Machine Learning Algorithms IN DEPTH

Vadim Smolyakov

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Machine Learning Algorithms in Depth

Version 7

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Machine Learning Algorithms from Scratch*. This book will take you on a journey from mathematical derivation to software implementation of some of the most intriguing algorithms in ML.

This book dives into the design of ML algorithms from scratch. Throughout the book, you will develop mathematical intuition for classic and modern ML algorithms, learn the fundamentals of Bayesian inference and deep learning, as well as the data structures and algorithmic paradigms in ML.

Understanding ML algorithms from scratch will help you choose the right algorithm for the task, explain the results, troubleshoot advanced problems, extend an algorithm to a new application, and improve performance of existing algorithms.

Some of the prerequisites for reading this book include basic level of programming in Python and intermediate level of understanding of linear algebra, applied probability and multivariate calculus.

My goal in writing this book is to distill the science of ML and present it in a way that will convey intuition and inspire the reader to self-learn, innovate and advance the field. Your input is important. I'd like to encourage you to post questions and comments in the liveBook discussion forum to help improve presentation of the material.

Thank you again for your interest and welcome to the world of ML algorithms!

— Vadim Smolyakov

brief contents

1

Machine Learning Algorithms

This chapter covers

- **Types of ML algorithms**
- **Importance of learning algorithms from scratch**
- **Introduction to Bayesian Inference and Deep Learning**
- **Software implementation of machine learning algorithms from scratch**

An algorithm is a sequence of steps required to achieve a particular task. An algorithm takes an input, performs a sequence of operations and produces a desired output. The simplest example of an algorithm is sorting: given a list of integers, we perform a sequence of operations to produce a sorted list. A sorted list enables us to organize information better and find answers in our data.

Two popular questions to ask about an algorithm is how fast does it run (run-time complexity) and how much memory does it take (memory complexity) for an input of size `n`. For example, a comparison-based sort, as we'll see later, has `O(nlogn)` run-time complexity and requires `O(n)` memory storage.

There are many approaches to sorting, and in each case, in the classic algorithmic paradigm, the algorithm designer creates a set of instructions. Imagine a world where you can *learn* the instructions based on a sequence of input and output examples available to you. This is a setting of ML algorithmic paradigm. Similar to how a human brain learns, when we are playing connect-the-dots game or sketching a nature landscape, we are comparing the desired output with what we have at each step and filling in the gaps. This in broad strokes is what (supervised) machine learning (ML) algorithms do. During training, ML algorithms are learning the rules (e.g. classification boundaries) based on training examples by optimizing an objective function. During testing, ML algorithms apply previously learned rules to new input data points to give a prediction as shown in Figure 1.1

1.1 Types of ML Algorithms

Let's unpack the previous paragraph a little bit and introduce some notation. This book focuses on machine learning algorithms that can be grouped together in the following categories: supervised learning, unsupervised learning and deep learning. In **supervised learning**, the task is to learn a mapping `f` from inputs `x` to outputs given a training dataset `D = {(x1, y1 ),…, (xn, yn )}` of `n` input-output pairs. In other words, we are given `n` examples of what the output should look like given the input. The output `y` is also often referred to as the label, and it is the supervisory signal that tells our algorithm what the correct answer is.

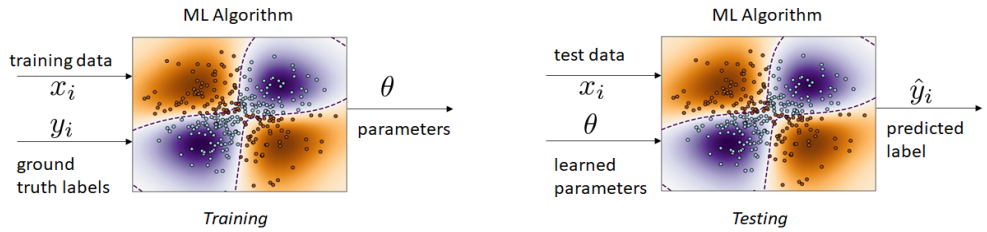


Figure 1.1 Supervised Learning: Training (left) and Testing (right)

Supervised learning can be sub-divided into *classification* and *regression* based on the quantity we are trying to predict. If our output `y` is a discrete quantity (e.g. K distinct classes) we have a classification problem. On the other hand, if our output `y` is a continuous quantity (e.g. a real number such as stock price) we have a regression problem.

Thus, the nature of the problem changes based on the quantity `y` we are trying to predict. We want to get as close as possible to the ground truth value of `y`.

A common way to measure performance or closeness to ground truth is the loss function. The loss function is computing a distance between the prediction and the true label. Let `y = f(x; θ)` be our ML algorithm that maps input examples `x` to output labels `y`, parameterized by `θ`, where `θ` captures all the learnable parameters of our ML algorithm. Then, we can write our classification loss function as follows in Equation 1.1:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} 1[y_i \neq f(x_i;\theta)]$$

Equation 1.1 Loss function for classification

Where `1[]` is an indicator function, which is equal to 1 when the argument inside is true and 0 otherwise. What the expression above says is we are adding up all the instances in which our prediction `f(xi; θ)` did not match the ground truth label `yi` and we are dividing by the total number of examples `n`. In other words, we are computing an average misclassification rate. Our goal is to minimize the loss function, i.e. find a set of parameters `θ`, that make

the misclassification rate as close to zero as possible. Note that there are other alternative loss functions for classification such as cross entropy that we will look into in later chapters.

For continuous labels or response variables, a common loss function is the Mean Square Error (MSE), defined as follows in Equation 1.2:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}[y_i - f(x_i;\theta)]^2$$

Equation 1.2 Loss function for regression

Essentially, we are subtracting our prediction from the ground truth label, squaring it and aggregating the result as an average over all data points. By taking the square we are eliminating the possibility of negative loss values, which would impact our summation.

One of the central goals of machine learning is to be able to generalize to unseen examples. We want to achieve high accuracy (low loss) on not just the training data (which is already labelled) but on new, unseen, test data examples. This generalization ability is what makes machine learning so attractive: if we can design ML algorithms that can see outside their training box, we'll be one step closer to Artificial General Intelligence (AGI).

In **unsupervised learning,** we are not given the label y nor are we learning the mapping between input and output examples, instead we are interested in making sense of the data itself. Usually, that implies in a lower dimension and with some properties so it could be easier to understand by human. In other words, our training dataset consists of $D = \{x_1,\ldots,x_n\}$ of n input examples without any corresponding labels y. The simplest example of unsupervised learning is finding clusters within data. Intuitively, data points that belong to the same cluster have similar characteristics. In fact, data points within a cluster can be represented by the cluster center as an exemplar and used as part of a data compression algorithm. Alternatively, we can look at the distances between clusters in a projected lower-dimensional space to understand the inter-relation between different groups. Additionally, a point that's far away from all the existing clusters can be considered an anomaly leading to an anomaly detection algorithm. As you can see, there's an infinite number of interesting uses cases that arise from unsupervised learning, and we'll be learning from scratch some of the most intriguing algorithms in that space.

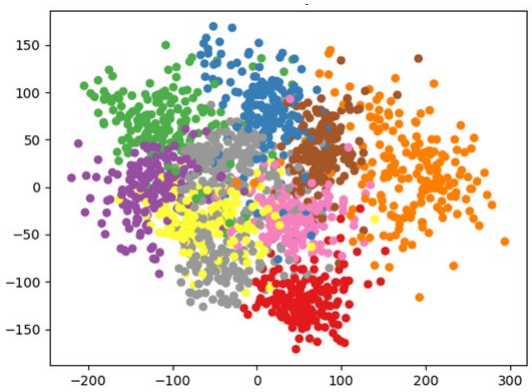


Figure 1.2 Unsupervised Learning: clusters of data points projected onto 2-dimensional space

Another very important area of modern machine algorithms is **deep learning**. The name comes from a stack of computational layers forming together a computational graph. The depth of this graph refers to sequential computation and the breadth to parallel computation.

As we'll see, deep learning models gradually refine their parameters through back-propagation algorithm until they meet the objective function. Deep learning models permeated the industry due to their ability to solve complex problems with high accuracy. For example, Figure 1.3 shows a deep learning architecture for sentiment analysis. We'll learn more about what individual blocks represent in future chapters.

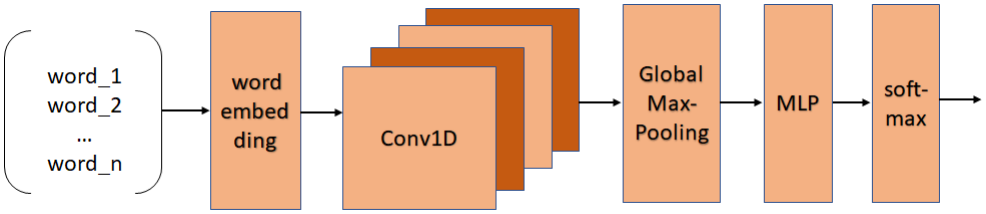


Figure 1.3 Deep Neural Network (DNN) architecture for sentiment analysis

Deep learning is a very active research area and we'll be focusing on the modern deep learning algorithms throughout this book. For example, in self-supervised learning, used in Transformer models, we are using the context and structure of the natural language as a supervisory signal thereby extracting the labels from the data itself. In addition to classic applications of deep learning in Natural Language Processing (NLP) and Computer Vision (CV), we'll be taking a look at generative models, learning how to predict time-series data and journey into relational graph data.

1.2 Why Learn Algorithms from Scratch?

Understanding ML algorithms from scratch has a number of valuable outcomes for the reader. First, you will be able to choose the right algorithm for the task. By knowing the innerworkings of the algorithm, you will understand its shortcomings, assumptions made in the derivation of the algorithm as well as advantages in different data scenarios. This will enable you to exercise judgement when selecting the right solution to a problem and save time by eliminating approaches that don't work.

Secondly, you will be able to explain the results of the given algorithm to the stakeholders. Being able to interpret the results and present them to the audience in the industrial or academic settings is an important trait of ML algorithm designer.

Thirdly, you will be able to use intuition developed by reading this book to troubleshoot advanced ML problems. Breaking down a complex problem into smaller pieces and understanding where things went wrong often requires a strong sense of fundamentals and algorithmic intuition. This book will allow the reader to construct minimum working examples and build upon existing algorithms to develop and be able to debug more complex models.

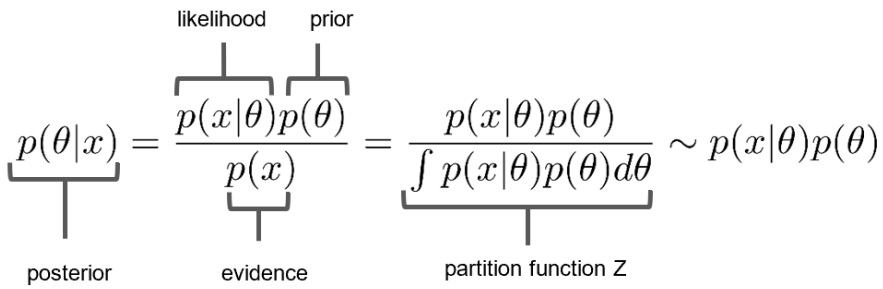
Fourthly, you will be able to extend an algorithm when a new situation arises in the real world, in particular, where the textbook algorithm or a library cannot be used as is. The in-depth understanding of ML algorithms that you will acquire in this book will help you modify existing algorithms to meet your needs.

Finally, we are often interested in improving performance of existing models. The principles discussed in this book will enable the reader to accomplish that. In conclusion, understanding ML algorithms from scratch will help you choose the right algorithm for the task, explain the results, troubleshoot advanced problems, extend an algorithm to a new situation and improve performance of existing algorithms.

1.3 Bayesian Inference and Deep Learning

Bayesian inference allows us to update our beliefs about the world given observed data. Our minds hold a variety of mental models explaining different aspects of the world, and by observing new data points, we can update our latent representation and improve our understanding of reality. Any **probabilistic model** is described by a set of parameters θ modelled as random variables, which control the behavior of the model, and associated data x.

The goal of Bayesian inference is to find the posterior distribution $p(\theta|x)$ in order to capture well a particular aspect of reality. The posterior distribution is proportional to the product of the likelihood $p(x|\theta)$ and the prior $p(\theta)$, which follows from the Bayes rule in Equation 1.3:



Equation 1.3 Bayes Rule

Prior `p(θ)` is our initial belief and can be either non-informative (e.g. uniform over all possible states) or informative (e.g. based on experience in a particular domain). Moreover, our inference results depend on the prior we choose: not only the value of prior parameters but also on the functional form of the prior. We can imagine a chain of updates in which the prior becomes a posterior as more data is obtained in a form of Bayes engine shown in Figure 1.4. We can see how our prior is updated to a posterior via Bayes rule as we observe more data.

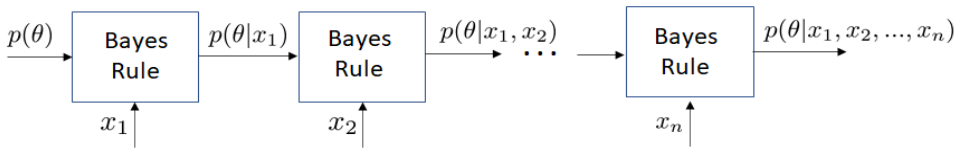


Figure 1.4 Bayes engine showing the transformation of a prior to a posterior as more data is observed

Posteriors that have the same form as the prior are known as **conjugate priors**, which are historically preferred since they simplify computation by having closed form updates. The denominator `Z=p(x)=∫p(x|θ)p(θ)dθ` is known as the normalizing constant or the partition function and is often intractable to compute due to integration in high dimensional parameter space. We'll look at a number of techniques in this book that work around the problem of estimating `Z`.

We can model relationships between different random variables in our model as a graph as shown in Figure 1.5 giving rise to **probabilistic graphical models**. Each node in the graph represents a random variable (RV) and each edge represents conditional dependency. The topology of the graph itself changes according to specific application you are trying to model. However, the goals of Bayesian inference remain the same.

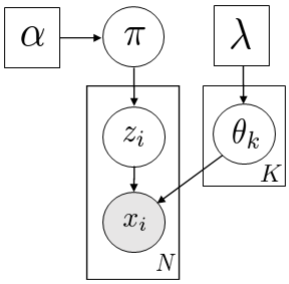


Figure 1.5 Probabilistic Graphical Model (PGM) for a Gaussian Mixture Model

In contrast to PGMs, where the connections are specified by domain experts, **deep learning** model learn representations of the world automatically through the algorithm of back-propagation that minimizes an objective function. Deep Neural Networks (DNNs) consist of multiple layers parameterized by weight matrices and bias parameters. Mathematically, DNNs can be expressed as a composition of individual layer functions as in Equation 1.4:

$$\mathrm{DNN}(x;\theta) = f_L(f_{L-1}\cdots(f_1(x;\theta_1))\cdots)$$

Equation 1.4 Deep Neural Network Composition

Where $f_l(x)=f(x;\theta_l)$ is the function at layer l. The compositional form of the DNNs reminds us of the chain rule when it comes to differentiating with respect to parameters as part of stochastic gradient descent. Throughout this book, we'll look at a number of different kinds of DNNs such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) as well as Transformers and Graph Neural Networks (GNNs) with applications ranging from computer vision to finance.

1.3.1 Two Main Camps of Bayesian Inference: MCMC and VI

Markov Chain Monte Carlo (MCMC) is a methodology of sampling from high dimensional parameter spaces in order to approximate the posterior distribution $p(\theta|x)$. There are many approaches to sampling in high dimensional parameter spaces. As we'll see in later chapters, MCMC is based on constructing a Markov chain whose stationary distribution is the target density of interest, i.e. posterior distribution. By performing a random walk over the state space, the fraction of time we spend in each state θ will be proportional to $p(\theta|x)$. As a result, we can use Monte Carlo integration to derive the quantities of interest associated with our posterior distribution.

Before we get into high dimensional parameter spaces, let's take a look at how we can sample from low dimensional spaces. The most popular method for sampling form univariate distributions is known as the inverse CDF method, where CDF stands for cumulative density function and it is defined as $\text{CDF}_X(x) = P(X \leq x)$

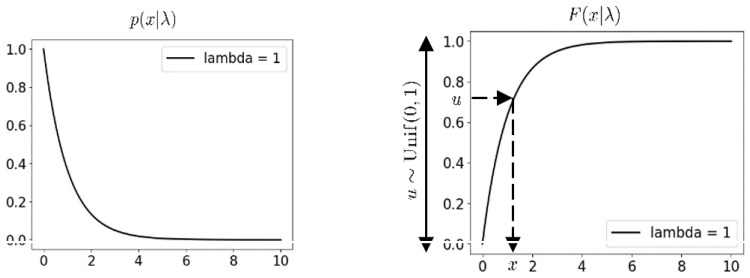


Figure 1.6 Exponential Random Variable PDF (left) and CDF (right)

For example, for an exponential random variable (RV) with probability density function (PDF) and CDF given by Equation 1.5:

$$p(x|\lambda) = \lambda e^{-\lambda x}, x \geq 0 \qquad F(x|\lambda) = \int_0^x p(x|\lambda)dx = 1 - e^{-\lambda x}, x \geq 0$$

Equation 1.5 Probability Density Function (PDF) and Cumulative Density Function (CDF) of Exponential RV

The inverse CDF can be found as follows:

$$F^{-1}(u) = -\ln(1-u)/\lambda$$

Equation 1.6 Inverse CDF for Exponential RV

Thus, to generate a sample from an exponential RV, we first need to generate a sample from uniform random variable `u ~ Unif(0,1)` and apply the transformation `-ln(1-u)/λ`. By generating enough samples, we can achieve an arbitrary level of accuracy. One challenge regarding MCMC is how to *efficiently* generate samples from high-dimensional distributions. We'll look at two ways of doing that in this book: Gibbs sampling and Metropolis-Hastings (MH) sampling.

Variational Inference (VI) is an optimization based approach to approximating the posterior distribution `p(x)`. The basic idea behind VI is to choose an approximate distribution `q(x)` from a family of tractable distributions and then make this approximation as close as possible to the true posterior distribution `p(x)`. As we will see in the mean-field

section of the book, the approximate `q(x)` can take on a fully factored representation of the joint posterior distribution. This factorization significantly speeds up computation.

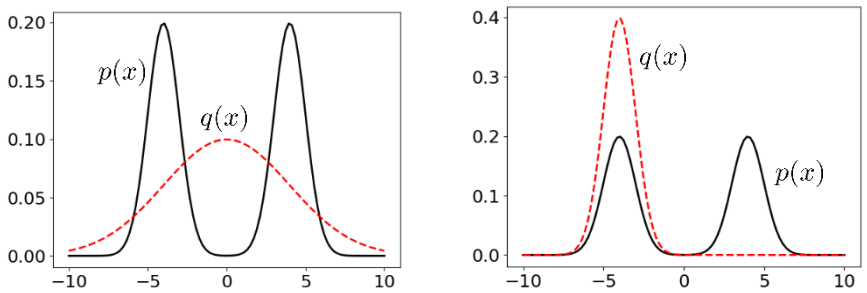


Figure 1.7 Forward KL (left) and Reverse KL (right) approximate distribution q(x) fit to Gaussian mixture p(x)

We will introduce KL divergence and use it as a way to measure closeness of our approximate distribution to the true posterior. Figure 1.7 shows the two versions of KL divergence.

$$KL(p||q) = \sum p(x) \log \frac{p(x)}{q(x)} \qquad KL(q||p) = \sum q(x) \log \frac{q(x)}{p(x)}$$

Equation 1.7 Forward KL (left) and Reverse KL (right) divergence definitions

The original distribution `p(x)` is a bi-modal Gaussian distribution (aka Gaussian mixture with two components), while the approximating distribution `q(x)` is a uni-modal Gaussian. As we can see from the Figure 1.7, we can approximate the distribution with two peaks either at the center with `q(x)` that has high variance so as to capture the support of the bimodal distribution or at one of its modes as shown on the right. This is a result of forward and reverse KL divergence definitions in Equation 1.7. As we'll see in later chapters, by minimizing KL divergence we will effectively convert VI into an optimization problem.

1.3.2 Modern Deep Learning Algorithms

Over the years, deep learning architecture has changed from the basic building blocks of LeNet convolutional neural network (CNN) to inception modules of GoogLeNet. We saw the emergence of certain architectural design themes such as residual connections in the ResNet model, which became the standard architectural choice of modern neural networks of arbitrary depth. In the later chapters of this book, we'll be taking a look at modern deep learning algorithms which include self-attention based Transformers, generative models such as Variational Auto-Encoders (VAE) and Graph Neural Networks.

We will also take a look at amortized variational inference which is an interesting research area as it combines the expressiveness and representation learning of deep neural networks with domain knowledge of probabilistic graphical models. We will see one such application of Mixture Density Networks, where we'll use a neural network to map from observation space to the parameters of the approximate posterior distribution.

A vast majority of deep learning models fall in the area of narrow AI showing high-performance on a specific dataset. While it is a useful skill to be able to do well on a narrow set of tasks, we would like to generalize away from narrow AI and towards Artificial General Intelligence (AGI).

1.4 Implementing Algorithms

A key part of learning algorithms from scratch is software implementation. It's important to write good code that is both efficient in terms of its use of data structures and has low algorithmic complexity. Throughout this chapter, we'll be grouping the functional aspects of the code into classes and implementing different computational methods from scratch. Thus, you'll be exposed to a lot of object oriented programming (OOP) which is common practice with popular ML libraries such as scikit-learn. While the intention of this book is to write all code from scratch (without reliance on third party libraries), we can still use ML libraries (such as scikit-learn: https://scikit-learn.org/stable/) to check the results of our implementation if available. We'll be using Python language throughout this book.

1.4.1 Data Structures

A data structure is a way of storing and organizing data. Each data structure offers different performance trade-offs and some are more suitable for the task than others. We'll be using a lot of **linear data structures** such as fixed size arrays in our implementation of ML algorithms since the time to access an element in the array is constant `O(1)`. We'll also frequently use dynamically resizable arrays (such as lists in Python) to keep track of data over multiple iterations.

Throughout the book, we'll be using **non-linear data structures**, such as map (dictionary) and set. The reason is that ordered dictionary and ordered set are built upon self-balanced binary search trees (BSTs) that guarantee `O(nlogn)` insertion, search and deletion operations. Finally, a hash table or unordered map is another commonly used, efficient data structure with `O(1)` access time assuming no collisions.

1.4.2 Problem-Solving Paradigms

Many ML algorithms in this book can be grouped into four main problem-solving paradigms: complete search, greedy, divide and conquer, and dynamic programming. **Complete search** is a method for solving a problem by traversing the entire search space in search of a solution. A machine learning example where complete search takes place is an exact inference by complete enumeration. During exact inference, we must completely specify a number of probability tables to carry out our calculations. A **greedy algorithm** takes a locally optimum choice at each step with the hope of eventually reaching a globally optimum solution. Greedy algorithms often rely on a greedy heuristic. A machine learning

example of a greedy algorithm consists of sensor placement. For example, given a room and several temperature sensors, we would like to place the sensors in a way that maximizes room coverage. **Divide and conquer** is a technique that divides the problem into smaller, *independent* sub-problems and then combines the solutions to each of the sub-problems. A machine learning example that uses divide and conquer paradigm can be found in CART decision tree algorithm. As we'll see in a future chapter, in CART algorithm an optimum threshold for splitting a decision tree is found by optimizing a classification objective (such as Gini index). The same procedure is applied to a tree of depth one greater resulting in a recursive algorithm. Finally, **Dynamic Programming (DP)** is a technique that divides a problem into smaller, *overlapping* sub-problems, computes a solution for each sub-problem and stores it in a DP table. A machine learning example that uses dynamic programming occurs in Reinforcement Learning (RL) in finding a solution to Bellman equations. For a small number of states, we can compute the Q-function in tabular way using dynamic programming.

1.5 Summary

- An algorithm is a sequence of steps required to achieve a particular task. Machine learning algorithms can be grouped into supervised learning, unsupervised learning and deep learning.
- Understanding algorithms from scratch will help you choose the right algorithm for the task, explain the results, troubleshoot advanced problems and improve performance of existing models
- Bayesian inference allows us to update our beliefs about the world given observed data, while deep learning models learn representations of the world through the algorithm of back-propagation that minimizes an objective function. There are two camps of Bayesian inference: Markov Chain Monte Carlo (MCMC) and Variational Inference (VI) that deal with sampling and approximating the posterior distribution, respectively
- It is important to write good code that is both efficient in terms of its use of data structures and that has low algorithmic complexity. Many machine learning algorithms in this book can be grouped into four main problem-solving paradigms: complete search, greedy, divide and conquer, and dynamic programming.

2

Markov Chain Monte Carlo

This chapter covers

- Introduction to Markov Chain Monte Carlo (MCMC)
- Estimating Pi via Monte Carlo Integration
- Binomial Tree Model Monte Carlo Simulation
- Self-Avoiding Random Walk
- Gibbs Sampling Algorithm
- Metropolis-Hastings Algorithm
- Importance Sampling

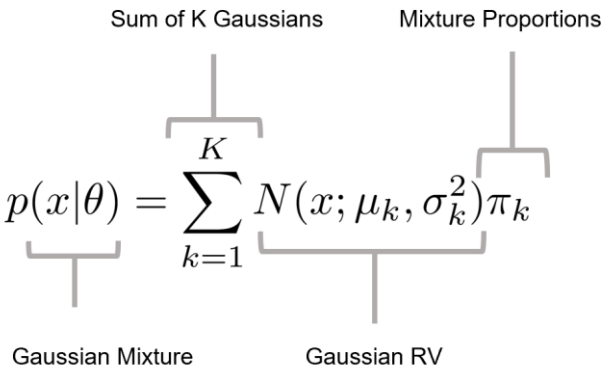
In the previous chapter we reviewed different types of ML algorithms and software implementation. Now we are going to focus on one popular class of ML algorithms known as Markov Chain Monte Carlo. Any probabilistic model that explains a part of reality contains multiple parameters and can be described by distributions that live in high dimensional parameter spaces. Markov Chain Monte Carlo (MCMC) is a methodology of sampling from high dimensional parameter spaces, in order to approximate the posterior distribution $p(\theta|x)$. Originally developed by physicists this method became popular in Bayesian statistics community because it allows one to estimate high dimensional posterior distributions using sampling. The basic idea behind MCMC is to construct a **Markov chain** whose stationary distribution is equal to the target posterior $p(\theta|x)$. In other words, if we perform a random walk across the parameter space, the fraction of time we spend in a particular state θ is proportional to $p(\theta|x)$. We'll start by introducing MCMC in the following section. We'll proceed with three warm-up Monte Carlo examples (Estimating Pi, Binomial Tree Model and self-avoiding random walk) before looking at three popular sampling algorithms (Gibbs sampling, Metropolis-Hastings and Importance sampling).

2.1 Introduction to Markov Chain Monte Carlo

Let's start by understanding high dimensional **parameter spaces** based on a simple example of classifying Iris. Iris is a species of flowers consisting of three types: Setosa, Versicolor and Virginica. The flowers are characterized by their petal and sepal length and width which can be used as features to determine the Iris type. Figure 2.1 shows the Iris pairplot.

Figure 2.1 Iris pairplot: pair-wise scatterplots color coded by Iris species

A pairplot is a matrix of plots where off-diagonal entries contain a scatter plot of every feature (such as petal length) against every other feature, while the main diagonal entries contain plots of every feature against itself color coded by the three Iris species. The pairplot captures pairwise relationships in the Iris dataset. Let's focus on the main diagonal entries and try to model the data we see. Since we have three types of Iris flowers, we can model the data as a mixture of three Gaussian distributions. Equation 2.1 captures this in mathematical terms:



Equation 2.1 Mixture of Gaussians

A mixture of Gaussians consists of `K` Gaussian RVs scaled by mixture proportions `πk` that are positive and add up to 1: `∑πk = 1, πk > 0` This is a high dimensional parameter problem because to fit the Gaussian mixture model we need to find the values for the means `μk`, covariances σ^2_k and mixture proportions `πk`. In the case of Iris dataset, we have `K=3`, which means that the number of parameters for Gaussian mixture is equal to `9-1=8`, i.e. we have `9` parameters in $\theta = \{\mu_k, \sigma^2_k, \pi_k\}^3_{(k=1)}$ and we subract `1` because of the sum-to-one constraint for `πk`. In a later chapter, we'll look at how to find the Gaussian mixture parameters via Expectation-Maximization (EM) algorithm.

2.1.1 Posterior Distribution of Coin Flips

In MCMC algorithms, we are going after approximating the posterior distribution through samples. In fact, most of Bayesian inference is designed to efficiently approximate the posterior distribution. Let's understand what a posterior distribution is in a little more detail. Posterior arises when we have a model with parameters `θ` that we are trying to fit to observed data `x`. Namely, it's the probability of parameters given the data: `p(θ|x)`. Consider an example of a sequence of coin flips where every coin is heads with probability `θ` and tails with probability `1-θ` as shown in Figure 2.2

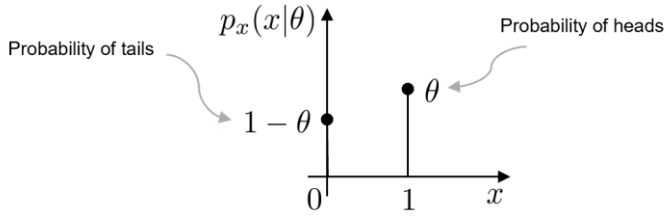


Figure 2.2 Bernoulli Random Variable

Figure 2.2 shows the probability mass function (PMF) of a Bernoulli RV modeling the coin flip. If $\theta=1/2$ we have a fair coin with equal chance of heads (1) or tails (0), otherwise, we say that the coin is biased with the bias equal to θ. We can write down the PMF of Bernoulli RV as follows:

$$p_x(x|\theta) = \theta^x(1-\theta)^{1-x},\ x \in \{0,1\}$$

Equation 2.2 Bernoulli Random Variable

The mean of Bernoulli RV is equal to, $E[X]= \sum_x x\ p_x\ (x|\theta) = 0(1-\theta)+1(\theta)= \theta$ while the variance is equal to $VAR(x)=E[x^2]-E[x]^2= \theta- \theta^2= \theta(1-\theta)$. Let's go back to our coin tossing example. Let $D=\{x_1,\ldots,x_n\}$ be a sequence of independent and identically distributed (iid) coin flips. Then we can compute the likelihood as follows:

$$p_x(D|\theta) = p_x(x_1,...,x_n|\theta) = \prod_{i=1}^{n} p(x_i|\theta) = \prod_{i=1}^{n} \theta^{x_i}(1-\theta)^{1-x_i} = \theta^{N_1}(1-\theta)^{N_0}$$

Equation 2.3 Likelihood of n coin flips

Where $N_1 = \sum^{n}_{(i=1)} x_i$ or the total number of counts for which the coin landed heads and $N_0=n-N_1$ or the total number of counts for which the coin landed tails.

Equations 2.2 and 2.3 have the form $p(x|\theta)$ which is by definition the likelihood of data x given the parameter θ. What if we have some prior information about the parameter θ, in other words we know something about $p(\theta)$, such as the number of heads and tails we obtained in another experiment with the same coins. We can capture this prior information as follows:

$$\text{Beta}(\theta|a,b) \propto \theta^{a-1}(1-\theta)^{b-1}$$

Equation 2.4 Unnormalized Beta Prior Distribution

where `a` is the number of heads and `b` is the number of tails in our previous experiment. When computing the posterior distribution, we are interested in computing `p(θ|x)`. We can use Bayes rule from Chapter 1 to express the posterior distribution as proportional to the product of the likelihood and the prior:

$$p(\theta|D) \propto p(D|\theta)p(\theta) = \theta^{N_1}(1-\theta)^{N_0}\theta^{a-1}(1-\theta)^{b-1} = \theta^{N_1+a-1}(1-\theta)^{N_0+b-1} \propto \text{Beta}(\theta|N_1+a, N_0+b)$$

Equation 2.5 Beta Posterior Distribution

Equation 2.5 computes the posterior distribution `p(θ|D)` which tells us the probability of heads for a sequence of coin flips. We can see that the posterior is distributed as a Beta random variable. Remember that our prior was also a Beta random variable. We say that the prior is conjugate to the posterior. In other words, the posterior can be computed in closed form by updating the prior counts with observed data.

2.1.2 Markov Chain for Page Rank

Before we dive into MCMC algorithms, we need to introduce the concept of a **Markov chain**. A Markov chain is a sequence of possible events, in which the probability of the current event depends only on the previous event. A first-order Markov chain can be written as follows. Let $x_1, \ldots, x_t$ be the samples from our posterior distribution, then we can write the joint as:

$$p(x_1, ..., x_t) = p(x_1)p(x_2|x_1)p(x_3|x_2)\cdots p(x_t|x_{t-1}) = p(x_1)\prod_{i=2}^{t} p(x_i|x_{i-1})$$

Equation 2.6 Factorized First Order Markov Chain

Notice how the joint factors as a product of conditional distributions where the current state is conditioned only on the previous state. A Markov chain is characterized by the **initial distribution** over the states $p(x_1 = i)$ and a **state transition matrix** $A_{ij}=p(x_t=j \mid x_{t-1} = i)$ from state `i` to state `j`. Let's motivate Markov chains via Google page rank example. Google uses page rank algorithm to rank billions of web pages. We can formulate a collection of `n` web-pages as a graph `G=(V,E)`. Let every node $v \in V$ represent a web-page and let every edge $e \in E$ represent a link from one page to another. Knowing how the pages are linked allows us to construct a giant, sparse transition matrix `A`, where `Aij` is the probability of following a link from page `i` to page `j` as shown in Figure 2.3

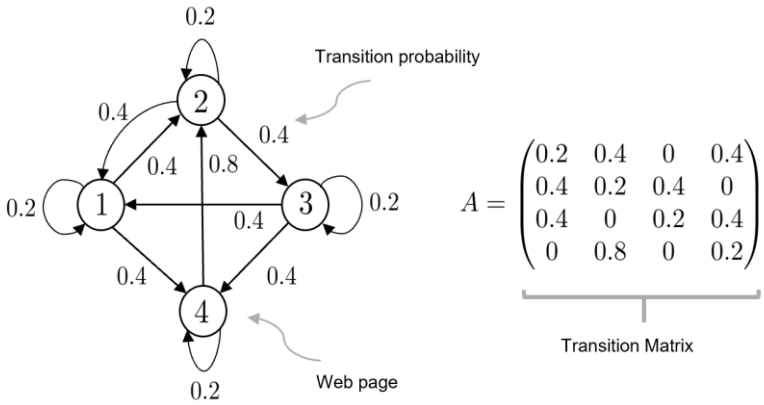


Figure 2.3 A graph of web-pages with transition probabilities

Note that every row in the matrix `A` sums to `1`, i.e. $\sum_j A_{ij} = 1$. A matrix with this property is known as a stochastic matrix. Note that `Aij` corresponds to transition probability from state `i` to state `j`. As we will see in a later chapter, the page rank is a stationary distribution over the states of the Markov chain. To make it scale to billions of web-pages, we'll derive from scratch and implement a power iteration algorithm. But for now, in the next few sections, we are going to look at different Markov Chain Monte Carlo sampling algorithms: from Gibbs sampler to Metropolis-Hastings to Importance Sampling. Let's start with a couple of warm up examples as our first exposure to Monte Carlo algorithms.

2.2 Estimating Pi

The first example we are going to look at is estimating the value of π via **Monte Carlo integration**. Monte Carlo (MC) integration has an advantage over numerical integration (which evaluates a function at a fixed grid of points) in that the function is only evaluated in places where there is non-negligible probability. Thus, MC integration scales better to high dimensional problems.

Let's take a look at the expected value of some function **f:** $\mathbb{R} \rightarrow \mathbb{R}$ of a random variable `Z=f(Y)`. We can approximate it by drawing samples `y` from the distribution `p(y)` as follows:

$$E[f(Y)] = \int f(y)p(y)dy \approx \frac{1}{S}\sum_{s=1}^{S} f(y_s), \text{ where } y_s \sim p(y)$$

Equation 2.7 Monte Carlo Expectation

Let's now use the same idea but apply it to evaluating an integral `I =∫f(x)dx`. We can approximate the integral as follows:

$$I = \int_a^b f(x)dx = \int_a^b w(x)p(x)dx = E_p[w(x)] = \frac{1}{n}\sum_{i=1}^{N} w(x_i), \text{ where } x_i \sim p(x)$$

Equation 2.8 Monte Carlo Integration

where `p(x) ~ Unif(a,b) = 1/(b-a)` is the pdf of a uniform random variable over the interval `(a,b)` and `w(x) = f(x)(b-a)` is our scaled function `f(x)`.

As we increase the number of samples `N`, our empirical estimate of the mean becomes more accurate. In fact, the standard error is equal to `σ/sqrt(N)`, where `σ` is the empirical standard deviation:

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(f(x_i) - I\right)^2$$

Equation 2.9 Empirical Variance

Now that we have an idea of how Monte Carlo integration works, let's use it to estimate `π`. We know that the area of a circle with radius `r` is `I=π r^2`. Alternatively, the area of the circle can be computed as an integral:

$$I = \int_{-r}^{r}\int_{-r}^{r} 1[x^2 + y^2 \leq r^2]\ dx\ dy = E_{x,y}[w(x,y)]$$

Equation 2.10 Area of a circle of radius r

where `1[x^2+y^2 ≤ r^2]` is an indicator function equal to `1` when a point is inside the circle of radius `r` and equal to `0` otherwise. Therefore, `π = I / r^2`. To compute `I` note that:

$$w(x,y) = (b_x - a_x)(b_y - a_y)1[x^2+y^2 \leq r^2] = (2r)(2r)1[x^2+y^2 \leq r^2] = 4r^2 1[x^2+y^2 \leq r^2]$$

Equation 2.11 Integrand used to estimate pi

We can summarize the Pi estimation algorithm in the following pseudo-code:

1: **function** pi_est(R, N):
2: **for** i = 1 to N:
3: $X[i] \sim \text{Unif}(-R, R)$
4: $Y[i] \sim \text{Unif}(-R, R)$ } Uniform RVs
5: $\text{IN[i]} = X[i]^2 + Y[i]^2 \leq R^2$
6: $\text{S[i]} = (2R) \times (2R) \times \text{IN[i]}$
7: **end for**
8: $\hat{I} = \frac{1}{N} \sum_{i=1}^{N} S[i]$
9: $\hat{\pi} = \hat{I}/R^2$ ← Pi Estimate
10: $\hat{\pi}_{se} = \sigma_S/\sqrt{N}$ ← Pi Standard Deviation
11: **return** $\hat{\pi} \pm \hat{\pi}_{se}$

We generate `N` samples from a Uniform distribution with support from `-R` to `R` and compute a Boolean expression of whether our sample is inside the circle or outside. If the sample is inside, it factors into the integral computation. Once we have an estimate of the integral, we divide the result by `R^2` to compute our estimate of pi.

We are now ready to implement our pi estimator!

Listing 2.1 Pi Estimator

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)    #A

def pi_est(radius=1, num_iter=int(1e4)):    #B


    X = np.random.uniform(-radius,+radius,num_iter)
    Y = np.random.uniform(-radius,+radius,num_iter)

    R2 = X**2 + Y**2
    inside = R2 < radius**2
    outside = ~inside

    samples = (2*radius)*(2*radius)*inside

    I_hat = np.mean(samples)
    pi_hat = I_hat/radius ** 2    #C
    pi_hat_se = np.std(samples)/np.sqrt(num_iter)    #D
    print("pi est: {} +/- {:f}".format(pi_hat, pi_hat_se))

    plt.figure()
    plt.scatter(X[inside],Y[inside], c='b', alpha=0.5)
    plt.scatter(X[outside],Y[outside], c='r', alpha=0.5)
    plt.show()

if __name__ == "__main__":

    pi_est()
```

#A fix a seed for reproducible results
#B experiment with different number of samples N
#C pi estimate
#D pi standard deviation

If we execute the code, we get π = 3.1348 ± 0.0164 which is a pretty good result (within the error bounds). Try experimenting with different numbers of samples `N` and see how fast we converge to π as a function of `N`. Figure 2.4 shows the accepted Monte Carlo samples in blue corresponding to samples inside the circle and the rejected samples in red.

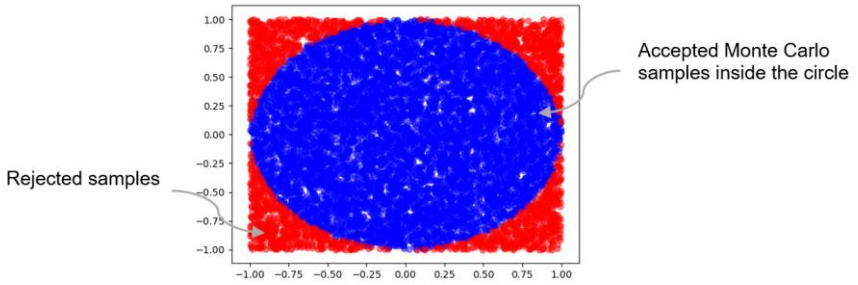


Figure 2.4 Monte Carlo samples used to estimate pi

In the above example, we assumed that samples came from a uniform distribution `X,Y ~ Unif(-R,R)`.In the following section, we are going to look at simulating a stock price over different time horizons using the binomial tree model.

2.3 Binomial Tree Model

Let's take a look now at how Monte Carlo sampling can be used in finance. In a binomial tree model of a stock price, we assume that at each time step, the stock could be in either up or down states with unequal payoffs characteristic for a risky asset. Assuming the initial stock price at time `t=0` is `$1`, at the next time step `t=1` the price is `u` in the up state and `d` in the down state with up-state transition probability `p`. A binomial tree model for the first two timesteps is shown in Figure 2.5. Note that the price in the next up (down) state is `u` (`d`) times the price of the previous state.

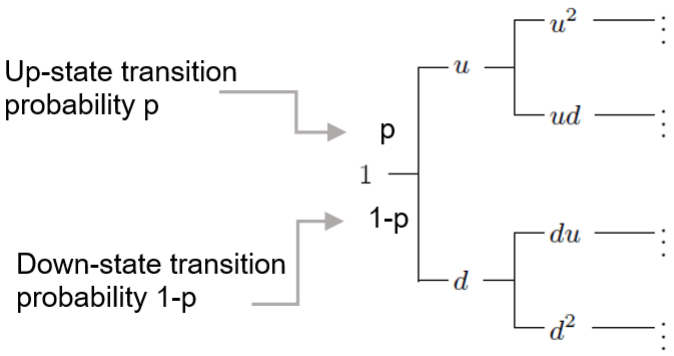


Figure 2.5 Binomial Tree Model

We can use Monte Carlo to generate uncertainty estimates of the terminal stock price after some time horizon `T`. When using a binomial model to describe the price process of the stock, we can use the following calibration:

$$
\begin{aligned}
u &= \exp\left(\sigma\sqrt{T/n}\right) = 1/d \\
p &= \frac{1}{2} + \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{T/n}
\end{aligned}
$$

Equation 2.12 Binomial Tree Calibration

where `T` is the length of prediction horizon in years and `n` is the number of time steps. We assume `1` year equals `252` trading days, `1` month equals `21` days, `1` week equals `5` days and `1` day equals `8` hours. Let's simulate the stock price using the binomial model with a daily time step for two different time horizons: `1` month from today and `1` year from today. We can summarize the algorithm in the following pseudo-code:

1: **function** binomial_tree($\mu, \sigma, S_0, N, T,$ step):
2: $u = \exp\left(\sigma\sqrt{T/n}\right)$ ← Up Price
3: $d = 1/u$ ← Down Price
4: $p = \frac{1}{2} + \frac{1}{2}\left(\frac{\mu}{\sigma}\right)\sqrt{T/n}$ ← Up State Transition Probability
5: up_times = Binomial(T/step, p, N)
6: down_times = T/step - up_times
7: $S_T = S_0 \times u^{up\_times} \times d^{down\_times}$
8: **return** S_T

We begin by initializing up price `u and` down price `d`, along with up state transition probability `p`. Next, we simulate all the up-state transitions by sampling from a Binomial random variable with the number of trials equal to `T/step`, success probability `p` and the number of monte carlo simulations equal to `N`. The down-state transitions are computed by complimenting up-state transitions. Finally, we compute the asset price by multiplying the initial price `S0` with the price of all the up-state transitions and all the down-state transitions.

Listing 2.2 Binomial Tree Stock Price Simulation

```
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt

np.random.seed(42)

def binomial_tree(mu, sigma, S0, N, T, step):

    #compute state price and probability
    u = np.exp(sigma * np.sqrt(step))    #A
    d = 1.0/u    #B
    p = 0.5+0.5*(mu/sigma)*np.sqrt(step)    #C

    #binomial tree simulation
    up_times = np.zeros((N, len(T)))
    down_times = np.zeros((N, len(T)))
    for idx in range(len(T)):
        up_times[:,idx] = np.random.binomial(T[idx]/step, p, N)
        down_times[:,idx] = T[idx]/step - up_times[:,idx]

    #compute terminal price
    ST = S0 * u**up_times * d**down_times

    #generate plots
    plt.figure()
    plt.plot(ST[:,0], color='b', alpha=0.5, label='1 month horizon')
    plt.plot(ST[:,1], color='r', alpha=0.5, label='1 year horizon')
    plt.xlabel('time step, day')
    plt.ylabel('price')
    plt.title('Binomial-Tree Stock Simulation')
    plt.legend()
    plt.show()

    plt.figure()
    plt.hist(ST[:,0], color='b', alpha=0.5, label='1 month horizon')
    plt.hist(ST[:,1], color='r', alpha=0.5, label='1 year horizon')
    plt.xlabel('price')
    plt.ylabel('count')
    plt.title('Binomial-Tree Stock Simulation')
    plt.legend()
    plt.show()

if __name__ == "__main__":

    #model parameters
    mu = 0.1    #D
    sigma = 0.15    #E
    S0 = 1    #F

    N = 10000    #G
    T = [21.0/252, 1.0]    #H
    step = 1.0/252    #I

    binomial_tree(mu, sigma, S0, N, T, step)
```

#A up state price

```
#B down state price
#C probability of up state
#D mean
#E volatility
#F starting price
#G number of simulations
#H time horizon in years
#I time step in years
```

Figure 2.6 shows that our yearly estimates have higher volatility compared to the monthly estimates.

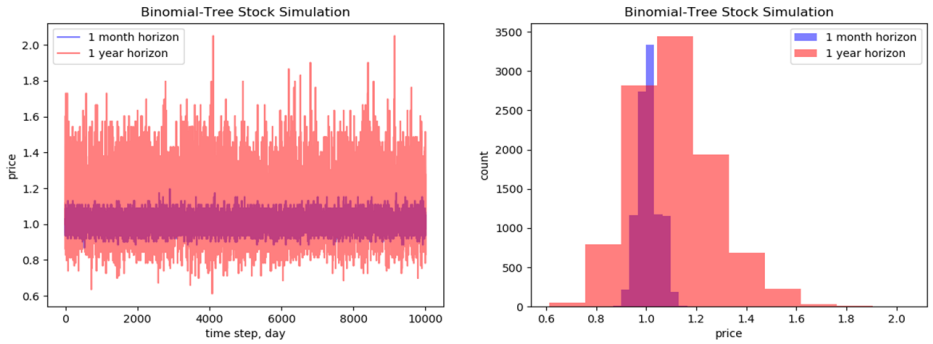


Figure 2.6 Binomial Tree Stock Simulation

This makes sense as we expect to encounter more uncertainty over long time horizons. In the next section, we are going to look into using Monte Carlo to simulate self-avoiding random walks.

2.4 Self-Avoiding Random Walk

Consider a random walk on a 2D grid. At each point on the grid, we take a random step with equal probability. This process forms a Markov chain $\{X\_i\}^{n}_{(i=1)}$ on $\mathbb{Z} \times \mathbb{Z}$ with $X_0=(0,0)$ and transition probabilities given by:

$$P(X_i = (k,l) | X_{i-1} = (i,j)) = \begin{cases} 1/4, & \text{if } |k-i| + |l-j| = 1 \\ 0, & \text{otherwise} \end{cases}$$

Equation 2.13 Random Walk Transition Matrix

In other words, we have an equal probability of $1/4$ for transitioning from a point on the grid to any of the four up, down, left, and right neighbors. In addition, *self-avoiding* random

walks are simply random walks that do not cross themselves. We can use monte carlo to simulate a self-avoiding random walk as shown in the following pseudo-code.

```
function rand_walk(num_step, num_iter, moves):
X, Y, lattice = 0, 0, 0
weight = 1
xx = num_step + 1 + X          <- Middle of the x-axis
yy = num_step + 1 + Y          <- Middle of the y-axis
lattice[xx, yy] = 1            <- Init grid position
for i = 1 to num_step:
    up = lattice[xx, yy+1]
    down = lattice[xx, yy-1]
    left = lattice[xx-1, yy]
    right = lattice[xx+1, yy]
    neighbors = [1, 1, 1, 1] - [up, down, left, right]   <- Available directions
    if sum(neighbors) == 0:
        break                  <- Self-loop
    end if
    weight = weight x sum(neighbors)   <- Compute importance weights
    direction ~ Cat(neighbors/sum(neighbors))   <- Sample a move direction
    X = X + moves[direction]
    Y = Y + moves[direction]
    //update grid coordinates
    xx = num_step + 1 + X
    yy = num_step + 1 + Y
    lattice[xx, yy] = 1
end for
return lattice
```

We start by initializing the grid (lattice) to an all zero matrix. We position the start of the random walk in the center of the grid as represented by `xx` and `yy`. Notice that `num_step` is the number of steps in a random walk. We begin each iteration by computing the `up, down, left, right` values which are equal to `1` if the grid is occupied and `0` otherwise. Therefore, we can compute available directions (neighbors) by computing `1` minus the direction. If the sum of neighbors is zero that means there are no available directions (all the neighboring grid cells are occupied) and the random walk is self-intersecting at which point we stop. Otherwise, we compute an importance weight as a product between the previous weight and sum of neighbors. The importance weights are used to compute the weighted mean square distances of the random walk. Next, we sample a move direction from the available directions

represented by a sample from the Categorical random variable. Since the neighbors array can only take the values of `0` and `1`, we are sampling uniformly from the available directions. Next, we update the grid coordinates `xx` and `yy`, and mark the occupancy by setting `lattice[xx, yy]=1`. While only a single iteration is shown in the pseudo-code, the code listing below wraps the pseudo-code in an additional for loop over the number of iterations `num_iter`. Each iteration is an attempt or a trial to produce a non-intersecting, i.e. self-avoiding random walk. The square distance of the random walk and the importance weight are recorded in each trial. We are now ready to look at the self-avoid random walk code.

Listing 2.3 Self-Avoiding Random Walk

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

np.random.seed(42)

def rand_walk(num_step, num_iter, moves):

 #random walk stats
 square_dist = np.zeros(num_iter)
 weights = np.zeros(num_iter)

 for it in range(num_iter):

     trial = 0
     i = 1

     while i != num_step-1:    #D

         #init
         X, Y = 0, 0
         weight = 1
         lattice = np.zeros((2*num_step+1, 2*num_step+1))
         lattice[num_step+1,num_step+1] = 1
         path = np.array([0, 0])
         xx = num_step + 1 + X
         yy = num_step + 1 + Y

         print("iter: %d, trial %d" %(it, trial))

         for i in range(num_step):

             up    = lattice[xx,yy+1]
             down  = lattice[xx,yy-1]
             left  = lattice[xx-1,yy]
             right = lattice[xx+1,yy]

             neighbors = np.array([1, 1, 1, 1]) - np.array([up, down, left, right])    #E

             if (np.sum(neighbors) == 0):    #F
                 i = 1
                 break
             #end if

             weight = weight * np.sum(neighbors) #G
```

```
            direction = np.where(np.random.rand() <
      np.cumsum(neighbors/float(sum(neighbors))))    #H

            X = X + moves[direction[0][0],0]
            Y = Y + moves[direction[0][0],1]

            path_new = np.array([X,Y])
            path = np.vstack((path,path_new))    #I

            #update grid coordinates
            xx = num_step + 1 + X
            yy = num_step + 1 + Y
            lattice[xx,yy] = 1
        #end for

        trial = trial + 1
    #end while

    square_dist[it] = X**2 + Y**2    #J

    weights[it] = weight
#end for

mean_square_dist = np.mean(weights * square_dist)/np.mean(weights)
print("mean square dist: ", mean_square_dist)

#generate plots
plt.figure()
for i in range(num_step-1):
    plt.plot(path[i,0], path[i,1], path[i+1,0], path[i+1,1], 'ob')
plt.title('random walk with no overlaps')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

plt.figure()
sns.displot(square_dist)
plt.xlim(0,np.max(square_dist))
plt.title('square distance of the random walk')
plt.xlabel('square distance (X^2 + Y^2)')
plt.show()


if __name__ == "__main__":

    num_step = 150    #A
    num_iter = 100    #B
    moves = np.array([[0, 1],[0, -1],[-1, 0],[1, 0]])    #C

    rand_walk(num_step, num_iter, moves)
```

#A number of steps in a random walk
#B number of iterations for averaging results
#C 2-D moves
#D iterate until we have a non-crossing random walk
#E compute available directions
#F avoid self-loops

```
#G compute importance weights
#H sample a move direction
#I store sampled path
#J compute square extension
```

Figure 2.7 shows a self-avoiding random walk generated by the last iteration.

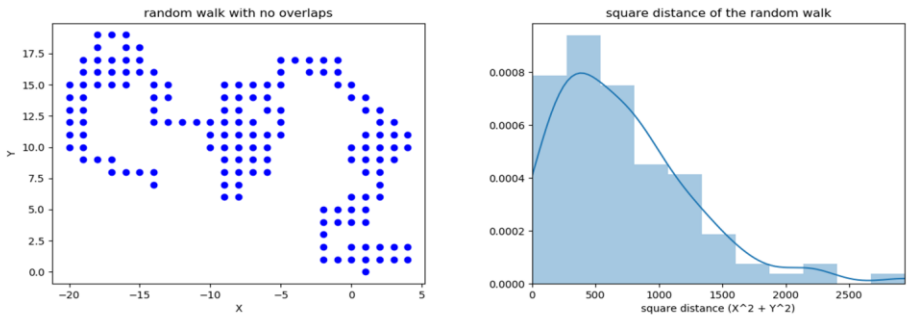


Figure 2.7 Self-Avoiding Random Walk (left) and Random Walk Square Distance (right)

There are `4^n` possible random walks of length `n` on the 2D lattice, and for large `n` it is very unlikely that a random walk will be self-avoiding. The Figure also shows a histogram of the square distance computed for each random walk iteration. The histogram is positively skewed showing a smaller probability of large distance walks. In the next section we are going to look at a popular sampling algorithm called the Gibbs sampling.

2.5 Gibbs Sampling

In this section we introduce one of the fundamental Markov Chain Monte Carlo (MCMC) algorithms called Gibbs sampling. It's based on the idea of sampling one variable at a time from a multi-dimensional distribution conditioned on the latest samples from all the other variables. For example, for a `d=3` dimensional distribution, given a starting sample `x^{k}`, we generate the next sample `x^{k+1}` as follows:

$$
\begin{aligned}
x_1^{k+1} &\sim p(x_1|x_2^k, x_3^k) \\
x_2^{k+1} &\sim p(x_2|x_1^{k+1}, x_3^k) \\
x_3^{k+1} &\sim p(x_3|x_1^{k+1}, x_2^{k+1})
\end{aligned}
$$

Equation 2.14 Gibbs Sampling

The distributions in the equations above are called fully conditional distributions. Also, notice that the naive Gibbs sampling algorithm is sequential (with the number of steps proportional to dimensionality of the distribution) and it assumes that we can easily sample

from the fully conditional distributions. The Gibbs sampling algorithm is applicable to scenarios where full conditional distributions in Equation 2.14 are easy to compute.

One instance where the fully conditional distributions are easy to compute is in the case of multi-variate Gaussians with pdf defined as follows:

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

Equation 2.15 Multivariate Gaussian Distribution

If we partition the Gaussian vector into two sets `x{1:D} = (xA,xB)` with parameters:

$$\begin{aligned}\mu &= \begin{pmatrix}\mu_A \\ \mu_B\end{pmatrix} \\ \Sigma &= \begin{pmatrix}\Sigma_{AA}\ \Sigma_{AB} \\ \Sigma_{BA}\ \Sigma_{BB}\end{pmatrix}\end{aligned}$$

Equation 2.16 Subsets of mean and variance

Then, it can be shown in Section 2.3 of Bishop, "Pattern Recognition and Machine Learning" that the full conditionals are given by:

$$\begin{aligned}p(x_A|x_B) &= \mathcal{N}(x_A|\mu_{A|B},\Sigma_{A|B}) \\ \mu_{A|B} &= \mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B-\mu_B) \\ \Sigma_{A|B} &= \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}\end{aligned}$$

Equation 2.17 Full Conditional Gaussian Distribution

Let's understand the Gibbs sampling algorithm by looking at the following pseudo-code.

1: **class** gibbs_gauss
2: **function** gauss_conditional(mu, Sigma, setA, x):
3: setU = set(range(len(mu))) ← Universal set
4: setB = setU \ setA
5: x_B, μ_A, μ_B = x[setB], mu[setA], mu[setB]
6: $\mu_{A|B} = \mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B)$
7: $\Sigma_{A|B} = \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}$
8: **return** $\mu_{A|B}$, $\Sigma_{A|B}$
9: **function** sample(mu, Sigma, xinit, num_samples):
10: x = xinit
11: dim = len(mu)
12: **for** s = 1 to num_samples:
13: **for** d = 1 to dim:
14: $\mu_{A|B}, \Sigma_{A|B}$ = gauss_conditional(mu, Sigma, set(d), x)
15: $x[d] \sim N(x; \mu_{A|B}, \Sigma_{A|B})$ ← Gibbs samples
16: **end for**
17: samples[s,:] = x
18: **end for**
19: **return** samples

Our `gibbs_gauss` class contains two functions: `gauss_conditional` and `sample`. The `gauss_conditional` function computes the conditional Gaussian distribution `p(xA|xB)` for any sets of variables `setA` and `setB`. `setA` is an input to the function, while `setB` is computed as a set difference between the universal set of dimension `D` and `setA`. Recall, that in Gibbs sampling we sample one dimension at a time while conditioning the distribution on all the other dimensions. In other words, we cycle through available dimensions and compute the conditional distribution in each iteration. That's exactly what `sample` function does. For each sample we iterate over each dimension and compute the mean and covariance of gauss conditional distribution from which we then take and record a sample. We repeat this process until the maximum number of samples is reached. Let's see the Gibbs sampling algorithm in action for sampling from a 2-D Gaussian distribution.

Listing 2.4 Gibbs Sampling

```
import numpy as np
import matplotlib.pyplot as plt

import itertools
from numpy.linalg import inv
from scipy.stats import multivariate_normal

np.random.seed(42)

class gibbs_gauss:

 def gauss_conditional(self, mu, Sigma, setA, x):    #A
     dim = len(mu)
     setU = set(range(dim))
     setB = setU.difference(setA)
     muA = np.array([mu[item] for item in setA]).reshape(-1,1)
     muB = np.array([mu[item] for item in setB]).reshape(-1,1)
     xB = np.array([x[item] for item in setB]).reshape(-1,1)

     Sigma_AA = []
     for (idx1, idx2) in itertools.product(setA, setA):
         Sigma_AA.append(Sigma[idx1][idx2])
     Sigma_AA = np.array(Sigma_AA).reshape(len(setA),len(setA))

     Sigma_AB = []
     for (idx1, idx2) in itertools.product(setA, setB):
         Sigma_AB.append(Sigma[idx1][idx2])
     Sigma_AB = np.array(Sigma_AB).reshape(len(setA),len(setB))

     Sigma_BB = []
     for (idx1, idx2) in itertools.product(setB, setB):
         Sigma_BB.append(Sigma[idx1][idx2])
     Sigma_BB = np.array(Sigma_BB).reshape(len(setB),len(setB))

     Sigma_BB_inv = inv(Sigma_BB)
     mu_AgivenB = muA + np.matmul(np.matmul(Sigma_AB, Sigma_BB_inv), xB - muB)
     Sigma_AgivenB = Sigma_AA - np.matmul(np.matmul(Sigma_AB, Sigma_BB_inv),
       np.transpose(Sigma_AB))

     return mu_AgivenB, Sigma_AgivenB

 def sample(self, mu, Sigma, xinit, num_samples):
     dim = len(mu)
     samples = np.zeros((num_samples, dim))
     x = xinit
     for s in range(num_samples):
         for d in range(dim):
             mu_AgivenB, Sigma_AgivenB = self.gauss_conditional(mu, Sigma, set([d]), x)
             x[d] = np.random.normal(mu_AgivenB, np.sqrt(Sigma_AgivenB))
         #end for
         samples[s,:] = np.transpose(x)
     #end for
     return samples

if __name__ == "__main__":

    num_samples = 2000
```

```
mu = [1, 1]
Sigma = [[2,1], [1,1]]
xinit = np.random.rand(len(mu),1)
num_burnin = 1000

gg = gibbs_gauss()
gibbs_samples = gg.sample(mu, Sigma, xinit, num_samples)

scipy_samples =
      multivariate_normal.rvs(mean=mu,cov=Sigma,size=num_samples,random_state=42)

plt.figure()
plt.scatter(gibbs_samples[num_burnin:,0], gibbs_samples[num_burnin:,1], label='Gibbs
      Samples')
plt.grid(True); plt.legend(); plt.xlim([-4,5])
plt.title("Gibbs Sampling of Multivariate Gaussian"); plt.xlabel("X1"); plt.ylabel("X2")
plt.show()

plt.figure()
plt.scatter(scipy_samples[num_burnin:,0], scipy_samples[num_burnin:,1], label='Ground
      Truth  Samples')
plt.grid(True); plt.legend(); plt.xlim([-4,5])
plt.title("Ground Truth Samples of Multivariate Gaussian"); plt.xlabel("X1");
plt.ylabel("X2")
plt.show()
```

#A computes P(X_A | X_B = x) = N(mu_{A|B}, Sigma_{A|B})

From Figure 2.8 we can see that Gibbs samples resemble the ground truth 2-D Gaussian distribution samples parameterized by $\mu = [1,1]^T$ and $\Sigma = [[2,1],[1,1]]$.

