CS337: Artificial Intelligence and Machine Learning

Krishna Agaram

Autumn 2023

Contents

Chapter 1

Unsupervised Learning: Clustering

Consider a set of points $X = \{x_1, \ldots, x_n\}$ in $\mathbb{R}^d$. We want to find similar (defined by close-by according to some metric on $\mathbb{R}^d$) points and group them together (assign them the same class). This is called *clustering* (into classes).

Definition 1.1 (Clustering). Let $X = \{x_1, \ldots, x_n\}$ be a set of points in $\mathbb{R}^d$. A *clustering* of X is a partition of X into k subsets $C_1, \ldots, C_K$ such that - in some well-defined sense - points in the same subset are similar and points in different subsets are dissimilar.

Of what utility is clustering? Consider the following applications:

1. *Market segmentation*: Given a set of customers, group them into clusters based on their purchasing behavior.
2. *Social network analysis*: Given a set of users, group them into clusters based on their social interactions.
3. *Image segmentation*: Given an image, group its pixels into clusters based on their color.
4. *Astronomical data analysis*: Given a set of stars, group them into clusters based on their brightness.

1.1 k-means Clustering

A simple algorithm for clustering into k clusters. The intuition is the following: every datapoint is close to the centre of the mean of its assigned cluster. This means that once we fix the *cluster centres*, it is very easy to complete the cluster (assign each point to the cluster whose centre is closest to it).

Here is the key point: a good heuristic for the cluster centres is to take the mean of the points in the cluster. Indeed, the mean minimizes the sum of squared distances to all the points in the cluster (does that hint at what a good loss could be?).

So what we could do is: pickup random points as the cluster centres, and repeat the following two steps until convergence:

1. Assign each point to the cluster whose centre is closest to it.
2. Update each cluster centre to be the mean of the points in the cluster.

This alternates between two steps: *assignment* (computing the cluster corresponding to a set of clusters) and *update* (update the cluster centres based on the cluster). This is called the *K-means algorithm*. In a way, it is similar to Generalized Policy Iteration (GPI) in Reinforcement Learning - where we alternate between *policy evaluation* (computing the values corresponding to a policy) and *policy improvement* (update the policy based on the values). Indeed, all these algorithms are instances of *Expectation Maximization* (EM).

Let us make this precise. Let $X = \{x_1, \ldots, x_n\}$ be a set of points in $\mathbb{R}^d$, a clustering $C_1, \ldots, C_k$ and a set of cluster centres $\mu_1, \ldots, \mu_k$. Define $C_j^i := \mathbb{1}_{[x_i \in C_j]}$.

Goal: find $\mu_1, \ldots, \mu_k$ such that the sum of squared distances of each point to its cluster centre is minimized. That is, we want to solve the following optimization problem:

$$\min_{\mu_1,\ldots,\mu_k} \min_{C_1,\ldots,C_k} \sum_{i\in[n]} \sum_{j\in[k]} C_j^i \|x_i - \mu_j\|^2 \tag{1.1}$$

yielding the loss:

$$\mathcal{L} = \sum_{i\in[n]} \sum_{j\in[k]} C_j^i \|x_i - \mu_j\|^2 = \sum_{j\in[k]} \sum_{x\in C_j} \|x - \mu_k\|^2$$

Suppose that we fix the clustering $C_1, \ldots, C_k$. Then the optimal $\mu_1, \ldots, \mu_k$ are given by setting the derivative $\frac{\partial \mathcal{L}}{\partial \mu_j} = 0$ for all $j \in [k]$. We have

$$0 = \frac{\partial \mathcal{L}}{\partial \mu_j} = 2 \sum_{x\in C_j} (x - \mu_k) \implies \mu_j = \frac{1}{|C_j|} \sum_{x\in C_j} x$$

that is, given a cluster, the optimal cluster centre is the mean of the points in the cluster - precisely the update step in the K-means algorithm.

Now suppose that we fix the cluster centres $\mu_1, \ldots, \mu_k$. Then the optimal $C_1, \ldots, C_k$ are given by assigning each point to the cluster whose centre is closest to it (We minimize the loss for fixed μ_i over each z_i - the label of each x_i).

