

Case Studies in Machine Learning

David McAllester

Draft of Nov. 23, 2009

Chapter 1

Fundamental Issues in Machine Learning

Any definition of machine learning is bound to be controversial. From a scientific perspective machine learning is the study of learning mechanisms — mechanisms for using past experience to make future decisions. From an engineering perspective machine learning is the study of algorithms for automatically constructing computer software from training data. For example, we might have a large database of translation pairs each of which is an English sentence paired with a French translation. A prominent machine learning problem is to automatically learn a machine translation system from translation pairs. State of the art machine translation systems are currently obtained this manner. Machine learning has become the dominant approach to most of the classical problems of artificial intelligence (AI). Machine learning now dominates the fields of computer vision, speech recognition, natural language question answering, computer dialogue systems, and robotic control. Machine learning has also achieved a prominent role in other areas of computer science such as information retrieval, database consistency, and spam detection. One can argue that machine learning is revolutionizing our understanding of the process of constructing computer software. Machine learning has become a new foundation for much of the practice of computer science.

Although usually viewed as an approach to computer science, machine learning is also closely related to classical statistics. Recently there has even been some debate about whether statistics and machine learning are really the same subject. They certainly share a common mathematical language. But they typically study quite different applications. Machine learning studies applications traditionally viewed as the purview of computer science or electrical engineering. In later chapters we present a variety of case studies mostly in computer vision, speech recognition, and machine translation. Statistics departments generally do not engage in the engineering required to study learning in these areas.

Machine learning also has a significant relationship to neuroscience. Of course the most powerful learning mechanism we know of is the human brain. However, many researchers in machine learning feel that the current understanding of neuroscience is too poor to be of value in the engineering of learning mechanisms. Furthermore, the understanding of learning mechanisms (the science of learning) makes more progress when we directly engineer systems rather than try to reverse engineer the brain. Even very basic properties of neural computation, such as how short term memory is stored, are only very poorly understood. Furthermore, it seems clear that anything neurons can do transistors can also do. And we understand transistors much better than neurons. The main contribution of neuroscience to the practice of machine learning is the artificial neural network as a model of computation. Artificial neural networks, whether or not they are directly related to actual neurons, are a significant tool in the engineering of learning mechanisms.

Having attempted to define machine learning, we now consider a variety of fundamental issues.

1.1 Specialized Learning versus General Learning

Consider a child learning their native language. Since the 1960s linguistic theory has focused on the notion of grammaticality — which strings of words of, say, English form grammatical sentences. It is clear that the number of possible strings of even ten words is so large that no person has heard even a very small fraction of them. Yet we easily judge the grammaticality of word strings that we have never seen before. Noam Chomsky has called this phenomenon “poverty of the stimulus”. For most input spaces of interest (not just language) the training data can never specify a behavior for every possible input. The obvious poverty of the stimulus lead Chomsky to claim that there must be innate knowledge of linguistic structure. This innate knowledge corresponds to the learning theorist’s notion of bias. Any learning system must consider some input-output relationships more likely than others and this preference is called “bias”. The poverty of the stimulus, and the need for bias, is sometimes called the “no free lunch theorem” — to generalize you must have a bias.

Learning bias is a subtle notion and must be examined carefully. Despite Chomsky’s claim, the fact that bias is needed for learning does not necessarily imply that humans have innate knowledge of linguistic structure. The alternative is that there is a single general learning mechanism which handles language along with all of the other things that humans learn. Since bias is necessary for learning, a general learning mechanism would need a general or “universal” bias. The simplest example of a universal bias is a programming language such as C++. It is reasonable to assume that the behavior we want to learn can be expressed as a C++ program. It turns out that there are many more subsets of

word strings, many more notions of grammaticality, than there are 1 million line C++ programs. Simply saying that the grammaticality rule must be expressible with a 1 million line C++ program is already a strong bias. But saying that the notion of grammaticality can be expressed with 1 million line C++ program appears to put no constraints on linguistic theory. It also seems that the choice of programming language is irrelevant, one would not expect a C++ bias to be very different from a JAVA bias or even a FORTRAN bias. All of these biases are universal in the sense that any rule written in one language can be written in any other language in roughly the same number of lines of code (if we let a good coder do the coding). In sharp contrast to Noam Chomsky, Geoff Hinton believes that a single universal bias underlies all human learning. Hinton believes that this universal bias corresponds to the prediction rules that can be implemented in reasonably sized artificial neural networks. This is closely related to the circuit model of computation and is indeed universal. It seems that any natural universal bias is information theoretically sufficient to handle language learning — the C++ program could always start with a coding of universal grammar which is, presumably, short enough to fit in the human DNA.

At a more philosophical level it is worth noting that the language of mathematics is more expressive than C++. In mathematics we can write expressions such as $\inf_x f(x)$, and show that this expression has a well defined value, even if we have no way of computing it. It is possible to define a single formal language, similar to a programming language, in which one can express all of mathematics. For example, we could use the formal language of set theory, although better options exist. In a mathematically universal language one can define any other language that can be defined in mathematics. The language of mathematics is in some sense the ultimate universal language and seems to give the most general possible learning bias.

Although universal bias is information theoretically possible, specialized biases currently play a central role in the practice of machine learning. A specialized bias is by definition not universal — either there are computable rules which can not be expressed at all, or computable rules that are simple under any universal bias but prohibitively complex under the specialized bias. In practice the construction of a specialized bias often takes the form of feature engineering. More generally, many learning systems restrict themselves to rules based on a linear combination of feature values for a specialized set of features. Although specialized features give a specialized bias, feature engineering is a central part of the practice of machine learning. David Lowe's introduction of SIFT features, a certain representation of edge distributions in an image patch, has revolutionized the field of computer vision. But Geoff Hinton, and others in the artificial neural network community, believe that features as useful as SIFT can be learned from the universal bias of neural circuits. But as of this writing SIFT features, and variants of SIFT such as SURF and HOG, dominate the engineering of computer vision systems.

Someday a combination of neuroscience and genomics will answer the ques-

tion of exactly what is humanly innate. It certainly seems plausible that the human DNA specifies system-specific biases for learning each of the visual system, the motor control system the auditory system, and perhaps a bias for a language system. But a single universal bias is also mathematically possible.

1.2 Bayesians versus Frequentists

In the foundations of probability theory there is a tension between Bayesians and frequentists. This tension has generated different schools of thought within the machine learning community. The Bayesian-frequentist tension is perhaps best discussed in terms of one’s beliefs about fundamental physics. In classical physics the world behaves deterministically — given the current state all future states are determined. Given physical determinism, probability is used to represent uncertain beliefs about the world. The interpretation of a probability as a degree of belief is fundamentally Bayesian. In quantum physics the objective world behaves randomly. The probability that an isolated neutron will undergo beta decay in a one minute interval is an objective quantity that can be measured and in principle calculated from the laws of subatomic physics. The interpretation of probability as part of objective reality is fundamentally frequentist. While Bayesians focus on the revision of subjective beliefs, frequentists focus on the measurement or calculation of objective quantities.

Bayesians and frequentists both use the same mathematics of probability (measure theory). Any theorem of probability theory is accepted by both. But the different interpretations of probability lead to different views of what theorems are interesting or important. Bayesians typically focus on theorems related to belief revision. Such theorems typically give a way of calculating a posterior distribution of the form $P(h|D)$ where P is a prior, D is observed data, and h is a hypothesis. This typically involves Bayes’ law $P(h|D) = P(h)P(D|h)/P(D)$. Hence the term “Bayesian”. Frequentists are not so interested in belief revision because in their view the prior P seems subjective and arbitrary. They prefer theorems about the measurement of objective quantities. A typical frequentist theorem is Hoeffding’s inequality. A simple version of this inequality states that for any biased coin where the (objective but unknown) probability of heads is P , if we flip the coin n times and observe a fraction \hat{P} of heads we can compute a certain “confidence interval” for the true value of P . More precisely, for any $\delta \in (0, 1)$ with probability at least $1 - \delta$ over the random measurement process generating \hat{P} we have the following.

$$P = \hat{P} \pm \sqrt{\frac{\ln\left(\frac{2}{\delta}\right)}{2m}} \quad (1.1)$$

If $\delta = .05$ then this is the 95% confidence interval. Note that because δ appears inside a logarithm, the 99% confidence interval is not much wider. There is no prior involved in (1.1), just the assumption that there exists some objective

bias P . The term “frequentist” comes from the tendency to measure objective probabilities by observing empirical frequencies.

Einstein objected to the probabilistic interpretation of quantum mechanics and is widely quoted as saying “God does not throw dice”. Perhaps Einstein was a Bayesian. But the frequentist framework is sensible even if the physical world is deterministic. Lottery numbers are drawn using a device in which ping-pong balls are suspending in moving air until one of the balls get close enough to an exiting air stream and is sucked out. Even if the ping-pong balls are modeled as obeying deterministic classical dynamics, this device is very well modeled as generating a truly random draw of lottery numbers. So even if fundamental physics is deterministic, it is still sensible to model reality as containing random processes whose properties can be objectively measured or calculated. Even if Einstein believed that the timing of beta decay was determined by hidden variables, he may also have believed that the “probability” of beta decay in a period of one minute was, for at least most experimental settings, a well defined physical quantity.

It should be noted that there are strong mathematical arguments supporting the use of probability as a representation of subjective belief. In particular, one can show that unless an agent bases betting decisions on subjective probabilities (satisfying the laws of probability theory) the agent is susceptible to arbitrage — a bookie can place a series of bets with the agent that guarantees that, whatever the outcome of the events, the bookie wins money from the agent. This is called the Dutch book argument for subjective probabilities. But the Dutch book argument does not provide any support for the correctness of subjective beliefs. In fact, it is not clear that the concept of correctness even applies to subjective beliefs. They just are what they are — beliefs. Most people would agree, however, that there can be pathological prior beliefs. For example, there are prior beliefs where belief revision takes far longer to learn the bias of a biased coin than the time needed for the direct frequentist measurement given by (1.1).

1.3 Prior Probability versus Prior Language

There is, of course, a close relationship between Bayesian prior beliefs and the concept of bias. From a Bayesian perspective it is natural to formulate a bias as a prior probability over predictive rules. From a frequentist perspective it is perhaps more natural to formulate bias as a language in which predictive rules are expressed where shorter (simpler) rules are preferred to longer rules. These two formulations are mathematically equivalent. Give a prior P we can interpret $\log_2(1/P(f))$ as the “length” (in bits) of the rule f . Given a language for writing down rules we can think of $2^{-|f|}$ as the prior probability of f where $|f|$ is the length of f in bits. The universal C++ bias can be formulated as a Bayesian prior by assigning a prior probability to each C++ program equal to 2^{-n} where n is the length of the program in bits. This conversion of length in bits to a

prior probability is an expression of Occam's razor. Occam's razor states that simple rules are preferable to complex rules. But the bias induced by Occam's razor depends on the choice of programming language. Choosing a programming language is mathematically equivalent to choosing a prior probability.

1.4 A Formal Notion of Generality

We say that a prior P_1 is as general as a prior P_2 if there exists a fixed ϵ such that for all f we have $P_1(f) \geq \epsilon P_2(f)$.¹ Thinking of P and Q as corresponding to languages, we have that $\log(1/\epsilon)$ is the maximum number of bits that must be added to a Q program to translate it into a P program. Intuitively, language P is as general as Q if any Q program can be converted to a P program by adding only a fixed small number of bits. We say that P is strictly more general than Q if P is as general as Q but Q is not as general as P . For example, C++ predicates are strictly more general than the sets definable by regular expressions or context free grammars. The C++ prior is as general as any other prior definable by an executable programming language. In the worst case we simply write an interpreter in C++ for the other language. The language of mathematics, which is not constrained to be executable, is strictly more general. From a Bayesian perspective there seems to be no foundation for preferring one prior to another. However it seems natural to prefer more general priors, especially when dealing with large quantities of training data.

1.5 Why is Learning Possible?

Given the poverty of the stimulus, why is learning possible at all? In what sense can a learning algorithm be correct? Bayesian belief updating is correct by definition — $P(h|D)$ is simply defined to be the probability of the hypothesis given the training data. But for Bayesian belief revision to actually lead to good predictions mustn't it be true that the prior is (approximately) correct? But what would this mean? We have argued above that a general purpose programming language such as C++ yields a universal bias. In a general purpose language one can represent any computable prediction rule without much increase in the number of bits required relative to a specialized language. But is the prior probability corresponding a universal programming language "correct" as a prior belief?

It turns out that there is a simple generalization of Hoeffding's confidence interval (1.1) which can be used to justify learning. Suppose that we are interested in classification. For example, in determining if a patient has cancer on the basis of a blood test. We assume a fixed but unknown probability distribution over patients. We also assume training data consisting of patients sampled

¹More general priors are often called "weaker" priors. We avoid this terminology here.

independently from this distribution with a blood test and a definitive diagnosis for whether the patient had cancer or not. Each of the infinitely many C++ predicates on word blood tests then has a measurable error rate on the training data. Each C++ predicate also has a true error rate — a probability of being wrong when a patient is sampled from the distribution. We can generalize (1.1) to give a confidence interval for each rule. More specifically, with probability at least $1 - \delta$ over the draw of n training examples we have that all rules have true error rates in the following confidence intervals where $\mathbf{err}(f)$ is the true error rate of f , $\widehat{\mathbf{err}}(f)$ is the fraction of training points on which f makes an error, and $|f|$ is the length of the rule f in bits.

$$\mathbf{err}(f) = \widehat{\mathbf{err}}(f) \pm \sqrt{\frac{(\ln 2)|f| + \ln(2/\delta)}{2m}} \quad (1.2)$$

It is possible to give tighter but less intuitive confidence intervals. For $\delta = .05$ there is at least a 95% chance that all of these infinitely many confidence intervals hold simultaneously. A confidence interval of the form in (1.2) becomes tight when $n \gg |f|$. This is the justification for saying that, for example, a universal bias is sufficient to overcome the poverty of the stimulus for learning English grammar. The number of bits that it takes to define universal grammar is presumably small compared to the number of bits required to specify an English dictionary. If we can give a confidence interval for the error rate of each rule, we can then select the rule with the strongest guarantee — the smallest upper bound on its actual error rate. Confidence intervals of this form are discussed in more detail in chapter 2.

1.6 Continuous Rule Parameters

Most learning systems use prediction rules involving numerical parameters. Many learning systems do nothing other than tune the parameters. But one should not underestimate the power of parameter tuning. A large neural network can simulate any Boolean circuit of roughly the same size simply by tuning the parameters of the circuit. A programmable gate array is a single big circuit which is “programmed” by setting parameters. A simpler example is a linear classifier. A linear classifier computes a weighted sum of feature values and then compares this sum with a threshold to determine whether to predict, say, cancer or not-cancer. It is possible to design an infinite feature space, and an appropriately general learning bias, such that a linear classifier over this set of features is universal — it can information theoretically learn any computable function with an amount of training data proportional to the length of the function.

Continuous parameters are easily handled within a Bayesian framework. One simply defines a prior probability mass density on the set of assignments of values to parameters. This can be done even for a linear classifiers with an infinite number of parameters. A Gaussian process is an example of such a prior. Given training data one can then compute a posterior mass density on

assignments of values to parameters. The learning algorithm can then set the parameters to the most probable setting given the training data. This is called the maximum a-posteriori (MAP) parameter setting. An alternative to using the MAP parameters settings is to compute the most probable output class by averaging over (integrating over) all possible parameter settings weighted by the posterior probability. This is called rule (or model) averaging.

But Bayesian methods have only a superficial notion of correctness — belief revision is correct by definition. Frequentist statements such as (1.1) provide a more sophisticated notion of correctness — a guarantee on true error rates assuming only the existence of some independently sampled data distribution. It is natural to ask how one can handle continuous parameters from a frequentist perspective.

For a rule with a finite number of parameters we can simply represent each parameter with a 32 bit floating point number. A rule, including a particular parameter setting, can then be written with a finite number of bits and we can apply (1.2). The smaller the number of bits we use for each parameter, the smaller the length of the rule and the tighter the confidence interval. One could then tune the precision of each parameter so as to minimize the upper bound on the confidence interval. This approach can in principle be used for very high dimensional linear classifiers provided that we represent a parameter setting by a list of pairs each which pairs a feature with a weight. Features not on the list are assigned weight zero. Such a rule can be written using $bN(1 + \log M)$ bits where b is the precision (number of bits) per weight, N is the number of nonzero weights, and M is the total number of available features. This can be generalized to an infinite number of available features if we allow different features to be named with different length bit strings.

While treating parameters as finite precision numbers might be reasonable, it is not typically done in current systems. Furthermore, treating parameters with finite precision numbers does not seem to handle certain commonly used biases. In particular it is common to use infinite dimensional Gaussian priors. This is essentially what is done in a nonlinear SVM, one of the most common machine learning mechanisms. As we will see in chapter 6, we can think of a nonlinear SVM as a linear classifier with an infinite number of features and trained relative to a prior probability (or learning bias) in which each feature is taken to have an independent Gaussian prior probability density. It does not seem possible to represent, or even approximate, an infinite dimensional Gaussian prior by assigning finite precision values for each parameter and measuring the bit length of the represented rule.

It turns out that the use of a Gaussian Prior for a high dimensional linear classifier can be understood from a frequentist perspective in terms of confidence intervals associated with “posterior” distributions. There is a generalization of (1.2) which gives a confidence interval for all possible “posterior weightings” over the space of rules. More specifically, let P be any prior measure (distribution or density) on classification rules. For any measure Q on rules (possibly different

from P) we define $\text{err}(Q)$ to be the expected value of $\text{err}(f)$ when f is drawn from Q . Similarly $\widehat{\text{err}}(Q)$ is the expected error rate $\widehat{\text{err}}(f)$ of a rule drawn from Q as measured on the training data. We then have that with probability at least $1 - \delta$ the following confidence intervals hold simultaneously for all posterior weightings (measures) Q on the rules.

$$\text{err}(Q) = \widehat{\text{err}}(Q) \pm \sqrt{\frac{KL(Q, P) + \ln\left(\frac{2(m+1)}{\delta}\right)}{2m}} \quad (1.3)$$

As with (1.2), tighter but less intuitive confidence intervals can be given. Using (1.3) with a Gaussian prior it is possible to derive frequentist guarantees for infinite dimensional support vector machines trained with finite training data. The details are presented in chapter 6.

1.7 Optimization

Most approaches to machine learning involve optimization. For example, one can consider the “simple” learning algorithm which selects the rule f minimizing the upper bound on true error in the interval given by (1.2). While (1.2) gives a natural optimization problem, for most rule sets this optimization problem is intractable. The issue of computational tractability, at least as related to the practice of machine learning, has focused on the formulation of convex objective functions and convex optimization. The formulation of a soft SVM (one involving hinge loss) is motivated mostly by the fact that it yields a convex optimization problem. For finite dimensional SVMs, or SVMs trained on finite samples, the soft SVM objective function is almost certainly “wrong” in the sense that optimizing a different objective function would yield rules with lower error rates. But the “right” objective function, or at least the ones known to have better generalization guarantees, are not convex and appear to lead to computationally intractable optimization problems.

The above discussion focuses on the optimization problems that arise in learning — optimization problems that define the search for a good rule or parameter setting. But optimization also arises in using a rule once it has been learned. Consider the machine translation problem — the problem of translating one human language, say French, into another, say English. It was mentioned above that this is one of the most prominent applications of machine learning today — state of the art systems are learned from a data base of translation pairs. The results of learning is a function f (system might be better word) such that for any French sentence x , we have that $f(x)$ is the (purported) English translation produced by f . A system which produces a complex output, such as word string, is sometimes called a *decoder*. We say that a translation system “decodes” French into English. A speech recognition system decodes a sound wave into a word string. Decoders generally define the decoding as an optimization problem of the following form where $E(x, y)$ is often called an

“energy”.

$$f(x) = \underset{y}{\operatorname{argmin}} E(x, y) \quad (1.4)$$

Machine learning researchers generally use the term “inference” to mean the process of computing the value $f(x)$ of a decoder f by solving an optimization problem of the form (1.4). The theory and practice of convex (and non-convex) optimization play a central role in both learning and inference.

1.8 Science versus Engineering

There are two notions of “science” in machine learning. The first involves learning theory — what statements or guarantees can be proved for learning systems. Such statements can be either statements of statistical inference, such as (1.2), or statements of computational complexity such as the statement that it is NP-hard to find a minimum-error linear classifier. The second is related to neuroscience. What learning algorithms actually exist in natural computational systems (nervous systems)? This is an empirical rather than a mathematical question. This empirical question is very difficult to approach in any direct way at this time. A more approachable question is what algorithms are capable of replicating the performance observed in nature? This latter question seems more approachable. However, the study of what algorithms are capable of effective learning is fundamentally a question of engineering. It cannot be answered entirely by the mathematics of learning theory. Many NP hard problems are easy in practice and many polynomial time algorithms are impractical. One can at least argue that the engineering of learning systems is a critical part of the understanding of learning as a computational phenomenon. In order to understand neurological learning systems it seems important to understand the general phenomenon of learning.

At the present time learning theory does not allow learning systems to be designed in the same way as bridges or dams. A bridge or dam can be designed directly from fundamental equations of mechanical and hydrodynamic systems. Building a learning system for visual object detection, speech recognition, or machine translation requires tremendous trial and error experimentation. Of course this experimentation should be guided whatever theoretical insight is possible.

1.9 Theory, Recipes, and Case Studies

The material in this book can be roughly divided into three types: theorems, recipes, and case studies. The theorems given here are typically about statistical inference or computational complexity. The recipes are learning mechanisms, often called “machines” as in “support vector machine” or “Boltzmann machine”. Sometimes, as in the case of (soft) SVM, there is little clear theoretical

justification for the precise structure of the machine. The case studies provide a context in which to evaluate, at these for these cases, which theorems and recipes seem important to understanding the phenomenon of learning. We consider the following four case study problems.

- **Visual Object Detection.** Given natural photographs (images from the web), and a target object class such as “person” or “car”, we want to build a system that will identify the objects of that type in the photographs and give their approximate locations. We consider the case where training data is given as pictures annotated with bounding boxes for the objects of the desired class.
- **Open Domain Continuous Speech Recognition.** Given a sound wave of human speech recover the sequence of words that were uttered. Training data is taken to consist of sound waves paired with transcriptions such as in closed caption television together with a large corpus of text not associated with any sound waves.
- **Natural Language Translation.** Given a sentence in one language translate it into another language. The training data consists of a set of translation pairs where each pair consists of a sentence and its translation.
- **The Netflix Challenge.** Given previous movie ratings of an individual predict the rating they would give to a movie they have not yet rated. The training data consists of a large set of ratings where each rating is a person identifier, a movie identifier, and a rating.

Each of these case study problems has been intensively worked on. In each case a large number of approaches have been found to be inferior to the current state of the art. The hope is that by examining the state of the art methods in problems that have been extensively studied one might be able to extract some insight into what methods or principles are important in practice. Each case study seems to have a moral. The object detection case study shows the power of SVMs and the significance of latent information. The speech recognition case study shows the power of multiple sources of training data, the importance of statistical inference in decoding, and the utility of EM for alignment. The machine translation case is surprisingly similar to speech recognition case study has many of the same morals. This serves to reinforce those morals. The netflix challenge shows the power of meta-systems and cue combination (at least in this application) together with the value of a variety of particular methods used in the combination.

The theory and practice of machine learning is still undergoing rapid evolution. Presumably the state of the art in the above case study problems will improve over time and the learning recipes will change. However, it also seems possible that the best established theorems and recipes of current machine learning are immortal — they will continue to be taught as long as the phenomenon of learning remains of interest.

Chapter 2

Occam's Razor

Entia non sunt multiplicanda praeter necessitatem.
(Entities should not be multiplied beyond necessity.)

William of Occam, c. 1320

Make everything as simple as possible, but not simpler.

Albert Einstein

Over the centuries Occam's razor has come to be interpreted as stating that if two theories explain the data equally well, the simpler one is preferable. Although we seem to understand the notion of simplicity at an intuitive level, to develop a theoretical understanding of Occam's razor we must formally examine both the notion of "explanation" and the notion of "simplicity". Here we develop a formal approach within the framework of classifier learning. A classifier takes an input, such as a blood sample, and returns a binary prediction (a predicted label) such as a prediction as to whether the patient has cancer. Classifier learning is an over simplification of the general phenomenon of learning. Classifier learning is not intended as a model of the scientific process, although presumably some form of Occam's does apply to science generally. Classifier learning is, however, a good place to begin to develop a formal understanding of learning.

In the classification problem we assume training data $(x_1, y_1), \dots, (x_n, y_n)$ with y_t being one of the two values 1 or -1 . We will assume that these training pairs have been drawn independently from a distribution (or density) ρ . Our objective is to construct a predictor f such that when a new pair (x, y) is drawn from this same data source we have a high probability that $f(x)$ correctly predicts y , i.e., $f(x) = y$. The probability over a fresh draw that $f(x) \neq y$ is called the error rate of f or the generalization error (to distinguish it from the

training error — the error rate on the training data). Our goal is to take the training data (past data) and construct a predictor f which has a low error rate on new (future) data.

A fundamental problem in machine learning is over-fitting. For any training data set we can always find a predictor with zero error on the training data. For example, the predictor can be a simple table giving the value of y for each input in the training data and predicting a random value if the input did not occur in the training data. Such a predictor will have no errors on the training data but will make many errors on new data. However, if the rule is simple — if it can be written down with a number of bits small compared to the number of training pairs, then its error rate on the training data (the training error) is a good predictor of its error rate on new data (the generalization error). This claim is proved formally in section 2.2 and proved again with a more refined theorem in section 2.6.

Perhaps the most interesting formulation of Occam's razor in this chapter is (2.21) which states a linear trade-off between accuracy and simplicity and where the relative weight of these two terms (the regularization constant) has been derived analytically from frequentist confidence intervals. In practice, the relative weight of training accuracy and simplicity is often determined empirically. However, machine learning packages such as SVMlight provide default settings for regularization parameters in apparent agreement with the analytic value in (2.21).

2.1 The Hoeffding Bound and the Hoeffding Confidence Interval

The Chernoff bound can be viewed as a variant of the law of large numbers. The law of large numbers concerns the process of estimating an average by numerically computing the average of a sample. For example, we can estimate the average height of an American adult male by sampling 100, or 1000, adult males at random and taking the average height of the sample. Different samples will give different sample averages but the sample averages should be near the true average and as the sample gets larger the sample average should become closer to the true average. More specifically, suppose we take a sample x_1, \dots, x_n where each x_i is drawn independently from a fixed distribution (or density) ρ and suppose that we estimate the mean of ρ as follows.