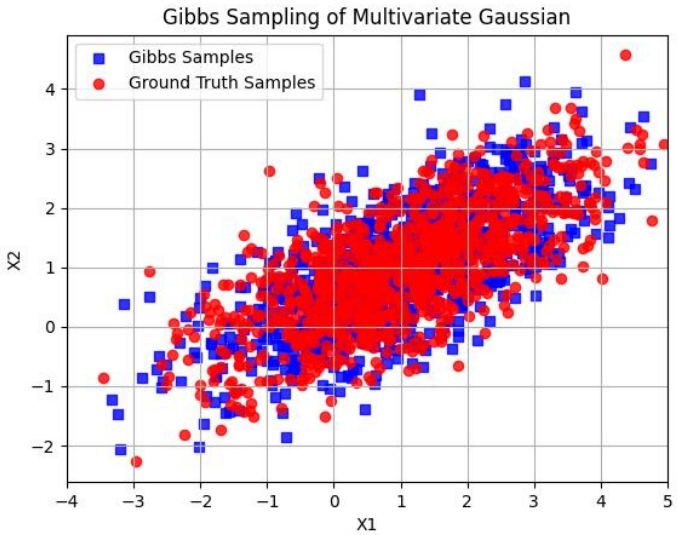


Figure 2.8 Gibbs Samples of Multivariate Gaussian

In the next section, we are going to look at a more general MCMC sampling algorithm called the Metropolis Hastings sampling.

2.6 Metropolis-Hastings Sampling

Let's look at a more general MCMC algorithm for sampling from distributions. Our goal is to construct a Markov chain whose stationary distribution is equal to our target distribution `p(x)`. The target distribution `p(x)` is the distribution (typically a posterior `p(θ|x)` or a density function `p(θ)` that we are interested in drawing samples from.

The basic idea in the Metropolis-Hastings (MH) algorithm is to propose a move from the current state `x` to a new state `x'` based on a proposal distribution `q(x'|x)` and then either accept or reject the proposed state according to MH ratio that ensures that detailed balance is satisfied, i.e.

$$p(x')q(x|x') = p(x)q(x'|x)$$

Equation 2.18 Detailed Balance Equation

The detailed balance equation says that the probability of transitioning out of state `x` is equal to the probability of transition into state `x`. To derive the MH ratio, assume for a moment that the detailed balanced equation above is not satisfied, then there must exist a correction factor `r(x'|x)`, s.t. the two sides are equal, solving for it leads to the MH ratio:

$$\begin{aligned} p(x')q(x|x') &= r(x'|x)p(x)q(x'|x) \\ r(x'|x) &= \min\left[1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right] \end{aligned}$$

Equation 2.19 MH ratio derivation

We can summarize the Metropolis-Hastings algorithm as follows:

1: Init x_0 at random
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: propose a new state $x' \sim q(x'|x_k)$ ← Sample from the proposal distribution
4: compute metropolis-hastings ratio:
5: $r(x'|x) = \min\left[1, \frac{p(x')q(x|x')}{p(x)q(x'|x)}\right]$
6: set $x_{k+1} = \begin{cases} x', & \text{with prob. } r(x'|x) \\ x_k, & \text{with prob. } 1 - r(x'|x) \end{cases}$ ← Accept the sample / ← Reject the sample
7: **end for**

Let's implement the MH algorithm for a Multivariate Mixture of Gaussian target distribution and a Gaussian proposal distribution:

$$\begin{aligned} p(x) &= \sum_{k=1}^{K} \pi(k)\mathcal{N}(x; \mu_k, \Sigma_k) \\ q(x'|x) &= \mathcal{N}(x'|x, \Sigma) \end{aligned}$$

Equation 2.20 Target and Proposal Distributions

Let's take a look at the code listing below. The `mh_gauss` class that consists of `target_pdf` function that defined the target distribution (in our case the Gaussian mixture `p(x)`), the `proposal_pdf` function that defines the proposal distribution (a multivariate normal), and the `sample` function, which samples a new state from the proposal `q(x'|xk)` conditioned on the previous state `xk`, computes the metropolis-hastings ratio and either accepts the new sample with probability `r(x'|x)` or rejects the sample with probability `1-r(x'|x)`

Listing 2.5 Metropolis Hastings Sampling

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.stats import uniform
from scipy.stats import multivariate_normal

np.random.seed(42)

class mh_gauss:

    def __init__(self, dim, K, num_samples, target_mu, target_sigma, target_pi,
       proposal_mu, proposal_sigma):
     self.dim = dim   #A
     self.K = K    #A
     self.num_samples = num_samples   #A
     self.target_mu = target_mu   #A
     self.target_sigma = target_sigma    #A
     self.target_pi = target_pi    #A
```

```
    self.proposal_mu = proposal_mu     #B
    self.proposal_sigma = proposal_sigma    #B

    self.n_accept = 0   #C
    self.alpha = np.zeros(self.num_samples)   #C
    self.mh_samples = np.zeros((self.num_samples, self.dim))    #C

def target_pdf(self, x):
    prob = 0    #D
    for k in range(self.K):    #D
        prob += self.target_pi[k]*\
            multivariate_normal.pdf(x,self.target_mu[:,k],self.target_sigma[:,:,k])    #D
    #end for    #D
    return prob

def proposal_pdf(self, x):
       return multivariate_normal.pdf(x, self.proposal_mu, self.proposal_sigma)    #E

def sample(self):
    x_init = multivariate_normal.rvs(self.proposal_mu, self.proposal_sigma, 1)   #F
    self.mh_samples[0,:] = x_init

    for i in range(self.num_samples-1):
        x_curr = self.mh_samples[i,:]
        x_new = multivariate_normal.rvs(x_curr, self.proposal_sigma, 1)

        self.alpha[i] = self.proposal_pdf(x_new) / self.proposal_pdf(x_curr)    #G
        self.alpha[i] = self.alpha[i] * (self.target_pdf(x_new)/self.target_pdf(x_curr))
      #G

        r = min(1, self.alpha[i])    #H
        u = uniform.rvs(loc=0, scale=1, size=1)
        if (u <= r):
            self.n_accept += 1
            self.mh_samples[i+1,:] = x_new  #I
        else:
            self.mh_samples[i+1,:] = x_curr #J
    #end for
    print("MH acceptance ratio: ", self.n_accept/float(self.num_samples))

if __name__ == "__main__":


    dim = 2
    K = 2
    num_samples = 5000
    target_mu = np.zeros((dim, K))
    target_mu[:,0] = [4,0]
    target_mu[:,1] = [-4,0]
    target_sigma = np.zeros((dim, dim, K))
    target_sigma[:,:,0] = [[2,1],[1,1]]
    target_sigma[:,:,1] = [[1,0],[0,1]]
    target_pi = np.array([0.4, 0.6])

    proposal_mu = np.zeros((dim,1)).flatten()
    proposal_sigma = 10*np.eye(dim)
```

```
    mhg = mh_gauss(dim, K, num_samples, target_mu, target_sigma, target_pi, proposal_mu,
       proposal_sigma)
    mhg.sample()

    plt.figure()
    plt.scatter(mhg.mh_samples[:,0], mhg.mh_samples[:,1], label='MH samples')
    plt.grid(True); plt.legend()
    plt.title("Metropolis-Hastings Sampling of 2D Gaussian Mixture")
    plt.xlabel("X1"); plt.ylabel("X2")
    plt.show()
```

```
#A target parameters: p(x) = \sum_k pi(k) N(x; mu_k, Sigma_k)
#B proposal parameters: q(x) = N(x; mu, Sigma)
#C sampling chain params
#D target pdf: p(x) = \sum_k pi(k) N(x; mu_k,Sigma_k)
#E proposal pdf: q(x) = N(x; mu, Sigma)
#F draw initial sample from the proposal
#G MH ratio
#H MH acceptance probability
#I accept
#J reject
```

From Figure 2.9, we can see that MH samples resemble the ground truth mixture of two 2-D Gaussian distributions with means `μ1 = [4,0]`, `μ2 = [-4,0]`, covariances `Σ_1 = [[2,1],[1,1]]` and `Σ_2 = [[1,0],[0,1]]` and mixture proportions `π = [0.4,0.6]`

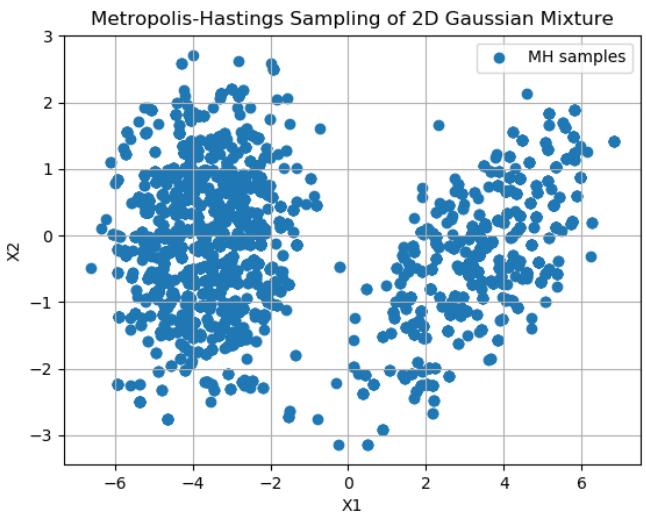


Figure 2.9 Metropolis-Hastings samples of multivariate Gaussian mixture

Notice that we are free to choose any proposal distribution `q(x'|x)` which makes the method flexible. A good choice of the proposal will result in high sample acceptance rate. In our implementation, we chose a symmetric Gaussian distribution centered on the current state:

$$q(x'|x) = \mathcal{N}(x'|x, \Sigma)$$

Equation 2.21 Proposal Distribution

This is known as a random walk Metropolis algorithm. If we use a proposal of the form `q(x'|x) = q(x')`, where the new state is independent of the old state, we get an independence sampler, which is similar to importance sampling that we will look at in the next section.

Also, notice that to compute the MH ratio we don't need to know the normalization constants `Z` of our distributions since we are taking the ratio and the `Z`'s cancel out. There's a connection between MH algorithm and Gibbs sampling: the Gibbs algorithm acceptance ratio is always `1`.

2.7 Importance Sampling

Importance Sampling (IS) is a Monte Carlo algorithm for estimating integrals of the form:

$$E[f(x)] = \int p(x)f(x)dx$$

Equation 2.22 Expectation of a function

The idea behind important sampling is to draw samples in interesting regions, i.e. where both `p(x)` and `|f(x)|` are large. Importance sampling works by drawing samples from an easier to sample proposal distribution `q(x)`. Thus, we can compute the expected value of `f(x)` with respect to the target distribution `p(x)` by drawing samples from the proposal `q(x)` and using Monte Carlo integration:

$$E[f(x)] = \int f(x)\frac{p(x)}{q(x)}q(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} w(x_i)f(x_i), \text{where } x_i \sim q(x)$$

Equation 2.23 Monte Carlo Expectation

where we defined the importance weights as `w(x) = p(x)/q(x)`. Let's look at an example where we use importance sampling to approximate an expected value of f(x) = 2 sin(πx/1.5), `x ≥ 0`. We are asked to take the expectation with respect to an unnormalized Chi distribution parameterized by a non-integer degree of freedom (DoF) parameter `k`:

$$p(x) \sim x^{(k-1)} \exp\{-\frac{x^2}{2}\}, x \geq 0$$

Equation 2.24 Unnormalized Chi distribution

We will use an easier to sample from proposal distribution q(x):

$$q(x) \sim \mathcal{N}(x; 0.8, 1.5)$$

Equation 2.25 Proposal distribution

Let's look at the following pseudo-code.

1: **class** importance_sampler
2: **function** sample(N):
3: **for** i = 1 to N:
4: $x_i \sim q(x) = N(x; \mu, \sigma^2)$ ← Sample from the proposal
5: $w(x_i) = p(x_i)/q(x_i)$ ← Compute importance weights
6: **end for**
7: $E[f(x)] = \frac{1}{N} \sum_{i=1}^{N} w(x_i) f(x_i)$
8: **return** $w(x), E[f(x)]$

Our `importance_sampler` class contains a function called `sample` that takes the number of samples `N` as its input. Inside the `sample` function, we loop over the number of samples and for each iteration we sample from the proposal distribution `q(x)` and compute the importance weight as the ratio of target distribution `p(x)` to proposal distribution `q(x)`. Finally, we compute Monte Carlo integral by weighting our function of interest `f(x)` by the importance weights, summing over all samples and dividing by `N`. Let's look at the following code listing that captures all the details.

Listing 2.6 Importance Sampling

```
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import quad
from scipy.stats import multivariate_normal

np.random.seed(42)

class importance_sampler:

 def __init__(self, k=1.5, mu=0.8, sigma=np.sqrt(1.5), c=3):
     self.k = k    #A

     self.mu = mu    #B
     self.sigma = sigma    #B
     self.c = c    #C

 def target_pdf(self, x):
     return (x**(self.k-1)) * np.exp(-x**2/2.0)    #D

 def proposal_pdf(self, x):
     return self.c * 1.0/np.sqrt(2*np.pi*1.5) * np.exp(-(x-self.mu)**2/(2*self.sigma**2))
       #E

 def fx(self, x):
        return 2*np.sin((np.pi/1.5)*x)    #F

 def sample(self, num_samples):
     x = multivariate_normal.rvs(self.mu, self.sigma, num_samples)   #G

     idx = np.where(x >= 0)  #H
     x_pos = x[idx]    #H

     isw = self.target_pdf(x_pos) / self.proposal_pdf(x_pos)    #I

     fw = (isw/np.sum(isw))*self.fx(x_pos)   #J
     f_est = np.sum(fw)    #J

     return isw, f_est


if __name__ == "__main__":

 num_samples = [10, 100, 1000, 10000, 100000, 1000000]

 F_est_iter, IS_weights_var_iter = [], []
 for k in num_samples:
     IS = importance_sampler()
     IS_weights, F_est = IS.sample(k)
     IS_weights_var = np.var(IS_weights/np.sum(IS_weights))
     F_est_iter.append(F_est)
     IS_weights_var_iter.append(IS_weights_var)

 #ground truth (numerical integration)
 k = 1.5
 I_gt, _ = quad(lambda x: 2.0*np.sin((np.pi/1.5)*x)*(x**(k-1))*np.exp(-x**2/2.0), 0, 5)
```

```
#generate plots
plt.figure()
xx = np.linspace(0,8,100)
plt.plot(xx, IS.target_pdf(xx), '-r', label='target pdf p(x)')
plt.plot(xx, IS.proposal_pdf(xx), '-b', label='proposal pdf q(x)')
plt.plot(xx, IS.fx(xx) * IS.target_pdf(xx), '-k', label='p(x)f(x) integrand')
plt.grid(True); plt.legend(); plt.xlabel("X1"); plt.ylabel("X2")
plt.title("Importance Sampling Components")
plt.show()

plt.figure()
plt.hist(IS_weights, label = "IS weights")
plt.grid(True); plt.legend();
plt.title("Importance Weights Histogram")
plt.show()

plt.figure()
plt.semilogx(num_samples, F_est_iter, label = "IS Estimate of E[f(x)]")
plt.semilogx(num_samples, I_gt*np.ones(len(num_samples)), label = "Ground Truth")
plt.grid(True); plt.legend(); plt.xlabel('iterations'); plt.ylabel("E[f(x)] estimate")
plt.title("IS Estimate of E[f(x)]")
plt.show()
```

```
#A target parameters p(x)
#B proposal parameters q(x)
#C fix c, s.t. p(x) < c q(x)
#D p(x) ~ Chi(k=1.5)
#E q(x) ~ N(mu,sigma)
#F function of interest f(x), x >= 0
#G sample from the proposal
#H discard netgative samples (since f(x) is defined for x >= 0)
#I compute importance weights
#J compute E[f(x)] = sum_i f(x_i) w(x_i), where x_i ~ q(x)
```

Figure 2.10 (left) shows the target pdf `p(x)`, the proposal pdf `q(x)`, and the integrand `p(x)f(x)`. Notice, that we scaled the proposal pdf `q(x)` by constant `c`, s.t. `p(x) < cq(x)` for all `x≥0`. Furthermore, we restricted the samples from `q(x)` to be positive (thus sampling from a truncated Gaussian) in order to meet our constraint of `x≥0` for `f(x)`. Figure (right) shows the improvement in the estimate of `E[f(x)]` vs the number of samples as compared to the ground truth computed via numerical integration.

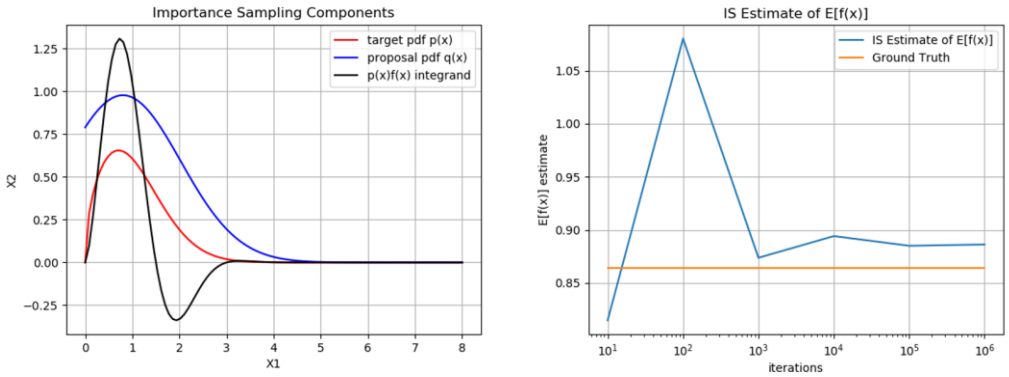


Figure 2.10 IS Components (left) and IS Estimate (right)

In the previous example, we used a normalized proposal distribution. However, for more complex distributions we often do not know the normalization constant (aka the partition function), since we are working with ratios of pdfs we'll see that the math holds for unnormalized distributions in addition we'll look at a way of estimating ratios of partition functions and discuss IS properties as an estimator.

Let's define a normalized pdf `p(x)` and `q(x)` as follows:

$$p(x) = \frac{1}{Z_p}\tilde{p}(x) \qquad q(x) = \frac{1}{Z_q}\tilde{q}(x)$$

Equation 2.26 Normalized PDFs

where `Zp` and `Zq` are normalization constants of `p(x)` and `q(x)`, respectively and $\tilde{p}(x)$ and $\tilde{q}(x)$ are unnormalized distributions. We can write our estimate as follows:

$$\begin{aligned} E[f(x)] &= \int f(x)p(x)dx = \int f(x)\frac{q(x)}{q(x)}p(x)dx = \frac{Z_q}{Z_p}\int f(x)\frac{\tilde{p}(x)}{\tilde{q}(x)}q(x)dx \\ &\approx \frac{Z_q}{Z_p}\frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s)f(s), \text{ where } s \sim q(x) \text{ and } \tilde{w}(s) = \frac{\tilde{p}(s)}{\tilde{q}(s)} \end{aligned}$$

Equation 2.27 Importance Sampling Estimate

Notice, in the equation above $\tilde{w}(s)$ are unnormalized importance weights. We can compute the ratio of normalizing constants as follows:

$$\frac{Z_p}{Z_q} = \frac{1}{Z_q}\int \tilde{p}(x)dx = \frac{1}{Z_q}\int \frac{\tilde{p}(x)}{q(x)}q(x)dx = \int \frac{\tilde{p}(x)}{\tilde{q}(x)}q(x)dx = \frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s)$$

Equation 2.28 Ratio of Normalization Constants

Combining the two expressions above, we get:

$$E[f(x)] \approx \frac{Z_q}{Z_p}\frac{1}{S}\sum_{s=1}^{S}\tilde{w}(s)f(s) = \frac{\frac{1}{S}\sum_s \tilde{w}(s)f(s)}{\frac{1}{S}\sum_s \tilde{w}(s)} = \sum_{s=1}^{S} w(s)f(s)$$

Equation 2.29 Importance Sampling Estimate

The equation above justifies our computation in the code example for the estimate of `E[f(x)]`. Let's look at a few properties of IS estimator and compute quantities relevant to characterizing the performance of IS. The IS estimator can be defined as follows:

$$\hat{a} = E[a(x)] = \frac{\frac{1}{n}\sum_{i=1}^{n}\tilde{w}^i a(x_i)}{\frac{1}{n}\sum_{i=1}^{n}\tilde{w}^i} = \frac{1}{n}\sum_{i=1}^{n} w_*^i a(x_i), \quad \text{where} \quad w_*^i = \tilde{w}^i / \frac{1}{n}\sum_i \tilde{w}^i$$

Equation 2.30 Importance Sampling Estimator

By the Weak Law of Large Numbers (WLLN), assuming independent samples `xi`, we know that the average of iid samples converges in probability to the true mean:

$$a = E[w_*(X_i)a(X_i)] \rightarrow b(\hat{a}) = E[\hat{a}] - a = 0$$

Equation 2.31 Unbiased Estimator

where `Xi` are assumed to be iid random variables distributed according to `q(x)`. As a result, IS is in theory an unbiased estimator. The variance of the IS estimator is given by:

$$\text{VAR}(\hat{a}) = \frac{\sigma^2}{n} = \frac{1}{n}\frac{\sum_{i=1}^{n}\left(\tilde{w}^i(a(x_i)-\hat{a})\right)^2}{\left(\sum_{i=1}^{n}\tilde{w}^i\right)^2}$$

Equation 2.32 Estimator Variance

To check whether IS estimate is consistent, we need to show that as we increase the number of samples n, the estimate converges to the true value a in probability:

$$\lim_{n\to\infty} P(|\hat{a}_n - a| > \epsilon) = 0$$

Equation 2.33 Convergence in Probability

Using Chebyshev's inequality, we can bound the deviation from a as follows:

$$\lim_{n\to\infty} P(|\hat{a} - a| > \epsilon) \leq \frac{\mathrm{VAR}(\hat{a})}{\epsilon^2} = \frac{\sigma^2}{n\epsilon^2} = 0$$

Equation 2.34 Chebyshev's Inequality

As a result, the IS estimator is consistent, here we didn't need to use an assumption that samples are independent but we needed to assume that the variance of importance weights is finite. Thus, we expect the variance around the estimate to shrink as we add more samples. One factor that affects the variance of the estimator is the magnitude of importance weights. Since `w(xi) = p(xi)/q(xi)` the weights are small if `q(xi)` is similar to `p(xi)` and has heavy tails. This is the desired case that leads to a small variance in our estimate.

We would like to have a way of telling when the importance weights are problematic, e.g. when only a couple of weights dominate the weighted sum. When samples are correlated the variance is `σ^2 / neff` , where `neff` is the effective sample size. To compute an expression for effective sample size, we set the variance of weighted average equal to the variance of unweighted average:

$$\mathrm{VAR}\left(\frac{1}{n}\sum_{i=1}^{n} a(x_i)\right) = \frac{\sigma^2}{n_{eff}} = \mathrm{VAR}\left(\frac{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i a(x_i)}{\frac{1}{n}\sum_{i=1}^{n} \tilde{w}^i}\right) \quad \text{where} \quad \mathrm{VAR}(a(x_i)) = \sigma^2$$

Equation 2.35 Effective Sample Size Derivation

Solving for `neff`, we get:

$$n_{eff} = \frac{(\sum_{i=1}^{n} w_i)^2}{\sum_{i=1}^{n} w_i^2} = \frac{n}{1 + \mathrm{VAR}(w_*^i)}$$

Equation 2.36 Effective Sample Size

We can use `neff` as a kind of diagnostics for the importance sampler where the target number `neff` depends on the application.

2.8 Exercises

2.1 Derive full conditionals `p(xA|xB)` for multivariate Gaussian distribution where `A` and `B` are sub-sets of `x1,x2,...,xn` of jointly Gaussian random variables.

2.2 Derive marginals `p(xA)` and `p(xB)` for multivariate Gaussian distribution where `A` and `B` are sub-sets of `x1,x2,...,xn` of jointly Gaussian random variables.

2.3 Let $y \sim N(\mu, \Sigma)$, where $\Sigma = LL^T$. Show that you can get samples y as follows: $x \sim N(0, I)$; $y = Lx + \mu$

2.9 Summary

- Markov Chain Monte Carlo (MCMC) is a methodology of sampling from high dimensional parameter spaces in order to approximate the posterior distribution $p(\theta|x)$
- Monte Carlo (MC) integration has an advantage over numerical integration (which evaluates a function at a fixed grid of points) in that the function is only evaluated in places where there is non-negligible probability
- In a binomial tree model of a stock price, we assume that at each time step, the stock could be in either up or down states with unequal payoffs characteristic for a risky asset.
- Self-avoiding random walks are simply random walks that do not cross themselves. We can use Monte Carlo to simulate a self-avoiding random walk.
- Gibbs sampling is based on the idea of sampling one variable at a time from a multi-dimensional distribution conditioned on the latest samples from all the other variables.
- The basic idea in the Metropolis-Hastings (MH) algorithm is to propose a move from the current state `x` to a new state `x'` based on a proposal distribution `q(x'|x)` and then either accept or reject the proposed state according to MH ratio that ensures that detailed balance is satisfied
- The idea behind important sampling is to draw samples in interesting regions, i.e. where both `p(x)` and `|f(x)|` are large. Importance sampling works by drawing samples from an easier to sample proposal distribution `q(x)`

3

Variational Inference

This chapter covers

- Introduction to KL Variational Inference
- Mean-field approximation
- Image Denoising in Ising Model
- Mutual Information Maximization

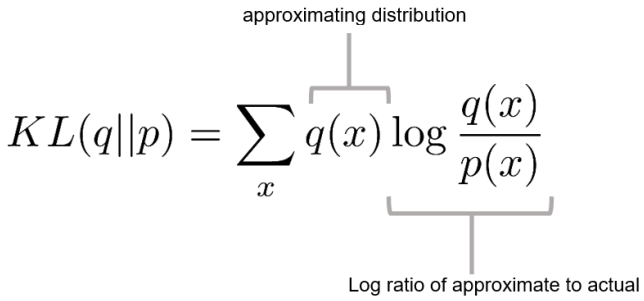
In the previous chapter, we covered one of the two main camps of Bayesian Inference: Markov Chain Monte Carlo. We examined different sampling algorithms and approximated the posterior distribution using samples. In this chapter, we are going to look at the second camp of Bayesian Inference: Variational Inference. Variational Inference (VI) is an important class of approximate inference algorithms. The basic idea behind VI is to choose an approximate distribution `q(x)` from a family of tractable or easy to compute distributions with trainable parameters and then make this approximation as close as possible to the true posterior distribution `p(x)`.

As we will see in the mean-field section, the approximate `q(x)` can take on a fully factored representation of the joint posterior distribution. This factorization significantly speeds up computation. We will introduce KL divergence and use it as a way to measure closeness of our approximate distribution to the true posterior. By optimizing KL divergence we will effectively convert VI into an optimization problem. In the following section, we will derive the Evidence Lower BOund (ELBO) and interpret it in three different ways, which will become handy during our implementation of mean-field for image denoising.

3.1 KL Variational Inference

We can use KL divergence to measure distance between probability distributions. This is particularly useful when making an approximation to the target distribution, since we want to

find out how close our approximation is. Let `q(x)` be our approximating distribution and `p(x)` be the target posterior distribution. Then the reverse KL is defined as follows:



Equation 3.1 Reverse KL definition

Consider a simple example, in which our target distribution is a standard univariate normal distribution `p(x)~N(0,1)` and our approximating distribution is a univariate normal with mean `μ` and variance σ^2: `q(x)~N(μ,σ²)` Then we can compute `KL(q||p)` as follows:

$$
\begin{aligned}
KL(q||p) &= \int q(x) \log \frac{q(x)}{p(x)} = -\int q(x) \log p(x) + \int q(x) \log q(x) \\
&= -\int q(x)\big[-\frac{1}{2}\log 2\pi - \frac{1}{2}x^2\big] + \int q(x)\big[-\frac{1}{2}\log 2\pi\sigma^2 - \frac{1}{2\sigma^2}(x-\mu)^2\big] \\
&= \big[\frac{1}{2}\log 2\pi + \frac{1}{2}(\sigma^2+\mu^2)\big] + \big[-\frac{1}{2}\log 2\pi\sigma^2 - \frac{1}{2}\big] \\
&= -\frac{1}{2}(1 + \log\sigma^2 - \mu^2 - \sigma^2)
\end{aligned}
$$

Equation 3.2 KL(q||p) where `p(x)~N(0,1)` and `q(x)~N(μ,σ²)`

We can visualize how `KL(q||p)` changes as we vary the parameters of our approximate distribution `q(x)`.Let's fix $\sigma^2=4$ and vary the mean `μ ∈[-4,4]`. We obtain Figure 3.1

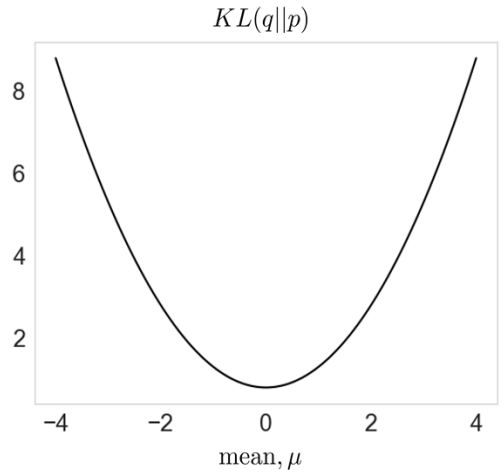


Figure 3.1 KL(q||p) for p(x)~N(0,1) and q(x)~N(μ,4)

Notice that the KL divergence is non-negative and it is smallest when `μ=0`, i.e. the mean of approximating distribution is equal to the mean of our target distribution `p(x)~N(0,1)`. We can interpret KL divergence as a measure of distance between distributions.

Let $\tilde{p}(x) = p(x)$ be the un-normalized distribution, then consider the following objective function:

$$J(q) = KL(q||\tilde{p}) = \sum_x q(x) \log \frac{q(x)}{p(x)Z} = \sum_x q(x) \log \frac{q(x)}{p(x)} - \log Z = KL(q||p) - \log Z$$

Equation 3.3 Objective Function

Since KL divergence is non-negative, `J(q)` is an upper bound on the marginal likelihood:

$$J(q) = KL(q||p) - \log Z \geq -\log Z = -\log p(D)$$

Equation 3.4 Upper Bound

when `q(x)` equals the true posterior `p(x)`, the KL divergence vanishes and the optimal value `J(q*)` equals the log partition function and for all other values of `q` it yields a bound. `J(q)` is called the **variational free energy** and can be written as:

$$\min_q J(q) = E_q[\log q(x)] + E_q[-\log \tilde{p}(x)] = -H(q) + E_q[E(x)]$$

Equation 3.5 Variational Free Energy

The variational objective function in Equation 3.4 is closely related to energy minimization in statistical physics. The first term acts as a regularizer by encouraging maximum entropy, while the second term is the expected energy and encourages the variational distribution `q` to explain the data.

The reverse KL that acts as a penalty term in the variational objective is also known as I-projection or information projection. In the reverse KL, `q(x)` will typically under-estimate the support of `p(x)` and will lock onto one of its modes. This is due to `q(x)=0` whenever `p(x)=0` to make sure the KL divergence stays finite. On the other hand, the forward KL, known as M-projection or moment projection is zero avoiding for `q(x)` and will over-estimate the support of `p(x)` as shown in Figure 3.2

Figure 3.2 Forward KL (left) q(x) over-estimates the support, while Reverse KL (right) q(x) locks onto a mode.

Figure 3.2 shows samples from a 2-D Gaussian mixture with 4 components `p(x)` as well as density ellipses of approximating distribution `q(x)`. We can see that optimizing forward KL leads to `q(x)` centered at zero (in low density region) as we over-estimate the support of `p(x)`. On the other hand, optimizing reverse KL leads to `q(x)` centered at one of the four modes of the Gaussian mixture.

We can use the Jensen's inequality to derive the Evidence Lower BOund (ELBO), an objective that we can maximize to learn the variational parameters of our model. Let `x` be our data and `z` be the latent variables, then we can derive our ELBO objective as follows:

$$\begin{aligned}\log p(x) &= \log \sum_z p(x,z) = \log \sum_z \frac{q(z)}{q(z)} p(x,z) = \log E_{q(z)}\left[\frac{p(x,z)}{q(z)}\right] \\ &\geq E_{q(z)}\left[\log \frac{p(x,z)}{q(z)}\right] = \underbrace{E_{q(z)}\left[\log p(x,z)\right]}_{\text{Energy Term}} - \underbrace{E_{q(z)}\left[\log q(x)\right]}_{\text{Entropy Term}} = \text{ELBO}\end{aligned}$$

Equation 3.6 ELBO derivation

Notice that the first term is the average negative energy and the second term is the entropy. Thus, a good posterior must assign most of its probability mass to regions of low energy (i.e. high joint probability density) while also maximizing the entropy of `q(z)`. Thus, Variational Inference, in contrast to MAP estimator, prevents `q(z)` from collapsing to an atom.

One form of ELBO emphasizes that the lower bound becomes tighter as the variational distribution better approximates the posterior:

$$\text{ELBO} = E_{q(z)}\left[\log \frac{p(x,z)}{q(z)}\right] = E_{q(z)}\left[\log \frac{p(z|x)p(x)}{q(z)}\right] = \underbrace{-KL(q(z)||p(z|x))}_{\text{distance between approximating q(z) and the posterior p(z|x)}} + \log p(x)$$

Equation 3.7 ELBO expression

Therefore, we can improve the ELBO by improving the model log evidence `log p(x)` through the prior `p(z)` or the likelihood `p(z|x)` or by improving the variational posterior approximation `q(z)`.

Finally, we can write the ELBO as follows:

$$\text{ELBO} = \frac{1}{n}\sum_{i=1}^{n}\left[\underbrace{E_{q(z)}\left[\log p(x_i|z_i)\right]}_{\text{sample likelihood}} - \underbrace{KL(q(z_i)||p(z_i))}_{\text{distance between approximating q(z) and the prior p(z) for sample i}}\right]$$

Equation 3.8 ELBO expression

this version emphasizes a likelihood term for the `i`-th observation and KL divergence term between each approximating distribution and the prior. In all cases above, the expectation wrt `q(z)` can be computed by sampling from our approximating distribution. Let's look at one of the most common variational approximations in the next section.

3.2 Mean-Field

One of the most popular forms of variational inference is called the **mean field approximation**, where we assume that the posterior is a fully factorized approximation of the form:

$$q(x) = \prod_i q_i(x_i)$$

Equation 3.9 Mean Field Approximation

where we optimize over the parameters of each marginal distribution `qi(xi)`. We can visualize fully factored distribution as in Figure 3.3.

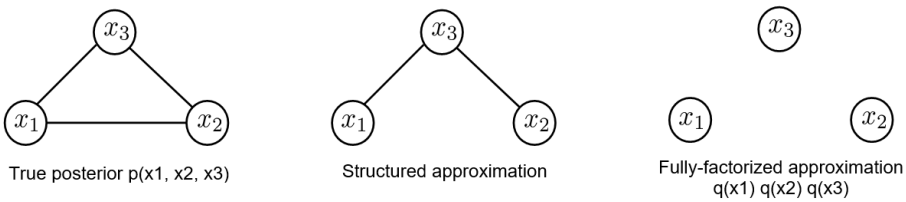


Figure 3.3 True Posterior (left), Structured Approximation (middle), Fully-Factored Approximation (right)

For a distribution with three random variables `x1`, `x2` and `x3`, we have the true posterior `p(x1,x2,x3)` that we are attempting to approximate by a fully factored distribution `q(x1)q(x2)q(x3)`.

Our goal is to minimize variational free energy `J(q)` or equivalently, maximize the lower bound:

$$L(q) = -J(q) = \sum_x q(x) \log \frac{\tilde{p}(x)}{q(x)}$$

Equation 3.10 Lower Bound

We can re-write the objective for each marginal distribution `qj`, keeping the rest of the terms as constants as in Section 21.3 of K. Murphy, "Probabilistic Machine Learning":

$$
\begin{aligned}
L(q_j) &= \sum_x \prod_i q_i(x_i) \left[\log \tilde{p}(x) - \sum_k \log q_k(x_k) \right] \\
&= \sum_{x_j} \sum_{x_{-j}} q_j(x_j) \prod_{i \neq j} q_i(x_i) \left[\log \tilde{p}(x) - \sum_k \log q_k(x_k) \right] \\
&= \sum_{x_j} q_j(x_j) \log f_j(x_j) - \sum_{x_j} q_j(x_j) \sum_{x_{-j}} \prod_{i \neq j} q_i(x_i) \left[\sum_{k \neq j} \log q_k(x_k) + \log q_j(x_j) \right] \\
&= \sum_{x_j} q_j(x_j) \log f_j(x_j) - \sum_{x_j} q_j(x_j) \log q_j(x_j) + \text{const}
\end{aligned}
$$

Mean field approx

Mean field approx

Factor out qj

Treat non-qj terms as constant

Equation 3.11 Objective for Marginal Distribution

Where we defined:

$$\log f_j(x_j) = \sum_{x_{-j}} \prod_{i \neq j} q_i(x_i) \log \tilde{p}(x) = E_{-q_j}[\log \tilde{p}(x)]$$

Equation 3.12 log f(x) definition

Since we are replacing the values by their mean value, the method is known as mean field. We can re-write `L(qj) = -KL(qj ||fj)` and therefore maximize the objective by setting `qj = fj` or equivalently:

$$\log q_j(x_j) = \log f_j(x_j) = E_{-q_j}[\log \tilde{p}(x)]$$

Equation 3.13 log q(x) definition

where the functional form of `qj` will be determined by the type of variables `xj` and their probability model. We will use this result in the next section in deriving the image denoising algorithm from scratch.

3.3 Image Denoising in Ising Model

The Ising model is an example of a Markov Random Field (MRF) and has its origins in statistical physics. A Markov Random Field is a set of random variables with a Markov property described by an undirected graph. The Ising model assumes that we have a grid of nodes, where each node can be in one of two possible states. The state of each node depends on the neighboring nodes through interaction potentials. In the case of images, this translates to a smoothness

constraint, i.e. a pixel prefers to be of the same color as the neighboring pixels. In the image denoising problem, we assume that we have a 2-D grid of noisy pixel observations of an underlying true image and we would like to recover the true image.

Let `yi` be noisy observations of binary latent variables `xi ∈ {-1,+1}`. We can write down the joint distribution as follows:

$$p(x,y) = p(x)p(y|x) = \prod_{(s,t)\in E} \Psi_{st}(x_s,x_t) \prod_{i=1}^{n} p(y_i|x_i) = \prod_{(s,t)\in E} \exp\{x_s w_{st} x_t\} \prod_{i=1}^{n} N(y_i|x_i,\sigma^2)$$

Equation 3.14 Ising Model Joint Distribution

where the interaction potentials are represented by `Ψst` for every pair of nodes `xs` and `xt` in a set of edges `E` and the observations `yi` are Gaussian with mean `xi` and variance σ^2. Here, `wst` is the coupling strength and assumed to be constant and equal to `J>0` indicating a preference for the same state as neighbors (i.e. the potential `Ψ(xs,xt)=exp{xs J xt}` is higher when `xs` and `xt` are both either `+1` or `-1)`.

To fit the model parameters using variational inference, we want to maximize the ELBO:

$$\begin{aligned} \text{ELBO} &= E_{q(x)}\Big[\log p(x,y)\Big] - E_{q(x)}\Big[\log q(x)\Big] = \\ &= E_{q(x)}\Big[\sum_{(s,t)\in E} x_s w_{st} x_t + \sum_{i=1}^{n} \log N(x_i;\sigma^2)\Big] - \sum_{i=1}^{n} E_{q_i(x)}\Big[\log q_i(x)\Big] \end{aligned}$$

Equation 3.15 Ising Model ELBO

where we are using the mean-field assumption of a fully-factored approximation `q(x)`:

$$q(x) = \prod_{i=1}^{n} q(x_i;\mu_i)$$

Equation 3.16 Mean-Field Approximation

Using the previously derived result in Equation 3.12, we state that `q(xi;μi)` that minimizes the KL divergence is given by:

$$q_i(x_i) = \frac{1}{Z_i} \exp\Big[E_{-q_i}\{\log p(x)\}\Big]$$

Equation 3.17 Optimum Approximating Distribution

where `E-q_i` denotes the expectation over every `qj` except for `j=i`. To compute, `qi(xi)` we only care about the terms that involve `xi`, i.e. we can isolate them as follows:

$$E_{-q_i}\{\log p(x)\} = E_{-q_i}\{x_i \sum_{j \in N(i)} w_{ij} x_j + \log N(x_i, \sigma^2) + \text{const}\} =$$
$$= x_i \sum_{j \in N(i)} J \times \mu_j + \log N(x_i, \sigma^2) + \text{const}$$

Equation 3.18 Expectation Derivation

where `N(i)` denotes the neighbors of node `i` and `μj` is the mean of a binary random variable.

$$\mu_j = E_{q_j}[x_j] = q_j(x_j = +1) \times (+1) + q_j(x_j = -1) \times (-1)$$

Equation 3.19 Mean of Binary Random Variable

Figure 3.4 shows the parametric form of our mean-field approximation for the Ising model:

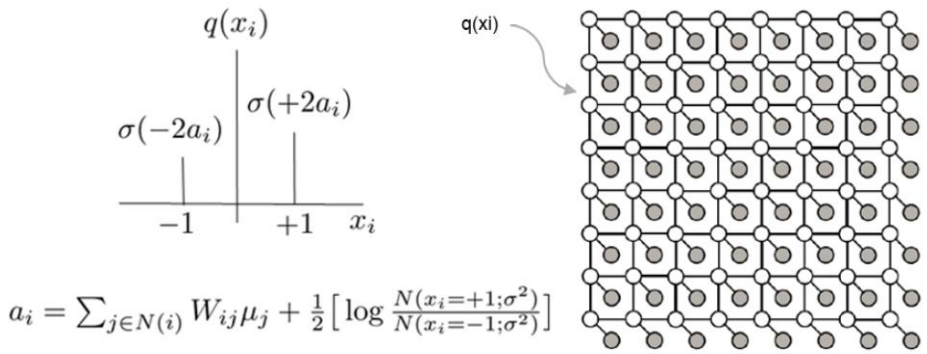


Figure 3.4 Ising model and its approximating distribution q(x)

In order to compute this mean, we need to know the values of `qj(xj=+1)` and `qj(xj=-1)`. Let $m_i = \Sigma_{j \in N(i)} w_{ij} \mu_j$ be the mean value of neighbors and let $L_i^+ = N(x_i = +1, \sigma^2)$ and $L_i^- = N(x_i = -1, \sigma^2)$, then we can compute the mean as follows:

$$
\begin{aligned}
q_i(x_i = +1) &= \frac{\exp\{m_i + L_i^+\}}{\exp\{m_i + L_i^+\} + \exp\{-m_i + L_i^-\}} \\
&= \frac{1}{1 + \exp\{-2m_i + L_i^- - L_i^+\}} = \frac{1}{1 + \exp\{-2a_i\}} = \sigma(2a_i)
\end{aligned}
$$

Equation 3.20 q(x) derivation

Where $a_i = m_i + 1/2(L^+_i - L^-_i)$ and `σ(x)` is a sigmoid function. Since `qi(xi=-1)=1-qi(xi=+1)=1-σ(2ai)=σ(-2ai)`, we can write the mean of our variational approximation `qi(xi)` as follows:

$$\mu_i = E_{q_i}[x_i] = \sigma(2a_i) - \sigma(-2a_i) = \tanh(a_i)$$

Equation 3.21 Mean of Binary Random Variable in the Ising Model

In other words, our mean-field updates of the variational parameters `μi` at iteration `k` are computed as follows:

$$\mu_i^{(k)} = \tanh\left(\sum_{j \in N(i)} w_{ij}\mu_j^{(k-1)} + \frac{1}{2}\left[\log\frac{N(x_i = +1, \sigma^2)}{N(x_i = -1, \sigma^2)}\right]\right) \times \lambda + (1 - \lambda) \times \mu_i^{(k-1)}$$

Equation 3.22 Mean field updates of variational parameters

where we added a learning rate parameter `λ ∈ (0,1]`. We further note that we can simplify the computation of ELBO term by term as follows:

$$
\begin{aligned}
&\sum_{(s,t)\in E} E_{q(x)}[x_s w_{st} x_t] = \frac{1}{2}\sum_{i=1}^{n}\sum_{j\in N(i)}\left(\sum_{x_i\in\{-1,+1\}}\sum_{x_j\in\{-1,+1\}} q_i(x_i)q_j(x_j)x_i J x_j\right) = \\
&\frac{1}{2}\sum_{i=1}^{n}\sum_{j\in N(i)}\Big(q_i(x_i = +1)JE[x_j] - q_i(x_i = -1)JE[x_j]\Big) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j\in N(i)} E[x_i]JE[x_j]
\end{aligned}
$$

ELBO 1st term

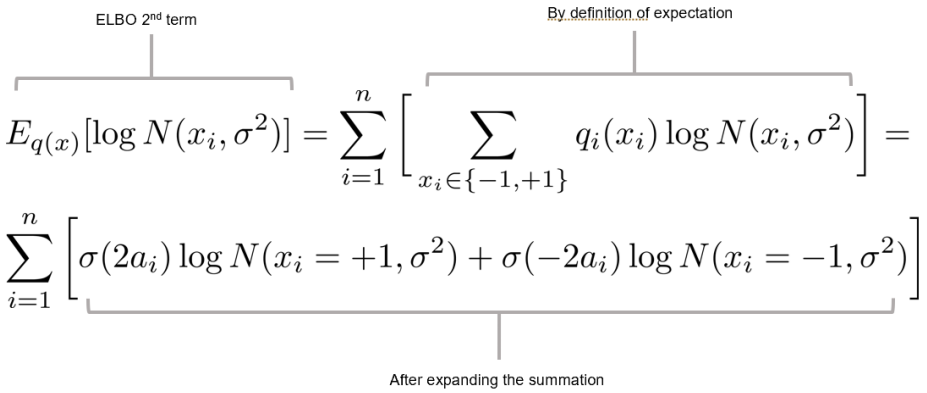
By definition of expectation

After substitution of values for xi and xj

By definition of expectation

Equation 3.23 Ising Model ELBO term

Similarly,



Equation 3.24 Ising Model ELBO term

To better understand the algorithm, let's look at the following pseudo-code.

class image_denoising
function mean_field(σ, y, w, λ, max_iter):
logp1 = $\log N(y; x_i = +1, \sigma^2)$
logm1 = $\log N(y; x_i = -1, \sigma^2)$
logodds = logp1 - logm1
p1 = sigmoid(logodds) //init
$\mu^{(0)}$ = 2 x p1 - 1 //init
for k=1 to max_iter:
    $\bar{S}_{ij} = \sum_{j \in N(i)} w_{ij} \mu_j^{(k-1)}$
    $\mu_i^{(k)} = \tanh(\bar{S}_{ij} + 1/2 \text{ logodds}) \times \lambda + (1 - \lambda) \times \mu_i^{(k-1)}$
    ELBO[k] = ELBO[k] + $1/2(\bar{S}_{ij} \times \mu_i^{(k)})$
    a = $\mu^{(k)}$ + 1/2 logodds
    qxp1 = sigmoid(+2a)
    qxm1 = sigmoid(-2a)
    Hx = $-\text{qxm1} \times \log(\text{qxm1}) - \text{qxp1} \times \log(\text{qxp1})$
    ELBO[k] = ELBO[k] + $\sum_{i=1}^{N}(\text{qxp1}[i] \times \text{logp1}[i] + \text{qxm1}[i] \times \text{logm1}[i]) + \sum_{i=1}^{N}(\text{Hx}[i])$
end for
return $\mu^{(k)}$

In the `image_denoising` class, we have a single method called `mean_field`, which takes as input the noise level sigma, noisy binary image `y`, the coupling strength `w=J`, learning rate lambda and max number of iterations. We start by computing logodds ratio, i.e. the probability of observing image pixel `y` under Gaussian random variable with means `+1` and `-1`. We then compute sigmoid function of logodds and use the result to initialize the mean variable. Next, we iterate until the max number of iterations and in each iteration we compute the influence of the neighbors `Sij`, which we include in the mean-field update equation. We then compute

our objective function `ELBO` and mean entropy `Hx` in order to monitor the convergence of the algorithm.

We now have all the tools we need to implement the mean-field variational inference for the Ising model in application to image denoising! In the following listing, we will read-in a noisy image and execute mean-field variational inference on a grid of pixels to denoise it.

Listing 3.1 Mean-Field Variational Inference in Ising Model

```
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

from PIL import Image
from tqdm import tqdm
from scipy.special import expit as sigmoid
from scipy.stats import multivariate_normal

np.random.seed(42)
sns.set_style('whitegrid')

class image_denoising:

    def __init__(self, img_binary, sigma=2, J=1):

        #mean-field parameters
        self.sigma  = sigma    #A
        self.y = img_binary + self.sigma*np.random.randn(M, N)    #B
        self.J = J    #C
        self.rate = 0.5    #D
        self.max_iter = 15
        self.ELBO = np.zeros(self.max_iter)
        self.Hx_mean = np.zeros(self.max_iter)

    def mean_field(self):

        #Mean-Field VI
        print("running mean-field variational inference...")
        logodds = multivariate_normal.logpdf(self.y.flatten(), mean=+1, cov=self.sigma**2)
       - \
                  multivariate_normal.logpdf(self.y.flatten(), mean=-1, cov=self.sigma**2)
        logodds = np.reshape(logodds, (M, N))

        #init
        p1 = sigmoid(logodds)
        mu = 2*p1-1    #E

        a = mu + 0.5 * logodds
        qxp1 = sigmoid(+2*a)  #q_i(x_i=+1)
        qxm1 = sigmoid(-2*a)  #q_i(x_i=-1)

        logp1 = np.reshape(multivariate_normal.logpdf(self.y.flatten(), mean=+1,
       cov=self.sigma**2), (M, N))
        logm1 = np.reshape(multivariate_normal.logpdf(self.y.flatten(), mean=-1,
       cov=self.sigma**2), (M, N))
```

```
        for i in tqdm(range(self.max_iter)):
            muNew = mu
            for ix in range(N):
                for iy in range(M):
                    pos = iy + M*ix
                    neighborhood = pos + np.array([-1,1,-M,M])
                    boundary_idx = [iy!=0,iy!=M-1,ix!=0,ix!=N-1]
                    neighborhood = neighborhood[np.where(boundary_idx)[0]]
                    xx, yy = np.unravel_index(pos, (M,N), order='F')
                    nx, ny = np.unravel_index(neighborhood, (M,N), order='F')

                    Sbar = self.J*np.sum(mu[nx,ny])
                    muNew[xx,yy] = (1-self.rate)*muNew[xx,yy] + self.rate*np.tanh(Sbar +
      0.5*logodds[xx,yy])
                    self.ELBO[i] = self.ELBO[i] + 0.5*(Sbar * muNew[xx,yy])
                #end for
            #end for
            mu = muNew

            a = mu + 0.5 * logodds
            qxp1 = sigmoid(+2*a) #q_i(x_i=+1)
            qxm1 = sigmoid(-2*a) #q_i(x_i=-1)
            Hx = -qxm1*np.log(qxm1+1e-10) - qxp1*np.log(qxp1+1e-10) #entropy

            self.ELBO[i] = self.ELBO[i] + np.sum(qxp1*logp1 + qxm1*logm1) + np.sum(Hx)
            self.Hx_mean[i] = np.mean(Hx)
        #end for
        return mu

if __name__ == "__main__":

    #load data
    print("loading data...")
    data = Image.open('./figures/bayes.bmp')
    img = np.double(data)
    img_mean = np.mean(img)
    img_binary = +1*(img>img_mean) + -1*(img<img_mean)
    [M, N] = img_binary.shape

    mrf = image_denoising(img_binary, sigma=2, J=1)
    mu = mrf.mean_field()

    #generate plots
    plt.figure()
    plt.imshow(mrf.y)
    plt.title("observed noisy image")
    plt.show()

    plt.figure()
    plt.imshow(mu)
    plt.title("after %d mean-field iterations" %mrf.max_iter)
    plt.show()

    plt.figure()
    plt.plot(mrf.Hx_mean, color='b', lw=2.0, label='Avg Entropy')
    plt.title('Variational Inference for Ising Model')
    plt.xlabel('iterations'); plt.ylabel('average entropy')
    plt.legend(loc='upper right')
    plt.show()
```

```
    plt.figure()
    plt.plot(mrf.ELBO, color='b', lw=2.0, label='ELBO')
    plt.title('Variational Inference for Ising Model')
    plt.xlabel('iterations'); plt.ylabel('ELBO objective')
    plt.legend(loc='upper left')
    plt.show()
```

#A noise level
#B y_i ~ N(x_i; sigma^2);
#C coupling strength (wij)
#D smoothing rate update
#E initial value of mu

Figure 3.5 shows experimental results for binary image denoising via mean-field variational inference.

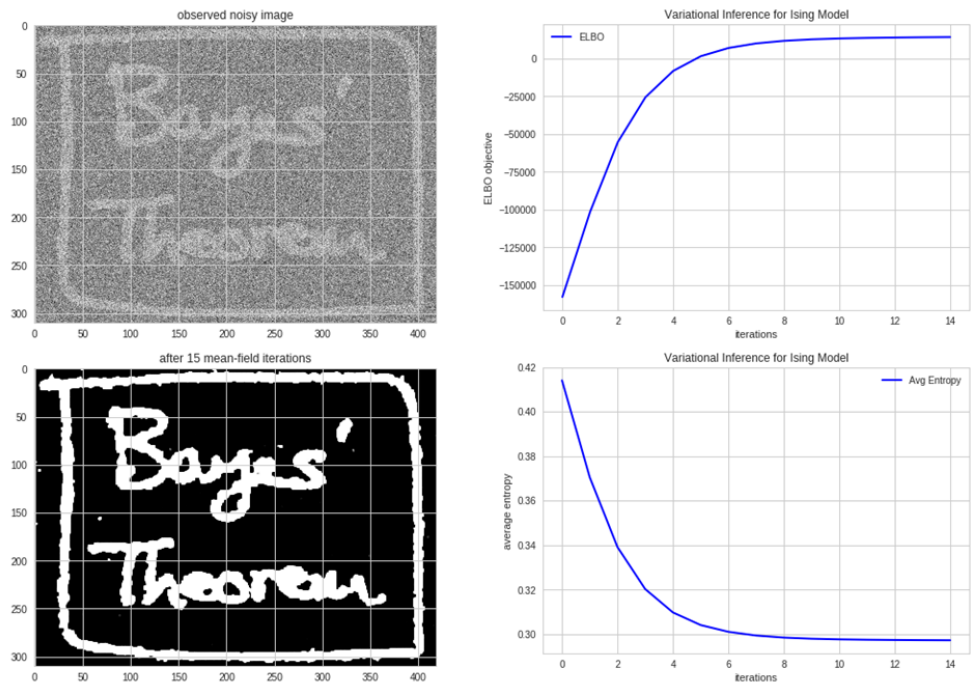


Figure 3.5 Mean-Field Variational Inference for Image Denoising in Ising Model

The noisy observed image is shown in the top-left obtained by adding Gaussian noise to each pixel and binarizing the image based on mean threshold. We then set the variational inference parameters such as the coupling strength `J=1`, noise level `σ=2`, smoothing rate `λ=0.5` and max number of iterations to `15`. The resulting de-noised binary image is shown in the bottom-left corner of the Figure. We can also see an increase in the ELBO objective (top-

right) and a decrease in the average entropy of our binary random variables `qi(xi)` representing the value of each pixel (bottom-right) as the number of mean-field iterations increases. The 2-D Ising model can be extended in multiple ways, for example: 3-D grids and `K`-states per node (aka Potts model).

3.4 MI Maximization

In this section, we look at mutual information (MI) maximization which commonly occurs in information planning and data communications settings. Consider a wireless communications scenario, in which we transmit a signal `x~px(x)`, it passes through a Multiple-Input Multiple-Output (MIMO) channel `H` and at the output we receive our signal, `y=Hx+n`, where `n~N(0,σ²I)` is an additive Gaussian noise. We would like to maximize the amount of information transmitted over the wireless channel. In other words, we would like to maximize the capacity or mutual information between the transmitted signal `X` and the received signal `Y`: `C=max I(X;Y)` where the maximization is taken over `p(x)`. To compute channel capacity, we discuss a general procedure based on KL divergence to approximately maximize mutual information.

$$I(X;Y) = H(X) - H(X|Y) = -E_{p(x)}\big[\log p(x)\big] - E_{p(x,y)}\big[\log p(x|y)\big]$$

Equation 3.25 Mutual Information (MI) definition

Let `q(x)` be an approximating distribution to `p(x)`, then consider KL divergence between the posterior distributions of `p` and `q`:

$$D_{KL}(p(x|y)||q(x|y)) \geq 0$$

Equation 3.26 Nonnegativity of KL divergence

We would like to derive a lower bound on Mutual Information (MI). Expanding the expression above, we can proceed as follows:

$$\sum_x p(x|y)\log p(x|y) - \sum_x p(x|y)\log q(x|y) \geq 0$$

Equation 3.27 KL divergence expansion

Multiplying both sides by `p(y)`, we get:

$$\sum_{x,y} p(y)p(x|y)\log p(x|y) \geq \sum_{x,y} p(x,y)\log q(x|y)$$

Equation 3.28 MI lower bound derivation

Recognizing the left-hand side as `-H(X|Y)`, we obtain the following MI lower bound:

$$I(X;Y) = H(X) - H(X|Y) \geq H(X) - E_{p(x,y)}\big[\log q(x|y)\big] = \tilde{I}(X;Y)$$

Equation 3.29 MI lower bound

Using the lower bound above, we can describe MI maximization Algorithm.

1: Choose approximating distribution family $Q(x;\theta)$
2: Initialize θ
3: **repeat**
4: for a fixed $q(x|y;\theta)$, find
5: $\theta^{new} = \arg\max_\theta \tilde{I}(X;Y)$
6: for a fixed θ, find
7: $q^{new}(x|y;\theta) = \arg\max_{q(x|y)\in Q} \tilde{I}(X;Y)$
8: **until** convergence

The above algorithm alternates between finding a set of parameters that maximize MI lower bound and finding the approximate distribution.

In this section, we saw how we can use the definitions of entropy, mutual information and KL divergence to derive a lower bound that could then be iteratively maximized by updating our approximation distribution `q`. In the following chapter, we will look at ML from computer science perspective and explore useful data-structures and algorithmic paradigms.

3.5 Exercises

3.1 Compute KL divergence between two univariate Guassians: $p(x) \sim N(\mu_1, \sigma^2_1)$ and $q(x) \sim N(\mu_2, \sigma^2_2)$.

3.2 Compute `E[X]`, `Var(X)`, and `H(X)` for a Bernoulli distribution.

3.3 Derive the mean, mode and variance of `Beta(a,b)` distribution.

3.6 Summary

- The main idea behind variational inference is to choose an approximate distribution `q(x)` from a family of tractable distributions and then make this approximation as close as possible to the true posterior distribution `p(x)`
- Evidence Lower BOund (ELBO) is an objective function that we seek to maximize to learn variational parameters of our model
- In mean-field approximation, we assume that the approximate distribution `q(x)` is fully factorized

- Mutual Information maximization can be carried out by deriving and maximizing MI lower bound.

4

Software Implementation

This chapter covers

- **Data Structures: Linear, Non-Linear, and Probabilistic**
- **Problem-Solving Paradigms: Complete Search, Greedy, Divide and Conquer, and Dynamic Programming**
- **ML Research: Sampling Methods and Variational Inference**

In the previous chapters, we looked at two main camps of Bayesian inference: Markov Chain Monte Carlo and Variational Inference. In this chapter, we review computer science concepts required for implementing algorithms from scratch. In order to write high quality code, it's important to have a good grasp of data structures and algorithm fundamentals. This chapter is designed to introduce common computational structures and problem-solving paradigms. Many of the concepts reviewed in this section are interactively visualized at https://visualgo.net/en

4.1 Data Structures

A data structure is a way of storing and organizing data. Data structures support a number of operations such as insertion, search, deletion, and updates, and the right choice of the data structure can simplify the run-time of an algorithm. Each data structure offers different performance trade-offs. As a result, it's important to understand how data structures work.

4.1.1 Linear

A data structure is considered linear if its elements are arranged in a linear fashion. The simplest example of a linear data structure is a **fixed size array** (where the size of the array may be specified as a constraint of the problem). The time it takes to access an element in an array is constant `O(1)`. In case the size of the array is not known ahead of time, it's better to

use a **dynamically resizable array** (e.g. a List in Python or a Vector in C++): these data structures are designed to handle resizing natively.

Two common operations applied to arrays are searching and sorting. The simplest search is a linear scan through all elements in `O(n)` time. If the array is sorted, we can use binary search in `O(log n)` time, which is an example of divide and conquer algorithm that we will discuss soon. Naive array sorting algoirthms such as selection sort and insertion sort have a complexity of $O(n^2)$ and only work well for small inputs. In general, comparison-based sorts where elements are compared pair-wise such as merge, heap, or quicksort, have the run-time of `O(n log n)` because the time complexity can be thought of as traversing a complete binary tree where each leaf represents one sorted ordering. In this representation, the height of the tree `h` is equal to algorithm run-time. Since there are `n!` possible orderings (leaf nodes), we can bound the run-time as follows:

$$\begin{aligned} h &= \log n! = \log n(n-1)(n-2) \times \cdots \times 1 = \log n + \log(n-1) + \cdots + \log 1 \\ &< \log n + \log n + \cdots + \log n = n \log n = \mathcal{O}(n \log n) \end{aligned}$$

Equation 4.1 Comparison-Based Sort Run-time Derivation

If we add additional constraints on the input, we can construct linear-time `O(n)` sorting algorithms such as Count sort, Radix sort and Bucket sort. For example, Count sort can be applied to integers in a small range and Radix sort works by applying Count sort digit by digit as discussed in Steven Halim's "Competitive Programming" book. Making algorithms distributed can further reduce run-time as described in "The Art of Multiprocessor Programming" by M. Herlihy and N. Shavit.

A **linked list** consists of nodes that store a value and a next pointer starting from the head node. It's usually avoided due to its linear `O(n)` search time. A **stack** allows `O(1)` insertions (push) and `O(1)` deletions (pop) in last in first out (LIFO) order. This is particularly useful in algorithms that are implemented recursively (e.g. bracket matching, topological sort). A **queue** allows `O(1)` insertion (enqueue) from the back and `O(1)` deletion (dequeue) from the front, thus following first in first out (FIFO) model. It's a commonly used data structure in algorithms such Breadth First Search (BFS) and algorithms based on BFS. In the next section, we are going to define and look at several examples of non-linear data structures.

4.1.2 Non-Linear

A data structure is considered non-linear if its elements do not follow a linear order. Examples of non-linear data structures include **map** (dictionary) and **set.** This reason is that ordered dictionary and ordered sets are built on self-balanced binary search trees (BST) that guarantee `O(n log n)` insertion, search, deletion operations. BSTs have the property that root node value is greater than its left child and less than its right child for every sub-tree. Self-balanced BSTs are typically implemented as Adelson-Velskii-Landis (AVL) or Red-Black (RB) trees, see CLRS "Introduction to Algorithms" for details. The difference between ordered map (dictionary) and ordered set data structures is that the map stores (key, value) pairs while the set only stores keys. A **heap** is another way to organize data in a tree representation. For example, an

array `A = [2,7,26,25,19,17,1,90,3,36]` can be represented as a tree as shown in Figure 4.1

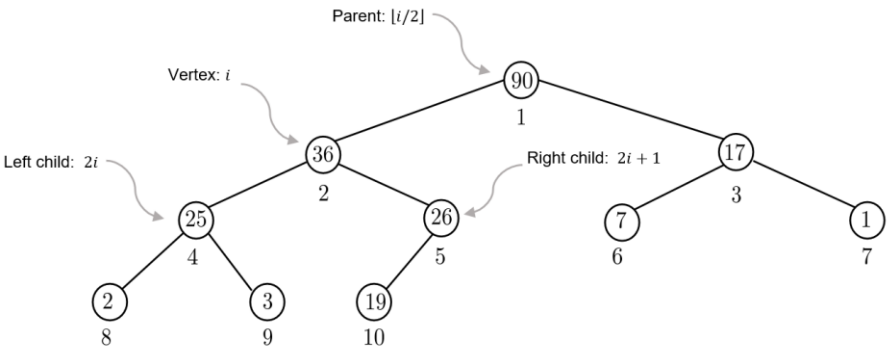


Figure 4.1 A binary heap

We can easily navigate the binary heap `A = [90,36,17,25,26,7,1,2,3,19]` by starting with a vertex `i` and using simple index arithmetic: `2i` to access left child, `2i+1` to access the right child and `⌊i/2⌋` to access the parent node. Instead of enforcing the BST property, the (max) heap enforces the heap property: in each subtree rooted at `x`, items on the left *and* right subtrees of `x` are smaller than (or equal to) `x`. This property guarantees that the top of the (max) heap is always the maximum element. Thus, (max) heap allows for fast extraction of the maximum element. Indeed, extract max and insert operations are achieved in `O(log n)` tree traversal, performing swapping operations to maintain the heap property whenever necessary. A heap forms the basis for a **priority queue** which is an important data structure in algorithms such as Prim and Kruskal Minimum Spanning Trees (MST), Dijkstra's Single-Source Shortest Paths (SSSP) and the `A*` search. Finally, a **hash table** or unordered map is a very efficient data structure with `O(1)` access assuming no collisions. One commonly used class of hash tables is direct addressing (DA), where they keys themselves are the indices. The goal of hash function is to uniformly distribute the elements in the table so as to minimize collisions. On the other hand, if you are looking to group similar items in the same bucket, locality sensitive hashing (LSH) allows you to find nearest neighbors as in "Beyond Locality-Sensitive Hashing", SODA 2014 by A. Andoni et al.

4.1.3 Probabilistic

Probabilistic data structures are designed to handle big data. They provide probabilistic guarantees and result in drastic memory savings. Probabilistic data structures tackle the following common big data challenges:

- Membership querying: Bloom filter, Counting Bloom filter, Quotient filter, Cuckoo filter
- Cardinality: Linear counting, probabilistic counting, LogLog, HyperLogLog,

HyperLogLog++

- Frequency: Majority algorithm, Frequent, Count Sketch, Count-Min Sketch
- Rank: Random sampling, q-digest, t-digest
- Similarity: LSH, MinHash, SimHash

For a comprehensive discussion on probabilistic data structures, please refer to "Probabilistic Data Structures and Algorithms for Big Data Applications" by A. Gakhov. In the next section, we are going to look at four main algorithmic paradigms.

4.2 Problem-Solving Paradigms

There are four main algorithmic paradigms: complete search, greedy, divide and conquer, and dynamic programming. Depending on the problem at hand, the solution can often be found by recalling the algorithmic paradigms. In this section, we'll discuss each strategy and provide an example.

4.2.1 Complete Search

Complete search (aka brute force) is a method for solving a problem by traversing the entire search space in search of a solution. During the search we can prune parts of the search space that we are sure do not lead to the required solution. For example, consider the problem of generating subsets. We can either use a recursive solution or an iterative one. In both cases, we terminate when we reach the required subset size. In the following listing, we will implement a complete search strategy based on generating subsets example.

Listing 4.1 Subset Generation

