5.1 Multiple Experts - Motivation:
Consider a situation in which you have several experts for a certain task. However, you don't have any prior information on the reliability of the different experts. We want to use the expert's guesses to predict, and hope to converge to the performance of the best expert.

Example:
A gambler, frustrated by persistent horse-racing losses and envious of his friend's winnings, decides to allow a group of his fellow gamblers to make bets on his behalf. He decides he will wager a fixed sum of money in every race, but that he will apportion his money among his friends based on how well they are doing. Certainly, if he knew psychically ahead of time which of his friends would win the most, he would naturally have that friend handle all his wagers. Lacking such clairvoyance, however, he attempts to allocate each race's wager in such a way that his total winnings for the season will be reasonably close to what he would have won had he bet everything with the luckiest of his friends.

5.1.1 Types of Multiple Experts
We can define the following types of experts

- A single expert on full observation space.

- A single expert for sub regions of observation space. That is, we divide the observation space into sub regions, one expert to each region and each point in the observation space get classification from a single expert.

- Multiple experts on full observation space and each point in the observation space can get classifications from several experts

- Multiple experts on sub regions of observation space and each point in the observation space can get classifications from multiple experts.

5.1.2 Types of Multiple Experts Training

- Use full observation space for each expert
- Use different observation features for each expert
- Use different observations for each expert
- Combine of the above
5.1.3 Online expert selection: the Online Model

In each step we have full information about the past, and no information about the future. That is, in each step we know exactly whether an expert was correct in his guess, or was wrong.

The model includes:
1. N different strategies (experts)
2. In step t, the learner A chooses a distribution $P_t$ on the N strategies. Such that $\sum_{i=1}^{N} P_{i}^{'} = 1$ where $P_{i}^{'} \geq 0$ is the amount allocated to strategy i.
3. After each step t, the learner receive the loss $l_{i}^{'}$ of each expert (denote by the vector $l_{t}$) which is determined by the (possibly adversarial) "environment".
4. The loss of A at time t is then $\sum_{i=1}^{N} P_{i}^{'} l_{i}^{'} = P^{'} \cdot l^{'}$ i.e., the average loss of the strategies with respect to A's chosen allocation rule.

The goal of algorithm A is to minimize its cumulative loss relative to the loss suffered by the best strategy. That is, A attempts to minimize its net loss

$$L_{A} = \min_{i} L_{i}$$

That is, $L_{A} = \sum_{t=1}^{T} P_{t} l_{t}^{'}$ is the total cumulative loss suffered by algorithm A on the first $T$ trails and the cumulative loss of the i-th expert is given by $L_{i} = \sum_{t=1}^{T} l_{i}^{'}$.

Without the loss of generality we can assume that $l_{i}^{'} \in [0,1]$ that is, the loss suffered by any strategy is bounded and of course the loss of A is at least the loss of the best expert.

5.1.4 Application example: Rock-paper-scissors

Two players play a game. Player 1 chooses a row, player 2 chooses a column. Each of the players has 3 different strategies (Experts). The loss of player 1 is as shown on the table below. Given that player 2 chooses his strategies in random (nature), no matter what will be the strategy of player 1, he will loose $n/2$ in $n$ games.

If on the other hand, player 2 has one fixed strategy, then player 1 must choose its best corresponding strategy.

Player 2

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>0.5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>S</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 1: loss of player 1
5.1.5 First Experts Algorithm: The Greedy Algorithm

Let $L_i$ be the cumulative loss of the $i$-th expert and let $L'_i = \sum_{j=1}^{t} I_{ij}$. That is, the loss of the $i$-th expert at step $t$, (time of the last prediction).

**Greedy**: choose the expert that achieve the minimum loss until time $t$. i.e. the algorithm decision at step $t+1$ is $\arg\min_i L'_i$

**Theory**: Let $L^T_G$ be the loss of the Greedy algorithm at time $T$. Than:

$$L^T_G \leq N(\min_i L^T_i + 1)$$

Where $N$ denotes the number of experts.