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i$$

We now have that $\hat{\mu}$ is a random variable — different samples will yield different sample averages. So there is a probability distribution (or density) for the quantity $\hat{\mu}$. Informally, the law of large numbers states that as the sample

2.1. THE Hoeffding Bound and the Hoeffding Confidence Interval 17

size n becomes large, the probability distribution of $\hat{\mu}$ becomes approximately Gaussian. The variance of this Gaussian will be σ^2/n where σ^2 is the variance of x . In particular we have the following (very approximate) relations for $\epsilon > 0$.

$$p(\hat{\mu}) \approx \frac{1}{(\sigma/\sqrt{n})\sqrt{2\pi}} e^{-\frac{(\hat{\mu}-\mu)^2}{2\sigma^2/n}}$$

$$p(\hat{\mu} \leq \mu - \epsilon) \approx e^{-\frac{n\epsilon^2}{2\sigma^2}} \quad (2.1)$$

Relation (2.1) is approximate. However if $x \in [0, 1]$ then we have $\sigma^2 \leq 1/4$ and, furthermore, we have the following rigorous inequality.

$$p(\hat{\mu} \leq \mu - \epsilon) \leq e^{-2n\epsilon^2} \quad (2.2)$$

(2.2) is called Hoeffding's inequality. An equivalent statement (applying (2.2) to $1 - x$) is the following.

$$p(\hat{\mu} \geq \mu + \epsilon) \leq e^{-2n\epsilon^2} \quad (2.3)$$

We can combine (2.2) and (2.3) to get a confidence interval. To do this we use the union bound. The union bound states that for any two statements Φ and Ψ we have that $P(\Phi \vee \Psi) \leq P(\Phi) + P(\Psi)$. This gives the following.

$$\begin{aligned} P(|\hat{\mu} - \mu| \geq \epsilon) &= P(\hat{\mu} \leq \mu - \epsilon \vee \hat{\mu} \geq \mu + \epsilon) \\ &\leq P(\hat{\mu} \leq \mu - \epsilon) + P(\hat{\mu} \geq \mu + \epsilon) \\ &\leq 2e^{-2n\epsilon^2} \end{aligned} \quad (2.4)$$

(2.4) can be converted to a confidence interval. Given a confidence parameter $\delta \in (0, 1)$ we have that with probability at least $1 - \delta$ over the draw of the sample the following holds.

$$|\hat{\mu} - \mu| \leq \sqrt{\frac{\ln \frac{2}{\delta}}{2n}} \quad (2.5)$$

To prove that (2.5) holds with probability at least $1 - \delta$ we simply calculate the probability that it is not true using (2.4).

$$P\left(|\hat{\mu} - \mu| \geq \sqrt{\frac{\ln \frac{2}{\delta}}{2n}}\right) \leq \delta$$

We can rewrite (2.5) as follows.

$$\mu = \hat{\mu} \pm \sqrt{\frac{\ln \frac{2}{\delta}}{2n}} \quad (2.6)$$

We will call (2.6) the Hoeffding confidence interval. If we set $\delta = .05$ then we get the 95% confidence interval.

2.2 A First Occam Confidence Interval

The square root Occam bound is perhaps the simplest generalization guarantee and is the starting point of our analysis. For this theorem we consider a countable class \mathcal{H} of binary predictors $h : \mathcal{X} \rightarrow \{-1, 1\}$. We will call $h \in \mathcal{H}$ a hypothesis. We will assume a fixed language, or code, in which each hypothesis can be named. Let $|h|$ be the number of bits needed to name the hypothesis h . Let $\text{err}(h)$ and $\widehat{\text{err}}(h)$ be defined as follows where $I[\Phi]$ is 1 if Φ is true and 0 otherwise.

$$\begin{aligned}\text{err}(h) &\equiv \mathbb{P}_{\langle x, y \rangle \sim \rho} [h(x) \neq y] \\ \widehat{\text{err}}(h) &\equiv \frac{1}{N} \sum_{t=1}^n I[h(x_t) \neq y_t]\end{aligned}$$

The Occam bound states that for IID draws of n training pairs, and for $\delta > 0$, with probability at least $1 - \delta$ over the draw of the training data D , we have the following simultaneously for all $h \in \mathcal{H}$.

$$|\text{err}(h) - \widehat{\text{err}}(h)| \leq \sqrt{\frac{(\ln 2)|h| + \ln \frac{2}{\delta}}{2n}} \quad (2.7)$$

The important point is that, with high probability over the draw of the sample, the infinitely many confidence intervals given by (2.7) (for the different hypotheses h) all hold simultaneously. The Proof is a simple application of the two-sided Hoeffding inequality (2.4) and the following two additional inequalities.

- **Union Bound:** $P(\exists x \Phi[x]) \leq \sum_x P(\Phi[x])$
- **Kraft Inequality:** $\sum_h 2^{-|h|} \leq 1$

The union bound is a simple generalization of the observation that $P(\Phi \vee \Psi)$ can be no larger than $P(\Phi) + P(\Psi)$. The Kraft inequality holds for prefix codes — a set of code words where there is a well defined way of determining when a code word ends, or more formally, where no code word is a proper prefix of any other code word. Null terminated byte strings have a well defined termination condition (when a null byte is found) and therefore form a prefix code. To prove the Kraft inequality consider randomly generating one bit at a time and stopping when you have a (complete) code word. Then the probability of stopping at the code word for h equals $2^{-|h|}$. Therefore the sum over h of $2^{-|h|}$ cannot exceed 1.

Given the two-sided Hoeffding inequality (2.4), the union bound, and the Kraft inequality we can prove (2.7) by considering the case where some confidence interval fails to hold. We need to show that the probability that some

interval fails to hold is no larger than δ . We will call h “bad” (for a given sample) if the confidence interval in (2.7) fails to hold for h .

$$\text{bad}(h) \equiv \left[|\text{err}(h) - \widehat{\text{err}}(h)| \geq \sqrt{\frac{(\ln 2)|h| + \ln \frac{2}{\delta}}{2n}} \right]$$

We now have the following proof that the probability that a bad h exists is no larger than δ .

$$\begin{aligned} P(\text{bad}(h)) &\leq 2e^{-2n\epsilon^2} \\ &= \delta 2^{-|h|} \\ P(\exists h \text{ bad}(h)) &\leq \sum_h \delta 2^{-|h|} \\ &= \delta \sum_h 2^{-|h|} \\ &\leq \delta \end{aligned}$$

2.3 Selecting a Hypothesis

We can rewrite (2.7) as follows.

$$\text{err}(h) = \widehat{\text{err}}(h) \pm \sqrt{\frac{(\ln 2)|h| + \ln \frac{2}{\delta}}{2n}}$$

We can select a hypothesis h by minimizing the upper bound on $\text{err}(h)$. More formally we can select the formula \hat{h} defined as follows.

$$\hat{h} = \underset{h}{\text{argmin}} \widehat{\text{err}}(h) + \sqrt{\frac{(\ln 2)|h| + \ln \frac{2}{\delta}}{2n}} \quad (2.8)$$

Note that \hat{h} is the result of optimizing a trade off between its ability to explain the training data (as measured by $\widehat{\text{err}}(h)$) and its simplicity (as measured by $|h|$). Hence, this rule for selecting \hat{h} is a formal version of Occam’s razor.

This version of Occam’s razor has two major issues. First, for many (most?) languages for stating hypotheses the optimization problem defined by (2.8) is computationally intractable (NP hard) (Problem: show that deciding whether there exists an h for which the right hand side of (2.8) is below a specified value is in NP.) We will ignore this for now and focus instead on information theoretic aspects of (2.8) — we will analyze (2.8) assuming infinite computational resources. A second, purely information theoretic issue, is that (2.7) is

actually a weak confidence interval — a tighter upper bound on generalization error is given in section 2.6. The use of a weak confidence interval leads to over-regularization — the selected formula \hat{h} is too simple.

2.4 An Example: Boolean Formulas

As an example we consider objects with Boolean features. More specifically, we write $x.f$ for the value of feature f of object x and assume $x.f \in \{0, 1\}$. We consider Boolean expressions constructed from feature names and the operations of negation \neg , disjunction \vee and conjunction \wedge . For example, we might have $\neg f \vee (\neg g \wedge w)$ where f , g , and w are feature names. For a Boolean formula Φ and an object x we define $\Phi(x) \in \{0, 1\}$ in the standard way. For example if $x.f$ is 1 and $x.g$ is 0 then $(f \wedge \neg g)(x) = 1$. It is possible to design a prefix code for Boolean formulas under which we have the following where n is the number of nodes in Φ , ℓ is the number of leaf nodes in Φ and d is the number of Boolean features.

$$|\Phi| = 2n + \lceil \log_2 d \rceil \ell \quad (2.9)$$

This code can be constructed by first representing formulas in prefix notation (also called Polish notation) in which the formula is represented as a string where operators precede their arguments. For example we can represent $\neg f \vee (g \wedge w)$ as $\text{wedge} \neg f \vee w g$. We then represent each symbol in the prefix notation symbol string with a bit string. We can use "11" for \wedge , "01" for \vee , "10" for \neg , and "00" followed by $\lceil \log_2 d \rceil$ bits for a feature name. It is possible to show that the resulting code is a prefix code — one can determine when a formula code ends.

We can also handle an infinite number of Boolean features. Of course we can not represent an object with an infinite number of features as a simple data structure storing the value of each feature. But we could represent such an object as a finite list of feature-value pairs where the set of potential features is infinite and the length of feature-value pair list is unbounded. What is needed in this case is a way of coding each feature name as a finite bit string (such as a character string). We can let $|f|$ be the number of bits needed to name feature f . If Φ is a Boolean formula (over an infinite number of features), and ℓ is a leaf node of Φ , then we can write $|\ell|$ for the number of bits in the code for the feature name at the leaf ℓ . In this case we get the following variant of (2.9) where n is the number of nodes in Φ and ℓ ranges over the leaf nodes of Φ .

$$|\Phi| = 2n + \sum_{\ell \in \Phi} |\ell| \quad (2.10)$$

2.5 Probability Codes

Let P be any fixed probability distribution on the hypothesis set \mathcal{H} . We define the probability code length $|h|_P$ of hypothesis h under probability distribution

P as follows.

$$|h|_P = \log_2 \frac{1}{P(h)}$$

Note that for this definition of $|h|$ we have the Kraft inequality $\sum_h 2^{-|h|} = \sum_h P(h) = 1$. Since the Kraft inequality holds, the interval theorem (2.7) holds as well even though we have no explicit code for h as a bit string.¹ By abuse of terminology we will often refer to a probability distribution on the hypothesis space \mathcal{H} as a “probability code” for the rules in \mathcal{H} and refer to $|h|_P$ as the (fractional) number of bits in the code for h .

Designing a probability code is often easier and gives better results than designing a bit code. For example, we can define a probability distribution over Boolean formulas by defining a stochastic process for generating formulas. First we select one of \vee , \wedge , \neg or “feature” with equal probability. If we select an operator (\vee , \wedge or \neg) then we recursively generate the arguments for the operator. If we select “feature” then we pick a feature giving equal probability to each feature. We then get the following where again n is the number of nodes and ℓ is the number of leaves.

$$|\Phi| = \log_2 \frac{1}{P(\Phi)} = 2n + (\log_2 d)\ell \quad (2.11)$$

The difference between (2.11) and (2.9) is simply that in (2.11) we do not need to round up $\log_2 d$ to an integral number of bits.

2.6 A Tighter Occam Bound

It turns out that Hoeffding’s inequality (2.2) can be greatly improved for the case where the error rate is small. Very small error rates typically occur in highly unbalanced classification problems, such as visual object detection, where the vast majority of inputs are negative (background images). In this case the error rate is very low — true negatives, which are the vast majority of inputs, are almost always correctly classified. In these cases a more meaningful notion of performance is precision (the fraction detections which are true positives) and recall (the fraction of true positives that are detected). Precision and recall are meaningful even if true positives are very rare.

For a random variable $x \in [0, 1]$ it is useful to observe that $\sigma^2 \leq \mu(1 - \mu)$ where μ and σ are the mean and standard deviation of x respectively. The worst case (largest variance) is when x is Bernoulli, i.e., $x \in \{0, 1\}$, in which case we have $\sigma^2 = \mu(1 - \mu)^2 + (1 - \mu)\mu^2 = \mu(1 - \mu) \leq \mu$. So smaller μ leads to smaller variance which leads to a tighter confidence interval. More precisely, let $\hat{\mu}$ be

¹Shannon’s source coding theorem gives a way of converting any probability code into an actual (block) code using bit strings with only minor overhead relative to $|h|_P$ caused by the discrete nature of bit strings. But here we do not require an actual bit string code.

the mean of a sample of n draws of a random variable $x \in [0, 1]$. We have the following rigorous bound.

$$p(\hat{\mu} \leq \mu - \epsilon) \leq e^{-\frac{n\epsilon^2}{2\mu}} \quad (2.12)$$

The bound (2.12) implies that with probability at least $1 - \delta$ we have the following.

$$\mu \leq \hat{\mu} + \sqrt{\mu c} \quad (2.13)$$

$$c = \frac{2 \ln \frac{1}{\delta}}{n}$$

To prove that (2.13) holds with probability $1 - \delta$ it suffices to use (2.12) to show that the probability that it fails to hold is no larger than δ . The problem with (2.13), of course, is that it bounds μ in terms of $\hat{\mu}$ and μ . But (2.13) implies the following for the case of $\mu \geq \hat{\mu}$.

$$\begin{aligned} \mu &\leq \hat{\mu} + \sqrt{\mu c} \\ (\mu - \hat{\mu})^2 &\leq \mu c \\ \mu^2 - 2\mu\hat{\mu} - \hat{\mu}^2 &\leq \mu c \\ \mu^2 - (c + 2\hat{\mu})\mu - \hat{\mu}^2 &\leq 0 \\ \mu &\leq \frac{c + 2\hat{\mu} + \sqrt{(c + 2\hat{\mu})^2 - 4\hat{\mu}^2}}{2} \\ &= \frac{c + 2\hat{\mu} + \sqrt{c^2 + 4c\hat{\mu}}}{2} \\ &\leq \frac{c + 2\hat{\mu} + \sqrt{c^2} + \sqrt{4c\hat{\mu}}}{2} \\ &= \hat{\mu} + \sqrt{\hat{\mu}c} + c \end{aligned}$$

We have now shown that with probability at least $1 - \delta$ over the draw of the sample we have the following upper bound on μ based on the sample mean $\hat{\mu}$.

$$\mu \leq \hat{\mu} + \sqrt{\hat{\mu}c} + c \quad (2.14)$$

$$c = \frac{2 \ln \frac{1}{\delta}}{n}$$

Again we consider a hypothesis space \mathcal{H} where we have a code length $|h|$ for each $h \in \mathcal{H}$ satisfying the Kraft inequality $\sum_h 2^{-|h|} \leq 1$. Using a proof similar to the proof of (2.7), and then a derivation similar to the derivation of (2.14), one can prove that with probability at least $1 - \delta$ over the draw of n training pairs (x_i, y_i) the following holds simultaneously for all $h \in \mathcal{H}$.

$$\mathbf{err}(h) \leq \widehat{\mathbf{err}}(h) + \sqrt{\widehat{\mathbf{err}}(h)c(h)} + c(h) \quad (2.15)$$

$$c(h) = \frac{2((\ln 2)|h| + \ln \frac{1}{\delta})}{n} \quad (2.16)$$

It is easy to show that if $\widehat{err}(h) \leq \epsilon$ and $c(h) \leq \epsilon$, then the left hand side of (2.15) is at most 3ϵ . Under the same conditions the left hand of (2.7) goes as $\sqrt{\epsilon}$ which can be much larger when ϵ is small. In highly unbalanced problems, such as a rare event detection problem, per sample error rates can be very low because most inputs are easily dismissed background events. In such cases (2.15) is far superior to (2.7).

2.7 Linearized Regularization

We can use (2.15) to select a hypothesis with an appropriate trade off between error rate and complexity as follows.

$$\hat{h} = \operatorname{argmin}_h \widehat{\mathbf{err}}(h) + \sqrt{\widehat{\mathbf{err}}(h)c(h)} + c(h) \quad (2.17)$$

To analyze this selection rule let $B(h)$ be defined to the quantity optimized in (2.17).

$$B(h) = \widehat{\mathbf{err}}(h) + \sqrt{\widehat{\mathbf{err}}(h)c(h)} + c(h)$$

Because the arithmetic mean bounds the geometric mean ($\sqrt{xy} \leq (x+y)/2$) we get the following.

$$\widehat{\mathbf{err}}(h) + c(h) \leq B(h) \leq \frac{3}{2}(\widehat{\mathbf{err}}(h) + c(h)) \quad (2.18)$$

This shows that $B(h)$ is never far from $\widehat{\mathbf{err}}(h) + c(h)$. So we can consider solving a simpler “linearized” optimization problem as follows.

$$\tilde{h} = \operatorname{argmin}_h \widehat{err}(h) + c(h) \quad (2.19)$$

We now have that (2.18) ensures that \tilde{h} is never much worse than \hat{h} , at least as measured by the generalization guarantee $B(h)$. In particular we have the following.

$$\begin{aligned} \mathbf{err}(\tilde{h}) &\leq B(\tilde{h}) \\ &\leq \frac{3}{2}(\widehat{\mathbf{err}}(\tilde{h}) + c(\tilde{h})) \\ &\leq \frac{3}{2}(\widehat{\mathbf{err}}(\hat{h}) + c(\hat{h})) \\ &\leq \frac{3}{2}B(\hat{h}) \end{aligned} \quad (2.20)$$

So at the cost of at most a 50% degradation of the bound value, we can use the linear regularizer (2.19) rather than (2.17). By multiplying the quantity being optimized by n and ignoring terms that do not depend on h we can rewrite (2.19) as follows.

$$\tilde{h} = \operatorname{argmin}_h \sum_i I[h(x_i) \neq y_i] + 2(\ln 2)|h| \quad (2.21)$$

So starting from the Angluin-Valiant bound (2.12) we have derived a linear trade off between the absolute number of errors and the hypothesis complexity in bits with a roughly equal weighting of the two terms ($2 \ln 2$ is about 1.4). This equal weighting is independent of the sample size n and independent of where the optimum occurs. We should expect that at the optimum the two terms are roughly balanced and therefore that $|\tilde{h}|$ (in bits) is approximately equal to the absolute number of errors made by \tilde{h} .

In practice it is generally preferable to allow more flexibility in the relative weight in the trade off between $\widehat{\text{er}}(h)$ and $|h|$. More generally we can use the following where λ is a parameter of the learning system.

$$\tilde{h} = \operatorname{argmin}_h \sum_i I[h(x_i) \neq y_i] + \lambda|h| \quad (2.22)$$

Although (2.21) suggests $\lambda = 2 \ln 2$, whose value can be selected by measuring generalization performance for different values of λ . This is typically done by removing some of the training to form a hold out set. For various values of λ one learns a classifier \tilde{h} using (2.22) and then test \tilde{h} on the hold out set to estimate the performance when training at this setting of λ . This is generally referred to as hold-out tuning.

2.8 Codes for Numerical Parameters

We are often interested in prediction rules involving numerical parameters. At a theoretical level numerical parameters are probably best treated by the methods described in chapter 6. However, it is worth noting that numerical parameters can also be treated naively by coding parameter values with finite precision bit strings. For the learning rule given by (2.22) to work well, especially with $\lambda = 2 \ln 2$, it is probably important that numerical parameters be coded with variable precision codes allowing extremely efficient coding of default values. Here we formulate some parameter codes that might plausibly work with (2.22).

We can represent a number in the open interval $(0, 1)$ by a bit string preceded by a decimal point. For example $.1 = 1/2$, or $.01 = 1/4$, or $.1101 = 1/2 + 1/4 + 1/16 = 13/16$. To ensure that we have a prefix code we define a stochastic process for generating such representations and then use the probability code associated with this process. More specifically, with probability $1/2$ we return the fraction $1/2$. Otherwise we generate $x \in (0, 1)$ recursively and, with equal probability, return either $x/2$ or $1/2 + x/2$. With probability 1 this process generates a number represent by a decimal point followed by a finite bit string. For any such x we can express $|x| = \log_2 1/P(x)$ as follows where $L(x)$ is the length of the binary representation of x .

$$|x| = 2L(x) - 1 \quad (2.23)$$

We get that $|1/2|$ is one bit; $|1/4|$ and $|3/4|$ are both three bits; $|k/8|$ is five bits and so on.

If we have prior knowledge of a plausible range for a parameter we can represent a parameter α as follows where $x \in (0, 1)$ is as described above.

$$\alpha(x) = \alpha_{\min} + (\alpha_{\max} - \alpha_{\min})x \quad (2.24)$$

Here we take α_{\min} and α_{\max} as parameters of the code. This gives the following complexity for the parameter $\alpha(x)$.

$$|\alpha(x)| = |x| \quad (2.25)$$

We then get that the midpoint $(\alpha_{\min} + \alpha_{\max})/2$ can be coded with one bit. More sophisticated parameter codes can be specified with nonlinear functions of the “random” value $x \in (0, 1)$.

Positive integers, such as the length of a list, can be coded with a probability code where the probability of a length k is defined as follows.

$$P(k) = \frac{1}{k(k+1)}$$

$$|k| = \log_2 k(k+1) \approx 2 \log_2 k \quad (2.26)$$

It is possible to show that under this definition we have that the sum of $P(k)$ for all positive k equals 1 and hence this is a valid probability code.

Chapter 3

Decision Trees and Sparse Linear Classifiers

Here we consider two standard machine learning recipes: decision trees and sparse linear classifiers. Decision trees, along with neural networks, are one of the oldest machine learning methods dating back to the 1960s. Sparse linear classifiers became popular in the machine learning community with the development of boosting in the 1990s as a method for learning them. Standard methods of learning decision trees or sparse linear classifiers use greedy optimization to select the next “test” or “feature” to incorporate into the rule. Boosting was originally motivated by something called the weak learning hypothesis. The weak learning hypothesis is an assumption which implies that greedy feature selection will always make progress. Here we view this assumption as largely unjustified and do not formally discuss it. Instead we base our theoretical understanding on the generalization bound (2.15) and the training equation (2.22). We informally assume that greedy feature selection is generally effective.

Decision trees and sparse linear classifiers are similar in that they are both “sparse” — each tree or linear classifier typically involves only a small fraction of the allowed tests or features. Although they are both sparse classifiers, they represent significantly different approaches to rule formation. A decision tree is a “hard” logical rule. It is equivalent to a logical formula (although perhaps more compactly represented as a tree). While a Boolean formula is a logical rule, a sparse linear classifier computes a “soft” score that adds together weighted evidence. There is a general feeling in the machine learning community that evidence weighing rules, like linear classifiers, provide a better learning bias than logical rule classifiers like Boolean formulas. This feeling is based primarily on empirical experience in machine learning applications. The power of linear classifiers seems related to a common sense intuition that it is often wiser to weigh evidence than to apply logical rules. The power of linear classifiers, and perhaps the wisdom of weighing evidence, may come from some combination of

generality in the sense of section 1.4 and some notion of approximability in the sense that many functions are well approximated by weighted summations. But in any case, adding together weighted evidence seems, empirically, to be a good way of making decisions.

It should be noted here that while (2.15) and (2.22) provide a reasonable framework for learning decision trees and sparse linear classifiers, the theory of sparse linear classifiers is usually justified in terms of the margin bounds. Margin bounds for linear classifiers are in chapter 6.

3.1 Decision Trees

We assume a set (or space) X of possible inputs. For example we could take \mathcal{X} to be the set of all possible results of a certain blood test where each result is a measured level of a certain set of proteins. We could denote the measured level of protein f in test results x by $x.f$ (as in the C programming language). A test on \mathcal{X} is a function Φ such that for $x \in \mathcal{X}$ we have that $\Phi(x) \in \{-1, 1\}$. For example for a protein f and fixed constant α we define the test $\Phi_{f,\alpha}$ as follows.

$$\Phi_{f,\alpha}(x) = \begin{cases} 1 & \text{if } x.f \geq \alpha \\ -1 & \text{otherwise} \end{cases}$$

In many applications we work with a highly restricted set of tests. For example, we might consider only tests of the form $\Phi_{f,\alpha}$. We assume a fixed language (code) in which the tests are expressed so that we have a well defined complexity $|\Phi|$ for each expressible test Φ .

A decision tree is a binary tree where every internal node is labeled with a test and every leaf node is labeled with truth value (either -1 or 1). A tree with root node labeled with f and with left and right subtrees T_L and T_R respectively will be written as $f ? T_L : T_R$ (by analogy with C conditional expressions). For example we might have the following tree.

$$\Phi_{f,2} ? (\Phi_{g,1/2} ? -1 : 1) : (\Phi_{h,-1} ? 1 : -1)$$

Each decision tree is itself a test and we write $T(x)$ for the value of T on input x . The value $T(x)$ can be defined by the following equations.

$$(\Phi ? T_L : T_R)(x) = \begin{cases} T_L(x) & \text{if } \Phi(x) = 1 \\ T_R(x) & \text{if } \Phi(x) = -1 \end{cases}$$

$$1(x) = 1$$

$$-1(x) = -1$$

3.1.1 Greedy Optimization of Log Loss

Most decision tree learning algorithms construct trees by repeatedly replacing a leaf with a branch. This is done in such a way as to greedily reduce some

measure of the quality of the tree. An obvious measure of quality is the error rate. But greedy reduction of error rate does not work well. To appreciate the difficulty consider a detection problem in which most inputs are “background” and labeled with -1. For concreteness suppose that only 1% of the labels are 1. Now suppose that there exists a feature f such that after branching on a test of f we have that one branch is has no 1s and the other branch has 30% 1s. This test has clearly made progress — although both branches should still be labeled -1, we have introduced a test that enriches the number of 1s from 1% to 30%. Although the error rate of the tree has not improved — the still always predicts -1 — another test on the 30% branch will presumably now improve the error rate. A less dramatic enrichment of the positive labels, from 1% to say 2% would also be quite useful. If the enrichment can be continued then we will eventually reduce the error rate. This gradual enrichment of the positives is the fundamental method used in construction of the Viola-Jones face detector described in chapter ??.

A common measure of the quality of a decision tree is the log loss of the tree on the training data. Consider training data $D = (x_1, y_1), \dots, (x_n, y_n)$. Let $\ell(T, x)$ be the leaf of the tree T which is reached on input x . This leaf has a number of positive instance and a number of negative instances in the training data defined as follows.

$$c(\ell, y, D) = \sum_{i=1}^n I[\ell(T, x_i) = \ell \wedge y_i = y]$$

We can now define $P(y|x; T, D)$ as follows.

$$P(y|x; T, D) = \frac{c(\ell(T, x), y, D)}{c(\ell(T, x), 1, D) + c(\ell(T, x), -1, D)}$$

This simply says that the probability of a label y for input x is the fraction of the training data D reaching the leaf $\ell(T, x)$ which has label y . We now define the log loss of a decision tree (on the training data) as follows.

$$L_{\log}(T, D) = \sum_{i=1}^n \ln \frac{1}{P(y_i|x_i; T, D)} \quad (3.1)$$

Minimizing log loss is equivalent to maximizing $\prod_{i=1}^n P(y_i|x_i; T, D)$. While the product of conditional probabilities may seem more natural than the negative sum of log probabilities, the negative sum of log probabilities (log loss) is analogous to error rate and often appears instead of error rates in additive training equations such as (2.22).

Introducing a branch reduce the log loss provided that it improves the segregation of positive from negative labels even if both branches have the same dominant label. In growing a tree we replace leaf nodes with branches. At each replacement one selects the test, among all available tests, which results in the largest reduction of log loss. Typically one restricts the tests to be very simple and does not worry about the complexity of the test.

3.1.2 Applying Occam's Razor

Here we consider how (2.22) might be applied to decision trees. Section 3.1.1 leaves open the question of when to stop growing the tree. Most decision tree learning algorithms first over expand the tree and then use some rule to prune it back. Typically there enough tests available to allow one to construct a zero loss tree — one in which every leaf has only positive or only negative training instances. With a sufficiently large number of independent features such a tree almost certainly can be found even if there is no statistical relationship between the features and the labels. To produce a tree that generalizes well the tree must be pruned.

