Introduction to Machine Learning
Draft Notes by Topic

Tommi Jaakkola and Regina Barzilay
MIT

Spring 2016

Contents

1 Introduction

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. Such predictions are invariably 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 or 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 the available data. We will try to understand how machine learning problems are formulated, which methods work and 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 to ground our discussion.

Classification as a learning problem

Much of the available data today are in unstructured textual form such as news articles, books, product reviews, emails, tweets, etc. Given the huge selection of possible things to view, it is not feasible for us to weed through them by hand. We need automated tools to filter the possibilities according to our preferences, similar to spam filtering that all of us employ implicitly or explicitly. How do we express our preferences? Should we write down "if-then" rules? This would be time consuming, cumbersome, and really would not perform well. Instead, it is substantially easier just to annotate a small number of documents as positive (those that we want) or negative (documents we don't want), and ask a machine learning algorithm to learn the "rules" automatically based on the information implicitly provided by these examples. The set of *labeled examples* – the set of documents we have labeled as positive (+1) or negative (-1) – is known as the *training set.* Our goal is to learn a *classifier* – a binary valued function – based on the training set. Once found, the classifier can be easily applied to predict ±1 labels for any (large) number of new documents.

In our simple setting the learning algorithm whose goal is to find a classifier is only given the training set, nothing else. This generality is very powerful. The examples in the training set could represent cell samples, snippets of computer code, molecular structures, or anything else. The same algorithm can be run to learn from these labeled examples no matter what their origin, and the resulting classifier can be applied to predict labels for new examples of the same kind.

Feature vectors

But we aren't we forgetting something? Computers do not really understand what "documents" or "cell samples" mean. What exactly is this classifier (binary ±1 valued function) taking in as an argument? We have to work a bit harder to describe each example (here a document) in a manner that can be easily used by automated methods. To this end, we will translate each example into a *feature vector* whose coordinates measure something useful about the corresponding example. A poor construction of feature vectors may loose or hide the relevant information that we would like the classifier to pick up. The more explicit the key pieces of information are in these vectors, the simpler the task is for the learning algorithm to solve. Note that all the examples in the training set as well as the new examples to be classified later must be mapped to their corresponding feature vectors by the same procedure. Otherwise the new examples would appear different to the classifier and result in poor predictions.

Documents are often mapped to feature vectors using the so-called "bag-of-words" representation. Each coordinate in this vector representation corresponds to a word in the vocabulary, and the value of the coordinate is the number of times the word appears in the document. The dimension of the resulting feature vector is the number of words we wish to query. This could be a well-crafted set of words pertinent to the classification problem or just all the typical words (e.g., in English, the vector could be 10,000 dimensional). No problem. The vectors are also commonly normalized by the length of the document so that documents that talk about similar things result in roughly the same vectors regardless of their lengths. It's worth emphasizing that the coordinates must mean the same from one document to another. For example, the first coordinate always represents the frequency of word "Machine", and so on.

We have finally managed to cast the problem in a form that we can easily learn from with automated tools. The training set consists of a set of labeled fixed-length vectors. Any problem that can be mapped to this form can then be solved using the same algorithms. The method is clearly completely oblivious to how the vectors were constructed, and what the coordinates mean. All that it can do is to try to statistically relate how the feature coordinates vary from one example to another in relation to the available label. We will operate on this level of abstraction.

Before moving on, let's take a couple of quick examples of building feature vectors in other contexts. Consider the problem of classifying whether a tissue sample contains tumor cells. While the samples themselves are relatively easy to acquire from willing participants, the task of determining whether they contain tumor cells often involves a combination of laboratory tests and specialist assessments. In order to easily screen large numbers of samples, we should try to develop automated methods that can classify whether new samples contain tumor cells (+1) or only normal cells (-1). The training set in this case would consist of tissue samples with verified labels. It remains to map tissue samples to fixed-length feature vectors. To do so, we could use readily available high-throughput biological assays that 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 expression of a particular gene (averaged across the cells in the sample). We are assuming here that the expression of genes provide some cues for catching tumor cells. Once the fixed-length feature vectors are available for each sample (however they were constructed), we can apply the same generic machine learning method to learn a mapping from these vectors to ±1 labels, and apply it to new samples.

In many cases the examples to be classified are images. How do we represent images as feature vectors? We could take a high resolution pixel image 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 learning algorithm to figure out. For example, hair, skin color, eyes, etc may be important "features" to pay attention to in order to predict the gender of a person but none of these features are immediately 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. Modern computer vision algorithms learn the features directly from the images along with solving the classification task. We will touch on these methods in the context of "deep learning" architectures. For now, we assume that the feature vectors are constructed prior to learning.

Formalizing the learning problem

Let's look at these types of classification problems a bit more formally. We will use $x = [x_1, \ldots, x_d]^T \in \mathbb{R}^d$ 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 $\phi(x) \in \mathbb{R}^d$ 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. Note that examples x can be also viewed geometrically as points in $\mathbb{R}^d$. If $d = 2$, we can easily plot them on the plane.

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 set S_n where the subscript n highlights the number of labeled examples we have.

A classifier h is a mapping from feature vectors to labels: $h : \mathcal{R}^d \to \{-1, 1\}$. When we apply the classifier to a particular example x, we write $h(x)$ as the predicted label. Any classifier therefore divides the space $\mathbb{R}^d$ into positive and negative regions depending on the predicted label. There could be a number of distinct regions that are marked as positive (i.e., x such that $h(x) = +1$). A learning algorithm entertains a *set of classifiers* $\mathcal{H}$ (set of hypotheses about the rules that govern how labels are related to examples), 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 key 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.

We refer to the classifier performance on the future test examples as *generalization*. In other words, a classifier that generalizes well performs on the new examples similarly to its performance on the training set (which we can measure). Designing classifiers that generalize well ties together the choices of feature vectors, how many training examples we have (i.e., n), the choice of the set of classifiers $\mathcal{H}$, as well as how $\hat{h} \in \mathcal{H}$ is chosen.

To understand these relationships a bit more precisely, let's start by defining how we evaluate the classifiers. There are two key measures. The *training error* of a classifier h is simply the fraction of mistakes that it makes on the training examples. More formally,

$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^{n} [\![\, h(x^{(i)}) \neq y^{(i)} \,]\!] \tag{1}$$

where $[\![\,\text{true}\,]\!] = 1$ (when h makes an error) and $[\![\,\text{false}\,]\!] = 0$ (when h is correct). We can evaluate the training error $\mathcal{E}_n(h)$ for any classifier h since we have the training examples. There are many classifiers that attain low or even zero training error for any specific training set S_n. But this is not the error we are actually interested in minimizing. Our objective is to minimize the *test* or *generalization error* (i.e., performance on future examples)

$$\mathcal{E}(h) = \frac{1}{n'} \sum_{i=n+1}^{n+n'} [\![\, h(x^{(i)}) \neq y^{(i)} \,]\!] \tag{2}$$

which is just like the training error but evaluated on the basis of the next n' (a large number of) new examples. The problem is that while we wish to minimize $\mathcal{E}(h)$, we cannot evaluate it since the test examples (e.g., future weather patterns) are not available at the time that we have to commit to a classifier $\hat{h}$. Why isn't our task hopeless then? We assume that the test examples are in many ways similar to the training examples. Specifically, we think of both the training and test examples along with their labels as drawn at random from some large underlying pool of labeled examples (formally, a joint distribution). So it is always helpful to have more training examples as the resulting training error (that we can measure) will then better reflect the test error (that we hope to minimize).

There are two key parts to designing a classifier that will generalize well. The choice of the set of classifiers $\mathcal{H}$ (a.k.a. *model selection*) and how $\hat{h} \in \mathcal{H}$ is selected (a.k.a. *parameter fitting*). It may sound strange but it is actually advantageous to entertain a set of classifiers $\mathcal{H}$ that is small. In other words, it is helpful to a priori restrict classifiers that we can choose in response to the training set S_n. This is precisely because our goal is prediction rather than minimizing the training error. As for the second part, we will talk a lot about *learning* or *training algorithms* that attempt to find some $\hat{h} \in \mathcal{H}$ for the chosen (small) set $\mathcal{H}$ that (approximately) minimize the training error. There are many ways to improve this part as well so as to generalize better.

Understanding the role of $\mathcal{H}$

Suppose $\mathcal{H}$ contains only a single classifier. In other words, we are not training anything. The only thing we can do is to evaluate how well this specific (already chosen) classifier performs on the training set. From this perspective, the training examples play the role of new test examples (which are not used for selecting h) so the classifier will generalize well in the sense that its training performance is very much like the test performance. Of course, it is probably not a good classifier as it wasn't tailored to the task at all. The more we make use of the training examples for selecting $\hat{h}$, the wider apart $\mathcal{E}_n(\hat{h})$ and $\mathcal{E}(\hat{h})$ can be.

Let's take another extreme example where $\mathcal{H}$ contains all binary functions. In other words, we can select any classifier we want so $\mathcal{H}$ is very large in this case. For example, given n distinct training examples $S_n = \{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, we can select

$$\hat{h}(x) = \begin{cases} +1 \text{ if there is a positively labeled } x^{(i)} \text{ such that } \|x - x^{(i)}\| \leq \epsilon \\ -1 \text{ otherwise} \end{cases} \tag{3}$$

Clearly $\hat{h} \in \mathcal{H}$ as it is a binary valued function. This classifier predicts $+1$ for a very small region around each positive example in the training set and -1 everywhere else. If the training examples are distinct, and ϵ small enough, then this $\hat{h}$ will have zero training error. Why isn't it an excellent classifier then? Suppose the test examples contain an equal number of $+1$ and -1 labeled examples. Also, as is typical, they are all distinct (our description of them in terms of feature vectors is rich enough that, overall, no two examples look exactly the same). What is the test error of our $\hat{h}$ in this setting? Exactly 1/2 (equivalent to random guessing). This is because all the positive test examples are misclassified as negative (they are not ϵ close to the positive training examples). On the other hand, all the negative examples are classified correctly since $\hat{h}$ predicts -1 nearly everywhere.

The goal is to explore sets of classifiers $\mathcal{H}$ that are small enough to ensure that we generalize well yet large enough that some $\hat{h} \in \mathcal{H}$ has low training error. We will start with a widely useful set of classifiers – *the set of linear classifiers.*

2 Linear Classification

Any classifier h divides the input space into two halves based on the predicted label, i.e., cuts $\mathbb{R}^d$ into two sets $\{x : h(x) = 1\}$ and $\{x : h(x) = -1\}$ which, as sets, can be quite complicated. Linear classifiers make this division in a simple geometric way. In two dimensions, each linear classifier can be represented by a line that divides the space into the two halves based on the predicted label. In three dimensions, the division is made by a plane, and in higher dimensions by a hyper-plane. Thus, as a set of classifiers, it is certainly constrained. We will see later on that it is easy to make linear classifiers very powerful by adding features to input examples, i.e., making the feature vectors higher dimensional. For now, we will assume that feature vectors are simple enough (low dimensional enough) that linear separation is an effective constraint and leads to good generalization.

2.1 The Set of Linear Classifiers

To gain a better understanding of what this set of linear classifiers really is, and what it can do, let's start with a slightly handicapped version, the set *linear classifiers through origin.* These are thresholded linear mappings from examples to labels. More formally, we only consider classifiers that can be written as

$$h(x; \theta) = \text{sign}\big(\, \theta_1 x_1 + \ldots + \theta_d x_d \,\big) = \text{sign}\big(\, \theta \cdot x \,\big) = \begin{cases} +1, & \theta \cdot x > 0 \\ -1, & \theta \cdot x \leq 0 \end{cases} \tag{4}$$

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, the parameter vector θ can be thought of as an index of a classifier. Any two classifiers corresponding to different parameters (other than overall scaling of θ) would produce a different prediction for at least some input examples x. We say that the set of linear classifiers through origin are *parameterized* by $\theta \in \mathcal{R}^d$.

We can understand these linear classifiers geometrically as alluded to earlier. Suppose we select one classifier, i.e., fix parameters θ, and look at what happens for different examples x. The classifier changes its prediction only when the argument to the sign function $\theta \cdot x$

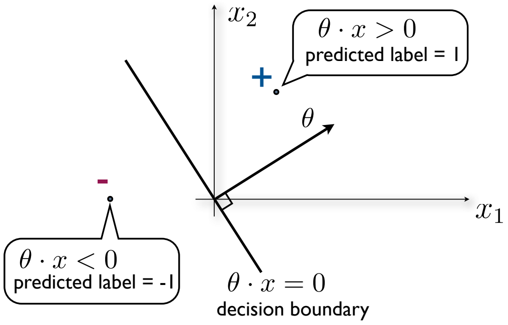


Figure 1: A linear classifier through origin.

changes from positive to negative (or vice versa). In the space of feature vectors, this transition corresponds to crossing the *decision boundary* where the argument to the sign function is exactly zero: all examples 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 line in 2d, plane in 3d and a hyper-plane in higher dimensions. This hyper-plane necessarily goes through the origin $x = 0$ (since $\theta \cdot 0 = 0$ so 0 lies on the boundary). 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 x-space along which the value of $\theta \cdot x$ would increase the most. So examples on the side of the boundary where θ points to are classified as positive ($\theta \cdot x > 0$). We can see it by rewriting $\theta \cdot x$ as $\|x\|\|\theta\| \cos(x, \theta)$, and observing that for all x on the right side of the boundary $\cos(x, \theta)$ is positive. Similarly, we can show that all the examples of the left side of the boundary are classified as negative. How are the points exactly on the boundary classified? This is a matter of definition (defined above as -1). Figure 1 illustrates these concepts in two dimensions.

We can extend 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

$$h(x; \theta, \theta_0) = \text{sign}\big(\, \theta \cdot x + \theta_0 \,\big) = \begin{cases} +1, & \theta \cdot x + \theta_0 > 0 \\ -1, & \theta \cdot x + \theta_0 \leq 0 \end{cases} \tag{5}$$

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 2 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 $\theta \cdot x + \theta_0 = 0$.

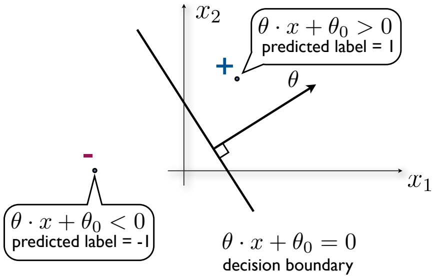

x_2
$\theta \cdot x + \theta_0 > 0$
predicted label = 1
+
θ
-
x_1
$\theta \cdot x + \theta_0 < 0$
predicted label = -1
$\theta \cdot x + \theta_0 = 0$
decision boundary

Figure 2: Linear classifier with offset parameter

2.2 Roadmap of Algorithms for Learning Linear Classifiers

Now that we have chosen a set of classifiers, we are left with the problem of selecting one in response to the training set $S_n = \{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$. There is a long history of designing algorithms and developing theory for learning linear classifiers from data. Broadly speaking, the algorithms are derived in one of two ways. First, we can design simple on-line algorithms that consider each training example in turn, updating parameters slightly in response to possible errors. Such algorithms are simple to understand (they respond by correcting mistakes), and they are efficient to run since the amount of computation they require scales linearly with the size of the training set. They often do not have a statically defined *objective function* that the algorithm minimizes with respect to the parameters. They are motivated by goals such as minimizing the training error but cannot be strictly seen to do so. They do, however, typically accompany theoretical guarantees of generalization which is what we are after in the first place. We will discuss the perceptron and passive-aggressive algorithms in this category. The second type of algorithm comes about as a method that directly minimizes a clearly specified objective function with respect to the classifier parameters. The objective function may reflect the “error”, “cost”, ”loss” or ”risk” of a linear classifier. By defining the objective, we decide what we are after based on the training set and the algorithms are designed so as to directly minimize this objective. Methods in this category include the Support Vector Machine (SVM) and its on-line variant the Pegasos algorithm. Note that we can have an on-line algorithm (Pegasos) designed to directly minimize an objective (defined later) as well as a mistake driven on-line algorithm (Perceptron) that does not.

2.3 Training error and Linear Separation

One possible objective function we could define is simply the number of training errors that the classifier makes. If we were to design an algorithm for linear classifiers based on this objective, our goal would be to find θ that results in the fewest mistakes on the training set:

$$\mathcal{E}_n(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} [\![\, y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0 \,]\!] \tag{6}$$

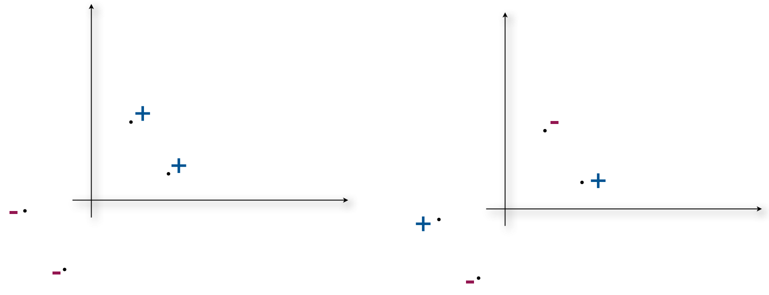
where $[\![\cdot]\!]$ returns 1 if the logical expression in the argument is true, and zero otherwise. The training error here is the fraction of training examples for which the classifier with parameters θ, θ_0 predicts the wrong label. Note that the last expression on the right calls all predictions for which $y(\theta \cdot x + \theta_0) \leq 0$ as mistakes. This may differ slightly from $[\![\, y^{(t)} \neq h(x^{(t)}; \theta, \theta_0)\,]\!]$ when points fall right on the boundary. We will use $y(\theta \cdot x + \theta_0) \leq 0$ as the more accurate measure of error (if we don't know, it should be an error). Now, the training error $\mathcal{E}_n(\theta, \theta_0)$ can be evaluated for any choice of parameters θ, θ_0. As a result, the error can be also minimized with respect to θ, θ_0.

So, what would a reasonable algorithm be for finding $\hat{\theta}$, $\hat{\theta}_0$ that minimizes $\mathcal{E}_n(\theta, \theta_0)$? 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. Note that we are assuming here that since our set of classifiers is quite constrained, finding a classifier that works well on the training set is likely to also work well on the test set, i.e., generalize well. We will discuss generalization more formally later.

We will initially consider a special case where a solution can be found – when there exists a linear classifier 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. We will provide here with two definitions *linear separation* depending on whether we include the offset parameter or not.

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

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



linearly separable through origin

not linearly separable

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?

2.4 The Perceptron algorithm

Some of the simplest and useful algorithms for learning linear classifiers are *mistake driven on-line algorithms*. They are guaranteed to find a solution with zero training error if such a solution exists (realizable case) though their behavior in the unrealizable case is less well-defined. For simplicity, we will begin with the problem of learning linear classifiers through origin and bring back the offset parameter later. Now, the mistake driven algorithms in this case start with a simple classifier, e.g., $\theta = 0$ (zero vector), and successively try to adjust the parameters, based on each training example in turn, so as to correct any mistakes. Of course, since each step in the algorithm is not necessarily decreasing $\mathcal{E}_n(\theta)$, we will have to separately prove that it will indeed find a solution with zero error.

The simplest (and oldest!) algorithm of this type is the *perceptron* algorithm.

Algorithm 1 Perceptron Algorithm (without offset)

```
procedure PERCEPTRON($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, $T$)
    $\theta = 0$ (vector)
    for $t = 1, \ldots, T$ do
        for $i = 1, \ldots, n$ do
            if $y^{(i)}(\theta \cdot x^{(i)}) \leq 0$ then
                $\theta = \theta + y^{(i)} x^{(i)}$
    return $\theta$
```

In this algorithm, we go through the training set T times, adjusting the parameters slightly in response to any mistake. No update is made if the example is classified correctly. Since parameters are changed only when we encounter a mistake, we can also track their evolution as a function of the mistakes, i.e., denote $\theta^{(k)}$ as the parameters obtained after exactly k mistakes on the training set. By definition $\theta^{(0)} = 0$. Looking at the parameters in this way will be helpful in understanding what the algorithm does.

We should first establish that the perceptron updates tend to correct mistakes. Too see this, consider a simple two dimensional example in figure 3. 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

$$\theta^{(1)} = \theta^{(0)} + x^{(1)} \tag{7}$$

$$\theta^{(2)} = \theta^{(1)} + (-1)x^{(2)} \tag{8}$$

In this simple case, both updates result in correct classification of the respective examples,

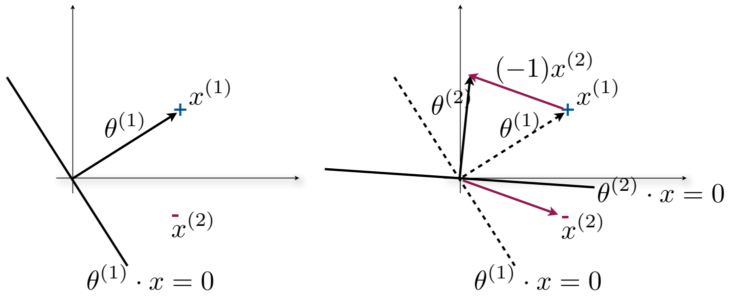


Figure 3: The perceptron update rule

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?

Let's look at the updates more algebraically. Note that when we make a mistake the product $y^{(i)}(\theta^{(k)} \cdot x^{(i)})$ is negative or zero. Suppose we make a mistake on $x^{(i)}$. Then the updated parameters are given by $\theta^{(k+1)} = \theta^{(k)} + y^{(i)}x^{(i)}$. If we consider classifying the same example $x^{(i)}$ again right after the update, using the new parameters $\theta^{(k+1)}$, then

$$\begin{aligned}
y^{(i)}(\theta^{(k+1)} \cdot x^{(i)}) &= y^{(i)}(\theta^{(k)} + y^{(i)}x^{(i)}) \cdot x^{(i)} && (9)\\
&= y^{(i)}(\theta^{(k)} \cdot x^{(i)}) + (y^{(i)})^2(x^{(i)} \cdot x^{(i)}) && (10)\\
&= y^{(i)}(\theta^{(k)} \cdot x^{(i)}) + \|x^{(i)}\|^2 && (11)
\end{aligned}$$

In other words, the value of $y^{(i)}(\theta \cdot x^{(i)})$ 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 $y^{(i)}(\theta \cdot x^{(i)})$ becomes strictly 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. Luckily, we can show that the algorithm does converge in the realizable case (we assume that T, the number of passes through the training set, is large enough to allow this).

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. The convergence guarantee holds independently of the order in which the points are traversed. While the order does impact the number of mistakes that the algorithm makes, it does not change the fact that it converges after a finite number of mistakes. Similarly, we could use any (finite) initial setting of θ without loosing the convergence guarantee but again affecting the number of mistakes accumulated.

The perceptron algorithm and the above statements about convergence naturally extend to the case with the offset parameter.

Algorithm 2 Perceptron Algorithm

```
procedure PERCEPTRON($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, $T$)
    $\theta = 0$ (vector), $\theta_0 = 0$ (scalar)
    for $t = 1, \ldots, T$ do
        for $i = 1, \ldots, n$ do
            if $y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0$ then
                $\theta = \theta + y^{(i)}x^{(i)}$
                $\theta_0 = \theta_0 + y^{(i)}$
    return $\theta$, $\theta_0$
```

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. In other words, we map our examples $x \in R^d$ to $x' \in R^{d+1}$ such that $x' = [x_1, \ldots, x_d, 1]^T$, and our parameters $\theta \in R^d$ to $\theta' \in R^{d+1}$ such that $\theta' = [\theta_1, \ldots, \theta_d, \theta_0]^T$. In this setup, the perceptron algorithm through origin will reduce exactly to the algorithm shown above. In terms of convergence, if training examples are linearly separable (not necessarily through the origin), then the above perceptron algorithm with the offset parameter converges after a finite number of mistakes.

What if the training examples are not linearly separable? In this case, the algorithm will continue to make updates during each of the T passes. It cannot converge. Better algorithms exist for this setting, and we will discuss them later on.

2.5 Perceptron Convergence Theorem

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 $S_n = \{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$ are linearly separable with margin γ if there exists a parameter vector $\hat{\theta}$ and offset parameter $\hat{\theta}_0$ such that $y^{(i)}(\hat{\theta} \cdot x^{(i)} + \hat{\theta}_0)/\|\hat{\theta}\| \geq \gamma$ for all $i = 1, ..., n$.