**Proof**: Let $G'_i$ be the loss of the greedy algorithm that was incurred when the algorithm choose the expert $i$.

$$G'_i = \begin{cases} G''_{i+1} + I'_i & i = \arg\min_k L''_{k-1} \\ G''_{i-1} & \text{else} \end{cases}$$

Clearly, $\forall i, L'_i \leq L^T_G$. Now we ready to show that:

$$\forall t, \min_i (L'_i) + 1 \leq G'_j \leq \min_i (L'_i) + 1$$

We consider three different cases regarding the last time the algorithm choose the $j$th expert.

1. At time $t$ i.e. $i=j$

$$G'_j = G'_i \leq L'_i \leq L'_i + 1$$

2. The algorithm never has chosen the $j$th expert

$$G'_j = 0 \leq L'_i \leq L'_i + 1$$

3. The last time the algorithm chooses the $j$th expert was at time $k$. Notice that the $j$th expert had the minimal loss at that time. Therefore,

$$\forall i, L''_{i-k} \leq L''_{i-k}$$

$$G'_j - k \leq L''_{i-k} \leq L''_{i-k} < L''_{i-k} + 1$$

Since the $j$th expert was not chosen for the last $k$ time steps

$$G'_j - k = G'_j$$

$$G'_j = G'_i - k \leq L''_{i-k} + 1 \leq L''_{i-k} + 1 + \sum_{n=t-k+1}^{t} I''_n = L'_i$$
\[
L_{G}^{T} = \sum_{j=1}^{N} G_{j}^{T} \leq \sum_{j=1}^{N} \min(L_{j}^{T}) + 1 \leq N(\min(L_{j}^{T}) + 1)
\]
Thus:

5.1.6 Second Experts Algorithm: The Hedge Algorithm

Next we present a better algorithm for multiple experts, (better bound).
At each step \( t \), the algorithm maintains a positive weight vector \( w_t \) for the experts.

\[
p' = \frac{w'}{\sum_{i=1}^{N} w'_i}
\]

Let \( P_t \) denote the probability vector where
\[
\forall i \ w_i^t = \frac{1}{N}
\]
Initialization:

Update: after receiving \( l_t \), we update the weights as follows:
\[
w_{i}^{t+1} = w_{i}^{t} U_{\beta}(l_{i}^{t})
\]
Where \( U_{\beta} : [0,1] \to [0,1] \), such that for any \( r, \beta \in [0,1] \), we have
\[
\beta r \leq U_{\beta}(r) \leq 1 - (1 - \beta)r
\]

Motivation: Let us try to motivate the update rule. Assume that \( U_{\beta}(r) = \beta r \) then
\[
w_{i}^{1} = w_{i}^{t-1} \beta^{L_{i}^{t-1}} = \ldots = w_{i}^{1} \beta^{L_{i}^{t-1}}
\]
Note, when an expert is wrong, his weight decreased and he will have a smaller effect on the following guesses. If we have one (or several) good experts, his weight in the final guess will dominate and the performance of the algorithm will be close to that best expert. Unlike the previous algorithm, the predictions will not be dominated by a single expert at each step but by the set of almost bests experts.

5.1.7 Analysis of Hedge

**Lemma:** for any sequence of loss vectors \( l_{1} \ldots l_{T} \), we have
\[
\ln\left(\sum_{i=1}^{N} w_{i}^{T+1}\right) \leq -(1 - \beta)L_{Hedge(\beta)}
\]

**Proof:** choose a certain time \( t \), and then we obtain:
\[
\sum_{i=1}^{N} w_{i}^{t+1} = \sum_{i=1}^{N} w_{i}^{t} U_{\beta}(l_{i}^{t}) \leq \sum_{i=1}^{N} w_{i}^{t} (1 - (1 - \beta)\ell_{i}^{t}) = (\sum_{i=1}^{N} w_{i}^{t})(1 - (1 - \beta)p'l')
\]
(For the last equality we use the identity \( w_{i}^{t} = p'_{i} (\sum_{j} w_{j}^{t}) \))
Using the inequality $\ln(1-x) \leq -x$ and taking the log of the expression we get:

$$\ln(\sum w_i^{t+1}) \leq \ln(\sum w_j^t) + \ln(1 - (1 - \beta) p^t l^t) \leq \ln(\sum w_i^t) - (1 - \beta) p^t l^t$$

Hence, the growth of the log of the weight in each step, is bounded by $-(1 - \beta) p^t l^t$

