CSCI-GA.2566-001
Foundations of Machine Learning

Problem Set 3 Aditya Ramesh

Exercise 3.1. Show that $\hat{R}_\rho(f)$ can be upper-bounded by

$$\hat{R}_\rho(f) \le \frac{1}{m} \sum_{i=1}^{m} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) + \rho \sum_{t=1}^{T} \alpha_t\right).$$

Solution. For all $i \in [1, m]$, we have

$$\begin{aligned}
1\{y_i f(x_i) \le \rho\} &= 1\{y_i f(x_i) - \rho \le 0\} \\
&= 1\left\{\left(\sum_{t=1}^{T} \alpha_t\right)^{-1} \left(y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) - \rho \sum_{t=1}^{T} \alpha_t\right) \le 0\right\} \\
&= 1\left\{y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) - \rho \sum_{t=1}^{T} \alpha_t \le 0\right\} \\
&\le \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) + \rho \sum_{t=1}^{T} \alpha_t\right).
\end{aligned}$$

The result follows.

Exercise 3.2. For any $\rho > 0$, let G_ρ be the objective function defined for all $\alpha > 0$ by

$$G_\rho(\alpha) = \frac{1}{m} \sum_{i=1}^{m} \exp\left(-y_i \sum_{j=1}^{N} \alpha_j h_j(x_i) + \rho \sum_{j=1}^{N} \alpha_j \right),$$

with $h_j \in H$ for all $j \in [1, N]$, with the notation used in class in the boosting lecture. Show that G_ρ is convex and differentiable.

Solution. For fixed $i \in [1, m]$ and $j \in [1, N]$, let

$$z_j := \rho - y_i h_j(x_i).$$

Then we have

$$\begin{aligned} g(\alpha) &:= -y_i \sum_{j=1}^{N} \alpha_j h_j(x_i) + \rho \sum_{j=1}^{N} \alpha_j \\ &= \sum_{j=1}^{N} (\rho - y_i h_j(x_i)) \alpha_j = \sum_{j=1}^{N} \alpha_j z_j. \end{aligned}$$

Thus $g : \mathbb{R}^N \to \mathbb{R}$ is linear, and hence convex and smooth. Since exp is convex and non-decreasing, we have by Lemma B1 in the appendix that $\exp(g)$ is convex. Since the set of convex functions is closed under sums and scalar multiplication, G_ρ is convex.

To show that G_ρ is smooth, we note that the composition of smooth maps is smooth. Hence $\exp(g)$ is smooth. The class C^∞ is closed under sums and scalar multiplication. Therefore G_ρ is smooth.

Exercise 3.3. Derive a boosting-style algorithm A_ρ by applying (maximum) coordinate descent to G_ρ. You should justify in detail the derivation of the algorithm, in particular the choice of the base classifier selected at each round and that of the step. Compare both to their counterparts in AdaBoost.

Solution. For reasons that will become apparent in the derivation of the step size, we define

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

for all $i \in [1, m]$ and $t \in [1, T]$, where α_t is the step size and Z_t is a normalization constant. It follows that

$$D_{t+1}(i) = \frac{\exp\left(\sum_{s=1}^{t}(\rho - y_i h_s(x_i))\alpha_s\right)}{m\prod_{s=1}^{t} Z_s}.$$

Expressing G_ρ as

$$G_\rho(\alpha) = \frac{1}{m}\sum_{i=1}^{m}\exp\left(\sum_{j=1}^{N}(\rho - y_i h_j(x_i))\alpha_j\right),$$

we have

$$\begin{aligned}\frac{dG_\rho(\alpha_{t-1} + \eta e_t)}{d\eta} &= \frac{1}{m}\sum_{i=1}^{m}(\rho - y_i h_t(x_i))\exp\left(\sum_{j=1}^{t-1}(\rho - y_i h_j(x_i))\alpha_j\right)\\ &\quad \exp((\rho - y_i h_t(x_i))\eta)\\ &= \frac{1}{m}\sum_{i=1}^{m}(\rho - y_i h_t(x_i))D_t(i)\left(m\prod_{s=1}^{t-1} Z_s\right)\exp((\rho - y_i h_t(x_i))\eta).\end{aligned}$$

So

$$\begin{aligned}
0 &= \frac{dG_\rho(\alpha_{t-1} + \eta e_t)}{d\eta} \\
\Leftrightarrow 0 &= \frac{1}{m} \sum_{i=1}^{m} (\rho - y_i h_t(x_i)) D_t(i) \left(m \prod_{s=1}^{t-1} Z_s \right) \exp((\rho - y_i h_t(x_i))\eta) \\
\Leftrightarrow 0 &= \sum_{i=1}^{m} (\rho - y_i h_t(x_i)) D_t(i) \exp((\rho - y_i h_t(x_i))\eta) \\
\Leftrightarrow 0 &= \sum_{i: y_i h_t(x_i) = 1} (\rho - y_i h_t(x_i)) D_t(i) \exp((\rho - y_i h_t(x_i))\eta) + \\
&\quad \sum_{i: y_i h_t(x_i) = -1} (\rho - y_i h_t(x_i)) D_t(i) \exp((\rho - y_i h_t(x_i))\eta) \\
\Leftrightarrow 0 &= \sum_{i: y_i h_t(x_i) = 1} (\rho - 1) D_t(i) \exp((\rho - 1)\eta) + \\
&\quad \sum_{i: y_i h_t(x_i) = -1} (\rho + 1) D_t(i) \exp((\rho + 1)\eta) \\
\Leftrightarrow 0 &= (1 - \epsilon_t)(\rho - 1) \exp((\rho - 1)\eta) + \epsilon_t(\rho + 1) \exp((\rho + 1)\eta).
\end{aligned}$$

