Study Notes

Yangyang Li
yangyang.li@northwestern.edu

Update on October 13, 2023

Contents

List of Figures

List of Theorems

List of Definitions

Acronyms

Preface

Contents

0.1   Features of this template

TeX, stylized within the system as LaTeX, is a typesetting system which was designed and written by Donald Knuth and first released in 1978. TeX is a popular means of typesetting complex mathematical formulae; it has been noted as one of the most sophisticated digital typographical systems.

- Wikipedia

0.1.1   crossref

different styles of clickable definitions and theorems

- nameref: Gaussian distribution

- autoref: Definition A.1, **??**

- cref: Definition A.1,

- hyperref: Gaussian,

0.1.2   ToC (Table of Content)

- mini toc of sections at the beginning of each chapter

- list of theorems, definitions, figures

- the chapter titles are bi-directional linked

0.1.3   header and footer

fancyhdr

- right header: section name and link to the beginning of the section

- left header: chapter title and link to the beginning of the chapter

- footer: page number linked to ToC of the whole document

0.1.4 bib

- titles of reference is linked to the publisher webpage e.g., [Kit+02]

- backref (go to the page where the reference is cited) e.g., [Chi09]

- customized video entry in reference like in [Bab16]

0.1.5 preface, index, quote (epigraph) and appendix

index page at the end of this document...

Part I

Machine Learning

Chapter 1

Probability

Contents

1.1   Basic Concepts

Permutation is an arrangement of objects in which the order is important.

$$P\left(n,r\right) = \frac{n!}{(n-r)!}$$

Combination is an arrangement of objects in which the order is not important.

$$C\left(n,r\right) = \frac{P\left(n,r\right)}{r!} = \frac{n!}{r!\left(n-r\right)!}$$

in which $0 \leq r \leq n$.

1.2   Maximum Likelihood Estimation

$$\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)^{\mathrm{T}}, \boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^{\mathrm{T}} \tag{1.1}$$

in which N is the number of samples, p is the number of features. The data is sampled from a distribution $p(\boldsymbol{x} \mid \theta)$, where θ is the parameter of the distribution.

For N i.i.d. samples, the likelihood function is $p(\boldsymbol{X} \mid \theta) = \prod_{i=1}^{N} p(\boldsymbol{x}_i \mid \theta))$

In order to get θ, we use Maximum Likelihood Estimation (MLE) to maximize the likelihood function.

$$\theta_{\mathtt{MLE}} = \operatorname*{argmax}_{\theta} \log p(\boldsymbol{X} \mid \theta) = \operatorname*{argmax}_{\theta} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid \theta) \tag{1.2}$$

1.3   Maximum A Posteriori Estimation

In Bayes' theorem, the θ is not a constant value, but $\theta \sim p(\theta)$. Hence,

$$p(\theta \mid \boldsymbol{X}) = \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{p(\boldsymbol{X})} = \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{\int_{\theta} p(\boldsymbol{X} \mid \theta)p(\theta)d\theta} \tag{1.3}$$

In order to get θ, we use Maximum A Posteriori Estimation (MAP) to maximize the posterior function.

$$\theta_{\texttt{MAP}} = \operatorname*{argmax}_{\theta} p(\theta \mid \boldsymbol{X}) = \operatorname*{argmax}_{\theta} \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{p(\boldsymbol{X})} \tag{1.4}$$

After θ is estimated, then calculating $\frac{p(\boldsymbol{X} \mid \theta) \cdot p(\theta)}{\int_{\theta} p(\boldsymbol{X} \mid \theta)p(\theta)d\theta}$ to get the posterior distribution. We can use the posterior distribution to predict the probability of a new sample $\boldsymbol{x}$.

$$p(x_{\texttt{new}} \mid \boldsymbol{X}) = \int_{\theta} p(x_{\texttt{new}} \mid \theta) \cdot p(\theta \mid \boldsymbol{X})d\theta \tag{1.5}$$

1.4   Gaussian Distribution

Gaussian distribution is also called normal distribution.

$$\theta = (\mu, \sigma^2), \quad \mu = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{1.6}$$

For MLE,

$$\theta = (\mu, \Sigma) = (\mu, \sigma^2), \quad \theta_{\texttt{MLE}} = \operatorname*{argmax}_{\theta} \log p(\boldsymbol{X} \mid \theta) = \operatorname*{argmax}_{\theta} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid \theta) \tag{1.7}$$

Generally, the Probability Density Function (PDF) of a Gaussian distribution is:

$$p(x \mid \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^{\mathrm{T}}\Sigma^{-1}(\boldsymbol{x} - \mu)\right) \tag{1.8}$$

in which μ is the mean vector, Σ is the covariance matrix, det is the determinant of matrix. det is the product of all eigenvalues of a matrix.