```
def search(k, n):
    if (k == n):
        print(subset)    #A
    else:
        search(k+1, n)
        subset.append(k)
        search(k+1, n)
        subset.pop()
    #end if

def bitseq(n):
    for b in range(1 << n):
        subset = []
        for i in range(n):
            if (b & 1 << i):
                subset.append(i)
        #end for
        print(subset)
    #end for

if __name__ == "__main__":
    n = 4
    subset = []
    search(0, n)    #B

    subset = []
    bitseq(n)   #C
```

#A process subset
#B recursive
#C iterative

A machine learning example where complete search takes place is in exact inference by complete enumeration, for details see Chapter 21 of "Information Theory, Inference and Learning Algorithms" by D. MacKay. Given a graphical model, we would like to factor a joint distribution according to conditional independence relations and use the Bayes rule to compute posterior probability of certain events. In this case, we need to completely fill out the necessary probability tables in order to carry out our calculation.

4.2.2 Greedy

A greedy algorithm takes a locally optimum choice at each step with the hope of eventually reaching a globally optimum solution. Greedy algorithms often rely on a greedy heuristic and one can often find examples in which greedy algorithms fail to achieve the global optimum. For example, consider the problem of fractional knapsack. A greedy knapsack problem is to select items to place in a knapsack of limited capacity `W` so as to maximize the total value of knapsack items, where each item has an associated weight and value. We can define a greedy heuristic to be a ratio of item value to item weight, i.e. we would like to greedily choose items that are simultaneously high value and low weight and sort the items based on this criteria. In the fractional knapsack problem, we are allowed to take fractions of an item (as opposed to

0–1 Knapsack). In the following listing, we will implement a greedy strategy based on fractional knapsack example.

Listing 4.2 Fractional Knapsack

```
class Item:
    def __init__(self, wt, val, ind):
        self.wt = wt
        self.val = val
        self.ind = ind
        self.cost = val // wt

    def __lt__(self, other):
        return self.cost < other.cost

class FractionalKnapSack:
    def get_max_value(self, wt, val, capacity):

        item_list = []
        for i in range(len(wt)):
            item_list.append(Item(wt[i], val[i], i))

        # sorting items by cost heuristic
        item_list.sort(reverse = True)  #O(nlogn)

        total_value = 0
        for i in item_list:
            cur_wt = int(i.wt)
            cur_val = int(i.val)
            if capacity - cur_wt >= 0:
                capacity -= cur_wt
                total_value += cur_val
            else:
                fraction = capacity / cur_wt
                total_value += cur_val * fraction
                capacity = int(capacity - (cur_wt * fraction))
                break
        return total_value

if __name__ == "__main__":
    wt = [10, 20, 30]
    val = [60, 100, 120]
    capacity = 50

    fk = FractionalKnapSack()
    max_value = fk.get_max_value(wt, val, capacity)
    print("greedy fractional knapsack")
    print("maximum value: ", max_value)
```

Since sorting is the most expensive operation, the algorithm runs in `O(n log n)` time. We can see that the input items are sorted in decreasing ratio of value / cost, after greedily selecting items `1` and `2`, we take a `2/3` fraction of item `3` for a total value of `60+100+(2/3)120 = 240`.

A machine learning example of a greedy algorithm consists of sensor placement. Given a room and several temperature sensors, we would like to place the sensors in a way that maximizes room coverage. A simple greedy approach is to start with an empty set `S0` and at

iteration `i` add the sensor `A` that maximizes the increment function, such as mutual information: $F_{MI}(A) = H(V \setminus A) - H(V \setminus A|A)$ where `V` is the set of all sensors. Where we used the identity $I(X;Y) = H(X) - H(X|Y)$. Turns out that mutual information is *submodular* if observed variables are indpendent given the latent state, which leads to efficient greedy submodular optimization algorithms with performance guarantees, e.g. see "Optimizing Sensing: Theory and Applications" PhD thesis by A. Kraus.

4.2.3 Divide and Conquer

Divide and Conquer is a technique that divides a problem into smaller, *independent* sub-problems and then combines solutions to each of the sub-problems. Examples of divide and conquer technique include sorting algorithms such as quick sort, merge sort and heap sort as well as binary search. The classic use of binary search is in searching for a value in a sorted array. First, we check the middle of the array to see if if contains what we are looking for. If it does or there are no more items to consider, we stop. Otherwise, we decide whether the answer is to the left or the right of the middle element and continue searching. As the size of the search space is halved after each check, the complexity of the algorithm is $O(\log n)$. In the following listing, we will implement a divide and conquer strategy based on binary search example.

Listing 4.3 Binary Search

```
def binary_search(arr, l, r, x):
    #assumes a sorted array
    if l <= r:
        mid = int(l + (r-l) / 2)

        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, l, mid-1, x)
        else:
            return binary_search(arr, mid+1, r, x)
        #end if
    else:
        return -1

if __name__ == "__main__":

    x = 5
    arr = sorted([1, 7, 8, 3, 2, 5])

    print(arr)
    print("binary search:")
    result = binary_search(arr, 0, len(arr)-1, x)

    if result != -1:
        print("element {} is found at index {}.".format(x, result))
    else:
        print("element is not found.")
```

A machine learning example that uses divide and conquer paradigm can be found in CART decision tree algorithm in which the threshold partitioning is done in a divide and conquer way,

and the nodes are split recursively until the maximum depth of the tree is reached. In CART algorithm, as we will see in the next chapter, an optimum threshold is found greedily by optimizing a classification objective (such as Gini index) and the same procedure is applied on a tree of depth one greater resulting in a recursive algorithm.

4.2.4 Dynamic Programming

Dynamic Programming (DP) is a technique that divides a problem into smaller *overlapping* sub-problems, computes a solution for each sub-problem and stores it in a DP table. The final solution is read off the DP table. Key skills in mastering dynamic programming is the ability to determine the problem states (entries of the DP table) and the relationships or transitions between the states. Then, having defined base cases and recursive relationships, one can populate the DP table in a top-down or bottom-up fashion. In the top-down DP, the table is populated recursively, as needed, starting from the top and going down to smaller sub-problems. In the bottom-up DP, the table is populated iteratively starting from the smallest sub-problems and using their solutions to build-on and arrive at solutions to bigger sub-problems. In both cases, if a sub-problem was already encountered, its solution is simply looked up in the table (as opposed to re-computing the solution from scratch). This dramatically reduces computational cost.

We use binomial coefficients example to illustrate the use of top-down and bottom-up DP. The code below is based on the recursion for binomial coefficients with overlapping sub-problems. Let `C(n,k)` denote `n` choose `k`, then, we have:

$$\begin{aligned} \text{Base case} &: C(n,0) = C(n,n) = 1 \\ \text{Recursion} &: C(n,k) = C(n-1,k-1) + C(n-1,k) \end{aligned}$$

Equation 4.2 Binomial Coefficients Recursion

Notice that we have multiple over-lapping sub-problems. E.g. For `C(n=5,k=2)` the recursion tree is as follows

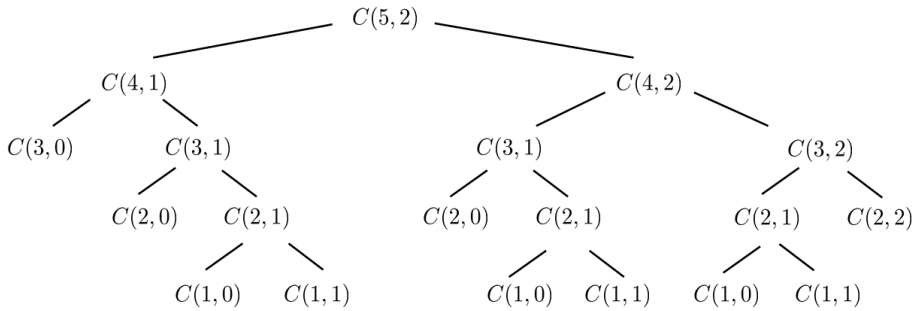


Figure 4.2 Binomial Coefficient C(5,2) Recursion

We can implement top-down and bottom-up DP as follows:

Listing 4.4 Binomial Coefficients

```
def binomial_coeffs1(n, k):
    #top down DP
    if (k == 0 or k == n):
        return 1
    if (memo[n][k] != -1):
        return memo[n][k]
    memo[n][k] = binomial_coeffs1(n-1, k-1) + binomial_coeffs1(n-1, k)
    return memo[n][k]

def binomial_coeffs2(n, k):
    #bottom up DP
    for i in range(n+1):
        for j in range(min(i,k)+1):
            if (j == 0 or j == i):
                memo[i][j] = 1
            else:
                memo[i][j] = memo[i-1][j-1] + memo[i-1][j]
            #end if
        #end for
    #end for
    return memo[n][k]
def print_array(memo):
    for i in range(len(memo)):
        print('\t'.join([str(x) for x in memo[i]]))

if __name__ == "__main__":
    n = 5
    k = 2
    print("top down DP")
    memo = [[-1 for i in range(6)] for j in range(6)]
    nCk = binomial_coeffs1(n, k)
    print_array(memo)
    print("C(n={}, k={}) = {}".format(n,k,nCk))

    print("bottom up DP")
    memo = [[-1 for i in range(6)] for j in range(6)]
    nCk = binomial_coeffs2(n, k)
    print_array(memo)
    print("C(n={}, k={}) = {}".format(n,k,nCk))
```

The time complexity is `O(nk)` and the space complexity is `O(nk)`. In the case of top-down DP, solutions to sub-problems are stored (memoized) as needed, whereas in the bottom-up DP, the entire table is computed starting from the base case.

A machine learning examples that uses dynamic programming occurs in Reinforcement Learning (RL) in finding solution to Bellman equations. We can write down the value of a state based on its reward at time `t` and sum of future discounted rewards as follows:

$$\begin{aligned} v_\pi(s) &= \Sigma_a \pi(a|s) \Sigma_{r,s'} p(r, s'|s, a)\left[r + \gamma v_\pi(s')\right] \\ &= E_\pi\left[R_t + \gamma v_\pi(S_{t+1})|S_t = s\right] \end{aligned}$$

Equation 4.3 Bellman Optimality

The above equation is known as Bellman optimality equation for `v(s)`. We can recover the optimum policy by solving for action that maximizes state reward described by the Q-function:

$$\pi(s) = \arg\max_a q(s, a)$$

Equation 4.4 Optimum Policy

For a small number of states and actions we can compute the `Q`–function in tabular way using dynamic programming. In RL, we often want to balance exploration and exploitation, in which case we take the argmax above with probability `1-ϵ` and take a random action with probability `ϵ`.

4.3 ML Research: Sampling Methods and Variational Inference

In this section, we focus on ML research. It is an important skill to have if you want to stay current in the field.

We focus on the latest developments in the area of sampling methods and variational inference. As we observed in this chapter, many modern ML algorithms include clever algorithms to approximate hard-to-compute posterior densities as a result of intractable, high-dimensional integrals involved in the computation of the posterior.

It is worth comparing briefly the differences between MCMC and variational inference. The advantages of variational inference is that for small to medium problems it is usually faster, it is deterministic, it is easy to determine when to stop, and it often provides a lower bound on log-likelihood. The advantages of sampling are that it is often simpler to implement, it is applicable to broader range of problems (e.g. problems without nice conjugate priors) and sampling can be faster when applied to really big models or datasets. See Chapter 24, "Machine Learning: A Probabilistic Perspective" by K. Murphy for additional discussion.

In addition to the classic MCMC sampling algorithms we studied in previous chapters, a few others deserving attention are Slice Sampling (R. Neal, "Slice Sampling", Annals of Statistics, 2003), Hamiltonian Monte Carlo (HMC) (R. Neal, "MCMC Using Hamiltonian Dynamics", arXiv, 2012) and the No-U-Turn Sampler (NUTS) (M. Hoffman et al, "The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo", JMLR, 2014). The state-of-the-art NUTS algorithm is commonly used as MCMC inference method for probabilistic graphical models and is implemented in the PyMC3, Stan, TensorFlow Probability, and Pyro probabilistic programming libraries.

There have been a number of attempts to scale MCMC for big data leading to Stochastic Monte Carlo methods that can be generally grouped into stochastic gradient based methods, methods using approximate Metropolis-Hastings with randomly sampled mini-batches and data augmentation. Another popular class of MCMC algorithms are streaming Monte Carlo that approximate the posterior for online Bayesian inference. Sequential Monte Carlo (SMC) rely on re-sampling and propagating samples over time with a large number of particles. Parallelizing Monte Carlo algorithms is another big area of research. If blocks of independent samples can be drawn from the posterior or a proposal distribution, the sampling algorithm could be parallelized by running multiple independent samplers on separate machines and then

aggregating the results. Additional methods include divide-and-conquer and pre-fetching (J. Zhu et al, "Big Learning with Bayesian Methods", National Science Review, 2017).

Advances in Variational Inference (VI) span scalable VI, which includes stochastic approximations, generic VI which extends the applicability of VI to non-conjugate models, accurate VI, that includes variational models beyond mean-field approximation and amortized VI which implements the inference over local latent variables with inference networks (C. Zhang, et al, "Advances in Variational Inference", IEEE Transactions on Pattern Analysis and Machine Intelligence, 2019). Scalable VI includes Stochastic Variational Inference (SVI) that applies stochastic optimization techniques of the variational objective. The efficiency of Stochastic Gradient Descent (SGD) depends on the variance of gradient estimates (smaller gradient noise allows for larger learning rates and leads to faster convergence). Techniques such as adaptive learning rates and mini-batch size as well as variance reduction such as control variates, non-uniform sampling and other approaches are used to speed up convergence.

In addition to stochastic optimization, leveraging model structure can help achieve the same objective. Examples include collapsed, sparse, parallel and distributed inference. The collapsed VI relies on the idea of analytically integrating out certain model parameters. Sparse inference exploits either sparsely distributed parameters or datasets that can be summarized by a small number of representative points. In addition, Structured Variational Inference examines variational distributions that are not fully factorized leading to more accurate approximations.

Finally, amortized variational inference is an interesting research area that combines probabilistic graphical models and neural networks. The term amortized refers to utilizing inference from past computations to support future computations. Amortized inference became a popular tool for inference in deep latent variable models (DLVM) such as Variational Auto-Encoder (VAE) (D. Kingma et al, "Auto-encoding variational Bayes", arXiv, 2013). Similarly, neural networks can be used to learn the parameters of conditional distributions in directed probabilistic graphical models (PGM) (D. Kingma, "Variational Inference and Deep Learning: A New Synthesis", PhD Thesis, 2017).

4.4 Exercises

4.1 Prove binomial identity: $C(n,k) = C(n-1,k-1) + C(n-1,k)$

4.2 Derive the Gibbs inequality: $H(p,q) \geq H(q)$, where $H(p,q) = -\sum_x p(x) \log q(x)$ is the cross-entropy and $H(q) = -\sum q(x) \log q(x)$ is the entropy

4.3 Use Jensen's inequality with $f(x) = \log(x)$ to prove $AM \geq GM$ inequality

4.4 Prove that $I(x;y) = H(x) - H(x|y) = H(y) - H(y|x)$

4.5 Summary

- A data structure is a way of storing and organizing data. Data structures can be categorized into: linear, non-linear and probabilistic.

- We looked at four algorithmic paradigms in this chapter: complete search, greedy, divide and conquer, and dynamic programming.
- Complete search (aka brute force) is a method for solving a problem by traversing the entire search space in search of a solution. During the search we can prune parts of the search space that we are sure do not lead to the required solution.
- A greedy algorithm takes a locally optimum choice at each step with the hope of eventually reaching a globally optimum solution.
- Divide and Conquer is a technique that divides a problem into smaller, *independent* sub-problems and then combines solutions to each of the sub-problems.
- Dynamic Programming (DP) is a technique that divides a problem into smaller *overlapping* sub-problems, computes a solution for each sub-problem and stores it in a DP table. The final solution is read off the DP table.
- Mastery of algorithms and software implementation can be achieved through practice of competitive programming (see Appendix for resources)
- Machine learning mastery requires a solid understanding of fundamentals. See recommended texts section in the Appendix for ideas on how to increase the depth and breadth of your knowledge.
- The field of machine learning is rapidly evolving and the best way to stay on top of latest research is by digesting conference papers.

5 Classification Algorithms

This chapter covers

- Introduction to Classification
- Perceptron Algorithm
- SVM Algorithm
- SGD Logistic Regression
- Bernoulli Naïve Bayes Algorithm
- Decision Tree (CART) Algorithm

In the previous chapter, we looked at computer science fundamentals required to implement ML algorithms from scratch. In this chapter, we focus on supervised learning algorithms. Classification is a fundamental class of algorithms and is widely used in machine learning. We will derive from scratch and implement a number of selected classification algorithms to build our experience with fundamentals and motivate the design of new ML algorithms.

5.1 Introduction to Classification

In **supervised learning**, we are given a dataset $D=\{(x_1,y_1),\ldots,(x_n,y_n)\}$ consisting of tuples of data x and labels y. The goal of a **classification algorithm** is to learn a mapping from inputs x to outputs y, where y is a discrete quantity, i.e. $y \in \{1,\ldots,K\}$. If $K=2$, we have a binary classification problem, while for $K>2$ we have multi-class classification.

A classifier h can be viewed as a mapping between a d-dimensional feature vector $\phi(x)$ and a k-dimensional label y, i.e. $h:R^d \rightarrow R^k$. We often have several models to choose from, let's call this set of classifier models H. Thus, for a given $h \in H$, we can obtain a prediction $y=h(\phi(x))$. Note that we are typically interested in predicting on new or unseen data. In other words, our classifier h must be able to **generalize** to new data samples.

Finding the right classifier is known as **model selection**. We want to choose a model that has sufficient number of parameters (degrees of freedom) so as to not under-fit and not over-fit to training data as shown in Figure 5.1

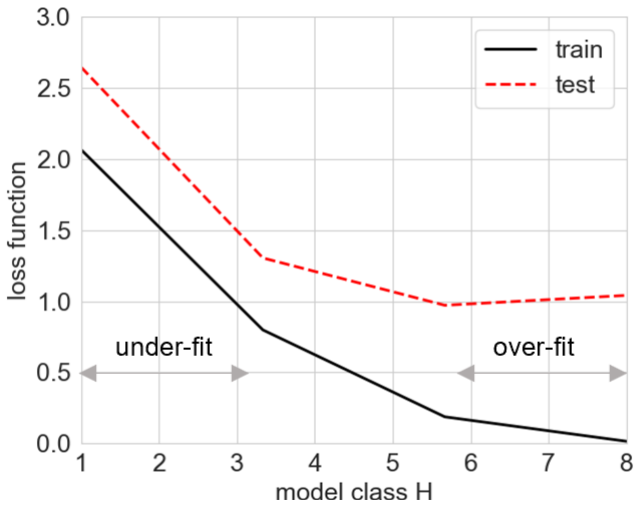


Figure 5.1 Model selection to avoid over-fitting and under-fitting to training data

For model classes `H=[1,2,3]`, training and test loss functions are both decreasing, which indicates that there's more capacity to learn and as a result these models under-fit the data. For model classes `H=[6,7,8]`,the training loss decreases while the test loss is starting to increase, which indicates that we are over-fitting the data.

5.2 Perceptron

Let's start with the most basic classification model: a **linear classifier**. We'll be using the perceptron classifier on the Iris dataset. We can define a linear classifier as follows:

$$h(x;\theta) = \text{sign}(\theta_1 x_1 + \cdots + \theta_d x_d + \theta_0) = \text{sign}(\theta \cdot x + \theta_0) = \begin{cases} +1 & \text{if } \theta \cdot x + \theta_0 \geq 0 \\ -1 & \text{if } \theta \cdot x + \theta_0 < 0 \end{cases}$$

Equation 5.1 Linear classifier

Notice how the sign function of the inner product between the parameter `θ` and the feature input `x` maps to `±1` labels. Geometrically, $\theta x + \theta_0 = 0$ describes a hyper-plane in `d`-dimensional space uniquely determined by the normal vector `θ`. Any point that lies on the same side as

the normal `θ` is labeled `+1`, while any point on the opposite side is labeled `-1`. As a result, `θx+θ0=0` represents the **decision boundary**. Figure 5.2 illustrates these concepts in `2` dimensions.

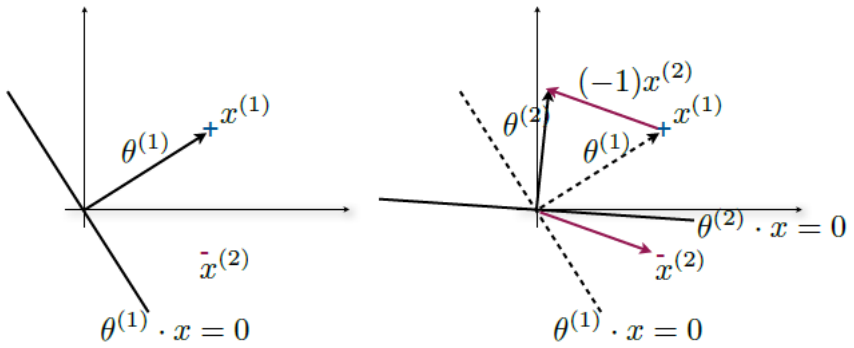


Figure 5.2 Linear classifier decision boundary

How do we measure performance of this classifier? One way is to count the number of mistakes it makes when compared to ground truth labels `y`. We can count the number of mistakes as follows:

$$\mathcal{E}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}[[y_i \neq h(x_i;\theta)]] = \frac{1}{n}\sum_{i=1}^{n}[[y_i(\theta \cdot x_i + \theta_0) \leq 0]]$$

Equation 5.2 Error rate of linear classifier

where `[[·]]` is an indicator function, which is equal to `1` when the expression inside is true and `0` otherwise. Notice, in the equation above, a mistake occurs whenever the label `yi ∈ [+1,-1]` disagrees with the prediction of the classifier `h(xi;θ) ∈ [+1,-1]`, i.e. their product is negative.

Another way to measure performance of a binary classifier is via a confusion matrix.

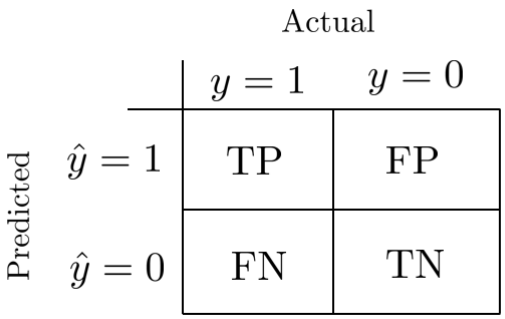


Figure 5.3 Confusion Matrix for a Binary Classifier

Figure 5.3 shows the table of errors called the **confusion matrix**. The prediction is correct when the predicted value matches the actual value as in the case of True Positive (TP) and True Negative (TN). Similarly, the prediction is wrong when there is a mismatch between predicted and actual values as in the case of False Positive (FP) and False Negative (FN). As we vary the classification threshold, we get different values for TP, FP, FN, and TN. In order to better visualize the performance of classifier under different classification thresholds, we can construct two additional figures: **Receiver Operating Characteristic (ROC)** and **Precision-Recall curve**.

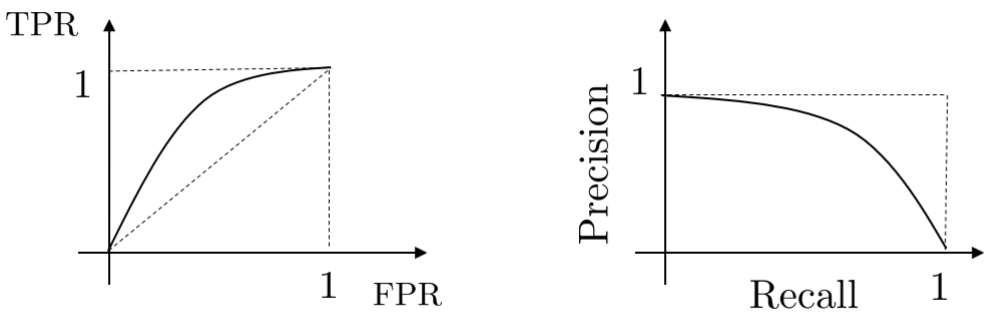


Figure 5.4 Receiver Operating Characteristic (ROC) plot (left) and Precision-Recall plot (right).

In the ROC plot on the left of Figure 5.4, TPR stands for True Positive Rate and can be computed as follows: `TPR=TP/(TP+FN)`. We can also compute the False Positive Rate (FPR) by using `FPR=FP/(FP+TN)`. By varying our classification threshold, we get different points along the ROC curve. A perfect classification results in `TPR=1` and `FPR=0`, in reality the closer

we are to upper left corner the better is the classifier. At chance level, we get the diagonal `TPR=FPR` line. The quality of ROC curve is often summarized by a single number using the area under the curve or AUC. Higher AUC scores are better with the maximum `AUC=1`.

In information retrieval, it is common to use Precision-Recall plot as shown on the right of Figure 5.4. The precision is defined as `Precision=TP/(TP+FP)` and the recall is defined as `Recall=TP/(TP+FN)`. A precision recall curve is a plot of precision vs recall as we vary the classification threshold. The curve can be summarized by a single number using the mean precision by averaging over the recall values, which approximates the area under the curve. Also, for a fixed threshold, we can summarize performance in a single statistic called the F1 score, which is the harmonic mean of precision and recall: `F1=2PR/(P+R)`.

Perceptron is a mistake driven algorithm: it starts with `θ=0` and successively adjusts the parameter `θ` for each training example until there are no more classification mistakes, assuming the data is linearly separable. The perceptron update rule can be summarized as follows:

$$\begin{aligned} \text{if} \quad y_i &\neq h(x_i;\theta^{(k)}) \quad \text{then} \\ \theta^{(k+1)} &= \theta^{(k)} + y_i x_i \\ \theta_0^{(k+1)} &= \theta_0^{(k)} + y_i \end{aligned}$$

Equation 5.3 Perceptron Update Rule

where index `k` denotes the number of times parameter updates, aka the number of mistakes. Think of θ_0 update as similar to `θ` update but with `x=1`. If the training examples are linearly separable then the above perceptron algorithm converges after a finite number of iterations. Notice, that the order of input data points makes a difference on how parameter `θ` is learned and therefore, we can randomize (shuffle) the training dataset. In addition, we can introduce a learning rate to help with `θ` convergence, the properties of which we'll discuss following the implementation. The perceptron algorithm can be summarized in the pseudo-code below:

1: **class** perceptron
2: **function** fit(X, y):
3: $k = 1$
4: **for** epoch=1,2,...,num_epochs
5: **for** i=1,2,...,N
6: **if** $y_i(\theta \cdot x_i + \theta_0) \leq 0$
7: $\eta = 1/(k+1)$
8: $k+ = 1$ — Update learning rate
9: $\theta = \theta + \eta\ y_i x_i$
10: $\theta_0 = \theta_0 + \eta\ y_i$ — Update theta
11: **end if**
12: **end for**
13: **end for**
14: **return** θ, θ_0
15: **function** predict(X):
16: $\hat{y} = \text{sign}(\theta \cdot X + \theta_0)$
17: **return** $\hat{y}$

The code consists of `Perceptron` class with two functions: `fit` and `predict`. In the `fit` function, we are taking the training data `X` and labels `y` and upon encountering an error (in which case the if statement condition is true), we update the learning rate and update theta as derived previously. Finally, in the `predict` function, we make a prediction for test data based on the sign of the decision boundary.

We now have all the tools to implement the perceptron algorithm from scratch! In the following code listing, we will be classifying Iris by training the perceptron algorithm on the training feature data and making a prediction based on the test data.

Listing 5.1 Perceptron Algorithm

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from scipy.stats import randint
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

class perceptron:
    def __init__(self, num_epochs, dim):
      self.num_epochs = num_epochs
      self.theta0 = 0
      self.theta = np.zeros(dim)

    def fit(self, X_train, y_train):
```

```
      n = X_train.shape[0]
      dim = X_train.shape[1]

      k = 1
      for epoch in range(self.num_epochs):
          for i in range(n):
              idx = randint.rvs(0, n-1, size=1)[0]    #A
              if (y_train[idx] * (np.dot(self.theta, X_train[idx,:]) + self.theta0) <= 0):
       #B
                  eta = pow(k+1, -1)    #C
                  k += 1

                  self.theta = self.theta + eta * y_train[idx] * X_train[idx, :]  #D
                  self.theta0 = self.theta0 + eta * y_train[idx]  #D
               #end if

          print("epoch: ", epoch)
          print("theta: ", self.theta)
          print("theta0: ", self.theta0)
          #end for
      #end for

    def predict(self, X_test):
      n = X_test.shape[0]
      dim = X_test.shape[1]

      y_pred = np.zeros(n)
      for idx in range(n):
          y_pred[idx] = np.sign(np.dot(self.theta, X_test[idx,:]) + self.theta0)
      #end for
      return y_pred

if __name__ == "__main__":

    iris = load_iris()    #E
    X = iris.data[:100,:]
    y = 2*iris.target[:100] - 1 #F

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
       random_state=42)

    #perceptron (binary) classifier
    clf = perceptron(num_epochs=5, dim=X.shape[1])
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    cmt = confusion_matrix(y_test, y_pred)
    acc = np.trace(cmt)/np.sum(np.sum(cmt))
    print("percepton accuracy: ", acc)

    #generate plots
    plt.figure()
    sns.heatmap(cmt, annot=True, fmt="d")
    plt.title("Confusion Matrix"); plt.xlabel("predicted"); plt.ylabel("actual")
    plt.savefig("./figures/perceptron_acc.png")
    plt.show()
```

#A sample random point
#B hinge loss

```
#C update learning rate
#D update theta and theta0
#E load dataset
#F map to {+1,-1} labels
```

After running the algorithm, we get the following classification accuracy results on test dataset.

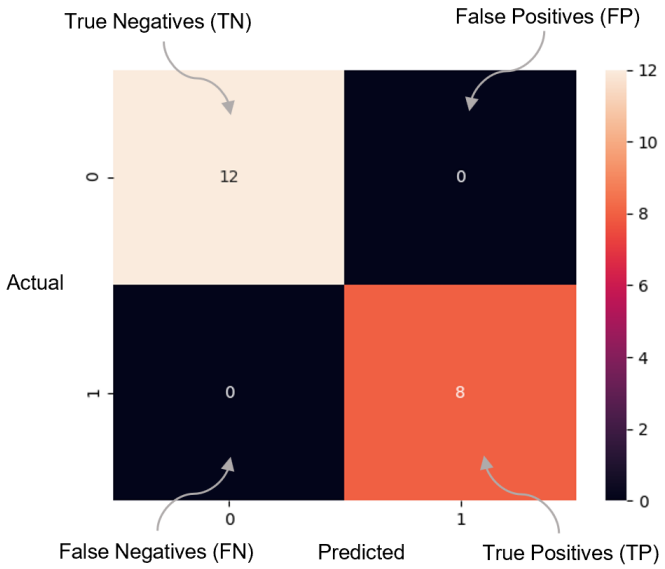


Figure 5.5 Perceptron Binary Classifier Confusion Matrix (Iris Dataset)

Let's take a second look at our implementation and understand it a little better. Perceptron algorithm can be formulated as stochastic gradient descent that minimizes a hinge loss function. Consider, a loss function that penalizes the *magnitude* of disagreement `zi=yi(θ·xi+θ0)` between the label `yi` and the prediction `h(xi;θ)`.

$$\text{Loss}_h(z) = \frac{1}{n}\sum_{i=1}^{n}\max\{1 - z_i, 0\} = \frac{1}{n}\sum_{i=1}^{n}\max\{1 - y_i(\theta \cdot x_i + \theta_0), 0\}$$

Equation 5.4 Hinge Loss Function

This is known as a hinge loss function as illustrated in Figure 5.6

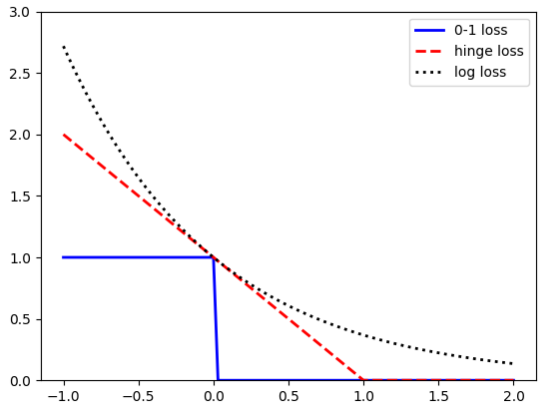


Figure 5.6 Hinge Loss, 0-1 Loss and log-loss functions

The stochastic gradient descent attempts to minimize the hinge loss by taking a gradient with respect to `θ`. However, the max operator is not differentiable at `zi=1`. In fact, we have several possible gradients at that point, collectively known as sub-differential. Since hinge loss is a piece-wise linear function, the gradient for `zi>1` is equal to `0`, while the gradient for `zi≤1` is equal to:

$$\begin{aligned}\nabla_\theta\big(1 - y_i(\theta \cdot x_i + \theta_0)\big) &= -y_i x_i \\ \nabla_{\theta_0}\big(1 - y_i(\theta \cdot x_i + \theta_0)\big) &= -y_i\end{aligned}$$

Equation 5.5 Gradient of Hinge Loss

Combining the expressions above with stochastic gradient descent update (where eta is the learning rate):

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta \text{Loss}_h(y_i(\theta \cdot x_i + \theta_0))$$

Equation 5.6 Perceptron Update Rule

We get the perceptron algorithm! In the next section, we'll talk about another important classification algorithm: Support Vector Machine (SVM).

5.3 SVM

In the previous section, we evaluated the performance of our classifier by minimizing expected loss function aka empirical risk. One problem with current formulation is there are multiple classifiers (multiple parameter values θ and θ_0) that can achieve the same empirical risk. So how do we choose the best model and what does best mean?

One solution is to regularize the loss function to favor small parameter values:

$$L_n(\theta, \theta_0) = \frac{\lambda}{2}||\theta||^2 + \frac{1}{n}\sum_{i=1}^{n} \text{Loss}\big(y_i(\theta \cdot x_i + \theta_0)\big)$$

Equation 5.7 Regularized Loss Function

where the regularization applies to θ and not θ_0. The reason is because θ specifies the orientation of the decision boundary, whereas θ_0 is related to its offset from the origin, which is unknown at the start.

Let's try to understand the decision boundary better from geometric point of view. It's desirable for the decision boundary to first of all classify all data points correctly and secondly, to be maximally removed from all the training examples, i.e. to have the maximum margin. Suppose, condition 1 is met and to optimize for condition 2, we need to compute and maximize the distance from every training example to the decision boundary. Geometrically, this distance is:

$$\gamma_i = \frac{y_i(\theta \cdot x_i + \theta_0)}{||\theta||}$$

Equation 5.8 Distance from training example to decision boundary

Since we want to maximize the margin, we would like to maximize the minimum distance to the decision boundary across all data points, i.e. to find $\max[\min_i \gamma_i]$. This can be formulated more simply as a **quadratic program** with linear constraints:

$$(\text{primal}) \quad \min \frac{1}{2}||\theta||^2 \text{ subject to } y_i(\theta \cdot x_i + \theta_0) \geq 1, \quad i = 1, ..., n$$

Equation 5.9 Quadratic Primal Optimization Program for SVM

Notice, how we are essentially minimizing the regularized loss function by choosing θ with small l_2 norm subject to the constraints that every training examples is correctly classified.

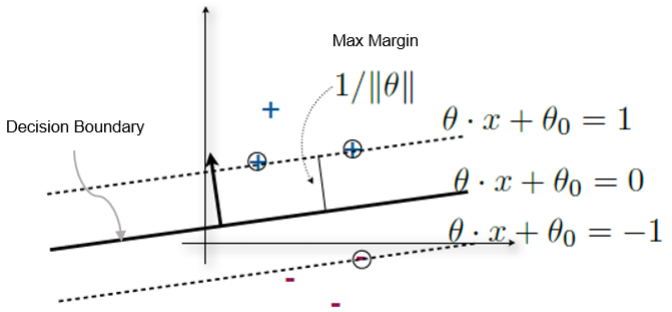


Figure 5.7 Max Margin Solution of SVM Classifier

Notice, as we minimize $||\theta||^2$, we are effectively increasing the distance $\gamma_i \propto 1/||\theta||$ between the decision boundary and the training data points indexed by i. Geometrically, we are pushing the margin boundaries away from each other as shown in Figure 5.7. At some point, they cannot be pushed further without violating classification constraints. At this point, the margin boundaries lock into unique maximum margin solution. The training data points that lie on the margin boundaries become **support vectors**. Notice that we only need a *subset* of training examples (support vectors) to fully learn SVM model parameters.

Let's see if we gain any advantages of solving the **dual** form of the quadratic program. We can obtain the dual form by writing out the Lagrangian (by adding constraints to the objective function with non-negative Lagrange multipliers):

$$\max_{\alpha \geq 0} L(\theta, \theta_0; \alpha) = \frac{1}{2}||\theta||^2 - \sum_{i=1}^{n} \alpha_i \left[y_i(\theta \cdot x_i + \theta_0) - 1 \right]$$

Equation 5.10 Lagrangian objective function for SVM

We can now compute the gradient with respect to our parameters:

$$\nabla_\theta L(\theta, \theta_0; \alpha) = \theta - \sum_{i=1}^{n} \alpha_i y_i x_i = 0$$

$$\frac{d}{d\theta_0} L(\theta, \theta_0; \alpha) = -\sum_{i=1}^{n} \alpha_i y_i = 0$$

Equation 5.11 Gradient of the Lagrangian

By substituting the expression for θ back into our Lagrangian, we get:

$$\text{(dual)} \quad \max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \left[x_i^T x_j \right] \quad \text{subject to} \quad \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0$$

Equation 5.12 Quadratic Dual Optimization Program for SVM

Notice, the big change in the dual formulation is that `d`-dimensional data points `xi` and `xj` interact via inner product. This has significant computational advantages over the primal formulation (in addition to simpler constraints in the dual).

The inner product measures the degree of similarity between two vectors and can be generalized via **kernels** `K(xi,xj)`. Kernels measure a degree of similarity between objects without explicitly representing them as feature vectors. This is particularly advantageous when we don't have access to or choose not to look into the internals of our objects. Typically, a kernel function that compares two objects `xi,xj` ∈ `X` is symmetric `K(xi,xj)=K(xj,xi)` and non-negative `K(xi,xj)≥0`. There is a wide variety of kernels, ranging from graph kernels to compute similarity between graphs to string kernels and document kernels. One popular kernel example that we will use in our SVM implementation is a radial basis function or RBF kernel:

$$K(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{2\sigma^2} \right)$$

Equation 5.13 Radial Basis Function Kernel

We are now ready to implement a binary SVM classifier from scratch using CVXOPT optimization package. CVXOPT is a free software package for convex optimization based on Python programming language and can be downloaded at cvxopt.org

The standard form of a quadratic program (QP) following CVXOPT notation is:

$$\min_x \frac{1}{2} x^T P x + q^T x \quad \text{subject to} \quad Gx \preceq h,\ Ax = b$$

Equation 5.14 Quadratic Program in CVXOPT notation

Note that the above objective function is convex if and only if matrix `P` is positive semi-definite. The CVXOPT QP expects the problem in the above form parameterized by `(P,q,G,h,A,b)`. Let us convert our dual QP into this form. Let `P` be a matrix such that

$$P_{ij} = y_i y_j [x_i^T x_j]$$

Equation 5.15 Definition of Matrix P

Then the optimization program becomes:

$$\max_\alpha \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \alpha^T P \alpha \quad \text{subject to} \quad \alpha_i \geq 0,\ \sum_{i=1}^{n} \alpha_i y_i = 0$$

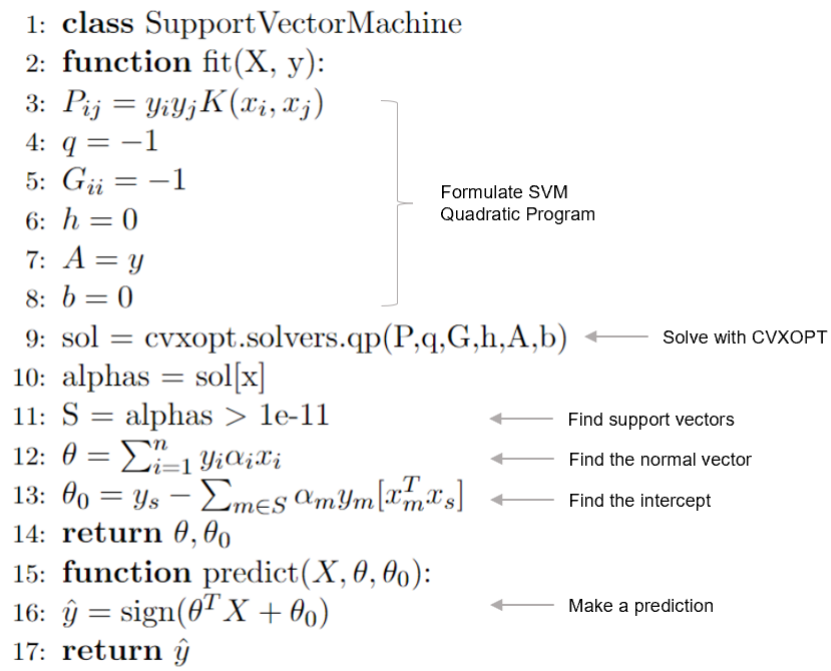
Equation 5.16 SVM quadratic program formulation

We can further modify the QP by multiplying the objective and the constraint by `-1` which turns this into minimization problem and reverses the inequality. In addition, we can convert the sum over alphas into a vector form by multiplying the alpha vector with an all ones vector.

$$\min_\alpha \frac{1}{2} \alpha^T P \alpha - 1^T \alpha \quad \text{subject to} \quad -\alpha_i \leq 0,\ y^T \alpha = 0$$

Equation 5.17 SVM quadratic program for CVXOPT

We can now use CVXOPT to solve our SVM quadratic program. Let's start by looking at the following pseudocode first.



The SVM class consists of two functions: `fit` and `predict`. In the `fit` function, we start off by formulating the quadratic problem to be solved by CVXOPT and defining all input parameters: `(P,q,G,h,A,b)`. After calling the solver, we find the support vectors as alphas greater than 0 (up to a rounding error). We compute the normal vector next and the intercept as discussed previously. In the `predict` function, we use the computed normal and intercept support vectors to make label prediction on test data.

Listing 5.2 SVM Algorithm

```
import cvxopt
import numpy as np

from sklearn.svm import SVC   #A
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

def rbf_kernel(gamma, **kwargs):
    def f(x1, x2):
     distance = np.linalg.norm(x1 - x2) ** 2
     return np.exp(-gamma * distance)
    return f

class SupportVectorMachine(object):
  def __init__(self, C=1, kernel=rbf_kernel, power=4, gamma=None):
      self.C = C    #B
```

```
    self.kernel = kernel  #C
    self.power = power    #C
    self.gamma = gamma    #C
    self.lagr_multipliers = None
    self.support_vectors = None
    self.support_vector_labels = None
    self.intercept = None

def fit(self, X, y):

    n_samples, n_features = np.shape(X)

    if not self.gamma:
        self.gamma = 1 / n_features

    self.kernel = self.kernel(    #D
        power=self.power,   #D
        gamma=self.gamma)   #D

    kernel_matrix = np.zeros((n_samples, n_samples))    #E
    for i in range(n_samples):    #E
        for j in range(n_samples):    #E
            kernel_matrix[i, j] = self.kernel(X[i], X[j])   #E

    P = cvxopt.matrix(np.outer(y, y) * kernel_matrix, tc='d')  #F
    q = cvxopt.matrix(np.ones(n_samples) * -1)   #F
    A = cvxopt.matrix(y, (1, n_samples), tc='d')  #F
    b = cvxopt.matrix(0, tc='d')    #F

    if not self.C: #if its empty
        G = cvxopt.matrix(np.identity(n_samples) * -1)
        h = cvxopt.matrix(np.zeros(n_samples))
    else:
        G_max = np.identity(n_samples) * -1
        G_min = np.identity(n_samples)
        G = cvxopt.matrix(np.vstack((G_max, G_min)))
        h_max = cvxopt.matrix(np.zeros(n_samples))
        h_min = cvxopt.matrix(np.ones(n_samples) * self.C)
        h = cvxopt.matrix(np.vstack((h_max, h_min)))

    minimization = cvxopt.solvers.qp(P, q, G, h, A, b)   #G

    lagr_mult = np.ravel(minimization['x'])    #H

    # Get indexes of non-zero lagr. multipiers   #I
    idx = lagr_mult > 1e-11   #I
    # Get the corresponding lagr. multipliers   #I
    self.lagr_multipliers = lagr_mult[idx]   #I
    # Get the samples that will act as support vectors   #I
    self.support_vectors = X[idx]   #I
    # Get the corresponding labels   #I
    self.support_vector_labels = y[idx]   #I

    self.intercept = self.support_vector_labels[0]   #J
    for i in range(len(self.lagr_multipliers)):    #J
      self.intercept -= self.lagr_multipliers[i] * self.support_vector_labels[
         i] * self.kernel(self.support_vectors[i], self.support_vectors[0])
```

```
  def predict(self, X):   #K
      y_pred = []
      for sample in X:
          prediction = 0
          # Determine the label of the sample by the support vectors
          for i in range(len(self.lagr_multipliers)):
              prediction += self.lagr_multipliers[i] * self.support_vector_labels[
                  i] * self.kernel(self.support_vectors[i], sample)
          prediction += self.intercept
          y_pred.append(np.sign(prediction))
      return np.array(y_pred)

def main():

  #load dataset
  iris = load_iris()
  X = iris.data[:100,:]
  y = 2*iris.target[:100] - 1 #L

  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
  clf = SupportVectorMachine(kernel=rbf_kernel, gamma = 1)
  clf.fit(X_train, y_train)
  y_pred = clf.predict(X_test)
  accuracy = accuracy_score(y_test, y_pred)
  print ("Accuracy (scratch):", accuracy)

  clf_sklearn = SVC(gamma = 'auto')
  clf_sklearn.fit(X_train, y_train)
  y_pred2 = clf_sklearn.predict(X_test)
  accuracy = accuracy_score(y_test, y_pred2)
  print ("Accuracy :", accuracy)

if __name__ == "__main__":
  main()
```

#A for comparison only
#B `regularization constant`
#C `kernel parameters`
#D initialize kernel method with parameters
#E calculate kernel matrix
#F define the quadratic optimization problem
#G solve the quadratic optimization problem using cvxopt
#H Lagrange multipliers
#I extract support vectors
#J calculate intercept with first support vector
#K iterate through list of samples and make predictions
#L map to {+1,-1} labels

We can see that SVM classification accuracy based on our implementation matches the accuracy of sklearn model!

5.4 Logistic Regression

Logistic regression is a classification algorithm. Let's dive into some of the theory behind logistic regression before implementing it from scratch! In a probabilistic view of classification, we are interested in computing `p(Ck|x)` the probability of class label `Ck` given

the input data `x`. Consider two classes `C1` and `C2`, we can use Bayes rule to compute our posterior probability:

$$p(C_1|x) = \frac{p(x|C_1)p(C_1)}{p(x|C_1)p(C_1) + p(x|C_2)p(C_2)}$$

Equation 5.18 Class posterior probability

where `p(Ck)` are prior class probabilities. We can divide the right hand side by the numerator and obtain:

$$p(C_1|x) = \frac{1}{1 + \frac{p(x|C_2)p(C_2)}{p(x|C_1)p(C_1)}} = \frac{1}{1 + \exp(-a)} = \sigma(a)$$

Equation 5.19 Class posterior probability

where we defined:

$$a = \ln \frac{p(x|C_1)p(C_1)}{p(x|C_2)p(C_2)}$$

Equation 5.20 Log-Likelihood Ratio

In the multi-class scenario `(K>2)`, we have:

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{\sum_i p(x|C_i)p(C_i)} = \frac{\exp(a_k)}{\sum_i \exp(a_i)}$$

Equation 5.21 Class posterior probability

Where `ak=ln p(x|Ck)p(Ck)`. The expression above is also known as a *softmax* function. Now, it's a matter of choosing class conditional densities that model the data well. In the case of binary logistic regression parameterized by `θ` and with class label `y=Ck`, we have

$$p(C_k|x) = p(y|x, \theta) = \text{Ber}(y|\sigma(\theta^T x))$$

Equation 5.22 Class posterior probability

We can compute the joint distribution as follows:

$$p(x_i, y_i|\theta) = p(y_i|x_i, \theta)p(x_i|\theta) = \text{Ber}(y|\sigma(\theta^T x))p(x_i|\theta)$$

Equation 5.23 Joint distribution

Since we are not modelling the distribution of data `p(xi|θ)=p(xi)`, we can write the log-likelihood as follows:

$$\begin{aligned}\log p(D|\theta) &= \log \prod_{i=1}^{n} p(x_i, y_i|\theta) = \sum_{i=1}^{n} \log p(x_i, y_i|\theta) \\ &= \sum_{i=1}^{n} \log \text{Ber}(y|\sigma(\theta^T x)) = \sum_{i=1}^{n} \log \left[\sigma(\theta^T x_i)^{y_i}(1-\sigma(\theta^T x_i))^{1-y_i}\right] \\ &= \sum_{i=1}^{n} \left[y_i \log \sigma(\theta^T x_i) + (1-y_i)\log(1-\sigma(\theta^T x_i))\right]\end{aligned}$$

Equation 5.24 Log-likelihood

Note that we are interested in maximizing the log-likelihood or equivalently minimizing the loss or the negative log-likelihood (NLL):

$$\min_{\theta} \text{Loss}(\theta) = \min_{\theta} \text{NLL}(\theta) = \max_{\theta} \log p(D|\theta)$$

Equation 5.25 Negative Log-likelihood Loss

We are planning on minimizing the logistic regression loss via Stochastic Gradient Descent (SGD) that can be written as follows:

$$\theta_{k+1} = \theta_k - \eta_k g_k$$

Equation 5.26 Gradient Descent

where `gk` is the gradient and `ηk` is the step size. To gaurantee convergence of SGD, the following conditions known as *Robbins-Monro* conditions on the learning rate must be satisfied:

$$\sum_{k=1}^{\infty} \eta_k = \infty$$
$$\sum_{k=1}^{\infty} \eta_k^2 < \infty$$

Equation 5.27 Robbins-Monro conditions

We can use the following learning rate schedule that satisfies the conditions above:

$$\eta_k = (\tau_0 + k)^{-\kappa}$$

Equation 5.28 Learning Rate Schedule

where $\tau 0 \geq 0$ slows down early iterations of the algorithm and $\kappa \in (0.5, 1]$ controls the rate at which old values are forgotten. To compute the steepest descent direction g_k, we need to differentiate our loss function $\text{NLL}(\theta)$:

$$\begin{aligned}
\frac{d}{d\theta} \log p(D|\theta) &= \sum_{i=1}^{n} \left[y_i \frac{d}{d\theta} \log \sigma(\theta^T x_i) + (1 - y_i) \frac{d}{d\theta} \log(1 - \sigma(\theta^T x_i)) \right] \\
&= \sum_{i=1}^{n} \left[y_i \frac{\sigma(\theta^T x_i)(1 - \sigma(\theta^T x_i))}{\sigma(\theta^T x_i)} x_i + (1 - y_i) \frac{\sigma(\theta^T x_i)(1 - \sigma(\theta^T x_i))}{1 - \sigma(\theta^T x_i)} (-x_i) \right] \\
&= \sum_{i=1}^{n} \left[y_i x_i (1 - \sigma(\theta^T x_i)) - (1 - y_i) x_i \sigma(\theta^T x_i) \right] \\
&= \sum_{i=1}^{n} \left[y_i - \sigma(\theta^T x_i) \right] x_i = \sum_{i=1}^{n} \left[y_i - \mu_i \right] x_i = -X^T (\mu - y)
\end{aligned}$$

Equation 5.29 Gradient computation

where we used the fact $d/dx \ \sigma(x) = (1 - \sigma(x))\sigma(x)$ and that the mean of the Bernoulli distribution $\mu i = \sigma(\theta \hat{} T \ xi)$. Note, that there exist a number of autograd libraries to avoid deriving the gradients by hand. Furthermore, we can add regularization to control parameter size. Our regularized objective and the gradient become:

$$\begin{aligned}
\min_{\theta} \text{Loss}(\theta) &= \min_{\theta} \left[\text{NLL}(\theta) + \lambda \theta^T \theta \right] \\
g_k &= X^T (\mu - y) + 2\lambda\theta
\end{aligned}$$

Equation 5.30 Regularized loss function and gradient

We are now ready to implement SGD for logistic regression. Let's start with the following pseudocode.

1: **class** sgdlr
2: **function** lr_objective(θ, X, y, λ)
3: $\mu_i = \text{sigmoid}(\theta^T X_i)$
4: $\text{cost} = -\sum_{i=1}^{n}[y_i \log \mu_i + (1 - y_i) \log(1 - \mu_i)] + \lambda\theta^T\theta$
5: $\text{grad} = X^T(\mu - y) + 2\lambda\theta$ ← Compute gradient
6: **return** cost, grad
7: **function** fit(X, y):
8: $\eta_i = (\tau + i)^{-\kappa}$ ← Set learning rate
9: **for** i = 1, 2, ... num_iter
10: cost, grad = lr_objective(θ, X, y, λ)
11: $\theta = \theta - \eta_i$ grad ← Update theta
12: **end for**
13: **return** θ
14: **function** predict(X, θ):
15: $\hat{y} = \text{sigmoid}(\theta^T X)$ ← Make a prediction
16: **return** $\hat{y}$

The `sgdlr` class consists of three main functions: `lr_objective`, `fit` and `predict`. In the `lr_objective` function, we compute the regularized objective function and the gradient of the objective as discussed in the text. In the `fit` function, we first set the learning rate, and for each iteration we update the theta parameters in the direction opposite to the gradient. Finally, in the `predict` function, we make a binary prediction of the label based on test data. In the following code listing, we'll be using a synthetic Gaussian mixture dataset to train the logistic regression model.

