Notes for EECS 445

Introduction to Machine Learning

Albon Wu

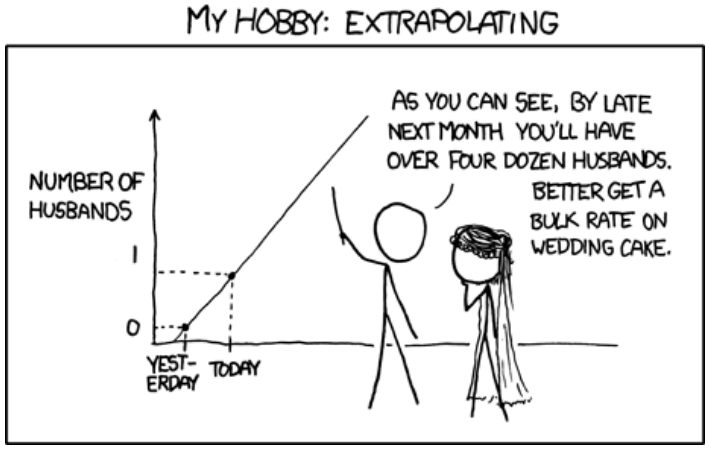
October 22, 2023

Contents

0 Introduction

Course covering theory and implementation of machine learning algorithms. Supervised and unsupervised learning.

Textbook: None.



1 Linear Classifiers

1.1 Basic Definitions

Supervised learning: training using a labeled dataset as experience

Classification: the process of assigning inputs to a finite number of discrete categories

We will consider the example of classifying Amazon reviews to determine their usefulness. Here, the parts of the review needed for classification are called **features**, and the process of selecting them is called **feature engineering**. The number of people who mark a review as "helpful" will be our **label**—what we are trying to predict.

The goal is to construct a model with these specifications that can predict the label of a previously unseen data point given its features.

In this example, we will use the following features: star rating as a fraction of 5 stars, and review length as a fraction of 200 words. The label will be a binary value: at least 10 votes for “helpful” corresponds to a positive label, and a negative label otherwise.

1.2 Classification as an ML Problem

Mathematically, features are statistics or attributes that describe the data, which we can represent in terms of vectors. Take the table below:

stars	length	label
0.6	0.7	+
0.2	0.2	-
1	0.9	+
0.2	0.9	-
0.6	0.2	?

Our feature vectors include $[0.6, 0.7]^\top$ and $[0.2.0.2]^\top$. Let n be the number of data points in the training dataset—the set of values used to train a model—and number each point sequentially. We denote the i^{th} feature vector by $\vec{x}^{(i)}$ and the corresponding label by $y^{(i)}$. The j^{th} feature of the i^{th} data point is denoted $x_j^{(i)}$.

In general, $\vec{x}^{(i)} \in \mathcal{X}$, where $\mathcal{X}$ is the feature space.

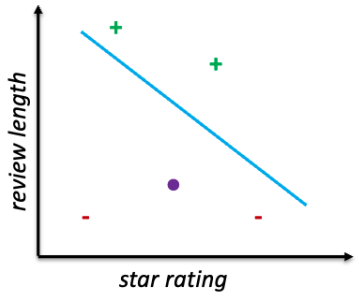
Each training data point i has a label $y^{(i)} \in \{+1, -1\}$ in the case of binary classification. Similarly, $y^{(i)} \in \mathcal{Y}$, where $\mathcal{Y}$ is the label space.

In general, we denote the training set of a supervised learning problem by

$$S_n = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^n = \{(\vec{x}^{(1)}, y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), \ldots, (\vec{x}^{(n)}, y^{(n)})\}.$$

The problem is thus to predict whether a new, unlabeled review is helpful (+1) or unhelpful (−1). There are a few ways to approach this task.

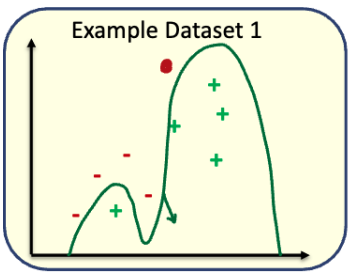
First, we will consider a geometric perspective. Let's plot the features in $\mathbb{R}^2$.

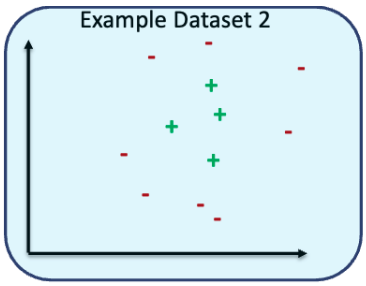


stars	length	label
0.6	0.7	+
0.2	0.2	-
1	0.9	+
0.2	0.9	-
0.6	0.2	?

Intuitively, the unclassified point should have a negative label. The data looks like it could be separated by a line with negative slope partitioning the positive and negative points. This is an example of a **linear** decision boundary.

But why restrict ourselves to linear boundaries? Wouldn't we get a better fit if we chose a classifier that could completely accommodate the training data? Consider the following datasets. We denote the set of possible classifiers by $\mathcal{H}$.





Observe the first dataset. If we use a partition that perfectly separates the data, like the one shown, it might misclassify obvious points. To accommodate the positive outlier, the curve becomes convoluted and incorrectly labels the red point that is clearly positive. This is called **overfitting**.

Thus, we might be inclined to constrain $\mathcal{H}$ so that we don't end up with a wildly esoteric decision boundary. On the other hand, if we constrain $\mathcal{H}$ too much, we might miss out on important patterns in the data. A linear decision boundary is obviously insufficient to classify the points in the second dataset; it would result in **underfitting**.

The process of optimally narrowing down $\mathcal{H}$ is called **model selection**.

1.3 Linear Classifiers

Now, we will look at some mathematical abstractions for these classifiers.

Our goal is to learn a linear decision boundary, meaning that we will constrain $\mathcal{H}$ to hyperplanes. For now, we also make the following simplifying assumptions:

- $\mathcal{H}$ contains only hyperplanes passing through the origin
- Any data sets we deal with are linearly separable

Recall the definition of hyperplane:

> **Definition 1.3.1.** A **hyperplane** in $\mathbb{R}^d$ specified by parameter vector $\vec{\theta} \in \mathbb{R}^d$ and offset $b \in \mathbb{R}$ is the set of points $\vec{x} \in \mathbb{R}^d$ such that
>
> $$\vec{\theta} \cdot \vec{x} + b = 0.$$

In $\mathbb{R}^2$, this is simply the set of lines:

$$\vec{\theta} \cdot \vec{x} + b = \theta_1 x_1 + \theta_2 x_2 + b = 0$$

$$\implies x_2 = -\frac{\theta_1}{\theta_2} x_1 + b.$$

Note that $\vec{\theta}$ is orthogonal to the hyperplane $\vec{\theta} \cdot \vec{x} = 0$ by the definition of the dot product.