If you take any linear classifier defined by $\hat{\theta}$ and $\hat{\theta}_0$ that correctly classifies the training examples, then

$$\frac{y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} \tag{12}$$

is the orthogonal distance of the 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),

$$\frac{y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} = \frac{(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0)}{\|\hat{\theta}\|} - \frac{\overbrace{(\hat{\theta} \cdot x^0 + \hat{\theta}_0)}^{=0}}{\|\hat{\theta}\|} = \frac{\hat{\theta} \cdot (x^{(t)} - x^0)}{\|\hat{\theta}\|} = \|x^{(t)} - x^0\| \tag{13}$$

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

Let's now go back to the perceptron algorithm, without the offset parameter to keep things simple. We will consider running the algorithm with large T such that it will no longer make any mistakes. Clearly, it can stop making mistakes only if the training examples are linearly separable. Our goal is to try to characterize when the algorithm converges after only a small number of mistakes (when the problem is "easy"). The result will say something about generalization as well.

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 $i = 1, \ldots, n$ and some $\gamma > 0$.

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

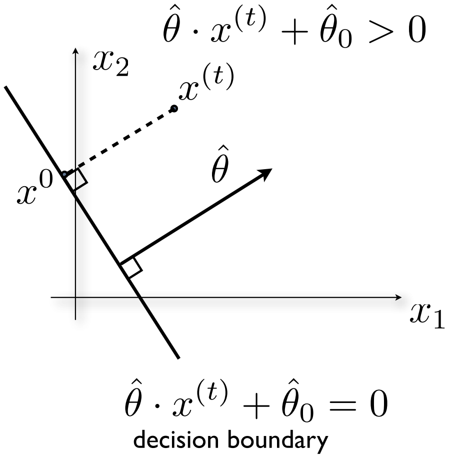

$\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0 > 0$
x_2
$x^{(t)}$
x^0
$\hat{\theta}$
x_1
$\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0 = 0$
decision boundary

Figure 4: Orthogonal projection onto the boundary

Note that the two assumptions constrain what the examples and labels can be, i.e., they constrain the training set. 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 on the examples and labels $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, n$

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 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

$$cos(\theta^{(k)}, \theta^*) = \frac{\theta^{(k)} \cdot \theta^*}{\|\theta^{(k)}\| \|\theta^*\|} \tag{14}$$

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

$$\frac{\theta^{(k)} \cdot \theta^*}{\|\theta^*\|} = \frac{(\theta^{(k-1)} + y^{(i)}x^{(i)}) \cdot \theta^*}{\|\theta^*\|} = \frac{\theta^{(k-1)} \cdot \theta^*}{\|\theta^*\|} + \frac{y^{(i)}x^{(i)} \cdot \theta^*}{\|\theta^*\|} \geq \frac{\theta^{(k-1)} \cdot \theta^*}{\|\theta^*\|} + \gamma \geq k\gamma \quad (15)$$

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. Therefore, we can prove the last transition by induction on the number of updates k. For the base case $k = 1$, $\frac{\theta^{(k-1)} \cdot \theta^*}{\|\theta^*\|} = 0$ due to our initialization of $\theta^{(0)}$ to zero, and $\frac{\theta^{(1)} \cdot \theta^*}{\|\theta^*\|} \geq \gamma$. Every inductive step, increases this value by γ.

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

$$\begin{aligned} \|\theta^{(k)}\|^2 = \|\theta^{(k-1)} + y^{(i)}x^{(i)}\|^2 &= \|\theta^{(k-1)}\|^2 + \overbrace{2y^{(i)}\theta^{(k-1)} \cdot x^{(i)}}^{<0} + \overbrace{\|x^{(i)}\|^2}^{\leq R^2} && (16) \\ &\leq \|\theta^{(k-1)}\|^2 + R^2 \leq kR^2 && (17) \end{aligned}$$

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,

$$cos(\theta^{(k)}, \theta^*) = \frac{\theta^{(k)} \cdot \theta^*}{\|\theta^{(k)}\|\|\theta^*\|} \geq \frac{k\gamma}{\sqrt{k}R} = \sqrt{k}\frac{\gamma}{R} \quad (18)$$

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.

One additional remarkable aspect of this result is that we can apply it to an infinite sequence of examples. If assumptions (A) and (B) hold for this infinite sequence, then the perceptron algorithm makes only $(R/\gamma)^2$ mistakes along this sequence! So, eventually it will be a perfect classifier even for test examples (that conform to (A) and (B)).

2.6 Beyond Realizable Case: Averaged Perceptron

When our training data is not linearly separable, the perceptron algorithm will not converge and continues updating until stopped (after T passes). We can adjust the algorithm a little bit to extract a reasonable classifier even in this case.

In the course of the algorithm, parameters that remain intact for a large number of examples are better than others. It seems therefore reasonable that we should track the parameters as the algorithm runs and take their average as the final answer. In other words, we average parameters that are present at each step, not just parameters after each update, so as to emphasize parameters that seem to work longer than others. This is called the *averaged perceptron* and works quite well in practice.

Algorithm 3 Averaged Perceptron Algorithm

```
procedure AVERAGED PERCEPTRON($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, $T$)
    $\theta = 0$ (vector), $\theta_0 = 0$ (scalar)
    $\bar{\theta} = 0$ (vector), $\bar{\theta}_0 = 0$ (scalar)
    $c = 1$
    for $t = 1, \ldots, T$ do
        for $i = 1, \ldots, n$ do
            if $y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \leq 0$ then
                $\theta = \theta + y^{(i)} x^{(i)}$
                $\theta_0 = \theta_0 + y^{(i)}$
                $\bar{\theta} = \bar{\theta} + c\, y^{(i)} x^{(i)}$
                $\bar{\theta}_0 = \bar{\theta}_0 + c\, y^{(i)}$
            $c = c - 1/(nT)$
    return $\bar{\theta}$, $\bar{\theta}_0$
```

Did you notice how we managed to insert the update of averaged parameters, that should be averaged on each step (update or not), inside the part that is executed only if there's an update? This is much more efficient, yet produces the same answer. Can you figure out why this is correct?

2.7 Linear Classification with Loss Functions

Our hope so far has been that if we find a linear classifier with few (or no) training errors, then this classifier will also make few mistakes on the test set. What's the basis for this belief? We have introduced two different ways to think about this. The perceptron convergence proof relies on the assumption that there exists some (unknown to us) linear classifier that separates the training examples with a large margin. If this is indeed so, then the algorithm convergence quickly to one of the many possible separating solutions. If we extend this assumption to apply to both training *and* test examples, then the perceptron algorithm is guaranteed to make a small number of test errors as well. Note that this is an assumption *about the problem*, that it really is easy to solve (when the margin is large). So the perceptron algorithm works well on easy problems.

The second rationale is that the set of linear classifiers is already constrained enough such that, regardless of how hard the task is, the fraction of training errors reflects the fraction of test errors. This rationale depends somewhat on the dimension of the feature vectors. Let's see how it breaks down (we will come back to it more formally later). Suppose we are classifying documents using bag-of-words feature vectors. In other words, each document is mapped to a vector where each coordinate corresponds to a count of the number of times a word appears in the document. The dimension could therefore easily be in the tens of thousands! Why is this a problem then? Well, if each document were just a single distinct word, i.e., each document is tied to a single coordinate x_j, then we could just set the sign of the associated parameter θ_j in the linear classifier to correctly classify all such documents. High dimensional feature vectors give linear classifiers a lot of power. Indeed, we will make them shortly infinitely powerful by expanding the feature vectors to be infinite dimensional! Clearly we should work a little harder to ensure that such classifiers still generalize well.

One thing we can do to further constrain the set of linear classifiers is to measure errors differently on the training set. This will not actually exclude any linear classifier from being selected but will strongly bias the selection towards particular kinds of classifiers, thereby effectively restricting our choices (in a good way) and improving generalization.

The key motivation is that not all errors should be treated the same. Points near the decision boundary are inherently uncertain and should be penalized less than those that are overtly placed on the wrong side of the boundary. Formally, we will use a "loss function" that simply specifies numerically how badly we classify each example. The perceptron algorithm implicitly assumes a simple zero-one loss (just the error), i.e.,

$$\text{Loss}_{0,1}(y(\theta \cdot x + \theta_0)x) = [\![\, y(\theta \cdot x + \theta_0) \leq 0 \,]\!] \tag{19}$$

Note that while this loss function is a function of the "agreement" $y(\theta \cdot x + \theta_0)$ it only cares about whether the agreement is positive (correct classification) or not (mistake). It doesn't depend on the magnitude of the agreement (or lack thereof). We can introduce a more sensitive loss function by penalizing any prediction for which the agreement drops below one (as opposed to zero), and increasing the penalty for greater violations. For example, we can use the so-called Hinge loss

$$\text{Loss}_h(y(\theta \cdot x + \theta_0)) = \max\{0, 1 - y(\theta \cdot x + \theta_0)\} = \begin{cases} 1 - y(\theta \cdot x + \theta_0) \text{ if } y(\theta \cdot x + \theta_0) \leq 1 \\ 0, \text{ o.w.} \end{cases} \tag{20}$$

Let's understand geometrically how different parameter choices θ, θ_0 affect the Hinge loss that we incur on each example. The relevant geometric object(s) are no longer the decision boundaries but parallel boundaries around the decision boundary known as the positive and negative margin boundaries $\{x : \theta \cdot x + \theta_0 = 1\}$ and $\{x : \theta \cdot x + \theta_0 = -1\}$, respectively. See Figure 5 below. In order for a positively labeled point to get zero Hinge loss, it must be on the correct (positive) side of the positive margin boundary $\{x : \theta \cdot x + \theta_0 = 1\}$. If we figuratively moved a training example past the relevant margin boundary, the Hinge loss would increase linearly as a function of the degree of violation. The margin boundaries are at distance $1/\|\theta\|$ 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, $y(\theta \cdot x + \theta_0) = 1$, and the orthogonal distance from the boundary is $y(\theta \cdot x + \theta_0)/\|\theta\| = 1/\|\theta\|$. If we kept the decision boundary intact but increased $\|\theta\|$, then the margin boundaries would draw closer to each other. In contrast, a small $|\theta\|$ implies that margin boundaries are far apart, increasing the likelihood of violations.

So, how do we search for linear classifier parameters θ, θ_0, while measuring the performance with respect to the Hinge loss? The goal is to bias our search towards classifiers with small $\|\theta\|$ so that the margin boundaries remain far apart. This forces us to select classifiers where the decision boundary has lots of "empty space" between positive and negative examples. In other words, in contrast to the perceptron algorithm, we try to find *large margin classifiers*. Of course, sometimes there will be examples that violate the margin boundaries but we seek to balance such violations against how large the margin is (how small $\|\theta\|$ can be). With this intuition, we will first introduce a useful perceptron-like classifier known as passive-aggressive classifier. After this, we will explicitly formulate the optimization problem for finding the maximum margin classifier, obtaining a method called Support Vector Machine (SVM) and its on-line variant Pegasos.

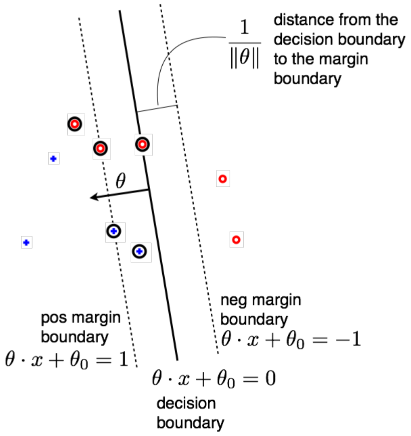

$\frac{1}{\|\theta\|}$
distance from the decision boundary to the margin boundary
θ
pos margin boundary
$\theta \cdot x + \theta_0 = 1$
neg margin boundary
$\theta \cdot x + \theta_0 = -1$
$\theta \cdot x + \theta_0 = 0$
decision boundary

Figure 5: Decision and margin boundaries for a linear classifier. The blue '+'s are positively labeled points while red 'o's are negative. Points that are circled incur a non-zero Hinge loss as they violate the margin boundaries.

2.8 Passive-Aggressive Algorithm

Passive-aggressive algorithm is an on-line algorithm just like perceptron. In other words, it will process each training example in turn, updating parameters slightly in response to the example (if needed), and moves on. The difference is that the algorithm tries to explicitly minimize the Hinge loss on that example (the aggressive part) while keeping the parameters close to where they were prior to seeing the example (the passive part). The passive part is necessary so as not to overwrite what was learned from the previous training examples. The algorithm has similar guarantees to the perceptron mistake bound but now expressed in terms of the Hinge loss. We will consider here the algorithm without the associated guarantees (see Crammer et al., 2006 for a more formal treatment)

To keep things simple, let's define the algorithm without the offset parameter θ_0. To include the offset after the fact, we can always think of adding a fixed coordinate to all feature vectors. Now, assume that $\theta^{(k)}$ are our current parameters which are initially set to zero, i.e., $\theta^{(0)} = 0$ (vector). If (x, y) is the training example in question, the algorithm finds $\theta = \theta^{(k+1)}$ that minimizes

$$\frac{\lambda}{2}\|\theta - \theta^{(k)}\|^2 + \text{Loss}_h(y\theta \cdot x) \tag{21}$$

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

$$\theta^{(k+1)} = \theta^{(k)} + \eta\, yx \tag{22}$$

The only difference is that we have a learning rate (step-size) parameter η which will depend on λ, $\theta^{(k)}$, as well as the training example (x, y).

How did we deduce that the update should be of this form? Well, the first term $\|\theta - \theta^{(k)}\|^2$ in Eq.(21) doesn't care about the direction at all, only how much we move away from $\theta^{(k)}$. The loss term, on the other hand, does have a directional preference. The gradient with respect to the parameters of the loss provides the steepest ascent direction. Thus the negative gradient specifies the steepest descent direction[1]:

$$-\nabla_\theta \text{Loss}_h(y\theta \cdot x) = -\nabla_\theta \left\{ \begin{array}{ll} 1 - y\theta \cdot x & \text{if } y\theta \cdot x \leq 1 \\ 0 \text{ o.w.} & \end{array} \right. = \left\{ \begin{array}{ll} yx & \text{if } y\theta \cdot x \leq 1 \\ 0 \text{ o.w.} & \end{array} \right. \tag{23}$$

Note that the direction (if we move at all) doesn't depend on θ. We can therefore safely assume that $\theta^{(k+1)} = \theta^{(k)} + \eta\, yx$ for some value of η which may be zero. To complete the derivation, we can plug this form of the solution back into Eq.(21) and minimize it with respect to this single real number η. This yields a closed form solution

$$\eta = \min \left\{ \frac{\text{Loss}_h(y\theta^{(k)} \cdot x)}{\|x\|^2}, \frac{1}{\lambda} \right\} \tag{24}$$

[1] Technically, this is a sub-gradient since the loss is not differentiable at a single point.

What happens if (x, y) is already beyond the margin boundaries relative to $\theta^{(k)}$? In this case, $\text{Loss}_h(y\theta^{(k)} \cdot x) = 0$, and the step-size evaluates to zero as well. In other words, we will not perform any updates on correctly classified 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 more restrained than what perceptron would make.

Algorithm 4 Passive-Aggressive Algorithm (without offset)

```
procedure PASSIVE-AGGRESSIVE($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, $\lambda$, $T$)
    $\theta = 0$ (vector)
    for $t = 1, \ldots, T$ do
        for $i = 1, \ldots, n$ do
            $\eta = \min\left\{\frac{\text{Loss}_h(y^{(i)}\theta \cdot x^{(i)})}{\|x^{(i)}\|^2}, \frac{1}{\lambda}\right\}$
            $\theta = \theta + \eta\, y^{(i)} x^{(i)}$
    return $\theta$
```

We can view the parameter λ in the algorithm as a *regularization parameter*. It's role is to keep each parameter update small. Moreover, since the initial parameters are zero, it also keeps the norm of the parameter vector $\|\theta\|$ small, consequently attempting to find a large margin solution. When such a solution doesn't exist, $\|\theta\|$ can slowly increase if we continue updating, i.e., if T, the number of passes, is large. Both λ and T affect the quality of the solution found. They can be se by optimizing the classifier performance on a separate *validation data* or via *cross-validation*.

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 $\theta^{(nT)}$. The averaging takes place whether or not the parameters receive a non-zero update, and can be implemented efficiently exactly as in the averaged perceptron algorithm. 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.

Figure 6 illustrates how passive-aggressive algorithm prefers solutions different from the perceptron algorithm.

2.9 Support Vector Machine

We have so far discussed only *on-line algorithms* that process each training example in turn with the overall goal of trying to minimize errors (perceptron) or losses (passive-aggressive) on the training set. However, neither goal was formulated directly as a minimization problem pertaining to all the training examples at once. We will start here with an *off-line algorithm* known as the support vector machine.

The support vector machine tries to 1) minimize the average Hinge loss on the training examples while 2) pushing margin boundaries apart by reducing $\|\theta\|$. These two goals oppose each other and need to be balanced. The more we emphasize the losses, the more the resulting classifier is determined by the few examples right near the decision boundary. In contrast, if we extend the margin boundaries, we incur losses on examples that are further away and the solution will be defined by where the bulk of the positive and negative

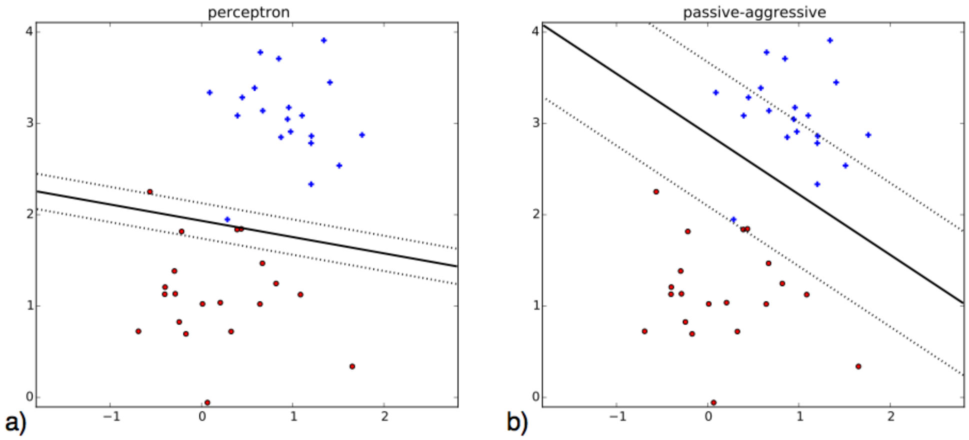


Figure 6: Decision and margin boundaries obtained after 10 passes through the training set via a) the perceptron algorithm or b) the passive-aggressive algorithm. Both algorithms included the offset parameter.

examples are. We will discuss a bit later how to find the right balance. For now, we will focus on formulating the problem with a prescribed balance. We find θ, θ_0 so as to minimize

$$\frac{1}{n}\sum_{i=1}^{n} \text{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\big) + \frac{\lambda}{2}\|\theta\|^2 \tag{25}$$

where the *regularization parameter* λ balances these two goals. The second term $\|\theta\|^2/2$ is a *regularization term* that pulls the parameters towards a default answer (here zero vector). By including the regularization term we have made the problem well-posed even when data is conflicting or absent.[2] Figure 7 shows how the solution changes if we change λ when the data are not linearly separable.

Quadratic program

How do we actually find the minimizing θ and θ_0? The SVM objective function can be reformulated as a *quadratic program* and solved with a variety of available optimization packages. Specifically, we can minimize

$$\frac{1}{n}\sum_{i=1}^{n} \xi_i + \frac{\lambda}{2}\|\theta\|^2 \quad \text{subject to} \quad y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \geq 1 - \xi_i,\;\; \xi_i \geq 0,\;\; i = 1, \ldots, n \tag{26}$$

with respect to θ, θ_0, and $\xi_i \geq 0$, $i = 1, \ldots, n$. The *slack variables* ξ_i are introduced to represent the Hinge loss in terms of linear constraints so that the overall problem has a quadratic objective with linear constraints. Can you figure out why optimizing ξ_i (keeping

[2] While $\theta = 0$ (zero vector) in the absence of data, the offset parameter cannot be determined in this case.

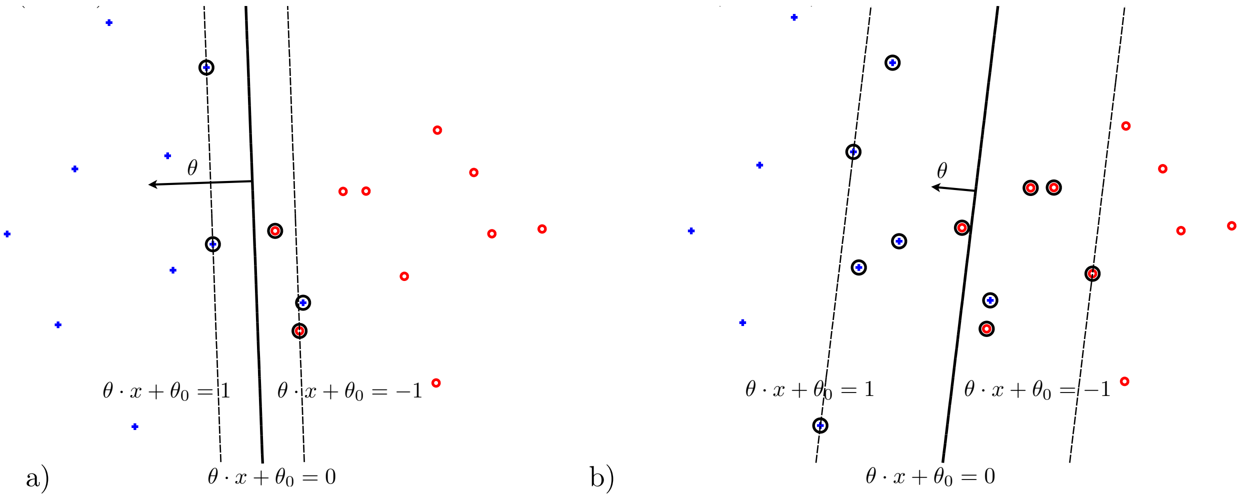


Figure 7: SVM solution when a) λ is small so as to minimize margin violations b) λ is large and we accept margin violations in favor of increasing the size of the margin. Note that $\|\theta\|$ is inversely related to the margin. We have circled all the points that either violate the margin boundaries or lie exactly on them.

other things fixed) will make it exactly the loss on the i^{th} example, i.e., reconstituting $\text{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\big)$ in the objective?

While it would be good to to find such a solution, this quadratic program doesn't scale well with the number of training examples. Indeed, it takes about $O(dn^3)$ computation to solve it reasonably well in practice, limiting it to smaller problems (on the order of 10,000 training examples without additional tricks).

2.9.1 Stochastic Gradient Descent and the Pegasos Algorithm

We can always try to solve the same SVM minimization problem approximately using much simpler (and scalable) *stochastic gradient methods*. These methods, as on-line algorithms, consider each training example in turn, move the parameters slightly in the negative gradient direction, and move on. This is similar but not exactly the same as the passive-aggressive algorithm. The difference is that the regularization term now explicitly controls the norm $\|\theta\|$ (i.e., 1/margin) rather than indirectly through keeping the sequence of updates small. Let's again drop the offset parameter θ_0 and rewrite the SVM objective as an average of terms

$$\frac{1}{n}\sum_{i=1}^{n} \text{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) + \frac{\lambda}{2}\|\theta\|^2 = \frac{1}{n}\sum_{i=1}^{n}\left[\text{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) + \frac{\lambda}{2}\|\theta\|^2\right] \qquad (27)$$

When the objective function to be minimized has this form we can always select a term at random and apply a stochastic gradient descent update. Specifically, in response to each example, we update[3]:

$$
\begin{aligned}
\theta &= \theta - \eta \nabla_\theta \left[\mathrm{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) + \frac{\lambda}{2}\|\theta\|^2 \right] && (28)\\
&= \theta - \eta \nabla_\theta \left[\mathrm{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) \right] - \eta \nabla_\theta \left[\frac{\lambda}{2}\|\theta\|^2 \right] && (29)\\
&= \theta - \eta \nabla_\theta \left[\mathrm{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) \right] - \eta \lambda \theta && (30)\\
&= (1 - \lambda\eta)\theta - \eta \nabla_\theta \left[\mathrm{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)})\big) \right] && (31)\\
&= (1 - \lambda\eta)\theta + \eta \begin{cases} y^{(i)}x^{(i)} & \text{if } y^{(i)}(\theta \cdot x^{(i)}) \le 1 \\ 0 & \text{o.w.} \end{cases} && (32)
\end{aligned}
$$

where η is the learning rate. If we decrease the learning rate appropriately, these gradient descent updates are indeed guaranteed to converge to the same SVM solution. For example, we can change η_k after each update k such that $\sum_{k=1}^{\infty} \eta_k = \infty$ (not summable) and $\sum_{k=1}^{\infty} \eta_k^2 < \infty$ (square summable). These two conditions are necessary to make sure that, regardless of where we start (or end up during optimization), we can move to the optimum, and that the "noise" inherent in the gradient updates (stochastic choice of terms, finite step-size) vanishes in the end. For example, $\eta_k = 1/(k+1)$ is a valid (albeit somewhat slow) choice.

