and then we saw the variance decreases by a factor of M. However, if the data are dependent on each other, the prediction made by each model is also not independent. Hence, we cannot claim the same. In fact, Variance is given by the following formula:

$$Var[\frac{1}{M}\sum_{i=1}^{M}\hat{y}_i] = \frac{1}{M}(1-\rho)\sigma^2 + \rho\sigma^2$$

Where ρ is the correction factor and σ is the standard deviation.

3.4 Random Forests

Random Forest is a bagging method with a learning model as decision trees. In addition, while selecting a bootstrap dataset for each decision tree, it chooses a random set of features on which the decision tree will split. This additional trick will ensure all predictions made by decision trees are not dependent.

Random forests are probably the best black-box machine learning algorithm. They often work well with no tuning whatsoever and are the most widely used algorithm in Kaggle competition.

4. What is boosting?

Boosting methods work in the same spirit as bagging methods: we build a family of models that are aggregated to obtain a strong learner that performs better. However, unlike bagging which mainly aims at reducing variance, boosting is a technique that consists of fitting sequentially multiple weak learners in a very adaptative way: each model in the sequence is fitted giving more importance to observations in the dataset that were badly handled by the previous models in the sequence. In this way, the bias is lowered.

The base model in boosting is the starting weak learning model. The base model is often a high-bias and low-variance model because boosting mainly focuses on reducing bias (and maybe increasing the variance as boosting can cause overfitting). Boosting algorithms also use weighted dataset which is discussed below.

4.1 Weighted dataset

Till now the loss function was giving equal treatment to all data points i.e

$$Loss = \frac{1}{M}\sum_{i=1}^{M} L_i(\hat{y}_i, y_i)$$

The key idea of having a weighted dataset is that the learning algorithm can give more focus on data points with higher weights. In this way, we can control which data points should the learning algorithm put focus on. This is used in boosting since we can increase the weights of misclassified data points so that the next weak learner focuses on it.

4.2 Different type of boosting algorithm

There are mainly 5 types of boosting algorithms:

1) AdaBoost (Adaptive Boosting)
2) GBM (Gradient Boosting Machine)
3) XGBM (Extreme Gradient Boosting Machine)
4) LightGBM
5) CatBoost

4.3 AdaBoost

An adaptative boosting (often called 'Adaboost'), we try to define our ensemble model as a weighted sum of L weak learners:

$$H_T(X) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

α_t and h_t are chosen such that we minimize the fitting loss which is given as follows:

$$\alpha_t, h_t = \arg\min_{\alpha_t, h_t} \frac{1}{N}\sum_{n=1}^{N} loss(y^{(n)}, H_{T-1}(x^{(n)}) + \alpha_t h_T(x^{(n)}))$$

We need to solve the above optimization problem. Here in this example, we will consider an exponential loss function given by:

$$Loss(y, \hat{y}) = \exp(-y\hat{y})$$

Let $\mathbb{L}(h(X^{(n)}) \neq y^{(n)}) = \frac{1}{2}(1-h(X^{(n)}).y^{(n)})$. Using this we can simplify the optimization problem as follow:

$$\begin{aligned}
\alpha_t, h_t &= \arg\min_{\alpha_t, h_t} \sum_{n=1}^{N} \exp(-y^{(n)}(H_{T-1}(x^{(n)}) + \alpha_t h_T(x))) \\
&= \arg\min_{\alpha_t, h_t} \sum_{n=1}^{N} \exp(-y^{(n)}(H_{T-1}(x^{(n)}))) \exp(-y^{(n)}\alpha_t h_T(x^{(n)})) \\
&= \arg\min_{\alpha_t, h_t} w_T^{(n)} \exp(-y^{(n)}\alpha_t h_T(x^{(n)}))
\end{aligned}$$

Solving the optimization problem we get

$$\alpha_t = \frac{1}{2} \log \frac{1 - err_t}{err_t}$$

$$\text{Where, } err_t = \frac{\sum_{n=1}^{N} w_t^{(n)} \mathbb{L}(h_t(X^{(n)}) \neq y^{(n)})}{\sum_{n=1}^{N} w^{(n)}}$$

With that, we also find that h_t minimizes the weighted 0/1-loss i.e.

$$h_t = \arg\min_{h} \sum_{i=0}^{N} w_t^{(i)} \mathbb{L}(h(X^{(i)}) \neq y^{(i)})$$