Therefore,

$$\ln(w_i^{t+1}) \leq \ln(\sum w_i^t) - (1 - \beta) \sum p^t l^t = -(1 - \beta) L_{Hedge(\beta)}$$

The lemma implies that:

$$L_{Hedge(\beta)} \leq -\frac{\ln(\sum_{i=1}^N w_i^{t+1})}{1 - \beta}$$

Since the weights are positive for any i, we get that:

$$\sum w_i^{r+1} \geq w_j^{r+1} \geq w_j^t \beta^{t_j} = \frac{1}{N} \beta^{t_j}$$

Therefore,

$$\forall i \quad L_{Hedge} \leq -\frac{\ln(\frac{1}{N} \beta^{L_i})}{1 - \beta} = \frac{\ln N + L_i \ln \frac{1}{\beta}}{1 - \beta}$$

Since the last equation holds for all of the experts, it holds in particularly for the best expert, where the best expert is the one whose loss is minimal. Let $k$ denote the best expert, and then we get:

$$L_{Hedge} \leq \frac{\ln N + L_k \ln \frac{1}{\beta}}{1 - \beta}$$

Having bounded the loss of the hedge algorithm in terms of the loss of the best expert all that is left is to set $\beta$ that provides the best bound.

If $\beta = \frac{1}{2}$, we get the bound of $2\ln N + (\ln 2)\text{mini}(L_i)$ which is much better then what we have had (A fairly weak dependency on the number of experts). We can choose $\beta$ to get even a better bound.
5.1.8 How to Choose $\beta$

Lemma:
For $0 \leq L \leq L'$ and $0 \leq R \leq R'$ if we choose $\beta = g\left(\frac{L'}{R'}\right)$

$$g(z) = \frac{1}{1 + \sqrt{2z}}$$

then

$$\frac{-L \ln \beta + R}{1 - \beta} \leq L + \sqrt{2L'R'} + R$$

Using the above lemma, we can choose $L' =$ maximum-loss and $R' = \ln N$ and for $\beta = g\left(\frac{L'}{\ln N}\right)$, we get that $L^A \leq \min(L_i) + \sqrt{2L'\ln N + \ln N}$

5.2 Weak and Strong Learners

Suppose a learning algorithm can do a little bit better than random, that is, its error rate is less than 50%, can we take this error rate and drive it down to zero?

We start by repeating the (standard) definition of PAC learning (strong learning) and then introduce the notion of weak-learning.

Definition 1: $C$ is learnable if $\exists$ algorithm $A$ such that, $\forall c \in C, \forall D, \forall \epsilon > 0, \forall \delta > 0$ whenever $A$ is given $m = poly\left(\frac{1}{\epsilon}, \frac{1}{\delta}\right)$ examples $(x_1, c(x_1)), ..., (x_m, c(x_m))$ $A$ returns $h$ for which

$$\Pr_D[err(h) > \epsilon] \leq \delta.$$

Definition 2: $C$ is weakly learnable if $\exists \gamma > 0, \exists$ algorithm $A$ such that, $\forall c \in C, \forall D, \delta > 0$ whenever $A$ is given $m = poly\left(\frac{1}{\delta}\right)$ examples $(x_1, c(x_1)), ..., (x_m, c(x_m))$ $A$ returns $h$ for which

$$\Pr_D\left[err(h) > \frac{1}{2} - \gamma\right] \leq \delta.$$

A weak learner can be trivial. For example if given a sample set of more than 60% positive examples, a learner can output a hypothesis that always predicts positive, this way the sample error rate is slightly smaller than 50%. However, such a concept class is not weakly learnable since the latter is defined on any distribution of examples. Thus, if we use an equalized distribution (mass of positive examples is equal to the mass of negative examples), then clearly such a hypothesis cannot reach an error different than 50%.

Many researchers have investigated the technique of combining the predictions of multiple classifiers to produce a single classifier. The resulting classifier (referred to as ensemble) is generally more accurate than any of the individual classifiers making up the ensemble. Two popular methods for creating ensembles are
Bagging (Breiman 1996) and Boosting (Freund & Schapire 1996). These methods rely on “resampling” to obtain different training sets for each of the classifiers.