Algorithm 5 Pegasos Algorithm (without offset)

```
procedure PEGASOS($\{(x^{(i)}, y^{(i)}), i = 1, \ldots, n\}$, $\lambda$, $T$)
    $\theta = 0$ (vector)
    for $t = 1, \ldots, T$ do
        Select $i \in \{1, \ldots, n\}$ at random
        $\eta = 1/t$
        if $y^{(i)}\theta \cdot x^{(i)} \le 1$ then
            $\theta = (1 - \eta\lambda)\theta + \eta\, y^{(i)} x^{(i)}$
        else
            $\theta = (1 - \eta\lambda)\theta$
    return $\theta$
```

2.9.2 Maximum Margin Hyperplane – Realizable Case

You may see the SVM optimization problem written slightly differently. First, let's multiply the objective by $1/\lambda$. Since λ is constant with respect to θ and θ_0, this is fine and doesn't change the minimizing solution. We get

$$
\overbrace{\left(\frac{1}{\lambda n}\right)}^{=C} \sum_{i=1}^{n} \mathrm{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\big) + \frac{1}{2}\|\theta\|^2 \qquad (33)
$$

[3] Technically, this is again a sub-gradient update

where the parameter C tells us how much to emphasize the training losses. Whether you prefer to use C or λ as the regularization parameter is immaterial. Just remember that they are inversely related.

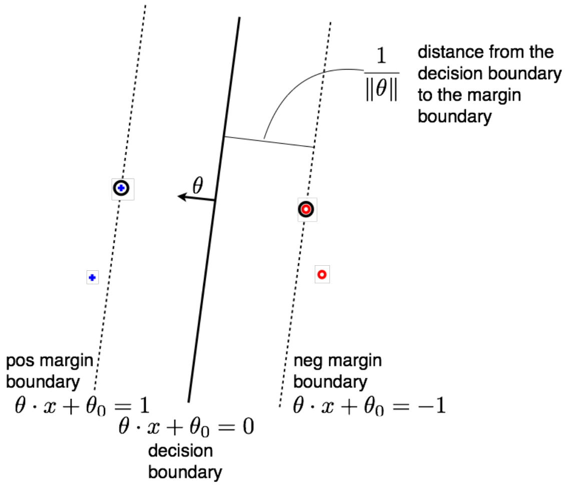
What will happen if $C \to \infty$ (or C is very large)? In other words, what do we get as the solution if we adamantly do not want any margin violations? Clearly, the question is relevant only when the data are linearly separable. In this extreme case, we can write the problem as

$$\text{minimize } \frac{1}{2}\|\theta\|^2 \text{ subject to } \text{Loss}_h\big(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)\big) = 0,\ i = 1, \ldots, n \tag{34}$$

or, equivalently,

$$\text{minimize } \frac{1}{2}\|\theta\|^2 \text{ subject to } y^{(i)}(\theta \cdot x^{(i)} + \theta_0) \geq 1,\ i = 1, \ldots, n \tag{35}$$

This is again a *quadratic program* (quadratic objective to be minimized subject to linear constraints). Geometrically, by minimizing $\|\theta\|$ we push the margin boundaries farther and farther apart until we can no longer do this without violating some of the constraints. This gives us the linear separator whose decision boundary is furthest away from all the training examples as shown in Figure 8 below. If the training examples are separable and contain at least one positive and one negative example, this maximum margin solution is unique. However, 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 better to be able to "give up" on trying to satisfy all the classification constraints.



$\frac{1}{\|\theta\|}$
distance from the decision boundary to the margin boundary
θ
pos margin boundary
$\theta \cdot x + \theta_0 = 1$
$\theta \cdot x + \theta_0 = 0$
decision boundary
neg margin boundary
$\theta \cdot x + \theta_0 = -1$

Figure 8: Maximum margin linear separator

2.10 Exercises

Exercise (2.1) Consider applying the perceptron algorithm (through the origin) based on a small training set containing three points: $x^{(1)} = [-1, 1]$, $x^{(2)} = [0, -1]$, and $x^{(3)} = [1.5, 1]$ with labels $y^{(1)} = 1$, $y^{(2)} = -1$, and $y^{(3)} = 1$. Given that the algorithm starts with $\theta^{(0)} = 0$, the first point that the algorithm sees is always considered a mistake. The algorithm starts with some data point and then cycles through the data until it makes no further mistakes.

1. How many mistakes does the algorithm make until convergence if the algorithm starts with data point $x^{(1)}$? How many mistakes does the algorithm make if it starts with data point $x^{(2)}$? Draw a diagram showing the progression of the plane as the algorithm cycles.

2. Now assume that $x^{(3)} = [10, 1]$. How many mistakes does the algorithm make until convergence if cycling starts with data point $x^{(1)}$? How many if it starts with data point $x^{(2)}$? Draw a diagram showing the progression of the plane as the algorithm cycles.

3. Explain why there is a difference between the number of mistakes made by the algorithm in part a) and part b).

4. Briefly describe an adversarial procedure for selecting the order and value of labeled data points so as to maximize the number of mistakes the perceptron algorithm makes before converging. Assume that the data is indeed linearly separable, by a hyperplane that you (but not the algorithm) know.

Exercise (2.2) We initialized the perceptron algorithm with $\theta = 0$. In this problem we will explore this initialization choice.

1. Given a set of linearly separable training examples, show that the initialization of θ does not impact the perceptron algorithm's ability to converge. You only need to show it for perceptron without offset.

2. Now suppose we run the perceptron algorithm twice, but initialize θ to two different values (possibly non-zero) for each run and train on the examples in the same order until convergence. Do they converge to the same θ? If so, provide a proof. If not, provide a counterexample. You may use either with or without offset.

3. Both classifiers converge, so their performance on the training set is the same. Does their performance on a test set perform the same as well?

4. The following table shows a data set and the number of times each point is misclassified during a run of the perceptron algorithm (with offset). θ is initialized to zero.

i	$x^{(i)}$	y	times misclassified
1	[-3, 2]	+1	1
2	[-1, 1]	+1	0
3	[-1, -1]	-1	2
4	[2, 2]	-1	1
5	[1, -1]	-1	0

Write down the post-training θ.

Exercise (2.3)

1. Consider the AND function defined over three binary variables: $f(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge x_3)$.

   We aim to find a θ such that, for any $x = [x_1, x_2, x_3]$, where $x_i \in \{0, 1\}$:

   $$\theta \cdot x + \theta_0 > 0 \text{ when } f(x_1, x_2, x_3) = 1, \text{ and}$$

   $$\theta \cdot x + \theta_o < 0 \text{ when } f(x_1, x_2, x_3) = 0.$$

   (a) If $\theta_0 = 0$ (no offset), would it be possible to learn such a θ?

   (b) Would it be possible to learn the pair θ and θ_o?

2. You are given the following labeled data points:

   - Positive examples: $[-1, 1]$ and $[1, -1]$,
   - Negative examples: $[1, 1]$ and $[2, 2]$.

   For each of the following parameterized families of classifiers, find the parameters of a family member that can correctly classify the above data, or explain (with a clear diagram and/or words) why no such family member exists.

   (a) Inside or outside of an origin-centered circle with radius r,

   (b) Inside or outside of an $[x, y]$-centered circle with radius r,

   (c) Above or below a line through the origin with normal θ,

   (d) Above or below a line with normal θ and offset θ_0.

3. Which of the above are families of linear classifiers?

Exercise (2.4) Consider a sequence of n-dimensional data points, $x^{(1)}, x^{(2)}, ...$, and a sequence of m-dimensional feature vectors, $z^{(1)}, z^{(2)}, ...$, extracted from the x's by a linear transformation, $z^{(i)} = Ax^{(i)}$. If m is much smaller than n, you might expect that it would be easier to learn in the lower dimensional feature space than in the original data space.

1. Suppose $n = 6$, $m = 2$, z_1 is the average of the elements of x, and z_2 is the average of the first three elements of x minus the average of fourth through sixth elements of x. Determine A.

2. Suppose $h(z) = sign(\theta_z \cdot z)$ is a classifier for the feature vectors, and $h(x) = sign(\theta_x \cdot x)$ is a classifier for the original data vectors. Given a θ_z that produces good classifications of the feature vectors, is there a θ_x that will identically classify the associated x's.

3. Given the same classifiers as in (b), if there is a θ_x that produces good classifications of the data vectors, will there *always* be a θ_z that will identically classify the associated z's? An explanation on intuition is sufficient.

   Under what conditions on A will such a θ_z always exist?

4. If $m < n$, can the perceptron algorithm converge more quickly when training in z-space? If so, provide an example.

5. If $m < n$, can we find a more accurate classifier by training in z-space, as measured on the training data? How about on unseen data?

Exercise (2.5) In this exercise, we will explore questions related to perceptron convergence rates. Consider a set of n labeled training data points $\{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$, where each $y^{(t)} \in \{-1, +1\}$ is the label for the point $x^{(t)}$, a d-dimensional vector defined as follows:

$$x_i^{(t)} = \begin{cases} \cos(\pi t) & i = t \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

Recall the no-offset perceptron algorithm, and assume that $\theta \cdot x = 0$ is treated as a mistake, regardless of label. Assume in this problem that $n = d$ and that we initialize $\theta = 0$ as usual.

1. Consider the case $d = 2$. For each assignment of labels $y^{(1)}, y^{(2)}$, sketch the θ that results by running the perceptron algorithm until convergence. Convince yourself that the algorithm will make 2 updates for any ordering (and labeling) of the feature vectors.

2. Now consider the general case. Show that the no-offset perceptron algorithm will make exactly d updates to θ, regardless of the order in which we present the feature vectors, and regardless of how these vectors are labeled, i.e., no matter how we choose $y^{(t)}$, $t = 1, \ldots, t$.

3. What is the vector θ that the perceptron algorithm converges to based on this d-dimensional training set? Does θ depend on the ordering? How about the labeling?

4. Is the number of perceptron algorithm mistakes made, d, a violation of the $\frac{R^2}{\gamma^2}$ bound on mistakes we proved in class? Why or why not (Hint: $\|x^{(t)}\| = 1$ for all t, what is $\|\theta\|$?)

Exercise (2.6) As we have shown in the Section ref, the passive-aggressive (PA) algorithm (without offset) responds to a labeled training example (x, y) by finding θ that minimizes

$$\frac{\lambda}{2}\|\theta - \theta^{(k)}\|^2 + \text{Loss}_h(y\theta \cdot x) \tag{37}$$

where $\theta^{(k)}$ is the current setting of the parameters prior to encountering (x, y) and $\text{Loss}_h(y\theta \cdot x) = \max\{0, 1 - y\theta \cdot x\}$ is the hinge loss. We could replace the loss function with something else (e.g., the zero-one loss). The form of the update is similar to the perceptron algorithm, i.e.,

$$\theta^{(k+1)} = \theta^{(k)} + \eta\, yx \tag{38}$$

but the real-valued step-size parameter η is no longer equal to one; it now depends on both $\theta^{(k)}$ and the training example (x, y).

1. **Loss functions and decision boundaries**

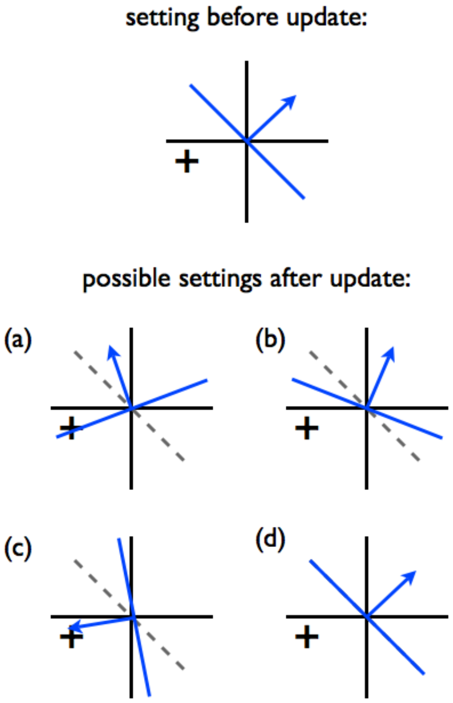
   (a) Consider minimizing the function in eq. 37 with the hinge loss. What happens as the value of λ increases? If the λ is large, should the step-size of the algorithm (η) be large or small? Explain.

   (b) Consider minimizing the function in eq. 37 and the setting of our decision boundary plotted below. We ran our PA algorithm on the next data point in our sequence - a positively-labeled vector (indicated with a +). We plotted the results of our algorithm after the update, by trying out a few different variations of loss function and λ as follows:

   1) hinge loss and a large λ
   2) hinge loss and a small λ
   3) 0-1 loss and a large λ
   4) 0-1 loss and a small λ

   Note that for a large λ, the $\frac{\lambda}{2}\|\theta - \theta^{(k)}\|^2$ term dominates and for a small λ, the $\text{Loss}_h(y\theta \cdot x)$ term dominates.

   Unfortunately, we forgot to label our result files, and want to find out which variation of algorithm corresponds to which update. Please help match up the 4 variations above with the resulting decision boundaries plotted in a-d below (the dotted lines correspond to the previous decision boundary, and the solid blue lines correspond to the new decision boundary; also, note that these are just sketches which ignore any changes to the magnitude of θ).



2. **Update equation, effect**

   (a) Suppose $\text{Loss}_h(y\theta^{(k+1)} \cdot x) > 0$ after the update $(\theta^{(k)} \rightarrow \theta^{(k+1)})$. What is the value of η that lead to this update? (Hint: you can simplify the loss function in this case).

   (b) Suppose we replace the perceptron algorithm in problem 1 with the passive aggressive algorithm. Will the number of mistakes made by the passive aggressive algorithm depend on feature vector ordering? Explain.

3 Linear regression (old version)

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

$$f(x; \theta, \theta_0) = \theta \cdot x + \theta_0 = \sum_{i=1}^{d} \theta_i x_i + \theta_0 \tag{39}$$

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.

3.1 Empirical risk and the least squares criterion

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

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) \tag{40}$$

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

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}(y^{(t)} - \theta \cdot x^{(t)}) = \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{41}$$

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

$$R_{n'}^{test}(\theta) = \frac{1}{n'}\sum_{t=n+1}^{n+n'} (y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{42}$$

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.

3.2 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

$$\nabla_\theta (y^{(t)} - \theta \cdot x^{(t)})^2/2 = (y^{(t)} - \theta \cdot x^{(t)})\nabla_\theta(y^{(t)} - \theta \cdot x^{(t)}) = -(y^{(t)} - \theta \cdot x^{(t)})x^{(t)} \tag{43}$$

As a result, the algorithm can be written as

```
set $\theta^{(0)} = 0$
    randomly select $t \in \{1, \ldots, n\}$                          (44)
        $\theta^{(k+1)} = \theta^{(k)} + \eta_k (y^{(t)} - \theta \cdot x^{(t)})x^{(t)}$
```

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.

3.2.1 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,

$$\begin{aligned}
\nabla R_n(\theta)_{\theta=\hat{\theta}} &= \frac{1}{n}\sum_{t=1}^{n} \nabla_\theta \{\, (y^{(t)} - \theta \cdot x^{(t)})^2/2 \,\}_{|\theta=\hat{\theta}} && (45)\\
&= \frac{1}{n}\sum_{t=1}^{n} \{\, -(y^{(t)} - \hat{\theta} \cdot x^{(t)})x^{(t)} \,\} && (46)\\
&= -\frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)} + \frac{1}{n}\sum_{t=1}^{n} (\hat{\theta} \cdot x^{(t)})x^{(t)} && (47)\\
&= \underbrace{-\frac{1}{n}\sum_{t=1}^{n} y^{(t)}x^{(t)}}_{=b} + \underbrace{\frac{1}{n}\sum_{t=1}^{n} x^{(t)}(x^{(t)})^T}_{=A}\ \hat{\theta} && (48)\\
&= -b + A\hat{\theta} = 0 && (49)
\end{aligned}$$

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

$$b = \frac{1}{n}X^T\vec{y},\ \ A = \frac{1}{n}X^TX \qquad (50)$$

3.3 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

$$J_{n,\lambda}(\theta) = \frac{\lambda}{2}\|\theta\|^2 + R_n(\theta) = \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot x^{(t)})^2/2 \tag{51}$$

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

$$\nabla_\theta \{ \frac{\lambda}{2}\|\theta\|^2 + (y^{(t)} - \theta \cdot x^{(t)})^2/2 \}_{|\theta=\theta^{(k)}} = \lambda\theta^{(k)} - (y^{(t)} - \theta^{(k)} \cdot x^{(t)})x^{(t)} \tag{52}$$

As a result, the algorithm can be rewritten as

$$\begin{aligned} &\text{set } \theta^{(0)} = 0 \\ &\quad \text{randomly select } t \in \{1, \ldots, n\} \\ &\qquad \theta^{(k+1)} = (1 - \lambda\eta_k)\theta^{(k)} + \eta_k(y^{(t)} - \theta \cdot x^{(t)})x^{(t)} \end{aligned} \tag{53}$$

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.

4 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.

4.1 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

$$\begin{aligned} J(\theta) &= \sum_{i \in D_a} \big(y^{(i)} - \theta \cdot x^{(i)}\big)^2/2 + \frac{\lambda}{2}\|\theta\|^2 && (54) \\ &= \sum_{i \in D_a} \big(y^{(i)} - \sum_{j=1}^{d} \theta_j x_j^{(i)}\big)^2/2 + \frac{\lambda}{2}\|\theta\|^2 && (55) \end{aligned}$$

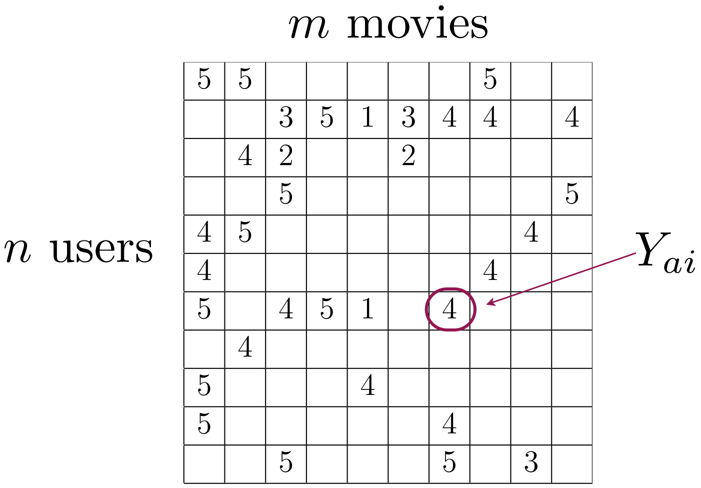
with respect to the vector of parameters θ. There are many ways to solve this optimization problem as we have already seen. We could use a simple stochastic gradient descent method or obtain the solution in closed-form. In the latter case, we set the derivative of the objective $J(\theta)$ with respect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

$$\begin{aligned} \frac{d}{d\theta_k} J(\theta) &= -\sum_{i \in D_a} \big(y^{(i)} - \sum_{j=1}^{d} \theta_j x_j^{(i)}\big) x_k^{(i)} + \lambda \theta_k && (56) \\ &= -\sum_{i \in D_a} y^{(i)} x_k^{(i)} + \sum_{j=1}^{d} \theta_j \sum_{i \in D_a} x_j^{(i)} x_k^{(i)} + \lambda \theta_k && (57) \\ &= -\sum_{i \in D_a} y^{(i)} x_k^{(i)} + \sum_{j=1}^{d} \sum_{i \in D_a} x_k^{(i)} x_j^{(i)} \theta_j + \lambda \theta_k && (58) \end{aligned}$$

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 $\lambda > 0$[4]

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.

4.2 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 9: 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

[4] $(\sum_{i \in D_a} x^{(i)} x^{(i)^T})$ 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 linear regression for movie features, when user representation is fixed, and solving it for users, when movie representation is fixed.

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 9. 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 one percent of the matrix entries had known ratings so Figure 9 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 9, 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., $Y_{ai} \in \mathcal{R}$.

4.2.1 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

$$\text{sim}(a,b) = \text{corr}(a,b) = \frac{\sum_{j \in CR(a,b)} (Y_{aj} - \tilde{Y}_a)(Y_{bj} - \tilde{Y}_b)}{\sqrt{\sum_{j \in CR(a,b)} (Y_{aj} - \tilde{Y}_a)^2} \sqrt{\sum_{j \in CR(a,b)} (Y_{bj} - \tilde{Y}_b)^2}} \tag{59}$$

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 $\text{sim}(a,b) \in [-1,1]$ 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 $\text{sim}(a,b) = 1$ 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,

$$\hat{Y}_{ai} = \bar{Y}_a + \frac{\sum_{b \in KNN(a,i)} \text{sim}(a,b)(Y_{bi} - \bar{Y}_b)}{\sum_{b \in KNN(a,i)} |\text{sim}(a,b)|} \tag{60}$$

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

$$\hat{Y}_{ai} = \bar{Y}_a + \frac{\text{sim}(a,a)(Y_{ai} - \bar{Y}_a)}{|\text{sim}(a,a)|} = \bar{Y}_a + \frac{1\,(Y_{ai} - \bar{Y}_a)}{|1|} = Y_{ai} \tag{61}$$

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.

4.2.2 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 9. 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

$$\sum_{ai \in D} (Y_{ai} - X_{ai})^2/2 + \frac{\lambda}{2} \sum_{ai} X_{ai}^2 \tag{62}$$

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,

$$\hat{X}_{ai} \quad = \quad \begin{cases} \frac{1}{1+\lambda} Y_{ai}, & (a,i) \in D \\ 0, & (a,i) \notin D \end{cases} \tag{63}$$

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

Figure 10: 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 10.

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. In 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 $n \times k$ matrix and $V = [v^{(1)}, \ldots, v^{(m)}]^T$ is $m \times k$, where $k \leq \min\{n, m\}$. We characterize each movie in terms of k features, i.e., as a vector $v^{(i)} \in \mathbb{R}^k$, while each user is represented by a vector of regression coefficients (features)

$u^{(a)} \in \mathbb{R}^k$. Both of these vectors must be learned from observations. The predicted rating is $X_{ai} = [UV^T]_{ai} = u^{(a)} \cdot v^{(i)} = \sum_{j=1}^{k} u_j^{(a)} v_j^{(i)}$ (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 $nk + mk - k^2$ where the scaling degree of freedom discussed in the context of rank 1 matrices now appears as any invertible $k \times k$ 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

$$\begin{aligned} J(U,V) &= \sum_{(a,i)\in D} (Y_{ai} - [UV^T]_{ai})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n}\sum_{j=1}^{k} U_{aj}^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\sum_{j=1}^{k} V_{ij}^2 &(64)\\ &= \sum_{(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n} \|u^{(a)}\|^2 + \frac{\lambda}{2}\sum_{i=1}^{m} \|v^{(i)}\|^2 &(65) \end{aligned}$$

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

$$\sum_{i:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|u^{(a)}\|^2 \tag{66}$$

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

$$\sum_{a:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|v^{(i)}\|^2 \tag{67}$$

where $\{a : (a,i) \in D\}$ is the set of users who have rated movie i. This is again a regression problem without offset. We know how to solve it 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

$$\sum_{i:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|u^{(a)}\|^2 \tag{68}$$

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

$$\sum_{a:(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|v^{(i)}\|^2 \tag{69}$$

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 with initialization 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. However, for computational reasons, fully representing X for realistic problems is infeasible. 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)

$$Y = \begin{bmatrix} 5 & ? & 7 \\ 1 & 2 & ? \end{bmatrix} \tag{70}$$

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

$$UV^T = \begin{bmatrix} 2u^{(1)} & 7u^{(1)} & 8u^{(1)} \\ 2u^{(2)} & 7u^{(2)} & 8u^{(2)} \end{bmatrix} \tag{71}$$

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

$$J_1(u^{(1)}) = \frac{(5 - 2u^{(1)})^2}{2} + \frac{(7 - 8u^{(1)})^2}{2} + \frac{\lambda}{2}(u^{(1)})^2 \tag{72}$$

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

$$\frac{dJ_1(u^{(1)})}{du^{(1)}} = -66 + (68 + \lambda)u^{(1)} = 0 \tag{73}$$

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

$$UV^T = \begin{bmatrix} \frac{66}{69}v^{(1)} & \frac{66}{69}v^{(2)} & \frac{66}{69}v^{(3)} \\ \frac{16}{54}v^{(1)} & \frac{16}{54}v^{(2)} & \frac{16}{54}v^{(3)} \end{bmatrix} \tag{74}$$

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

$$\frac{(5 - \frac{66}{69}v^{(1)})^2}{2} + \frac{(1 - \frac{16}{54}v^{(1)})^2}{2} + \frac{\lambda}{2}(v^{(1)})^2 \tag{75}$$

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.