We can prune a decision tree using the Occam's razor regularization specified by (2.22). To do this we must first define a code for decision trees so that we have a well defined complexity (in bits) for each tree. We can code a decision tree as a bit string as follows. First we write a bit to determine whether this is a leaf node or a branching node. If it is a leaf node we write one more bit to indicate the value of that node (-1 or 1). If it is branch node we first write a code for the test of the node and then (recursively) write code a code for the left branch and a code for the right branch. The total number of bits can then be written as follows where $\text{Branches}(T)$ is the set tests used in of branch nodes of the tree T , $\text{Test}(b)$ is the test used at branch node b , and $\text{Leaves}(T)$ is the set of leaves of T .

$$|T| = \left(\sum_{b \in \text{Branches}(T)} 1 + |\text{Test}(b)| \right) + 2|\text{Leaves}(T)| \quad (3.2)$$

Note that we have allowed different tests at the branch nodes to have different complexities. In some cases the tests often all have the same complexity which is $\log_2 M$ where M is the number of possible tests. For the blood test results described above, a test of the form $|\Phi_{f,\alpha}|$ is $\log_2 M + |\alpha|$ where M is the number of proteins and $|\alpha|$ is the complexity in bits of the numerical value α as given, perhaps, by (2.25).

We can now prune a tree starting at the leaves and working back toward the root. For a node both of whose children are leaves we must decide whether to keep the node or to replace it by a single leaf. The training equation (2.22) can be used to make this decision. We prune a branch b if the number of errors introduced by the pruning is no larger than $\lambda(3 + |\text{Test}(b)|)$. Often the default of $\lambda = 2 \ln 2$ works well for decision trees.

Rather than simply decide whether to prune or not, one might alternatively consider simplifications of the test. For example, one might simply $\Phi_{f,\alpha}$ by reducing the numerical precision of the parameter α . One could then select between pruning, simplifying, or leaving the branch unchanged based on the training equation (2.22).

3.2 Sparse Linear Classifiers

As with decision trees, we assume a set (or space) X of possible inputs. A feature on \mathcal{X} is a function Φ from such that for $x \in \mathcal{X}$ we have that $\Phi(x)$ is a number. For example, for blood test results we might define Φ_f and $\Phi_{f,2}$ as follows.

$$\begin{aligned}\Phi_f(x) &= x.f \\ \Phi_{f,\text{squared}} &= (x.f)^2\end{aligned}$$

In practice one often works with a highly restricted set of features such as the set of features of the form Φ_f and $\Phi_{f,\text{squared}}$. Note that a test is a special case of a feature with $\Phi(x) \in \{-1, 1\}$. So we could also include the tests $\Phi_{f,\alpha}$ defined in section 3.1 as features. We assume a fixed language (code) in which the features are expressed so that we have a well defined complexity $|\Phi|$ for each expressible feature Φ .

A linear discriminator L_k consists of a set of features $\Phi_1, \Phi_2, \dots, \Phi_k$ plus a set of weights w_1, w_2, \dots, w_k and has an associated score $L_k(x)$ on input x defined as follows.

$$L_k(x) = \sum_{i=1}^k w_i \Phi_i(x) \quad (3.3)$$

We will generally write L_k for a linear classifier using k features. The classifier L_k is sparse if k is small compared to the number of features expressible under the given feature code.

It is common to fix the first feature Φ_1 at the constant function $\Phi_1(x) = 1$. Under this convention equation (3.3) is equivalent to the following.

$$L_k(x) = w_1 + \sum_{i=2}^k w_i \Phi_i(x) \quad (3.4)$$

In this case the weight w_1 is called the bias term.

For example, in the case of a blood test results x we might have the following linear classifier.

$$L_4(x) = w_1 + w_2 \Phi_f(x) + w_3 \Phi_{f,\text{squared}}(x) + w_4 \Phi_g(x) \quad (3.5)$$

Just as a decision tree is itself a test on \mathcal{X} , a linear discrimination function is itself a feature on \mathcal{X} . The score $L_k(x)$ can be interpreted as a test, or classifier, by predicting $y = 1$ when $L(x) \geq 0$ and we predict $y = -1$ when $L(x) < 0$.

3.2.1 Greedy Optimization of Log Loss (Boosting)

As with decision trees, it is easier to train sparse linear classifiers by optimizing log loss rather than optimizing error rate. As with decision trees, the log loss

can be improved even if the error rate is not improved and improving the log loss eventually leads to improvements in the error rate. To learn sparse linear classifiers in this way we must first define log loss which involves first defining $P(y|x; L)$. Intuitively, however, the larger the value $L(x)$ the more confident one should be that $y = 1$ and the further below zero $L(x)$ the more confident one should be that $y = -1$. We define the probability model $P(y|x; L)$ as follows.

$$P(y|x; L) = \frac{e^{\frac{1}{2}yL(x)}}{e^{\frac{1}{2}yL(x)} + e^{-\frac{1}{2}yL(x)}} \quad (3.6)$$

It is easy to check that $P(y|x; L) + P(-y|x; L) = 1$ and so (3.6) defines a well formed conditional distribution on y given x . Equation (3.6) can be rewritten as follows.

$$P(y|x; L) = \frac{1}{1 + e^{-yL(x)}} \quad (3.7)$$

We can now write the log loss on training data $D = (x_1, y_1), \dots, (x_n, y_n)$ as follows.

$$L_{\log}(L, D) = \sum_{i=1}^n \ln \frac{1}{P(y_i|x_i; L)} = \sum_{i=1}^n \ln \left(1 + e^{-y_i L(x_i)} \right) \quad (3.8)$$

We can grow a sparse linear classifier one term at a time. We can start by setting Φ_1 to the constant function $\Phi_1(x) = 1$ and setting w_1 as follows.

$$w_1 = \ln \frac{\sum_{i=1}^n I[y_i = 1]}{\sum_{i=1}^n I[y_i = -1]}$$

This minimizes the log loss of the constant function $L_1(x) = w_1$. Now given a linear classifier $L_k(x) = w_1 + w_2\Phi_2(x) + \dots + w_k\Phi_k(x)$ we greedily construct a new term $w_{k+1}\Phi_{k+1}(x)$ so that we get the following.

$$L_{k+1}(x) = L_k(x) + w_{k+1}\Phi_{k+1}(x)$$

To do this we first define the derivative of the loss with respect to score at each training point i .

$$\begin{aligned} \alpha_i &= -\frac{d \ln(1 + e^{-y_i L})}{dL} \\ &= \frac{y_i e^{-y_i L(x_i)}}{1 + e^{-y_i L(x_i)}} \\ &= y_i P(-y_i|x_i; L) \end{aligned}$$

Here α_i is the rate at which the log loss is reduced as we increase the score $L(x_i)$ at single input x_i . Note that if the model confidently predicts y_i , as opposed to $-y_i$, for the input x_i then α_i will be near zero. The vector $(\alpha_1, \dots, \alpha_n)$

gives the rate of loss reduction for increasing score as a function of the training data index i . We can now select the feature Φ such that the vector $\vec{\Phi}^T = (\Phi(x_1), \dots, \Phi(x_n))$ is the most aligned, or most anti-aligned, with the direction of the vector $\vec{\alpha}^T = (\alpha_1, \dots, \alpha_n)$. If $\vec{\Phi}_{k+1}$ is anti-aligned with $\vec{\alpha}$ then w_{k+1} will be negative.

$$\Phi_{k+1} = \operatorname{argmax}_{\Phi} \max \left(\frac{\vec{\alpha}^T \vec{\Phi}}{\|\vec{\Phi}\|}, \frac{-\vec{\alpha}^T \vec{\Phi}}{\|\vec{\Phi}\|} \right) \quad (3.9)$$

Given Φ_{k+1} we can select w_{k+1} so as to minimize the log loss on the training data.

$$w_{k+1} = \operatorname{argmin}_w \sum_{i=1}^n \ln \left(1 + e^{-y_i(L_k(x_i) + w\Phi_{k+1}(x_i))} \right) \quad (3.10)$$

3.2.2 Applying Occam's Razor

Here we consider how (2.22) might be applied to sparse linear classifiers. A first question is how do we measure the complexity, in bits, of a sparse linear classifier. Note that for $2 \leq i \leq k$ we can rearrange the terms of the form $w_i\Phi_i$ to be in any order and get the same function. Therefore, for a given linear discriminator function L_k we can get the following statement about the probability of L_k for $k > 2$.

$$\begin{aligned} P(L_k) &\geq (k-1)! P((w_1, (w_2, \Phi_2), \dots, (w_k, \Phi_k))) \\ |L_k| &\leq |k| + |w_1| + \dots + |w_k| + |\Phi_2| + \dots + |\Phi_k| - \log_2((k-1)!) \\ &\leq |k| + |w_1| + \dots + |w_k| + |\Phi_2| + \dots + |\Phi_k| - (k-2)(\log_2(k-2) - 1) \end{aligned}$$

If a decision tree has zero error rate on the training data then it also has zero log loss. To have zero loss, either error rate or log loss, every leaf of the decision tree must be "pure" — either only positive data points reach that leaf or only negative data points reach that leaf. However, a linear discriminator with zero classification training error still has nonzero log loss. In particular $P(y|x; L)$ is always strictly greater than 0 and strictly less than 1. This implies that the log loss is never zero. A training point that is correctly classified but where $L_k(x)$ is near zero is barely correct and such near-miss correct classifications should be avoided if possible. We would like all of the correct classifications to be correct by a sufficient safety margin. So rather than use the training equation (2.22), we use the following log-loss training equation.

$$L^* = \operatorname{argmin}_L \sum_{i=1}^n \ln \left(1 + e^{-y_i L(x)} \right) + \lambda |L| \quad (3.11)$$

We can continue to grow L using (3.9) and (3.10) until the growth fails to improve the objective in (3.11). In particular, assuming that $|k|$ is a negligible component of $|L_k|$, we add a new term $w_{k+1}\Phi_{k+1}$ provided that the reduction in total log loss exceeds $\lambda(|w_k| + |\Phi_k| - \log_2(k-2) + 1)$. As with decision trees,

it might be useful to simplify weights and tests after they have been constructed by (3.9) and (3.10).

By analogy with (2.21) we might predict that λ near unity would work well. If we take $\lambda = \ln 2$ then we have that (3.11) is equivalent to the following.

$$L^* = \operatorname{argmax}_L P(L) \prod_{i=1}^n P(y_i | x_i; L) \quad (3.12)$$

So with $\lambda = \ln 2$ we get that L^* is selected to have maximum a-posteriori probability (MAP). But somewhat different values of λ will generally perform better for different applications.

An interpretation of regularization of linear classifiers which does not rely on finite precision parameters is given in chapter 6.

Chapter 5

Linear Prediction

Sparse linear classifiers were considered in chapter 3 in conjunction with the boosting algorithm and Occam's razor. Here we consider linear classifiers more generally. In particular we consider least squares regression, logistic regression, and support vector machines all under L_2 regularization. In all cases we assume a set (or space) X of possible inputs and a feature map Φ from \mathcal{X} to \mathbb{R}^d . For $w \in \mathbb{R}^d$ we define the linear predictor $L_w(x)$ as follows.

$$L_w(x) = w^t \Phi(x) = \sum_{i=1}^d w_i \Phi_i(x) \quad (5.1)$$

As in section 3.2 we adopt the convention that $\Phi_1(x) = 1$ for all $x \in \mathcal{X}$.

In section 3.2 we used linear predictors based on a small subset of the available or expressible features. In (5.1), on the other hand, we assume that the features Φ_i for $1 \leq i \leq d$ are all of the available or expressible features. We say that (5.1) is dense while (3.3) is sparse.

It is common in machine learning applications to have many more available or expressible features than training data points. In such cases regularization, such as in (2.21), is essential. The sparse linear predictors considered in section 3.2 can be regularized by limiting the number of terms in the sum as well as the precision of numerical parameters. In this chapter we focus on dense linear classifiers and, in particular, L_2 regularization. In L_2 regularization we use $\|w\|^2$ as the measure of the complexity of w . The complexity $\|w\|^2$ is analogous to the complexity $|h|$ in (2.22) and corresponds to a Gaussian prior (probability code) for w . However, in L_2 regularization, we do not work with finite precision numerical parameters. Instead, at a theoretical level at least, we work with infinite precision real numbers. The Gaussian prior is a continuous probability density and the probability of any particular weight vector w is zero. It takes infinitely many bits to specify an infinite precision number. In this case the generalization bounds of chapter 2 cannot be applied and a new theoretical

approach is needed. In this chapter we present training equations for dense linear predictors, and discuss some properties of these training equations. This chapter does not give any theoretical justifications for the training equations themselves other than self-justifying Bayesian assumptions. Chapter 6 develops a frequentist theoretical foundation for using continuous probability densities to be used as “probability codes” in generalization bounds such as (2.15). This general framework can then be used to give frequentist generalization bounds for regularization based on a Gaussian prior.

5.1 Ridge Regression

Ridge regression is L_2 -regularized least squares regression. In least squares regression one typically assumes training data of the form $(x_1, y_1), \dots, (x_n, y_n)$ with $x \in \mathbb{S}$ and $y \in \mathbb{R}$. This is a regression problem defined by $y_t \in \mathbb{R}$ rather than a classification problem defined by $y_t \in \{-1, 1\}$. L_2 -regularized least squares regression is defined by the following training equation.

$$w^* = \operatorname{argmin}_w \left(\sum_{t=1}^n \frac{1}{2} (y - w^t \Phi(x_t))^2 \right) + \frac{1}{2} \lambda \|w\|^2 \quad (5.2)$$

Ridge regression should work reasonably well with $\lambda = 1$, with y scaled to have unit variance, and with the non-bias features (the features other than ϕ_1) scaled so as to satisfy the following.

$$\frac{1}{n} \sum_{t=1}^n \sum_{i=2}^d \Phi_i^2(x_t) = 1 \quad (5.3)$$

Some adjustment of λ will be needed to achieve optimal performance. Under these conditions ridge regression should perform reasonably well even for $d \gg n$. A frequentist justification for similar parameter settings for the classification case is given in chapter 6.

Equation (5.2) is equivalent to the following with $\sigma^2 = 1/\lambda$.

$$w^* = \operatorname{argmax}_w e^{-\frac{\|w\|^2}{2\sigma^2}} \prod_{t=1}^n e^{-\frac{(y - w^t \Phi(x_t))^2}{2}} \quad (5.4)$$

The first term can be interpreted as being proportional to a Gaussian prior on w and the second term as the product of terms proportional to a Gaussian probability for $P(y|x; w)$. So (5.4) can be rewritten as follows.

$$w^* = \operatorname{argmax}_w p(w) \prod_{t=1}^n p(y_t|x_t; w) \quad (5.5)$$

Equation (5.5) gives a Bayesian interpretation of (5.2). Unfortunately, this Bayesian interpretation does not explain why it is reasonable to take $\lambda = 1$ or

why it is reasonable to scale the features Φ_i for $i > 1$ in the way described above.

By setting the gradient of the objective in (5.2) to zero it is possible to derive the following closed form solution where Φ is the matrix defined by $\Phi_{i,t} = \Phi_i(x_t)$ and y is vector with components y_t .

$$w^* = (\Phi\Phi^T - \lambda I)^{-1}\Phi y$$

5.2 Logistic Regression

In logistic regression one typically assumes classification training data, i.e., training data of the form $(x_1, y_1), \dots, (x_n, y_n)$ with $x \in \mathcal{X}$ and $y \in \{-1, 1\}$. Logistic regression is formulated in terms of certain probability model $P(y|x; L)$ where L is a linear predictor. In particular, we define $P(y|x; w)$ as in section 3.2.1 as follows.

$$P(y|x; w) = \frac{e^{\frac{1}{2}yw^T\Phi(x)}}{e^{\frac{1}{2}yw^T\Phi(x)} + e^{-\frac{1}{2}yw^T\Phi(x)}} \quad (5.6)$$

One can check that $P(y|x; w) + P(-y|x; w) = 1$ and so (5.6) defines a well formed conditional distribution on y given x . Also note that if $w^t\Phi(x) > 0$ then $P(1|x_t; w) > 1/2$ and we should predict $y = 1$. If $w^t\Phi(x) < 0$ then we have $P(-1|x_t; w) > 1/2$ and we should predict -1 . Equation (5.6) for a training pair (x_t, y_t) can be rewritten as follows.

$$P(y_t|x_t; L) = \frac{1}{1 + e^{-m_t}} \quad (5.7)$$

$$m_t = y_t w^t \Phi(x_t) \quad (5.8)$$

The quantity m_t defined by (5.8) is called the margin. For $m_t > 0$ we have that the score $w^t\Phi(x_t)$ has the same sign as y_t and therefor the score predicts the correct label. For $m_t < 0$ we have that we have that the score $w^t\Phi(x_t)$ has the opposite sign as y_t and we predict the incorrect label. We now write the log loss on training data $D = (x_1, y_1), \dots, (x_n, y_n)$ as follows.

$$L_{\log}(w, D) = \sum_{i=1}^n \ln \frac{1}{P(y_i|x_i; L)} = \sum_{i=1}^n \ln (1 + e^{-m_i}) \quad (5.9)$$

L_2 -regularized logistic regression is defined by the following training equation.

$$w^* = \underset{w}{\operatorname{argmin}} \left(\sum_{t=1}^n \ln (1 + e^{-m_t}) \right) + \frac{1}{2} \lambda \|w\|^2 \quad (5.10)$$

The training equation (5.10) should work reasonably well for $\lambda = 1$ and for the features Φ_i for $i > 1$ scaled to satisfy 5.3).

The training equation (5.10) can be rewritten as follows where $\sigma^2 = 1/\lambda$.

$$w^* = \operatorname{argmax}_w e^{-\frac{\|w\|^2}{2\sigma^2}} \prod_{i=1}^n P(y_i|x_i; w) \quad (5.11)$$

$$= \operatorname{argmax}_w p(w) \prod_{i=1}^n P(y_i|x_i; w) \quad (5.12)$$

Equation (5.12) is the same as (5.5) except that the two equations assign different meanings to the conditional probability $P(y|x; w)$. In (5.5) we have that y is assumed to have a Gaussian distribution centered at $w^T\Phi(x)$. In (5.12) we have that $P(y|x; w)$ is defined by (5.6). While (5.12) gives a Bayesian interpretation of the logistic regression training equation (5.10), it again does not explain why we should take λ near 1 or Φ scaled as in (5.3).

There is no closed form solution for the logistic regression training equation. However the quantity being optimized in the equation is convex and so can be optimized in polynomial time using generic methods for the optimization of convex functions. The logistic regression training equation is often solved approximately using gradient descent. Gradient descent methods are described in more detail in section 5.5.

5.3 Support Vector Machines SVMs

For support vector machines (SVMs) one typically assumes classification training data, i.e., training data of the form $(x_1, y_1), \dots, (x_n, y_n)$ with $x \in \mathcal{X}$ and $y \in \{-1, 1\}$. Support vector machines are perhaps best motivated as a sparse version of logistic regression. Consider the log loss function defined as follows.

$$L_{\log}(m) = \ln(1 + e^{-m})$$

We can approximate this function for $m < -1$ or $m > 1$ as follows.

$$L_{\log}(m) \approx -m \quad \text{for } m < -1$$

$$L_{\log}(m) \approx e^{-m} \approx 0 \quad \text{for } m > 1$$

We can further approximate this with the following definition of hinge loss.

$$\begin{aligned} L_{\text{hinge}}(m) &= \begin{cases} 1 - m & \text{if } m \leq 1 \\ 0 & \text{for } m \geq 1 \end{cases} \\ &= \max(0, 1 - m) \end{aligned}$$

The training equation for an SVM is then the following.

$$w^* = \operatorname{argmin}_w \left(\sum_{t=1}^n \max(0, 1 - m) \right) + \frac{1}{2} \lambda \|w\|^2 \quad (5.13)$$

Again, this equation tends to work well with $\lambda = 1$ and with Φ_i , for $i > 1$, scaled so as to satisfy (5.3).

Hinge loss is motivated here as a simplification of logistic loss. One way in which hinge loss is simpler is that, unlike (5.10), the SVM training equation (5.13) is equivalent to a convex quadratic program. This allows the training equation to be solved by quadratic programming algorithms rather than by generic gradient descent. Although gradient descent still works reasonably well for SVMs (better than most quadratic programming algorithms) in many practical settings. To convert (5.13) to a convex quadratic program we replace $\max(0, 1 - m)$ with a slack variable η constrained to be larger than both 0 and $1 - m$ as follows.

$$\begin{aligned} & \text{minimize} && \sum_{t=1}^n \eta_t + \frac{1}{2} \lambda \|w\|^2 \\ & \text{subject to} && \eta_t \geq 0 \\ & && \eta_t \geq 1 - y_t w^T \Phi(x_t) \end{aligned} \quad (5.14)$$

The optimization problem (5.14) minimizes a convex quadratic function subject to linear constraints which is the definition of a convex quadratic program.

5.4 The Representer Theorem

For classification training data, Ridge regression, logistic regression, and SVMs are all instances of the following generic training equation for an arbitrary loss function L .

$$w^* = \operatorname{argmin}_w \left(\sum_{t=1}^n L(m_t) \right) + \frac{1}{2} \lambda \|w\|^2 \quad (5.15)$$

For the case of ridge regression with we have the following.

$$\begin{aligned} (y_t - w^T \Phi(x_t))^2 &= y_t^2 (y_t - w^T \Phi(x_t))^2 \\ &= (y_t^2 - y_t w^T \Phi(x_t))^2 \\ &= (1 - m_t)^2 \end{aligned}$$

So for classification data, all of the loss functions are functions of the margin. The representer theorem is the following.

Theorem 1. *For any function $L : \mathbb{R} \rightarrow \mathbb{R}$, if w^* is a minimizer of (5.15) then there exists α_t for $1 \leq t \leq n$ such that w^* can be written as follows.*

$$w^* = \sum_{t=1}^n \alpha_t \Phi(x_t)$$

The representer theorem states that w^* is in the span of the feature vectors. It can be proved by observing that if w^* was not in the span then we would have

$$w^* = w_{\parallel} + w_{\perp}$$

where w_{\parallel} is the projection of w^* onto the span of the vectors $\Phi(x_t)$ and w_{\perp} is $w^* - w_{\parallel}$ which is perpendicular to the space spanned by the vectors $\Phi(x_t)$. We then have the following.

$$\begin{aligned} \|w^*\|^2 &= \|w_{\parallel}\|^2 + \|w_{\perp}\|^2 \\ \|w_{\parallel}\| &< \|w^*\| \end{aligned}$$

$$\begin{aligned} (w^*)^T \Phi(x_t) &= (w_{\parallel} + w_{\perp})^T \Phi(x_t) \\ &= w_{\parallel}^T \Phi(x_t) + w_{\perp}^T \Phi(x_t) \\ &= w_{\parallel}^T \Phi(x_t) \end{aligned}$$

These relations imply that w_{\parallel} has smaller norm and yields the same margin values and therefore achieves a better value of the optimization problem (5.15) than does w^* . Hence we have a contradiction, and it must be true that w^* is already in the span of the vectors $\Phi(x_t)$.

If the loss function L is differentiable then by setting the derivative of the objective in (5.15) to zero we can derive the following.

$$w^* = \frac{1}{\lambda} \sum_{t=1}^n -y_t \left(\frac{dL}{dm} \Big|_{m_t} \right) \Phi(x_t) \quad (5.16)$$

Equation (5.16) implies the representer theorem in the case where L is differentiable. For ridge regression (even with general regression data) we get the following.

$$w^* = \frac{1}{\lambda} \sum_{t=1}^n (y - (w^*)^T \Phi(x_t)) \Phi(x_t) \quad (5.17)$$

We then get that the weight on $\Phi(x_t)$ is proportional to the residual value $y - (w^*)^T \Phi(x_t)$. So the weight is largest (in absolute value) at training points where the predictor has the most error. For logistic regression we get the following.

$$\begin{aligned} w^* &= \frac{1}{\lambda} \sum_{t=1}^n \frac{e^{-m_t}}{1 + e^{-m_t}} \Phi(x_t) \\ &= \frac{1}{\lambda} \sum_{t=1}^n \frac{1}{1 + e^{m_t}} \Phi(x_t) \\ &= \frac{1}{\lambda} \sum_{t=1}^n P(-y_t | x_t; w) \Phi(x_t) \end{aligned} \quad (5.18)$$

So for logistic regression the weight on $\Phi(x_t)$ is the probability of the opposite label from the one given in the training data. As with ridge regression, the training points on which the predictor is most wrong are weighted most highly. However, unlike ridge regression, in logistic regression the weight on $\Phi(x_t)$ never exceeds 1. This limits the influence of any single training data point making the learning process more robust to errors in the training data.

For SVMs we have that the hinge loss is not differentiable at $m = 1$. However, it is possible to prove the following.

$$w^* = \frac{1}{\lambda} \left(\sum_{t:m_t < 1} y_t \Phi(x_t) + \sum_{t:m_t = 1} y_t \alpha_t \Phi(x_t) \right) \quad m_t = y_t (w^*)^T \Phi(x_t), \quad \alpha_t \in [0, 1] \quad (5.19)$$

Here we have that well classified points, points satisfying $m_t > 1$, do not contribute to w^* at all. The points that do contribute to w^* , those with $m_t \leq 1$, are called the support vectors. In many cases the support vectors form only a small subset of the training points. For example in a detection problem where the vast majority of inputs are negative the support vectors typically include only a small fraction of the negative training points — the so-called hard negatives. Also, typically a large number of the support vectors will have margin exactly equal to 1. These are the so-called critical support vectors. Critical support vectors are at an intermediate stage between having full weight and having zero weight and the weight can be adjusted so as to keep them critical. If we knew what the support vectors were, with k of the vectors specified as being critical, we could solve for the k weights α_t by solving the k equations of the form $m_t = 1$ for the critical support vectors.

5.5 Stochastic Gradient Descent

Modern machine learning problems typically involve huge training data sets. The training set is sometimes so large that it is reasonable to think of it as being infinite. This is sometimes referred to as the data-rich regime. In the data-rich regime we cannot use any training method that requires reading all the training data. For example, most quadratic programming algorithms are infeasible. We should be able to think of the training data as a resource with the property that more training data is always better and infinite training data is best of all. A simple training algorithm whose performance improves with increasing data size, even to the limit of infinite training data, is stochastic gradient descent.

Here we consider the generic L_2 -regularized training equation (5.15) and

consider the gradient with respect to w of quantity being optimized.

$$Q(w) = \left(\sum_{t=1}^n L(y_t w^t \Phi(x_t)) \right) + \frac{1}{2} \lambda \|w\|^2 \quad (5.20)$$

$$\nabla Q = \left(\sum_{t=1}^n y_t \left(\frac{dL}{dm} \right) \Phi(x_t) \right) + \lambda w \quad (5.21)$$

Gradient descent computes a sequence of weight vectors w^1, w^2, w^3, \dots by initializing w^1 somehow (perhaps to zero) and then repeating the following update where η can be taken to be a fixed learning rate.

$$w^{t+1} = w^t - \eta \nabla_w Q \quad (5.22)$$

Note that computing even a single value of $\nabla_w Q$ as given by equation (5.21) is impossible for infinite training data as it involves a summation over the entire training set. For finite data sets, but a bound on the amount available running time, the performance of gradient descent degrades as the amount of training data increases. This is counter to the view of training data as a resource where more is always better.

In the data-rich regime stochastic gradient descent is more effective. Stochastic gradient descent repeats a stochastic update where one first selects s at random uniformly between 1 and n (the number of training points) and then performs the following update where η_t is a learning rate that is typically reduced over time.

$$w^{t+1} = w^t + \eta_t \left(y_s \left(\frac{-dL}{dm} \right) \Phi(x_s) - \frac{\lambda}{n} w^t \right) \quad (5.23)$$

Chapter 6

Margin Bounds

This chapter attempts to provide some theoretical understanding of L_2 regularization and the default settings of the parameters of the training equations discussed in chapter 5. Our basic approach is to relate L_2 regularization to a Gaussian prior density on weight vectors. We saw in chapter 2 that it is possible to define a probability code by $|h| = \log_2(1/P(h))$ where P is a probability distribution on classifiers. However, for a Gaussian prior density the probability of any particular weight vector w is zero and we get that the bit complexity $|w|$ is infinite. It takes infinitely many bits to specify an infinite precision number. Even if we specify individual weights with finite precision numbers, a Gaussian prior allows the weights to be evenly spread over a number of parameters larger than the sample size. It does not seem to be possible to model a Gaussian prior with a discrete distribution over finite precision weights. To properly handle the Gaussian prior we use the PAC-Bayesian theorem described below.

