

Lecture 9: January 1, 2016

Lecturer: Yishay Mansour

Scribe: Yishay Mansour

9.1 Weak and Strong Learners

In the PAC model there is a distribution D on a domain X . Random examples $\langle x, c^*(x) \rangle$ are drawn according to the distribution D and labeled using the target function $c^* \in C$. The goal of the learner is to find a hypothesis $h \in H$ such that $\text{error}(h, c^*) \leq \epsilon$, with probability $1 - \delta$. This is a *strong learning model*, since ϵ and δ can be arbitrarily small.

Recall that ϵ is the error rate of the algorithm and $1 - \delta$ represents the confidence. However, suppose we have an algorithm with low error rate but also low confidence, say confidence 50%, or alternatively an algorithm with an error rate of 49% (slightly better than flipping a coin) but high confidence level.

Is it possible to drive those *weak* algorithms to be *strong learners*? Intuitively, it is easier to find hypothesis that is correct only 51 percent of the time, rather than a hypothesis that is correct 99 percent of the time.

9.1.1 Boosting the confidence ($1 - \delta$)

Suppose algorithm A returns with probability $1 - \delta \geq \frac{1}{2}$ a hypothesis h such that $\text{error}(h, c^*) \leq \epsilon$. An interesting question is whether it is possible to build a PAC learning algorithm A' (from A)? The answer is positive.

Algorithm *BoostConfidence*(A):

1. Run A for $k = \log \frac{2}{\delta}$ times (on fresh sample S_i each time) with parameter $\epsilon' = \frac{\epsilon}{3}$.
2. Algorithm A on input S_i outputs hypotheses h_i , so we have hypotheses h_1, \dots, h_k .
3. Draw a new sample S of size $m = \frac{9}{\epsilon^2} \ln \frac{4k}{\delta} = O\left(\frac{1}{\epsilon^2} \ln \frac{k}{\delta}\right)$ and for each hypothesis h_i compute its error on S , i.e., the observed error $\widehat{\text{error}}(h_i)$.
4. Return $\hat{h}^* = \arg \min_{h_i} (\widehat{\text{error}}(h_i(S)))$.

Analysis of Algorithm $\text{BoostConfidence}(A)$

After the first stage of the algorithm, we would like at least one hypotheses h_i to have error at most $\epsilon/3$. With probability at most $(\frac{1}{2})^k$, $\forall i : \text{error}(h_i) > \frac{\epsilon}{3}$. Hence, with probability at least $1 - (\frac{1}{2})^k$, $\exists i : \text{error}(h_i) \leq \frac{\epsilon}{3}$.

Therefore, if we set $k = \log_{\frac{2}{\delta}}$, then with probability $1 - \frac{\delta}{2}$ for at least one of h_1, \dots, h_k we have $\text{error}(h_i) \leq \frac{\epsilon}{3}$. Denote by h_+ this hypothesis.

Now we will show that after the second stage of the algorithm $\text{BoostConfidence}(A)$, with probability $1 - \frac{\delta}{2}$, outputs the hypothesis \hat{h}^* (with minimum errors on S) such that,

$$\text{error}(\hat{h}^*) \leq \frac{\epsilon}{2} + \min_i(\text{error}(h_i)) \leq \epsilon \quad .$$

Proof: First, we use the Chernoff Bound to bound the probability for “bad” event, i.e., the difference between the empirical error of any h_i and its real error is greater than $\frac{\epsilon}{3}$:

$$\Pr[|\widehat{\text{error}}(h_i) - \text{error}(h_i)| \geq \frac{\epsilon}{3}] \leq 2e^{-(\frac{\epsilon}{3})^2 m}$$

Second, we will bound by $\frac{\delta}{2}$ the probability that such bad event will happen to any of the k hypothesis h_i using a Union Bound:

$$2ke^{-(\frac{\epsilon}{3})^2 m} \leq \frac{\delta}{2} \quad .$$

Then, by isolating m , we will get:

$$\begin{aligned} \frac{1}{e^{\frac{\epsilon^2}{9} m}} &\leq \frac{\delta}{4k} \\ \frac{4k}{\delta} &\leq e^{\frac{\epsilon^2}{9} m} \\ \ln \frac{4k}{\delta} &\leq \frac{\epsilon^2}{9} m \\ \frac{9}{\epsilon^2} \ln \frac{4k}{\delta} &\leq m \quad . \end{aligned}$$