5.2.1 Bagging
Generate a random sample from training set by selecting $m' < m$ elements with replacement. This procedure is repeated to get $k$ “independent” training sets. Each of these bootstrap data sets is used to train a different component classifier $h_1, h_2, \ldots, h_k$. The final classification decision is based on the vote of each component classifier. The global decision rule in bagging can be a simple vote among the component classifiers $H(x) = \text{MAJ}(h_1(x), \ldots, h_k(x))$. Traditionally the component classifiers are of the same general form — i.e., all hidden Markov models, or all neural networks, or all decision trees — merely the final parameter values differ among them due to their different sets of training patterns.

A classifier/learning algorithm combination is informally called unstable if “small” changes in the training data lead to significantly different classifiers and relatively “large” changes in accuracy. For example, decision tree classifiers trained by a greedy algorithm can be unstable — a slight change in the position of a single training point can lead to a radically different tree. In general, bagging improves recognition for unstable classifiers since it effectively averages over such discontinuities, although there are no convincing theoretical derivations or simulation studies proving it.

5.2.2 Boosting
Boosting is a general method for improving the accuracy of any given learning algorithm.

The idea of Boosting has its roots in PAC learning. Kearns and Valiant proved that learners, each performing only slightly better than random, can be “boosted” and combined into an arbitrarily accurate “strong” learning algorithms (when enough data is available).
5.2.3 AdaBoost

AdaBoost is a boosting algorithm, running a given weak learner several times on slightly altered training data, and combining the hypotheses to one final hypothesis, in order to achieve higher accuracy than the weak learner's hypothesis would have.

**In detail:**

**Input:**

- A set $S=\{(x_1, y_1), \ldots, (x_m, y_m)\}$ of classified examples, where $x_i \in X$ and $y_i \in Y$, for $i = 1, \ldots, m$. The following code is for the case of discrete binary hypotheses - we assume $Y = \{-1,+1\}$.

- A weak learning algorithm that can deal with weighted example sets. Such a learning algorithm reads an example set $S$ and a distribution $D$. In the simplest case, where all hypotheses that can be output are functions $h(x) \rightarrow \{-1,+1\}$, the algorithm tries to find a hypothesis $H$ with minimal probability of misclassification, given that an example is drawn from $X$ with respect to $D$.

**The algorithm:**

Let $w_i$ denote the weight of example $i$ in iteration $t$.

- Initialization: Assign each example $(x_i, y_i) \in S$ the weight $w_i^t = \frac{1}{m}$.
- For $t = 1$ to $T$:
  1. Call the weak learning algorithm with example set $S$ and weights given by $w_i^t$.
  2. Get a weak hypothesis $h_t : X \rightarrow \{-1,+1\}$
  3. Compute the error of $h_t$ on the sample: $\varepsilon_t = \Pr_{i \sim w^t} [h_t(x_i) \neq y_i] = \sum_{i : h_t(x_i) \neq y_i} w_i^t$
  4. Choose $\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \varepsilon_t}{\varepsilon_t} \right)$ which denotes the importance of hypothesis $h_t$. 


Update the weights of all examples:

\[
    w_{t+1}^{i} = \frac{w_{t}^{i}}{Z_{t}} \times \begin{cases} 
        e^{-\alpha_{t}} & \text{if } h_{t}(x_{i}) = y_{i} \\
        e^{\alpha_{t}} & \text{if } h_{t}(x_{i}) \neq y_{i}
    \end{cases}
\]

(5) Update the weights of all examples:

\[
    = \frac{w_{t}}{Z_{t}} \exp(-\alpha_{t} y_{i} h_{t}(x_{i}))
\]

where \(Z_{t}\) is a normalization factor.

- Output the final hypothesis \(H\): \(X \rightarrow \{-1, +1\}\), generated from the hypotheses of rounds 1 to \(T\):

\[
    H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_{t} h_{t}(x) \right)
\]

The main idea of AdaBoost is to assign each example of the given training set a weight. At the beginning all weights are equal. In each iteration the weak learner returns a hypothesis, and the weights of all examples classified wrong by that hypothesis are increased. That way the weak learner is forced to focus on the difficult examples of the training set.