Hence,

$$\log p(\boldsymbol{X} \mid \theta) = \sum_{i=1}^{N} \log p(x_i \mid \theta) = \sum_{i=1}^{N} \log \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^{\mathrm{T}}\Sigma^{-1}(\boldsymbol{x} - \mu)\right) \tag{1.9}$$

Let's only consider 1 dimension case for brevity, then

$$\log p(\boldsymbol{X} \mid \theta) = \sum_{i=1}^{N} \log p(x_i \mid \theta) = \sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x - \mu)^2}{\sigma^2}\right) \tag{1.10}$$

Let's get the optimal value for μ,

$$\mu_{\text{MLE}} = \underset{\mu}{\text{argmax}} \log p(\boldsymbol{X} \mid \theta) = \underset{\mu}{\text{argmin}} \sum_{i=1}^{N} \frac{1}{2} (x_i - \mu)^2 \tag{1.11}$$

So,

$$\frac{\partial \log p(\boldsymbol{X} \mid \theta)}{\partial \mu} = \sum_{i=1}^{N} (\mu - x_i) = 0 \rightarrow \mu_{\text{MLE}} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{1.12}$$

Let's get the optimal value for σ^2,

$$
\begin{aligned}
\sigma_{\text{MLE}} &= \underset{\sigma}{\text{argmax}} \log p(\boldsymbol{X} \mid \theta) \\
&= \underset{\sigma}{\text{argmax}} \sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x - \mu)^2}{\sigma^2}\right) \\
&= \underset{\sigma}{\text{argmax}} \sum_{i=1}^{N} \left[-\log \sqrt{2\pi\sigma^2} - \frac{(x - \mu)^2}{2\sigma^2} \right] \\
&= \underset{\sigma}{\text{argmin}} \sum_{i=1}^{N} \left[\log \sigma + \frac{(x - \mu)^2}{2\sigma^2} \right]
\end{aligned}
$$

Hence,

$$\frac{\partial}{\partial \sigma} \sum_{i=1}^{N} \left[\log \sigma + \frac{(x_i - \mu)^2}{2\sigma^2} \right] = 0 \rightarrow \sigma_{\text{MLE}}^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \tag{1.13}$$

$\mathbb{E}_D\left[\mu_{\text{MLE}}\right]$ is unbaised.

$$\mathbb{E}_D\left[\mu_{\text{MLE}}\right] = \mathbb{E}_D\left[\frac{1}{N} \sum_{i=1}^{N} x_i\right] = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_D\left[x_i\right] = \frac{1}{N} \sum_{i=1}^{N} \mu = \mu \tag{1.14}$$

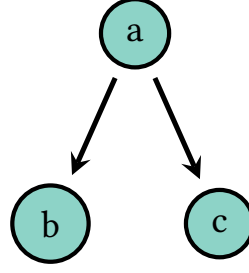However, $\mathbb{E}_D\left[\sigma_{\text{MLE}}^2\right]$ is biased.

Figure 1.1: A simple Bayesian network.

$$\mathbb{E}_D\left[\sigma_{\text{MLE}}^2\right] = \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu_{\text{MLE}}\right)^2\right] \tag{1.15}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu_{\text{MLE}}\right)^2\right] \tag{1.16}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i^2 - 2x_i\mu_{\text{MLE}} + \mu_{\text{MLE}}^2\right)\right] = \mathbb{E}_D\left[\sum_{i=1}^{N}x_i^2 - 2\frac{1}{N}\sum_{i=1}^{N}x_i\mu_{\text{MLE}} + \mu_{\text{MLE}}^2\right] \tag{1.17}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i^2 - \mu^2\right) + \mu^2 - \mu_{\text{MLE}}^2\right] \tag{1.18}$$

