Lecture Notes

Machine Learning

Winter Semester 24/25

Emma Bach

Lecture by
Dr. Daniele Cattaneo

March 3, 2025

Table of contents

Chapter 1

Introduction

Given a set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, i.e. a set of features (inputs) $x = \{x_1, \ldots, x_n\}$ and a target (desired output) $y = \{y_1, \ldots, y_n\}$, Machine Learning is the process of algorithmically finding the best possible function $f(x)$ such that

$$\forall x_i [f(x_i) \approx y_i].$$

y is also known as the **ground truth**. In **unsupervised learning**, y is not explicitly given to the algorithm (and may not even exist). The focus of the course, and thus these notes, was **supervised learning**, where y is explicitly given.

In order to find an optimal $f(x)$, we first rephrase the function in terms of **learnable parameters** θ to get a function $f(x, \theta)$. We often write $f(x_i, \theta) = \hat{y}_i$, where the hat $\hat{}$ is supposed to show that $\hat{y}_i$ is an approximation of y_i. The full image $\hat{y}$ is often called the **prediction model**.

We now need to define a seperate function that we can use to judge how good our values of θ are. Such a function is known as a **loss function**, which we write as $\mathcal{L}(y, \hat{y})$. The problem of finding the best possible f then comes down to a minimization problem of the form

$$\underset{\theta}{\text{argmin}} \sum_{n=1}^{N} \mathcal{L}(y_n, \hat{y}_n)$$

This function that we want to minimize is also called the **empirical risk**.

1.1 Basic Prediction Models

Common basic prediction models include:

- Linear Models:

$$\hat{y_n} = \theta_0 + \sum_{m=1}^{M} \theta_m x_{n,m}$$

- Polynomial Regression Models:

$$\begin{aligned}\hat{y_n} = \theta_0 &+ \sum_{m=1}^{M} \theta_m x_{n,m} \\ &+ \sum_{m=1}^{M} \sum_{m'=1}^{M} \theta_{m,m'} x_{n,m} x_{n,m'} \\ &+ \sum_{m=1}^{M} \sum_{m'=1}^{M} \sum_{m''=1}^{M} \theta_{m,m',m''} x_{n,m} x_{n,m'} x_{n,m''} \\ &+ \ldots\end{aligned}$$

- Decision Trees
- Neural Networks

1.1.1 Decision Trees

In a Decision Tree, the model gives its answer by starting at the root node of a binary tree and then moving from each node to one of its children by answering a yes/no question at each leaf. A decision tree can be thought of as approximating an arbitrary function through a piecewise combination of constant functions.

1.1.2 Neural Networks

In a Neural Network, a given neuron labeled i consists of a non-linear function $f_i(x, \theta_i)$, such that if the output of a neuron i is connected to a neuron j, then the output $\hat{y_{n,j}}$ of the neuron j is

$$\hat{y_{n,j}} = f_j(f_i(x, \theta_i), \theta_j)$$

1.2 Common Loss Functions

A **Regression Problem** is a problem where the output is an arbitrary real number. Common loss functions for regression include:

- Least Squares:

$$\mathcal{L}(y_n, \hat{y}_n) = (y_n - \hat{y}_n)^2$$

  Least Square Loss is an approximation of the maximum likelihood estimator of a linear model with normally distributed error (with a mean of 0).

- L1 Loss:

$$\mathcal{L}(y_n, \hat{y}_n) = |y_n - \hat{y}_n|$$

For **Binary Classification Problems**, where y_n can only take on two possible values, more specialized loss functions are used:

- Logistic Loss, $y_n \in \{0, 1\}$:

$$\mathcal{L}(y_n, \hat{y}_n) = -y_n \log(\hat{y}_n) - (1 - y) \log(1 - \hat{y}_n)$$

- Hinge Loss, $y_n \in \{-1, 1\}$:

$$\mathcal{L}(y_n, \hat{y}_n) = \max(0, 1 - y_n \hat{y}_n)$$

A **Softmax** is a type of function used to express **Non-Binary Classification Problems**, where targets $y_n \in 1, \ldots, C$, as a set of binary classification targets:

$$y_{n,k} = \begin{cases} 1 & y_n = k \\ 0 & y_n \neq k \end{cases}$$

The final result is then obtained by re-expressing the results as probabilities among classes as follows:

$$\hat{y}_{n,k} = \frac{e^{f(x_n, \theta_k)}}{\sum_{i=1}^{C} e^{f(x_n, \theta_i)}}$$