Listing 5.3 SGD Logistic Regression

```
import numpy as np
import matplotlib.pyplot as plt

def generate_data():

 n = 1000
 mu1 = np.array([1,1])
 mu2 = np.array([-1,-1])
 pik = np.array([0.4,0.6])

 X = np.zeros((n,2))
 y = np.zeros((n,1))

 for i in range(1,n):
     u = np.random.rand()
```

```
    idx = np.where(u < np.cumsum(pik))[0]

    if (len(idx)==1):
        X[i,:] = np.random.randn(1,2) + mu1
        y[i] = 1
    else:
        X[i,:] = np.random.randn(1,2) + mu2
        y[i] = -1
 return X, y

class sgdlr:

    def __init__(self):

     self.num_iter = 100
     self.lmbda = 1e-9

     self.tau0 = 10
     self.kappa = 1
     self.eta = np.zeros(self.num_iter)

     self.batch_size = 200
     self.eps = np.finfo(float).eps


 def fit(self, X, y):

     theta = np.random.randn(X.shape[1],1)   #A

     for i in range(self.num_iter):
         self.eta[i] = (self.tau0+i)**(-self.kappa)   #B

     batch_data, batch_labels = self.make_batches(X,y,self.batch_size)   #C
     num_batches = batch_data.shape[0]
     num_updates = 0

     J_hist = np.zeros((self.num_iter * num_batches,1))
     t_hist = np.zeros((self.num_iter * num_batches,1))

     for itr in range(self.num_iter):
         for b in range(num_batches):
             Xb = batch_data[b]
             yb = batch_labels[b]

             J_cost, J_grad = self.lr_objective(theta, Xb, yb, self.lmbda)
             theta = theta - self.eta[itr]*(num_batches*J_grad)

             J_hist[num_updates] = J_cost
             t_hist[num_updates] = np.linalg.norm(theta,2)
             num_updates = num_updates + 1
         print("iteration %d, cost: %f" %(itr, J_cost))

     y_pred = 2*(self.sigmoid(X.dot(theta)) > 0.5) - 1
     y_err = np.size(np.where(y_pred - y)[0])/float(y.shape[0])
     print("classification error:", y_err)

     self.generate_plots(X, J_hist, t_hist, theta)
     return theta
```

```
def make_batches(self, X, y, batch_size):
    n = X.shape[0]
    d = X.shape[1]
    num_batches = int(np.ceil(n/batch_size))

    groups = np.tile(range(num_batches),batch_size)
    batch_data=np.zeros((num_batches,batch_size,d))
    batch_labels=np.zeros((num_batches,batch_size,1))

    for i in range(num_batches):
        batch_data[i,:,:] = X[groups==i,:]
        batch_labels[i,:] = y[groups==i]

    return batch_data, batch_labels

def lr_objective(self, theta, X, y, lmbda):    #D

    n = y.shape[0]
    y01 = (y+1)/2.0

    mu = self.sigmoid(X.dot(theta))

    mu = np.maximum(mu,self.eps)     #E
    mu = np.minimum(mu,1-self.eps)   #E

    cost = -(1/n)*np.sum(y01*np.log(mu)+(1-y01)*np.log(1-mu))+np.sum(lmbda*theta*theta)
      #F

    grad = X.T.dot(mu-y01) + 2*lmbda*theta   #G

    #compute the Hessian of the lr objective
    #H = X.T.dot(np.diag(np.diag( mu*(1-mu) ))).dot(X) + 2*lmbda*np.eye(np.size(theta))

    return cost, grad

def sigmoid(self, a):
    return 1/(1+np.exp(-a))

def generate_plots(self, X, J_hist, t_hist, theta):

    plt.figure()
    plt.plot(J_hist)
    plt.title("logistic regression")
    plt.xlabel('iterations')
    plt.ylabel('cost')
    #plt.savefig('./figures/lrsgd_loss.png')
    plt.show()

    plt.figure()
    plt.plot(t_hist)
    plt.title("LR theta l2 norm")
    plt.xlabel('iterations')
    plt.ylabel('theta l2 norm')
    #plt.savefig('./figures/lrsgd_theta_norm.png')
    plt.show()

    plt.figure()
    plt.plot(self.eta)
    plt.title("LR learning rate")
```

```
    plt.xlabel('iterations')
    plt.ylabel('learning rate')
    #plt.savefig('./figures/lrsgd_learning_rate.png')
    plt.show()

    plt.figure()
    x1 = np.linspace(np.min(X[:,0])-1,np.max(X[:,0])+1,10)
    plt.scatter(X[:,0], X[:,1])
    plt.plot(x1, -(theta[0]/theta[1])*x1)
    plt.title('LR decision boundary')
    plt.grid(True)
    plt.xlabel('X1')
    plt.ylabel('X2')
    #plt.savefig('./figures/lrsgd_clf.png')
    plt.show()

if __name__ == "__main__":

 X, y = generate_data()
 sgd = sgdlr()
 theta = sgd.fit(X,y)
```

#A random init
#B learning rate schedule
#C divide data into batches
#D compute the objective
#E bound away from zero and one
#F compute cost
#G compute the gradient of the lr objective

Figure 5.8 shows the stochastic nature of loss function that decreases with the number of iterations as well as the decision boundary learned by our binary logistic regression.

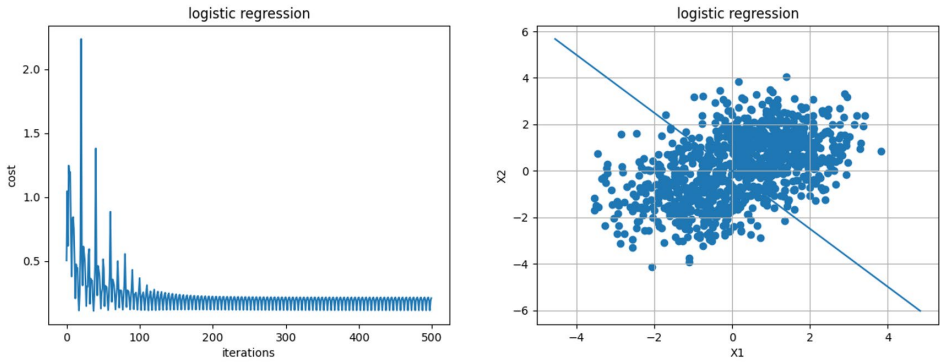


Figure 5.8 SGD Logistic Regression: cost (left) and decision boundary (right)

A natural extension to the binary logistic regression is a multinomial logistic regression that handles more than `2` classes.

5.5 Naïve Bayes

This section focuses on understanding, deriving, and implementing the Naive Bayes algorithm. The fundamental assumption of the algorithm is that the features are conditionally independent given the class label. This allows us to write the class conditional density as a product of one-dimensional densities.

$$p(x_i|y=c,\theta) = \prod_{j=1}^{D} p(x_{ij}|y=c,\theta_{jc})$$

Equation 5.31 Naïve Bayes Class-Conditional Density

The model is called naive because we don't expect the features to be conditionally independent. However, even if the assumption is false, the model performs well in many scenarios. Here we will focus on Bernoulli Naive Bayes for document classification with graphical model shown in Figure 5.9. Note the shaded nodes represent the observed variables.

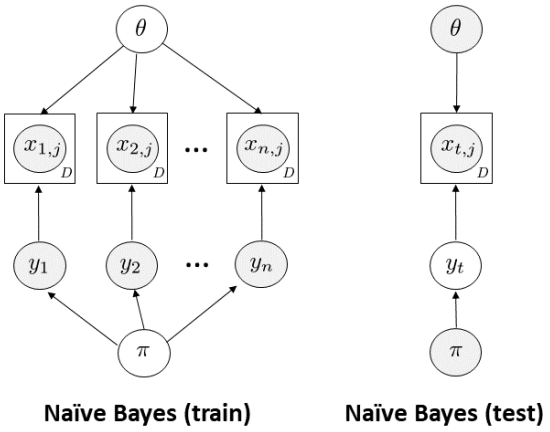


Figure 5.9 Naïve Bayes Probabilistic Graphical Model (PGM)

The choice of class conditional density `p(x|y=c,θ)` determines the type of Naive Bayes classifier such as Gaussian, Bernoulli or Multinomial. In the section, we focus on Bernoulli Naive Bayes due to its high performance in classifying documents.

Let x_{ij} be Bernoulli random variables indicating the presence $(x_{ij}=1)$ or absence $(x_{ij}=0)$ of a word $j \in \{1,\ldots,D\}$ for document $i \in \{1,\ldots,n\}$, parameterized by θ_{jc} for a given class label $y=c \in \{1,\ldots,C\}$. In addition, let π be a Dirichlet distribution representing the prior over the class labels. Thus, the total number of learnable parameters is $|\theta|+|\pi|=O(DC)+O(C)=O(DC)$, where D is the dictionary size and C is the number of classes. Due to the small number of parameters, the Naive Bayes model is immune to over-fitting.

We can write-down the class conditional density as follows:

$$p(x|y=c,\theta)=\prod_{i=1}^{n}\prod_{j=1}^{D}p(x_{ij}|y=c,\theta_{jc})=\prod_{i=1}^{n}\prod_{j=1}^{D}\text{Bernoulli}(\theta_{jc})$$

Equation 5.32 Naïve Bayes Class-Conditional Density

We can derive the Naive Bayes inference algorithm by maximizing the log-likelihood. Consider words x_i in a single document i:

$$p(x_i,y_i|\theta)=p(y_i|\pi)\prod_{j=1}^{D}p(x_{ij}|y_i,\theta)=\prod_{c=1}^{C}\pi_c^{1[y_i=c]}\prod_{j=1}^{D}\prod_{c=1}^{C}p(x_{ij}|\theta_{jc})^{1[y_i=c]}$$

Equation 5.33 Naïve Bayes Likelihood for a Single Sample

Using the Naive Bayes assumption, we can compute the log-likelihood objective:

$$\log p(D|\theta)=\log\prod_{i=1}^{n}p(x_i,y_i|\theta)=\sum_{i=1}^{n}\log p(x_i,y_i|\theta)=\sum_{c=1}^{C}N_c\log\pi_c+\sum_{j=1}^{D}\sum_{c=1}^{C}\sum_{i:y_i=c}\log p(x_{ij}|\theta_{jc})$$

Equation 5.34 Naïve Bayes Log-Likelihood for Training Dataset

Note this is a constrained optimization program since the probabilities of class labels must sum to one: $\sum_c \pi_c=1$. We can solve the above optimization problem using a Lagrangian by including the constraint into the objective function and setting the gradient of the Lagr angian $L(\theta,\lambda)$ wrt to model parameters to zero:

$$L(\theta,\lambda)=\log p(D|\theta)+\lambda\big(1-\sum_c\pi_c\big)$$

Equation 5.35 Naïve Bayes Lagrangian

Differentiating wrt π_c, we get:

$$\frac{d}{d\pi_c} L(\theta, \lambda) = \frac{d}{d\pi_c} \log p(D|\theta) - \lambda = N_c \frac{1}{\pi_c} - \lambda = 0$$

Equation 5.36 Derivative of Lagrangian wrt pi

Which gives us an expression for `πc` in terms of `λ`: `πc =(1/λ) Nc`. To solve for `λ`, we use our sum to one constraint:

$$\sum_c \pi_c = 1 \rightarrow \sum_c \frac{1}{\lambda} N_c = 1 \rightarrow \lambda = \sum_c N_c$$

Equation 5.37 Solving for lambda

Substituting `λ` back into expression for `πc`, we get: `πc=Nc/∑Nc=Nc/Ntot`. Similarly, we can compute the optimum `θjc` parameters by setting the gradient of objective wrt `θjc` to zero:

$$\begin{aligned} \frac{d}{d\theta_{jc}} \log p(D|\theta) &= \frac{d}{d\theta_{jc}} \sum_{i:y_i=c} \left[x_{ij} \log(\theta_{jc}) + (1 - x_{ij}) \log(1 - \theta_{jc}) \right] = 0 \\ &= \sum_{i:y_i=c} \left[\frac{x_{ij}}{\theta_{jc}} - \frac{1 - x_{ij}}{1 - \theta_{jc}} \right] = 0 \\ \rightarrow \quad \frac{N_{jc}}{\theta_{jc}} &= \frac{1}{1 - \theta_{jc}} \left[N_c - N_{jc} \right] \rightarrow N_{jc} = N_c \theta_{jc} \end{aligned}$$

Equation 5.38 Derivative of Lagrangian wrt theta

As a result, the optimum Maximum Likelihood Estimate (MLE) value of `θjc=Njc/Nc`, where `Nc=∑1[y_i=c]` Note, that it's straight forward to add a Beta conjugate prior for the Bernoulli random variables and a Dirichlet conjugate prior for the class density to smooth the MLE counts:

$$\begin{aligned} p(\pi|D) &= \text{Dir}(N_1 + \alpha_1, ..., N_c + \alpha_c) \\ p(\theta_{jc}|D) &= \text{Beta}([N_c - N_{jc}] + \beta_0, N_{jc} + \beta_1) \end{aligned}$$

Equation 5.39 Conjugate priors for pi and theta

where we use *conjugate prior* for computational convenience since the posterior and the prior have the same form which enables closed form updates.

During test time, we would like to predict the class label `y` given the training data `D` and the learned model parameters. Applying the Bayes rule:

$$
\begin{aligned}
p(y=c|x_{i,1},...,x_{i,D},D) &\propto p(y=c|D)p(x_{i,1},...,x_{i,D}|y=c,D) \\
&= p(y=c|D)\prod_{j=1}^{D} p(x_{ij}|y=c,D)
\end{aligned}
$$

Equation 5.40 Conjugate priors for pi and theta

Substituting the distributions for `p(y=c|D)` and `p(xij|y=c,D)` and taking the log, we get:

$$
\log p(y=c|x,D) \propto \log \hat{\pi}_c + \sum_{j=1}^{D}\left(1[x_{ij}=1]\log\hat{\theta}_{jc} + 1[x_{ij}=0]\log(1-\hat{\theta}_{jc})\right)
$$

Equation 5.41 log class conditional density

Where `πc` and `θjc` are the MLE estimates obtained during training. The Naive Bayes algorithm is summarized in pseudo-code below.

Training:
$N_c = 0,\ N_{jc} = 0$
for $i = 1, 2, ..., n$ **do**
    $c = y_i$ //class label for ith example
    $N_c = N_c + 1$
    for $j = 1, ..., D$ **do**
        if $x_{ij} = 1$ then
            $N_{jc} = N_{jc} + 1$
    end for
end for
$\hat{\pi}_c = \frac{N_c}{N},\ \hat{\theta}_{jc} = \frac{N_{jc}}{N}$
return $\hat{\pi}_c,\ \hat{\theta}_{jc}$
Testing (for a single test document):
for $c = 1, 2, ..., C$ **do**
    $\log p[c] = \log \pi_c$
    for $j = 1, 2, ..., D$ **do**
        if $x_j = 1$ then
            $\log p[c] += \log \hat{\theta}_{jc}$
        else
            $\log p[c] += \log(1 - \hat{\theta}_{jc})$
    end for
end for
$c = \arg\max_c \log p[c]$
return c

Figure 5.10 Naïve Bayes Algorithm

The run-time complexity of MLE inference during training is `O(ND)` where `N` is the number of training documents and `D` is the dictionary size. The run-time complexity during test time is `O(TCD)` where `T` is the number of test documents, `C` is the number of classes and `D` is the dictionary size. Similarly, space complexity is the size of arrays required to store model parameters that grows as `O(DC)`.

We are now ready to implement Bernoulli Naive Bayes algorithm!

Listing 5.4 Bernoulli Naïve Bayes Algorithm

```
import numpy as np
import seaborn as sns
```

```
import matplotlib.pyplot as plt

from time import time
from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer

from sklearn.metrics import accuracy_score
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer

sns.set_style("whitegrid")
tokenizer = RegexpTokenizer(r'\w+')
stop_words = set(stopwords.words('english'))
stop_words.update(['s','t','m','1','2'])

class naive_bayes:
    def __init__(self, K, D):
      self.K = K   #A
      self.D = D   #B

      self.pi = np.ones(K)  #C
      self.theta = np.ones((self.D, self.K))  #D

  def fit(self, X_train, y_train):

      num_docs = X_train.shape[0]
      for doc in range(num_docs):

          label = y_train[doc]
          self.pi[label] += 1

          for word in range(self.D):
              if (X_train[doc][word] > 0):
                  self.theta[word][label] += 1
              #end if
          #end for
      #end for

      #normalize pi and theta
      self.pi = self.pi/np.sum(self.pi)
      self.theta = self.theta/np.sum(self.theta, axis=0)

  def predict(self, X_test):

      num_docs = X_test.shape[0]
      logp = np.zeros((num_docs,self.K))
      for doc in range(num_docs):
          for kk in range(self.K):
              logp[doc][kk] = np.log(self.pi[kk])
              for word in range(self.D):
                  if (X_test[doc][word] > 0):
                      logp[doc][kk] += np.log(self.theta[word][kk])
                  else:
                      logp[doc][kk] += np.log(1-self.theta[word][kk])
                  #end if
              #end for
          #end for
      #end for
```

```
      return np.argmax(logp, axis=1)

if __name__ == "__main__":

  import nltk
  nltk.download('stopwords')

  #load data
  print("loading 20 newsgroups dataset...")
  tic = time()
  classes = ['sci.space', 'comp.graphics', 'rec.autos', 'rec.sport.hockey']
  dataset = fetch_20newsgroups(shuffle=True, random_state=0,
       remove=('headers','footers','quotes'), categories=classes)
  X_train, X_test, y_train, y_test = train_test_split(dataset.data, dataset.target,
       test_size=0.5, random_state=0)
  toc = time()
  print("elapsed time: %.4f sec" %(toc - tic))
  print("number of training docs: ", len(X_train))
  print("number of test docs: ", len(X_test))

  print("vectorizing input data...")
  cnt_vec = CountVectorizer(tokenizer=tokenizer.tokenize, analyzer='word',
       ngram_range=(1,1), max_df=0.8, min_df=2, max_features=1000, stop_words=stop_words)
  cnt_vec.fit(X_train)
  toc = time()
  print("elapsed time: %.2f sec" %(toc - tic))
  vocab = cnt_vec.vocabulary_
  idx2word = {val: key for (key, val) in vocab.items()}
  print("vocab size: ", len(vocab))

  X_train_vec = cnt_vec.transform(X_train).toarray()
  X_test_vec = cnt_vec.transform(X_test).toarray()

  print("naive bayes model MLE inference...")
  K = len(set(y_train)) #number of classes
  D = len(vocab) #dictionary size
  nb_clf = naive_bayes(K, D)
  nb_clf.fit(X_train_vec, y_train)

  print("naive bayes prediction...")
  y_pred = nb_clf.predict(X_test_vec)
  nb_clf_acc = accuracy_score(y_test, y_pred)
  print("test set accuracy: ", nb_clf_acc)
```

#A number of classes
#B dictionary size
#C class priors
#D Bernoulli parameters

As we can see from the output, we achieve 82% accuracy on the 20 newsgroups test dataset.

5.6 Decision Tree (CART)

This section focuses on Classification and Regression Trees (CART) algorithm. Tree based algorithms partition the input space into axis parallel regions such that each leaf represents a region. They can then be used to either classify the region by taking a majority vote or

regress the region by computing the expected value. Tree based models are interpretable and provide insight into feature importance. They are based on a greedy, recursive algorithm since optimum partitioning of space is NP complete.

In tree-based models during training we are interested in constructing a binary tree in a way that optimizes an objective function and does not lead to under or over fitting. A key determinant in growing a decision tree is the choice of the feature and the threshold to use when classifying the data points. Consider an input data matrix `Xn×d` with `n` data points of dimension (feature size) `d`. We would like to find the optimum feature and threshold for that feature that results in the split of data with minimum cost. Let `j∈ {1,...,d}` represent feature dimension and `t ∈ τj` represent a threshold for feature `j` out of all possible thresholds `τj` (constructed by taking mid-points of our data `xij`), then we would like to compute:

$$j^*, t^* = \arg \min_{j \in \{1,\ldots,d\}} \min_{t \in \tau_j} \text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_{ij} > t\})$$

Equation 5.42 Objective Function

Before we look at an example, let's look at potential costs we can use for optimizing the tree for classification. Our goal in defining a cost function is to evaluate how good our data partition is. We would like the leaf nodes to be pure, i.e. contain data from the same class and still be able to generalize to test data, i.e. we would like to limit the depth of the tree (to prevent overfitting) while minimizing impurity. One notion of impurity is the Gini index:

$$\sum_{k=1}^{K} \pi_k(1 - \pi_k) = \sum_k \pi_k - \sum_k \pi_k^2 = 1 - \sum_k \pi_k^2$$

Equation 5.43 Gini Index

where `πk` is a fraction of points in the region that belongs to cluster `k`:

$$\pi_k = \frac{1}{|D|} \sum_{i \in D} 1[y_i = k]$$

Equation 5.44 Definition of pi_k

Notice that since `πk` is the probability of a random point in the leaf belonging to class `k` and `1-πk` is the error rate, the Gini index is the expected error rate. If the leaf cluster is pure `(πk=1)` then the Gini index is zero. Thus, we are interested in minimizing the Gini index.

An alternative objective is the entropy:

$$H(\pi) = -\sum_{k=1}^{K} \pi_k \log \pi_k$$

Equation 5.45 Entropy Definition

Entropy measures the amount of uncertainty. If we are certain that the leaf cluster is pure (i.e. `πk=1`) then the entropy is zero. Thus, we are interested in minimizing the entropy when it comes to CART.

Let's look at a 1-D example of choosing the optimum splitting feature and its threshold. Let `X=[1.5,1.7,2.3,2.7,2.7]` and class label `y=[1,1,2,2,3]`. Since the data is one dimensional, our task is to find a threshold that will split `X` in a way that minimizes the Gini index. If we choose a threshold `t1=2` as a midpoint between `1.7` and `2.3` and compute the resulting Gini index we get:

$$G = \frac{2}{5} G_{left} + \frac{3}{5} G_{right} = \frac{2}{5} \times 0 + \frac{3}{5} \times \left(1 - \frac{2}{3}^2 - \frac{1}{3}^2\right) = 0.27$$

Equation 5.46 Gini Index Example

where `Gleft` is the Gini index of `{xi,yi: xij≤2}` and is equal to zero since both class labels are equal to `1`, i.e. a pure leaf cluster; and `Gright` is the Gini index of `{xi,yi: xij>2}` and contains a mix of class labels `yright = [2,2,3]`.

The key to CART algorithm is finding the optimal feature and threshold such that the cost (such as Gini index) is minimized. During training, we'll need to iterate through every feature one by one and compute the Gini cost for all possible thresholds for that feature. But how do we compute `τj` a set of all possible threshold for feature `j`? We can sort the training data `X[:,j]` in `O(nlogn)` time and consider all mid-points between two adjacent data values. Next, we'll need to compute the Gini index for each threshold that can be done as follows. Let `m` be the size of the node and `mk` be the number of points in the node that belong to class `k`, then

$$G = 1 - \sum_{k=1}^{K} \pi_k^2 = 1 - \sum_{k=1}^{k} \left(\frac{m_k}{m}\right)^2$$

Equation 5.47 Gini Index Computation

We can iterate through the sorted thresholds `τj` in `O(n)` time and in each iteration compute the Gini index that would result in applying that threshold. For `i`-th threshold, we get

$$
\begin{aligned}
G_i &= \frac{i}{m} G_i^{left} + \frac{m-i}{m} G_i^{right} \\
G_i^{left} &= 1 - \sum_k \left(\frac{m_k^{left}}{i}\right)^2 \\
G_i^{right} &= 1 - \sum_k \left(\frac{m_k^{right}}{m-i}\right)^2
\end{aligned}
$$

Equation 5.48 Gini Index Computation for the i-th threshold

Having found the optimum feature and threshold, we split each node recursively until the maximum depth is reached. Once we've constructed a tree during training, given test data, we simply traverse the tree from root to leaf which stores our class label. We can summarize CART algorithm in the following pseudo-code:

```
class TreeNode(gini, num_samples, num_samples_class, class_label):
self.gini = gini //gini cost
self.num_samples = num_samples //size of node
self.num_samples_class = num_samples_class //number of pts with label k
self.class_label = class_label //predicted class label
self.feature_idx = 0 //idx of feature to split on
self.threshold = 0 //best threshold to split on
self.left = None //left subtree pointer
self.right = None //right subtree pointer
function grow_tree(X_train, y_train, depth)
class_label = majority_vote(y_train)
gini = compute_gini(y_train)
node = new TreeNode(gini, class_label)
//split recursively until max depth is reached
if depth < max_depth:
    idx, threshold = best_split(X_train, y_train)
    if idx is not None:
        indices_left = X_train[:,idx] < threshold
        node.feature_index = idx
        node.threshold = threshold
        node.left = grow_tree(X_left, y_left, depth+1)
        node.right = grow_tree(X_right, y_right, depth+1)
return node
```

As we can see from the definition of `TreeNode`, it stores the predicted class label, id of feature to split on and the best threshold to split on, pointers to left and right subtrees as

well as the gini cost and the size of node. We can grow the decision tree recursively by calling the `grow_tree` function as long as the depth of the tree is less than the maximum depth determined ahead of time. First, we compute the class label via majority vote and the gini index for training labels. Next, we determine the best split by iterating over all features and over all possible splitting thresholds. Once, we determined the best feature idx and feature threshold to split on, we initialize left and right pointers of the current node with new `TreeNode` objects that contain data less the splitting threshold and greater than the splitting threshold, respectively. And we iterate in this fashion until we reach the maximum tree depth. We are now ready to implement the CART algorithm.

Listing 5.5 CART Decision Tree Algorithm

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

class TreeNode():
    def __init__(self, gini, num_samples, num_samples_class, class_label):
     self.gini = gini  #A
     self.num_samples = num_samples #B
     self.num_samples_class = num_samples_class #C
     self.class_label = class_label #D
     self.feature_idx = 0 #E
     self.treshold = 0  #F
     self.left = None #G
     self.right = None #H

class DecisionTreeClassifier():
 def __init__(self, max_depth = None):
     self.max_depth = max_depth

 def best_split(self, X_train, y_train):
     m = y_train.size
     if (m <= 1):
         return None, None

     mk = [np.sum(y_train == k) for k in range(self.num_classes)]   #I

     best_gini = 1.0 - sum((n / m) ** 2 for n in mk)  #J
     best_idx, best_thr = None, None

     #iterate over all features
     for idx in range(self.num_features):

         thresholds, classes = zip(*sorted(zip(X[:, idx], y)))   #K

         num_left = [0]*self.num_classes
         num_right = mk.copy()

         for i in range(1, m):   #L

             k = classes[i-1]
```

```
            num_left[k] += 1
            num_right[k] -= 1

            gini_left = 1.0 - sum(
                (num_left[x] / i) ** 2 for x in range(self.num_classes)
            )

            gini_right = 1.0 - sum(
                (num_right[x] / (m - i)) ** 2 for x in range(self.num_classes)
            )

            gini = (i * gini_left + (m - i) * gini_right) / m

            if thresholds[i] == thresholds[i - 1]:
                continue

            if (gini < best_gini):
                best_gini = gini
                best_idx = idx
                best_thr = (thresholds[i] + thresholds[i - 1]) / 2  #M
            #end if
        #end for
    #end for
    return best_idx, best_thr

def gini(self, y_train):
    m = y_train.size
    return 1.0 - sum((np.sum(y_train == k) / m) ** 2 for k in range(self.num_classes))

def fit(self, X_train, y_train):
    self.num_classes = len(set(y_train))
    self.num_features = X_train.shape[1]
    self.tree = self.grow_tree(X_train, y_train)

def grow_tree(self, X_train, y_train, depth=0):

    num_samples_class = [np.sum(y_train == k) for k in range(self.num_classes)]
    class_label = np.argmax(num_samples_class)

    node = TreeNode(
        gini=self.gini(y_train),
        num_samples=y_train.size,
        num_samples_class=num_samples_class,
        class_label=class_label,
    )

    if depth < self.max_depth:   #N
        idx, thr = self.best_split(X_train, y_train)
        if idx is not None:
            indices_left = X_train[:, idx] < thr
            X_left, y_left = X_train[indices_left], y_train[indices_left]
            X_right, y_right = X_train[~indices_left], y_train[~indices_left]
            node.feature_index = idx
            node.threshold = thr
            node.left = self.grow_tree(X_left, y_left, depth + 1)
            node.right = self.grow_tree(X_right, y_right, depth + 1)

    return node
```

```
 def predict(self, X_test):
     return [self.predict_helper(x_test) for x_test in X_test]

 def predict_helper(self, x_test):
     node = self.tree
     while node.left:
         if x_test[node.feature_index] < node.threshold:
             node = node.left
         else:
             node = node.right
     return node.class_label


if __name__ == "__main__":

 #load data
 iris = load_iris()
 X = iris.data[:, [2,3]]
 y = iris.target

 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

 print("decision tree classifier...")
 tree_clf = DecisionTreeClassifier(max_depth = 3)
 tree_clf.fit(X_train, y_train)

 print("prediction...")
 y_pred = tree_clf.predict(X_test)

 tree_clf_acc = accuracy_score(y_test, y_pred)
 print("test set accuracy: ", tree_clf_acc)
```

#A gini cost
#B size of node
#C number of node points with label k
#D predicted class label
#E idx of feature to slit on
#F best threshold to split on
#G left subtree pointer
#H right subtree pointer
#I number of points of class k
#J gini of current node
#K sort data along selected feature
#L iterate over all possible split positions
#M midpoint
#N split recursively until maximum depth is reached

As we can see from the output, we achieve test classification accuracy of 80% on the Iris dataset.

5.7 Exercises

5.1 Given a data point $y \in R^d$ and a hyper-plane $\theta \cdot x + \theta_0 = 0$, compute Euclidean distance from the point to the hyper-plane.

5.2 Given a primal linear program (LP) `min c^T x` subject to `Ax <= b, x >= 0`, write down the dual version of the LP.

5.3 Show that Radial Basis Function (RBF) kernel is equivalent to computing similarity between two infinite dimensional feature vectors.

5.4 Verify that the learning rate schedule `ηk=(τ0+k)^(-κ)` satisfies Robbins-Monro conditions.

5.5 Compute the derivative of the sigmoid function `σ(a) = [1+exp(-a)]^(-1)`

5.6 Compute run-time and memory complexity of Bernoulli Naive Bayes Algorithm.

5.8 Summary

- The goal of a classification algorithm is to learn a mapping from inputs x to outputs y, where y is a discrete quantity
- Perceptron is a classification algorithm that updates the decision boundary until there are no more classification mistakes
- SVM is max-margin classifier. The training data points that lie on the margin boundaries become support vectors.
- Logistic regression is a classification algorithm that computes class conditional density based on softmax function.
- Naive Bayes algorithm assumes that features are conditionally independent given the class label. It's commonly used in document classification.
- CART decision tree is greedy, recursive algorithm that finds the optimum feature splits by minimizing an objective function such as the Gini index.

6

Regression Algorithms

This chapter covers

- **Introduction to Regression**
- **Bayesian Linear Regression**
- **Hierarchical Bayesian Regression**
- **KNN Regression**
- **Gaussian Process Regression**

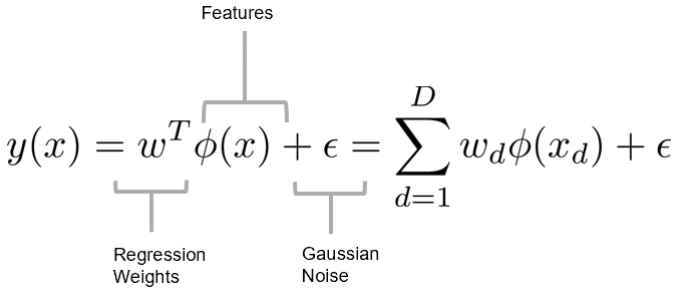
In the previous chapter, we looked at supervised algorithms for classification. In this chapter, we focus on supervised learning in which we are trying to predict a continuous quantity. We are going to study four intriguing regression algorithms: Bayesian Linear Regression, Hierarchical Bayesian Regression, KNN regresion and Gaussian Process Regression. We will derive them from first principles. Regression algorithms that predict a continuous quantity find many uses in a variety of applications. For example, predicting the price of financial assets or predicting CO2 levels in the atmosphere. Let's begin by reviewing the fundamentals of regression.

6.1 Introduction to Regression

In supervised learning, we are given a dataset $D=\{(x_1,y_1),\ldots,(x_n,y_n)\}$ consisting of tuples of data x and labels y. The goal of a regression algorithm is to learn a mapping from inputs x to outputs y, where y is a continuous quantity, i.e. $y \in R$.

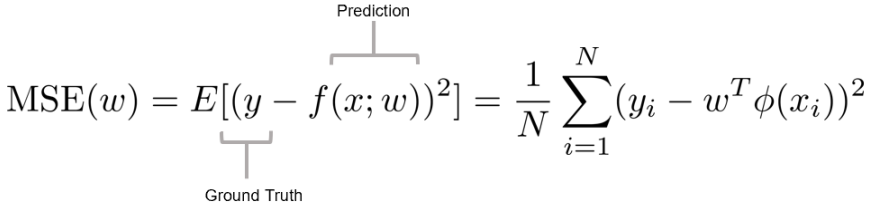
A regressor f can be viewed as a mapping between a d-dimensional feature vector $\phi(x)$ and a label y, i.e. $f: R^d \rightarrow R$. Regression problems are typically harder (to achieve higher accuracy) compared to classification problems because we are trying to predict a *continuous* quantity. Moreover, we are often interested in predicting *future* response variable *y* based on past training data.

One of most widely used models for regression is **linear regression**, which models the response variable `y` as a linear combination of input feature vectors `ϕ(x)`:



Equation 6.1 Formulation of Linear Regression

where `ϵ` is the residual error between our linear predictions and the true response. We can characterize the quality of our regressor based on the Mean Squared Error (MSE):



Equation 6.2 Mean Squared Error Definition

In this section, we are going to look at several important regression models starting with KNN and Bayesian regression, and their extensions to hierarchical regression models and conclude with Gaussian Process (GP) regression. We'll focus on both the theory and implementation of each model from scratch.

6.2 Bayesian Linear Regression

Recall, that we can write the linear regression as `y(x) = w^T x +ϵ`. If we assume that `ϵ~N(0,σ^2)` is a zero-mean Gaussian RV with variance `σ^2`, then we can formulate linear regression as follows:

$$p(y|x,\theta) = N(y|w^T x, \sigma^2)$$

Equation 6.3 Bayesian Linear Regression Formulation

where w are regression coefficients. To fit a linear regression model to data, we minimize the negative log likelihood:

$$\begin{aligned}
\mathrm{NLL}(w, \sigma^2) &= -\log p(y|x, \theta) \quad \longleftarrow \text{Definition of NLL} \\
&= -\log \prod_{n=1}^{N} \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2\sigma^2}(y_n - w^T x)^2 \right] \right] \\
&= -\sum_{n=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) + \sum_{n=1}^{N} \frac{1}{2\sigma^2}(y_n - w^T x)^2 \\
&= \frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - w^T x)^2 + \frac{N}{2} \log(2\pi\sigma^2)
\end{aligned}$$

Equation 6.4 Negative Log Likelihood Derivation

Keeping σ^2 fixed and differentiating with respect to w, we get:

$$2X^T X w - 2X^T y = 0$$

Equation 6.5 Derivative of NLL wrt w in matrix form

from which we can write the following:

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

Equation 6.6 Optimum Regression Weights

One problem with above estimation is that it can result in over-fitting. To make the Bayesian linear regression robust against overfitting, we can encourage the parameters to be small by placing a zero-mean Gaussian prior:

$$p(w) = \prod_{d} N(w_d|0, \tau^2)$$

Equation 6.7 Zero-Mean Gaussian Prior Over Regression Weights

Thus, we can re-write our regularized objective as:

$$\min_w \text{NLL}(w, \sigma^2) + \lambda ||w||_2^2$$

Equation 6.8 Regularized Bayesian Linear Regression Objective

Solving for w as before, we get the following coefficients:

$$\hat{w}_{ridge} = (X^T X + \lambda I)^{-1} X^T y$$

Equation 6.9 Regularized Optimum Regression Weights

We can learn the parameters `w` using gradient descent! Let's look at the following pseudo-code:

```
class ridge_reg:
function fit(X, y)
for i = 1, 2, ..., num_iter
    ŷ = w^T X
    grad = −(y − ŷ)^T X + λw
    w = w − η_i grad        ← Update regression weights
end for
return w
function predict(w, X)
ŷ = w^T X                   ← Make a prediction
return ŷ
```

The `ridge_reg` class consists of `fit` and `predict` function. In the `fit` function, we compute the gradient of the objective function wrt `w` and updated the weight parameters. In the `predict` function, we use the learned regression weights to make a prediction on test data.

Listing 6.1 Bayesian Linear Regression

```
import math
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing

class ridge_reg():
```

```
    def __init__(self, n_iter=20, learning_rate=1e-3, lmbda=0.1):
     self.n_iter = n_iter
     self.learning_rate = learning_rate
     self.lmbda = lmbda

  def fit(self, X, y):
     X = np.insert(X, 0, 1, axis=1)    #A

     self.loss = []
     self.w = np.random.rand(X.shape[1])

     for i in range(self.n_iter):
         y_pred = X.dot(self.w)
         mse = np.mean(0.5*(y - y_pred)**2 + 0.5*self.lmbda*self.w.T.dot(self.w))
         self.loss.append(mse)
         print(" %d iter, mse: %.4f" %(i, mse))
         grad_w = - (y - y_pred).dot(X) + self.lmbda*self.w   #B
         self.w -= self.learning_rate * grad_w   #C

 def predict(self, X):
     X = np.insert(X, 0, 1, axis=1)  #D
     y_pred = X.dot(self.w)
     return y_pred

if __name__ == "__main__":

    X, y = fetch_california_housing(return_X_y=True)
    X_reg = X[:,2].reshape(-1,1) #E
    X_std = (X_reg - X_reg.mean())/X.std() #F
    y_std = (y - y.mean())/y.std() #F

    X_std = X_std[:200,:]
    y_std = y_std[:200]

    rr = ridge_reg()
    rr.fit(X_std, y_std)
    y_pred = rr.predict(X_std)

    print(rr.w)

    plt.figure()
    plt.plot(rr.loss)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.tight_layout()
    plt.show()

    plt.figure()
    plt.scatter(X_std, y_std)
    plt.plot(np.linspace(-1,1), rr.w[1]*np.linspace(-1,1)+rr.w[0], c='red')
    plt.xlim([-0.01,0.01])
    plt.xlabel("scaled avg num of rooms")
    plt.ylabel("scaled house price")
    plt.show()
```

#A insert const 1 for bias term
#B compute gradient of NLL(w) wrt w
#C update the weights

```
#D insert const 1 for bias term
#E average number of rooms
#F standard scaling
```

Figure 6.1 shows the output of Bayesian linear regression algorithm.

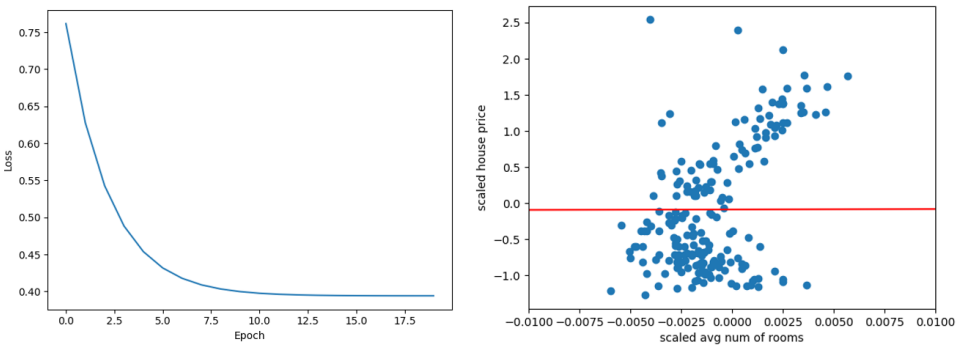


Figure 6.1 Bayesian Linear Regression loss function (left) and plot (right)

We can see the decrease in loss function over epochs on the left and a fit to California house pricing dataset projection on the right. Note that both axis are standardized. The Bayesian linear regression is able to capture the trend of increasing house price with average number of rooms. In the next section, we are going to look into the benefits of hierarchical model of linear regression.

6.3 Hierarchical Bayesian Regression

Hierarchical models enable sharing of features among groups. The parameters of the model are assumed to be sampled from a common distribution that models similarity between groups. Figure 6.2 shows three different scenarios that illustrate the benefit of hierarchical modeling. In the figure on the left, we have a single set of parameters θ that model the entire sequency of observations referred to as a pooled model. Here any variation in data is not modelled explicitly since we are assuming a common set of parameters that give rise to the data. On the other hand, we have an unpooled scenario where we model a different set of parameters for each observation. In the unpooled case, we are assuming that there is no sharing of parameters between groups of observations and that each parameter is independent. The hierarchical model combines the best of both worlds: it assumes that there's a common distribution from which individual parameters are sampled and therefore captures similarities between groups.

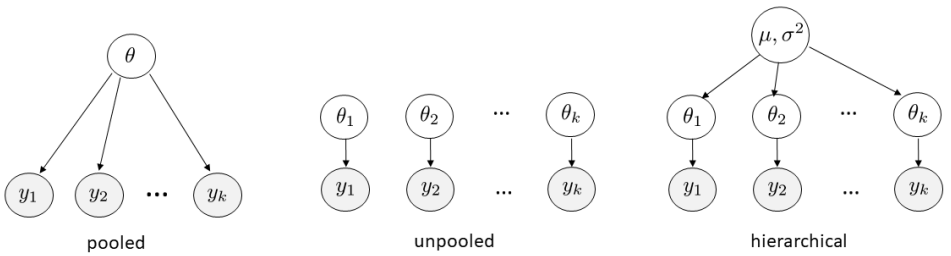


Figure 6.2 Pooled, Unpooled, and Hierarchical Graphical Models

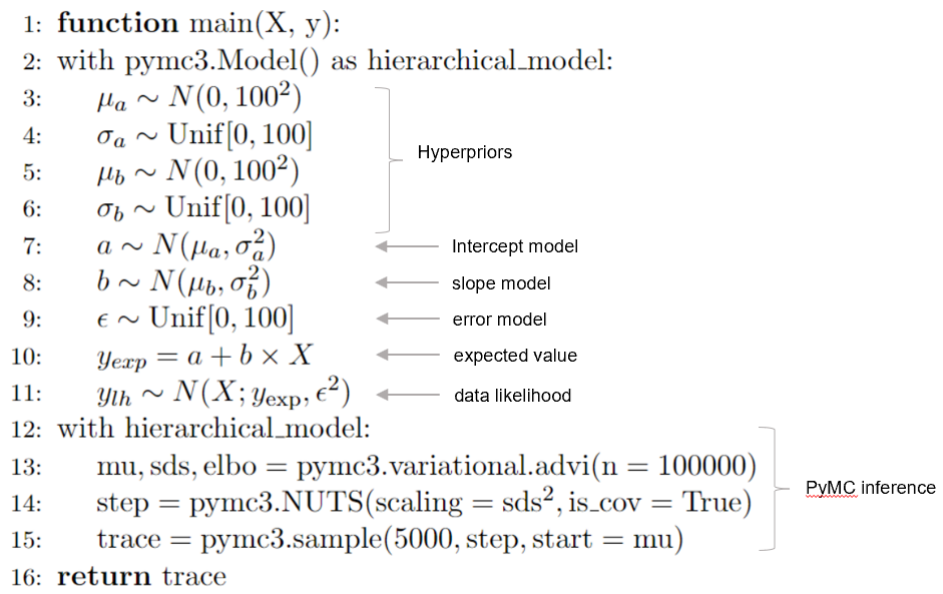
In Bayesian Hierarchical Regression, we can assign priors on model parameters and use MCMC sampling to infer posterior distributions. We use the radon dataset to regress radon gas levels in houses of different counties based on the floor number (in particular if there's a basement or not). Thus, our regression model looks like the following:

$$\begin{aligned} \alpha_c &\sim N(\mu_a, \sigma_a^2) \\ \beta_c &\sim N(\mu_\beta, \sigma_\beta^2) \\ \text{radon}_c &= \alpha_c + \beta_c \times \text{floor}_{i,c} + \epsilon_c \end{aligned}$$

Equation 6.10 Hierarchical Bayesian Regression Model

Notice that subscript c indicates a county, thus we are learning an intercept and a slope for each county sampled from a shared Gaussian distribution. Thus, we are assuming a hierarchical model in which our parameters (α_c and β_c) are sampled from common distributions. In the following code listing, we are going to define probability distributions over regression coefficients and model the data likelihood as the normal distribution with uniform standard deviation. Having specified the graphical model, we can run inference using the No-U-Turn Sampler (NUTS) with the help of PyMC library. PyMC is probabilistic programming library in Python which can be downloaded from https://docs.pymc.io/ It is an excellent tool for Bayesian modeling and can be considered from scratch since we are defining the probabilistic graphical model from scratch and then using off-the-shelf tools for sampling the posterior distribution. If this is your first exposure to probabilistic programming languages, I highly recommend going through PyMC online examples to learn more about its capabilities.

Let's take a look at the following pseudo-code.



The code consists of a single `main` function. In the first section of the code, we are defining the probabilistic model and in the second section we are using PyMC3 library for variational inference. First, we set the hyperpriors for the mean and variance of the regression intercept and slope models. Next, we define the intercept and slope model as the Gaussian RVs and we define the error model as a Uniform RV. Finally, we compute the regression expression and set it as a mean in the data likelihood model. We then proceed with NUTS inference implemented in PyMC.

Listing 6.2 Hierarchical Bayesian Regression

```
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

import pymc3 as pm

def main():

 data = pd.read_csv('./data/radon.txt')   #A

 county_names = data.county.unique()
 county_idx = data['county_code'].values

 with pm.Model() as hierarchical_model:

     mu_a = pm.Normal('mu_alpha', mu=0., sd=100**2)   #B
     sigma_a = pm.Uniform('sigma_alpha', lower=0, upper=100)  #B
```

```
    mu_b = pm.Normal('mu_beta', mu=0., sd=100**2)   #B
    sigma_b = pm.Uniform('sigma_beta', lower=0, upper=100)   #B

    a = pm.Normal('alpha', mu=mu_a, sd=sigma_a, shape=len(data.county.unique()))   #C
    b = pm.Normal('beta', mu=mu_b, sd=sigma_b, shape=len(data.county.unique()))    #D

    eps = pm.Uniform('eps', lower=0, upper=100)   #E

    radon_est = a[county_idx] + b[county_idx] * data.floor.values   #F

    y_like = pm.Normal('y_like', mu=radon_est, sd=eps, observed=data.log_radon)  #G


with hierarchical_model:
    # Use ADVI for initialization
    mu, sds, elbo = pm.variational.advi(n=100000)
    step = pm.NUTS(scaling=hierarchical_model.dict_to_array(sds)**2, is_cov=True)
    hierarchical_trace = pm.sample(5000, step, start=mu)


pm.traceplot(hierarchical_trace[500:])
plt.show()

if __name__ == "__main__":
 main()
```

#A load data
#B hyperpriors
#C intercept for each county
#D slope for each county
#E model error
#F expected value
#G data likelihood

From the trace plots in Figure 6.3 we can see convergence in our posterior distributions for α_c and β_c indicating different intercepts and slopes for different counties.

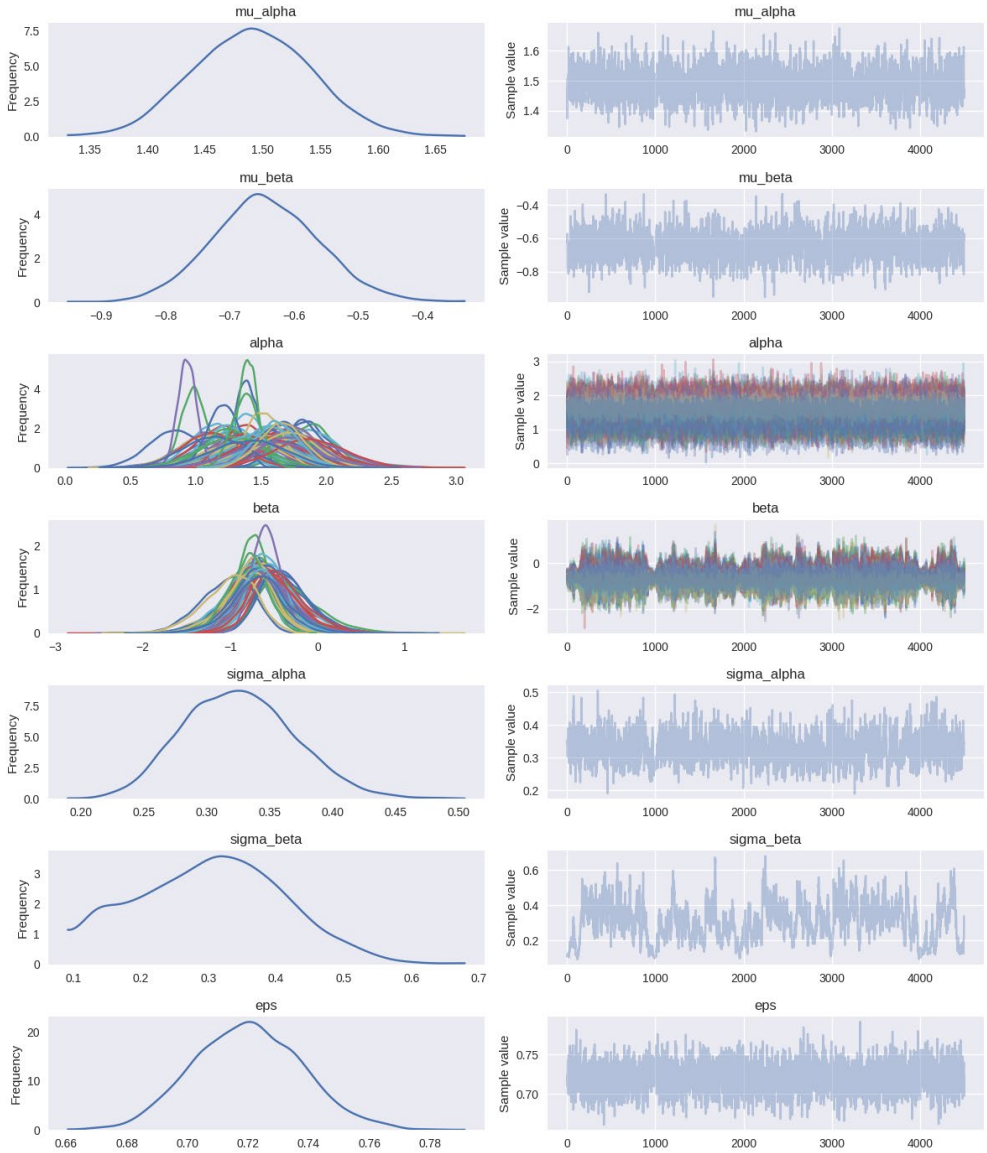


Figure 6.3 MCMC traceplots for hierarchical Bayesian regression

In addition, we also recover the posterior distribution of the shared parameters. μ_a tells us that the group mean of log radon levels is close to 1.5, while μ_b tells us that the slope is

negative with a mean of `-0.65` and therefore having no basement decreases radon levels. In the next section, we are going to look at an algorithm suitable for non-linear data.

6.4 KNN Regression

K Nearest Neighbors (KNN) regression is an example of a non-parametric model, in which for a given query data point `q`, we find its `k` nearest neighbors in the training set and compute the average response variable `y`. In this section, we'll compute the average of KNN target labels for the Iris dataset. The average is taken over the local neighborhood of `K` points that are closest to our query `q`:

$$y_q = \frac{1}{K} \sum_{i \in N_K(q,D)} y_i$$

Equation 6.11 K Nearest Neighbors Regression

where $N_K(q,D)$ denotes the local neighborhood of `k` nearest neighbors to query `q` from the training dataset `D`. To find the local neighborhood $N_K(q,D)$, we can compute a distance between the query point `q` and each of the training dataset points `xi ∈ D`, sort these distances in ascending order and take the top `K` data points. The run-time complexity of this approach is $O(n \log n)$, where `n` is the size of the training dataset due to the sort operation. We are now ready to implement a KNN regressor from scratch! In the following listing, KNN regression is computed by averaging the labels of K nearest neighbors based on Euclidean distance.

```
class KNN:
function knn_search(K, X, y, Q)
for query in Q:
    idx = argsort(euclidean_dist(query, X))[:K]   <-- KNN IDs
    knn_labels = [y[i] for i in idx]              <-- KNN labels
    y_pred = mean(knn_labels)                     <-- KNN regression
end for
return y_pred
```

The code consists of `knn_search` function in which for every `K` nearest neighbor query `Q`, we compute the Euclidean distance between the query and all of the data points, sort the results and pick out K lowest distance IDs. We then form KNN region by collecting the labels with KNN IDs. Finally, we compute our result by averaging over KNN labels.

Listing 6.3 K Nearest Neighbors Regression

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn import datasets
from sklearn.model_selection import train_test_split

np.random.seed(42)

class KNN():

  def __init__(self, K):
     self.K = K

  def euclidean_distance(self, x1, x2):
     dist = 0
     for i in range(len(x1)):
         dist += np.power((x1[i] - x2[i]), 2)
     return np.sqrt(dist)

  def knn_search(self, X_train, y_train, Q):
     y_pred = np.empty(Q.shape[0])

     for i, query in enumerate(Q):
         idx = np.argsort([self.euclidean_distance(query, x) for x in X_train])[:self.K]
       #A
         knn_labels = y_train[idx]   #B
         y_pred[i] = np.mean(knn_labels)    #C

     return y_pred


if __name__ == "__main__":

    plt.close('all')

    #iris dataset
    iris = datasets.load_iris()
    X = iris.data[:,:2]
    y = iris.target

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
       random_state=42)

    K = 4
    knn = KNN(K)
    y_pred = knn.knn_search(X_train, y_train, X_test)

    plt.figure(1)
    plt.scatter(X_train[:,0], X_train[:,1], s = 100, marker = 'x', color = 'r', label =
       'data')
    plt.scatter(X_test[:,0], X_test[:,1], s = 100, marker = 'o', color = 'b', label =
       'query')
    plt.title('K Nearest Neighbors (K=%d)'% K)
    plt.legend()
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.grid(True)
```

```
    plt.show()
```

#A get K nearest neighbors to query point
#B `extract KNN training labels`
#C `compute average of KNN training labels`

Finding the exact nearest neighbors in high-dimensional space is often computationally intractable, and therefore there exist approximate methods. There are two classes of approximate methods: ones that partition the space into regions such as k-d tree implemented in FLANN (fast library for approximate nearest neighbors) library and hashing based methods such as locality sensitive hashing (LSH). In the next section, we are going to look at a different type of regression over functions.

6.5 Gaussian Process Regression

Gaussian processes (GPs) define a prior over functions that can be updated to a posterior once we have observed data (C. E. Rasmussen et al, "Gaussian Processes for Machine Learning", The MIT Press, 2006). In a supervised setting, the function gives a mapping between the data points `xi` and the target value `yi: yi = f(xi)`. Gaussian processes infer a distribution over functions given the data `p(f|x,y)` and then use it to make predictions given new data. A GP assumes that the function is defined at a finite and arbitrary chosen set of points `x1,...,xn`, such that `p(f(x1),...,f(xn))` is jointly Gaussian with mean `μ(x)` and covariance `Σ(x)`, where `Σ_ij=κ(x_i,x_j)` and `κ` is a positive definite kernel function.One example which can be solved by Gaussian Process regression is predicting the CO2 level based on observed measurements. In our code listing, we will assume a sinusoidal model and a radial basis function kernel.

Consider a simple regression problem:

$$f(x) = x^T w \qquad y = f(x) + \epsilon \qquad \epsilon \sim N(0, \sigma_n^2)$$

Equation 6.12 Bayesian Linear Regression Definition

Assuming independent and identically distributed noise, we can write down the likelihood function:

$$p(y|X,w) = \prod_{i=1}^{n} p(y_i|x_i,w) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\{-\frac{(y_i - x_i^T w)^2}{2\sigma_n^2}\} \sim N(Xw, \sigma_n^2 I)$$

Equation 6.13 Likelihood function used in posterior computation

In Bayesian framework, we need to specify a prior over the parameters: `w~N(0,Σp)`. Writing only the terms of the likelihood and the prior which depend on the weights, we get:

$$
\begin{aligned}
p(w|X,y) &\propto \exp\{-\frac{1}{2\sigma_n^2}||y - Xw||^2\}\exp\{-\frac{1}{2}w^T\Sigma_p^{-1}w\} \\
&\propto \exp\{-\frac{1}{2}(w-\bar{w})(\frac{1}{\sigma_n^2}XX^T + \Sigma_p^{-1})(w-\bar{w})\} \\
&\sim N(\frac{1}{\sigma_n^2}A^{-1}Xy, A^{-1})
\end{aligned}
$$

Equation 6.14 Posterior distribution over the weights

Where

$$A = \sigma_n^{-2}XX^T + \Sigma_p^{-1}$$

Thus, we have a closed form posterior distribution over the parameters `w`. To make predictions using this equation, we need to invert the matrix `A` of size `p x p`.

Assuming the observations are noiseless, we want to predict the function outputs `y*=f(x*)`. Consider the following joint GP distribution:

$$\begin{pmatrix} f \\ f_* \end{pmatrix} \sim N\left(\begin{pmatrix} \mu \\ \mu_* \end{pmatrix}, \begin{pmatrix} K & K_* \\ K_*^T & K_{**} \end{pmatrix}\right)$$

Equation 6.15 Joint GP distribution for prediction

where `K=k(X,X)`, `K*=k(X,X*)` and `K**=k(X*,X*)`. Using standard rules for conditioning Gaussians, the posterior has the following form:

$$
\begin{aligned}
p(f_*|X_*, X, f) &\sim N(f_*|\mu_*, \Sigma_*) \\
\mu_* &= \mu(X_*) + K_*^T K^{-1}(f - \mu(X)) \\
\Sigma_* &= K_{**} - K_*^T K^{-1} K_*
\end{aligned}
$$

Equation 6.16 Posterior predictive distribution

In the following code listing, we are going to use Radial Basis Function kernel as a measure of similarity defined in Equation 6.17. We are now ready to implement GP regression from scratch!

1: **class** GPreg:
2: **function** kernel_func(x, z)
3: $\text{Kfn} = \exp\{-\frac{1}{2\sigma^2}||x - z||^2\}$ ← Radial Basis Function Kernel
4: **return** Kfn
5: **function** compute_posterior(X)
6: K = kernel_func(X_train, X_train)
7: Ks = kernel_func(X_train, X_test)
8: Kss = kernel_func(X_test, X_test)
9: $\mu_{post} = \mu(X_{test}) + K_s^T K^{-1}(f - \mu(X_{train}))$ } Gaussian Process Posterior
10: $\Sigma_{post} = K_{ss} - K_s^T K^{-1} K_s$
11: **return** $\mu_{post}, \Sigma_{post}$

The code consists of `kernel_func` and `compute_posterior` functions. The `kernel_func` returns Radial Basis Function (RBF) kernel which measures similarity between two inputs `x` and `z`. In `computer_posterior` function, we first compute the ingredients required for posterior mean and covariance equations and then compute and return posterior mean and covariance as derived in the text.