> **Example 1.3.1.** Consider the hyperplane given by $\vec{\theta} = [10, 10]^\top$ and $b = 0$. How will it classify points in quadrant 1 versus 3?
>
> Let $\vec{x}^{(1)}$ be strictly in Q1. Then $x_1^{(1)}, x_2^{(1)} > 0$, so $\vec{\theta} \cdot \vec{x} = 10x_1^{(1)} + 10x_2^{(1)} > 0$. In other words, $\text{sign}(\vec{\theta} \cdot \vec{x}^{(1)}) = 1$.
>
> Analogously, if $\vec{x}^{(2)}$ is strictly in Q3, we find $\text{sign}(\vec{\theta} \cdot \vec{x}^{(2)}) = -1$.
>
> Incidentally, $\vec{x}^{(1)}$ is on the same side of the hyperplane as $\vec{\theta}$, while $\vec{x}^{(2)}$ is on the opposite. This leads us to the following claim:
>
> **Claim.** Any data point $\vec{x}^{(i)}$ on the opposite side of the hyperplane as $\vec{\theta}$ satisfies
>
> $$\text{sign}(\vec{\theta} \cdot \vec{x}^{(i)}) = -1$$

and any data point on the same side satisfies

$$\text{sign}(\vec{\theta} \cdot \vec{x}^{(i)}) = 1.$$

Proof. Recall the following property of the dot product:

$$\vec{\theta} \cdot \vec{x} = \|\theta\|_2 \|\vec{x}\|_2 \cos \alpha,$$

where α is the angle between $\vec{\theta}$ and $\vec{x}$. If $0^\circ \leq \alpha < 90^\circ$ or $270^\circ < \alpha \leq 360^\circ$, then $\vec{\theta}$ and $\vec{x}$ are on the same side of the hyperplane. In this instance, cosine (and thus the dot product) is positive.

If $90^\circ < \alpha < 270^\circ$, then the dot product is negative. In the case that $\alpha \in \{90^\circ, 270^\circ\}$, it is normal to the hyperplane, so the dot product is 0. □

So we can intuitively define a linear classifier $\vec{\theta}$ by predicting the class label of $\vec{x}$ using $\text{sign}(\vec{\theta} \cdot \vec{x})$. Any $\vec{x}$ on the same side of $\vec{\theta}$ has $\text{sign}(\vec{\theta} \cdot \vec{x}) = 1$, hence positive classification, and any on the other side is -1, hence negative classification. A data point that lies exactly on the hyperplane is not classified decisively, so by convention we say it is misclassified. Mathematically, we define the linear classifier as

$$h(\vec{x}; \vec{\theta}) = \text{sign}(\vec{\theta} \cdot \vec{x}).$$

Note that the choice of $\vec{\theta}$ determines not just how the hyperplane is oriented but also which side corresponds to which label. For instance, flipping the direction of $\vec{\theta}$ will preserve the magnitude of any $\vec{\theta} \cdot \vec{x}$, but it will invert its sign. This also inverts the classification of every point.

How do we choose $\vec{\theta}$? We try to minimize **training error**, which is a measure of the fraction of misclassified points in the dataset. Consider the quantity $y^{(i)} h(\vec{x}^{(i)}; \vec{\theta})$, which is the product of the actual and predicted labels for data point i.

Recall that both values are in $\{\pm 1\}$. Therefore, by simple casework, their product is 1 when they are equal and -1 when they differ. Therefore, we define training error as follows:

$$\begin{aligned} E_n(\vec{\theta}) &:= \frac{1}{n} \sum_{i=1}^{n} [\![y^{(i)} \neq h(\vec{x}^{(i)}; \vec{\theta})]\!] \\ &= \frac{1}{n} \sum_{i=1}^{n} [\![y^{(i)} h(\vec{x}^{(i)}) \leq 0]\!] \end{aligned}$$

Next, we will look at the perceptron algorithm, which chooses an optimal $\vec{\theta}$ by minimizing E_n.

1.4 The Perceptron Algorithm

Definition 1.4.1. Given training examples $S_n = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^n$, we say the data are **linearly separable** (without offset) if there exists $\vec{\theta} \in \mathbb{R}^d$ such that

$$y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) > 0$$

for any $i \in \{1, \dots, n\}$.

$\vec{\theta} \cdot \vec{x} = 0$ is called the **separating hyperplane**.

Here, we will introduce a way to find such a $\vec{\theta}$ given a linearly separable dataset.

Algorithm 1 Perceptron algorithm

on input $S_n = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^n$:
$k \leftarrow 0$, $\vec{\theta}^{(0)} \leftarrow \vec{0}$
while there exists a misclassified point **do**
  for $i \in \{1, \dots, n\}$ **do**
    if $y^{(i)}(\vec{\theta^{(k)}} \cdot \vec{x}^{(i)}) \leq 0$ **then**
      $\vec{\theta}^{(k+1)} = \vec{\theta}^{(k)} + y^{(i)}\vec{x}^{(i)}$
      k++
    end if
  end for
end while

If a point $\vec{x}^{(i)}$ is misclassified, the perceptron algorithm "includes" it in the newest $\vec{\theta}^{(k)}$ based on its actual label.

1.5 The Perceptron Algorithm (with offset)

Now we will consider hyperplanes that do not necessarily pass through the origin. Given a data point $\vec{x}^{(i)} \in \mathbb{R}^d$, we add a constant component to obtain $\vec{x}^{(i)\prime} = [1, \vec{x}^{(i)}]^\top$. This also implies $\vec{\theta}^{(k)\prime} \in \mathbb{R}^{d+1}$, where the first component is $b^{(k)} \in \mathbb{R}$, the offset of the hyperplane.

Note that if $\vec{\theta}^{(k)\prime}$ is a separating hyperplane, we have

$$y^{(i)}\begin{bmatrix} b^{(k)} \\ \vec{\theta}^{(k)} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \vec{x}^{(i)} \end{bmatrix} > 0 \implies y^{(i)}(b^{(k)} + \vec{\theta}^{(k)} \cdot \vec{x}^{(i)}) > 0,$$

where in the second inequality, we have the equation of a hyperplane with offset.