4.3 Exercises

Exercise (4.1) Suppose we are trying to predict how a user would rate songs. Each song x is represented by a feature vector $\phi(x)$ with coordinates that pertain to acoustic properties of the song. The ratings are binary valued $y \in \{0, 1\}$ ("need earplugs" or "more like this"). Given n already rated songs, we can use regularized linear regression to predict the binary ratings. The training criterion is

$$J(\theta) = \frac{\lambda}{2}\|\theta\|^2 + \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot \phi(x^{(t)}))^2/2 \tag{76}$$

(a) If $\hat{\theta}$ is the optimal setting of the parameters with respect to the above criterion, which of the following conditions must be true:

A ; $\lambda\,\hat{\theta} - \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \hat{\theta} \cdot \phi(x^{(t)}))\phi(x^{(t)}) = 0$

B ; $J(\hat{\theta}) \geq J(\theta)$, for all $\theta \in \mathcal{R}^d$

C ; If we increase λ, the resulting $\|\hat{\theta}\|$ will decrease

D ; If we add features to $\phi(x)$ (whatever they may be), the resulting squared training error will NOT increase

(b) Once we have the estimated parameters $\hat{\theta}$, we must decide how to predict ratings for new songs. Write down an expression that shows which label we should choose for a new song x

(c) What will happen to the predicted ratings if we increase the regularization parameter λ during training?

Exercise (4.2) In this question, we apply low-rank factorization algorithm to the matrix of observations Y:

$$Y = \begin{bmatrix} 5 & ? & 7 \\ ? & 2 & ? \\ ? & 1 & 4 \\ 4 & ? & ? \\ ? & 3 & 6 \end{bmatrix} \tag{77}$$

Assume that $k = 1$ and $\lambda = 1$. After one iteration of the algorithm, we obtain the following values for U and V: $U = [6, 2, 3, 3, 5]^T$, and $V = [4, 1, 5]^T$.

1. evaluate matrix X of predicted ranks.
2. evaluate the squared error and the regularization terms for the current estimate X.
3. evaluate the new estimate of $u^{(1)}$ if Vs are kept fixed.

Exercise (4.3) Suppose we try to estimate a simple rank-1 model $X_{ai} = u_a v_i$ where $u_1, \ldots, u_n$ and $v_1, \ldots, v_m$ are just scalar parameters associated with users u_a and items v_i, respectively. Please select which (if any) of the following 2x2 matrices cannot be reproduced by the rank-1 model.

	v_1	v_2
u_a	+1	+1
u_b	+1	-1

(   )

	v_1	v_2
u_a	-1	+1
u_b	+1	-1

(   )

Exercise (4.4) Consider the case where a movie i has been ranked by a single user. What can you say about the direction of $v^{(i)}$ after one update?

Exercise (4.5) In our description of the matrix factorization algorithm, we initialized the model randomly. Instead, consider an initialization procedure where all the movie vectors, $v^{(1)}, \ldots, v^{(m)}$, are initially the same, i.e., $v^{(i)} = v$, $i = 1, \ldots, m$, for some non-zero v. What can you say about the resulting $u^{(a)}$, $a = 1, \ldots, n$, after the first iteration? What about at convergence?

Exercise (4.6) Assume you are given a procedure for estimating rank-1 matrices on the basis of a partially observed target matrix.

1. Describe an algorithm that utilizes this rank-1 procedure to find a low-rank factorization.

2. What are the advantages of this algorithm over a generic low-rank factorization algorithm described in the text?

Exercise (4.7) Content recommendation (based on existing features) and collaborative filtering (based on collective ratings) rely on complementary sources of information. It should therefore be advantageous to combine them. We would like to implement this hybrid model in the framework of low-rank factorization, expanding the approach to handle existing features for users and movies.

1. We will first represent users and movies using one-hot vectors. In other words, user a will be represented by a vector of length n with 1 in position a and 0 elsewhere. Similarly, each movie i has an associated one-hot vector. Note that these feature vectors simply identify the row and column of the rating matrix. Formulate the standard matrix factorization approach for collaborative filtering in terms of these feature vectors.

2. Now consider a more general case where each movie i is represented by a feature vector $\phi^{(i)}$ and each user a is represented by a feature vector $\psi^{(a)}$. Note that these vectors may consist of one-hot indicator vectors together with other features. Formulate the low-rank factorization algorithm for this case.

5 Non-linear Classifiers – Kernels (old version)

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

$$\phi(x) = [x_1, x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2]^T \tag{78}$$

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

$$y = \text{sign}(\theta \cdot \phi(x) + \theta_0) \tag{79}$$

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

$$\hat{\theta} \cdot \phi(x) + \hat{\theta}_0 = 0 \;\; \Leftrightarrow \;\; \hat{\theta}_1 x_1 + \hat{\theta}_2 x_2 + \hat{\theta}_3\sqrt{2}x_1x_2 + \hat{\theta}_4 x_1^2 + \hat{\theta}_5 x_2^2 + \hat{\theta}_0 = 0 \tag{80}$$

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!)

$$\phi(x) \cdot \phi(x') = (x \cdot x') + (x \cdot x')^2 \tag{81}$$

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 $x_i x_j$, x_i^2, and so on. The resulting predictors

$$\begin{aligned}
\text{Perceptron :} \quad & y = \text{sign}\big(\,\theta \cdot \phi(x) + \theta_0\big) && (82)\\
\text{SVM :} \quad & y = \text{sign}\big(\,\theta \cdot \phi(x) + \theta_0\big) && (83)\\
\text{Linear regression :} \quad & y = \theta \cdot \phi(x) + \theta_0 && (84)
\end{aligned}$$

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.

5.0.1 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

$$\theta = \sum_{i=1}^{n} \alpha_i y^{(i)} \phi(x^{(i)}) \quad (85)$$

$$\theta_0 = \sum_{i=1}^{n} \alpha_i y^{(i)} \quad (86)$$

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

$$\theta \cdot \phi(x) = \sum_{i=1}^{n} \alpha_i y^{(i)} (\phi(x^{(i)}) \cdot \phi(x)) = \sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x) \quad (87)$$

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

$$\theta \cdot \phi(x) + \theta_0 = \sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x) + \sum_{i=1}^{n} \alpha_i y^{(i)} = \sum_{i=1}^{n} \alpha_i y^{(i)} [K(x^{(i)}, x) + 1] \tag{88}$$