6.1 The PAC-Bayesian Theorem

Like the Occam bound (2.15), the PAC-Bayesian theorem is extremely general. We consider an arbitrary input space \mathcal{X} and an arbitrary set \mathcal{H} of classifiers such that for $h \in \mathcal{H}$ and $x \in \mathcal{X}$ we have $h(x) \in \{-1, 1\}$. The PAC-Bayesian theorem applies to the loss of a Gibbs classifier. A Gibbs classifier stochastically selects a weight vector w from a proposal distribution Q and then uses w to make the prediction. For any distribution Q on \mathcal{H} we define $\text{err}(Q, x, y)$ to be the error rate of the Gibbs classifier defined by Q on the pair (x, y) . In other words $\text{err}(Q, x, y)$ is the probability that when h is drawn from Q we have $h(x) \neq y$. Or in formal notation we have the following.

$$\text{err}(Q, x, y) = P_{h \sim Q}[h(x) \neq y]$$

We now consider a fixed probability distribution (or density) ρ on pairs (x, y) with $x \in \mathcal{X}$ and $y \in \{-1, 1\}$. We let $(x_1, y_1), \dots, (x_n, y_n)$ be n pairs drawn

independently from ρ . We define $\widehat{\text{err}}(Q)$ to be the error rate of the Gibbs classifier defined by Q on the training sample.

$$\widehat{\text{err}}(Q, x, y) = \frac{1}{n} \sum_{t=1}^n \text{err}(Q, x_t, y_t)$$

We define the generalization error $\text{err}(Q)$ to be the error rate of the Gibbs classifier on fresh data drawn from ρ .

$$\text{err}(Q) = \mathbb{E}_{(x,y) \sim \rho} [\text{err}(Q, x, y)]$$

As with ordinary classifiers, a Gibbs classifier can over-fit the training data in which case $\widehat{\text{err}}(Q)$ will be small but the error on fresh data $\text{err}(Q)$ will be large. To control over-fitting we need a notion of complexity for Q . We can then apply Occam's razor and prefer "simple" Gibbs classifiers to complex ones. To define a notion of complexity we assume a probability distribution P on \mathcal{H} . The probability distribution P defines a kind of probability code which determines the complexity of Q as follows.

$$|Q| = KL(Q, P) = \mathbb{E}_{h \sim Q} \left[\ln \frac{Q(h)}{P(h)} \right]$$

We now state a version of the PAC-Bayesian theorem directly analogous to (2.15). With probability at least $1 - \delta$ over the draw of the training sample, the following holds simultaneously for all Gibbs classifiers Q .

$$\text{err}(Q) \leq \widehat{\text{err}}(Q) + \sqrt{\widehat{\text{err}}(Q)c(Q)} + c(Q) \quad (6.1)$$

$$c(Q) \doteq \frac{2(|Q| + \ln \frac{n}{\delta})}{n-1}$$

The term "PAC-Bayesian" comes from the idea that the theorem states that Q is "probably approximately correct" (PAC) when $\widehat{\text{err}}(Q)$ and $|Q|$ are both small and, furthermore, the complexity $|Q|$ is defined by a "Bayesian" prior. A proof (6.1) is given in section 6.5. For now, however, we take this theorem for granted and consider how it can be used to give frequentist generalization bounds for L_2 regularization.

6.2 L_2 Regularization

L_2 regularization corresponds to a Gaussian prior. From the frequentist perspective the Gaussian prior has no notion of "truth" or "validity". It is simply a way of assigning complexity to classifiers such that Occam's razor can be applied. In particular we apply (6.1) with the complexity $|Q|$ defined by taking P to be Gaussian.

More formally, we now take $\mathcal{H} = \mathbb{R}^d$. We also assume a feature map $\Psi : \mathcal{X} \rightarrow \mathbb{R}^d$ and for $w \in \mathcal{H}$ and $x \in \mathcal{X}$ we consider the classifier defined by the sign of $w^T \Phi(x)$: if $w^T \Phi(x) \geq 0$ we predict 1 and if $w^T \Phi(x) < 0$ we predict -1. We take the prior P to be the following Gaussian density where Z is a normalizing constant.

$$P_\sigma(w) = \frac{1}{Z} e^{-\frac{\|w\|^2}{2\sigma^2}}$$

We consider Gibbs classifiers defined by proposal distributions of the following form.

$$Q_{(\mu,\sigma)}(w) = \frac{1}{Z} e^{-\frac{\|w-\mu\|^2}{2\sigma^2}}$$

In other words the proposal distribution is also a Gaussian distribution but centered at the vector μ rather than the origin. We can calculate the complexity of the Gibbs classifier $Q_{(\mu,\sigma)}$ under the prior P as follows.

$$\begin{aligned} |Q_{(\mu,\sigma)}| &= KL(Q_{\mu,\sigma}, P_\sigma) \\ &= \mathbb{E}_{w \sim Q_{(\mu,\sigma)}} \left[\ln \frac{Q_\mu(w)}{P(w)} \right] \\ &= \mathbb{E}_{w \sim Q_{(\mu,\sigma)}} \left[\frac{\|w\|^2 - \|w - \mu\|^2}{2\sigma^2} \right] \\ &= \mathbb{E}_{w \sim Q_{(\mu,\sigma)}} \left[\frac{\|w\|^2 - (\|w\|^2 - 2w^T \mu + \|\mu\|^2)}{2\sigma^2} \right] \\ &= \mathbb{E}_{w \sim Q_{(\mu,\sigma)}} \left[\frac{2w^T \mu}{2\sigma^2} \right] - \frac{\|\mu\|^2}{2\sigma^2} \\ &= \frac{2\mu^T \mathbb{E}_{w \sim Q_{(\mu,\sigma)}} [w]}{2\sigma^2} - \frac{\|\mu\|^2}{2\sigma^2} \\ &= \frac{\|\mu\|^2}{2\sigma^2} \end{aligned} \tag{6.2}$$

Next we analyze $\text{err}(Q_u, x, y)$.

$$\text{err}(Q, x, y) = P_{w \sim Q_{(\mu,\sigma)}} [yw^T \Phi(x) < 0] \tag{6.3}$$

In other words the error rate on (x, y) is just the probability that the margin is negative when w is drawn from $Q_{(\mu,\sigma)}$. A random weight vector w drawn from Q_μ can be written as follows where ϵ is a random Gaussian noise vector.

$$w = \mu + \epsilon$$

We then have the following.

$$\begin{aligned}
\text{err}(Q_{(\mu,\sigma)}, x, y) &= P[yw^T\Phi(x) < 0] \\
&= P[y(\mu + \epsilon)^T\Phi(x) < 0] \\
&= P[-y\epsilon^T\Phi(x) \geq y\mu^T\Phi(x)] \\
&= P[\epsilon^T\Phi(x) \geq m] \\
&= P\left[\left(\frac{\epsilon}{\sigma}\right)^T \frac{\Phi(x)}{\|\Phi(x)\|} \geq \frac{m}{\sigma\|\Phi(x)\|}\right] \\
&= L_{\text{probit}}\left(y\left(\frac{\mu}{\sigma}\right)^T \left(\frac{\Phi(x)}{\|\Phi(x)\|}\right)\right) \tag{6.4}
\end{aligned}$$

$$L_{\text{probit}}(m) = P_{\gamma \sim \mathcal{N}(0,1)}[\gamma > m]$$

The function $L_{\text{probit}}(m)$ is called probit loss. Note that for $m < -2$ we have $L_{\text{probit}}(m) \approx 1$ and for $m > 2$ we have $L_{\text{probit}}(m) \approx 0$. The probit function L is a form of sigmoid that transitions smoothly from near 1 to near 0 over the interval $[-2, 2]$.

Equations (6.2) and (6.4) imply the following ‘‘symmetry of scale’’ relations.

$$KL(Q_{(\mu,\sigma)}, P_\sigma) = KL(Q_{(\mu/\sigma,1)}, P_1)$$

$$\text{err}(Q_{(\mu,\sigma)}, x, y) = \text{err}(Q_{(\mu/\sigma,1)}, x, y)$$

These symmetry of scale relations hold whenever we simultaneously apply a scaling to both the prior and the posterior. However, these scaling relations do not apply when we consider loss functions other than classification error rate. These symmetry of scale relations imply that the performance guarantee provided by the PAC-Bayesian theorem for $Q_{\mu,\sigma}$ under prior P_σ is identical to the performance guarantee for $Q_{\mu/\sigma,1}$ under prior P_1 . So without loss of generality we can assume $\sigma = 1$ and we will write P and Q_μ for P_1 and $Q_{\mu,1}$ respectively.

Taking $\sigma = 1$, equation (6.4) implies the following.

$$\begin{aligned}
\text{err}(Q_\mu) &= \mathbb{E}_{(x,y) \sim \rho} \left[L_{\text{probit}} \left(y\mu^T \left(\frac{\Phi(x)}{\|\Phi(x)\|} \right) \right) \right] \\
\widehat{\text{err}}(Q_\mu) &= \frac{1}{n} \sum_{i=1}^n L_{\text{probit}} \left(y\mu^T \left(\frac{\Phi(x)}{\|\Phi(x)\|} \right) \right)
\end{aligned}$$

We now have that both $\text{err}(Q_\mu)$ and $\widehat{\text{err}}(Q_\mu)$ are functions of the normalized feature vectors $\Phi(x)/\|\Phi(x)\|$. So normalizing each feature vector to have unit

norm has no influence on the bound and simplifies the expressions. So with these simplifications we now have the following where we now introduce the notation $L_{\text{probit}}(\mu)$ and $\hat{L}_{\text{probit}}(\mu)$.

$$\begin{aligned} L_{\text{probit}}(\mu) &\doteq \mathbb{E}_{(x,y)\sim\rho} [L_{\text{probit}}(y\mu^T\Phi(x))] = \text{err}(Q_\mu) \\ \hat{L}_{\text{probit}}(\mu) &\doteq \frac{1}{n} \sum_{t=1}^n L_{\text{probit}}(m_t) = \widehat{\text{err}}(Q_\mu) \end{aligned}$$

We now insert $\text{err}(Q_\mu)$, $\widehat{\text{err}}(Q_\mu)$, and $|Q_\mu|$ into (6.1). We get that with probability at least $1 - \delta$ over the draw of the training data the following holds simultaneously for all vectors μ .

$$\begin{aligned} L_{\text{probit}}(\mu) = \text{err}(Q_\mu) &\leq \hat{L}_{\text{probit}}(\mu) + \sqrt{\hat{L}_{\text{probit}}(\mu)c(\mu) + c(\mu)} \quad (6.5) \\ c(\mu) &\doteq \frac{\|\mu\|^2 + 2 \ln \frac{2\sqrt{n}}{\delta}}{n} \end{aligned}$$

As observed in section 2.7, we can achieve a $3/2$ approximation of the minimum in (6.5) with the following linear training equation.

$$\mu^* = \underset{\mu}{\text{argmin}} \left(\sum_{t=1}^n L_{\text{probit}}(m_t) \right) + \|\mu\|^2 \quad (6.6)$$

The training equation (6.6) is called probit regression (with $\lambda = 2$). We have derived probit regression from the PAC-Bayesian theorem for classification error together with a Gaussian prior. In this process we showed that without loss of generality we can work with normalized feature vectors and with $\sigma = 1$ in the Gaussian prior. The analysis predicts $\lambda = 2$.

6.3 L_1 Regularization

Here we assume the same technical setting as in section 6.2 except that we now assume a Laplace prior (probability code) rather than a Gaussian prior. The prior is defined with parameter γ as follows.

$$P_\gamma(w) = \frac{1}{Z} e^{-\frac{\|w\|_1}{\gamma}} \quad (6.7)$$

$$\|w\|_1 = \sum_i |w_i| \quad (6.8)$$

When using the prior P_γ we will use proposal distributions of the form $Q_{(\mu,\gamma)}$ defined as follows.

$$Q_{(\mu,\gamma)}(w) = \frac{1}{Z} e^{-\frac{\|w_i - \mu\|_1}{\gamma}} \quad (6.9)$$

We can think of the posterior as being defined by $w = \mu + \epsilon$ where ϵ is a Laplace random variable (as defined by P_γ). As in section 6.2 we have “symmetry of scale” relations as follows.

$$KL(Q_{(\mu,\gamma)}, P_\gamma) = KL(Q_{(\mu/\gamma,1)}, P_1)$$

$$\text{err}(Q_{(\mu,\gamma)}, x, y) = \text{err}(Q_{(\mu/\gamma,1)}, x, y)$$

These symmetry of scale relations imply that the performance guarantee provided by the PAC-Bayesian theorem for $Q_{(\mu,\gamma)}$ under prior P_γ is identical to the performance guarantee for $Q_{(\mu/\gamma,1)}$ under prior P_1 . So without loss of generality we can assume $\gamma = 1$ and we will write P and Q_μ for P_1 and $Q_{\mu,1}$ respectively.

As in section 6.2, we now analyze the complexity $|Q_\mu|$ and the error $\text{err}(Q_\mu, x, y)$ in more detail.

$$\begin{aligned} KL(Q_\mu, P) &= \mathbb{E}_{w \sim Q_\mu} \left[\sum_{i=1}^d |w_i| - |w_i - \mu_i| \right] \\ &\leq \mathbb{E}_{w \sim Q_\mu} \left[\sum_{i=1}^d |\mu_i| \right] dz \\ &= \sum_{i=1}^d |\mu_i| \\ &= \|\mu\|_1 \end{aligned} \quad (6.10)$$

$$\text{err}(Q_\mu, x, y) = \mathbb{P}_{\epsilon \sim P} [\epsilon \cdot \Phi(x) \geq y \mu^T \Phi(x)] \quad (6.11)$$

For each component ϵ_i of the random variable ϵ has standard deviation equal to 1 and is independent of the other components. This implies that the random variable $\epsilon^T \Phi(x)$ has variance equal to $\|\Phi(x)\|^2$. Furthermore, since the random variable $\epsilon^T \Phi(x)$ is a sum of independent random variable it will typically be approximately normally distributed. This gives the following.

$$\begin{aligned} \text{err}(Q_\mu, x, y) &\approx \mathbb{P}_{\epsilon \sim \mathcal{N}(0, \|\Phi(x)\|)} [\epsilon \geq y \mu^T \Phi(x)] \\ &= \mathbb{P}_{\epsilon \sim \mathcal{N}(0,1)} \left[\epsilon \geq \frac{y \mu^T \Phi(x)}{\|\Phi(x)\|} \right] \\ &= L_{\text{probit}} \left(y \mu^T \left(\frac{\Phi(x)}{\|\Phi(x)\|} \right) \right) \end{aligned}$$

So, as for a Gaussian prior, we get that both the empirical and generalization loss are functions of $\Phi(x)/\|\Phi(x)\|$ and so without loss of generality we can assume that all feature vectors are normalized. Assuming all feature vectors are normalized we get that with probability at least $1 - \delta$ over the draw of the training data the following holds simultaneously for all $\mu \in \mathbb{R}^d$.

$$\begin{aligned} c(\mu) &\doteq \frac{2(\|\mu\|_1 + \ln \frac{n}{\delta})}{n-1} \\ \text{err}(Q_\mu) &\leq \widehat{\text{err}}(Q_\mu) + \sqrt{\widehat{\text{err}}(Q_\mu)c(\mu) + c(\mu)} \\ &\approx \hat{L}_{\text{probit}}(\mu) + \sqrt{\widehat{\text{err}}(Q_\mu)c(\mu) + c(\mu)} \end{aligned} \quad (6.12)$$

As in previous sections, we note that a $3/2$ approximation of the minimization problem defined by (6.12) is achieved at the optimum of the following simpler optimization problem

$$\mu^* = \underset{\mu}{\text{argmin}} \hat{L}_{\text{probit}}(\mu) + \|\mu\|_1 \quad (6.13)$$

So we again get a form of probit regression but with regularizer $\|\mu\|_1$ rather than $\|\mu\|^2$. As in previous sections, the analysis predicts a specific relative weighting between $\hat{L}_{\text{probit}}(\mu)$ and $\|\mu\|_1$.

6.4 An Alternate L_1 Regularization

As of this writing the most commonly cited generalization bounds for classification error rate under L_1 regularization are not based on the Laplace prior but based instead on a prior which yields a different complexity measure also involving the L_1 norm of w . To describe the difference in the two bounds we first define $\|\Phi\|_2$ and $\|\Phi\|_\infty$ as follows.

$$\begin{aligned} \|\Phi\|_2 &= \sup_{x \in \mathcal{X}} \|\Phi(x)\|_2 \\ \|\Phi\|_\infty &= \sup_{x \in \mathcal{X}} \|\Phi(x)\|_\infty \\ &= \sup_{x \in \mathcal{X}} \max_i |\Phi_i(x)| \end{aligned}$$

The alternate L_1 analysis yields a dependence on $\|\Phi\|_\infty$ rather than $\|\Phi\|_2$. More specifically, the most commonly L_1 analysis yields a bound with a complexity measure involving $\|w\|_1^2 \|\Phi\|_\infty^2$ while the Laplace prior, when formulated in a comparable way, involves $\|w\|_1 \|\Phi\|_2$. These quantities are incomparable. We have $\|\Phi\|_\infty < \|\Phi\|_2$ but (typically) $\|w\|_1^2 > \|w\|_1$.

From an empirical perspective both analyses suggest a training equation of the following form.

$$w^* = \operatorname{argmin}_w \sum_{t=1}^n \hat{L}_{\text{probit}}(m_t) + \lambda \|w\|_1$$

We can tune λ by measuring performance on hold-out data. If the feature vectors are scaled such that $\|\Phi\|_\infty = 1$ then the Laplace prior predicts λ near $\|\Phi\|_2$ while alternate prior predicts λ near $\|w^*\|_1$. Since both bounds hold simultaneously with probability at least $1 - 2\delta$, the combination of the two analyses predicts λ near the minimum of $\|\Phi\|_2$ and $\|w^*\|_1$.

For the alternate L_1 analysis we first transform the feature map into a more convenient form. Given a feature map $\Phi : \mathcal{X} \rightarrow \mathbb{R}^d$ we define a new feature map $\Psi : \mathcal{X} \rightarrow \mathbb{R}^{2d}$ as follows for $i \in \{1, \dots, d\}$.

$$\Psi_i(x) = \Phi_i(x)$$

$$\Psi_{d+i}(x) = -\Phi_i(x)$$

Working with the feature vector Ψ we can then restrict our attention to weight vectors w with $w_j \geq 0$ for $1 \leq j \leq 2d$. We define a prior P on weight vectors by first selecting parameter $N \geq 1$ where the probability of N equals $1/(N(N+1))$. Given N we define a distribution P_N on weight vectors w by sampling N features uniformly from the $2d$ features (possibly with repetitions) and weighting each drawn feature with $1/N$. For w with $w_j \geq 0$ and $\|w\|_1 = 1$, and $N \geq 1$, we define $Q_{(\mu, N)}$ to be the proposal distribution which draws N features independently from the probability distribution defined by w . By applying the PAC-Bayesian theorem to P and $Q_{(\mu, N)}$ it is possible to prove that with probability at least $1 - \delta$ over the draw of the training data the following holds simultaneously for all w (here we do not assume that feature vectors are normalized). We omit any further details of the analysis.

$$\begin{aligned} \operatorname{err}(w) &\leq \hat{L}_{\text{margin}}(w) + \sqrt{\hat{L}_{\text{margin}}(w)c(w)} + c(w) \\ c(w) &= \tilde{O}\left(\frac{\|w\|_1^2 \|\Phi\|_\infty^2 + \ln \frac{1}{\delta}}{n}\right) \end{aligned} \tag{6.14}$$

$$\hat{L}_{\text{margin}}(w) = \frac{1}{n} \sum_{t=1}^n L_{\text{margin}}(m_t)$$

$$L_{\text{margin}}(m) = \begin{cases} 0 & \text{if } m \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

To make these comparable we state (without proof) analogous theorems derived from the Gaussian and Laplace prior respectively. From the Gaussian

prior we get the following where, for comparison, we again do not assume that feature vectors are normalized.

$$\begin{aligned} \text{err}(w) &\leq \hat{L}_{\text{margin}}(w) + \sqrt{\hat{L}_{\text{margin}}(w)c(w)} + c(w) \\ c(w) &= \tilde{O}\left(\frac{\|w\|_2^2 \|\Phi\|_2^2 + \ln \frac{1}{\delta}}{n}\right) \end{aligned} \quad (6.15)$$

$$\|\Phi\|_2^2 = \sup_x \|\Phi(x)\|^2 \quad (6.16)$$

For the Laplace prior we get the following.

$$\text{err}(w) \leq \hat{L}_{\text{margin}}(w) + \sqrt{\hat{L}_{\text{margin}}(w)c(w)} + c(w) \quad (6.17)$$

$$c(w) = \tilde{O}\left(\frac{\|w\|_1 \|\Phi\|_2 + \ln \frac{1}{\delta}}{n}\right) \quad (6.18)$$

When we take the hidden constants into account, the performance guarantees given by (6.15) and (6.18) are considerably weaker than the performance guarantees stated in sections 6.2 and 6.3. The bounds (6.15) and (6.18) are given here only to provide a direct comparison with 6.14.

6.5 Proof of the PAC-Bayesian Theorem

Let μ be the expectation of a random variable $x \in [0, 1]$ and let $\hat{\mu}$ be the average of a sample of n IID values of x . For $q \leq \mu$ we have the following concentration inequality which we do not prove here.

$$p(\hat{\mu} \leq q) \leq e^{-nKL(p,\mu)} \quad (6.19)$$

$$KL(p, q) \doteq p \ln \frac{p}{q} + (1-p) \ln \frac{1-q}{1-p}$$

The inequality (6.19) is the strongest concentration inequality on Bernoulli variables provably by Chernoff's exponential moment method. The concentration inequality (2.12) follows from (6.19) and the following inequality which holds for $p \leq q$ (and which we also do not prove here).

$$KL(p, q) \geq \frac{(q-p)^2}{2q} \quad (6.20)$$

Our first step is to prove the following from (6.19).

$$\mathbb{E} \left[e^{(n-1)KL^<(\hat{\mu}, \mu)} \right] \leq n \quad (6.21)$$

$$KL^<(p, \mu) \doteq \begin{cases} 0 & \text{if } p \geq \mu \\ KL(p, \mu) & \text{otherwise} \end{cases}$$

To prove (6.21) we first note the following which holds for $p < \mu$.

$$\begin{aligned} P(\hat{\mu} \leq p) &= P\left(e^{(n-1)KL^<(\hat{\mu}, \mu)} \geq e^{(n-1)KL(p, \mu)}\right) \\ &\leq e^{-nKL(p, \mu)} \end{aligned}$$

$$P\left(e^{(n-1)KL^<(\hat{\mu}, \mu)} \geq \eta\right) \leq \max 1, \eta^{\frac{-n}{n-1}}$$

We can then use the general fact that for W non-negative we have $\mathbb{E}[W] = \int_0^\infty P(W \geq \nu) d\nu$. This gives the following.

$$\begin{aligned} \mathbb{E} \left[e^{(n-1)KL^<(\hat{\mu}, \mu)} \right] &\leq 1 + \int_1^\infty \nu^{-n/(n-1)} d\nu \\ &= 1 - (n-1) \left[\nu^{-1/(n-1)} \right]_1^\infty \\ &= n \end{aligned}$$

We now consider a space \mathcal{H} of predictors and a sample $D = (x_1, y_1), \dots, (x_n, y_n)$ sampled IID from an underlying distribution ρ . We have that (6.21) implies the following for any individual $h \in \mathcal{H}$.

$$\mathbb{E}_{D \sim \rho^n} \left[e^{(n-1)KL^<(\widehat{\text{err}}(Q), \text{err}(Q))} \right] \leq n$$

This implies the following where P is any prior probability (measure) on \mathcal{H} .

$$\begin{aligned} \mathbb{E}_{h \sim P} \left[\mathbb{E}_{D \sim \rho^n} \left[e^{(n-1)KL^<(\widehat{\text{err}}(Q), \text{err}(Q))} \right] \right] &\leq n \\ \mathbb{E}_{D \sim \rho^n} \left[\mathbb{E}_{h \sim P} \left[e^{(n-1)KL^<(\widehat{\text{err}}(Q), \text{err}(Q))} \right] \right] &\leq n \end{aligned} \quad (6.22)$$

Markov's inequality can be phrased as saying that for any random variable x we have that with probability at least $1 - \delta$ we have $x \leq \mathbb{E}[x]/\delta$. So (6.22) implies that probability at least $1 - \delta$ over the draw of the training data we have the following.

$$\mathbb{E}_{h \sim P} \left[e^{(n-1)KL^<(\widehat{\text{err}}(h), \text{err}(h))} \right] \leq \frac{n}{\delta} \quad (6.23)$$

We now prove the following shift of measure lemma.

$$\begin{aligned}
\mathbb{E}_{h \sim Q} [f(h)] &= \mathbb{E}_{h \sim Q} \left[\ln e^{f(h)} \right] \\
&= \mathbb{E}_{h \sim Q} \left[\ln \frac{dP(h)}{dQ(h)} e^{f(h)} + \ln \frac{dQ(h)}{dP(h)} \right] \\
&= KL(Q, P) + \mathbb{E}_{h \sim Q} \left[\ln \frac{dP(h)}{dQ(h)} e^{f(h)} \right] \\
&\leq KL(Q, P) + \ln \mathbb{E}_{h \sim Q} \left[\frac{dP(h)}{dQ(h)} e^{f(h)} \right] \\
&= KL(Q, P) + \ln \mathbb{E}_{h \sim P} \left[e^{f(h)} \right]
\end{aligned}$$

We now apply the shift of measure lemma with $f(h) = (n-1)KL^<(\widehat{\text{err}}(h), \text{err}(h))$ and then apply (6.23) to $e^{f(h)}$ to get the following.

$$\mathbb{E}_{h \sim Q} [(n-1)KL^<(\widehat{\text{err}}(h), \text{err}(h))] \leq KL(Q, P) + \ln \frac{n}{\delta} \quad (6.24)$$

We now use the following joint convexity property of KL -divergence which we do not prove here.

$$\mathbb{E} [KL^<(x, y)] \geq KL^<(\mathbb{E}[x], \mathbb{E}[y]) \quad (6.25)$$

Combining (6.24) and (6.25) gives the following.

$$KL^<(\widehat{\text{err}}(Q), \text{err}(Q)) \leq \frac{KL(Q, P) + \ln \frac{n}{\delta}}{n-1} \quad (6.26)$$

Next we use (6.20) to get the following.

$$\frac{(\widehat{\text{err}}(Q) - \text{err}(Q))^2}{2\text{err}(Q)} \leq \frac{KL(Q, P) + \ln \frac{n}{\delta}}{n-1} \quad (6.27)$$

Finally by solving a quadratic formula, as in section 2.6, one can show that (6.27) implies the following, which is the PAC-Bayesian theorem stated in section 6.1.

$$\begin{aligned}
\text{err}Q &\leq \widehat{\text{err}}(Q) + \sqrt{\widehat{\text{err}}(Q)c(Q)} + c(Q) \\
c(Q) &\doteq \frac{2(KL(Q, P) + \ln \frac{n}{\delta})}{n-1} \\
&= \frac{2(|Q| + \ln \frac{n}{\delta})}{n-1}
\end{aligned}$$

Using a somewhat more elaborate argument the PAC-Bayesian theorem can be improved to the following.

$$KL(\widehat{\text{err}}(Q), \text{err}(Q)) \leq \frac{KL(Q, P) + \ln \frac{2\sqrt{n}}{\delta}}{n}$$

Chapter 7

Kernels

The representer theorem of section 5.4 for L_2 regularization makes it possible to work with inner products without ever computing actual feature vectors. When this is done for SVMs the result is a kernel SVM, also commonly called a “nonlinear” SVM. A commonly used nonlinear SVM is the Gaussian kernel SVM. Kernel SVMs bear a certain structural similarity to weighted nearest neighbor rules and superficially seem quite different from the “linear” SVMs presented in section 5.3. However the theory of kernel SVMs is essentially the same as the theory of linear SVMs — at a theoretical level all SVMs are linear.