Algorithm 2 Perceptron algorithm with offset

on input $S_n = \{(\vec{x}^{(i)}, y^{(i)})\}_{i=1}^n$:
$k \leftarrow 0$, $\vec{\theta}^{(0)} \leftarrow \vec{0}$, $b^{(0)} = 0$
while there exists a misclassified point **do**
  for $i \in \{1, \dots, n\}$ **do**
    if $y^{(i)}(\vec{\theta}^{(k)} \cdot \vec{x}^{(i)}) \leq 0$ **then**
      $\vec{\theta}^{(k+1)} \leftarrow \vec{\theta}^{(k)} + y^{(i)}\vec{x}^{(i)}$
      $b^{(k+1)} \leftarrow b^{(k)} + y^{(i)}$
      $k \leftarrow k + 1$
    end if
  end for
end while

Theorem 1.5.1. *The perceptron algorithm converges.*

Proof. Exercise. :) □

2 (Stochastic) Gradient Descent

2.1 Empirical Risk

What if our data is not linearly separable? Then the perceptron algorithm does not converge, but our objective is the same: minimize the empirical risk (in this problem, the training error). Recall that this is given by

$$\begin{aligned} R_n(\vec{\theta}) &:= \frac{1}{n}\sum_{i=1}^{n}[\![y^{(i)} \neq h(\vec{x}^{(i)};\vec{\theta})]\!] \\ &= \frac{1}{n}\sum_{i=1}^{n}[\![y^{(i)}h(\vec{x}^{(i)}) \leq 0]\!] \end{aligned}$$

for linear classifiers. Let $z = y^{(i)}(\vec{\theta}\cdot\vec{x}^{(i)})$. Intuitively, this is how "badly" $\vec{\theta}$ misclassifies data point i; note that z is greatest in magnitude when $\vec{\theta}$ and $\vec{x}^{(i)}$ are either closest or furthest apart, which conveys "certainty." So, when $z < 0$, the classifier is highly certain of its incorrect prediction.

Note that the empirical risk above uses the 0-1 loss function:

$$\text{Loss}_{0-1}(z) = \begin{cases} 0 & \text{for } z > 0 \\ 1 & \text{otherwise} \end{cases}$$

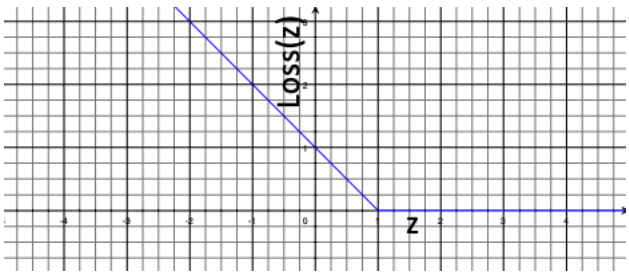
where, to compute empirical risk, we evaluate the average value of this loss function over every point in the data set. Unfortunately, minimizing this function is NP-hard. So, let's consider some alternative loss functions. Firstly, we write the general formula for empirical risk:

$$R_n(\vec{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\text{Loss}(h(\vec{x}^{(i)};\vec{\theta}), y^{(i)}).$$

Now consider the hinge loss function:

$$\text{Loss}_{\text{hinge}}(z) = \max(1-z, 0).$$

Its graph is as follows:



This yields the following empirical risk:

$$R_n(\vec{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\max(1 - y^{(i)}(\vec{\theta}\cdot\vec{x}^{(i)}), 0).$$

Immediately, we note some advantages. Hinge loss applies a greater penalty to more severe misclassifications, but it also penalizes for being only slightly correct. This forces the classifier to be more than "somewhat accurate."

More importantly, however, this R_n is a convex function, meaning that we can minimize it using **gradient descent**.

2.2 Gradient Descent

Algorithm 3 Gradient descent

$k \leftarrow 0$, $\vec{\theta}^{(0)} \leftarrow \vec{0}$
while convergence criteria not met **do**
    $\vec{\theta}^{(k+1)} = \vec{\theta}^{(k)} - \eta \nabla_{\vec{\theta}} R_n(\vec{\theta})|_{\vec{\theta}=\vec{\theta}^{(k)}}$
    $k \leftarrow k + 1$
end while

A convex function, like $f(x) = x^2$, is "bowl-shaped." This means that the gradient at any point on the function points away from the bottom of the bowl; consider $f(x)$ as an example. When $x > 0$, the derivative is positive, which points away from the vertex at $x = 0$.

For convex minimization problems, we leverage this by taking small steps in the direction *opposite* the gradient. In our case, we have a function of $\vec{\theta}$ versus $R_n(\vec{\theta})$ that we want to minimize by finding the value of $\vec{\theta}$ that minimizes $R_n(\vec{\theta})$. So, we take small steps updating $\vec{\theta}$ opposite the gradient of $R_n(\vec{\theta})$.

For convergence criteria, we can use a combination of the following:

- $R_n(\vec{\theta})$ is less than some amount
- $\nabla_{\vec{\theta}} R_n(\vec{\theta})$ is less than some amount
- $\vec{\theta}$ does not change by much
- Some amount of iterations have passed

We can set η, the step size, either as a constant or a variable in terms of k. A constant η can pose issues if we don't choose it appropriately; too large and the algorithm will oscillate, but too small and it will take too long. We can also opt to set η as such:

$$\eta_k = \frac{1}{k+1}.$$

Intuitively, this says "make large updates at the beginning but slow down once we've made a few," which is sensible.

However, a major flaw of gradient descent is that it, in every update step, it computes the gradient of R_n. Recall that R_n is the average loss across all data points, and thus one GD update requires us to look at *every* training data point.

Clearly, this is infeasible if our dataset is large. We want some way to update $\vec{\theta}^{(k)}$ without looking at every point.

2.3 Stochastic Gradient Descent

The idea of SGD is to reduce computational cost by updating the hyperplane based on just one point, rather than the whole data set.

Algorithm 4 Stochastic gradient descent

$k \leftarrow 0$, $\vec{\theta}^{(0)} \leftarrow \vec{0}$
while convergence criteria not met **do**
  randomly shuffle points
  for $i \in \{1, \ldots, n\}$ **do**
    $\vec{\theta}^{(k+1)} = \vec{\theta}^{(k)} - \eta \nabla_{\vec{\theta}} \text{Loss}_h(y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}))|_{\vec{\theta}=\vec{\theta}^{(k)}}$
    $k \leftarrow k + 1$
  end for
end while

Note that, in each update, we only compute the gradient of the loss of a single point. For hinge loss, there are two possible outcomes of the update step. If $z \geq 1$, then the loss and gradient are 0, so no update is made. If $z < 1$, then hinge loss becomes $1 - y^{(i)}(\vec{\theta} \cdot \vec{x})$, whose gradient with respect to $\vec{\theta}$ is $-y^{(i)}\vec{x}^{(i)}$.