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 $\alpha_t \leftarrow \alpha_t + 1$
```

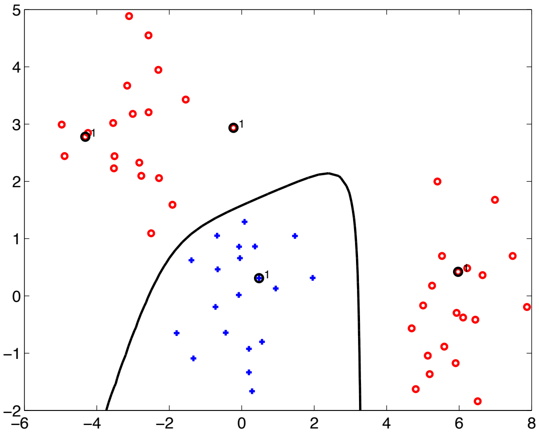


Figure 11: Kernel perceptron example. We have circled the four training examples that the algorithm makes a mistake on.

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 11 shows the decision boundary (compare to Eq.(88) above)

$$\sum_{i=1}^{n} \alpha_i y^{(i)} [K(x^{(i)}, x) + 1] = 0 \tag{89}$$

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

$$K(x, x') = \exp(-\|x - x'\|^2/2) \tag{90}$$

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!

5.0.2 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

$$J(\theta) = \frac{1}{n}\sum_{t=1}^{n}(y^{(t)} - \theta \cdot \phi(x^{(t)}))^2/2 + \frac{\lambda}{2}\|\theta\|^2 \tag{91}$$

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

$$\begin{aligned} \frac{\partial}{\partial\theta}J(\theta) &= -\frac{1}{n}\sum_{t=1}^{n}\underbrace{(y^{(t)} - \theta \cdot \phi(x^{(t)}))}_{=n\lambda\,\alpha_t}\phi(x^{(t)}) + \lambda\theta & (92) \\ &= -\lambda\sum_{t=1}^{n}\alpha_t\phi(x^{(t)}) + \lambda\theta = 0 & (93) \end{aligned}$$

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

$$\begin{aligned} n\lambda\,\alpha_t &= y^{(t)} - \theta \cdot \phi(x^{(t)}) & (94) \\ &= y^{(t)} - \left(\sum_{i=1}^{n}\alpha_i\phi(x^{(i)})\right) \cdot \phi(x^{(t)}) & (95) \\ &= y^{(t)} - \sum_{i=1}^{n}\alpha_i K(x^{(i)}, x^{(t)}) & (96) \end{aligned}$$

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)

$$\sum_{i=1}^{n}\alpha_i K(x^{(i)}, x^{(t)}) = \sum_{i=1}^{n}K(x^{(t)}, x^{(i)})\alpha_i = [K\vec{\alpha}]_t \tag{97}$$

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

$$n\lambda\,\vec{\alpha} = \vec{y} - K\vec{\alpha} \quad \text{or} \quad (n\lambda\,I + K)\vec{\alpha} = \vec{y} \tag{98}$$

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

$$\hat{\theta} \cdot \phi(x) = \sum_{i=1}^{n} \hat{\alpha}_i \phi(x^{(i)}) \cdot \phi(x) = \sum_{i=1}^{n} \hat{\alpha}_i K(x^{(i)}, x) \tag{99}$$

Figure 12 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.

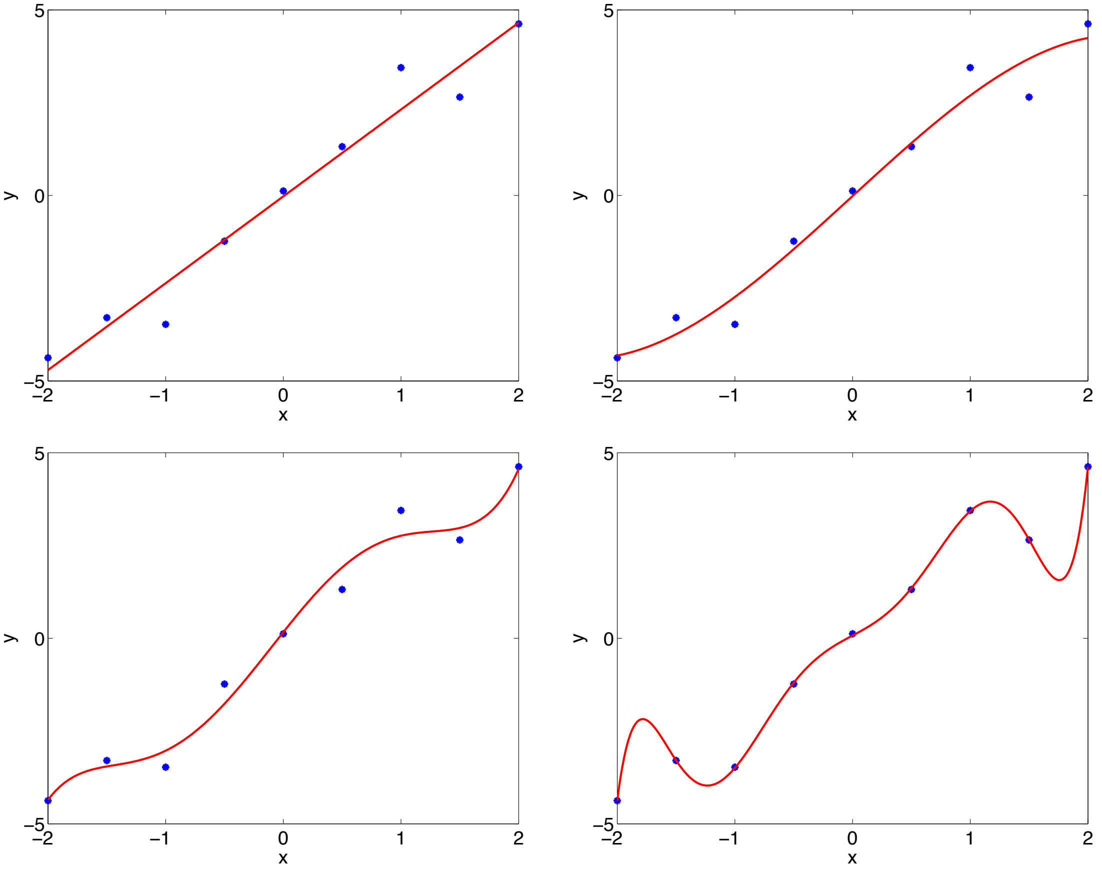


Figure 12: Kernel regression with a linear kernel (top left), 3rd order polynomial kernel (top right), 5th order polynomial kernel (bottow left), and a 7th order polynomial kernel (bottom right).

5.0.3 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 $\tilde{K}(x, x') = f(x)K(x, x')f(x')$

(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)

$$\begin{aligned} K(x, x') &= \begin{bmatrix} \phi^{(1)}(x) \\ \phi^{(2)}(x) \end{bmatrix} \cdot \begin{bmatrix} \phi^{(1)}(x) \\ \phi^{(2)}(x) \end{bmatrix} && (100) \\ &= \phi^{(1)}(x) \cdot \phi^{(1)}(x') + \phi^{(2)}(x) \cdot \phi^{(2)}(x') && (101) \\ &= K_1(x, x') + K_2(x, x') && (102) \end{aligned}$$

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

$$x \cdot x' = x_1 x'_1 + \ldots + x_d x'_d = f_1(x)\, 1\, f_1(x') + \cdots + f_d(x)\, 1\, f_d(x') \qquad (103)$$

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

$$K(x, x') = (x \cdot x') + (x \cdot x')^2 \qquad (104)$$

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,

$$K(x, x') = \exp(x \cdot x') = 1 + (x \cdot x') + \frac{1}{2!}(x \cdot x')^2 + \cdots \qquad (105)$$

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,

$$\begin{aligned} K(x, x') &= \exp(-\|x - x'\|^2/2) && (106) \\ &= \exp(-\|x\|^2/2) \exp(x \cdot x') \exp(-\|x'\|^2/2) && (107) \\ &= f(x) \exp(x \cdot x') f(x') && (108) \end{aligned}$$

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[5]. 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.

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

6 Learning representations – Neural networks

We have so far covered two 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

$$\hat{y} = \text{sign}(\theta \cdot \phi(x)) \tag{109}$$

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

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x)) \tag{110}$$

We can always map the kernel classifier back into the explicit form in Eq.(109) 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.(110) 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

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} \alpha_i y^{(i)} K(x^{(i)}, x)) = \text{sign}(\alpha \cdot \tilde{\phi}(x)) \tag{111}$$

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.

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

6.1 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 13 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.

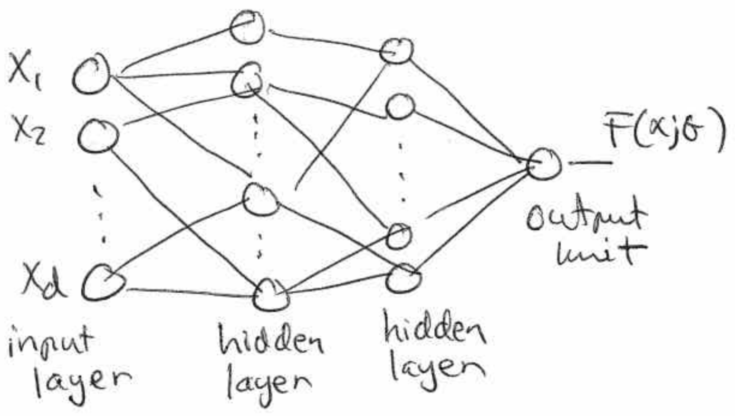


Figure 13: A feed-forward neural network with two hidden layers and a single output unit.

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 14. 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).

$$
\begin{aligned}
z &= \sum_{i=1}^{d} x_i V_i + V_0 = x \cdot V + V_0 \ \text{(weighted summed input to the unit)} \quad (112) \\
F(x;\theta) &= f(z) = z \ \text{(network output)} \quad (113)
\end{aligned}
$$

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.

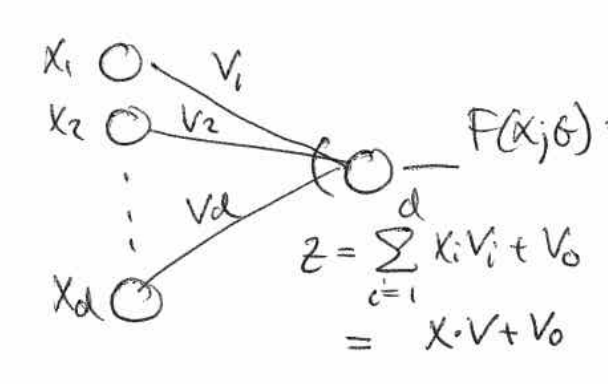


Figure 14: A simple neural network with no hidden units.

Hidden layers, representation

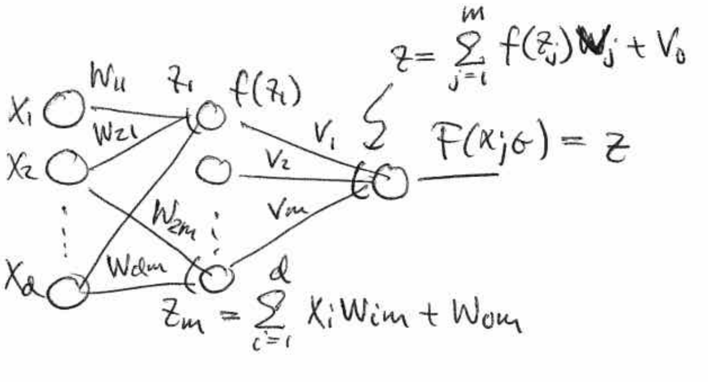


Figure 15: A neural network with one hidden layer.

Let's extend the model a bit and consider a neural network with one hidden layer. This is shown in Figure 15. 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

$$\begin{aligned} z_j &= \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \;\; \text{(weighted input to the } j^{th} \text{ hidden unit)} && (114) \\ f(z_j) &= \max\{0, z_j\} \;\; \text{(rectified linear activation function)} && (115) \end{aligned}$$

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

$$\begin{aligned} z &= \sum_{j=1}^{m} f(z_j) V_j + V_0 \;\; \text{(weighted summed input to the unit)} && (116) \\ F(x;\theta) &= z \;\; \text{(network output)} && (117) \end{aligned}$$

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 16. 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

$$\begin{aligned} z_1 &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{11} \\ W_{21} \end{bmatrix} + W_{01} && (118) \\ f(z_1) &= \max\{0, z_1\} \;\; \text{(activation of the 1st hidden unit)} && (119) \\ z_2 &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \cdot \begin{bmatrix} W_{12} \\ W_{22} \end{bmatrix} + W_{02} && (120) \\ f(z_2) &= \max\{0, z_2\} \;\; \text{(activation of the 2nd hidden unit)} && (121) \end{aligned}$$

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 16. 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.

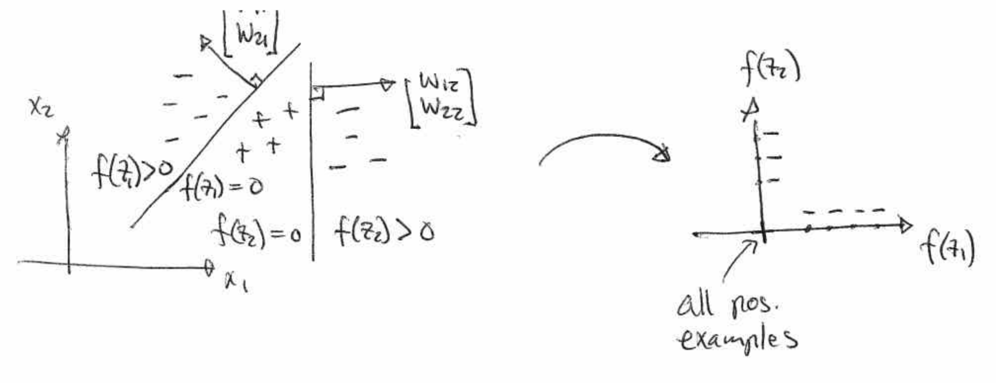


Figure 16: Hidden unit activations and examples represented in hidden unit coordinates

6.2 Learning, back-propagation

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,

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{122}$$

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

$$\text{(0)}\ \ \text{Initialize } \theta \text{ to small random values} \tag{123}$$

$$\text{(1)}\ \ \text{Select } i \in \{1, \ldots, n\} \text{ at random or in a random order} \tag{124}$$

$$\text{(2)}\ \ \theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) \tag{125}$$

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

$$\nabla_\theta \text{Loss}(y^{(i)} F(x^{(i)}; \theta)) = [\frac{\partial}{\partial \theta_1} \text{Loss}(y^{(i)} F(x^{(i)}; \theta)), \ldots, \frac{\partial}{\partial \theta_D} \text{Loss}(y^{(i)} F(x^{(i)}; \theta))]^T \tag{126}$$

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 17. 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.

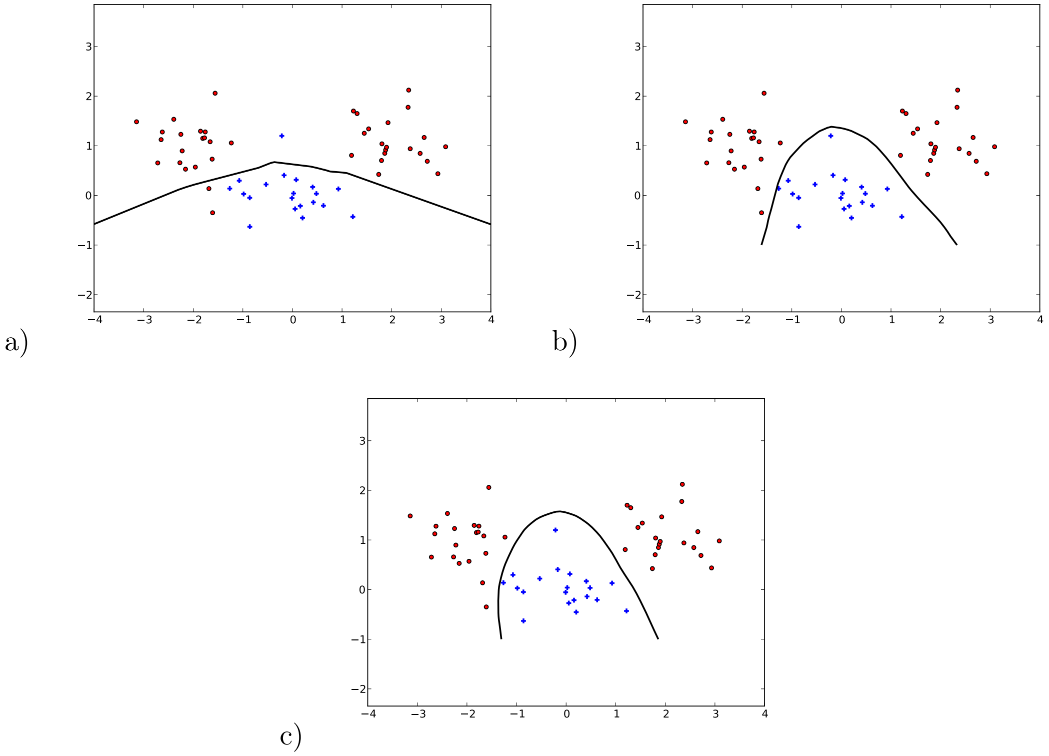


Figure 17: Decision boundaries resulting from training neural network models with one hidden layer and varying number of hidden units: a) 10; b) 100; and c) 500.

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

$$\theta = [V_1, \ldots, V_d, V_0]^T \tag{127}$$

$$\theta = [W_{11}, \ldots, W_{1m}, \ldots, W_{d1}, \ldots, W_{dm}, W_{01}, \ldots, W_{0m}, V_1, \ldots, V_d, V_0]^T \tag{128}$$

respectively, as illustrated in Figures 18a) and b) below.

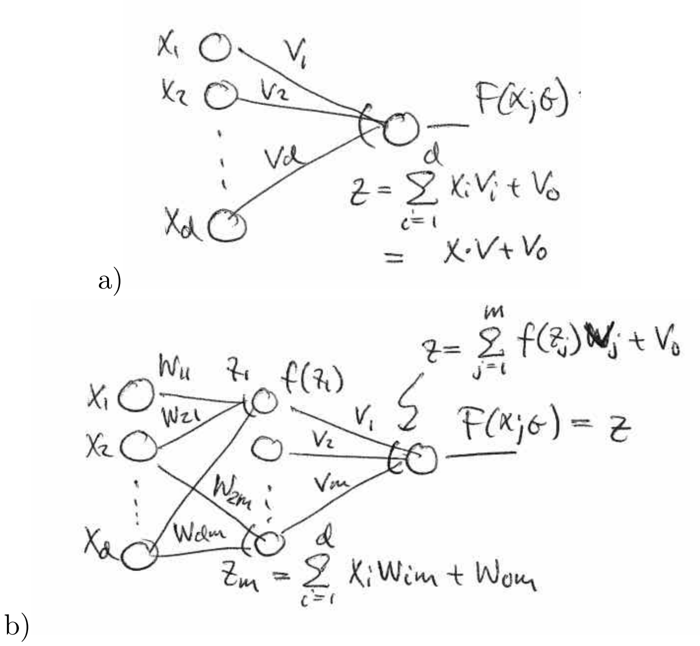

a)
X1
X2
Xd
V1
V2
Vd
F(x;θ)
z = Σ_{i=1}^{d} x_i V_i + V_0
= x·V + V_0
b)
z = Σ_{j=1}^{m} f(z_j) W_j + V_0
F(x;θ) = z
W11
W21
z1
f(z1)
X1
X2
Xd
V1
V2
Vm
W2m
Wdm
z_m = Σ_{i=1}^{d} x_i W_im + W_0m

Figure 18: 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.(122). 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

$$\theta \leftarrow \theta - \eta_k \nabla_\theta \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big) \tag{129}$$

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

$$\theta_i \leftarrow \theta_i - \eta_k \frac{\partial}{\partial \theta_i} \text{Loss}\big(y^{(t)} F(x^{(t)}; \theta)\big), \quad i = 1, \ldots, D \tag{130}$$

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 18a) 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

$$\frac{\partial}{\partial V_i}\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) \;=\; \frac{\partial}{\partial V_i}\text{Loss}\big(y^{(t)}z^{(t)}\big) \tag{131}$$

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

$$\begin{aligned}
\frac{\partial}{\partial V_i}\text{Loss}\big(y^{(t)}z^{(t)}\big) &\overset{\text{chain rule}}{=} \left[\frac{\partial z^{(t)}}{\partial V_i}\right]\left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big)\right] && (132)\\
&= \left[\frac{\partial(\sum_{j=1}^{d} x_j^{(t)} V_j + V_0)}{\partial V_i}\right]\left[\frac{\partial}{\partial z^{(t)}}\text{Loss}\big(y^{(t)}z^{(t)}\big)\right] && (133)\\
&= \left[x_i^{(t)}\right]\left[\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)}z^{(t)}\big) > 0 \\ \text{and zero otherwise}\end{array}\right] && (134)
\end{aligned}$$

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*[6] 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

$$\begin{aligned}
V_i &\leftarrow V_i + \eta_k y^{(t)} x_i^{(t)}, \;\; i = 1, \ldots, d && (135)\\
V_0 &\leftarrow V_0 + \eta_k y^{(t)} && (136)
\end{aligned}$$

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 19 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 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

[6] 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.

step-size based on the gradients (AdaGrad):

$$\begin{aligned} g_i &\leftarrow \frac{\partial}{\partial V_i}\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) \;\; \text{(gradient or sub-gradient)} && (137) \\ G_i &\leftarrow G_i + g_i^2 \;\; \text{(cumulative squared gradient)} && (138) \\ V_i &\leftarrow V_i - \frac{\eta}{\sqrt{G_i}} g_i \;\; \text{(adaptive gradient update)} && (139) \end{aligned}$$

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.

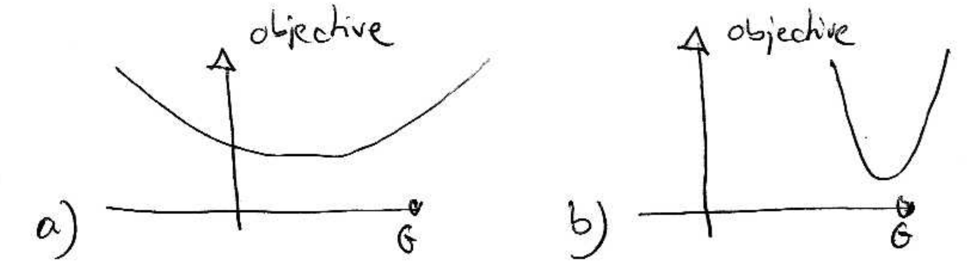


Figure 19: Objective functions that vary a) little b) a lot as a function of the parameter.

Stochastic gradient descent for two layer networks

Let us now consider the two layer network in Figure 18b). Recall that the network output is now obtained in stages, activating the hidden units, before evaluating the output. In other words,

$$\begin{aligned} z_j &= \sum_{i=1}^{d} x_i W_{ij} + W_{0j} \;\; \text{(input to the } j^{th} \text{ hidden unit)} && (140) \\ f(z_j) &= \max\{0, z_j\} \;\; \text{(output of the } j^{th} \text{ hidden unit)} && (141) \\ z &= \sum_{j=1}^{m} f(z_j) V_j + V_0 \;\; \text{(input to the last unit)} && (142) \\ F(x;\theta) &= z \;\; \text{(network output)} && (143) \end{aligned}$$

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,

$$V_j \leftarrow V_j + \eta_k y^{(t)} f(z_j^{(t)}), \;\; j = 1, \ldots, m \tag{144}$$

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

$$\begin{aligned}
\frac{\partial}{\partial W_{ij}} \text{Loss}\big(y^{(t)} z^{(t)}\big) &= \left[\frac{\partial z_j^{(t)}}{\partial W_{ij}}\right] \left[\frac{\partial f(z_j^{(t)})}{\partial z_j^{(t)}}\right] \left[\frac{\partial z^{(t)}}{\partial f(z_j^{(t)})}\right] \left[\frac{\partial}{\partial z^{(t)}} \text{Loss}\big(y^{(t)} z^{(t)}\big)\right] && (145) \\
&= [x_i] \, [\![\, z_j^{(t)} > 0 \,]\!] \, [V_j] \left[\begin{array}{l} -y^{(t)} \text{ if } \text{Loss}\big(y^{(t)} z^{(t)}\big) > 0 \\ \text{and zero otherwise} \end{array} \right] && (146)
\end{aligned}$$

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., $\text{Loss}\big(y^{(t)} z^{(t)}\big) > 0$,

$$W_{ij} \leftarrow W_{ij} + \eta_k \, x_i^{(t)} [\![\, z_j^{(t)} > 0 \,]\!] V_j y^{(t)}, \;\; i = 1, \ldots, d, \; j = 1, \ldots, m \tag{147}$$

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

$$\frac{1}{n}\sum_{t=1}^{n} \text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2 \;=\; \frac{1}{n}\sum_{t=1}^{n} \overbrace{\left[\text{Loss}\big(y^{(t)}F(x^{(t)};\theta)\big) + \frac{\lambda}{2}\|\theta\|^2\right]}^{\text{modified loss}} \tag{148}$$

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

$$V_j \leftarrow V_j + \eta_k\Bigg(-\lambda V_j + y^{(t)} f(z_j^{(t)})\Bigg),\;\; j = 1, \ldots, m \tag{149}$$

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

$$V_j \leftarrow V_j + \eta_k\Bigg(-\lambda V_j + 0\Bigg),\;\; j = 1, \ldots, m \tag{150}$$

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).

6.3 Multi-way classification with Neural Networks

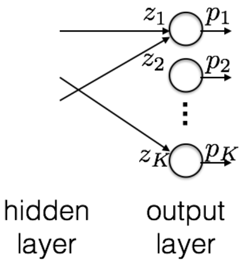
Many classification problems involve more than two possible class labels. For example, we might wish to assign images to one of 1000 different categories based on the object that is primarily displayed in the image. Suppose then that we have K possible class labels, and the goal is to predict $y \in \{1, \ldots, K\}$ for each example x. Broadly speaking, there are two ways to solve such multi-way classification problems. One possible approach is to first decompose the multi-way classification problem into many binary classification tasks (known as output codes). For example, we could train a binary classifier to predict whether x should be assigned to class 1 or not, and similarly for all the other labels. The problem is that the resulting binary classifiers may be mutually inconsistent. Indeed, classifier 1 might strongly prefer label 1 while another classifier supports a different label. The predicted multi-way label must be then obtained by reconciling individual predictions. There are many ways to break problems into binary tasks (one vs all, all pairs, etc.) and many ways to combine the resulting binary predictions.

We can also solve the multi-way classification problem directly by introducing not one but K output units. Each output unit then corresponds to a specific class label but shares the feature representation of the example as captured by the preceding layer. A natural way to resolve competing predictions from the output units is to map them to probabilities. In other words, our goal is not only to predict which label is the most likely but also how certain we are in classifying it as such. We need to therefore map net inputs $z_1, \ldots, z_K$ received by the output units to probability values $p_1, \ldots, p_K$ where $p_j \in [0, 1]$. Note that it is important to also ensure that $\sum_{j=1}^{K} p_j = 1$ since x should be assigned to only one label (we are just uncertain which one it is). If we didn't enforce the normalization, we could easily say that x has label 1 with probability 1 and label 2 with probability 1. What would this mean? It means that the labels are not exclusive but rather non-exclusive properties that we predict for examples (multi-label classification). Our focus here is on multi-way classification where there is exactly one correct label for each example.

The most common way to map real numbers (net inputs) to probabilities is via the softmax function

$$p_j = \frac{\exp(z_j)}{\sum_{l=1}^{K} \exp(z_l)}, \quad j = 1, \ldots, K \tag{151}$$

Clearly, $p_j \in [0,1]$ and $\sum_{j=1}^{K} p_j = 1$. The larger the net input z_j is, the greater the corresponding probability p_j.



Now that our predictions are probabilities rather than specific labels, we need to define a new loss function for training. The goal overall is clearly to maximize the probability that we predict the correct label for each training example. Let's say training example x has label y. We would then wish to maximize p_y where $p_1, \ldots, p_K$ are obtained by passing the example x through the feed-forward network, calculating the net inputs to the output units, and passing the resulting values through the softmax function. The probability that we predict all the training labels correctly is just a product of p_y across the training examples. If we take a logarithm of this product, we get a sum of log-probabilities $\log p_y$. This additivity is much nicer for optimization. Finally, if we maximize $\log p_y$, we can just as well minimize $-\log p_y$. As a result, we should define our loss (log-loss) as

$$\text{Loss}(y, p) = -\log p_y = -\left(z_y - \log \sum_{l=1}^{K} \exp(z_l) \right) \tag{152}$$

Note that p_y implicitly depends on all the net inputs $z_1, \ldots, z_K$ due to normalization. Indeed, for back-propagation we get

$$\delta_j = \frac{\partial}{\partial z_j} \text{Loss}(y, p) = -\big([\![\, j = y \,]\!] - p_j \big), \quad j = 1, \ldots, K \tag{153}$$

which is just the difference between the "target distribution" $[\![\, j = y \,]\!]$, $j = 1, \ldots, K$, that places all the probability mass on the correct label, and our predicted distribution p_j, $j = 1, \ldots, K$.

Now that we know how to assess the loss on each training example and how to back-propagate errors, we can train the multi-way classification model analogously to binary classification.

6.4 Convolutional Neural Networks

We will discuss here a new neural network architecture that is specifically tailored to images. As a starting point, let's try to solve an image classification problem with a simple feed-forward architecture that we already know about and see what types of problems we run into. If we view a digital image as a matrix of pixels, we can vectorize it, and feed it "as is" to a feed-forward neural network. Thus we designate an input unit to hold the value of each pixel in the image. If the image has millions of pixels, and we use a few hundred hidden units, then the first layer in the feed-forward architecture already has over 100 million parameters. While the number is large, and it seems that we will need lots of labeled images to learn the associated parameters, the main problem is actually in what these weights are supposed to do. For example, it is often useful to first identify small features in the image such as edges, and later combine such features to detect objects etc. But in this architecture, we would have to separately learn the weights for recognizing the same simple feature in each possible location in the image (nothing carries over from one part of the image to another). Another issue is accounting for scale. Features are sometimes small, sometimes large, as are objects in the image. How do we vary the resolution at which we look at the image? Moreover, the objects should eventually be classified correctly regardless of where they appear in the image, i.e., our classifier should be (at least partly) translation invariant. Do we have to separately learn to recognize the same object in each position in the image? The straightforward feed-forward architecture doesn't seem well-suited for this task.

