Introduction to Machine Learning
Draft Notes by Topic

Tommi Jaakkola and Regina Barzilay
MIT

Spring 2015

Copyright © 2015 Jaakkola and Barzilay. All rights reserved.

Contents

1 Classification 3

2 Linear Classification, Perceptron 8

3 Perceptron, Passive-Aggressive 13

4 Linear regression 18

5 Support Vector Machines and Kernels 23

6 Ensembles and Boosting 32

7 Generalization and Model Selection 37

8 Recommender Problems 42

9 Representation and Neural networks 52

10 Unsupervised Learning, Clustering 68

11 Generative Models and Mixtures 76

12 Bayesian networks 83

13 Hidden Markov Models 88

14 Reinforcement Learning 98

1 Classification

Much of what we do in engineering and sciences involves prediction. We can predict the weather tomorrow, properties of new materials, what people will purchase, and so on. Many of these predictions are data driven, i.e., based on past observations or measurements. For example, we can look at past weather patterns, measure properties of materials, record which products different people have bought/viewed before, and so on. In this course, we will look at such prediction problems as machine learning problems where the predictors are learned from data. We will try to understand how machine learning problems are formulated, which methods work when, what type of assumptions are needed, and what guarantees we might be able to provide for different methods. We will start with the simplest type of prediction problem – classification. In this case, the goal is to learn to classify examples such as biological samples, images, or text. Let's begin with a few scenarios.

- *Tumor classification.* Most tumors can be identified from tissue samples. While the samples themselves are easy to acquire from willing participants, the task of determining whether a sample contains tumor cells often involves a combination of laboratory tests and physician assessment. In order to easily screen large numbers of samples, we should develop automated methods that are capable of predicting a tumor/normal label for any new tissue sample. In machine learning, we call this problem a *classification problem.* Given some tissue samples with verified tumor/normal labels as *training data*, the goal is to find a good mapping from samples to labels – a *classifier.* Once found, the classifier can be easily applied to predict labels for new samples.

  But computers do not understand "tissue samples". We have to work a bit harder to describe each sample in a manner that can be used by automated methods. We could, for example, try to represent each sample with a *feature vector.* To construct such a vector, we could use readily available gene expression assays to first measure how active each gene is in a population of cells represented by the sample. In this case, each coordinate of the feature vector would correspond to the resulting expression of a particular gene. The assumption here is that the expression of genes would provide sufficient information to determine whether the sample is a tumor. Once feature vectors are available, we can apply a generic machine learning method to learn a mapping from vectors to labels. Such a method would be completely oblivious to how the vectors were constructed, and what the coordinates mean. All it would do is relate different coordinate values (or their combinations) to the corresponding labels in the training data.

  It is important that we follow the same protocol to construct feature vectors for training samples and any new samples to be classified. Otherwise the new samples would "look" different to the classifier and result in poor predictions. There are many subtleties here that we will address more formally later on in the course. For example, if the classifier is trained on samples from one type of tumor but tested on samples taken from another type of tumor, there's little reason to expect (in general) that the classifier would do well. We are making a tacit assumption that the samples used for training are somehow representative of the samples that the classifier sees (and tries to classify) later on.

- *Gender from images.* Most people would find it relatively easy to determine the gender of a person based on portraits. But a human-powered solution is expensive, and we would rather use an automated method for predicting gender across large sets of images. In order to set up gender classification as a machine learning problem, we will first need a bit of human assistance to create a set of labeled images, i.e., (image,gender) pairs. These labeled faces constitute the training set for the classification method. As in the tissue classification problem, the learning task is to relate the descriptions of images (as feature vectors) to the corresponding gender labels. Once the mapping from images to labels is found, we can easily apply the mapping to label large numbers of new images.

  How do we represent images as feature vectors? We could take a high resolution pixel image of a face and simply concatenate all the pixel values (color, intensity) into a long feature vector. While possible, this may not work very well. The reason is that we are leaving everything for the classification method to figure out. For example, hair, skin color, eyes, etc may be important "features" to pay attention to in order to predict gender but none of these features are deducible from individual pixels. Indeed, images in computer vision are often mapped to feature vectors with the help of simple detectors that act like classifiers themselves. They may detect edges, color patches, different textures, and so on. It is more useful to concatenate the outputs of these detectors into a feature vector and use such vectors to predict gender. More generally, it is important to *represent* the examples to be classified in a way that the information pertaining to the labels is more easily accessible.

Classification as machine learning

Let's look at these types of classification problems a bit more formally. We will use $x = [x_1, \ldots, x_d]^T \in \mathcal{R}^d$ to denote each feature (column) vector of dimension d. To avoid confusion, when x is used to denote the original object (e.g., sample, image, document), we will use $\phi(x) \in \mathcal{R}^d$ for the feature vector constructed from object x. But, for now, let's simply use x as a column feature vector, as if it was given to us directly. In other words, let's look at the problem as the classification method sees it.

Each training example x is associated with a binary label $y \in \{-1, 1\}$. For new examples, we will have to predict the label. Let's say that we have n training examples available to learn from. We will index the training examples with superscripts, $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ and similarly for the corresponding labels $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$. All that the classification method knows about the problem is the training data as n pairs $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, n$. Let's call this training data S_n where the subscript n highlights the number of examples we have.

A classifier h is a mapping from feature vectors to labels: $h : \mathcal{R}^d \rightarrow \{-1, 1\}$. When we apply the classifier to a particular example x, we write $h(x)$ as the predicted label. Any learning algorithm entertains a set of alternative classifiers $\mathcal{H}$, and then selects one $\hat{h} \in \mathcal{H}$ based on the training set S_n. The goal is to select $\hat{h} \in \mathcal{H}$ that would have the best chance of correctly classifying new examples that were not part of the training set. Note the difficulty here. All the information we have about the classification problem is the training set S_n but we are actually interested in doing well on examples that were not part of the training set. In other words, we are interested in prediction.

Our brief discussion is already touching on some of the key aspects of learning problems:

1. *Set of classifiers $\mathcal{H}$*. Depending on the context, this set is also known as *the model* or *the hypothesis class*. The larger this set is, the more powerful the classification method is. In other words, there are many classifiers to choose from in response to the training set, many alternative hypotheses about how feature vectors relate to labels. We will later formalize exactly how to think about the size of $\mathcal{H}$. For now, to gain intuition, you can think of it as a discrete set, where each member is a (slightly) different classifier.

2. *Learning algorithm/criterion*. The problem of finding $\hat{h} \in \mathcal{H}$ based on the training set S_n is solved by the learning algorithm. This training problem is often cast as an optimization problem, and we will talk about the *objective function or estimation criterion* for selecting $\hat{h}$ from $\mathcal{H}$. Note that many learning algorithms or corresponding estimation criteria might use the same set of classifiers $\mathcal{H}$ but still select different classifiers in responds to the training set.

3. *Generalization*. The goal of the learning algorithm is to find a classifier $\hat{h} \in \mathcal{H}$ that will work well on yet unseen examples x. Put another way, we say that we want a classifier that *generalizes* well. How well the resulting classifier will work on new examples depends on the choice of $\mathcal{H}$, the training data S_n, as well as the learning algorithm. A classifier that predicts all the training labels correctly may not generalize well. In order to generalize, we must capture something about how the feature vectors relate to the labels.

Let's take a simple example to see how these concepts relate, and what we must do to generalize well. Consider the gender classification task based on images discussed above. Assume that each image (grayscale) is represented as a column vector x of dimension d. The pixel intensity values in the image, column by column, are concatenated into a single column vector. If the image has 128 by 128 pixels, then $d = 16\,384$. We assume that all the images are of the same size. You may remember that we had already argued against using this simple vector representation for images. Indeed, there are many better ones. However, we will use this easy-to-understand representation to illustrate some of the basic learning issues.

A classifier in this context is a binary valued function $h : \mathcal{R}^d \rightarrow \{-1, 1\}$ chosen on the basis of the training set alone. For our task here we assume that the classifier knows nothing about images (or faces for that matter) beyond the labeled training set. So, for example, from the point of view of the classifier, the images could have been measurements of weight, height, etc. rather than pixel intensities. The classifier only has a set of n training vectors $x^{(1)}, \ldots, x^{(n)}$ with binary ± 1 labels $y^{(1)}, \ldots, y^{(n)}$. This is the only information about the task that we can use to constraint what the classifier $\hat{h}$ should be.

In order to learn anything, we will have to constrain the set of classifiers $\mathcal{H}$. Put another way, effective learning requires constraints. Let's see first what would happen without any constraints. To this end, suppose we have $n = 50$ labeled 128×128 pixel images where the pixel intensities range from 0 to 255. Given the small number of training examples, it is likely that one of the pixels, say pixel i, has a distinct value in each of the training images. If $\mathcal{H}$ includes all possible classifiers (no constraints), then we could also find a classifier that relies on the value of this single pixel alone, yet perfectly maps all the training images to their correct labels. Let $x_i^{(t)}$ refer to pixel i in the t^{th} training image, and suppose x'_i is the i^{th} pixel in any image x'. Then our simple single pixel classifier could be written as

$$\hat{h}(x') = \begin{cases} y^{(t)}, \text{ if } x_i^{(t)} = x'_i \text{ for some } t = 1, \ldots, n \text{ (in this order)} \\ -1, \text{ otherwise} \end{cases} \tag{1}$$

You should verify that this classifier does indeed map training examples to their correct labels. In fact, when there are no constraints on $\mathcal{H}$, it is always possible to come up with such a "perfect" classifier if the training images are all distinct (no two images have identical pixel intensities for all the pixels). Any training algorithm that tries to find classifiers that make few errors on the training set could end up selecting such $\hat{h}$.

But we are forgetting here that the task is *not* to correctly classify the training images; the training set is merely a helpful source of information. Our task is to do well on yet unseen images. Do we expect our single pixel classifier to correctly classify images not in the training set? New images are likely to portray different people, in different orientations, under varying lighting conditions, etc. The value of the single pixel (e.g., background) is likely to bear no relevance to the gender label.

What went wrong? Our set $\mathcal{H}$ (no constraints) is too large. It contains classifiers which do well on the training set but perform poorly on new images. This is a subtle but important sentence. Any fixed classifier, however complicated it may be, would statistically speaking do about the same on the training set as on new images. The problem here is not a particular classifier but the fact that there are so many choices available in $\mathcal{H}$ that we may end up choosing the one that does do substantially better on the specific training set we have than on yet unseen images. We say that in such cases $\mathcal{H}$, i.e., the set of classifiers, *overfits* the training data. The training set is too small in relation to $\mathcal{H}$ in order for us to have any statistical power to distinguish between all the available choices $h \in \mathcal{H}$.

In order to find classifiers that *generalize* well, we must constrain the set $\mathcal{H}$. We would like to find a set of classifiers such that if a classifier chosen from this set works well on the training set, it is also likely to work well on the unseen images. This right set of classifiers cannot be too large in the sense of containing too many clearly different functions. Otherwise we are likely to find classifiers such as the trivial ones that are close to perfect on the training set but do not generalize well. The set of classifiers should not be too small either or we run the risk of not finding any classifiers that work well even on the training set. For example, suppose $\mathcal{H}$ contains only one classifier. There's really no learning here, we will choose the same $h \in \mathcal{H}$ regardless of the training set. But, we do know that how well it does on the training set is probably a good measure of how well it does on new images. Finding the right set is a key problem in machine learning, also known as the *model selection* problem. We will discuss it more later.

Linear classification

Let's start by considering a particular constrained set of classifiers. Specifically, we will look at *linear classifiers through origin.* These are thresholded linear mappings from images to labels. More formally, we only consider classifiers of the form

$$h(x;\theta) = \text{sign}\big(\,\theta_1 x_1 + \ldots + \theta_d x_d\,\big) = \text{sign}\big(\,\theta \cdot x\,\big) = \begin{cases} +1, & \theta \cdot x > 0 \\ -1, & \theta \cdot x \leq 0 \end{cases} \tag{2}$$

where $\theta \cdot x = \theta^T x$ and $\theta = [\theta_1, \ldots, \theta_d]^T$ is a column vector of real valued parameters. Different settings of the parameters give rise to different classifiers in this set. In other words, θ indexes the classifiers. Any two classifiers corresponding to different parameters would produce a different prediction for some input images x. We say that the classifiers in this set are *parameterized* by $\theta \in \mathcal{R}^d$.

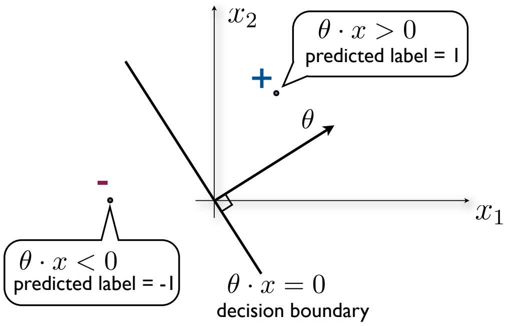


Figure 1: A linear classifier through origin.

We can also understand these linear classifiers geometrically. Suppose we select one classifier, i.e., fix parameters θ, and look at what happens for different images x. The classifier changes its prediction only when the argument to the sign function $\theta \cdot x$ changes from positive to negative (or vice versa). Geometrically, in the space of image vectors, this transition corresponds to crossing the *decision boundary* where the argument is exactly zero: all images x such that $\theta \cdot x = 0$ lie exactly on the decision boundary. This is a linear equation in x (θ is fixed). It defines a plane in d-dimensions, a plane that goes through the origin $x = 0$ (since $\theta \cdot 0 = 0$). What is the direction of this plane? The parameter vector θ is normal (orthogonal) to this plane; this is clear since the plane is defined as all x for which $\theta \cdot x = 0$. The θ vector as the normal to the plane also specifies the direction in the image space along which the value of $\theta \cdot x$ would increase the most. Figure **??** tries to illustrate these concepts in two dimensions.

2 Linear Classification, Perceptron

Learning linear classifiers

During the last lecture we started to explore linear classifiers through origin. These classifiers were defined as

$$h(x;\theta) = \text{sign}\big(\,\theta_1 x_1 + \ldots + \theta_d x_d\,\big) = \text{sign}\big(\,\theta \cdot x\,\big) = \begin{cases} +1, & \theta \cdot x > 0 \\ -1, & \theta \cdot x \leq 0 \end{cases} \tag{3}$$

where x is the feature (input) vector, $\theta \cdot x = \theta^T x$, and $\theta = [\theta_1, \ldots, \theta_d]^T$ is a column vector of real valued parameters. The parameter vector serves as an "index" that specifies the particular classifier among all the linear classifiers. Since the parameters are continuous valued, this is still a large (uncountably infinite) set of possible classifiers.

The name linear classifier comes from the fact that the *decision boundary* is linear (line in 2d, plane in 3d, hyper-plane in higher dimensions). Specifically, the decision boundary is defined as the set of all $x \in \mathcal{R}^d$ for which $\theta \cdot x = 0$. These points lie exactly on the decision boundary. Another way to understand binary classifiers is to explicitly evaluate how they divide the space (here $\mathcal{R}^d$) into two regions based on the label. For linear classifiers, both

$$\mathcal{X}^+(\theta) \;\; = \;\; \{x \in \mathcal{R}^d :\; h(x;\theta) = +1\} \tag{4}$$
$$\mathcal{X}^-(\theta) \;\; = \;\; \{x \in \mathcal{R}^d :\; h(x;\theta) = -1\} \tag{5}$$

are half-spaces, separated by the decision boundary. Note that, clearly, these half spaces as well as the decision boundary, depend on how we set θ, i.e., which classifier we choose.

Now that we have chosen a set of classifiers, we still need to choose one of them in response to the training set of labeled examples $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$. For simplicity, we assume that since our classifiers are linear, i.e., highly constrained, we can just find one that does well on the training set (we will revisit this issue later). For example, we could find θ that results in the fewest mistakes on the training set, i.e., we would minimize the training error

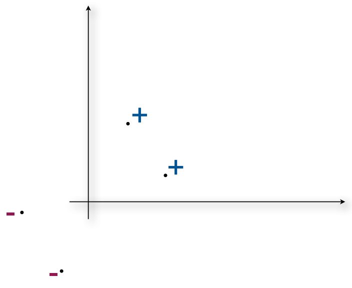
$$\mathcal{E}_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} [\![\, y^{(t)} \neq h(x^{(t)};\theta)\,]\!] = \frac{1}{n}\sum_{t=1}^{n} [\![\, y^{(t)}(\theta \cdot x^{(t)}) \leq 0\,]\!] \tag{6}$$

where $[\![\, \cdot \,]\!]$ returns 1 if the logical expression in the argument is true, and zero otherwise. The training error here is the fraction of training examples for which the classifier with parameters θ predicts the wrong label. Note that incorrect prediction happens if $y\theta \cdot x \leq 0$, i.e., when the label y does not have the same sign as $\theta \cdot x$ or lies exactly on the decision boundary (which we will count as an error). The training error $\mathcal{E}_n(\theta)$ is calculated as a function of the parameters θ, and can be minimized with respect to θ.

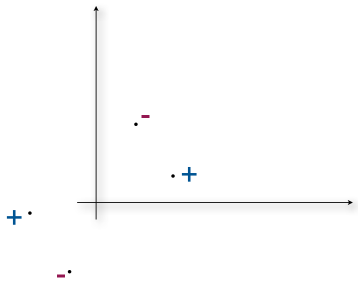
What would a reasonable algorithm be for finding $\hat{\theta}$ that minimizes $\mathcal{E}_n(\theta)$? Unfortunately, this is not an easy problem to solve in general, and we will have to settle for an algorithm that approximately minimizes the training error. However, for this lecture, we consider a special case where there exists a linear classifier (through origin) that achieves zero training error. This is also known as the *realizable case.* Note that "realizability" depends on both the training examples as well as the set of classifiers we have adopted. Specifically, for linear classifiers, we assume that the training examples are *linearly separable through origin*:

Definition: Training examples $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$ are linearly separable through origin if there exists a parameter vector $\hat{\theta}$ such that $y^{(t)}(\hat{\theta} \cdot x^{(t)}) > 0$ for all $t = 1, ..., n$.

Here are a couple of examples:



linearly separable through origin



not linearly separable

The perceptron algorithm

We'll consider here an algorithm that is *mistake driven*. In other words, it starts with a simple classifier, e.g., $\theta = 0$ (zero vector), and successively tries to adjust the parameters, based on each training example, so as to correct any mistakes. The simplest algorithm of this type is the so-called *perceptron* update rule. In this algorithm, we set $\theta = 0$, and subsequently consider each training example one by one, cycling through all them, and adjusting the parameters according to:

$$\begin{aligned} &\text{if } \; y^{(t)} \neq h(x^{(t)}; \theta^{(k)}) \; \text{ then} && (7) \\ &\theta^{(k+1)} = \theta^{(k)} + y^{(t)} x^{(t)} && (8) \end{aligned}$$

where $\theta^{(k)}$ denotes the parameters after k mistakes ($\theta^{(0)} = 0$). In other words, the parameters are changed only if we make a mistake, and we track the evolution of the parameters as a function of the mistakes. Normally, we would simply overwrite the earlier parameters through the update but we will here keep track of the parameters as they are changed so as to understand the algorithm better.

The perceptron updates do tend to correct mistakes. Too see this, consider a simple two dimensional example in figure 2. The points $x^{(1)}$ and $x^{(2)}$ in the figure are chosen such that the algorithm makes a mistake on both of them during its first pass. As a result, the updates become: $\theta^{(0)} = 0$ and

$$\begin{aligned} \theta^{(1)} &= \theta^{(0)} + x^{(1)} && (9) \\ \theta^{(2)} &= \theta^{(1)} + (-1)x^{(2)} && (10) \end{aligned}$$

In this simple case, both updates result in correct classification of the respective examples, and the algorithm would terminate. However, each update can also undershoot in the sense that the example that triggered the update would be misclassified even after the update. Can you construct a setting where an update would undershoot?

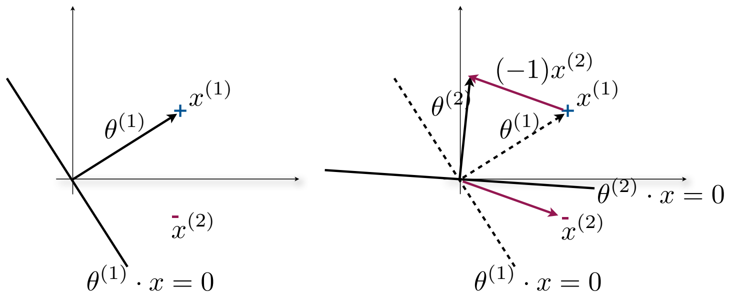


Figure 2: The perceptron update rule

Let's look at the updates more algebraically. Note that when we make a mistake the sign of $(\theta^{(k)} \cdot x^{(t)})$ disagrees with $y^{(t)}$ and the product $y^{(t)}(\theta^{(k)} \cdot x^{(t)})$ is non-positive; the product is positive for correctly classified examples. Suppose we make a mistake on $x^{(t)}$. Then the updated parameters are given by $\theta^{(k+1)} = \theta^{(k)} + y^{(t)}x^{(t)}$. If we consider classifying the same example $x^{(t)}$ after the update, using the new parameters $\theta^{(k+1)}$, then

$$\begin{aligned}
y^{(t)}(\theta^{(k+1)} \cdot x^{(t)}) &= y^{(t)}(\theta^{(k)} + y^{(t)}x^{(t)}) \cdot x^{(t)} && (11)\\
&= y^{(t)}(\theta^{(k)} \cdot x^{(t)}) + (y^{(t)})^2(x^{(t)} \cdot x^{(t)}) && (12)\\
&= y^{(t)}(\theta^{(k)} \cdot x^{(t)}) + \|x^{(t)}\|^2 && (13)
\end{aligned}$$

In other words, the value of $y^{(t)}(\theta \cdot x^{(t)})$ increases as a result of the update (becomes more positive). If we consider the same example repeatedly, then we will necessarily change the parameters such that the example will be classified correctly, i.e., the value of $y^{(t)}(\theta \cdot x^{(t)})$ becomes positive. Of course, mistakes on other examples may steer the parameters in different directions so it may not be clear that the algorithm converges to something useful if we repeatedly cycle through the training examples. The algorithm does converge in the realizable case:

Theorem: The perceptron update rule converges after a finite number of mistakes when the training examples are linearly separable through origin.

We will see later that the number of mistakes that the algorithm makes as it passes through the training examples depends on how easy or hard the classification task is. If the training examples are well-separated by a linear classifier (a notion which we will define formally), the perceptron algorithm converges quickly, i.e., it makes only a few mistakes in total until all the training examples are correctly classified.

What if the training examples are not linearly separable? In this case, the algorithm cannot converge. There would always be a mistake in each pass through the training examples, and the parameters would be changed. Better algorithms exist in this sense, and will be discussed them later on.

Linear classifiers with offset

We extend here the set of linear classifiers slightly by including a scalar offset parameter θ_0. This parameter will enable us to place the decision boundary anywhere in $\mathcal{R}^d$, not only through the origin. Specifically, a linear classifier with offset, or simply linear classifier, is defined as

$$h(x;\theta,\theta_0) = \text{sign}\big(\,\theta\cdot x + \theta_0\,\big) = \begin{cases} +1, & \theta\cdot x + \theta_0 > 0 \\ -1, & \theta\cdot x + \theta_0 \le 0 \end{cases} \tag{14}$$

Clearly, if $\theta_0 = 0$, we obtain a linear classifier through origin. For a non-zero value of θ_0, the resulting decision boundary $\theta\cdot x + \theta_0 = 0$ no longer goes through the origin (see figure 3 below). The hyper-plane (line in 2d) $\theta\cdot x + \theta_0 = 0$ is oriented parallel to $\theta\cdot x = 0$. If they were not, then there should be some x that satisfies both equations: $\theta\cdot x + \theta_0 = \theta\cdot x = 0$. This is possible only if $\theta_0 = 0$. We can conclude that vector θ is still orthogonal to the decision boundary, and also defines the positive direction in the sense that if we move x in this direction, the value of $\theta\cdot x + \theta_0$ increases. In the figure below, $\theta_0 < 0$ because we have to move from the origin (where $\theta\cdot x = 0$) in the direction of θ (increasing $\theta\cdot x$) until we hit $\theta\cdot x + \theta_0 = 0$.

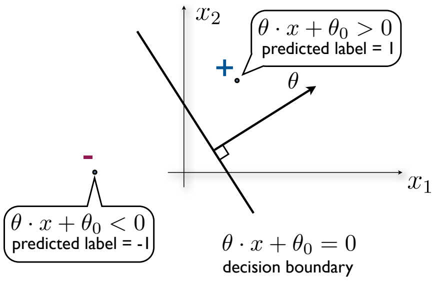

x_2
$\theta\cdot x + \theta_0 > 0$
predicted label = 1
+
θ
-
x_1
$\theta\cdot x + \theta_0 < 0$
predicted label = -1
$\theta\cdot x + \theta_0 = 0$
decision boundary

Figure 3: Linear classifier with offset parameter

Both linear separability and the perceptron algorithm for learning linear classifiers generalize easily to the case of linear classifiers with offset. Specifically,

Definition: Training examples $S_n = \{(x^{(t)}, y^{(t)}\}), t = 1,\ldots,n\}$ are linearly separable if there exists a parameter vector $\hat{\theta}$ and offset parameter $\hat{\theta}_0$ such that $y^{(t)}(\hat{\theta}\cdot x^{(t)} + \hat{\theta}_0) > 0$ for all $t = 1, ..., n$.

If training examples are linearly separable through origin, they are clearly also linearly separable. The converse is not true in general, however. Can you find such an example?

The perceptron algorithm is also modified only slightly: initialize $\theta^{(0)} = 0$ (vector) and $\theta_0^{(0)} = 0$ (scalar). Cycle through the training examples $t = 1,\ldots,n$ and update parameters according to

$$\text{if } \; y^{(t)} \neq h(x^{(t)}; \theta^{(k)}, \theta_0^{(k)}) \; \text{ then} \tag{15}$$

$$\theta^{(k+1)} = \theta^{(k)} + y^{(t)}x^{(t)} \tag{16}$$

$$\theta_0^{(k+1)} = \theta_0^{(k)} + y^{(t)} \tag{17}$$

Why is the offset parameter updated in this way? Think of it as a parameter associated with an additional coordinate that is set to 1 for all examples. If training examples are linearly separable (not necessarily through the origin), then the above perceptron algorithm converges after a finite number of mistakes.

Linear separation with margin

We can understand which problems are easy or hard for the perceptron algorithm. Easy problems mean that the algorithm converges quickly (and will also have good guarantees of generalization, i.e., will accurately classify new examples). Hard problems, even if still linearly separable, will require many passes through the training set before the algorithm finds a separating solution. More formally, the notion of "easy" relates to how far the training examples are from the decision boundary.

Definition: Training examples $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$ are linearly separable with margin γ if there exists a parameter vector $\hat{\theta}$ and offset parameter $\hat{\theta}_0$ such that $y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)/\|\hat{\theta}\| \geq \gamma$ for all $t = 1, ..., n$. If you take any linear classifier defined by $\hat{\theta}$ and $\hat{\theta}_0$ that correctly classifies the training examples, then

$$\frac{y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} \tag{18}$$

is exactly the orthogonal distance of point $x^{(t)}$ from the decision boundary $\hat{\theta} \cdot x + \hat{\theta}_0 = 0$. To see this, consider the figure below, where we have one positively labeled point $x^{(t)}$ and its orthogonal projection onto the boundary, the point x^0. Now, because $y^{(t)} = 1$ and $\hat{\theta} \cdot x^0 + \hat{\theta}_0 = 0$ (on the boundary),

$$\frac{y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} = \frac{(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} - \frac{\overbrace{(\hat{\theta} \cdot x^0 + \hat{\theta}_0)}^{=0}}{\|\hat{\theta}\|} = \frac{\hat{\theta} \cdot (x^{(t)} - x^0)}{\|\hat{\theta}\|} = \|x^{(t)} - x^0\| \tag{19}$$

where the last step follows from the fact that $\hat{\theta}$ is, by definition, oriented in the same direction as $x^{(t)} - x^0$.

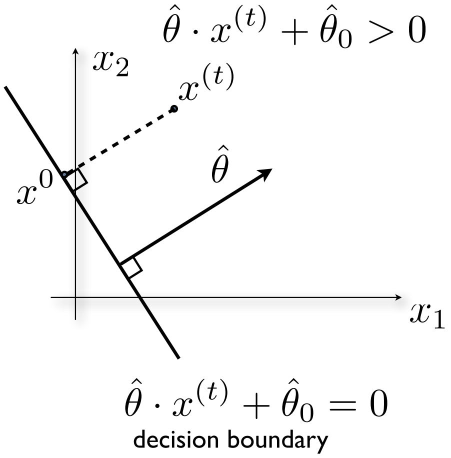

$\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0 > 0$
x_2
$x^{(t)}$
$\hat{\theta}$
x^0
x_1
$\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0 = 0$
decision boundary

Figure 4: Orthogonal projection onto the boundary

Next time we will discuss and show stronger guarantees for the perception algorithm based on linear separation with margin.

3 Perceptron, Passive-Aggressive

Perceptron convergence theorem

We will consider here a sequential prediction task (prediction game) where examples and labels are given one after the other without ever going back. For example, a high-speed trading problem can be viewed in this way. If cast as a classification problem, the task would be to predict price movement (up/down) on the basis of past pricing information. On the i^{th} round of the game, we get to see the feature vector $x^{(i)}$ (consisting of past prices), have to make a prediction about the next price movement (e.g., evaluating the sign of $\theta \cdot x^{(i)}$), and finally obtain the correct label $y^{(i)}$ (whether price went up/down). If we made a mistake, we can update the parameters θ with hopes of doing better in the future. Our goal is to understand how many mistakes the perceptron algorithm will make along this infinite sequence of examples.

Clearly, we need to assume something. If an adversary was selecting the prices (the target labels) with full knowledge of our prediction strategy, they could always choose to counter our predictions. In other words, we would make an error on each example! We must assume something about the sequence of examples and labels in order to provide a meaningful guarantee for the perceptron (or any) algorithm. For simplicity, we will formulate the assumptions as well as the guarantee for linear separators through origin. The two assumptions we will use

(A) There exists θ^* such that $y^{(i)}(\theta^* \cdot x^{(i)})/\|\theta^*\| \geq \gamma$ for all $i = 1, 2, \ldots$, and some $\gamma > 0$.

(B) All the examples are bounded $\|x^{(i)}\| \leq R$, $i = 1, 2, \ldots$

Note that the two assumptions constrain the sequence of examples and labels. They are not properties of the prediction algorithm (perceptron) we use. Assumption (A) requires that there exists at least one linear classifier through origin such that all the examples lie on the correct side of the boundary and are at least distance (margin) γ away from the boundary. Assumption (B), on the other hand, ensures that we cannot just scale all the examples to increase the separation. Indeed, it is the ratio R/γ that matters in terms of the number of mistakes. We will prove the following theorem:

Perceptron convergence theorem If (A) and (B) hold, then the perceptron algorithm will make at most $(R/\gamma)^2$ mistakes along the infinite sequence of examples and labels $(x^{(i)}, y^{(i)})$, $i = 1, 2, \ldots$

There are a few remarkable points about this result. In particular, what it does not depend on. It does not depend on the dimensionality of the feature vectors at all! They could be very high or even infinite dimensional provided that they are well-separated as required by (A) and bounded as stated in (B). Also, the number of examples considered (the infinite sequence) is not relevant. The only thing that matters is how easy the classification task is as defined by the ratio of the magnitude of examples to how well they are separated.

Now, to show the result, we will argue that each perceptron update helps steer the parameters a bit more towards the unknown but assumed to exist θ^*. In fact, each perceptron update (each mistake) will help with a finite increment. These increments (resulting from mistakes) cannot continue forever since there's a limit to how close we can get (cannot be better than identical).

To begin with, we note that the decision boundary $\{x : \theta \cdot x = 0\}$ only depends on the orientation, not the magnitude of θ. It suffices therefore to find θ that is close enough in angle to θ^*. So, we will see how

$$cos(\theta^{(k)}, \theta^*) = \frac{\theta^{(k)} \cdot \theta^*}{\|\theta^{(k)}\| \|\theta^*\|} \tag{20}$$

behaves as a function of k (number of mistakes) where $\theta^{(k)}$ is the parameter vector after k mistakes (updates). Let's break the cosine into two parts $\theta^{(k)} \cdot \theta^* / \|\theta^*\|$ and $\|\theta^{(k)}\|$. The cosine is the ratio of the two. First, based on the form of the perceptron update, if the kth mistake happened on example $(x^{(i)}, y^{(i)})$, we can write

$$\frac{\theta^{(k)} \cdot \theta^*}{\|\theta^*\|} = \frac{(\theta^{(k-1)} + y^{(i)} x^{(i)}) \cdot \theta^*}{\|\theta^*\|} = \frac{\theta^{(k-1)} \cdot \theta^*}{\|\theta^*\|} + \frac{y^{(i)} x^{(i)} \cdot \theta^*}{\|\theta^*\|} \geq \frac{\theta^{(k-1)} \cdot \theta^*}{\|\theta^*\|} + \gamma \geq k\gamma \tag{21}$$

where we have used assumption (A) since θ^* correctly classifies all the examples with margin γ. So, clearly, this term will get an increment γ every time the update is made. Note that for us to get this increment, the updates don't have to be made based only in response to mistakes. We could update on each example! Why only on mistakes then? This is because the expression for cosine is divided by $\|\theta^{(k)}\|$ and we have to keep this magnitude in check in order to guarantee progress towards increasing the angle. By expanding $\theta^{(k)}$ again, we get

$$\begin{aligned} \|\theta^{(k)}\|^2 = \|\theta^{(k-1)} + y^{(i)} x^{(i)}\|^2 &= \|\theta^{(k-1)}\|^2 + \overbrace{2y^{(i)} \theta^{(k-1)} \cdot x^{(i)}}^{<0} + \overbrace{\|x^{(i)}\|^2}^{\leq R^2} && (22) \\ &\leq \|\theta^{(k-1)}\|^2 + R^2 \leq kR^2 && (23) \end{aligned}$$

since $\theta^{(0)} = 0$ by assumption. Here we have explicitly used the fact that $\theta^{(k-1)}$ makes a mistake on $(x^{(i)}, y^{(i)})$ as well as the fact that the examples are bounded (assumption (B)). As a result, we get $\|\theta^{(k)}\| \leq \sqrt{k}R$. Taken together,

$$cos(\theta^{(k)}, \theta^*) = \frac{\theta^{(k)} \cdot \theta^*}{\|\theta^{(k)}\| \|\theta^*\|} \geq \frac{k\gamma}{\sqrt{k}R} = \sqrt{k}\frac{\gamma}{R} \tag{24}$$

Since cosine is bounded by one, we cannot continue making mistakes (continue increasing k). If we solve for k from $1 \geq \sqrt{k}(\gamma/R)$, we get $k \leq (R/\gamma)^2$ as desired.

Perceptron issues

We can apply the perceptron convergence theorem also to the problem of running the algorithm on a fixed set of training examples. By cycling through the training examples, we will simply face the same set of n examples again and again. This is a possible (infinite) sequence of examples and labels. The result holds just the same but now the separation and boundedness are properties of the training set, and we are counting mistakes on the training examples.