3 SVMs and Kernels

3.1 Hard Margin SVMs

Assume linearly separable data.

The principle behind picking a separating hyperplane is that we want it to classify the training set correctly and be maximally removed from training examples closest to the decision boundary (maximum margin separator). We define the **suport vectors** to lie parallel to the hyperplane on these closest training points.

Mathematically, we want $\vec{\theta}$, b such that

$$y^{(i)}(\vec{\theta} \cdot \vec{x} + b) \geq 1$$

for all i (recall hinge loss).

Now define

$$\gamma^{(i)}(\vec{\theta}, b) = \frac{y^{(i)}(\vec{\theta} \cdot \vec{x} + b)}{\|\vec{\theta}\|}$$

as the distance from $\vec{x}^{(i)}$ to the hyperplane. Note that $\gamma^{(i)}$ has a lower bound of $\frac{1}{\|\vec{\theta}\|}$ due to the above constraint.

If we have multiple separating hyperplanes, we want to pick the $\vec{\theta}$ and b that maximize the minimal distance from the hyperplane to a training point. In other words, we want

$$\max_{\vec{\theta}, b} \min_i \gamma^{(i)}(\vec{\theta}, b) \text{ subject to } y^{(i)}(\vec{\theta} \cdot \vec{x} + b) \geq 1.$$

Now we will look for a closed form expression of the min term. Recalling the definition of γ and the hinge loss constraint, we claim that $\min_i \gamma^{(i)}(\vec{\theta}, b) = \frac{1}{\|\vec{\theta}\|}$.

It is true that

$$y^{(i)}(\vec{\theta} \cdot \vec{x} + b) > 1 \implies \gamma^{(i)}(\vec{\theta}, b) > \frac{1}{\|\vec{\theta}\|},$$

but note that in this case we can scale $\vec{\theta}$ and b down to obtain strict equality in the constraint. Therefore, if the above holds for every point, we can obtain a smaller margin by modifying $\vec{\theta}$ and b:

$$\vec{\theta} \longrightarrow \frac{\vec{\theta}}{y^{(i)}(\vec{\theta} \cdot \vec{x} + b)}$$

$$b \longrightarrow \frac{b}{y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)} + b)},$$

where i is the closest data point.

So $\min_i \gamma^{(i)}(\vec{\theta}, b) = \frac{1}{\|\vec{\theta}\|}$, which lets us rewrite the optimization problem as such:

$$\max_{\vec{\theta},b} \min_i \gamma^{(i)}(\vec{\theta}, b)$$
$$\implies \max_{\vec{\theta},b} \frac{1}{\|\vec{\theta}\|}$$
$$\implies \min_{\vec{\theta},b} \frac{\|\vec{\theta}\|^2}{2}.$$

We have just obtained a **quadratic program**, which is computationally easy to minimize. The term inside the "min" is called the **objective function**, which we wish to minimize while satisfying the constraint $y^{(i)}(\vec{\theta} \cdot \vec{x} + b) \geq 1$.

This process of manipulating $\vec{\theta}$ and b reflects the fact that the *constraint cannot be violated*, and that we may only modify the objective function in finding an optimal solution.

3.2 Soft Margin SVMs

What if the dataset is not linearly separable? In this situation, not every point will satisfy $y^{(i)}(\vec{\theta} \cdot \vec{x} + b) \geq 1$. Either a point is on the wrong side, in which case $y^{(i)}(\vec{\theta} \cdot \vec{x} + b) < 0$, or it is classified correctly but without certainty: $0 < y^{(i)}(\vec{\theta} \cdot \vec{x} + b) < 1$.

To account for this, we will modify the optimization problem as follows:

$$\min_{\vec{\theta},b} \frac{\|\vec{\theta}\|}{2} + C \sum_{i=1}^{n} \xi_i$$

$$\text{subject to } y^{(i)}(\vec{\theta} \cdot \vec{x} + b) \geq 1 - \xi_i.$$

ξ_i is called the slack variable, and it allows the constraint to be violated at the expense of an additional penalty in the objective function. Therefore, the classifier still has an incentive to classify points according to the original constraint.

Note that if a point is so misclassified that it appears on the wrong side, ξ_i will be very large. Recall that the support vectors are a distance $\frac{1}{\|\vec{\theta}\|}$ from the hyperplane; a point's slack variable is determined relative to the support vector on its side.

C is a hyperparameter that affects how much the classifier should be penalized for using slack variables. When $C \longrightarrow \infty$, the penalty for misclassifications becomes infinite, and we approach the behavior of a hard-margin SVM. For lower values of C, the margin loosens and we start to see data points within the margins.

Apart from being able to classify data sets that are not linearly separable, soft-margin SVMs also have higher tolerance to overfitting because they are willing to misclassify outliers to obtain better overall performance.

3.3 Feature Maps

What if we want a circular decision boundary? Recall the equation of a circle of radius 1 centered at $(2, 2)$.

$$(x_1 - 2)^2 + (x_2 - 2)^2 = 1$$

$$\implies x_1^2 + x_2^2 - 4x_1 - 4x_2 + 7 = 0.$$

Now we will define the following mapping from $\mathbb{R}^2$ to $\mathbb{R}^5$:

$$\phi(\vec{x}) = [x_1^2, x_2^2, x_1, x_2, 1]^\top.$$

Since any point in $\mathbb{R}^5$ can also be predicted using a linear classifier, we will construct one in terms of the points of the above form. Consider the following $\vec{\theta}$:

$$\vec{\theta} = [1, 1, -4, -4, 7]^\top.$$

and note that $\phi(\vec{x}) \cdot \vec{\theta} = 0$, when expanded, is exactly the expression for the circular decision boundary.

This is the idea behind **feature maps**, which send data to a higher-dimensional space in which a separating hyperplane exists; in the lower-dimensional space, the hyperplane will then correspond to a non-linear decision boundary.