Once the weak hypothesis \(h_{t}\) has been received, AdaBoost chooses a parameter \(\alpha_{t}\) as in the algorithm described above. Intuitively, \(\alpha_{t}\) measures the importance that is assigned to \(h_{t}\). Note that \(\alpha_{t} \geq 0\) if \(\varepsilon_{t} \leq \frac{1}{2}\) (which we can assume - \(h_{t}\) is a weak learner – without the loss of generality), and that \(\alpha_{t}\) gets larger as \(\varepsilon_{t}\) gets smaller.

The final hypothesis is a combination of the hypotheses of all iterations, namely a weighted majority vote, where hypotheses with lower classification error have higher weight.

5.2.4 Analyzing the training error of AdaBoost

The most basic theoretical property of AdaBoost concerns its ability to reduce the training error. The empirical loss theorem can be stated as follows.

**Theorem:** Suppose that in each step \(t\) a weak learner generated hypothesis \(h_{t}\). The error of each hypothesis is denoted by \(\varepsilon_{t}\). Given \(T\) errors \(\varepsilon_{1}, \ldots, \varepsilon_{T}\), the error \(\varepsilon\) of \(H\) is at most:
\[ \varepsilon \leq 2 \prod_{i=1}^{\mathcal{R}} \sqrt{\varepsilon_i (1 - \varepsilon_i)} \]

Let us write the error \( \varepsilon_i \) of \( h_i \) as \( \frac{1}{2} - \gamma_i \). Since a hypothesis that guesses each instance’s class at random has an error rate of \( \frac{1}{2} \) (on binary problems), \( \gamma_i \) thus measures how much better than random are \( h_i \)'s predictions. Based on the theorem it is simple to show that,

\[ \varepsilon \leq 2 \prod_{i=1}^{\mathcal{R}} \sqrt{\varepsilon_i (1 - \varepsilon_i)} \]
\[ = \prod_i \sqrt{1 - 4\gamma_i^2} \]
\[ \leq \exp(-2\sum_i \gamma_i^2). \]

Furthermore if each weak hypothesis is slightly better than random so that \( \gamma_i \geq \gamma \) for some \( \gamma > 0 \), then the training error drops exponentially fast. \( \varepsilon \leq \exp(-2\gamma^2\mathcal{T}). \)

Specifically, if \( \mathcal{T} > \frac{1}{2\gamma^2} \ln m \), then \( \varepsilon(H) < \frac{1}{m} \), which implies the training error will be zero. Note the theorem does not have any assumptions on \( h_i \) and where samples are form. We are going to prove the theorem in 3 steps.

1. Show that \( \omega_\mathcal{T+1} = \exp(-y_i f(x_i)) \frac{\prod_{\mathcal{T}} Z_{\mathcal{T}}}{m} \) where \( f(x_i) = \sum_{i=1}^{\mathcal{R}} \alpha_i h_i(x_i) \)

   \textbf{Proof:}

   \[ \omega_{\mathcal{T+1}} = \frac{w_{\mathcal{T}} \cdot \exp(-\alpha_{\mathcal{T}, y_i h_\mathcal{T}(x_i))}{Z_\mathcal{T}} = \]
   \[ = \frac{w_{\mathcal{T}} \cdot \exp(-\alpha_{\mathcal{T}, y_i h_\mathcal{T-1}(x_i)) \cdot \exp(-\alpha_{\mathcal{T}, y_i h_\mathcal{T}(x_i))}{Z_{\mathcal{T}-1} \cdot Z_\mathcal{T}} = \]
   \[ \vdots \]
   \[ = 1 \cdot \frac{\exp(-y_i \cdot \sum_{i=1}^{\mathcal{R}} \alpha_i h_i(x_i))}{\prod_i Z_i} \]

2. Show that \( \varepsilon(H) \leq \prod_i Z_i \)

   \textbf{Proof:}
Here, \( \Pi \) is 1 if \( \Pi \) is true, and 0 otherwise. Note the third expression follows because for each term that \( y_i f(x_i) \leq 0 \), the corresponding \( e^{-y_i f(x_i)} \) is greater than 1 while for each term \( y_i f(x_i) > 0 \) the term \( e^{-y_i f(x_i)} \) is greater than zero. The fourth expression comes from step 1 above.