Listing 6.4 Gaussian Process Regression

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist

np.random.seed(42)

class GPreg:

 def __init__(self, X_train, y_train, X_test):

     self.L = 1.0
     self.keps = 1e-8

     self.muFn = self.mean_func(X_test)
     self.Kfn = self.kernel_func(X_test, X_test) + 1e-15*np.eye(np.size(X_test))

     self.X_train = X_train
     self.y_train = y_train
     self.X_test = X_test

 def mean_func(self, x):
     muFn = np.zeros(len(x)).reshape(-1,1)
     return muFn

 def kernel_func(self, x, z):
     sq_dist = cdist(x/self.L, z/self.L, 'euclidean')**2
```

```
    Kfn = 1.0 * np.exp(-sq_dist/2)
    return Kfn

 def compute_posterior(self):
    K = self.kernel_func(self.X_train, self.X_train)
    Ks = self.kernel_func(self.X_train, self.X_test)
    Kss = self.kernel_func(self.X_test, self.X_test) +
      self.keps*np.eye(np.size(self.X_test))
    Ki = np.linalg.inv(K) #A

    postMu = self.mean_func(self.X_test) + np.dot(np.transpose(Ks), np.dot(Ki,
      (self.y_train - self.mean_func(self.X_train))))
    postCov = Kss - np.dot(np.transpose(Ks), np.dot(Ki, Ks))

    self.muFn = postMu
    self.Kfn = postCov

    return None

 def generate_plots(self, X, num_samples=3):
    plt.figure()
    for i in range(num_samples):
        fs = self.gauss_sample(1)
        plt.plot(X, fs, '-k')
        #plt.plot(self.X_train, self.y_train, 'xk')

    mu = self.muFn.ravel()
    S2 = np.diag(self.Kfn)
    plt.fill(np.concatenate([X, X[::-1]]), np.concatenate([mu - 2*np.sqrt(S2), (mu +
      2*np.sqrt(S2))[::-1]]), alpha=0.2, fc='b')
    plt.show()

 def gauss_sample(self, n):   #B
    A = np.linalg.cholesky(self.Kfn)
    Z = np.random.normal(loc=0, scale=1, size=(len(self.muFn),n))
    S = np.dot(A,Z) + self.muFn   #C
    return S

def main():

    # generate noise-less training data
    X_train = np.array([-4, -3, -2, -1, 1])
    X_train = X_train.reshape(-1,1)
    y_train = np.sin(X_train)

    # generate  test data
    X_test = np.linspace(-5, 5, 50)
    X_test = X_test.reshape(-1,1)

    gp = GPreg(X_train, y_train, X_test)
    gp.generate_plots(X_test,3)  #D
    gp.compute_posterior()
    gp.generate_plots(X_test,3)  #E


if __name__ == "__main__":
    main()
```

```
#A O(N_train^3)
#B returns n samples from multivariate Gaussian distribution
#C S = AZ + mu
#D samples from GP prior
#E samples from GP posterior
```

Figure 6.4 shows three functions drawn at random from a GP prior (left) and GP posterior (right) after observing five data points in the case of noise-free observations. The shaded area corresponds to two times the standard deviation around the mean (`95%` confidence region). We can see that the model perfectly interpolates the training data and that the predictive uncertainty increases as we move further away from the observed data.

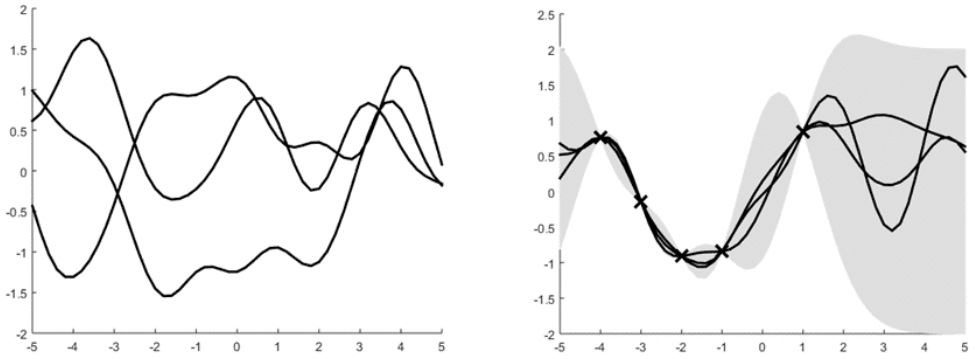


Figure 6.4 Gaussian Process Regression: samples from the prior (left) and posterior (right)

Since our algorithm is defined in terms of inner products in the input space, it can be lifted into feature space by replacing the inner products with `k(x,x')`, this is often referred to as the *kernel trick*. The kernel measures similarity between objects and it doesn't require pre-processing them into feature vector format. For example, a common kernel function is a *radial basis function*:

$$k(x, x') = \exp\left(- \frac{||x - x'||^2}{2\sigma^2} \right)$$

Equation 6.17 Radial Basis Function (RBF) Kernel

In the case of a Gaussian kernel, the feature map lives in an infinite dimensional space. In this case, it is clearly infeasible to explicitly represent the feature vectors.

Regression algorithms help us predict continuous quantities. Based on the nature of data (e.g. linear vs non-linear relationship between the variables), we can choose either a linear algorithm such as Bayesian linear regression or a non-linear K nearest neighbors regression. We may benefit from a hierarchical model in which certain features are shared among the

population. Also, in the case of predicting functional relationships between variables, Gaussian Process Regression provides the answer to do so. In the following chapter, we are going to look at more advanced supervised learning algorithms.

6.6 Exercises

6.1 Compute run-time and memory complexity of a KNN regressor

6.2 Derive Gaussian Process (GP) update equations based on the rules for conditioning of multivariate Gaussian random variables.

6.7 Summary

- The goal of a regression algorithm is to learn a mapping from inputs `x` to outputs `y`, where `y` is a continuous quantity.
- In Bayesian linear regression defined by $y(x) = w^T x + \epsilon$, we assume that the noise term is a zero-mean Gaussian random variable.
- Hierarchical models enable sharing of features among groups. A hierarchical model assumes that there's a common distribution from which individual parameters are sampled and therefore captures similarities between groups.
- K Nearest Neighbors (KNN) regression is a non-parametric model, in which for a given query data point `q`, we find its `k` nearest neighbors in the training set and compute the average response variable `y`.
- Gaussian processes (GPs) define a prior over functions that can be updated to a posterior once we have observed data. A GP assumes that the function is defined at a finite and arbitrary chosen set of points `x1,...,xn`, such that `p(f(x1),...,f(xn))` is jointly Gaussian with mean `μ(x)` and covariance `Σ(x)`, where `Σ_ij=κ(xi,xj)` and `κ` is a positive definite kernel function.

7

Selected Supervised Learning Algorithms

This chapter covers

- **Markov Models: Page Rank and HMM**
- **Imbalanced Learning: Undersampling and Oversampling strategies**
- **Active Learning: Uncertainty Sampling and Query by Committee Strategies**
- **Model Selection: Hyperparameter Tuning**
- **Ensemble Methods: Bagging, Boosting, and Stacking**
- **ML Research: Supervised Learning Algorithms**

In the previous two chapters, we looked at supervised algorithms for classification and regression. In this chapter, we focus on a selected set of supervised learning algorithms. The algorithms are selected to give an exposure to a variety of applications: from time-series models used in computational finance, to imbalanced learning used in fraud detection, to acive learning used to reduce the number of training labels, to model selection and ensemble methods used in all data science competitions. Finally, we conclude with ML research and exercises. Let's begin by reviewing the fundamentals of Markov models.

7.1 Markov Models

In this section, we discuss probabilistic models for sequence of observations. Timeseries models have a wide range of applications from computational finance to speech recognition to computational biology. We are going to look at two popular algorithms build upon properties of Markov chains: the page rank algorithm and the EM algorithm for Hidden Markov Models (HMMs).

However, before we dive into individual algorithms, let's start with common fundamentals. A Markov model for a sequence of random variables $X_1, \ldots, X_T$ of order one is a joint probability model that can be factorized as follows:

$$p(x_1, ..., x_T) = p(x_1)p(x_2|x_1) \cdots p(x_T|x_{T-1}) = p(x_1) \prod_{t=2}^{T} p(x_t|x_{t-1})$$

Equation 7.1 Markov model factorization

Notice, the memory of one in the above equation, which says that conditioned on the present state, the future is independent of the past. In other words, $X_{(t-1)}$ serves as a **sufficient statistic** for X_t.

Let's look at the transition probability $p(x_t|x_{(t-1)})$ in more detail. For a discrete state sequence $x_t \in \{1, \ldots, K\}$, we can represent the **transition probability** as a $K \times K$ stochastic matrix (in which the rows sum to 1):

$$A_{ij} = p(X_t = j|X_{t-1} = i), \text{where} \sum_j A_{ij} = 1 \ \forall i$$

Equation 7.2 Transition Probability

Figure 7.1 shows a simple Markov model with two states.

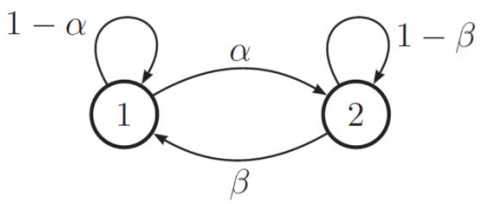


Figure 7.1 A two-state Markov model

Where `alpha` is the transition probability out of state `1` and `1-alpha` is the probability of staying in state `1`. Notice how the transition probabilities out of each state add up to `1`. We can write down the corresponding transition probability matrix as follows:

$$A = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}$$

Equation 7.3 Transition Probability Matrix

If α and β do not vary over time, in other words if the transition matrix A is independent of time, we call the Markov chain stationary or time-invariant.

We are often interested in the long-term behavior of the Markov chain, namely a distribution over states based on the frequency of visits to each state as time passes. Such distribution is known as the **stationary distribution**. Let's compute it! Let $\pi 0$ be the initial distribution over the states. Then after the first transition, we have:

$$\pi_1(j) = \sum_i \pi_0(i) A_{ij}$$

Equation 7.4 Distribution over states after 1 transition

Or in matrix notation:

$$\pi_1 = \pi_0 A$$

Equation 7.5 One transition in matrix form

We can keep going and write down the second transition as follows:

$$\pi_2 = \pi_1 A = \pi_0 A^2$$

Equation 7.6 Two transitions in matrix form

We see that raising the transition matrix A to a power of n is equivalent to modeling n hops (or transitions) of the Markov chain. After some time, we reach a state when left-multiplying the row state vector π by the matrix A gives us the same vector π:

$$\pi = \pi A$$

Equation 7.7 Stationary Distribution

In the above case, we found the stationary distribution and it's equal to π which is an eigen-vector of A that corresponds to the eigen-value of 1. We will also state here (with proof left as an exercise) that a stationary distribution exists if and only if the chain is **recurrent** (i.e. it can return to any state with probability 1) and **aperiodic** (i.e. it doesn't oscillate).

7.1.1 Page Rank Algorithm

Google uses page rank algorithm to rank millions of web pages. We can formulate a collection of n web-pages as a graph $G=(V,E)$. Let every node $v \in V$ represent a web-page and let every edge $e \in E$ represent a directed link from one page to another. Then G is a sparse graph with $|V|=n$. Intuitively, a web-page that receives a lot of incoming links from reputable sources should be promoted towards the top of ranked list. Knowing how the pages are linked allows us to construct a giant transition matrix A, where Aij is the probability of following a link from page i to page j. Given this formulation, we can interpret the entry πj as the importance or rank of page j. In light of Markov chain discussion in the previous section, the page rank is the stationary distribution π, i.e. the eigen-vector of A corresponding to the eigen-value of 1:

$$\pi_j = \sum_i A_{ij}\pi_i$$

Equation 7.8 Transition from i to j

For the stationary distribution π (i.e. page rank) to exist we need to guarantee that the Markov chain described by the transition matrix is recurrent and aperiodic. Let's look at how we can construct such transition matrix. Let's model the interaction with web-pages as follows. If we have outgoing links for a specific page, then with probability $p>0.5$ (p can be chosen using a simulation), we jump to one of the outgoing links (decided uniformly at random), and with probability $1-p$ we open a new page (also uniformly at random). In the event that there are no outgoing links, we open a new page (decided uniformly at random). These two conditions ensure that the Markov chain is recurrent and aperiodic since random jumps can include self-transitions and thus every state is reachable from every other state. Let Gij be the adjacency matrix and $dj = \sum_i G\_ij$ represent the out-degree of page j. Then if the out-degree is 0, we jump to a random page with probability $1/n$, else with probability p we follow a link with probability equal to the out-degree $1/d\_j$ and with probability $1-p$, we jump to a random page. Let's summarize our transition matrix below:

$$A_{ij} = \begin{cases} pG_{ij}/d_j + (1-p)/n & \text{if } d_j \neq 0 \\ 1/n & \text{if } d_j = 0 \end{cases}$$

Equation 7.9 Page Rank Transition Matrix

However, how do we find the eigen-vector π given the enormous size of our transition matrix A? We are going to use an iterative algorithm called **power method** for computing the page rank. Let's $v0$ be an arbitrary vector in the range of A, we can initialize $v0$ at random. Consider now repeatedly multiplying v by A and re-normalizing v:

$$\begin{aligned} v_t &= Av_{t-1} \\ v_t &= v_t/||v_t|| \\ \lambda &= v_t^T A v_t \end{aligned}$$

Equation 7.10 Power Iterations

If we repeat the above algorithm until convergence ($||v\_t|| \approx ||v(t-1)||$) or until maximum number of iterations T is reached, we get our stationary distribution $\pi = v_T$. The reason why this works is because we can write out our matrix $A = U\Lambda U^T$, then

$$\begin{aligned} v_t &= Av_{t-1} = A^t v_0 = (U\Lambda U^T)^t v_0 = U(\Lambda^t U^T v_0) \\ &= a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots + a_n\lambda_n^t u_n \\ &= \lambda_1^t(a_1 u_1 + a_2(\lambda_2/\lambda_1)^t u_2 + \cdots + a_n(\lambda_n/\lambda_1)^t u_n) \\ &\rightarrow \lambda_1^t a_1 u_1 \end{aligned}$$

Equation 7.11 Power Iterations Derivation

for some coefficiens ai and since U is an ortho-normal matrix (i.e. $U^TU=I$) and $|\lambda k|/|\lambda 1|<1$ for $k>1$, vt converges to $u1$, which is equal to our stationary distribution! We can summarize page rank algorithm in the following pseudo code:

1: **class** page_rank
2: **function** power_iteration(A):
3: $v_0 \sim \text{Unif}[0,1]^d$
4: **while** (not converged) and (iter $\leq$ max_iter):
5: $v_t = Av_{t-1}$
6: $v_t = v_t/||v_t||$
7: $\lambda = v_t^T A v_t$
8: converged $= ||v_t - v_{t-1}|| \leq$ tolerance
9: **end while**
10: **return** λ, v_t

We start by sampling the initial value of our vector `v` from a `d`-dimensional uniform distribution. Next, we iterate by multiplying `A` and `v` and re-normalizing `v`.We repeat the process until we either converge or exceed the maximum number of iterations. We are now ready to implement the power method algorithm for computing page rank from scratch.

Listing 7.1 Page Rank Algorithm

```
import numpy as np
from numpy.linalg import norm

np.random.seed(42)

class page_rank():

    def __init__(self):
        self.max_iter = 100
        self.tolerance = 1e-5

    def power_iteration(self, A):
        n = np.shape(A)[0]
        v = np.random.rand(n)
        converged = False
        iter = 0

        while (not converged) and (iter < self.max_iter):
            old_v = v
            v = np.dot(A, v)
            v = v / norm(v)
            lambd = np.dot(v, np.dot(A, v))
            converged = norm(v - old_v) < self.tolerance
            iter += 1
        #end while

        return lambd, v

if __name__ == "__main__":

    X = np.random.rand(10,5)    #A
    A = np.dot(X.T, X)    #A

    pr = page_rank()
    lambd, v = pr.power_iteration(A)

    print(lambd)
    print(v)

    #compare against np.linalg implementation
    eigval, eigvec = np.linalg.eig(A)    #B
    idx = np.argsort(np.abs(eigval))[::-1]    #B
    top_lambd = eigval[idx][0]    #B
    top_v = eigvec[:,idx][0]    #B
```

#A construct a symmetric random real matrix for simplicity
#B compare against np.linalg implementation

As we can see, our implementation successfully finds the eigen-vector of `A` corresponding the eigen-value of `1`, which is equal to the page rank.

7.1.2 Hidden Markov Model

Hidden Markov Models (HMMs) represent time-series data in applications ranging from stock market prediction to DNA sequencing. HMMs are based on discrete-state Markov chain with

latent states `zt ∈ {1,...,K}`, a transition matrix `A` and an emission matrix `E` that models observed data `X` emitted from each state. A HMM graphical model is shown in Figure 7.2:

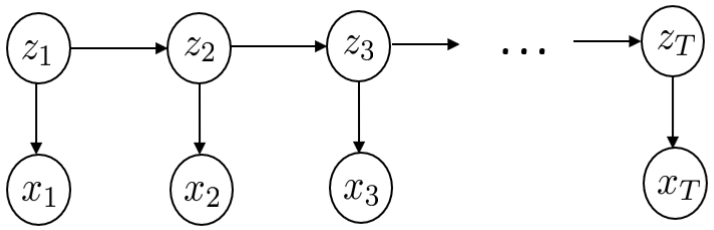


Figure 7.2 Hidden Markov Model

We can write down the joint probability density as follows:

$$\underbrace{p(z_{1:T}, x_{1:T}|\theta)}_{\text{Joint Distribution}} = \overbrace{p(z_{1:T}|\theta)p(x_{1:T}|z_{1:T},\theta)}^{\text{Factorization according to HMM model}} = \underbrace{p(z_1|\pi)}_{\text{Initial state distribution}}\overbrace{\left[\prod_{t=2}^{T} p(z_t|z_{t-1},A)\right]}^{\text{State transition distribution}}\underbrace{\left[\prod_{t=1}^{T} p(x_t|z_t,E)\right]}_{\text{State emission distribution}}$$

Equation 7.12 HMM joint probability distribution

where `θ = {π,A,E}` are HMM parameters with `π` being the initial state distribution. The number of states `K` is often determined by the application. For example, we can model sleep / wake cycle using `K=2` states. The data itself can be either discrete or continuous. We are going to focus on the discrete case in which the emission matrix `Ekl=p(xt=l|zt=k)`. The transition matrix is assumed to be time-invariant and equal to `Aij = p(zt=j|z(t-1)=i)`.

We are typically interested in predicting the unobserved latent state `z` based on emitted observations `x` at any given time. In mathematical notation, we are after `p(zt=j|x_(1:t))`. Let's call this quantity $\alpha_t(j)$:

$$\alpha_t(j) = p(z_t = j|x_{1:t})$$

Equation 7.13 Definition of alpha

Let's compute it! Notice a recurrent relationship between successive values of alpha. Let's try to develop this recurrence.

Prediction Step:

$$\begin{aligned} p(z_t = j|x_{1:t-1}) &= \sum_{z_{t-1}} p(z_t = j, z_{t-1}|x_{1:t-1}) = \sum_i p(z_t = j|z_{t-1} = i)p(z_{t-1} = i|x_{1:t-1}) \\ &= \sum_i A(i,j)\alpha_{t-1}(i) \end{aligned}$$

Equation 7.14 Prediction Step

In the prediction step, we are summing over all possible i states and multiplying the alpha by the transition matrix into `j` state. Let's use the above result in the update step.

Update Step:

$$\begin{aligned} \alpha_t(j) &= p(z_t = j|x_{1:t}) = p(z_t = j|x_t, x_{1:t-1}) \\ &= \frac{p(x_t|z_t = j, x_{1:t-1})p(z_t = j|x_{1:t-1})}{p(x_t|x_{1:t-1})} \\ &= \frac{p(x_t|z_t = j)p(z_t = j|x_{1:t-1})}{\sum_{z_t} p(z_t, x_t|x_{1:t-1})} \\ &= \frac{p(x_t|z_t = j)p(z_t = j|x_{1:t-1})}{\sum_j p(x_t|z_t = j)p(z_t = j|x_{1:t-1})} \\ &= \frac{1}{Z_t} E_t(j) \sum_i A(i,j)\alpha_{t-1}(i) \end{aligned}$$

Equation 7.15 Update Step

where we used the result of prediction step in the update step. We can summarize our derivation in matrix notation as follows:

$$\alpha_t \propto E_t\left(A^T \alpha_{t-1}\right)$$

Equation 7.16 Alpha Recursion in Matrix Notation

Given the recursion in alpha and the initial condition, we can compute the latent state marginals. The above algorithm is referred to as the **forward algorithm** due to its forward recursive pass to compute the alphas. It's a real-time algorithm suitable (with appropriate

features) for applications such as hand-writing or speech recognition. However, we can improve our estimates of the marginals by considering all of the data up to time `T`. Let's figure out how we can do that by introducing the backward pass. The basic idea behind **forward-backward algorithm** is to partition the Markov chain into past and future by conditioning on `zt`. Let `T` be the time our time-series ends. Let's define the marginal probability over all observed data as:

$$\begin{aligned}\gamma_t(j) &= p(z_t = j|x_{1:T}) = p(z_t = j|x_{1:t}, x_{t+1:T}) \\ &= \frac{1}{Z_t} p(z_t = j|x_{1:t}) p(x_{t+1:T}|z_t = j) \\ &\propto \alpha_t(j)\beta_t(j)\end{aligned}$$

Equation 7.17 Marginal Probability

where we defined `βt(j)=p(x(t+1:T)|zt=j)` and used conditional independence of the past and future chain conditioned on state `zt`. The question remains: how to compute `βt(j)`? We can use a similar recursion going backwards from time `t=T`:

$$\begin{aligned}\beta_{t-1}(i) &= p(x_{t:T}|z_{t-1} = i) = \sum_j p(z_t = j, x_t, x_{t+1:T}|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j, x_t, z_{t-1} = i) p(z_t = j, x_t|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j) p(z_t = j, x_t|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j) p(x_t|z_t = j, z_{t-1} = i) p(z_t = j|z_{t-1} = i) \\ &= \sum_j \beta_t(j) E_t(j) A(i, j)\end{aligned}$$

Equation 7.18 Derivation of beta

We can summarize our derivation in the matrix notation as follows:

$$\beta_{t-1} = A(E_t \beta_t)$$

Equation 7.19 Beta Recursion in Matrix Notation

where the base case is given by:

$$\beta_T(i) = p(X_{T+1:T}|z_T = i) = 1$$

Equation 7.20 Beta Base Case

since the sequence ends at time `T` and `X(T+1:T)` is a non-event with probability `1`. Having computed both alpha and beta messages we can combine them to produce our smoothed marginals: `γt(j) ∝ αt(j) βt(j)`.

Let's now look at how we can decode maximum likelihood sequence of transitions between the latent state variables `zt`. In other words, we would like to find:

$$z^* = \arg\max_{z_{1:T}} p(z_{1:T}|x_{1:T})$$

Equation 7.21 Optimal Latent State Sequence

Let `δt(j)` be the probability of ending up in state `j` given the most probable path sequence `z(1:t-1)*`:

$$\delta_t(j) = \max_{z_1,\ldots,z_{t-1}} p(z_t = j, z_{1:t-1}|x_{1:t})$$

Equation 7.22 Definition of delta

We can represent it recursively as

$$\delta_t(j) = \max_i \delta_{t-1}(i) A(i,j) E_t(j)$$

Equation 7.23 Recursion of delta

The above algorithm is known as **Viterbi algorithm**. Let's summarize what we studied so far in the following pseudo-code.

1: **class** HMM
2: **function** forward_backward:
3: **for** t = 1 to n:
4: $\alpha_t = \text{normalize}(E_t A^T \alpha_{t-1})$
5: **end for**
6: **for** t = n-1 to 1:
7: $\beta_t = \text{normalize}(A(E_t \beta_{t+1}))$
8: **end for**
9: $\gamma = \alpha \cdot \beta$
10: **return** γ, α, β
11: **function** viterbi:
12: **for** t = 1 to n:
13: $\delta_t[j] = \max_i(\delta_{t-1}[i] A[i,j] E_t[j])$
14: **end for**

We have two main functions `forward_backward` and `viterbi`. In the `forward_backward` function, we construct sparse matrix `X` of emission indicators and compute the forward probabilities `alpha`, normalizing in every iteration. Similarly, we compute the backward probabilities `beta` as previously derived. Finally, we compute the marginal probabilities `gamma` by multiplying `alpha` and `beta`. In the `viterbi` function, we apply log scale for numerical stability and replace multiplication with addition, we compute the expression for `delta` as derived earlier. We are now ready to implement the inference of Hidden Markov Model (HMM) from scratch!

Listing 7.2 Forward-Backward HMM Algorithm

```
import numpy as np
from scipy.sparse import coo_matrix
import matplotlib.pyplot as plt

np.random.seed(42)

class HMM():
    def __init__(self, d=3, k=2, n=10000):
        self.d = d    #A
        self.k = k    #B
        self.n = n    #C

        self.A = np.zeros((k,k))    #D
        self.E = np.zeros((k,d))    #E
        self.s = np.zeros(k)     #F

        self.x = np.zeros(self.n)    #G

    def normalize_mat(self, X, dim=1):
        z = np.sum(X, axis=dim)
        Xnorm = X/z.reshape(-1,1)
        return Xnorm

    def normalize_vec(self, v):
        z = sum(v)
        u = v / z
        return u, z

    def init_hmm(self):

        #initialize matrices at random
        self.A = self.normalize_mat(np.random.rand(self.k,self.k))
        self.E = self.normalize_mat(np.random.rand(self.k,self.d))
        self.s, _ = self.normalize_vec(np.random.rand(self.k))

        #generate markov observations
        z = np.random.choice(self.k, size=1, p=self.s)
        self.x[0] = np.random.choice(self.d, size=1, p=self.E[z,:].ravel())
        for i in range(1, self.n):
            z = np.random.choice(self.k, size=1, p=self.A[z,:].ravel())
            self.x[i] = np.random.choice(self.d, size=1, p=self.E[z,:].ravel())
        #end for

    def forward_backward(self):

        #construct sparse matrix X of emission indicators
        data = np.ones(self.n)
        row = self.x
        col = np.arange(self.n)
        X = coo_matrix((data, (row, col)), shape=(self.d, self.n))

        M = self.E * X
        At = np.transpose(self.A)
        c = np.zeros(self.n)  #normalization constants
        alpha = np.zeros((self.k, self.n))  #alpha = p(z_t = j | x_{1:T})
        alpha[:,0], c[0] = self.normalize_vec(self.s * M[:,0])
        for t in range(1, self.n):
```

```
            alpha[:,t], c[t] = self.normalize_vec(np.dot(At, alpha[:,t-1]) * M[:,t])
        #end for

        beta = np.ones((self.k, self.n))
        for t in range(self.n-2, 0, -1):
            beta[:,t] = np.dot(self.A, beta[:,t+1] * M[:,t+1])/c[t+1]
        #end for
        gamma = alpha * beta

        return gamma, alpha, beta, c

    def viterbi(self):

        #construct sparse matrix X of emission indicators
        data = np.ones(self.n)
        row = self.x
        col = np.arange(self.n)
        X = coo_matrix((data, (row, col)), shape=(self.d, self.n))

        #log scale for numerical stability
        s = np.log(self.s)
        A = np.log(self.A)
        M = np.log(self.E * X)

        Z = np.zeros((self.k, self.n))
        Z[:,0] = np.arange(self.k)
        v = s + M[:,0]
        for t in range(1, self.n):
            Av = A + v.reshape(-1,1)
            v = np.max(Av, axis=0)
            idx = np.argmax(Av, axis=0)
            v = v.reshape(-1,1) + M[:,t].reshape(-1,1)
            Z = Z[idx,:]
            Z[:,t] = np.arange(self.k)
        #end for
        llh = np.max(v)
        idx = np.argmax(v)
        z = Z[idx,:]

        return z, llh


if __name__ == "__main__":

    hmm = HMM()
    hmm.init_hmm()

    gamma, alpha, beta, c = hmm.forward_backward()
    z, llh = hmm.viterbi()
```

#A dimension of data
#B dimension of latent state
#C number of data points
#D transition matrix
#E emission matrix
#F initial state vector
#G emitted observations

After the code is executed, we return `alpha`, `beta`, and `gamma` parameters for a randomly initialized HMM model. After calling viterbi decoder, we retrieve the maximum likelihood sequence of state transitions `z`.

In the following section, we are going to look at imbalanced learning, in particular, undersampling and oversampling strategies for equalizing the number of samples among different classes.

7.2 Imbalanced Learning

Most classification algorithms will only perform optimally when the number of samples in each class is roughly the same. Highly skewed datasets where the minority class is outnumbered by one or more classes commonly occur in fraud detection, medical diagnosis and computational biology. One way of addressing this issue is by re-sampling the dataset to offset the imbalance and arrive at a more robust and accurate decision boundary. Re-sampling techniques can be broadly divided into four categories: undersampling the majority class, over-sampling the minority class, combining over and under sampling, and creating ensemble of balanced datasets.

7.2.1 Undersampling Strategies

Undersampling methods remove data from the majority class of the original dataset as shown in Figure 7.3

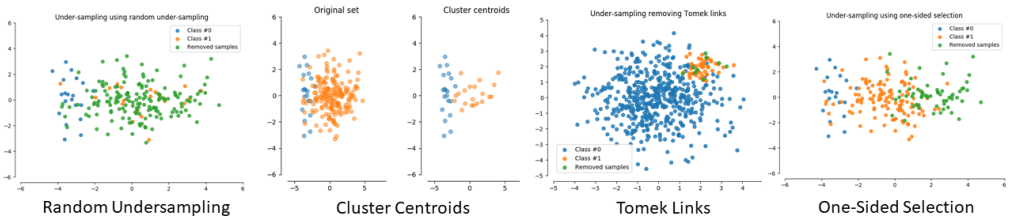


Figure 7.3 Undersampling Strategies: Random, Cluster Centroids, Tomek Links and One-Sided Selection.

Random Under Sampler simply removes data points from the majority class uniformly at random. **Cluster Centroids** is a method that replaces cluster of samples by the cluster centroid of a K-means algorithm, where the number of clusters is set by the level of undersampling. Another effective undersampling method is **Tomek links** that removes unwanted overlap between classes. Tomek links are removed until all minimally distanced nearest neighbor pairs are of the same class. A Tomek link is defined as follows: given an instance pair `(xi,xj)`, where `xi ∈ S_min`, `xj ∈ S_maj` and `d(xi,xj)` is the distance between `xi` and `xj`, then the `(xi,xj)` pair is called a Tomek link if there's no instance `xk` such that `d(xi,xk) < d(xi,xj)` or `d(xj,xk) < d(xi,xj)`. In this way, if two instances form a Tomek link then either one of these instances is noise or both are near a border. Therefore, one can use Tomek links to clean up overlap between classes. By removing overlapping examples, one can establish well-defined clusters in the training set and lead to

improved classification performance. The **One Sided Selection** (OSS) method selects a representative subset of the majority class `E` and combines it with the set of all minority examples `S_min` to form `N={E ∪ S_min}`. The reduced set `N` is further processed to remove all majority class Tomek links.

Let's experiment with Tomek links undersampling using the imbalanced-learn library.

Listing 7.3 Tomek Links Algorithm

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from imblearn.under_sampling import TomekLinks

rng = np.random.RandomState(42)

def main():

    n_samples_1 = 500
    n_samples_2 = 50
    X_syn = np.r_[1.5 * rng.randn(n_samples_1, 2), 0.5 * rng.randn(n_samples_2, 2) + [2,
       2]]    #A
    y_syn = np.array([0] * (n_samples_1) + [1] * (n_samples_2))
    X_syn, y_syn = shuffle(X_syn, y_syn)
    X_syn_train, X_syn_test, y_syn_train, y_syn_test = train_test_split(X_syn, y_syn)

    tl = TomekLinks(sampling_strategy='auto')
    X_resampled, y_resampled = tl.fit_resample(X_syn, y_syn)    #B
    idx_resampled = tl.sample_indices_
    idx_samples_removed = np.setdiff1d(np.arange(X_syn.shape[0]),idx_resampled)

    fig = plt.figure()    #C
    ax = fig.add_subplot(1, 1, 1)

    idx_class_0 = y_resampled == 0
    plt.scatter(X_resampled[idx_class_0, 0], X_resampled[idx_class_0, 1], alpha=.8,
       label='Class #0')
    plt.scatter(X_resampled[~idx_class_0, 0], X_resampled[~idx_class_0, 1], alpha=.8,
       label='Class #1')
    plt.scatter(X_syn[idx_samples_removed, 0], X_syn[idx_samples_removed, 1], alpha=.8,
       label='Removed samples')
    plt.title('Undersampling: Tomek links')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

#A generate data
#B remove Tomek links
#C generate plots

Figure 7.4 shows the resulting output.

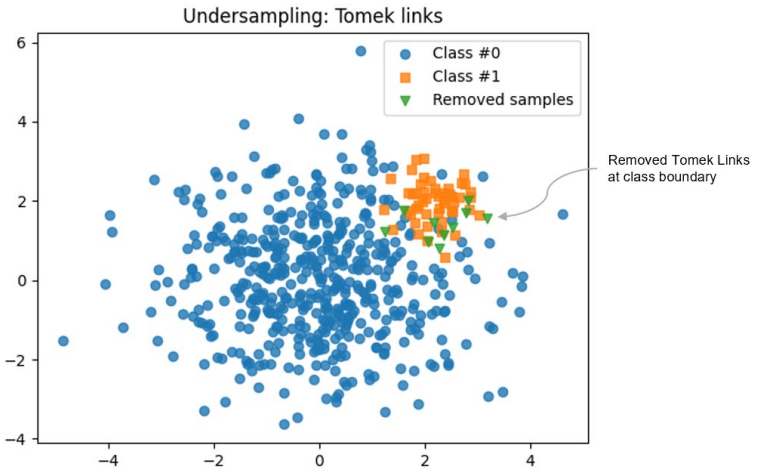


Figure 7.4 Tomek Links Algorithm: showing a number of removed samples at the classification boundary.

Thus, we can see that removing unwanted class overlap can increase robustness of our decision boundary and improve classification accuracy.

7.2.2 Oversampling Strategies

Oversampling methods append data to the minority class of the original dataset as shown in Figure 7.5.

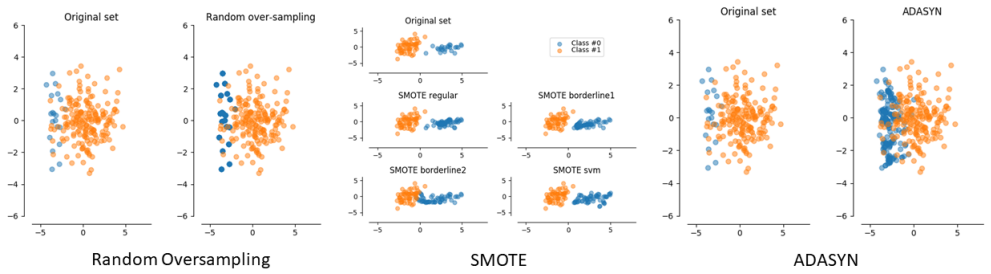


Figure 7.5 Oversampling Strategies.

Random Over Sampler simply adds data points to the minority class uniformly at random. **Synthetic Minority Oversampling Technique** (SMOTE) generates synthetic examples by finding `K` nearest neighbors in the feature space and generating a new data point along the line segments joining any of the `K` minority class nearest neighbors. Synthetic samples are generated in the following way: take the difference between the feature vector (sample) under consideration and its nearest neighbor, multiply this difference by a random number

between `0` and `1` and add it to the feature vector under consideration thus augmenting the dataset with a new data point. Adaptive Synthetic Sampling (ADASYN) uses a weighted distribution for different minority class examples according to their level of difficulty in learning, where more synthetic data is generated for minority class examples that are harder to learn. As a result, the ADASYN approach improves learning of imbalanced dataset in two ways: reducing the bias introduced by class imbalance and adaptively shifting the classification decision boundary toward the difficult examples.

Let's experiment with SMOTE oversampling using the imbalanced-learn library.

Listing 7.4 SMOTE Algorithm

```
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.decomposition import PCA

from imblearn.over_sampling import SMOTE

def plot_resampling(ax, X, y, title):
    c0 = ax.scatter(X[y == 0, 0], X[y == 0, 1], label="Class #0", alpha=0.5)
    c1 = ax.scatter(X[y == 1, 0], X[y == 1, 1], label="Class #1", alpha=0.5)
    ax.set_title(title)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.spines['left'].set_position(('outward', 10))
    ax.spines['bottom'].set_position(('outward', 10))
    ax.set_xlim([-6, 8])
    ax.set_ylim([-6, 6])

    return c0, c1

def main():
    X, y = make_classification(n_classes=2, class_sep=2, weights=[0.3, 0.7],
                               n_informative=3, n_redundant=1, flip_y=0,
                               n_features=20, n_clusters_per_class=1,
                               n_samples=80, random_state=10)    #A

    pca = PCA(n_components=2)
    X_vis = pca.fit_transform(X)    #B

    method = SMOTE()
    X_res, y_res = method.fit_resample(X, y)    #C
    X_res_vis = pca.transform(X_res)

    f, (ax1, ax2) = plt.subplots(1, 2)    #D
    c0, c1 = plot_resampling(ax1, X_vis, y, 'Original')
    plot_resampling(ax2, X_res_vis, y_res, 'SMOTE')
    ax1.legend((c0, c1), ('Class #0', 'Class #1'))
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

```
#A generate the dataset
#B fit PCA for visualization
#C apply regular SMOTE
#D generate plots
```

In the above code listing we generate a high-dimensional dataset with 2 imbalanced classes. We apply regular SMOTE oversampling algorithm and fit a 2-component PCA to visualize the data in 2 dimensions.

Figure 7.6 shows the resulting output.

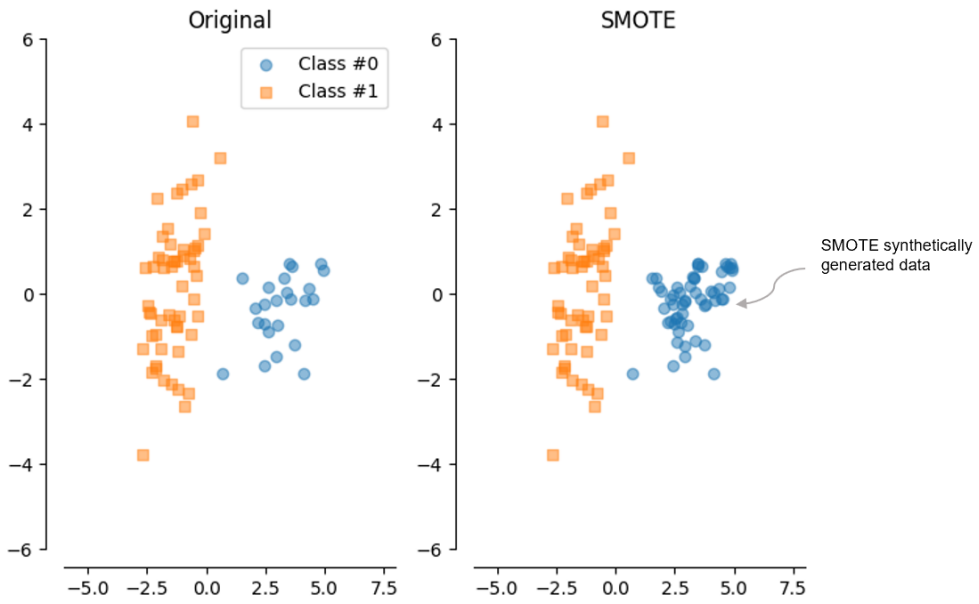


Figure 7.6 SMOTE Algorithm

Thus, we can see that SMOTE densely populates the minority class with synthetic data. It's possible to combine over-sampling and under-sampling techniques into a hybrid strategy. Common examples include SMOTE and Tomek links or SMOTE and Edited Nearest Neighbors (ENN). Additional ways of learning on imbalanced datasets include weighing training instances, introducing different misclassification costs for positive and negative examples and bootstrapping.

In the following section, we will focus on a very important principal in supervised ML that can reduce the number of required training examples: active learning.

7.3 Active Learning

The key idea behind active learning is that a machine learning algorithm can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns. Active learning is well motivated in many modern machine learning problems where unlabeled data may be abundant, but labels are expensive to obtain. Active learning is sometimes called query learning or optimal experimental design because an active learner poses queries in the form of unlabelled data instances to be labeled by an oracle. In this way, the active learner seeks to achieve high accuracy using as few labeled instances as possible. For a review, see B. Settles, "Active Learning Literature Survey", Technical Report, 2009.

We focus on pool-based sampling that assumes that there is a small set of labeled data `L` and a large pool of unlabeled data `U`. Queries are selectively drawn from the pool according to an informativeness measure. Pool based methods rank the entire collection of unlabeled data to select the best query. Therefore, for very large datasets, stream-based sampling maybe more appropriate where the data is scanned sequentially and query decisions are evaluated individually.

Figure 7.7 shows an example of pool-based active learning based on binary classification of a synthetic dataset with two balanced classes using Logistic Regression (LR).

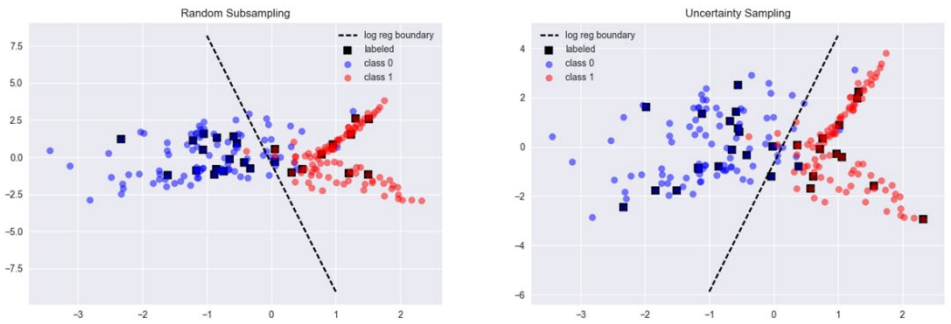


Figure 7.7 Active Learning for Logistic Regression

On the left, we can see the LR decision boundary as a result of training on a randomly subsampled set of `30` labels that achieves a classification accuracy of `90%` on held out data. On the right, we can see the LR decision boundary as a result of training on `30` queries selected by uncertainty sampling based on entropy. Uncertainty sampling achieves a higher classification accuracy of `92.5%` on the held out set.

7.3.1 Query Strategies

Query strategies refer to criteria we use to select a subset of the training examples as part of active learning. We are going to look at two query strategies: uncertainty sampling and query by committee.

Uncertainty Sampling. One of the simplest and most commonly used query framework is uncertainty sampling. In this framework, an active learner queries the label about which it is least certain. For example, in binary logistic regression, uncertainty sampling queries points near the boundary where the probability of being positive is close to `1/2`. For multi-class problems, uncertainty sampling can query points that are least confident:

$$x^*_{LC} = \arg\max_x 1 - P_\theta(\hat{y}|x)$$

Equation 7.24 Least confident definition

where `y ∈ {1,...,K}` is the class label with the highest posterior probability under the model `θ`. The criterion for the least confident strategy only considers information about the most probable label. We can use max margin sampling to preserve information about the remaining label distribution:

$$x^*_M = \arg\min_x P_\theta(\hat{y}_1|x) - P_\theta(\hat{y}_2|x)$$

Equation 7.25 Max Margin Strategy

where `y1` and `y2` are the first and second most probable class labels under the model, respectively. Intuitively, instances with large margins are easy to classify. Thus, points with small margins are ambiguous and knowing their labels would help the model to more effectively discriminate between them. However, for multi-class problems with very large label sets, the margin strategy still ignores much of the output distribution for the remaining classes. A more general uncertainty sampling strategy is based on entropy:

$$x^*_H = \arg\max_x -\sum_i P_\theta(y_i|x) \log P_\theta(y_i|x)$$

Equation 7.26 Entropy Strategy

By learning labels that have highest entropy we can reduce label uncertainty. Uncertainty sampling also works for regression problems, in which case the learner queries the point with highest output variance in its prediction.

Query by Committee. Another query selection framework is the Query By Committee (QBC) algorithm that involves maintaining a committee $C=\{\theta^1, \ldots, \theta^C\}$ of models which are all trained on the current labeled set `L` but represent competing hypotheses. Each committee member is then allowed to vote on the labelings of query candidates and the most informative query is considered to be an instance about which they most disagree. The objective of QBC is to minimize a set of hypotheses that are consistent with the current

labeled training data `L`. For measuring the level of disagreement two main approaches have been proposed: the vote entropy and KL divergence. The vote entropy is defined as follows:

$$x^*_{VE} = \arg\max_x - \sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C}$$

Equation 7.27 Vote Entropy Definition

Where `yi ∈ {1,...,K}` is the class label, `V(yi)` is the number of votes that a label received from the committee members and `C` is the size of the committee. Notice the similarity to equation 7.26. The KL divergence for QBC voting is defined as follows:

$$\begin{aligned} x^*_{KL} &= \arg\max_x \frac{1}{C} \sum_{c=1}^{C} KL(P_{\theta^{(c)}} || P_C) \\ KL(P_{\theta^{(c)}} || P_C) &= \sum_i P_{\theta^{(c)}}(y_i|x) \log \frac{P_{\theta^{(c)}}(y_i|x)}{P_C(y_i|x)} \\ P_C(y_i|x) &= \frac{1}{C} \sum_{c=1}^{C} P_{\theta^{(c)}}(y_i|x) \end{aligned}$$

Equation 7.27 KL divergence for QBC Definition

where θ^c represents a member model of the committee and `Pc(yi|x)` is the consensus probability that `yi` is the predicted label. The KL divergence metric considers the most informative query to be the one with the largest average difference between the label distributions of any one committee member and the consensus distribution.

Variance Reduction. We can reduce the generalization error by minimizing output variance. Consider a regression problem for which the learning objective is to minimize the mean squared error. Let $\bar{\theta} = E[\hat{\theta}]$ be the expected value of the parameter estimate $\hat{\theta}$ and let θ^* be the ground truth, then

$$\begin{aligned}
\text{MSE} &= E\left[(\hat{\theta}-\theta^*)^2\right] = E\left[\left[(\hat{\theta}-\bar{\theta})+(\bar{\theta}-\theta^*)\right]^2\right] \\
&= E\left[(\hat{\theta}-\bar{\theta})^2\right] + 2(\bar{\theta}-\theta^*)E\left[\hat{\theta}-\bar{\theta}\right] + (\bar{\theta}-\theta^*)^2 \\
&= E\left[(\hat{\theta}-\bar{\theta})^2\right] + (\bar{\theta}-\theta^*)^2 \\
&= \text{VAR}[\hat{\theta}] + \text{bias}^2(\hat{\theta})
\end{aligned}$$

Equation 7.28 MSE derivation

This is called the **bias-variance tradeoff**. Thus, it is possible to achieve lower MSE with a biased estimator as long as it reduces the variance. A natural question is how low can the variance be? The answer is given by the Cramer-Rao lower bound that provides a lower bound on the variance of any unbiased estimator.

Cramer-Rao Lower Bound. Assuming `p(x|θ)` satisfies the regularity condition, the variance of any unbiased estimator $\hat{\theta}$ satisfies:

$$\text{VAR}(\hat{\theta}) \geq \frac{1}{-E\left[\frac{\partial^2 \log p(x|\theta)}{\partial \theta^2}\right]} = \frac{1}{I(\theta)}$$

Equation 7.29 Cramer-Rao Lower Bound

where `I(θ)` is the Fisher information matrix. Thus, the Minimum Variance Unbiased (MVU) estimator achieves the minimum variance equal to the inverse of the Fisher information matrix. To minimize variance of parameter estimates, an active learner should select data that maximizes its Fisher information. For multi-variate models with `K` parameters, Fisher information takes the form of a `KxK` matrix:

$$[I(\theta)]_{ij} = -E\left[\frac{\partial^2 \log p(x|\theta)}{\partial \theta_i \partial \theta_j}\right]$$

Equation 7.30 Fisher Information Matrix

As a result, there are several options for minimizing the inverse information matrix: A-optimality minimizes the trace: $Tr(I^{-1}(\theta))$, D-optimality minimizes the determinant: $|I^{-1}(\theta)|$ and E-optimality minimizes the maximum eigenvalue: $\lambda_{max}[I^{-1}(\theta)]$.

However, there are some computational disadvantages to the variance reduction methods. Estimating output variance requires inverting a `KxK` matrix for each unlabeled instance, resulting in a time complexity of $O(UK^3)$, where `U` is the size of the query pool. As a result variance reduction methods are empirically slower than simpler query strategies like uncertainty sampling.

Let's look at the pseudocode for Active Learner class.

```
class ActiveLearner
function uncertainty_sampling(clf, X):
P_θ(ŷ|x) = clf.predict_proba(X)
if strategy = "least confident":
    return arg max_x 1 − P_θ(ŷ|x)
else if strategy = "max margin":
    return arg min_x P_θ(ŷ_1|x) − P_θ(ŷ_2|x)
else if strategy = "entropy":
    return arg max_x − Σ_i P_θ(y_i|x) log P_θ(y_i|x)
end if
function query_by_committee(clf, X):
if strategy = "vote entropy":
   C = len(clf)
   for model in clf:
       y_i = clf.predict(X)
    end for
    V(y_i) = (1/C) Σ_{i=1}^{C} 1[[y_i]]
    return arg max_x − Σ_i (V(y_i)/C) log (V(y_i)/C)
else if strategy = "average kl divergence":
    P_C(y_i|x) = (1/C) Σ_{c=1}^{C} P_θ(c)(y_i|x)
    KL(P_θ(c)||P_C) = Σ_i P_θ(c)(y_i|x) log P_θ(c)(y_i|x) / P_C(y_i|x)
    return arg max_x (1/C) Σ_{c=1}^{C} KL(P_θ(c)||P_C)
end if
```

We have two main functions: `uncertainty_sampling` and `query_by_committee`. In the `uncertainty_sampling` function, we pass in the classifier model `clf` and the unlabeled data `X` as inputs. We then predict the probability of the label given the classifier model and the training data and use this prediction to compute one of three uncertainty sampling strategies as discussed in the text. In the `query_by_committee` function, we implement two committee

methods: vote entropy and average kl divergence. Except, we now pass a set of models `clf`, the predictions of which are used to vote on the predicted label thus forming a distribution for evaluating the entropy. In the KL divergence case, we make use of averages of model predictions in computation of KL divergence between each model and the average. We return the training sample that maximizes this KL divergence. We are now ready to take a look at the following code listing.

Listing 7.5 Active Learner Class

```
from __future__ import unicode_literals, division
from scipy.sparse import csc_matrix, vstack
from scipy.stats import entropy
from collections import Counter
import numpy as np


class ActiveLearner(object):

    uncertainty_sampling_frameworks = [
        'entropy',    #A
        'max_margin',    #A
        'least_confident',    #A
    ]

    query_by_committee_frameworks = [
        'vote_entropy',    #B
        'average_kl_divergence',    #B
    ]

    def __init__(self, strategy='least_confident'):
        self.strategy = strategy

    def rank(self, clf, X_unlabeled, num_queries=None):

        if num_queries == None:
            num_queries = X_unlabeled.shape[0]

        elif type(num_queries) == float:
            num_queries = int(num_queries * X_unlabeled.shape[0])

        if self.strategy in self.uncertainty_sampling_frameworks:
            scores = self.uncertainty_sampling(clf, X_unlabeled)

        elif self.strategy in self.query_by_committee_frameworks:
            scores = self.query_by_committee(clf, X_unlabeled)

        else:
            raise ValueError("this strategy is not implemented.")

        rankings = np.argsort(-scores)[:num_queries]
        return rankings

    def uncertainty_sampling(self, clf, X_unlabeled):
        probs = clf.predict_proba(X_unlabeled)

        if self.strategy == 'least_confident':
            return 1 - np.amax(probs, axis=1)    #C
```

```
        elif self.strategy == 'max_margin':
            margin = np.partition(-probs, 1, axis=1)
            return -np.abs(margin[:,0] - margin[:, 1])    #D

        elif self.strategy == 'entropy':
            return np.apply_along_axis(entropy, 1, probs)    #E

    def query_by_committee(self, clf, X_unlabeled):
        num_classes = len(clf[0].classes_)
        C = len(clf)
        preds = []

        if self.strategy == 'vote_entropy':
            for model in clf:
                y_out = map(int, model.predict(X_unlabeled))
                preds.append(np.eye(num_classes)[y_out])

            votes = np.apply_along_axis(np.sum, 0, np.stack(preds)) / C
            return np.apply_along_axis(entropy, 1, votes)    #F

        elif self.strategy == 'average_kl_divergence':
            for model in clf:
                preds.append(model.predict_proba(X_unlabeled))

            consensus = np.mean(np.stack(preds), axis=0)
            divergence = []
            for y_out in preds:
                divergence.append(entropy(consensus.T, y_out.T))

            return np.apply_along_axis(np.mean, 0, np.stack(divergence))    #G
```

#A uncertainty sampling frameworks
#B query by committee frameworks
#C least confident
#D max margin
#E entropy
#F vote entropy
#G average KL divergence

We apply our Active Learner to logistic regression example below:

Listing 7.6 Active Learner for Logistic Regression

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from active_learning import ActiveLearner
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

np.random.seed(42)

def main():

    num_queries = 30    #A

    data, target = make_classification(n_samples=200, n_features=2, n_informative=2,\
                                       n_redundant=0, n_classes=2, weights = [0.5, 0.5],
       random_state=0)    #B

    X_train, X_unlabeled, y_train, y_oracle = train_test_split(data, target, test_size=0.2,
       random_state=0)     #C

    rnd_idx = np.random.randint(0, X_train.shape[0], num_queries)   #D
    X1 = X_train[rnd_idx,:]
    y1 = y_train[rnd_idx]

    clf1 = LogisticRegression()
    clf1.fit(X1, y1)

    y1_preds = clf1.predict(X_unlabeled)
    score1 = accuracy_score(y_oracle, y1_preds)
    print("random subsampling accuracy: ", score1)

    #plot 2D decision boundary: w2x2 + w1x1 + w0 = 0
    w0 = clf1.intercept_
    w1, w2 = clf1.coef_[0]
    xx = np.linspace(-1, 1, 100)
    decision_boundary = -w0/float(w2) - (w1/float(w2))*xx

    plt.figure()
    plt.scatter(data[rnd_idx,0], data[rnd_idx,1], c='black', marker='s', s=64,
       label='labeled')
    plt.scatter(data[target==0,0], data[target==0,1], c='blue', marker='o', alpha=0.5,
       label='class 0')
    plt.scatter(data[target==1,0], data[target==1,1], c='red', marker='o', alpha=0.5,
       label='class 1')
    plt.plot(xx, decision_boundary, linewidth = 2.0, c='black', linestyle = '--',
       label='log reg boundary')
    plt.title("Random Subsampling")
    plt.legend()
    plt.show()

    AL = ActiveLearner(strategy='entropy')    #E
    al_idx = AL.rank(clf1, X_unlabeled, num_queries=num_queries)

    X2 = X_train[al_idx,:]
```

```
    y2 = y_train[al_idx]

    clf2 = LogisticRegression()
    clf2.fit(X2, y2)

    y2_preds = clf2.predict(X_unlabeled)
    score2 = accuracy_score(y_oracle, y2_preds)
    print("active learning accuracy: ", score2)

    #plot 2D decision boundary: w2x2 + w1x1 + w0 = 0
    w0 = clf2.intercept_
    w1, w2 = clf2.coef_[0]
    xx = np.linspace(-1, 1, 100)
    decision_boundary = -w0/float(w2) - (w1/float(w2))*xx

    plt.figure()
    plt.scatter(data[al_idx,0], data[al_idx,1], c='black', marker='s', s=64,
       label='labeled')
    plt.scatter(data[target==0,0], data[target==0,1], c='blue', marker='o', alpha=0.5,
       label='class 0')
    plt.scatter(data[target==1,0], data[target==1,1], c='red', marker='o', alpha=0.5,
       label='class 1')
    plt.plot(xx, decision_boundary, linewidth = 2.0, c='black', linestyle = '--',
       label='log reg boundary')
    plt.title("Uncertainty Sampling")
    plt.legend()
    plt.show()

if __name__ == "__main__":

    main()
```

#A number of labeled points
#B generate data
#C split into labeled and unlabeled pools
#D random sub-sampling
#E active learning

Active learning and semi-supervised learning both try to make the most of unlabeled data. For example, a basic semi-supervised technique is self-training in which the learner is first trained with a small amount of labeled data and then used to classify the unlabeled data. The most confident unlabeled instances together with their predicted labels are added to the training set and the process repeats.

Figure 7.8 compares `3` uncertainty sampling techniques: least confident, max margin and entropy with a random subsampling baseline on `20` newsgroups dataset classified with Logistic Regression.

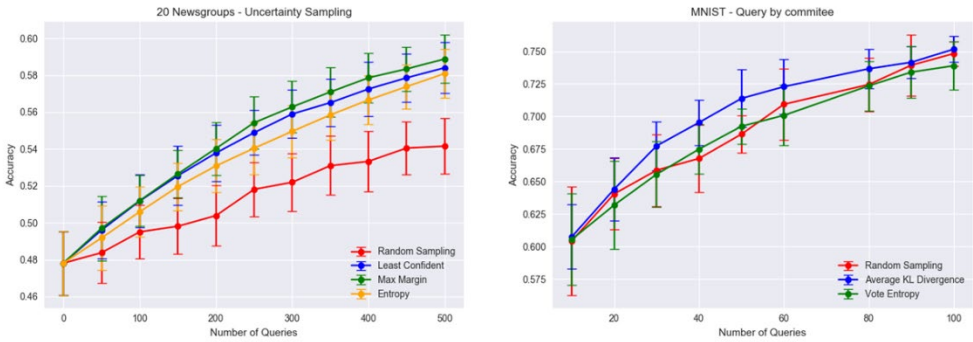


Figure 7.8 Uncertainty Sampling Techniques Comparison. The left figure shows that Active Learning is better than Random Sampling. The right figure shows that Average KL Divergence is better than Random Sampling.

All three methods achieve higher accuracy in comparison to baseline that highlights the benefit of active learning. On the right, we can see the performance of Query By Committee strategy applied to MNIST dataset. The committee consists of `5` instances of Logistic Regression. Two methods are compared against the random subsampling baseline: vote entropy and average KL divergence. We can see that average KL divergence achieves highest classification accuracy. All experiments were repeated `10` times.

7.4 Model Selection: Hyperparameter Tuning

In our machine learning journey, we are often faced with multiple models. Model selection is focused on choosing the optimal model, i.e. the model that has sufficient number of parameters (degrees of freedom) so as to not under-fit and not over-fit to training data. We can summarize model selection using **Occam's razor** principle: "choose the simplest model that explains the data well". In other words, we want to penalize model complexity out of all possible solutions.

In addition, we'd like to explain not just the training data but more so the future data. As a result, we want our model to **generalize** to new and unseen data. A model's behavior is often characterized by its hyper-parameters (such as the number of nearest neighbors or the number of clusters in the K-means algorithm). Let's look at several ways in which we can find the optimum hyper-parameters for model selection.

We often operate in a multi-dimensional hyper-parameter space in which we would like to find an optimum point that leads to the highest performing model on the **validation dataset**. For example, in the case of Support Vector Machines (SVMs), we may try different kernels, kernel parameters and regularization constants.

There are several strategies we can take for hyper-parameter tuning. The most straightforward strategy is called **grid search** which is an exhaustive search over all possible combinations of hyper-parameters. When using grid search, we can set the hyper-parameter values using the log scale (e.g. `0.1, 1, 10, 100`) to arrive at the right order of magnitude. Grid search works well for smaller hyper-parameter spaces.

An alternative to grid search is **random search** which is based on sampling the hyper-parameters from corresponding prior distributions. For example, in finding the number of clusters `K`, we may sample `K` from a discrete exponential distribution. The random search has the computational advantage of finding the optimum set faster compared to an exhaustive grid search. Morever, for random search the number of iterations can be chosen independent of the number of parameters and adding parameters that do not influence performance does not decrease efficiency. A third and most interesting alternative that we are going to look at is called **Bayesian optimization.**

7.4.1 Bayesian Optimization

Rather than exploring the parameter space randomly (according to a chosen distribution), it would be great to adapt an active learning approach that selects continuous parameter values in a way that reduces uncertainty and provides a balance between exploration and exploitation. Bayesian optimization provides an automated Bayesian framework by utilizing Gaussian Processes (GPs) to model algorithm's generalization performance (see J. Snoek, "Practical Bayesian Optimization of Machine Learning Algorithms", NeurIPS, 2012).

Bayesian optimization assumes that a suitable performance function was sampled from a Gaussian Process and maintains a posterior distribution for this function as observations are made: `f(x)~ GP(m(x),k(x,x'))`. To choose which hyperparameters to explore next, one can optimize the Expected Improvement (EI) over the current best result or the Gaussian process Upper Confidence Bound (UCB). EI and UCB have been shown to be efficient in the number of function evaluations required to find the global optimum of multi-modal black-box functions.

Bayesian optimization uses all of the information available from previous evaluations of the objective function as opposed to relying on local gradient and Hessian approximations. This results in an automated procedure that can find an optimum of non-convex functions with relatively few evaluations, at the cost of performing more computation to determine the next point to try. This is particularly useful when evaluations are expensive to perform such as in selecting hyperparameters for deep neural networks.

Bayesian Optimization algorithm is summarized in Figure 7.9.

1: **for** $n = 1, 2, \ldots$ **do**
2: select new x_{n+1} by optimizing acquisition function α
$x_{n+1} = \arg\max_x \alpha(x; D_n, \theta)$
3: query objective function to obtain $y_{n+1} = f(x_{n+1})$
4: augment data $D_{n+1} = \{D_n, (x_{n+1}, y_{n+1})\}$
5: update GP posterior and acquisition function
6: **end for**

Figure 7.9 Bayesian Optimization Algorithm

The following listing applies Bayesian Optimization for hyper-parameter search in SVM and Random Forrest Classifier.

Listing 7.7 Bayesian Optimization for SVM and RF

```
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.svm import SVC

from bayes_opt import BayesianOptimization

np.random.seed(42)

# Load data set and target values
data, target = make_classification(
    n_samples=1000,
    n_features=45,
    n_informative=12,
    n_redundant=7
)
target = target.ravel()

def svccv(gamma):    #A
    val = cross_val_score(
        SVC(gamma=gamma, random_state=0),
        data, target, scoring='f1', cv=2
    ).mean()

    return val

def rfccv(n_estimators, max_depth):    #B
    val = cross_val_score(
        RFC(n_estimators=int(n_estimators),
            max_depth=int(max_depth),
            random_state=0
        ),
        data, target, scoring='f1', cv=2
    ).mean()
    return val

if __name__ == "__main__":

    gp_params = {"alpha": 1e-5}

    #SVM
    svcBO = BayesianOptimization(svccv,
        {'gamma': (0.00001, 0.1)})

    svcBO.maximize(init_points=3, n_iter=4, **gp_params)

    #Random Forest
    rfcBO = BayesianOptimization(
        rfccv,
        {'n_estimators': (10, 300),
         'max_depth': (2, 10)
```

```
        }
    )
    rfcBO.maximize(init_points=4, n_iter=4, **gp_params)

    print('Final Results')
    print(svcBO.max)
    print(rfcBO.max)
```

#A SVM classifier
#B Random Forest (RF) classifier

Figure 7.10 shows Bayesian optimization applied to SVM and Random Forest Classifier.

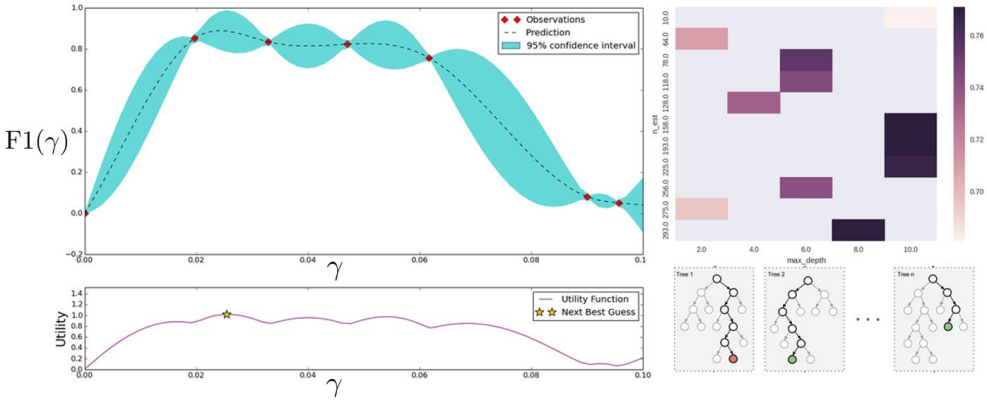


Figure 7.10 Bayesian Optimization applied to SVM and RF

F1 score was used as performance objective function for a classification task. The figure on the left shows Bayesian optimization of `F1` score as a function of the gamma parameter of the SVM RBF kernel: `K(x,x') = exp{-γ||x-x'||^2}`. We can see that after only `7` iterations we have discovered the gamma parameter that gives the maximum `F1` score. The peak of EI utility function at the bottom tells us which experiment to perform next. The figure on the right shows Bayesian optimization of `F1` score as a function of maximum depth and the number of estimators of a Random Forest classifier. From the heatmap, we can tell that the maximum `F1` score is achieved for `158` estimators with depth equal to `10`.

7.5 Ensemble Methods

Ensemble methods are meta-algorithms that combine several machine learning techniques into one predictive model in order to decrease variance (bagging), bias (boosting), or improve predictions (stacking). Ensemble methods can be divided into two groups: *sequential* ensemble methods where the base learners are generated sequentially (e.g. AdaBoost) and *parallel* ensemble methods where the base learners are generated in parallel (e.g. Random Forest). The basic motivation of sequential methods is to exploit the

dependence between the base learners since the overall performance can be boosted by weighing previously mislabeled examples with higher weight. The basic motivation of parallel methods is to exploit independence between the base learners since the error can be reduced dramatically by averaging.

Most ensemble methods use a single base learning algorithm to produce homogeneous base learners, i.e. learners of the same type leading to *homogeneous ensembles*. There are also some methods that use heterogeneous learners, i.e. learners of different types, leading to *heterogeneous ensembles*. In order for ensemble methods to be more accurate than any of its individual members the base learners have to be as accurate as possible and as diverse as possible.

7.5.1 Bagging

Bagging stands for bootstrap aggregation. One way to reduce the variance of an estimate is to average together multiple estimates. For example, we can train `M` different trees `fm` on different subsets of the data (chosen randomly with replacement) and compute the ensemble:

$$f(x) = \frac{1}{M} \sum_{m=1}^{M} f_m(x)$$

Equation 7.31 Tree Ensemble

Bagging uses bootstrap sampling to obtain the data subsets for training the base learners. For aggregating the outputs of base learners, bagging uses voting for classification and averaging for regression.

The following listing shows bagging ensemble experiments.

Listing 7.8 Bagging Ensemble