We have that for a sample of size at least m , with probability $1 - \frac{\delta}{2}$, for each of those h_i :

$$|\widehat{\text{error}}(h_i) - \text{error}(h_i)| < \frac{\epsilon}{3}$$

thus,

$$\text{error}(h^*) < \widehat{\text{error}}(h^*) + \frac{\epsilon}{3} \quad .$$

From the first stage of the algorithm we already know that

$$\widehat{\text{error}}(h_+) < \text{error}(h_+) + \frac{\epsilon}{3} < \frac{2\epsilon}{3} \quad .$$

Since $\widehat{\text{error}}(h_+) \geq \widehat{\text{error}}(h^*)$, and we conclude that, with probability $1 - \delta$, we have $\text{error}(\hat{h}^*) \leq \epsilon$. □

9.1.2 Boosting the accuracy (ϵ)

One question we can ask: given an algorithm that outputs hypothesis with $\epsilon = \frac{1}{2}$, can we drive it to learn PAC? The answer is No, because such an algorithm will do exactly like flipping a coin. Therefore we will need a stronger hypothesis.

Definition: Weak learning

Algorithm A learns Weak-PAC a concept class C with H if:

$\exists \gamma > 0$,

$\forall c^* \in C$, (target function)

$\forall D$, (distribution)

$\forall \delta < \frac{1}{2}$,

With probability $1 - \delta$, algorithm A outputs an hypothesis $h \in H$ such that $error(h) \leq \frac{1}{2} - \gamma$.

Intuitively, A will guarantee an error rate of 49% instead of 1% of the PAC model. We show, that if a concept class has a weak learning algorithm, then there is a PAC learning algorithm for the class.

Note that running A multiple times on the same distribution D , does not work because A might return the same hypothesis over and over again.

Example

Suppose we have the following target function c^* (over bits) with a uniform distribution D :

if $x_1 = x_2 = 0 \implies c^*(x) = \text{some very hard function}$

otherwise $\implies c^*(x) = 0$

(e.g., the hardness depends on the first and the second bits.)

We can easily achieve 87.5% accuracy by flipping a coin if $x_1 = x_2 = 0$ and otherwise predicting zero. The probability for the event $x_1 = x_2 = 0$ is 0.25 which gives us a total accuracy of 87.5%. On the other hand, getting better than 87.5% accuracy is hard. For this reason we want our weak learner to perform well with any distribution D ! (In the example a natural distribution is $x_1 = x_2 = 0$ and uniform otherwise).

Conclusion: An important requirement in the weak learning model is: *for any distribution* (in the example we assumed a specific distribution).

9.2 Three Weak Learners

9.2.1 Algorithm Description

Let A be a weak learning algorithm, and p the error probability of A .

Step 1: Run A with the initial distribution D_1 to obtain h_1 (*error* $\leq \frac{1}{2} - \gamma$).

Step 2: Define a new distribution D_2 , such that

$$\begin{aligned} S_c &= \{x | h_1(x) = c^*(x)\} \\ S_e &= \{x | h_1(x) \neq c^*(x)\} \\ D_2(S_c) &= D_2(S_e) = \frac{1}{2} \end{aligned}$$

To do so we will define D_2 as follows:

$$D_2(x) = \begin{cases} \frac{0.5}{1-p} \cdot D_1(x) & x \in S_c \\ \frac{0.5}{p} \cdot D_1(x) & x \in S_e, \end{cases}$$

where $p = D_1(S_e)$. For simplicity we assume that all the weak learners have error $p = 1/2 - \gamma$. To obtain h_2 we will run A with D_2 . In order to sample from D_2 , at each step with select a random bit b . If $b = 0$, we sample x from D_1 until we find an x for which $h_1(x) = c^*(x)$. If $b = 1$, we sample until $h_1(x) \neq c^*(x)$.

Step 3: The distribution D_3 would be defined only on examples x for which $h_1(x) \neq h_2(x)$:

$$D_3(x) = \begin{cases} \frac{D_1(x)}{Z} & h_1(x) \neq h_2(x) \\ 0 & \text{otherwise,} \end{cases}$$

where $Z = P[h_1(x) \neq h_2(x)]$. To obtain h_3 we will run A with D_3 . In order to sample from D_3 , we sample x -s from D_1 until we get $h_1(x) \neq h_2(x)$.