3. Finally we show that \( Z_t \leq 2 \sqrt{\varepsilon_t (1 - \varepsilon_t)} \)

Proof:

\[
Z_t = \sum_{i=1}^{m} w_i^t \cdot e^{-\alpha_i y_i h_t(x_i)} = \sum_{i: h_t(x_i) \neq y_i} w_i^t \cdot e^{\alpha_i} + \sum_{i: h_t(x_i) = y_i} w_i^t \cdot e^{-\alpha_i} =
\]

\[
= \varepsilon_t \cdot e^{\alpha_t} + (1 - \varepsilon_t) \cdot e^{-\alpha_t}
\]

The value \( \alpha_t \) was in fact chosen so that \( Z_t \) is minimized. In this case,

\[
\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \varepsilon_t}{\varepsilon_t} \right)
\]

and

\[
Z_t = 2 \sqrt{\varepsilon_t (1 - \varepsilon_t)}
\]

Note the third expression follow the definition of weighted training error.

5.2.5 AdaBoost – Characteristics

Practically, AdaBoost has many advantages. It is fast, simple and easy to program. It has no parameters to tune (except for the number of iterations \( T \)). It requires no prior knowledge about the weak learner and so can be flexibly combined with any method for finding weak hypotheses. There is no over fit mechanism. Finally, Instead of trying to design a learning algorithm that is accurate over the entire space, we can focus we can find weak learning algorithms that only need to be better than random.
On the other hand, the actual performance of boosting on a particular problem is dependent on the data and the weak learner. Consistent with the theory, boosting can fail to perform well given insufficient data, or given weak hypotheses which are too weak or with considerably mutual disagreement. Boosting seems to be susceptible to missing values and especially susceptible to noise.

5.2.6 Boosting as Additive Model

The final prediction in boosting $f(x)$ can be expressed as an additive expansion of $T$ individual classifiers $h_i$ (dependent on the examples weight $w'_i$), with weights on the hypotheses ($\alpha_t$).

$$f(x) = \sum_{t=1}^{T} \alpha_t h_t(x, w')$$

The process is iterative and can be expressed as follows:

$$f_T(x) = f_{T-1} + \alpha_T h_T(x, w_T)$$

Typically we would try to minimize a loss function on the training examples:

$$\min \sum_{t=1}^{T} L(y_i, \sum_{t=1}^{T} \alpha_t h_t(x, w'))$$

**Squared-error loss**

In the simple case the loss is squared-error loss:

$$L(y, f(x)) = \frac{1}{2} (y - f(x))^2$$

Forward stage-wise modeling amounts to just fitting the residuals from previous iteration:

$$L(y_i, f_{T-1}(x_i) + \alpha_T h_T(x_i, w'_T))$$

$$= (y_i - f_{T-1}(x_i) - \alpha_T h_T(x_i, w'_T))^2$$

$$= (r_{iT} - \alpha_T h_T(x_i, w'_T))^2$$

Squared-error loss is not robust for classifications.

**Exponential loss function (AdaBoost)**

The exponential loss function is: $L(y, f(x)) = \exp(-y \cdot f(x))$

In this case we need at each step $t$ to find $f_t$ that minimizes the training error
\[
\begin{align*}
\text{arg min } & \sum_{i=1}^{m} L(y_i, f_i(x_i)) = \\
& = \text{arg min } \sum_{i=1}^{m} \exp(-y_i \cdot [f_{i-1}(x_i) + \alpha_i \cdot h_i(x_i)]) \\
& = \text{arg min } \sum_{i=1}^{m} \exp(-y_i \cdot f_{i-1}(x_i)) \cdot \exp(-y_i \cdot \alpha_i \cdot h_i(x_i)) = \\
\end{align*}
\]

first assume that \( \alpha_i \) is constant, and minimize w.r.t. \( h_i \):

