# Probability for Machine Learning

## Discover How To Harness Uncertainty With Python

---

Jason Brownlee

MACHINE LEARNING MASTERY

## Disclaimer



## Acknowledgements

Special thanks to my copy editor Sarah Martin and my technical editors Michael Sanderson and Arun Koshy.

## Copyright



Edition: v1.9

# Contents

# Preface

Probability is foundational to machine learning and required background for machine learning practitioners.

- Probability is a prerequisite in most courses and books on applied machine learning.
- Probability methods are used at each step in an applied machine learning project.
- Probabilistic frameworks underlie the training of many machine learning algorithms.

Data distributions and statistics don't make sense without probability. Fitting models on a training dataset does not make sense without probability. Interpreting the performance of a stochastic learning algorithm does not make sense without probability. A machine learning practitioner cannot be effective without an understanding and appreciation of the basic concepts and methods from the field of probability.

## Practitioners Don't Know Probability

Developers don't know probability, and this is a huge problem. Programmers don't need to know or use probability in order to develop software. Software engineering and computer science courses focus on deterministic programs, with inputs, outputs, and no randomness or noise. As such, it is common for machine learning practitioners coming from the computer science or developer tradition not to know and not value probabilistic thinking. This is a problem given that the bedrock of a predictive modeling project is probability.

## Practitioners Study the Wrong Probability

Eventually, machine learning practitioners realize the need for skills in probability. This might start with a need to better interpret descriptive statistics and may progress to the need to understand the probabilistic frameworks behind many popular machine learning algorithms. The problem is, they don't seek out the probability information they need. Instead, they try to read through a textbook on probability or work through the material for an undergraduate course on probabilistic methods. This approach is slow, it's boring, and it covers a breadth and depth of material on probability that is beyond the needs of the machine learning practitioner.

## Practitioners Study Probability Wrong

It's worse than this. Regardless of the medium used to learn probability, be it books, videos, or course material, machine learning practitioners study probability the wrong way. Because the material is intended for undergraduate students that need to pass a test, the material is focused on the math, theory, proofs, and derivations. This is great for testing students but terrible for practitioners that need results. Practitioners need methods that clearly state when they are appropriate and instruction on how to interpret the results. They need intuitions behind the complex equations. They need code examples that they can use immediately on their project.

## A Better Way

I set out to write a playbook for machine learning practitioners that gives them only those parts of probability that they need to be more effective at working through a predictive modeling project. I set out to present probability methods in the way that practitioners learn: that is with simple language and working code examples. Probability is important to machine learning, and I believe that if it is taught at the right level for practitioners, it can be a fascinating, fun, directly applicable, and immeasurably useful area of study. I hope that you agree.

Jason Brownlee
2020

# Part I

# Introduction

# Welcome

Welcome to *Probability for Machine Learning*. The field of probability is hundreds of years old and probabilistic methods are central to working through predictive modeling problems with machine learning. The simplest concepts when working with data come from the field of probability, such as the most likely value and the idea of a probability distribution over observations. Building on these simple ideas, probabilistic frameworks like maximum likelihood estimation underly a range of machine learning algorithms, from logistic regression to neural networks. I designed this book to teach you step-by-step the basics of probability with concrete and executable examples in Python.

## Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

- You know your way around basic Python for programming.
- You may know some basic NumPy for array manipulation.
- You want to learn probability to deepen your understanding and application of machine learning.

This guide was written in the top-down and results-first machine learning style that you're used to from Machine Learning Mastery.

## About Your Outcomes

This book will teach you the basics of probability that you need to know as a machine learning practitioner. After reading and working through this book, you will know:

- About the field of probability, how it relates to machine learning, and how to harness probabilistic thinking on a machine learning project.
- How to calculate different types of probability, such as joint, marginal, and conditional probability.

- How to consider data in terms of random variables and how to recognize and sample from common discrete and continuous probability distribution functions.
- How to frame learning as maximum likelihood estimation and how this important probabilistic framework is used for regression, classification, and clustering machine learning algorithms.
- How to use probabilistic methods to evaluate machine learning models directly without evaluating their performance on a test dataset.
- How to calculate and consider probability from the Bayesian perspective and to calculate conditional probability with Bayes theorem for common scenarios.
- How to use Bayes theorem for classification with Naive Bayes, optimization with Bayesian Optimization, and graphical models with Bayesian Networks.
- How to quantify uncertainty using measures of information and entropy from the field of information theory and calculate quantities such as cross-entropy and mutual information.
- How to develop and evaluate naive classifiers using a probabilistic framework.
- How to evaluate classification models that predict probabilities and calibrate probability predictions.