Our combined hypothesis would be:

$$\bar{h}(x) = \begin{cases} h_1(x) & h_1(x) = h_2(x) \\ h_3(x) & \text{otherwise} \end{cases}$$

Which is equivalent to $\bar{h}(x) = MAJ(h_1(x), h_2(x), h_3(x))$.

9.2.2 Estimation of the Error

Suppose each hypothesis h_i errs with a probability of p , independently. What would be the error of the majority of h_1, h_2, h_3 ?

$$\text{Error} = 3p^2(1 - p) + p^3 = 3p^2 - 2p^3 = p^2(3 - 2p)$$

We would like to show that this is the error probability without assuming the hypotheses are independent. To do so we would partition the space into four subspaces:

$$\begin{aligned} S_{cc} &= \{x | h_1(x) = c^*(x) \wedge h_2(x) = c^*(x)\} \\ S_{ee} &= \{x | h_1(x) \neq c^*(x) \wedge h_2(x) \neq c^*(x)\} \\ S_{ec} &= \{x | h_1(x) \neq c^*(x) \wedge h_2(x) = c^*(x)\} \\ S_{ce} &= \{x | h_1(x) = c^*(x) \wedge h_2(x) \neq c^*(x)\} \end{aligned}$$

Let $P_{cc} = D_1(S_{cc})$, $P_{ee} = D_1(S_{ee})$, $P_{ce} = D_1(S_{ce})$ and $P_{ec} = D_1(S_{ec})$.

The error probability, with respect to the initial distribution D_1 , is $P_{ee} + (P_{ec} + P_{ce})p$.

Let us define $\alpha = D_2(S_{ce})$. Therefore, from the definition of D_2 , in terms of D_1 we get $P_{ce} = 2(1 - p)\alpha$.

Since $D_2(S_{*e}) = p$, we have,

$$\begin{aligned} D_2(S_{ee}) &= p - \alpha \\ P_{ee} &= 2p(p - \alpha). \end{aligned}$$

From the construction of D_2 , since $D_2(S_{e*}) = D_2(S_{ee}) + D_2(S_{ec}) = 1/2$, we have

$$\begin{aligned} D_2(S_{ec}) &= \frac{1}{2} - (p - \alpha) \\ P_{ec} &= 2p\left(\frac{1}{2} - p + \alpha\right). \end{aligned}$$

Therefore the error is:

$$P_{ee} + (P_{ec} + P_{ce})p = 2p(p - \alpha) + p\left(2p\left(\frac{1}{2} - p + \alpha\right) + 2(1 - p)\alpha\right) = 3p^2 - 2p^3$$

One can now build a recursive construction to derive an arbitrary PAC learner.

9.3 Adaptive Boosting - AdaBoost

The AdaBoost algorithm is an iterative boosting algorithm that enables us to create a strong learning algorithm from a weak learning algorithm. The general idea of this algorithm is to maintain a distribution on the input sample, and increase the weight of the harder to classify examples so the algorithm would focus on them.

9.3.1 Algorithm Description

Input: A set of m classified examples: $S = \{\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_m, y_m \rangle\}$ where $y_i \in \{-1, 1\}$. A set H of weak classifiers.

Definitions: Let D_t denote the distribution of weights of the examples at time t , and $D_t(i)$ the weight of example x_i at time t .

Initialization:

$$D_1(i) = \frac{1}{m} \quad \forall i \in \{1, \dots, m\}$$

Step: At each iteration we use a classifier $h_t \in H : X \mapsto \{-1, +1\}$ that minimizes the error on the current distribution (defined as $\epsilon_t = \Pr_{D_t}[h_t(x) \neq c^*(x)]$ where c^* is the target function). At time $t + 1$ we update the weights in the following manner:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & y_i = h_t(x_i) \\ e^{\alpha_t} & y_i \neq h_t(x_i) \end{cases} \\ &= \frac{D_t(i)}{Z_t} \cdot e^{-y_i \alpha_t h_t(x_i)} \end{aligned}$$

where Z_t is a normalizing factor to keep D_{t+1} a distribution and $\alpha_t = \frac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}$.