The exponential function is used to avoid negative probabilities. A common loss function for Softmax problems is **Logloss**:

$$\mathcal{L}(y_n, \hat{y}_n) = -\sum_{i=i}^{C} y_{n,c} \log(\hat{y}_{n,c})$$

1.3 Overfitting and Underfitting

Two common issues every model needs to deal with are **Underfitting**, where the bias of a model is too high, leading to a model that is unable to capture the data's complexity, and **Overfitting**, where the variance of a model is too high, meaning that the model has captured noise in the training data and is unable to generalize to pieces of data not included in the training data. Very simple models like linear models tend to be prone to underfitting, while most modern models that are more complex tend to be more prone to overfitting. Decision Trees in particular are very prone to overfitting.

1.3.1 Regularization

Regularization is the process of fighting overfitting by reducing the the size of the parameters of a model. This is expressed by adding a **Regularization Function** as a penalty term to the underlying optimization problem of minimizing the empirical risk:

$$\underset{\theta}{\text{argmin}} \sum_{n=1}^{N} \mathcal{L}(y_n, \hat{y}_n) + \alpha\Omega(\theta)$$

Just as there are many different possible loss functions, there are also many different possible regularization functions. Using a regularization always comes with the tradeoff of increasing model bias (but is done in the vast majority of models, since with complex modern models, overfitting tends to be a more common issue than underfitting).

1.4 ML Design Cycle

1. Pre-processing
2. Feature extraction / Feature encoding
3. Feature selection
4. Machine learning
5. Evaluation / Model Selection
6. Post-processing

1.5 Non-Parametric Models

1.5.1 Nearest Neighbors

Predict target y_i by averaging over the targets of the k nearest points $x_1, \ldots, x_k$ to the point x_i. For a continuus target the aggregation is simply the mean:

$$f(x_i) = \frac{1}{k} \sum_{y_j \in Y_{near}}^{k} y_j \tag{1.1}$$

For classification, we get our probability of the target being a certain class by taking the number of points belonging to each class and dividing by the total number of points we sampled:

$$f_c(x) = \frac{|y_j y_j \in Y_{near} \wedge y_j = c|}{k} \tag{1.2}$$

1.5.2 Naive Bayes

Bayes Rule:

$$P(y|x) = P(y)\frac{P(x|y)}{P(x)} \tag{1.3}$$

Using the law of total probability this can be extended to:

$$P(y|x) = P(y)\frac{P(x|y)}{\sum_{y' \in Y} P(x|y')P(y')} \tag{1.4}$$

Note that all of this only formally works if all features are conditionally independent.

Chapter 2

Linear Methods

2.1 Linear Regression

The following problem is also known as the regression problem:

- Dataset of N instances (x_i, y_i) (x_i is generally a vector)
- Find a model $\hat{y}$ that minimizes a loss function $\mathcal{L}$:

$$\underset{w \in W}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}(y_i f(x_i; w))$$

- Very simple and basic model: **Linear Regression**

$$f(x_i; w) = w_0 + x_i w^T$$

  w_0 is a fixed offset and thus often called the **bias**, while w is the **weight vector** or **parameter vector**.

Linear regression is more powerful than it looks - for example, $f(x) = (x + 1)^2$ can be expressed as a linear function by increasing the dimensionality and writing it as

$$f(a, b) = a + 2b + 1$$

(such that $b = x$ and $a = x^2$). Formally, we can take any set of arbitrary **basis functions** $\phi_j(x_i)$ and arrive at a linear model

$$f(x_i; w) = w_0 + \sum_{j=1}^{M} w_j \phi_j(x_i)$$

Linear models are therefore linear in the weights w_i, but they don't have to be linear in the inputs x_i.

The **residual error** of a linear model is given by:

$$y_i - f(x_i; w)$$

For the loss function of a linear model, we just sum the losses of the individual predictions. Typical choices for the loss function are L2 loss $(y - \hat{y})^2$ and L1 loss $|y - \hat{y}|$

A closed-form solution for linear regression using L2 loss exists under the following assumptions:

- The expected value of the residuals is 0: $\forall i : E(\varepsilon_i) = 0$
- Residuals are uncorrelated and share the same variance: $\forall i : Var(\varepsilon_i) = \sigma_2$
- Residuals follow a normal distribution: $\forall i : \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$

If these hold, then the closed-form solution is:

$$w = (X^T X)^{-1} X^T y \tag{2.1}$$

Where X is the input matrix, i.e.