This new basic understanding of probability will impact your practice of machine learning in the following ways:

- Confidently calculate and wield both frequentist probability (counts) and Bayesian probability (beliefs) generally and within the context of machine learning datasets.
- Confidently select and use loss functions and performance measures when training machine learning algorithms, backed by a knowledge of the underlying probabilistic framework (e.g. maximum likelihood estimation) and the relationships between metrics (e.g. cross-entropy and negative log-likelihood).
- Confidently evaluate classification predictive models, including establishing a robust baseline in performance, probabilistic performance measures, and calibrated predicted probabilities.

This book is not a substitute for an undergraduate course in probability or a textbook for such a course, although it could complement such materials. For a good list of top courses, textbooks, and other resources on probability, see the *Further Reading* section at the end of each tutorial.

## How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific technique or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation,

on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them. This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then applying your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

## About the Book Structure

This book was designed around major probabilistic techniques that are directly relevant to applied machine learning. There are a lot of things you could learn about probability, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. I designed the tutorials to focus on how to get things done with probability. They give you the tools to both rapidly understand and apply each technique or operation. Each tutorial is designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I recommend picking a schedule and sticking to it. The tutorials are divided into seven parts; they are:

- **Part 1: Foundations**. Discover a gentle introduction to the field of probability, the relationship to machine learning, and the importance that probability has when working through predictive modeling problems.
- **Part 2: Basics**. Discover the different types of probability, such as marginal, joint, and conditional, and worked examples that develop an intuition for how to calculate each.
- **Part 3: Distributions**. Discover that probability distributions summarize the likelihood of events and common distribution functions for discrete and continuous random variables.
- **Part 4: Maximum Likelihood**. Discover the maximum likelihood estimation probabilistic framework that underlies how the parameters of many machine learning algorithms are fit on training data.
- **Part 5: Bayesian Probability**. Discover Bayes theorem and some of the most important uses in applied machine learning, such as the naive Bayes algorithm and Bayesian optimization.
- **Part 6: Information Theory**. Discover the relationship between probability and information theory and some of the most important concepts to applied machine learning, such as cross-entropy and information gain.
- **Part 7: Classification**. Discover the relationship between classification and probability, including models that predict probabilities for class labels, evaluation metrics, and probability calibration.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions. The tutorials were not designed to teach you everything there is to know about each of the theories or techniques of probability. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

## About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third-parties needed beyond the installation of the required packages. A complete working example is presented with each tutorial for you to inspect and copy-and-paste. All source code is also provided with the book, and I would recommend running the provided files whenever possible to avoid any copy-paste issues. The provided code was developed in a text editor and is intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead. All code examples were tested on a POSIX-compatible machine with Python 3.

## About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Research papers.
- Books and book chapters.
- Web Pages and Articles.
- API documentation.

Wherever possible, I tried to list and link to the relevant API documentation for key objects and functions used in each lesson so you can learn more about them. I have tried to link to books on Amazon. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

## About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.

- **Help with APIs?** If you need help with using a specific Python function, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.
- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.
- **Help in general?** You can shoot me an email. My details are in *Appendix A*.

# Next

Are you ready? Let's dive in! Next up you will discover that probability is an important foundational field of mathematics.

# Part II

# Background

# Chapter 1

# What is Probability?

Uncertainty involves making decisions with incomplete information, and this is the way we generally operate in the world. Handling uncertainty is typically described using everyday words like chance, luck, and risk. Probability is a field of mathematics that gives us the language and tools to quantify the uncertainty of events and reason in a principled manner. In this tutorial, you will discover a gentle introduction to probability. After reading this tutorial, you will know:

- Certainty is unusual and the world is messy, requiring operating under uncertainty.
- Probability quantifies the likelihood or belief that an event will occur.
- Probability theory is the mathematics of uncertainty.