Output: The hypothesis we return after running the algorithm for T iterations is:

$$\tilde{h}(x) = \text{Sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right)$$

An advantage using the AdaBoost algorithm is that it removes the need of knowing the parameter γ . Another advantage is that it is easy to implement and runs efficiently.

9.3.2 Bounding the Error

Theorem 9.1 *Let \tilde{h} be the output hypothesis of AdaBoost. Then:*

$$\begin{aligned} \widehat{\text{error}}(\tilde{h}) &\leq \prod_{t=1}^T 2\sqrt{\epsilon_t(1-\epsilon_t)} \\ &= \prod_{t=1}^T \sqrt{1-4\gamma_t^2} \\ &\leq e^{-2\sum_t \gamma_t^2} \end{aligned}$$

where the last inequality is obtained from the inequality $1 + x \leq e^x$.

Conclusion: The training error drops exponentially fast in T for a constant $\gamma_t = \gamma$.

Proof: The proof follows in three steps:

1. First, obtain the following expression for $D_{T+1}(i)$:

$$D_{T+1}(i) = \frac{D_1(i)e^{-y_i f(x_i)}}{\prod_t Z_t}$$

where $f(x) = \sum_{t=1}^T \alpha_t h_t(x)$.

Proof: Since $D_{t+1}(i)$ is given by:

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} e^{-y_i \alpha_t h_t(x_i)}$$

we can unravel the recurrence to obtain:

$$\begin{aligned} D_{T+1}(i) &= D_1(i) \prod_{t=1}^T \frac{e^{-y_i \alpha_t h_t(x_i)}}{Z_t} \\ &= D_1(i) \frac{e^{-y_i \sum_{t=1}^T \alpha_t h_t(x_i)}}{\prod_{t=1}^T Z_t} \\ &= D_1(i) \frac{e^{-y_i f(x_i)}}{\prod_t Z_t} \end{aligned}$$

□

2. Second, we bound the training error of \hat{h} by the product of the normalizing factors Z_t :

$$\widehat{error}(\hat{h}) \leq \prod_{t=1}^T Z_t$$

Proof:

$$\begin{aligned}
 \widehat{error}(\mathbf{h}) &= \frac{1}{m} \sum_{i=1}^m I(y_i \neq \mathbf{h}(x_i)) \\
 &= \frac{1}{m} \sum_{i=1}^m I(y_i f(x_i) \leq 0) \\
 &\leq \frac{1}{m} \sum_{i=1}^m e^{-y_i f(x_i)} \\
 &= \frac{1}{m} \sum_{i=1}^m m \left(\prod_{t=1}^T Z_t \right) D_{T+1}(i) \\
 &= \left(\prod_{t=1}^T Z_t \right) \sum_{i=1}^m D_{T+1}(i) \\
 &= \prod_{t=1}^T Z_t,
 \end{aligned}$$

where I is the indicator function. The third line follows from the observation that when $I(y_i f(x_i) \leq 0) = 1$, then $y_i f(x_i) \leq 0$ and so $e^{-y_i f(x_i)} \geq 1 = I(y_i f(x_i) \leq 0)$. (Also, clearly when $I(y_i f(x_i) \leq 0) = 0$, then $e^{-y_i f(x_i)} \geq 0$). The fourth line follows from step 1. The last line is obtained from the fact that D_{T+1} is a probability distribution over the examples.

□

3. Now that the training error has been bounded in step 2 by the product of the normalizing weights Z_t , the last step is to express Z_t in terms of ϵ_t :

$$Z_t = 2\sqrt{\epsilon_t(1 - \epsilon_t)}$$

Proof: By definition,

$$\begin{aligned}
 Z_t &= \sum_{i=1}^m D_t(i) e^{-y_i \alpha_t h_t(x_i)} \\
 &= \sum_{i: y_i = h_t(x_i)} D_t(i) e^{-\alpha_t} + \sum_{i: y_i \neq h_t(x_i)} D_t(i) e^{\alpha_t} \\
 &= (1 - \epsilon_t) e^{-\alpha_t} + \epsilon_t e^{\alpha_t},
 \end{aligned}$$

where the last step follows from the definition of ϵ_t :

$$\sum_{i: y_i \neq h_t(x_i)} D_t(i) = \epsilon_t,$$

Since the expression above for Z_t is valid for all α_t , minimizing Z_t with respect to α_t for each t will produce the minimum training error $\widehat{error}(\bar{h})$.