Solving for η in the last expression on the RHS yields

$$\eta = \frac{1}{2} \log \left(\frac{(1 - \epsilon_t)(1 - \rho)}{\epsilon_t(1 + \rho)} \right), \tag{1}$$

provided

$$(1 - \epsilon_t)(1 - \rho) \neq 0 \quad \text{and} \quad \epsilon_t(1 + \rho) \neq 0.$$

Substituting $\rho = 0$ in Equation 1 yields the step size used for AdaBoost.

The procedure used to select the base classifier at each round of boosting is the same as the one used in AdaBoost. To see why, observe that at boosting round t, we choose

$$\begin{aligned}
h_t &\in \operatorname*{arg\,min}_{h \in H} \Pr_{i \sim D_t} [y_i h(x_i) \leq \rho] \\
&= \operatorname*{arg\,min}_{h \in H} \sum_{i=1}^{m} D_t(i) 1\{y_i h(x_i) \leq \rho\} \\
&= \operatorname*{arg\,min}_{h \in H} \sum_{i=1}^{m} D_t(i) 1\{y_i \neq h(x_i)\}.
\end{aligned}$$

Exercise 3.4. What is the equivalent of the weak learning assumption for A_ρ?

Solution. Observe that

$$\begin{aligned}
0 < \eta &= \frac{1}{2}\log\left(\frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)}\right) \\
\Leftrightarrow 1 &< \frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)} \\
\Leftrightarrow 0 &< 1 - 2\epsilon_t - \rho.
\end{aligned}$$

So the equivalent of the weak learning assumption is that $0 < \gamma < 1 - 2\epsilon_t - \rho$, where γ is the edge of the weak learner. Substituting $\rho = 0$ yields the weak learning assumption for AdaBoost.

Exercise 3.5. Give the full pseudocode of the algorithm A_ρ. What can you say about the A_0 algorithm?

Algorithm 1 Boosting-style algorithm to minimize the empirical margin loss.

```
Precondition: $A$ is a weak learner than returns a hypothesis $h_t$ with small error
    $\epsilon_t := \Pr_{i \sim D_t}[y_i h_t(x_i) \le \rho]$.
Precondition: $\rho \in [0, 1)$ is the margin.
Precondition: $T > 0$ is the number of rounds of boosting.
Precondition: $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ is the training sample.

function MarginLossBoost($A, \rho, T, S$)
    for $i \in [1, m]$ do
        $D_t(i) \leftarrow \frac{1}{m}$
    for $t \in [1, T]$ do
        $(h_t, \epsilon_t) \leftarrow A(S, D_t)$                 ▷ Select the base classifier.
        if $\epsilon_t = 0$ then
            return $h_t$
        else if $1 - 2\epsilon_t \le \rho$ then        ▷ If the step size would be non-positive.
            if $t = 1$ then
                return $h_t$
            else
                return sgn $\left(\sum_{s=1}^{t-1} \alpha_s h_s\right)$
        $\alpha_t \leftarrow \frac{1}{2} \log \left(\frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)}\right)$
        $Z_t \leftarrow (1 - \epsilon_t) \exp((\rho - 1)\alpha_t) + \epsilon_t \exp((\rho + 1)\alpha_t)$
        for $i \in [1, m]$ do
            $D_{t+1}(i) \leftarrow (D_t(i) \exp((\rho - y_i h_t(x_i)\alpha_t)))/Z_t$
    return sgn $\left(\sum_{s=1}^{T} \alpha_s h_s\right)$
```

Solution. The pseudocode for the margin loss boosting algorithm is given in Algorithm 1. For $\rho = 0$, this algorithm is equivalent to AdaBoost.

Exercise 3.6. Implement the A_ρ algorithm. Use a grid search to determine the best values of ρ and T, and report the test error obtained. Compare the result with the one obtained by running AdaBoost using the same setup. Also, compare these results with those obtained in the second homework assignment using SVMs.

Solution. The results from the cross-validation procedure are given in Table 1. The best cross-validation error obtained using SVMs in the previous homework assignment was about 28.9%, with a standard deviation of about 2%. The best cross-validation error using boosting was obtained using the margin loss boosting algorithm, with $\rho = 2^{-5}$ and $T \in \{100, 200, 500, 1000\}$. This model achieved a cross-validation error of about 25.9%, with a standard deviation of 2.4%. So boosted decision stumps performed about 3% better than SVMs for this classification problem.

The best test error obtained using boosting was 28.6% for AdaBoost, and 27.3% for the margin loss boosting algorithm. Using SVMs, the best test error obtained was 31.6%. So on the test set, boosting was 4.3% better than SVMs.