Let's get started.

## 1.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Uncertainty is Normal
2. Probability of an Event
3. Probability Theory
4. Two Schools of Probability

## 1.2 Uncertainty is Normal

Uncertainty refers to imperfect or incomplete information. Much of mathematics is focused on certainty and logic. Much of programming is this way too, where we develop software with the assumption that it will execute deterministically. Yet, under the covers, computer hardware is subject to noise and errors that are being checked and corrected all of the time. Certainty with perfect and complete information is unusual. It is the place of games and contrived examples.

Almost everything we do or are interested involves information on a continuum between uncertainty or wrongness. The world is messy and imperfect and we must make decisions and

operate in the face of this uncertainty. For example, we often talk about luck, chance, odds, likelihood, and risk. These are words that we use to interpret and negotiate uncertainty in the world. When making inferences and reasoning in an uncertain world, we need principled, formal methods to express and solve problems. Probability provides the language and tools to handle uncertainty.

## 1.3 Probability of an Event

Probability is a measure that quantifies the likelihood that an event will occur. For example, we can quantify the probability of a fire in a neighborhood, a flood in a region, or the purchase of a product. The probability of an event can be calculated directly by counting all of the occurrences of the event, dividing them by the total possible outcomes of the event.

$$\text{probability} = \frac{\text{occurrences}}{\text{non-occurrences} + \text{occurrences}} \tag{1.1}$$

The assigned probability is a fractional value and is always in the range between 0 and 1, where 0 indicates no probability and 1 represents full probability. Together, the probability of all possible events sums to the probability value one. If all possible occurrences are equally likely, the probability of their occurrence is 1 divided by the total possible occurrences or trials. For example, each of the numbers 1 to 6 are equally likely from the roll of a fair die, therefore each has a probability of $\frac{1}{6}$ or 0.166 of occurring.

Probability is often written as a lowercase $p$ and may be stated as a percentage by multiplying the value by 100. For example, a probability of 0.3 can be stated as 30% (given $0.3 \times 100$). A probability of 50% for an event, often spoken of as a *50-50 chance*, means that it is likely to happen half of the time. The probability of an event, like a flood, is often denoted as a function (e.g. the probability function) with an uppercase $P$. For example:

$$P(flood) = \text{probability of a flood} \tag{1.2}$$

It is also sometimes written as a function of lowercase $p$ or $Pr$. For example: $p(flood)$ or $Pr(flood)$. The complement of the probability can be stated as one minus the probability of the event. For example:

$$1 - P(flood) = \text{probability of no flood} \tag{1.3}$$

The probability, or likelihood, of an event is also commonly referred to as the odds of the event or the chance of the event. These all generally refer to the same notion, although odds often has its own notation of wins to losses, written as w:l; e.g. 1:3 for a 1 win and 3 losses or $\frac{1}{4}$ (25%) probability of a win. We have described naive probability, although probability theory allows us to be more general.

## 1.4 Probability Theory

More generally, probability is an extension of logic that can be used to quantify, manage, and harness uncertainty. As a field of study, it is often referred to as probability theory to differentiate it from the likelihood of a specific event.

> Probability can be seen as the extension of logic to deal with uncertainty. [...] Probability theory provides a set of formal rules for determining the likelihood of a proposition being true given the likelihood of other propositions.
>
> — Page 56, *Deep Learning*, 2016.

Probability theory has three important concepts:

- **Event ($A$)**. An outcome to which a probability is assigned.
- **Sample Space ($S$)**. The set of possible outcomes or events.
- **Probability Function ($P$)**. The function used to assign a probability to an event.

The likelihood of an event ($A$) being drawn from the sample space ($S$) is determined by the probability function ($P$). The shape or distribution of all events in the sample space is called the probability distribution. Many domains have a familiar shape to the distribution of probabilities to events, such as uniform if all events are equally likely or Gaussian if the likelihood of the events forms a normal or bell-shape. Probability forms the foundation of many applied fields of mathematics, including statistics, and is an important foundation of many higher-level fields of study, including physics, biology, and computer science.