The perceptron algorithm stops after all the training examples are correctly classified. However, there are potentially many possible such separators. We might, for example, end up with

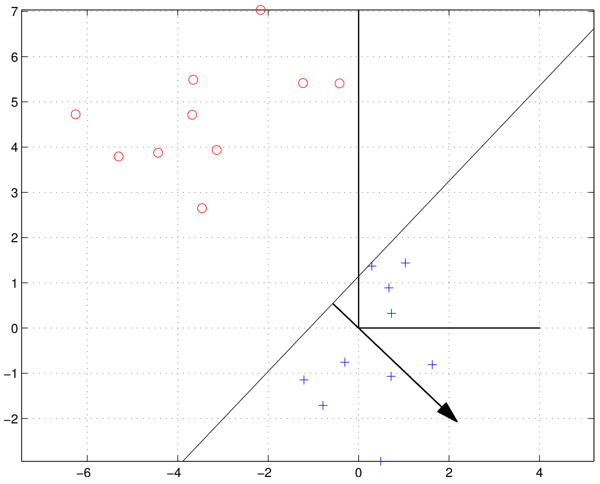


Figure 5: Linear classifier with offset that correctly classifies the training examples but is unlikely to generalize well

The problem here is that the positively labeled examples remain too close to the boundary. It would be better if the algorithm found a solution closer to the middle of the two sets of labeled points. If the boundary is far from the training examples, then any test points that are perturbed versions of the training examples would be classified correctly. In order to ensure this, we need to measure errors in a more refined manner so that the algorithm will update the parameters based on examples that are correctly classified but (in some sense) too close to the boundary.

From errors to loss functions

Instead of counting whether or not we make a mistake, we will assess our "loss" on each training example. The loss function measures how badly we classify each example. The perceptron algorithm implicitly assumes that we use the zero-one loss (simply the error), i.e.,

$$\text{Loss}_{0,1}(y\theta \cdot x) = [\![\, y\theta \cdot x \leq 0 \,]\!] \tag{25}$$

Note that the loss function is a function of the "agreement" $y\theta \cdot x$ but only cares about whether this agreement is positive (correct classification) or not. We can introduce a better loss function by starting to measure penalty already when the agreement drops below one. For example, we can use the Hinge loss

$$\text{Loss}_h(y\theta \cdot x) = \max\{0, 1 - y\theta \cdot x\} = \begin{cases} 1 - y\theta \cdot x \text{ if } y\theta \cdot x \leq 1 \\ 0, \text{ o.w.} \end{cases} \tag{26}$$

Geometrically, we are introducing parallel boundaries around the decision boundary known as the margin boundaries $\{x : \theta \cdot x = 1\}$ and $\{x : \theta \cdot x = -1\}$ (see the figure below). In order for a positively labeled point to get zero Hinge loss, it would have to be on the correct side of the decision boundary and also beyond the margin boundary $\{x : \theta \cdot x = 1\}$. The margin boundaries are at distance exactly $1/\|\theta\|$ away from the decision boundary. To see this, consider a point (x, y) that is correctly classified and lies exactly on the margin boundary. As a result, $y\theta \cdot x = 1$, and the orthogonal distance from the boundary is $y\theta \cdot x/\|\theta\| = 1/\|\theta\|$.

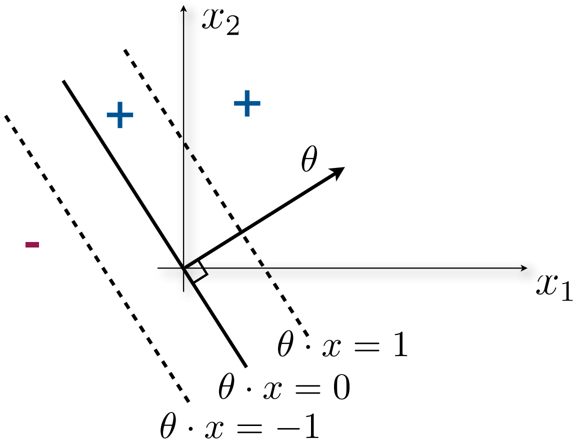


Figure 6: Decision and margin boundaries for a linear classifier through origin

Passive aggressive algorithm

We are now looking for an algorithm that finds parameters that result in a small Hinge loss over the training examples. As before, we are not trying to solve just an optimization problem. We need to minimize the loss in a guarded fashion so as to generalize well. We will talk about generalization in a more formal way later. For now, the algorithm we want is also an on-line algorithm that updates the parameters in response to each training example in turn. As such, it must balance two opposing tendencies. First, it should perform an update so as to better classify the current example, i.e., reduce the Hinge loss on this example. At the same time, it shouldn't forget the updates made so far, i.e., it should keep the parameters close to what they were before, parameters that were adjusted based on the other examples.

More formally, if $\theta^{(k)}$ are our current parameters, and (x, y) is the training example in question, we find $\theta = \theta^{(k+1)}$ that minimizes

$$\frac{\lambda}{2}\|\theta - \theta^{(k)}\|^2 + \text{Loss}_h(y\theta \cdot x) \tag{27}$$

The parameter λ determines how passive or aggressive our update is. Larger values will result in parameters close to $\theta^{(k)}$, smaller values of λ permit larger deviations so as to minimize $\text{Loss}_h(y\theta \cdot x)$. The resulting update, i.e., the minimizing solution, looks very much like the perceptron update

$$\theta^{(k+1)} = \theta^{(k)} + \eta\, yx \tag{28}$$

where η is a step-size (learning rate) that depends on both the current parameters $\theta^{(k)}$ and

(x, y). In our case, the step-size can be solved in closed form

$$\eta = \min\left\{ \frac{\mathrm{Loss}_h(y\theta^{(k)} \cdot x)}{\|x\|^2}, \frac{1}{\lambda} \right\} \tag{29}$$

What happens if (x, y) is already beyond the margin boundaries relative to $\theta^{(k)}$? In this case, $\mathrm{Loss}_h(y\theta^{(k)} \cdot x) = 0$, and the step-size evaluates to zero as well. In other words, we will not perform any updates on examples beyond the margin boundaries. Note also that if λ is large, η will be small because it cannot exceed $1/\lambda$. The resulting updates will be small (smaller than what perceptron would make).

We can write the passive aggressive algorithm in a similar form as the perceptron algorithm (here over a fixed training set). We initialize $\theta^{(0)} = 0$ (vector) and cycle through the training examples $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, n$, performing updates according to

$$\eta = \min\left\{ \frac{\mathrm{Loss}_h(y^{(i)}\theta^{(k)} \cdot x^{(i)})}{\|x^{(i)}\|^2}, \frac{1}{\lambda} \right\} \tag{30}$$

$$\theta^{(k+1)} = \theta^{(k)} + \eta\, y^{(i)} x^{(i)} \tag{31}$$

A better version of the algorithm would add, in addition, an averaging step over the parameters, i.e., return $\frac{1}{nT}(\theta^{(1)} + \ldots + \theta^{(nT)})$ instead of just the last parameter vector $\theta^{(nT)}$, where T is the number of times we have gone through the training set. The averaging takes place whether or not the parameters receive a non-zero update, and can be implemented more efficiently than keeping all the parameters. One reason for this averaging is that for any finite λ, even after many steps, the parameters may keep on changing, bouncing around some “mean solution” that we want.

4 Linear regression

A linear regression function is simply a linear function of the feature vectors, i.e.,

$$f(x;\theta,\theta_0) = \theta \cdot x + \theta_0 = \sum_{i=1}^{d} \theta_i x_i + \theta_0 \tag{32}$$

Each setting of the parameters θ and θ_0 gives rise to a slightly different regression function. Collectively, different parameter choices $\theta \in \mathcal{R}^d$, $\theta_0 \in \mathcal{R}$, give rise to the set of functions $\mathcal{F}$ that we are entertaining. While this set of functions seems quite simple, just linear, the power of $\mathcal{F}$ is hidden in the feature vectors. Indeed, we can often construct different feature representations for objects. For example, there are many ways to map past values of financial assets into a feature vector x, and this mapping is typically completely under our control. This "freedom" gives us a lot of power, and we will discuss how to exploit it later on. For now, we assume that a proper representation has been found, denoting feature vectors simply as x.

Our learning task is to choose one $f \in \mathcal{F}$, i.e., choose parameters $\hat{\theta}$ and $\hat{\theta}_0$, based on the training set $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$, where $y^{(t)} \in \mathcal{R}$ (response). As before, our goal is to find $f(x;\hat{\theta},\hat{\theta}_0)$ that would yield accurate predictions on yet unseen examples. There are several problems to address:

(1) How do we measure error? What is the criterion by which we choose $\hat{\theta}$ and $\hat{\theta}_0$ based on the training set?

(2) What algorithm can we use to optimize the training criterion? How does the algorithm scale with the dimension (feature vectors may be high dimensional) or the size of the training set (the dataset may be large)?

(3) When the size of the training set is not large enough in relation to the number of parameters (dimension) there may be degrees of freedom, i.e., directions in the parameter space, that remain unconstrained by the data. How do we set those degrees of freedom? This is a part of a broader problem known as *regularization*. The question is how to softly constrain the set of functions $\mathcal{F}$ to achieve better generalization.

(1) Empirical risk and the least squares criterion

As in the classification setting, we will measure training error in terms of *empirical risk*

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) \tag{33}$$

where, for simplicity, we have dropped the parameter θ_0. It will be easy to add it later on in the appropriate place. Note that, unlike in classification, the loss function now depends on the difference between the real valued target value $y^{(t)}$ and the corresponding linear prediction $\theta \cdot x^{(t)}$. There are many possible ways of defining the loss function. We will use here a simple squared error: $\text{Loss}(z) = z^2/2$. The idea is to permit small discrepancies (we expect the responses to include noise) but heavily penalize large deviations (that typically indicate poor parameter choices). As a result, we have

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) = \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{34}$$

Our learning goal is not to minimize $R_n(\theta)$; it is just the best we can do (for now). Minimizing $R_n(\theta)$ is a surrogate criterion since we don't have a direct access to the test or generalization error

$$R_{n'}^{test}(\theta) = \frac{1}{n'}\sum_{t=n+1}^{n+n'} (y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{35}$$

Let's briefly consider how $R_n(\theta)$ and $R_{n'}^{test}(\theta)$ are related. If we select $\hat{\theta}$ by minimizing $R_n(\theta)$, our performance will be measured according to $R_{n'}^{test}(\hat{\theta})$. This test error can be large for two different reasons. First, we may have a large *estimation error*. This means that, even if the true relationship between x and y is linear, it is hard for us to estimate it on the basis of a small (and potentially noisy) training set S_n. Our estimated parameters $\hat{\theta}$ will not be entirely correct. The larger the training set is, the smaller the estimation error will be. The second type of error on the test set is *structural error*. This means that we may be estimating a linear mapping from x to y when the true underlying relationship is highly non-linear. Clearly, we cannot do very well in this case, regardless of how large the training set is. In order to reduce structural error, we would have to use a larger set of functions $\mathcal{F}$. But, given a noisy training set S_n, it will be harder to select the correct function from such larger $\mathcal{F}$, and our estimation error will increase. Finding the balance between the estimation and structural errors lies at the heart of learning problems.

When we formulate linear regression problem as a statistical problem, we can talk about the structural error as “bias”, while the estimation error corresponds to the “variance” of the *estimator* $\hat{\theta}(S_n)$. The parameters $\hat{\theta}$ are obtained with the help of training data S_n and thus can be viewed as functions of S_n. An estimator is a mapping from data to parameters.

(2) Optimizing the least squares criterion

Perhaps the simplest way to optimize the least squares objective $R_n(\theta)$ is to use the stochastic gradient descent method discussed earlier in the classification context. Our case here is easier, in fact, since $R_n(\theta)$ is everywhere differentiable. At each step of the algorithm, we select one training example at random, and nudge parameters in the opposite direction of the gradient

$$\nabla_\theta(y^{(t)} - \theta \cdot x^{(t)})^2/2 = (y^{(t)} - \theta \cdot x^{(t)})\nabla_\theta(y^{(t)} - \theta \cdot x^{(t)}) = -(y^{(t)} - \theta \cdot x^{(t)})x^{(t)} \tag{36}$$

As a result, the algorithm can be written as

```
set $\theta^{(0)} = 0$
    randomly select $t \in \{1, \ldots, n\}$                                    (37)
        $\theta^{(k+1)} = \theta^{(k)} + \eta_k(y^{(t)} - \theta \cdot x^{(t)})x^{(t)}$
```

where η_k is the learning rate (e.g., $\eta_k = 1/(k+1)$). Recall that in classification the update was performed only if we made a mistake. Now the update is proportional to discrepancy $(y^{(t)} - \theta \cdot x^{(t)})$ so that even small mistakes count, just less. As in the classification context, the update is "self-correcting". For example, if our prediction is lower than the target, i.e., $y^{(t)} > \theta \cdot x^{(t)}$, we would move the parameter vector in the positive direction of $x^{(t)}$ so as to increase the prediction next time $x^{(t)}$ is considered. This would happen in the absence of updates based on other examples.

Closed form solution

We can also try to minimize $R_n(\theta)$ directly by setting the gradient to zero. Indeed, since $R_n(\theta)$ is a convex function of the parameters, the minimum value is obtained at a point (or a set of points) where the gradient is zero. So, formally, we find $\hat{\theta}$ for which $\nabla R_n(\theta)_{\theta=\hat{\theta}} = 0$. More specifically,

$$\begin{aligned}
\nabla R_n(\theta)_{\theta=\hat{\theta}} &= \frac{1}{n}\sum_{t=1}^{n} \nabla_\theta \{ \, (y^{(t)} - \theta \cdot x^{(t)})^2/2 \, \}_{|\theta=\hat{\theta}} && (38) \\
&= \frac{1}{n}\sum_{t=1}^{n} \{ \, -(y^{(t)} - \hat{\theta} \cdot x^{(t)}) x^{(t)} \, \} && (39) \\
&= -\frac{1}{n}\sum_{t=1}^{n} y^{(t)} x^{(t)} + \frac{1}{n}\sum_{t=1}^{n} (\hat{\theta} \cdot x^{(t)}) x^{(t)} && (40) \\
&= \underbrace{-\frac{1}{n}\sum_{t=1}^{n} y^{(t)} x^{(t)}}_{=b} + \underbrace{\frac{1}{n}\sum_{t=1}^{n} x^{(t)} (x^{(t)})^T}_{=A} \, \hat{\theta} && (41) \\
&= -b + A\hat{\theta} = 0 && (42)
\end{aligned}$$

where we have used the fact that $\hat{\theta} \cdot x^{(t)}$ is a scalar and can be moved to the right of vector $x^{(t)}$. We have also subsequently rewritten the inner product as $\hat{\theta} \cdot x^{(t)} = (x^{(t)})^T \hat{\theta}$. As a result, the equation for the parameters can be expressed in terms of a $d \times 1$ column vector b and a $d \times d$ matrix A as $A\theta = b$. When the matrix A is invertible, we can solve for the parameters directly: $\hat{\theta} = A^{-1} b$. In order for A to be invertible, the training points $x^{(1)}, \ldots, x^{(n)}$ must *span* $\mathcal{R}^d$. Naturally, this can happen only if $n \geq d$, and is therefore more likely to be the case when the dimension d is small in relation to the size of the training set n. Another consideration is the cost of actually inverting A. Roughly speaking, you will need $\mathcal{O}(d^3)$ operations for this. If $d = 10,000$, this can take a while, making the stochastic gradient updates more attractive.

In solving linear regression problems, the matrix A and vector b are often written in a slightly different way. Specifically, define $X = [x^{(1)}, \ldots, x^{(n)}]^T$. In other words, X^T has each training feature vector as a column; X has them stacked as rows. If we also define $\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^T$ (column vector), then you can easily verify that

$$b = \frac{1}{n} X^T \vec{y}, \;\; A = \frac{1}{n} X^T X \qquad (43)$$

(3) Regularization

What happens when A is not invertible? In this case the training data provide no guidance about how to set some of the parameter directions. In other words, the learning problem is ill-posed. The same issue inflicts the stochastic gradient method as well though the initialization $\theta^{(0)} = 0$ helps set the parameters to zero for directions outside the span of the training examples. The simple fix does not solve the broader problem, however. How should we set the parameters when we have insufficient training data?

We will modify the estimation criterion, the mean squared error, by adding a *regularization term*. The purpose of this term is to bias the parameters towards a default answer such as zero. The regularization term will "resist" setting parameters away from zero, even when the training data may weakly tell us otherwise. This resistance is very helpful in order to ensure that our predictions generalize well. The intuition is that we opt for the "simplest answer" when the evidence is absent or weak.

There are many possible regularization terms that fit the above description. In order to keep the resulting optimization problem easily solvable, we will use $\|\theta\|^2/2$ as the penalty. Specifically, we will minimize

$$J_{n,\lambda}(\theta) = \frac{\lambda}{2}\|\theta\|^2 + R_n(\theta) = \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{44}$$

where the *regularization parameter* $\lambda \geq 0$ quantifies the trade-off between keeping the parameters small – minimizing the squared norm $\|\theta\|^2/2$ – and fitting to the training data – minimizing the empirical risk $R_n(\theta)$. The use of this modified objective is known as *Ridge regression*.

While important, the regularization term introduces only small changes to the two estimation algorithms. For example, in the stochastic gradient descent algorithm, in each step, we will now move in the reverse direction of the gradient

$$\nabla_\theta\left\{ \frac{\lambda}{2}\|\theta\|^2 + (y^{(t)} - \theta \cdot x^{(t)})^2/2 \right\}_{|\theta=\theta^{(k)}} = \lambda\theta^{(k)} - (y^{(t)} - \theta^{(k)} \cdot x^{(t)})x^{(t)} \tag{45}$$

As a result, the algorithm can be rewritten as

```
set $\theta^{(0)} = 0$
    randomly select $t \in \{1, \ldots, n\}$
        $\theta^{(k+1)} = (1 - \lambda\eta_k)\theta^{(k)} + \eta_k(y^{(t)} - \theta \cdot x^{(t)})x^{(t)}$        (46)
```

As you might expect, there's now a new factor $(1 - \lambda\eta_k)$ multiplying the current parameters $\theta^{(k)}$, shrinking them towards zero during each update.

When solving for the parameters directly, the regularization term only modifies the $d \times d$ matrix $A = \lambda I + (1/n)\, X^T X$, where I is the identity matrix. The resulting matrix is *always* invertible so long as $\lambda > 0$. The cost of inverting it remains the same, however.

The effect of regularization

The regularization term shifts emphasis away from the training data. As a result, we should expect that larger values of λ will have a negative impact on the training error. Specifically, let $\hat{\theta} = \hat{\theta}(\lambda)$ denote the parameters that we would find by minimizing the regularized objective $J_{n,\lambda}(\theta)$. We view $\hat{\theta}(\lambda)$ here as a function of λ. We claim then that $R_n(\hat{\theta}(\lambda))$, i.e., mean squared training error, increases as λ increases. If the training error increases, where's the benefit? Larger values of λ actually often lead to lower generalization error as we are no longer easily swayed by noisy data. Put another way, it becomes harder to over-fit to the training data. This benefit accrues for a while as λ increases, then turns to hurt us. Biasing the parameters towards zero too strongly, even when the data tells us otherwise, will eventually hurt generalization performance. As a result, you will see a typical U-shaped curve in terms of how the generalization error depends on the regularization parameter λ.

5 Support Vector Machines and Kernels

Maximum margin separator

Let's begin by assuming that the training examples are linearly separable. Clearly, in this case there are many linear separators that can perfectly classify the training examples. For example, if we run the perceptron algorithm with two different initializations, or cycle through the training examples in a slightly different order, we would potentially arrive at different classifiers (linear separators). Here we are trying to explicitly find a single linear separator that is "optimal" in some sense.

If we imagine that (yet unseen) test examples are noisy versions of the training examples, then it seems sensible to draw the linear decision boundary in a manner that a) classifies all the training examples correctly, and b) is maximally removed from all the training examples. This is known as the *maximum margin separator* or the *optimal hyperplane.*

We now turn to the question of how to find the maximum margin separator. Recall that in the context of the passive-aggressive algorithm, we sought to find a linear classifier that approximately minimizes the Hinge loss on the training examples (rather than just classifying them correctly). Zero Hinge loss on the training set means that all the examples are on the correct sides of the margin boundaries $\{x : \theta \cdot x + \theta_0 = 1\}$ and $\{x : \theta \cdot x + \theta_0 = -1\}$. Figure 7a) below shows an example of such a linear classifier. The margin boundaries are exactly $1/\|\theta\|$ distance away from the decision boundary. We call $1/\|\theta\|$ the *margin.* Note that the margin depends on $\|\theta\|$ and we will make use of this fact here. As discussed earlier, the distance of each training example $(x^{(t)}, y^{(t)})$ from the decision boundary is given by

$$\frac{y^{(t)}(\theta \cdot x^{(t)} + \theta_0)}{\|\theta\|} \tag{47}$$

So, one way to find the maximum margin separator is to maximize the margin $1/\|\theta\|$ while keeping all the examples beyond the margin boundaries. In other words, we can

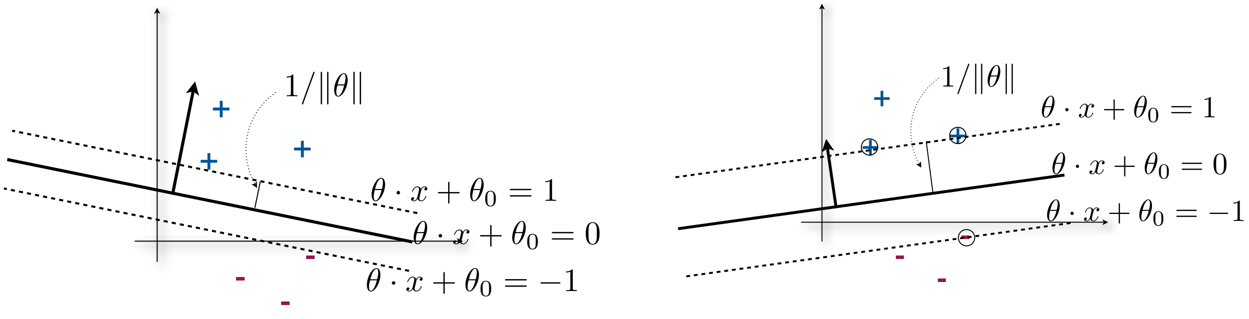
$$\max \frac{1}{\|\theta\|} \;\; \text{subject to} \;\; \frac{y^{(t)}(\theta \cdot x^{(t)} + \theta_0)}{\|\theta\|} \geq \frac{1}{\|\theta\|}, \; t = 1, \ldots, n \tag{48}$$

where the maximization is with respect to both θ and θ_0. Note that the margin depends implicitly on θ_0 since θ_0 affects the location of the decision boundary. As we push the margin boundaries further than further away from the decision boundary, there will be a point where we can no longer increase the margin, however we would draw the decision boundary. At this point, the solution "locks in place" as shown in Figure 7b). The resulting solution is the maximum margin separator.

We can simplify the optimization problem a bit more. Instead of maximizing $1/\|\theta\|$ we can minimize $\|\theta\|$ or $\|\theta\|^2/2$ as it will be more convenient to do. We can also multiply both sides of the constraints in Eq.(48) by $\|\theta\|$. As a result, we get a simpler optimization problem for the maximum margin separator.

$$\min \frac{1}{2}\|\theta\|^2 \;\; \text{subject to} \;\; y^{(t)}(\theta \cdot x^{(t)} + \theta_0) \geq 1, \;\; t = 1, \ldots, n \tag{49}$$

Using this quadratic programming problem for solving the maximum margin separator is known as the *support vector machine.* If the examples are linearly separable, and we have at least one positive and one negative example, then the solution is unique. Moreover, as you can see in Figure 7b), the margin boundaries touch only a small subset of examples (circled in the figure). These examples that are right on the margin boundaries are called *support vectors*. They are also the only examples we would need to obtain the maximum margin separator (it didn't matter where the rest of them were so long as they are beyond the margin boundaries).


$1/\|\theta\|$
$\theta \cdot x + \theta_0 = 1$
$\theta \cdot x + \theta_0 = 0$
$\theta \cdot x + \theta_0 = -1$
+
-
$1/\|\theta\|$
$\theta \cdot x + \theta_0 = 1$
$\theta \cdot x + \theta_0 = 0$
$\theta \cdot x + \theta_0 = -1$
+
-

a)

b)

Figure 7: a) a linear separator with margin boundaries that satisfy the classification constraints. b) the maximum margin separator

The maximum margin linear separator can be easily affected by a single misclassified example (do you see why?). For this reason, and to be able to use it even in case of non-separable data, it is good to be able to "give up" on trying to satisfy all the classification constraints. Indeed, we can introduce "slack variables" $\xi_t \geq 0$, $t = 1, \ldots, n$, that measure how much each constraints are violated, and try to minimize the amount of violation. The overall optimization problem is then

$$\min \frac{1}{2}\|\theta\|^2 + C\sum_{t=1}^{n}\xi_t \;\text{ subject to }\; y^{(t)}(\theta \cdot x^{(t)} + \theta_0) \geq 1 - \xi_t,\;\; \xi_t \geq 0,\;\; t = 1, \ldots, n \tag{50}$$

where larger values of C enforce the constraints more strongly (if possible). See Figure 8 below. Note that the margin is larger for smaller C as pushing the margin boundaries past the points is penalized less in the objective.

While the maximum margin separator is well-motivated as a linear classifier, solving the associated quadratic programming (QP) problem, with or without slack, can be challenging in practice. In many cases the dimension d and/or the number of training examples n is large. The time to solve the QP can easily scale as n^3. In contrast, the passive-aggressive algorithm performs a few linear scans over the training set. It's training time therefore scales with n rather than n^3.

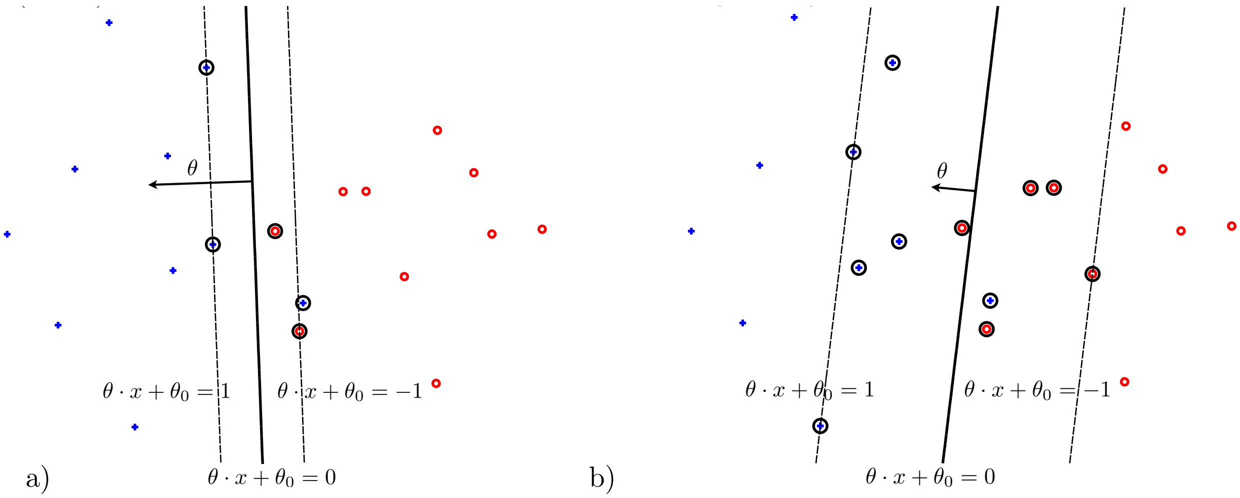


Figure 8: a) SVM solution for large C (few violations) b) small C (many violations)

Non-linear classifiers – kernels

Let's start by considering how we can use linear classifiers to make non-linear predictions. The easiest way to do this is to first map all the examples $x \in \mathcal{R}^d$ to different feature vectors $\phi(x) \in \mathcal{R}^p$ where typically p is much larger than d. We would then simply use a linear classifier on the new (higher dimensional) feature vectors, pretending that they were the original input vectors. As a result, all the linear classifiers we have learned remain applicable, yet produce non-linear classifiers in the original coordinates.

There are many ways to create such feature vectors. For example, we can build $\phi(x)$ by concatenating polynomial terms of the original coordinates. For example, in two dimensions, we could map $x = [x_1, x_2]^T$ to a five dimensional feature vector

$$\phi(x) = [x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2]^T \tag{51}$$

We can then train a "linear" classifier, linear in the new ϕ-coordinates,

$$y = \text{sign}(\theta \cdot \phi(x) + \theta_0) \tag{52}$$

by mapping each training example $x^{(t)}$ to the corresponding feature vector $\phi(x^{(t)})$. In other words, our training set is now $S_n^\phi = \{(\phi(x^{(t)}), y^{(t)}), t = 1, \ldots, n\}$. The resulting parameter estimates $\hat{\theta}$, $\hat{\theta}_0$ define a linear decision boundary in the ϕ-coordinates but a non-linear boundary in the original x-coordinates

$$\hat{\theta} \cdot \phi(x) + \hat{\theta}_0 = 0 \quad \Leftrightarrow \quad \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \hat{\theta}_3 \sqrt{2} x_1 x_2 + \hat{\theta}_4 x_1^2 + \hat{\theta}_5 x_2^2 + \hat{\theta}_0 = 0 \tag{53}$$

Such a non-linear boundary can represent, e.g., an ellipse in the original two dimensional space.

The main problem with the above procedure is that the feature vectors $\phi(x)$ can become quite high dimensional. For example, if we start with $x \in \mathcal{R}^d$, where $d = 1000$, then compiling $\phi(x)$ by concatenating polynomial terms up to the 2nd order would have dimension $d+d(d+1)/2$ or about $500,000$. Using higher order polynomial terms would become quickly infeasible. However, it may still be possible to *implicitly* use such feature vectors. If our training and prediction problems can be formulated only in terms of inner products between the examples, then the relevant computation for us is $\phi(x) \cdot \phi(x')$. Depending on how we define $\phi(x)$, this computation may be possible to do efficiently even if using $\phi(x)$ explicitly is not. For example, when $\phi(x) = [x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2]^T$, we see that (please check!)

$$\phi(x) \cdot \phi(x') = (x \cdot x') + (x \cdot x')^2 \tag{54}$$

So the inner product is calculated easily on the basis of the original input vectors without having to explicitly represent $\phi(x)$. We will try to turn our linear classifiers into a form that relies only on the inner products between the examples.

Kernel methods

We have discussed several linear prediction methods such as the perceptron algorithm, passive-aggressive algorithm, support vector machine, as well as linear (Ridge) regression. All these methods can be transformed into non-linear methods by mapping examples $x \in \mathcal{R}^d$ into feature vectors $\phi(x) \in \mathcal{R}^p$. Typically $p > d$ and $\phi(x)$ is constructed from x by appending polynomial (or other non-linear) terms involving the coordinates of x such as x_ix_j, x_i^2, and so on. The resulting predictors

$$\begin{aligned} \text{Perceptron :} \quad & y = \text{sign}\big(\,\theta \cdot \phi(x) + \theta_0\big) && (55)\\ \text{SVM :} \quad & y = \text{sign}\big(\,\theta \cdot \phi(x) + \theta_0\big) && (56)\\ \text{Linear regression :} \quad & y = \theta \cdot \phi(x) + \theta_0 && (57) \end{aligned}$$

differ from each other based on how they are trained in response to (expanded) training examples $S_n = \{(\phi(x^{(t)}), y^{(t)}), t = 1, \ldots, n\}$. In other words, the estimated parameters $\hat{\theta}$ and $\hat{\theta}_0$ will be different in the three cases even if they were all trained based on the same data. Note that, in the regression case, the responses $y^{(t)}$ are typically not binary labels. However, there's no problem applying the linear regression method even if the training labels are all ±1.

Kernel perceptron

We can run the perceptron algorithm (until convergence) when the training examples are linearly separable in the given feature representation. Recall that the algorithm is given by

(0) Initialize: $\theta = 0$ (vector), $\theta_0 = 0$

(1) Cycle through the training examples $t = 1, \ldots, n$

If $y^{(t)}(\theta \cdot \phi(x^{(t)}) + \theta_0) \leq 0$ (mistake)
then $\theta \leftarrow \theta + y^{(t)}\phi(x^{(t)})$ and $\theta_0 \leftarrow \theta_0 + y^{(t)}$

It is clear from this description that the parameters θ and θ_0 at any point in the algorithm can be written as

$$\theta = \sum_{i=1}^{n} \alpha_i y^{(i)} \phi(x^{(i)}) \tag{58}$$

$$\theta_0 = \sum_{i=1}^{n} \alpha_i y^{(i)} \tag{59}$$

where α_i is the number of times that we have made a mistake on the corresponding training example $(\phi(x^{(i)}), y^{(i)})$. Our goal here is to rewrite the algorithm so that we just update α_i's, never explicitly constructing θ which may be high dimensional. To this end, note that

$$\theta \cdot \phi(x) = \sum_{i=1}^{n} \alpha_i y^{(i)} (\phi(x^{(i)}) \cdot \phi(x)) = \sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x) \tag{60}$$

where the inner product $K(x^{(i)}, x) = \phi(x^{(i)}) \cdot \phi(x)$ is known as the *kernel function*. It is a function of two arguments, and is always defined as the inner product of feature vectors corresponding to the input arguments. Indeed, we say that a kernel function is *valid* if there is some feature vector $\phi(x)$ (possibly infinite dimensional!) such that $K(x, x') = \phi(x) \cdot \phi(x')$ for all x and x'. Can you think of some function of two arguments which is not a kernel in this sense?