Clearly, the minimum is achieved at

$$z_i = \arg\min_{j\in[k]} \|x_i - \mu_j\|^2$$

(so $C_j^i = \mathbb{1}_{\left[j = \arg\min_{j\in[k]} \|x_i - \mu_j\|^2\right]}$ for each i, j) Thus, given the cluster centres, the optimal clustering is the one that assigns each point to the closest cluster centre - precisely the assignment step in the k-means algorithm.

1.2 An algorithm for k-means

Algorithm 1 k-means

1: Initialize $\mu_1, \ldots, \mu_k$ randomly
2: **while** not converged **do**
3: Assign each point to the cluster whose centre is closest to it
4: Update each cluster centre to be the mean of the points in the cluster
5: **end while**

Another algorithm that makes explicit the two alternating steps is (the set of possible cluster centres is Θ, and of the possible assignments (clusters) is Φ):

Algorithm 2 k-means

Initialize $\theta \in \Theta$ randomly
flag = True
prev = -1
while True **do**
    if flag **then**
        $\phi = \arg\min_{\phi' \in \Phi} \mathcal{L}(\theta, \phi')$ // Assign each point to the cluster closest to it
    else
        $\theta = \arg\min_{\theta' \in \Theta} \mathcal{L}(\theta', \phi)$ // Update each cluster centre to be the mean of the cluster
    end if
    flag = **not** flag
    new = $\mathcal{L}(\theta, \phi)$
    if new == prev **then**
        break
    end if
end while

1.3 Convergence of k-means

Convergence:

Convergence in finitely many steps: there are only finitely many clusterings and cluster centres, so the algorithm must converge in finitely many steps.

k-means is doing gradient descent on the loss function $\mathcal{L}$!

(However, it is not guaranteed to converge to the global minimum. In fact, it is NP-hard (need to check a lot of the k^n possible clusterings) to find the global minimum of the loss.)

Chapter 2

Gaussian Mixtures: the generative model for k-means

Let us take a different route - the route of probabilistic modelling to solve the clustering problem. We will see that the k-means algorithm is doing MLE on a Gaussian Mixture Model (GMM).

Definition 2.1 (Generative model). A *generative model* is a model that tries to *learn* the distribution of the data. Typically, this is done by estimating either the joint $p(x, y)$ or the two distributions $p(x|y)$ and $p(y)$ or $p(x)$ and the conditional $p(y|x)$.

Definition 2.2 (Discriminative Model). A *discriminative model* is a model that tries to *learn* the conditional distribution $p(y|x)$ - it tries to learn to discriminate between the outputs y given an x.

2.1 Motivation

The probabilistic model for regression was a linear model with additive Gaussian noise - that is, the underlying generative model "looked linear" in a certain sense. What does the underlying generative model for clustering look like?

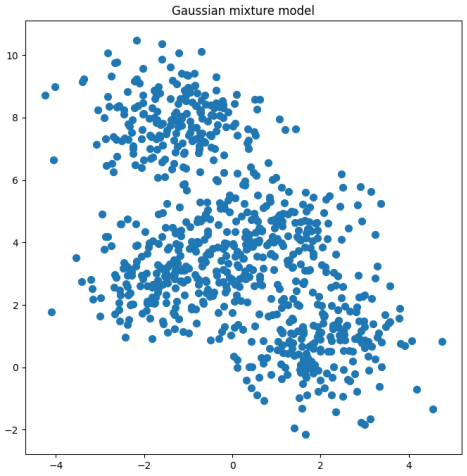
One could think about how data for clustering may have been generated by a friendly alien: The alien decides that there will be k clusters $\{1, \ldots, k\}$, and picks k points $\mu_1, \ldots, \mu_k$ in $\mathbb{R}^d$ as the cluster centres. He also picks a probability distribution π over the k clusters.

Then, when the alien wants to generate a datapoint, he simply picks a cluster j with probability π_j and then picks a point x from the distribution $\mathcal{N}(\mu_j, I)$. (For simplicity of exposition, we assume that the covariance of the Gaussian is the identity - it's not a bad assumption).