We also find the relationship between weights as well:

$$w_{t+1}^{(n)} = w_t^{(n)} \exp(-y^{(n)}\alpha_t h_t(x^{(n)}))$$

The algorithm is given as follows:

```
Input: Data $D$ with $N$ data points, a family of weak classifier, number of iterations T
Initialize all weights ($w \in \mathbb{R}^N$) to $\frac{1}{N}$
for $t = 1 \dots T$ do
    Train the classifier $h_t$ on the weighted dataset.
    Compute weighted error as follows:
        $err_t \leftarrow \frac{\sum_{n=1}^{N} w^{(n)} \mathbb{L}(h_t(X^{(n)}) \neq y^{(n)})}{\sum_{n=1}^{N} w^{(n)}}$
    Compute the classifier coefficient as follows:
        $\alpha_t \leftarrow \frac{1}{2} \log \frac{1 - err_t}{err_t}$
    Update data weights as follows:
        $w^{(n)} \leftarrow w^{(n)} \exp(-\alpha_t y^{(n)} h_t(X^{(n)}))$
end
Output: $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$
```

Algorithm 1: AdaBoost

4.4 Implementing AdaBoost

The full working code can be found on this link. The AdaBoost model is made completely using NumPy. We have used sci-kit-learn to import only the decision tree that can find the best split.

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier

class AdaBoost(object):
    def __init__(self) -> None:
        self.hypothesis = None
        self.hypothesis_weights = None

    def train(self,X, y, num_iteration):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        y: ndarray of shape (num,)
        num_itteration: int
        --------------------------------

        Output:
        --------------------------------
        hist: list
        --------------------------------
        """
        n = X.shape[0]
        w = np.full(n, 1/n)
        self.hypothesis = []
        self.hypothesis_weights = []
        hist = []
        for t in range(num_iteration):
            # training a model
            stump = DecisionTreeClassifier(max_depth=1)
            stump.fit(X,y,w)
            self.hypothesis.append(stump)
            # finding predicted value
            y_pred = stump.predict(X)
            # finding wieghted error
            err = np.sum(w*(y_pred != y))/np.sum(w)
            # finding alpha
            alpha = (1/2) * np.log((1-err)/err)
            self.hypothesis_weights.append(alpha)
            #updating value of w
            w = w*np.exp(-alpha*y*y_pred)
            # conputing the loss
            loss = np.mean(w)
            hist.append(loss)
        return hist

    def predict(self,X):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        --------------------------------
        Output:
        --------------------------------
        y: ndarray of shape (Num,)
        --------------------------------
        """
        num = X.shape[0]
        y = np.zeros(num)
        for alpha,h in zip(self.hypothesis_weights,self.hypothesis):
            y += alpha*(h.predict(X))
        pos = (y >= 0).astype(mystyle"mystyleintmystyle")
        neg = (y < 0).astype(mystyle"mystyleintmystyle")
        y = pos - neg
        return y
```

4.5 GBM

Just like AdaBoost, we try to define our ensemble model as a weighted sum of L weak learners. Compared to AdaBoost, gradient boosting does not penalize missed-classified cases but uses a loss function instead. Similar to the gradient descent algorithm, gradient boosting casts the problem into a gradient descent one i.e. at each iteration, we fit a weak learner to the opposite of the gradient of the current fitting error concerning the current ensemble model. To put this mathematical perspective, the ensemble can be written as:

$$H_T(x) = H_{T-1}(x) - \alpha \nabla_{H_{T-1}} E(H_{T-1})(x)$$

Where $E(.)$ is the fitting error which can be written as:

$$E(H_T)(x) = \frac{1}{N} \sum_{n=1}^{N} loss(y^{(n)}, H_T(x^{(i)}))$$

The coefficient α is the learning rate that can be computed following a one-dimensional optimization process (line-search to obtain the best step size α). Basically, unlike Adaboost there is no analytic expression of α. Because of this, the gradient descent approach can more easily be adapted to a large number of loss functions. Thus, gradient boosting can be considered as a generalization of AdaBoost to arbitrary differentiable loss functions.

The algorithm for the regression problem with MSE as a loss function is given below:

```
Input: $\alpha$, $T$, Data D with N training points, family of weak learner
Train a weak learner $h_1$ on original data
$\forall i \in (1 \dots N)$ $r_i \leftarrow -\nabla_{h_1(x_i)} loss(y_i, h_1(x_i))$
$\implies \forall i \in 1 \dots N$ $r_i \leftarrow y_i - h_1(x_i)$
for $t = 2$ to $T$ do
    Train data $(x_1, r_1) \dots (x_N, r_N)$ on weak learner $h_t$
    $\forall i \in (1 \dots N)$ update $r_i$ as follows
    $r_i \leftarrow r_i - \alpha h_t(x_i)$
end
Output: Final model $H_T(x) = h_1(x) + \alpha \sum_{t=2}^{T} h_t(x)$
```

Algorithm 2: Gradient boosting

4.6 Implementing Gradient Boosting

The full working code can be found on this link. The GBM model is made completely using NumPy. We have used sci-kit-learn to import only the decision tree that can find the best split.