We want the perceptron algorithm to use only kernel values – comparisons between examples – rather than feature vectors directly. To this end, we will write the discriminant function $\theta \cdot \phi(x) + \theta_0$ solely in terms of the kernel function and α's

$$\theta \cdot \phi(x) + \theta_0 = \sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x) + \sum_{i=1}^{n} \alpha_i y^{(i)} = \sum_{i=1}^{n} \alpha_i y^{(i)} [K(x^{(i)}, x) + 1] \tag{61}$$

This is all that we need for prediction or for assessing whether there was a mistake on a particular training example $(\phi(x^{(t)}), y^{(t)})$. We can therefore write the algorithm just in terms of α's, updating them in response to each mistake. The resulting *kernel perceptron* algorithm is given by

```
Initialize: $\alpha_t = 0$, $t = 1, \ldots, n$

Cycle through training examples $t = 1, \ldots, n$
    If $y^{(t)}(\sum_{i=1}^{n} \alpha_i y^{(i)} [K(x^{(i)}, x^{(t)}) + 1]) \leq 0$ (mistake)
    then $\alpha_t \leftarrow \alpha_t + 1$
```

Note that the algorithm can be run with any valid kernel function $K(x, x')$. Also, typically only a few of the counts α_i will be non-zero. This means that only a few of the training examples are relevant for finding a separating solution, the rest of the counts α_i remain exactly at zero. So, just as with support vector machines, the solution can be quite *sparse*. Figure 9 shows the decision boundary (compare to Eq.(61) above)

$$\sum_{i=1}^{n} \alpha_i y^{(i)} [K(x^{(i)}, x) + 1] = 0 \tag{62}$$

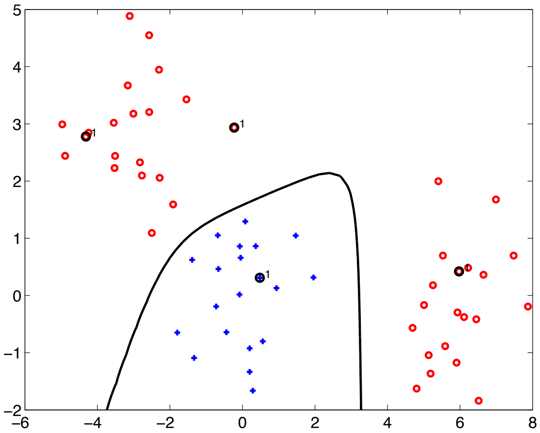


Figure 9: Kernel perceptron example. We have circled the four training examples that the algorithm makes a mistake on.

resulting from running the kernel perceptron algorithm with the radial basis kernel (see later for why this is a valid kernel)

$$K(x, x') = \exp(-\|x - x'\|^2/2) \tag{63}$$

The algorithm makes only four mistakes until a separating solution is found. In fact, with this kernel, the perceptron algorithm can perfectly classify any set of distinct training examples!

Kernel linear regression

For simplicity, let's consider here the case where $\theta_0 = 0$, i.e., that the regression function we wish to estimate is given by $\theta \cdot \phi(x)$. With this change, the estimation criterion for parameters θ was given by

$$J(\theta) = \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot \phi(x^{(t)}))^2/2 + \frac{\lambda}{2}\|\theta\|^2 \tag{64}$$

This is a convex function (bowl-shaped) and thus the minimum is obtained at the point where the gradient is zero. To this end,

$$\begin{aligned}\frac{\partial}{\partial\theta}J(\theta) &= -\frac{1}{n}\sum_{t=1}^{n}\underbrace{(y^{(t)} - \theta \cdot \phi(x^{(t)}))}_{=n\lambda\,\alpha_t}\phi(x^{(t)}) + \lambda\theta && (65)\\ &= -\lambda\sum_{t=1}^{n}\alpha_t\phi(x^{(t)}) + \lambda\theta = 0 && (66)\end{aligned}$$

or, equivalently, that $\theta = \sum_{t=1}^{n}\alpha_t\phi(x^{(t)})$, where α_t are proportional to prediction errors (recall that α's were related to errors in the perceptron algorithm). The above equation holds only if α_t and θ relate to each other in a specific way, i.e., only if

$$\begin{aligned} n\lambda\,\alpha_t &= y^{(t)} - \theta \cdot \phi(x^{(t)}) && (67) \\ &= y^{(t)} - \left(\sum_{i=1}^n \alpha_i \phi(x^{(i)})\right) \cdot \phi(x^{(t)}) && (68) \\ &= y^{(t)} - \sum_{i=1}^n \alpha_i K(x^{(i)}, x^{(t)}) && (69) \end{aligned}$$

which should hold for all $t = 1, \ldots, n$. Let's write the equation in a vector form. $\vec{\alpha} = [\alpha_1, \ldots, \alpha_n]^T$, $\vec{y} = [y^{(1)}, \ldots, y^{(n)}]^T$, and because $K(x^{(i)}, x^{(t)}) = K(x^{(t)}, x^{(i)})$ (inner product is symmetric)

$$\sum_{i=1}^n \alpha_i K(x^{(i)}, x^{(t)}) = \sum_{i=1}^n K(x^{(t)}, x^{(i)})\alpha_i = [K\vec{\alpha}]_t \tag{70}$$

where K is a $n \times n$ matrix whose i, j element is $K(x^{(i)}, x^{(j)})$ (a.k.a. the Gram matrix). Now, in a vector form, we have

$$n\lambda\,\vec{\alpha} = \vec{y} - K\vec{\alpha} \quad \text{or} \quad (n\lambda\, I + K)\vec{\alpha} = \vec{y} \tag{71}$$

The solution is simply $\vec{\alpha} = (n\lambda\, I + K)^{-1}\vec{y}$ (always invertible for $\lambda > 0$). In other words, estimated coefficients $\hat{\alpha}_t$ can be computed only in terms of the kernel function and the target responses, never needing to explicitly construct feature vectors $\phi(x)$. Once we have the coefficients, prediction for a new point x is similarly easy

$$\hat{\theta} \cdot \phi(x) = \sum_{i=1}^n \hat{\alpha}_i \phi(x^{(i)}) \cdot \phi(x) = \sum_{i=1}^n \hat{\alpha}_i K(x^{(i)}, x) \tag{72}$$

Figure 10 below shows examples of one dimensional regression problems with higher order polynomial kernels. Which of these kernels should we use? (the right answer is linear; this is how the data was generated, with noise). The general problem selecting the kernel (or the corresponding feature representation) is a *model selection* problem. We can always try to use cross-validation as a model selection criterion.

Kernel functions

All of the methods discussed above can be run with any valid kernel function $K(x, x')$. A kernel function is valid if and only if there exists some feature mapping $\phi(x)$ such that $K(x, x') = \phi(x) \cdot \phi(x')$. We don't need to know what $\phi(x)$ is (necessarily), only that one exists. We can build many common kernel functions based only on the following four rules

(1) $K(x, x') = 1$ is a kernel function.

(2) Let $f : \mathcal{R}^d \to \mathcal{R}$ be any real valued function of x. Then, if $K(x, x')$ is a kernel function, then so is $\tilde{K}(x, x') = f(x)K(x, x')f(x')$

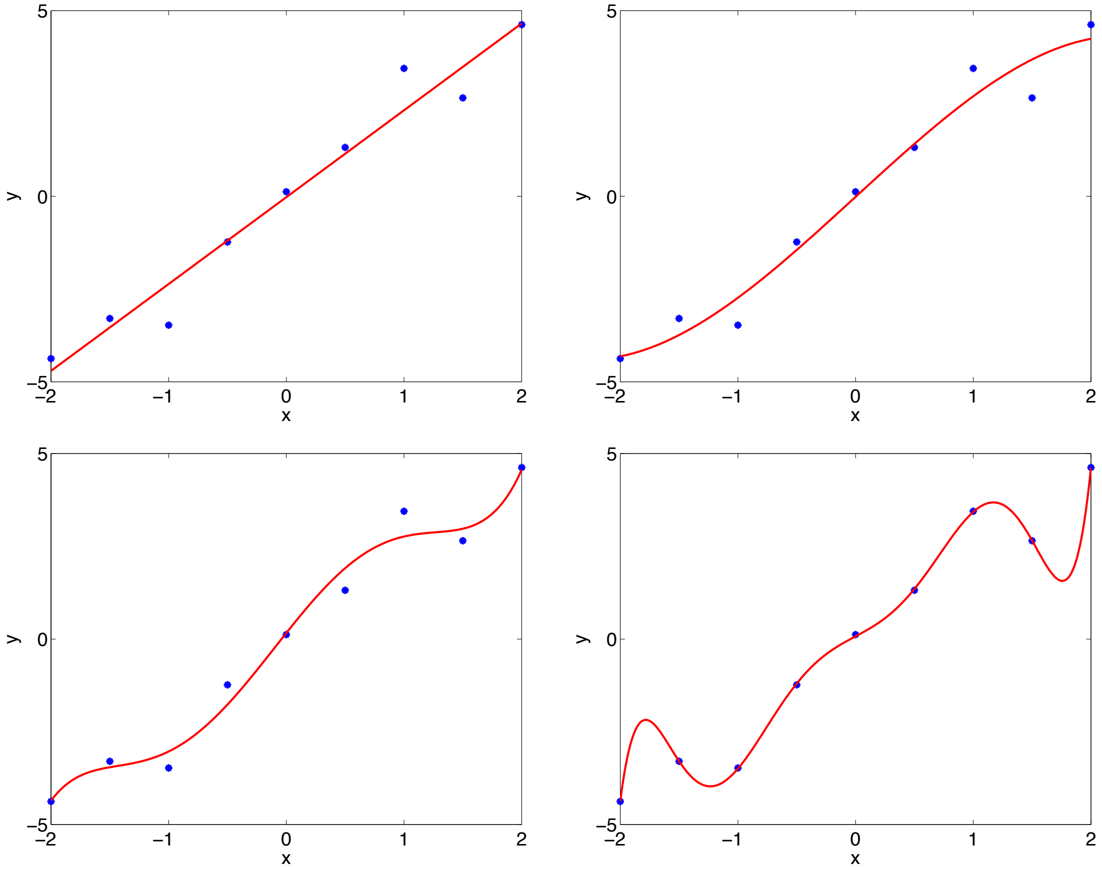


Figure 10: Kernel regression with a linear kernel (top left), 3rd order polynomial kernel (top right), 5th order polynomial kernel (bottow left), and a 7th order polynomial kernel (bottom right).

(3) If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then so is their sum. In other words, $K(x, x') = K_1(x, x') + K_2(x, x')$ is a kernel.

(4) If $K_1(x, x')$ and $K_2(x, x')$ are kernels, then so is their product $K(x, x') = K_1(x, x')K_2(x, x')$

To understand these composition rules, let's figure out how they relate to the underlying feature mappings. For example, a constant kernel $K(x, x') = 1$ simply corresponds to $\phi(x) = 1$ for all $x \in \mathcal{R}^d$. Similarly, if $\phi(x)$ is the feature mapping for kernel $K(x, x')$, then $\tilde{K}(x, x') = f(x)K(x, x')f(x')$ (rule 2) corresponds to $\tilde{\phi}(x) = f(x)\phi(x)$. Adding kernels means appending feature vectors. For example, let's say that $K_1(x, x')$ and $K_2(x, x')$ correspond to feature mappings $\phi^{(1)}(x)$ and $\phi^{(2)}(x)$, respectively. Then (see rule 3)

$$\begin{aligned} K(x, x') &= \begin{bmatrix} \phi^{(1)}(x) \\ \phi^{(2)}(x) \end{bmatrix} \cdot \begin{bmatrix} \phi^{(1)}(x) \\ \phi^{(2)}(x) \end{bmatrix} && (73) \\ &= \phi^{(1)}(x) \cdot \phi^{(1)}(x') + \phi^{(2)}(x) \cdot \phi^{(2)}(x') && (74) \\ &= K_1(x, x') + K_2(x, x') && (75) \end{aligned}$$

Can you figure out what the feature mapping is for $K(x, x')$ in rule 4, expressed in terms of the feature mappings for $K_1(x, x')$ and $K_2(x, x')$?

Many typical kernels can be constructed on the basis of these rules. For example, $K(x, x') = x \cdot x'$ is a kernel based on rules (1), (2), and (3). To see this, let $f_i(x) = x_i$ (i^{th} coordinate mapping), then

$$x \cdot x' = x_1 x_1' + \ldots + x_d x_d' = f_1(x)\, 1\, f_1(x') + \cdots + f_d(x)\, 1\, f_d(x') \tag{76}$$

where each term uses rules (1) and (2), and the addition follows from rule (3). Similarly, the 2nd order polynomial kernel

$$K(x, x') = (x \cdot x') + (x \cdot x')^2 \tag{77}$$

can be built from assuming that $(x \cdot x')$ is a kernel, using the product rule to realize the 2nd term, i.e., $(x \cdot x')^2 = (x \cdot x')(x \cdot x')$, and finally adding the two. More interestingly,

$$K(x, x') = \exp(x \cdot x') = 1 + (x \cdot x') + \frac{1}{2!}(x \cdot x')^2 + \cdots \tag{78}$$

is also a kernel by the same rules. But, since the expansion is an infinite sum, the resulting feature representation for $K(x, x')$ is infinite dimensional! This is also why the radial basis kernel has an infinite dimensional feature representation. Specifically,

$$\begin{aligned} K(x, x') &= \exp(-\|x - x'\|^2/2) && (79) \\ &= \exp(-\|x\|^2/2) \exp(x \cdot x') \exp(-\|x'\|^2/2) && (80) \\ &= f(x) \exp(x \cdot x') f(x') && (81) \end{aligned}$$

where $f(x) = \exp(-\|x\|^2/2)$. The radial basis kernel is special in many ways. For example, any distinct set of training examples are always perfectly separable with the radial basis kernel[1]. In other words, running the perceptron algorithm with the radial basis kernel will always converge to a separable solution provided that the training examples are all distinct.

[1] Follows from a Michelli theorem about monotone functions of distance and the invertibility of the corresponding Gram matrices; theorem not shown

6 Ensembles and Boosting

Ensembles

We can combine any set of classifiers into an ensemble where each classifier in the ensemble votes for one label versus another. Ensembles can be useful even if they were generated through randomization, i.e., through little variations on how the same type of classifier is trained. For example, we can generate random subsets of smaller training sets from the original one, and train a classifier, e.g., a perceptron, based on each such set. The outputs of these classifiers, trained with slightly different training sets, can be then combined into an ensemble classifier where each perceptron is given an equal vote. The ensemble provides more stable predictions (only the majority has to be correct) and therefore often generalizes better. Weighting each perceptron output equally in the ensemble, regardless of how well they solve their corresponding subtasks, does not not lead to a more complex classifier (closer to the true underlying relation), just to one that is more stable.

Here we will explore methods such as *boosting* for combining simple base classifiers into a stronger (more complex) ensemble. The ensemble has potential to give rise to an arbitrary complex decision boundary, yielding a stronger classifier than any of the weak learners used to construct it. To achieve this, we need to design an algorithm that carefully selects weak learners so that they compensate for each other's weakness. Note that the process of combining simple "weak" classifiers into one "strong" classifier is analogous to the use of kernels to go from a simple linear classifier to a non-linear classifier. The difference is that here we are *learning* a small number of highly *non-linear features* from the inputs rather than using a fixed process of generating a large number of features from the inputs as in the polynomial kernel.

Decision Stumps

The boosting algorithm we will cover makes use of simple weak classifiers, called *decision stumps*. A decision stump is a linear classifier that relies only on a single coordinate k, and is defined as:

$$h(x; \theta) = sign(s(x_k - \mu))$$

Each stump is parametrized by $\theta = (k, s, l)$, where k is the chosen coordinate to rely on, s is the direction (which side is positive) and μ is the location of the boundary along the chosen coordinate. Figure 11 shows an example of a stump with $\theta = (2, -1, \mu)$ in two dimensions. Given the simplicity of this classifier, it is not surprising that, on its own, it cannot separate examples in our training set. However, it is very easy to train. Remember we are looking at a single coordinate. Our stump learning algorithm follows the following steps:

1. Consider each coordinate in turn

2. Consider n possible locations along the coordinate (e.g., place μ in between every pair of adjacent points along the chosen coordinate, or outside all the points).

3. Compute the number of errors for each split (μ) and orientation (s).

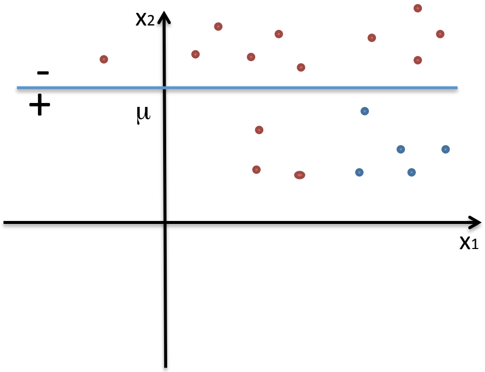


Figure 11: A possible decision boundary from a trained decision stump $\theta = (2, -1, \mu)$. The stump in the figure depends only on the vertical x_2-axis.

4. Select the classifier with the smallest number (weight) of errors.

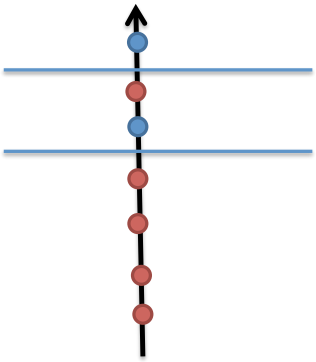


Figure 12: Two possible decision stumps applied to the training set. Each one of them obtains one mistake on the training set.

Figure 12 shows a set of training points, along with two possible stumps that minimize the error on this training set. Note that each of the candidates misclassifies exactly one point. On this dataset, we cannot find a stump which achieves better classification.

In previous lectures, we assumed that all the training examples have equal weight. However, in some scenarios we may want to associate a weight with each example. For instance, we may want to associate the second point with weight 10, while the rest of the points with the weight 1. In this case, we will prefer the first decision stump since it classifies this point correctly, yileding the loss of 1, in comparison to the second stump which gets loss of 10. Our algorithm for learning stumps can be easily applied to the setting where training instances are weighted. To this end, the third step of the algorithm for estimating a stump should use weighted errors.

Ensemble of Decision Stumps

On their own, decision stumps are indeed weak classifier. However, by combining them we can achieve much stronger predictors. Let's add a new decision stump to the example in Firgure 11. This vertical decision stump operates over x_1, and is defined as:

$$h(x;\theta_2) = sign(x_1 - \nu)$$

.

Firgure 13 shows the two stumps. Note that each stump returns -1 or $+1$. One way to combine these two stumps is to add their predictions:

$$h_2(x) = h(x;\theta_1) + h(x;\theta_2)$$

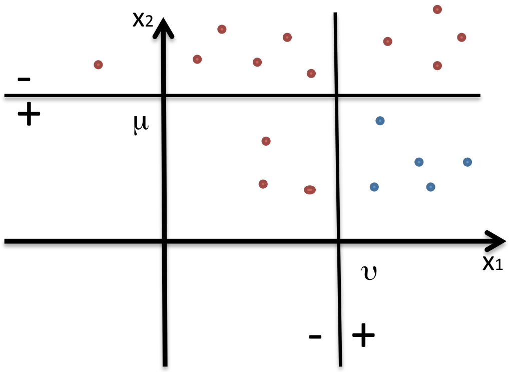


Figure 13: Two stumps applied to the same training set.

For instance, $h_2(\mu/2, \nu + 1) = 2$, since both decision stumps return $+1$ for this point. It is easy to see that our ensemble still cannot classify all the points. One of such points is $(\mu/2, \nu/2)$, where the first stump returns $+1$ and the second stump returns -1. As a result, h_2 does not give a prediction for that point.

We will further enrich our ensemble by augmenting individual decision stumps with non-negative votes:

$$h_m(x) = \sum_{j=1}^{m} \alpha_j h(x;\theta_j)$$

,

where $\alpha_j \geq 0$. In some cases, we may want to normalize the votes to sum to 1. We can always normalize the "votes" α_j such that $\sum_{j=1}^{m} \alpha_j = 1$. The ensemble can be viewed as a voting combination. Given x, each stump votes for a label $h(x;\theta_j)$ using α_j votes allocated to it. The ensemble then classifies the example according to which label receives the most votes. Note that $h_m(x) \in [-1, 1]$ whenever the votes are normalized to sum to one. So, $h_m(x) = 1$ only if all the stumps agree that the label should be $y = 1$.

The ensemble can also be viewed as a linear classifier based on a feature vector

$$\phi(x;\theta) = [h(x;\theta_1), \ldots, h(x;\theta_m)]^T \tag{82}$$

The linear parameters (the votes) we learn are constrained to be positive $\alpha_j \geq 0$. As a result, $h_m(x) = \phi(x;\theta) \cdot \alpha$, which is linear in α. However, the key difference here is that we learn both which features to include, i.e., the "coordinates" $h(x;\theta_j)$ in the feature vector, as well as how they are combined, the α's. This is a difficult learning problem to solve.

We will need a loss function, and there are multiple choices here. Following common practice in boosting, we will use the *exponential loss*:

$$\text{Loss}(y, h(x)) = \exp(-yh(vx)) \tag{83}$$

The loss is small if the ensemble classifier agrees with the label y (the stronger the agreement, the smaller the loss is). The loss is large if the ensemble strongly disagrees with the label. Note that the number of mistakes on a training set is bounded by the exponential loss on this set. Therefore, when we minimize exponential loss, we minimize an upper bound on the number of mistakes.

The simplest way to optimize the ensemble is to do it sequentially in stages. In other words, we will first find a single stump (an ensemble of one), then add another while keeping the first one fixed, and so on, never retraining those already added into the ensemble. To facilitate this type of estimation, we will assume that $\alpha_j \geq 0$ but won't require that they will sum to one (we can always renormalize the votes after having trained the ensemble).

Since our iterative algorithm considers one stamp at a time, we need to find a way to pass information about the results of previous estimations. Specifically, in finding a new stump, we should know how well the ensemble already handles various training instances. We will encode this information by augmenting training instances with weights W. These weights capture the exponential loss of the ensemble on inidivual training points.

Suppose now that we have already added $m-1$ base learners into the ensemble and call this ensemble $h_{m-1}(x)$. This part will be fixed for the purpose of adding $\alpha_m h(x;\theta_m)$. We can then try to minimize the training loss corresponding to the ensemble

$$\begin{aligned} h_m(x) &= \sum_{j=1}^{m-1} \hat{\alpha}_j h(x;\hat{\theta}_j) + \alpha_m h(x;\theta_m) && (84) \\ &= h_{m-1}(x) + \alpha_m h(x;\theta_m) && (85) \end{aligned}$$

with respect to α_m and θ_m. To this end, let's write

$$\begin{aligned} J(\alpha_m, \theta_m) &= \sum_{t=1}^{n} \text{Loss}(y_t, h_m(x_t)) && (86) \\ &= \sum_{t=1}^{n} \exp\Big(- y_t h_{m-1}(x_t) - y_t \alpha_m h(x_t;\theta_m)\Big) && (87) \\ &= \sum_{t=1}^{n} \overbrace{\exp\Big(- y_t h_{m-1}(x_t)\Big)}^{W_{m-1}(t)} \exp\Big(- y_t \alpha_m h(x_t;\theta_m)\Big) && (88) \\ &= \sum_{t=1}^{n} W_{m-1}(t) \exp\Big(- y_t \alpha_m h(x_t;\theta_m)\Big) && (89) \end{aligned}$$

In other words, for the purpose of estimating the new base learner, all we need to know from the previous ensemble are the weights $W_{m-1}(t)$ associated with the training examples. These weights are exactly the losses of the $m-1$ ensemble on each of the training example. Thus, the new base learner will be "directed" towards examples that were misclassified by the $m-1$ ensemble $h_{m-1}(x)$.

The estimation problem that couples α_m and θ_m is still a bit difficult. We will simplify this further by figuring out how to estimate θ_m first and then decide how many votes α_m we should assign to the new base learner (i.e., how much we should rely on its predictions).

We have now essentially all the components to define the *Adaboost algorithm*. We will make one modification which is that the weights will be normalized to sum to one. This is advantageous as they can become rather small in the course of adding the base learners. The normalization won't change the optimization of θ_m nor α_m. We denote the normalized weights with $\tilde{W}_{m-1}(t)$. The boosting algorithm is now defined as

(0) Set $\tilde{W}_0(t) = 1/n$ for $t = 1, \ldots, n$.

(1) At stage m, find a base learner (stump) $h(x; \hat{\theta}_m)$ that achieves the best weighted classification error (zero-one loss) on the training examples, weighted by the normalized weights $\tilde{W}_{m-1}(t)$:

$$\epsilon_m = \sum_{t=1}^{n} \tilde{W}_{m-1}(t)[[y_t \neq h(x_t; \theta_m)]] \tag{90}$$

We already know how to find the best decision stump (see Section on decision stamps). We will just try each possible coordinate, each possible split/orientation along each coordinate, and then select the stump with the smallest value of ϵ_m.

(2) Given $\hat{\theta}_m$, we set

$$\hat{\alpha}_m = 0.5 \log\left(\frac{1-\hat{\epsilon}_m}{\hat{\epsilon}_m}\right) \tag{91}$$

where $\hat{\epsilon}_m$ is the weighted error corresponding to $\hat{\theta}_m$ chosen in step (1). This $\hat{\alpha}_m$ exactly minimizes the weighted training loss (loss of the ensemble):

$$J(\alpha_m, \hat{\theta}_m) = \sum_{t=1}^{n} \tilde{W}_{m-1}(t) \exp\Big(-y_t \alpha_m h(x_t; \hat{\theta}_m)\Big) \tag{92}$$

We can show it by differentiating the loss with respect to $\hat{\alpha}_m$, and setting the resulting expression to zero.

(3) Update the weights on the training examples based on the new base learner:

$$\tilde{W}_m(t) = c_m \cdot \tilde{W}_{m-1}(t) \exp\Big(-y_t \hat{\alpha}_m h(x_t; \hat{\theta}_m)\Big) \tag{93}$$

where c_m is the normalization constant to ensure that $\tilde{W}_m(t)$ sum to one after the update. The new weights can be again interpreted as normalized losses for the new ensemble $h_m(x_t) = h_{m-1}(x_t) + \hat{\alpha}_m h(x_t; \hat{\theta}_m)$.

The Adaboost algorithm sequentially adds base learners to the ensemble so as to decrease the training loss.

7 Generalization and Model Selection

Let's define our classification task precisely. We assume that training and test examples are drawn independently at random from some fixed but unknown distribution P^* over (x, y). So, for example, each $(x^{(t)}, y^{(t)})$ in the training set $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$, is sampled from P^*. You can view P^* as a large (infinite) pool of (x, y) pairs and each (x, y), whether training or test, is simply drawn at random from this pool. We do not know P^* (though could, and will, try to estimate it later). The major assumption is that the training and test examples and labels come from the *same* underlying distribution P^*.

The training error of any classifier $h(x) \in \{-1, 1\}$ is measured, as before, as counting the number of mistakes on the training set S_n

$$\mathcal{E}_n(h) = \frac{1}{n}\sum_{t=1}^{n} [\![\, y^{(t)} h(x^{(t)}) \leq 0 \,]\!] \tag{94}$$

The test or generalization error is defined as the expected value

$$\mathcal{E}(h) = E_{(x,y)\sim P^*} [\![\, y h(x) \leq 0 \,]\!] \tag{95}$$

where you can imagine drawing (x, y) from the large "pool" P^* and averaging the results. Note that the training error will change if we measure it based on another set of n examples S'_n drawn from P^*. The generalization error does not vary for any fixed $h(x)$, however, as this error is measured across the whole pool of examples already. Also note that the generalization error $\mathcal{E}(h)$ is also the probability that $h(x)$ would misclassify a randomly drawn example from P^*.

Given a set of classifiers $\mathcal{H}$, we would like to choose h^* that minimizes the generalization error, i.e., $\mathcal{E}(h^*) = \min_{h\in\mathcal{H}} \mathcal{E}(h)$. If we had access to $\mathcal{E}(h)$, there would be no model selection problem either. We would simply select the largest set $\mathcal{H}$ so as to find a classifier that minimizes the error. But we only have access to $\hat{h}$ that minimizes the training error, $\hat{h} \in \mathrm{argmin}_{h\in H}\, \mathcal{E}_n(h)$, and still wish to guarantee that the generalization error $\mathcal{E}(\hat{h})$ is low. A large $\mathcal{H}$ can hurt us in this setup. The basic intuition is that if $\mathcal{H}$ involves too many choices, we may select $\hat{h}$ that fits the noise rather than the signal in the training set. Any characteristics of $\hat{h}$ that are based on noise will not generalize well. We must select the appropriate "model" $\mathcal{H}$.

Suppose we have sets of classifiers $\mathcal{H}_i$, $i = 1, 2, \ldots,$, ordered from simpler to complex. We assume that these sets are nested in the sense that the more complex ones always include the simpler ones, i.e., $H_1 \subseteq H_2 \subseteq H_3 \subseteq \ldots$. So, for example, if $h \in H_1$, then $h \in H_2$ as well. The nested sets ensure, for example, that the training error will go down if we adopt a more complex set of classifiers. In terms of linear classifiers,

$$H = \left\{ h:\ s.t.\ h(x) = \mathrm{sign}(\phi(x)\cdot\theta + \theta_0),\ \text{for some}\ \ \theta \in \mathcal{R}^p,\ \theta_0 \in \mathcal{R} \right\}, \tag{96}$$

the sets H_i could correspond to the degree of polynomial features in $\phi(x)$. For example, in two dimensions,

$$H_1 :\ \phi(x) = [x_1, x_2]^T \tag{97}$$

$$H_2 :\ \phi(x) = [x_1, x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2]^T \tag{98}$$

$$\ldots$$

Note that H_2 contains the first order features as well as the additional 2nd order ones. So, any $h \in H_1$ has the equivalent classifier in H_2 simply by setting the parameters corresponding to the 2nd order features to zero.

Our goal here is to find the set of classifiers $\mathcal{H}_i$ such that, if we choose $\hat{h}_i \in \mathcal{H}_i$ by minimizing the training error, $\mathcal{E}_n(\hat{h}_i)$, we obtain the best guarantee of generalization error $\mathcal{E}(\hat{h}_i)$. In other words, we select the model (set of classifiers) for which we can guarantee the best generalization error. The remaining problem is to find such generalization guarantees.

Generalization guarantees

Our goal here is to provide some generalization guarantees for classifiers chosen based on the training set S_n. We assume, for simplicity, that the training algorithm finds a classifier $\hat{h} \in \mathcal{H}$ that minimizes the training error $\mathcal{E}_n(\hat{h})$. What can we say about the resulting generalization error $\mathcal{E}(\hat{h})$? Well, since the training set S_n is a random draw from P^*, our trained classifier $\hat{h}$ also varies according to this random draw. So, the generalization error $\mathcal{E}(\hat{h})$ also varies through $\hat{h}$. We cannot simply give a absolute bound on the generalization error. We can only say that with high probability, where the probability refers to the choice of the training examples in S_n, we find $\hat{h}$ that would generalize well. After all, we might have been very unlucky with the training set, and therefore obtain a bad $\hat{h}$ that generalizes poorly. But we want the procedure to work most of the time, in all "typical" cases.

More formally, we are looking for PAC (Probably Approximately Correct) guarantees of the form: *With probability at least $1-\delta$ over the choice of the training set S_n from P^*, any classifier $\hat{h}$ that minimizes the training error $\mathcal{E}_n(\hat{h})$ has generalization error $\mathcal{E}(\hat{h}) \leq \epsilon$.* Note that the statement is always true if we set $\epsilon = 1$ (since generalization error is bounded by one). Moreover, it is easy to satisfy the statement if we increase δ, i.e., require that good classifiers are found only in response to a small fraction $1-\delta$ of possible training sets. A key task here is to find the smallest ϵ for a given δ, n, and $\mathcal{H}$. In other words, we wish to find the best guarantee of generalization (i.e., ϵ) that we can give with confidence $1-\delta$ (fraction of training sets for which the guarantee is true), the number of training examples n, and the set of classifiers $\mathcal{H}$ (in particular, some measure of the size of this set). In looking for such guarantees, we will start with a simple finite case.

A finite set of classifiers

Suppose $\mathcal{H} = \{h_1, \ldots, h_K\}$, i.e., there are at most K distinct classifiers in our set. We'd like to understand, in particular, how $|\mathcal{H}| = K$ relates to ϵ, the guarantee of generalization. To make our derivations simpler, we assume, in addition, that there exists some $h^* \in \mathcal{H}$ with zero generalization error, i.e., $\mathcal{E}(h^*) = 0$. This additional assumption is known as the *realizable* case: there exists a perfect classifier in our set, we just don't know which one.

Our derivation proceeds as follows. We will fix ϵ, $\mathcal{H}$, and n, and try to obtain the smallest δ so that the guarantee holds (recall, δ is the probability over the choice of the training set that our guarantee fails). To this end, let $h \in \mathcal{H}$ be any classifier that generalizes poorly, i.e., $\mathcal{E}(h) > \epsilon$. What is the probability that we would consider it after seeing the training set? It is the probability that this classifier makes no errors on the training set. This is at most $(1-\epsilon)^n$ since the probability that it makes an error on any example drawn from P^* is at least ϵ. But there may be many such "offending" classifiers that have high generalization error (above ϵ), yet appear good on the training set (zero training error). Clearly, there cannot be more than $|\mathcal{H}|$ of such classifiers. Taken together, we clearly have that $\delta \leq |\mathcal{H}|(1-\epsilon)^n$. So,

$$\log \delta \leq \log |\mathcal{H}| + n \log(1-\epsilon) \leq \log |\mathcal{H}| - n\epsilon \tag{99}$$

where we used the fact that $\log(1-\epsilon) \leq -\epsilon$. Solving for ϵ, we get

$$\epsilon \leq \frac{\log |\mathcal{H}| + \log(1/\delta)}{n} \tag{100}$$

In other words, the generalization error of any classifier $\hat{h}$ that achieves zero training error (under our two assumptions) is bounded by the right hand side in the above expression. Note that

- For good generalization (small ϵ), we must ensure that $\log |\mathcal{H}|$ is small compared to n. In other words, the "size" of the set of classifiers cannot be too large. What matters is the logarithm of the size.
- The more confident we wish to be about our guarantee (the smaller δ is), the more training examples we need. Clearly, the more training examples we have, the more confident we will be that a classifier which achieves zero training error will also generalize well.

The analysis is slightly more complicated if we remove the assumption that there has to be one perfect classifier in our set. The resulting guarantee is weaker but with similar qualitative dependence on the relevant quantities: with probability at least $1-\delta$ over the choice of the training set,

$$\mathcal{E}(\hat{h}) \leq \mathcal{E}_n(\hat{h}) + \sqrt{\frac{\log |\mathcal{H}| + \log(2/\delta)}{2n}} \tag{101}$$

where $\hat{h}$ is the classifier that minimizes the training error. In fact, in this case, the guarantee holds for all $\hat{h} \in \mathcal{H}$, not only for the classifier that we would choose based on the training set. The result merely shows how many examples we would need in relation to the size of the set of classifiers so that the generalization error is close to the training error *for all* classifiers in our set.

What happens to our analysis if $\mathcal{H}$ is not finite? $\log |\mathcal{H}|$ is infinite, and the results are meaningless. We must count the size of the set of classifiers differently when there are continuous parameters as in linear classifiers.