T	$\hat{R}_{CV}$	Standard Deviation
100	0.265899	0.0266990
200	0.270968	0.0308599
500	0.277880	0.0310315
1000	0.280184	0.0317642

$-\log_2(\rho)$	T	$\hat{R}_{CV}$	Standard Deviation
12	100	0.272350	0.0271720
11	100	0.272811	0.0231742
10	100	0.266820	0.0240388
9	100	0.271429	0.0302927
8	100	0.267742	0.0269978
7	100	0.266359	0.0260503
6	100	0.262212	0.0310619
5	100	0.259447	0.0241954
4	100	0.263134	0.0201984
3	100	0.287097	0.0253386
2	100	0.331797	0.0196717
1	100	0.331797	0.0196717
12	200	0.268203	0.0232758
11	200	0.268664	0.0238021
10	200	0.269124	0.0250718
9	200	0.271889	0.0240928
8	200	0.268203	0.0238763
7	200	0.268203	0.0231742
6	200	0.263594	0.0298494
5	200	0.259447	0.0241954
4	200	0.263134	0.0201984
3	200	0.287097	0.0253386
2	200	0.331797	0.0196717
1	200	0.331797	0.0196717
12	500	0.277419	0.0317642
11	500	0.278802	0.0308178
10	500	0.275576	0.0286391
9	500	0.272811	0.0263207
8	500	0.270046	0.0262668
7	500	0.266820	0.0226229
6	500	0.263594	0.0295315
5	500	0.259447	0.0241954
4	500	0.263134	0.0201984
3	500	0.287097	0.0253386
2	500	0.331797	0.0196717
1	500	0.331797	0.0196717
12	1000	0.281106	0.0301795
11	1000	0.279724	0.0307258
10	1000	0.276498	0.0306451
9	1000	0.277419	0.0309363
8	1000	0.270046	0.0260865
7	1000	0.267281	0.0225759
6	1000	0.263594	0.0295315
5	1000	0.259447	0.0241954
4	1000	0.263134	0.0201984
3	1000	0.287097	0.0253386
2	1000	0.331797	0.0196717
1	1000	0.331797	0.0196717

Table 1: Cross-validation results for AdaBoost (left) and the margin loss boosting algorithm (right).

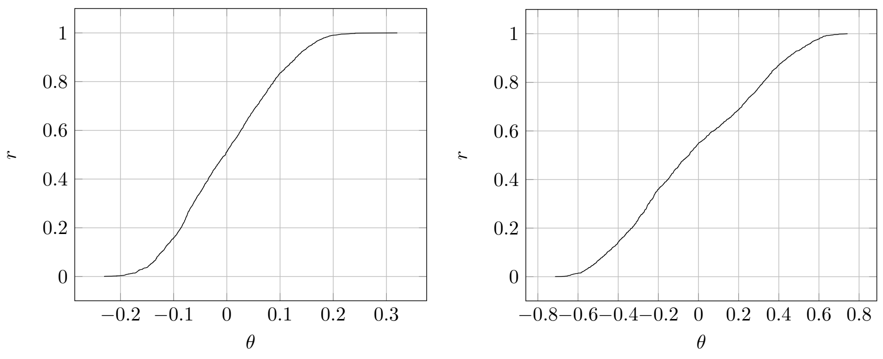

Figure 1: Plots of the fraction of the training points with margin less than or equal to θ for AdaBoost with $T = 500$ (left) and the margin loss boosting algorithm with $\rho = 2^{-5}$ and $T = 500$ (right).

Exercise 3.7. For both the A_ρ results and AdaBoost, plot the cumulative margins after 500 rounds of boosting. That is, plot the fraction of training points with margin less than or equal to θ as a function of $\theta \in [0, 1]$.

Solution. The relevant plots are given in Figure 1. Apparently, the margin loss boosting algorithm classifies the training data with a much larger margin, on average, than AdaBoost. Whereas the cumulative margins for AdaBoost are distributed in the range $(0, 0.32]$, for the margin loss boosting algorithm, they are distributed in the range $(0, 0.74]$. As expected, the margin loss boosting algorithm did a better job of minimizing the empirical margin loss than AdaBoost.

Exercise 3.8.

(a) Prove the upper bound

$$\hat{R}_\rho(f) \le \exp\left(\rho \sum_{t=1}^{T} \alpha_t\right) \prod_{t=1}^{T} Z_t,$$

where the normalization factors Z_t are defined as in the case of AdaBoost.

(b) Give the expression of Z_t as a function of ρ and ϵ_t. Use this to prove that

$$\hat{R}_\rho(f) \le \left(u^{\frac{1+\rho}{2}} + u^{\frac{\rho-1}{2}}\right)^T \prod_{t=1}^{T} \sqrt{\epsilon_t^{1-\rho}(1-\epsilon_t)^{1+\rho}},$$

where

$$u = \frac{1-\rho}{1+\rho}.$$

(c) Assume that for all $t \in [1, T]$,

$$\frac{1-\rho}{2} - \epsilon_t > \gamma > 0.$$

Use the result of the previous question to show that

$$\hat{R}_\rho(f) \le \exp\left(-\frac{2\gamma^2 T}{1-\rho^2}\right).$$

Show that for

$$T \ge \frac{(\log m)(1-\rho^2)}{2\gamma^2},$$

all points of the training data have margin at least ρ.

Solution.