$$= \sigma^2 - \mathbb{E}_D\left[\mu_{\text{MLE}}^2 - \mu^2\right] \tag{1.19}$$

$$= \sigma^2 - \left(\mathbb{E}_D\left[\mu_{\text{MLE}}^2\right] - \mathbb{E}_D\left[\mu_{\text{MLE}}^2\right]\right) \tag{1.20}$$

$$= \sigma^2 - \text{Var}\left[\mu_{\text{MLE}}\right] = \sigma^2 - \text{Var}\left[\frac{1}{N}\sum_{i=1}^{N}x_i\right] \tag{1.21}$$

$$= \sigma^2 - \frac{1}{N^2}\sum_{i=1}^{N}\text{Var}\left[x_i\right] = \frac{N-1}{N}\sigma^2 \tag{1.22}$$

$$\tag{1.23}$$

1.5 Bayesian Network

1.6 Probability Graph

> this section is not finished yet. Need to be reviewed p54

1.6.1 Variables Elimination

1.6.2 Belief propagation

Belief propagation is mainly used for tree data structure, and equals Section with caching.
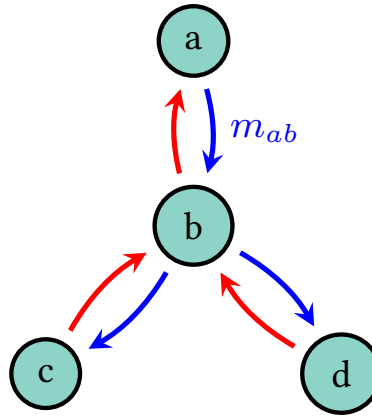
Figure 1.2: Belief propagation.

1.6.3   Max-product Algorithm

1.6.4   Factor Graph

1.7   Expectation Maximum

$$\Theta^{(t+1)} = \operatorname*{argmax}_{\Theta} \int_Z \log P(x, z \mid \theta) \cdot P(z \mid x, \Theta^{(t)}) \, dz$$

1.8   Gaussian Mixture Model

$$Q\left(\Theta, \Theta^{(t)}\right) = \int_Z \log P(x, z \mid \theta) \cdot P(z \mid x, \Theta^{(t)}) \, dz$$

1.9   Hidden Markov Model

Chapter 2

Notes

2.1 Complexity of matrix multiply

Assume A is $m \times n$ and B is $n \times p$. The naive algorithm takes $O\left(mnp\right)$ time.

2.2 Representation of matrix

row major, col major, stride

Part II

Algorithm and Data Structure

Chapter 3

Algorithm

Contents

3.1   Graph

Part III

Programming

Chapter 4

C++

Contents

4.1   Code Snippets

4.1.1   Random Number Generation

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>

int main() {
  std::random_device rd;
  std::mt19937 rng(rd());
  std::uniform_int_distribution<int> dist6(1, 6);
  std::generate_n(std::ostream_iterator<int>(std::cout, " "), 10,
                  [&dist6, &rng]() { return dist6(rng); });
  return 0;
}
```

4.1.2   Quick Sort

```cpp
#include <vector>
using std::vector;

class Solution {
public:
    void sort(vector<int>& nums, int low, int high){
        if (low >= high)  return;

        int p = partition(nums, low, high);

        sort(nums, low, p - 1); // Changed p to p-1
```

```cpp
        sort(nums, p + 1, high);
    }

    int partition(vector<int>& nums, int low, int high){
        int pivot  = low, l = pivot + 1, r = high;

        while(l <= r) {
            if (nums[l] < nums[pivot]) ++l;
            else if (nums[r] >= nums[pivot]) --r;
            else std::swap(nums[l], nums[r]);
        }
        std::swap(nums[pivot], nums[r]);
        return r;
    }

    void shuffle(vector<int>& nums){
        std::srand((unsigned) time(nullptr));
        int n = nums.size();

        for (int i = 0; i < n; ++i){
            int r = i + rand() % (n - i);
            std::swap(nums[i], nums[r]);
        }
    }