3.4 Dual SVMs

Recall the quadratic program formulation for hard-margin SVM with no offset:

$$\min_{\vec{\theta}} \frac{\|\vec{\theta}\|^2}{2}$$

$$\text{subject to } y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) \geq 1.$$

We want to rewrite this in the **dual form**:

$$\max_{\vec{\alpha}, \alpha_i \geq 0} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \vec{x}^{(i)} \cdot \vec{x}^{(j)}.$$

The output is a $\vec{\theta}^*$ in terms of the α_i, which is the same $\vec{\theta}$ we would have gotten from solving the QP.

$$\vec{\theta}^* = \sum_{i=1}^{n} \hat{\alpha}_i y^{(i)} \vec{x}^{(i)}.$$

This is important because mapping to a higher dimensional space requires us to compute the computationally expensive quantity $\phi(\vec{x}^{(i)})$. With the dual form, we only need $\phi(\vec{x}^{(i)}) \cdot \phi(\vec{x}^{(j)})$, which is sometimes easier to compute than separately finding each term.

Now we will derive the dual form in general. Recall the general form of a constrained optimization problem:

$$\min_{\vec{w}} f(\vec{w}) \text{ such that } h_i(\vec{w}) \leq 0$$

for $i \in \{1, \ldots, n\}$. In other words, we want to minimize $f(\vec{w})$ while satisfying $h_1(\vec{w}) \leq 0$, $h_2(\vec{w}) \leq 0$, etc.

We write the Lagrangian:

$$L(\vec{w}, \vec{\alpha}) = f(\vec{w}) + \sum_{i=1}^{n} \alpha_i h_i(\vec{w})$$

for non-negative α_i. Now we define the following function:

$$g_p(\vec{w}) = \max_{\vec{\alpha}, \alpha_i \geq 0} L(\vec{w}, \vec{\alpha}).$$

If all constraints are satisfied, then $h_i(\vec{w}) \leq 0$ for all i. Since α_i is by definition non-negative, the only way to maximize $L(\vec{w}, \vec{\alpha})$ is to send all the α_i to zero. In this case, then, $g_p(\vec{w}) = f(\vec{w})$.

If constraint j is violated, then $h_j(\vec{w}) > 0$. To maximize $L(\vec{w}, \vec{\alpha})$, we then send α_j to infinity.

Therefore, $g_p(\vec{w})$ has the property that it is $f(\vec{w})$ is all constraints are satisfied, and ∞ otherwise. In other words, if all constraints are satisfied, then

$$\min_{\vec{w}} g_p(\vec{w}) = \min_{\vec{w}} \max_{\vec{\alpha}, \alpha_i \geq 0} L(\vec{w}, \vec{\alpha}) = \min_{\vec{w}} f(\vec{w}).$$

There is a name for this representation of the problem: the **primal formulation**.

$$\min_{\vec{w}} \max_{\vec{\alpha}, \alpha_i \geq 0} L(\vec{w}, \vec{\alpha}).$$

Swapping the max and min gives the **dual formulation**.

$$\max_{\vec{\alpha}, \alpha_i \geq 0} \min_{\vec{w}} L(\vec{w}, \vec{\alpha}).$$

The difference between these solutions is called the **duality gap**, which gives a lower bound on the size of the primal. For our problem, the dual gap is zero, so we can solve the dual form for a solution to both formulations.

With this in mind, we find the dual formulation for the QP formulation of the hard-margin SVM without offset:

$$\min_{\vec{\theta}} \frac{\|\vec{\theta}\|^2}{2} \text{ subject to } y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) \geq 1.$$

There are n constraints for this problem; one for every data point. First, we write the Lagrangian:

$$L(\vec{\theta}, \vec{\alpha}) = \frac{\|\vec{\theta}\|^2}{2} + \sum_{i=1}^{n} \alpha_i (1 - y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)})).$$

Now we write the dual formulation:

$$\max_{\alpha, \alpha_i \geq 0} \min_{\vec{\theta}} L(\vec{\theta}, \vec{\alpha}).$$

To minimize $L(\vec{\theta}, \vec{\alpha})$, we set $\nabla_{\vec{\theta}} L(\vec{\theta}, \vec{\alpha})|_{\vec{\theta} = \vec{\theta}^*} = 0$ and solve for $\vec{\theta}^*$:

$$\nabla_{\vec{\theta}} \left[\frac{\|\vec{\theta}\|^2}{2} + \sum_{i=1}^{n} \alpha_i - \sum_{i=1}^{n} \alpha_i y^{(i)} (\vec{\theta} \cdot \vec{x}^{(i)}) \right] = \vec{\theta}^* - \sum_{i=1}^{n} \alpha_i y^{(i)} \vec{x}^{(i)} = 0.$$

Plugging this value of $\vec{\theta}^*$ back into the Lagrangian, we obtain

$$\max_{\vec{\alpha}, \alpha_i \geq 0} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \vec{x}^{(i)} \cdot \vec{x}^{(j)}.$$

The solutions $\hat{\alpha}_i$ of this optimization problem have an interesting property. Recall the Lagrangian:

$$L(\vec{\theta}, \vec{\alpha}) = \frac{\|\vec{\theta}\|^2}{2} + \sum_{i=1}^{n} \alpha_i (1 - y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)})).$$

If $\vec{\theta}^*$ satisfies the constraints, as discussed in the general case, then the sum term is zero. This implies that $\alpha_i(1 - y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)})) = 0$. Therefore, if point i satisfies $\hat{\alpha}_i > 0$ (since α is always non-negative), we know $y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) = 1$. This is exactly the condition for $\vec{x}^{(i)}$ being on the margin boundary (making it a **support vector** for a hard-margin SVM).

On the other hand, if $y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) > 1$, then $\hat{\alpha}_i = 0$. Barring edge cases, either the primal inequality is satisfied with equality or the dual variable is zero. This provides insight into why support vectors are so important to SVMs; the only vectors that contribute to $\vec{\theta}^*$,

$$\vec{\theta}^* = \sum_{i=1}^{n} \alpha_i y^{(i)} \vec{x}^{(i)},$$

are the ones with nonzero α_i. These are simply the support vectors.

In a hard-margin SVM, support vectors can only lie on the margin because they are the only points that satisfy the hard-margin constraint with equality: $y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) = 1$. In a soft-margin SVM, though, we account for slack variables: $y^{(i)}(\vec{\theta} \cdot \vec{x}^{(i)}) = 1 - \xi_i$. This implies that any point within the margin on its side can be a support vector. Note that this includes misclassified points as well as weakly-classified points.