The representer theorem applies to a wide variety of unregularized and L_2 -regularized training equations. Most such training equations, and other learning algorithms can be kernelized.

7.1 Basic Definitions

As before we assume an input space \mathcal{X} and a feature map $\Phi : \mathcal{X} \rightarrow \mathbb{R}^d$. A feature map Φ determines a kernel function $K : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ defined by the following equation for $x, x' \in \mathcal{X}$.

$$K(x, x') = \Phi^T(x)\Phi(x') \quad (7.1)$$

Note that this gives $K(x, x') = K(x', x)$ and $K(x, x) = \|\Phi(x)\|^2 \geq 0$. Kernel methods assume that we are given a kernel function K on \mathcal{X} but not give the feature map Φ . One can then ask what learning methods can be implemented if we have access to K but not to Φ . This is important in cases where $d \gg n$ so that $\Phi(x)$ is difficult or impossible to compute and yet there exists a simple method of computing $K(x, x')$. One example of this is the Gaussian kernel where \mathcal{X} is itself a vector space and $K(x, x')$ can be computed as follows.

$$K(x, x') = e^{-\frac{\|x-x'\|^2}{2\sigma^2}}$$

It is rather surprising that there exists a feature map Φ such that the Gaussian kernel satisfies (7.1). We will prove this in section 7.4. For now we are interested in reformulating the learning algorithms of chapter 5 for the case where we are given a kernel function K but do not have access to the feature map Φ .

We consider again the generic training equation (5.15) which we repeat here for convenience.

$$\begin{aligned} w^* &= \operatorname{argmin}_w \left(\sum_{t=1}^n L(m_t) \right) + \frac{1}{2} \lambda \|w\|^2 \\ m_t &= y_t w^T \Phi(x_t) \end{aligned}$$

Here $L(m)$ can be quadratic loss $(1/2)(1 - m)^2$, or log loss $\ln(1 + e^{-m})$, or hinge loss $\max(0, 1 - m)$. The representer theorem of section 5.4 states that for an arbitrary loss function L , there exists $\alpha_1^*, \dots, \alpha_n^*$ such that we have the following.

$$w^* = \sum_{t=1}^n \alpha_t^* \Phi(x_t)$$

For any vector α with components $\alpha_1, \dots, \alpha_n$ we can define w_α as follows.

$$w_\alpha = \sum_{t=1}^n \alpha_t \Phi(x_t)$$

The basic idea of kernel methods is to reformulate algorithms in terms of the training instance weight vector α rather the feature component weight vector w . First we define $f_\alpha(x)$ as follows.

$$\begin{aligned} f_\alpha(x) &= w_\alpha^T \Phi(x) \\ &= \left(\sum_{t=1}^n \alpha_t \Phi^T(x_t) \right) \Phi(x) \\ &= \sum_{t=1}^n \alpha_t \Phi^T(x_t) \Phi(x) \\ &= \sum_{t=1}^n \alpha_t K(x_t, x) \end{aligned} \tag{7.2}$$

We now have that $f_\alpha(x) = w_\alpha^T \Phi(x)$ can be computed from the weight vector α and the kernel function K without any use of the feature map Φ . The margin m_t can be defined using f_α as follows.

$$\begin{aligned} m_t &= y_t w_\alpha^T \Phi(x) \\ &= y_t f_\alpha(x) \end{aligned} \tag{7.3}$$

Finally we consider the norm $\|w_\alpha\|^2$. To express $\|w_\alpha\|^2$ in terms of the instance weight vector α and the kernel function we first define the kernel matrix K as follows.

$$K_{s,t} = \Phi^T(x_s)\Phi(x_t) \quad (7.4)$$

We can then expression $\|w_\alpha\|^2$ as follows.

$$\begin{aligned} \|w_\alpha\|^2 &= w_\alpha^T w_\alpha \\ &= \left(\sum_{s=1}^T \alpha_s \Phi^T(x_s) \right) \left(\sum_{t=1}^n \alpha_t \Phi(x_t) \right) \\ &= \sum_{s,t} \alpha_s \Phi^T(x_s) \Phi(x_t) \alpha_t \\ &= \alpha^T K \alpha \end{aligned} \quad (7.5)$$

We now put together (7.2), (7.3) and (7.5) to reformulate the general training equation as follows.

$$\alpha^* = \underset{\alpha}{\operatorname{argmin}} \left(\sum_{t=1}^n L(m_t) \right) + \frac{1}{2} \lambda \alpha^T K \alpha \quad (7.6)$$

This training equation is now a well defined optimization problem on the example weight vector α using only the kernel function with no reference to the underlying feature map. Furthermore we have that the margin m_t is linear in α . More specifically we have the following.

$$\begin{aligned} m_t &= y_t f_\alpha(x_t) \\ &= y_t \left(\sum_{s=1}^n \alpha_s K(x_s, x_t) \right) \end{aligned} \quad (7.7)$$

Since m_t is linear in α we get that if $L(m)$ is a convex function of m then $L(m_t)$ is a convex function of the instance weight vector α . Furthermore, (7.5) implies that for any vector α we have $\alpha^T K \alpha = \|w_\alpha\|^2 \geq 0$. This implies that the matrix K is positive semidefinite and hence the term $\alpha^T K \alpha$ is also convex in α . So for any convex loss function L we have that the training equation (7.6) defines a convex optimization problem in the instance weight vector α .

7.2 Kernel Ridge Regression

In the case of square loss (ridge regression) the training equation can be solved in closed form. In the general case of regression, with y_t an arbitrary scalar

value, the kernel training equation can be written as follows.

$$\alpha^* = \operatorname{argmin}_{\alpha} \left(\sum_{t=1}^n \frac{1}{2} (y_t - f_{\alpha}(x_t))^2 \right) + \frac{1}{2} \lambda \alpha^T K \alpha \quad (7.8)$$

Now let y and $f_{\alpha}(x)$ each be a vector indexed by training instance, i.e., the t th component of y is y_t and the t th component of $f_{\alpha}(x)$ is $f_{\alpha}(x_t)$. The vector $f_{\alpha}(x)$ can then be written in terms of the kernel matrix K as follows.

$$f_{\alpha}(x) = K\alpha$$

We can now rewrite the first term in (7.8) as follows.

$$\alpha^* = \operatorname{argmin}_{\alpha} \frac{1}{2} \|y - K\alpha\|^2 + \frac{1}{2} \lambda \alpha^T K \alpha$$

Setting the gradient with respect to α to zero gives the following.

$$\begin{aligned} 0 &= -K(y - K\alpha^*) + \lambda K\alpha^* \\ 0 &= K(K\alpha^* + \lambda\alpha^* - y) \end{aligned}$$

We now assume that the kernel matrix has full rank which then implies the following.

$$(K\alpha^* + \lambda\alpha^* - y) = 0$$

$$\alpha^* = (K + \lambda I)^{-1} y$$

7.3 Kernel SVMs

In the case of hinge loss (the SVM case) the kernel optimization problem (7.6) becomes a convex quadratic program which can be solved with off the shelf quadratic programming packages. To see this we first rewrite (7.6) explicitly for SVMs.

$$\alpha^* = \operatorname{argmin}_{\alpha} \sum_{t=1}^N \max(0, 1 - m_t) + \frac{1}{2} \lambda \alpha^T K \alpha \quad (7.9)$$

$$(7.10)$$

This optimization problem can be reformulated equivalently as follows.

$$\begin{aligned} &\text{minimize} && \sum_{t=1}^n \eta_t + \frac{1}{2} \lambda \|w\|^2 \\ &\text{subject to} && \eta_t \geq 0 \\ &&& \eta_t \geq 1 - m_t \end{aligned} \quad (7.11)$$

Equation (7.3) shows that m_t is a linear function of α . We have also observed earlier that K is a positive semidefinite matrix. These two observations together imply that (7.11) is a convex quadratic program — a minimization problem involving a convex quadratic objective function subject to linear constraints.

7.4 Infinite Dimensional Feature Maps

We now consider the case of $d = \infty$. The space ℓ_2 is defined to be the set of sequences w_1, w_2, w_3, \dots which have finite norm, i.e., where we have the following.

$$\|w\|^2 = \sum_{i=1}^{\infty} w_i^2 < \infty \quad (7.12)$$

We are now interested in regression and classification with infinite dimensional feature vectors and weight parameters. In other words we have $\Phi(x) \in \ell_2$ and $w \in \ell_2$. In practice there is little difference between the infinite dimensional case and the finite dimensional case with $d \gg n$.

Definition: A function K on $\mathcal{X} \times \mathcal{X}$ is called a *kernel function* if there exists a function Φ mapping \mathcal{X} into ℓ_2 such that for any $x_1, x_2 \in \mathcal{X}$ we have that $K(x_1, x_2) = \Phi(x_1) \cdot \Phi(x_2)$.

We will show below that for $x_1, x_2 \in R^q$ the functions $(x_1 \cdot x_2 + 1)^p$ and $\exp(-\frac{1}{2}(x_1 - x_2)^T \Sigma^{-1}(x_1 - x_2))$ are both kernels. The first is called a polynomial kernel and the second is called a Gaussian kernel. The Gaussian kernel is particularly widely used. For the Gaussian kernel we have that $K(x_1, x_2) \leq 1$ where the equality is achieved when $x_1 = x_2$. In this case $K(x_1, x_2)$ expresses a nearness of x_1 to x_2 . When K is a Gaussian kernel we get that $f_\alpha(x)$ as computed by (kernels:eq:f) can be viewed as a classifying x using a weighted nearest neighbor rule where $K(x, x_t)$ is interpreted as giving the “nearness” of x to x_t . Empirically (7.9) works better for setting the weights α_t than other weight setting heuristics for weighted nearest neighbor rules.

7.5 Some Closure Properties on Kernels

Note that any kernel function K must be symmetric, i.e., $K(x_1, x_2) = K(x_2, x_1)$. It must also be positive semidefinite, i.e., $K(x, x) \geq 0$.

If K is a kernel and $\alpha > 0$ then αK is also a kernel. To see this let Φ be a feature map for K . Define Φ_2 so that $\Phi_2(x) = \sqrt{\alpha}\Phi_1(x)$. We then have that $\Phi_2(x_1) \cdot \Phi_2(x_2) = \alpha K(x_1, x_2)$. Note that for $\alpha < 0$ we have that αK is not positive semidefinite and hence cannot be a kernel.

If K_1 and K_2 are kernels then $K_1 + K_2$ is a kernel. To see this let Φ_1 be a feature map for K_1 and let Φ_2 be a feature map for K_2 . Let Φ_3 be the feature

map defined as follows.

$$\Phi_3(x) = f_1(x), g_1(x), f_2(x), g_2(x), f_3(x), g_3(x), \dots$$

$$\Phi_1(x) = f_1(x), f_2(x), f_3(x), \dots$$

$$\Phi_2(x) = g_1(x), g_2(x), g_3(x), \dots$$

We then have that $\Phi_3(x_1) \cdot \Phi_3(x_2)$ equals $\Phi_1(x_1) \cdot \Phi_1(x_2) + \Phi_2(x_1) \cdot \Phi_2(x_2)$ and hence Φ_3 is the desired feature map for $K_1 + K_2$.

If K_1 and K_2 are kernels then so is the product $K_1 K_2$. To see this let Φ_1 be a feature map for K_1 and let Φ_2 be the feature map for K_2 . Let $f_i(x)$ be the i th feature value under feature map Φ_1 and let $g_i(x)$ be the i th feature value under the feature map Φ_2 . We now have the following.

$$\begin{aligned} K_1(x_1, x_2)K_2(x_1, x_2) &= (\Phi_1(x_1) \cdot \Phi_1(x_2))(\Phi_2(x_1) \cdot \Phi_2(x_2)) \\ &= \left(\sum_{i=1}^{\infty} f_i(x_1)f_i(x_2) \right) \left(\sum_{j=1}^{\infty} g_j(x_1)g_j(x_2) \right) \\ &= \sum_{i,j} f_i(x_1)f_i(x_2)g_j(x_1)g_j(x_2) \\ &= \sum_{i,j} (f_i(x_1)g_j(x_1)) (f_i(x_2)g_j(x_2)) \end{aligned}$$

We can now define a feature map Φ_3 with a feature $h_{i,j}(x)$ for each pair $\langle i, j \rangle$ defined as follows.

$$h_{i,j}(x) = f_i(x)g_j(x)$$

. We then have that $K_1(x_1, x_2)K_2(x_1, x_2)$ is $\Phi_3(x_1) \cdot \Phi_3(x_2)$ where the inner product sums over all pairs $\langle i, j \rangle$. Since the number of such pairs is countable, we can enumerate the pairs in a linear sequence to get $\Phi_3(x) \in \ell_2$.

It follows from these closure properties that if p is a polynomial with positive coefficients, and K is a kernel, then $p(K(x_1, x_2))$ is also a kernel. This proves that polynomial kernels are kernels. One can also give a direct proof that if K is a kernel and p is a convergent infinite power series with positive coefficients (an convergent infinite polynomial) then $p(K(x_1, x_2))$ is a kernel. The proof is similar to the proof that a product of kernels is a kernel but uses a countable set of higher order moments as features. The result for infinite power series can then be used to prove that a Gaussian kernel is a kernel. These proofs are homework problems for these notes. Unlike most proofs in the literature, we do not require compactness of the set X on which the Gaussian kernel is defined.

7.6 Hilbert Space

The set ℓ_2 is an infinite dimensional Hilbert space. In fact, all Hilbert spaces with a countable basis are isomorphic to ℓ_2 . So ℓ_2 is really the only Hilbert space we need to consider. But different feature maps yield different interpretations of the space ℓ_2 as functions on \mathcal{X} . A particularly interesting feature map is the following.

$$\Phi(x) = 1, x, \frac{x^2}{\sqrt{2}}, \frac{x^3}{\sqrt{3!}}, \dots, \frac{x^n}{\sqrt{n!}}, \dots$$

Now consider any function f all of whose derivatives exist at 0. Define $w(f)$ to be the following infinite sequence.

$$w(f) = f(0), f'(0), \frac{f''(0)}{\sqrt{2}}, \dots, \frac{f^k(0)}{\sqrt{k!}}, \dots$$

For any f with $w(f) \in \ell_2$ (which is many familiar functions) we have the following.

$$f(x) = w(f) \cdot \Phi(x) \tag{7.13}$$

So under this feature map, the parameter vectors w in ℓ_2 represent essentially all functions whose Taylor series converges. For any given feature map Φ on \mathcal{X} define $\mathcal{H}(\Phi)$ to be the set of functions f from \mathcal{X} to R such that there exists a parameter vector $w(f) \in \ell_2$ satisfying (7.13). Equation (7.6) can then be written as follows where $\|f\|^2$ abbreviates $\|w(f)\|^2$.

$$f^* = \operatorname{argmin}_{f \in \mathcal{H}(\Phi)} \left(\sum_{t=1}^T L(y_t, f(x_t)) \right) + \lambda \|f\|^2$$

This way of writing the equation emphasizes that with a rich feature map selecting w is equivalent to selecting a function from a rich space of functions.

7.7 Problems

1. Let $P(z)$ be an infinite power series (where z is a single real number) with positive coefficients such that $P(z)$ converges for all $z \in R$.

$$P(z) = \sum_{k=0}^{\infty} a_k z^k, \quad a_k \geq 0, \quad P(z) \text{ finite } \forall z$$

Let K be a kernel on a set \mathcal{X} . This problem is to show that that $P(K(x, y))$ is a kernel on \mathcal{X} .

a. Let $\Phi : \mathcal{X} \rightarrow \ell_2$ be the feature map for K . Let s range over all finite sequences of positive integers where $|s|$ is the length of the sequence s and for

$1 \leq j \leq |s|$ we have $s_j \geq 1$ is the j th integer in the sequence s . For $x \in \mathcal{X}$, and s a sequence of indices, let $\Psi_s(x)$ be defined as follows.

$$\Psi_s(x) = \sqrt{a_{|s|}} \prod_{j=1}^{|s|} \Phi_{s_j}(x)$$

We note that the set of all sequences s is countable and hence can be enumerated in a single infinite sequence of sequences. Hence the map Ψ maps x into an infinite series. Show that $\Psi(x) \in \ell_2$, i.e., that $\sum_s \Psi_s^2(x) < \infty$. (Consider $P(K(x, x))$).

b. Show that Ψ is the feature map for $P(K(x, y))$.

2. Here will show that the Gaussian kernel is indeed a kernel. Consider $x, y \in R^d$. The problem is to show that there exists a feature map Ψ , with $\Psi(x), \Psi(y) \in \ell_2$, such that $\exp(-\frac{1}{2}(x-y)^T \Sigma^{-1}(x-y)) = \Psi(x) \cdot \Psi(y)$.

a. Show

$$\exp\left(-\frac{1}{2}(x-y)^T \Sigma^{-1}(x-y)\right) = \exp\left(-\frac{1}{2}x^T \Sigma^{-1}x\right) \exp\left(-\frac{1}{2}y^T \Sigma^{-1}y\right) \exp(x^T \Sigma^{-1}y)$$

c. Show that $x^T \Sigma^{-1}y$ is a kernel in x and y .

b. Show that $\exp(-\frac{1}{2}x^T \Sigma^{-1}x) \exp(-\frac{1}{2}y^T \Sigma^{-1}y)$ is a kernel in x and y (Hint: you only need a single feature.)

c. Use the result of part 1 and a, b, and c, plus the result in the notes that the product of kernels is a kernel, to show that the Gaussian kernel is a kernel.

3. Limits of Kernels: In this problem we show that under quite general conditions a limit of kernels is a kernel. More specifically, we consider kernels K^1, K^2, K^3, \dots such that for any x, x' we have that the limit as i goes to infinity of $K^i(x, x')$ exists (the sequence of kernels is pointwise convergent). To state a condition that implies that a limit of kernels is a kernel we need a little terminology. Let $x_1, x_2, x_3 \dots$ be a fixed countable subset of \mathcal{X} and let Φ be a feature map on \mathcal{X} . For any $x \in \mathcal{X}$ we can define $\tilde{\Phi}_N(x)$ to be the projection of $\Phi(x)$ into the space spanned by $\Phi(x_1), \dots, \Phi(x_N)$ as follows.

$$\tilde{\Phi}_N(x) = \sum_{i=1}^N \alpha_i \Phi(x_i) \tag{7.14}$$

$$\vec{\alpha} = \underset{\vec{\alpha}}{\operatorname{argmin}} \left\| \Phi(x) - \sum_{j=1}^N \alpha_j \Phi(x_j) \right\|^2 \tag{7.15}$$

We say that the set x_1, x_2, x_3, \dots spans feature map $\Phi : \mathcal{X} \rightarrow \ell_2$ if for all $x \in \mathcal{X}$ we have the following.

$$\lim_{N \rightarrow \infty} \|\Phi(x) - \Phi_N(x)\|^2 = 0 \quad (7.16)$$

We will call a kernel K on \mathcal{X} *separable* if there exists a countable subset S of \mathcal{X} , and a feature map Φ for K , such that S spans Φ .

a. Give an example of two different feature maps for the same kernel K (you only need a two dimensional feature map).

b. Show that if a countable subset S of \mathcal{X} spans any feature map for K then it spans all feature maps for K . Hint: Rewrite the optimization problem (7.15) to depend only on the kernel and then express condition (7.16) purely in terms of the kernel. If S spans any (and hence all) feature maps for K we say that S spans K .

c. Show that a pointwise convergent sequence of separable kernels is also a separable kernel. Hint: first show that for any countable set of separable kernels there exists a single countable subset of \mathcal{X} which spans all of the kernels.

Chapter 8

Reducing Dimensionality

In this chapter we consider some basic methods for constructing feature maps of reduced dimensionality. More specifically, we present principal component analysis (PCA), canonical correlation analysis (CCA), k-means clustering, Gaussian clustering, and EM for Gaussian mixtures. All of these methods can be applied to an unlabeled set of points in \mathbb{R}^d . Each method produces a new representation of each point. In the case of PCA and CCA each point is mapped to a lower dimensional point. In the case of clustering each point is represented by a discrete cluster name (a symbolic name). Fitting a mixture model is similar to clustering but represents each point as a distribution over discrete clusters.

8.1 Principal Component Analysis (PCA)

To develop PCA we need some mathematical background.

8.1.1 Covariance and The Central Limit Theorem

Consider a probability density ρ on the real numbers. The mean and variance for this density is defined as follows.

$$\mu = \mathbb{E}_{x \sim \rho}[x] = \int xp(x) dx \quad (8.1)$$

$$\sigma^2 = \mathbb{E}_{x \sim \rho}[(x - \mu)^2] = \int (x - \mu)^2 p(x) dx \quad (8.2)$$

We now generalize mean and variance to dimension larger than 1. Consider a probability density ρ on \mathbb{R}^d . We can define a mean (or average) vector as

follows.

$$\mu = \mathbb{E}_{x \sim \rho} [x] = \int x \rho(x) dx \quad (8.3)$$

$$\mu_i = \mathbb{E}_{x \sim \rho} [x_i] = \int x_i \rho(x) dx \quad (8.4)$$

We now consider a large sample x_1, \dots, x_n drawn IID from ρ and we consider the vector average of these vectors.

$$\hat{\mu} = \frac{1}{n} \sum_{t=1}^n x_t$$

Assuming that the distribution mean μ is well defined (that the integral defining μ converge), we expect that as $n \rightarrow \infty$ we have that $\hat{\mu}$ should approach the distribution average μ .

We now consider the generalization of variance. For variance we are interested in how the distribution varies around its mean. The variance of different dimensions can be different and, perhaps more importantly, the dimensions need not be independent. We define the *covariance matrix* by the following equation with i, j ranging from 1 to d .

$$\Sigma = \mathbb{E}_{x \sim \rho} [(x - \mu)(x - \mu)^T] \quad (8.5)$$

$$\Sigma_{i,j} = \mathbb{E}_{x \sim \rho} [(x_i - \mu_i)(x_j - \mu_j)] \quad (8.6)$$

The definition immediately implies that Σ is symmetric, i.e., $\Sigma_{i,j} = \Sigma_{j,i}$. We also have that Σ is positive semidefinite meaning that $x \Sigma x \geq 0$ for all vectors x . We first give a general interpretation of Σ by noting the following.

$$u \Sigma v = \mathbb{E}_{x \sim \rho} [u(x - \mu)(x - \mu)^T v] \quad (8.7)$$

$$= \mathbb{E}_{x \sim \rho} [(u^T(x - \mu))(v^T(x - \mu))] \quad (8.8)$$

$$(8.9)$$

So for fixed (nonrandom) vectors u and v we have that $u \Sigma v$ is the expected product of the two random variables $u^T(x - \mu)$ and $v^T(x - \mu)$. In particular we have that $u \Sigma u$ is the expected value of $((x - \mu) \cdot u)^2$. This implies that $u \Sigma u \geq 0$. A matrix satisfying this property for all u is called positive semi-definite. The covariance matrix is always both symmetric and positive semi-definite.

8.1.2 The Multivariate Central Limit Theorem

We now consider the standard estimator $\hat{\mu}$ of μ where $\hat{\mu}$ is derived from a sample x_1, \dots, x_n drawn independently according to the density ρ .

$$\hat{\mu} = \frac{1}{n} \sum_{t=1}^n x_t \quad (8.10)$$

Note that $\hat{\mu}$ can have different values for different samples — $\hat{\mu}$ is a random variable. As the sample size increases we expect that $\hat{\mu}$ becomes near μ and hence the length of the vector $\hat{\mu} - \mu$ goes to zero. In fact the length of this vector decreases like $1/\sqrt{n}$. In fact, the probability density for the quantity $\sqrt{n}(\hat{\mu} - \mu)$ converges to a multivariate Gaussian with zero mean covariance equal to the covariance of the density ρ .

Theorem 2. *For any density ρ on \mathbb{R}^d with finite mean and positive semi-definite covariance, and any (measurable) subset U of \mathbb{R}^d we have the following where μ and Σ are the mean and covariance matrix of the density ρ .*

$$\lim_{n \rightarrow \infty} \mathbb{P} [\sqrt{n}(\hat{\mu} - \mu) \in U] = \mathbb{P}_{x \sim \mathcal{N}(0, \Sigma)} [x \in U]$$

$$\mathcal{N}(\nu, \Sigma)(x) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp \left(-\frac{1}{2} (x - \nu)^T \Sigma^{-1} (x - \nu) \right)$$

Note that we need that the the covariance Σ is positive semi-definite, i.e., $x^T \Sigma x > 0$, or else we have $|\Sigma| = 0$ in which case the normalization factor is infinite. If $|\Sigma| = 0$ we simply work in the linear subspace spanned by the non-zero eigenvectors of Σ .

8.1.3 Eigenvectors

An eigenvector of a matrix A is a vector β such that $A\beta = \lambda\beta$ where λ is a scalar called the eigenvalue of β . In general the components of an eigenvector can be complex numbers and the eigenvalues can be complex. However, any symmetric positive semi-definite real-valued matrix Σ has real-valued orthogonal eigenvectors with real eigenvalues.

We can see that two eigenvectors with different eigenvalues must be orthogonal by observing the following for any two eigenvectors β_1 and β_2 with eigenvalues λ_1 and λ_2 respectively.

$$\begin{aligned} \beta_1^T \Sigma \beta_2 &= \lambda_1 (\beta_1^T \beta_2) \\ &= \beta_2^T \Sigma \beta_1 \\ &= \lambda_2 (\beta_2^T \beta_1) \end{aligned}$$

So if $\lambda_1 \neq \lambda_2$ then we must have $\beta_1^T \beta_2 = 0$.

If two eigenvectors have the same eigenvalue then any linear combination of those eigenvectors are also eigenvectors with that same eigenvalue. So for a given eigenvalue λ the set of eigenvectors with eigenvalue λ forms a linear subspace and we can select orthogonal vectors spanning this subspace. So there always exists an orthonormal set of eigenvectors of Σ .

It is often convenient to work in an orthonormal coordinate system where the coordinate axes are eigenvectors of Σ . In this coordinate system we have that Σ is a diagonal matrix with $\Sigma_{i,i} = \lambda_i$, the eigenvalue of coordinate i .