Growth function and the VC-dimension

The set of linear classifiers is an uncountably infinite set. How do we possibly count them? We will try to understand this set in terms of how classifiers from this set label training examples. In other words, we can think of creating a matrix where each row corresponds to a classifier and each column corresponds to a training example. Each entry of the matrix tells us how a particular classifier labels a specific training example. Note that there are infinite rows in this matrix and exactly n columns.

$$\begin{array}{rcccc} & x^{(1)} & x^{(2)} & \cdots & x^{(n)} \\ h \in \mathcal{H}: & +1 & -1 & \cdots & -1 \\ h' \in \mathcal{H}: & +1 & -1 & \cdots & -1 \\ h'' \in \mathcal{H}: & +1 & +1 & \cdots & -1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{array} \tag{102}$$

Not all the rows in this matrix are distinct. In fact, there can be only at most 2^n distinct rows. But our set of classifiers may not be able to generate all the 2^n possible labelings. Let $N_{\mathcal{H}}(x^{(1)}, \ldots, x^{(n)})$ be the number of distinct rows in the matrix if we choose classifiers from $\mathcal{H}$. Since this depends on the specific choice of the training examples, we look at instead the maximum number of labelings (distinct rows) that can be obtained with the same number of points:

$$N_{\mathcal{H}}(n) = \max_{x^{(1)},\ldots,x^{(n)}} N_{\mathcal{H}}(x^{(1)}, \ldots, x^{(n)}) \tag{103}$$

This is known as the *growth function* and measures how powerful the set of classifiers is. The relevant measure of the size of the set of classifiers is now $\log N_{(n)}$ (again, the logarithm of the "number"). Indeed, using this as a measure of size already gives us a generalization guarantee similar to the case of finite number of classifiers: with probability at least $1 - \delta$ over the choice of the training set,

$$\mathcal{E}(\hat{h}) \leq \mathcal{E}_n(\hat{h}) + \sqrt{\frac{\log N_{\mathcal{H}}(2n) + \log(4/\delta)}{n}}, \text{ for all } \hat{h} \in \mathcal{H} \tag{104}$$

The fact that the guarantee uses $\log N_{\mathcal{H}}(2n)$ rather than $\log N_{\mathcal{H}}(n)$ comes from a particular technical argument (symmetrization) used to derive the result. The key question here is how the new measure of size, i.e., $\log N_{\mathcal{H}}(n)$, grows relative to n. When $N_{\mathcal{H}}(n) = 2^n$, we have $\log N_{\mathcal{H}}(n) = n \log(2)$ and the guarantee remains vacuous. Indeed, our guarantee becomes interesting only when $\log N_{\mathcal{H}}(n)$ grows much slower than n. When does this happen?

This key question motivates us to define a measure of complexity of a set of classifiers, the *Vapnik-Chervonenkis* dimension or VC-dimension for short. It is the largest number of points for which $N_{\mathcal{H}}(n) = 2^n$, i.e., the largest number of points that can be labeled in all possible ways by choosing classifiers from $\mathcal{H}$. Let $d_{\mathcal{H}}$ be the VC-dimension. It turns out that $N_{\mathcal{H}}(n)$ grows much slower after n is larger than the VC-dimension $d_{\mathcal{H}}$. Indeed, when $n > d_{\mathcal{H}}$,

$$\log N_{\mathcal{H}}(2n) \leq d_{\mathcal{H}}(\log(2n/d_{\mathcal{H}}) + 1) \tag{105}$$

In other words, the number of labelings grows only logarithmically. As a result, the VC-dimension $d_{\mathcal{H}}$ captures a clear threshold for learning: when we can and cannot guarantee generalization.

So, what is the VC-dimension of a set of linear classifiers? It is exactly $d + 1$ (the number of parameters in d-dimensions). This relation to the number of parameters is often but not always true. The figure below illustrates that the VC-dimension of the set of linear classifiers in two dimensions is exactly 3.

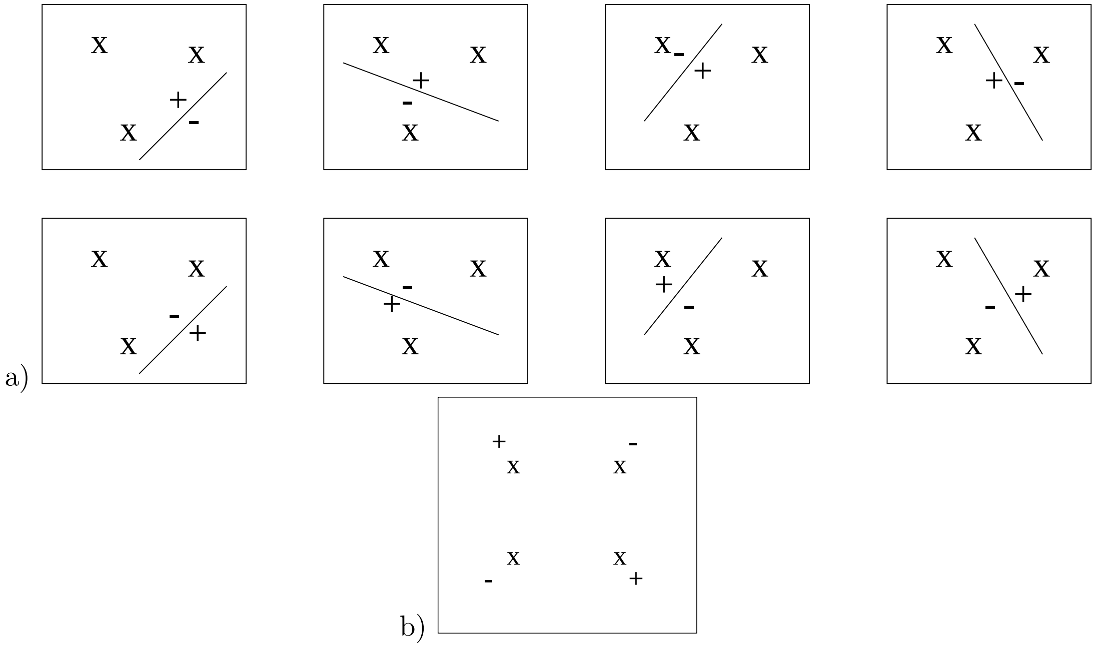


Figure 14: a) The set of linear classifiers in 2d can label three points in all possible ways; b) a labeling of four points that cannot be obtained with a linear classifier.

8 Recommender Problems

Which movie will you watch today? There are so many choices that you may have to consult a few friends for recommendations. Chances are, however, that your friends are struggling with the exact same question. In some sense we are truly lucky to have access to vast collections of movies, books, products, or daily news articles. But this "big data" comes with big challenges in terms of finding what we would really like. It is no longer possible to weed through the choices ourselves or even with the help of a few friends. We need someone a bit more "knowledgeable" to narrow down the choices for us at this scale. Such knowledgeable automated friends are called recommender systems and they already mediate much of our access to information. Whether you watch movies through Netflix or purchase products from Amazon, for example, you will be guided by recommender systems. We will use the Netflix movie recommendation problem as a running example in this chapter to illustrate how these systems actually work.

In order to recommend additional content, modern systems make use of little feedback from the user in the form of what they have liked (or disliked) in the past. There are two complementary ways to formalize this problem – *content based recommendation* and *collaborative filtering*. We will take a look at each of them individually but, ideally, they would be used in combination with each other. The key difference between them is the type of information that the machine learning algorithm is relying on. In content based recommendations, we first represent each movie in terms of a feature vector, just like before in typical classification or regression problems. For example, we may ask whether the movie is a comedy, whether Jim Carey is the lead actor, and so on, compiling all such information into a potentially high dimensional feature vector, one vector for each movie. A small number of movies that the user has seen and rated then constitutes the training set of examples and responses. We should be able to learn from this training set and be able to predict ratings for all the remaining movies that the user has not seen (the test set). Clearly, the way we construct the feature vectors will influence how good the recommendations will be. In collaborative filtering methods, on the other hand, movies are represented by how other users have rated them, dispensing entirely with any explicit features about the movies. The patterns of ratings by others do carry a lot of information. For example, if we can only find the "friend" that likes similar things, we can just borrow their ratings for other movies as well. More sophisticated ways of "borrowing" from others lies at the heart of collaborative filtering methods.

Content based recommendations

The problem of content based recommendations based on explicit movie features is very reminiscent of regression and classification problems. We will nevertheless recap the method here as it will be used as a subroutine later on for collaborative filtering. So, a bit more formally, the large set of movies in our database, m of them in total, are represented by vectors $x^{(1)}, \ldots, x^{(m)}$, where $x^{(i)} \in \mathbb{R}^d$. How the features are extracted from the movies, including the overall dimension d, may have substantial impact on our ability to predict user preferences. Good features are those that partition the movies into sets that people might conceivably assign different preferences to. For example, genre might be a good feature but the first letter of the director's last name would not be. It matters, in addition, whether the information in the features appears in a form that the regression method can make use of it. While properly engineering and tailoring the features for the method (or the other way around) is important, we will assume that the vectors are given and reasonably useful for linear predictions.

Do you rate the movies you see? Our user has rated at least some of them. Let $y^{(i)}$ be a star rating (1-5 scale) for movie i represented by feature vector $x^{(i)}$. The ratings are typically integer valued, i.e., $1, 2, \ldots, 5$ stars, but we will treat them as real numbers for simplicity. The information we have about each user (say user a) is then just the training set of rated movies $S_a = \{(x^{(i)}, y^{(i)}), i \in D_a\}$, where D_a is the index set of movies user a has explicitly rated so far. Our method takes the typically small training set (a few tens of rated movies) and turns it into predicted ratings for all the remaining movies. The movies with the highest predicted ratings could then be presented to the user as possible movies to watch. Of course, this is a very idealized interaction with the user. In practice, for example, we would need to enforce some diversity in the proposed set (preferring action movies does not mean that those are the only ones you ever watch).

We will try to solve the rating problem as a linear regression problem. For each movie $x^{(i)}$, we will predict a real valued rating $\hat{y}^{(i)} = \theta \cdot x^{(i)} = \sum_{j=1}^{d} \theta_j x_j^{(i)}$ where θ_j represents the adjustable "weight" that we place on the j^{th} feature coordinate. Note that, unlike in a typical regression formulation, we have omitted the offset parameter θ_0 for simplicity of exposition.

We estimate parameters θ by minimizing the squared error between the observed and predicted ratings on the training set while at the same time trying to keep the parameters small (regularization). More formally, we minimize

$$J(\theta) = \sum_{i \in D_a} \big(y^{(i)} - \theta \cdot x^{(i)}\big)^2/2 + \frac{\lambda}{2}\|\theta\|^2 \tag{106}$$

$$= \sum_{i \in D_a} \big(y^{(i)} - \sum_{j=1}^{d} \theta_j x_j^{(i)}\big)^2/2 + \frac{\lambda}{2}\|\theta\|^2 \tag{107}$$

with respect to the vector of parameters θ. There are many ways to solve this optimization problem as we have already seen. We could use a simple stochastic gradient descent method or obtain the solution in closed-form. In the latter case, we set the derivative of the objective $J(\theta)$ with respect to each parameter θ_k to zero, and obtain d linear equations that constrain the parameters

$$\frac{d}{d\theta_k} J(\theta) = -\sum_{i \in D_a} \big(y^{(i)} - \sum_{j=1}^{d} \theta_j x_j^{(i)}\big) x_k^{(i)} + \lambda \theta_k \tag{108}$$

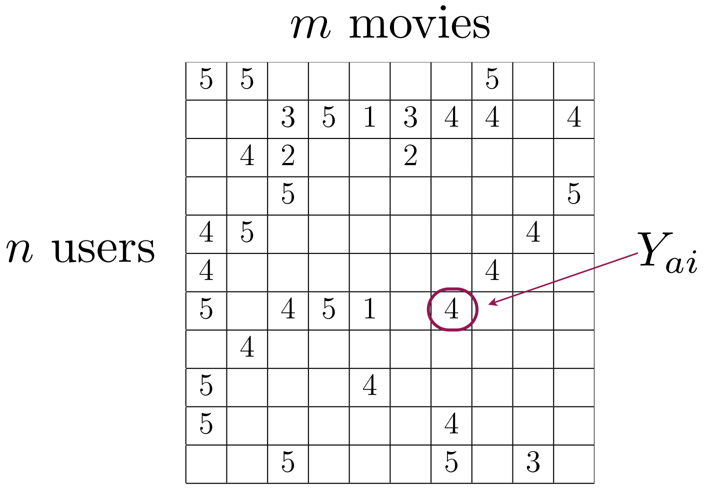
$$= -\sum_{i \in D_a} y^{(i)} x_k^{(i)} + \sum_{j=1}^{d} \theta_j \sum_{i \in D_a} x_j^{(i)} x_k^{(i)} + \lambda \theta_k \tag{109}$$

$$= -\sum_{i \in D_a} y^{(i)} x_k^{(i)} + \sum_{j=1}^{d} \sum_{i \in D_a} x_k^{(i)} x_j^{(i)} \theta_j + \lambda \theta_k \tag{110}$$

where $k = 1, \ldots, d$. It is easier to solve these in a matrix form. To this end, define a vector $b = \sum_{i \in D_a} y^{(i)} x^{(i)}$ and matrix $A = (\sum_{i \in D_a} x^{(i)} x^{(i)^T}) + \lambda I$ so that the linear equations above can be written as $-b_k + \sum_{j=1}^{d} A_{kj}\theta_j = 0$ for $k = 1, \ldots, d$. The resulting matrix form $-b + A\theta = 0$ gives $\hat{\theta} = A^{-1}b$. Recall that A is guaranteed to be invertible provided that $\lambda > 0$[2]

How well will this method work? There are several factors that affect this. The accuracy will clearly depend on the quality of features we add, i.e., whether the features represent natural distinctions along which human preferences vary. E.g., genre, director, lead actor/actress, and so on, are likely to be all good features to add. The more features we add, the more intricate preferences we can model by adjusting the parameters. However, creating and using high dimensional feature vectors for movies also means that we must have more ratings available during training in order to be able to distinguish good regression methods from a large set of other possibilities. It is not atypical to have just a few tens of rated movies. This is a problem, but it could be alleviated by "borrowing" data from other users. We will do that next with collaborative filtering. Another subtle issue is that there may be features about the movies that are not even possible to extract without considering other people's reactions to the content. For example, someone may strongly prefer movies with happy endings. While it is hard to automatically extract such a feature, it may be easy to recognize that your preferences align with such a person. Collaborative filtering tries to relate users based on their ratings and has a chance to leverage it.

Collaborative Filtering



5	5						5		
		3	5	1	3	4	4		4
	4	2			2				
		5							5
4	5							4	
4							4		
5		4	5	1		4			
	4								
5				4					
5						4			
		5				5		3	

Figure 15: Sparse data matrix Y for a collaborative filtering problem with users and movies. Empty spaces represent so far unseen (unrated) movies. The goal is to fill in these missing ratings.

How could we recommend movies to you if we have no information about the movies

[2] $(\sum_{i \in D_a} x^{(i)} x^{(i)^T})$ is positive semi-definite by construction and λI is positive definite. Thus all of A's eigenvalues are strictly positive and the matrix is invertible.

beyond their arbitrary ids? Content based recommendations (as discussed above) are indeed doomed to failure in this setting. For example, if you like movie 243, what could we say about your preference for movie 4053? Not much. No features, no basis for prediction beyond your average rating, i.e., whether you tend to give high or low ratings overall. But we can solve this problem quite well if we step up and consider all the users at once rather than individually. This is known as collaborative filtering. The key underlying idea is that we can somehow leverage experience of other users. There are, of course, many ways to achieve this. For example, it should be easy to find other users who align well with you based on a small number of movies that you have rated. The more users we have, the easier this will be. Some of them will have rated also other movies that you have not. We can then use those ratings as predictions for you, driven by your similarity to them as users. As a collaborative filtering method, this is known as nearest neighbor prediction. Alternatively, we can try to learn explicit feature vectors for movies guided by people's responses. Movies that users rate similarly would end up with similar feature vectors in this approach. Once we have such movie feature vectors, it would suffice to predict ratings separately for each user, just as we did in content based recommendations. The key difference is that now the feature vectors are "behaviorally" driven and encode only features that impact ratings. In a matrix factorization approach, the recommendation problem is solved iteratively by alternating between solving for movie features and linear regression parameters for users.

Let's make the problem a bit more formal. We have n users and m movies along with ratings for some of the movies as shown in Figure 15. Both n and m are typically quite large. For example, in the Netflix challenge problem given to the research community there were over $400,000$ users and over $17,000$ movies. Only about a few percent of the matrix entries had known ratings so Figure 15 is a bit misleading (the matrix would look much emptier in reality). We will use Y_{ai} to denote the rating that user $a \in \{1, \ldots, n\}$ has given to movie $i \in \{1, \ldots, m\}$. For clarity, we adopt different letters for indexing users $(a, b, c, \ldots)$ and movies $(i, j, k, \ldots)$. The rating Y_{ai} could be in $\{1, \ldots, 5\}$ (5 star rating) as in Figure 15, or $Y_{ai} \in \{-2, -1, 0, 1, 2\}$ if we subtract 3 from all the ratings, reducing the need for the offset term. We omit any further consideration of the rating scale, however, and instead simply treat the matrix entries as real numbers, i.e., $Y_{ai} \in \mathcal{R}$.

Nearest-neighbor prediction

We can understand the nearest neighbor method by focusing on the problem of predicting a single entry Y_{ai}. In other words, we assume that user a has not yet rated movie i and we must somehow fill in this value. It is important that there are at least some other users who have seen movie i and provided a rating for it. Indeed, collaborative filtering is powerless if faced with an entirely new movie no-one has seen.

In the nearest neighbor approach, we gauge how similar user a is to those users who have already seen movie i, and combine their ratings. Note that the set of users whose experience we "borrow" for predicting Y_{ai} changes based on the target movie i as well as user a. Now, to develop this idea further, we need to quantitatively measure how similar any two users are. A popular way to do this is via sample correlation where we focus on the subset of movies that both users have seen, and ask how these ratings co-vary (with normalization). Let $CR(a, b)$ denote the set of movies a and b have both rated. $CR(a, b)$

does not include movie i or other movies that a has yet to see. Following this notation, the sample correlation is defined as

$$\mathrm{sim}(a,b) = \mathrm{corr}(a,b) = \frac{\sum_{j\in CR(a,b)}(Y_{aj}-\tilde{Y}_a)(Y_{bj}-\tilde{Y}_b)}{\sqrt{\sum_{j\in CR(a,b)}(Y_{aj}-\tilde{Y}_a)^2}\sqrt{\sum_{j\in CR(a,b)}(Y_{bj}-\tilde{Y}_b)^2}} \tag{111}$$

where the mean rating $\tilde{Y}_a = (1/|CR(a,b)|)\sum_{j\in CR(a,b)} Y_{aj}$ is evaluated based on the movies the two users a and b have in common. It would potentially change when comparing a and c, for example. This subtlety is not visible in the notation as it would become cluttered otherwise. Note that $\mathrm{sim}(a,b) \in [-1,1]$ where -1 means that users are dissimilar, while value 1 indicates that their ratings vary identically (up to an overall scale) around the mean. So, for example, let's say user a rates movies j and k as 5 and 3, respectively, while user b assigns 4 and 2 to the same movies. If these are the only movies they have in common, $CR(a,b) = \{j,k\}$ and $\mathrm{sim}(a,b) = 1$ since the overall mean rating for the common movies does not matter in the comparison.

Once we have a measure of similarity between any two users, we can exploit it to predict Y_{ai}. Clearly, it would be a good idea to emphasize users who are highly similar to a while downweighting others. But we must also restrict these assessments to the subset of users who have actually seen i. To this end, let $KNN(a,i)$ be the top K most similar users to a who have also rated i. K here is an adjustable parameter (integer) we can set to improve the method. For now, let's fix it to a reasonable value such as 10 so as to build some statistical support (including more data from others) while at the same time avoiding users who are no longer reasonably similar. Now, the actual formula for predicting Y_{ai} is composed of two parts. The first part is simply the average rating $\bar{Y}_a$ computed from all the available ratings for a. This would be our prediction in the absence of any other users. The second part is a weighted sum of correction terms originating from other similar users. More formally,

$$\hat{Y}_{ai} = \bar{Y}_a + \frac{\sum_{b\in KNN(a,i)} \mathrm{sim}(a,b)(Y_{bi}-\bar{Y}_b)}{\sum_{b\in KNN(a,i)} |\mathrm{sim}(a,b)|} \tag{112}$$

Let's disentangle the terms a bit. $(Y_{bi}-\bar{Y}_b)$ measures how much user b's rating for i deviates from their overall mean $\bar{Y}_b$. Note that, in contrast to $\tilde{Y}_b$ appearing in the similarity computation, $\bar{Y}_b$ is based on all the ratings from b. We deal with deviations from the mean as corrections so as to be indifferent towards varying rating styles of users (some users tend to give high ratings to every movie while others are more negative). This is taken out by considering deviations from the mean. Now, the correction terms in the formula are weighted by similarity to a and normalized. The normalization is necessary to ensure that we effectively select (in a weighted manner) whose correction we adopt. For example, suppose $KNN(a,i)$ includes only one user and a did in fact rate i. So, $KNN(a,i) = \{a\}$ and Y_{ai} is available. In this case, our formula would give

$$\hat{Y}_{ai} = \bar{Y}_a + \frac{\mathrm{sim}(a,a)(Y_{ai}-\bar{Y}_a)}{|\mathrm{sim}(a,a)|} = \bar{Y}_a + \frac{1\,(Y_{ai}-\bar{Y}_a)}{|1|} = Y_{ai} \tag{113}$$

as it clearly should.

The nearest neighbor approach to collaborative filtering has many strengths. It is conceptually simple, relatively easy to implement, typically involves few adjustable parameters (here only K), yet offers many avenues for improvement (e.g., the notion of similarity). It works quite well for users who have stereotypical preferences in the sense that there are groups of others who are very close as evidenced by ratings. But the method degenerates for users with mixed interests relative to others, i.e., when there is no other with the same pattern across all the movies even though strong similarities hold for subsets. For example, a person who dislikes clowns (coulrophobic) might have a pattern of rating very similar to a group of others except for movies that involve clowns. The simple nearest neighbor method would not recognize that this subset of movies for this user should be treated differently, continuing to insist on overall comparisons and borrowing ratings from non-coulrophobic users. Since all of us are idiosyncratic in many ways, the method faces an inherent performance limitation. The matrix factorization method discussed next avoids this problem in part by decoupling the modeling of how preferences vary across users from how exactly to orient each user in this space.

Matrix factorization

We can think of collaborative filtering as a matrix problem rather than a problem involving individual users. We are given values for a small subset of matrix entries, say $\{Y_{ai}, (a,i) \in D\}$, where D is an index set of observations, and the goal is to fill in the remaining entries in the matrix. This is the setting portrayed in Figure 15. Put another way, our objective is to uncover the full underlying rating matrix Y based on observing (possibly with noise) only some of its entries. Thus the object for learning is a matrix rather a typical vector of parameters. We use X to denote the matrix we entertain and learn to keep it separate from the underlying target matrix Y (known only via observations). As in any learning problem, it will be quite necessary to constrain the set of possible matrices we can learn; otherwise observing just a few entries would in no way help determine remaining values. It is through this process of "squeezing" the set of possible matrices X that gives us a chance to predict well and, as a byproduct, learn useful feature vectors for movies (as well as users).

It is instructive to see first how things fail in the absence of any constraints on X. The learning problem here is a standard regression problem. We minimize the squared error between observations and predictions while keeping the parameters (here matrix entries) small. Formally, we minimize

$$\sum_{ai \in D} (Y_{ai} - X_{ai})^2/2 + \frac{\lambda}{2} \sum_{ai} X_{ai}^2 \tag{114}$$

with respect to the full matrix X. What is the solution $\hat{X}$? Note that there's nothing that ties the entries of X together in the objective so they can be solved independently of each other. Entries $X_{ai}, (a,i) \in D$ are guided by both observed values Y_{ai} and the regularization term X_{ai}^2. In contrast, $X_{ai}, (a,i) \notin D$ only see the regularization term and will therefore be set to zero. In summary,

$$\hat{X}_{ai} = \begin{cases} \frac{1}{1+\lambda} Y_{ai}, & (a,i) \in D \\ 0, & (a,i) \notin D \end{cases} \tag{115}$$

$$\begin{bmatrix} 5 & 7 \\ 10 & 14 \\ 30 & 42 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 6 \end{bmatrix} \times \begin{bmatrix} 5 & 7 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \\ 3 \end{bmatrix} \times \begin{bmatrix} 10 & 14 \end{bmatrix}$$

Figure 16: An example rank-1 matrix and its two possible factorizations. Note that the factorizations differ only by an overall scaling of its component vectors.

It's not exactly useful to predict all zeros for unseen movies regardless of the observations. In order to remedy the situation, we will have to introduce fewer adjustable parameters than there are entries in the matrix.

Low-rank factorization The typical way to control the effective number of parameters in a matrix is to the control its rank, i.e., how it can be written as a product of two smaller matrices. Let's commence with the simplest (most constrained) setting, where X has rank one. In this case, by definition, it must be possible to write it as an outer product of two vectors that we call U ($n \times 1$ vector) and V ($m \times 1$ vector). We will adopt capital letters for these vectors as they will be later generalized to matrices for ranks greater than one. Now, if X has rank 1, it can be written as $X = UV^T$ for some vectors U and V. Note that U and V are not unique. We can multiply U by some non-zero constant β and V by the inverse $1/\beta$ and get back the same matrix: $X = UV^T = (\beta U)(V/\beta)^T$. The factorization of a rank 1 matrix into two vectors is illustrated in Figure 16.

We will use $X = UV^T$ as the set of matrices to fit to the data where vectors $U = [u^{(1)}, \ldots, u^{(n)}]^T$ and $V = [v^{(1)}, \ldots, v^{(m)}]^T$ are adjustable parameters. Here $u^{(a)}$, $a = 1, \ldots, n$ are just scalars, as are $v^{(i)}$, $i = 1, \ldots, m$. Notationally, we are again gearing up to generalizing $u^{(a)}$ and $v^{(i)}$ to vectors but we are not there yet. Once we have U and V, we would use $X_{ai} = [UV^T]_{ai} = u^{(a)}v^{(i)}$ (simple product of scalars) to predict the rating that user a assigns to movie i. This is quite restrictive. Indeed, the set of matrices $X = UV^T$ obtained by varying U and V has only $n + m - 1$ degrees of freedom (we loose one degree of freedom to the overall scaling exchange between U and V that leaves matrix X intact). Since the number of parameters is substantially smaller than nm in the full matrix, we have a chance to actually fit these parameters based on the observed entries, and obtain an approximation to the underlying rating matrix.

Let's understand a bit further what rank 1 matrices can or cannot do. We can interpret the scalars $v^{(i)}$, $i = 1, \ldots, m$, specifying V as scalar features associated with movies. For example, $v^{(i)}$ may represent a measure of popularity of movie i. In this view $u^{(a)}$ is the regression coefficient associated with the feature, controlling how much user a likes or dislikes popular movies. But the user cannot vary this preference from one movie to another. Instead, all that they can do is to specify an overall scalar $u^{(a)}$ that measures the degree to which their movie preferences are aligned with $[v^{(1)}, \ldots, v^{(m)}]$. All users gauge their preferences relative to the same $[v^{(1)}, \ldots, v^{(m)}]$, varying only the overall scaling of this vector. Restrictive indeed.

We can generalize the idea to rank k matrices. kIn this case, we can still write $X = UV^T$ but now U and V are matrices rather than vectors. Specifically, if X has rank at most k then $U = [u^{(1)}, \ldots, u^{(n)}]^T$ is an $n \times d$ matrix and $V = [v^{(1)}, \ldots, v^{(m)}]^T$ is $m \times d$, where $d \leq \min\{n, m\}$. We characterize each movie in terms of k features, i.e., as a vector $v^{(i)} \in \mathbb{R}^d$, while each user is represented by a vector of regression coefficients (features)

$u^{(a)} \in \mathbb{R}^d$. Both of these vectors must be learned from observations. The predicted rating is $X_{ai} = [UV^T]_{ai} = u^{(a)} \cdot v^{(i)} = \sum_{j=1}^{d} u_j^{(a)} v_j^{(i)}$ (dot product between vectors) which is high when the movie (as a vector) aligns well with the user (as a vector). The set of rank k matrices $X = UV^T$ is reasonably expressive already for small values of k. The number of degrees of freedom (independent parameters) is $nd + md - d^2$ where the scaling degree of freedom discussed in the context of rank 1 matrices now appears as any invertible $d \times d$ matrix B since $UV^T = (UB)(VB^{-T})^T$. Clearly, there are many possible Us and Vs that lead to the same predictions $X = UV^T$. Suppose $n > m$ and $k = m$. Then the number of adjustable parameters in UV^T is $nm + m^2 - m^2 = nm$, the same as in the full matrix. Indeed, when $k = \min\{n, m\}$ the factorization $X = UV^T$ imposes no constraints on X at all (the matrix has full rank). In collaborative filtering only small values of k are relevant. This is known as the low rank assumption.

Learning Low-Rank Factorizations The difficulty in estimating U and V is that neither matrix is known ahead of time. A good feature vector for a movie depends on how users would react to them (as vectors), and vice versa. Both types of vectors must be learned from observed ratings. In other words, our goal is to find $X = UV^T$ for small k such that $Y_{ai} \approx X_{ai} = [UV^T]_{ai} = u^{(a)} \cdot v^{(i)}, \;\; (a,i) \in D$. The estimation problem can be formulated again as a least squares regression problem with regularization. Formally, we minimize

$$\begin{aligned} J(U,V) &= \sum_{(a,i)\in D} (Y_{ai} - [UV^T]_{ai})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n}\sum_{j=1}^{k} U_{aj}^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\sum_{j=1}^{k} V_{ij}^2 && (116) \\ &= \sum_{(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n} \|u^{(a)}\|^2 + \frac{\lambda}{2}\sum_{i=1}^{m} \|v^{(i)}\|^2 && (117) \end{aligned}$$

with respect to matrices U and V, or, equivalently, with respect to feature vectors $u^{(a)}$, $a = 1, \ldots, n$, and $v^{(i)}$, $i = 1, \ldots, m$. The smaller rank k we choose, the fewer adjustable parameters we have. The rank k and regularization parameter λ together constrain the resulting predictions $\hat{X} = \hat{U}\hat{V}^T$.

Is there a simple algorithm for minimizing $J(U,V)$? Yes, we can solve it in an alternating fashion. If the movie feature vectors $v^{(i)}$, $i = 1, \ldots, m$ were given to us, we could minimize $J(U,V)$ (solve the recommendation problem) separately for each user, just as before, finding parameters $u^{(a)}$ independently from other users. Indeed, the only part of $J(U,V)$ that depends on the user vector $u^{(a)}$ is

$$\sum_{i:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|u^{(a)}\|^2 \qquad (118)$$

where $\{i : (a,i) \in D\}$ is the set of movies that user a has rated. This is a standard least squares regression problem (without offset) for solving $u^{(a)}$ when $v^{(i)}$, $i = 1, \ldots, m$ are fixed. Note that other users do influence how $u^{(a)}$ is set but this influence goes through the movie feature vectors.

Once we have estimated $u^{(a)}$ for all the users $a = 1, \ldots, n$, we can instead fix these vectors, and solve for the movie vectors. The objective $J(U,V)$ is entirely symmetric in terms of $u^{(a)}$ and $v^{(i)}$, so solving $v^{(i)}$ also reduces to a regression problem. Indeed, the only part of $J(U,V)$ that depends on $v^{(i)}$ is

$$\sum_{a:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|v^{(i)}\|^2 \tag{119}$$

where $\{a : (a,i) \in D\}$ is the set of users who have rated movie i. This is again a regression problem without offset that we can solve since $u^{(a)}$, $a = 1,\ldots,n$ are fixed. Note that the movie feature vector $v^{(i)}$ is adjusted to help predict ratings for all the users who rated the movie. This is how they are tailored to user patterns of ratings.

Taken together, we have an alternating minimization algorithm for finding the latent feature vectors, fixing one set and solving for the other. All we need in addition is a starting point. The complete algorithm is given by

(0) Initialize the movie feature vectors $v^{(1)},\ldots,v^{(m)}$ (e.g., randomly)

(1) Fix $v^{(1)},\ldots,v^{(m)}$ and separately solve for each $u^{(a)}$, $a = 1,\ldots,n$ by minimizing

$$\sum_{i:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|u^{(a)}\|^2 \tag{120}$$

(2) Fix $u^{(1)},\ldots,u^{(n)}$ and separately solve for each $v^{(i)}$, $i = 1,\ldots,m$, by minimizing

$$\sum_{a:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|v^{(i)}\|^2 \tag{121}$$

Each minimization step in our alternating algorithm utilizes old parameter values for the other set that is kept fixed. It is therefore necessary to iterate over steps (1) and (2) in order to arrive at a good solution. The resulting values for U and V can depend quite a bit on the initial setting of the movie vectors. For example, if we initialize $v^{(i)} = 0$ (vector), $i = 1,\ldots,m$, then step (1) of the algorithm produces user vectors that are also all zero. This is because $u^{(a)} \cdot v^{(i)} = 0$ regardless of $u^{(a)}$. In the absence of any guidance from the error terms, the regularization term drives the solution to zero. The same would happen in step (2) to the new movie vectors since user vectors are now zero, and so on. Can you figure out what would happen if we initialized the movie vectors to be all the same but non-zero? (left as an exercise). The issue here is that while each step, (1) or (2), offers a unique solution, the overall minimization problem is not jointly convex (bowl shaped) with respect to (U,V). There are locally optimal solutions. The selection of where we end up is based on the initialization (the rest of the algorithm is deterministic). It is therefore a good practice to run the algorithm a few times with randomly initialized movie vectors and either select the best one (the one that achieves the lowest value of $J(U,V)$) or combine the solutions from different runs. A theoretically better algorithm would use the fact that there are many Us and Vs that result in the same $X = UV^T$ and therefore cast the problem directly in terms of X. The alternating algorithm we have presented is often sufficient and widely used in practice.

Alternating minimization example For concreteness, let's see how the alternating minimization algorithm works when $k = 1$, i.e., when we are looking for a rank-1 solution. Assume that the observed ratings are given in a 2×3 matrix Y (2 users, 3 movies)

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix} \tag{122}$$

where the question marks indicate missing ratings. Our goal is to find U and V such that $X = UV^T$ closely approximates the observed ratings in Y. We start by initializing the movie features $V = [v^{(1)}, v^{(2)}, v^{(3)}]^T$ where $v^{(i)}$, $i = 1, 2, 3$, are scalars since $d = 1$. In other words, V is just a 3×1 vector which we set as $[2, 7, 8]^T$. Given this initialization, our predicted rating matrix $X = UV^T$, as a function of $U = [u^{(1)}, u^{(2)}]^T$, where $u^{(a)}$, $a = 1, 2$, are scalars, becomes

$$UV^T = \begin{bmatrix} 2u^{(1)} & 7u^{(1)} & 8u^{(1)} \\ 2u^{(2)} & 7u^{(2)} & 8u^{(2)} \end{bmatrix} \tag{123}$$

We are interested in finding $u^{(1)}$ and $u^{(2)}$ that best approximates the ratings (step (1) of the algorithm). For instance, for user 1, the observed ratings 5 and 7 are compared against predictions $2u^{(1)}$ and $8u^{(1)}$. The combined loss and regularizer for this user in step (1) of the algorithm is

$$J_1(u^{(1)}) = \frac{(5 - 2u^{(1)})^2}{2} + \frac{(7 - 8u^{(1)})^2}{2} + \frac{\lambda}{2}(u^{(1)})^2 \tag{124}$$

To minimize this loss, we differentiate it with respect to $u^{(1)}$ and equate it to zero.

$$\frac{dJ_1(u^{(1)})}{du^{(1)}} = -66 + (68 + \lambda)u^{(1)} = 0 \tag{125}$$

resulting in $u^{(1)} = \frac{66}{\lambda + 68}$. We can similarly find $u^{(2)} = \frac{16}{\lambda + 53}$.