(a) As in AdaBoost, we have

$$D_{t+1}(i) = \frac{\exp\left(-y_i \sum_{s=1}^{t} a_s h_s(x_i)\right)}{m \prod_{s=1}^{t} Z_s}.$$

Thus

$$\begin{aligned}\hat{R}_\rho(g) &\leq \frac{1}{m} \sum_{i=1}^{m} \exp\left(-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i) + \rho \sum_{t=1}^{T} \alpha_t\right) \\ &= \frac{1}{m} \exp\left(\rho \sum_{t=1}^{T} \alpha_t\right) \left(\sum_{i=1}^{m} D_{T+1}(i)\right) \left(m \prod_{s=1}^{T} Z_s\right) \\ &= \exp\left(\rho \sum_{t=1}^{T} \alpha_t\right) \prod_{s=1}^{T} Z_s,\end{aligned}$$

as required, where we note that D_{T+1} sums to one, as it is a normalized distribution.

(b) Before we can answer this question, we need to re-derive the step size α_t using the new definition of D_t. Fortunately, however, the solution for the step size is unchanged by this modification. To see why, observe that

$$\begin{aligned}\frac{dG_\rho(\alpha_{t-1}+\eta e_t)}{d\eta} &= \frac{1}{m}\sum_{i=1}^{m}(\rho - y_i h_t(x_i))\exp\left(\sum_{j=1}^{t-1}(\rho - y_i h_j(x_i))\alpha_j\right)\\ &\quad \exp((\rho - y_i h_t(x_i))\eta)\\ &= \frac{1}{m}\exp\left(\rho\sum_{j=1}^{t-1}\alpha_j\right)\sum_{i=1}^{m}(\rho - y_i h_t(x_i))\exp\left(\sum_{j=1}^{t-1}-y_i h_j(x_i)\alpha_j\right)\\ &\quad \exp((\rho - y_i h_t(x_i))\eta)\\ &= \frac{1}{m}\exp\left(\rho\sum_{j=1}^{t-1}\alpha_j\right)\sum_{i=1}^{m}(\rho - y_i h_t(x_i))D_t(i)\left(m\prod_{s=1}^{t}Z_s\right)\\ &\quad \exp((\rho - y_i h_t(x_i))\eta)\\ &= \left(\prod_{s=1}^{t}Z_s\right)\exp\left(\rho\sum_{j=1}^{t-1}\alpha_j\right)\sum_{i=1}^{m}(\rho - y_i h_t(x_i))D_t(i)\\ &\quad \exp((\rho - y_i h_t(x_i))\eta).\end{aligned}$$

The last expression on the RHS is equal to zero if and only if

$$0 = \sum_{i=1}^{m}(\rho - y_i h_t(x_i))D_t(i)\exp((\rho - y_i h_t(x_i))\eta).$$

From this point, the derivation of the step size is the same as it was using the other definition of D_t. Therefore

$$\eta = \frac{1}{2}\log\left(\frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)}\right).$$

Solving for Z_t, we get

$$\begin{aligned}Z_t &= \sum_{i=1}^{m}D_t(i)\exp(-y_i\eta h_t(x_i))\\ &= (1-\epsilon_t)e^{-\eta} + \epsilon_t e^{\eta}\\ &= (1-\epsilon_t)\sqrt{\frac{\epsilon_t(1+\rho)}{(1-\epsilon_t)(1-\rho)}} + \epsilon_t\sqrt{\frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)}}\\ &= \sqrt{\frac{\epsilon_t(1-\epsilon_t)(1+\rho)}{1-\rho}} + \sqrt{\frac{\epsilon_t(1-\epsilon_t)(1-\rho)}{1+\rho}}\\ &= \sqrt{\epsilon_t(1-\epsilon_t)}\left(\sqrt{u^{-1}} + \sqrt{u}\right).\end{aligned}$$

Substituting this expression for Z_t in $\hat{R}_\rho(f)$, we get

$$\begin{aligned}
\hat{R}_\rho(f) &\leq \exp\left(\rho \sum_{t=1}^{T} a_t\right) \left(\sqrt{u^{-1}} + \sqrt{u}\right)^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1-\epsilon_t)} \\
&= \left(\prod_{t=1}^{T} \sqrt{\frac{(1-\epsilon_t)(1-\rho)}{\epsilon_t(1+\rho)}}\right)^\rho \left(\sqrt{u^{-1}} + \sqrt{u}\right)^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1-\epsilon_t)} \\
&= u^{\frac{\rho T}{2}} \left(\sqrt{u^{-1}} + \sqrt{u}\right)^T \prod_{t=1}^{T} \left(\frac{1-\epsilon_t}{\epsilon_t}\right)^{\frac{\rho}{2}} \sqrt{\epsilon_t(1-\epsilon_t)} \\
&= \left(u^{\frac{\rho+1}{2}} + u^{\frac{\rho-1}{2}}\right)^T \prod_{t=1}^{T} \sqrt{\epsilon_t^{1-\rho}(1-\epsilon_t)^{1+\rho}},
\end{aligned}$$

as desired.