```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 0:2], iris.target

    clf1 = DecisionTreeClassifier(criterion='entropy', max_depth=None)    #A
    clf2 = KNeighborsClassifier(n_neighbors=1)    #B

    bagging1 = BaggingClassifier(base_estimator=clf1, n_estimators=10, max_samples=0.8,
       max_features=0.8)
    bagging2 = BaggingClassifier(base_estimator=clf2, n_estimators=10, max_samples=0.8,
       max_features=0.8)

    label = ['Decision Tree', 'K-NN', 'Bagging Tree', 'Bagging K-NN']
    clf_list = [clf1, clf2, bagging1, bagging2]

    fig = plt.figure(figsize=(10, 8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    for clf, label, grd in zip(clf_list, label, grid):
        scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')
        print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.std(), label))

        clf.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)
        plt.title(label)

    plt.show()
    #plt.savefig('./figures/bagging_ensemble.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
       random_state=0)

    plt.figure()
```

```
    plot_learning_curves(X_train, y_train, X_test, y_test, bagging1, print_model=False,
       style='ggplot')
    plt.show()
    #plt.savefig('./figures/bagging_ensemble_learning_curve.png')

    #Ensemble Size
    num_est = list(map(int, np.linspace(1,100,20)))
    bg_clf_cv_mean = []
    bg_clf_cv_std = []
    for n_est in num_est:
        print("num_est: ", n_est)
        bg_clf = BaggingClassifier(base_estimator=clf1, n_estimators=n_est,
       max_samples=0.8, max_features=0.8)
        scores = cross_val_score(bg_clf, X, y, cv=3, scoring='accuracy')
        bg_clf_cv_mean.append(scores.mean())
        bg_clf_cv_std.append(scores.std())

    plt.figure()
    (_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_std, c='blue',
       fmt='-o', capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
    plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('Bagging Tree
       Ensemble');
    plt.show()
    #plt.savefig('./figures/bagging_ensemble_size.png')

if __name__ == "__main__":
    main()
```

#A decision tree classifier
#B K nearest neighbors classifier

Figure 7.11 shows the decision boundary of a decision tree and k-NN classifiers along with their bagging ensembles applied to the Iris dataset.

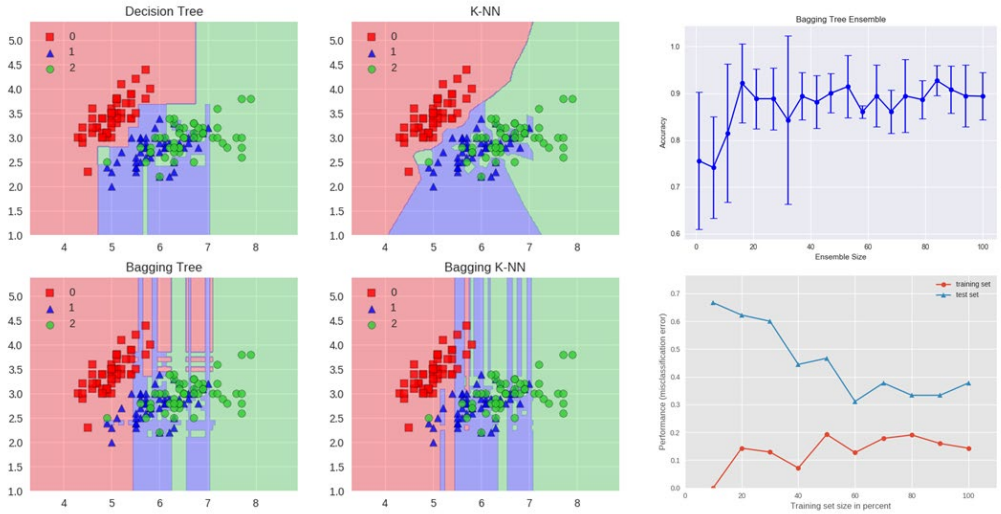


Figure 7.11 Bagging Ensemble Applied to Iris Dataset as generated by code listing

The decision tree shows axes parallel boundaries while the `k=1` nearest neighbors fits closely to the data points. The bagging ensembles were trained using `10` base estimators with `0.8` subsampling of training data and `0.8` subsampling of features. The decision tree bagging ensemble achieved higher accuracy in comparison to k-NN bagging ensemble because k-NN are less sensitive to perturbation on training samples and therefore they are called *stable learners*. Combining stable learners is less advantageous since the ensemble will not help improve generalization performance. Notice also that the decision boundary of the bagging k-NN looks similar to the decision boundary of the bagging tree as a result of voting. The figure also shows how the test accuracy improves with the size of the ensemble. Based on cross-validation results, we can see the accuracy increases until approximately `20` base estimators and then plateaus afterwards. Thus, adding base estimators beyond `20` only increases computational complexity without accuracy gains for the Iris dataset. The figure also shows learning curves for the bagging tree ensemble. We can see an average error of `0.15` on the training data and a U-shaped error curve for the testing data. The smallest gap between training and test errors occurs at around `80%` of the training set size.

A commonly used class of ensemble algorithms are forests of randomized trees. In a **random forest**, each tree in the ensemble is built from a sample drawn with replacement (i.e. a bootstrap sample) from the training set. In addition, instead of using all the features, a random subset of features is selected further randomizing the tree. As a result, the bias of the forest increases slightly but due to averaging of less correlated trees, its variance decreases resulting in an overall better model.

In **extremely randomized trees** algorithm randomness goes one step further: the splitting thresholds are randomized. Instead of looking for the most discriminative threshold, thresholds are drawn at random for each candidate feature and the best of these randomly-

generated thresholds is picked as the splitting rule. This usually allows to reduce the variance of the model a bit more, at the expense of a slightly greater increase in bias.

7.5.2 Boosting

Boosting refers to a family of algorithms that are able to convert weak learners to strong learners. The main principle of boosting is to fit a sequence of weak learners (models that are only slightly better than random guessing, such as small decision trees) to weighted versions of the data, where more weight is given to examples that were mis-classified by earlier rounds. The predictions are then combined through a weighted majority vote (classification) or a weighted sum (regression) to produce the final prediction. The principal difference between boosting and the committee methods such as bagging is that base learners are trained in sequence on a weighted version of the data.

The algorithm below describes the most widely used form of boosting algorithm called **AdaBoost** which stands for adaptive boosting.

1: Init data weights $\{w_n\}$ to $1/N$
2: **for** $m = 1$ to M **do**
3: fit a classifier $y_m(x)$ by minimizing weighted error function J_m:
4: $J_m = \sum_{n=1}^{N} w_n^{(m)} 1[y_m(x_n) \neq t_n]$
5: compute $\epsilon_m = \sum_{n=1}^{N} w_n^{(m)} 1[y_m(x_n) \neq t_n] / \sum_{n=1}^{N} w_n^{(m)}$
6: evaluate $\alpha_m = \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$
7: update the data weights: $w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m 1[y_m(x_n) \neq t_n]\}$
8: **end for**
9: Make predictions using the final model: $Y_M(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(x)\right)$

Figure 7.12 AdaBoost Algorithm for classification

We see that the first base classifier `y1(x)` is trained using weighting coefficients w_n^1 that are all equal. In subsequent boosting rounds, the weighting coefficients w_n^m are increased for data points that are misclassified and decreased for data points that are correctly classified. The quantity ϵ_m represents a weighted error rate of each of the base classifiers. Therefore, the weighting coefficients α_m give greater weight to the more accurate classifiers.

The following listing trains AdaBoost classifier with different number of learners.

Listing 7.9 Boosting Ensemble

```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 0:2], iris.target

    #XOR dataset
    #X = np.random.randn(200, 2)
    #y = np.array(map(int,np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)))

    clf = DecisionTreeClassifier(criterion='entropy', max_depth=1)    #A

    num_est = [1, 2, 3, 10]
    label = ['AdaBoost (n_est=1)', 'AdaBoost (n_est=2)', 'AdaBoost (n_est=3)', 'AdaBoost
       (n_est=10)']

    fig = plt.figure(figsize=(10, 8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    for n_est, label, grd in zip(num_est, label, grid):
        boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
        boosting.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=boosting, legend=2)
        plt.title(label)

    plt.show()
    #plt.savefig('./figures/boosting_ensemble.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
       random_state=0)

    boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=10)

    plt.figure()
    plot_learning_curves(X_train, y_train, X_test, y_test, boosting, print_model=False,
       style='ggplot')
```

```
    plt.show()
    #plt.savefig('./figures/boosting_ensemble_learning_curve.png')

    num_est = list(map(int, np.linspace(1,100,20)))    #B
    bg_clf_cv_mean = []
    bg_clf_cv_std = []
    for n_est in num_est:
        print("num_est: ", n_est)
        ada_clf = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
        scores = cross_val_score(ada_clf, X, y, cv=3, scoring='accuracy')
        bg_clf_cv_mean.append(scores.mean())
        bg_clf_cv_std.append(scores.std())

    plt.figure()
    (_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean, yerr=bg_clf_cv_std, c='blue',
       fmt='-o', capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
    plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('AdaBoost Ensemble');
    plt.show()
    #plt.savefig('./figures/boosting_ensemble_size.png')

if __name__ == "__main__":
    main()
```

#A base classifier
#B ensemble size

The boosting ensemble is illustrated in Figure 7.13

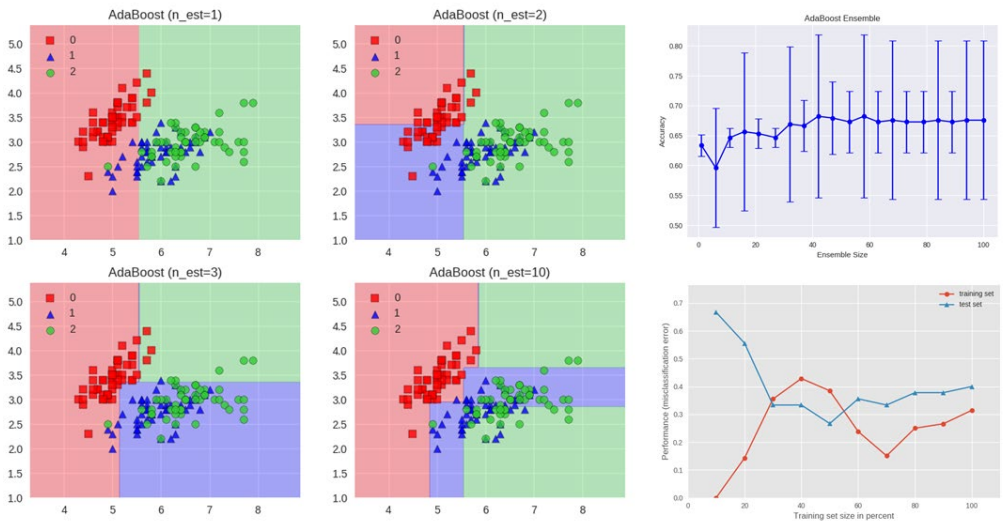


Figure 7.13 Boosting Ensemble

Each base learner consists of a decision tree with depth `1`, thus classifying the data based on a feature threshold that partitions the space into two regions separated by a linear decision surface that is parallel to one of the axes. The figure also shows how the test accuracy improves with the size of the ensemble and the learning curves for training and testing data.

Gradient Tree Boosting is a generalization of boosting to arbitrary differentiable loss functions. It can be used for both regression and classification problems. Gradient Boosting builds the model in a sequential way:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Equation 7.32 Sequential Model Construction

At each stage the decision tree $h_m(x)$ is chosen to minimize a loss function L given the current model $F_{(m-1)}(x)$:

$$F_m(x) = F_{m-1}(x) + \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h(x_i))$$

Equation 7.33 Sequential Loss Minimization

Gradient Boosting attempts to solve this minimization problem numerically via steepest descent. The steepest descent direction is the negative gradient of the loss function evaluated at the current model $F_{(m-1)}$. The algorithms for regression and classification differ in the type of loss function used.

7.5.3 Stacking

Stacking is an ensemble learning technique that combines multiple classification or regression models via a meta-classifier or a meta-regressor. The base level models are trained based on complete training set then the meta-model is trained on the outputs of base level model as features. The base level often consists of different learning algorithms and therefore stacking ensembles are often heterogeneous. The algorithm below summarizes stacking.

1: Input: training data $D = \{x_i, y_i\}_{i=1}^{m}$
2: Ouput: ensemble classifier H
3: *Step 1: learn base-level classifiers*
4: **for** $t = 1$ to T **do**
5: learn h_t based on D
6: **end for**
7: *Step 2: construct new data set of predictions*
8: **for** $i = 1$ to m **do**
9: $D_h = \{x'_i, y_i\}$, where $x'_i = \{h_1(x_i), ..., h_T(x_i)\}$
10: **end for**
11: *Step 3: learn a meta-classifier*
12: learn H based on D_h
13: return H

Figure 7.14 Stacking Algorithm

The listing below shows the stacking classifier in action.

Listing 7.10 Stacking Ensemble

```
import itertools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier

from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 1:3], iris.target

    clf1 = KNeighborsClassifier(n_neighbors=1)
    clf2 = RandomForestClassifier(random_state=1)
    clf3 = GaussianNB()
    lr = LogisticRegression()
    sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                              meta_classifier=lr)    #A

    label = ['KNN', 'Random Forest', 'Naive Bayes', 'Stacking Classifier']
    clf_list = [clf1, clf2, clf3, sclf]

    fig = plt.figure(figsize=(10,8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    clf_cv_mean = []
    clf_cv_std = []
    for clf, label, grd in zip(clf_list, label, grid):

        scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')
        print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(), scores.std(), label))
        clf_cv_mean.append(scores.mean())
        clf_cv_std.append(scores.std())

        clf.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=clf)
        plt.title(label)

    plt.show()
    #plt.savefig("./figures/ensemble_stacking.png")

    #plot classifier accuracy
    plt.figure()
```

```
    (_, caps, _) = plt.errorbar(range(4), clf_cv_mean, yerr=clf_cv_std, c='blue', fmt='-o',
       capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
    plt.xticks(range(4), ['KNN', 'RF', 'NB', 'Stacking'], rotation='vertical')
    plt.ylabel('Accuracy'); plt.xlabel('Classifier'); plt.title('Stacking Ensemble');
    plt.show()
    #plt.savefig('./figures/stacking_ensemble_size.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
       random_state=0)

    plt.figure()
    plot_learning_curves(X_train, y_train, X_test, y_test, sclf, print_model=False,
       style='ggplot')
    plt.show()
    #plt.savefig('./figures/stacking_ensemble_learning_curve.png')


if __name__ == "__main__":
    main()
```

#A stacking classifier

The stacking ensemble is illustrated in Figure 7.15.

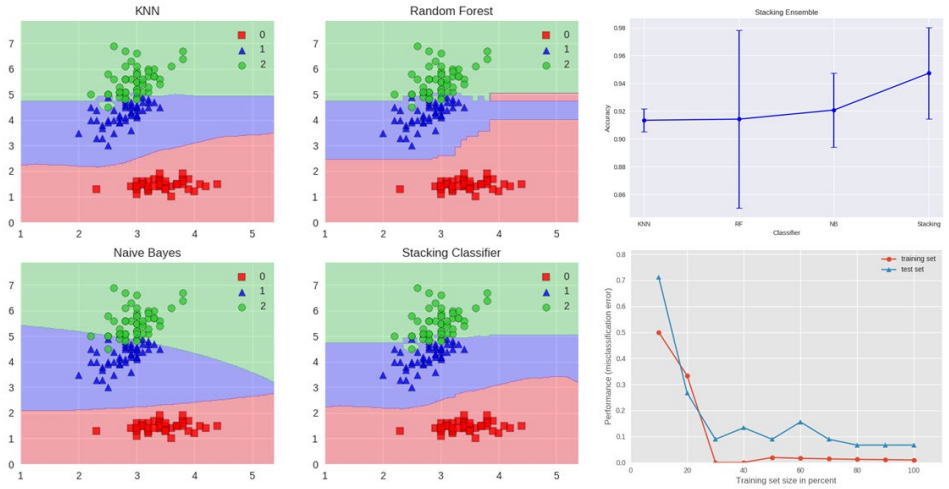


Figure 7.15 Stacking Ensemble

It consists of k-NN, Random Forest and Naive Bayes base classifiers whose predictions are combined by Logistic Regression as a meta-classifier. We can see the blending of decision boundaries achieved by the stacking classifier. The figure also shows that stacking achieves

higher accuracy than individual classifiers and based on learning curves, it shows no signs of overfitting.

7.6 ML Research: Supervised Learning Algorithms

In this section, we cover additional insights and research related to the topics presented above. In the classification section, we derived a number of classic algorithms and built a solid foundation for the design of new algorithms. For the perceptron algorithm, we saw how we can modularize the algorithm by introducing different loss functions leading to slightly different update rules. Moreover, the loss function itself can be weighted to account for imbalanced datasets. Similarly, in the case of SVM, we have the flexibility of choosing the appropriate kernel function. We looked at multi-class SVM, however, a 1-class SVM with RBF kernel can be used for anomaly detection.

There are a number of extensions to solvers besides SGD for logistic regression. Some of them include: liblinear (which uses coordinate descent), newton-cg (a second order method that can lead to faster convergence), and LBFGS (a powerful optimization framework which is both memory and computation efficient). Similarly, we have a number of extensions to the Naive Bayes algorithm based on the type of data it models, such as Multinomial Naive Bayes and Gaussian Naive Bayes. The derivations for which are very similar to the Bernoulli Naive Bayes that we looked at in detail.

In the section of regression, we had our first encounter with a non-parametric model, namely the KNN regressor. We are going to look into this rich set of models in a later chapter when we discuss Bayesian non-parametric models in which the number of parameters grows with data. For example, in the Dirichlet Process (DP) HMM, we have a potentially infinite number of states with probability mass concentrated on the few states supported by the data (e.g. see E. Fox et al, "A Sticky HDP-HMM with Application to Speaker Diarization", Annals of Applied Statistics, 2011)

We observed the advantages of hierarchical models. This technique can be used whenever different data groups have characteristics in common. This allows sharing of statistical strength from the different data groups. Hierarchical models often provide greater accuracy with fewer observations such as hierarchical HMM and hierarchical Dirichlet process mixture models (e.g. J. Chang et al, "Parallel Sampling of HDPs using sub-cluster splits", NeurIPS, 2014.

We touched on scalable ML when we discussed page rank due to computations on millions of web-pages. Scalable ML is an important area of research. In the industry, Spark ML is typically used to process big data. However, to understand how parallelizable algorithms are constructed it helps to implement them from scratch. For more information about scalable algorithms, the reader is encouraged to check out "Scaling Up Machine Learning: Parallel and Distributed Approaches" by R. Bekkerman et al.

There are a number of generalizations of HMMs. One example is a semi-Markov HMM which models state transitions of variable duration commonly found in genomics data. Another example is Hierarchical HMMs that model data with hierarchical structure often present in speech applications. Finally, there are Factorial HMMs that consist of multiple inter-linked Markov chains running in parallel to capture different aspects of the input signal.

There are a number of research directions in the area of active learning. One of them is semi-supervised learning, one example of which is self-training. The learner is first trained on a small amount of labelled data and later augments its own dataset with most-confidently classified new training examples. Another direction is the study of exploration-exploitation trade-off commonly present in reinforcement learning. In particular, the active learner must be proactive and be able to choose well and explore examples for instances that it is unsure how to label. Finally, there's an important area of submodular optimization (see A. Kraus, "Optimizing Sensing: Theory and Applications", PhD Thesis, 2008). In problems with fixed budget for gathering data, its advantageous to formulate the objective function for data selection as a submodular function, which could then be optimized using greedy algorithms with performance guarantees.

In the area of model selection, Reversible Jump (RJ) Markov Chain Monte Carlo (MCMC) is an interesting sampling based method for selecting between models with different number of parameters. RJ-MCMC samples in parameter spaces of different dimensionality; for example, when selecting the best Gausian mixture model with different number of clusters `K`. The interesting part arises in the Metropolis-Hastings sampling when we sample from distributions with different dimensions. RJ-MCMC augments the low dimensional space with extra random variables so that the two spaces have a common measure (see P. Green et al, "Tutorial on Transdimensional MCMC", Highly Structured Stochastic Systems, 2003).

Finally, ensemble methods coupled with model selection and hyper-parameter tuning are the winning models in any data science competition!

7.7 Exercises

7.1 Explain how temperature softmax works for different values of the temperature parameter `T`.

7.2 In the forward-backward HMM algorithm, store the latent state variable `z` (as part of the HMM class) and compare the inferred `z` against the ground truth `z`.

7.8 Summary

- Markov models have the Markov property that conditioned on the present state, the future states are independent of the past. In other words, the present state serves as a sufficient statistic for the next state.
- Page rank is a stationary distribution of the Markov chain described by the transition matrix which is recurrent and aperiodic. At scale, the page rank algorithm is computed using power iterations.
- Hidden Markov Models (HMMs) model time series data and consist of a latent state Markov chain, a transition matrix between the latent states and an emission matrix that models observed data emitted from each state. Inference is carried out using either EM algorithm or Forward-Backward algorithm with Viterbi maximum likelihood sequence decoder
- There are two main strategies for imbalanced learning: undersampling the majority class and over-sampling the minority class. Additional methods include introducing class weights in the loss function.
- In active learning, the ML algorithm chooses the data samples to train on in a way that maximizes learning and requires fewer training examples. There are two main query strategies: uncertainty sampling and query by committee.
- Bias-variance trade-off says that mean square error (MSE) is equal to bias squared plus variance. The minimum variance is given by the inverse of Fisher information which measures the curvature of log-likelihood.
- In model selection, we often want to follow the Occam's razor principle and choose the simplest model that explains the data well. In hyper-parameter optimization, in addition to grid search and random search, Bayesian Optimization uses an active learning approach in a way that reduces uncertainty and provides a balance between exploration and exploitation.
- Ensemble methods are meta-algorithms that combine several machine learning techniques into one predictive model in order to decrease variance (bagging), bias (boosting), or improve predictions (stacking).

8

Fundamental Unsupervised Learning Algorithms

This chapter covers

- **Dirichlet-Process K-Means**
- **Gaussian Mixture Models (GMMs)**
- **Dimensionality Reduction**

In the previous chapters, we looked at supervised algorithms for classification and regression. This chapter focuses on unsupervised learning algorithms. Unsupservised learning takes place when no training labels are available. In this case, we are interested in discovering patterns in data and learning data representations. Applications of unsupervised learning span from clustering customer segments in e-commerce to extracting features from image data. In this chapter, we'll start by looking at Bayesian non-parametric extension of K-means algorithm followed by the EM algorithm for Gaussian Mixture Models (GMMs). We will then look at two different dimensionality reduction techniques, namely PCA and t-SNE in application to learning an image manifold.

8.1 Dirichlet Process K-Means

Dirichlet Process (DP) K-means is a Bayesian non-parametric extension of the K-Means algorithm. K-means is a clustering algorithm that can be applied, for instance, to customer segmentation, where customers are grouped by purchase history, interests, and geographical location. DP-means is similar to K-Means except that new clusters are created whenever a data point is sufficiently far away from all existing cluster centroids. Therefore, the number of clusters grows with data. DP means converges to a local optimum of a K means like objective that includes a penalty for the number of clusters.

We select the cluster K based on the smallest value of:

$$\arg\min_k \{||x_i - \mu_1||^2, ..., ||x_i - \mu_k||^2, \lambda\}$$

Equation 8.1 Cluster Center Selection

The resulting update is analogous to the K means reassignment step, where we reassign a point to the cluster corresponding to the closest mean or we start a new cluster if the squared Euclidean distance is greater than `λ`. The DP means algorithm is summarized below.

1: Init. $K = 1$, $l_1 = \{x_1, ...x_n\}$ and μ_1 the global mean
2: Init. labels $z_i = 1$ for all $i = 1, ..., n$
3: Init. $\lambda = \text{kpp\_init}(X, \text{K}_{\text{init}})$
4: Repeat until convergence:
5: for each x_i:
6: compute $d_{ic} = ||x_i - \mu_c||^2$ for $c = 1, ..., K$
7: if $\min_c d_{ic} > \lambda$, set $K = K + 1$ and $\mu_k = x_i$
8: otherwise, set $z_i = \text{argmin}_c\ d_{ic}$
9: for each cluster $l_j = \{x_i | z_i = j\}$, compute $\mu_j = \frac{1}{|l_j|} \sum_{x \in l_j} x$
10: Compute the objective: $\Sigma_{c=1}^{K} \Sigma_{x \in l_c} ||x - \mu_c||^2 + \lambda K$

Figure 8.1 DP means algorithm

To evaluate cluster performance, we can use the following metrics: Normalized Mutual Information (NMI), Variation of Information (VI) and Adjusted Rand Index (ARI). The NMI is defined as follows:

$$\text{NMI}(X, Y) = \frac{I(X;Y)}{(H(X) + H(Y))/2}$$

Equation 8.2 Normalized Mutual Information

where `H(X)` is the entropy of `X` and `I(X;Y)` is the mutual information between the ground truth label assignments `X` (when they are available) and the computed assignments `Y`. For a review of information measures see the Appendix or T. Cover and J. Thomas, "Elements of Information Theory", Wiley, 2006 for details.

Let p_{XY}`(i,j) = |xi ∩ yj|/N` be the probability that a label belongs to cluster `xi` in `X` and `yj` in `Y`. Define, p_X`(i)=|xi|/N` and p_Y`(j)=|yj|/N` similarly. Then

$$I(X;Y) = \sum_i \sum_j p_{XY}(i,j) \log \frac{p_{XY}(i,j)}{p_X(i)p_Y(j)}$$

Equation 8.3 Mutual Information

Thus, NMI lies between `0` and `1` with higher values indicating more similar label assignments. The Variation of Information (VI) is defined as:

$$VI(X;Y) = H(X) + H(Y) - 2I(X;Y) = H(X|Y) + H(Y|X)$$

Equation 8.4 Variation of Information

Thus, VI decreases as the overlap between label asignments `X` and `Y` increases. The Adjusted Rand Index computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusterings:

$$\begin{aligned} \text{ARI} &= \frac{RI - E[RI]}{\max RI - E[RI]} \\ RI &= \frac{TP + TN}{TP + FP + FN + TN} \end{aligned}$$

Equation 8.5 Adjusted Rand Index

Thus, ARI approaches `1` for cluster assignments that are similar to each other.

Let's implement the Dirichlet Process K-Means algorithm from scratch!

Listing 8.1 Dirichlet-Process K-means

```
import numpy as np
import matplotlib.pyplot as plt

import time
from sklearn import metrics
from sklearn.datasets import load_iris

np.random.seed(42)

class dpmeans:

    def __init__(self,X):   #A
        self.K = 1
        self.K_init = 4
        self.d = X.shape[1]
        self.z = np.mod(np.random.permutation(X.shape[0]),self.K)+1
```

```
        self.mu = np.random.standard_normal((self.K, self.d))
        self.sigma = 1
        self.nk = np.zeros(self.K)
        self.pik = np.ones(self.K)/self.K

        self.mu = np.array([np.mean(X,0)])   #B

        self.Lambda = self.kpp_init(X,self.K_init)   #C

        self.max_iter = 100
        self.obj = np.zeros(self.max_iter)
        self.em_time = np.zeros(self.max_iter)

    def kpp_init(self,X,k):   #D

        [n,d] = np.shape(X)
        mu = np.zeros((k,d))
        dist = np.inf*np.ones(n)

        mu[0,:] = X[int(np.random.rand()*n-1),:]
        for i in range(1,k):
            D = X-np.tile(mu[i-1,:],(n,1))
            dist = np.minimum(dist, np.sum(D*D,1))
            idx = np.where(np.random.rand() < np.cumsum(dist/float(sum(dist))))
            mu[i,:] = X[idx[0][0],:]
            Lambda = np.max(dist)

        print("Lambda: ", Lambda)

        return Lambda   #E

    def fit(self,X):

        obj_tol = 1e-3
        max_iter = self.max_iter
        [n,d] = np.shape(X)

        obj = np.zeros(max_iter)
        em_time = np.zeros(max_iter)
        print('running dpmeans...')

        for iter in range(max_iter):
            tic = time.time()
            dist = np.zeros((n,self.K))

            for kk in range(self.K):   #F
                Xm = X - np.tile(self.mu[kk,:],(n,1))    #F
                dist[:,kk] = np.sum(Xm*Xm,1)    #F

            dmin = np.min(dist,1)    #G
            self.z = np.argmin(dist,1)   #G
            idx = np.where(dmin > self.Lambda)  #G

            if (np.size(idx) > 0):    #G
                self.K = self.K + 1
                self.z[idx[0]] = self.K-1 #cluster labels in [0,...,K-1]
                self.mu = np.vstack([self.mu,np.mean(X[idx[0],:],0)])
                Xm = X - np.tile(self.mu[self.K-1,:],(n,1))
                dist = np.hstack([dist, np.array([np.sum(Xm*Xm,1)]).T])
```

```
            self.nk = np.zeros(self.K)  #H
            for kk in range(self.K):   #H
                self.nk[kk] = self.z.tolist().count(kk)   #H
                idx = np.where(self.z == kk)    #H
                self.mu[kk,:] = np.mean(X[idx[0],:],0)   #H

            self.pik = self.nk/float(np.sum(self.nk))

            for kk in range(self.K):   #I
                idx = np.where(self.z == kk)   #I
                obj[iter] = obj[iter] + np.sum(dist[idx[0],kk],0)   #I
            obj[iter] = obj[iter] + self.Lambda * self.K   #I

            if (iter > 0 and np.abs(obj[iter]-obj[iter-1]) < obj_tol*obj[iter]):  #J
                print('converged in %d iterations\n'% iter)
                break
            em_time[iter] = time.time()-tic
        #end for
        self.obj = obj
        self.em_time = em_time
        return self.z, obj, em_time

    def compute_nmi(self, z1, z2):   #K

        n = np.size(z1)
        k1 = np.size(np.unique(z1))
        k2 = np.size(np.unique(z2))

        nk1 = np.zeros((k1,1))
        nk2 = np.zeros((k2,1))

        for kk in range(k1):
            nk1[kk] = np.sum(z1==kk)
        for kk in range(k2):
            nk2[kk] = np.sum(z2==kk)

        pk1 = nk1/float(np.sum(nk1))
        pk2 = nk2/float(np.sum(nk2))

        nk12 = np.zeros((k1,k2))
        for ii in range(k1):
            for jj in range(k2):
                nk12[ii,jj] = np.sum((z1==ii)*(z2==jj))
        pk12 = nk12/float(n)

        Hx = -np.sum(pk1 * np.log(pk1 + np.finfo(float).eps))
        Hy = -np.sum(pk2 * np.log(pk2 + np.finfo(float).eps))

        Hxy = -np.sum(pk12 * np.log(pk12 + np.finfo(float).eps))

        MI = Hx + Hy - Hxy;
        nmi = MI/float(0.5*(Hx+Hy))

        return nmi

    def generate_plots(self,X):

        plt.close('all')
```

```
        plt.figure(0)
        for kk in range(self.K):
            #idx = np.where(self.z == kk)
            plt.scatter(X[self.z == kk,0], X[self.z == kk,1], \
                        s = 100, marker = 'o', c = np.random.rand(3,), label = str(kk))
        #end for
        plt.xlabel('X1')
        plt.ylabel('X2')
        plt.legend()
        plt.title('DP-means clusters')
        plt.grid(True)
        plt.show()

        plt.figure(1)
        plt.plot(self.obj)
        plt.title('DP-means objective function')
        plt.xlabel('iterations')
        plt.ylabel('penalized l2 squared distance')
        plt.grid(True)
        plt.show()

if __name__ == "__main__":

    iris = load_iris()
    X = iris.data
    y = iris.target

    dp = dpmeans(X)
    labels, obj, em_time = dp.fit(X)
    dp.generate_plots(X)

    nmi = dp.compute_nmi(y,labels)
    ari = metrics.adjusted_rand_score(y,labels)

    print("NMI: %.4f" % nmi)
    print("ARI: %.4f" % ari)
```

#A initialize parameters for DP means
#B initialize mean
#C initialize lambda
#D K++ initialization
#E Lambda is max distance to k++ means
#F assignment step
#G update labels
#H update step
#I compute objective
#J check convergence
#K compute normalized mutual information

The performance of DP-Means algorithm is shown in Figure 8.2

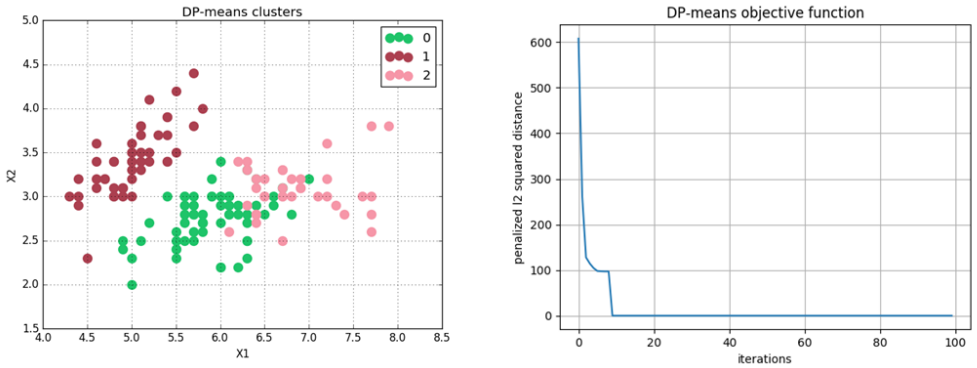


Figure 8.2 DP-means cluster centers (left) and objective (right) for Iris dataset

The performance of DP means algorithm was evaluated on the Iris dataset. We can see good clustering results based on high NMI and ARI metrics. In the following section, we'll take a look at a popular clustering algorithm that models each cluster as a Gaussian distribution taking into account not just the mean but also the covariance information.

8.2 Gaussian Mixture Models (GMMs)

Mixture models are commonly used to model complex density distributions. For example, you may be interested in discovering patterns in census data consisting of information about the person's age, income, occupation, and other dimensions. If we plot the resulting data in high-dimensional space, we'll likely discover non-uniform density characterized by groups or clusters of data points. We can model each cluster using a base probability distribution. Mixture models consist of a convex combination of K base models. In the case of Gaussian mixture models, the base distribution is Gaussian and can be written as follows:

$$p(x_i|\theta) = \sum_{k=1}^{K} \pi_k p_k(x_i|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i|\mu_k, \Sigma_k)$$

Equation 8.6 Gaussian Mixture Model

Where π_k are the mixing proportions that satisfy: $0 \leq \pi_k \leq 1$ and $\sum \pi_k = 1$. In contrast to K-means that only model cluster means, GMM models the cluster covariance as well. Thus, GMMs are able to capture the data more accurately.

8.2.1 EM Algorithm

The EM algorithm provides a way of computing ML/MAP estimates when we have unobserved latent variables and/or missing data. EM exploits the fact that if the data were fully observed,

then the ML/MAP estimates would be easy to compute. In particular, EM is an iterative algorithm which alternates between inferring the latent variables given the parameters (E step) and then optimizing the parameters given filled in data (M step).

In the EM algorithm, we define the complete data log-likelihood `lc(θ)` where `xi` are the observed random variables and `zi` are unobserved. Since we don't know `zi` we can't compute `p(xi,zi|θ)` but we can compute an expectation of `lc(θ)` wrt to parameters $\theta^{(k-1)}$ from the previous iteration.

$$l_c(\theta) = \sum_{i=1}^{N} \log p(x_i, z_i|\theta)$$

Equation 8.7 Complete data log-likelihood

The goal of the E-step is to compute $Q(\theta, \theta^{(k-1)})$ on which the ML/MAP estimates depend. The goal of the M-step is to re-compute θ by finding the ML/MAP estimates:

$$\begin{aligned} \text{E} - \text{step} \quad &: \quad Q(\theta, \theta^{(k-1)}) = E_{\theta^{(k-1)}}\left[l_c(\theta)|D, \theta^{(k-1)}\right] \\ \text{M} - \text{step} \quad &: \quad \theta^{(k)} = \arg\max_{\theta} Q(\theta, \theta^{(k-1)}) + \log p(\theta) \end{aligned}$$

Equation 8.8 EM algorithm

To derive the EM algorithm for GMM, we first need to compute the expected complete data log-likelihood:

$$
\begin{aligned}
Q(\theta, \theta^{(k-1)}) &= E\left[\sum_i \log p(x_i, z_i|\theta)\right] \\
&= \sum_i E\left[\log\left[\prod_{k=1}^{K}(\pi_k p(x_i|\theta_k))^{1[z_i=k]}\right]\right] \\
&= \sum_i \sum_k E\big[1[z_i=k]\big] \log\big[\pi_k p(x_i|\theta_k)\big] \\
&= \sum_i \sum_k p(z_i=k|x_i, \theta^{(k-1)}) \log\big[\pi_k p(x_i|\theta_k)\big] \\
&= \sum_i \sum_k r_{ik} \log \pi_k + \sum_i \sum_k r_{ik} \log p(x_i|\theta_k)
\end{aligned}
$$

Equation 8.8 Derivation of Q for GMM

where $r_{ik}=p(z_i=k|x_i,\theta^{(k-1)})$ is the soft assignment of point x_i to cluster k.

E-step. Given $\theta^{(k-1)}$, we want to compute the soft assignments:

$$
r_{ik} = p(z_i=k|x_i,\theta^{(k-1)}) = \frac{p(z_i=k, x_i|\theta^{(k-1)})}{\sum_{k=1}^{K} p(z_i=k, x_i|\theta^{(k-1)})} = \frac{p(x_i|z_i=k,\theta^{(k-1)})\pi_k}{\sum_{k=1}^{K} p(x_i|z_i=k,\theta^{(k-1)})\pi_k}
$$

Equation 8.9 E-step for GMM

where $\pi_k=p(z_i=k)$ are the mixture proportions.

M-step. In the M step, we maximize Q with respect to model parameters π and θ_k. First, let's find π that maximizes the Lagrangian:

$$
\frac{\partial Q}{\partial \pi_k} = \frac{\partial}{\partial \pi_k}\left[\sum_i \sum_k r_{ik} \log \pi_k + \lambda(1 - \sum_k \pi_k)\right] = \sum_i r_{ik} \frac{1}{\pi_k} - \lambda = 0
$$

Equation 8.10 Lagrangian

Substituting the above expression into the constraint, we get

$$\sum_k \pi_k = 1 \implies \sum_k \frac{1}{\lambda} \sum_i r_{ik} = 1 \implies \lambda = \sum_i \sum_k r_{ik} = \sum_i 1 = N$$

Equation 8.11 Solving for lambda

Therefore, `πk = 1/λ ∑rik = 1/N ∑rik`. To find the optimum parameters `θk={μk,Σk}`, we want to optimize the terms of `Q` that depend on `θk`:

$$\begin{aligned} l(\mu_k, \Sigma_k) &= \sum_i r_{ik} \log p(x_i|\theta_k) \\ &\propto -\frac{1}{2} \sum_i r_{ik} \left[\log |\Sigma_k| + (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) \right] \end{aligned}$$

Equation 8.12 Terms of `Q` that depend on `θk`

To find the optimimum `μk`, we differentiate the above expression. First, focusing on the second term inside the sum, we can make a substitution `yi = xi-μk`

$$\frac{\partial}{\partial \mu_k} (x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k) = \frac{\partial}{\partial y_i} y_i^T \Sigma^{-1} y_i \frac{\partial y_i}{\partial \mu_k} = -1 \times (\Sigma^{-1} + \Sigma^{-T}) y_i$$

Equation 8.13 Derivative wrt `μk`

Substituting the above expression, we get:

$$\frac{\partial}{\partial \mu_k} l(\mu_k, \Sigma_k) \propto -\frac{1}{2} \sum_i r_{ik} \left[-2\Sigma^{-1}(x_i - \mu_k) \right] = \Sigma^{-1} \sum_i r_{ik} (x_i - \mu_k) = 0$$

Equation 8.14 Derivative wrt `μk`

which implies that

$$\sum_i r_{ik} (x_i - \mu_k) = 0 \implies \mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}$$

Equation 8.15 Optimal `μk`

To compute optimum `Σk`, we can use the trace identity:

$$x^T Ax = \mathrm{tr}(x^T Ax) = \mathrm{tr}(xx^T A) = \mathrm{tr}(Axx^T)$$

Equation 8.16 Trace Identity

Using $\Lambda=\Sigma^{-1}$ notation, we have:

$$\begin{aligned} l(\Lambda) &\propto -\frac{1}{2}\sum_i r_{ik}\log|\Lambda| - \frac{1}{2}\sum_i r_{ik}\mathrm{tr}\left[(x_i-\mu)(x_i-\mu)^T\Lambda\right] \\ &= -\frac{1}{2}\sum_i r_{ik}\log|\Lambda| - \frac{1}{2}\mathrm{tr}(S_\mu\Lambda) \end{aligned}$$

Equation 8.17 Complete Log-Likelihood

Taking matrix derivative, we get:

$$\begin{aligned} \frac{\partial l(\Lambda)}{\partial\Lambda} &= -\frac{1}{2}\sum_i r_{ik}\Lambda^{-T} - \frac{1}{2}S_\mu^T = 0 \\ \Lambda^{-1}\sum_i r_{ik} &= S_\mu^T \implies \Sigma = \frac{S_\mu^T}{\sum_i r_{ik}} \\ \Sigma &= \frac{\sum_i r_{ik}(x_i-\mu_k)(x_i-\mu_k)^T}{\sum_i r_{ik}} = \frac{\sum_i r_{ik}x_i x_i^T}{\sum_i r_{ik}} - \mu_k\mu_k^T \end{aligned}$$

Equation 8.18 Derivative wrt to inverse covariance

These equations make intuitive sense, the mean of cluster `k` is a weighted by `rik` average of all points assigned to cluster `k`, while the covariance is the weighted empirical scatter matrix. We are now ready to implement EM algorithm for GMM from scratch! Let's start by looking at the following pseudo-code.

1: **class** GMM
2: **function** gmm_em(X, k):
3: $\pi_{init} = 1/K$
4: $\mu_{init} = \text{KMeans}(X, k)$ ← Initialize with K-means
5: $\Sigma_{init} = I_{d \times d}$
6: **for** iter=1,2,...,max_iter
7: $r_{ik} = \text{estep}(\pi_k, \mu_k, \Sigma_k, X)$ ← Compute responsibilities
8: $\pi_k, \mu_k, \Sigma_k = \text{mstep}(r_{ik}, X)$ ← Compute model parameters
9: **end for**
10: **return** π_k, μ_k, Σ_k
11: **function** estep$(\pi_k, \mu_k, \Sigma_k, X)$:
12: $r_{ik} = \frac{N(x_i|\mu_k, \Sigma_k)\pi_k}{\sum_{k=1}^{K} N(x_i|\mu_k, \Sigma_k)\pi_k}$
13: **return** r_{ik}
14: **function** mstep(r_{ik}, X):
15: $\pi_k = \frac{1}{N} \sum_i r_{ik}$
16: $\mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}$
17: $\Sigma_k = \frac{\sum_i r_{ik} x_i x_i^T}{\sum_i r_{ik}} - \mu_k \mu_k^T$
18: **return** π_k, μ_k, Σ_k

The `GMM` class consists of three main functions: `gmm_em`, `estep` and `mstep`. In `gmm_em`, we initialize the means using the K-means algorithm and initialize covariances as identity matrices. Next, we iterate between the `estep` that computes responsibilities (aka soft assignments) of data point `i` to each of the `k` clusters and the `mstep` that takes the responsibilities as input and computes model parameters: mean, covariance and mixture proportions for each cluster in the Gaussian Mixture Model. The code in the `estep` and `mstep` functions follows the derivations in text. In the following code listing, we will use a synthetic dataset, in order to compare our ground truth GMM parameters with parameters inferred by the EM algorithm.

Listing 8.2 Expecation-Maximization (EM) for Gaussian Mixture Models (GMM)

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn.cluster import KMeans
from scipy.stats import multivariate_normal
```

```
from scipy.special import logsumexp
from scipy import linalg

np.random.seed(3)

class GMM:

    def __init__(self, n=1e3, d=2, K=4):
        self.n = int(n)  #A
        self.d = d  #B
        self.K = K  #C

        self.X = np.zeros((self.n, self.d))

        self.mu = np.zeros((self.d, self.K))
        self.sigma = np.zeros((self.d, self.d, self.K))
        self.pik = np.ones(self.K)/K

    def generate_data(self):    #D
        alpha0 = np.ones(self.K)
        pi = np.random.dirichlet(alpha0)

        #ground truth mu and sigma
        mu0 = np.random.randint(0, 10, size=(self.d, self.K)) - 5*np.ones((self.d, self.K))
        V0 = np.zeros((self.d, self.d, self.K))
        for k in range(self.K):
            eigen_mean = 0
            Q = np.random.normal(loc=0, scale=1, size=(self.d, self.d))
            D = np.diag(abs(eigen_mean + np.random.normal(loc=0, scale=1, size=self.d)))
            V0[:,:,k] = abs(np.transpose(Q)*D*Q)

        #sample data
        for i in range(self.n):
            z = np.random.multinomial(1,pi)
            k = np.nonzero(z)[0][0]
            self.X[i,:] = np.random.multivariate_normal(mean=mu0[:,k], cov=V0[:,:,k],
       size=1)

        plt.figure()
        plt.scatter(self.X[:,0], self.X[:,1], color='b', alpha=0.5)
        plt.title("Ground Truth Data"); plt.xlabel("X1"); plt.ylabel("X2")
        plt.show()

        return mu0, V0

    def gmm_em(self):

        kmeans = KMeans(n_clusters=self.K, random_state=42).fit(self.X)   #E
        self.mu = np.transpose(kmeans.cluster_centers_)

        #init sigma
        for k in range(self.K):
            self.sigma[:,:,k] = np.eye(self.d)

        #EM algorithm
        max_iter = 10
        tol = 1e-5
        obj = np.zeros(max_iter)
        for iter in range(max_iter):
```

```
            print("EM iter ", iter)
            #E-step
            resp, llh = self.estep()    #F
            #M-step
            self.mstep(resp)     #G
            #check convergence
            obj[iter] = llh
            if (iter > 1 and obj[iter] - obj[iter-1] < tol*abs(obj[iter])):
                break
            #end if
        #end for
        plt.figure()
        plt.plot(obj)
        plt.title('EM-GMM objective'); plt.xlabel("iter"); plt.ylabel("log-likelihood")
        plt.show()

    def estep(self):

        log_r = np.zeros((self.n, self.K))
        for k in range(self.K):
            log_r[:,k] = multivariate_normal.logpdf(self.X, mean=self.mu[:,k],
       cov=self.sigma[:,:,k])
        #end for
        log_r = log_r + np.log(self.pik)
        L = logsumexp(log_r, axis=1)
        llh = np.sum(L)/self.n  #log likelihood
        log_r = log_r - L.reshape(-1,1) #normalize
        resp = np.exp(log_r)
        return resp, llh

    def mstep(self, resp):

        nk = np.sum(resp, axis=0)
        self.pik = nk/self.n
        sqrt_resp = np.sqrt(resp)
        for k in range(self.K):
            #update mu
            rx = np.multiply(resp[:,k].reshape(-1,1), self.X)
            self.mu[:,k] = np.sum(rx, axis=0) / nk[k]

            #update sigma
            Xm = self.X - self.mu[:,k]
            Xm = np.multiply(sqrt_resp[:,k].reshape(-1,1), Xm)
            self.sigma[:,:,k] = np.maximum(0, np.dot(np.transpose(Xm), Xm) / nk[k] + 1e-5 *
       np.eye(self.d))
        #end for

if __name__ == '__main__':

    gmm = GMM()
    mu0, V0 = gmm.generate_data()
    gmm.gmm_em()


    for k in range(mu0.shape[1]):
        print("cluster ", k)
        print("-----------")
        print("ground truth means:")
        print(mu0[:,k])
```

```
        print("ground truth covariance:")
        print(V0[:,:,k])
    #end for

    for k in range(mu0.shape[1]):
        print("cluster ", k)
        print("-----------")
        print("GMM-EM means:")
        print(gmm.mu[:,k])
        print("GMM-EM covariance:")
        print(gmm.sigma[:,:,k])

      plt.figure()
    ax = plt.axes()
    plt.scatter(gmm.X[:,0], gmm.X[:,1], color='b', alpha=0.5)

    for k in range(mu0.shape[1]):

        v, w = linalg.eigh(gmm.sigma[:,:,k])
        v = 2.0 * np.sqrt(2.0) * np.sqrt(v)
        u = w[0] / linalg.norm(w[0])

        # plot an ellipse to show the Gaussian component
        angle = np.arctan(u[1] / u[0])
        angle = 180.0 * angle / np.pi  # convert to degrees
        ell = mpl.patches.Ellipse(gmm.mu[:,k], v[0], v[1], 180.0 + angle, color='r',
       alpha=0.5)
        ax.add_patch(ell)

        # plot cluster centroids
        plt.scatter(gmm.mu[0,k], gmm.mu[1,k], s=80, marker='x', color='k', alpha=1)
    plt.title("Gaussian Mixture Model"); plt.xlabel("X1"); plt.ylabel("X2")
       plt.show()
```

#A number of data points
#B data dimension
#C number of clusters
#D GMM generative model
#E init mu with k-means
#F E step
#G M step

As we can see from the output, the cluster means and covariances match closely to the ground truth.

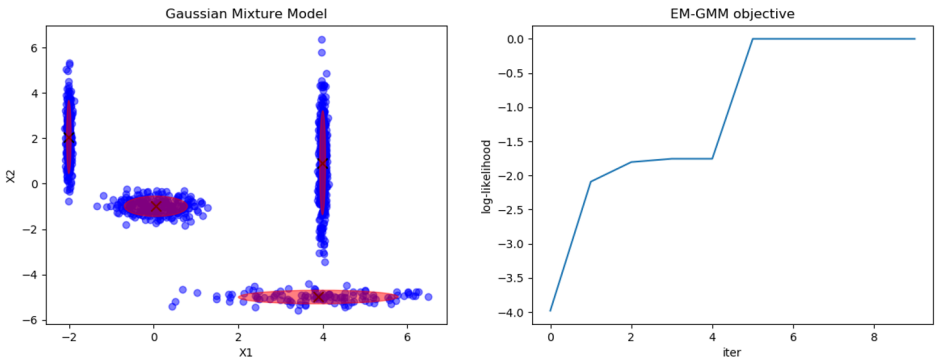


Figure 8.3 EM-GMM clustering results (left) and log-likelihood objective function (right)

Figure 8.3 (left) shows the inferred Gaussian mixture overlayed with the data. We see that the Gaussian ellipses closely fit the data. This can also be confirmed by a monotonic increase of the log-likelihood objective in Figure 8.3 (right). In the next section we are going to look at two popular dimensionality reduction techniques: Principal Component Analysis (PCA) and T-distributed Stochastic Neighbor Embedding (t-SNE).

8.3 Dimensionality Reduction

It is often useful to project high-dimensional data $x \in \mathbb{R}^D$ onto lower-dimensional subspace $z \in \mathbb{R}^L$ in a way that preserves the unique characteristics of the data. In other words, it is desirable to capture the essence of the data in the low dimensional projection. For example, if you trained word embeddings on Wikipedia corpus and you are trying to understand the relationships between different word vectors, it is much easier to visualize the relationships in two dimensions by means of dimensionality reduction. In this section, we are going to look at two ML techniques that help accomplish that: principal component analysis (PCA) and t-SNE.

8.3.1 Principal Component Analysis

In principal component analysis, we would like to project our data $x \in \mathbb{R}^D$ onto $z \in \mathbb{R}^L$ with $L<D$ such that the variance of the projected data is maximized. We can measure the quality of our projection as a reconstruction error:

$$E = \frac{1}{N}\sum_{i=1}^{N}||x_i - \hat{x}_i||^2 = ||X - WZ||_F^2$$

Equation 8.19 Reconstruction Error

where \hat{xi} is the reconstructed (lifted to higher dimensional space) vector zi and W is an orthonormal matrix. In other words, if our PCA projection is zi=W^{T}xi, then \hat{xi}= Wzi. We will show that the optimal projection matrix W (one that maximizes the variance of projected data) is equal to a matrix of L eigenvectors corresponding to the largest eigenvalues of the empiricial covariance matrix \hat{Σ} = 1/N ∑ xi xi^T.

We can write down the variance of the projected data as follows:

$$\frac{1}{N}\sum_{i=1}^{N} z_i^2 = \frac{1}{N}\sum_{i=1}^{N} w^T x x^T w = w^T \hat{\Sigma} w$$

Equation 8.20 Variance of Projected Data

we would like to maximize the quantity above subject to orthonormal constraint, i.e. ||w||$_2$ = 1. We can write down the Lagrangian as follows:

$$\max J(w) = w^T \hat{\Sigma} w + \lambda(w^T w - 1)$$

Equation 8.21 PCA objective function

Taking the derivative and setting it to zero, gives us the following expression:

$$\begin{aligned} \frac{\partial}{\partial w} J(w) &= 2\hat{\Sigma} w - 2\lambda w = 0 \\ \hat{\Sigma} w &= \lambda w \end{aligned}$$

Equation 8.21 Optimum w

Thus, the direction that maximizes the variance is the eigenvector of the covariance matrix. Left multiplying by w and using orthonormality constraint, we get w^T\hat{Σ}w =λ. Therefore, to maximize the variance for the first principal component, we want to choose an eigenvector that corresponds to the largest eigenvalue. We can repeat the above procedure by subtracting the first principal component from xi and we'll discover that \hat{Σ}w2 = λw2. By induction, we can show that the PCA matrix W$_{DL}$ consists of L eigenvectors corresponding to largest eigenvalues of the empirical covariance matrix \hat{Σ}. We are now ready to implement the PCA algorithm from scratch! The following pseudo-code summarizes the algorithm.

1: **class** PCA
2: **function** transform(X, K):
3: Σ =covariance_matrix(X) ← Compute empirical covariance
4: $V, \lambda = \text{eig}(\Sigma)$ ← Eigen value decomposition
5: idx = argsort(λ) ← Sort from largest to smallest
6: $V_{pca} = V[\text{idx}][:\text{K}]$
7: $\lambda_{pca} = \lambda[\text{idx}][:\text{K}]$
Select top K eigen vectors and eigen values
8: $X_{pca} = XV_{pca}$ ← Project data onto principal components
9: **return** X_{pca}

The main function in the `PCA` class is called `transform`. We first compute the empirical covariance matrix from high dimensional data matrix `X`. Next, we compute the eigen value decomposition of the covariance matrix. We sort the eigenvalues from largest to smallest and use the sorted index to select top K largest eigen values and their corresponding eigen vectors. Finally, we compute the PCA representation of our data by multiplying the data matrix with the matrix of top K eigenvalues. In the following code listing, we project a random `d`-dimensional matrix onto `2` principal components.