\[
\begin{align*}
\text{arg min } & \sum_{i=1}^{m} \exp(-y_i \cdot f_{i-1}(x_i)) \cdot \exp(-y_i \cdot \alpha_i \cdot h_i(x_i)) = \\
& = \text{arg min } \sum_{i=1}^{m} w_i^f \cdot \exp(-y_i \cdot \alpha_i \cdot h_i(x_i)) ; \quad \text{where } w_i^f = \exp(-y_i \cdot f_{i-1}(x_i)) \\
& = \text{arg min } \sum_{i=1}^{m} w_i^f \cdot e^{-\alpha_i} + \sum_{y_i \neq h_i(x_i)} w_i^f \cdot e^{\alpha_i} = \\
& = \text{arg min } \left( e^{\alpha_i} - e^{-\alpha_i} \right) \sum_{i=1}^{m} \left[ w_i^f \cdot I(y_i \neq h_i(x_i)) \right] + e^{-\alpha_i} \sum_{i=1}^{m} w_i^f = \\
& = \text{arg min } \left( e^{\alpha_i} - e^{-\alpha_i} \right) \frac{\sum_{i=1}^{m} \left[ w_i^f \cdot I(y_i \neq h_i(x_i)) \right]}{\sum_{i=1}^{m} w_i^f} + e^{-\alpha_i} = \\
& = \text{arg min } \left( e^{\alpha_i} - e^{-\alpha_i} \right) \cdot \text{err}_i + e^{-\alpha_i} = F(\alpha_i) \\
\end{align*}
\]

\( err_i \): It is the training error on the weighted samples.

The last equation tells us that in each iteration we must find a classifier that minimizes the training error on the weighted samples.

Now that we have found \( h_i \), we minimize w.r.t. \( \alpha_i \):

\[
\begin{align*}
F(\alpha_i) &= err_i \cdot (e^{\alpha_i} - e^{-\alpha_i}) + e^{-\alpha_i} \\
\frac{\partial F}{\partial \alpha_i} &= err_i \cdot (e^{\alpha_i} + e^{-\alpha_i}) - e^{-\alpha_i} = 0 \\
\end{align*}
\]
\[
1 - e^{\alpha_i} \cdot err_i (e^{\alpha_i} + e^{-\alpha_i}) = 0 \\
1 - e^{2\alpha_i} \cdot err_i - err_i = 0 \\
\frac{1 - err_i}{err_i} = e^{2\alpha_i} \\
\alpha_i = \frac{1}{2} \ln \left( \frac{1 - err_i}{err_i} \right)
\]

Notice that AdaBoost chose \( \alpha_i \) to be \( \frac{1}{2} \ln \left( \frac{1 - err_i}{err_i} \right) \).

### 5.3 Boosting Trees

#### 5.3.1 Basics of boosting trees

A \( J \)-terminal node regression tree can be model as:

\[
T(x, \Theta) := T(x, \{ \gamma_j, R_j \}_{j=1}^J) = \sum_{j=1}^J \gamma_j I(x \in R_j)
\]

Where the collection of \( \{ R_j \}_{j=1}^J \) are disjoint partitions of the predictor variable \( Y(x) \) and the \( \{ \gamma_j \}_{j=1}^J \) are the means of the response variable associate with predictor variables in \( R_j \) meaning that \( \gamma_j = \text{mean}_{x \in R_j} (Y(x)) \), and \( I \) is the indicator function.

The parameters \( \Theta = \{ R_j, \gamma_j \}_{j=1}^J \) can be found by minimizing the empirical risk.

\[
\hat{\Theta} = \arg \min_{\Theta} \sum_{j=1}^J \sum_{x \in R_j} L(Y(x), \gamma_j)
\]

Where \( L \) is the loss function.

In this way Finding \( \{ R_j \}_{j=1}^J \) is computational hard but a solution exists.

In order to simplify the calculation we can use an approximated criterion for optimizing \( \Theta \):

\[
\hat{\Theta} = \arg \min_{\Theta} \sum_{i=1}^N \tilde{L}(y_i, T(x_i, \Theta))
\]

Where \( N \) is the number of training samples and \( y_i = Y(x_i) \) (the response to the \( i^{th} \) sample).

In this way we can divide the minimum into two parts and iterate:

1) Finding \( \{ \gamma_j \}_{j=1}^J \) given \( \{ R_j \}_{j=1}^J \).