If we set $\lambda = 1$, then the current estimate of U is $[66/69, 16/54]^T$. We will next estimate V based on this value of U. Now, writing $X = UV^T$ as a function of $V = [v^{(1)}, v^{(2)}, v^{(3)}]^T$, we get

$$UV^T = \begin{bmatrix} \frac{66}{69}v^{(1)} & \frac{66}{69}v^{(2)} & \frac{66}{69}v^{(3)} \\ \frac{16}{54}v^{(1)} & \frac{16}{54}v^{(2)} & \frac{16}{54}v^{(3)} \end{bmatrix} \tag{126}$$

As before, in step (2) of the algorithm, we separately solve for $v^{(1)}$, $v^{(2)}$, and $v^{(3)}$. The combined loss and regularizer for the first movie is now

$$\frac{(5 - \frac{66}{69}v^{(1)})^2}{2} + \frac{(1 - \frac{16}{54}v^{(1)})^2}{2} + \frac{\lambda}{2}(v^{(1)})^2 \tag{127}$$

We would again differentiate this objective with respect to $v^{(1)}$ and equate it to zero to solve for the new updated value for $v^{(1)}$. The remaining $v^{(2)}$ and $v^{(3)}$ are obtained analogously.

9 Representation and Neural networks

We have so far covered three different ways of performing non-linear classification. The first one maps examples x explicitly into feature vectors $\phi(x)$ which contain non-linear terms of the coordinates of x. The classification decisions are based on

$$\hat{y} = \text{sign}(\theta \cdot \phi(x)) \tag{128}$$

where we have omitted the bias/offset term for simplicity. The second method translates the same problem into a kernel form so as to reduce the computations involved into comparisons between examples (via the kernel) rather than evaluating the feature vectors explicitly. This approach can be considerably more efficient in cases where the inner product between feature vectors, i.e., the kernel, can be evaluated without ever enumerating the coordinates of the feature vectors. The decision rule for a kernel method can be written as

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x)) \tag{129}$$

We can always map the kernel classifier back into the explicit form in Eq.(128) with the idea that $K(x, x') = \phi(x) \cdot \phi(x')$ and $\theta = \sum_{i=1}^{n} \alpha_i y^{(i)} \phi(x^{(i)})$ (recall kernel perceptron). However, we can also view Eq.(129) as a linear classifier with parameters $\alpha = [\alpha_1, \ldots, \alpha_n]^T$. In this view, the feature vector corresponding to each example x would be $\tilde{\phi}(x) = [y^{(1)}K(x^{(1)}, x), \ldots, y^{(n)}K(x^{(n)}, x)]^T$, i.e., each new x is compared to all the training examples, multiplied by the corresponding labels, and concatenated into a vector of dimension n. The predicted label for x is then

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x)) = \text{sign}(\alpha \cdot \tilde{\phi}(x)) \tag{130}$$

Note that the feature vector $\tilde{\phi}(x)$ is now "adjusted" based on the training set (in fact, it is explicitly constructed by comparing examples to training examples) but it is not learned specifically to improve classification performance.

Finally, we have shown how to build a strong classifier by combining simple weak/base classifiers into an ensemble through boosting. The resulting ensemble classifier looks like

$$\hat{y} \quad = \quad \text{sign}(\sum_{j=1}^{m} \alpha_j h(x; \theta_j)) = \text{sign}(\vec{\alpha} \cdot \vec{\phi}(x)) \tag{131}$$

where, again, we have viewed the ensemble as a linear classifier with parameters (votes) $\vec{\alpha} = [\alpha_1, \ldots, \alpha_m]^T$ and feature vectors $\vec{\phi}(x) = [h(x; \theta_1), \ldots, h(x; \theta_m)]^T$ composed of the outputs of the weak learners. Note that in this case the feature representation $\phi(\vec{x})$ is explicitly tailored to optimize classification performance. However, since learning the parameters $\vec{\alpha}$ together with the feature representation $\vec{\phi}(x)$ is challenging, boosting approximates this process by performing the optimization sequentially, one coordinate (weak learner) at a time while fixing those already optimized.

We will next formulate models where the feature representation is learned jointly with the classifier.

Feed-forward Neural Networks

Neural networks consist of a large number of simple computational units/neurons (e.g., linear classifiers) which, together, specify how the input vector x is processed towards the final classification decision. Neural networks can do much more than just classification but we will use classification problems here for continuity. In a simple case, the units in the neural network are arranged in layers, where each layer defines how the input signal is transformed in stages. These are called *feed-forward* neural networks. They are loosely motivated by how our visual system processes the signal coming to the eyes in massively parallel stages. The layers in our models include

1. *input layer* where the units simply store the coordinates of the input vector (one unit assigned to each coordinate). The input units are special in the sense that they don't compute anything. Their activation is just the value of the corresponding input coordinate.

2. possible *hidden layers* of units which represent complex transforms of the input signal, from one layer to the next, towards the final classifier. These units determine their activation by aggregating input from the preceding layer

3. *output layer*, here a single unit, which is a linear classifier taking as its input the activations of the units in the penultimate (hidden or the input) layer.

Figure 17 shows a simple feed-forward neural network with two hidden layers. The units correspond to the nodes in the graph and the edges specify how the activation of one unit may depend on the activation of other units. More specifically, each unit aggregates input from other preceding units (units in the previous layer) and evaluates an output/activation value by passing the summed input through a (typically non-linear) transfer/activation/link function. All the units except the input units act in this way. The input units are just clamped to the observed coordinates of x.

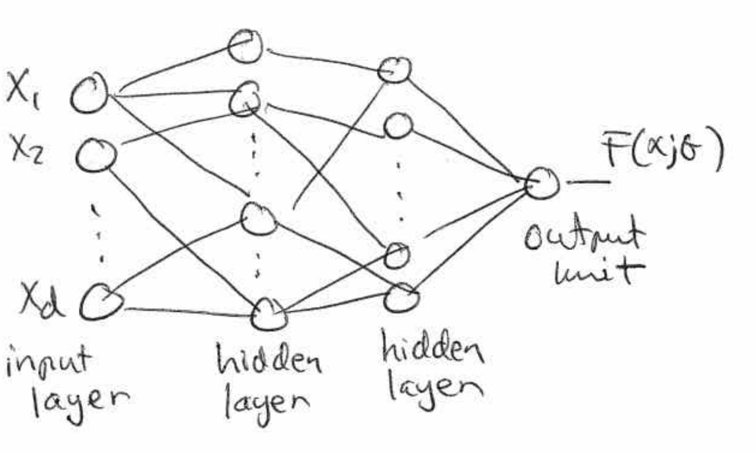


Figure 17: A feed-forward neural network with two hidden layers and a single output unit.

Simplest neural network

Let's begin with the simplest neural network (a linear classifier). In this case, we only have d input units corresponding to the coordinates of $x = [x_1, \ldots, x_d]^T$ and a single linear output unit producing $F(x;\theta)$ shown in Figure 18. We will use θ to refer to the set of all parameters in a given network. So the number of parameters in θ varies from one architecture to another. Now, the output unit receives as its aggregated input a weighted combination of the input units (plus an overall offset V_0).

$$\begin{aligned} z &= \sum_{i=1}^{d} x_i V_i + V_0 = x \cdot V + V_0 \;\; \text{(weighted summed input to the unit)} & (132) \\ F(x;\theta) &= f(z) = z \;\; \text{(network output)} & (133) \end{aligned}$$

where the activation function $f(\cdot)$ is simply linear $f(z) = z$. So this is just a standard linear classifier if we classify each x by taking the sign of the network output. We will leave the network output as a real number, however, so that it can be easily fed into the Hinge or other such loss function. The parameters θ in this case are $\theta = \{V_1, \ldots, V_d, V_0\} = \{V, V_0\}$ where V is a vector.

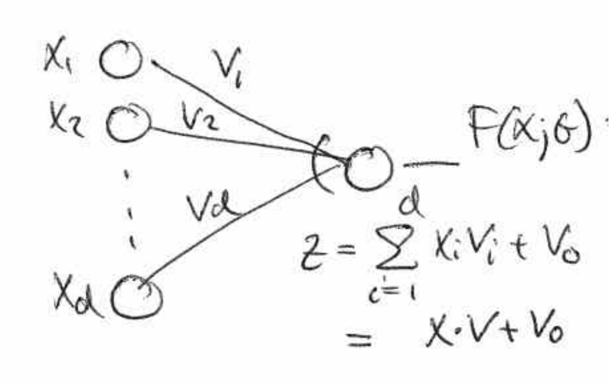


Figure 18: A simple neural network with no hidden units.

Hidden layers, representation

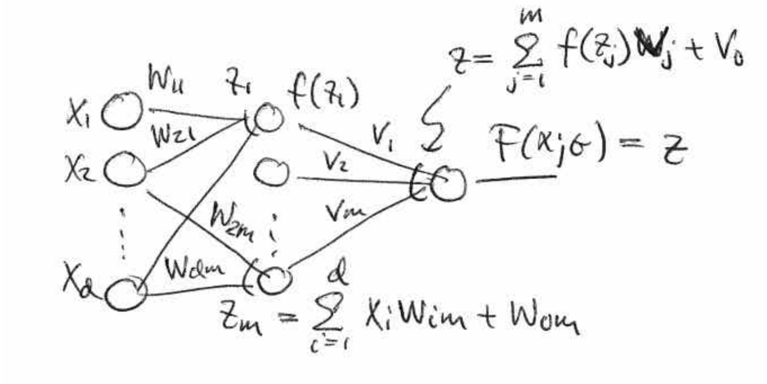


Figure 19: A neural network with one hidden layer.

Let's extend the model a bit and consider a neural network with one hidden layer. This is shown in Figure 19. As before, the input units are simply clamped to the coordinates of the input vector x. In contrast, each of the m hidden units evaluate their output in two steps

$$z_j = \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \quad \text{(weighted input to the } j^{th} \text{ hidden unit)} \tag{134}$$

$$f(z_j) = \max\{0, z_j\} \quad \text{(rectified linear activation function)} \tag{135}$$

where we have used so called *rectified linear* activation function which simply passes any positive input through as is and squashes any negative input to zero. A number of other activation functions are possible, including $f(z) = \text{sign}(z)$, $f(z) = (1 + \exp(-z))^{-1}$ (sigmoid), and so on. We will use rectified linear hidden units which are convenient when we derive the learning algorithm. Now, the single output unit no longer sees the input example directly but only the activations of the hidden units. In other words, as a unit, it is exactly as before but takes in each $f(z_j)$ instead of the input coordinates x_i. Thus

$$z = \sum_{j=1}^{m} f(z_j) V_j + V_0 \quad \text{(weighted summed input to the unit)} \tag{136}$$

$$F(x;\theta) = z \quad \text{(network output)} \tag{137}$$

The output unit is again linear and functions as a linear classifier on a new feature representation $[f(z_1), \ldots, f(z_m)]^T$ of each example. Note that z_j are functions of x but we have suppressed this in the notation. The parameters θ in this case include both the weights $\{W_{ij}, W_{0j}\}$ that mediate hidden unit activations in response to the input, and $\{V_j, V_0\}$ which specify how the network output depends on the hidden units.

How powerful is the neural network model with one hidden layer? It turns out that it is already a universal approximator in the sense that it can approximate any mapping from the input to the output if we increase the number of hidden units. But it is not necessarily easy to find the parameters θ that realize any specific mapping exemplified by the training examples. We will return to this question later.

Let's take a simple example to see where the power lies. Consider the labeled two dimensional points shown in Figure 20. The points are clearly not linearly separable and therefore cannot be correctly classified with a simple neural network without hidden units. Suppose we introduce only two hidden units such that

$$\begin{aligned}
z_1 &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{11} \\ W_{21} \end{bmatrix} + W_{01} && (138)\\
f(z_1) &= \max\{0, z_1\} \ \text{(activation of the 1st hidden unit)} && (139)\\
z_2 &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{12} \\ W_{22} \end{bmatrix} + W_{02} && (140)\\
f(z_2) &= \max\{0, z_2\} \ \text{(activation of the 2nd hidden unit)} && (141)
\end{aligned}$$

Each example x is then first mapped to the activations of the hidden units, i.e., has a two-dimensional feature representation $[f(z_1), f(z_2)]^T$. We choose the parameters W as shown pictorially in the Figure 20. Note that we can represent the hidden unit activations similarly to a decision boundary in a linear classifier. The only difference is that the output is not binary but rather is identically zero in the negative half, and increases linearly in the positive part as we move away from the boundary. Now, using these parameters, we can map the labeled points (approximately) to their feature coordinates $[f(z_1), f(z_2)]^T$ as shown on the right in the same figure. The labeled points are now linearly separable in these feature coordinates and therefore the single output unit – a linear classifier – can find a separator that correctly classifies these training examples.

As an exercise, think about whether the examples remain linearly separable in the feature coordinates $[f(z_1), f(z_2)]^T$ if we flip the positive/negative sides of the two hidden units. In other words, now the positive examples would have $f(z_1) > 0$ and $f(z_2) > 0$. This example highlights why parameter estimation in neural networks is not an easy task.

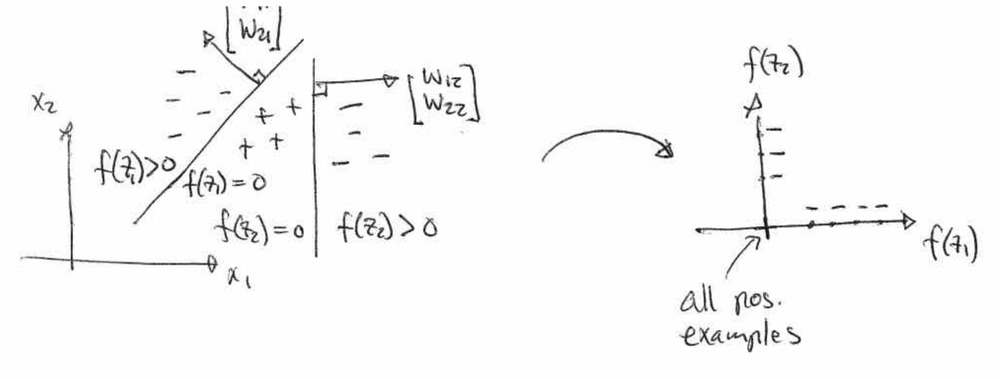


Figure 20: Hidden unit activations and examples represented in hidden unit coordinates

Learning Neural Networks

Given a training set $\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$ of examples $x \in \mathcal{R}^d$ and labels $y \in \{-1, 1\}$, we would like to estimate parameters θ of the chosen neural network model so as to minimize the average loss over the training examples,

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{142}$$

where we assume that the loss is the Hinge loss $\text{Loss}(z) = \max\{0, 1 - z\}$. Clearly, since these models can be very complex, we should add a regularization term as well. We could use, for example, $(1/2)\|\theta\|^2$ as the regularizer so as to squash parameters towards zero if they are not helpful for classification. There is a better way to regularize neural network models so we will leave the regularization out for now, and return to it later.

In order to minimize the average loss, we will resort to a simple stochastic optimization procedure rather than performing gradient descent steps on $J(\theta)$ directly. The stochastic version, while simpler, is also likely to work better with complex models, providing the means to randomize the exploration of good parameter values. On a high level, our algorithm is simply as follows. We sample a training example at random, and nudge the parameters towards values that would improve the classification of that example. Many such small steps will overall move the parameters in a direction that reduce the average loss. The algorithm is known as *stochastic gradient descent* or SGD, written more formally as

$$\text{(0)} \;\; \text{Initialize } \theta \text{ to small random values} \tag{143}$$

$$\text{(1)} \;\; \text{Select } i \in \{1, \ldots, n\} \text{ at random or in a random order} \tag{144}$$

$$\text{(2)} \;\; \theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{145}$$

where we iterate between (1) and (2). Here the gradient

$$\nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) = [\frac{\partial}{\partial \theta_1} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)), \ldots, \frac{\partial}{\partial \theta_D} \text{Loss}(y^{(i)} F(x^{(i)}; \theta))]^T \tag{146}$$

has the same dimension as the parameter vector, and it points in a direction in the parameter space where the loss function increases. We therefore nudge the parameters in the opposite direction. The *learning rate* η_k should decrease slowly with the number of updates. It should be small enough that we don't overshoot (often), i.e., if η_k is large, the new parameter values might actually increase the loss after the update. If we set, $\eta_k = \eta_0/(k{+}1)$ or, more generally, set η_k, $k = 1, 2, \ldots$, such that $\sum_{k=1}^{\infty} \eta_k = \infty$, $\sum_{k=1}^{\infty} \eta_k^2 < \infty$, then we would be guaranteed in simple cases (without hidden layers) that the algorithm converges to the minimum average loss. The presence of hidden layers makes the problem considerably more challenging. For example, can you see why it is critical that the parameters are NOT initialized to zero when we have hidden layers? We will make the algorithm more concrete later, actually demonstrating how the gradient can be evaluated by propagating the training signal from the output (where we can measure the discrepancy) back towards the input layer (where many of the parameters are).

While our estimation problem may appear daunting with lots of hidden units, it is surprisingly easier if we increase the number of hidden units in each layer. In contrast, adding layers (deeper architectures) are tougher to optimize. The argument for increasing the size of each layer is that high-dimensional intermediate feature spaces yield much room for gradient steps to traverse while permitting continued progress towards minimizing the objective. This is not the case with small models. To illustrate this effect, consider the classification problem in Figure 21. Note that the points are setup similarly to the example discussed above where only two hidden units would suffice to solve the problem. We will try networks with a single hidden layer, and 10, 100, and 500 hidden units. It may seem that the smallest of them – 10 hidden units – should already easily solve the problem. While it clearly has the power to do so, such parameters are hard to find with gradient based algorithms (repeated attempts with random initialization tend to fail as in the figure). However, with 100 or 500 hidden units, we can easily find a nice decision boundary that separates the examples.

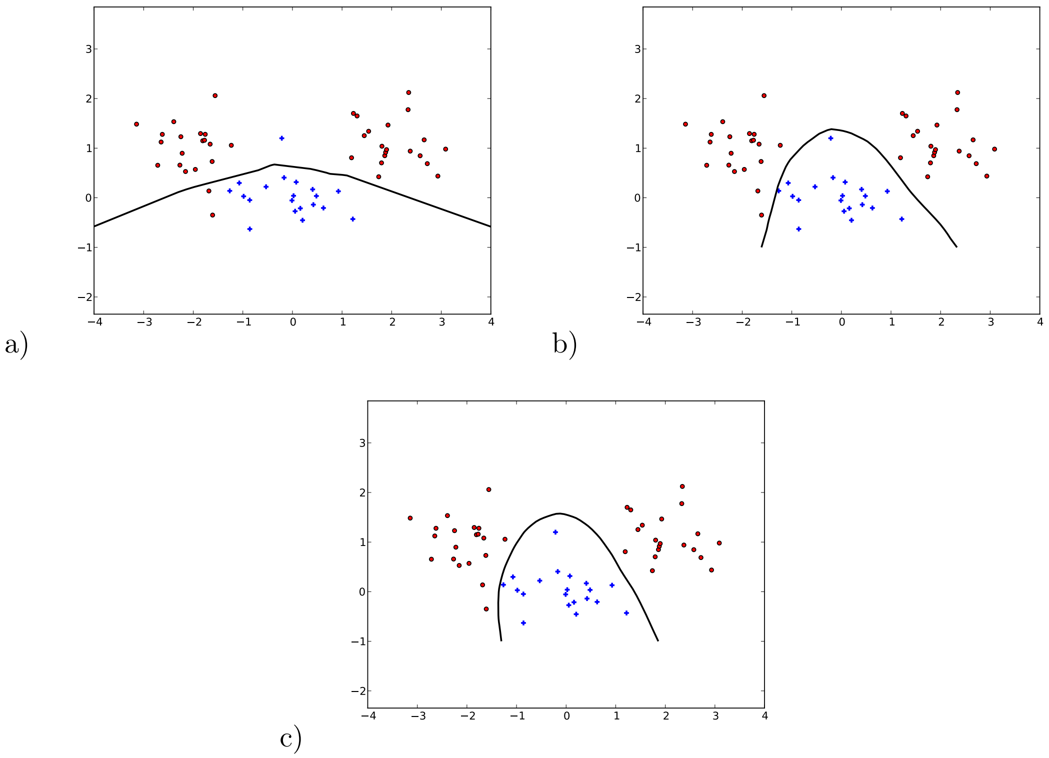


Figure 21: Decision boundaries resulting from training neural network models with one hidden layer and varying number of hidden units: a) 10; b) 100; and c) 500.

Stochastic gradient descent, back-propagation

Our goal here is to make the algorithm more concrete, adapting it to simple example networks, and seeing back-propagation (chain rule) in action. We will also discuss little adjustments to the algorithm that may help make it work better in practice.

If viewed as a classifier, our network generates a real valued output $F(x;\theta)$ in response to any input vector $x \in \mathcal{R}^d$. This mapping is mediated by parameters θ that represent all the weights in the model. For example, in case of a) one layer or b) two layer neural networks, the parameters θ correspond to vectors

$$\theta = [V_1, \ldots, V_d, V_0]^T \tag{147}$$

$$\theta = [W_{11}, \ldots, W_{1m}, \ldots, W_{d1}, \ldots, W_{dm}, W_{01}, \ldots, W_{0m}, V_1, \ldots, V_d, V_0]^T \tag{148}$$

respectively, as illustrated in Figures 22a) and b) below.

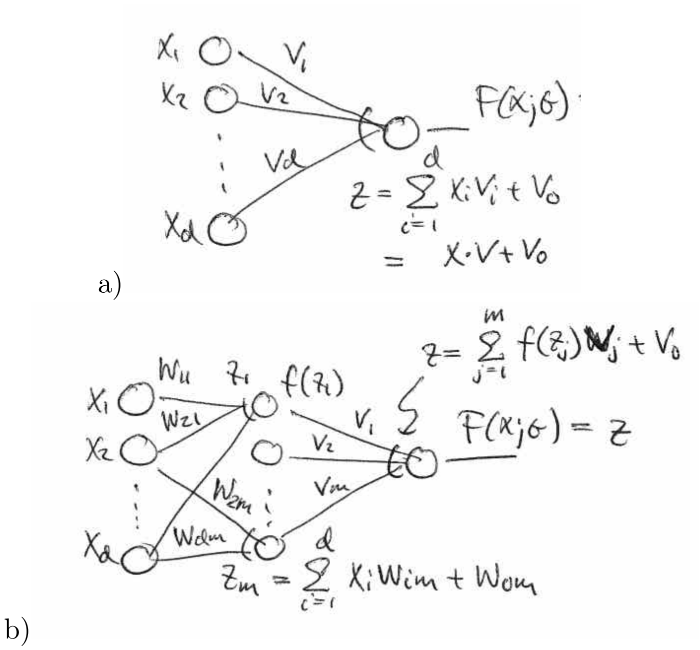

X1
X2
Xd
V1
V2
Vd
F(x;θ)
z = $\sum_{i=1}^{d}$ xiVi + V0
= x·V + V0
a)
z = $\sum_{j=1}^{m}$ f(zj)Wj + V0
W11
W21
z1
f(z1)
V1
V2
Vm
W2m
Wdm
F(x;θ) = z
X1
X2
Xd
zm = $\sum_{i=1}^{d}$ xiWim + W0m
b)

Figure 22: Example networks

We will illustrate here first how to use stochastic gradient descent (SDG) to minimize average loss over the training examples shown in Eq.(142). The algorithm performs a series of small updates by focusing each time on a randomly chosen loss term. After many such small updates, the parameters will have moved in a direction that reduces the overall loss above. More concretely, we iteratively select a training example $(x^{(t)}, y^{(t)})$ at random (or in a random order) and move the parameters in the opposite direction of the gradient of the loss associated with that example. Each parameter update is therefore of the form

$$\theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big) \tag{149}$$

where η_k is the learning rate/step-size parameter after k updates. Note that we will update all the parameters in each step. In other words, if we break the update into each coordinate, then

$$\theta_i \leftarrow \theta_i - \eta_k \frac{\partial}{\partial \theta_i} \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big), \quad i = 1, \ldots, D \tag{150}$$

In order to use SGD, we need to specify three things: 1) how to evaluate the derivatives, 2) how to initialize the parameters, and 3) how to set the learning rate.

Stochastic gradient descent for single layer networks

Let's begin with the simplest network in Figure 22a) where $\theta = [V_1, \ldots, V_d, V_0]^T$. This is just a linear classifier and we might suspect that SGD reduces to similar updates as perceptron or passive-aggressive algorithm. This is indeed so. Now, given any training example $(x^{(t)}, y^{(t)})$, our first task is to evaluate

$$\frac{\partial}{\partial V_i} \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big) \quad = \quad \frac{\partial}{\partial V_i} \text{Loss}\big(y^{(t)} z^{(t)}\big) \tag{151}$$

where $z^{(t)} = \sum_{j=1}^{d} x_j^{(t)} V_j + V_0$ and we have assumed (as before) that the output unit is linear. Moreover, we will assume that the loss function is the Hinge loss, i.e., $\text{Loss}(yz) = \max\{0, 1 - yz\}$. Now, the effect of V_i on the network output, and therefore the loss, goes entirely through the summed input $z^{(t)}$ to the output unit. We can therefore evaluate the derivative of the loss with respect to V_i by appeal to chain rule

$$\begin{aligned}
\frac{\partial}{\partial V_i} \text{Loss}\big(y^{(t)} z^{(t)}\big) \quad & \overset{\text{chain rule}}{=} \quad \left[\frac{\partial z^{(t)}}{\partial V_i}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] && (152) \\
& = \quad \left[\frac{\partial(\sum_{j=1}^{d} x_j^{(t)} V_j + V_0)}{\partial V_i}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] && (153) \\
& = \quad \left[x_i^{(t)}\right] \left[\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)} z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array}\right] && (154)
\end{aligned}$$

You may notice that $\text{Loss}(yz)$ is not actually differentiable at a single point where $yz = 1$. For our purposes here it suffices to use a *sub-gradient*[3] rather than a derivative at that point. Put another way, there are many possible derivatives at $yz = 1$ and we opted for one of them (zero). SGD will work fine with sub-gradients.

We can now explicate SGD for the simple network. As promised, the updates look very much like perceptron or passive-aggressive updates. Indeed, we will get a non-zero update when $\text{Loss}\big(y^{(t)} z^{(t)}\big) > 0$ and

$$\begin{aligned}
V_i \quad & \leftarrow \quad V_i + \eta_k y^{(t)} x_i^{(t)}, \;\; i = 1, \ldots, d && (155) \\
V_0 \quad & \leftarrow \quad V_0 + \eta_k y^{(t)} && (156)
\end{aligned}$$

We can also initialize the parameters to all zero values as before. This won't be true for more complex models (as discussed later).

It remains to select the learning rate. We could just use a decreasing sequence of values such as $\eta_k = \eta_0/(k+1)$. This may not be optimal, however, as Figure 23 illustrates. In SGD, the magnitude of the update is directly proportional to the gradient (slope in the figure). When the gradient (slope) is small, so is the update. Conversely, if the objective varies sharply with θ, the gradient-based update would be large. But this is exactly the wrong way around. When the objective function varies little, we could/should make larger steps

[3] Think of the knot at $yz = 1$ of $\max\{0, 1 - yz\}$ as a very sharp but smooth turn. Little changes in yz would change the derivative from zero (when $yz > 1$) to $-y$ (when $yz < 1$). All the possible derivatives around the point constitute a *sub-differential*. A *sub-gradient* is any one of them.

so as to get to the minimum faster. Moreover, if the objective varies sharply as a function of the parameter, the update should be smaller so as to avoid overshooting. A fixed and/or decreasing learning rate is oblivious to such concerns. We can instead adaptively set the step-size based on the gradients (AdaGrad):

$$\begin{aligned} g_i &\leftarrow \frac{\partial}{\partial V_i}\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) \text{ (gradient or sub-gradient)} && (157)\\ G_i &\leftarrow G_i + g_i^2 \text{ (cumulative squared gradient)} && (158)\\ V_i &\leftarrow V_i - \frac{\eta}{\sqrt{G_i}} g_i \text{ (adaptive gradient update)} && (159) \end{aligned}$$

where the updates, as before, are performed for all the parameters, i.e., for all $i = 0, 1, \ldots, D$, in one go. Here η can be set to a fixed value since $\sqrt{G_i}$ reflects both the magnitude of the gradients as well as the number of updates performed. Note that we have some freedom here in terms of how to bundle the adaptive scaling of learning rates. In the example above, the learning rate is adjusted separately for each parameter. Alternatively, we could use a common scaling per node in the network such that all the incoming weights to a node are updated with the same learning rate, adjusted by the cumulative squared gradient that is now a sum of the individual squared derivatives.

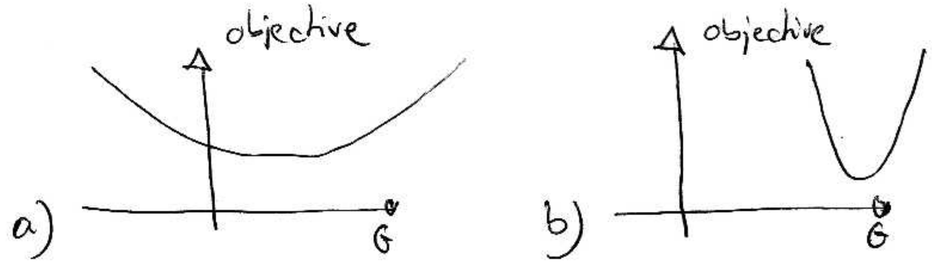


Figure 23: Objective functions that vary a) little b) a lot as a function of the parameter.

Stochastic gradient descent for two layer networks

Let us now consider the two layer network in Figure 22b). Recall that the network output is now obtained in stages, activating the hidden units, before evaluating the output. In other words,

$$
\begin{aligned}
z_j &= \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \ \text{ (input to the } j^{th} \text{ hidden unit)} && (160)\\
f(z_j) &= \max\{0, z_j\} \ \text{ (output of the } j^{th} \text{ hidden unit)} && (161)\\
z &= \sum_{j=1}^{m} f(z_j) V_j + V_0 \ \text{ (input to the last unit)} && (162)\\
F(x;\theta) &= z \ \text{ (network output)} && (163)
\end{aligned}
$$

The last layer (the output unit) is again simply a linear classifier but bases its decisions on the transformed input $[f(z_1), \ldots, f(z_m)]^T$ rather than the original $[x_1, \ldots, x_d]^T$. As a result, we can follow the SGD updates we have already derived for the single layer model. Specifically,

$$V_j \leftarrow V_j + \eta_k y^{(t)} f(z_j^{(t)}), \ \ j = 1, \ldots, m \tag{164}$$

where $z_j^{(t)}$ is the input to the j^{th} hidden unit resulting from presenting $x^{(t)}$ to the network, i.e., $z_j^{(t)} = \sum_{i=1}^d x_i^{(t)} W_{ij} + W_{0j}$. Note that now $f(z_j^{(t)})$ serves the same role as the input coordinate $x_j^{(t)}$ did in the single layer network.

In order to update W_{ij}, we must consider how it impacts the network output. Changing W_{ij} will first change z_j, then $f(z_j)$, then finally z. In this case the path of influence of the parameter on the network output is unique. There are typically many such paths in multi-layer networks (and must all be considered). To calculate the derivative of the loss with respect to W_{ij} in our case, we simply repeatedly apply the chain rule

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}} \text{Loss}\big(y^{(t)} z^{(t)}\big) &= \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right] \left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right] \left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] && (165)\\
&= [x_i] \, [\![\, z_j^{(t)} > 0 \,]\!] \, [V_j] \left[\begin{array}{l} -y^{(t)} \text{ if Loss}\big(y^{(t)} z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array} \right] && (166)
\end{aligned}
$$

Note that $f(z_j) = \max\{0, z_j\}$ is not differentiable at $z_j = 0$ but we will again just take a sub-gradient $[\![\, z_j > 0 \,]\!]$ which is one if $z_j > 0$ and zero otherwise. When there are multiple layers of units, the gradients can be evaluated efficiently by propagating them backwards from the output (where the signal lies) back towards the inputs (where the parameters are). This is because each previous layer must evaluate how the next layer affects the output as the influence of the associated parameters goes through the next layer. The process of propagating the gradients backwards towards the input layer is called *back-propagation*.

We can now write down the simple SGD rule for W_{ij} parameters as well. Whenever $x^{(t)}$ isn't classified sufficiently well, i.e., $\text{Loss}\big(y^{(t)} z^{(t)}\big) > 0$,

$$W_{ij} \leftarrow W_{ij} + \eta_k \, x_i^{(t)} [\![\, z_j^{(t)} > 0 \,]\!] V_j y^{(t)}, \ \ i = 1, \ldots, d, \ j = 1, \ldots, m \tag{167}$$

Note that the further back the parameters are, the more multiplicative terms appear in the update. This can have the effect of easily either exploding or vanishing the gradients, precluding effective learning. The issue of learning rate is therefore quite important for these parameters but can be mitigated as discussed before (AdaGrad).

Properly initializing the parameters is much more important in networks with two or more layers. For example, if we set all the parameters (W_{ij}'s and V_j's) to zero, then also $z_j^{(t)} = 0$ and $f(z_j^{(t)}) = 0$. Thus the output unit only sees an all-zero "input vector" $[f(z_1^{(t)}), \ldots, f(z_m^{(t)})]^T$ resulting in no updates. Similarly, the gradient for W_{ij} includes both $[\![\, z_j^{(t)} > 0 \,]\!]$ and V_j which are both zero. In other words, SGD would forever remain at the all-zero parameter setting. Can you see why it would work to initialize W_{ij}'s to non-zero values while V_j's are set initially to zero?

Since hidden units (in our case) all have the same functional form, we must use the initialization process to break symmetries, i.e., help the units find different roles. This is typically achieved by initializing the parameters randomly, sampling each parameter value from a Gaussian distribution with zero mean and variance σ^2 where the variance depends on the layer (the number of units feeding to each hidden unit). For example, unit z_j receives d inputs in our two-layer model. We would like to set the variance of the parameters such that the overall input to the unit (after randomization) does not strongly depend on d (the number of input units feeding to the hidden unit). In this sense, the unit would be initialized in a manner that does not depend on the network architecture it is part of. To this end, we could sample each associated W_{ij} from a zero-mean Gaussian distribution with variance $1/d$. As a result, the input $z_j = \sum_{i=1}^{d} x_i W_{ij} + 0$ to each hidden unit (with zero offset) corresponds to a different random realization of W_{ij}'s. We can ask how $z_1, \ldots, z_m$ vary from one unit to another. This variance is exactly $(1/d)\sum_{i=1}^{d} x_i^2$ which does not scale with d.