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

2.2 Linear Classification

Logistic regression is a way of using linear regression to solve classification problems, by bounding the output of a linear model to $0 \leq h_w(x) \leq 1$ by taking the sigmoid of our output. The output is then interpreted as the probability that $y = 1$ (formally, $h_w(x) = P(y = 1|x; w)$). It's kind of fucked up that it's called "logistic regression" instead of "logistic classification".

$$h_w(x) = \sigma(w^T x) \tag{2.2}$$

Alternative idea: predict $y = 1$ if $wx^T \geq 0$ and $y = 0$ otherwise (see support vector machines in a later chapter.)

We use **Logistic Loss**, as already seen in the introduction:

$$\mathcal{L}(y_i, h_w(x_i)) = \begin{cases} -log(1 - h_w(x_i)) & y_i = 0 \\ -log(h_w(x_i)) & y_i = 1 \end{cases} \tag{2.3}$$

Note: I wrote $h_w(x_i)$ here because that's what the lecture does, but there's no real reason not to just write $\hat{y}(x_i)$ instead.

2.2.1 Gradient Descent

:

- For linear regression using L2 loss:

$$\mathcal{L}(w) = \sum_{i=1}^{N} (y_i - f(x_i; w))^2 \tag{2.4}$$

$$\implies \frac{\partial J(w)}{\partial w} = \sum_{i=1}^{N} -2(y_i - f(x_i; w))x_i \tag{2.5}$$

- For logistic regression using logistic loss:

$$\mathcal{L}(w) = \sum_{i=1}^{N} -y_i log(h_w(x_i)) - (1 - y_i) log(1 - h_w(x_i)) \tag{2.6}$$

$$\implies \frac{\partial J(w)}{\partial w} = \sum_{i=1}^{N} -(y_i - h_w(x_i))x_i \tag{2.7}$$

Chapter 3

Principles of Regularization

Reminder: for $x \in \mathbb{R}$, the polynomial prediction model of degree M is simply:

$$\hat{y} = f(x;\theta) = \sum_{j=0}^{M} \theta_j x^j \tag{3.1}$$

Optimal values $\theta 1^*$ are learned by minimizing the empirical risk (which is just the L2 loss? Why is a new term needed here?)

$$\theta^* := \underset{\theta}{\text{argmin}} \sum_{i=1}^{N} (f(x_i;\theta) - y_i)^2 \tag{3.2}$$

Small M leads to underfitting, large M leads to overfitting.

3.1 Bias-Variance Tradeoff

The expected target of y given x is written

$$\overline{y}(x) := E_{y|x}(x) = \int_y y p(y|x) \tag{3.3}$$

The expected prediction model $\overline{f}(x)$ over sample datasets is the model we expect to obtain given that we train randomly sampled data D that was sampled following a distribution $\mathcal{P}$:

$$\overline{f}(x) := E_{D \sim P^N}[\hat{f}(x;D)] \tag{3.4}$$

The expected value of the loss function is also called the **expected test error**:

$$E_{(x,y)\sim P}[(\hat{f}(x;D) - y)^2] \tag{3.5}$$

It turns out that the expected test error can be directly decomposed into a simple function of the variance, the bias squared, and the expected noise:

$$E_{x,y,D}[(\hat{f}(x;D) - y)^2] = E_{x,D}[(\hat{f}(x;D) - \overline{f}(x;D))^2] \tag{3.6}$$

$$+ E_{x,D}[(\overline{f}(x;D) - \overline{y}(x))^2] \tag{3.7}$$

$$+ E_{x,y}[(\overline{y}(x) - y)^2] \tag{3.8}$$

Recall the definitions of noise, bias and variance:

- The **variance** of a random variable X is defined as:

$$\sigma = E[(X - E[X])^2] = E[(f - E(f))^2] = E[f - \overline{f}] \tag{3.9}$$

- The **bias** of a predictor X that predicts y is defined as:

$$b = E[E[X] - y] = E[\overline{f} - \overline{y}] \tag{3.10}$$

  (Note: Why $\overline{y}$ instead of y? I assume both work because its wrapped up in an expected value anyways?)