    vector<int> sortArray(vector<int>& nums) {
        shuffle(nums); // Shuffle the array before sorting
        sort(nums, 0, nums.size() - 1); // Changed nums.size() to nums.size() - 1
        return nums;
    }
};
```

On average, the algorithm takes $O(n \log n)$ time. In the worst case, it takes $O(n^2)$ time. We shuffle the array before sorting to avoid the worst case.

Chapter 5

Rust

Part IV

Research

Chapter 6

Paper Reading

Chapter 7

Dive into Deep Learning

7.1 Linear Neural Networks

7.1.1 Exercise 3.1.6

Exercise 1: Let's solve this analytically.

To minimize the sum of squared differences, we take the derivative of the function with respect to (b), set it equal to zero, and solve for (b).

The function to minimize is:

$$f(b) = \sum_{i=1}^{n}(x_i - b)^2$$

Taking the derivative with respect to (b) gives:

$$f'(b) = \sum_{i=1}^{n} -2(x_i - b)$$

Setting this equal to zero and solving for (b) gives:

$$0 = \sum_{i=1}^{n} -2(x_i - b)$$

Solving for (b) gives:

$$b = \frac{1}{n}\sum_{i=1}^{n} x_i$$

So the value of (b) that minimizes the sum of squared differences is the mean of the x_i.

This problem and its solution relate to the normal distribution because the sum of squared differences is the basis for the maximum likelihood estimation of the mean of a normal distribution. In other words, the mean of a normal distribution is the value that maximizes the likelihood of the observed data, which is equivalent to minimizing the sum of squared differences from the mean.

If we change the loss function to the sum of absolute differences, the problem becomes finding the median of the x_i. This is because the median is the value that minimizes the sum of absolute differences. Unlike the mean, the median is not sensitive to extreme values, so it is a more robust measure of central tendency when there are outliers in the data.

Exercise 2:   An affine function is a function composed of a linear transformation and a translation. 'In' the context of machine learning, an affine function often takes the form $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b)$, where $\mathbf{x}$ is an input vector, $\mathbf{w}$ is a weight vector, and (b) is a bias term.

We can show that this affine function is equivalent to a linear function on the augmented vector $(\mathbf{x}, 1)$ by considering a new weight vector $\mathbf{w}' = (\mathbf{w}, b)$ and an augmented input vector $\mathbf{x}' = (\mathbf{x}, 1)$. Then the affine function can be written as a linear function:

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b = \mathbf{x'}^\top \mathbf{w'}$$

Here's the proof:

The original affine function is $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b$.

We define $\mathbf{x}' = (\mathbf{x}, 1))and(\mathbf{w}' = (\mathbf{w}, b)$.

The dot product $\mathbf{x'}^\top \mathbf{w'})is(\mathbf{x}^\top \mathbf{w} + 1 \cdot b$, which is exactly the original affine function.

Therefore, the affine function $\mathbf{x}^\top \mathbf{w} + b$ is equivalent to the linear function $\mathbf{x'}^\top \mathbf{w'}$ on the augmented vector $(\mathbf{x}, 1)$. This equivalence is often used in machine learning to simplify the notation and the implementation of algorithms.

Exercise 3:   A quadratic function of the form $f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$ can be implemented in a deep network using a fully connected layer followed by an element-wise multiplication operation and another fully connected layer. Here's how you can do it:

Input Layer: The input to the network is the vector $\mathbf{x}$.

First Fully Connected Layer: This layer applies a linear transformation to the input vector. The weights of this layer represent the w_i terms in the quadratic function. The output of this layer is $b + \sum_i w_i x_i$.

Element-wise Multiplication Layer: This layer computes the element-wise product of the input vector with itself, resulting in the vector $\mathbf{x} \odot \mathbf{x}$, where $\odot$ denotes element-wise multiplication. This operation corresponds to the $x_i x_j$ terms in the quadratic function.

Second Fully Connected Layer: This layer applies a linear transformation to the output of the element-wise multiplication layer. The weights of this layer represent the w_{ij} terms in the quadratic function. The output of this layer is $\sum_{j \leq i} w_{ij} x_i x_j$.

Summation Layer: This layer adds the outputs of the first fully connected layer and the second fully connected layer to produce the final output of the network, which is the value of the quadratic function.

Note that this network does not include any activation functions, because the quadratic function is a polynomial function, not a nonlinear function. If you want to model a more complex relationship between the input and the output, you can add activation functions to the network.

Exercise 4:   If the design matrix $\mathbf{X}^\top \mathbf{X}$ does not have full rank, it means that there are linearly dependent columns in the matrix $\mathbf{X}$. This can lead to several problems:

The matrix $\mathbf{X}^\top \mathbf{X}$ is not invertible, which means that the normal equations used to solve the linear regression problem do not have a unique solution. This makes it impossible to find a unique set of regression coefficients that minimize the sum of squared residuals.

The presence of linearly dependent columns in $\mathbf{X}$ can lead to overfitting, as the model can "learn" to use these redundant features to perfectly fit the training data, but it will not generalize well to new data.

One common way to fix this issue is to add a small amount of noise to the entries of $\mathbf{X}$. This can help to make the columns of $\mathbf{X}$ linearly independent, which ensures that $\mathbf{X}^\top \mathbf{X}$ has full rank and is invertible. Another common approach is to use regularization techniques, such as ridge regression or lasso regression, which add a penalty term to the loss function that discourages the model from assigning too much importance to any one feature.

If you add a small amount of coordinate-wise independent Gaussian noise to all entries of $\mathbf{X}$, the expected value of the design matrix $\mathbf{X}^\top \mathbf{X}$ remains the same. This is because the expected value of a Gaussian random

variable is its mean, and adding a constant to a random variable shifts its mean by that constant. So if the noise has mean zero, the expected value of $\mathbf{X}^\top \mathbf{X}$ does not change.

Stochastic gradient descent (SGD) can still be used when $\mathbf{X}^\top \mathbf{X}$ does not have full rank, but it may not converge to a unique solution. This is because SGD does not rely on the invertibility of $\mathbf{X}^\top \mathbf{X}$, but instead iteratively updates the regression coefficients based on a randomly selected subset of the data. However, the presence of linearly dependent columns in $\mathbf{X}$ can cause the loss surface to have multiple minima, which means that SGD may converge to different solutions depending on the initial values of the regression coefficients.

Exercise 5: The negative log-likelihood of the data under the model is given by:

The likelihood function for the data is $P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} \frac{1}{2} \exp\big(-|y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|\big)$, where y_i is the (i)-th target value, $\mathbf{x}_i$ is the (i)-th input vector, $\mathbf{w}$ is the weight vector, and (b) is the bias term.

Taking the negative logarithm of this gives the negative log-likelihood:

$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \big(\log 2 + |y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|\big)$.

Unfortunately, there is no closed-form solution for this problem. The absolute value in the log-likelihood function makes it non-differentiable at zero, which means that we cannot set its derivative equal to zero and solve for $\mathbf{w}$ and b.

A minibatch stochastic gradient descent algorithm to solve this problem would involve the following steps: Initialize $\mathbf{w}$ and b with random values. For each minibatch of data: Compute the gradient of the negative log-likelihood with respect to $\mathbf{w}$ and b. The gradient will be different depending on whether $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$ is positive or negative. Update $\mathbf{w}$ and b by taking a step in the direction of the negative gradient. Repeat until the algorithm converges. One issue that could arise with this algorithm is that the updates could become very large if $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$ is close to zero, because the gradient of the absolute value function is not defined at zero. This could cause the algorithm to diverge.

One possible solution to this problem is to use a variant of gradient descent that includes a regularization term, such as gradient descent with momentum or RMSProp. These algorithms modify the update rule to prevent the updates from becoming too large. Another solution is to add a small constant to the absolute value inside the logarithm to ensure that it is always differentiable.

Exercise 6: The composition of two linear layers in a neural network essentially results in another linear layer. This is due to the property of linearity: the composition of two linear functions is another linear function.

Mathematically, if we have two linear layers L_1 and L_2 such that $L_1(x) = Ax + b$ and $L_2(x) = Cx + d$, then the composition $L_2(L_1(x)) = L_2(Ax + b) = C(Ax + b) + d = (CA)x + (Cb + d)$, which is another linear function.

This means that a neural network with two linear layers has the same expressive power as a neural network with a single linear layer. It cannot model complex, non-linear relationships between the input and the output.

To overcome this limitation, we typically introduce non-linearities between the linear layers in the form of activation functions, such as the ReLU (Rectified Linear Unit), sigmoid, or tanh functions. These non-linear activation functions allow the neural network to model complex, non-linear relationships and greatly increase its expressive power.

Exercise 7: Regression for realistic price estimation of houses or stock prices can be challenging due to the complex nature of these markets. Prices can be influenced by a wide range of factors, many of which may not be easily quantifiable or available for use in a regression model.

The additive Gaussian noise assumption may not be appropriate for several reasons. First, prices cannot be negative, but a Gaussian distribution is defined over the entire real line, meaning it allows for negative values. Second, the Gaussian distribution is symmetric, but price changes in real markets are often not symmetric. For

example, prices may be more likely to increase slowly but decrease rapidly, leading to a skewed distribution of price changes. Finally, the Gaussian distribution assumes that large changes are extremely unlikely, but in real markets, large price fluctuations can and do occur.

Regression to the logarithm of the price can be a better approach because it can help to address some of the issues with the Gaussian noise assumption. Taking the logarithm of the prices can help to stabilize the variance of the price changes and make the distribution of price changes more symmetric. It also ensures that the predicted prices are always positive, since the exponential of any real number is positive.

When dealing with penny stocks, or stocks with very low prices, there are several additional factors to consider. One issue is that the prices of these stocks may not be able to change smoothly due to the minimum tick size, or the smallest increment by which the price can change. This can lead to a discontinuous distribution of price changes, which is not well modeled by a Gaussian distribution. Another issue is that penny stocks are often less liquid than higher-priced stocks, meaning there may not be enough buyers or sellers to trade at all possible prices. This can lead to large price jumps, which are also not well modeled by a Gaussian distribution.

The Black-Scholes model for option pricing is a celebrated model in financial economics that makes specific assumptions about the distribution of stock prices. It assumes that the logarithm of the stock price follows a geometric Brownian motion, which implies that the stock price itself follows a log-normal distribution. This model has been very influential, but it has also been criticized for its assumptions, which may not hold in real markets. For example, it assumes that the volatility of the stock price is constant, which is often not the case in reality.

Exercise 8:   The Gaussian additive noise model may not be appropriate for estimating the number of apples sold in a grocery store for several reasons: The Gaussian model allows for negative values, but the number of apples sold cannot be negative. The Gaussian model is a continuous distribution, but the number of apples sold is a discrete quantity. The Gaussian model assumes that large deviations from the mean are extremely unlikely, but in reality, there could be large fluctuations in the number of apples sold due to various factors (e.g., seasonal demand, promotions).

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events (in this case, the number of apples sold) occurring in a fixed interval of time or space. The parameter λ is the rate at which events occur. The expected value of a Poisson-distributed random variable is indeed λ. This can be shown as follows:

The expected value (E[k]) of a random variable (k) distributed according to a Poisson distribution is given by:

$$E[k] = \sum_{k=0}^{\infty} k \cdot p(k \mid \lambda) = \sum_{k=0}^{\infty} k \cdot \frac{\lambda^k e^{-\lambda}}{k!}$$

By the properties of the Poisson distribution, this sum equals λ.

The loss function associated with the Poisson distribution is typically the negative log-likelihood. Given observed counts $y_1, y_2, \ldots, y_n$ and predicted rates $\lambda_1, \lambda_2, \ldots, \lambda_n$, the negative log-likelihood is:

$$L(\lambda, y) = \sum_{i=1}^{n} (\lambda_i - y_i \log \lambda_i)$$

If we want to estimate $\log \lambda$ instead of λ, we can modify the loss function accordingly. One possible choice is to use the squared difference between the logarithm of the predicted rate and the logarithm of the observed count:

$$L(\log \lambda, y) = \sum_{i=1}^{n} (\log \lambda_i - \log y_i)^2$$

This loss function penalizes relative errors in the prediction of the rate, rather than absolute errors. It can be more appropriate when the counts vary over a wide range, as it gives equal weight to proportional errors in the prediction of small and large counts.

7.1.2   Exercise 3.3.5

1. What will happen if the number of examples cannot be divided by the batch size. How would you change this behavior by specifying a different argument by using the framework's API?   PyTorch: In PyTorch's *DataLoader*, there's an argument called *drop_last*. If set to True, it will drop the last incomplete batch. By default, it's set to *False*.