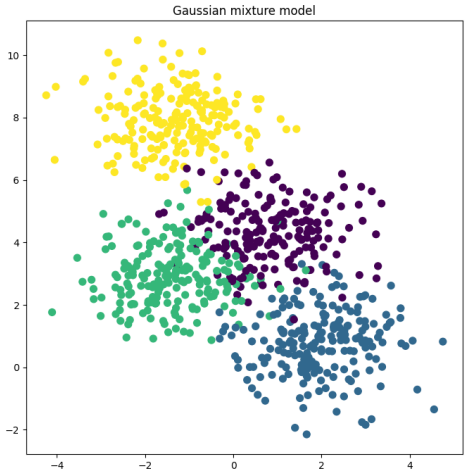
Think about it - it is a very reasonable modelling for a clustering generative model. The values π_j indicate the density of each cluster and the values μ_j indicate the centre of each cluster. The model is essentially saying that the clusters are isotropic Gaussian blobs about the μ_j with density π_j.

2.2 Formal definition

Suppose that the dataset $\mathcal{D} = \{x_1, \ldots, x_n\}$ was sampled from the parameterized generative model defined below with some parameters. We use MLE to estimate the parameters of the model (which, in the limit of infinite training data, will be exact - the ML estimate is consistent).



(a) Alien generates $n = 800$ datapoints with $k = 4$ clusters.



(b) The clusters that the alien used for each datapoint colored for clarity

Figure 2.1: Alien generates data from a GMM

- Pick a cluster $j \in [k]$ with probability π_j. We write the random variable corresponding to the cluster chosen as z.
- Pick x from the distribution $\mathcal{N}(\mu_j, I)$ (notice how this depends on j). For simplicity of exposition, we assume that the covariance of the Gaussian is the identity.

Essentially, we have parameterized the generative model by μ_j and π_j that define the joint by:

$$p(z = j) = \pi_j$$
$$p(x|z = j) = \mathcal{N}(\mu_j, I)$$

A model with the above parameterization is called a Gaussian Mixture Model (GMM). The joint distribution is given by:

$$p(x, z) = p(z)p(x|z) = \pi_j \mathcal{N}(\mu_j, I)$$

Okay. We'd like to estimate parameters of our model and find the one that matches the underlying GMM of the data. In the unsupervised learning situation, we are not given the labels z (which cluster each point belongs to): we need to estimate the parameters μ_j and π_j simply from the data $\mathcal{D} = \{x_1, \ldots, x_n\}$.

To this end, we marginalize out z to work with the probabilities

$$p(x) = \sum_{j \in [k]} p(x, z = j) = \sum_{j \in [k]} \pi_j \mathcal{N}(\mu_j, I)$$

The log-likelihood of the data is given by:

$$\mathcal{L}(\mu, \pi; \mathcal{D}) = \sum_{i \in [n]} \log p(x_i) = \sum_{i \in [n]} \log \sum_{j \in [k]} \pi_j \mathcal{N}(\mu_j, I)$$

A very messy expression. $p(x)$ is not a great term to put in a log. Let's take a small detour to think about how we can simplify the expression.

Suppose we were given labels $\mathcal{Z} = \{z_1, \ldots, z_n\}$ (generated by the underlying generative model) corresponding to each point x_i. Then the log-likelihood would be given by:

$$\begin{aligned}\mathcal{L}(\mu, \pi; \mathcal{D}, \mathcal{Z}) &= \sum_{i \in [n]} \log p(x_i, z_i | \mu, \pi) = \sum_{i \in [n]} \log \pi_{z_i} \mathcal{N}(\mu_{z_i}, I) \\ &= \sum_{i \in [n]} \log \pi_{z_i} + \sum_{i \in [n]} -\log \sqrt{2\pi} + \sum_{i \in [n]} -\frac{1}{2} \|x_i - \mu_{z_i}\|^2\end{aligned}$$

Let's write the likelihood in a more symmetric way:

$$\mathcal{L}(\mu, \pi; \mathcal{D}, \mathcal{Z}) = \sum_{j \in [k]} \sum_{i \in [n]} \mathbb{1}_{[z_i = j]} \log \pi_j + \sum_{i \in [n]} -\log \sqrt{2\pi} - \frac{1}{2} \sum_{j \in [k]} \sum_{i \in [n]} \mathbb{1}_{[z_i = j]} \|x_i - \mu_j\|^2$$

Maximizing the likelihood yields (for pik, cant differentiate, so use lagrange multipliers):

$$\mu_j = \frac{\sum_{i \in [n]} \mathbb{1}_{[z_i = j]} x_i}{\sum_{i \in [n]} \mathbb{1}_{[z_i = j]}} \tag{2.1}$$

$$\pi_j = \frac{\sum_{i \in [n]} \mathbb{1}_{[z_i = j]}}{n} \tag{2.2}$$

Notice how the ML estimate for the cluster centre is the mean of the points in the cluster - the update step in k-means. Since the labels are given to us, the clusters are fixed, so the k-means reduces to just the one step of updating the cluster centres. Thus, the k-means algorithm is just an ML estimate for a GMM model on the underlying data.