- The **noise** is the difference in our data from the expected value of the data, i.e:

$$n = (\overline{y} - y) \tag{3.11}$$

3.2 Regularization

An increase in model complexity will always lead to an increase in variance, but a decrease in bias. In practice, this means that any method of decreasing a models variance will increase the bias and vice versa. Since sensible models are more likely to overfit, Regularization is very commonly used, since it decreases variance. Common regularization functions are:

$$L1 : \Omega(\theta) = \frac{1}{|\theta|} \sum_{k=1}^{|\theta|} |\theta_k| \tag{3.12}$$

$$L2 : \Omega(\theta) = \frac{1}{|\theta|} \sum_{k=1}^{|\theta|} \theta_k^2 \tag{3.13}$$

Chapter 4

Support Vector Machines

Motivation: Classify data by seperating it using a linear decision boundary. We want to find the boundary that maximizes the margin between the line and the current data.

Seperation is generalized to arbitrary dimensions by hyperplanes:

$$H(x) : w^T x + w_0 = 0 \tag{4.1}$$

Classification of x_i then comes down to simply taking $sign(H(x_i)$.

As a loss function, we can take a variant of the hinge loss:

$$\sum_{i=1, f(x_i;w)=-1}^{N} -y_i(w^T x_i + w_0) \tag{4.2}$$

Since this will be negative if y_i and $sign(w^T x_i + w_0)$ don't agree. The size of the margin scales inversely with the size of the weights.

If the data cant be linearly seperate, we introduce slack margins, i.e. we enforce a margin greater than $1 - \xi_i$ and then minimize ξ_i.

Maximizing the margin is equivalent to minimizing the squared weights $w^T w$. (I'm skipping a significant amount of stuff on the lagrange dual form of the SVM problem at this point due to time constraints)

The set of points whose removal would allow the creation of a different hyperplane with a larger margin is known as the set of **support vectors**. The dual formulation allows us to do inference on a new point by calculating its dot product with the support vectors, instead of having to evaluate the entire hyperplane. To be honest, I'm not entirely sure how this is more efficient.

We can still use SVMs on "strongly" non-linearly seperable data, where slack margins aren't enough, by applying a nonlinear mapping $\phi(x)$ to the data to make it linearly seperable in a higher dimensional space. We can efficiently do the necessary calculations using the **Kernel Trick**, where we replace instances of $\phi(x_i)^T \phi(x_j)$ in relevant equations with a kernel function $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ that doesn't need to explicitly calculate the mapping ϕ.

Chapter 5

Decision Trees

- Equivalent to step function with steps in the middle between data points
- Depth D is a property of individual nodes, the maximum depth of any node is known as the **height** of the tree.
- Trees partition feature space into regions R_i and then assign a constant prediction to each region.
- The constant is obtained by aggregating the data points in that region, i.e. taking the mean:

$$\hat{y}_{R_i}(x) = \frac{1}{|R_i|} \sum_{(x,y)\in R_i} y \tag{5.1}$$

- In this lecture, we only use **binary splits**.

5.1 Choosing Splits

We assign to each set D an **Impurity** $H(D)$. We then define the **information gain** $I(x \leq v)$ of the split $x \leq v$ as the difference in impurity before and after the split:

$$I(x \leq v) = |D| \cdot H(Y) - |D_L| \cdot H(Y_L) - |D_R| \cdot H(Y_R) \tag{5.2}$$

Where:

- D is the original set of data
- $D_L = \{(x, y) \in D \mid y \leq v\}$
- $D_R = \{(x, y) \in D \mid y > v\}$

Choosing correct splits then naturally comes down to maximizing the information gain.

5.1.1 For regression

In regression, we want to minimize the variance:

$$|D| \cdot H(D) = \sum_{(x,y)\in D} (y - \hat{y}_L(x))^2 \tag{5.3}$$

5.1.2 For classification

In classification, where $y \in \{1, \ldots, k\}$, we want to minimize the **entropy** of the data in the chosen split:

$$H(D) = -\sum_{k=1}^{K} p(y = k) log_2(p(y = k)) \tag{5.4}$$

5.2 Pros and Cons of Decision Trees

Pros:

- Flexible with many tunable hyperparameters (splitting criterion, leaf model, split type, depth etc.)
- Easily interpretable
- Easily deals with different types of inputs
- Handles features of different priorities well
- Easily scalable for large datasets

Cons:

- Tends to overfit
- Fully deterministic, making them bad at bagging (see later section).