```
train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
    shuffle=True, drop_last=True)
```

2. Suppose that we want to generate a huge dataset, where both the size of the parameter vector w and the number of examples num_examples are large.   1. What happens if we cannot hold all data in memory?

2. How would you shuffle the data if it is held on disk? Your task is to design an efficient algorithm that does not require too many random reads or writes. Hint: pseudorandom permutation generators 75 allow you to design a reshuffle without the need to store the permutation table explicitly (Naor and Reingold, 1999).

1. What happens if we cannot hold all data in memory?

If we cannot hold all the data in memory:

Performance Impact: Reading data directly from disk is much slower than reading from memory. If the training algorithm frequently accesses the disk, it can significantly slow down the training process.

Out-of-Core Processing: You'll need to use "out-of-core" or "external memory" algorithms, which are designed to process data that is too large to fit into a computer's main memory. These algorithms break the data into smaller chunks that can fit into memory, process each chunk, and then combine the results.

Streaming: Data can be streamed from the disk in mini-batches. Only one mini-batch is loaded into memory at a time, and after processing it, the next mini-batch is loaded. This is common in deep learning when dealing with large datasets.

Distributed Computing: Another approach is to distribute the dataset across multiple machines in a cluster. Each machine processes a subset of the data. Frameworks like TensorFlow and PyTorch have support for distributed training.

2. How would you shuffle the data if it is held on disk?

Shuffling large datasets on disk without too many random reads or writes can be challenging. Here's a method using pseudorandom permutation generators:

1. Pseudorandom Permutation: Use a pseudorandom permutation generator to generate a sequence of indices that represents the shuffled order of your data. The beauty of pseudorandom permutation generators is that they can generate the nth element of the sequence without generating the first n-1 elements, allowing for efficient random access.

2. Sequential Reads and Writes: - Read the data from the disk sequentially in large chunks (to benefit from sequential read speeds). - For each item in the chunk, use the pseudorandom permutation generator to determine its new location in the shuffled dataset. - Write the shuffled data back to the disk sequentially in large chunks.

3. In-Place Shuffling: If you have some memory available (but not enough to hold the entire dataset), you can load chunks of data into memory, shuffle them using the pseudorandom permutation, and then write them back to disk. This reduces the number of disk writes.

4. External Sorting: Another approach is to use techniques from external sorting. Divide the data into chunks that fit into memory, shuffle each chunk in memory, and then merge the chunks together in a shuffled manner.

5. Temporary Storage: If possible, use a fast intermediate storage (like an SSD) to temporarily store data chunks during the shuffling process. This can speed up both reads and writes.

Remember, the key is to minimize random disk accesses, as they are much slower than sequential accesses. Leveraging pseudorandom permutations allows for an efficient reshuffling without the need to store large permutation tables explicitly.

3. Implement a data generator that produces new data on the fly, every time the iterator is called.