(c) Using the result from part (b) and the identity in the hint, we get

$$\begin{aligned}
\hat{R}_\rho(f) &\leq \prod_{t=1}^{T} \left(u^{\frac{\rho+1}{2}} + u^{\frac{\rho-1}{2}}\right) \sqrt{\epsilon_t^{1-\rho}(1-\epsilon_t)^{1+\rho}} \\
&\leq \prod_{t=1}^{T} \left(1 - 2\frac{\left(\frac{1-\rho}{2} - \epsilon_t\right)^2}{1-\rho^2}\right) \\
&< \prod_{t=1}^{T} \left(1 - \frac{2\gamma^2}{1-\rho^2}\right) \\
&\leq \prod_{t=1}^{T} \exp\left(-\frac{2\gamma^2}{1-\rho^2}\right) \\
&= \exp\left(-\frac{2\gamma^2 T}{1-\rho^2}\right),
\end{aligned}$$

where we used the identity $1 + x \leq e^x$, valid for all $x \in \mathbb{R}$. If

$$T \geq \frac{(\log m)(1-\rho^2)}{2\gamma^2},$$

we get

$$\frac{2\gamma^2 T}{1-\rho^2} \geq \log m.$$

Hence

$$\hat{R}_\rho(f) \leq \exp\left(-\frac{2\gamma^2 T}{1-\rho^2}\right) \leq \frac{1}{m},$$

which implies that

$$\sum_{i=1}^{m} 1\{y_i f(x_i) \leq \rho\} < 1.$$

So no point in the training sample can have margin less than or equal to ρ. In order words, all points in the training sample have margin greater than ρ.

Appendix

I have included the C++ source listings of the most important files used to implement the boosted decision stumps in the cross-validation procedure. The same cross-validation files that were prepared for the previous homework assignment were used for this one as well.

`boost.hpp`

This header file implements the AdaBoost and margin loss boosting algorithms.

```
#ifndef Z94575332_0295_4270_A342_29A7C43D1217
#define Z94575332_0295_4270_A342_29A7C43D1217

#include <algorithm>
#include <cmath>
#include <iterator>
#include <type_traits>
#include <vector>
#include <stump.hpp>

template <class Classifier>
class boosted_classifier
{
private:
	std::vector<Classifier> bases{};
	std::vector<double> coeffs{};
	double z{};
public:
	boosted_classifier() {}

	boosted_classifier(const size_t n)
	{
		bases.reserve(n);
		coeffs.reserve(n);
	}

	size_t size() const { return bases.size(); }

	void add(double coeff, const Classifier& c)
	{
		bases.push_back(c);
		coeffs.push_back(coeff);
		z += coeff;
	}

	void add(double coeff, Classifier&& c)
	{
		bases.push_back(std::move(c));
		coeffs.push_back(coeff);
		z += coeff;
	}

	template <class Iterator>
	double operator()(Iterator it) const
	{
		auto r = double{};
		for (size_t i = 0; i != bases.size(); ++i) {
			r += coeffs[i] * bases[i](it);
		}
		return std::copysign(1.0, r);
```

```
        }

        template <class Iterator>
        double margin(Iterator it) const
        {
                auto r = double{};
                for (size_t i = 0; i != bases.size(); ++i) {
                        r += coeffs[i] * bases[i](it);
                }
                return r / z;
        }
};

template <class Iterator, class WeakLearner>
auto adaboost(const Iterator f, const Iterator l, const size_t T,
        const WeakLearner& wl)
{
        using return_type =
        typename std::result_of<WeakLearner(
                const Iterator, const Iterator,
                std::vector<double>
        )>::type;

        using base_classifier =
        typename std::tuple_element<0, return_type>::type;

        auto m = std::distance(f, l);
        auto dist = std::vector<double>(m, 1.0 / m);
        auto g = boosted_classifier<base_classifier>{T};
        auto h = base_classifier{};
        auto err = double{};

        for (size_t t{0}; t != T; ++t) {
                std::tie(h, err) = wl(f, l, dist);
                assert(err != 0);
                assert(err < 0.5 && "Weak learning assumption violated.");

                auto step = 0.5 * std::log((1 - err) / err);
                auto z = 2 * std::sqrt(err * (1 - err));
                assert(step > 0);

                for (size_t i{0}; i != m; ++i) {
                        auto x = std::begin(std::get<0>(f[i]));
                        auto y = std::get<1>(f[i]);
                        dist[i] = dist[i] * std::exp(-step * y * h(x)) / z;
                }
                assert(std::fabs(std::accumulate(begin(dist), end(dist), 0.0) - 1) < 1e-10);
                g.add(step, h);

                auto wrong = size_t{0};
                for (size_t i{0}; i != m; ++i) {
                        auto x = std::begin(std::get<0>(f[i]));
                        auto y = std::get<1>(f[i]);
                        wrong += (g(x) != y);
                }
                //cc::println("Round $: error: $.", t + 1, double(wrong) / m);
        }
        return g;
}

template <class Iterator, class WeakLearner>
auto margin_loss_boost(const Iterator f, const Iterator l, const double marg,
        const size_t T, const WeakLearner& wl)
{
        using return_type =
        typename std::result_of<WeakLearner(
                const Iterator, const Iterator,
                std::vector<double>
        )>::type;

        using base_classifier =
        typename std::tuple_element<0, return_type>::type;

        assert(marg >= 0 && marg < 1);

        auto m = std::distance(f, l);
        auto dist = std::vector<double>(m, 1.0 / m);
        auto g = boosted_classifier<base_classifier>{T};
        auto h = base_classifier{};
        auto err = double{};

        for (size_t t{0}; t != T; ++t) {
                std::tie(h, err) = wl(f, l, dist);
                assert(err * (marg + 1) != 0);
                assert((1 - err) * (1 - marg) != 0);

                /*
                ** If the weak learning assumption is violated, we return
                ** whatever we have so far.
                **
                ** assert((1 - 2 * err > marg) && "Weak learning assumption violated.");
                */
                if (1 - 2 * err <= marg) {
                        if (g.size() > 0) {
                                return g;
                        }
                        else {
                                g.add(1, h);
                                return g;
                        }
                }

                auto step = 0.5 * std::log((1 - err) * (1 - marg) / (err * (marg + 1)));
                auto z = (1 - err) * std::exp((marg - 1) * step) + err *
                        std::exp((marg + 1) * step);
                assert(step > 0);

                for (size_t i{0}; i != m; ++i) {
                        auto x = std::begin(std::get<0>(f[i]));
                        auto y = std::get<1>(f[i]);
                        dist[i] = dist[i] * std::exp((marg - y * h(x)) * step) / z;
                }
                assert(std::fabs(std::accumulate(begin(dist), end(dist), 0.0) - 1) < 1e-10);
                g.add(step, h);

                auto wrong = size_t{0};
                for (size_t i{0}; i != m; ++i) {
                        auto x = std::begin(std::get<0>(f[i]));
                        auto y = std::get<1>(f[i]);
                        wrong += (g(x) != y);
                }
                //cc::println("Round $: error: $.", t + 1, double(wrong) / m);
        }
        return g;
```

```
}

#endif
```