Chapter 6

Neural Networks

Common activation functions and their gradients:

- ReLu:

$$\text{ReLu}(x) = x \cdot \mathbf{1}(x \geq 0) \tag{6.1}$$

$$\frac{\partial ReLu}{\partial x} = \mathbf{1}(x \geq 0) \tag{6.2}$$

$$\tag{6.3}$$

- Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{6.4}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)(1 - \sigma(x)) \tag{6.5}$$

- Hyperbolic Tan:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6.6}$$

$$\frac{\partial \tanh}{\partial x} = 1 - \tanh(x)^2 \tag{6.7}$$

- Softmax:

$$\sigma_{SM}(x_i) = \frac{e^{x_i}}{\sum_{n=1}^{N} e^{x_n}} \tag{6.8}$$

$$\frac{\partial \sigma_{SM}}{\partial x} = oof,\ not\ doing\ this\ one \tag{6.9}$$

Chapter 7

Ensembles

7.1 Bagging

Idea: If we have many models with low biases but a high variance, and they have independent outputs, then by the law of large numbers we can get closer to the correct value by averaging the prediction of these different models.

Since we only have one big dataset D, our models aren't actually independent in practice. We try to get close by creating many different smaller datasets D_i from D by **sampling with replacement**, i.e. we allow a dataset to draw the same element multiple times, and then training our models on those datasets. This is known as **Bootstrap Aggregation**.

The output of our bagged model is then simply the mean of the ouputs of the individual models:

$$\hat{f}(x) = \frac{1}{K}\sum_{k=1}^{K}\hat{f}^k(x) \tag{7.1}$$

The models still have to be **uncorrelated** to each other, i.e. their covariances have to be 0, otherwise this process doesn't work. To be precise, the expected squared error of an ensemble of K models with a shared variance v and a covariamce c is:

$$\varepsilon = E\left[\left(\frac{1}{K}\sum_{k=1}^{k}\varepsilon_k\right)^2\right] = \frac{1}{K^2}E\left[\left(\sum_{k=1}^{k}\varepsilon_k^2 + \sum_{k\neq l}\varepsilon_k\varepsilon_l\right)\right] = \frac{1}{K}v + \frac{K-1}{K}c \tag{7.2}$$

Trees are low bias, but often end up being correlated. We can decrease their correlation by randomizing which feature we decide to split on at each point. Bagged Trees that split on random features are known as **Random Forests**. Generally, random forests don't split on entirely random features, instead they pick m features randomly out of the full set of M features and then pick the best split among the chosen features.

Bagged ensembles provide an estimate of the test error via the out-of-bag training error, i.e. the output we get from the bagged model if we only take the output $f_k(x_i)$ of each submodel k if x_i wasn't part of its training data D_k:

$$\varepsilon_{OOB} = \frac{1}{|D|}\sum_{i=1}^{|D|}\mathcal{L}\left(y_i, \frac{1}{S(x_i)}\sum_{k\in S(x_i)}f_k(x_i)\right) \tag{7.3}$$

$$S(X) = \{k \in \{1,\dots,K\} \mid (x,y) \notin D_k\} \tag{7.4}$$

The variance of an ensemble is:

$$\sigma(x) = \sqrt{\frac{1}{K-1}\sum_{k=1}^{K}\left(\hat{f}_k(x) - \hat{f}(x)\right)^2} \tag{7.5}$$

7.2 Boosting

Boosting creates a low-bias model out of an esemble of high-bias models, by training each model to correct the error of the last model. The final ensemble F_K is a weighted sum of a set of models f_k:

$$F_K(x) = \sum_{k=1}^{K} \alpha f_k(x) \tag{7.6}$$

We train each model sequentially while keeping the parameters of past models fixed:

$$\theta_{k+1} = \underset{\theta}{\text{argmin}} \sum_{n=1}^{N} \mathcal{L}(y_n, F_k(x_n; \theta_{fixed}) + \alpha f(x_n; \theta)) \tag{7.7}$$

7.2.1 Gradient Boosting

1. Initialize model with a constant value:

$$F_0(x) = \underset{\alpha_0}{\text{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, \alpha_0) \tag{7.8}$$

   In the lecture, this calculation was skipped and 0 was always used instead.

2. Let M be an arbitrary number of maximum iterations. For each $m = 1, \ldots, M$:
   (a) Compute *pseudo-residuals*, i.e. the gradient of the loss function by the ensemble output:

$$z_m = \frac{\partial \mathcal{L}(y_n, F_k(x_n))}{\partial F_k(x_n)} \tag{7.9}$$

   (b) Train model f_m on the set $\{(x_i, r_i)\}_{i=1}^n$
   (c) Compute the best possible α_n, i.e. $\alpha_m = \underset{\alpha}{\text{argmin}} \sum_{i=1}^{n} \mathcal{L}(y_i, F_{m-1} + \alpha f_m(x_i))$
   (d) Update the ensemble model: $F_k(x) = F_{m-1}(x) + \alpha_m f_m(x)$
3. Output the finished ensemble model F_M.