$$\frac{\partial Z_t}{\partial \alpha_t} = -(1 - \epsilon_t)e^{-\alpha_t} + \epsilon_t e^{\alpha_t} = 0$$

Solving, we find:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right).$$

□

Using this value of α_t in the expression for Z_t , and then plugging that into the bound on the training error for \bar{h} , we end up with:

$$\widehat{error}(\bar{h}) \leq \prod_{t=1}^T \left(2\sqrt{\epsilon_t(1 - \epsilon_t)} \right)$$

which proves the theorem. □

9.4 Ensemble Methods

9.4.1 Basic idea and rationale

The idea behind ensemble methods is very simple, combine multiple hypotheses to build a (hopefully) more accurate hypothesis. The two most important questions that we should answer, when considering a specific ensemble method are the following:

1. How do we generate the multiple hypotheses. Recall that we have only one sample, so we need to specify how to use a single sample to generate multiple hypotheses.
2. How do we combine the hypotheses. Once we have multiple hypotheses we need to specify how do we combine them to a single prediction. The most natural is a majority rule or a weighted majority rule, but many other alternatives exist.

There are a few basic reasons why ensemble methods can offer an advantage. (See also Figure 9.1.)

1. **Statistical reasoning:** The sample does not have enough information to tightly specify the target hypothesis. There is a significant uncertainty about which hypothesis is the “right” one. One can envision that we have multiple good hypotheses to better specify the target function. This reason stresses the fact that we have limited amount of data.

2. **Computational reasons:** Many times we have to resort to heuristic methods when searching for the best hypotheses given the data, given computational constraints. The back-propagation algorithm for neural networks is an excellent example of this computational limitation. This suggests that generating multiple hypotheses might give a better approximation of the target function. This reason stresses the fact that many times we have a limited amount of computation power.
3. **Representation reasons:** If the space of hypotheses is not convex, then averaging multiple hypotheses might represent a hypothesis which is outside our hypotheses class. This is true even with infinite data and unlimited computational power.

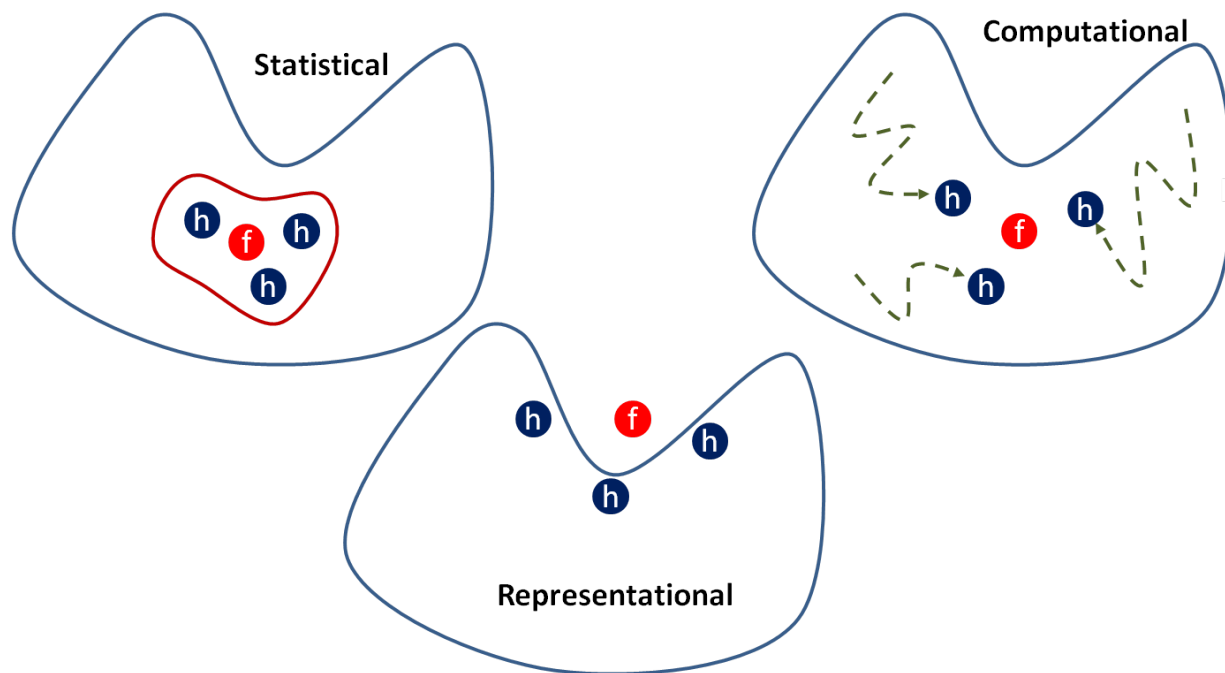


Figure 9.1: Reasons for using ensemble methods (Source: <http://web.engr.oregonstate.edu/~tgd/publications/mcs-ensembles.pdf>)