In the coordinate system where the axes are the eigenvectors of Σ the multivariate Gaussian distribution has the following form where σ_i^2 is the eigenvalue of eigenvector i .

$$\mathcal{N}(\nu, \Sigma)(x) = \frac{1}{(2\pi)^{d/2} \prod_i \sigma_i} \exp\left(-\frac{1}{2} \sum_i \frac{(x_i - \nu_i)^2}{\sigma_i^2}\right) \quad (8.11)$$

$$= \prod_i \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{(x_i - \nu_i)^2}{2\sigma_i^2}\right) \quad (8.12)$$

8.1.4 Computing Eigenvectors

To compute an eigenvector of Σ with largest eigenvalue we take a random vector x and compute $\Sigma^M x = \Sigma \Sigma \dots \Sigma x$ with M applications of Σ . Note that x can be written as follows the β^j are orthonormal eigenvectors with eigenvalues λ_j and where $\lambda_j \geq \lambda_h$ for j, h .

$$x = (x^T \beta_1) \beta_1 + \dots + (x^T \beta_d) \beta_d$$

We then get the following.

$$\Sigma^M x = (x^T \beta_1) \lambda_1^M \beta_1 + \dots + (x^T \beta_d) \lambda_d^M \beta_d$$

So the components of x in the directions of the largest eigenvectors grow exponentially in M relative to other components of x . For M large we will have that $\Sigma^M x$ points in the direction of a an eigenvector with largest eigenvalue. We can normalize the vector after each application of Σ to keep things numerically stable. After a first eigenvector has been found we can then work in the subspace orthogonal to that eigenvector to find the second eigenvector and so on.

8.1.5 Principle Component Analysis (PCA)

In PCA we assume a sample x_1, \dots, x_n drawn from ρ . We estimate the mean and covariance matrix of ρ as follows.

$$\hat{\mu} = \frac{1}{n} \sum_{t=1}^n x_t$$

$$\hat{\Sigma} = \frac{1}{n-1} \sum_{t=1}^n (x_t - \hat{\mu})(x_t - \hat{\mu})^T$$

Let β^1, \dots, β^K be G orthonormal eigenvectors of $\hat{\Sigma}$ with the K largest eigenvalues. We can define a “reduced” feature map Ψ with $\Psi(x) \in \mathbb{R}^K$ as follows.

$$\Psi_j(x) = (x - \hat{\mu})^T \beta^j \quad (8.13)$$

The feature map Ψ is the PCA feature map. We can now represent each $x \in \mathbb{R}^d$ by the K dimensional vector $\Psi(x)$ with $K < d$.

PCA gives the least distortion linear map into a lower dimensional space. More specifically, suppose we wish to define a K -dimensional feature vector $\Psi(x)$ by a $K \times D$ matrix A and an offset vector a as follows.

$$\Psi(x) = A\Phi(x) + a \quad (8.14)$$

Suppose we also want to be able to “decompress” a low dimensional feature vector z into an input vector x' using a $D \times K$ “inverse” matrix B and constant b with the decompression given by $x' = Bz + b$. We can define the “optimal” A , a , B and b as follows.

$$A^*, a^*, B^*, b^* = \operatorname{argmin}_{A, a, B, b} \sum_{t=1}^n \|x_t - (B\Psi(x_t) + b)\|^2 \quad (8.15)$$

It can be shown that PCA gives the solution to this problem as follows where β^1, \dots, β^K are orthonormal eigenvectors of $\hat{\Sigma}$ with the K largest eigenvalues.

$$A_{j,i}^* = \beta_i^j \quad (8.16)$$

$$a^* = -A^* \hat{\mu} \quad (8.17)$$

$$B^* = (A^*)^T \quad (8.18)$$

$$b^* = \hat{\mu} \quad (8.19)$$

8.1.6 Eigenfaces

The result of applying PCA to images of faces is known as “eigenfaces”. We can think of an image of a face as a high dimensional vector with one dimension

for every pixel of the image. We can then compute k largest eigenvectors of $\hat{\Sigma}$. Each resulting eigenvector is an “image” — it has a value for each pixel. These images look like faces and are called eigenfaces. An individual image of a face can then be represented by a linear combination of K eigenfaces.

8.2 Singular Value Decomposition (SVD)

SVD is a general method of factoring matrices and of finding low-rank approximation to matrices. We describe a use of SVD in dimensionality reduction similar to PCA.

8.2.1 SVD Machine for Dimensionality Reduction

SVD is can be computed more efficiently than naive PCA when $D \gg N$, i.e., there are many more features than objects in the sample. In this case the time series space has lower dimension (N) than does the feature vector space (dimension D). For the word histogram feature map on documents it is common to have $D \gg N$. Rather than work with the covariance matrix, we work with the Gramm matrix defined as follows.

$$K_{s,t} = \Phi(x_s) \cdot \Phi(x_t) \quad (8.20)$$

$$K = \Phi\Phi^T \quad (8.21)$$

K is also symmetric and positive semi-definite. The eigenvectors of K are time series. Let $\alpha^1, \dots, \alpha^G$ be orthonormal eigenvectors of K with eigenvalues $\lambda_1, \dots, \lambda_G$ respectively. We can define the reduced dimensionality feature map Ψ as follows.

$$\Psi_k(x) = \frac{1}{\sqrt{\lambda_k}} \alpha^k \cdot \Phi\Phi(x) \quad (8.22)$$

$$(\Phi\Phi(x))_t = \Phi(x_t) \cdot \Phi(x) \quad (8.23)$$

Note that equations (8.20), (8.22) and (8.23) imply that $\Psi(x)$ can be computed provided that we can compute $K(x, y) = \Phi(x) \cdot \Phi(y)$ even if we cannot compute the feature vector $\Phi(x)$. In some cases we can have $D = \infty$ but can still compute $K(x, y)$. The reduced feature vector $\Psi(x)$ can still be computed in this case using (8.20), (8.22) and (8.23).

8.2.2 The Similarity with PCA

SVD is closely related to PCA. SVD solves the following optimization problem for a given reduced dimension G where the matrix A must be $G \times D$ and the matrix B must be $N \times G$.

$$A^*, B^* = \operatorname{argmin}_{A, B} \sum_{t=1}^N \|\Phi(x_t) - BA\Phi(x_t)\|^2 \quad (8.24)$$

$$= \operatorname{argmin}_{A, B} \|\Phi - BA\|^2 \quad (8.25)$$

Equation (8.25) suggests that there is a symmetry between Φ and Φ^T . It is possible to solve (8.25) either by computing eigenvectors of $\Phi^T\Phi$ or by computing eigenvectors $\Phi\Phi^T$. We define the correlation matrix $\hat{\Gamma}$ as follows.

$$\hat{\Gamma} = \frac{1}{N} \sum_{t=1}^N \Phi(x) \Phi^T(x) \quad (8.26)$$

$$= \frac{1}{N} \Phi^T \Phi \quad (8.27)$$

Like the covariance matrix, the correlation matrix $\hat{\Gamma}$ is symmetric and positive semi-definite and hence has orthogonal eigenvectors. Let β^1, \dots, β^G be orthonormal eigenvectors of $\hat{\Gamma}$ with the G largest eigenvalues. We will now show that the following definition of $\Psi(x)$ is equivalent to that given above.

$$\Psi_k(x) = \Phi(x) \cdot \beta^k \quad (8.28)$$

Equation (8.28) is the same as PCA except that it uses eigenvectors of the correlation matrix $\hat{\Gamma}$ rather than eigenvectors of the covariance matrix $\hat{\Sigma}$. To show the equivalence of the two definitions of Ψ we first note that if α is an eigenvector of $K = \Phi\Phi^T$ with eigenvalue λ then we have the following.

$$\Phi^T \Phi (\Phi^T \alpha) = \Phi^T (\Phi \Phi^T) \alpha \quad (8.29)$$

$$= \Phi^T \lambda \alpha \quad (8.30)$$

$$= \lambda (\Phi^T \alpha) \quad (8.31)$$

Hence, if α is an eigenvector of $K = \Phi\Phi^T$ then $\Phi^T \alpha$ is an eigenvector of $\hat{\Gamma} = (1/N)\Phi^T\Phi$ with the same eigenvalue. Equation (8.22) is equivalent to $\Psi_k(x) = (1/\sqrt{\lambda_k})(\Phi^T \alpha^k) \cdot \Phi(x)$. This implies that $\Psi_k(x)$ as defined by (8.28) is proportional to $\Psi_k(x)$ as defined by (8.22). The comments in the next subsection imply that if α is a unit norm eigenvector of K then $\Phi^T \alpha$ has norm $\sqrt{\lambda_i}$ which implies that the two definitions are the same. Note that a unit norm eigenvector of $K = (1/N)\Phi\Phi^T$ is the same as a unit norm eigenvector of $\Phi\Phi^T$ — the factor of $1/N$ is removed when we normalize the eigenvector.

8.2.3 SVD as General Matrix Factorization

In general, singular value decomposition is a statement about an arbitrary matrix Φ with arbitrary dimensions $N \times D$. Let G be the rank of Φ which is no larger than the minimum of N and D . In general we can write Φ as $B\Lambda A$ where B is $N \times G$, Λ is $G \times G$, A is $G \times D$ and where the vectors $B_{\cdot,k}$ are orthonormal eigenvectors of $\Phi\Phi^T$, the vectors $A_{k,\cdot}$ are orthonormal nonsingular eigenvectors $\Phi^T\Phi$, and Λ is diagonal with $\Lambda_{k,k} = \sqrt{\lambda_k}$ where λ_k is the eigenvalue of $B_{\cdot,k}$ which is also the eigenvalue of $A_{k,\cdot}$. The optimization problem (8.25) is solved by dropping the eigenvectors of smallest eigenvalue from the G -dimensional intermediate representation.

8.3 Latent Semantic Indexing (LSI)

SVD applied to English documents, with feature vectors defined by word histograms or related word count based feature vectors (such as tf-idf), is called latent semantic indexing (LSI). In this case the feature value $\Psi_k(x_t)$ defined by (8.20), (8.22) and (8.23) gives the component of document x along the “topic” or “concept” dimension k .

8.4 Kernel PCA

Like SVD, Kernel PCA is appropriate for $D \gg N$ as is common with word-based feature vectors for documents. Kernel PCA is SVD applied to the centered data matrix defined as follows.

$$\tilde{\Phi}_{t,i} = \Phi_i(x_t) - \hat{\mu}_i \quad (8.32)$$

For the centered data matrix we have the following.

$$\hat{\Sigma} = \frac{1}{N} \tilde{\Phi}^T \tilde{\Phi} \quad (8.33)$$

$$\tilde{K} = \tilde{\Phi} \tilde{\Phi}^T \quad (8.34)$$

So SVD on $\tilde{\Phi}$ yields a representation of eigenvectors of the covariance matrix $\hat{\Sigma}$ in terms of the eigenvectors of \tilde{K} . The trick is to compute \tilde{K} using only $K(x, y)$, i.e., without computing explicit feature vectors $\Phi(x)$. Note that the (uncentered) Gram matrix K can be computed from $K(x, y)$ alone. One can show that the centered Gram matrix \tilde{K} can be written in terms of K as follows

where $\mathbf{1}$ denotes the $N \times N$ matrix in which every entry has the value $1/N$.

$$\tilde{K}_{s,t} = (\Phi(x_s) - \hat{\mu}) \cdot (\Phi(x_t) - \hat{\mu}) \quad (8.35)$$

$$= K(x_s, x_t) - \hat{\mu} \cdot \Phi(x_t) - \hat{\mu} \cdot \Phi(x_s) + \|\hat{\mu}\|^2 \quad (8.36)$$

$$\hat{\mu} = \frac{1}{N} \sum_{t=1}^N \Phi(x_t) \quad (8.37)$$

$$\hat{\mu} \cdot \Phi(x) = \frac{1}{N} \sum_{t=1}^N \Phi(x_t) \cdot \Phi(x) \quad (8.38)$$

$$= \frac{1}{N} \sum_{t=1}^N K(x, x_t) \quad (8.39)$$

$$\|\hat{\mu}\|^2 = \hat{\mu} \cdot \hat{\mu} \quad (8.40)$$

$$= \frac{1}{N^2} \sum_{t=1}^N \sum_{s=1}^N K(x_t, x_s) \quad (8.41)$$

Equations (8.36), (8.39) and (8.41) allow \tilde{K} to be computed entirely from inner products. Now let $\alpha^1, \dots, \alpha^G$ be G orthonormal eigenvectors of \tilde{K} with the G largest eigenvalues $\sigma_1^2, \dots, \sigma_G^2$. By the arguments given for SVD dimension reduction, the following reduced feature map implements inner products with the principle eigenvectors of $\hat{\Sigma} = (1/N)\tilde{\Phi}^T\tilde{\Phi}$.

$$\Psi_k(x) = \frac{1}{\sigma_k} \alpha^k \cdot \tilde{\Phi}\Phi(x) \quad (8.42)$$

$$(\tilde{\Phi}\Phi(x))_t = (\Phi(x_t) - \hat{\mu}) \cdot \Phi(x) \quad (8.43)$$

$$= K(x_t, x) - \frac{1}{N} \sum_{s=1}^N K(x, x_s) \quad (8.44)$$

Equations (8.42) and (8.44) allow the PCA feature vector $\Psi(x)$ to be computed entirely with inner products.

8.5 Problems

1. Let ρ be the density on \mathbb{R}^2 defined as follows.

$$p(x, y) = \begin{cases} \frac{1}{2} & \text{if } 0 \leq x \leq 2 \text{ and } 0 \leq y \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad (8.45)$$

Give the mean and covariance matrix of this density. Drawn some iso-density contours of the Gaussian with the same mean and covariance as ρ .

2. Consider the following density.

$$p(x, y) = \begin{cases} \frac{1}{2} & \text{if } 0 \leq x + y \leq 2 \text{ and } 0 \leq y - x \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad (8.46)$$

Give the mean of the distribution and the eigenvectors and eigenvalues of the covariance matrix. Hint: draw the density.

Problem 3. This problem is related to kernel PCA. We consider a set of data points x_1, \dots, x_T and a feature map Φ with $\Phi(x) \in \ell_2$ (we are considering infinite dimensional vectors). In kernel methods vectors are represented by linear combinations of sample points. For a series of weights $\alpha_1, \dots, \alpha_T$ we define $\Psi(\alpha) \in \ell_2$ to be the following vector.

$$\Psi(\alpha) = \sum_{t=1}^T \alpha_t \Phi(x_t)$$

Suppose that inner products can be computed easily using a kernel function $\Phi(x_1) \cdot \Phi(x_2) = K(x_1, x_2)$. Let K denote the kernel matrix defined as follows.

$$K_{s,t} = K(x_s, x_t)$$

a. Consider two weight sequences α and β . Derive an expression for $\Psi(\alpha) \cdot \Psi(\beta)$ in terms of the sequences α and β , the data points x_1, \dots, x_T and the kernel function K and should not reference the feature map Φ . Your derivation should start from the definition of $\Psi(\alpha)$ and $\Psi(\beta)$.

Now suppose that kernel PCA has produced n weight series $\alpha^1, \dots, \alpha^n$ defining n vectors Ψ^1, \dots, Ψ^n as follows.

$$\begin{aligned} \Psi^1 &= \Psi(\alpha^1) = \sum_{t=1}^T \alpha_t^1 \Phi(x_t) \\ &\vdots \\ \Psi^n &= \Psi(\alpha^n) = \sum_{t=1}^T \alpha_t^n \Phi(x_t) \end{aligned}$$

Kernel PCA produces orthonormal vectors, i.e., $\|\Psi^i\|^2 = 1$ and $\Psi^i \cdot \Psi^j = 0$ for $i \neq j$. We can now reduce the dimensionality of a new point x by defining the vector “projection” $\pi(\Psi(x))$ as follows.

$$\begin{aligned} \pi(\Phi(x)) &= \sum_{i=1}^n \gamma_i(x) \Psi^i \\ \gamma_i(x) &= \Phi(x) \cdot \Psi^i \end{aligned}$$

b. Derive an expression for $\gamma_i(x)$ in terms of the kernel function, the sample, and the new point x . The vector $\gamma(x) \in R^n$ is the reduced-dimension feature map produced by kernel PCA.

Chapter 9

Hidden Markov Models and Speech Recognition

Here we assume a finite set of hidden (latent) states. We will represent a hidden state by an integer $i, j \in \{1, \dots, S\}$. We also assume a set of observations. We will represent an observation by an integer $k \in \{1, \dots, O\}$ where O is the number of observations. A run of a hidden Markov model consists of a sequence z_1, \dots, z_N of hidden states and a sequence x_1, \dots, x_N of observations where the first hidden state z_1 is generated from a fixed distribution over state and each successor hidden state z_{t+1} is generated stochastically from the preceding hidden state z_t . Each observation x_t is generated stochastically from the hidden state z_t at that time. More formally an HMM has parameters Θ which defines the probability $P_{\Theta}(z_1, \dots, z_N, x_1, \dots, x_N)$ as follows.

$$\Theta = \langle \pi, A, C \rangle$$

$$\pi_i = P(z_1 = s), \quad \sum_{i=1}^S \pi_i = 1$$

$$A_{i,j} = P(z_{t+1} = i | z_t = j), \quad \sum_{i=1}^S A_{i,j} = 1$$

$$C_{k,j} = P(x_t = k | z_t = j), \quad \sum_{k=1}^O C_{k,j} = 1$$

$$P_{\Theta}(z_1, \dots, z_N, x_1, \dots, x_N) = \pi_{z_1} \left(\prod_{t=1}^{N-1} A_{z_t, z_{t+1}} \right) \left(\prod_{t=1}^N C_{x_t, z_t} \right)$$

Applications of HMMs:

- **Speech Recognition.** The hidden states are word positions and the observable tokens are acoustic feature vectors.
- **Part of speech tagging.** The hidden states are the parts of speech (noun, verb, adjective, and so on).
- **DNA sequence analysis.** The hidden states might be protein secondary structure or a position in a homologous sequence.

9.1 Trigram Language Models

Let $\#(w)$ be the number of times that the word w appears in a certain training corpus. Let $\#(w_1w_2)$ be the number of times that the pair of words w_1, w_2 occurs and similarly for $\#(w_1, w_2, w_3)$ for the triple of words w_1, w_2, w_3 . Let N be the total number of word occurrences. A interpolated trigram model predicts the word w_3 following a given pair w_1, w_2 as follows.

$$P(w_3|w_1, w_2) = \lambda_1 \left(\frac{\#(w_1, w_2, w_3)}{\#(w_1, w_2)} \right) + \lambda_2 \left(\frac{\#(, w_2, w_3)}{\#(w_2)} \right) + \lambda_3 \left(\frac{\#(w_3)}{N} \right) \quad (9.1)$$

Here λ_1, λ_2 , and λ_3 are non-negative weights which sum to one:

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

A weighted sum, such as (9.1), where the weights are non-negative and sum to one, is called a *convex combination*. Any convex combination of probability distributions is also a probability distribution. A convex combination of distributions is often called an *interpolated* model. In a trigram language model the weights λ_1, λ_2 and λ_3 are usually taken to depend on some way on the pair w_1, w_2 . This is ok since we can hold w_1, w_2 fixed in defining the conditional distribution $P(w_3|w_1, w_2)$.

A trigram language model defines the hidden states used in standard HMM-based speech recognition. A hidden state is a triple of words w_1, w_2, w_3 together with an index for a position in the last word. For example “we the pe*ople” is a state specifying that the two preceding words were “we” and “the” and that we are currently at the o in the word “people” so we should expect to be hearing an “o” sound. The state transition probabilities can be taken to be the following.

$$T(w_1, w_2, [\alpha_1 \dots \alpha_i * \alpha_{i+1} \alpha_{i+2} \dots \alpha_k] | w_1, w_2, [\alpha_1 \dots \alpha_i * \alpha_{i+1} \alpha_{i+2} \dots \alpha_k]) = 1/2$$

$$T(w_1, w_2, [\alpha_1 \dots \alpha_i \alpha_{i+1} * \alpha_{i+2} \dots \alpha_k] | w_1, w_2, [\alpha_1 \dots \alpha_i * \alpha_{i+1} \dots \alpha_k]) = 1/2$$

$$T(w_2, w_3, *w_4 | w_1, w_2, w_3*) = P(w_4|w_2, w_3)$$

The output probabilities are determined by a “acoustic model” specifying the probability distribution over acoustic feature vectors given the current phoneme of the hidden state. Actually, the distribution on acoustic features is usually taken to depend on a “triphone” — the preceding phoneme, the current phoneme, and the next phoneme. Pentaphones are even used in some systems.

9.2 The Viterbi Algorithm

In typical applications of an HMM we are given x_1, \dots, x_N and must infer z_1, \dots, z_N . The Viterbi algorithm is used to compute the following.

$$\begin{aligned}
z_1^*, \dots, z_N^* &= \operatorname{argmax}_{z_1, \dots, z_N} P_{\Theta}(z_1, \dots, z_N \mid x_1, \dots, x_N) \\
&= \operatorname{argmax}_{z_1, \dots, z_N} P_{\Theta}(z_1, \dots, z_N, x_1, \dots, x_N)
\end{aligned} \tag{9.2}$$

In the Viterbi algorithm we define the following $S \times N$ matrix V .

$$V_{i,t} = \max_{z_1, \dots, z_{t-1}} P(z_1, \dots, z_{t-1}, x_1, \dots, x_{t-1}, z_t = i)$$

Given this definition of the matrix V it is possible to prove the following identities.

$$V_{i,1} = \pi_i \tag{9.3}$$

$$V_{i,t+1} = \max_j V_{j,t} C_{x_t,j} A_{i,j} \tag{9.4}$$

Equation (9.3) follows directly from the definition of V . Equation (9.4) can be derived as follows.

$$\begin{aligned}
V_{i,t+1} &= \max_{z_1, \dots, z_t} P_{\Theta}(z_1, \dots, z_t, x_1, \dots, x_t, z_{t+1} = i) \\
&= \max_{z_1, \dots, z_t} \pi_{z_1} \left(\prod_{s=1}^{t-1} A_{z_s, z_{s+1}} \right) \left(\prod_{s=1}^t C_{x_s, z_s} \right) A_{i, z_t} \\
&= \max_{z_1, \dots, z_{t-1}} \max_j \pi_{z_1} \left(\prod_{s=1}^{t-2} A_{z_s, z_{s+1}} \right) \left(\prod_{s=1}^{t-1} C_{x_s, z_s} \right) A_{j, z_{t-1}} C_{x_t, j} A_{i, j} \\
&= \max_j \max_{z_1, \dots, z_{t-1}} \left(\pi_{z_1} \left(\prod_{s=1}^{t-2} A_{z_s, z_{s+1}} \right) \left(\prod_{s=1}^{t-1} C_{x_s, z_s} \right) A_{j, z_{t-1}} \right) C_{x_t, j} A_{i, j} \\
&= \max_j \left(\max_{z_1, \dots, z_{t-1}} \pi_{z_1} \left(\prod_{s=1}^{t-2} A_{z_s, z_{s+1}} \right) \left(\prod_{s=1}^{t-1} C_{x_s, z_s} \right) A_{j, z_{t-1}} \right) C_{x_t, j} A_{i, j} \\
&= \max_j V_{j,t} C_{x_t, j} A_{i, j}
\end{aligned} \tag{9.5}$$

We note that (9.3) and (9.4) provide a way of computing the matrix V starting by using (9.3) to compute $V_{\cdot,1}$ and then using (9.4) to compute $V_{\cdot,t+1}$ from $V_{\cdot,t}$. We can then compute z_t^* backward from $t = N$ as follows.

$$z_N^* = \operatorname{argmax}_i V_{i,N} C_{x_N,i}$$

$$z_{t-1}^* = \operatorname{argmax}_i V_{i,t-1} C_{x_{t-1},i} A_{z_t^*,i}$$

Rather than compute best predecessors backward in this way, we can record the best predecessor of for each pair $\langle i, t \rangle$ during the forward computation of the matrix $V_{i,t}$.

9.3 The Forward-Backward Procedure

We now consider the following problem.

$$z_t^* = \operatorname{argmax}_i P_{\Theta}(z_t = i \mid x_1, \dots, x_N) \quad (9.6)$$

It is important to note that z_t^* as defined by (9.6) is different from z_t^* as defined by (9.2). To see this consider a case where $N = 3$ and $S = 100$ and the state transition matrix has the property that 99 states deterministically transition into state 1 and state 1 uniformly transitions into one of the 99 other states. Suppose that the initial distribution π is such that $P(z_1 = 1) = 1/3$ with the remaining probability uniformly distributed among the other states. Suppose that all observations are equally likely for all states so that the observations have no influence on the state probabilities. In this case the most likely sequence has $z_1 = 1$ and $z_2 \neq 1$. So for z_t^* as defined by (9.2) we have $z_2^* \neq 1$. But $P(x_2 = 1) = 2/3$ so for z_t^* as defined by (9.6) we have $z_2^* = 1$.

Consider the case where we generate z_1, \dots, z_N and x_1, \dots, x_N from the distribution P_{Θ} and then have to guess z_1, \dots, z_n given only x_1, \dots, x_N . There are two different ways that the guess might be scored. In the first way, which we will call 01-loss, the guess is scored correct only if the entire state sequence is correct. If we are to be scored in this way then the optimal guess is given by (9.2). The second method of scoring counts the number of times t where our guess for z_t is correct. This score is one minus the hamming distance from our guess to the actual hidden state sequence where the Hamming distance is just the number of times where the guess is different from the actual. This will be called Hamming loss. If we are to be scored by hamming loss then the optimal guess is given by (9.6). Note that the sequence defined by (9.6) can be extremely unlikely and can even contain impossible state stansition — a time t where $A_{z_t^*, z_{t+1}^*} = 0$.

We can solve (9.6) using the forward-backward procedure. As in the case of the Viterbi algorithm, we first define matrices and then derive an efficient

way of computing them. We define the matrices F (for forward) and B (for backward) as follows.

$$\begin{aligned} F_{i,t} &= P_{\Theta}(x_1, \dots, x_{t-1}, z_t = s) \\ B_{i,t} &= P_{\Theta}(x_t, \dots, x_N \mid z_t = s) \end{aligned}$$

From these definitions we can prove the following equations.

$$F_{i,1} = \pi_i \quad (9.7)$$

$$F_{i,t+1} = \sum_{j=1}^S F_{j,t} C_{x_t,j} A_{i,j} \quad (9.8)$$

$$B_{i,N} = C_{x_n,i} \quad (9.9)$$

$$B_{i,t-1} = \sum_{j=1}^S C_{x_{t-1},i} A_{j,i} B_{j,t} \quad (9.10)$$

Equation (9.7) can be used to compute $F_{\cdot,1}$ and equation (9.8) can be used to compute $F_{\cdot,t+1}$ from $F_{\cdot,t}$. Equation (9.7) follows directly from the definition of F . Equation (9.8) can be derived as follows.

$$\begin{aligned} F_{i,t+1} &= P(x_1, \dots, x_t, z_{t+1} = i) \\ &= \sum_{j=1}^S P(x_1, \dots, x_t, z_t = j, z_{t+1} = i) \\ &= \sum_{j=1}^S P(x_1, \dots, x_{t-1}, z_t = j) C_{x_t,j} A_{i,j} \\ &= \sum_{j=1}^S F_{j,t} C_{x_t,j} A_{i,j} \end{aligned}$$

Equation (9.9) can be used to compute $B_{\cdot,N}$ and equation (9.10) can be used to compute $B_{\cdot,t-1}$ from $B_{\cdot,t}$. Equation (9.9) follows directly from the definition of B . Equation (9.10) can be derived as follows.

$$\begin{aligned}
B_{i,t-1} &= P(x_{t-1}, x_t, \dots, x_N \mid z_{t-1} = i) \\
&= \sum_{j=1}^S P(z_t = j, x_{t-1}, x_t, \dots, x_N \mid z_{t-1} = i) \\
&= \sum_{j=1}^S P(z_t = j, x_{t-1} \mid z_{t-1} = i) P(x_t, \dots, x_N \mid z_{t-1} = i, z_t = j, x_{t-1}) \\
&= \sum_{j=1}^S C_{x_{t-1}, i} A_{j,i} P(x_t, \dots, x_N \mid z_t = j,) \\
&= \sum_{j=1}^S C_{x_{t-1}, i} A_{j,i} B_{j,t} \\
&= C_{x_{t-1}, i} \sum_{j=1}^S A_{j,i} B_{j,t}
\end{aligned}$$

We can now compute z_t^* as defined by (9.6) using the following.

$$\begin{aligned}
P(z_t = i \mid x_1, \dots, x_N) &= \frac{F_{i,t} B_{i,t}}{P(x_1, \dots, x_N)} \quad (9.11) \\
P(x_1, \dots, x_N) &= \sum_{j=1}^S \pi_j B_{j,1}
\end{aligned}$$

9.4 Problems

1. Suppose that we have two hidden states ($S = 2$), two observations ($O = 2$), and Θ is given as follows.

$$\begin{aligned}
\pi_1 &= \pi_2 = \frac{1}{2} \\
A_{1,1} &= A_{2,2} = 1 - \epsilon \\
A_{2,1} &= A_{1,2} = \epsilon \\
C_{1,1} &= C_{2,2} = 1 - \delta \\
C_{2,1} &= C_{1,2} = \delta
\end{aligned}$$

Now suppose that we observe x_1, \dots, x_N with $x_t = 1$ for all $1 \leq t \leq N$.