7.2.2 AdaBoost

AdaBoost is a special case of Gradient Boosting with a particular loss function that results in weighting points misclassified by previous models higher than correctly classified ones.

Specifically, AdaBoost uses an approximation of **instance-penalty loss**:

$$\mathcal{L}(y_i, f(x_i)) = e^{-y_i f(x_i)} \tag{7.10}$$

$$\approx \frac{\sum_{i=1}^{N} w_i^{(k)} \cdot 1(y_i \neq f(x_i))}{\sum_{i=1}^{N} w_i^{(k)}} \tag{7.11}$$

$$:= \varepsilon \tag{7.12}$$

In natural language, this just means dividing the sum of the weights of incorrectly classified data points by the total sum of the weights.

The model weights are then given by:

$$\alpha_k = ln\left(\frac{1-\varepsilon_k}{\varepsilon_k}\right) \tag{7.13}$$

The feature weights w_i are initialized as $\frac{1}{N}$, i.e. all features are weighted equally, and then updated according to the following rule:

$$w_i^{(k+1)} = w_i^{(k)} e^{(\alpha_k \cdot 1(y_i \neq f(x_i)))} \tag{7.14}$$

This means that the weights of correctly classified features stay the same, while the weights of incorrectly classified features increase by a factor of $\frac{1-\varepsilon_k}{\varepsilon_k}$.

7.3 Gradient-Boosted Decision Trees

Tree overfitting happens if there are too many leaves (coresponding to too many steps in the final function) or if the leaves' output values are too high (correspnding to large jumps in the final function). A common regularization function for trees is thus:

$$\Omega(f) = \gamma T + \frac{\lambda}{2}\sum_{j=1}^{T} w_j^2 \tag{7.15}$$

Where T denotes the number of leaves.

For Gradient Boosted Trees, the function we want to minimize at each step is:

$$J(y_i, f^{(k)}) = \left(\sum_{n=1}^{N} \mathcal{L}_n^{k-1} G_n f^{(k)}(x_n) + \frac{1}{2} H_n (f^{(k)}(x_n))^2\right) + \Omega(f^{(k)}) \tag{7.16}$$

i.e. we approximate $\mathcal{L}_n^{(k)}$ via a Taylor expansion:

$$\mathcal{L}_n^{(k)} \approx \mathcal{L}_n^{(k-1)} + G_n^{(k)} f^k(x_n) + \frac{1}{2} H_n (f^{(k)}(x_n))^2 \tag{7.17}$$

Minimizing this loss function corresponds to minimizing the following Objective Function O_j for each Leaf j:

$$O_j = -\frac{1}{2}\frac{\left(\sum_{n=1}^{N} G_n\right)^2}{\left(\lambda + \sum_{n=1}^{N} H_n\right)} + \gamma \tag{7.18}$$

Where G_n is the gradient of the loss function over the model output and H_n is the Hessian matrix, i.e. the multidimensional equivalent of the second derivative.

If we let I_J denotes the indices of all data points in the region R_J belonging to leaf J, the optimal leaf weights are then given by:

$$w^* = -\frac{\sum_{n=1}^{N} G_n}{\lambda + \sum_{n=1}^{N} H_n} \tag{7.19}$$

The gain of splitting a leaf is simply:

$$Gain_j = O_j - O_L - O_R \tag{7.20}$$

just like the information gain of regular trees.

We can decide to stop splitting further if $Gain_j \leq 0$. This means that γ directly controls the minimum gain needed for the model to perform a split.

The most common loss function for classification is logistic loss, so here are the gradient and hessian pre-calculated:

$$\mathcal{L}_n = -y_n \ln(\sigma(\hat{y}_n)) - (1 - y_n)\ln(1 - \sigma(\hat{y}_n)) \tag{7.21}$$

$$G_n = \frac{\partial \mathcal{L}_n}{\partial \hat{y}_n} = \sigma(\hat{y}_n) - y_n \tag{7.22}$$

$$H_n = \frac{\partial^2 \mathcal{L}_n}{\partial^2 \hat{y}_n} = \sigma(\hat{y}_n)(1 - \sigma(\hat{y}_n)) \tag{7.23}$$

Remember that $\hat{y}_n^{(k)} = \hat{y}_n^{(k-1)} + f^{(k)}(x_n)$. (Or $\hat{y}_n^{(k)} = \hat{y}_n^{(k-1)} + \alpha f^{(k)}(x_n)$ if we want to do a weighted sum). In this lecture, we assume $\hat{y}^{(0)} = 0$.