## 1.5 Two Schools of Probability

There are two main ways of interpreting or thinking about probability. The perhaps simpler approach is to consider probability as the actual likelihood of an event, called the Frequentist probability. Another approach is to consider probability a notion of how strongly it is believed the event will occur, called Bayesian probability. It is not that one approach is correct and the other is incorrect; instead, they are complementary and both interpretations provide different and useful techniques.

### 1.5.1 Frequentist Probability

The frequentist approach to probability is objective. Events are observed and counted, and their frequencies provide the basis for directly calculating a probability, hence the name *frequentist*.

> Probability theory was originally developed to analyze the frequencies of events.
>
> — Page 55, *Deep Learning*, 2016.

Methods from frequentist probability include p-values and confidence intervals used in statistical inference and maximum likelihood estimation for parameter estimation.

### 1.5.2 Bayesian Probability

The Bayesian approach to probability is subjective. Probabilities are assigned to events based on evidence and personal belief and are centered around Bayes' theorem, hence the name *Bayesian*. This allows probabilities to be assigned to very infrequent events and events that have not been observed before, unlike frequentist probability.

> One big advantage of the Bayesian interpretation is that it can be used to model our uncertainty about events that do not have long term frequencies.
>
> — Page 27, *Machine Learning: A Probabilistic Perspective*, 2012.

Methods from Bayesian probability include Bayes factors and credible interval for inference and Bayes estimator and maximum a posteriori estimation for parameter estimation.

## 1.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 1.6.1 Books

- *Probability Theory: The Logic of Science*, 2003.
  https://amzn.to/2lnW2pp
- *Introduction to Probability*, 2nd edition, 2019.
  https://amzn.to/2xPvobK
- *Introduction to Probability*, 2nd edition, 2008.
  https://amzn.to/2llA3PR

### 1.6.2 Articles

- Uncertainty, Wikipedia.
  https://en.wikipedia.org/wiki/Uncertainty
- Probability, Wikipedia.
  https://en.wikipedia.org/wiki/Probability
- Odds, Wikipedia.
  https://en.wikipedia.org/wiki/Odds
- Probability theory, Wikipedia.
  https://en.wikipedia.org/wiki/Probability_theory

## 1.7 Summary

In this tutorial, you discovered a gentle introduction to probability. Specifically, you learned:

- Certainty is unusual and the world is messy, requiring operating under uncertainty.
- Probability quantifies the likelihood or belief that an event will occur.
- Probability theory is the mathematics of uncertainty.

### 1.7.1 Next

In the next tutorial, you will discover the source of uncertainty in machine learning.

# Chapter 2

# Uncertainty in Machine Learning

Applied machine learning requires managing uncertainty. There are many sources of uncertainty in a machine learning project, including variance in the specific data values, the sample of data collected from the domain, and in the imperfect nature of any models developed from such data. Managing the uncertainty that is inherent in machine learning for predictive modeling can be achieved via the tools and techniques from probability, a field specifically designed to handle uncertainty. In this tutorial, you will discover the challenge of uncertainty in machine learning. After reading this tutorial, you will know:

- Uncertainty is the biggest source of difficulty for beginners in machine learning, especially developers.
- Noise in data, incomplete coverage of the domain, and imperfect models provide the three main sources of uncertainty in machine learning.
- Probability provides the foundation and tools for quantifying, handling, and harnessing uncertainty in applied machine learning.

Let's get started.

## 2.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Uncertainty in Machine Learning
2. Noise in Observations
3. Incomplete Coverage of the Domain
4. Imperfect Model of the Problem
5. How to Manage Uncertainty

## 2.2 Uncertainty in Machine Learning

Applied machine learning requires getting comfortable with uncertainty. Uncertainty means working with imperfect or incomplete information. Uncertainty is fundamental to the field of machine learning, yet it is one of the aspects that causes the most difficulty for beginners, especially those coming from a developer background. For software engineers and developers, computers are deterministic. You write a program, and the computer does what you say. Algorithms are analyzed based on space or time complexity and can be chosen to optimize whichever is most important to the project, like execution speed or memory constraints.

Predictive modeling with machine learning involves fitting a model to map examples of inputs to an output, such as a number in the case of a regression problem or a class label in the case of a classification problem. Naturally, the beginner asks reasonable questions, such as:

- What are the best features that I should use?
- What is the best algorithm for my dataset?

The answers to these questions are unknown and might even be unknowable, at least exactly.