- Give values for $F_{1,1}$ and $F_{2,1}$.
- Give expressions for $F_{1,t+1}$ and $F_{2,t+1}$ in terms of $F_{1,t}$, $F_{2,t}$, ϵ and δ .

c. Give expressions for $B_{1,N}$ and $B_{2,N}$ in terms of δ .

d. Give expressions for $B_{1,t-1}$ and $B_{2,t-1}$ in terms of $B_{1,t}$, $B_{2,t}$, ϵ and δ .

Extra Credit: Give closed form solutions for $F_{1,t}$, $F_{2,t}$, $B_{1,t}$ and $B_{2,t}$ as functions of t , ϵ and δ . Use your answers to give a closed form solution for $P(z_t = 1 \mid x_1, \dots, x_N)$ as a function of t .

2. Give an expression for $P_{\Theta}(z_t = i, z_{t+1} = j \mid x_1, \dots, x_N)$ in terms of the matrices A and C , the forward value $F_{i,t}$, the backward value $B_{j,t+1}$ and the observed data probability $P_{\Theta}(x_1, \dots, x_N)$. Hint: the answer is similar to (9.11).

Problem 3. A hierarchical hidden Markov model (HMM) has a hierarchy of hidden states. A three layer model has a top level state that changes slowly, a second level state that changes more rapidly and with state transitions depending on the higher level state, and a third level state that changes more rapidly than the second level states with transition probabilities that depend on both higher level states. The state transition probabilities at higher levels do not depend on the lower level states. Furthermore, when a higher level state changes between time t and $t+1$ it causes a “reinitialization” of all lower states so that the lower level states at time $t+1$ do not depend on their values at time t .

a) Implement a three level hierarchical hidden Markov model as traditional (one-level) hidden Markov model. More specifically, define the state space, emission probabilities and transition probabilities of the one-level hidden Markov model implementing the three level model.

b) Implement a three level hierarchical HMM as a probabilistic context free grammar.

Problem 5. A Dynamical Bayesian network (DBN) is a special case of an HMM where each hidden state variable is replaced by a set of hidden variables at that point in time. We let n be the number of hidden variables at one point in time. In a DBN the state transition probability is given by a Bayesian network (see the figure on the white board). We can “unroll” a DBN for T time steps into one large Bayesian network with nT hidden variables and T observed variables. We also assume that each observation variable for time i has all n state variables for time i as parents. This unrolled DBN defines a hypergraph whose nodes are the state and observation variables and where there is one hyperedge for each variable consisting of that variable and its parents.

a) Give an upper bound on the tree width of the unrolled DBN as a function of n and T .

b) Assuming that the Bayesian network defining $P(X_{i_1} = \sigma \mid X_i = \gamma)$ has width w , give the width of the unrolled network as a function of n , w and T .

Chapter 10

Expectation Maximization (EM)

The Expectation Maximization (EM) algorithm is one approach to unsupervised, semi-supervised, or lightly supervised learning. In this kind of learning either no labels are given (unsupervised), labels are given for only a small fraction of the data (semi-supervised), or incomplete labels are given (lightly supervised). Completely unsupervised learning is believed to take place in human babies. Babies learn to divide continuous speech into words. There is significant experimental evidence that babies can do this simply by listening to tapes of speech. Computer systems are not yet able to learn to do speech recognition simply by listening to speech sound. In practice the EM algorithm is most effective for lightly supervised data. For example, the text in closed caption television is a light labeling of the television speech sound. Although the sequence of words is given, the alignment between the words and the sound is not given. The text-to-speech alignment can be inferred by EM. Another example of partially labeled data is translation pairs, for example English text paired with a French translation. Translation pairs are used to train machine translation systems. Although the translation is given, the alignment between the words is not (and the word order is different in different languages). Word alignment can be inferred by EM. Although EM is most useful in practice for lightly supervised data, it is more easily formulated for the case of unsupervised learning.

10.1 Hard EM

Here we assume a model with parameters Θ defining probabilities of the form $P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$.¹ In a mixture of Gaussian model we have that each x_t is a point in R^D and each z_t is a class label with $z_t \in \{1, \dots, K\}$. Gaussian mixture models are defined in the next section. In an HMM we have that x_1, \dots, x_N is a sequence of observations and z_1, \dots, z_N is a sequence of hidden states. Alternatively, for an HMM we could take each x_t to be an observation sequence (such as a protein amino acid sequence) and each z_t a hidden state sequence (such a sequence of secondary structure classifications). It is more common with HMMs to consider one very long sequence where each x_t is a single observation of the underlying model. For a PCFG we typically have that x_1, \dots, x_N is a sequence of strings and z_1, \dots, z_n is a sequence of parse trees where z_t is a parse tree for the string x_t .

We now consider an arbitrary model defining $P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n)$. In unsupervised training we are given only x_1, \dots, x_n and, given only this information, we set the parameter values Θ of the model. Hard EM approximatey solves the following optimization problem.

$$\Theta^* = \operatorname{argmax}_{\Theta} \max_{z_1, \dots, z_n} P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n) \quad (10.1)$$

A local optimum of (10.1) can be found by coordinate ascent. We wish to find values of the two “coordinates” Θ and (z_1, \dots, z_n) maximizing $P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n)$. In coordinate ascent we alternately optimize each coordinate holding the other coordinates fixed. Hard EM is the following coordinate ascent algorithm.

1. Initialize Θ (the initialization is important but application specific).
2. Repeat the following until $P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n)$ converges.
 - (a) $(z_1, \dots, z_N) := \operatorname{argmax}_{z_1, \dots, z_N} P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$
 - (b) $\Theta := \operatorname{argmax}_{\Theta} P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$

10.2 K -Means Clustering as an Example of Hard EM

K -means clustering is a special case of hard EM. In K -means clustering we consider sequences x_1, \dots, x_n and z_1, \dots, z_N with $x_t \in R^D$ and $z_t \in \{1, \dots, K\}$.

¹We view $P_{\Theta}(x, z)$ as a special case with $N = 1$. Alternatively, one can view $P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$ as a special case of $P_{\Theta}(x, z)$ where x is the sequence x_1, \dots, x_N and z is the sequence z_1, \dots, z_N . All the examples we consider have the form $P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$ and so it seems natural to keep sequences throughout the formulation.

In other words, z_t is a class label, or cluster label, for the data point x_t . We can define a K -means probability model as follows where $\mathcal{N}(\mu, I)$ denotes the D -dimensional Gaussian distribution with mean $\mu \in R^D$ and with the identity covariance matrix.

$$\begin{aligned}\Theta &= \langle \mu^1, \dots, \mu^K \rangle, \mu^k \in R^D \\ P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n) &= \prod_{t=1}^N P_{\Theta}(z_t) P_{\Theta}(x_t | z_t) \\ &= \prod_{t=1}^N \frac{1}{K} \mathcal{N}(\mu^{z_t}, I)(x_t)\end{aligned}$$

We now consider the optimization problem defined by (10.1) for this model. For this model one can show that (10.1) is equivalent to the following.

$$(\mu^1, \dots, \mu^K)^* = \underset{\mu^1, \dots, \mu^K}{\operatorname{argmin}} \min_{z_1, \dots, z_n} \sum_{t=1}^N \|\mu^{z_t} - x_t\|^2 \quad (10.2)$$

The optimization problem (10.2) defines K -means clustering (under quadratic distortion). This problem is nonconvex and in fact is NP-hard (worse than nonconvex). The K means algorithm is coordinate descent applied to this objective and is equivalent to hard EM under the above probability model. The K -means clustering algorithm can be written as follows where we specify a typical initialization step.

1. Initialize μ^z to be equal to a randomly selected point x_t .
2. Repeat the following until (z_1, \dots, z_n) stops changing.
 - (a) $z_t := \operatorname{argmin}_z \|\mu^z - x_t\|^2$
 - (b) $N_z := |\{t : z_t = z\}|$
 - (c) $\mu^z := \frac{1}{N_z} \sum_{t: z_t=z} x_t$

In words, the K -means algorithm first assigns a class center μ^z for each class z . It then repeatedly classifies each point x_t as belonging to the class whose center is nearest x_t and then recomputes the class centers to be the mean of the point placed in that class. Because it is a coordinate descent algorithm for (10.2), the sum of squares of the difference between each point and its class center is reduced by each update. This implies that the classification must eventually stabilize. The procedure terminates when the class labels stop changing.

10.3 Hard EM for Mixtures of Gaussians

Again we consider sequences x_1, \dots, x_n and z_1, \dots, z_n with $x_t \in R^D$ and $z_t \in \{1, \dots, K\}$. Now however we consider a probability model that is a mixture of

Gaussians including mixture weights and covariance matrices.

$$\Theta = \langle \pi^1, \dots, \pi^K, \mu^1, \dots, \mu^K, \Sigma^1, \dots, \Sigma^K \rangle$$

$$P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n) = \prod_{t=1}^n P_{\Theta}(z_t) P_{\Theta}(x_t | z_t)$$

$$= \prod_{t=1}^n \pi^{z_t} \mathcal{N}(\mu^{z_t}, \Sigma^{z_t})(x_t)$$

Again we can consider the optimization problem defined by (10.1) for this model. It can now be shown that step 2b of the hard EM is equivalent to the following.

$$N^k = |\{t : z_t = k\}|$$

$$\pi^k = N^k / N$$

$$\mu^k = \frac{1}{N^k} \sum_{t: z_t^* = k} x_t$$

$$\Sigma^k = \frac{1}{N^k} \sum_{t: z_t^* = k} (x_t - \mu^k)(x_t - \mu^k)^T \quad (10.3)$$

10.4 An Informal Statement of General Soft EM

Again we assume a model with parameters Θ defining probabilities of the form $P_{\Theta}(x_1, \dots, x_N, z_1, \dots, z_N)$. Again we are given only x_1, \dots, x_n and, given only this information, we set the parameter values Θ of the model. Soft EM approximately solves the following optimization problem.

$$\Theta^* = \operatorname{argmax}_{\Theta} \sum_{z_1, \dots, z_n} P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n) \quad (10.4)$$

$$= P_{\Theta}(x_1, \dots, x_N) \quad (10.5)$$

One should compare (10.4) with (10.1) and note that the objective in hard EM is different from the objective in soft EM. Which objective is more appropriate can depend on how the model is to be used. If the model is used to infer the most likely value of z_1, \dots, z_N , as in speech recognition, then (10.1) seems more appropriate than (10.4). But if the model is to be used for computing $P_{\Theta}(x_1, \dots, x_N)$ then (10.4) seems more appropriate. Soft EM can be described informally as the following iterative improvement algorithm for the objective given in (10.4).

1. Initialize Θ (the initialization is important but application specific).
2. Repeat the following until $P_{\Theta}(x_1, \dots, x_n, z_1, \dots, z_n)$ converges.
 - (a) $\rho(z_1, \dots, z_N) := P_{\Theta}(z_1, \dots, z_N | x_1, \dots, x_N)$ (The E step.)
 - (b) $\Theta :=$ The best fit of Θ to x_1, \dots, x_N and the distribution ρ on z_1, \dots, z_N . (The M step.)

10.5 Soft EM for Mixtures of Gaussians

Again consider the mixture of Gaussian model defined in section 10.3. We can implement step 2a (the E step) by computing probabilities (or weights) p_t^k as follows.

$$\begin{aligned}
 p^k &= P_{\Theta_w}(z_t = k | x_t) \\
 &= \frac{\pi_w^k \mathcal{N}(\mu_w^k, \Sigma_w^k)(x_t)}{\sum_{k=1}^K \pi_w^k \mathcal{N}(\mu_w^k, \Sigma_w^k)(x_t)}
 \end{aligned}$$

Given the numbers (the matrix) p_t^k , step 2b is then implemented as follows.

$$\begin{aligned}
 N^k &= \sum_{t=1}^N p_t^k \\
 \pi^k &= \frac{N^k}{N} \\
 \mu_{w+1}^k &= \frac{1}{N^k} \sum_{t=1}^N p_t^k x_t \\
 \Sigma_{w+1}^k &= \frac{1}{N^k} \sum_{t=1}^N p_t^k (x_t - \mu^k)(x_t - \mu^k)^T
 \end{aligned}$$

It is instructive to compare these update equations with the update equations of section 10.3.

10.6 Soft EM for HMMs

EM for HMMs is called the Baum-Welch algorithm. The Baum-Welch algorithm predates the general formulation of EM — most special cases of EM predate the general formulation. Recall that in an HMM we have $x_t \in \{1, \dots, O\}$ and $z_t \in \{1, \dots, S\}$. An HMM is defined as follows where π is a S -dimensional

vector, A is a $S \times S$ hidden state transition probability matrix, and C is an $O \times S$ emission probability matrix.

$$\Theta = \langle \pi, A, C \rangle$$

$$P_{\Theta}(z_1, \dots, z_N, x_1, \dots, x_N) = \pi_{z_1} \left(\prod_{t=1}^{N-1} A_{z_{t+1}, z_t} \right) \left(\prod_{t=1}^N C_{x_t, z_t} \right)$$

Here we will use superscript indices for model generation so that we are computing Θ^{w+1} from Θ^w . Given a model Θ^w , and a sequence x_1, \dots, x_n , we can use the forward-backward procedure to compute the following two matrices.

$$\begin{aligned} F_{i,t} &= P_{\Theta^w}(x_1, \dots, x_{t-1}, z_t = i) \\ B_{i,t} &= P_{\Theta^w}(x_t, \dots, x_N \mid z_t = i) \end{aligned}$$

We will also use the following matrix computable from F and B .

$$\begin{aligned} P_{i,t} &= P_{\Theta^w}(z_t = i \mid x_1, \dots, x_N) \\ &= \frac{F_{i,t} B_{i,t}}{P(x_1, \dots, x_N)} \\ P(x_1, \dots, x_N) &= \sum_{i=1}^S F_{i,t} B_{i,t} \end{aligned}$$

The model Θ^{w+1} is then computed as follows.

$$\begin{aligned} \pi_i^{w+1} &= P_{i,1} \\ C_{k,i}^{w+1} &= \frac{\sum_{t: x_t=k} P_{i,t}}{\sum_{t=1}^N P_{i,t}} \\ A_{i,j}^{w+1} &= \frac{\sum_{t=1}^{N-1} P_{\Theta^w}(z_t = j \wedge z_{t+1} = i \mid x_1, \dots, x_N)}{\sum_{t=1}^{N-1} P_{j,t}} \\ &= \frac{\sum_{t=1}^{N-1} F_{j,t} C_{x_t, j}^w A_{i,j}^w B_{j,t}}{P(x_1, \dots, x_N) \sum_{t=1}^{N-1} P_{j,t}} \end{aligned}$$

Intuition into these equations can be gained by thinking about the distribution over z_1, \dots, z_N as training data. Each entry in the model Θ^{w+1} can be viewed as a conditional probability $P(\Phi \mid \Psi)$ which is being set to a “count” of $\Phi \wedge \Psi$ divided by a “count” of Ψ . In each case the count is an *expected* number of occurrences. Hence the term “expectation” for the E-step of EM.

10.7 A Formal Treatment of General Soft EM

Here we let x abbreviate (x_1, \dots, x_n) and let z abbreviate (z_1, \dots, z_n) . More generally we can consider any model $P_\Theta(x, z)$ on a pair of variables x and z . Soft EM is the following iterative improvement algorithm for the objective given in (10.4).

1. Initialize Θ (the initialization is important but application specific).
2. Repeat the following until $P_\Theta(x, z)$ converges.
 - (a) $\rho(z) := P_\Theta(z | x)$
 - (b) $\Theta := \operatorname{argmax}_\Theta \mathbb{E}_{z \sim \rho} [\ln P_\Theta(x, z)]$

First we give an intuition as to why step 2b in the formal soft EM algorithm corresponds to step 2b in the informal version. Step 2b of the informal version can now be stated as “fit Θ to x and ρ ”. Intuitively, we can think of fitting Θ to x and ρ as fitting Θ to a very large sample $(x, z_1), \dots, (x, z_M)$ where each z_i is drawn at random from ρ . For such a sample we have the following.

$$\begin{aligned}
 P_\Theta((x, z_1), \dots, (x, z_M)) &= \prod_{i=1}^M P_\Theta(x, z_i) \\
 \ln P_\Theta((x, z_1), \dots, (x, z_M)) &= \sum_{i=1}^M \ln P_\Theta(x, z_i) \\
 &= \sum_z \operatorname{count}(z) \ln P(x, z) \\
 \frac{1}{M} \ln P_\Theta((x, z_1), \dots, (x, z_M)) &= \frac{\operatorname{count}(z)}{M} \ln P(x, z) \\
 &\approx \sum_z \rho(z) \ln P(x, z) \\
 &= \mathbb{E}_{z \sim \rho} [\ln P(x, z)]
 \end{aligned}$$

$$\operatorname{argmax}_\Theta P_\Theta((x, z_1), \dots, (x, z_M)) \approx \operatorname{argmax}_\Theta \mathbb{E}[z \sim \rho] \ln P_\Theta(x, z)$$

So we should think of the distribution ρ as providing a “virtual sample” to which we fit Θ in step 2b.

We now show that the update 2b improves the objective function (10.4) unless we are already at a local optimum. Here we consider an update starting at the parameter vector Θ and ending in the parameter vector $\Theta + \epsilon$. Step 2b can be defined as follows.

$$\epsilon = \operatorname{argmax}_{\epsilon} Q(\Theta + \epsilon) \quad (10.6)$$

$$Q(\Psi) = \mathbb{E}_{z \sim \rho} [\ln P_{\Psi}(x, z)] \quad (10.7)$$

$$\rho(z) = P_{\Theta}(z|x) \quad (10.8)$$

We now write $P_{\Theta}(\cdot | x)$ for the distribution on z defined by $P_{\Theta}(z|x)$ and show the following.

$$\begin{aligned} \ln P_{\Theta+\epsilon}(x) &= \ln P_{\Theta}(x) + Q(\Theta + \epsilon) - Q(\Theta) + KL(P_{\Theta}(\cdot|x), P_{\Theta+\epsilon}(\cdot|x)) \\ &\geq \ln P_{\Theta}(x) + Q(\Theta + \epsilon) - Q(\Theta) \end{aligned} \quad (10.9)$$

Proof:

$$\begin{aligned} Q(\Theta + \epsilon) - Q(\Theta) &= \mathbb{E}_{z \sim \rho} \left[\ln \frac{P_{\Theta+\epsilon}(x, z)}{P_{\Theta}(x, z)} \right] \\ &= \mathbb{E}_{z \sim \rho} \left[\ln \frac{P_{\Theta+\epsilon}(x) P_{\Theta+\epsilon}(z|x)}{P_{\Theta}(x) P_{\Theta}(z|x)} \right] \\ &= \mathbb{E}_{z \sim \rho} \left[\ln \frac{P_{\Theta+\epsilon}(x)}{P_{\Theta}(x)} \right] + \mathbb{E}_{z \sim \rho} \left[\ln \frac{P_{\Theta+\epsilon}(z|x)}{P_{\Theta}(z|x)} \right] \\ &= \ln \frac{P_{\Theta+\epsilon}(x)}{P_{\Theta}(x)} - KL(P_{\Theta}(\cdot|x), P_{\Theta+\epsilon}(\cdot|x)) \end{aligned}$$

Formula (10.10) gives a differentiable lower bound on a differentiable objective function to be maximized. Furthermore, the lower bound equals the objective function at the “current point” Θ . We can now make a very general observation. For any differentiable lower bound on a differentiable objective function where the bound equals the objective at the current point, and where the gradient of the objective is nonzero at the current point, maximizing the lower bound will strictly improve the objective. To see this note that at the current point the gradient of the bound must equal the gradient of the objective. Hence the lower bound itself can be strictly improved. At any point where the lower bound is larger, the objective must also be larger. The EM update corresponds to maximizing the lower bound in (10.10).

10.8 Problems

1. Suppose that we want to cluster elements of \mathcal{X} into one of K classes using a single feature vector $\Phi(x)$ where each feature is either 0 or 1, i.e., for each feature i and $x \in \mathcal{X}$ we have $\Phi_i(x) \in \{0, 1\}$. The naive Bayes model is a

generative model for generating pairs $\langle x, y \rangle$ with $x \in \mathcal{X}$ and y a class label, i.e., $y \in \{1, \dots, K\}$. The naive Bayes model can be defined by the following equations.

$$\begin{aligned}\beta_j &= P(z = j) \\ \beta_{i,j} &= P(\Phi_i(x) = 1 \mid z = j) \\ P_\beta(x, z) &= \beta_z \prod_i \begin{cases} \beta_{i,z} & \text{if } \Phi_i(x) = 1 \\ (1 - \beta_{i,z}) & \text{if } \Phi_i(x) = 0 \end{cases}\end{aligned}$$

Suppose that we have a sample D of unlabeled values x_1, \dots, x_n from \mathcal{X} .

- a. Give equations for $P_\beta(y_t = j | x_t)$ assuming the model parameters β are given.
- b. Specify the hard EM algorithm for this model. Give an initialization that you think is reasonable.
- c. Specify the soft EM algorithm for this model. Again given an initialization that you think is reasonable.

2. Consider a Bayesian network with structure $X \leftarrow Y \rightarrow Z$ where each of X , Y , and Z take values from the finite sets \mathcal{X} , \mathcal{Y} and \mathcal{Z} respectively. This network has the following parameters where $x \in \mathcal{X}$, $y \in \mathcal{Y}$ and $z \in \mathcal{Z}$.

$$\begin{aligned}\Pi_y &= P(Y = y) \\ R_{x,y} &= P(X = x \mid Y = y) \\ L_{z,y} &= P(Z = z \mid Y = y)\end{aligned}$$

Give both the hard and soft EM algorithms for this model.

- 3.** Give the hard EM algorithm for PCFGs (using the Viterbi parsing algorithm) and the soft EM algorithm for PCFGs (using the inside-outside algorithm).
- 4.** Suppose we initialize the PCFG to be deterministic in the sense that for any word string there is at most one parse of that string with nonzero probability and suppose that each x_h does have a parse under the initial grammar. How rapidly does EM converge in this case? What does it converge to? Justify your answer.
- 5.** Let γ be a distribution on $\{1, \dots, S\}$. In other words we have the following.

$$\begin{aligned}\gamma_i &\geq 0 \\ \sum_{i=1}^S \gamma_i &= 1\end{aligned}\tag{10.11}$$

Suppose that we want to fit a distribution (or model) π to the distribution γ . Intuitively, the best fit is $\pi = \gamma$. But the general EM update fits using a formula similar to the following.

$$\pi^* = \operatorname{argmax}_{\pi} \mathbb{E}_{j \sim \gamma} [\ln \pi_j]$$

Show that $\pi^* = \gamma$. Hint: express the problem as a minimization of cross entropy and use the fact that cross entropy is minimized when the distributions are equal, or perhaps better known, that $KL(P, Q) \geq 0$ and $KL(P, Q) = 0$ only if $P = Q$. Alternatively, you can use Lagrange multipliers and KKT conditions.

6. Let ρ be a distribution (density) on the real numbers R . Suppose that we want to fit a Gaussian with mean μ and standard deviation σ to the distribution ρ using the following “distribution fit equation”.

$$\begin{aligned} \mu^*, \Sigma^* &= \operatorname{argmax}_{\mu, \Sigma} \mathbb{E}_{x \sim \rho} [\ln \mathcal{N}(\mu, \sigma)(x)] \\ \mathcal{N}(\mu, \sigma)(x) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x - \mu)^2}{2\sigma^2}\right) \end{aligned}$$

Show that $\mu^* = \mu_{\rho} = \mathbb{E}_{x \sim \rho} [x]$ and $\sigma^{*2} = \sigma_{\rho}^2 = \mathbb{E}_{x \sim \rho} [(x - \mu_{\rho})^2]$.

Chapter 11

Graphical Models

11.1 Bayesian Networks

We will use capital letters for random variables and lower case letters for values of those variables. A Bayesian network is a triple $\langle V, G, \mathcal{P} \rangle$ where V is a set of random variables X_1, \dots, X_n , G is a directed acyclic graph (DAG) whose nodes are the variables in V , and \mathcal{P} is a set of conditional probability tables as described below. The conditional probability tables determine a probability distribution over the values of the variables. If there is a directed edge in G from X_j to X_i then we will say that X_j is a parent of X_i and X_i is a child of X_j . To determine values for the variables one first selects values for variables that have no parents and then repeatedly picks a value for any node all of whose parents already have values. When we pick a value for a variable we look only at the values of the parents of that variable. We will write $P(x_i \mid \text{parents of } x_i)$ to abbreviate $P(x \mid x_{i_1}, \dots, x_{i_k})$ where x_{i_1}, \dots, x_{i_k} are the parents of x . For example, if x_7 has parents x_2 and x_4 then $P(x_7 \mid \text{parents of } x_7)$ abbreviates $P(x_7 \mid x_2, x_4)$. Formally, the probability distribution on the variables of a Bayesian network is determined by the following equation.

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i \mid \text{parents of } x_i) \quad (11.1)$$

The conditional probabilities of the form $P(x_i \mid \text{parents of } x_i)$ are called conditional probability tables (CPTs). Suppose that each of the variables x_i has d possible values (this is not required in general). In this case if a variable x has k parents then $P(x \mid \text{parents of } x)$ has d^{k+1} values (with $(d-1)d^k$ degrees of freedom). These d^{k+1} values can be stored in a table with $k+1$ indices. Hence the term “table”. Note that the number of indices of the CPTs (conditional probability tables) is different for the different variables.

Note that an HMM is a Bayesian network with a variable for each hidden state and each observable token.

Bayesian networks are often used in medical diagnosis where variables represent the presence or absence of certain diseases or the presence or absence of certain measurable quantities such as blood sugar or presence of a certain protein in the blood.

In a Bayesian network the edges of the directed graph are often interpreted as “causation” with the parent causally influencing the child and parents getting assigned values temporally before children.

We are interested in “Bayesian inference” which means, intuitively, inferring causes by observing their effects using Bayes’ rule. In an HMM for example, we want to infer the hidden states from the observations that they cause.

In general we can formulate the inference problem as the problem of determining the probability of an unknown variable (a hidden cause) from observed values of other variables. In general we can consider the variables in any order.

$$P(x_5, x_7 \mid x_3, x_2) = \frac{P(x_5, x_7, x_2, x_3)}{P(x_2, x_3)} \quad (11.2)$$

So in general, for inference it suffices to be able to compute probabilities of the form $P(x_{i_1}, \dots, x_{i_k})$. We give an algorithm for doing this in section 11.5.