```python
import torch
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):
    def __init__(self, num_samples, tensor_size):
        """
        Args:
            num_samples (int): Number of samples in the dataset.
            tensor_size (tuple): Size of the tensor to be generated.
        """
        self.num_samples = num_samples
        self.tensor_size = tensor_size

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Generate a random tensor on the fly
        sample = torch.randn(self.tensor_size)
        return sample

# Parameters
num_samples = 1000
tensor_size = (3, 32, 32)  # Example: 3-channel, 32x32 image

# Create dataset and dataloader
random_dataset = RandomDataset(num_samples, tensor_size)
dataloader = DataLoader(random_dataset, batch_size=32, shuffle=True)

# Iterate over the dataloader
for batch in dataloader:
    print(batch.shape)  # Should print torch.Size([32, 3, 32, 32])
                        # for all batches except possibly the last one
```

4. How would you design a random data generator that generates the same data each time it is called?

```python
torch.manual_seed(idx)
```

Appendix A

Formulas

A.1 Gaussian distribution

Definition A.1 (Gaussian distribution). *Gaussian distribution*

Theorem A.1 (Central limit theorem).

Bibliography

[Bab16]   László Babai. "Graph Isomorphism in Quasipolynomial Time". Jan. 19, 2016. arXiv: 1512.03547 [cs, math] (cit. on p. 10). Online video

[Chi09]   Andrew M. Childs. *Universal Computation by Quantum Walk*. Physical Review Letters 102.18 (May 4, 2009), p. 180501. arXiv: 0806.1972 (cit. on p. 10).

[Kit+02]   Alexei Yu Kitaev et al. *Classical and quantum computation*. 47. American Mathematical Soc., 2002 (cit. on p. 10).

Alphabetical Index