> Many branches of computer science deal mostly with entities that are entirely deterministic and certain. [...] Given that many computer scientists and software engineers work in a relatively clean and certain environment, it can be surprising that machine learning makes heavy use of probability theory.
>
> — Page 54, *Deep Learning*, 2016.

This is the major cause of difficulty for beginners. The reason that the answers are unknown is because of uncertainty, and the solution is to systematically evaluate different solutions until a good or good-enough set of features and/or algorithm is discovered for a specific prediction problem. There are three main sources of uncertainty in machine learning, and in the following sections, we will take a look at the three possible sources in turn.

## 2.3 Noise in Observations

Observations from the domain are not crisp; instead, they contain noise. An observation from the domain is often referred to as an *instance* or a *example* and is one row of data. It is what was measured or what was collected. It is the data that describes the object or subject. It is the input to a model and the expected output. An example might be one set of measurements of one iris flower and the species of flower that was measured in the case of training data.

```
Sepal length:  5.1 cm
Sepal width:  3.5 cm
Petal length:  1.4 cm
Petal width:  0.2 cm
Species:   Iris setosa
```

Listing 2.1: Example of measurements and expected class label.

In the case of new data for which a prediction is to be made, it is just the measurements without the species of flower.

```
Sepal length:  5.1 cm
Sepal width:  3.5 cm
Petal length:  1.4 cm
Petal width:  0.2 cm
Species:   ?
```

Listing 2.2: Example of measurements and missing class label.

Noise refers to variability in the observation. Variability could be natural, such as a larger or smaller flower than normal. It could also be an error, such as a slip when measuring or a typo when writing it down. This variability impacts not just the inputs or measurements but also the outputs; for example, an observation could have an incorrect class label. This means that although we have observations for the domain, we must expect some variability or randomness.

The real world, and in turn, real data, is messy or imperfect. As practitioners, we must remain skeptical of the data and develop systems to expect and even harness this uncertainty. This is why so much time is spent on reviewing statistics of data and creating visualizations to help identify those aberrant or unusual cases: so-called data cleaning.

## 2.4 Incomplete Coverage of the Domain

Observations from a domain used to train a model are a sample and incomplete by definition. In statistics, a random sample refers to a collection of observations chosen from the domain without systematic bias (e.g. uniformly random). Nevertheless, there will always be some limitation that will introduce bias. For example, we might choose to measure the size of randomly selected flowers in one garden. The flowers are randomly selected, but the scope is limited to one garden. Scope can be increased to gardens in one city, across a country, across a continent, and so on.

An appropriate level of variance and bias in the sample is required such that the sample is representative of the task or project for which the data or model will be used. We aim to collect or obtain a suitably representative random sample of observations to train and evaluate a machine learning model. Often, we have little control over the sampling process. Instead, we access a database or CSV file and the data we have is the data we must work with. In all cases, we will never have all of the observations. If we did, a predictive model would not be required. This means that there will always be some unobserved cases. There will be part of the problem domain for which we do not have coverage. No matter how well we encourage our models to generalize, we can only hope that we can cover the cases in the training dataset and the salient cases that are not.

This is why we split a dataset into train and test sets or use resampling methods like $k$-fold cross-validation. We do this to handle the uncertainty in the representativeness of our dataset and estimate the performance of a modeling procedure on data not used in that procedure.

## 2.5 Imperfect Model of the Problem

A machine learning model will always have some error. This is often summarized as *all models are wrong*, or more completely in an aphorism by George Box:

> All models are wrong but some are useful

This does not apply just to the model, the artifact, but the whole procedure used to prepare it, including the choice and preparation of data, choice of training hyperparameters, and the interpretation of model predictions. Model error could mean imperfect predictions, such as predicting a quantity in a regression problem that is quite different to what was expected, or predicting a class label that does not match what would be expected. This type of error in prediction is expected given the uncertainty we have about the data that we have just discussed, both in terms of noise in the observations and incomplete coverage of the domain.

Another type of error is an error of omission. We leave out details or abstract them in order to generalize to new cases. This is achieved by selecting models that are simpler but more robust to the specifics of the data, as opposed to complex models that may be highly specialized to the training data. As such, we might and often do choose a model known to make errors on the training dataset with the expectation that the model will generalize better to new cases and have better overall performance.