11.2 Inference in Bayesian Network is #P hard

A Boolean variable (also called Bernoulli variable) is a variable that has only the two possible values of 0 or 1. A disjunctive clause is a disjunction of literals where each literal is either a Boolean variable or the negation of a Boolean variable. For example we have that $(X_5 \vee \neg X_2 \vee X_3)$ is a clause with three literals. A 3SAT problem is a set of clauses with three literals in each clause. It is hard (in fact #P hard) to take a 3SAT problem and determine the number of complete assignments of values to variables that satisfy the clauses, i.e., that make every clause true.

Take X_1, \dots, X_n be independent Boolean (Bernoulli) variables with $P(X_i = 1) = 1/2$. Let Σ be a set of clauses over these variables where each clause has only three literals. Let C_j be a random variable which is 1 if the j th clause is satisfied. Since we can compute c_j from x_1, \dots, x_n using a (deterministic) conditional probability table having only three parents. Let A_j be a Boolean variable that is true if all of C_1, \dots, C_j are true. a_1 can be computed with a CPT from c_1 and for $j > 1$ we have that a_j can be computed using a CPT from a_{j-1} and c_j .

Now we have that $P(A_k = 1)$ is proportional to the number of truth assignments that satisfying all the clauses. This implies that computing the probability of a partial assignment in a Bayesian network is #P hard. It is widely believed that there are no polynomial time algorithms for #P hard problems.

11.3 Value Assignments

First we introduce some formal notation which will allow certain equation (11.1) to be written more precisely. We let V be a finite set of random variables (the nodes of the network) where for $X \in V$ we let $\mathcal{D}(X)$ be the set of values that variable X can have (the support of the variable X). We let σ range over assignments of values to the variables — for $X \in V$ we let $\sigma(X)$ denote the value that σ assigns to the variable X . We require $\sigma(X) \in \mathcal{D}(X)$. For an assignment σ , and a subset E of V , we write $\sigma|_E$ to be σ restricted to the variables in E so that $\sigma|_E$ is an assignment only to the variables in E .

Now consider a Bayesian network as defined in section 11.1. For $X \in V$ let $\mathcal{P}(X)$ be the set of parents of the variable X — $\mathcal{P}(X)$ is the set of Y such that there is a directed arc in the Bayesian network from Y to X . Equation (11.1) can now be written as follows where Ψ_X is a the function on assignments to the set $\{X\} \cup \mathcal{P}(X)$ giving the conditional probability in (11.3).

$$P(\sigma) = \prod_{X \in V} P(X = \sigma(X) \mid \sigma|_{\mathcal{P}(X)}) \quad (11.3)$$

$$= \prod_{X \in V} \Psi_X(\sigma|_{\{X\} \cup \mathcal{P}(X)}) \quad (11.4)$$

We now introduce notation which allows a more general and preciser version of equation (11.2). We let ρ range over partial assignments of values to variables — ρ assigns values to *some* of the variables in V . We can implement ρ as a finite list $\langle\langle X_1, x_1 \rangle\rangle, \dots, \langle\langle X_K, x_K \rangle\rangle$ where $X_k \in V$ and $x_k \in \mathcal{D}(X_k)$. We say that a partial assignment ρ' is an extension of ρ , written $\rho' \sqsubseteq \rho$, if ρ' assigns a value to every variable that ρ assigns a value to and every variable where ρ assigns a value, ρ' assigns the same value. We can think of a partial assignment ρ as representing a statement about, or constraint on, total assignments. We then have that the probability of partial assignment satiafies the following where σ ranges over total assignments.

$$P(\rho) = \sum_{\sigma \sqsubseteq \rho} P(\sigma) \quad (11.5)$$

The notation of partial assignments also allows us to write a general and more precise verion of (11.2) as follows where ρ' is any extension of a partial assignment ρ .

$$P(\rho'|\rho) = \frac{P(\rho')}{P(\rho)} \quad \text{for } \rho' \sqsubseteq \rho \quad (11.6)$$

To compute arbitrary conditional probabilities between partial assignments it suffices to be able to compute probabilities of the form $P(\rho)$.

11.4 Factor Graphs

A factor graph is a hypergraph with a factor term associated with each hyperedge. More specifically, a factor graph consists of a set V of random variables and a tuple $\langle E_1, \dots, E_k \rangle$ of “hyperedges” where each hyperedge E_k is a subset of V and for each hyperedge E_k there is a factor term Ψ_k described below. The set V and the hyperedges $\langle E_1, \dots, E_k \rangle$ together form a hypergraph.¹ A factor graph determines a probability distribution on total assignments σ as follows.

$$P(\sigma) = \frac{1}{Z} \prod_{k=1}^K \Psi_k(\sigma|_{E_k}) \quad (11.7)$$

$$Z = \sum_{\sigma} \prod_{k=1}^K \Psi_k(\sigma|_{E_k}) \quad (11.8)$$

Note that the factor term Ψ_k is a function on assignments of values to the variables on the hyperedge E_k . We now have that (11.4) is a special case of (11.7) — for the case of Bayesian networks we have that $Z = 1$.

For a partial assignment ρ we define the partition function $Z(\rho)$ as follows.

$$Z(\rho) = \sum_{\sigma \sqsubseteq \rho} \prod_{k=1}^K \Psi_k(\sigma|_{E_k}) \quad (11.9)$$

The partition function values $Z(\rho)$ should be thought of as an unnormalized probabilities. In particular we have the following where Z is defined by (11.8).

$$\begin{aligned} P(\rho) &= \sum_{\sigma \sqsubseteq \rho} P(\sigma) \\ &= \sum_{\sigma \sqsubseteq \rho} \frac{1}{Z} \prod_{k=1}^K \Psi_k(\sigma|_{E_k}) \\ &= \frac{1}{Z} \sum_{\sigma \sqsubseteq \rho} \prod_{k=1}^K \Psi_k(\sigma|_{E_k}) \\ &= \frac{Z(\rho)}{Z} \end{aligned} \quad (11.10)$$

¹Factor graphs are often formulated as bipartite graphs with one partition representing the random variables and the other partition representing the hyperedges (the “factors”). The formulation as a bipartite graph is equivalent to the formulation as a hypergraph.

For $\rho' \sqsubseteq \rho$ we have the following.

$$\begin{aligned}
 P(\rho'|\rho) &= \frac{P(\rho')}{P(\rho)} \\
 &= \frac{\left(\frac{Z(\rho')}{Z}\right)}{\left(\frac{Z(\rho)}{Z}\right)} \\
 &= \frac{Z(\rho')}{Z(\rho)} \tag{11.11}
 \end{aligned}$$

Equation (11.11) shows that to compute conditional probabilities between partial assignments it suffices to compute the unnormalized values $Z(\rho)$.

11.5 The Case-Factor Algorithm

Consider a fixed case-factor graph with nodes (random variables) V and with K factors so that for $1 \leq k \leq K$ we have the hyperedge $E_k \subseteq V$ and we have the function Ψ_k assigning real values scores to assignments of values to the variables in the hyperedge E_k . We want consider subgraphs of this factor graph. A subgraph will be taken to consist of a subset of nodes $\mathcal{V} \subseteq V$ and a subset of the factors $\mathcal{F} \subset \{1, \dots, K\}$ such that for all $k \in \mathcal{F}$ we have that the hyperedge E_k is a subset of \mathcal{V} . The case-factor algorithm is based on computing Z values for subgraphs. These values have the form $Z(\rho, \mathcal{V}, \mathcal{F})$ defined as follows where ρ is a partial assignment to the variables in \mathcal{V} and σ ranges over total assignments to the variables in \mathcal{V} .

$$Z(\rho, \mathcal{V}, \mathcal{F}) = \sum_{\sigma \sqsubseteq \rho} \prod_{k \in \mathcal{F}} \Psi_k(\sigma|_{E_k}) \tag{11.12}$$

To define the case-factor algorithm we some additional notation. For a partial assignment ρ we write $\text{dom}(\rho)$ for the set of variables that are assigned values by ρ . For $X \notin \text{dom}(\rho)$ and $x \in \mathcal{D}(X)$ we write $\rho[X; = x]$ for the partial assignment that is ρ extended with the one additional assignment that assigns X the value x . We write $S \setminus U$ for set difference — $S \setminus U$ is the subset of elements of S that are not elements of U . Equation (11.12) is the definition of $Z(\rho, \mathcal{V}, \mathcal{F})$ and from this definition we can prove the following two equations which define the case-factor

algorithm.

$$Z(\rho, \mathcal{V}, \mathcal{F}) = \sum_{x \in \mathcal{D}(X)} Z(\rho[X := x], \mathcal{V}, \mathcal{F}) \text{ if } \begin{cases} X \in \mathcal{V} \\ X \notin \text{dom}(\rho) \end{cases} \quad (11.13)$$

$$Z(\rho, \mathcal{V}, \mathcal{F}) = Z(\rho|_{\mathcal{V}_1}, \mathcal{V}_1, \mathcal{F}_1) Z(\rho|_{\mathcal{V}_2}, \mathcal{V}_2, \mathcal{F}_2) \quad (11.14)$$

$$\text{if } \begin{cases} E_k \subseteq \mathcal{V}_1 \text{ for } k \in \mathcal{F}_1 \\ E_k \subseteq \mathcal{V}_2 \text{ for } k \in \mathcal{F}_2 \\ (\mathcal{V}_1 \setminus \text{dom}(\rho)) \cap (\mathcal{V}_2 \setminus \text{dom}(\rho)) = \emptyset \\ (\mathcal{V}_1 \setminus \text{dom}(\rho)) \cup (\mathcal{V}_2 \setminus \text{dom}(\rho)) = \mathcal{V} \setminus \text{dom}(\rho) \\ \mathcal{F}_1 \cap \mathcal{F}_2 = \emptyset \\ \mathcal{F}_1 \cup \mathcal{F}_2 = \mathcal{F} \end{cases}$$

$$Z(\rho, \mathcal{V}, \mathcal{F}) = \prod_{k \in \mathcal{F}} \Psi_k(\rho) \text{ if } \text{dom}(\rho) = \mathcal{V} \quad (11.15)$$

Each of these three equations gives a potential way to make progress in computing $Z(\rho, \mathcal{V}, \mathcal{F})$. Equation (11.13) is the case rule which does a case analysis on the value of one of the variables not in $\text{dom}(\rho)$. Equation (11.14) gives a way to factor the problem into a product of two subproblems. Equation (11.14) can be used whenever the variables in $\mathcal{V} \setminus \text{dom}(\rho)$ can be divided into two disjoint sets such that no factor involves variables from both of these sets. To computing the factoring we simply need to compute the connected components of the graph on $\mathcal{V} \setminus \text{dom}(\rho)$ where two nodes are connected if they occur in the same factor in \mathcal{F} . To apply a case-factor analysis we first see if the factor rule can be used. If not, we apply the case rule unless all variables in \mathcal{V} are already in $\text{dom}(\rho)$ in which case we apply the base case equation (11.15). In the case-factor algorithm it is important to memoize the values of subcomputations.

11.6 The Junction Tree of the Case-Factor Algorithm

Consider using the junction tree algorithm to compute $Z(\rho, \mathcal{V}, \mathcal{F})$. The choice of which rule to apply — case, factor, or the base case — is determined by the three sets $\text{dom}(\rho)$, \mathcal{V} , and \mathcal{F} independent of the particular partial assignment ρ . Therefore one can construct an abstract problem-subproblem graph each node of which is labeled by the three sets $\text{dom}(\rho)$, \mathcal{V} and \mathcal{F} and where there is a directed arc from one node to another if either the case or factor rule generates the second node a subproblem the first. Because the factor rule generates subproblems with disjoint sets of unassigned variables, this graph forms a tree. The junction tree

generated by the case-factor algorithm is the tree that results from erasing the sets \mathcal{V} and \mathcal{F} at each node of the abstract problem-supproblem tree so that the junction tree is a tree each node of which is labeled with the set $\text{dom}(\rho)$.

11.7 General Junction Trees and Tree Width

The junction tree of the case-factor algorithm is an instance of the more general concept of junction tree. The concept of a junction tree can be defined for any hypergraph. Consider a hypergraph with node set V and hyperedges E_1, \dots, E_k . A junction tree for this hypergraph is a tree T each node n of which is associated with a set a variables denoted $\text{dom}(n)$ with $\text{dom}(n) \subseteq V$ and such that the following conditions hold.

- **Cover Property:** For each E_k there exists a node n with $E_k \subseteq \text{dom}(n)$.
- **Running Intersection Property:** For any node $X \in V$, the set of nodes n with $X \in \text{dom}(n)$ is a subtree of T .

The width of a junction tree is the maximum size of the set $\text{dom}(n)$ over all nodes n of the tree minus 1. The tree width of a hypergraph is the minimum width of any junction tree for that hypergraph. Note that a graph is a special case of a hypergraph and it is common to talk about the tree width of a graph. We subtract one from $\text{dom}(n)$ so that the width of a tree equals 1.

It is NP hard to determine tree width even for graphs. However, good heuristics exists for constructing junction trees of small width. It can be shown that there exists a variable ordering, used to select the variable used in the case rule, such that the case-factor junction tree generated by that variable-selection ordering has minimum width. So one can also define tree width to be the minimum over all variable orderings of the width of the case-factor junction tree for that ordering. Again, good heuristics exist for selecting the next variable to case on.

11.8 The Junction Tree Algorithm

Let n and m be any two nodes connected by an edge in the junction tree. It turns out that we do not need to root the tree — we do not need to distinguish which node is the parent and which node is the child. We write $n \rightarrow m$ if there is an edge in the tree from n to m . We will treat the direction $n \rightarrow m$ differently from $m \rightarrow n$ — for each direction there is a message and the two messages are different. We write $s \rightarrow^* n \rightarrow m$ if the path in the tree from s to m includes n . We allow s to be n so that we have $n \rightarrow^* n \rightarrow m$. To define the messages we also need to assign each factor to a single node of the junction tree — in general there might be several different nodes n of the junction tree with $E_k \subseteq \text{dom}(n)$.

So for each factor we pick one node to cover that factor and then let $\mathcal{F}(n)$ be the set of factors whose selected node is n . We now define the message $Z_{n \rightarrow m}$ to be a function of assignments to $\text{dom}(n) \cap \text{dom}(m)$ as follows.

$$Z_{n \rightarrow m}(\rho) = Z(\rho, \mathcal{V}_{n \rightarrow m}, \mathcal{F}_{n \rightarrow m}) \quad (11.16)$$

$$\mathcal{V}_{n \rightarrow m} = \{X \in V : \exists s \ s \rightarrow^* n \rightarrow m, \ X \in \text{dom}(s)\} \quad (11.17)$$

$$\mathcal{F}_{n \rightarrow m} = \{k : \exists s \ s \rightarrow^* n \rightarrow m, \ k \in \mathcal{F}(s)\} \quad (11.18)$$

Given this definition of $Z_{n \rightarrow m}(\rho)$ we have the following equation which can be used to compute these messages recursively and where ρ' ranges over assignments to the variables in $\text{dom}(n)$.

$$Z_{n \rightarrow m}(\rho) = \sum_{\rho' \sqsubseteq \rho} \left(\prod_{k \in \mathcal{F}(n)} \Psi_k(\rho') \right) \left(\prod_{s \rightarrow n, \ s \neq m} Z_{s \rightarrow n}(\rho' |_{\text{dom}(s)}) \right) \quad (11.19)$$

Equation (11.19) defines the junction tree algorithm. It can be viewed as a recursive definition of the messages, although the semantics of the messages is defined by (11.16) and (11.12). The recursion in (11.19) is well founded — the set of nodes s with $s \rightarrow^* n \rightarrow m$ is reduced on each recursive call. Therefore the recursion terminates and equation (11.19) can be used to compute the messages. It is important to memoize the messages so that each message is only computed once.

11.9 Problems

1. Consider a Bayesian network with three variables X_1, X_2, X_3 where X_1 and X_2 have no parents and X_3 has parents X_1 and X_2 so that we have the following equation.

$$P(X_1 = x_1, X_2 = x_2, X_3 = x_3) = \Psi_1(x_1)\Psi_2(x_2)\Psi_3(x_3, x_1, x_2) \quad (11.20)$$

a) Prove that X_1 and X_2 are independent, i.e., prove the following.

$$P(X_1 = x_1, X_2 = x_2) = P(X_1 = x_1)P(X_2 = x_2) \quad (11.21)$$

Now consider a Markov random field on $X_1, X_2,$ and X_3 with hyperedges $\{X_1\}, \{X_2\},$ and $\{X_1, X_2, X_3\}$ so that we have the following.

$$P(X_1 = x_1, X_2 = x_2, X_3 = x_3) = \frac{1}{Z} \Psi_1(x_1)\Psi_2(x_2)\Psi_3(x_1, x_2, x_3) \quad (11.22)$$

Note the similarity between (11.20) and (11.22). Suppose that each of the variables X_1, X_2 and X_3 only take on values in the two element set $\{-1, 1\}$.

b) Given an example of the energy functions E_1 , E_2 and E_3 for which equation (11.21) does *not* hold.

2. Consider an image defined by an $n \times m$ array I of integers in the range 0 to 255. A segmentation of an image is a division of the image into segments. We assume that each segment has an identifier which is also an integer in the range from 0 to 256. Let $I(x, y)$ be the image value at coordinates x, y and let $S(x, y)$ be the segment index for the segment containing pixel x, y . Suppose that we want to find a segmentation S minimizing the following energy where $[\Phi]$ is the indicator function for Φ , i.e., $[\Phi] = 1$ if Φ is true and $[\Phi] = 0$ if Φ is false.

$$E(I, S) = \left(\sum_{x,y} (I(x, y) - S(x, y))^2 \right) + \left(\sum_{x,y, \delta \in \{-1,1\}, \gamma \in \{-1,1\}} \lambda [S(x, y) \neq S(x + \delta, y + \gamma)] \right)$$

λ is a parameter of the energy function called a “regularization parameter”. Explain the effect of increasing or decreasing λ . Also formalize the problem of minimizing this energy as a Markov random field problem. What are the variables, the hyperedges, and the energy functions associated with each hyperedge.

3. Consider the Bayesian network $X \rightarrow Z \leftarrow Y \rightarrow W$. Show that X and W are independent. Give an instance of this network (particular conditional probability tables) where X and W are not independent given Z .

4. Consider a graph with nodes A_1, A_2, \dots, A_n and B_1, B_2, \dots, B_n and edges from A_i to B_i , from A_i to A_{i+1} and from B_i to B_{i+1} . These edges form a “ladder”. Consider a Markov random field where \mathcal{F} has a function for each edge of this graph. Give a junction tree for this MRF with width 2.

5. A “series parallel” graph is a graph with special structure (defined below) and with two distinguished nodes called the “interface nodes” of the graph. Series parallel graphs can be defined recursively as follows.

- **edge:** A two node graph with one edge between the two nodes is a series parallel graph where the two given nodes are the interface nodes.
- **series combination:** If G_1 is a series parallel graph with interface nodes A and B and G_2 is a series parallel graph with interface nodes B and C , and B is the only node in both G_1 and G_2 , then $G_1 \cup G_2$ is a series parallel graph with interface nodes A and C .
- **parallel combination:** If G_1 is a series parallel graph with interface nodes A and B and G_2 is a series parallel graph which also has interface nodes A and B , and A and B are the only nodes in both G_1 and G_2 , then $G_1 \cup G_2$ is a series parallel graph with interface nodes A and B .

- a) Draw a series-parallel graph that is a parallel combination of two series combinations of edges.
- b) Consider a Markov random field (MRF) where each edge of the graph in part a) is considered to be a hyperedge of the field. Draw a junction tree of width 2 for this MRF.
- c) Prove, by structural induction on the definition of a series-parallel graph, that for every series-parallel graph there exists a width 2 junction tree.

Chapter 12

Loopy Belief Propagation

12.1 Graph-Structured Factor Graphs

A graph-structured factor graph is a factor graph in which the factors involve at most two nodes. Graph structured factor graphs can be represented by a set of random variables (nodes) V and a set of edges E on V where, for $X, Y \in V$, we will write $\{X, Y\} \in E$ if the edge $\{X, Y\}$ is an edge in E . For convenience here we assume that for each $X \in V$ the support of X (the set of values that the variable X can have) is finite and we denote the support of X by $\mathcal{D}(X)$. For each $X \in V$ we assume a factor Ψ_X such that for each $x \in \mathcal{D}(X)$ we have that $\Psi_X(x)$ is a nonnegative real number. We also assume a factor $\Psi_{X,Y}$ for each edge $\{X, Y\} \in E$ such that for $x \in \mathcal{D}(X)$ and $y \in \mathcal{D}(Y)$ we have that $\Psi_{X,Y}(x, y)$ is a nonnegative real number. The edges in E are undirected and we have $\Psi_{X,Y}(x, y) = \Psi_{Y,X}(y, x)$.

We let σ range over complete assignments of values to variables in V — for any $X \in V$ we have $\sigma(X) \in \mathcal{D}(X)$. We then define the following unnormalized probability.

$$Z(\sigma) = \left(\prod_{X \in V} \Psi_X(\sigma(X)) \right) \left(\prod_{\{X,Y\} \in E} \Psi_{X,Y}(\sigma(X), \sigma(Y)) \right) \quad (12.1)$$

$$P(\sigma) = \frac{Z(\sigma)}{Z} \quad (12.2)$$

$$Z = \sum_{\sigma} Z(\sigma) \quad (12.3)$$

The quantity Z is called the partition function. If the factors Ψ_X and $\Psi_{X,Y}$ involve parameters Θ the one writes $Z(\Theta)$ and the partition function is a func-

tion of the parameters defining the factors. Here we will not be concerned with parameters.

We let ρ range over partial assignments of values to variables — ρ is a finite set of pairs each of which specifies a value for one of the variables in V . We write $\sigma \sqsubseteq \rho$ if σ satisfies all of the assignments in ρ . We define $Z(\rho)$ as follows.

$$Z(\rho) = \sum_{\sigma \sqsubseteq \rho} Z(\sigma) \quad (12.4)$$

$$P(\rho) = \sum_{\sigma \sqsubseteq \rho} P(\sigma) \quad (12.5)$$

$$= \frac{Z(\rho)}{Z} \quad (12.6)$$

If ρ' is a partial assignments that extends ρ , i.e., ρ' contains all the pairs in ρ plus possibly additional pairs, written $\rho' \sqsupseteq \rho$, then we have the following.

$$P(\rho' | \rho) = \frac{P(\rho')}{P(\rho)} \quad (12.7)$$

$$= \frac{Z(\rho')}{Z(\rho)} \quad (12.8)$$

To compute conditional probabilities of the form $P(\rho' | \rho)$ it therefore suffices to be able to compute values of the form $Z(\rho)$.

12.2 Belief Propagation for Trees

We now consider the case where the edges in E form a tree. In this case the junction tree algorithm can be formulated directly on the graph E with a message $Z_{X \rightarrow Y}$ for $\{X, Y\} \in E$. The message $Z_{X \rightarrow Y}$ is a function on $\mathcal{D}(Y)$. To define the semantics of the message we first define $\mathcal{V}_{X \rightarrow Y}$ to be the set of $U \in V$ such that the path in the tree E from U to Y includes X . Note that $X \in \mathcal{V}_{X \rightarrow Y}$. Intuitively, the subtree $\mathcal{V}_{X \rightarrow Y}$ is the “input” to the message $Z_{X \rightarrow Y}$. The semantics of the message is defined as follows where σ ranges over assignments of values to the set $\mathcal{V}_{X \rightarrow Y}$.

$$Z_{X \rightarrow Y}(y) = \sum_{\sigma} \begin{cases} \Psi_{X,Y}(\sigma(X), y) \\ \prod_{U \in \mathcal{V}_{X \rightarrow Y}} \Psi_U(\sigma(U)) \\ \prod_{\{U,W\} \in E, U,W \in \mathcal{V}_{X \rightarrow Y}} \Psi_{U,W}(\sigma(U), \sigma(W)) \end{cases} \quad (12.9)$$

In words, the message $Z_{X \rightarrow Y}(y)$ is the partition function for the factor graph consisting of the input subtree $\mathcal{V}_{X \rightarrow Y}$ plus the edge from X to Y but with

the value of Y fixed at y . The messages can be efficiently computed using the following equations which can be proved as a lemma from the definition of the message.

$$Z_{X \rightarrow Y}(y) = \sum_{x \in \mathcal{D}(X)} \Psi_X(x) \left(\prod_{\{U, X\} \in E; U \neq Y} Z_{U \rightarrow X}(x) \right) \Psi_{X, Y}(x, y) \quad (12.10)$$

Equation (12.10) defines the belief propagation algorithm. As noted above, this is equivalent to the junction tree algorithm but has a simpler form due to the simple form of a tree structured factor graph. In the junction tree algorithm there would be an additional node representing each edge in E (so that the cover property holds), and we get a message into the node for the edge $X \rightarrow Y$ which is defined on $\mathcal{D}(X)$ as well as a message out of the node for the edge $X \rightarrow Y$ which is defined on $\mathcal{D}(Y)$. In (12.10) we have only the outgoing messages. Note that the recursion in (12.10) terminates because the recursive call always involves computing the partition function of a smaller subtree.

We can compute $Z(\rho)$ for any ρ by restricting the domain of each variables assigned in ρ to consist of the single value assigned in ρ and then applying the belief propagation algorithm under this restriction of domains. For any variable X we also have the following.

$$Z(X = x) = \Psi_X(x) \left(\prod_{\{Y, X\} \in E} Z_{Y \rightarrow X}(x) \right) \quad (12.11)$$

12.3 Loopy Belief Propagation

We now consider a graph structured factor graph but where the graph E is not a tree (it is “loopy”). Loopy belief propagation (loopy BP) uses a form of equation (12.10) even though the equations no longer have a clear semantics. The intuition is that if the loops are long then the effect of the loops fades out as the messages propagate and the resulting answer is accurate in any case. Loopy BP has proved very effective in many applications. In loopy BP we assume a message $m_{X \rightarrow Y}^t$ for every edge $\{X, Y\} \in E$ and compute a new system of messages as follows.

$$m_{X \rightarrow Y}^{t+1}(y) = \frac{1}{Z} \sum_{x \in \mathcal{D}(X)} \Psi_X(x) \left(\prod_{\{U, X\} \in E; U \neq Y} m_{U \rightarrow X}^t(x) \right) \Psi_{X, Y}(x, y) \quad (12.12)$$

In (12.12) the constant Z is selected so that the messages are normalized, i.e., $\sum_{y \in \mathcal{D}(Y)} m_{X \rightarrow Y}(y) = 1$. It is useful to compare (12.12) with (12.10). We have

replaced $Z_{X \rightarrow Y}$ by $m_{X \rightarrow Y}$ because in the loopy case we no longer have a semantics for the messages as partition function values. The messages $m_{X \rightarrow Y}^0$ are typically initialized to be uniform. Normalization is needed in the loopy case to prevent the message values from diverging exponentially as t increases. Equation (12.12) can be computed more efficiently as follows.

$$B_X^t(x) = \Psi_X(x) \prod_{\{X,Y\} \in E} m_{Y \rightarrow X}^t(x) \quad (12.13)$$

$$m_{X \rightarrow Y}^{t+1}(y) = \frac{1}{Z_{X \rightarrow Y}^{t+1}} \sum_{x \in \mathcal{D}(X)} \frac{B_X^t(x)}{m_{Y \rightarrow X}^t(x)} \Psi_{X,Y}(x, y) \quad (12.14)$$