Regularization, dropout training

Our network models can be quite complex (have a lot of power to overfit to the training data) as we increase the number of hidden units or add more layers (deeper architecture). There are many ways to regularize the models. The simplest way would be to add a squared penalty on the parameter values to the training objective, i.e., use SGD to minimize

$$\frac{1}{n}\sum_{t=1}^{n} \text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2 \quad = \quad \frac{1}{n}\sum_{t=1}^{n} \overbrace{\left[\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2\right]}^{\text{modified loss}} \tag{168}$$

where λ controls the strength of regularization. We have rewritten the objective so that the regularization term appears together with each loss term. This way we can derive the SGD algorithm as before but with a modified loss function that now includes a regularization term. For example, V_j's in the two layer model would be now updated as

$$V_j \leftarrow V_j + \eta_k\Big(-\lambda V_j + y^{(t)} f(z_j^{(t)})\Big), \;\; j = 1, \ldots, m \tag{169}$$

when $\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) > 0$ and

$$V_j \leftarrow V_j + \eta_k\Big(-\lambda V_j + 0\Big), \;\; j = 1, \ldots, m \tag{170}$$

when $\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) = 0$. The additional term $-\lambda V_j$ pushes the parameters towards zero and this term remains even when the loss is zero since it comes about as the negative gradient of $\lambda\|\theta\|^2/2 = \lambda(\sum_{j=1}^{m} V_j^2 + \ldots)/2$.

Let's find a better way to regularize large network models. In the two layer model, we could imagine increasing m, the size of the hidden layer, to create an arbitrarily powerful model. How could we regularize this model such that it wouldn't (easily) overfit to noise in the training data? In order to extract complex patterns from the input example, each hidden unit must be able to rely on the behavior of its neighbors so to complement each other. We can make this co-adaption harder by randomly turning off hidden units. In other words, with probability 1/2, we set each hidden unit to have output zero, i.e., the unit is simply *dropped out*. This randomization is done anew for each training example. When a unit is turned off, the input/output weights coming in/going out will not be updated either. In a sense, we have dropped those units from the model in the context of the particular training example. This process makes is much harder for units to co-adapt. Instead, units can rely on the signal from their neighbors only if a larger numbers of neighbors support it (as about half of them would be present).

What shall we do at test time? Would we drop units as well? No, we can include all of them as we would without randomization. But we do need a slight modification. Since during training each hidden unit was present at about half the time, the signal to the units ahead is also about half of what it would be if all the hidden units were present. As a result, we can simply multiply the *outgoing* weights from each hidden unit by 1/2 to compensate. This works well in practice. Theoretical justification comes from thinking about the randomization during training as a way of creating large ensembles of networks (different configurations due to dropouts). This halving of outgoing weights is a fast approximation to the ensemble output (which would be expensive to compute).

Unsupervised neural networks – autoencoders

We have so far learned feed-forward neural networks in a *supervised* setting where each training input $x^{(t)} = [x_1^{(t)}, \ldots, x_d^{(t)}]^T \in \mathbb{R}^d$ is paired with the corresponding target output $y^{(t)} \in \{-1, 1\}$. The hidden layers (feature transformations) in such networks are adjusted through learning so as to improve classification performance. Here we focus on *unsupervised* learning where only input examples $x^{(t)}$ are available.

What kind of neural networks can we learn from $x^{(1)}, \ldots, x^{(n)}$ alone? We can think of the learning problem in this context as a form of compression. We map each x into a feature representation of lower dimension. This lower dimensional representation serves two purposes. First, it is a compressed version of the original example, summarizing it. The representation should retain key aspects of the example while omitting inessential detail. Second, as a feature representation of the example, we should be able to use it to approximately reconstruct the original example if desired. The reconstruction will not be perfect, of course, since the low dimensional feature representation has lost information (it is a summary) but it may still capture many of the key aspects. The feature representation should be learned so as to balance these two goals: 1) compression (how little information to retain) and 2) reconstruction (how well we can reconstitute each example).

Many unsupervised learning methods can be viewed in this manner. For example, we can think of k-means clustering as a form of compression. Suppose we have k clusters represented by cluster centroids $z^{(1)}, \ldots, z^{(k)}$. We can then "compress" each example x by mapping it to its closest centroid $\hat{j} = \text{argmin}_{j=1,\ldots,k} \|x - z^{(j)}\|^2$. The selected cluster $\hat{j}$ represents the example. It is a highly compressed representation (when k is small) since all that we retain is the cluster identity. The reconstruction step is therefore only approximate. We simply use the corresponding centroid as the reconstructed example $\hat{x} = z^{(\hat{j})}$. Note that it is critical to keep the feature representation – here cluster id – simple. For example, if we introduced one cluster per training example then the corresponding cluster id (the associated centroid) would perfectly reconstruct each training example. There's no compression. The feature representation must *summarize* the examples to be useful.

So, let's setup a feed-forward neural network in a manner that tries to summarize training examples. Each example is mapped to a feature representation in stages as in a typical feed-forward network. But now we also add a feed-forward reconstruction step where the feature representation is expanded in stages to an output layer that reconstructs the example. In this overall feed-forward architecture the feature representation compresses or summarizes the example, i.e., creates a *bottleneck*. This is illustrated in Figure 24 below. Neural networks that map examples to itself, via a bottleneck, are known as *autoencoders*.

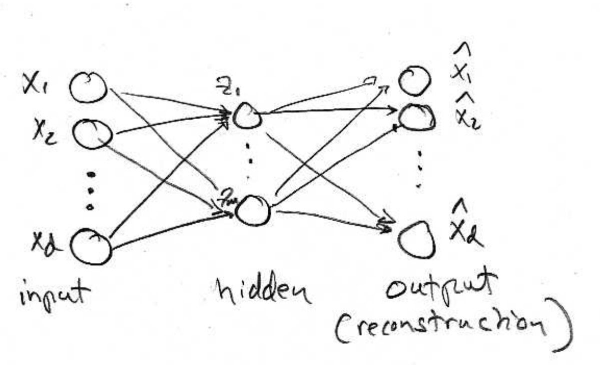


Figure 24: A simple autoencoder with one hidden layer

Simple autoencoder, PCA

If we assume a single hidden layer and rectified linear hidden units, our feed-forward architecture would evaluate, in stages

$$
\begin{aligned}
z_j &= \sum_{i=1}^{d} x_i W_{ij} + W_{0j}, \;\; j = 1, \ldots, m \;\; \text{(inputs to hidden units)} \quad (171) \\
f(z_j) &= \max\{0, z_j\}, \;\; j = 1, \ldots, m \;\; \text{(hidden unit outputs)} \quad (172) \\
\hat{x}_i &= \sum_{j=1}^{m} f(z_j) W'_{ji} + W'_{0i}, i = 1, \ldots, d \;\; \text{(reconstruction)} \quad (173)
\end{aligned}
$$

The parameters W and W' could be learned by minimizing the overall reconstruction error across training examples $x^{(1)}, \ldots, x^{(n)}$, i.e.,

$$\frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|x^{(t)} - \hat{x}^{(t)}\|^2 \tag{174}$$

Our goal is to understand what type of hidden layer representation the model would learn. In order to make progress towards answering this question, we will simplify the model drastically. We make the hidden units linear, i.e., $f(z) = z$ instead of rectified linear. We will also start with a single hidden unit, call it z. Then

$$\begin{aligned} z &= \sum_{i=1}^{d} x_i W_i + W_0 && (175)\\ \hat{x}_i &= zW_i' + W_{0i}', \;\; i = 1, \ldots, d && (176) \end{aligned}$$

It'll be helpful to change the notation a bit. Let $w = [W_1, \ldots, W_d]^T$ and $W_0 = -w^T\mu$ (if w is a non-zero vector then we can always find a vector μ such that $W_0 = -w^T\mu$). We will also write $w' = [W_1', \ldots, W_d']^T$ and $\mu' = [W_{01}', \ldots, W_{0d}']^T$ for the reconstruction part. As a result, in this new notation,

$$\begin{aligned} z &= w^T x + W_0 = w^T x - w^T\mu = w^T(x - \mu) \;\; \text{(scalar hidden feature)} && (177)\\ \hat{x} &= w'\,z + \mu' = w'w^T(x - \mu) + \mu' \;\; \text{(reconstruction)} && (178) \end{aligned}$$

You might suspect now that w is the first principal component, μ the mean of examples, and so on. Indeed, this is roughly how things go. But there are degrees of freedom that are unset. For example, we could multiply w by 2 and divide w' by 2 without affecting the reconstruction $\hat{x}$ at all. The setup here is *under-constrained* in this sense. We will set $\|w\| = 1$ to remove this particular degree of freedom (others remain). As a result, $w^T(x-\mu)$ becomes the length of the orthogonal projection of $(x-\mu)$ in the direction w. This is all that the hidden unit activation is telling us. How should we reconstruct x with this information? The most reasonable vector to produce is just the orthogonal projection as a vector, i.e., w multiplied by the length of the projection or $w(w^T(x - \mu)) = ww^T(x - \mu)$. We don't have any information about the other directions so best to assume they are zero. Since we subtracted μ from x before evaluating the hidden unit activation, we should also put it back at the time of reconstructing x. In other words, we should reconstruct by

$$\hat{x} = ww^T(x - \mu) + \mu \tag{179}$$

so that $\mu' = \mu$ and $w' = w$ where also $\|w\| = 1$. Here μ indeed plays the role of the mean of the examples. For brevity, we will set

$$\mu = \frac{1}{n}\sum_{t=1}^{n} x^{(t)} \tag{180}$$

without further proof. With these shortcuts (which were derived by relying on intuition rather than proof), we can now see more formally what the vector parameters w come out to be. To this end, we minimize

$$\begin{aligned}
\frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|x^{(t)}-\hat{x}^{(t)}\|^2 &= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|x^{(t)}-w'w^T(x^{(t)}-\mu)-\mu'\|^2 &(181)\\
&= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|x^{(t)}-ww^T(x^{(t)}-\mu)-\mu\|^2 &(182)\\
&= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|(x^{(t)}-\mu)-ww^T(x^{(t)}-\mu)\|^2 &(183)\\
&= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|(I-ww^T)(x^{(t)}-\mu)\|^2 &(184)
\end{aligned}$$

where we have used the fact that $\mu' = \mu$ and $w' = w$. Now, what is the matrix $I - ww^T$ when $\|w\| = 1$? For any vector v, $(I - ww^T)v$ removes the component of v in the direction of w and leaves the rest intact. ww^Tv in contrast keeps only the part parallel to w. In fact, $(I-ww^T)v$ and ww^Tv represent an orthogonal decomposition of vector v. Clearly then, for any v,

$$\|v\|^2 = \|(I-ww^T)v\|^2 + \|ww^Tv\|^2 \qquad (185)$$

Minimizing $\|(I - ww^T)v\|^2$ is therefore equivalent to *maximizing* $\|ww^Tv\|^2$ since their sum is constant. In other words, in our earlier context, we can instead maximize

$$\begin{aligned}
\frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}\|ww^T(x^{(t)}-\mu)\|^2 &= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}(w^T(x^{(t)}-\mu))^2 &(186)\\
&= \frac{1}{n}\sum_{t=1}^{n}\frac{1}{2}w^T(x^{(t)}-\mu)(x^{(t)}-\mu)^Tw &(187)\\
&= \frac{1}{2}w^T\underbrace{\left[\frac{1}{n}\sum_{t=1}^{n}(x^{(t)}-\mu)(x^{(t)}-\mu)^T\right]}_{\Sigma}w &(188)
\end{aligned}$$

were the first equality comes from the fact that $\|w\| = 1$ and $w^T(x - \mu)$ is a scalar. The second equality follows from $w^T(x - \mu) = (x - \mu)^Tw$ since these are just scalars (so their transposes are themselves). The resulting Σ, when μ is the sample mean, is the sample covariance matrix evaluated from the training examples. Thus the best setting of w is the largest eigenvector of the sample covariance matrix, i.e., it is the first *principal component*.

10 Unsupervised Learning, Clustering

In our previous lectures, we considered supervised learning scenarios, where we have access to both examples and the corresponding target labels or responses: $\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$. The goal was to learn a mapping from examples to labels that would work well on (generalize to) yet unseen examples. In contrast, here we have only examples $S_n = \{x^{(i)}, i = 1, \ldots, n\}$. What is the learning task now? The goal of *unsupervised learning* is to uncover useful structure in the data S_n such as identify groups or clusters of similar examples.

Clustering is one of the key problems in exploratory data analysis. Examples of clustering applications include mining customer purchase patterns, modeling language families, or grouping search results according to topics. Clustering can be also used for data compression. For example, consider vector quantization for image compression. A typical image consists of 1024*1024 pixels, where each pixel is represented by three integers ranging from 0 to 255 (8 bits), encoding red, green, and blue intensities of that point in the image. As a result, we need 24 bits to store each pixel, and the full image requires about 3MB of storage. One way to compress the image is to select only a few representative pixels and substitute each pixel with the closest representative. For example, if we use only 32 colors (5 bits), then we will need a codebook of 32 representative pixels (points in the red-green-blue color space). Now, instead of requiring 24bits for each pixel, we only use 5bits to store the identity of the closest representative pixel in our codebook. In addition, we need to store the codebook itself which has 32 points in the color space. Without compressing these further, each of the 32 representative pixels would require 24 bits to store. Taken together, the compressed image would require 640KB.

A bit more formally, the clustering problem can we written as:

Input: training set $S_n = \{x^{(i)}, i = 1, \ldots, n\}$, where $x^{(i)} \in R^d$, integer k
Output: a set of clusters $C_1, \ldots, C_k$.

For example, in the context of the previous example, each cluster is represented by one pixel (a point in color space). The cluster as a set would then consist of all the points in the color space that are closest to that representative. This example highlights the two ways of specify the output of the clustering algorithm. We can either return the groups (clusters) as sets, or we can return the representatives[4] that implicitly specify the clusters as sets. Which view is more appropriate depends on the clustering problem. For example, when clustering news, the output can be comprised of groups of articles about the same event. Alternatively, we can describe each cluster by its representative. In the news example, we may want to select a single news story for each event cluster. In fact, this output format is adopted in Google News.

Note that we have yet to specify any criterion for selecting clusters or the representatives. To this end, we must be able to compare pairs of points to determine whether they are indeed similar (should be in the same cluster) or not (should be in a different cluster). The comparison can be either in terms of similarity such as *cosine similarity* or dissimilarity as

[4] In the clustering literature, the terms "representative", "center" and "exemplar" are used interchangeably.

in Euclidean distance. Cosine similarity is simply the angle between two vectors (elements):

$$cos(x^{(i)}, x^{(j)}) = \frac{x^{(i)} \cdot x^{(j)}}{\|x^{(i)}\|\|x^{(j)}\|} = \frac{\sum_{l=1}^{d} x_l^{(i)} x_l^{(j)}}{\sqrt{\sum_{l=1}^{d} (x_l^{(i)})^2}\sqrt{\sum_{l=1}^{d} (x_l^{(j)})^2}} \tag{189}$$

Alternatively, we can focus on dissimilarity as in pairwise distance. In this lecture, we will primarily use squared Euclidian distance

$$dist(x^{(i)}, x^{(j)}) = \|x^{(i)} - x^{(j)}\|^2 = \sum_{l=1}^{d} (x_l^{(i)} - x_l^{(j)})^2 \tag{190}$$

but there are many alternatives. For example, in your homework (and in the literature) you may often encounter l_1 distance

$$dist(x^{(i)}, x^{(j)}) = \|x^{(i)} - x^{(j)}\|_1 = \sum_{l=1}^{d} |x_l^{(i)} - x_l^{(j)}| \tag{191}$$

The choice of which distance metric to use is important as it will determine the type of clusters you will find. A reasonable metric or similarity is often easy to find based on the application. Another issue with the metric is that the available clustering algorithms such as k-means discussed below may rely on a particular metric.

Once we have the distance metric, we can specify an objective function for clustering. In other words, we specify the cost of choosing any particular set of clusters or their representatives (a.k.a. centroids). The "optimal" clustering is then obtained by minimizing this cost. The cost is often cast in terms of *distortion* associated with individual clusters. For instance, the cost – distortion – associated with cluster C could be the sum of pairwise distances within the points in C, or the diameter of the cluster (largest pairwise distance). We will define the distortion here slightly differently: the sum of (squared) distances from each point in the cluster to the corresponding cluster representative z. For cluster C with centroid z, the distortion is defined as $\sum_{i \in C} \|x^{(i)} - z\|^2$. The cost of clustering $C_1, C_2, \ldots, C_k$, is simply the sum of costs of individual clusters:

$$cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) = \sum_{j=1 \ldots k} \sum_{i \in C_j} \|x^{(i)} - z^{(j)}\|^2 \tag{192}$$

Our goal is to find a clustering that minimizes this cost. Note that the cost here depends on both the clusters and how the representatives (centroids) are chosen for each cluster. It seems unnecessary to have to specify both clusters and centroids and, indeed, one will imply the other. We will see this below. Note also that we only consider valid clusterings $C_1, \ldots, C_k$, those that specify a partition of the indexes $\{1, \ldots, n\}$. In other words, each point must belong to one and only one cluster.

K-means

We introduced two ways to characterize the output of a clustering algorithm: the cluster itself or the corresponding representative (centroid). For some cost functions these two representations are interchangeable: knowing the representatives, we can compute the corresponding clusters and vice versa. In fact, this statement holds for the cost function introduced above. We can define clusters by their representatives:

$$C_j = \{i \in \{1, \ldots, n\} \; s.t. \; the \; closest \; representative \; of \; x^{(i)} \; is \; z^{(j)}\} \tag{193}$$

These clusters define an optimal clustering with respect to our cost function for a fixed setting of the representatives $z^{(1)}, \ldots, z^{(k)}$. In other words,

$$\begin{aligned} cost(z^{(1)}, \ldots, z^{(k)}) &= \min_{C_1,\ldots,C_k} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) && (194) \\ &= \min_{C_1,\ldots,C_k} \sum_{j=1\ldots k} \sum_{i \in C_j} \|x^{(i)} - z^{(j)}\|^2 && (195) \\ &= \sum_{i=1,\ldots,n} \min_{j=1,\ldots,k} \|x^{(i)} - z^{(j)}\|^2 && (196) \end{aligned}$$

where in the last expression we are simply assigning each point to its closest representative (as we should). Geometrically, the partition induced by the centroids can be visualized as as Voronoi partition of R^d, where R^d is divided into k convex cells. The cell is the region of space where the corresponding centroid z is the closest representative. See Figure 10.

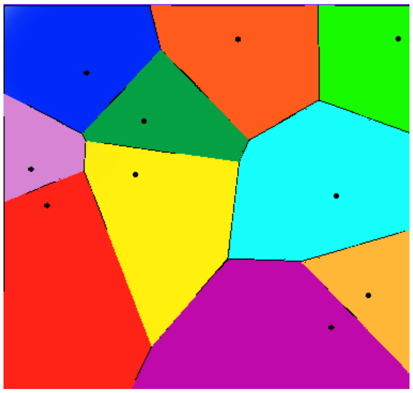


Figure 25: An example of Voronoi diagram

The K-means algorithm

Now, given an optimization criterion, we need to find an algorithm that tries to minimize it. Directly enumerating and selecting the best clustering out of all the possible clusterings is prohibitively expensive. We will instead rely here on an approximate method known as the k-means algorithm. This algorithm alternatingly finds best clusters for centroids, and best centroids for clusters. The iterative algorithm is given by

1. Initialize centroids $z^{(1)}, \ldots, z^{(k)}$

2. Repeat until there is no further change in cost

(a) for each j=1,...,k: $C_j = \{i \text{ s.t. } x^{(i)} \text{ is closest to } z^{(j)}\}$

(b) for each j=1,...,k: $z^{(j)} = \frac{1}{|C_j|}\sum_{i \in C_j} x^{(i)}$ (cluster mean)

Each iteration requires $\mathcal{O}(kn)$ operations.

Convergence The k-means algorithm does converge albeit not necessarily to a solution that is optimal with respect to the cost defined above. However, each iteration of the algorithm necessarily lowers the cost. Given that the algorithm alternates between choosing clusters and centroids, it will be helpful to look at

$$cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) \tag{197}$$

as the objective function for the algorithm. Consider $(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)})$ as the starting point. In the first step of the algorithm, we find new clusters $C'_1, C'_2, \ldots, C'_k$ corresponding to fixed centroids $z^{(1)}, \ldots, z^{(k)}$. These new clusters are chosen such that

$$\begin{aligned} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) &\overset{(a)}{\geq} \min_{C_1, \ldots, C_k} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) && (198) \\ &= cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) && (199) \end{aligned}$$

This is because $C'_1, C'_2, \ldots, C'_k$ are clusters induced from assigning each point to its closest centroid. No other clusters can achieve lower cost for these centroids. The inequality (a) is equality only when the algorithm converges. In the 2nd step of the algorithm, we fix the new clusters $C'_1, C'_2, \ldots, C'_k$ and find new centroids $z'^{(1)}, \ldots, z'^{(k)}$ such that

$$\begin{aligned} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) &\overset{(b)}{\geq} \min_{z^{(1)}, \ldots, z^{(2)}} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) && (200) \\ &= cost(C'_1, C'_2, \ldots, C'_k, z'^{(1)}, \ldots, z'^{(k)}) && (201) \end{aligned}$$

where the inequality in (b) is tight only when the centroids are already optimal for the given clusters, i.e., when the algorithm has converged. The problem of finding the new centroids decomposes across clusters. In other words, we can find them separately for each cluster. In particular, the new centroid $z'^{(j)}$ for a fixed cluster C'_j is found by minimizing

$$\sum_{i \in C'_j} \|x^{(i)} - z^{(j)}\|^2 \tag{202}$$

with respect to $z^{(j)}$. The solution to this is the mean of the points in the cluster

$$z'^{(j)} = \frac{1}{|C'_j|} \sum_{i \in C'_j} x^{(i)} \tag{203}$$

as specified in the k-means algorithm. We will explore this point further in a homework problem.

Now, taken together, (a) and (b) guarantee that the k-means algorithm monotonically decreases the objective function. As the cost has a lower bound (non-negative), the algorithm must converge. Moreover, after the first step of each iteration, the resulting cost is exactly Eq.(196), and it too will decrease monotonically.

The K-medoids algorithm

We had previously defined the cost function for the K-means algorithm in terms of squared Euclidean distance of each point $x^{(i)}$ to the closest cluster representative. We showed that, for any given cluster, the best representative to choose is the mean of the points in the cluster. The resulting cluster mean typically does not correspond to any point in the original dataset. The K-medoids algorithm operates exactly like K-means but, instead of choosing the cluster mean as a representative, it chooses one of the original points as a representative, now called an *exemplar*. Selecting exemplars rather than cluster means as representatives can be important in applications. Take, for example, Google News, where a single article is used to represent a news cluster. Blending articles together to evaluate the "mean" would not make sense in this context. Another advantage of K-medoids is that we can easily use other distance measures, other than the squared Euclidian distance.

The K-medoids objective is very similar to the K-means objective:

$$Cost(C^1, \ldots, C^k, z^{(1)}, \ldots, z^{(k)}) = \sum_{j=1}^{k} \sum_{i \in C^j} d(x^{(i)}, z^{(j)}) \tag{204}$$

The algorithm:

1. Initialize exemplars: $\{z^{(1)}, \ldots, z^{(k)}\} \subseteq \{x^{(1)}, \ldots, x^{(n)}\}$ (exemplars are k points from the original dataset)
2. Repeat until there is no further change in cost:
   (a) for each j: $C^j = \{i : x^{(i)}\text{'s closest exemplar is } z^{(j)}\}$
   (b) for each j: set $z^{(j)}$ to be the point in C^j that minimizes $\sum_{i \in C^j} d(x^{(i)}, z^{(j)})$

In order to update $z^{(j)}$ in step (b), we can consider each point in turn as a candidate exemplar and compute the associated cost. Among the candidate exemplars, the point that produces the minimum cost is chosen as the exemplar.

Initialization

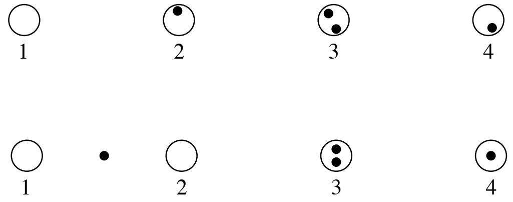
In the previous lecture, we demonstrated that the K-means algorithm monotonically decreases the cost (the same holds for the K-medoids algorithm). However, K-means (or K-medoids) only guarantees that we find a local minimum of the cost, not necessarily the optimum. The quality of the clustering solution can depend greatly on the initialization, as shown in the example below[5]. The example is tailored for K-means.

Given: N points in 4 clusters with small radius δ, and a large distance B between the clusters. The cost of the optimal clustering will be $\approx O(\delta^2 N)$. Now consider the following initialization:

After one iteration of K-means, the center assignment will be as follows:

This cluster assignment will not change during subsequent iterations. The cost of the resulting clustering is $O(B^2 N)$. Given that B can be arbitrary large, K-means produces a solution that is far from the optimal.

[5] This example is taken from Sanjoy Dasgupta.



One failure of the above initialization was that two centers were placed in close proximity to each other (see cluster 3) and therefore many points were left far away from any center/representative. One possible approach to fixing this problem is to pick k points that are far away from each other. In the example above, this initialization procedure would indeed yield one center per cluster. But this solution is sensitive to outliers. To correct this flaw, we will add some randomness to the selection process: the initializer will pick the k-centers one at a time, choosing each center at random from the set of remaining points. The probability that a given point is chosen as a center is proportional to that point's squared distance from the centers chosen already thereby steering them towards distinct clusters. This initialize procedure is known as the K-means++ initializer.

K-means++ initializer

```
pick x uniformly at random from the dataset, and set T=x
while $|T| \leq k$
pick x at random, with probability proportional to the cost $\min_{z \in T} ||x - z||^2$
$T = T \cup \{x\}$
```

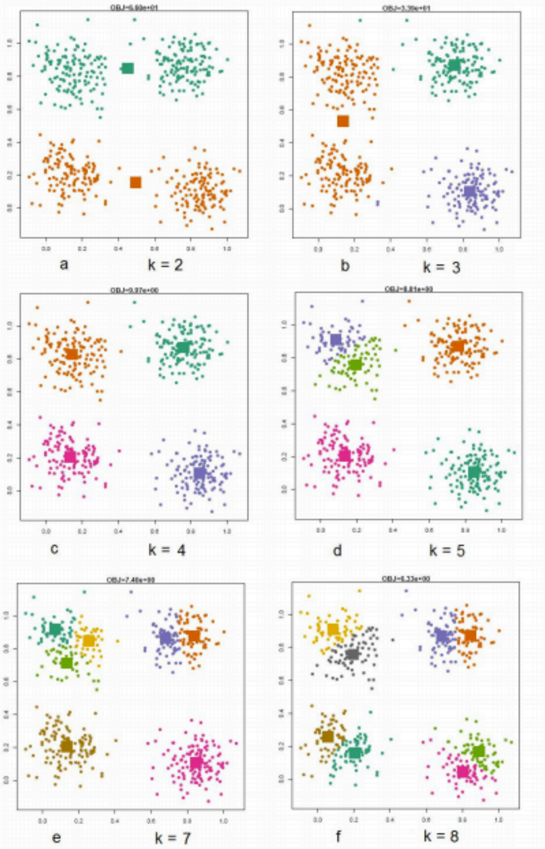
We can offer some guarantees for this initialization procedure:

K-means++ guarantee Let T be the initial set of centers chosen by K-means++. Let T^* be the set of optimal cluster centers. Then, $E|cost(T)| \leq cost(T^*)O(logK)$, where the expectation is over the randomness in the initialization procedure, and $cost(T) = \sum_{i=1}^{n} \min_{z \in T} \|x^{(i)} - z\|^2$

Choosing K

The selection of k greatly impacts the quality of your clustering solution. Figure 10 shows how the output of K-means changes as a function of k. In some applications, the desired k is intuitively clear based on the problem definition. For instance, if we wish to divide students into recitations based on their interests, we will choose k to be 4 (the number of recitation times for the class). In most problems, however, the optimal k is not given and we have to select it automatically.

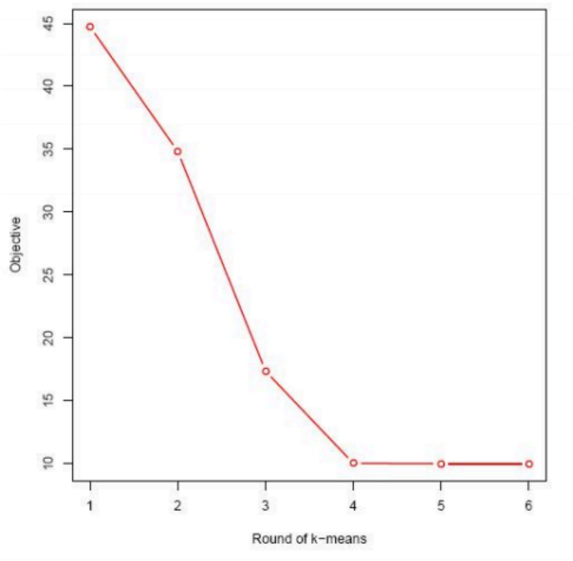
Let's start by understanding the connection between the number of clusters k and the cost function. If every point belongs to its own cluster, then the cost is equal to zero. At the other extreme, if all the points belong to a single cluster with a center z, then the cost is the maximum: $\sum_{i=1}^{n} ||x^{(i)} - z||^2$. Figure 10 shows how the cost decreases as a function of k. Notice that for $l = 4$, we observe a sharp decrease in cost. While the decrease continues for $k \geq 4$, it levels off. This observation motivates one of the commonly used heuristics for selecting k, called the "elbow method". This method suggests that we should choose the value of k that results in the highest relative drop in the cost (which corresponds to an



"elbow" in the graph capturing as a function of k). However, it may be hard to identify (or justify) such a sharp point. Indeed, in many clustering applications the cost decreases gradually.

There are a number of well-founded statistical criteria for choosing the number of clusters. These include, for example, the *minimum description length principle* (casting clustering as a communication problem) or the *Gap statistics* (characterizing how much we would expect the cost to decrease when no additional cluster structure exists). We will develop here instead a simpler approach based on assessing how useful the clusters are as inputs to other methods. In class, we used clustering to help semi-supervised learning.

In a semi-supervised learning problem we assume that we have access to a small set of labeled examples as well as a large amount of unannotated data. When the input vectors are high dimensional, and there are only a few labeled points, even a linear classifier would likely overfit. But we can use the unlabeled data to reduce the dimensionality. Consider, for instance, a document classification task where the goal is to label documents based on whether they involve a specific topic such as ecology. As you have seen in project 1, a typical mapping from documents to feature vectors is bag-of-words. In this case, the feature vectors would be of dimension close to the size of the English vocabulary. However, we can take advantage of the unannotated documents by clustering them into semantically coherent groups. The clustering would not tell us which topics each cluster involve, but



it would put similar documents in the same group, hopefully placing documents involving pertinent topics in distinct clusters. If so, knowing the group to which the document belongs should help us classify it. We could therefore replace the bag-of-words feature vector by one that indicates only to which cluster the document belongs to. More precisely, given k clusters, a document that belongs to cluster j can be represented by a k dimensional vector with the j-th coordinate equal to 1 and the rest set to zero. This representation is admittedly a bit coarse – all the documents in the same cluster will be assigned the same feature vector, and therefore end up with the same label. A bit better representation would be to compile the feature vectors by appending relative distances of the document to the k clusters. In either case, we obtain low dimensional feature vectors that can be more useful for topic classification. At the very least, the lower dimensionality will guard against overfitting. But how to choose k in this case? Note that none of the training labels were used to influence what the clusters were. As a result, we can use cross-validation, with the k dimensional feature vectors, to get a sense of how well the process generalizes. We would select k (the number of clusters) that minimizes the cross-validation error.

11 Generative Models and Mixtures

So far we have primarily focused on classification where the goal is to find a separator that sets apart two classes of points. The internal structure of the points in each class is not directly captured or cared for by the classifier. In fact, datasets with very different structures (e.g., multiple clusters) may have exactly the same separator for classification. Intuitively, understanding the structure of data should be helpful for classification as well.

The idea of generative models is that we specify a mechanism by which we can generate (sample) points such as those given in the training data. This is a powerful idea that goes beyond classification, and allows us to automatically discover many hidden mechanism that underly the data. We will start here with a brief introduction to the type of distributions we will use, and then focus on mixture models.

Spherical Gaussian

Consider a cluster of points shown in Figure 26. To summarize this data, our distribution should capture (1) the center of the group (mean); (2) how spread the points are from the center. The simplest model, one that makes fewest assumptions above and beyond the mean and the spread, is a Gaussian. In probabilistic terms, we assume that points $x \in \mathcal{R}^d$ are generated as samples from a *spherical* Gaussian distribution:

$$P(x|\theta) = N(x; \mu, \sigma^2 I) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp(-\frac{1}{2\sigma^2}\|x-\mu\|^2) \tag{205}$$

where d is the dimension. This is a simple spherically symmetric distribution around the mean (centroid) μ. The probability of generating points away from the mean μ decreases the same way (based on the squared distance) regardless of the direction. A typical representation of this distribution is by a circle centered at the mean μ with radius σ (called the standard deviation). See Figure 26 below. σ^2 is the average squared variation of coordinates of x from the coordinates of the mean μ (see below). The two parameters, μ and σ^2, summarize how the data points in the cluster are expected to vary. As a density, $N(x; \mu, \sigma^2 I)$ integrates to one over $\mathcal{R}^d$.

Given a training set of points $S_n = \{x^{(t)}, t = 1, \ldots, n\}$, we can estimate the parameters of the Gaussian distribution to best match the data. Note that we can do this regardless of what the points look like (there may be several clusters or just one). We are simply asking what is the best Gaussian that fits the data. The criterion we use is *maximum likelihood* or ML for short. We evaluate the probability of generating all the data points, each one independently. This means that the likelihood of the training data is a product

$$L(S_n; \mu, \sigma^2) = \prod_{t=1}^{n} N(x^{(t)}; \mu, \sigma^2 I) \tag{206}$$

Since the training data S_n is fixed (given), we view this as a function of the parameters μ and σ^2. The higher the value of $L(S_n; \mu, \sigma^2)$ we can find, the better we think the Gaussian fits the data.

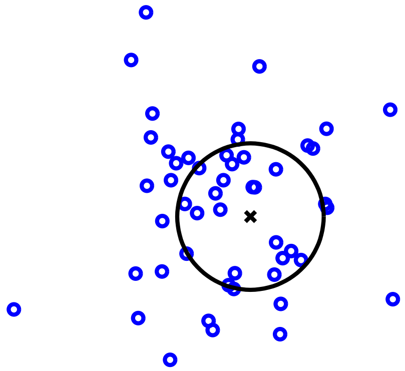


Figure 26: Samples from a spherical Gaussian distribution and the corresponding representation in terms of the mean (center) and the standard deviation (radius).