> In many cases, it is more practical to use a simple but uncertain rule rather than a complex but certain one, even if the true rule is deterministic and our modeling system has the fidelity to accommodate a complex rule.
>
> — Page 55, *Deep Learning*, 2016.

Nevertheless, predictions are required. Given we know that the models will make errors, we handle this uncertainty by seeking a model that is good enough. This often is interpreted as selecting a model that is skillful as compared to a naive method or other established learning models, e.g. good relative performance.

## 2.6 How to Manage Uncertainty

Uncertainty in applied machine learning is managed using probability. Probability is the field of mathematics designed to handle, manipulate, and harness uncertainty.

> A key concept in the field of pattern recognition is that of uncertainty. It arises both through noise on measurements, as well as through the finite size of data sets. Probability theory provides a consistent framework for the quantification and manipulation of uncertainty and forms one of the central foundations for pattern recognition.
>
> — Page 12, *Pattern Recognition and Machine Learning*, 2006.

In fact, probability theory is central to the broader field of artificial intelligence.

> Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience.
>
> — Page 802, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

The methods and tools from probability provide the foundation and way of thinking about the random or stochastic nature of the predictive modeling problems addressed with machine learning; for example:

- **In terms of noisy observations**, probability and statistics help us to understand and quantify the expected value and variability of variables in our observations from the domain.
- **In terms of the incomplete coverage of the domain**, probability helps to understand and quantify the expected distribution and density of observations in the domain.
- **In terms of model error**, probability helps to understand and quantify the expected capability and variance in performance of our predictive models when applied to new data.

But this is just the beginning, as probability provides the foundation for the iterative training of many machine learning models, called maximum likelihood estimation, behind models such as linear regression, logistic regression, artificial neural networks, and much more. Probability also provides the basis for developing specific algorithms, such as Naive Bayes, as well as entire subfields of study in machine learning, such as graphical models like the Bayesian Belief Network.

> Probabilistic methods form the basis of a plethora of techniques for data mining and machine learning.

— Page 336, *Data Mining: Practical Machine Learning Tools and Techniques*. 4th edition, 2016.

The procedures we use in applied machine learning are carefully chosen to address the sources of uncertainty that we have discussed, but understanding why the procedures were chosen requires a basic understanding of probability and probability theory.

## 2.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 2.7.1 Books

- Chapter 3: Probability Theory, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Chapter 2: Probability, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- Chapter 2: Probability Distributions, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I

### 2.7.2 Articles

- Uncertainty, Wikipedia.
  https://en.wikipedia.org/wiki/Uncertainty
- Stochastic, Wikipedia.
  https://en.wikipedia.org/wiki/Stochastic
- All models are wrong, Wikipedia.
  https://en.wikipedia.org/wiki/All_models_are_wrong

## 2.8 Summary

In this tutorial, you discovered the challenge of uncertainty in machine learning. Specifically, you learned:

- Uncertainty is the biggest source of difficulty for beginners in machine learning, especially developers.
- Noise in data, incomplete coverage of the domain, and imperfect models provide the three main sources of uncertainty in machine learning.
- Probability provides the foundation and tools for quantifying, handling, and harnessing uncertainty in applied machine learning.

### 2.8.1 Next

In the next tutorial, you will discover why probability is so important in machine learning.

# Chapter 3

# Why Learn Probability for Machine Learning

Probability is a field of mathematics that quantifies uncertainty. It is undeniably a pillar of the field of machine learning, and many recommend it as a prerequisite subject to study prior to getting started. This is misleading advice, as probability makes more sense to a practitioner once they have the context of the applied machine learning process in which to interpret it. In this tutorial, you will discover why machine learning practitioners should study probability to improve their skills and capabilities. After reading this tutorial, you will know:

- Not everyone should learn probability; it depends where you are in your journey of learning machine learning.
- Many algorithms are designed using the tools and techniques from probability, such as Naive Bayes and Probabilistic Graphical Models.
- The maximum likelihood framework that underlies the training of many machine learning algorithms comes from the field of probability.

Let's get started.