2) Finding \( \{ R_j \}_{j=1}^J \) using greedy recursive partitioning.
Boosted tree model is sum of such trees induced in a forward stage wise manner:

\[
\hat{\Theta}_m = \arg \min_{\Theta_m} \sum_{i=1}^{N} \tilde{L}(y_i, \sum_{m=1}^{M} T(x, \Theta_m))
\]

with \( \Theta_m = \{ R_{j,m}, y_{j,m} \}_{j=1}^{j_m} \).

In the case of binary classification and exponential loss function \( L(y, f(x)) = \exp(-y \cdot f(x)) \) this reduces to the AdaBoost algorithm.

### 5.3.2 Numerical Optimization

Goal: Find \( f \) that minimizes the loss function over the training data.

\[
\hat{f} = \arg \min_{f} L(f) = \arg \min_{f} \sum_{i=1}^{N} L(y_i, f(x_i))
\]

Where \( f(x) := T(x, \Theta) \).

The solution can be found by using the Steepest Descent search method on the unconstrained function space.

At each step, compute

\[
g_{i,m} = \left[ \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x_i) = f_{m-1}(x_i)}
\]

\[
\rho_m = \arg \min_{\rho} L(f_{m-1} - \rho \cdot g_m)
\]

\[
f_m = f_{m-1} - \rho \cdot g_m
\]

Where \( f_m \) is the tree at step \( m \) and \( \rho_m \) is the descent size of step \( m \).

This algorithm converge meaning that the loss on the training data converge to 0:

\[
f_m = \{ f_m(x_1), f_m(x_2), ..., f_m(x_N) \} \xrightarrow{m \to \infty} \{ y_1, y_2, ..., y_N \}
\]
5.3.3 The MART Algorithm

The disadvantages of using the gradient search method on the unconstrained function space are over fitting and robustness.

Therefore we will use the gradient search on constrained function space: Introduce a tree at the $m^{th}$ iteration whose predictions are as close as possible to the negative gradient.

$$\tilde{\Theta}_m = \arg \min_{\Theta} \sum_{i=1}^{N} \left(-g_{i,m} - T(x_i, \Theta_m)\right)^2$$

The MART (Multiple Additive Regression Trees) Algorithm:

1. Initialize $f_0(x)$ to a single terminal node tree: $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$

2. For $m=1$ to $M$:
   a. For $i=1,2,\ldots,N$ compute $g_{i,m}$ based on the loss function.
   b. Fit a regression tree to $-g_{i,m}$ (find the terminal regions $R_{j,m}$ $j=1,2,\ldots,J_m$)
   c. For $j=1,2,\ldots,J_m$:
      find the optimal value of coefficient within different region $R_{j,m}$:
      $$\gamma_{j,m} = \arg \min_{\gamma} \sum_{x_i \in R_{j,m}} L(y_i, f_{m-1}(x_i) + \gamma)$$
   d. Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{j,m} I(x \in R_{j,m})$.

3. Output $\hat{f}(x) = f_M(x)$.

The parameters that can be tuned in this algorithm are:
- The size of the constituent tree $J$.
- The number of boosting iteration $M$.

Finding the right size tree (finding $J$):

Problems:
- The optimal tree size for one step might not be the optimal for all steps.
- Using very large tree usually degrade performance and increase computation.

Solution:
Restrict the value of $J$ to be the same for all trees ($\forall m J_m = J$).

For trees the higher order interactions effects present in large trees suffer inaccuracies. $J$ is the factor that helps control the higher order interactions. Thus we would like to keep $J$ small. In practice the value of $4 \leq J \leq 8$ is seen to have worked the best.
**Controlling M (The number of boosting iteration):**

As $M \rightarrow \infty$ the training error goes to 0, but this would risk over fitting the training data. To avoid this we can use 2 methods:

1. Stop the loop when the training error is lower than a specific threshold.
2. Split the data to a training samples and validation samples and monitor the prediction risk on the validation samples (stop if the error is not reduced).

**Shrinkage:**
Shrinkage is an addition method to prevent over fitting:
The idea is to scale the contribution of each tree by a factor $0 < \nu < 1$ to control the learning rate:

$$f_m = f_{m-1}(x) + \nu \sum_{j=1}^{J} \gamma_{j,m} I(x \in R_{j,m})$$

Smaller values of $\nu$ lead to large values of $M$.

**References:**
2. Y.mansure Lecture Notes
3. Boosting Trees by Rishi Sinha