GBDTs have very low biases while also retaining a relatively low variance. They are also very fast to train, requiring a runtime of $O(kMN\log(T))$, where k is the number of trees in the ensemble, M is the number of features and N is the number of data points.

Chapter 8

Hyperparameter Optimization

A function $f(\lambda, D)$ is called a **response function** if its output is the loss of a given model that was trained on the dataset D using the hyperparameters λ. It can also be a function of the hyperparameters and dataset that we want to maximize, such as classification accuracy.

8.1 Bayesian Optimization

Straightforwardly optimizing f is costly enough to generally be infeasible, since it would involve retraining the entire model for every new set of hyperparameters. Instead, we trade a surrogate model Ψ to predict the output of f based on known values. We then obtain new hyperparameters by picking the ones that maximize/minimize Ψ, training a model using these new hyperparameters, and retraining Ψ on the augmented dataset that includes the performance of the model using the new hyperparameters.

Ψ is often called a **Gaussian Process Regressor**, and the process of optimizing hyperparameters by getting them from the surrogate model is also known as **Bayesian Optimization** or **Sequantial Model Based Optimization (SMBO)**.

If we know the mean μ and the variance σ of the surrogate model, we can also pick our hyperparameters based on the **lower confidence bound**:

$$LCB(\lambda) = -\mu + \beta\sigma \tag{8.1}$$

or, for classification, where we want to maximize μ, based on the **upper confidence bound**:

$$UCB(\lambda) = \mu + \beta\sigma \tag{8.2}$$

8.2 Gray-Box Search

Bayesian Optimization is an example of black box optimization, where we don't know the structure of the problem. **Gray-box optimization** does have knowledge about the structure of the problem. This means that we can:

- We can do k**-fold cross-validation** to discard bad hyperparameters early:
  - Split data D into k subsets
  - For every subset D_i, use $D \setminus D_i$ to train a model and D_i as the test data
  - Retain evaluation score of each model
- If we use iterative ML algorithms, we can simply stop poor runs early.
  - Hyperband: Start with a lot of models with random hyperparameters, then discard low-performing half / two thirds / a different fractions of the models each time a certain number of learning epochs is up.

Hyperband is faster than Bayesian Optimization, but Bayesian Optimization generally produces better results if given enough time (but "enough time" may be a factor on the order of 10^2)

Chapter 9

Advanced Regularization Or Whatever

9.1 Dropout

During training of a neural network, at each step, train as if a random set of node outputs were actually 0. We then take the output of multiple versions of a neural network with different dropped weights, making dropout a form of bagging.

9.2 Residual Networks

A **residual network** is a type of neural network whose architecture includes connections that "skip" a certain number of layers, i.e. a neuron h_n may be connected to a neuron h_{n+c} for $c \geq 2$.

Residual networks were the key advancement needed to start training ever deeper models, since they fix issues deep neural networks run into, like vanishing/exploding gradients during back propagation.

Shake-Shake is a method of regularization for residual networks that randomly assigns weights to the output of each "branch" during training (?)

9.3 Data Augmentation

Data Augmentation is the process of artificially creating additional data before training. It is primarily used in Image Processing Application, where it is easy to create realistic, sensible data through changes to images such as rotation, translation etc. Additional methods of data augmentation for images include:

- Cutout, where a random patch of the image is replaced with black pixels
- Mixup, where two images are linearly interpolated (i.e. made transparent and then layered over each other)
- Cutmix, where a random patch of an image is replaced with a patch from a different image.

These can help a model deal with images in unusual circumstances, such as recognizing an object that is partially covered by something else.

9.4 Cocktails

A combination of multiple regularizers is known as a **Cocktail**. Using cocktails is generally better than using an individual regularizer. Finding the best cocktail for a given problem is an active area of research.

Chapter 10

Error Measures

If a loss function we wish to use for a given problem is non-differentiable, we can use another arbitrary function as a **Proxy Loss** as long as we know that the minima of the proxy loss are at the same points as the minima of the original loss.