3.5 The Kernel Trick

Recall the motivation behind feature maps: finding non-linear decision boundaries. This involves mapping each data point to a higher dimension and then learning a linear decision boundary in that space. In the dual form, this looks like:

$$\max_{\vec{\alpha}, \alpha_i \geq 0} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \vec{x}^{(i)} \cdot \vec{x}^{(j)}$$

$$\implies \max_{\vec{\alpha}, \alpha_i \geq 0} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y^{(i)} y^{(j)} \phi(\vec{x}^{(i)}) \cdot \phi(\vec{x}^{(j)}).$$

The idea of a kernel is to define a function $K : \mathbb{R}^d \times \mathbb{R}^d \longrightarrow \mathbb{R}$ such that

$$K(\vec{x}^{(i)}, \vec{x}^{(j)}) = \phi(\vec{x}^{(i)}) \cdot \phi(\vec{x}^{(j)}),$$

which is theoretically easier to compute than finding $\phi(\vec{x}^{(i)})$ and $\phi(\vec{x}^{(j)})$ explicitly. Intuitively, $K(\vec{x}^{(i)}, \vec{x}^{(j)})$ is a measure of similarity between $\vec{x}(i)$ and $\vec{x}^{(j)}$, just like the dot product.

Example 3.5.1. Consider a feature map $\phi(\vec{u}) = [u_1^2, u_2^2, \sqrt{2}u_1u_2]^\top$ for $\vec{x} \in \mathbb{R}^2$. Then

$$\begin{aligned} \phi(\vec{u}) \cdot \phi(\vec{v}) &= [u_1^2, u_2^2, \sqrt{2}u_1u_2]^\top \cdot [v_1^2, v_2^2, \sqrt{2}v_1v_2]^\top \\ &= u_1^2v_1^2 + u_2v_2^2 + 2u_1u_2v_1v_2 = (\vec{u} \cdot \vec{v})^2. \end{aligned}$$

So we can define $K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v})^2$ as a proxy for $\phi(\vec{u}) \cdot \phi(\vec{v})$ given this particular feature map ϕ.

But the decision boundary learned in a higher-dimensional space will also be in that higher dimension, so if we want to classify a point $\vec{x}$ in the original space, we have to evaluate $\phi(\vec{x})$, right?

No. Note that we can rewrite the classification as:

$$\left(\sum_{i=1}^{n} \alpha_i y^{(i)} \phi(\vec{x}^{(i)})\right) \phi(\vec{x})$$
$$= \sum_{i=1}^{n} \alpha_i y^{(i)} K(\vec{x}^{(i)}, \vec{x}).$$

4 Regression and Regularization

4.1 Regression

The practice of predicting discrete labels is called classification, while predicting continuous labels is called **regression**. In binary classification, the labels are of the form $y \in \{-1, 1\}$, while in regression, they are $y \in \mathbb{R}$.

To perform linear regression, we use a function linear in $\vec{\theta}$:

$$f(\vec{x}; \vec{\theta}.b) = \vec{\theta} \cdot \vec{x} + b.$$

Whereas in classification, the input z to the empirical risk is the product of the label and prediction, the input to empirical risk for regression is the *difference* between the label and prediction.

With a different loss input also comes a different loss function. Here, we will consider the **squared loss** function:

$$\text{Loss}(z) = \frac{z^2}{2}$$

for $z = y^{(i)} - \vec{\theta} \cdot \vec{x}^{(i)}$. The idea of this function is to allow small inaccuracies but quadratically penalize larger ones. Since it is clearly convex, we can use SGD to minimize it; the only novel part is calculating the update term.

$$\eta_k \nabla_{\vec{\theta}} \text{Loss}(y^{(i)} - \vec{\theta} \cdot \vec{x})$$
$$= \eta_k \nabla_{\vec{\theta}} \left(\frac{(y^{(i)} - \vec{\theta} \cdot \vec{x}^{(i)})^2}{2}\right)$$
$$= \eta_k (y^{(i)} - \vec{\theta} \cdot \vec{x}^{(i)})(-\vec{x}^{(i)})$$

It turns out we can derive a closed-form solution for empirical risk using squared loss. We first write the expression for empirical risk:

$$R_n(\vec{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \frac{(y^{(i)} - \vec{\theta} \cdot \vec{x}^{(i)})^2}{2}.$$

Now we set its gradient to 0 and solve for $\vec{\theta}^*$:

$$\begin{aligned}
\nabla_{\vec{\theta}} R_n(\vec{\theta}) &= \frac{1}{n}\sum_{i=1}^{n} \nabla\vec{\theta}\left(\frac{(y^{(i)} - \vec{\theta}\cdot\vec{x}^{(i)})^2}{2}\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \vec{\theta}\cdot\vec{x}^{(i)})(-\vec{x}^{(i)}) \\
&= -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\vec{x}^{(i)} + \frac{1}{n}\sum_{i=1}^{n}(\vec{\theta}\cdot\vec{x}^{(i)})\vec{x}^{(i)} \\
&= -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\vec{x}^{(i)} + \frac{1}{n}\sum_{i=1}^{n}\vec{x}^{(i)}(\vec{x}^{(i)})^\top\vec{\theta}^*
\end{aligned}$$

Now let

$$\vec{b} = \frac{1}{n}\sum_{i=1}^{n} y^{(i)}\vec{x}^{(i)}$$

$$A = \frac{1}{n}\sum_{i=1}^{n}\vec{x}^{(i)}(\vec{x}^{(i)})^\top\vec{\theta}^*.$$

Define the following:

$$X = [\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}]^\top = \begin{bmatrix} x_1^{(1)} & \cdots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \cdots & x_d^{(n)} \end{bmatrix}$$

$$\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^\top.$$

Therefore,

$$\vec{b} = \frac{1}{n}X^\top\vec{y}$$

$$A = \frac{1}{n}X^\top X.$$

We can easily verify these by looking at a few rows/entries.

Returning to the risk function, we obtain a solution for $\vec{\theta}^*$.

$$\begin{aligned}
0 = R_n(\vec{\theta}) &= -\frac{1}{n}\sum_{i=1}^{n} y^{(i)}\vec{x}^{(i)} + \frac{1}{n}\sum_{i=1}^{n}\vec{x}^{(i)}(\vec{x}^{(i)})^\top\vec{\theta}^* \\
&= -\vec{b} + A\vec{\theta}* \\
&\implies \vec{\theta}^* = A^{-1}\vec{b} \\
&= \left(\frac{1}{n}X^\top X\right)^{-1}\left(\frac{1}{n}X^\top\vec{y}\right) \\
&= (X^\top X)^{-1}X^\top\vec{y}
\end{aligned}$$

This is a closed-form solution to minimize empirical risk with squared loss. In some cases, we might still want to use SGD if the computations involved in computing the closed form are expensive.