stump.hpp

This header file implements the procedure to find best decision stump at each round of boosting.

```
#ifndef Z0935E106_AF53_40AF_A0A7_6C3AB8BF0645
#define Z0935E106_AF53_40AF_A0A7_6C3AB8BF0645

#include <numeric>
#include <vector>
#include <tuple>

class stump
{
private:
        // Note that the component is zero-indexed.
        size_t comp{};
        double thres{};
        bool leq{};
public:
        stump() noexcept {}

        stump(size_t comp, double thres, bool leq)
        noexcept : comp{comp}, thres{thres}, leq{leq} {}

        double threshold() const { return thres; }
        size_t component() const { return comp; }

        template <class Iterator>
        double operator()(Iterator it) const
        {
                if (leq) {
                        return it[comp] <= thres ? 1 : -1;
                }
                else {
                        return it[comp] > thres ? 1 : -1;
                }
        }
};

/*
** Returns the best decision stump for a specific component of the feature
** vectors in the sample, along with the stump's associated loss $\epsilon_t$
** for round $t$ of boosting.
**
** Overview of algorithm:
**
** - Store the cumulative losses $p_k$ for classifying all bins with index less
**   than or equal to $k$ as positive.
** - Store the cumulative losses $n_k$ for classifying all bins with index
**   greater than $k$ as negative.
** - Iterate through $k \in [1, m]$, and compute the minimum and maximum net
**   losses $\epsilon_k$ for classification.
** - Return $\min(\epsilon_min, 1 - \epsilon_max)$, along with the associated
**   decision stump.
*/

template <class Iterator>
std::tuple<stump, double>
best_component_stump(
        // The component for which we are finding the best decision stump.
        const size_t comp,
        // Cumulative losses $p_k$ for classifying all bins of index less than
        // or equal to $k$ as positive.
        std::vector<double>& p_losses,
        // Cumulative losses $l_k$ for classifying all bins of index greater
        // than $k$ as negative.
        std::vector<double>& n_losses,
        // An iterator to a list of pairs. The first component of each pair is
        // the feature vector; the second component is the label.
        const Iterator sample,
        // Maps each unique feature in component $k$ of the feature vectors to a
        // vector of indices.
        const std::vector<std::tuple<double, std::vector<size_t>>>& bins,
        // The discrete distribution of sample boosting weights.
        const std::vector<double>& dist
)
{
        assert(bins.size() != 0);
        // For $m$ bins, there are actually $m + 1$ losses. But the extra loss
        // is the one which classifies all bins one way (positive or negative).
        // The corresponding loss for the other label is zero, so there is no
        // need to store it.
        p_losses.resize(bins.size());
        n_losses.resize(bins.size());

        /*
        ** Compute the cumulative losses. In the first loop, only the per-bin
        ** losses for negative classification are stored.
        */
        auto i = size_t{0};
        auto k = 0u;
        auto cum_p_loss = double{0};
        for (const auto& pair : bins) {
                // Loss for classifying this bin positive.
                auto p_loss = double{};
                // Loss for classifying this bin negative.
                auto n_loss = double{};
                for (const auto& index : std::get<1>(pair)) {
                        if (std::get<1>(sample[index]) == -1) {
                                p_loss += dist[index];
                        }
                        else {
                                n_loss += dist[index];
                        }
                }
                cum_p_loss += p_loss;
                p_losses[i] = cum_p_loss;
                n_losses[i] = n_loss;
                ++i;
                k += std::get<1>(pair).size();
        }

        /*
        ** Compute the cumulative losses for negative classification given the
        ** per-bin losses.
        */
        if (bins.size() > 1) {
                for (auto i = bins.size() - 2; i != size_t(-1); --i) {
                        n_losses[i] += n_losses[i + 1];
                }
        }

        /*
        ** The error resulting from classifying everything negative should be
        ** one minus the error resulting from classifying everything positive.
        */
        assert(std::fabs(n_losses.front() + p_losses.back() - 1) < 1e-10);

        /*
        ** Compute the minimum and maximum net losses. We only have to do this
        ** for one kind of threshold (left positive, right negative), since the
        ** loss for the other kind of decision stump at the same threshold is
        ** one minus the the loss of the former. The initial values correspond
        ** to classifying all of the bins as positive.
        */
        auto leq = true;
        auto l_min = p_losses.back();
        auto l_max = l_min;
        auto thres_min = std::get<0>(bins.back());
        auto thres_max = thres_min;
        for (size_t i = 0; i != bins.size() - 1; ++i) {
                auto loss = p_losses[i] + n_losses[i + 1];
                if (loss < l_min) {
                        l_min = loss;
                        thres_min = std::get<0>(bins[i]);
                }
                else if (loss > l_max) {
                        l_max = loss;
                        thres_max = std::get<0>(bins[i]);
                }
        }

        if (1 - l_max < l_min) {
                leq = false;
                l_min = 1 - l_max;
                thres_min = thres_max;
        }
        return std::make_tuple(stump{comp, thres_min, leq}, l_min);
}

template <class Iterator>
std::tuple<stump, double>
best_stump(
        std::vector<double>& p_losses,
        std::vector<double>& n_losses,
        const Iterator sample,
        // One feature to index list map for each component.
        const std::vector<std::vector<
                std::tuple<double, std::vector<size_t>>
        >>& comp_bins,
        const std::vector<double>& dist
)
{
        assert(comp_bins.size() != 0);
        assert(std::fabs(std::accumulate(begin(dist), end(dist), 0.0) - 1.0) < 1e-10);

        /*
        ** Return the best decision stump among those for all components of the
        ** feature vectors.
        */
        auto best = std::make_tuple(stump{}, double{1});
        for (size_t i = 0; i != comp_bins.size(); ++i) {
                auto cur = best_component_stump(i, p_losses, n_losses, sample,
                        comp_bins[i], dist);
                if (std::get<1>(cur) < std::get<1>(best)) {
                        best = cur;
                }
        }
        return best;
}

#endif
```