9.4.2 Boosting

Boosting is actually an ensemble method. In boosting we are generating different hypotheses by changing the sample distribution, and reweighing the examples. Based on the weak-learner hypotheses we are guarantee to generate different hypotheses.

In boosting we combine different the hypotheses using weighted linear threshold. The coefficient of the different hypotheses are determine when the weak learners are selected.

9.4.3 Bias versus variance tradeoff

Much of the reasoning behind the ensemble methods can be traced to the *bias* versus *variance* tradeoff. To understand the tradeoff, it is best to consider the Mean Square Error (MSE).

Consider a point x and let $f(x)$ be the true label and $h(x)$ be the predicted one, where h is a random function (which depends on the sample, for example). We can rewrite the MSE as:

$$MSE(x) = E_h[(f(x) - h(x))^2] = bias^2 + variance$$

where we define $bias = E[f(x) - h(x)]$ and $variance = VAR(h(x)) = E[h^2(x)] - E^2[h(x)]$.

We can see that for small sample we will have a high *bias* (not enough sample to fit well) and low *variance*. For a very large sample we will have a low *bias* (fitting the data well) while having high *variance* (due to the many points). The tradeoff would have a point which will minimize the sum (which is exactly the MSE).

9.4.4 Bagging

The idea behind bagging is to generate different samples and use them to learn different hypotheses. The problem is that many times we do not have a huge data set and we like to make the best use of it. For this reason we will sample the data set multiple times to generate the different hypotheses.

More precisely, in *bagging* the inputs are a single learning algorithm A and a sample S . We perform the following steps:

1. Given S we generate sub-samples S_1, \dots, S_k by selecting items from S with repetitions. (The repetitions guarantee that each example by itself has the same distribution as the underlying distribution.)
2. Given S_1, \dots, S_k , we run A on each S_i and learn a hypothesis h_i .
3. We combine the different hypotheses using a simple majority.

A good question is *why are we making progress*. Note that the expected error of each generated h_i is identical (before sampling S_i), and slightly higher than learning on S (since we use less examples). This error can be viewed as a *bias* which is inherent between the target function and the learned hypotheses, given the sample.

The main gain in averaging is done by reducing the *variance*. The variance of a error single hypothesis fluctuates considerably. In contrast, the majority of many hypotheses is much more stable. This suggests that we should be getting better generalization bounds.

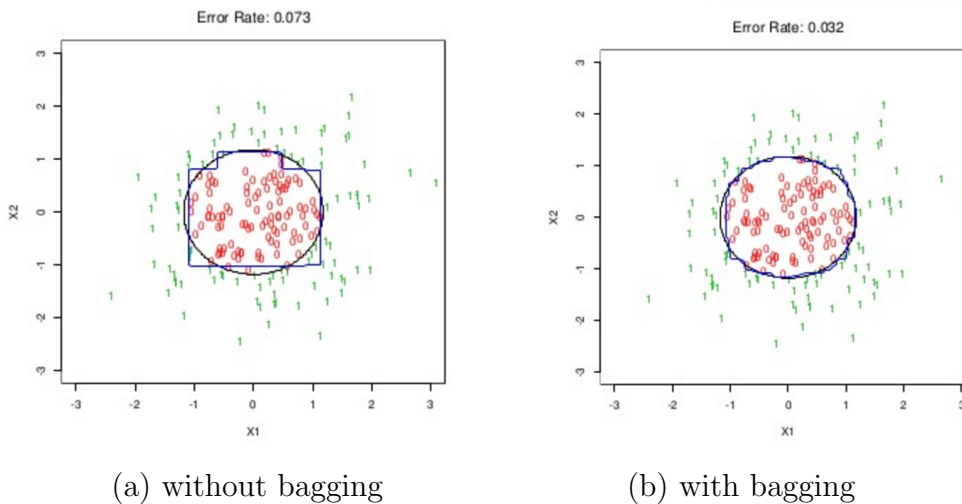


Figure 9.2: Comparing the output of decision tree with and without bagging on same data set. (source: <http://www.slideshare.net/0xdatagbm-27891077>)

9.4.5 Stacking

We like a general methodology of combining multiple hypotheses. For example we might learn a decision tree, a large margin classifier and AdaBoost. How can we combine them? Majority is only one option!