If $X^\top X$ is not invertible, then the columns are linearly dependent, which implies redundancy in the features. The process of removing these redundancies is called **regularization**.

4.2 Regularization

In general, there is a tradeoff between bias and variance in which high bias corresponds to more resilience to noise but less accurate predictions, and high variance corresponds to higher training accuracy but greater risk of overfitting.

The function with smallest variance is the constant function—it is highly rigid and thus would yield low variance if we were to test it on a new dataset. Obviously, though, it would have very high error in general, making it high-bias.

On the other hand, to minimize bias, we could use something like a high degree polynomial or RBF kernel. These methods yield low training error (i.e., low bias) but are likely to vary greatly on new data due to overfitting (i.e., high variance).

Here, we will look at regularization, which is a way to minimize variance. The idea is to impose a penalty on the model that increases as the model increases in complexity. We do this by adding a term to the empirical risk:

$$J_{n,\lambda} = \lambda Z(\vec{\theta}) + R_n(\vec{\theta}).$$

λ is a hyperparameter. $Z(\vec{\theta})$ should be convex and smooth so that it fits into our convex optimization strategy. It should also force the components of $\vec{\theta}$ to be very small in magnitude since $\vec{\theta} \cdot \vec{x}$ and $\vec{\theta} \cdot \vec{x'}$ differ greatly if $\vec{\theta}$ is large.

We will first consider **ridge (L_2) regression**, which means that $Z(\vec{\theta}) = \frac{\|\vec{\theta}\|^2}{2}$. If we apply this to squared loss, we get

$$J_{n,\lambda}(\vec{\theta}) = \lambda \frac{\|\vec{\theta}\|^2}{2} + \frac{1}{n} \sum_{i=1}^{n} \frac{(y^{(i)} - \vec{\theta} \cdot \vec{x}^{(i)})^2}{2}.$$

When $\lambda = 0$, the regularization has no effect and this is simply squared loss regression. When $\lambda \to \infty$, the model prioritizes minimizing the regularization term, so it sends $\vec{\theta}$ to $\vec{0}$.

To find a closed form solution that minimizes $J_{n,\lambda}$, we set the gradient with respect to $\vec{\theta}$ as 0 and find $\vec{\theta}^*$:

$$\vec{\theta}^* = (\lambda I + A)^{-1} b$$
$$= (\lambda' I + X^\top X)^{-1} X^\top \vec{y}$$

4.3 Feature Selection

This topic is motivated by the need to remove uninformative features that could make our models prone to overfitting.

The first is the **filter approach**, where we evaluate each feature using some metric (e.g., Pearson's correlation with output) and discard the ones below a threshold. We can also use the **wrapper approach**, which selects the best subset of features by scoring a series of subsets using a learning algorithm.

Finally, we can use **embedded methods**, which use variable selection as part of the training process, like L_1 or L_2 regularization.

5 Decision Trees

5.1 Basic Definitions

A **decision tree** is a tree in which each internal node tests a feature x_i, each branch considers an outcome $x_i = v$, and each leaf assigns a label. Thus, decision trees are generally very interpretable.

Here is an example of a decision tree constructed from a small dataset:

x_1	x_2	y
1	1	F
0	1	T
1	0	T
0	0	F

With this dataset, the choice to split on x_1 first was arbitrary. However, there are cases where order matters. Take the following example:

x_1	x_2	y
1	1	T
0	1	F
1	0	T
0	0	F

Since x_1 decides y entirely on its own, it makes sense to split on x_1 first to minimize the depth of the decision tree. If we split on x_2 first, we would need an additional layer. This reflects our overall goal: to find the *smallest* decision tree that minimizes error.

Unfortunately, this problem is NP-hard. However, we can use a heuristic that takes advantage of **entropy**, which measures the uncertainty of a value. Intuitively, we want to split on whichever feature reduces uncertainty the most.

For binary classification, entropy is defined as follows:

$$H(S_n) := -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus,$$

where $p_\oplus$ and $p_\ominus$ are the proportion of positive and negative examples, respectively. For a discrete random variable Y that takes on values $\{y_1, \ldots, y_k\}$, the entropy of Y is generalized to:

$$H(Y) := -\sum_{i=1}^{k} P(Y = y_i) \log_2 P(Y = y_i).$$

High entropy corresponds to a uniform distribution because there is high uncertainty in predicting the value. Low entropy corresponds to a more varied distribution.

More relevant to our decision trees is the idea of **conditional entropy**. Below is how to compute the entropy of Y given $X = x$:

$$H(Y|X = x) = -\sum_{i=1}^{k} P(Y = y_i|X = x) \log_2 P(Y = y_i|X = x),$$

which we use to compute the conditional entropy of Y given X:

$$H(Y|X) = \sum_{j=1}^{m} P(X = x_j)H(Y|X = x_j).$$

This suggests a way to decide which feature to split on; evaluate the conditional entropy $H(Y|X)$ for each feature X and pick whichever minimizes it. We can formulate this problem the opposite way–maximize the decrease in entropy, which we call information gain:

$$IG(X, Y) = H(Y) - H(Y|X).$$

Note that this value will always be in the range $[0, H(Y)]$.

With this in mind, we can devise a primitive algorithm for generating decision trees: starting on an empty tree, recursively split on the feature that maximizes information gain. We would like to have stopping criteria, though, in case the process is computationally expensive or the dataset is large.

The two we will consider are:

- When every point has the same label
- When every point has the same features

In the first case, we set every remaining leaf to the remaining label. In the second, we take the majority label. This yields the following algorithm:

Algorithm 5 Learning decision trees

if $y^{(i)} = y$ for all examples in DS **then return** y
end if
if $\vec{x}^{(i)} = \vec{x}$ for all examples in DS **then return** majority label
end if
$j \leftarrow \arg\min H(y|x_j)$
for each value v of x_j **do**
    $DS_v \leftarrow$ {examples in DS where $x_j = v$}
    recurse on DS_v
end for

In theory, decision trees can achieve zero training error, assuming no noise. Therefore, we need regularization to minimize complexity. We can set a maximum depth, set a minimum number of samples per leaf, or grow the tree fully and prune it.

6 Ensemble Methods

6.1 Bagging

Since decision trees are prone to overfitting, we want to increase variance without decreasing their bias. This leads to the idea of training multiple decision trees and averaging their predictions, due to the fact that averaging reduces variance.