To maximize the likelihood, it is convenient to maximize the log-likelihood instead. In other words, we maximize

$$\begin{aligned} l(S_n;\mu,\sigma^2) &= \sum_{t=1}^{n} \log N(x^{(t)};\mu,\sigma^2 I) && (207)\\ &= \sum_{t=1}^{n} \left[-\frac{d}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\|x^{(t)}-\mu\|^2 \right] && (208)\\ &= -\frac{nd}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}\|x^{(t)}-\mu\|^2 && (209) \end{aligned}$$

By setting $\partial/\partial\mu \; l(S_n;\mu,\sigma^2)=0$, $\partial/\partial\sigma \; l(S_n;\mu,\sigma^2)=0$, and solving for the parameters, we obtain the ML estimates

$$\hat{\mu} = \frac{1}{n}\sum_{t=1}^{n} x^{(t)}, \quad \hat{\sigma}^2 = \frac{1}{dn}\sum_{t=1}^{n}\|x^{(t)}-\hat{\mu}\|^2 \tag{210}$$

In other words, the mean μ is simply the sample mean (cf. the choice of centroid for k-means), and σ^2 is the average squared deviation from the mean, averaged across the points and across the d-dimensions (because we use the same σ^2 for each dimension).

Mixture models

Consider data shown in Figure 27 where the points are divided into two or more clusters. A single Gaussian is no longer a good model for this data. Instead, we can describe the data using two Gaussians, one for each cluster. The model should include the different locations and (possibly different) spreads of the two Gaussians, but also how many points are in each cluster (mixing proportions). Models built from this perspective are known as *mixture models*

Mixture models assume a two-stage generative process: first we select the type of point to generate (which cluster the point belongs to), and then we generate the point from the

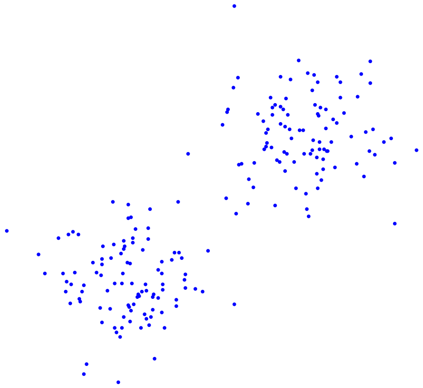


Figure 27: Clusters of points.

corresponding model (cluster). You can think of mixture models as probabilistic extensions of k-means clustering where we actually model what the clusters look like, their sizes, permitting also overlapping clusters.

A mixture of spherical Gaussians

Assuming there are exactly k clusters (this is our hypothesis, not necessarily true) we would define

$$N(x; \mu^{(j)}, \sigma_j^2 I), \quad j = 1, \ldots, k \tag{211}$$

and have to somehow estimate $\mu^{(1)}, \ldots, \mu^{(k)}$, $\sigma_1^2, \ldots, \sigma_k^2$ without knowing a priori where the clusters are, i.e., which points belong to which cluster. Since the clusters may vary by size as well, we also include parameters $p_1, \ldots, p_k$ (mixing proportions) that specify the fraction of points we would expect to see in each cluster. Note that k-means clustering did not include either different spreads (different σ_i^2's) nor different a priori sizes of clusters (different p_i's).

So how do we generate data points from the mixture? We first sample index j to see which cluster we should use. In other words, we sample i from a multinomial distribution governed by $p_1, \ldots, p_k$, where $\sum_{j=1}^{k} p_j = 1$. Think of throwing a biased k-faced die. Larger p_i means that we generate more points from that clusters. Once we know the cluster, we can sample x from the corresponding Gaussian. More precisely,

$$j \sim \text{Multinomial}(p_1, \ldots, p_k) \tag{212}$$

$$x \sim P(x|\mu^{(j)}, \sigma_j^2) \tag{213}$$

Figure 28 below shows data generated from the mixture model with colors identifying the clusters. We have also drawn the corresponding Gaussians, one for each cluster, as well as the prior fractions[6] (mixing proportions) p_j

[6] For this figure, the prior frequencies appear to match exactly the cluster sizes. This is not true in general, only on average, as the labels are sampled.

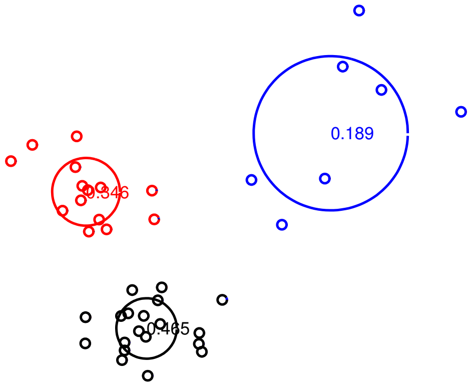


Figure 28: Samples from a mixture of Gaussian distribution with colors indicating the sampled cluster labels. The Figure also shows the corresponding Gaussian cluster models, included the prior cluster frequencies as numbers.

We typically don't have labels identifying the clusters. Indeed, the main use of mixture models (as in clustering) is to try to uncover these hidden labels, i.e., find the underlying clusters. To this end, we must evaluate the probability that each data point x could come as a sample from our mixture model, and adjust the model parameters so as to increase this probability. Each x could have been generated from any cluster, just with different probabilities. So, to evaluate $P(x|\theta)$, where θ specifies all the parameters in our mixture model

$$\theta = \{\mu^{(1)}, \ldots, \mu^{(k)}, \sigma_1^2, \ldots, \sigma_k^2, p_1, \ldots, p_k\}, \tag{214}$$

we must sum over all the alternative ways we could have generated x (all the ways that Eq.(213) could have resulted in x). In other words,

$$P(x|\theta) = \sum_{j=1}^{k} p_j N(x; \mu^{(j)}, \sigma_j^2 I) \tag{215}$$

This is the mixture model we must estimate from data $S_n = \{x^{(t)}, t = 1, \ldots, n\}$. It is not easy to resolve where to place the clusters, and how they should be shaped. We'll start with a simpler problem of estimating the mixture from labeled points, then generalize the solution to estimated mixtures from S_n alone.

Estimating mixtures: labeled case

If our data points came labeled, i.e., each point would be assigned to a single cluster, we could estimate our Gaussian models as before. In addition, we could evaluate the cluster sizes just based on the actual numbers of points. For later utility, let's expand on this a bit. Let $\delta(j|t)$ be an indicator that tells us whether $x^{(t)}$ should be assigned to cluster j. In other words,

$$\delta(j|t) = \begin{cases} 1, & \text{if } x^{(t)} \text{ is assigned to } j \\ 0, & \text{otherwise} \end{cases} \tag{216}$$

Using this notation, our maximum likelihood objective is

$$\sum_{t=1}^{n}\left[\sum_{j=1}^{k}\delta(j|t)\log(\,p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\,)\right]=\sum_{j=1}^{k}\left[\sum_{t=1}^{n}\delta(j|t)\log(\,p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\,)\right] \tag{217}$$

where, in the first expression, the inner summation over clusters simply selects the Gaussian that we should use to generate the corresponding data point, consistent with the assignments. In the second expression, we exchanged the summations to demonstrate that the Gaussians can be solved separately from each other, as in the single Gaussian case. Note that we also include p_j in generating each point, i.e., the probability that we would a priori select cluster j for point $x^{(t)}$. The ML solution based on labeled points is given by

$$\hat{n}_j = \sum_{t=1}^{n}\delta(j|t) \text{ (number of points assigned to cluster } j) \tag{218}$$

$$\hat{p}_j = \frac{\hat{n}_j}{n} \text{ (fraction of points in cluster } j) \tag{219}$$

$$\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j}\sum_{t=1}^{n}\delta(j|t)x^{(t)} \text{ (mean of points in cluster } j) \tag{220}$$

$$\hat{\sigma}_j^2 = \frac{1}{d\hat{n}_j}\sum_{t=1}^{n}\delta(j|t)\|x^{(t)}-\hat{\mu}^{(j)}\|^2 \text{ (mean squared spread in cluster } j) \tag{221}$$

Estimating mixtures: the EM-algorithm

Our goal is to maximize the likelihood that our mixture model generated the data. In other words, on a log-scale, we try to maximize

$$l(S_n;\theta)=\sum_{t=1}^{n}\log P(x^{(t)}|\theta)=\sum_{t=1}^{n}\log\left(\sum_{j=1}^{k}p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\right) \tag{222}$$

with respect to the parameters θ. Unfortunately, the summation inside the logarithm makes this a bit nasty to optimize. More intuitively, it is hard to consider different arrangements of k Gaussians that best explain the data.

Our solution is an iterative algorithm known as the *Expectation-Maximization algorithm* or the EM-algorithm for short. The trick we use is to return the problem back to the simple labeled case. In other words, we can use the current mixture model to assign examples to clusters (see below), then re-estimate each cluster model separately based on the points assigned to it, just as in the labeled case. Since the assignments were based on the current model, and the model was just improved by re-estimating the Gaussians, the assignments would potentially change as well. The algorithm is therefore necessarily iterative. The setup is very analogous to k-means. However, here we cannot fully assign each example to a single cluster. We have to entertain the possibility that the points were generated by different cluster models. We will make soft assignments, based on the relative probabilities that each cluster explains (can generate) the point.

We need to first initialize the mixture parameters. For example, we could initialize the means $\mu^{(1)}, \ldots, \mu^{(k)}$ as in the k-means algorithm, and set the variances σ_j^2 all equal to the overall data variances:

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{t=1}^{n} \|x^{(t)} - \hat{\mu}\|^2 \tag{223}$$

where $\hat{\mu}$ is the mean of all the data points. This ensures that the Gaussians can all "see" all the data points (spread is large enough) that we do not a priori assign points to specific clusters too strongly. Since we have no information about the cluster sizes, we will set $p_j = 1/k$, $j = 1, \ldots, k$.

The EM-algorithm is then defined by the following two steps.

- **E**-step: softly assign points to clusters according to the posterior probabilities

  $$p(j|t) = \frac{p_j N(x; \mu^{(j)}, \sigma_j^2 I)}{P(x|\theta)} = \frac{p_j N(x; \mu^{(j)}, \sigma_j^2 I)}{\sum_{l=1}^{k} p_l N(x; \mu^{(l)}, \sigma_l^2 I)} \tag{224}$$

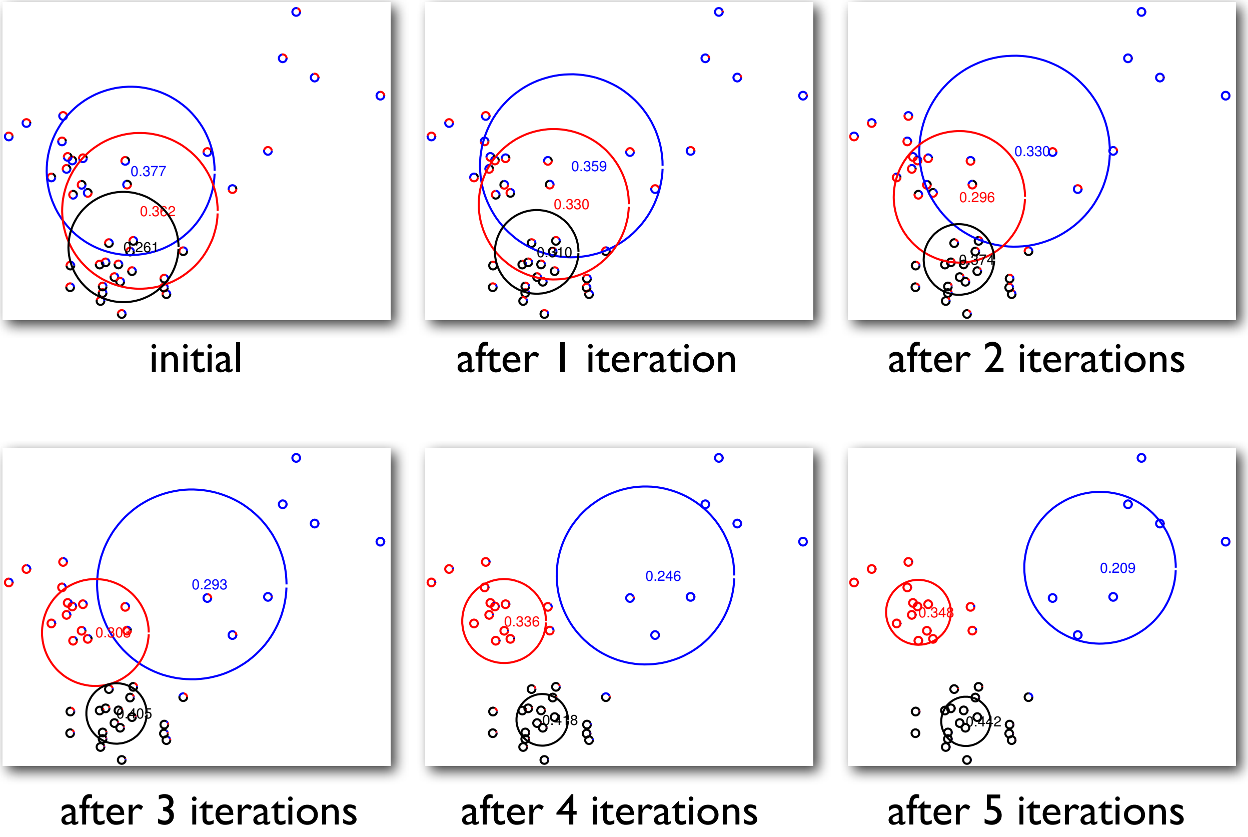
  Here $\sum_{j=1}^{k} p(j|t) = 1$. These are exactly analogous to (but soft versions of) $\delta(j|t)$ in the labeled case. Each point $x^{(t)}$ is assigned to cluster j with weight $p(j|t)$. The larger this weight, the more strongly we believe that it was cluster j that generated the point.

- **M**-step: Once we have $p(j|t)$, we pretend that we were given these assignments (as softly labeled examples) and can use them to estimate the Gaussians separately, just as in the labeled case.

  $$\begin{aligned}
  \hat{n}_j &= \sum_{t=1}^{n} p(j|t) \text{ (effective number of points assigned to cluster } j) && (225)\\
  \hat{p}_j &= \frac{\hat{n}_j}{n} \text{ (fraction of points in cluster } j) && (226)\\
  \hat{\mu}^{(j)} &= \frac{1}{\hat{n}_j} \sum_{t=1}^{n} p(j|t) x^{(t)} \text{ (weighted mean of points in cluster } j) && (227)\\
  \hat{\sigma}_j^2 &= \frac{1}{d\hat{n}_j} \sum_{t=1}^{n} p(j|t) \|x^{(t)} - \hat{\mu}^{(j)}\|^2 \text{ (weighted mean squared spread)} && (228)
  \end{aligned}$$

  We will then use these parameters in the E-step again, resulting in revised soft assignments $p(j|t)$, and iterate.

This simple algorithm is guaranteed to monotonically increase the log-likelihood of the data under the mixture model (cf. k-means). Just as in k-means, however, it may only find a locally optimal solution. But it is less "brittle" than k-means due to the soft assignments. Figure 29 below shows an example of running a few steps of the EM-algorithm. The points are colored based on the soft assignments in the E-step. Initially, many of the points are assigned to multiple clusters. The assignments are clarified (mostly in one cluster) as the algorithm finds where the clusters are.



initial

after 1 iteration

after 2 iterations

after 3 iterations

after 4 iterations

after 5 iterations

Figure 29: An example of running the EM-algorithm for five iterations

12 Bayesian networks

Overview

Bayesian networks are generative probability models that were developed for representing and using probabilistic information. A Bayesian network consists of random variables represented as nodes in the graph and arcs between the nodes representing dependences among the variables. Each Bayesian network is a combination of the graph (qualitative description) and the distribution over the variables (quantitative description).

All generative models involve variables. For example, the choice of mixture component is a variable, the state in a Hidden Markov Model (HMM) is a variable, and so on. How we select values for these variables is governed by a probability distribution. For example, mixture models specify a probability distribution over the selection of the mixture component as well as the (Gaussian) output variable. HMMs specify a distribution over the sequence of hidden states as well as the corresponding observation symbols. As generative models, Bayesian networks subsume mixture models, Hidden Markov Models, and many others. In fact, Bayesian networks provide a simple language for specifying generative probability models.

There are two parts to any Bayesian network model: 1) directed graph over the variables and 2) the associated probability distribution. The graph represents qualitative information about the random variables (conditional independence properties), while the associated probability distribution, consistent with such properties, provides a quantitative description of how the variables relate to each other. If we already have the distribution, as we have for mixture models or HMMs, why do we need the graph? The graph structure serves two important functions. First, it explicates the properties about the underlying distribution that would be otherwise hard to extract from a given distribution. For example, it tells us whether two sets of variables are independent from each other, and in which scenarios (known values for some variables). The graph is a compact summary of such statements. Given that the graph constraints the distribution, it affects how we can generate data. As a result, the graph structure can be learned from available data, i.e., we can explicitly learn qualitative properties from data. Second, since the graph pertains to independence properties about the random variables, it is very useful for understanding how we can use the probability models efficiently to evaluate various marginal and conditional properties.

This is exactly why we were able to carry out efficient computations in HMMs. The forward-backward algorithms relied on simple Markov properties which are independence properties, and these are generalized in Bayesian networks. We can make use of independence properties whenever they are explicit in the model.

Bayesian networks: examples, properties

Let's start with a simple example model over three binary variables. We imagine that two people are flipping coins independently from each other. The resulting values of their unbiased coin flips are stored in binary (H/T) variables X_1 and X_2. Another person checks whether the coin flips resulted in the same value and the outcome of the comparison is a binary (T/F) variable $X_3 = [\![X_1 = X_2]\!]$ (logical true/false). We will first construct the distribution, then look at how we should represent it as a graph.

The two coin flips are governed by simple uniform probability distributions. For example, $P(X_1 = H) = 0.5$ and $P(X_1 = T) = 0.5$. We can represent these probabilities as tables

$$X_1 : \begin{array}{|cc} H & T \\ \hline 0.5 & 0.5 \end{array}, \quad X_2 : \begin{array}{|cc} H & T \\ \hline 0.5 & 0.5 \end{array} \tag{229}$$

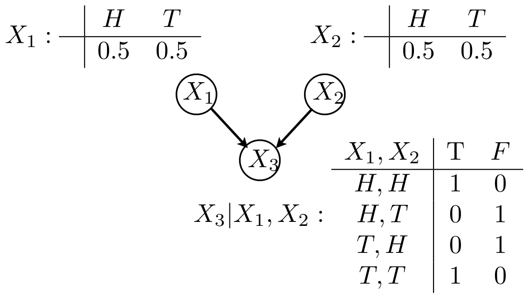
where each row in the table must sum to one. The value of X_3, on the other hand, depends on (in fact, is a function of) X_1 and X_2 and cannot be determined until we know which values X_1 and X_2 take. We must therefore specify a conditional distribution $P(X_3 = x_3|X_1 = x_1, X_2 = x_2)$ for this variable. The conditional probability can also be represented as a table where we introduce a row for each possible setting of X_1 and X_2.

$$X_3|X_1, X_2 : \begin{array}{c|cc} X_1, X_2 & \mathrm{T} & F \\ \hline H,H & 1 & 0 \\ H,T & 0 & 1 \\ T,H & 0 & 1 \\ T,T & 1 & 0 \end{array} \tag{230}$$

Again, each row of the probability table sums to one. Note that the probability values are extreme valued (zero and one) because X_3 is a function of X_1 and X_2. So, for example, $P(X_3 = T|X_1 = H, X_2 = H) = 1$ while $P(X_3 = F|X_1 = H, X_2 = H) = 0$, the first row in the above table. Now, since the two coins are flipped independently of each other, we can write the joint distribution over the three variables as

$$P(X_1 = x_1, X_2 = x_2, X_3 = x_3) = P(X_1 = x_1)P(X_2 = x_2)P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{231}$$

In order to represent this as a Bayesian network, we will use a directed graph over the variables X_1, X_2, and X_3 in addition to the distribution. The nodes in the graph represent variables while the directed edges specify dependences, i.e., whether one variable directly *depends on* another. We know in this example that X_1 and X_2 do not directly depend on each other, while X_3 depends on both X_1 and X_2. As a result, the directed graph for this model is as given by Figure 30.



X_1 :

H	T
0.5	0.5

X_2 :

H	T
0.5	0.5

$X_3|X_1, X_2$:

X_1, X_2	T	F
H,H	1	0
H,T	0	1
T,H	0	1
T,T	1	0

Figure 30: a) A directed graph for the coin toss example with the associated conditional probability tables

Typically, we would write down the distribution in response to the graph rather than the other way around. In fact, how the distribution *factors* is determined directly by the graph. We need a bit of terminology for this. In the graph, X_1 is a *parent* of X_3 since there's a directed edge from X_1 to X_3 (the value of X_3 depends on X_1). Analogously, we can say that X_3 is a *child* of X_1. Now, X_2 is also a parent of X_3 so that the value of X_3 depends on both X_1 and X_2. We will discuss later what the graph means more formally (it captures independence properties). For now, we just note that Bayesian networks always define acyclic graphs (no directed cycles) and represent how values of the variables depend on their parents, i.e., how we can generate values for the variables. Any joint distribution consistent with the graph, i.e., any distribution we could imagine associating with the graph, has to be able to be written as a product of conditional probabilities of each variable given its parents. If a variable has no parents (as is the case with X_1) then we just write $P(X_1 = x_1)$. Eq.(231) is exactly a product of conditional probabilities of variables given their parents.

Marginal independence and induced dependence

Let's analyze the properties of the simple model a bit. For example, what is the marginal probability over X_1 and X_2? This is obtained from the joint simply by summing over the values of X_3

$$P(X_1 = x_1, X_2 = x_2) = \sum_{x_3} P(X_1 = x_1)P(X_2 = x_2)P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{232}$$

$$= P(X_1 = x_1)P(X_2 = x_2)\sum_{x_3} P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{233}$$

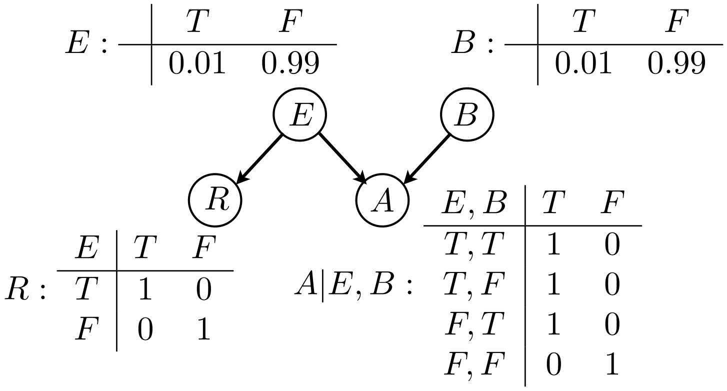
$$= P(X_1 = x_1)P(X_2 = x_2) \tag{234}$$

Thus X_1 and X_2 are *marginally independent* of each other (a product distribution means that the variables are independent). In other words, if we don't know the value of X_3 then there's nothing that ties the coin flips together (they were, after all, flipped independently in the description). This is also a property we could have extracted directly from the graph. We will provide shortly a formal way of deriving this type of independence properties from the graph.

Another typical property of probabilistic models is *induced dependence*. Suppose now that the coins X_1 and X_2 were flipped independently but we don't know their outcomes. All we know that $X_3 = T$, i.e., that the outcomes where identical. What do we know about X_1 and X_2 in this case? We know that either $X_1 = X_2 = H$ or $X_1 = X_2 = T$. So their values are clearly *dependent*. The dependence was *induced by additional knowledge*, in this case observing the value of X_3. This is again a property we could have read off directly from the graph (explained below). Both marginal independence and induced dependence are typical properties of realistic models.

Explaining away

Another typical phenomenon that probabilistic models can capture is *explaining away*. Consider the following typical example (Pearl 1988) in Figure 31. We have four variables A, B, E, and R capturing possible causes for why a burglary alarm went off. All the variables are binary (T/F) and, for example, $A = T$ means that the alarm went off. In our example here all the observed values are T (property is true). In general, observations in the



E:

T	F
0.01	0.99

B:

T	F
0.01	0.99

R:

E	T	F
T	1	0
F	0	1

$A\|E,B$:

E,B	T	F
T,T	1	0
T,F	1	0
F,T	1	0
F,F	0	1

Figure 31: Alarm example with four variables, E, B, R, and A representing true/false values of earthquake, burglary, radio report, and alarm, respectively. The corresponding probability tables are given next to the variables.

graph would be represented by shaded nodes. We assume that earthquakes ($E = T$) and burglaries ($B = T$) are equally unlikely events $P(E = 1) = P(B = 1) = 0.01$. Alarm is likely to go off only if either $E = 1$ or $B = 1$ or both. Moreover, either event will trigger the alarm so that $P(A = T|E, B) = 1$ whenever either $E = T$ or $B = T$ or $E = B = T$, and $P(A = T|E, B) = 0$ when $E = B = F$. An earthquake ($E = T$) is likely to be followed by a radio report ($R = T$) where $P(R = T|E = T) = 1$, and we assume that the report never occurs unless an earthquake actually took place: $P(R = T|E = F) = 0$. Based on the graph, or based on how we constructed the distribution, we can write down the joint distribution over all the binary variables as

$$\begin{aligned} & P(E=e, B=b, A=a, R=r) = \\ & = \quad P(E=e)P(B=b)P(A=a|E=e,B=b)P(R=r|E=e) \end{aligned} \tag{235}$$

Note that it again factors as a product of "variable given its parents".

What do we believe about the values of the variables if we only observe that the alarm went off ($A = T$)? At least one of the potential causes $E = T$ or $B = T$ should have occurred. However, since both are unlikely to occur by themselves, we are basically left with either $E = T$ or $B = T$ but (most likely) not both. We therefore have two alternative or competing explanations for the observation and both explanations are equally likely. We can evaluate the posterior probability that there was a burglary $P(B = T|A = T)$ as follows.

Let's first evaluate the marginal probability over the variables we are interested in:

$$\begin{aligned}
& P(B=b, A=T) = \\
& = \sum_{e\in\{T,F\}} \sum_{r\in\{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b)P(R=r|E=e) && (236)\\
& = \sum_{e\in\{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b) \sum_{r\in\{T,F\}} P(R=r|E=e) && (237)\\
& = \sum_{e\in\{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b) && (238)\\
& = P(B=b) \sum_{e\in\{T,F\}} P(E=e)P(A=T|E=e,B=b) && (239)
\end{aligned}$$

Note how the radio report (R) dropped out since it is a variable downstream from E, and we did not observe its value. It represents an observation we could have made but didn't. Such "imagined" possibilities will not affect our calculations. Now,

$$P(B=T|A=T) = \frac{P(B=T, A=T)}{\sum_{b\in\{T,F\}} P(B=b, A=T)} \tag{240}$$

and evaluates just slightly above 0.5. Why not exactly 0.5? Because there's a slight chance that both $B=T$ and $E=T$, not just one or the other.

If we now hear, in addition, that there was a radio report about an earthquake, we believe that $E=T$ because $R=T$ only if $E=T$. As a result, $E=T$ perfectly explains the alarm $A=T$, removing any evidence about $B=T$. In other words, the additional observation about the radio report *explained away* the evidence for $B=T$. Thus, $P(B=T|A=T,R=T) = P(B=T) = 0.01$ (prior probability) whereas $P(E=T|A=T,R=T) = 1$.

Note that we have implicitly captured in our calculations here that R and B are *dependent* given $A=T$ (induced dependence). If they were not, we would not be able to learn anything about the value of B as a result of also observing $R=T$. Here the effect is drastic and the variables are strongly dependent. We could have, again, deduced this dependence from the graph directly. In the next lecture, we will look at independence a bit more formally.

13 Hidden Markov Models

Motivation

In many practical problems, we would like to model pairs of sequences. Consider, for instance, the task of part-of-speech (POS) tagging. Given a sentence, we would like to compute the corresponding tag sequence:

Input:“Faith is a fine invention”
Output:“Faith/N is/V a/D fine/A invention/N”

More generally, a *sequence labeling problem* involves mapping a sequence of observations $x_1, x_2, \ldots, x_n$ into a sequence of tags $y_1, y_2, \ldots, y_n$. In the example above, every word x is tagged by a single label y. One possible approach for solving this problem would be to label each word independently. For instance, a classifier could predict a part-of-speech tag based on the word, its suffix, its position in the sentence, etc. In other words, we could construct a feature vector on the basis of the observed “context” for the tag, and use the feature vector in a linear classifier. However, tags in a sequence are dependent on each other and this classifier would make each tagging decision independently of other tags. We would like our model to directly incorporate these dependencies. For instance, in our example sentence, the word *"fine"* can be either noun (N), verb (V) or adjective (A). The label V is not suitable since a tag sequence "D V" is very unlikely. Today, we will look at a model – a Hidden Markov Model – that allows us to capture some of these correlations.

Generative Tagging Model

Assume a finite set of words Σ and a finite set of tags $\mathcal{T}$. Define S to be the set of all sequence tag pairs $(x_1, \ldots, x_n, y_1, \ldots, y_n)$, $x_i \in \Sigma$ and $y_i \in \mathcal{T}$ for $i = 1 \ldots n$. S here contains sequences of different lengths as well, i.e., n varies as well. A generative tagging model is a probability distribution p over pairs of sequences:

- $p(x_1, \ldots, x_n, y_1, \ldots, y_n) \geq 0 \ \forall (x_1, \ldots, x_n, y_1, \ldots, y_n) \in S$
- $\sum_{(x_1,\ldots,x_n,y_1,\ldots,y_n)\in S} p(x_1, \ldots, x_n, y_1, \ldots, y_n) = 1$

If we have such a distribution, then we can use it to predict the most likely sequence of tags $y_1, \ldots, y_n$ for any observed sequence of words $x_1, \ldots, x_n$, as follows

$$f(x_1, \ldots, x_n) = \text{argmax}_{y_1,\ldots,y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_n) \tag{241}$$

where we view f as a mapping from word sequences to tags.

Three key questions:

- How to specify $p(x_1, \dots, x_n, y_1, \dots, y_n)$ with a few number of parameters (degrees of freedom)
- How to estimate the parameters in this model based on observed sequences of words (and tags).
- How to predict, i.e., how to find the most likely sequence of tags for any observed sequence of words: evaluate $\text{argmax}_{y_1,\dots,y_n} p(x_1, \dots, x_n, y_1, \dots, y_n)$

Model definition

Let $X_1, \dots, X_n$ and $Y_1, \dots, Y_n$ be sequences of random variables of length n. We wish to specify a joint probability distribution

$$P(X_1 = x_1, \dots, X_n = x_n, Y_1 = y_1, \dots, Y_n = y_n) \tag{242}$$

where $x_i \in \Sigma$, $y_i \in \mathcal{T}$. For brevity, we will write it as $p(x_1, \dots, x_n, y_1, \dots, y_n)$, i.e., treat it as a function of values of the random variables without explicating the variables themselves. We will define one additional random variable Y_{n+1}, which always takes the value STOP. Since our model is over variable length sequences, we will use the end symbol to model when to stop. In other words, if we observe $x_1, \dots, x_n$, then clearly the symbol after $y_1, \dots, y_n$, i.e., y_{n+1}, had to be STOP (otherwise we would have continued generating more symbols).

Now, let's start by rewriting the distribution a bit according to general rules that apply to any distribution. The goal is to put the distribution in a form where we can easily explicate our assumptions. First,

$$p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) = p(y_1, \dots, y_{n+1}) p(x_1, \dots, x_n | y_1, \dots, y_{n+1}) \qquad \text{(chain rule)}$$

Then we will use the chain rule repeatedly along the sequence of tags

$$p(y_1, \dots, y_{n+1}) = p(y_1) p(y_2|y_1) p(y_3|y_1, y_2) \dots p(y_{n+1}|y_1, \dots, y_n) \qquad \text{(chain rule)}$$

So far, we have made no assumptions about the distribution at all. Since we don't expect tags to have very long dependences along the sequence, we will simply say that the next tag only depends on the current tag. In other words, we will "drop" the dependence on tags further back

$$\begin{aligned} p(y_1, \dots, y_{n+1}) &\approx p(y_1) p(y_2|y_1) p(y_3|y_2) \dots p(y_{n+1}|y_n) \qquad \text{(independence assumption)} \\ &= \prod_{i=1}^{n+1} p(y_i|y_{i-1}) \end{aligned}$$

Put another way, we assume that the tags form a Markov sequence (future tags are independent of the past tags given the current one). Let's now make additional assumptions about the observations as well

$$\begin{aligned} p(x_1, \dots, x_n | y_1, \dots, y_{n+1}) &= \prod_{i=1}^{n} p(x_i | x_1, \dots, x_{i-1}, y_1, \dots, y_{n+1}) \qquad \text{(chain rule)} \\ &\approx \prod_{i=1}^{n} p(x_i|y_i) \qquad \text{(independence assumption)} \end{aligned}$$

In other words, we say that the identity of each word only depends on the corresponding tags. This is a drastic assumption but still (often) leads to a reasonable tagging model, and simplifies our calculations. A more formal statement here is that the random variable X_i is conditionally independent of all the other variables in the model given Y_i (see more on conditional independence in the Bayesian networks lectures).

Now, we have a much simpler tagging model

$$p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) = p(y_1) \prod_{i=2}^{n+1} p(y_i|y_{i-1}) \prod_{i=1}^{n} p(x_i|y_i) \tag{243}$$

For notational convenience, we also assume a special fixed START symbol $y_0 = *$ so that $p(y_1)$ becomes $p(y_1|y_0)$. As a result, we can write

$$p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) = \prod_{i=1}^{n+1} p(y_i|y_{i-1}) \prod_{i=1}^{n} p(x_i|y_i) \tag{244}$$

Let's understand this model a bit more carefully by looking at how the pairs of sequences could be generated from the model. Here's the recipe

1. Set $y_0 = *$ (we always start from the START symbol) and let $i = 1$.
2. Generate tag y_i from the conditional distribution $p(y_i|y_{i-1})$ where y_{i-1} already has a value (e.g., $y_0 = *$ when $i = 1$)
3. If y_i = STOP, we halt the process and return $y_1, \ldots, y_i, x_1, \ldots, x_{i-1}$. Otherwise we generate x_i from the output/emission distribution $p(x_i|y_i)$
4. Set $i = i + 1$, and return to step 2.

HMM formal definition

The model we have defined is a Hidden Markov Model or HMM for short. An HMM is defined by a tuple $\langle N, \Sigma, \theta \rangle$, where

- N is the number of states $1, \ldots, N$ (assume the last state N is the final state, i.e. what we called "STOP" earlier).
- Σ is the alphabet of output symbols. For example, $\Sigma = \{$"the", "dog"$\}$.
- $\theta = \langle a, b, \pi \rangle$ consists of three sets of parameters
  - Parameter $a_{i,j} = p(y_{next} = j|y = i)$ for $i = 1, \ldots, N - 1$ and $j = 1, \ldots, N$ is the probability of transitioning from state i to state j: $\sum_{k=1}^{N} a_{i,k} = 1$
  - Parameter $b_j(o) = p(x = o|y = j)$ for $j = 1 \ldots N - 1$ and $o \in \Sigma$ is the probability of emitting symbol o from state j: $\sum_{o \in \Sigma} b_j(o) = 1$.
  - Parameter $\pi_i = p(y_1 = i)$ for $i = 1 \ldots N$ specifies probability of starting at state i: $\sum_{i=1}^{N} \pi_i = 1$.

Note that θ is a vector of $N + N \cdot (N-1) + (N-1) \cdot |\Sigma|$ parameters.