source/cross_validate.cpp

This header file implements the cross-validation procedure for the AdaBoost and margin loss boosting algorithms. The file used to generate the cumulative margin distributions is very similar.

```
#include <algorithm>
#include <numeric>
#include <tuple>
#include <utility>
#include <ccbase/format.hpp>
#include <parse.hpp>
#include <stump.hpp>
#include <boost.hpp>

template <class X, class Y>
static auto sort_components(const std::vector<std::tuple<X, Y>>& samp)
{
	using bin = std::tuple<double, std::vector<size_t>>;
	using bin_list = std::vector<bin>;

	const auto components = std::get<0>(samp[0]).size();
	auto comp_bins = std::vector<bin_list>{components};
	auto max_comp = size_t{0};

	for (size_t i{0}; i != comp_bins.size(); ++i) {
		for (size_t j{0}; j != samp.size(); ++j) {
			auto comp = std::get<0>(samp[j])[i];
			auto it = std::lower_bound(
				std::begin(comp_bins[i]),
				std::end(comp_bins[i]),
				comp,
				[](auto x, auto y) { return std::get<0>(x) < y; }
			);
			if (it != std::end(comp_bins[i]) && std::get<0>(*it) == comp) {
				std::get<1>(*it).push_back(j);
			}
			else {
				comp_bins[i].emplace(it, comp, std::vector<size_t>{j});
			}
		}
		max_comp = std::max(max_comp, comp_bins[i].size());
	}
	return std::make_tuple(comp_bins, max_comp);
}

template <class Sample, class BinList>
static auto adaboost_cross_validate(
	const std::vector<Sample>& train_data,
	const std::vector<Sample>& validation_data,
	const std::vector<BinList>& train_bins,
	std::vector<double>& p_losses,
	std::vector<double>& n_losses,
	const size_t t
)
{
	auto mean = double{};
	auto errors = std::vector<double>{};
	errors.reserve(train_data.size());

	for (size_t i{0}; i != train_data.size(); ++i) {
		auto g = adaboost(
                        begin(train_data[i]), end(train_data[i]), t,
                        [&](auto f, auto l, auto dist) {
                                return best_stump(p_losses, n_losses, f,
                                        train_bins[i], dist);
                        });

                auto wrong = size_t{0};
                for (const auto& pair : validation_data[i]) {
                        auto x = std::begin(std::get<0>(pair));
                        auto y = std::get<1>(pair);
                        wrong += (g(x) != y);
                }

                auto err = double(wrong) / validation_data[i].size();
                mean += err;
                errors.push_back(err);
        }

        mean /= errors.size();
        auto stddev = double{};
        for (const auto& err : errors) {
                stddev += std::pow(err - mean, 2);
        }
        stddev = std::sqrt(stddev / (errors.size() - 1));
        return std::make_tuple(mean, stddev);
}

template <class Sample, class BinList>
static auto margin_loss_cross_validate(
        const std::vector<Sample>& train_data,
        const std::vector<Sample>& validation_data,
        const std::vector<BinList>& train_bins,
        std::vector<double>& p_losses,
        std::vector<double>& n_losses,
        const double rho,
        const size_t t
)
{
        auto mean = double{};
        auto errors = std::vector<double>{};
        errors.reserve(train_data.size());

        for (size_t i{0}; i != train_data.size(); ++i) {
                auto g = margin_loss_boost(
                        begin(train_data[i]), end(train_data[i]), rho, t,
                        [&](auto f, auto l, auto dist) {
                                return best_stump(p_losses, n_losses, f,
                                        train_bins[i], dist);
                        });

                auto wrong = size_t{0};
                for (const auto& pair : validation_data[i]) {
                        auto x = std::begin(std::get<0>(pair));
                        auto y = std::get<1>(pair);
                        wrong += (g(x) != y);
                }

                auto err = double(wrong) / validation_data[i].size();
                mean += err;
                errors.push_back(err);
        }
        mean /= errors.size();
        auto stddev = double{};
        for (const auto& err : errors) {
                stddev += std::pow(err - mean, 2);
        }
        stddev = std::sqrt(stddev / (errors.size() - 1));
        return std::make_tuple(mean, stddev);
}

template <class Instance, class Bin>
static void adaboost_test(
        const std::vector<Instance>& train_data,
        const std::vector<Instance>& test_data,
        const std::vector<Bin>& train_bins,
        std::vector<double>& p_losses,
        std::vector<double>& n_losses,
        const size_t t
)
{
        auto g = adaboost(begin(train_data), end(train_data), t,
                [&](auto f, auto l, auto dist) {
                        return best_stump(p_losses, n_losses, f,
                                train_bins, dist);
                });