Stacking give a general methodology of combining multiple hypotheses. The input is as sample S , combining algorithm C , and k learning algorithms A_1, \dots, A_k . We run the staking procedure as follows.

1. We run algorithm A_i on S to generate hypotheses h_i .
2. Given h_1, \dots, h_k , we build a new sample S' , such that for any $(x, y) \in S$ we have $((h_1(x), \dots, h_k(x)), y) \in S'$.
3. We run C on S' to generate the hypotheses H .

We can view bagging as a special case of stacking where A_i sub-samples S and C is a simple majority. Similarly, boosting can be also viewed as a special case of stacking, where A_i generates the i th weak hypotheses, and H is the weighted majority using the coefficients α_i . Random forest, presented in the next section, is another example of stacking.

9.4.6 Random Forest

Decision trees construction is highly sensitive to the sample. A tiny change in the sample can generate a different predicate in the root, which will result in a completely different decision

tree. In bagging we tried to overcome this problem by generating multiple decision tree and taking their majority. In the original paper a sub-sample of about 66% was observe to be the best in those experiments. This is one source of generating different decision tree.

Another source of generating different decision tree, is selecting which subset of attributes will be considered in a given node. There is a parameter M which controls how many of the N attributes we will consider. When $M = N$ we consider all attributes, and we are back to the regular decision tree algorithm. For $M < N$ we select a random subset of M attributes, and consider only them as candidates for the current node. We select the attribute that minimizes the splitting index function. Note that in different nodes we select different subsets randomly! Also, note that for $M = 1$ we simply generate a random tree. A reasonable setting of M is \sqrt{N} (or even $\log N$).

To combine the multiple decision trees, similar to bagging, we are using a simple majority rule.

The main benefits of the random forest are: (1) Fast to run, (2) Easy to parallelize, and (3) Competitive performance with leading machine learning algorithms (AdaBosot and SVM).

The main weaknesses are: (1) Random forest losses the interpretability of decision trees, (2) Many parameters around that need to be tuned, and (3) Feature selection collides with sampling attributes.

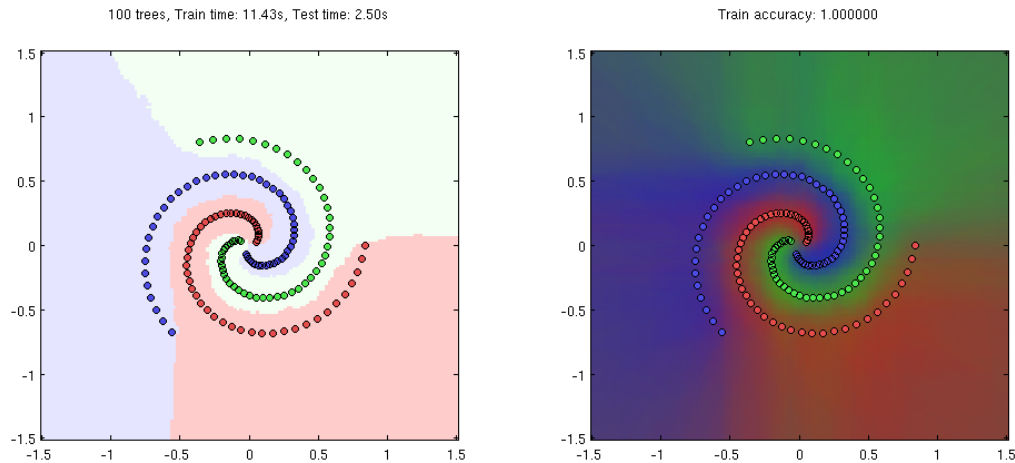


Figure 9.3: Classification of spiral using decision forest (source: <http://cs.stanford.edu/people/karpathy/randomForestSpiral.png>)