A convolutional neural network (CNN for short) is a feed-forward neural network but it has quite a bit of special structure. It consists of interspersed layers of convolution and pooling operations (which we will outline in more detail below). The convolution layer applies simple local image "filters" across the image, producing new images (known as feature maps) where the "pixels" in the feature map represent how much the simple features were present in corresponding location. You can think of this process as first breaking the image into overlapping little image patches, and applying a linear classifier to each patch. The size of the patches, and how much they overlap, can vary. The pooling layer, on the other hand, abstracts away from the location where the features are, only storing the maximum activation within each local area, capturing "what" is there rather than "where" it is.

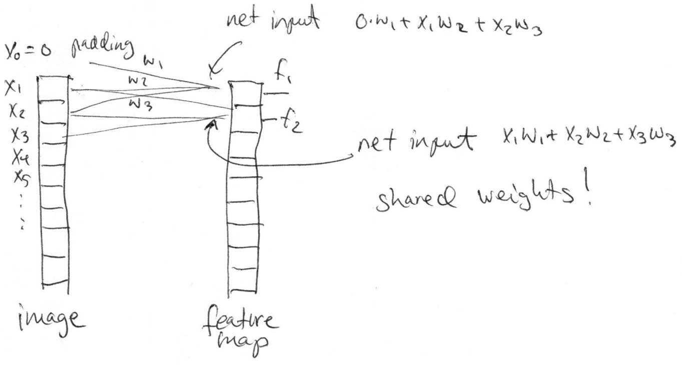
6.4.1 Convolution

To make the ideas a little more concrete, let's assume that our image is 1-dimensional with n pixels, $x_1, \ldots, x_n$. The filter applied to the image could be just $[w_1, w_2, \ldots, w_k]^T$, i.e., of size k (an odd number). Of course, the filter size can vary but we will use $k = 3$ for simplicity. Figure below illustrates how the filter is applied to obtain the corresponding feature map (another 1-dimensional image). For example, if stride is set to one, filter size is k, and the added non-linearity is ReLU, then the i^{th} coordinate value of the feature map is obtained as

$$f_i = \max\left\{0,\ \sum_{j=1}^{k} x_{i+j-k/2-1}\, w_j\right\} \tag{154}$$

where $i = 1, \ldots, n$ (the same size as the original image). We assume that the image is padded with zeros when the index $i + j - k/2 - 1$ extends beyond the limits. For example,

$x_{-1} = 0$. Note that if we increase the stride beyond one, the resulting feature map will be of size n/stride.



The feature map is parameterized just as any feed-forward layer in a neural network. However, if viewed as a feed-forward layer, the weights in this layer are *shared* in the sense that one unit in the feature map layer uses the exact same weights to calculate its net input as any other unit in that layer. Thus the only parameters that need to be estimated are $[w_1, w_2, \ldots, w_k]^T$, i.e., the parameters of the filter. This makes the mapping very compact with number of parameters that is independent of the image size (cf. straightforward feed-forward layer). Nevertheless, the filter can be adjusted to look for different features and the resulting feature map identifies where the chosen feature is active in the image. Of course, we don't just wish to look for a single feature (e.g., horizontal edge). Instead, we typically introduce a number of different feature maps (also called channels) and the parameters in the associated filters are estimated jointly in the overall CNN model. The feature maps may also be introduced across channels.

6.4.2 Pooling

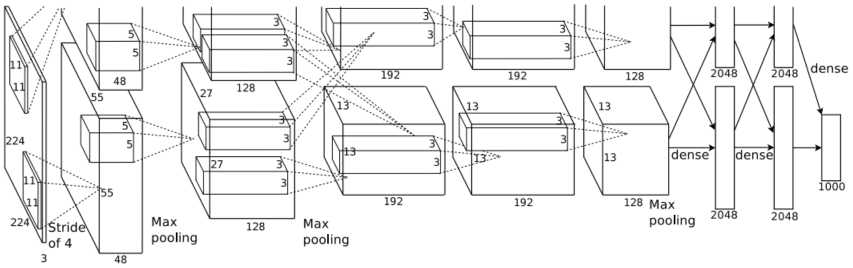
Another key operation in CNNs is pooling. There are many different types of pooling operations but the simplest one is just max-pooling. This is similar to convolution in terms of the filter size, and how it is applied across the image. However, there are no parameters, and the weighted average that the convolution produces is just replaced by the maximum. In other words, if the pooling filter size is k with stride one, then

$$f_i = \max_{j \in \{1,\ldots,k\}} x_{i+j-k/2-1} \tag{155}$$

for $i = 1, \ldots, n$. Max-pooling is typically applied with a relatively small k (can be an even number) but with larger stride (e.g., 2) so as to reduce the resulting feature map size.

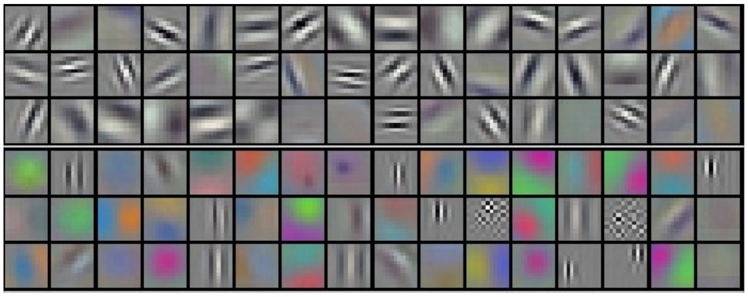
6.4.3 CNN architecture

There are a large number of possible CNN architectures that combine convolution and pooling operations (among other things). For example, the architecture that won the 2012 imageNet image classification challenge (1.2 million images, 1000 categories), begins by mapping the image to 96 feature maps using 11x11x3 convolution filters (size 11x11 filters over the image locations plus color) with ReLU non-linearity and stride equal to 4. This is followed by contrast normalization and 2x2 max pooling. Further layers operate similarly. Towards the output layer, the model introduces fully connected layers that combine the information from all the channels. The final output layer is a softmax layer with 1000 output units, one unit per category. The model is trained end-to-end, i.e., as a differentiable map from the image all the way to the output units. A simple schematic is shown below.



(Krizhevsky et al. 2012)

We can visualize what the resulting filters look like in a trained model. Figure below shows the first 96 filters 11x11x3 as color images (color image patches that would maximize the filter response).



(Krizhevsky et al. 2012)

More details about the model and tricks used for training it (data augmentation, dropout) can be found in Krizhevsky et al. (2012).

6.5 Recurrent Neural networks

Our goal here is to model sequences such as natural language sentences using neural networks. We will try to first adapt feed-forward networks for this purpose by incorporating one, two, or more previous words as inputs, aiming to always to predict the next word in the sequence. We will then generalize the simple feedforward model to a recurrent neural network (RNN) which has greater flexibility of storing key aspects of the sequence seen so far. RNNs maintain a notion of “state” (continuous vector) that evolves along the sequence and serves as the memory of what has been seen so far. Next word is then predicted relative to the information stored in the state. Basic RNNs suffer from vanishing/exploding gradient problems but can be considerably improved by introducing “gating”. Gates are simple networks that are used to control the flow of information into the state (memory) as well as what is read from the state for the purpose of predictions (e.g., next symbol in the sequence). Most modern neural network sequence models are variants of Long-Short-Term-Memory (LSTM), introduced already two decades ago. We will only consider simple versions of these models, pruning away details when possible for clarity of exposition. Such variants do perform similarly in practice, however.

As a starting point, consider a simple English sentence

$$\begin{array}{cccccccc} y_1 & y_2 & y_3 & y_4 & y_5 & y_6 & y_7 & y_8 \\ \text{midterm} & \text{will} & \text{have} & \text{a} & \text{neural} & \text{networks} & \text{question} & \text{<end>} \end{array} \tag{156}$$

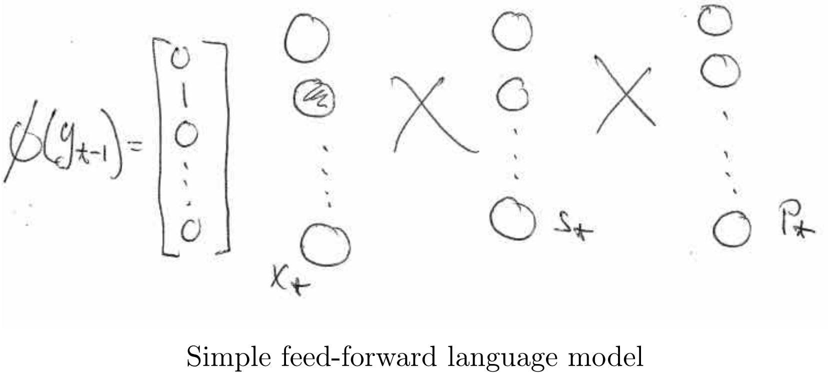
where we will represent words by discrete variables $y_i \in \{1, \ldots, V\}$ where V is the size of the vocabulary we consider plus one (the <end> symbol). Note that in order to predict a sentence we also need to determine when it ends. For this reason, we always have the <end> symbol appearing at the end of each sentence. All of our models predict sentences by generating words in a sequence until they predict <end> as the next symbol. Formally, we will specify a probability distribution over the sentences, unfolded sequentially as in

$$P(y_1)P(y_2|y_1)P(y_3|y_2, y_1) \cdots P(y_{n-1}|y_{n-2}, \ldots, y_1)P(y_n|y_{n-1}, \ldots, y_1) \tag{157}$$

where y_n is necessarily <end> if $y_1, \ldots, y_n$ represents a complete sentence. The distribution involves parameters that come from the neural network that is used to realize the individual probability terms. For example, we could use a simple feed-forward neural network model that takes in just the previous word and predicts a probability distribution over the next word. As such, it will not be able to capture dependences beyond the previous symbol. In this model, $P(y_3|y_2, y_1)$ could only be represented as $P(y_3|y_2)$ (dependence on y_1 is lost). Such a model would then assign probability

$$P(y_1)P(y_2|y_1)P(y_3|y_2) \cdots P(y_{n-1}|y_{n-2})P(y_n|y_{n-1}) \tag{158}$$

to the sentence $y_1, \ldots, y_n$. Figure below illustrates a possible feed-forward neural network for modeling $P(y_t|y_{t-1})$. The same model is applied for all $t = 1, \ldots, n$ where the symbol y_0 can be taken to be <end> (end of previous sentence). We first map index y_{t-1} to a one-hot vector $\phi(y_{t-1})$ of dimension V that can be used as an input vector x_t. Note that, in this representation, only one of the input units is on (one) while all the rest are zero. With this input, we obtain hidden unit activations s_t (vector of dimension m), and finally softmax output probabilities p_t (a vector of dimension V). Thus the kth component of p_t represents $P(y_t = k|y_{t-1})$.



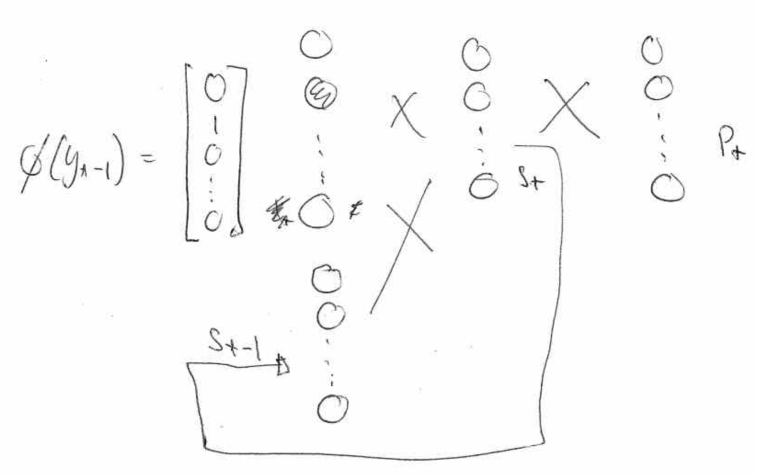
Simple feed-forward language model

Algebraically, our model corresponds to equations

$$x_t = \phi(y_{t-1}), \;\; s_t = \tanh(W^{s,x} x_t), \;\; p_t = \text{softmax}(W^o s_t) \tag{159}$$

where, for clarity, we have omitted any offset parameters, and the equations are "vectorized" in the sense that, e.g., tanh is applied element-wise to a vector of inputs. Here $W^{s,x}$ is a matrix of dimension $m \times V$ and W^o is a $V \times m$ matrix.

The problem with the simple feed-forward model is that the prediction about what comes next is made only on the basis of the previous word. We can, of course, expand the model to use the previous two or three words as inputs rather than just one. However, even though the resulting model clearly becomes more powerful with any additional input added, it still lacks the ability to memorize an important cue early on in the sentence and store it for use later on. We need a way to carry the information we knew before forward as well. To address this, we will feed the previous hidden state vector s_{t-1} (summary prior to seeing y_{t-1}) as an input to the model along with $\phi(y_{t-1})$ as in the feed-forward model. The idea of recurrent neural networks is precisely to incorporate previous "state" as an input.



Simple recurrent neural network language model

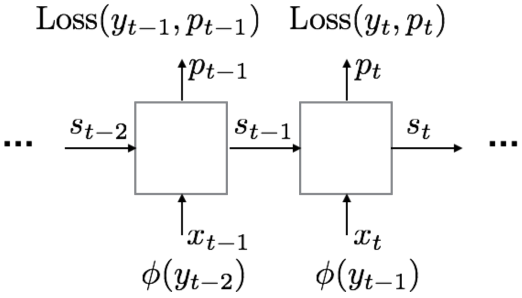
Algebraically, the simple RNN model is characterized by

$$x_t = \phi(y_{t-1}), \;\; s_t = \tanh(W^{s,x} x_t + W^{s,s} s_{t-1}), \;\; p_t = \text{softmax}(W^o s_t) \tag{160}$$

where we simply added the previous state s_{t-1} as an additional input to the state update. The associated new parameters are represented by an $m \times m$ matrix $W^{s,s}$. Such a recurrent neural network model has the ability to generate quite complex behaviors. Indeed, note that since s_{t-1} was calculated when y_{t-2} was fed as an input, the distribution p_t must depend on y_{t-2} as well. Going further, s_{t-1} also depends on s_{t-2} which was determined in part by y_{t-3}, and so on. As a result, the distribution over the possible next words p_t is affected by the whole history $y_{t-1}, \ldots, y_1$. Of course, the way this dependence is manifested results from the specific parameterization in the update equations. The recurrent model does not have enough parameters to specify any arbitrary $P(y_t|y_{t-1}, \ldots, y_1)$ (this would require $(V-1)V^{t-1}$ parameters) unless we increase m quite a bit. But it nevertheless gives us a parametric way to leverage the whole sequence seen so far in order to predict the next word.

6.5.1 Back-propagation through time

Training a recurrent neural network model doesn't differ much from training a simple feed-forward model. We can back-propagate the error signals measured at the outputs, and update parameters according to the resulting gradients. It is often helpful to unfold the RNN in time so that it looks more like a feed-forward structure. For example, we can represent each application of the update equations as a block, taking x_t, s_{t-1} as inputs and producing s_t, p_t. In other words, we feed $\phi(y_{t-1})$ as an input x_t together with the previous state s_{t-1}, calculate the updated state s_t and the distribution p_t over possible words y_t. Once y_t is revealed, we can measure $\text{Loss}(y_t, p_t)$, and proceed a step forward. The overall goal is to minimize the sum of losses incurred as the model is unfolded along the sequence. Note that the weights are shared across the blocks, i.e., each block uses exactly the same parameters $W^{s,x}$, $W^{s,s}$, W^o.



Unfolded block view of the simple RNN

Suppose we have already evaluated

$$\delta_j^{s_t} = \frac{\partial}{\partial s_{t;j}} \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l), \quad j = 1, \ldots, m \tag{161}$$

i.e., the impact of state s_t on the future losses (after t). Here j indexes the m units that are used to represent the state vector. Note that $\delta_j^{s_n}$ is simply zero as there are no future losses from that point on. We wish to calculate $\delta_i^{s_{t-1}}$, $i = 1, \ldots, m$, i.e., go one block backwards. Note that in our simple RNN, s_{t-1} affects $\text{Loss}(y_t, p_t)$ as well as the future losses only via the next state s_t. As a result,

$$\begin{aligned}
\delta_i^{s_{t-1}} &= \frac{\partial}{\partial s_{t-1;i}} \left[\text{Loss}(y_t, p_t) + \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (162) \\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \frac{\partial}{\partial s_{t;j}} \left[\text{Loss}(y_t, p_t) + \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (163) \\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \frac{\partial}{\partial s_{t;j}} \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (164) \\
&= \sum_{j=1}^{m} \frac{\partial s_{t;j}}{\partial s_{t-1;i}} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \delta_j^{s_t} \right] && (165) \\
& && (166)
\end{aligned}$$

where, for example,

$$\begin{aligned}
\frac{\partial s_{t;j}}{\partial s_{t-1;i}} &= \frac{\partial}{\partial s_{t-1;i}} \tanh([W^{s,x} x_t]_j + [W^{s,s} s_{t-1}]_j) && (167)\\
&= (1 - s_{t;j}^2) \frac{\partial}{\partial s_{t-1;i}} [W^{s,s} s_{t-1}]_j && (168)\\
&= (1 - s_{t;j}^2) W_{ji}^{s,s} && (169)\\
\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) &= -\sum_{k=1}^{V} \left([\![\, k = y_t \,]\!] - p_{t;k} \right) W_{k,j}^{o} && (170)
\end{aligned}$$

Now, suppose we have calculated all $\delta_j^{s_t}$, $t = 1, \ldots, n$, $j = 1, \ldots, m$. We would like to use these to evaluate the derivatives with respect to the parameters that are shared across the unfolded model. For example, $W_{ji}^{s,s}$ is used in all the state updates and therefore it will have an impact on the losses through $s_1, \ldots, s_n$. We will decompose this effect into a sum of immediate effects, i.e., how $W_{ji}^{s,s}$ impacts s_t (and losses there on) when s_{t-1} is assumed to be constant. So, when we write $\partial s_{t;k} / \partial W_{ji}^{s,s}$, we mean this immediate impact (as if the parameter was specific to the t^{th} box, just set equal to $W_{ji}^{s,s}$). Accordingly,

$$\begin{aligned}
\frac{\partial}{\partial W_{ji}^{s,s}} \left[\sum_{l=1}^{n} \text{Loss}(p_l, s_l) \right] &= \sum_{t=1}^{n} \sum_{k=1}^{m} \frac{\partial s_{t;k}}{\partial W_{ji}^{s,s}} \frac{\partial}{\partial s_{t;k}} \left[\sum_{l=1}^{n} \text{Loss}(y_l, p_l) \right] && (171)\\
&= \sum_{t=1}^{n} \sum_{k=1}^{m} \frac{\partial s_{t:k}}{\partial W_{ji}^{s,s}} \left[\frac{\partial}{\partial s_{t;k}} \text{Loss}(y_t, p_t) + \frac{\partial}{\partial s_{t;k}} \sum_{l=t+1}^{n} \text{Loss}(y_l, p_l) \right] && (172)\\
&= \sum_{t=1}^{n} \sum_{k=1}^{m} \frac{\partial s_{t:k}}{\partial W_{ji}^{s,s}} \left[\frac{\partial}{\partial s_{t;k}} \text{Loss}(y_t, p_t) + \delta_k^{s_t} \right] && (173)\\
&= \sum_{t=1}^{n} \frac{\partial s_{t:j}}{\partial W_{ji}^{s,s}} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \delta_j^{s_t} \right] && (174)\\
&= \sum_{t=1}^{n} (1 - s_{t:j}^2) s_{t-1;i} \left[\frac{\partial}{\partial s_{t;j}} \text{Loss}(y_t, p_t) + \delta_j^{s_t} \right] && (175)
\end{aligned}$$

The problem with learning RNNs is not the equations but rather how the gradients tend to vanish/explode as the number of time steps increases. You can see this by examining how $\delta^{s_{t-1}}$ is obtained on the basis of δ^{s_t}. It involves multiplication with the weights as well as the derivative of the activation function (tanh). Performing a series of such multiplications can easily escalate or rapidly vanish. One way to remedy this problem is to change the architecture a bit.

6.5.2 RNNs with gates

We would like to make the state update more "additive" than multiplicative. As a result, the gradients would also perform better over time. One way to achieve this is to control the flow of information into the state by a gating network. A gate is a simple network that determines how much each coordinate of state s_t is updated. Specifically, we introduce

$$g_t = \text{sigmoid}(W^{g,x}x_t + W^{g,s}s_{t-1}) \tag{176}$$

where the activation function is again applied element-wise, and the gate vector g_t has the same dimension as the state, i.e., $m \times 1$. Recall that $\text{sigmoid}(z) = (1 + \exp(-z))^{-1}$ and therefore all of its coordinates lie in $[0, 1]$. We combine gate and state updates as follows

$$\begin{aligned}
x_t &= \phi(y_{t-1}), & (177)\\
g_t &= \text{sigmoid}(W^{g,x}x_t + W^{g,s}s_{t-1}) & (178)\\
s_t &= (1 - g_t) \odot s_{t-1} + g_t \odot \tanh(W^{s,x}x_t + W^{s,s}s_{t-1}) & (179)\\
p_t &= \text{softmax}(W^o s_t) & (180)
\end{aligned}$$

where $\odot$ refers to element-wise product of vector coordinates. If some of the coordinates of g_t are close to zero, s_t retains the same value as s_{t-1} for those coordinates. As a result, the model can easily carry information from the beginning of the sentence towards the end, and this property is directly controlled by the gate. Models of this kind are much easier to learn than simple RNNs and are therefore preferred. LSTMs, General Recurrent Units (GRUs), and other variants involve further gating to better control the flow of information in and out of the state.

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

$$\mathcal{E}_n(h) = \frac{1}{n}\sum_{t=1}^{n} [\![\, y^{(t)}h(x^{(t)}) \leq 0 \,]\!] \tag{181}$$

The test or generalization error is defined as the expected value

$$\mathcal{E}(h) = E_{(x,y)\sim P^*} [\![\, yh(x) \leq 0 \,]\!] \tag{182}$$

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, $\hat{h} \in \text{argmin}_{h \in H} \mathcal{E}_n(h)$, 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,

$$H = \left\{ h : \ s.t.\ h(x) = \text{sign}(\phi(x) \cdot \theta + \theta_0), \text{ for some } \ \theta \in \mathcal{R}^p, \ \theta_0 \in \mathcal{R} \right\}, \tag{183}$$

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

$$H_1 : \ \phi(x) = [x_1, x_2]^T \tag{184}$$

$$H_2 : \ \phi(x) = [x_1, x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2]^T \tag{185}$$

$$\ldots$$

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,

$$\log \delta \leq \log |\mathcal{H}| + n \log(1-\epsilon) \leq \log |\mathcal{H}| - n\epsilon \tag{186}$$

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

$$\epsilon \leq \frac{\log |\mathcal{H}| + \log(1/\delta)}{n} \tag{187}$$

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,

$$\mathcal{E}(\hat{h}) \leq \mathcal{E}_n(\hat{h}) + \sqrt{\frac{\log |\mathcal{H}| + \log(2/\delta)}{2n}} \tag{188}$$