Listing 8.3 Principal Component Analysis (PCA)

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

class PCA():

    def __init__(self, n_components = 2):
        self.n_components = n_components

    def covariance_matrix(self, X, Y=None):
        if Y is None:
            Y = X
        n_samples = np.shape(X)[0]
        covariance_matrix = (1 / (n_samples-1)) * (X - X.mean(axis=0)).T.dot(Y -
       Y.mean(axis=0))
        return covariance_matrix

    def transform(self, X):
        Sigma = self.covariance_matrix(X)
        eig_vals, eig_vecs = np.linalg.eig(Sigma)

        idx = eig_vals.argsort()[::-1]    #A
        eig_vals = eig_vals[idx][:self.n_components]
```

```
        eig_vecs = np.atleast_1d(eig_vecs[:,idx])[:, :self.n_components]

        X_transformed = X.dot(eig_vecs)   #B

        return X_transformed

if __name__ == "__main__":

    n = 20
    d = 5
    X = np.random.rand(n,d)

    pca = PCA(n_components=2)
    X_pca = pca.transform(X)

    print(X_pca)

    plt.figure()
    plt.scatter(X_pca[:,0], X_pca[:,1], color='b', alpha=0.5)
    plt.title("Principal Component Analysis"); plt.xlabel("X1"); plt.ylabel("X2")
    plt.show()
```

#A sort from largest to smallest eigenvalue
#B project the data onto principal components

Figure 8.4 shows a plot of the first 2 principal components applied to a random matrix.

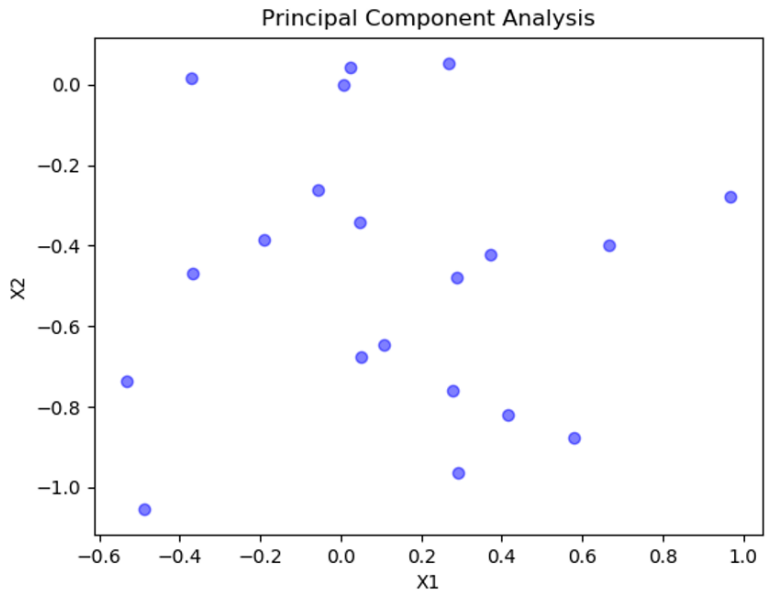


Figure 8.4 PCA projection of a random matrix

Recall that in PCA, the variance of the projected data is maximized, and therefore, we can discover trends and patterns in the projected data.

8.3.2 t-SNE Manifold Learning on Images

Images are high dimensional objects that live on manifolds. A manifold is a topological space that locally resembles Euclidean space. We can visualize high dimensional objects with the help of an embedding. We consider two such embeddings: t-SNE and Isomap on MNIST digits dataset.

Listing 8.4 t-SNE Manifold on MNIST digits dataset

```
import numpy as np
import matplotlib.pyplot as plt

from time import time
from sklearn import manifold

from sklearn.datasets import load_digits
from sklearn.neighbors import KDTree

def plot_digits(X):

    n_img_per_row = np.amin((20, np.int(np.sqrt(X.shape[0]))))
    img = np.zeros((10 * n_img_per_row, 10 * n_img_per_row))
    for i in range(n_img_per_row):
        ix = 10 * i + 1
        for j in range(n_img_per_row):
            iy = 10 * j + 1
            img[ix:ix + 8, iy:iy + 8] = X[i * n_img_per_row + j].reshape((8, 8))

    plt.figure()
    plt.imshow(img, cmap=plt.cm.binary)
    plt.xticks([])
    plt.yticks([])
    plt.title('A selection from the 64-dimensional digits dataset')

def mnist_manifold():

    digits = load_digits()

    X = digits.data
    y = digits.target

    num_classes = np.unique(y).shape[0]

    plot_digits(X)

    #TSNE
    #Barnes-Hut: O(d NlogN) where d is dim and N is the number of samples
    #Exact: O(d N^2)
    t0 = time()
    tsne = manifold.TSNE(n_components = 2, init = 'pca', method = 'barnes_hut', verbose =
       1)
    X_tsne = tsne.fit_transform(X)
    t1 = time()
    print('t-SNE: %.2f sec' %(t1-t0))
```

```
    tsne.get_params()

    plt.figure()
    for k in range(num_classes):
        plt.plot(X_tsne[y==k,0], X_tsne[y==k,1],'o')
    plt.title('t-SNE embedding of digits dataset')
    plt.xlabel('X1')
    plt.ylabel('X2')
    axes = plt.gca()
    axes.set_xlim([X_tsne[:,0].min()-1,X_tsne[:,0].max()+1])
    axes.set_ylim([X_tsne[:,1].min()-1,X_tsne[:,1].max()+1])
    plt.show()

    #ISOMAP
    #1. Nearest neighbors search: O(d log k N log N)
    #2. Shortest path graph search: O(N^2(k+log(N))
    #3. Partial eigenvalue decomposition: O(dN^2)

    t0 = time()
    isomap = manifold.Isomap(n_neighbors = 5, n_components = 2)
    X_isomap = isomap.fit_transform(X)
    t1 = time()
    print('Isomap: %.2f sec' %(t1-t0))
    isomap.get_params()

    plt.figure()
    for k in range(num_classes):
        plt.plot(X_isomap[y==k,0], X_isomap[y==k,1], 'o', label=str(k), linewidth = 2)
    plt.title('Isomap embedding of the digits dataset')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.show()

    #Use KD-tree to find k-nearest neighbors to a query image
    kdt = KDTree(X_isomap)
    Q = np.array([[-160, -30],[-102, 14]])
    kdt_dist, kdt_idx = kdt.query(Q,k=20)
    plot_digits(X[kdt_idx.ravel(),:])

if __name__ == "__main__":
    mnist_manifold()
```

Figure 8.5 shows a t-SNE embedding with 10 clusters in 2-D, where each cluster corresponds to a digit 0-9.

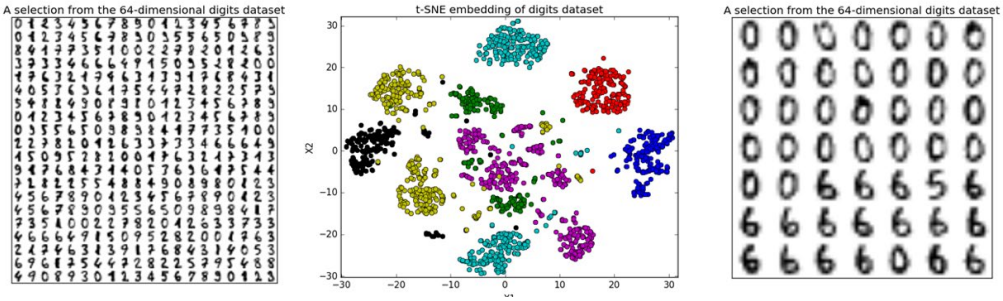


Figure 8.5 t-SNE embedding showing 10 clusters in 2-D, where each cluster corresponds to a digit 0-9.

We can see that without labels, we are able to discover 10 clusters using t-SNE embedding in the 2-dimensional space. Moreover, we expect adjacent clusters of digits to be similar to each other. The figure on the right shows sample digits from two adjacent clusters (digit 0 and digit 6). We can visualize adjacent clusters by constructing a KD-tree to find K nearest neighbors to a query point.

It's important to be aware of some of the pitfalls of t-SNE. For example, t-SNE results may vary based on the perplexity hyper-parameter. The t-SNE algorithm also doesn't always produce similar outputs on successive runs and there are additional hyperparameters related to the optimization process. Moreover, the cluster sizes (as measured by standard deviation) and distances between clusters might not mean anything. Thus, the high flexibility of t-SNE also makes the results of the algorithm tricky to interpret.

8.4 Exercises

8.1 Show that the Dirichlet distribution $\mathrm{Dir}(\theta|\alpha)$ is a conjugate prior to the Multinomial likelihood by computing the posterior. How does the shape of the posterior vary as a function of the posterior counts?

8.2 Explain the principle behind k-means++ initialization

8.3 Prove the cyclic permutation property of the trace: $\mathrm{tr}(ABC) = \mathrm{tr}(BCA) = \mathrm{tr}(CAB)$

8.4 Compute the run-time of the Principal Component Analysis (PCA) algorithm

8.5 Summary

- Unsupervised learning takes place when no training labels are available.
- Dirichlet Process (DP) K-means is a Bayesian non-parametric extension of the K-means algorithm in which the number of clusters grows with data.
- Gaussian Mixture Models (GMMs) are commonly used to model complex density distributions. GMM parameters (means, covariances and mixture proportions) can be inferred using Expectation-Maximization (EM) algorithm.
- Expectation-Maximization (EM) is an iterative algorithm that alternates between inferring the missing values given the parameters (E step) and then optimizing the parameters given filled in data (M step).
- In dimensionality reduction, we project high dimensional data into a lower dimensional sub-space in a way that preserves unique characteristics of the data.
- In Principal Component Analysis (PCA) algorithm, the variance of the projected data is maximized.
- Common pitfalls of t-SNE algorithm include variability of results due to perplexity hyper-parameter, dissimilar outputs on successive runs, lack of interpretability of cluster sizes and distances between clusters.

9

Selected Unsupervised Learning Algorithms

This chapter covers

- Latent Dirichlet Allocation for Topic Discovery
- Density Estimators in Computational Biology and Finance
- Structure Learning for Relational Data
- Simulated Annealing for Energy Minimization
- Genetic Algorithm in Evolutionary Biology
- ML Research: Unsupervised Learning

In the previous chapter, we looked at unsupervised ML algorithms to help learn patterns in our data. This chapter continues the discussion based on selected unsupervised learning algorithms. The algorithm presented in this chapter are selected to cover the breadth of unsupervised learning and they are important to learn because they cover a range of applications from computational biology to physics and finance. We'll start by looking at Latent Dirichlet Allocation (LDA) for learning topic models, followed by density estimators and structure learning algorithms and concluding with simulated annealing (SA) and genetic algorithm (GA).

9.1 Latent Dirichlet Allocation

A topic model is a latent variable model for discrete data such as text documents. Latent Dirichlet Allocation (LDA) is a topic model that represents each document as a finite mixture of topics, where a topic is a distribution over words. The objective is to learn the shared topic distribution and topic proportions for each document. LDA assumes a bag of words model in which the words are exchangeable and as a result sentence structure is not preserved, i.e.

only the word counts matter. Thus, each document is reduced to a vector of counts over the vocabulary `V` and the entire corpus of `D` documents is summarized in a *term-document* matrix A_{VD}. LDA can be seen as a non-negative matrix factorization problem that takes the term-document matrix and factorizes it into a product of topics W_{VK} and topic proportions H_{KD}: `A=WH`.

A common method for adjusting the word counts is *tf-idf* that logarithmically drives down to zero word counts that occur frequently across documents: `A log(D/nt)`, where `D` is the total number of documents in the corpus and `nt` is the number of documents where term `t` appears. The tf-idf smoothing of word counts identifies the sets of words that are discriminative for documents and leads to better model performance. The term-document matrix generalizes from counts of individual words (unigrams) to larger structural units such as n-grams. In the case of n-grams different smoothing of word counts techniques (such as Laplace smoothing) are used to address the lack of observations in a very large feature space.

Figure 9.1 shows the graphical model for the Latent Dirichlet Allocation (LDA).

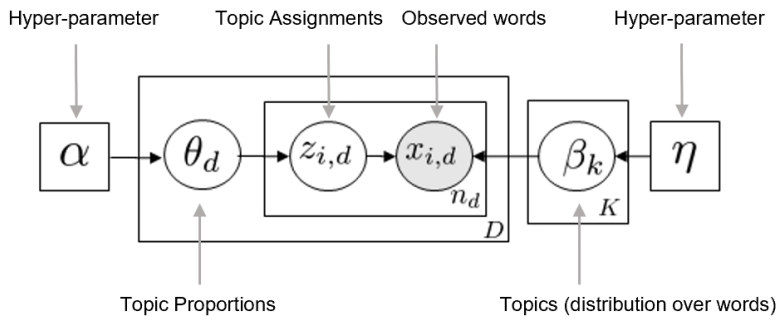


Figure 9.1 Latent Dirichlet Allocation (LDA) graphical model

The LDA topic model associates each word `xid` with a topic label `zid` $\in$ `{1,2,...,K}`. Each document is associated with topic proportions `θd` that could be used to measure document similarity. The topics `βk` are shared across all documents. The hyper-parameters `α` and `η` capture our prior knowledge of topic proportions and topics, respectively, e.g. from past on-line training of the model. The full generative model can be specified as follows:

$$\begin{aligned} \theta_d|\alpha &\sim \mathrm{Dir}(\alpha) \\ z_{i,d}|\theta_d &\sim \mathrm{Cat}(\theta_d) \\ \beta_k|\eta &\sim \mathrm{Dir}(\eta) \\ x_{i,d}|z_{i,d}=k,\beta &\sim \mathrm{Cat}(\beta_k) \end{aligned}$$

Equation 9.1 LDA generative model

The joint distribution for a single document `d` can be written as follows (as in D. Blei et al, "Latent Dirichlet Allocation", Journal of Machine Learning Research, 2003):

$$p(x,z,\theta,|\alpha,\beta) = p(\theta_d|\alpha)\prod_{i=1}^{n_d} p(z_{i,d}|\theta_d)p(x_{i,d}|z_{i,d},\beta)$$

Equation 9.2 Full joint distribution for LDA

The parameters `α` and `β` are corpus-level parameters, the variable `θd` is sampled once every document, while `zid` and `xid` are word-level variables sampled once for each word in each document. Unlike a multinomial clustering model where each document is associated with a single topic, LDA represents each document as a mixture of topics.

9.1.1 Variational Bayes

The key inference problem that we need to solve in order to use LDA is that of computing the posterior distribution of the latent variables for a given document: `p(θ,z|x,α,β)`. The posterior can be approximated with the following variational distribution:

$$q(\theta,z|\gamma,\phi) = q(\theta|\gamma)\prod_{i=1}^{n} q(z_i|\phi_i)$$

Equation 9.3 Variational Approximation of LDA Posterior Distribution

The variational parameters are optimized to maximize the Evidence Lower BOund (ELBO). Recall from Chapter 3, the definition of ELBO as a difference between the energy term and the entropy term:

$$\log p(x|\alpha,\eta) \geq L(x,\phi,\gamma,\lambda) = E_q[\log p(x,z,\theta,\beta|\alpha,\eta)] - E_q[\log q(z,\theta,\beta)]$$

Equation 9.4 ELBO objective function for LDA

We choose a fully factored distribution `q` of the form:

$$q(z_{id}=k) = \phi_{dwk}; \quad q(\theta_d) \sim \mathrm{Dir}(\theta_d|\gamma_d); \quad q(\beta_k) \sim \mathrm{Dir}(\beta_k|\lambda_k)$$

Equation 9.5 Approximating distributions

We can expand the lower bound by using the factorizations of `p` and `q` (following D. Blei et al, "Latent Dirichlet Allocation", Journal of Machine Learning Research, 2003):

$$L(\gamma,\phi;\alpha,\beta) = E_q[\log p(\theta|\alpha)] + E_q[\log p(z|\theta)] + E_q[\log p(x|z,\beta)] - E_q[\log q(\theta)] - E_q[\log q(z)]$$

Equation 9.10 ELBO objective function for LDA

Each of the five terms in `L(γ,ϕ;α,β)` can be expanded as follows:

$$\begin{aligned}
L(\gamma,\phi;\alpha,\beta) &= \log\Gamma(\sum_{j=1}^{k}\alpha_j) - \sum_{i=1}^{k}\log\Gamma(\alpha_i) + \sum_{i=1}^{k}(\alpha_i-1)(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{k}\gamma_j)) \\
&+ \sum_{n=1}^{N}\sum_{i=1}^{k}\phi_{ni}(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{k}\gamma_j)) \\
&+ \sum_{n=1}^{N}\sum_{i=1}^{k}\sum_{j=1}^{V}\phi_{ni}x_n^j\log\beta_{ij} \\
&- \log\Gamma(\sum_{j=1}^{k}\gamma_j) + \sum_{i=1}^{k}\log\Gamma(\gamma_i) - \sum_{i=1}^{k}(\gamma_i-1)(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{k}\gamma_j)) \\
&- \sum_{n=1}^{N}\sum_{i=1}^{k}\phi_{ni}\log\phi_{ni}
\end{aligned}$$

Equation 9.11 ELBO objective function for LDA

Where `Ψ(x)=d/dx log Γ(x)` is the digamma function. `L(γ,ϕ;α,β)` can be maximized using coordinate ascent over the variational parameters `ϕ, γ, λ`.

$$\begin{aligned}\phi_{dwk} &\propto \exp\{E_q[\log \theta_{dk}] + E_q[\log \beta_{kw}]\} \\ \gamma_{dk} &= \alpha + \sum_w n_{dw}\phi_{dwk} \\ \lambda_{kw} &= \eta + \sum_d n_{dw}\phi_{dwk}\end{aligned}$$

Equation 9.12: Variational Parameter Updates

where the expectations under `q` of `log θ` and `log β` are:

$$E_q[\log \theta_{dk}] = \Psi(\gamma_{dk}) - \Psi(\sum_{i=1}^{K} \gamma_{di}) \quad E_q[\log \beta_{kw}] = \Psi(\lambda_{kw}) - \Psi(\sum_{i=1}^{W} \lambda_{ki})$$

Equation 9.13: Expectations Expressions

The variational parameter updates can be used in an online setting that does not require a full pass through the entire corpus at each iteration. An online update of variational parameters enables topic analysis for very large datasets including streaming data. Online VB for LDA is described in the algorithm below.

1: Define $\rho_t = (\tau_0 + t)^{-\kappa}$
2: Initialize λ randomly
3: for t = 1 to ∞ do
4: *E step:*
5: Initialize $\gamma_{tk} = 1$
6: **repeat**
7: Set $\phi_{twk} \propto \exp\{E_q[\log \theta_{tk}] + E_q[\log \beta_{kw}]\}$ ← Topic Assignments
8: Set $\gamma_{tk} = \alpha + \sum_w \phi_{twk} n_{tw}$ ← Topic Proportions
9: **until** $\frac{1}{K}\sum_k |\Delta\gamma_{tk}| < \epsilon$
10: *M step:*
11: Compute $\tilde{\lambda}_{kw} = \eta + D n_{tw}\phi_{twk}$
12: Set $\lambda = (1 - \rho_t)\lambda + \rho_t\tilde{\lambda}$
13: end for

As the t-th vector of word counts `nt` is observed, we perform an E step to find locally optimal values of `γt` and `φt`, holding `λ` fixed. We then compute `\tilde{λ}` that would be

optimal if our entire corpus consisted of the single document n repeated D times. We then update λ as a weighted average of its previous value and $\tilde{\lambda}$, where the weight is given by the learning parameter ρt for $\kappa \in (0.5,1]$, controlling the rate at which old values of $\tilde{\lambda}$ are forgotten. We are now ready to implement variational Bayes for LDA from scratch!

Listing 9.1 Variational Bayes for Latent Dirichlet-Allocation

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from wordcloud import WordCloud
from scipy.special import digamma, gammaln

np.random.seed(12)

class LDA:
    def __init__(self, A, K):
        self.N = A.shape[0]    #A
        self.D = A.shape[1]    #B
        self.K = num_topics    #C

        self.A = A    #D

        #init word distribution beta
        self.eta = np.ones(self.N)    #E
        self.beta = np.zeros((self.N, self.K))    #F
        for k in range(self.K):
            self.beta[:,k] = np.random.dirichlet(self.eta)
            self.beta[:,k] = self.beta[:,k] + 1e-6 #to avoid zero entries
            self.beta[:,k] = self.beta[:,k]/np.sum(self.beta[:,k])
        #end for

        #init topic proportions theta and cluster assignments z
        self.alpha = np.ones(self.K)    #G
        self.z = np.zeros((self.N, self.D))    #H
        for d in range(self.D):
            theta = np.random.dirichlet(self.alpha)
            wdn_idx = np.nonzero(self.A[:,d])[0]
            for i in range(len(wdn_idx)):
                z_idx = np.argmax(np.random.multinomial(1, theta))
                self.z[wdn_idx[i],d] = z_idx  #topic id
            #end for
        #end for

        #init variational parameters
        self.gamma = np.ones((self.D, self.K))    #I
        for d in range(self.D):
            theta = np.random.dirichlet(self.alpha)
            self.gamma[d,:] = theta
        #end for

        self.lmbda = np.transpose(self.beta) #np.ones((self.K, self.N))/self.N #J

        self.phi = np.zeros((self.D, self.N, self.K))    #K
        for d in range(self.D):
```

```
            for w in range(self.N):
                theta = np.random.dirichlet(self.alpha)
                self.phi[d,w,:] = np.random.multinomial(1, theta)
            #end for
        #end for

    def variational_inference(self, var_iter=10):

        llh = np.zeros(var_iter)
        llh_delta = np.zeros(var_iter)

        for iter in range(var_iter):
            print("VI iter: ", iter)
            J_old = self.elbo_objective()
            self.mean_field_update()
            J_new = self.elbo_objective()

            llh[iter] = J_old
            llh_delta[iter] = J_new - J_old
        #end for

        #update alpha and beta
        for k in range(self.K):
            self.alpha[k] = np.sum(self.gamma[:,k])
            self.beta[:,k] = self.lmbda[k,:] / np.sum(self.lmbda[k,:])
        #end for

        #update topic assignments
        for d in range(self.D):
            wdn_idx = np.nonzero(self.A[:,d])[0]
            for i in range(len(wdn_idx)):
                z_idx = np.argmax(self.phi[d,wdn_idx[i],:])
                self.z[wdn_idx[i],d] = z_idx  #topic id
            #end for
        #end for

        plt.figure()
        plt.plot(llh); plt.title('LDA VI');
        plt.xlabel('mean field iterations'); plt.ylabel("ELBO")
        plt.show()

        return llh

    def mean_field_update(self):

        ndw = np.zeros((self.D, self.N))    #L
        for d in range(self.D):
            doc = self.A[:,d]
            wdn_idx = np.nonzero(doc)[0]

            for i in range(len(wdn_idx)):
                ndw[d,wdn_idx[i]] += 1
            #end for

            #update gamma
            for k in range(self.K):
                self.gamma[d,k] = self.alpha[k] + np.dot(ndw[d,:], self.phi[d,:,k])
            #end for
```

```
            #update phi
            for w in range(len(wdn_idx)):
                self.phi[d,wdn_idx[w],:] = np.exp(digamma(self.gamma[d,:]) -
    digamma(np.sum(self.gamma[d,:])) + digamma(self.lmbda[:,wdn_idx[w]]) -
    digamma(np.sum(self.lmbda, axis=1)))
                if (np.sum(self.phi[d,wdn_idx[w],:]) > 0): #to avoid 0/0
                    self.phi[d,wdn_idx[w],:] = self.phi[d,wdn_idx[w],:] /
    np.sum(self.phi[d,wdn_idx[w],:]) #normalize phi
                #end if
            #end for

        #end for

        #update lambda given ndw for all docs
        for k in range(self.K):
            self.lmbda[k,:] = self.eta
            for d in range(self.D):
                self.lmbda[k,:] += np.multiply(ndw[d,:], self.phi[d,:,k])
            #end for
        #end for

    def elbo_objective(self):
        #see Blei 2003

        T1_A = gammaln(np.sum(self.alpha)) - np.sum(gammaln(self.alpha))
        T1_B = 0
        for k in range(self.K):
            T1_B +=  np.dot(self.alpha[k]-1, digamma(self.gamma[:,k]) -
    digamma(np.sum(self.gamma, axis=1)))
        T1 = T1_A + T1_B

        T2 = 0
        for n in range(self.N):
            for k in range(self.K):
                T2 += self.phi[:,n,k] * (digamma(self.gamma[:,k]) -
    digamma(np.sum(self.gamma, axis=1)))

        T3 = 0
        for n in range(self.N):
            for k in range(self.K):
                T3 += self.phi[:,n,k] * np.log(self.beta[n,k])

        T4 = 0
        T4_A = -gammaln(np.sum(self.gamma, axis=1)) + np.sum(gammaln(self.gamma), axis=1)
        T4_B = 0
        for k in range(self.K):
            T4_B = -(self.gamma[:,k]-1) * (digamma(self.gamma[:,k]) -
    digamma(np.sum(self.gamma, axis=1)))
        T4 = T4_A + T4_B

        T5 = 0
        for n in range(self.N):
            for k in range(self.K):
                T5 += -np.multiply(self.phi[:,n,k], np.log(self.phi[:,n,k] + 1e-6))

        T15 = T1 + T2 + T3 + T4 + T5
        J = sum(T15)/self.D  #averaged over documents
        return J
```

```
if __name__ == "__main__":

    #LDA parameters
    num_features = 1000  #vocabulary size
    num_topics = 4      #fixed for LD

    #20 newsgroups dataset
    categories = ['sci.crypt', 'comp.graphics', 'sci.space', 'talk.religion.misc']

    newsgroups = fetch_20newsgroups(shuffle=True, random_state=42, subset='train',
                 remove=('headers', 'footers', 'quotes'), categories=categories)

    vectorizer = TfidfVectorizer(max_features = num_features, max_df=0.95, min_df=2,
       stop_words = 'english')
    dataset = vectorizer.fit_transform(newsgroups.data)
    A = np.transpose(dataset.toarray())  #term-document matrix

    lda = LDA(A=A, K=num_topics)
    llh = lda.variational_inference(var_iter=10)
    id2word = {v:k for k,v in vectorizer.vocabulary_.items()}

    #display topics
    for k in range(num_topics):
        print("topic: ", k)
        print("----------")
        topic_words = ""
        top_words = np.argsort(lda.lmbda[k,:])[-10:]
        for i in range(len(top_words)):
            topic_words += id2word[top_words[i]] + " "
            print(id2word[top_words[i]])

        wordcloud = WordCloud(width = 800, height = 800,
                    background_color ='white',
                    min_font_size = 10).generate(topic_words)

        plt.figure()
        plt.imshow(wordcloud)
        plt.axis("off")
        plt.tight_layout(pad = 0)
        plt.show()
```

#A word dictionary size
#B number of documents
#C number of topics
#D term-document matrix
#E uniform Dirichlet prior on words
#F NxK topic matrix
#G uniform Dirichlet prior on topics
#H cluster assignments z
#I topic proportions
#J word frequencies
#K assignments
#L word counts for each document

Figure 9.2 shows the increase Evidence Lower Bound (ELBO) over the number of mean field iterations.

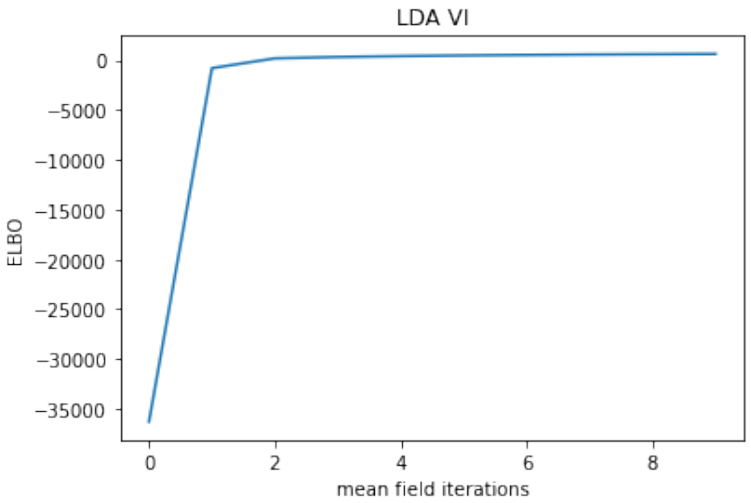


Figure 9.2 Increase in ELBO vs the number of mean-field iterations

Figure 9.3 shows the inferred topic distributions visualized as word clouds.

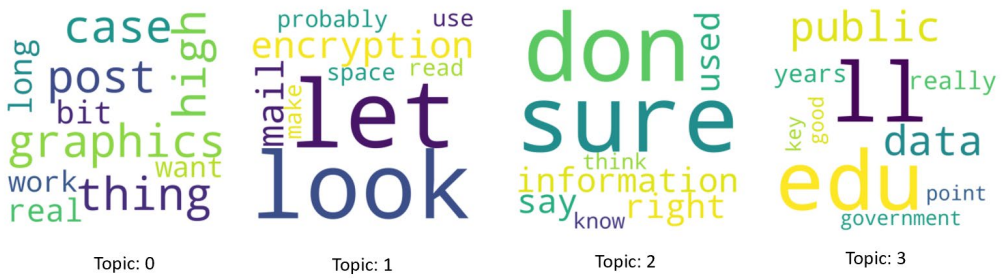


Figure 9.3 Inferred topic distributions via LDA mean field variational inference

As we can see from the output the top-K words for each topic match the categories in the 20 newsgroups dataset. In the following section, we are going to look into ways of modeling probability density of data in application to computational biology and finance.

9.2 Density Estimators

The goal of density estimation is to model the probability density of data. In this section, we are going to look at Kernel Density Estimators (KDEs) in application to computational biology and also look at how we can optimize a portfolio of stocks using tangent portfolio theory.

9.2.1. Kernel Density Estimator

An alternative approach to a K component mixture model is a Kernel Density Estimator (KDE) that allocates one cluster center per data point. KDE is application of kernel smoothing to probability density estimation that uses kernels as weights. In the case of a Gaussian kernel, we have:

$$p(x|D) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(x|x_i, \sigma^2 I)$$

Equation 9.14 Gaussian Kernel Density Estimator

Notice, that we are averaging `N` Gaussians, with each Gaussian centered at the data point `xi`. We can generalize the above expression to any kernel `K(x)`:

$$p(x|D) = \frac{1}{N}\sum_{i=1}^{N}\kappa_h(x - x_i)$$

Equation 9.15 General Kernel Density Estimator

The advantage of KDEs over parametric models such as density mixtures is that no model fitting is required (except for fine-tuning the bandwidth parameter `h`) and there is no need to pick the number of mixtures `K`. The disadvantage is that the model takes a lot of memory to store and a lot of time to evaluate. In other words, KDE is suitable when an accurate density estimate is required for a relatively small dataset (small number of points `N`). Let's look at an example that analyzes RNA-seq data in order to estimate the flux of a T7 promoter.

Listing 9.2 Kernel Density Estimate

```
import numpy as np
import matplotlib.pyplot as plt

class KDE():

    def __init__(self):
        #Histogram and Gaussian Kernel Estimator used to
        #analyze RNA-seq data for flux estimation of a T7 promoter
        self.G = 1e9    #A
        self.C = 1e3    #B
        self.L = 100    #C
        self.N = 1e6    #D
        self.M = 1e4    #E
        self.LN = 1000    #F
        self.FDR = 0.05    #G

        #uniform sampling (poisson model)
        self.lmbda = (self.N * self.L) / self.G    #H
        self.C_est = self.M/(1-np.exp(-self.lmbda))    #I
        self.C_cvrg = self.G - self.G * np.exp(-self.lmbda)    #J
        self.N_gaps = self.N * np.exp(-self.lmbda)   #K

        #gamma prior sampling (negative binomial model)
        #X = "number of failures before rth success"
        self.k = 0.5    #L
        self.p = self.lmbda/(self.lmbda + 1/self.k)    #M
        self.r = 1/self.k    #N

        #RNAP binding data (RNA-seq)
        self.data = np.random.negative_binomial(self.r, self.p, size=self.LN)

    def histogram(self):
        self.bin_delta = 1    #O
        self.bin_range = np.arange(1, np.max(self.data), self.bin_delta)
        self.bin_counts, _ = np.histogram(self.data, bins=self.bin_range)

        #histogram density estimation
        #P = integral_R p(x) dx, where X is in R^3
        #p(x) = K/(NxV), where K=number of points in region R
        #N=total number of points, V=volume of region R

        rnap_density_est = self.bin_counts/(sum(self.bin_counts) * self.bin_delta)
        return rnap_density_est

    def kernel(self):
        #Gaussian kernel density estimator with smoothing parameter h
        #sum N Guassians centered at each data point, parameterized by common std dev h

        x_dim = 1    #P
        h = 10    #Q

        rnap_density_support = np.arange(np.max(self.data))
        rnap_density_est = 0
        for i in range(np.sum(self.bin_counts)):
            rnap_density_est += (1/(2*np.pi*h**2)**(x_dim/2.0))*np.exp(-
       (rnap_density_support - self.data[i])**2 / (2.0*h**2))
        #end for
```

```
        rnap_density_est = rnap_density_est / np.sum(rnap_density_est)
        return rnap_density_est

if __name__ == "__main__":

    kde = KDE()
    est1 = kde.histogram()
    est2 = kde.kernel()

    plt.figure()
    plt.plot(est1, c='b', label='histogram')
    plt.plot(est2, c='r', label='gaussian kernel')
    plt.title("RNA-seq density estimate based on negative binomial sampling model")
    plt.xlabel("read length, [base pairs]"); plt.ylabel("density"); plt.legend()
    plt.show()
```

#A length of genome in base pairs (bp)
#B number of unique molecules
#C length of a read, bp
#D number of reads, L bp long
#E number of unique read sequences, bp
#F total length of assembled / mapped RNA-seq reads
#G false discovery rate
#H expected number of bases covered
#I library size estimate
#J base coverage
#K number of gaps (uncovered bases)
#L dispersion parameter (fit to data)
#M success probability
#N number of successes
#O smoothing parameter
#P dimension of x
#Q standard deviation

Figure 9.4 shows the RNA-seq density estimate based on the negative binomial model.

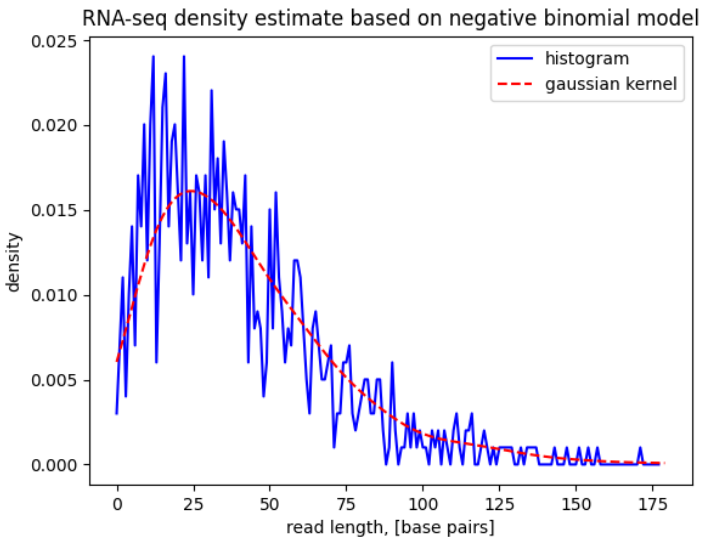


Figure 9.4 RNA-seq density estimate via histogram and Gaussian KDE

The figure shows two density estimates one based on histogram and the other based on Gaussian Kernel Density Estimator. We can see that the Gaussian density is much smoother and that we can further fine tune bin size smoothing parameter in the histogram estimator.

9.2.2. Tangent Portfolio Optimization

The objective of mean-variance analysis is to maximize the expected return of a portfolio for a given level of risk as measured by the standard deviation of past returns. By varying the mixing proportions of each asset, we can achieve different risk-return trade-offs.

In the code listing below, we first retrieve a data frame of closing prices for a list of stocks. We then examine stock price correlations via a scatter matrix plot. We create a randomized portfolio and compute the portfolio risk. Next, we generate 1000 randomly weighted portfolios and compute their value and risk. Finally, we choose a nearest neighbor portfolio weights in a way that minimizes standard deviation and maximizes portfolio value.

Listing 9.3 Tangent Portfolio Optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.neighbors import KDTree
from pandas.plotting import scatter_matrix
from scipy.spatial import ConvexHull

import pandas_datareader.data as web
from datetime import datetime
import pytz

STOCKS = ['SPY','LQD','TIP','GLD','MSFT']

np.random.seed(42)

if __name__ == "__main__":

    plt.close("all")

    #load data
    #year, month, day, hour, minute, second, microsecond
    start = datetime(2012, 1, 1, 0, 0, 0, 0, pytz.utc)
    end = datetime(2017, 1, 1, 0, 0, 0, 0, pytz.utc)

    data = pd.DataFrame()
    series = []
    for ticker in STOCKS:
        price = web.DataReader(ticker, 'stooq', start, end)    #A
        series.append(price['Close'])

    data = pd.concat(series, axis=1)
    data.columns = STOCKS
    data = data.dropna()

    scatter_matrix(data, alpha=0.2, diagonal='kde')   #B
    plt.show()

    cash = 10000
    num_assets = np.size(STOCKS)    #C
    cur_value = (1e4-5e3)*np.random.rand(num_assets,1) + 5e3
    tot_value = np.sum(cur_value)
    weights = cur_value.ravel()/float(tot_value)

    Sigma = data.cov().values
    Corr = data.corr().values
    volatility = np.sqrt(np.dot(weights.T, np.dot(Sigma, weights)))    #D

    plt.figure()
    plt.title('Correlation Matrix')
    plt.imshow(Corr, cmap='gray')
    plt.xticks(range(len(STOCKS)),data.columns)
    plt.yticks(range(len(STOCKS)),data.columns)
    plt.colorbar()
    plt.show()

    num_trials = 1000
```

```
    W = np.random.rand(num_trials, np.size(weights))    #E
    W = W/np.sum(W,axis=1).reshape(num_trials,1)    #F

    pv = np.zeros(num_trials)    #G
    ps = np.zeros(num_trials)    #H

    avg_price = data.mean().values
    adj_price = avg_price

    for i in range(num_trials):
        pv[i] = np.sum(adj_price * W[i,:])
        ps[i] = np.sqrt(np.dot(W[i,:].T, np.dot(Sigma, W[i,:])))

    points = np.vstack((ps,pv)).T
    hull = ConvexHull(points)

    plt.figure()
    plt.scatter(ps, pv, marker='o', color='b', linewidth = 3.0, label = 'tangent
       portfolio')
    plt.scatter(volatility, np.sum(adj_price * weights), marker = 's', color = 'r',
       linewidth = 3.0, label = 'current')
    plt.plot(points[hull.vertices,0], points[hull.vertices,1], linewidth = 2.0)
    plt.title('expected return vs volatility')
    plt.ylabel('expected price')
    plt.xlabel('portfolio std dev')
    plt.legend()
    plt.grid(True)
    plt.show()

    #query for nearest neighbor portfolio
    knn = 5
    kdt = KDTree(points)
    query_point = np.array([2, 115]).reshape(1,-1)
    kdt_dist, kdt_idx = kdt.query(query_point,k=knn)
    print("top-%d closest to query portfolios:" %knn)
    print("values: ", pv[kdt_idx.ravel()])
    print("sigmas: ", ps[kdt_idx.ravel()])
```

#A load data
#B plot data correlations
#C get current portfolio
#D compute portfolio risk
#E generate random portfolio weights
#F normalize
#G portoflio value w'v
#H portfolio sigma: sqrt(w'Sw)

Figure 9.5 shows regression results between pairs of portfolio assets (left).

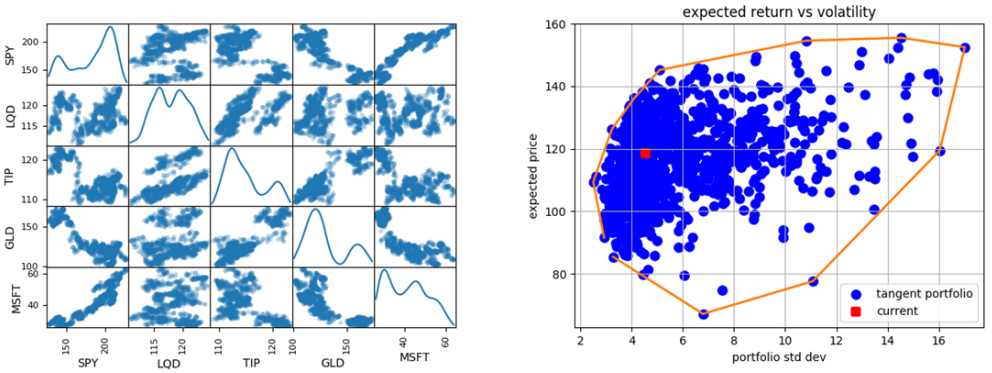


Figure 9.5 Pair Plot (left) and Tangent Portfolio (right)

Notice, for example, how SPY is uncorrelated with TIP and anti-correlated with GLD. Also, the diagonal densities are multi-modal and show negative skewness for riskier assets (e.g. SPY vs LQD). Figure 9.3 also shows the expected return vs risk trade-off for a set of randomly generated portfolios (right). The efficient frontier is defined by a set of portfolios at the top of the curve that correspond to maximum expected return for a given standard deviation. By adding a risk-free asset, we can choose a portfolio along a tangent line with slope equal to the Sharpe ratio. In the following section, we are going to learn how to discover structure in relational data.

9.3 Structure Learning

In this section, we are going to learn how to discover the graph structure given relational data. In other words, we would like to evaluate the probability of graph `G=(V,E)` given observed data `D`. One challenge in inferring the graph structure is the exponential number of possible graphs. For example, in a directed graph `G`, we can have `V` choose `2` edges and with each edge having two possible directions, we have `O(2^V(V-1)/2)` possible graphs. Since the problem of structure learning for general graphs is NP hard, we are going to focus on approximate methods. Namely, we'll look at Chow-Liu algorithm for tree-based graphs as well as inverse covariance estimation for general graphs.

9.3.1. Chow-Liu Algorithm

We can define the joint probability model for a tree `T` as follows:

$$p(x|T) = \prod_{t \in V} p(x_t) \prod_{(s,t) \in E} \frac{p(x_s, x_t)}{p(x_s)p(x_t)}$$

Equation 9.16 Joint Probability for a Tree T

where `p(xt)` is a node marginal and `p(xs,xt)` is an edge marginal. For example, for a `|V|=3` node V-shaped undirected tree, we have:

$$\begin{aligned} p(x_1, x_2, x_3|T) &= p(x_1)p(x_2)p(x_3)\frac{p(x_1, x_2)p(x_2, x_3)}{p(x_1)p(x_2)p(x_2)p(x_3)} \\ &= \frac{p(x1, x2)p(x_2, x_3)}{p(x_2)} = p(x_2)p(x_1|x_2)p(x_3|x_2) \end{aligned}$$

Equation 9.17 Joint Probability for a |V|=3 Node Tree T

To derive the Chow-Liu algorithm, we can use the above tree decomposition to write down the likelihood:

$$\begin{aligned} \log P(D|\theta, T) &= \sum_t \sum_k N_{tk} \log p(x_t = k|\theta) \\ &+ \sum_{s,t} \sum_{j,k} N_{stjk} \log \frac{p(x_s = j, x_t = k|\theta)}{p(x_s = j|\theta)p(x_t = k|\theta)} \end{aligned}$$

Equation 9.18 Log-Likelihood for a Tree T

where `Nstjk` is the number of times node `s` is in state `j` and node `t` is in state `k`, and `Ntk` is the number of times node `t` is in state `k`. We can re-write `Ntk` as `N x p(xt=k)` and similarly, `Nstjk` as `N x p (xs=j,xt=k)`. If we plug-in the above into our expression for log likelihood, we get:

$$\frac{1}{N}\log P(D|\theta,T) = \sum_{t\in V}\sum_{k}\hat{p}(x_t=k)\log\hat{p}(x_t=k) + \sum_{(s,t)\in E} I(x_s,x_t|\hat{\theta}_{st})$$

Equation 9.19 Normalized Log Likelihood for a Tree T

where `I(xs,xt|θ)` is the mutual information between `xs` and `xt`. Therefore, the tree topology that maximizes the log likelihood can be computed by the maximum weight spanning tree, where the edge weights are pairwise mutual information `I(xs,xt|θ)`. The above algorithm is known as the Chow-Liu algorithm. Note, to compute the maximum spanning tree (MST), we can use either Prim's algorithm or Kruskal's algorithms that can be implemented in `O(E log V)` time. In the following section, we are going to look at how to infer the structure of a general graph based on inverse covariance estimation.

9.3.2. Inverse Covariance Estimation

Identifying stock clusters helps discover similar companies which can be useful for comparable analysis or a pairs trading strategy. We can find similar clusters by estimating the inverse covariance (precision) matrix that can be used to construct a graph network of dependencies using the fact that zeros in the precision matrix correspond to absence of edges in the constructed graph. Let's represent our unknown graph structure as a Gaussian graphical model. Let $\Lambda=\Sigma^{-1}$ represent the precision matrix of the multivariate normal. Then, the log-likelihood of Λ can be derived as follows:

$$l(\Lambda) = \log\det\Lambda - \mathrm{tr}[S\Lambda]$$

Equation 9.20 Lambda log likelihood

where `S` is the empirical covariance matrix:

$$S = \frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)(x_i-\mu)^T$$

Equation 9.21 Empirical Covariance Matrix

To encourage sparse structure, we can add a penalty term for non-zero entries in the precision matrix. Thus, our graph lasso negative log likelihood objective becomes:

$$\mathrm{NLL}(\Lambda) = -\log\det\Lambda + \mathrm{tr}[S\Lambda] + c||\Lambda||_1$$

Equation 9.22 Regularized Negative Log Likelihood (NLL)

In the following example, we'll be using the difference between opening and closing daily price to compute empirical covariance used to fit graph lasso algorithm to estimate sparse precision matrix. Affinity propagation is used to compute the stock clusters and a linear embedding is used to display high dimensional data in 2D.

Listing 9.4 Inverse Covariance Estimation

```
import numpy as np
import pandas as pd
from scipy import linalg

from datetime import datetime
import pytz

from sklearn.datasets import make_sparse_spd_matrix
from sklearn.covariance import GraphicalLassoCV, ledoit_wolf
from sklearn.preprocessing import StandardScaler
from sklearn import cluster, manifold

import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection

import pandas_datareader.data as web

np.random.seed(42)

def main():

    #generate data (synthetic)
    #num_samples = 60
    #num_features = 20
    #prec = make_sparse_spd_matrix(num_features, alpha=0.95, smallest_coef=0.4,
       largest_coef=0.7)
    #cov = linalg.inv(prec)
    #X = np.random.multivariate_normal(np.zeros(num_features), cov, size=num_samples)
    #X = StandardScaler().fit_transform(X)

    #generate data (actual)
    STOCKS = {
        'SPY': 'S&P500',
        'LQD': 'Bond_Corp',
        'TIP': 'Bond_Treas',
        'GLD': 'Gold',
        'MSFT': 'Microsoft',
        'XOM':  'Exxon',
        'AMZN': 'Amazon',
        'BAC':  'BofA',
        'NVS':  'Novartis'}

    symbols, names = np.array(list(STOCKS.items())).T
```

```
#load data
#year, month, day, hour, minute, second, microsecond
start = datetime(2015, 1, 1, 0, 0, 0, 0, pytz.utc)
end = datetime(2017, 1, 1, 0, 0, 0, 0, pytz.utc)

qopen, qclose = [], []
data_close, data_open = pd.DataFrame(), pd.DataFrame()
for ticker in symbols:
    price = web.DataReader(ticker, 'stooq', start, end)
    qopen.append(price['Open'])
    qclose.append(price['Close'])

data_open = pd.concat(qopen, axis=1)
data_open.columns = symbols
data_close = pd.concat(qclose, axis=1)
data_close.columns = symbols

variation = data_close - data_open   #A
variation = variation.dropna()

X = variation.values
X /= X.std(axis=0)  #B

graph = GraphicalLassoCV()   #C
graph.fit(X)

gl_cov = graph.covariance_
gl_prec = graph.precision_
gl_alphas = graph.cv_alphas_
gl_scores = graph.cv_results_['mean_test_score']

plt.figure()
sns.heatmap(gl_prec, xticklabels=names, yticklabels=names)
plt.xticks(rotation=45)
plt.yticks(rotation=45)
plt.tight_layout()
plt.show()

plt.figure()
plt.plot(gl_alphas, gl_scores, marker='o', color='b', lw=2.0, label='GraphLassoCV')
plt.title("Graph Lasso Alpha Selection")
plt.xlabel("alpha")
plt.ylabel("score")
plt.legend()
plt.show()

_, labels = cluster.affinity_propagation(gl_cov)   #D
num_labels = np.max(labels)

for i in range(num_labels+1):
    print("Cluster %i: %s" %((i+1), ', '.join(names[labels==i])))

node_model = manifold.LocallyLinearEmbedding(n_components=2, n_neighbors=6,
   eigen_solver='dense')   #E
embedding = node_model.fit_transform(X.T).T

#generate plots
plt.figure()
plt.clf()
```

```
    ax = plt.axes([0.,0.,1.,1.])
    plt.axis('off')

    partial_corr = gl_prec
    d = 1 / np.sqrt(np.diag(partial_corr))
    non_zero = (np.abs(np.triu(partial_corr, k=1)) > 0.02)  #F

    #plot the nodes
    plt.scatter(embedding[0], embedding[1], s = 100*d**2, c = labels, cmap =
       plt.cm.Spectral)

    #plot the edges
    start_idx, end_idx = np.where(non_zero)
    segments = [[embedding[:,start], embedding[:,stop]] for start, stop in zip(start_idx,
       end_idx)]
    values = np.abs(partial_corr[non_zero])
    lc = LineCollection(segments, zorder=0, cmap=plt.cm.hot_r,
       norm=plt.Normalize(0,0.7*values.max()))
    lc.set_array(values)
    lc.set_linewidths(2*values)
    ax.add_collection(lc)

    #plot the labels
    for index, (name, label, (x,y)) in enumerate(zip(names, labels, embedding.T)):
        plt.text(x,y,name,size=12)

    plt.show()

if __name__ == "__main__":
    main()
```

#A per day variation in price for each symbol
#B standardize to use correlation rather than covariance
#C estimate inverse covariance
#D cluster using affinity propagation
#E find a low dimensional embedding for visualization
#F connectivity matrix

Figure 9.6 shows the sparse precision matrix estimated by the graph lasso algorithm.

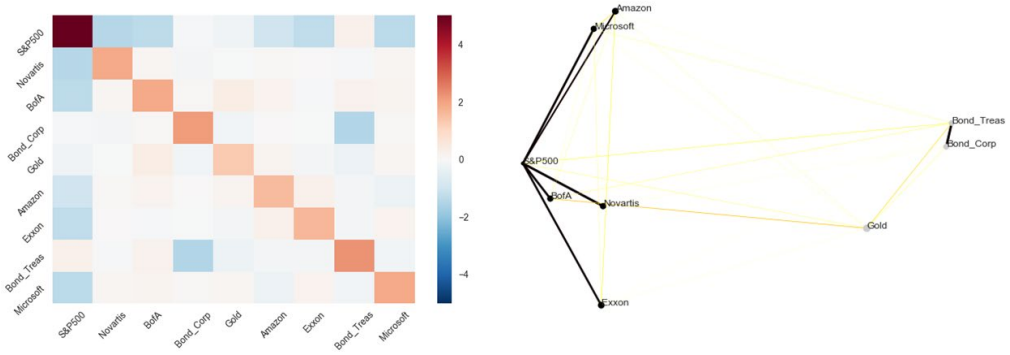


Figure 9.6 Graph Lasso estimated precision matrix (left) and stock clusters (right)

Edge values in the above precision matrix that are greater than a threshold correspond to connected components from which we can compute stock clusters as visualized on the right. In the next section, we are going to learn an energy minimization algorithm called simulated annealing.

9.4 Simulated Annealing

Simulated Annealing (SA) is a heuristic search method that allows for occasional transitions to less favorable states in order to escape local optima. We can formulate SA as an energy minimization problem with temperature parameter `T` as follows:

$$\begin{aligned} \alpha &= \exp\{(E_{old} - E_{new})/T\} \\ p &= \min(1, \alpha) \end{aligned}$$

Equation 9.23 Simulated Annealing Transition Probability

As we transition to a new state, we would like the energy in the new state to be lower, in which case $\alpha>1$ and $p=1$, meaning we accept the state transition with probability $p=1$. On the other hand, if the energy of the new state is higher, this will cause $\alpha<1$ and in that case we accept the transition with probability $p=\alpha$. In other words, we accept unfavorable transitions with probability proportional to the difference in energies between states and inversely proportional to the temperature parameter `T`. Initially, the temperature `T` is high allowing for a lot of random transitions. As the temperature decreases (according to a cooling schedule), the difference in energy becomes more pronounced. The above formulation allows simulated annealing to escape local optima. We are now ready to look at the following pseudo code.

```
class simulated_annealing
function run(x_init, y_init):
converged = False
T = 1
x_old, y_old = x_init, y_init
energy_old = target(x_init, y_init)
while not converged:
    x_new, y_new = proposal(x_old, y_old)        <- Evaluate proposal
    energy_new = target(x_new, y_new)            <- Compute energy
    converged = check_convergence()
    alpha = exp((energy_old - energy_new)/T)
    r = min(1, alpha)                            <- Transition probability
    u = Unif[0, 1]
    if u < r
        x_old, y_old = x_new, y_new              } Accept proposed state
        energy_old = energy_new
    end if
    T = temperature_schedule()
end while
x_opt, y_opt = x_old, y_old
return x_opt, y_opt
```

The `simulated_annealing` class contains the main `run` function. We begin by initializing the annealing temperature `T` and evaluating our `target` function at the initial location. Recall, that we are interested in finding a minimum point in a complex energy landscape represented by the target function. We sample from our proposal distribution to obtain a new set of coordinates and evaluate the energy the proposed coordinates. We check for convergence to decide whether to break out of the loop or to continue. Next, we compute the simulated annealing transition probability `alpha` as a difference between old and new energy states divided by the temperature `T`. In the case of `r>1`, we accept the low energy state with probability `1`, and in the case of `0<r<1` (i.e. the energy of the new state is higher), we accept the transition with probability `r=`α. Finally, we adjust the temperature according to a cooling schedule and upon convergence return the optimal coordinates (ones that achieve the minimum energy found by simulated annealing). We are now ready to implement simulated annealing algorithm from scratch.

Listing 9.5 Simulated Annealing

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

class simulated_annealing():
    def __init__(self):
        self.max_iter = 1000
        self.conv_thresh = 1e-4
        self.conv_window = 10

        self.samples = np.zeros((self.max_iter, 2))
        self.energies = np.zeros(self.max_iter)
        self.temperatures = np.zeros(self.max_iter)

    def target(self, x, y):    #A
        z = 3*(1-x)**2 * np.exp(-x**2 - (y+1)**2) \
            - 10*(x/5 -x**3 - y**5) * np.exp(-x**2 - y**2) \
            - (1/3)*np.exp(-(x+1)**2 - y**2)
        return z

    def proposal(self, x, y):
        mean = np.array([x, y])
        cov =  1.1 * np.eye(2)
        x_new, y_new = np.random.multivariate_normal(mean, cov)
        return x_new, y_new

    def temperature_schedule(self, T, iter):
        return 0.9 * T

    def run(self, x_init, y_init):

        converged = False
        T = 1
        self.temperatures[0] = T
        num_accepted = 0
        x_old, y_old = x_init, y_init
        energy_old = self.target(x_init, y_init)

        iter = 1
        while not converged:
            print("iter: {:4d}, temp: {:.4f}, energy = {:.6f}".format(iter, T, energy_old))
            x_new, y_new = self.proposal(x_old, y_old)
            energy_new = self.target(x_new, y_new)

            if iter > 2*self.conv_window:    #B
                vals = self.energies[iter-self.conv_window : iter-1]
                if (np.std(vals) < self.conv_thresh):
                    converged = True
                #end if
            #end if

            alpha = np.exp((energy_old - energy_new)/T)
            r = np.minimum(1, alpha)
            u = np.random.uniform(0, 1)
            if u < r:
                x_old, y_old = x_new, y_new
```

```
                num_accepted += 1
                energy_old = energy_new
            #end if
            self.samples[iter, :] = np.array([x_old, y_old])
            self.energies[iter] = energy_old

            T = self.temperature_schedule(T, iter)
            self.temperatures[iter] = T

            iter = iter + 1

            if (iter > self.max_iter): converged = True
        #end while

        niter = iter - 1
        acceptance_rate = num_accepted / niter
        print("acceptance rate: ", acceptance_rate)

        x_opt, y_opt = x_old, y_old

        return x_opt, y_opt, self.samples[:niter,:], self.energies[:niter],
       self.temperatures[:niter]

if __name__ == "__main__":

    SA = simulated_annealing()

    nx, ny = (1000, 1000)
    x = np.linspace(-2, 2, nx)
    y = np.linspace(-2, 2, ny)
    xv, yv = np.meshgrid(x, y)

    z = SA.target(xv, yv)
    plt.figure()
    plt.contourf(x, y, z)
    plt.title("energy landscape")
    plt.show()

    #find global minimum by exhaustive search
    min_search = np.min(z)
    argmin_search = np.argwhere(z == min_search)
    xmin, ymin = argmin_search[0][0], argmin_search[0][1]
    print("global minimum (exhaustive search): ", min_search)
    print("located at (x, y): ", x[xmin], y[ymin])

    #find global minimum by simulated annealing
    x_init, y_init = 0, 0
    x_opt, y_opt, samples, energies, temperatures = SA.run(x_init, y_init)
    print("global minimum (simulated annealing): ", energies[-1])
    print("located at (x, y): ", x_opt, y_opt)

    plt.figure()
    plt.plot(energies)
    plt.title("SA sampled energies")
    plt.show()

    plt.figure()
    plt.plot(temperatures)
    plt.title("Temperature Schedule")
```

```
    plt.show()
```

#A energy landscape
#B check convergence

Figure 9.7 shows the target energy landscape (left) and SA sampled energies (left).

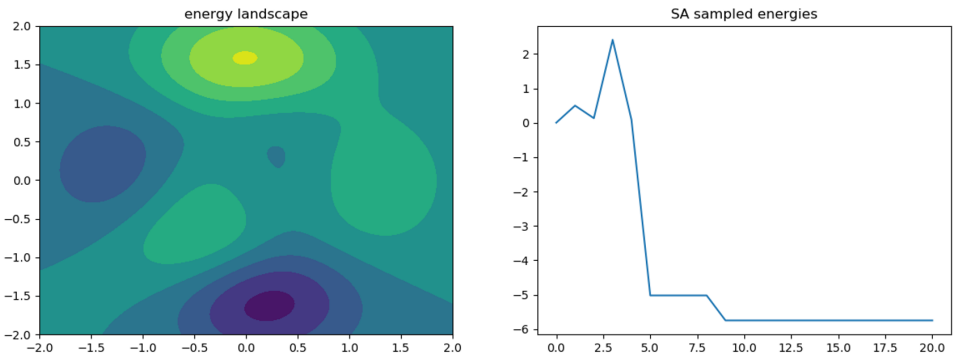


Figure 9.7 Energy landscape (left) and SA sampled energies (right)

In the above example, we were able to find the global minimum of an energy land scape using simulated annealing that matches the global minimum by exhaustive search. In the following section, we are going to study a genetic algorithm inspired by evolutionary biology.

9.5 Genetic Algorithm

Genetic Algorithm (GA) is inspired by and modeled after evolutionary biology. It consists of a population of (randomly initialized) individual genomes that are evaluated for their fitness with respect to a target. Two individuals combine and cross-over their genome to produce an off-spring. This cross-over step is followed by a mutation by which individual bases can change according to a mutation rate. These new individuals are added to the population and scored by the fitness function that determines whether the individual survives in the population or not. Let's look at the following pseudo-code. In this example, we are going to evolve a random string to match a string target.

```
class GeneticAlgorithm
function calculate_fitness(population, target):
for individual in population:
    compute loss as a distance between individual and target
    compute fitness = 1 / (loss + epsilon)
end for
return population_fitness
function mutate(individual, mutation_rate):
randomly change characters with probability equal to mutation_rate
return new_individual
function crossover(parent1, parent2):
cross_idx = random_integer(0, len(parent1))
child1 = parent1[:cross_idx] + parent2[cross_idx:]
child2 = parent2[:cross_idx] + parent1[cross_idx:]
return child1, child2
function run(target, population_size, mutation_rate):
init_population()
for epoch in num_iter:
    population_fitness = calculate_fitness(population, target)
    fittest_individual = population(argmax(population_fitness))
    if fittest_individual == target
        return fittest_individual
    end if
    parent_probabilities = fitness / sum(population_fitness)
    new_population = []
    for i in population_size:
        parent1, parent2 = random_choice(population, p=parent_probabilities)
        child1, child2 = crossover(parent1, parent2)
        new_population += [mutate(child1), mutate(child2)]
    end for
end for
```

The `GeneticAlgorithm` class consists of the following functions: `calculate_fitness`, `mutate`, `crossover` and `run`. In `calculate_fitness` function, we compute the fitness score as 1/loss where loss is defined as a distance between individual and target ascii string characters. In the `mutate` function, we randomly change string characters of an individual according to a mutation rate. In the `crossover` function, we choose an index at which to cross the parents genomes at random and then carry out the cross-over of parents genomes at the chosen index producing two children. Finally, in the `run` function, we initialize the population and compute population fitness. After which, we find the fittest individual in the

population and compare it with the target. If the two match, we exit the algorithm and return the fittest individual. Otherwise, we select two parents according to parent probabilities ranked by fitness, crossover to produce offspring, mutate each offspring and add them back to the new population. We repeat this process a fixed number of times or until the target is found. Let's now look at the genetic algorithm implementation in detail.

Listing 9.6 Genetic Algorithm

```
import numpy as np
import string

class GeneticAlgorithm():

    def __init__(self, target_string, population_size, mutation_rate):
        self.target = target_string
        self.population_size = population_size
        self.mutation_rate = mutation_rate
        self.letters = [" "] + list(string.ascii_letters)

    def initialize(self):   #A
        self.population = []
        for _ in range(self.population_size):
            individual = "".join(np.random.choice(self.letters, size=len(self.target)))
            self.population.append(individual)

    def calculate_fitness(self):   #B
        population_fitness = []
        for individual in self.population:
            loss = 0   #C
            for i in range(len(individual)):
                letter_i1 = self.letters.index(individual[i])
                letter_i2 = self.letters.index(self.target[i])
                loss += abs(letter_i1 - letter_i2)
            fitness = 1 / (loss + 1e-6)
            population_fitness.append(fitness)
        return population_fitness

    def mutate(self, individual):   #D
        individual = list(individual)
        for j in range(len(individual)):
            if np.random.random() < self.mutation_rate:
                individual[j] = np.random.choice(self.letters)
        return "".join(individual)

    def crossover(self, parent1, parent2):  #E
        cross_i = np.random.randint(0, len(parent1))
        child1 = parent1[:cross_i] + parent2[cross_i:]
        child2 = parent2[:cross_i] + parent1[cross_i:]
        return child1, child2

    def run(self, iterations):
        self.initialize()

        for epoch in range(iterations):
            population_fitness = self.calculate_fitness()

            fittest_individual = self.population[np.argmax(population_fitness)]
```

```
            highest_fitness = max(population_fitness)

            if fittest_individual == self.target:
                break

            #select individual as a parent proportional to individual's fitness
            parent_probabilities = [fitness / sum(population_fitness) for fitness in
      population_fitness]

            #next generation
            new_population = []
            for i in np.arange(0, self.population_size, 2):
                #select two parents
                parent1, parent2 = np.random.choice(self.population, size=2,
      p=parent_probabilities, replace=False)
                #crossover to produce offspring
                child1, child2 = self.crossover(parent1, parent2)
                #save mutated offspring for next generation
                new_population += [self.mutate(child1), self.mutate(child2)]

            print("iter %d, closest candidate: %s, fitness: %.4f" %(epoch,
      fittest_individual, highest_fitness))
            self.population = new_population

        print("iter %d, final candidate: %s" %(epoch, fittest_individual))

if __name__ == "__main__":

    target_string = "Genome"
    population_size = 50
    mutation_rate = 0.1

    ga = GeneticAlgorithm(target_string, population_size, mutation_rate)
    ga.run(iterations = 1000)
```

#A init population with random strings
#B calculate fitness of each individual in population
#C calculate loss as the distance between characters
#D randomly change characters with probability equal to mutation rate
#E create children from parents by cross-over

As we can see from the output, we were able to produce the target sequence "Genome" by evolving randomly initialized letter sequences. In the next section, we are going to expand on the topics learned by reviewing research literature on unsupervised learning.

9.6 ML Research: Unsupervised Learning

In this section, we cover additional insights and research related to the topics presented in this chapter. We had our first encounter with a Bayesian non-parametric model in the form of Dirichlet Process (DP) K-means. The number of parameters in such models increases with data and therefore Bayesian non-parametric models are better able to model real world scenarios. DP-means can be seen as small variance asymptotics (SVA) approximation of the Dirichlet Process Mixture Model. One of the main advantages of Bayesian non-parametric models is that they can be used for modeling infinite mixtures and hierarchical extensions can be utilized for sharing clusters across multiple data groups.

We looked at the EM algorithm which is a powerful optimization framework used widely in machine learning. There are a number of extensions to EM algorithm such as online EM that deals with online or streaming datasets, annealed EM that uses the temperature parameter to smooth the energy landscape during optimization in order to track the global optimum, variational EM that replaces exact inference in the E step with variational inference, Monte Carlo EM that draws samples in the E step from the intractable distribution and several others.

We looked at several ways to reduce dimensionality for the purpose of feature selection or data visualization. There a number of other ways to learn the underlying data manifold such as Isomap which is an extension of Kernel PCA that seeks to maintain geodesic distances between all points, Locally Linear Embedding (LLE) which can be thought of as a series of local PCA that are globally compared to find the best non-linear embedding, Spectral Embedding based on decomposition of the graph Laplacian, and the Multi-Dimensional Scaling (MDS) in which the distances in the embedding reflect well the distances in the original high-dimensional space. Note that some of these methods can be combined such as PCA used to pre-process the data and initialize t-SNE to reduce the computational complexity.

We looked at a powerful way to discover topics in text documents using Variational Bayes algorithm for Latent Dirichlet Allocation. There are a number of extensions to LDA as well. For example, correlated topic model captures correlations between topics, dynamic topic model tracks the evolution of topics over time, and supervised LDA model can be used to grade or assign scores to documents to evaluate their quality.

We looked at the problem of density estimation using kernels and discovered the effect of smoothing parameter on the resulting estimate. We can implement KDE more efficiently by using Ball Tree or KD Tree to reduce the time complexity required to query the data.