10.1 Classification

Missclassification Rate:

$$MCR(y, \hat{y}) = \frac{FP + FN}{TP + TN + FP + FN} \tag{10.1}$$

An effective proxy loss of MCR is the cross-entropy (presented here in the general form used for multiclass-classification, where $y_{n,c}$ is 1 if $y_n = c$ and 0 otherwise:

$$-\sum_{n=1}^{N}\sum_{c=1}^{C} y_{n,c} \log(\hat{y}_c(x_n; \theta)) \tag{10.2}$$

True Positive Rate or Recall:

$$TPR(y, \hat{y}) = \frac{TP}{TP + FN} \tag{10.3}$$

False Positive Rate ("Ratio of false alarms")

$$FPR(y, \hat{y}) = \frac{FP}{TN + FP} \tag{10.4}$$

The curve we get by plotting TPR on the y-axis against FPR on the x-axis for different activation thresholds is known as the *ROC* curve ("Receiver Operating Characteristic"). The Area under the *ROC* curve ("*AUC*") tells us the likelihood that, for a given pair of $y_i = 0$ and $y_j = 1$, our model predicts a higher value to $\hat{y}_j$ than $\hat{y}_i$.

Precision (Percentage of predicted positives that are correct):

$$PRC = \frac{TP}{TP + FP} \tag{10.5}$$

If we plot precision and recall against each other as we did for the *ROC*-curve, the result is known as a PR(precision-recall)-curve. Using a PR curve makes sense in imbalanced datasets where we care more about the positive class.

F1 (Harmonic mean between Precision and Recall):

$$F1 = 2\frac{PRC \cdot REC}{PRC + REC} \tag{10.6}$$

F1 is hard to optimize for because it is non-differentiable and non-decomposable. We have to train models using variants of regular classification loss functions (logistic loss, hinge loss etc.) as proxies. It also helps to oversample the positive or negative class, such that the training minibatches have approximately 50% positive and 50% negative instances.

10.2 Regression

Scale-dependent errors:

- Mean Absolute Error:

$$MAE = \frac{1}{N}\sum_{n=1}^{N}|y_n - \hat{y}_n| \tag{10.7}$$

- Root Mean Square Error:

$$RMSE = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2} \tag{10.8}$$

Scale independent errors: Percentage errors, such as Mean Absolute Percentage Error:

$$MAPE = \frac{1}{n}\sum_{n=1}^{N}\left|\frac{y_n - \hat{y}_n}{y_n}\right| \tag{10.9}$$

An issue with percentage errors is that they are undefined when $y_n = 0$ and produce extreme values when $y_n \approx 0$. One fix is to use scaled errors such as **Mean Absolute Scaled Error**:

$$MASE = \frac{1}{N}\sum_{n=1}^{N}\left|\frac{y_n - \hat{y}_n}{\frac{1}{N-1}\sum_{n'=2}^{N}|y_{n'} - y_{n'-1}|}\right| \tag{10.10}$$

10.3 Ranking

DCG (**Discounted Cumulative Gain**):

$$DCG = \sum_{i=1}^{K}\frac{2^{l_i} - 1}{\log(i+1)} \tag{10.11}$$

Where l_i is the "ground truth relevance" of the data point that our model assigns i-th most relevant. The DCG of the ground truth ranking is known as the $IDCG$, and a good error measure for our ranking model is then given by the normalized DCG:

$$NDCG = \frac{DCG}{IDCG} \tag{10.12}$$

Pairwise Ranking Loss (Logistic Loss adapted for ranking problems):

$$\mathcal{L}_{i,j} = -P_{i,j}\log(\hat{P}_{i,j}) - (1 - P_{i,j})\log(1 - \hat{P}_{i,j}) \tag{10.13}$$

Where P_{ij} is given by:

$$P_{ij} = \begin{cases} 1 & l_i > l_j \\ 0.5 & l_i = l_j \\ 0 & l_i < l_j \end{cases} \tag{10.14}$$

And $\hat{P}_{i,j}$ is an estimation of the probability that the pair i, j is correctly ranked:

$$\hat{P}_{i,j} = \sigma(s_i - s_j) \tag{10.15}$$

Where s_i are the predicted relevances.

Alternatively, we may reexpress P_{ij} in terms of the following term S_{ij}:

$$S_{ij} = \begin{cases} 1 & l_i > l_j \\ 0 & l_i = l_j \\ -1 & l_i < l_j \end{cases} \tag{10.16}$$

Sadly, Pairwise Loss is provably non-optimal for maximizing $NDCG$.