        auto wrong = size_t{0};
        for (const auto& pair : test_data) {
                auto x = std::begin(std::get<0>(pair));
                auto y = std::get<1>(pair);
                wrong += (g(x) != y);
        }
        auto err = double(wrong) / test_data.size();
        cc::println("AdaBoost test error: $.", err);
}

template <class Instance, class Bin>
static void margin_loss_boost_test(
        const std::vector<Instance>& train_data,
        const std::vector<Instance>& test_data,
        const std::vector<Bin>& train_bins,
        std::vector<double>& p_losses,
        std::vector<double>& n_losses,
        const double rho,
        const size_t t
)
{
        auto g = margin_loss_boost(begin(train_data), end(train_data), rho, t,
                [&](auto f, auto l, auto dist) {
                        return best_stump(p_losses, n_losses, f,
                                train_bins, dist);
                });

        auto wrong = size_t{0};
        for (const auto& pair : test_data) {
                auto x = std::begin(std::get<0>(pair));
                auto y = std::get<1>(pair);
                wrong += (g(x) != y);
        }
        auto err = double(wrong) / test_data.size();
        cc::println("Margin loss boost test error: $.", err);
}
```

```
int main()
{
        using sample_type = std::vector<std::tuple<std::array<double, 60>, int>>;
        auto train_data = std::vector<sample_type>{};
        auto validation_data = std::vector<sample_type>{};

        const char* train_file = "data/out/splice_noise_train_scaled_shuffled.txt";
        const char* test_file = "data/out/splice_noise_test_scaled_shuffled.txt";
        cc::println("Parsing $.", train_file);
        auto train_data_full = parse(train_file).get();
        cc::println("Parsing $.", test_file);
        auto test_data = parse(test_file).get();

        /*
        ** Parse the data for cross-validation.
        */
        for (size_t i{1}; i != 11; ++i) {
                auto train_file = std::string{"data/out/train_cv_"};
                train_file += std::to_string(i);
                train_file += ".txt";

                auto test_file = std::string{"data/out/train_"};
                test_file += std::to_string(i);
                test_file += ".txt";

                cc::println("Parsing $.", train_file);
                train_data.push_back(parse(train_file.c_str()).get());
                cc::println("Parsing $.", test_file);
                validation_data.push_back(parse(test_file.c_str()).get());
        }

        using pair_type = decltype(sort_components(test_data));
        using bin_list  = std::tuple_element<0, pair_type>::type;
        auto train_bins = std::vector<bin_list>{};
        train_bins.reserve(train_data.size());
        auto train_bins_full = std::get<0>(sort_components(train_data_full));

        /*
        ** Pre-sort the components.
        */
        cc::println("Sorting components.");
        auto max_comp = size_t{};
        for (size_t i{0}; i != train_data.size(); ++i) {
                auto pair = sort_components(train_data[i]);
                max_comp = std::max(max_comp, std::get<1>(pair));
                train_bins.push_back(std::move(std::get<0>(pair)));
        }

        auto p_losses = std::vector<double>{};
        auto n_losses = std::vector<double>{};
        p_losses.reserve(max_comp);
        n_losses.reserve(max_comp);

        // cc::println("AdaBoost Cross-Validation");
        //for (const auto& t : {100, 200, 500, 1000}) {
        //      auto stats = adaboost_cross_validate(train_data,
        //              validation_data, train_bins, p_losses, n_losses, t);
        //      //cc::println("T: $. Mean error: $.", t, std::get<0>(stats));
        //      cc::println("$, $, $", t, std::get<0>(stats),
        //              std::get<1>(stats));
        //}
        // cc::println("Margin Loss Cross-Validation");
        //for (const auto& t : {100, 200, 500, 1000}) {
        //      for (const auto& nlrho : {12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}) {
        //              auto rho = std::pow(2, -nlrho);
        //              auto stats = margin_loss_cross_validate(train_data,
        //                      validation_data, train_bins, p_losses, n_losses,
        //                      rho, t);
        //              //cc::println("nlrho: $. T: $. Mean error: $.", nlrho, t,
        //              //      std::get<0>(stats));
        //              cc::println("$, $, $, $", nlrho, t,
        //                      std::get<0>(stats), std::get<1>(stats));
        //      }
        //}

        adaboost_test(train_data_full, test_data, train_bins_full, p_losses,
                n_losses, 100);
        margin_loss_boost_test(train_data_full, test_data, train_bins_full,
                p_losses, n_losses, std::pow(2.0, -5), 1000);
}
```