Concretely, let $Z_1, \dots, Z_n$ be IID random variables. Then

$$\begin{aligned} &\mathrm{Var}[Z_i] = \sigma^2 \\ = \mathrm{Var}&\left[\frac{1}{k}\sum_{i=1}^{k} Z_i\right] = \frac{1}{k^2}\mathrm{Var}\left[\sum_{i=1}^{k} Z_i\right] \\ &= \frac{1}{k^2}\sum_{i=1}^{k}\sigma^2 = \frac{1}{k^2}\cdot k\sigma^2 \\ &= \frac{\sigma^2}{k}. \end{aligned}$$

This approaches 0 as $k \to \infty$.

To train multiple trees, we need multiple datasets. We can achieve this using **bootstrap sampling**, where we sample n data points independently from S_n at random with replacement. This ensures that each dataset is produced from the same distribution.

After we obtain k datasets, we can train k decision trees and use majority vote to find the aggregate decision from all. This process is called **bootstrap aggregating**, or **bagging**. However, note that the datasets obtained from bootstrap sampling are not independent, so we have to redo our variance analysis.

The $Z_1, \dots, Z_k$ are identically distributed as before, but now each pair of Z_i and Z_j has a nonzero correlation $\rho(Z_i, Z_j) \in [0, 1]$. So, for the aggregate variance, we have

$$\mathrm{Var}\left[\frac{1}{k}\sum_{i=1}^{k} Z_i\right] = \rho\sigma^2 + \frac{1-\rho}{k}\sigma^2.$$

We can send the second term to 0 as before by letting $k \to \infty$, but the first term still lingers.

6.2 Boosting

The idea of **boosting** is to combine many weak learners into a strong learner. Our only requirement for the weak learner is that it has > 50% error i.e., it is better than a random guess. For this course, we will study **adaptive boosting**, or **AdaBoost**.

For our weak classifiers, we will use decision stumps, which are essentially primitive linear classifiers. Let C be the set of decision stumps we start with. Then each $h \in C$ is defined as

$$h(\vec{x}; \vec{\theta}) = \mathrm{sign}(\theta_1(x_k - \theta_0)).$$

Here, $\vec{\theta} = [k, \theta_0, \theta_1]^\top$. k is the coordinate; that is, the stump classifies points along the x_k axis. θ_0 is the threshold, which is the point on the axis through which the stump passes. θ_1 is the

direction the stump classifies "positive." Essentially, a decision stump is a linear classifier that lies parallel to an axis.

We can also write $\vec{\theta} \in \{1, 2, \dots, d\} \times \mathbb{R} \times \{1, -1\}$. It is easy to see that we can find the optimal decision stump in linear time relative to the number of points.

The idea of AdaBoost is to assign a weight to each training example. In each iteration, we train a decision stump on the dataset and its weights; the weight of misclassified points is increased. When this process finishes, we output the final classifier as a weighted sum of the decision stumps.

Mathematically, we start by initializing the weights $\tilde{w}_0^{(i)} = \frac{1}{n}$ for $i \in \{1, \dots, n\}$. Then, in each iteration, starting from $m = 1$, we find the best decision stump with respect to the weights from iteration $m - 1$:

$$h_m = \arg\min_{h \in C} \sum_{i=1}^{n} \tilde{w}_{m-1}^{(i)} [\![y^{(i)} \neq h(\vec{x}^{(i)})]\!].$$

We denote the weighted error of h_m by ϵ_m:

$$\epsilon_m = \sum_{i=1}^{n} \tilde{w}_{m-1}^{(i)} [\![y^{(i)} \neq h_m(\vec{x}^{(i)})]\!].$$

This always satisfies $\epsilon_m \in [0, \frac{1}{2}]$ because the weak classifier, by definition, has classification accuracy of 50% or higher. The final weight of each decision stump is denoted α_m:

$$\alpha_m = \frac{1}{2} \ln \frac{1 - \epsilon_m}{\epsilon_m}.$$

Finally, we update weights of the training examples:

$$\tilde{w}_m^{(i)} = \frac{\tilde{w}_{m-1}^{(i)} \cdot \exp(-\alpha_m y^{(i)} h_m(\vec{x}^{(i)}))}{Z_m},$$

where the Z_m is simply a normalization constant to make sure the sum of the $\tilde{w}_m^{(i)}$ is 1. Note the similarity between this update step and the perceptron update step. In both algorithms, we either increase or decrease some quantity based on $y^{(i)} h_m(\vec{x}^{(i)})$, or whether the classifier correctly predicts point i.

The difference is that in this update step, we increase the weight when a point is classified incorrectly, as per above. The final classifier is given by

$$\text{sign}\left(\sum_{i=1}^{M} \alpha_m h_m(\vec{x}) \right).$$

Now we will look at some extreme cases. If $\epsilon_m = \frac{1}{2}$, then $\alpha_m = 0$, which implies that h_m does not contribute to the final classifier. Moreover, by the weight update step, $\alpha_m = 0$ implies that iteration m does not alter the weights. So the next iteration yields the same weak classifier with the same α, and so on. Therefore, if $\epsilon_m = \frac{1}{2}$, the algorithm terminates.

If $\epsilon_m = 0$, then $\alpha_m \to \infty$. Then h_m should have infinite weight, meaning that it is the only classifier necessary. Intuitively, this makes sense. $\epsilon_m = 0$ means that $y^{(i)} = h_m(\vec{x}^{(i)})$ holds for every point, so we only need h_m to perfectly classify the dataset.

Another property of AdaBoost is that it minimizes the exponential loss function $\exp(-z)$. Let $H_{m-1} = \sum_{t=1}^{m-1} \alpha_t h_t$ be the ensemble model before iteration m.

Theorem 6.2.1. *At iteration m, AdaBoost greedily adds a new weak classifier to minimize exponential loss:*

$$h_m, \alpha_m = \arg\min_{h \in C, \alpha \geq 0} L(H_{m-1}, \alpha h).$$

Theorem 6.2.2. *Suppose that the weighted error in each iteration satisfies $\epsilon_m \leq \frac{1}{2} - \gamma$ for positive γ. Then if the number of iterations satisfies $m > \frac{\ln n}{2\gamma^2}$, the AdaBoost output classifier must have zero training error.*

Theorem 6.2.3. *The weighted error of h_m under new weights $\{\tilde{w}_m^{(i)}\}_{i=1}^n$ is always $\frac{1}{2}$.*

7 Neural Networks