```python
import numpy as np
from sklearn.tree import DecisionTreeRegressor

class GBM(object):
    def __init__(self) -> None:
        self.alpha = None
        self.weak_learners = None

    def train(self,X,y,num_itr,alpha = 1):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        y: ndarray of shape (num,)
        num_itteration: int
        --------------------------------

        Output:
        --------------------------------
        hist: list
        --------------------------------
        """
        # Initialization
        self.weak_learners = []
        self.alpha = alpha
        hist = []
        # fitting data
        stump = DecisionTreeRegressor(max_depth=1)
        stump.fit(X,y)
        self.weak_learners.append(stump)
        # calculating residual
        r = y - stump.predict(X)
        # calculating loss
        loss = np.mean(r**2)
        hist.append(loss)
        for i in range(1,num_itr):
            # fitting data
            stump = DecisionTreeRegressor(max_depth=1)
            stump.fit(X,r)
            self.weak_learners.append(stump)
            # calculating residual
            r = r - alpha*stump.predict(X)
            # calculating loss
            loss = np.mean(r**2)
            hist.append(loss)
        return hist

    def predict(self,X):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        --------------------------------

        Output:
```

```
        --------------------------------
        y: ndarray of shape (Num,)
        --------------------------------
        """
        for i,h in enumerate(self.weak_learners):
            if i == 0:
                y = h.predict(X)
                continue
            y += self.alpha*h.predict(X)

        return y
```

4.7 XGBM

XGBoost is an enhanced version of the gradient boosting method. Firstly, it improves overfitting by using regularisation. Secondly, it improves the runtime speed by optimizing sorting using parallel running. It is also a state-of-the-art model. In this handout, we will discuss the mathematical aspect of gradient boosting with L2 regularisation. We can write the ensemble as

$$H_T(X_i) = \hat{y}_i^{(T)} = H_{T-1}(X_i) + h_T(X_i)$$

The loss function can be written as:

$$L^{(T)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(T-1)} + h_T(X_i)) + \Omega(h_T)$$

Where Ω is regularisation. Using L2 regularisation and applying Taylor expansion we get,

$$L^{(T)} = \sum_{j=1}^{T} \left[\left(\sum_{i\in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i\in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T$$

Where,

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

$$h_i = \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$$

The minimum value of loss found is

$$L^{(T)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T + const$$

We need a split such that the decrease in this loss function increases. To formulate this mathematically, let I_R and I_L be the instance sets of left and right nodes after a split such that $I_R \cup I_L = I$. The loss reduction after the split is given by,

$$L_{split} = \frac{1}{2} \left[\frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} + \frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda} \right] - \gamma$$

We choose a split such that we increase L_{split} and hence it will be our splitting criteria when creating a decision stump.

```
Input: Instance of node I, $g_i$,$\lambda$, Number of features: m
$L_{split} \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$
$H \leftarrow len(I)$
for $k$ = 1 to $m$ do
    $G_L \leftarrow 0$
    $H_L \leftarrow 0$
    for $j$ in sorted(I, by $x_{jk}$) do
        $G_L \leftarrow G_L + g_j$
        $H_L = H_L + 1$
        $G_R \leftarrow G - G_L$
        $H_R \leftarrow H - H_L$
        $L_{split} \leftarrow max(L_{split}, \frac{G_R^2}{H_R+\lambda} + \frac{G_L^2}{H_L+\lambda} - \frac{G^2}{H+\lambda})$
    end
end
Output: Split with maximum $L_{split}$
```

Algorithm 3: Exact Greedy Algorithm for Split Finding with MSE as loss function

```
Input: Data, $\gamma$, $\alpha$, $\lambda$
Fit the discussion stump $h_1$ on the data.
$\forall i \in 1 \dots N \; g_i \leftarrow y_i - h_1(x_i)$
for $t$ = 2 to $T$ do
    Train data $(x_1, y_1) \dots (x_N, y_N)$ on decision stump $h_t$ using values of $g_i$ and $\lambda$
    $\forall i \in (1 \dots N)$ update $g_i$ as follows
    $g_i \leftarrow g_i + \alpha h_t(x_i)$
end
Output: Final model $H_T(x) = h_1(x) + \alpha \sum_{t=2}^{T} h_t(x)$
```

Algorithm 4: extreme Gradient boosting

4.8 Implementing XGBoost

The XGBM model and decision stump are made from scratch using NumPy.