This is not a coincidence - just like in linear regression, the log of the Gaussian gives the squared distance which is precisely the loss we are trying to minimize in the k-means algorithm.

Back to reality. Of course, we are not given the labels. But we can still use the above expression to simplify the log-likelihood; interpreting the result as a sort of conditional over the values in $\mathcal{Z}$. Introduce the *responsibilities* or *occupancies* γ_j^i of each cluster j for each point i (the probability that point i belongs to cluster j):

$$\gamma_j^i = p(z = j | x_i) = \frac{p(x_i | z = j) p(z = j)}{p(x_i)} = \frac{\pi_j \mathcal{N}(\mu_j, I)}{\sum_{j' \in [k]} \pi_{j'} \mathcal{N}(\mu_{j'}, I)}$$

Remark. In case the covariance is not the identity - suppose it is a diagonal matrix for each cluster - that simply brings more parameters for the MLE, the rest works out similarly.

-> maximize wrt uk, pik assuming that he gamma variables are independent of these -> and then update the gamma variables

2.3 Expectation Maximization for GMMs

A general algorithm applying the same idea as above is called the Expectation Maximization (EM) algorithm. It is an iterative algorithm that alternates between two steps: the *expectation* step and the *maximization* step.

The algorithm finds a *lower bound* on the true log-likelihood of the data, and tries to maximize this lower bound. The lower bound is called the *evidence lower bound* (ELBO) and is given by:

$$\mathbb{E}_{z|x,\theta^{(t)}} [\log p(x, z|\theta)] - \mathbb{E}_{z|x,\theta^{(t)}} [\log q(z)]$$

Algorithm 3 Expectation Maximization

1: Initialize parameters θ
2: **while** not converged **do**
3: **E-step**: Compute the expected value of the log-likelihood wrt the conditional distribution of the latent variables given the data and the current estimate of the parameters:

$$\mathbb{E}_{z|x,\theta^{(t)}}\left[\log p(x, z|\theta)\right]$$

4: **M-step**: Update latent variables by maximizing the expected log-likelihood wrt parameters:

$$\theta^{(t+1)} = \arg\max_{\theta} \mathbb{E}_{z|x,\theta^{(t)}}\left[\log p(x, z|\theta)\right]$$

5: **end while**

2.3.1 The EM algorithm to fit GMMs to data

1. Initialize $\mu_k \leftarrow \mu_k^{\text{init}}$, $\pi_k \leftarrow \pi_k^{\text{init}}$ for $k \in [K]$.
2. **E-step**: Compute the responsibilities $\gamma_{k,i}$ for each point i and cluster j:

$$\gamma_{k,i} = p(z = k|x_i) = \frac{\pi_k \mathcal{N}(\mu_k, I)}{\sum_{k' \in [k]} \pi_{k'} \mathcal{N}(\mu_{k'}, I)}$$

3. **M-step**: Estimate μ_k, π_k using the maximum likelihood estimates that we found on keeping the $\gamma_{k,i}$ fixed:

$$\mu_k = \frac{\sum_{i \in [n]} \gamma_{k,i} x_i}{\sum_{i \in [n]} \gamma_{k,i}}$$
$$\pi_k = \frac{\sum_{i \in [n]} \gamma_{k,i}}{n}$$

4. Repeat steps 2 and 3 until convergence.

It is guaranteed to converge to a local optimum of the log-likelihood. The proof is a bit involved, but the intuition is that the log-likelihood is a concave function of the parameters, and the EM algorithm is a coordinate ascent algorithm on the log-likelihood.