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.

$$\begin{array}{rcccc} & x^{(1)} & x^{(2)} & \cdots & x^{(n)} \\ h \in \mathcal{H}: & +1 & -1 & \cdots & -1 \\ h' \in \mathcal{H}: & +1 & -1 & \cdots & -1 \\ h'' \in \mathcal{H}: & +1 & +1 & \cdots & -1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{array} \tag{189}$$

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:

$$N_{\mathcal{H}}(n) = \max_{x^{(1)},\ldots,x^{(n)}} N_{\mathcal{H}}(x^{(1)}, \ldots, x^{(n)}) \tag{190}$$

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,

$$\mathcal{E}(\hat{h}) \leq \mathcal{E}_n(\hat{h}) + \sqrt{\frac{\log N_{\mathcal{H}}(2n) + \log(4/\delta)}{n}}, \text{ for all } \hat{h} \in \mathcal{H} \tag{191}$$

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 $n > d_{\mathcal{H}}$,

$$\log N_{\mathcal{H}}(2n) \leq d_{\mathcal{H}}(\log(2n/d_{\mathcal{H}}) + 1) \tag{192}$$

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.

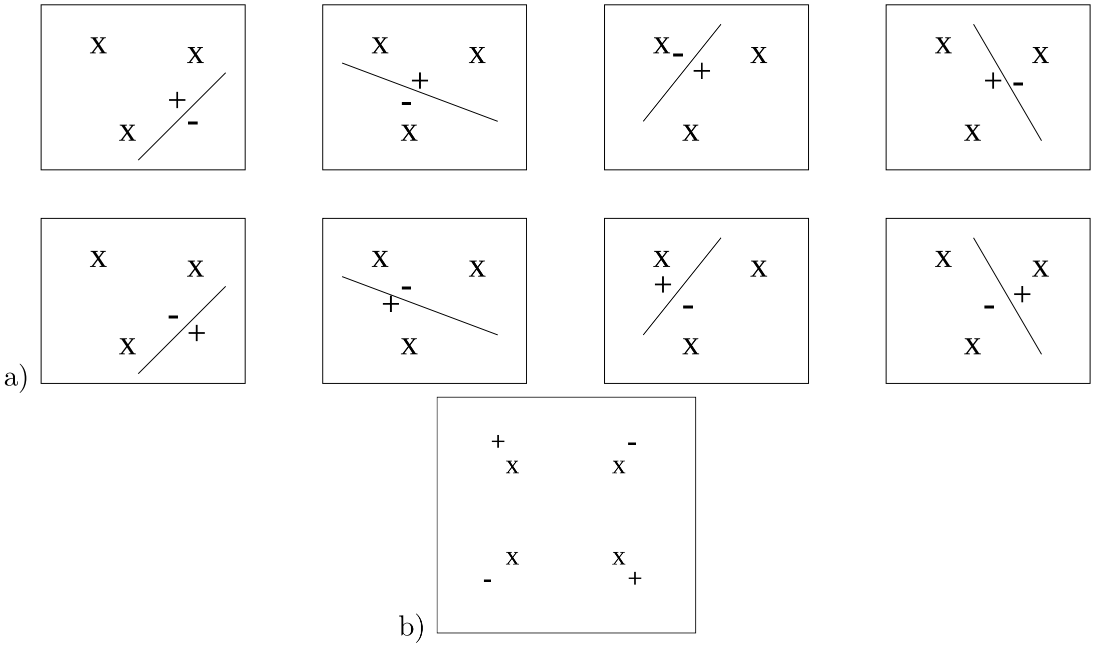


Figure 20: a) The set of linear classifiers in 2d can label three points in all possible ways; b) a labeling of four points that cannot be obtained with a linear classifier.

8 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[7] 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 in Euclidean distance. Cosine similarity is simply the angle between two vectors (elements):

$$cos(x^{(i)}, x^{(j)}) = \frac{x^{(i)} \cdot x^{(j)}}{\|x^{(i)}\| \|x^{(j)}\|} = \frac{\sum_{l=1}^{d} x_l^{(i)} x_l^{(j)}}{\sqrt{\sum_{l=1}^{d} (x_l^{(i)})^2} \sqrt{\sum_{l=1}^{d} (x_l^{(j)})^2}} \tag{193}$$

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

$$dist(x^{(i)}, x^{(j)}) = \|x^{(i)} - x^{(j)}\|^2 = \sum_{l=1}^{d} (x_l^{(i)} - x_l^{(j)})^2 \tag{194}$$

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

$$dist(x^{(i)}, x^{(j)}) = \|x^{(i)} - x^{(j)}\|_1 = \sum_{l=1}^{d} |x_l^{(i)} - x_l^{(j)}| \tag{195}$$

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

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

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:

$$cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) = \sum_{j=1\ldots k} \sum_{i \in C_j} \|x^{(i)} - z^{(j)}\|^2 \tag{196}$$

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:

$$C_j = \{i \in \{1, \ldots, n\} \textit{ s.t. the closest representative of } x^{(i)} \textit{ is } z^{(j)}\} \tag{197}$$

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,

$$\begin{aligned} cost(z^{(1)}, \ldots, z^{(k)}) &= \min_{C_1, \ldots, C_k} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) && (198) \\ &= \min_{C_1, \ldots, C_k} \sum_{j=1\ldots k} \sum_{i \in C_j} \|x^{(i)} - z^{(j)}\|^2 && (199) \\ &= \sum_{i=1,\ldots,n} \min_{j=1,\ldots,k} \|x^{(i)} - z^{(j)}\|^2 && (200) \end{aligned}$$

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 8.

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

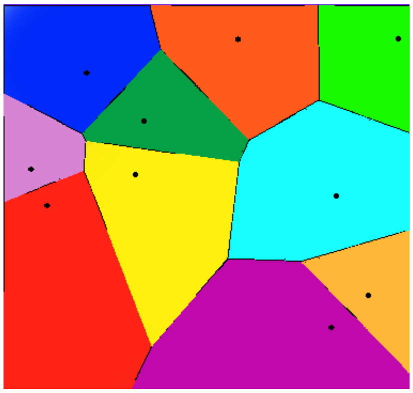


Figure 21: An example of Voronoi diagram

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

$$cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) \tag{201}$$

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

$$cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) \overset{(a)}{\geq} \min_{C_1,\ldots,C_k} cost(C_1, C_2, \ldots, C_k, z^{(1)}, \ldots, z^{(k)}) \tag{202}$$

$$= cost(C_1', C_2', \ldots, C_k', z^{(1)}, \ldots, z^{(k)}) \tag{203}$$

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

$$\begin{aligned} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) &\overset{(b)}{\geq} \min_{z^{(1)},\ldots,z^{(2)}} cost(C'_1, C'_2, \ldots, C'_k, z^{(1)}, \ldots, z^{(k)}) && (204) \\ &= cost(C'_1, C'_2, \ldots, C'_k, z'^{(1)}, \ldots, z'^{(k)}) && (205) \end{aligned}$$

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

$$\sum_{i \in C'_j} \|x^{(i)} - z^{(j)}\|^2 \tag{206}$$

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

$$z'^{(j)} = \frac{1}{|C'_j|} \sum_{i \in C'_j} x^{(i)} \tag{207}$$

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.(200), 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:

$$Cost(C^1, \ldots, C^k, z^{(1)}, \ldots, z^{(k)}) = \sum_{j=1}^{k} \sum_{i \in C^j} d(x^{(i)}, z^{(j)}) \tag{208}$$

The algorithm:

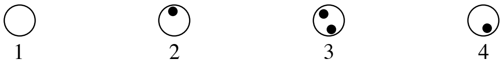
1. Initialize exemplars: $\{z^{(1)}, \ldots, z^{(k)}\} \subseteq \{x^{(1)}, \ldots, x^{(n)}\}$ (exemplars are k points from the original dataset)
2. Repeat until there is no further change in cost:
   (a) for each j: $C^j = \{i : x^{(i)}\text{'s closest exemplar is } z^{(j)}\}$
   (b) for each j: set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

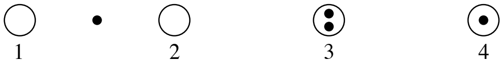
Initialization

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[8]. 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.

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.

[8] This example is taken from Sanjoy Dasgupta.

K-means++ initializer

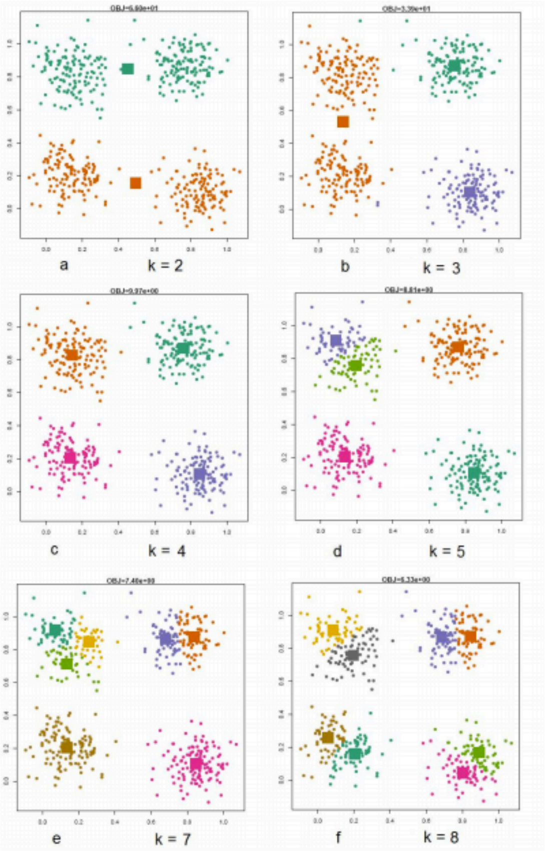
```
pick x uniformly at random from the dataset, and set T=x
while $|T| \leq k$
pick x at random, with probability proportional to the cost $\min_{z \in T} ||x - z||^2$
$T = T \cup \{x\}$
```

We can offer some guarantees for this initialization procedure:

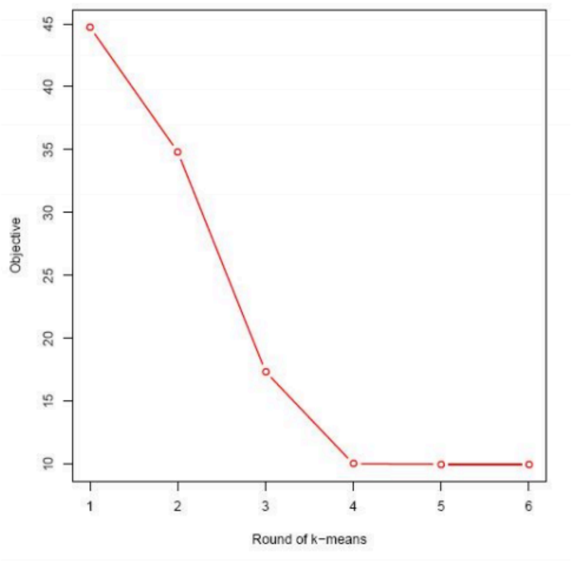
K-means++ guarantee Let T be the initial set of centers chosen by K-means++. Let T^* be the set of optimal cluster centers. Then, $E|cost(T)| \leq cost(T^*)O(logK)$, where the expectation is over the randomness in the initialization procedure, and $cost(T) = \sum_{i=1}^{n} \min_{z \in T} \|x^{(i)} -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

Choosing K

The selection of k greatly impacts the quality of your clustering solution. Figure 8 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 8 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.

9 Generative models

So far we have primarily focused on classification, where the goal is to find a separator that sets apart two classes. The internal structure of the classes is not directly captured by (or cared for) the classifier. In fact, datasets with very different structure may have exactly the same separator. Intuitively, understanding the structure of the data should be helpful for classification as well. We have touched this issue briefly in the context of K-means clustering. However, clustering provided a limited way of capturing the properties of input examples. For example, simply providing a cluster representative shows nothing about how the points are spread. Moreover, not all data have clusters of the type that K-means is geared to find. An example of such a dataset is shown in Figure 22.

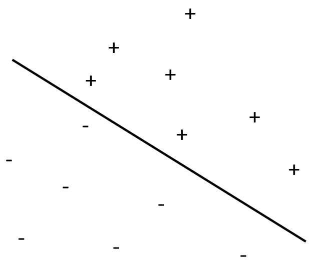


Figure 22: A labeled dataset where the class distinction cuts across a single cluster

Today, we will discuss using probabilistic models for data. We will demonstrate how these models can be directly used in classification: once we have a model for each class of examples, we can estimate how likely it was generated by each class-specific model, and determine the predicted label accordingly. A central question in probabilistic modeling is how to select a model that best fits the properties of the data we are trying to capture. We will describe several models and their properties.

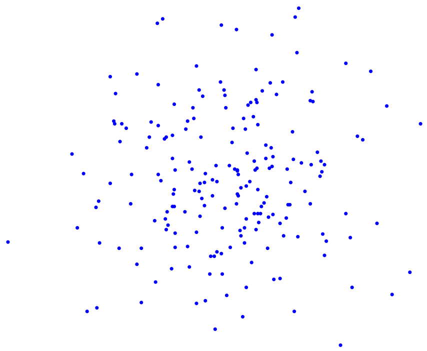


Figure 23: A cluster of points.

9.1 Multinomials

We can use probabilistic models with data other than real valued points in $\mathcal{R}^d$. Consider, for example, documents represented by bags-of-words. While in supervised classification we mapped each document into a point in space, here we will look at a different representation: we will treat text as a bag of words to be *generated.* Our model will generate different bags of words with different probability. We are looking for a model that generates the observed bags (i.e., documents in our dataset) with high probability.

To make our task simpler, we will further simplify the setting by assuming that words in a document are generated one word at a time, each word independently from others, but with a specific distribution. To specify such a model, each word $w \in W$ is associated with parameter θ_w

$$\theta_w \geq 0$$

$$\sum_{w \in W} \theta_w = 1$$

$$P(w|\theta) = \theta_w$$

For example, consider two probability distributions over four words: dog, cat, tulip, rose. The first distribution has parameters $\theta_{dog} = 0.5$, $\theta_{cat} = 0.4$, $\theta_{tulip} = 0.05$, $\theta_{rose} = 0.05$, while the second one is parametrized as $\theta_{dog} = 0.1$, $\theta_{cat} = 0.1$, $\theta_{tulip} = 0.5$, $\theta_{rose} = 0.3$. Now consider the following document (i.e., bag of words): dog, dog, cat, dog, cat, tulip. Which distribution is more likely to generate this document? To answer this question, we need to compute the likelihood of the document $D = \{w_!, w_2, \ldots, w_n\}$ under a given parameter setting θ:

$$P(D|\theta) = \prod_{i=1}^{N} \theta_{w_i} = \prod_{w} \theta_w^{n(w)},$$

where $n(w)$ is a count of w in document D.

In oder to use these probabilistic models, we need a way to estimate the parameters. Let's start with the case of a single document. We want to find θ that maximize the likelihood of observed data

$$\max_\theta P(D|\theta) \Leftrightarrow \max_\theta \log P(D|\theta)$$

$$\log P(D|\theta) = \log \prod_w \theta_w^{n(w)} = \sum_w n(w) \log \theta_w$$

We need to find θ that maximizes this expression. To see how this works, let's make it even simpler and assume that our vocabulary is of size 2, i.e., $W = \{0, 1\}$. Then

$$\log P(D|\theta) = n(0) \log \theta_0 + n(1) \log \theta_1 = n(0) \log(1 - \theta_1) + n(1) \log \theta_1$$

$$\frac{\partial}{\partial \theta_1} (n(0) \log(1 - \theta_1) + n(1) \log \theta_1) = 0$$

$$-\frac{n(0)}{1 - \theta_1} + \frac{n(1)}{\theta_1} = 0$$

$$
\begin{aligned}
-n(0)\theta_1 + n(1)(1-\theta_1) = 0 \\
-n(0)\theta_1 + n(1) - n(1)\theta_1 = 0 \\
-\theta_1(n(0)+n(1)) = -n(1) \\
\hat{\theta}_1 = \frac{n(1)}{n(0)+n(1)}
\end{aligned}
$$

Using Lagrange multipliers, you can show that parameters will have a similar form for an arbitrary size vocabulary:

$$\hat{\theta}_w = \frac{n(w)}{\sum_{w' \in W} n(w')}$$

In the discussion above, we considered training from a single document. Similar argument can be applied to the case where we are estimating parameters from multiple documents. Due to our independence assumption, the documents are "collapsed" into a giant bag, so we compute word counts across the documents.

$$P(D_1, \ldots, D_T|\theta) = \prod_{t=1}^{T} P(D_t|\theta) = \prod_{w \in W} \theta_w^{\sum_{t=1}^{T} n_t(w)}$$

Until now, we have seen two models: Gaussians and multinomials. Let's put these models in use for classification problems. We will start with the task of document classification. We are given two sets of documents, with known labels "+" and "-". We are interested in a model that generates documents for each class. Those are *class-conditional* distributions:

$$
\begin{aligned}
P(D|\theta^+) = \prod_w (\theta_w^+)^{n(w)} \\
P(D|\theta^-) = \prod_w (\theta_w^-)^{n(w)}
\end{aligned}
$$

We estimate θ_w^+ and θ_w^- from the corresponding labeled documents using the method described above. Now, we have two multinomials, each tailored to generate one class of documents. Given a document with an unknown label, how can we classify it? Assume that the likelihood of "+" and "-" is the same (later, we will remove this assumption). We will select the label based on which model assigns a higher likelihood to the document:

$$\log \frac{P(D|\theta^+)}{P(D|\theta^-)} = \begin{cases} \geq 0, & + \\ < 0, & - \end{cases}$$

where the log-likelihood ratio as a discriminant function can be written as

$$
\begin{aligned}
\log \frac{P(D|\theta^+)}{P(D|\theta^-)} &= \log P(D|\theta^+) - \log P(D|\theta^-) \\
&= \sum_w n(w)(\log \theta_w^+ - \log \theta_w^-) \\
&= \sum_w n(w) \underbrace{\log \frac{\theta_w^+}{\theta_w^-}}_{=\theta_w}
\end{aligned}
$$

This expression represents a linear classifier in a feature representation that concatenates word counts:

$$\log \frac{P(D|\theta^+)}{P(D|\theta^-)} = \Phi(D) \cdot \theta = \begin{bmatrix} n(w_1) \\ \vdots \\ n(w_{|W|}) \end{bmatrix} \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_{|W|} \end{bmatrix} \tag{209}$$

Now let's consider the case where the likelihood of the label classes is not the same. We can add the prior class frequencies, $P(y=+)$ and $P(y=-)$, directly to the log-likelihood ratio, i.e.,

$$\log \frac{P(D|\theta^+)P(y=+)}{P(D|\theta^-)P(y=-)} = \sum_w n(w) \underbrace{\log \frac{\theta_w^+}{\theta_w^-}}_{\theta_w} + \underbrace{\log \frac{P(y=+)}{P(y=-)}}_{=\theta_0} = \sum_w n(w)\theta_w + \theta_0 \tag{210}$$

or, equivalently, classify according to the posterior probability

$$P(y=+|D) = \frac{P(D|y=+)P(y=+)}{P(D)} = \frac{P(D|y=+)P(y=+)}{P(D|y=+)P(y=+) + P(D|y=-)P(y=-)},$$

where $P(D|y=+) = P(D|\theta^+) = \prod_w (\theta_w^+)^{n(w)}$. Once we estimated θ_w^+ and θ_w^- from the training data, we can classify an unlabeled document D by comparing the values of $P(y=+|D)$ and $P(y=-|D)$.

Smoothing One important issue related to parameter estimation is accounting for unseen events. Consider what happens if our training sample does not include word w? In this case, $\hat{\theta}_w = 0$. This problem will be particularly acute if we have little training data and many words are unseen during training.

One way to deal with the the sparsity problem is to introduce a prior distribution over parameters θ:

$$P(\theta|\lambda_1, \ldots, \lambda_n) = P(\theta|\lambda) = C \cdot \prod_w \theta_w^{\lambda_w},$$

where $\lambda_1, \ldots, \lambda_n$ are *hyper-parameters.* θ that maximizes this prior is simply $\hat{\theta}_w = \lambda_w / \sum_{w'} \lambda_{w'}$ and represents the "default" value that the prior highlights. During estimation, the hyper-parameters $\lambda_1, \ldots, \lambda_n$ act as *pseudo-counts* that supplement the observed counts in the data:

$$P(\theta|D) \propto P(D|\theta)P(\theta|\lambda) = \prod_w [\theta_w^{n(w)} \cdot \theta_w^{\lambda_w}] = \prod_w \theta_w^{n(w)+\lambda_w}$$

In other words, when we find the parameter values that maximize the posterior $P(\theta|D)$, we simply combine the observed counts with the pseudo counts from the prior. When observed counts are much larger than the hyper-parameters, the effect of the prior is miniscule. However, if the dataset is small, the pseudo counts will push the parameter estimates towards the default values specified by the prior.

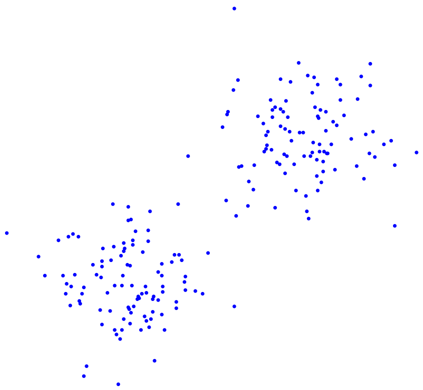


Figure 24: Two cluster of points.

9.2 Gaussians

Consider a cluster of points shown in Figure 29. 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:

$$P(x|\theta) = N(x; \mu, \sigma^2 I) = \frac{1}{(2\pi\sigma^2)^{d/2}} \exp(-\frac{1}{2\sigma^2} \|x - \mu\|^2) \tag{211}$$

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 29 below. σ^2 is the average squared variation of coordinates of x from the coordinates of the mean μ (see below). The value of σ determines how the probability of seeing points away from the mean decays. A small σ will result in a tight cluster of points (points close to the mean), while a large value of σ corresponds to a widely spread cluster of points. 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

$$L(S_n; \mu, \sigma^2) = \prod_{t=1}^{n} N(x^{(t)}; \mu, \sigma^2 I) \tag{212}$$

Since the training data S_n is fixed (given), we view this as a function of the parameters μ

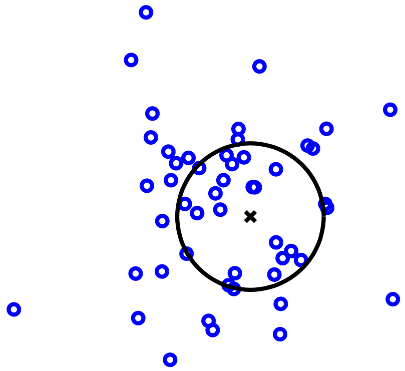


Figure 25: Samples from a spherical Gaussian distribution and the corresponding representation in terms of the mean (center) and the standard deviation (radius).

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.

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

$$\begin{aligned} l(S_n; \mu, \sigma^2) &= \sum_{t=1}^{n} \log N(x^{(t)}; \mu, \sigma^2 I) && (213) \\ &= \sum_{t=1}^{n} \left[-\frac{d}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \|x^{(t)} - \mu\|^2 \right] && (214) \\ &= -\frac{nd}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{t=1}^{n} \|x^{(t)} - \mu\|^2 && (215) \end{aligned}$$

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

$$\hat{\mu} = \frac{1}{n} \sum_{t=1}^{n} x^{(t)}, \quad \hat{\sigma}^2 = \frac{1}{dn} \sum_{t=1}^{n} \|x^{(t)} - \hat{\mu}\|^2 \qquad (216)$$

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).

Classification using Gaussians Similarly to multinomials, we can use Gaussians for classification. We assume that each class is associated with its own Gaussian. For simplicity, we assume that both of them have the same variance. Thus, our two Gaussians will have parameters μ^+, σ and μ^-, σ. As a result,

$$
\begin{aligned}
\log \frac{P(x|\mu^+,\sigma)}{P(x|\mu^-,\sigma)} &= \log P(x|\mu^+,\sigma) - \log P(x|\mu^-,\sigma) \\
&= \log[C \cdot e^{-\frac{1}{2\sigma^2}||x-\mu^+||^2}] - \log[C \cdot e^{-\frac{1}{2\sigma^2}||x-\mu^-||^2}] \\
&= \frac{1}{2\sigma^2}(2x \cdot \mu^+ - 2x \cdot \mu^-) - ||\mu^+||^2 + ||\mu^-||^2 \\
&= \frac{x \cdot (\mu^+ - \mu^-)}{\sigma^2} - \frac{1}{2\sigma^2}(||\mu^+||^2 - ||\mu^-||^2)
\end{aligned}
$$

As in the case with multinomials, the derived classifier is linear in x. The offset θ_0 is given by $\frac{1}{2\sigma^2}(||\mu^+||^2 - ||\mu^-||^2)$, while $\theta = \frac{(\mu^+ - \mu^-)}{\sigma^2}$.

9.3 Mixture of Gaussians

Consider data shown in Figure 30 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*

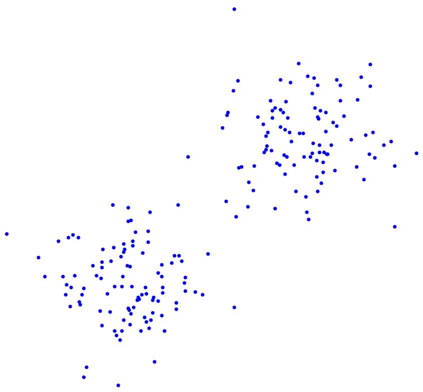


Figure 26: Clusters of points.

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 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.

9.3.1 A mixture of spherical Gaussians

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

$$
N(x; \mu^{(j)}, \sigma_j^2 I), \quad j = 1, \ldots, k \qquad (217)
$$

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,

$$\begin{aligned} j &\sim \text{Multinomial}(p_1, \ldots, p_k) && (218) \\ x &\sim P(x|\mu^{(j)}, \sigma_j^2) && (219) \end{aligned}$$

Figure 31 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[9] (mixing proportions) p_j

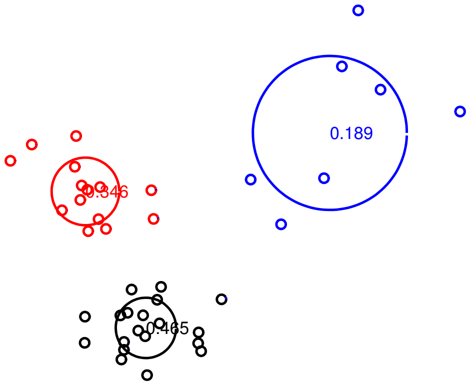


Figure 27: Samples from a mixture of Gaussian distribution with colors indicating the sampled cluster labels. The Figure also shows the corresponding Gaussian cluster models, included the prior cluster frequencies as numbers.

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

$$\theta = \{\mu^{(1)}, \ldots, \mu^{(k)}, \sigma_1^2, \ldots, \sigma_k^2, p_1, \ldots, p_k\}, \tag{220}$$

[9] 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.

we must sum over all the alternative ways we could have generated x (all the ways that Eq.(243) could have resulted in x). In other words,

$$P(x|\theta) = \sum_{j=1}^{k} p_j N(x; \mu^{(j)}, \sigma_j^2 I) \tag{221}$$

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.

9.3.2 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,

$$\delta(j|t) = \begin{cases} 1, & \text{if } x^{(t)} \text{ is assigned to } j \\ 0, & \text{otherwise} \end{cases} \tag{222}$$

Using this notation, our maximum likelihood objective is

$$\sum_{t=1}^{n} \left[\sum_{j=1}^{k} \delta(j|t) \log(\, p_j N(x^{(t)}; \mu^{(j)}, \sigma_j^2 I)\,) \right] = \sum_{j=1}^{k} \left[\sum_{t=1}^{n} \delta(j|t) \log(\, p_j N(x^{(t)}; \mu^{(j)}, \sigma_j^2 I)\,) \right] \tag{223}$$