```python
import numpy as np

class DescisionStump:
    def __init__(self,g = None,reg = 0) -> None:
        """
        Input
        -------------------------------
        g: ndarray of shape (Num,)
        reg: int
        -------------------------------
        """
        self.g = g
        self.reg = reg
        self.right = None
        self.left = None
        self.threshold = None
        self.right_weight = None
        self.left_weight = None
        self.feature_criteria = None
    def split(self,X,y):
        gain = 0
        G = np.sum(self.g)
        H = self.g.shape[0]
        m = X.shape[1]
        bestGR = None
        bestGL = None
        for k in range(m):
            GL = 0
            HL = 0
            for j in np.argsort(X[:,k]):
                GL += self.g[j]
                HL += 1
                GR = G - GL
                HR = H - HL
                Lsplit = GR**2/(HR + self.reg) + GL**2/(HL + self.reg) - G
                   **2/(H + self.reg)
                if (gain > Lsplit):
                    gain = Lsplit
                    self.feature_criteria = k
                    self.threshold = j
                    self.left, self.right = np.split(X,[j+1])
                    bestGR = GR
                    bestGL = GL

        self.left_weight = bestGL/(self.left_weight.shape[0] + self.reg)
        self.right_weight = bestGR/(self.right_weight.shape[0] + self.reg)

    def fit(self,X,y):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        y: ndarray of shape (Num,)
        --------------------------------
        """
        if self.g is None:
            self.g = np.random.randn(X.shape[0])
        self.split(X,y)

    def predict(self,X):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        --------------------------------

        Output:
        --------------------------------
        y_pred: ndarray of shape (Num,)
        --------------------------------
        """
        right_mask = X[:,self.feature_criteria] > self.threshold
        left_mask = ~right_mask
        y_pred = np.empty(X.shape[0])
```

```python
        y_pred[right_mask] = self.right_weight
        y_pred[left_mask] = self.left_weight

        return y_pred


class XGBM:
    def __init__(self) -> None:
        self.hypothesis = None
        self.alpha = None

    def train(self,X,y,reg,alpha,num_itr):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        y: ndarray of shape (num,)
        num_itteration: int
        --------------------------------
        """
        # initialization
        self.hypothesis = []
        self.alpha = alpha
        # training
        stump = DescisionStump(reg=reg)
        stump.fit(X,y)
        # updating g
        g = stump.predict(X) - y
        for i in range(1,num_itr):
            # Training data
            stump = DescisionStump(reg=reg,g=g)
            stump.fit(X,y)
            self.hypothesis.append(stump)
            # updating g
            g += alpha*stump.predict(X)

    def predict(self,X):
        """
        Input
        --------------------------------
        X: ndarray of shape (Num,features)
        --------------------------------

        Output:
        --------------------------------
        y: ndarray of shape (Num,)
        --------------------------------
        """

        for i,h in enumerate(self.hypothesis):
            if i == 0:
                y = h.predict(X)
                continue
            y += self.alpha*h.predict(X)
        return y
```

4.9 Comparison of different boosting algorithms

Points you can consider for choosing boosting algorithm

1) AdaBoost:
   (a) Focuses on misclassified cases.
   (b) It forms the foundation of boosting algorithm.
2) GBM:
   (a) You can use any type of loss function.
   (b) Gradient descent is used to minimize the loss function.
3) XGBoost
   (a) Improves on overfitting
   (b) Optimize running time by tree parallelism and tree pruning
4) LightGBM
   (a) further improves the speed of leaf-wise growth
   (b) Allow tuning of more parameter
5) CatBoost:
   (a) Handles categorial features automatically.
   (b) Works efficiently with categorial type data.

5. Pros and cons of boosting

1) Pros of boosting and cons of bagging:
   (a) boosting reduces the bias whereas bagging doesn't decrease bias.
   (b) In bagging we need to make sure predictions made by models are independent which is sometimes difficult in practice. Whereas, boosting doesn't need to worry about this.
2) Pros of bagging and cons of boosting:
   (a) Since boosting is a sequential algorithm, it does not support parallel programming. whereas bagging can support parallel programming.
   (b) Bagging decreases the variance whereas boosting can increase variance due to overfitting.

6. Questions

Subjective :

1) Can you justify briefly that the optimal prediction made for an input x is $y_* = E[y|x]$?
2) Can you prove $E[(\hat{y} - y)^2|x)] = (E[t|x] - \hat{y})^2 + Var[t|x]$?
3) Using the equation proved in question 2 can you answer question 1 again?
4) Can you prove $E[(\hat{y} - y)^2)] = (y_* - \hat{y})^2 + Var(\hat{y}) + Var(t)$?
5) Many machine libraries make use of parallel programming. Will libraries be more efficient in boosting algorithms or bagging algorithms? Justify
6) In Adaboost we constructed a optimization problem, can you show that $h_t(x)$ is the minimizer of the weighted 0/1-loss?
7) In bagging if the predictions of models are dependent then what will be the variance in terms of the Correlation factor (ρ) and standard deviation of one prediction(σ)?

Objective :

1) which of the following will be a good choice for the base model in boosting?
a) SVM **b)** Neural network with 3 hidden layer **c)** decision stump **d)** decision tree