2.4 The EM algorithm for k-means

Chapter 3

Dimensionality reduction

Most datapoints today have a very large number of features - that is, the vectors x are in a high-dimensional space. Images are an obvious example - a 100×100 color image has $30,000$ features. This is an obvious issue for many algorithms - iterative optimization algorithms like k-means are quite slow in high dimensions; in high dimensions, the data is very sparse and the curse of dimensionality makes it hard to find meaningful clusters.

Even for exact regression, the matrices that need to be multiplied become rather large, and since matrix multiplication is lower bounded by $\Omega(n^2)$, this becomes a problem.

In this chapter, we will look at two methods of dimensionality reduction: *Principal Component Analysis* (PCA) and *t-SNE*.

3.1 Principal Component Analysis

The most popular dimensionality reduction technique. It is a linear technique - that is, it tries to find a linear subspace of the original space that *captures most of the variance* in the data. It is also a *projection* technique - that is, it projects data onto this subspace.

3.1.1 Motivation

It is typically true that though the input space may have very large dimensionality, the data may be concentrated in a lower-dimensional subspace.

Thus, we aim to compute this subspace. We will do this by finding a set of orthonormal vectors $u_1, \ldots, u_k \in \mathbb{R}^d$ such that the projection of the data onto the subspace spanned by these vectors is as close in spirit as possible to the original data.

That is, we transform x to

$$z = U^T x \in \mathbb{R}^k$$

where $U = [u_1, \ldots, u_k] \in d \times k$ with $k \ll d$. Note that $U^T U$ is the Grammian of the vectors $u_1, \ldots, u_k$ and is the identity matrix since the vectors are orthonormal. Thus U is one-sided orthogonal.

The transformation is a projection:

$$z = \begin{bmatrix} u_1^T \\ \vdots \\ u_k^T \end{bmatrix} x = \begin{bmatrix} u_1^T x \\ \vdots \\ u_k^T x \end{bmatrix}$$

.

3.1.2 What is the subspace, exactly?

3.1.3 Objective

We wish to minimize the *reconstruction error*

$$\|x - Uz\|^2 = \left\| x - \sum_{i=1}^{k} u_i z_i \right\|^2$$

where $z_i = u_i^T x$.

A second objective of dimensionality reduction could be to maximize the variance of the projeected data:

$$\max_{U^T U = I_k} \sum_{i=1}^{k} \text{var}(z_i)$$

For PCA, both of these can be shown to be equivalent!

We have

$$\begin{aligned} x &= UU^T x + (I - UU^T)x \\ \implies \|x\|^2 &= \left\|UU^T x\right\|^2 + \left\|x - UU^T x\right\|^2 \\ \implies \mathbb{E}\left[\|x\|^2\right] &= \mathbb{E}\left[\left\|UU^T x\right\|^2\right] + \mathbb{E}\left[\left\|x - UU^T x\right\|^2\right] \end{aligned}$$

thus the variance of the projected data is maximized iff the reconstruction error is minimized.

3.1.4 Finding the subspace

We use objective 2. We want to maximize

$$\sum_{i=1}^{n} \left\|x_i^T u\right\|^2 = u^T \left(\sum_{i=1}^{n} x_i x_i^T \right) u = u^T S u.$$

It is standard to see that the maximum is achieved when U is the matrix of eigenvectors of S corresponding to the k largest eigenvalues. That's PCA.

Remark. The matrix S is called the *covariance matrix* of the data. It is a symmetric matrix, and its diagonal entries are the variances of the features. The off-diagonal entries are the covariances of the features.

Remark. Diagonalization to find eigenvectors can be very expensive (at least quadratic in the dimensionality). A faster algorithm uses the singular value decomposition: the top few eigenvectors of the covariance matrix are the same as the top few singular vectors of the data matrix.

Chapter 4

Ensemble methods

4.1 Aside: A few interesting papers

- How to avoid machine learning pitfalls: a guide for academic researchers
- Pen and Paper exercises in Machine Learning
-

4.2 A brief look at Autoencoders

Autoencoders are a class of neural networks that are trained to reconstruct their input. They are used for dimensionality reduction, denoising, and generative modeling. To avoid trivial solutions, we usually impose a bottleneck in the network, forcing it to learn a compressed representation of the input.