Example:

- N $= 3$. States are $\{1, 2, 3\}$
- Alphabet $\Sigma = \{the, dog\}$
- Distribution over initial states: $\pi_1 = 1, \pi_2 = \pi_3 = 0$.
- Parameters $a_{i,j}$ are

	$j = 1$	$j = 2$	$j = 3$
$i = 1$	0.5	0.5	0
$i = 2$	0	0.8	0.2

- Parameters $b_j(o)$ are

	$o =$the	$o =$dog
$i = 1$	0.9	0.1
$i = 2$	0.1	0.9

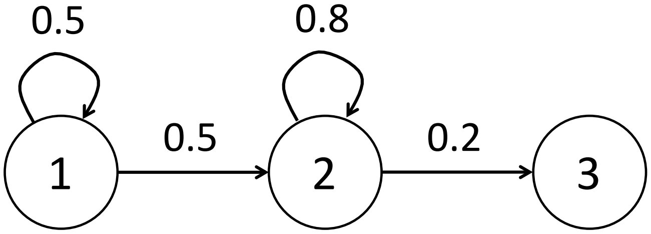


Figure 32: Transition graph for the example.

An HMM specifies a probability for each possible (x, y) pair, where $x = (x_1, \ldots, x_n)$ is a sequence of symbols drawn from Σ and $y = (y_1, \ldots, y_n)$ is a sequence of states drawn from the integers $1, \ldots, (N-1)$.

$$\begin{aligned} p(x, y|\theta) =& \pi_{y_1} \cdot && \text{(prob. of choosing } y_1 \text{ as an initial step)} \\ & a_{y_n, N} \cdot && \text{(prob. of transitioning to the final step)} \\ & \prod_{i=2}^{n} a_{y_{i-1}, y_i} \cdot && \text{(transition probability)} \\ & \prod_{i=1}^{n} b_{y_i}(x_i) && \text{(emission probability)} \end{aligned}$$

Consider the example: “the/1, dog/2, the/1”. The probability of such sequence is:

$$\pi_1 b_1(the) a_{1,2} b_2(dog) a_{2,1} b_1(the) a_{2,3} = 0 \tag{245}$$

Parameter estimation

We will first look at the fully observed case (complete data case), where our training data contains both xs and ys. We will do maximum likelihood estimation. Consider the examples: $\Sigma = \{e, f, g, h\}, N = 3$. Observation: $(e/1, g/2), (e/1, h/2), (f/1, h/2), (f/1, g/2)$. To find the MLE, we will simply look at the counts of events, i.e., the number of transitions between tags, the number of times we saw an output symbol together with an specific tag (state). After normalizing the counts to yield valid probability estimates, we get

$$a_{i,j} = \frac{count(i,j)}{count(i)} \tag{246}$$

$$a_{1,2} = \frac{count(1,2)}{count(1)} = \frac{4}{4} = 1,\ a_{2,2} = \frac{count(2,2)}{count(2)} = \frac{0}{4} = 0, \ldots \tag{247}$$

where $count(i,j)$ is the number of times we have (i,j) as two successive states and $count(i)$ is the number of times state i appears in the sequence. Similarly,

$$b_i(o) = \frac{count(i \to o)}{count(i)} \tag{248}$$

$$b_1(e)\frac{count(1 \to e)}{count(1)} = \frac{2}{4} = 0.5, \ldots \tag{249}$$

where $count(i \to o)$ is the number of times we see that state i emits symbol o. $count(i)$ was already defined above.

Decoding with HMM

Suppose now that we have the HMM parameters θ (see above) and the problem is to infer the underlying tags $y_1, \ldots, y_n$ corresponding to an observed sequence of words $x_1, \ldots, x_n$. In other words, we wish to evaluate

$$\text{argmax}_{y_1,\ldots,y_n}\, p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) \tag{250}$$

where

$$p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) = \prod_{i=1}^{n+1} a_{y_{i-1},y_i} \prod_{i=1}^{n} b_{y_i}(x_i) \tag{251}$$

and $y_0 = *, y_{n+1} = \text{STOP}$. Note that by defining a fixed starting state, we have again folded the initial state distribution π into the transition probabilities $\pi_i = a_{*,i},\ i \in \{1, \ldots, N\}$ (where $N =$STOP).

One possible solution for finding the most likely sequence of tags is to do brute force enumeration. Consider the example: $\Sigma = \{\text{the, dog}\}$, $x =$"the the the dog". The possible state sequences include:

$$1\ 1\ 1\ 2\ \text{STOP} \tag{252}$$

$$1\ 1\ 2\ 2\ \text{STOP} \tag{253}$$

$$1\ 2\ 2\ 2\ \text{STOP} \tag{254}$$

$$\vdots \tag{255}$$

But there are $|\mathcal{T}|^n$ possible sequences in total! Solving the tagging problem by enumerating the tag sequences will be prohibitively expensive.

Viterbi algorithm

The HMM has a simple dependence structure (recall, tags form a Markov sequence, observations only depend on the underlying tag). We can exploit this structure in a dynamic programming algorithm.

Input: $x = x_1, \ldots, x_n$ and model parameters θ.

Output: $\text{argmax}_{y_1,\ldots,y_{n+1}} p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1})$.

Now, let's look at a truncated version of the joint probability, focusing on the first k tags for any $k \in \{1, \ldots, n\}$. In other words, we define

$$r(y_1, \ldots, y_k) = \prod_{i=1}^{k} a_{y_{i-1},y_i} \prod_{i=1}^{k} b_{y_i}(x_i) \tag{256}$$

where y_k does not equal STOP. Note that our notation $r(y_1, \ldots, y_k)$ suppresses any dependence on the observation sequence. This is because we view $x_1, \ldots, x_n$ as given (fixed). Note that, according to our definition,

$$\begin{aligned} p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) =& r(y_1, \ldots, y_n) \cdot a_{y_n,y_{n+1}} \\ =& r(y_1, \ldots, y_n) \cdot a_{y_n,\text{STOP}} \end{aligned}$$

Let $S(k, v)$ be the set of tag sequences $y_1, \ldots, y_k$ such that $y_k = v$. In other words, $S(k, v)$ is a set of all sequences of length k whose last tag is v. The dynamic programming algorithm will calculate

$$\pi(k, v) = \max_{(y_1,\ldots,y_k) \in S(k,v)} r(y_1, \ldots, y_k) \tag{257}$$

recursively in the forward direction. In other words, $\pi(k, v)$ can be thought as solving the maximization problem partially, over all the tags $y_1, \ldots, y_{k-1}$ with the constraint that we use tag v for y_k. If we have $\pi(k, v)$, then $\max_v \pi(k, v)$ evaluates $\max_{y_1,\ldots,y_k} r(y_1, \ldots, y_k)$. We leave v in the definition of $\pi(k, v)$ so that we can extend the maximization one step further as we unravel the model in the forward direction. More formally,

- Base case:

$$\begin{aligned} &\pi(0, *) = 1 \ \ \text{(starting state, no observations)} \\ &\pi(0, v) = 0, \ \ \text{if } v \neq * \ \ \text{(an actual state has observations)} \end{aligned}$$

This definition reflects the assumption that $y_0 = *$.

- Moving forward recursively: for any $k \in \{1, \dots, n\}$

$$\pi(k, v) = \max_{u \in \mathcal{T}} \{\pi(k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\} \tag{258}$$

  In other words, when extending $\pi(k-1, u)$, $u \in \mathcal{T}$, to $\pi(k, v)$, $v \in \mathcal{T}$, we must transition from $y_{k-1} = u$ to $y_k = v$ (part $a_{u,v}$) and generate the corresponding observation x_k (part $b_v(x_k)$). Then we maximize over the previous tag $y_{k-1} = u$ so that $\pi(k, v)$ only depends on the value of y_k.

Finally, we must transition from y_n to STOP so that

$$\max_{y_1,\dots,y_n} p(x_1, \dots, x_n, y_1, \dots, y_n, y_{n+1} = \text{STOP}) = \max_{v \in \mathcal{T}} \{\pi(n, v) \cdot a_{v,\text{STOP}}\} \tag{259}$$

The whole calculation can be done in time $O(n|\mathcal{T}|^2)$, linear in length, quadratic in the number of tags.

Now, having values $\pi(k, v)$, how do we reconstruct the most likely sequence of tags which we denote as $\hat{y}_1, \dots, \hat{y}_n$? We can do this via *back-tracking.* In other words, at the last step, $\pi(n, v)$ represents maximizations of all $y_1, \dots, y_n$ such that $y_n = v$. What is the best value for this last tag v, i.e., what is $\hat{y}_n$? It is

$$\hat{y}_n = \text{argmax}_v \left\{\pi(n, v) a_{v,\text{STOP}}\right\} \tag{260}$$

Now we can fix $\hat{y}_n$ and work backwards. What is the best value $\hat{y}_{n-1}$ such that we end up with tag $\hat{y}_n$ in position n? It is simply

$$\hat{y}_{n-1} = \text{argmax}_u \{\pi(n-1, u) a_{u,\hat{y}_n}\} \tag{261}$$

and so on.

Hidden Variable Problem

When we no longer have the tags, we must resort to other ways of estimating the HMMs. It is not trivial to construct a model that agrees with the observations except in very simple scenarios. Here is one:

- We have an HMM with $N = 3, \Sigma = \{a, b, c\}$
- We see the following output sequences in training data: $(a, b), (a, c), (a, b)$.

How would you choose the parameter values for $\pi_i, a_{i,j}$ and $b_i(o)$? A reasonable choice is:

$$\pi_1 = 1.0, \pi_2 = \pi_3 = 0 \tag{262}$$

$$b_1(a) = 1.0, b_1(b) = b_1(c) = 0 \tag{263}$$

$$b_2(a) = 0, b_2(b) = b_2(c) = 0.5 \tag{264}$$

$$a_{1,2} = 1.0, a_{1,1} = a_{1,3} = 0 \tag{265}$$

$$a_{2,3} = 1.0, a_{2,1} = a_{2,2} = 0 \tag{266}$$

Expectation-Maximization (EM) for HMM

Suppose now that we have multiple observed sequences of outputs (no observed tags). We will denote these sequences with superscripts, i.e., $x^1, x^2, \ldots, x^m$. In the context of each sequence, we must evaluate a posterior probability over possible tag sequences. For estimation, we only need expected counts that are used in the re-estimation step (M-step). To this end, let $\mathrm{count}(x^i, y, p \to q)$ be the numbers of times a transition from state p to state q occurs in a tag sequence y corresponding to observation x^i. We will only show here the derivations for transition probabilities; the equations for emission and initial state parameters are obtained analogously.

E-step: calculate expected counts, added across sequences

$$\overline{count}(u \to v) = \sum_{i=1}^{m} \sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, u \to v) \tag{267}$$

M-step: re-estimate transition probabilities based on the expected counts

$$a_{u,v} = \frac{\overline{count}(u \to v)}{\sum_{k=1}^{N} \overline{count}(u \to k)} \tag{268}$$

where the denominator ensures that $\sum_{k=1}^{N} a_{u,k} = 1$.

The main problem in running the EM algorithm is calculating the sum over the possible tag sequences in the E-step.

$$\sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, u \to v) \tag{269}$$

The sum is over an exponential number of possible hidden state sequences y. Next we will discuss a dynamic programming algorithm – forward-backward algorithm. The algorithm is analogous to the Viterbi algorithm for maximizing over the hidden states.

The Forward-Backward Algorithm for HMMs

Suppose we could efficiently calculate marginal posterior probabilities

$$p(y_j = p, y_{j+1} = q|x, \theta) = \sum_{y: y_j = p, y_{j+1} = q} p(y|x, \theta) \tag{270}$$

for any $p \in 1 \ldots (N-1), q \in 1 \ldots N, j \in 1 \ldots n$. These are the posterior probabilities that the state in position j was p and we transitioned into q at the next step. The probability is conditioned on the observed sequence x and the current setting of the model parameters θ. Now, under this assumption, we could rewrite the difficulty computation in Eq. 269 as:

$$\sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, p \to q) = \sum_{j=1}^{n} p(y_j = p, y_{j+1} = q|x^i, \theta^{t-1}) \tag{271}$$

The key remaining question is how to calculate these posterior marginals effectively. In other words, our goal is to evaluate $p(y_j = p, y_{j+1} = q|x^i, \theta^{t-1})$.

Now, consider a single observation sequence $x_1, \ldots, x_n$. We will make use of the following forward probabilities:

$$\alpha_p(j) = p(x_1, \ldots, x_{j-1}, y_j = p|\theta) \tag{272}$$

for all $j \in 1 \ldots n$, for all $p \in 1 \ldots N-1$. $\alpha_p(j)$ is the probability of emitting the symbols $x_1, \ldots, x_{j-1}$ and ending in state p in position j without (yet) emitting the corresponding output symbol. These are analogous to the $\pi(k, v)$ probabilities in the Viterbi algorithm with the exception that $\pi(k, v)$ included generating the corresponding observation in position k. Note that, unlike before, we are summing over all the possible sequences of states that could give rise to the observations $x_1, \ldots, x_{j-1}$. In the Viterbi algorithm, we maximized over the tag sequences.

Similarly to the forward probabilities, we can define the backward probabilities:

$$\beta_p(j) = p(x_j, \ldots, x_n|y_j = p, \theta) \tag{273}$$

for all $j \in 1 \ldots n$, for all $p \in 1 \ldots N-1$. $\beta_p(j)$ is the probability of emitting symbols $x_j, \ldots, x_n$ and transitioning into the final (STOP) state, given that we begun in state p in position j. Again, this definition involves summing over all the tag sequences that could have generated the observations from x_j onwards, provided that the tag at j is p.

Why are these two definitions useful? Suppose we had been able to evaluate α and β probabilities effectively. Then the marginal probability we were after could be calculated as:

$$p(y_j = p, y_{j+1} = q|x, \theta) = \frac{1}{Z}\alpha_p(j) a_{p,q} b_p(o_j) \beta_q(j+1)$$
$$Z = p(x_1, \ldots, x_n|\theta) = \sum_p \alpha_p(j)\beta_p(j) \quad \text{for any } j = 1 \ldots n$$

This is just the sum over all possible tag sequences that include the transition $y_j = p$ and $y_{j+1} = q$ and generates the observations, divided by the sum over all tag sequences that generate the observations. As a result, we obtain the relative probability of the transition, relative to all the alternatives given the observations, i.e., the posterior probability. Note that $\alpha_p(j)$ involves all the summations over tags $y_1, \ldots, y_{j-1}$, and $\beta_q(j+1)$ involves all the summations over the tags $y_{j+2}, \ldots, y_n$.

Let's finally discuss how we can calculate α and β.

As Fig. 33 shows, for every state sequence $y_1, y_2, \ldots, y_n$ there is

- a path through graph that has the sequence of states $s, \langle 1, y_1 \rangle, \ldots, \langle n, y_n \rangle, f$.
- The path associated with state sequence $y_1, \ldots, y_n$ has weight equal to $p(x, y|\theta)$.
- $\alpha_p(j)$ is the sum of weights at all paths from s to the state (j, p).
- $\beta_p(j)$ is the sum of weights at all paths from state $\langle j, p \rangle$ to the final state f.

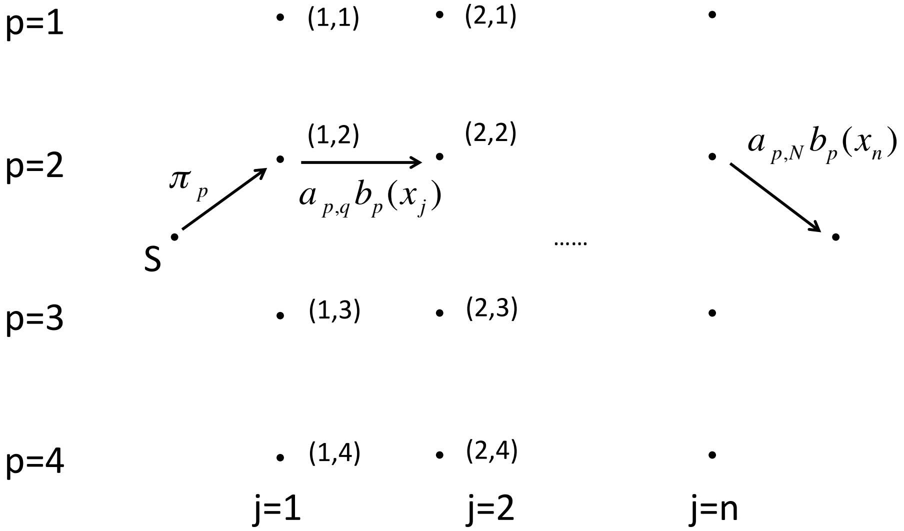


Figure 33: A path associated with state sequence.

Given an input sequence $x_1, \ldots, x_n$, for any $p \in 1 \ldots N, j \in 1 \ldots n$, the forward and backward probability can be calculated recursively.

Forward probability:

$$\alpha_p(j) = p(x_1, \ldots, x_{j-1}, y_j = p|\theta) \tag{274}$$

- Base case:

$$\alpha_p(1) = \pi_p \quad \forall p \in 1 \ldots N-1 \tag{275}$$

- Recursive case

$$\alpha_p(j+1) = \sum_q \alpha_q(j) a_{q,p} b_q(x_j) \quad \forall p \in 1 \ldots N-1, j = 1 \ldots n-1 \tag{276}$$

Backward probability:

$$\beta_p(j) = p(x_j, \ldots, x_n | y_j = p, \theta) \tag{277}$$

- Base case:

$$\beta_p(n) = a_{p,N} b_p(x_n) \quad \forall p \in 1 \ldots N-1 \tag{278}$$

- Recursive case

$$\beta_p(j) = \sum_q a_{p,q} b_p(x_j) \beta_q(j+1) \quad \forall p \in 1 \ldots N-1, j = 1 \ldots n-1 \tag{279}$$

14 Reinforcement Learning

Learning from Feedback

Let's consider a robot navigation problem (see Figure 34 for illustration). At each point in time, our robot is located at a certain position on the grid. Our robot also has sensors which can help predict (with some noise) its position within the grid. Our goal is to bring the robot to its desired final destination (e.g., charging station). The robot can from one position to another in small increments. However, we assume that the movements are not deterministic. In other words, with some probability the robot moves to the desired next position, but there is also a chance that it ends up in another nearby location (e.g., applying a bit too much power). Let's assume for simplicity that the states are discrete. That is, the positions correspond to grid points. Figure 35 illustrates this abstraction.

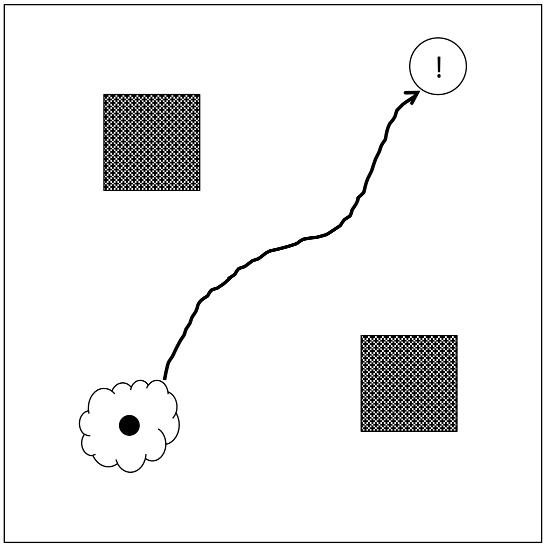


Figure 34: Robot navigation task.

At the first glance, this problem may resemble HMMs: we use states y_t to encode the position of the robot, and observations x_t capture the sensor readings. The process is clearly Markov in the sense that the transition to the next state is only determined by the current state.

In the case of HMMs, we represent transitions as $T(i,j) = p(y_{t+1} = j|y_t = i)$. What is missing? This parametrization does not account for the fact that the robot can select actions such as the direction that it wants to move in. We will therefore expand the transition probabilities to incorporate selected actions — $T(i,k,j) = p(y_{t+1} = j|y_t = i, a_t = k)$ specifies the probability of transitioning to j from i after taking action k.

In contrast to the more realistic setting discussed above, we will make the model simpler here by assuming that the states are directly observable. I.e., at every point, the robot knows its exact location in the greed. Put another way, the observation x_t fully determines y_t, and we will drop x_t as a result.

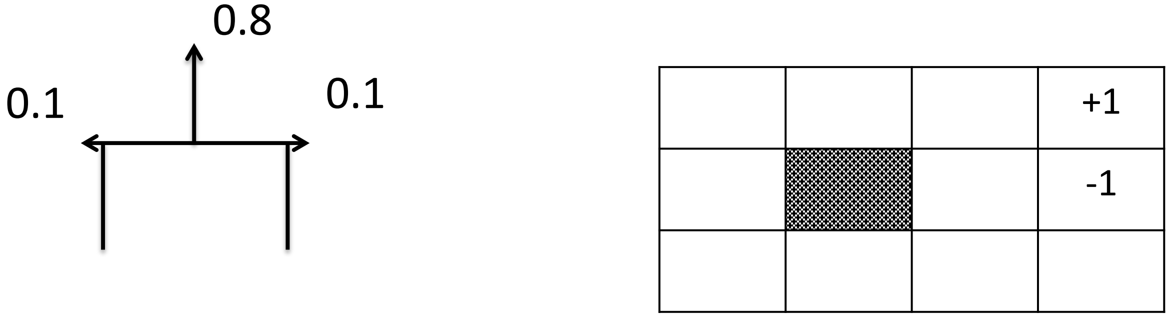


Figure 35: Robot in the grid. With probability 0.8, the robot will move into a specified direction. With probability 0.1, it will move into one of the orthogonal directions.

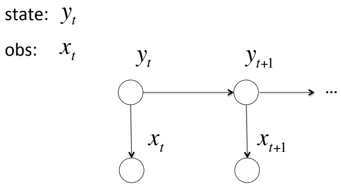


Figure 36: HMM.

The remaining piece in our robot navigation problem is to specify costs or rewards. Rewards can be associated with states $R(s_t)$ (e.g., the target charging station has a high reward) or they can also take into account the action that brings the robot to the follow-up state $R(s_t, a_t, s_{t+1})$. Here is a simple example of a reward function:

$$R_{s_t} = \begin{cases} 1 & \text{if on target} \\ 0 & \text{otherwise} \end{cases} \tag{280}$$

Utility Function Intuitively, a utility function would be the sum of all the rewards that the robot accumulates. However, this definition may result in acquiring infinite reward (for instance, when the robot loops around). Also, it may be better to acquire high rewards sooner than later. One possible setting is to assume that the robot has a finite horizon: after N steps, the utility value doesn't change at all:

$$U([s_0, s_1, \ldots, s_{N+k}]) = U_n([s_0, s_1, \ldots, s_N]) \;\; \forall k$$

However, under this assumption the optimal strategy depends on how long the agent has to live (see Figure 39). Thus, the optimal action would not only depend on the state that the robot is in, but would also depend on the time that it has left.

An alternative approach is to use so called *discounted rewards*. Even for infinite sequences, this utility function is guaranteed to have a finite value so long as the individual

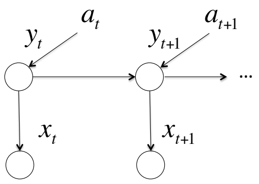


Figure 37: HMM with action.

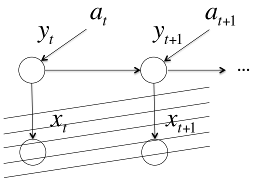


Figure 38: HMM without x_t, assuming the hidden state is observed

rewards are finite. For $0 \leq \gamma < 1$, we define it as

$$U([s_0, s_1, s_2 \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \tag{281}$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma} \tag{282}$$

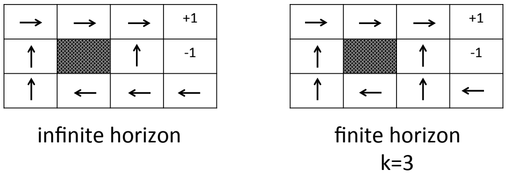
This formulations assigns higher utilities to rewards that come early. The discounting also helps us solve for these utilities (important for convergence of the algorithms we will cover later in the lecture).

Another way to think of the discounted utility is by augmenting our state space with an additional *"halt"* state. Once the agent reaches this state, the experience stops. At any point, we will transition to this halt state with probability $1-\gamma$. Thus the probability that we continue to move on is γ. Larger γ means longer horizon.

Policy A policy π is a function that specifies an action for each state. Our goal is to compute an optimal policy that maximizes the expected utility, i.e., the action that the robot takes in any state is chosen with the idea of maximizing the discounted future rewards. As illustrated in Figure 40, an optimal policy depends heavily on the reward function. In fact, it is the reward function that specifies the goal.

We will consider here two ways to learn the optimal policy:

- **Markov Decision Process (MDP)**. We assume that reward function and transition probabilities are known and available to the robot. More specifically, we are provided with
  - a set of states S
  - a set of actions A
  - a transition probability function $T(s, a, s') = p(s'|a, s)$
  - a reward function $R(s, a, s')$ (or just $R(s')$)



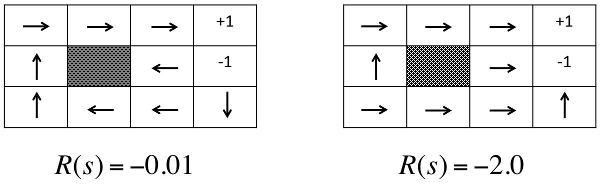
→	→	→	+1
↑	▨	↑	-1
↑	←	←	←

infinite horizon

→	→	→	+1
↑	▨	↑	-1
↑	←	↑	←

finite horizon
k=3

Figure 39: Policy with different horizons.



→	→	→	+1
↑	▨	←	-1
↑	←	←	↓

$R(s) = -0.01$

→	→	→	+1
↑	▨	→	-1
→	→	→	↑

$R(s) = -2.0$

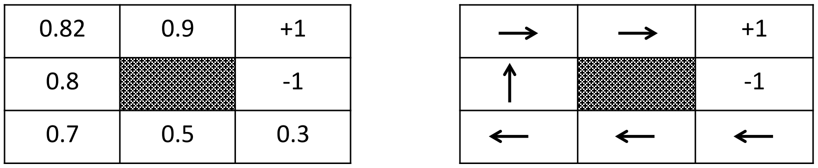
Figure 40: Policy with different rewards. $R(s)$ is reward associated with visiting a state (excluding absorbing states).

- **Reinforcement Learning** The reward function and transition probabilities are unknown (except for their form), and the robot only knows
  - a set of states S
  - a set of actions A

Value iteration

Value iteration is an algorithm for finding the (an) optimal policy for a given MDP. The algorithm iteratively estimates the value of each state; in turn, these values are used to compute the optimal policy. We will use the following notations:

- $V^*(s)$ – The value of state s, i.e., the expected utility of starting in state s and acting optimally thereafter.
- $Q^*(s,a)$ – The Q value of state s and action s. It is the expected utility of starting in state s, taking action a and acting optimally thereafter.
- $\pi^*(s)$ –The optimal policy. $\pi^*(s)$ specifies the action we should take in state s. Following policy π^* would, in expectation, result in the highest expected utility (see Figure 41).



0.82	0.9	+1
0.8	▨	-1
0.7	0.5	0.3

→	→	+1
↑	▨	-1
←	←	←

Figure 41: V-values and associated policy.

The above quantities are clearly related:

$$V^*(s) = \max_a Q^*(s,a) = Q^*(s, \pi^*(s)) \tag{283}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{284}$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{285}$$

$$= \sum_{s'} T(s,\pi^*(s),s')[R(s,\pi^*(s),s') + \gamma V^*(s')] \tag{286}$$

For example, in Eq.(284), we evaluate the expected reward of taking action a in state s. We sum over the possible next states s', weighted by the transition probabilities corresponding to the state s and action a. Each summand combines the immediate reward with the expected utility we would get if we started from s' and followed the optimal policy thereafter. The future utility is discounted by γ as discussed above. In other words, we have simple one-step lookahead relationship among the utility values.

Based on these equations, we can recursively estimate $V_k^*(s)$, the optimal value considering next k steps. As $k \to \infty$, the algorithm converges to the optimal values $V^*(s)$.

The Value Iteration Algorithm

- Start with $V_0^*(s) = 0$, for all $s \in S$
- Given V_i^*, calculate the values for all states $s \in S$ (depth $i+1$):

$$V_{i+1}^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i^*(s')]$$

- Repeat until convergence (until $V_{i+1}(s)$ is nearly $V_i(s)$ for all states)

The convergence of this algorithm is guaranteed. We will not show a full proof here but just illustrate the basic idea. Consider a simple MDP with a single state and a single action. In such a case:

$$V_{i+1} = R + \gamma V_i$$

We also know that for the optimal V* the following must hold:

$$V^* = R + \gamma V^*$$

By subtracting these two expressions, we get:

$$(V_{i+1} - V^*) = \gamma(V_i - V^*)$$

Thus, after each iteration, the difference between the estimate and the optimal value decreases by a factor $\gamma < 1$.

Once the values are computed, we can turn them into the optimal policy:

$$Q^*(s,a) = \sum_s T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

As we rely on the Q-values to get the policy, we could alternatively compute these values directly. Here is an algorithm that does exactly that.

The Q-Value Iteration Algorithm

- Start with $Q_0^*(s,a) = 0$ for all $s \in S$, $a \in A$.
- Given $Q_i^*(s,a)$, calculate the q-values for all states (depth $i+1$) and for all actions a:

$$Q_{i+1}^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \max_{a'} Q_i^*(s',a')]$$

This algorithm has the same convergence guarantees as its value iteration counterpart.

Reinforcement learning

Now, we will consider a set-up where neither reward no transitions are known a priori. Our robot can travel in the grid, moving from one state to another, collecting rewards along the way. The model of the world in unknown to the robot other than the overall Markov structure. The robot could do one of two things. First, it could try to learn the model, the reward and transition probabilities, and then solve the optimal policy using the algorithms for MDPs described above. Another option is to try to learn the Q-values directly.

Model-based learning We first assume that we can collect information about transitions involving any state s and action a. Under this assumption, we can learn T and R through experience, by collecting outcomes for each s and a.

$$T(s,a,s') = \frac{count(s,a,s')}{\sum_{s'} count(s,a,s')}$$

$$R(s,a,s') = \frac{\sum_t R_t(s,a,s')}{count(s,a,s')}$$

where $R_t(s,a,s')$ is the reward we observed when starting in state s, taking action a, and transitioning to s'. If the reward is noisy, observed rewards $R_t()$ may vary from one instance to another. In reality, this naive approach is highly ineffective for any non-trivial state space. The best we can do is randomly explore, taking actions and moving from one state to another. Most likely, we will be unable to reach many parts of the state space in any complex environment. Moreover, the learned model would be quite large as we'd have to store all the states and possible transitions.

Model-free learning Can we learn how to act without learning a full model? Remember:

$$pi^*(s) = \arg\max_a Q^*(s,a)$$

We have shown that Q-values can be learned recursively, assuming we have access to T and R. Since this information is not provided to us, we will consider Q-learning algorithm, a sample based Q-value iteration procedure.

To better understand the difference between model-based and model-free estimation, consider the task of computing the expected value of a function $f(x)$: $E[f(x)] = \sum_x p(x)f(x)$

- **Model-based computation**: First estimate $p(x)$ from samples and then compute expectation:

$$x_i \sim p(x),\ i = 1, \ldots, k$$

$$\hat{p}(x) = \frac{count(x)}{k}$$

$$E[f(x)] \approx \sum_x \hat{p}(x)f(x)$$

- **Model-free estimation**: estimate expectation directly from samples

$$x_i \sim p(x),\ i = 1, \ldots, k$$

$$E[f(x)] \approx \frac{1}{k}\sum_{i=1}^{k} f(x_i)$$

Now we will apply the model-free learning approach to the estimation of Q-values. Recall,

$$Q^*_{i+1}(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*_i(s',a')]$$

We will repeatedly take one action at a time, and observe the reward and the next state. We can compute:

$$sample_1 : R(s,a,s'_1) + \gamma \max_{a'} Q_i(s'_1, a')$$

$$sample_2 : R(s,a,s'_2) + \gamma \max_{a'} Q_i(s'_2, a')$$

$$\ldots$$

$$sample_k : R(s,a,s'_k) + \gamma \max_{a'} Q_i(s'_k, a')$$

Now we can average all the samples, to obtain the Q-value estimate:

$$Q_{i+1}(s,a) = \frac{1}{k}\sum_{l=1}^{k}\left[R(s,a,s'_l) + \gamma \max_{a'} Q_i(s'_l,a')\right]$$

which, for large k, would be very close to the Q-value iteration step. We are almost there. In practice, we only observe the states when we actually move. Therefore, we cannot really collect all these sample at once. Instead, we will update the Q-values after every experience (s,a,s',r), where r is the reward. To this end, we will use exponential running average:

$$\bar{x}_n = \frac{x_n + (1-\alpha)*x_{n-1} + (1-\alpha)^2 * x_{n-2}}{1 + (1-\alpha) + (1-\alpha^2) + \dots}$$

This averaging procedures emphasizes recent samples, downplaying the past. It enables us to easily compute running average:

$$\bar{x}_n = \alpha * x_n + (1-\alpha) * \bar{x}_{n-1}$$

.

The key step of Q-learning algorithm is compute new values of Q by repeatedly incorporating a new sample into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[sample]$$

$$sample = R(s,a,s') + \gamma \max_{a'} Q_i(s',a')$$

Q-learning Algorithm

- Collect a sample: s, a, s', and $R(s,a,s')$.
- Update Q-values, by incorporating the new sample into a running average over samples:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right] \tag{287}$$

$$= Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{288}$$

where the learning rate α takes the role of $1/k$ in the earlier sample average example. During the iterations of the algorithm, likely s' will contribute more often to the Q-values estimate. As the algorithm progresses, old estimates fade, making the Q-value more consistent with more recent samples.

You may have noticed that the form of the update closely resembles stochastic gradient decent. In fact, it has the same convergence conditions as the gradient ascent algorithm. Each sample corresponds to (s,a), i.e., being in state s and taking action a. We can assign a separate learning rate for each such case, i.e., $\alpha = \alpha_k(s,a)$, where k is the number of times that we saw (s,a). Then, in order to ensure convergence, we should have

$$\sum_k \alpha_k(s,a) \to \infty$$

$$\sum_k \alpha_k^2(s, a) < \infty$$

Exploration/Exploitation Trade-Off In the Q-learning algorithm, we haven't specified how to select an action for a new sample. One option is to do it fully randomly. While this exploration strategy has a potential to cover a wide spectrum of possible actions, most likely it will select plenty of suboptimal actions, and leads to a poor exploration of the relevant (high reward) part of the state space. Another option is to exploit the knowledge we have already obtained during previous iterations. Remember that once we have Q estimates, we can compute a policy. Since our estimates are noisy in the beginning, and the corresponding policy is weak, we wouldn't want to follow this policy completely. To allow for additional exploration, we select a random action every once in a while. Specifically, with prob ε, the action is selected at random and with probability $(1 - \varepsilon)$, we follow the policy induced by the current Q-values. Over time, we can decrease ε, to rely more heavily on the learned policy as it improves based on the Q-learning updates.