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

$$\begin{aligned}
\hat{n}_j &= \sum_{t=1}^{n} \delta(j|t) \;\; \text{(number of points assigned to cluster } j\text{)} && (224) \\
\hat{p}_j &= \frac{\hat{n}_j}{n} \;\; \text{(fraction of points in cluster } j\text{)} && (225) \\
\hat{\mu}^{(j)} &= \frac{1}{\hat{n}_j} \sum_{t=1}^{n} \delta(j|t) x^{(t)} \; \text{(mean of points in cluster } j\text{)} && (226) \\
\hat{\sigma}_j^2 &= \frac{1}{d\hat{n}_j} \sum_{t=1}^{n} \delta(j|t) \|x^{(t)} - \hat{\mu}^{(j)}\|^2 \; \text{(mean squared spread in cluster } j\text{)} && (227)
\end{aligned}$$

9.3.3 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

$$l(S_n;\theta) = \sum_{t=1}^{n} \log P(x^{(t)}|\theta) = \sum_{t=1}^{n} \log\left(\sum_{j=1}^{k} p_j N(x^{(t)};\mu^{(j)},\sigma_j^2 I)\right) \tag{228}$$

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:

$$\hat{\sigma}^2 = \frac{1}{dn}\sum_{t=1}^{n} \|x^{(t)} - \hat{\mu}\|^2 \tag{229}$$

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

$$p(j|t) = \frac{p_j N(x;\mu^{(j)},\sigma_j^2 I)}{P(x|\theta)} = \frac{p_j N(x;\mu^{(j)},\sigma_j^2 I)}{\sum_{l=1}^{k} p_l N(x;\mu^{(l)},\sigma_l^2 I)} \tag{230}$$

  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.

$$\hat{n}_j = \sum_{t=1}^{n} p(j|t) \text{ (effective number of points assigned to cluster } j\text{)} \tag{231}$$

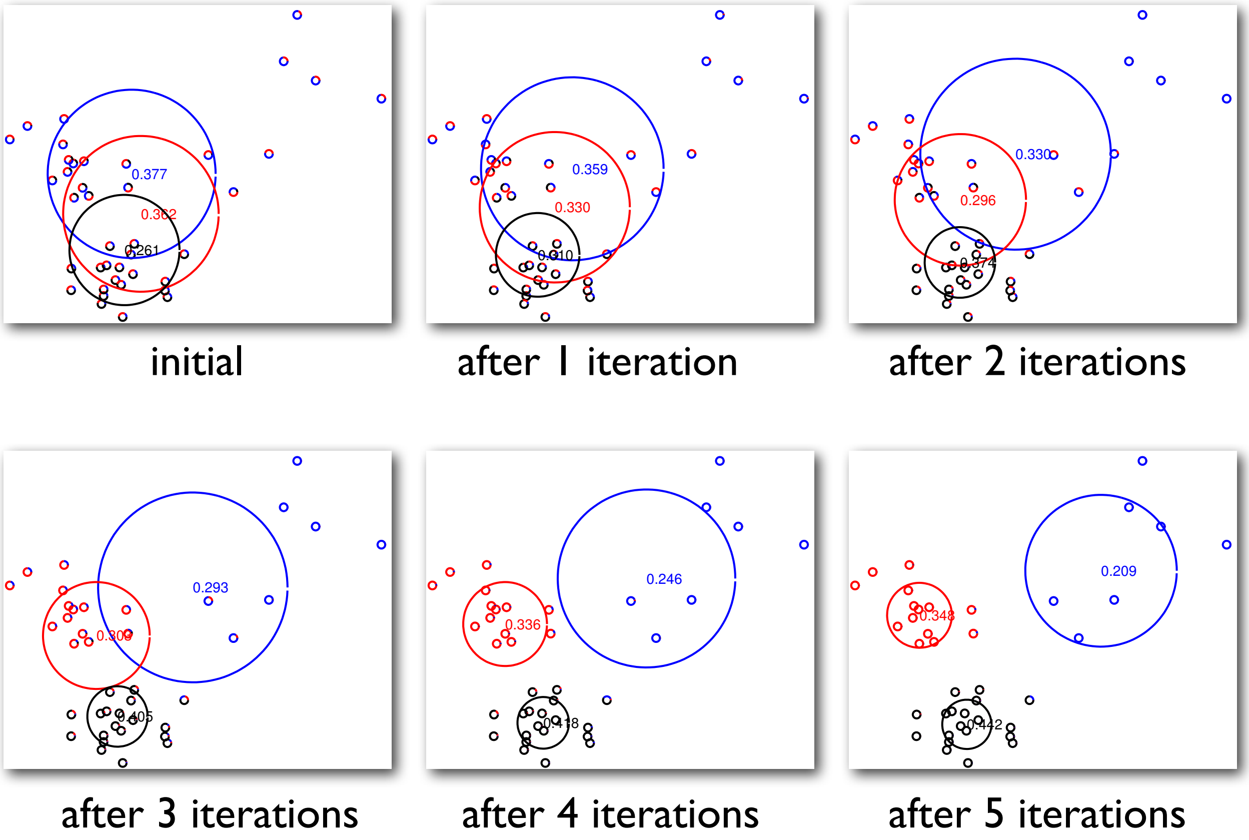
$$\hat{p}_j = \frac{\hat{n}_j}{n} \text{ (fraction of points in cluster } j\text{)} \tag{232}$$

$$\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{t=1}^{n} p(j|t) x^{(t)} \text{ (weighted mean of points in cluster } j\text{)} \tag{233}$$

$$\hat{\sigma}_j^2 = \frac{1}{d\hat{n}_j} \sum_{t=1}^{n} p(j|t) \|x^{(t)} - \hat{\mu}^{(j)}\|^2 \text{ (weighted mean squared spread)} \tag{234}$$

  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 32 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 28: An example of running the EM-algorithm for five iterations

10 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 29. 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:

$$P(x|\theta) = N(x;\mu,\sigma^2 I) = \frac{1}{(2\pi\sigma^2)^{d/2}}\exp(-\frac{1}{2\sigma^2}\|x-\mu\|^2) \tag{235}$$

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 29 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

$$L(S_n;\mu,\sigma^2) = \prod_{t=1}^{n} N(x^{(t)};\mu,\sigma^2 I) \tag{236}$$

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.

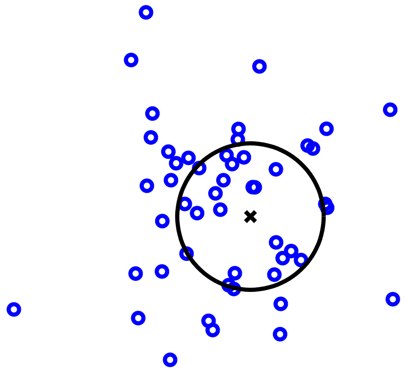


Figure 29: Samples from a spherical Gaussian distribution and the corresponding representation in terms of the mean (center) and the standard deviation (radius).

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

$$\begin{aligned} l(S_n;\mu,\sigma^2) &= \sum_{t=1}^{n} \log N(x^{(t)};\mu,\sigma^2 I) && (237)\\ &= \sum_{t=1}^{n} \left[-\frac{d}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\|x^{(t)}-\mu\|^2 \right] && (238)\\ &= -\frac{nd}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}\|x^{(t)}-\mu\|^2 && (239)\end{aligned}$$

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

$$\hat{\mu} = \frac{1}{n}\sum_{t=1}^{n} x^{(t)}, \quad \hat{\sigma}^2 = \frac{1}{dn}\sum_{t=1}^{n}\|x^{(t)}-\hat{\mu}\|^2 \tag{240}$$

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 30 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

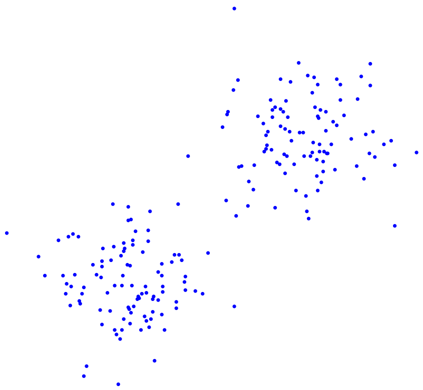


Figure 30: Clusters of points.

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

$$N(x; \mu^{(j)}, \sigma_j^2 I), \quad j = 1, \ldots, k \tag{241}$$

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,

$$j \quad \sim \quad \text{Multinomial}(p_1, \ldots, p_k) \tag{242}$$

$$x \quad \sim \quad P(x|\mu^{(j)}, \sigma_j^2) \tag{243}$$

Figure 31 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[10] (mixing proportions) p_j

[10] 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.

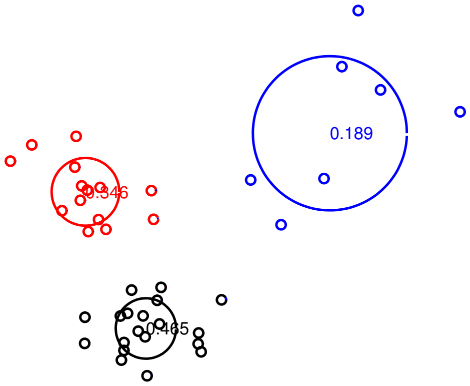


Figure 31: Samples from a mixture of Gaussian distribution with colors indicating the sampled cluster labels. The Figure also shows the corresponding Gaussian cluster models, included the prior cluster frequencies as numbers.

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

$$\theta = \{\mu^{(1)}, \ldots, \mu^{(k)}, \sigma_1^2, \ldots, \sigma_k^2, p_1, \ldots, p_k\}, \tag{244}$$

we must sum over all the alternative ways we could have generated x (all the ways that Eq.(243) could have resulted in x). In other words,

$$P(x|\theta) = \sum_{j=1}^{k} p_j N(x; \mu^{(j)}, \sigma_j^2 I) \tag{245}$$

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,

$$\delta(j|t) = \begin{cases} 1, & \text{if } x^{(t)} \text{ is assigned to } j \\ 0, & \text{otherwise} \end{cases} \tag{246}$$

Using this notation, our maximum likelihood objective is

$$\sum_{t=1}^{n}\left[\sum_{j=1}^{k}\delta(j|t)\log(\,p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\,)\right] = \sum_{j=1}^{k}\left[\sum_{t=1}^{n}\delta(j|t)\log(\,p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\,)\right] \quad (247)$$

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

$$\begin{aligned}
\hat{n}_j &= \sum_{t=1}^{n}\delta(j|t) \text{ (number of points assigned to cluster } j) && (248)\\
\hat{p}_j &= \frac{\hat{n}_j}{n} \text{ (fraction of points in cluster } j) && (249)\\
\hat{\mu}^{(j)} &= \frac{1}{\hat{n}_j}\sum_{t=1}^{n}\delta(j|t)x^{(t)} \text{ (mean of points in cluster } j) && (250)\\
\hat{\sigma}_j^2 &= \frac{1}{d\hat{n}_j}\sum_{t=1}^{n}\delta(j|t)\|x^{(t)}-\hat{\mu}^{(j)}\|^2 \text{ (mean squared spread in cluster } j) && (251)
\end{aligned}$$

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

$$l(S_n;\theta) = \sum_{t=1}^{n}\log P(x^{(t)}|\theta) = \sum_{t=1}^{n}\log\left(\sum_{j=1}^{k}p_jN(x^{(t)};\mu^{(j)},\sigma_j^2I)\right) \quad (252)$$

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:

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{t=1}^{n} \|x^{(t)} - \hat{\mu}\|^2 \tag{253}$$

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

  $$p(j|t) = \frac{p_j N(x; \mu^{(j)}, \sigma_j^2 I)}{P(x|\theta)} = \frac{p_j N(x; \mu^{(j)}, \sigma_j^2 I)}{\sum_{l=1}^{k} p_l N(x; \mu^{(l)}, \sigma_l^2 I)} \tag{254}$$

  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.

  $$\hat{n}_j = \sum_{t=1}^{n} p(j|t) \text{ (effective number of points assigned to cluster } j) \tag{255}$$

  $$\hat{p}_j = \frac{\hat{n}_j}{n} \text{ (fraction of points in cluster } j) \tag{256}$$

  $$\hat{\mu}^{(j)} = \frac{1}{\hat{n}_j} \sum_{t=1}^{n} p(j|t) x^{(t)} \text{ (weighted mean of points in cluster } j) \tag{257}$$

  $$\hat{\sigma}_j^2 = \frac{1}{d\hat{n}_j} \sum_{t=1}^{n} p(j|t) \|x^{(t)} - \hat{\mu}^{(j)}\|^2 \text{ (weighted mean squared spread)} \tag{258}$$

  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 32 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.

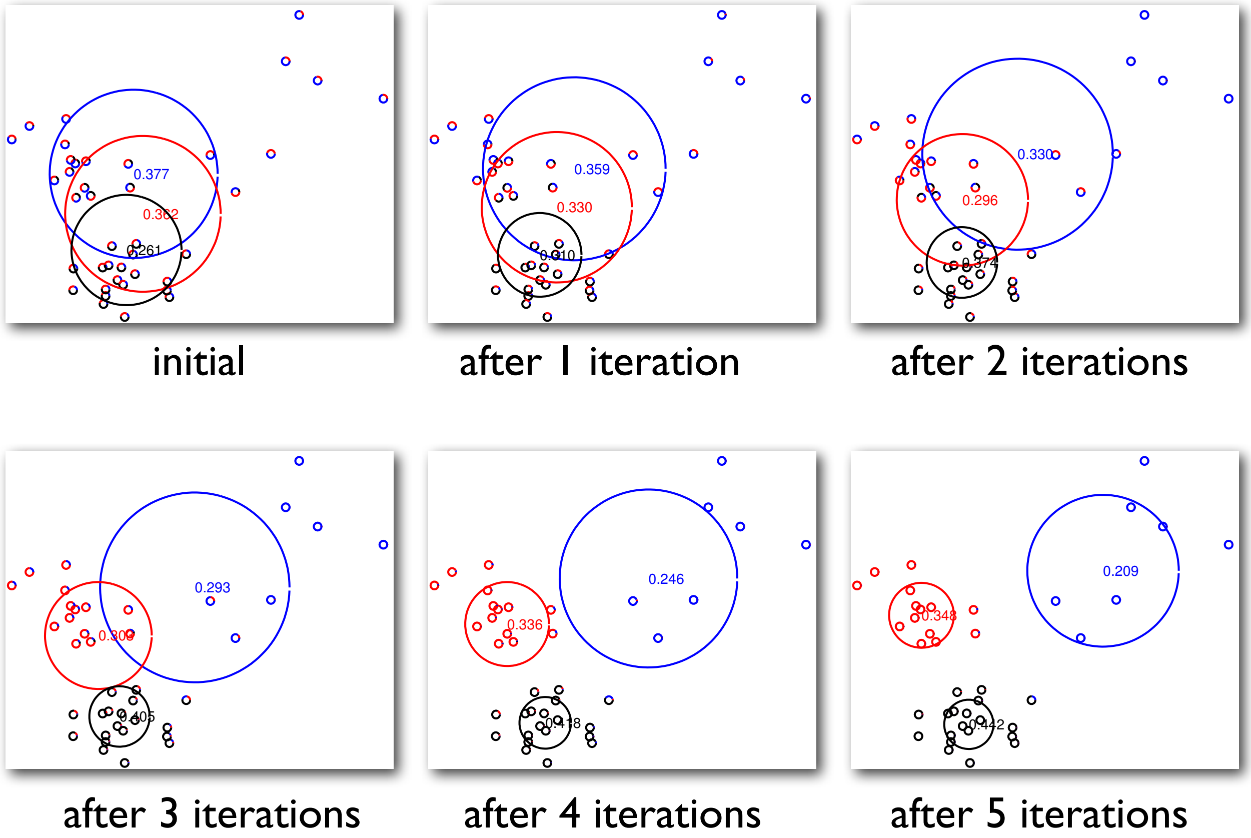


Figure 32: An example of running the EM-algorithm for five iterations

11 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

$$X_1 : \begin{array}{|cc} H & T \\ \hline 0.5 & 0.5 \end{array}, \quad X_2 : \begin{array}{|cc} H & T \\ \hline 0.5 & 0.5 \end{array} \tag{259}$$

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.

$$X_3|X_1, X_2 : \begin{array}{c|cc} X_1, X_2 & \text{T} & F \\ \hline H, H & 1 & 0 \\ H, T & 0 & 1 \\ T, H & 0 & 1 \\ T, T & 1 & 0 \end{array} \tag{260}$$

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

$$P(X_1 = x_1, X_2 = x_2, X_3 = x_3) = P(X_1 = x_1)P(X_2 = x_2)P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{261}$$

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 33.

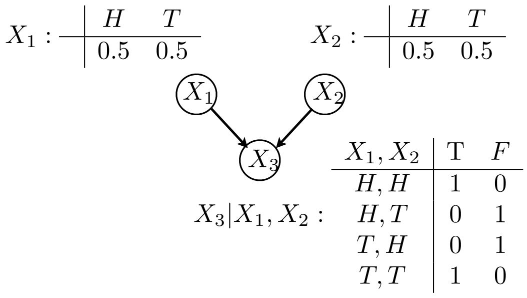


Figure 33: a) A directed graph for the coin toss example with the associated conditional probability tables

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.(261) 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

$$P(X_1 = x_1, X_2 = x_2) = \sum_{x_3} P(X_1 = x_1)P(X_2 = x_2)P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{262}$$

$$= P(X_1 = x_1)P(X_2 = x_2)\sum_{x_3} P(X_3 = x_3|X_1 = x_1, X_2 = x_2) \tag{263}$$

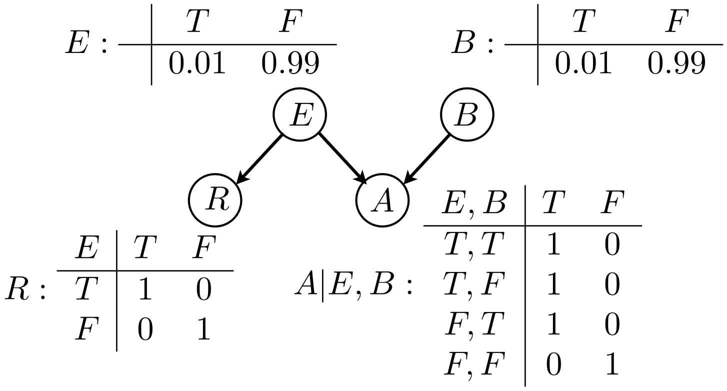
$$= P(X_1 = x_1)P(X_2 = x_2) \tag{264}$$

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 34. 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 34: 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

$$\begin{aligned} & P(E = e, B = b, A = a, R = r) = \\ = \quad & P(E = e)P(B = b)P(A = a|E = e, B = b)P(R = r|E = e) \end{aligned} \tag{265}$$

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:

$$\begin{aligned}
& P(B=b, A=T) = \\
&= \sum_{e \in \{T,F\}} \sum_{r \in \{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b)P(R=r|E=e) && (266) \\
&= \sum_{e \in \{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b) \sum_{r \in \{T,F\}} P(R=r|E=e) && (267) \\
&= \sum_{e \in \{T,F\}} P(E=e)P(B=b)P(A=T|E=e,B=b) && (268) \\
&= P(B=b) \sum_{e \in \{T,F\}} P(E=e)P(A=T|E=e,B=b) && (269)
\end{aligned}$$

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,

$$P(B=T|A=T) = \frac{P(B=T, A=T)}{\sum_{b \in \{T,F\}} P(B=b, A=T)} \qquad (270)$$

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.

12 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

$$f(x_1, \ldots, x_n) = \text{argmax}_{y_1, \ldots, y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_n) \tag{271}$$

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

Three key questions:

- How to specify $p(x_1, \ldots, x_n, y_1, \ldots, y_n)$ 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 $\text{argmax}_{y_1,\ldots,y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_n)$

Model definition

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

$$P(X_1 = x_1, \ldots, X_n = x_n, Y_1 = y_1, \ldots, Y_n = y_n) \tag{272}$$

where $x_i \in \Sigma$, $y_i \in \mathcal{T}$. For brevity, we will write it as $p(x_1, \ldots, x_n, y_1, \ldots, y_n)$, 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 $x_1, \ldots, x_n$, then clearly the symbol after $y_1, \ldots, y_n$, 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,

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

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

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

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

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

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

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

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

$$p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) = p(y_1) \prod_{i=2}^{n+1} p(y_i|y_{i-1}) \prod_{i=1}^{n} p(x_i|y_i) \tag{273}$$

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

$$p(x_1, \ldots, x_n, y_1, \ldots, y_{n+1}) = \prod_{i=1}^{n+1} p(y_i|y_{i-1}) \prod_{i=1}^{n} p(x_i|y_i) \tag{274}$$

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

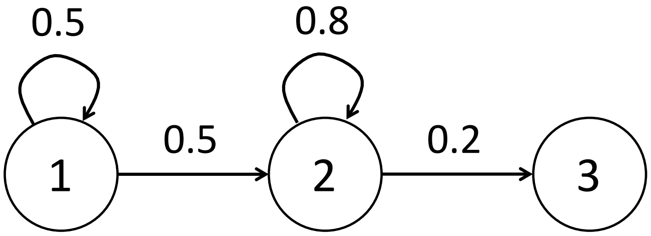


Figure 35: Transition graph for the example.

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:

$$\pi_1 b_1(the) a_{1,2} b_2(dog) a_{2,1} b_1(the) a_{2,3} = 0 \tag{275}$$

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

$$a_{i,j} = \frac{count(i,j)}{count(i)} \tag{276}$$

$$a_{1,2} = \frac{count(1,2)}{count(1)} = \frac{4}{4} = 1,\ a_{2,2} = \frac{count(2,2)}{count(2)} = \frac{0}{4} = 0, \dots \tag{277}$$

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,

$$b_i(o) = \frac{count(i \to o)}{count(i)} \tag{278}$$

$$b_1(e)\frac{count(1 \to e)}{count(1)} = \frac{2}{4} = 0.5, \dots \tag{279}$$

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 $y_1, \dots, y_n$ corresponding to an observed sequence of words $x_1, \dots, x_n$. In other words, we wish to evaluate

$$\text{argmax}_{y_1,\dots,y_n}\, p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) \tag{280}$$

where

$$p(x_1, \dots, x_n, y_1, \dots, y_{n+1}) = \prod_{i=1}^{n+1} a_{y_{i-1},y_i} \prod_{i=1}^{n} b_{y_i}(x_i) \tag{281}$$

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 $\pi_i = a_{*,i},\ i \in \{1, \dots, N\}$ (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:

$$1\ 1\ 1\ 2\ \text{STOP} \tag{282}$$

$$1\ 1\ 2\ 2\ \text{STOP} \tag{283}$$

$$1\ 2\ 2\ 2\ \text{STOP} \tag{284}$$

$$\vdots \tag{285}$$

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

$$r(y_1, \ldots, y_k) = \prod_{i=1}^{k} a_{y_{i-1},y_i} \prod_{i=1}^{k} b_{y_i}(x_i) \tag{286}$$

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

$$\pi(k, v) = \max_{(y_1,\ldots,y_k)\in S(k,v)} r(y_1, \ldots, y_k) \tag{287}$$

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 $k \in \{1, \ldots, n\}$

$$\pi(k, v) = \max_{u \in \mathcal{T}} \{\pi(k-1, u) \cdot a_{u,v} \cdot b_v(x_k)\} \tag{288}$$

  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

$$\max_{y_1,\ldots,y_n} p(x_1, \ldots, x_n, y_1, \ldots, y_n, y_{n+1} = \text{STOP}) = \max_{v \in \mathcal{T}} \{\pi(n, v) \cdot a_{v,\text{STOP}}\} \tag{289}$$

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 $\hat{y}_1, \ldots, \hat{y}_n$? We can do this via *back-tracking.* In other words, at the last step, $\pi(n, v)$ represents maximizations of all $y_1, \ldots, y_n$ such that $y_n = v$. What is the best value for this last tag v, i.e., what is $\hat{y}_n$? It is

$$\hat{y}_n = \text{argmax}_v \left\{\pi(n, v) a_{v,\text{STOP}}\right\} \tag{290}$$

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

$$\hat{y}_{n-1} = \text{argmax}_u \{\pi(n-1, u) a_{u,\hat{y}_n}\} \tag{291}$$

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:

$$\pi_1 = 1.0, \pi_2 = \pi_3 = 0 \tag{292}$$

$$b_1(a) = 1.0, b_1(b) = b_1(c) = 0 \tag{293}$$

$$b_2(a) = 0, b_2(b) = b_2(c) = 0.5 \tag{294}$$

$$a_{1,2} = 1.0, a_{1,1} = a_{1,3} = 0 \tag{295}$$

$$a_{2,3} = 1.0, a_{2,1} = a_{2,2} = 0 \tag{296}$$

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 $\text{count}(x^i, y, p \to q)$ 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

$$\overline{count}(u \to v) = \sum_{i=1}^{m} \sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, u \to v) \tag{297}$$

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

$$a_{u,v} = \frac{\overline{count}(u \to v)}{\sum_{k=1}^{N} \overline{count}(u \to k)} \tag{298}$$

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.

$$\sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, u \to v) \tag{299}$$

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

$$p(y_j = p, y_{j+1} = q|x, \theta) = \sum_{y: y_j = p, y_{j+1} = q} p(y|x, \theta) \tag{300}$$

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. 299 as:

$$\sum_{y} p(y|x^i, \theta^{t-1}) count(x^i, y, p \to q) = \sum_{j=1}^{n} p(y_j = p, y_{j+1} = q|x^i, \theta^{t-1}) \tag{301}$$

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:

$$\alpha_p(j) = p(x_1, \ldots, x_{j-1}, y_j = p|\theta) \tag{302}$$

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:

$$\beta_p(j) = p(x_j, \ldots, x_n|y_j = p, \theta) \tag{303}$$

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:

$$p(y_j = p, y_{j+1} = q|x, \theta) = \frac{1}{Z}\alpha_p(j)a_{p,q}b_p(o_j)\beta_q(j+1)$$

$$Z = p(x_1, \ldots, x_n|\theta) = \sum_p \alpha_p(j)\beta_p(j) \text{ for any } j = 1 \ldots n$$

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. 36 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.

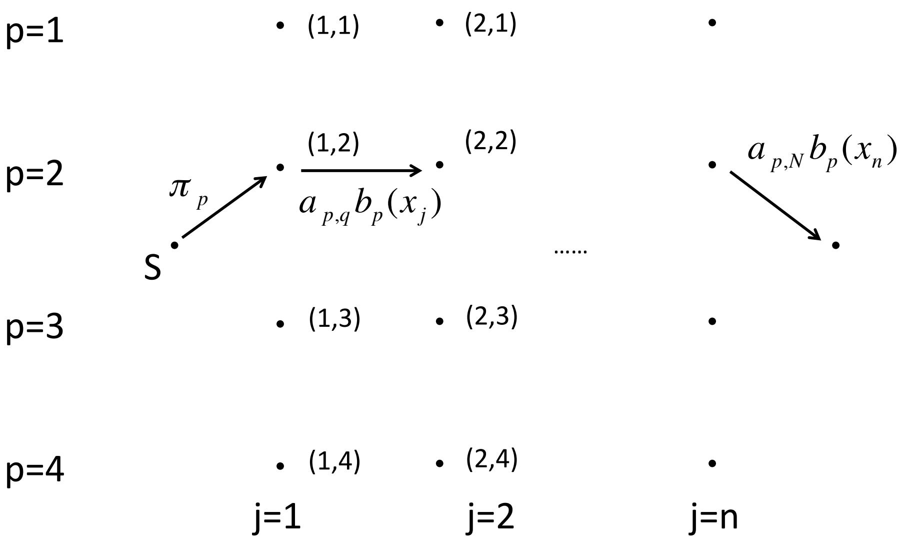


Figure 36: A path associated with state sequence.

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:

$$\alpha_p(j) = p(x_1, \ldots, x_{j-1}, y_j = p|\theta) \tag{304}$$

- Base case:

$$\alpha_p(1) = \pi_p \quad \forall p \in 1 \ldots N-1 \tag{305}$$

- Recursive case

$$\alpha_p(j+1) = \sum_q \alpha_q(j) a_{q,p} b_q(x_j) \quad \forall p \in 1 \ldots N-1, j = 1 \ldots n-1 \tag{306}$$

Backward probability:

$$\beta_p(j) = p(x_j, \ldots, x_n|y_j = p, \theta) \tag{307}$$

- Base case:

$$\beta_p(n) = a_{p,N} b_p(x_n) \quad \forall p \in 1 \ldots N-1 \tag{308}$$

- Recursive case

$$\beta_p(j) = \sum_q a_{p,q} b_p(x_j) \beta_q(j+1) \quad \forall p \in 1 \ldots N-1, j = 1 \ldots n-1 \tag{309}$$

13 Reinforcement Learning

Learning from Feedback

Let's consider a robot navigation problem (see Figure 37 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 38 illustrates this abstraction.

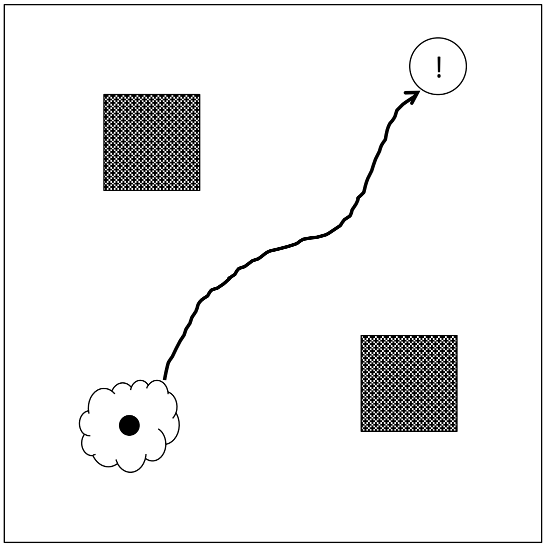


Figure 37: Robot navigation task.

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.

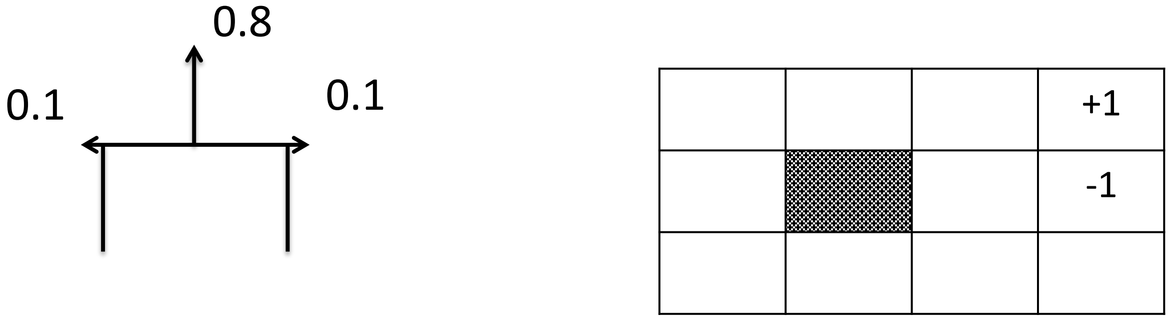


Figure 38: Robot in the grid. With probability 0.8, the robot will move into a specified direction. With probability 0.1, it will move into one of the orthogonal directions.

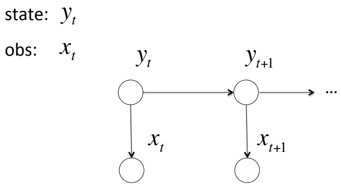


Figure 39: HMM.

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:

$$R_{s_t} = \begin{cases} 1 & \text{if on target} \\ 0 & \text{otherwise} \end{cases} \tag{310}$$

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 42). 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

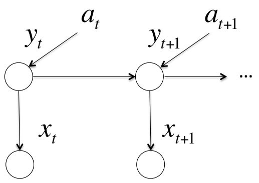


Figure 40: HMM with action.

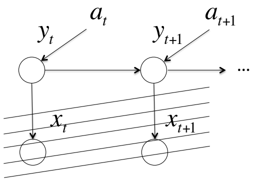


Figure 41: HMM without x_t, assuming the hidden state is observed

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

$$U([s_0, s_1, s_2 \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots \tag{311}$$

$$= \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{max} = \frac{R_{max}}{1-\gamma} \tag{312}$$

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 43, 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 $R(s')$)

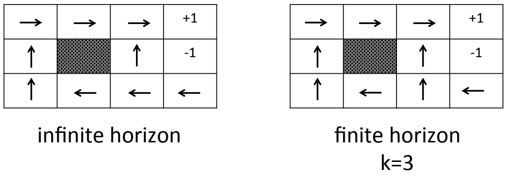


Figure 42: Policy with different horizons.

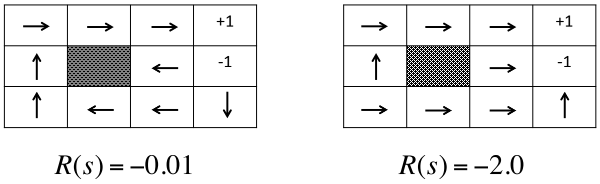


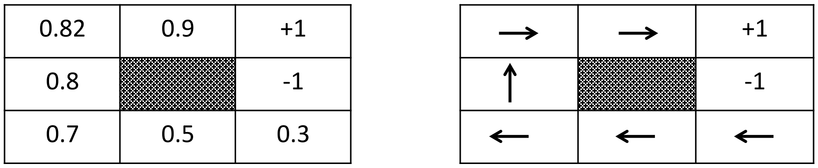
Figure 43: Policy with different rewards. $R(s)$ is reward associated with visiting a state (excluding absorbing states).

- **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 44).



0.82	0.9	+1
0.8	▨	-1
0.7	0.5	0.3

→	→	+1
↑	▨	-1
←	←	←

Figure 44: V-values and associated policy.

The above quantities are clearly related:

$$V^*(s) = \max_a Q^*(s,a) = Q^*(s, \pi^*(s)) \tag{313}$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{314}$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')] \tag{315}$$

$$= \sum_{s'} T(s,\pi^*(s),s')[R(s,\pi^*(s),s') + \gamma V^*(s')] \tag{316}$$

For example, in Eq.(314), 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:

$$x_i{\sim}p(x),\ i = 1,\dots,k$$
$$\hat{p}(x) = \frac{count(x)}{k}$$
$$E[f(x)]{\approx}\sum_x \hat{p}(x)f(x)$$

- **Model-free estimation**: estimate expectation directly from samples

$$x_i{\sim}p(x),\ i = 1,\dots,k$$
$$E[f(x)]{\approx}\frac{1}{k}\sum_{i=1}^{k} f(x_i)$$

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:

$$sample_1 : R(s,a,s'_1) + \gamma \max_{a'} Q_i(s'_1,a')$$
$$sample_2 : R(s,a,s'_2) + \gamma \max_{a'} Q_i(s'_2,a')$$
$$\dots$$
$$sample_k : R(s,a,s'_k) + \gamma \max_{a'} Q_i(s'_k,a')$$

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:

$$\bar{x}_n = \frac{x_n + (1-\alpha) * x_{n-1} + (1-\alpha)^2 * x_{n-2}}{1 + (1-\alpha) + (1-\alpha^2) + \ldots}$$

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:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a')\right] \tag{317}$$

$$= Q(s,a) + \alpha\left[R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a)\right] \tag{318}$$

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

$$\sum_k \alpha_k(s,a) \rightarrow \infty$$

$$\sum_k \alpha_k^2(s,a) < \infty$$

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.