A linear autoencoder in the most general case is a pair of matrices $W_1 \in \mathbb{R}^{d\times k}$ and $W_2 \in \mathbb{R}^{k\times d}$. If $k \ll d$, we achieve dimensionality reduction. The function $f : \mathbb{R}^d \to \mathbb{R}^k$ defined by $f(x) = W_1 x$ is called the encoder, and the function $g : \mathbb{R}^k \to \mathbb{R}^d$ defined by $g(x) = W_2 x$ is called the decoder. The autoencoder is the composition $g \circ f$. The goal is to find W_1 and W_2 such that the autoencoder is a good approximation of the identity function, even with the bottleneck.

The loss is usually the mean squared error between the input and the output (called the reconstruction error):

$$\mathcal{L}(x, \tilde{x}) = \|x - \tilde{x}\|^2$$

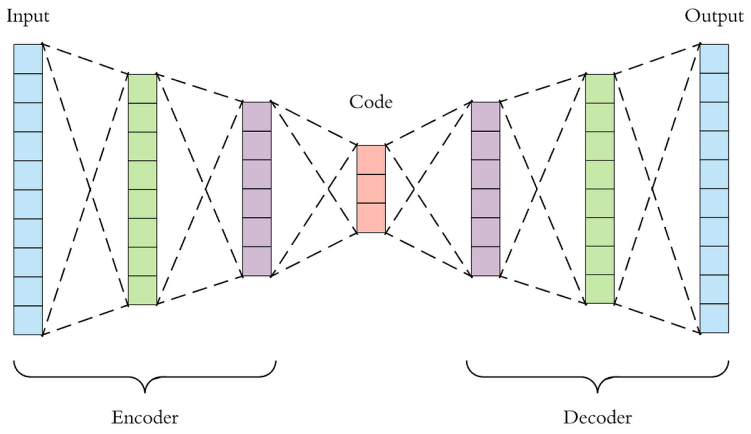


Figure 4.1: A linear autoencoder. Taken from here.

It can be shown that the minimum reconstruction error for $k < d$ is achieved by $W_1 = W_2^T = \Sigma_k^{-1/2} U_k^T$, where U_k is the matrix of the first k eigenvectors of the covariance matrix XX^T of the data. k is a hyperparameter that can be chosen by validation on the validation set.

Nonlinear autoencoders are, of course, just when the functions f and g are nonlinear. The most common choice is to use neural networks with nonlinear activation functions. Another way is to use nonlinear kernels and then use PCA - this is called kernel PCA.

4.3 Boosting

A very popular ensemble learning algorithm. The intuition is as filows: Can we start with a weaker model (a learner that is slightly better than the random model in expectation) and make it stronger by focusing on the difficult/hard to classify examples?

The main template of boosting algorithms is:

1. Train a weak learner on the training data.
2. Computer the error of the weak learner on the training data.
3. Increase the weights of the examples that were misclassified by the weak learner. This is the *boosting* step.
4. Go back to step 1.

This is a greedy algorithm over the space of possible functions. The weights of the examples are initialized to 1/N and are increased if the example is misclassified. The loss function is a weighted sum of the loss of each example.

Remark. Boosting algorithms are traditionally called meta-algorithms because they take as *argument* any algorithm for classifation. The boosting algorithm itself is not a classifier, but rather a procedure that takes a classifier and returns a better one.

4.3.1 AdaBoost

The most popular boosting algorithm. The algorithm is described in Algorithm 4 for binary classification. Here is the remarkable thing: the training error of AdaBoost is guaranteed to decrease exponentially fast with the number of iterations T as long as ϵ_t's correspond to learners better than chance. Even more remarkably, there are guarantees on its generalization error as well.

Algorithm 4 AdaBoost for Binary Classification

Initialize $w_i = 1/N$ for $i = 1, \ldots, N$

Initialize weak learner h_t to be a decision stump: $h_t(x) = \begin{cases} 1 & \text{if } x_j \leq \theta \\ -1 & \text{if } x_j > \theta \end{cases}$

for $t = 1, \ldots, T$ **do**

  Train a weak learner h_t on the training data with weights w_i

  Compute the error ϵ_t of h_t on the training data:

$$\epsilon_t = \sum_{i=1}^{N} w_i \mathbb{1}_{[y_i \neq h_t(x_i)]}$$

  Compute $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$. Note that $\alpha_t > 0$ iff $\epsilon_t < 1/2$.