In the area of structure learning we discovered the exponential number of possible graph topologies and touched on the topic of causality. We looked at how we could construct simpler tree graphs using mutual information between nodes as edge weights in the maximum weight spanning tree. We also saw how regularizing the inverse covariance matrix for general graphs led to more interpretable topologies with fewer edges in the inferred graph.

Finally, we looked at two unsupervised algorithms inspired by statistical physics (simulated annealing) and evolutionary biology (the genetic algorithm). We saw how by using the temperature parameter and a cooling schedule, we can modify the energy landscape, and how selecting unfavorable in the short-term moves can lead to better long-term optima. There are a number of NP hard problems that could be approximated with Simulated Annealing such as the Traveling Salesman Problem (TSP). We can often use several restarts and different initialization points to arrive at better optima. The genetic algorithm on the other hand, although satisfying in its parallel to nature, can take a long time to converge. However, it can lead to a number of interesting applications such as neural network architecture search.

9.7 Exercises

9.1 Explain how Latent Dirichlet Allocation can be interpreted as non-negative matrix factorization

9.2 Explain why sparsity is desirable in inferring general graph structure

9.3 List several NP-hard problems that can be approximated with Simulated Annealing (SA) algorithm

9.4 Brainstorm problems that can be efficiently solved by applying Genetic Algorithm (GA)

9.8 Summary

- Latent Dirichlet Allocation (LDA) is a topic model that represents each document as a finite mixture of topics, where a topic is a distribution over words. The objective is to learn the shared topic distribution and topic proportions for each document.
- A common method for adjusting the word counts is *tf-idf* that logarithmically drives down to zero word counts that occur frequently across documents: `A log(D/nt)`, where `D` is the total number of documents in the corpus and `nt` is the number of documents where term `t` appears.
- The goal of density estimation is to model the probability density of data. Kernel Density Estimator (KDE) allocates one cluster center per data point.
- The objective of mean-variance analysis is to maximize the expected return of a portfolio for a given level of risk as measured by the standard deviation of past returns
- Since the problem of structure learning for general graphs is NP hard, we focused on approximate methods. Namely, we looked at Chow-Liu algorithm for tree-based graphs as well as inverse covariance estimation for general graphs.
- Simulated Annealing (SA) is a heuristic search method that allows for occasional transitions to less favorable states in order to escape local optima.
- We can formulate simulated annealing as an energy minimization problem with temperature parameter T, with which we can modify the energy landscape and select unfavorable in the short-term moves can lead to better long-term optima.
- Genetic Algorithm (GA) is inspired by and modeled after evolutionary biology. It consists of a population of (randomly initialized) individual genomes that are evaluated for their fitness with respect to a target.
- In Genetic Algorithm, two individuals combine and cross-over their genome to produce an off-spring. This cross-over step is followed by a mutation by which individual bases can change according to a mutation rate. The resulting offspring are added to the population and scored according to their fitness level.

10

Fundamental Deep Learning Algorithms

This chapter covers

- Multi-Layer Perceptron
- Convolutional Neural Nets: LeNet on MNIST and ResNet image search
- Recurrent Neural Nets: LSTM sequence classification and multi-input neural net
- Neural Network Optimizers

In the previous chapter, we looked at selected unsupervised ML algorithms to help discover patterns in our data. In this chapter, we introduce deep learning algorithms. Deep learning algorithms are part of supervised learning that we encountered in Chapters 5, 6, and 7. Deep learning algorithms revolutionized the industry and enabled a number of research and business applications that were previously thought to be out of reach by classic ML algorithms. We'll begin this chapter with the fundamentals such as Multi-Layer Perceptron (MLP) and LeNet convolutional model for MNIST digit classification. Followed by more advanced applications such as image search based on ResNet50 Convolutional Neural Network (CNN). We will delve into Recurrent Neural Networks (RNNs) applied to sequence classification using LSTMs and implement from scratch a multi-input model for sequence similarity. We'll then discuss different optimization algorithms used for training neural networks and do a comparative study. We will be using Keras/TensorFlow deep learning library throughout this chapter.

10.1 Multi-Layer Perceptron

Multi-Layer Perceptron (MLP) is commonly used for classification and regression predictions tasks. A MLP consists of multiple densely connected layers that perform the following linear (affine) operation:

$$f(x;\theta) = Wx + b$$

Equation 10.1 MLP layer

where `x` is the input vector, `W` is the weight matrix and `b` is the bias term. While the linearity of the operation makes it easy to compute, it is limiting when it comes to stacking multiple layers. By introducing non-linearity between layers via activation functions enables the model to have greater expressive power. One common non-linearity is ReLU defined as follows:

$$\mathrm{ReLU}(x) = \max(0, x)$$

Equation 10.2 ReLU activation function

We can define MLP with `L` layers as a composition of $f_l(x;\mathrm{theta}) = \mathrm{ReLU}(W_l x + b_l)$ functions as follows:

$$\mathrm{MLP}(x;\theta) = f_L(f_{L-1}(\cdots(f_1(x;\theta_1))\cdots))$$

Equation 10.3 L-layer MLP

The activation of the last layer function f_L will depend on the task at hand: softmax for classification or identity for regression. The softmax activation computes class probabilities that sum to one based on real-valued input and is defined as follows:

$$\mathrm{softmax}(x) = \frac{e^x}{\sum_{i=1}^{K} e^{x_i}}$$

Equation 10.4 Softmax function

where `K` is the number of classes. Figure 10.1 shows a MLP architecture.

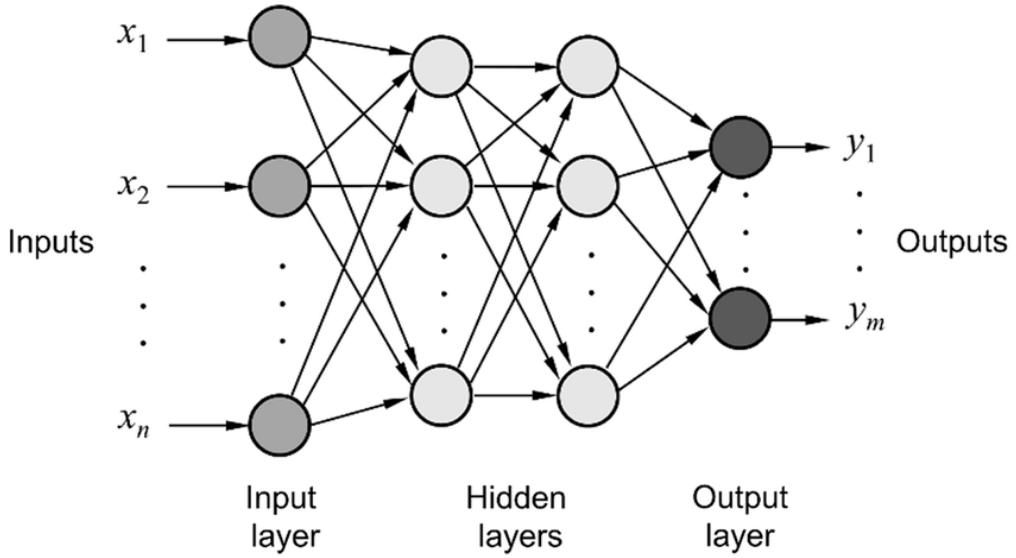


Figure 10.1 MLP Architecture

In order to train the neural network, we need to introduce the loss function. A loss function tells us how far away the network output is from expected ground truth labels. The loss depends on the task we are optimizing for, and in the case of our MLP, can be for example cross-entropy `H(p,q)` loss for classification or Mean Square Error (MSE) loss for regression.

$$L_H(y, \hat{y}) = H(p, q) = -\sum_{x \in X} p(x) \log q(x) = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

$$L_{MSE}(y, \hat{y}) = \text{MSE}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Equation 10.5 Cross-Entropy vs MSE loss functions

where `y` is the true label and `p(x)` is the true distribution, and `\hat{y}` is the predicted label and `q(x)` is the predicted distribution.

MLPs are known as feed-forward networks because they have a direct computational graph from input to output. A neural network training consists of a forward and a backward pass. During inference (forward pass), each layer produces a forward message `z = f(x;\theta)` as the output (given the current weights of the network) followed by computation of the loss function. During the backward pass, each layer takes a backward

message `dL/dz` of the next layer and produces two backward messages at the output: `dL/dx` gradient wrt to previous layer `x` and `dL/d\theta` gradient wrt to parameters of the current layer.

This back-propagation algorithm is basfed on the chain rule and can be summarized as follows:

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial x_i}$$

$$\frac{\partial L}{\partial \theta_i} = \sum_j \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial \theta_i}$$

Equation 10.6 Backpropagation Algorithm

Once we know the gradient with respect to parameters for each layer, we can update the layer parameters as follows:

$$\theta_i^t = \theta_i^{t-1} - \lambda \frac{\partial L}{\partial \theta_i}$$

Equation 10.7 Parameter Update

where `\lambda` is the learning rate. Note, that the gradients are computed automatically by deep learning frameworks such as TensorFlow/PyTorch.

During training, we may want to adopt a learning rate schedule to avoid local optima and converge on a solution. We typically start with the learning rate of some constant and reduce it according to a schedule over epochs, where one epoch is a single pass through the training dataset. Throughout this chapter, we'll be using an exponential learning rate schedule. Other alternatives include, a piece-wise linear schedule with successive halving, cosine annealing schedule and one-cycle schedule.

Before we get to the implementation of MLP, it's advantageous to talk about model capacity to avoid under-fitting and regularization to prevent over-fitting. Regularization can occur at multiple levels such as weight decay, early stopping, and dropout. At a high level, we would like to increase model capacity by changing the architecture (e.g. increasing the number of layers and number of hidden units per layer) if we find that the model is under-fitting or not achieving high validation accuracy. On the other hand, to avoid over-fitting, we can introduce weight decay or l2 regularization applied to non-bias weights (`W` but not `b`) of

each layer. Weight decay encourages smaller weights and therefore simpler models. Another form of regularization is stopping the training early when we notice that validation loss is starting to increase away from the training loss. This early stopping criterion can be used to save time and computational resources while saving a checkpoint of the model. Finally, dropout is an effective regularization technique, where the dense connections are dropped or zeroed at random according to a fixed rate as shown in Figure 10.2. Dropout enables better generalization (see Srivastava et al, "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014).

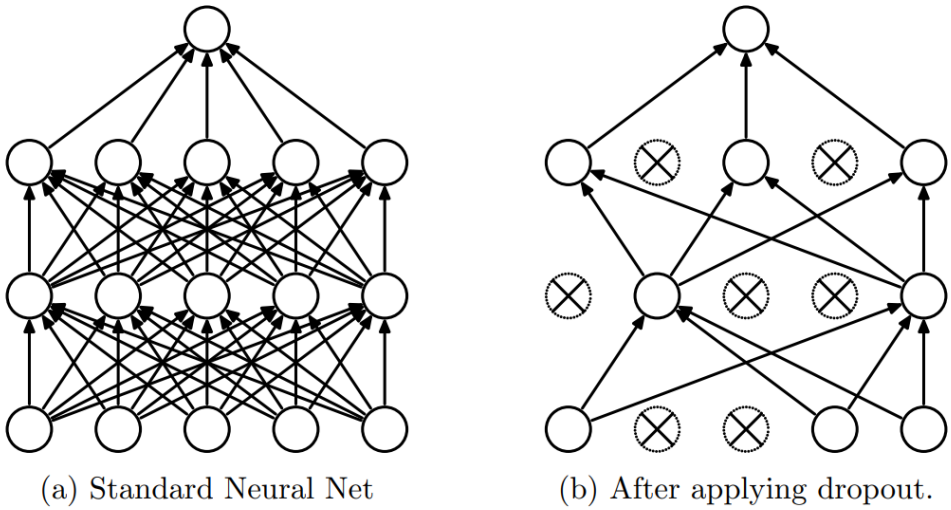


Figure 10.2 Dropout applied to Multi-Layer Perceptron at 1 Training Epoch.

Let's combine the principles we learned so far in our first implementation of the Multi-Layer Perceptron (MLP) in Keras/TensorFlow! For more information regarding the Keras library the reader is encouraged to visit https://keras.io/ or review F. Chollet, "Deep Learning with Python", Manning, 2021.

Listing 10.1 Multi-Layer Perceptron (MLP)

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

from keras.models import Sequential
from keras.layers import Dense, Dropout

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard    #A
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping
```

```
import math
import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):    #B
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

if __name__ == "__main__":

    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_train = x_train.reshape(x_train.shape[0], x_train.shape[1] *
       x_train.shape[2]).astype("float32") / 255
    x_test = x_test.reshape(x_test.shape[0], x_test.shape[1] *
       x_test.shape[2]).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 64    #C
    num_epochs = 16    #C

    model = Sequential()
    model.add(Dense(128, input_shape=(784, ), activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    model.compile(
        loss=keras.losses.CategoricalCrossentropy(),
        optimizer=tf.keras.optimizers.RMSprop(),
        metrics=["accuracy"]
    )

    model.summary()

    #define callbacks
    file_name = SAVE_PATH + 'mlp-weights-checkpoint.h5'
    checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
       save_best_only=True, mode='min')
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
    early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=16,
       verbose=1)
    #tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
    callbacks_list = [checkpoint, reduce_lr, early_stopping]

    hist = model.fit(x_train, y_train_label, batch_size=batch_size, epochs=num_epochs,
       callbacks=callbacks_list, validation_split=0.2)    #D

    test_scores = model.evaluate(x_test, y_test_label, verbose=2)    #E

    print("Test loss:", test_scores[0])
```

```
print("Test accuracy:", test_scores[1])

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()

plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
plt.title('MLP model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

#A for visualizing metrics
#B learning rate schedule
#C training params
#D model training
#E model evaluation

Figure 10.3 shows the cross-entropy loss and accuracy for both training and validation datasets.

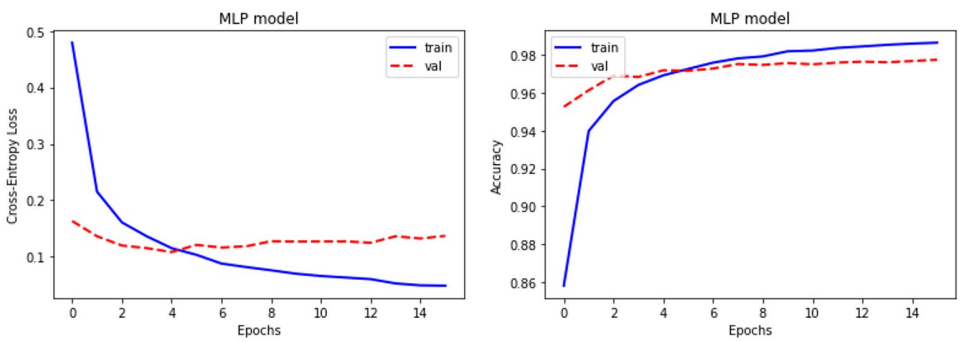


Figure 10.3 MLP cross-entropy loss and accuracy for training and validation datasets

The MLP model achieves `98%` accuracy on the test dataset after only `16` training epochs. Notice how the training loss drops below the validation and similarly the accuracy on the training set is higher than the validation set. Since we are interested in how our model generalizes to unseen data, the validation set is the real indicator of performance. In the next section, we are going to look at a class of neural networks better suited for processing image data and therefore widely used in computer vision.

10.2 Convolutional Neural Nets

Convolutional Neural Networks (CNNs) use convolution and pooling operations in place of vectorized matrix multiplications. In this section, we'll motivate the use of these operations and describe two architectures: LeNet for MNIST digit classification and ResNet50 applied to image search.

CNNs work exceptionally well for image data. Images are high dimensional objects of size `WxHxC`, where `W` is the width, `H` is the height, and `C` is the number of channels (e.g. `C=3` for RGB image and `C=1` for grayscale). CNNs consists of trainable filters or kernels that get convolved with the input to produce a feature map. This results in a vast reduction in the number of parameters and ensures invariance to translations of the input (since we would like to classify an object at any location of the image). The convolution operation is 2-D in the case of images, 1-D in case of time-series and 3-D in the case of volumetric data. A 2-D convolution is defined as follows:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, i - b)$$

Equation 10.8 2-D convolution

In other words, we slide the kernel across every possible position of the input matrix. At each position, we perform a dot-product operation and compute a scalar value. Finally, we gather these scalar values in a feature map output.

Convolutional layers can be stacked. Since convolutions are linear operators, we include non-linear activation functions in between just as we did in fully connected layers.

Many popular CNN architectures include a pooling layer. Pooling layers down-sample the feature maps by aggregating information. For example, max-pooling computes a maximum over incoming input values, while average pooling replaces the max operation with the average. Similarly, global average pooling could be used to reduce a `WxHxD` feature map into `1x1xD` aggregate, which can then be reshaped to a `D`-dimensional vector for further processing. Let's look at a few applications of CNNs.

10.2.1 LeNet on MNIST

In this section, we are going to study the classic LeNet architecture (LeCun et al, "Gradient-Based Learning Applied to Document Recongition", Proc of the IEEE, 1998) developed by

Yann LeCun for handwritten digit classification trained on MNIST dataset. It follows the design pattern of a convolutional layer, followed by ReLU activation, followed by max-pooling operation as shown in Figure 10.4

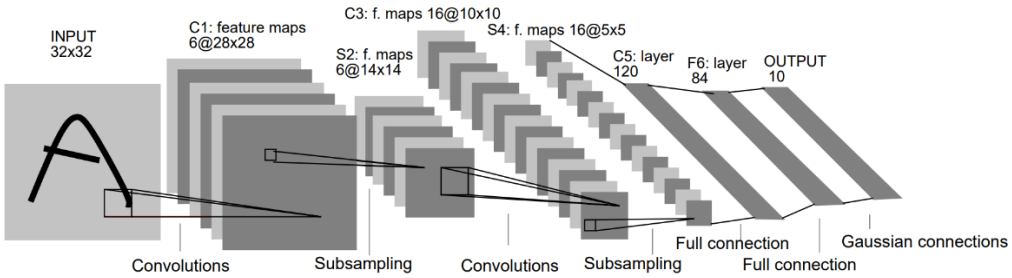


Figure 10.4 LeNet Architecture for MNIST digits classification

This sequence is stacked and the number of filters is increased as we go from the input to the output of the CNN. Notice, that we are both learning more features by increasing the number of filters and reducing the feature map dimensions through the max-pooling operations. In the last layer we flatten the feature map and add a dense layer followed by a softmax classifier.

In the following code listing, we start off by loading the MNIST dataset and reshaping the images to correct image size. We define our training parameters and model parameters followed by the definition of CNN architecture. Notice, a sequence of convolutional, ReLU and max-pooling operations. We compile the model and define a set of callback functions that will be executed during model training. We record the training history and evaluate the model on the test dataset. Finally, we save the prediction results and generate accuracy and loss plots.

We are now ready to implement a simple MNIST CNN architecture from scratch using Keras/TensorFlow!

Listing 10.2 Simple CNN for MNIST classification

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, Activation

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import math
import matplotlib.pyplot as plt
```

```
tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):    #A
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)


if __name__ == "__main__":

    img_rows, img_cols = 28, 28
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1).astype("float32") /
       255
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 128    #B
    num_epochs = 8    #B

    num_filters_l1 = 32    #C
    num_filters_l2 = 64    #C

    #CNN architecture
    cnn = Sequential()
    cnn.add(Conv2D(num_filters_l1, kernel_size = (5, 5), input_shape=(img_rows, img_cols,
       1), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Conv2D(num_filters_l2, kernel_size = (5, 5), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Flatten())
    cnn.add(Dense(128))
    cnn.add(Activation('relu'))

    cnn.add(Dense(num_classes))
    cnn.add(Activation('softmax'))

    cnn.compile(
        loss=keras.losses.CategoricalCrossentropy(),
        optimizer=tf.keras.optimizers.Adam(),
        metrics=["accuracy"]
    )

    cnn.summary()

    #define callbacks
    file_name = SAVE_PATH + 'lenet-weights-checkpoint.h5'
    checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
```

```
    save_best_only=True, mode='min')
reduce_lr = LearningRateScheduler(scheduler, verbose=1)
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=16,
    verbose=1)
#tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
callbacks_list = [checkpoint, reduce_lr, early_stopping]

hist = cnn.fit(x_train, y_train_label, batch_size=batch_size, epochs=num_epochs,
    callbacks=callbacks_list, validation_split=0.2)    #D

test_scores = cnn.evaluate(x_test, y_test_label, verbose=2)    #E

print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])

y_prob = cnn.predict(x_test)
y_pred = y_prob.argmax(axis=-1)

submission = pd.DataFrame(index=pd.RangeIndex(start=1, stop=10001, step=1),
    columns=['Label'])
submission['Label'] = y_pred.reshape(-1,1)
submission.index.name = "ImageId"
submission.to_csv(SAVE_PATH + '/lenet_pred.csv', index=True, header=True)

plt.figure()
plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Loss')
plt.legend(loc='upper right')
plt.show()

plt.figure()
plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='upper left')
plt.show()

plt.figure()
plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
plt.title('LeNet model')
plt.xlabel('Epochs')
plt.ylabel('Learning Rate')
plt.legend()
plt.show()
```

#A learning rate schedule
#B training parameters
#C model parameters
#D model training
#E model evaluation

With CNN architecture, we are able to achieve `99%` accuracy on the test set, which is impressive!

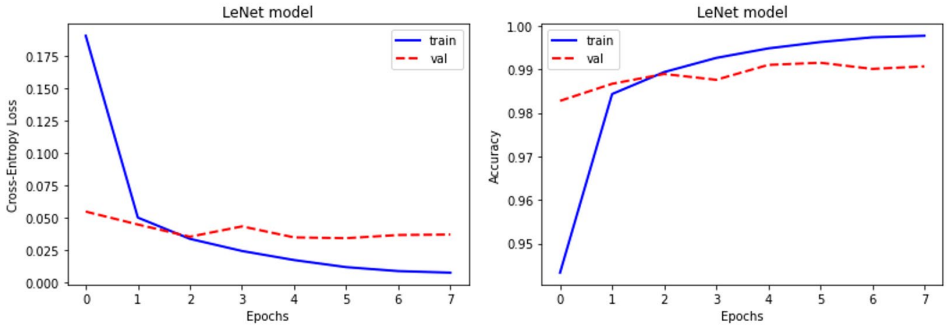


Figure 10.5 LeNet CNN loss and accuracy plots for training and validation datasets.

Notice how the training and validation curves differ for both cross entropy loss and accuracy. Early stopping enables us to stop the training when the validation loss does not improve over several epochs.

10.2.2 ResNet Image Search

The goal of image search is to retrieve images from a database similar to the query image. In this section, we'll be using a pre-trained ResNet50 CNN to encode a collection of images into dense vectors. This allows us to find similar images by computing distances between vectors.

The ResNet50 architecture (He et al, "Deep Residual Learning for Image Recognition", CVPR 2016) uses skip connections to avoid the vanishing gradient problem. Skip connections skip some layers in the network as shown in Figure 10.6.

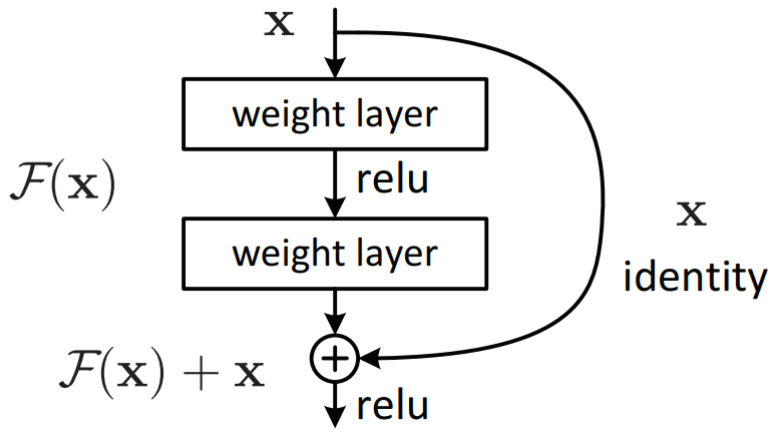


Figure 10.6 Skip Connection

The core idea behind ResNet is to back-propagate through the identity function in order to preserve the gradient. The gradient is then multiplied by one and its value will be maintained in the earlier layers. This enables us to stack many of such layers and create very deep architectures. Let `H=F(x)+x`, then:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial H}\frac{\partial H}{\partial x} = \frac{\partial L}{\partial H}(\frac{\partial F}{\partial x} + 1) = \frac{\partial L}{\partial H}\frac{\partial F}{\partial x} + \frac{\partial L}{\partial H}$$

Equation 10.9 Skip Connection Gradient

In this section, we'll be using the pre-trained on ImageNet convolutional base of ResNet50 CNN to encode every image in the Caltech 101 dataset. We start off by downloading the Caltech 101 dataset from http://www.vision.caltech.edu/datasets/ We select the pre-trained on ImageNet ResNet50 model as our base model: we set the output layer as the average pool layer, which effectively encodes an input image into a 2048-dimensional vector. We compute ResNet50 encoding of the dataset and store them in the activations list. To further save space, we compress the 2048-dimensional ResNet50 encoded vectors using Principal Component Analysis (PCA) down to 300-dimensional vectors. We retrieve the nearest neighbor images by sorting vectors according to cosine similarity. We are now ready to implement our image search architecture from scratch using Keras/TensorFlow.

Listing 10.3 ResNet50 Image Search

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras import Model
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import os
import random
from PIL import Image
from scipy.spatial import distance
from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"
DATA_PATH = "/content/drive/MyDrive/data/101_ObjectCategories/"

def get_closest_images(acts, query_image_idx, num_results=5):

    num_images, dim = acts.shape
    distances = []
    for image_idx in range(num_images):
        distances.append(distance.euclidean(acts[query_image_idx, :], acts[image_idx, :]))
    #end for
    idx_closest  = sorted(range(len(distances)), key=lambda k:
       distances[k])[1:num_results+1]

    return idx_closest

def get_concatenated_images(images, indexes, thumb_height):

    thumbs = []
    for idx in indexes:
        img = Image.open(images[idx])
        img = img.resize((int(img.width * thumb_height / img.height), int(thumb_height)),
       Image.ANTIALIAS)
        if img.mode != "RGB":
            img = img.convert("RGB")
        thumbs.append(img)
    concat_image = np.concatenate([np.asarray(t) for t in thumbs], axis=1)

    return concat_image

if __name__ == "__main__":

    num_images = 5000
```

```
images = [os.path.join(dp,f) for dp, dn, filenames in os.walk(DATA_PATH) for f in
   filenames \
          if os.path.splitext(f)[1].lower() in ['.jpg','.png','.jpeg']]
images = [images[i] for i in sorted(random.sample(range(len(images)), num_images))]

#CNN encodings
base_model = ResNet50(weights='imagenet')    #A
model = Model(inputs=base_model.input, outputs=base_model.get_layer('avg_pool').output)

activations = []
for idx, image_path in enumerate(images):
    if idx % 100 == 0:
        print('getting activations for %d/%d image...' %(idx,len(images)))
    img = image.load_img(image_path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)
    features = model.predict(x)
    activations.append(features.flatten().reshape(1,-1))

print('computing PCA...')
acts = np.concatenate(activations, axis=0)
pca = PCA(n_components=300)    #B
pca.fit(acts)
acts = pca.transform(acts)

print('image search...')
query_image_idx = int(num_images*random.random())
idx_closest = get_closest_images(acts, query_image_idx)
query_image = get_concatenated_images(images, [query_image_idx], 300)
results_image = get_concatenated_images(images, idx_closest, 300)

plt.figure()
plt.imshow(query_image)
plt.title("query image (%d)" %query_image_idx)
plt.show()

plt.figure()
plt.imshow(results_image)
plt.title("result images")
plt.show()
```

#A pre-trained on ImageNet ResNet50 model
#B reduce activation dimension

Figure 10.7 shows the query image of a chair (left) and a retrieved nearest neighbor image (right).

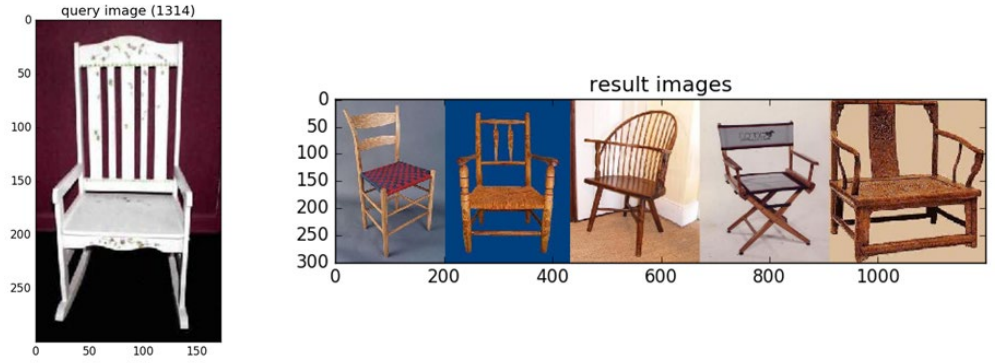


Figure 10.7 ResNet50 Image Search Results

The retrieved images closely resemble the query image. In the next section, we are going to look at neural networks for sequential data.

10.3 Recurrent Neural Nets

Recurrent Neural Nets (RNNs) are designed to process sequential data. RNNs maintain an internal state of the past and provide a natural way of encoding a sequence (Seq) into a vector (Vec) and vice versa. Application of RNNs range from language generation (Vec2Seq) to sequence classification (Seq2Vec) to sequence translation (Seq2Seq). In this section, we'll focus on sequence classification using bi-directional LSTM and sequence similarity using pre-trained word embeddings.

- RNNs use a hidden layer that incorporates current input `xt` and prior state `ht-1:`

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

Equation 10.10 LSTM cell: hidden layer output

where `Whh` are the hidden-to-hidden weights, `Wxh` are the input-to-hidden weights and `bh` is the bias term. Optionally, the outputs `yt` can be produced at every step

$$y_t = g(W_{hy}h_t + b_y)$$

Equation 10.11 LSTM cell: sequence output

The overall architecture is captured in Figure 10.8:

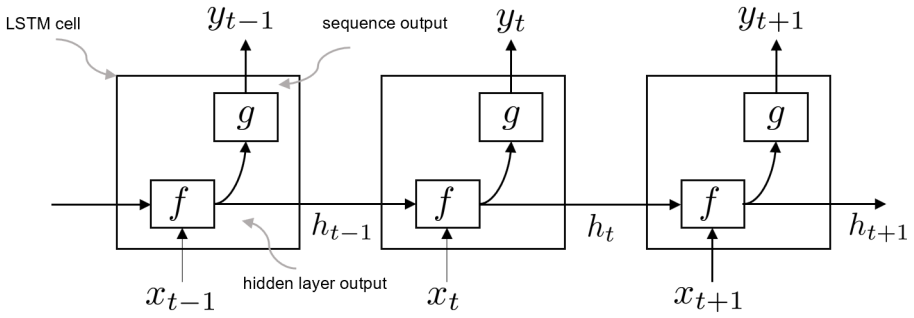


Figure 10.8 LSTM Recurrent Neural Network Architecture

Let's look at a few applications of RNNs.

10.3.1 LSTM Sequence Classification

In this section, our goal is to determine the sentiment of IMDB movie reviews. We start by tokenizing each word in the review and converting it into a vector via a trainable embedding layer. The sequence of word vectors forms the input to a forward LSTM model which encodes the sequence into a vector. In parallel, we input our sequence into a backward LSTM model and concatenate its vector output with the forward model to produce a latent representation of the movie review.

$$
\begin{aligned}
h_t^{\rightarrow} &= f(W_{hh}^{\rightarrow} h_{t-1}^{\rightarrow} + W_{xh}^{\rightarrow} x_t + b_h^{\rightarrow}) \\
h_t^{\leftarrow} &= f(W_{hh}^{\leftarrow} h_{t-1}^{\leftarrow} + W_{xh}^{\leftarrow} x_t + b_h^{\leftarrow}) \\
h_t &= [h_t^{\rightarrow}, h_t^{\leftarrow}]
\end{aligned}
$$

Equation 10.12 Hidden Layer Representation of bi-directional LSTM

where `ht` takes into account information from the past and the future. The two LSTMs combined are known as bi-directional LSTM and process input data as if time is running forward and backward. Figure 10.9 shows the architecture of bi-directional RNN.

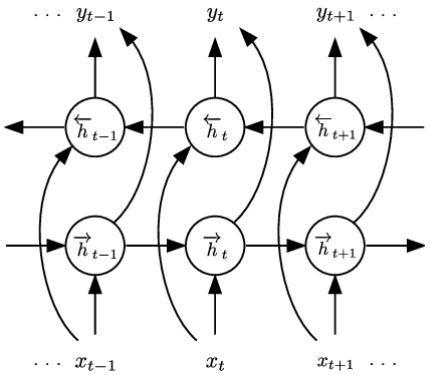


Figure 10.9 Bi-directional RNN architecture

Having encoded our review into a vector, we average several dense layers with l2 regularization and dropout to classify the sentiment as positive or negative via the sigmoid activation function in the output layer. Let's have a look at the code for LSTM sequence classification based on Keras/TensorFlow library.

Listing 10.4 Bi-diretional LSTM for Sentiment Classification

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

from keras.models import Sequential
from keras.layers import LSTM, Bidirectional
from keras.layers import Dense, Dropout, Activation, Embedding

from keras import regularizers
from keras.preprocessing import sequence
from keras.utils import np_utils

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):    #A
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)
```

```
if __name__ == "__main__":

    #load dataset
    max_words = 20000    #B
    seq_len = 200    #C
    (x_train, y_train), (x_val, y_val) = keras.datasets.imdb.load_data(num_words=max_words)

    x_train = keras.utils.pad_sequences(x_train, maxlen=seq_len)
    x_val = keras.utils.pad_sequences(x_val, maxlen=seq_len)

    batch_size = 256    #D
    num_epochs = 8    #D

    hidden_size = 64    #E
    embed_dim = 128    #E
    lstm_dropout = 0.2    #E
    dense_dropout = 0.5    #E
    weight_decay = 1e-3    #E

    #LSTM architecture
    model = Sequential()
    model.add(Embedding(max_words, embed_dim, input_length=seq_len))
    model.add(Bidirectional(LSTM(hidden_size, dropout=lstm_dropout,
       recurrent_dropout=lstm_dropout)))
    model.add(Dense(hidden_size, kernel_regularizer=regularizers.l2(weight_decay),
       activation='relu'))
    model.add(Dropout(dense_dropout))
    model.add(Dense(hidden_size/4, kernel_regularizer=regularizers.l2(weight_decay),
       activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    model.compile(
        loss=keras.losses.BinaryCrossentropy(),
        optimizer=tf.keras.optimizers.Adam(),
        metrics=["accuracy"]
    )

    model.summary()

    #define callbacks
    file_name = SAVE_PATH + 'lstm-weights-checkpoint.h5'
    checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
       save_best_only=True, mode='min')
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
    early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=16,
       verbose=1)
    #tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
    callbacks_list = [checkpoint, reduce_lr, early_stopping]

    hist = model.fit(x_train, y_train, batch_size=batch_size, epochs=num_epochs,
       callbacks=callbacks_list, validation_data=(x_val, y_val))    #F

    test_scores = model.evaluate(x_val, y_val, verbose=2)    #G

    print("Test loss:", test_scores[0])
    print("Test accuracy:", test_scores[1])

    plt.figure()
```

```
    plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
    plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
    plt.title('LSTM model')
    plt.xlabel('Epochs')
    plt.ylabel('Cross-Entropy Loss')
    plt.legend(loc='upper right')
    plt.show()

    plt.figure()
    plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
    plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
    plt.title('LSTM model')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend(loc='upper left')
    plt.show()

    plt.figure()
    plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
    plt.title('LSTM model')
    plt.xlabel('Epochs')
    plt.ylabel('Learning Rate')
    plt.legend()
    plt.show()
```

#A learning rate schedule
#B top 20K most frequent words
#C first 200 words of each movie review
#D training params
#E model parameters
#F model training
#G model evaluation

As we can see from Figure 10.10, we achieve test classification accuracy of `85%` on the IMDB movie review database.

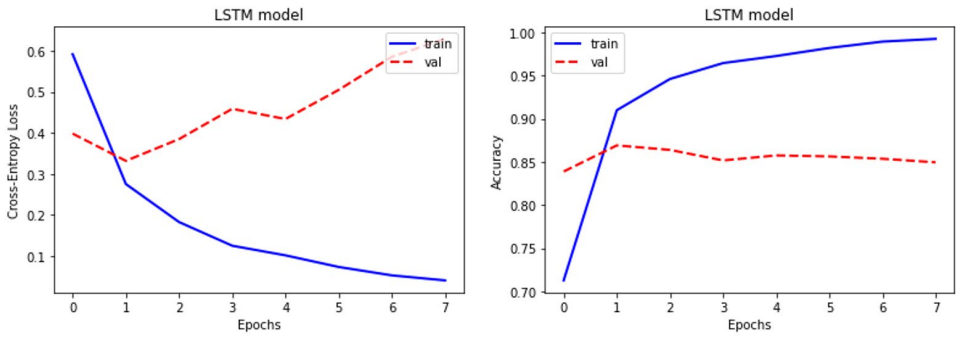


Figure 10.10 Bi-directional LSTM for sentiment classification loss and accuracy on training and validation datasets.

In the next section, we'll explore multiple-input neural network models.

10.3.2 Multi-Input Model

In this section, we study a multi-input model in application to sequence similarity. We are going to compare two questions (one for each input branch) and decide whether they have similar meaning or not. This will help us determine whether the questions are redundant or duplicate of each other. This challenge appeared as part of Quora Questions Pairs Kaggle data science competition and the model we'll develop could be used as part of an ensemble leading to a high performing solution.

In this case, we are going to use pre-trained Glove word embeddings (Pennington et al "Glove: Global Vectors for Word Representation", EMNLP, 2014) to encode each word into a 300-dimensional dense vector. We'll use 1-D convolutions and max-pooling operation to processes the sequence of data for each input branch and then concatenate the results into a single vector. After several dense layers, we will compute a probability of duplicate questions based on the sigmoid activation function.

Notice, the advantage of using 1-D convolutions is that they are computationally faster and can be done in parallel (since there's no recurrent loop to unroll). The accuracy will depend on whether the information in recent past is more important than distant past (in which case LSTMs will be more accurate) or whether the information is equally important in different time-frames of the past (in which case 1-D convolution will be more accurate).

We'll be using the Quora question pairs dataset available for download from Kaggle:

https://www.kaggle.com/datasets/quora/question-pairs-dataset Let's look at how we can implement multi-input sequence similarity model end-to-end in Keras/TensorFlow!

Listing 10.5 Multi-Input Neural Network Model for Sequence Similarity

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras

import os
import re
import csv
import codecs

from keras.models import Model
from keras.layers import Input, Flatten, Concatenate, LSTM, Lambda, Dropout
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import Conv1D, MaxPooling1D
from keras.layers import TimeDistributed, Bidirectional, BatchNormalization

from keras import backend as K
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer

from keras import regularizers
```

```
from keras.preprocessing import sequence
from keras.utils import np_utils

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"
DATA_PATH = "/content/drive/MyDrive/data/"

GLOVE_DIR = DATA_PATH
TRAIN_DATA_FILE = DATA_PATH + 'quora_train.csv'
TEST_DATA_FILE = DATA_PATH + 'quora_test.csv'
MAX_SEQUENCE_LENGTH = 30
MAX_NB_WORDS = 200000
EMBEDDING_DIM = 300
VALIDATION_SPLIT = 0.01

def scheduler(epoch, lr):
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

def text_to_wordlist(row, remove_stopwords=False, stem_words=False):
    # Clean the text, with the option to remove stopwords and to stem words.

    text = row['question']
    # Convert words to lower case and split them
    if type(text) is str:
        text = text.lower().split()
    else:
        return " "

    # Optionally, remove stop words
    if remove_stopwords:
        stops = set(stopwords.words("english"))
        text = [w for w in text if not w in stops]

    text = " ".join(text)

    # Clean the text
    text = re.sub(r"[^A-Za-z0-9^,!.\/'+-=]", " ", text)

    # Optionally, shorten words to their stems
    if stem_words:
        text = text.split()
        stemmer = SnowballStemmer('english')
        stemmed_words = [stemmer.stem(word) for word in text]
        text = " ".join(stemmed_words)

    # Return a list of words
    return(text)
```

```
if __name__ == "__main__":

    #load embeddings
    print('Indexing word vectors...')
    embeddings_index = {}
    f = codecs.open(os.path.join(GLOVE_DIR, 'glove.6B.300d.txt'), encoding='utf-8')    #A
    for line in f:
        values = line.split(' ')
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
    f.close()
    print('Found %s word vectors.' % len(embeddings_index))

    train_df = pd.read_csv(TRAIN_DATA_FILE)    #B
    test_df  = pd.read_csv(TEST_DATA_FILE)    #B

    q1df = train_df['question1'].reset_index()
    q2df = train_df['question2'].reset_index()
    q1df.columns = ['index', 'question']
    q2df.columns = ['index', 'question']
    texts_1 = q1df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    texts_2 = q2df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    labels = train_df['is_duplicate'].astype(int).tolist()
    print('Found %s texts.' % len(texts_1))
    del q1df
    del q2df

    q1df = test_df['question1'].reset_index()
    q2df = test_df['question2'].reset_index()
    q1df.columns = ['index', 'question']
    q2df.columns = ['index', 'question']
    test_texts_1 = q1df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    test_texts_2 = q2df.apply(text_to_wordlist, axis=1, raw=False).tolist()
    test_labels = np.arange(0, test_df.shape[0])
    print('Found %s texts.' % len(test_texts_1))
    del q1df
    del q2df

    #tokenize, convert to sequences and pad
    tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
    tokenizer.fit_on_texts(texts_1 + texts_2 + test_texts_1 + test_texts_2)
    sequences_1 = tokenizer.texts_to_sequences(texts_1)
    sequences_2 = tokenizer.texts_to_sequences(texts_2)
    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))

    test_sequences_1 = tokenizer.texts_to_sequences(test_texts_1)
    test_sequences_2 = tokenizer.texts_to_sequences(test_texts_2)

    data_1 = pad_sequences(sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
    data_2 = pad_sequences(sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
    labels = np.array(labels)
    print('Shape of data tensor:', data_1.shape)
    print('Shape of label tensor:', labels.shape)

    test_data_1 = pad_sequences(test_sequences_1, maxlen=MAX_SEQUENCE_LENGTH)
    test_data_2 = pad_sequences(test_sequences_2, maxlen=MAX_SEQUENCE_LENGTH)
    test_labels = np.array(test_labels)
```

```
del test_sequences_1
del test_sequences_2
del sequences_1
del sequences_2

#embedding matrix
print('Preparing embedding matrix...')
nb_words = min(MAX_NB_WORDS, len(word_index))

embedding_matrix = np.zeros((nb_words, EMBEDDING_DIM))
for word, i in word_index.items():
    if i >= nb_words:
        continue
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
print('Null word embeddings: %d' % np.sum(np.sum(embedding_matrix, axis=1) == 0))

#Multi-Input Architecture
embedding_layer = Embedding(nb_words,
                            EMBEDDING_DIM,
                            weights=[embedding_matrix],
                            input_length=MAX_SEQUENCE_LENGTH,
                            trainable=False)

sequence_1_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_1 = embedding_layer(sequence_1_input)
x1 = Conv1D(128, 3, activation='relu')(embedded_sequences_1)
x1 = MaxPooling1D(10)(x1)
x1 = Flatten()(x1)
x1 = Dense(64, activation='relu')(x1)
x1 = Dropout(0.2)(x1)

sequence_2_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences_2 = embedding_layer(sequence_2_input)
y1 = Conv1D(128, 3, activation='relu')(embedded_sequences_2)
y1 = MaxPooling1D(10)(y1)
y1 = Flatten()(y1)
y1 = Dense(64, activation='relu')(y1)
y1 = Dropout(0.2)(y1)

merged = Concatenate()([x1, y1])
merged = BatchNormalization()(merged)
merged = Dense(64, activation='relu')(merged)
merged = Dropout(0.2)(merged)
merged = BatchNormalization()(merged)
preds = Dense(1, activation='sigmoid')(merged)

model = Model(inputs=[sequence_1_input,sequence_2_input], outputs=preds)

model.compile(
    loss=keras.losses.BinaryCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(),
    metrics=["accuracy"]
)

model.summary()
```

```
    #define callbacks
    file_name = SAVE_PATH + 'multi-input-weights-checkpoint.h5'
    checkpoint = ModelCheckpoint(file_name, monitor='val_loss', verbose=1,
       save_best_only=True, mode='min')
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
    early_stopping = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=16,
       verbose=1)
    #tensor_board = TensorBoard(log_dir='./logs', write_graph=True)
    callbacks_list = [checkpoint, reduce_lr, early_stopping]

    hist = model.fit([data_1, data_2], labels, batch_size=1024, epochs=10,
       callbacks=callbacks_list, validation_split=VALIDATION_SPLIT)    #C

    num_test = 100000
    preds = model.predict([test_data_1[:num_test,:], test_data_2[:num_test,:]])    #D

    quora_submission = pd.DataFrame({"test_id":test_labels[:num_test],
       "is_duplicate":preds.ravel()})
    quora_submission.to_csv(SAVE_PATH + "quora_submission.csv", index=False)

    plt.figure()
    plt.plot(hist.history['loss'], 'b', lw=2.0, label='train')
    plt.plot(hist.history['val_loss'], '--r', lw=2.0, label='val')
    plt.title('Multi-Input model')
    plt.xlabel('Epochs')
    plt.ylabel('Cross-Entropy Loss')
    plt.legend(loc='upper right')
    plt.show()
    #plt.savefig('./figures/lstm_loss.png')

    plt.figure()
    plt.plot(hist.history['accuracy'], 'b', lw=2.0, label='train')
    plt.plot(hist.history['val_accuracy'], '--r', lw=2.0, label='val')
    plt.title('Multi-Input model')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend(loc='upper left')
    plt.show()
    #plt.savefig('./figures/lstm_acc.png')

    plt.figure()
    plt.plot(hist.history['lr'], lw=2.0, label='learning rate')
    plt.title('Multi-Input model')
    plt.xlabel('Epochs')
    plt.ylabel('Learning Rate')
    plt.legend()
    plt.show()
    #plt.savefig('./figures/lstm_learning_rate.png')
```

#A load embeddings
#B load dataset
#C model training
#D model evaluation

We can see the accuracy hovering around `69%` on the validation dataset. We can see that there's room for improvement. We can potentially increase the capacity of our model or improve data representation.

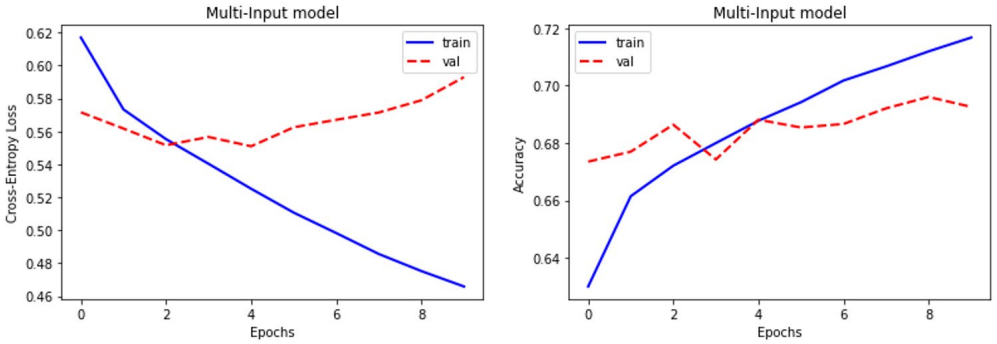


Figure 10.11 Multi-Input Model loss and accuracy for training and validation datasets

In this section, we focused on understanding an important class of neural nets: recurrent neural networks. We studied how they work and focused on two applications: sequence classification and sequence similarity. In the next section, we are going to look at different neural network optimizers.

10.4 Neural Network Optimizers

What are some of the popular optimization algorithm used for training neural networks? We will attempt to answer this question using a Convolutional Neural Network (CNN) trained on CIFAR-100 dataset with Keras/TensorFlow.

Stochastic Graident Descent (SGD) updates parameters `\theta` in the negative direction of the gradient `g` by taking a sub-set or a mini-batch of data of size `m`:

$$
\begin{aligned}
g &= \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}) \\
\theta &= \theta - \epsilon_k \times g
\end{aligned}
$$

Equation 10.13 SGD update

where `f(xi;\theta)` is a neural network trained on examples `xi` and labels `yi`, and `L` is the loss function. The gradient of the loss `L` is computed with respect to model parameters `\theta`. The learning rate `\epsilon_k` determines the size of the step that the algorithm takes along the gradient (in the negative direction in the case of minimization and in the positive direction in the case of maximization).

The learning rate is a function of iteration `k` and is the single most important hyper-parameter. A learning rate that is too high (e.g. `>0.1`) can lead to parameter updates that

miss the optimum value, a learning rate that is too low (e.g. `<1e-5`) will result in unnecessarily long training time. A good strategy is to start with a learning rate of `1e-3` and use a learning rate schedule that reduces the learning rate as a function of iterations. In general, we want the learning rate to satisfy the Robbins-Monroe conditions:

$$\sum_k \epsilon_k = \infty$$

$$\sum_k \epsilon_k^2 < \infty$$

Equation 10.14 Robbins-Monroe conditions for learning rate epsilon_k

The first condition ensures that the algorithm will be able to find a locally optimal solution regardless of the starting point and the second one controls oscillations.

Momentum accumulates exponentially decaying moving average of past gradients and continues to move in their direction, thus step size depends on how large and how aligned the sequence of gradients are, common values of momentum parameter `\alpha` are `0.5` and `0.9`:

$$v = \alpha v - \epsilon \nabla_\theta \Big(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta), y^{(i)}) \Big)$$

$$\theta = \theta + v$$

Equation 10.15 Momentum update

Nesterov Momentum inspired by Nesterov's accelerated gradient method, difference betweent Nesterov and standard momentum is where the gradient is evaluated, with Nesterov's momentum the gradient is evaluated after the current velocity is applied, thus Nesterov's momentum adds a correction factor to the gradient:

$$v = \alpha v - \epsilon \nabla_\theta \Big(\frac{1}{m} \sum_i L(f(x^{(i)}; \theta + \alpha \times v), y^{(i)}) \Big)$$

$$\theta = \theta + v$$

Equation 10.16 Nesterov Momentum update

AdaGrad is an adaptive method for setting the learning rate (Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR, 2011). Consider the two scenarios in Figure 10.12.

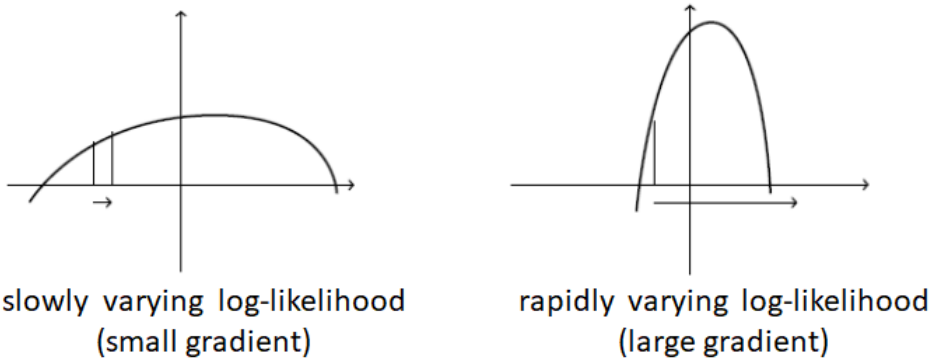


Figure 10.12 Slow varying (left) and rapidly varying (right) log likelihood

In the case of a slowly varying objective (left), the gradient would typically (at most points) have a small magnitude. As a result, we would need a large learning rate to quickly reach the optimum. In the case of a rapidly varying objective (right), the gradient would typically be very large. Using a large learning rate would result in very large steps, oscillating around but not reaching the optimum. These two situations occur because the learning rate is set independent of the gradient. AdaGrad solves this by accumulating squared norms of gradients seen so far and dividing the learning rate by the square root of this sum:

$$
\begin{aligned}
g &= \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}) \\
s &= s + g^T g \\
\theta &= \theta - \epsilon_k \times g/\sqrt{s + eps}
\end{aligned}
$$

Equation 10.17 AdaGrad update

As a result, parameters that receive high gradients will have their effective learning rate reduced and parameters that receive small gradients will have their effective learning rate increased. The net effect is greater progress in the more gently sloped directions of parameter space and more cautious updates in the presence of large gradients.

RMSProp modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average, i.e. it discards history from the distant past:

$$
\begin{aligned}
g &= \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}) \\
s &= \text{decay\_rate} \times s + (1 - \text{decay\_rate}) g^T g \\
\theta &= \theta - \epsilon_k \times g / \sqrt{s + eps}
\end{aligned}
$$

Equation 10.18 RMSProp update

Notice that AdaGrad implies a decreasing learning rate even if the gradients remain constant due to accumulation of gradients from the beginning of training. By introducing exponentially weighted moving average we are weighing recent past more heavily in comparison to distant past. As a result, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam derives from "adaptive moments", it can be seen as a variant on the combination of RMSProp and momentum, the update looks like RMSProp except that a smooth version of the gradient is used instead of the raw stochastic gradient, the full adam update also includes a bias correction mechanism (Kingma et al, "Adam: A Method for Stochastic Optimization, ICLR, 2015).

$$
\begin{aligned}
g &= \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)}) \\
m &= \beta_1 m + (1 - \beta_1) g \\
s &= \beta_2 v + (1 - \beta_2) g^T g \\
\theta &= \theta - \epsilon_k \times m / \sqrt{s + eps}
\end{aligned}
$$

Equation 10.19 Adam update

The recommended values in the paper are `\epsilon=1e-8`, `\beta1=0.9`, `\beta2=0.999`.

Let's examine the performance of different optimizers using Keras/TensorFlow!

Listing 10.6 Neural Network Optimizers

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras

from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D, Activation

from keras.callbacks import ModelCheckpoint
from keras.callbacks import TensorBoard
from keras.callbacks import LearningRateScheduler
from keras.callbacks import EarlyStopping

import math
import matplotlib.pyplot as plt

tf.keras.utils.set_random_seed(42)

SAVE_PATH = "/content/drive/MyDrive/Colab Notebooks/data/"

def scheduler(epoch, lr):
    if epoch < 4:
        return lr
    else:
        return lr * tf.math.exp(-0.1)

if __name__ == "__main__":

    img_rows, img_cols = 32, 32
    (x_train, y_train), (x_test, y_test) = keras.datasets.cifar100.load_data()    #A
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 3).astype("float32") /
       255
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 3).astype("float32") / 255

    y_train_label = keras.utils.to_categorical(y_train)
    y_test_label = keras.utils.to_categorical(y_test)
    num_classes = y_train_label.shape[1]

    batch_size = 256    #B
    num_epochs = 32    #B

    num_filters_l1 = 64    #C
    num_filters_l2 = 128    #C

    #CNN architecture
    cnn = Sequential()
    cnn.add(Conv2D(num_filters_l1, kernel_size = (5, 5), input_shape=(img_rows, img_cols,
       3), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    cnn.add(Conv2D(num_filters_l2, kernel_size = (5, 5), padding='same'))
    cnn.add(Activation('relu'))
    cnn.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
```

```
cnn.add(Flatten())
cnn.add(Dense(128))
cnn.add(Activation('relu'))

cnn.add(Dense(num_classes))
cnn.add(Activation('softmax'))

opt1 = tf.keras.optimizers.SGD()    #D
opt2 = tf.keras.optimizers.SGD(momentum=0.9, nesterov=True)    #D
opt3 = tf.keras.optimizers.RMSprop()    #D
opt4 = tf.keras.optimizers.Adam()    #D

optimizer_list = [opt1, opt2, opt3, opt4]

history_list = []

for idx in range(len(optimizer_list)):

    K.clear_session()

    opt = optimizer_list[idx]

    cnn.compile(
        loss=keras.losses.CategoricalCrossentropy(),
        optimizer=opt,
        metrics=["accuracy"]
    )

    #define callbacks
    reduce_lr = LearningRateScheduler(scheduler, verbose=1)
    callbacks_list = [reduce_lr]

    #training loop
    hist = cnn.fit(x_train, y_train_label, batch_size=batch_size, epochs=num_epochs,
   callbacks=callbacks_list, validation_split=0.2)
    history_list.append(hist)

#end for

plt.figure()
plt.plot(history_list[0].history['loss'], 'b', lw=2.0, label='SGD')
plt.plot(history_list[1].history['loss'], '--r', lw=2.0, label='SGD Nesterov')
plt.plot(history_list[2].history['loss'], ':g', lw=2.0, label='RMSProp')
plt.plot(history_list[3].history['loss'], '-.k', lw=2.0, label='ADAM')
plt.title('LeNet, CIFAR-100, Optimizers')
plt.xlabel('Epochs')
plt.ylabel('Cross-Entropy Training Loss')
plt.legend(loc='upper right')
plt.show()
#plt.savefig('./figures/lenet_loss.png')

plt.figure()
plt.plot(history_list[0].history['val_accuracy'], 'b', lw=2.0, label='SGD')
plt.plot(history_list[1].history['val_accuracy'], '--r', lw=2.0, label='SGD Nesterov')
plt.plot(history_list[2].history['val_accuracy'], ':g', lw=2.0, label='RMSProp')
plt.plot(history_list[3].history['val_accuracy'], '-.k', lw=2.0, label='ADAM')
plt.title('LeNet, CIFAR-100, Optimizers')
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
```

```
    plt.legend(loc='upper right')
    plt.show()
    #plt.savefig('./figures/lenet_loss.png')

    plt.figure()
    plt.plot(history_list[0].history['lr'], 'b', lw=2.0, label='SGD')
    plt.plot(history_list[1].history['lr'], '--r', lw=2.0, label='SGD Nesterov')
    plt.plot(history_list[2].history['lr'], ':g', lw=2.0, label='RMSProp')
    plt.plot(history_list[3].history['lr'], '-.k', lw=2.0, label='ADAM')
    plt.title('LeNet, CIFAR-100, Optimizers')
    plt.xlabel('Epochs')
    plt.ylabel('Learning Rate Schedule')
    plt.legend(loc='upper right')
    plt.show()
    #plt.savefig('./figures/lenet_loss.png')
```

#A cifar100 dataset
#B training parameters
#C model parameters
#D optimizers

Figure 10.13 shows the loss and validation dataset accuracy for LeNet neural network trained on cifar100 dataset using different optimizers.

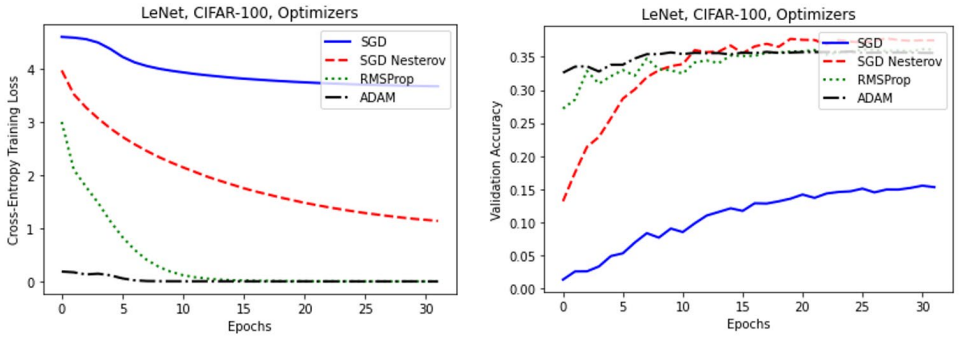


Figure 10.13 Cross-entropy loss and validation accuracy of LeNet trained on cifar100 using different optimizers.

We can see that Adam and Nesterov Momentum optimizers experimentally produce highest validation accuracy. In the following chapter, we will focus on advanced deep learning algorithms. We will learn to detect anomalies in time-series data using a Variational Auto-Encoder (VAE), determining clusters using a Mixture Density Network (MDN), learn to classify text with Transformers, and classify a citation graph using a Graph Neural Network (GNN).

10.4.1 Exercises

10.1 Explain the purpose of non-linearities in neural networks.

10.2 Explain the vanishing / exploding gradient problem.

10.3 Describe some of the ways to increase neural model capacity and avoid over-fitting.

10.4 Why does the number of filters increase in LeNet architecture as we go from input to output?

10.5 Explain how Adam optimizer works.

10.5 Summary

- Multi-Layer Perceptron (MLP) consist of multiple densely connected layers followed by a non-linearity.
- Cross-Entropy loss is used in classification tasks while Mean Square Error (MSE) loss is used in regression tasks.
- Neural networks are optimized via back-propagation algorithm which is based on the chain rule
- Increasing model capacity to avoid uner-fitting can be done by changing the model architecture (e.g. increasing the number of layers and number of hidden units per layer).
- Regularization to avoid over-fitting can occur on multiple levels: weight decay, early stopping, and dropout.
- Convolutional Neural Nets (CNNs) work exceptionally well for image data and use convolution and pooling layers instead of vectorized matrix multiplications.
- Classic CNN architectures consist of convolutional layer followed ReLU non-linear activation function followed by max pooling layer.
- Pre-trained CNNs can be used to extract feature vectors for images and used in applications such as image search
- Recurrent Neural Nets (RNNs) are designed to process sequential data.
- Application of RNNs include language generation (Vec2Seq), sequence classification (Seq2Vec) and sequence translation (Seq2Seq).
- Multi-Input Neural Nets can be constructed by concatenating the feature vector representations from individual input branches.
- Neural network optimizers include stochastic gradient descent (SGD), momentum, Nesterov momentum, AdaGrad, RMSProp, and Adam.