  Update the weights: $w_i \leftarrow w_i \exp\left(\alpha_t (-1)^{\mathbb{1}_{[y_i \neq h_t(x_i)]}}\right)$

  Normalize the weights: $w_i \leftarrow \frac{w_i}{\sum_{j=1}^{N} w_j}$

end for

Output the boosted classifier $H(x) = \sigma\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

Chapter 5

Search

We move now to artificial intelligence, more generally. So far, we have seen reflex-based models, where given an input, we make a prediction using a set of rules (the trained model). All computations were *forward*, with no *backtracking*. We now look at state-based models: state information (the representation of the environment and current calculation), action information, goals (the 'input' to the model), and a *plan*. In some sense, state-based models are a bit like rational models.

5.1 The search problem

For now, we operate under the assumption that the environment is fully known. A search problem can be defined using:

1. A set of *states* S - called the state space, and a set of actions A (the action space).
2. A start state $s \in S$, and a goal state $f \in S$.
3. A transition funtion that maps $s \xrightarrow{a} s'$ on action a.

A *solution* is a sequence of states from the start state to the goal state. An optimal solution is one with minimum cost.

5.1.1 The frontier

At any time, there are three sets of states: the explored set E, the frontier F, and the unexplored set U. The explored set is the set of states that have been visited and expanded from, the frontier/fringe is the set of states that have been visited but not expanded, and the unexplored set is the set of states that have not been visited. The frontier is the set of states that are on the boundary between the explored and unexplored sets.

A step in the search would look like so: pick a state from the frontier based on some *strategy* (based on some sort of priority, or randomly), expand it to child nodes. Move the chosen frontier state to explored, and add the children to the frontier.

5.2 Uninformed search

This kind of search algorithms do not use any information about the goal state. They do not use some sort of proximity to the goal to guide their strategy. DFS - expands the 'deepest' node first/node on top of the stack - and BFS - expand shallowest node first/node on top of the queue - on the state-transition graph are examples of uninformed search algorithms.

Another example is uniform cost search, which expands the node with the lowest path cost (cost of the path from the start state to this node). The strategy is to pick up the lowest cost node from the frontier (using a priority queue, say). Why the name? - because we explore nodes in the order of their cost: all nodes with the same cost are visited in turn, and then we move on to the next cost.

Algorithm 5 Uniform cost search

```
explored ← ∅
frontier ← {s}
unexplored ← S \ {s}
while frontier ≠ ∅ do
    n, p ← pop(frontier) // pop the lowest cost node. p stores the cost of a minimal path to n
    explored ← explored ∪ {n}
    if n = f then
        return n
    end if
    for a ∈ A do
        n′ ← child of n on action a
        if n′ ∉ explored then
            frontier.update_priority(n′, p + cost(n, a, n′)) // the + operator need to satisfy the triangle inequality for the proof of optimality
        end if
    end for
end while
return failure // no solution found
```

Note that all costs must be non-negative and the $+$ operator must satisfy the triangle inequality for the proof of optimality to work.

Uninformed search is optimal, yes, but there is an important problem here: UCS may readliy move "away" from the goal state, because they do not bother with it (and are not informed about their distance from the goal anyway). Informed search essentially provides a fix to this problem.

5.3 Best-first search

The ordering of nodes in the frontier is crucial. Suppose the order is based on a function $f : S \to \mathbb{R}$. The node n in teh frontier with the least value of $f(n)$ is chosen to be expanded. In general, this is best-first search (the node in the frontier with the best value of f is chosen to be expanded). The function f is called the evaluation function.

UCS is a special case of best-first search, where $f(n) = p(n)$, the min path cost from the start state to n. A best-first search that avoids the problem with UCS is greedy best-first search, where $f(n) = h(n)$, the *estimated* cost from n to the goal state. It is quite good if the estimate is good. But note that it is not optimal, because it does not take into account the cost of the path from the start state to n.

5.3.1 A* search

This algorithm picks up the best of both worlds: it is fast, and it is optimal if the heuristic h (estimate for the distance to the goal) is consistent. The evaluation function used is $f(n) = g(n) + h(n)$. A consistent heuristic satisfies a certain triangle inequality: $h(n) \leq c(n, a, n') + h(n')$. A consistent heuristic is also admissible: the value $h(n)$ is at most the true cost from n to the goal state.