## 3.1 Tutorial Overview

This tutorial is divided into seven parts; they are:

1. Reasons to NOT Learn Probability
2. Class Membership Requires Predicting a Probability
3. Some Algorithms Are Designed Using Probability
4. Models Are Trained Using a Probabilistic Framework
5. Models Can Be Tuned With a Probabilistic Framework
6. Probabilistic Measures Are Used to Evaluate Model Skill
7. One More Reason

## 3.2 Reasons to NOT Learn Probability

Before we go through the reasons that you should learn probability, let's start off by taking a small look at the reasons why you should not. I think you should not study probability if you are just getting started with applied machine learning.

- **It's not required**. Having an appreciation for the abstract theory that underlies some machine learning algorithms is not required in order to use machine learning as a tool to solve problems.
- **It's slow**. Taking months to years to study an entire related field before starting machine learning will delay you achieving your goals of being able to work through predictive modeling problems.
- **It's a huge field**. Not all of probability is relevant to theoretical machine learning, let alone applied machine learning.

I recommend a breadth-first approach to getting started in applied machine learning. I call this the results-first approach. It is where you start by learning and practicing the steps for working through a predictive modeling problem end-to-end (e.g. how to get results) with a tool (such as scikit-learn and Pandas in Python). This process then provides the skeleton and context for progressively deepening your knowledge, such as how algorithms work and, eventually, the math that underlies them. After you know how to work through a predictive modeling problem, let's look at why you should deepen your understanding of probability.

## 3.3 Class Membership Requires Predicting a Probability

Classification predictive modeling problems are those where an example is assigned a given label. An example that you may be familiar with is the iris flowers dataset where we have four measurements of a flower and the goal is to assign one of three different known species of iris flower to the observation. We can model the problem as directly assigning a class label to each observation.

- **Input**: Measurements of a flower.
- **Output**: One iris species.

A more common approach is to frame the problem as a probabilistic class membership, where the probability of an observation belonging to each known class is predicted.

- **Input**: Measurements of a flower.
- **Output**: Probability of membership to each iris species.

Framing the problem as a prediction of the probability of class membership simplifies the modeling problem and makes it easier for a model to learn. It allows the model to capture ambiguity in the data, which allows a process downstream, such as the user, to interpret the probabilities in the context of the domain. The probabilities can be transformed into a crisp class label by choosing the class with the largest probability. The probabilities can also be scaled or transformed using a probability calibration process. This choice of a class membership framing of the problem interpretation of the predictions made by the model requires a basic understanding of probability.

## 3.4 Some Algorithms Are Designed Using Probability

There are algorithms that are specifically designed to harness the tools and methods from probability. These includes individual algorithms, like Naive Bayes algorithm, which is constructed using Bayes Theorem with some simplifying assumptions.

- Naive Bayes

It also extends to whole fields of study, such as probabilistic graphical models, often called graphical models or PGM for short, and designed around Bayes Theorem.

- Probabilistic Graphical Models

A notable graphical model is Bayesian Belief Networks, or Bayes Nets, which are capable of capturing the conditional dependencies between variables.

- Bayesian Belief Networks

## 3.5 Models Are Trained Using a Probabilistic Framework

Many machine learning models are trained using an iterative algorithm designed under a probabilistic framework. Perhaps the most common is the framework of maximum likelihood estimation, sometimes shorted as MLE. This is a framework for estimating model parameters (e.g. weights) given observed data. This is the framework that underlies the ordinary least squares estimate of a linear regression model. The expectation-maximization algorithm, or EM for short, is an approach for maximum likelihood estimation often used for unsupervised data clustering, e.g. estimating $k$ means for $k$ clusters, also known as the $k$-Means clustering algorithm.

For models that predict class membership, maximum likelihood estimation provides the framework for minimizing the difference or divergence between an observed and a predicted probability distribution. This is used in classification algorithms like logistic regression as well as deep learning neural networks. It is common to measure this difference in probability distributions during training using entropy, e.g. via cross-entropy. Entropy, differences between distributions measured via KL divergence, and cross-entropy are from the field of information theory that directly builds upon probability theory. For example, entropy is calculated directly as the negative log of the probability.

## 3.6 Models Can Be Tuned With a Probabilistic Framework

It is common to tune the hyperparameters of a machine learning model, such as $k$ for $k$NN or the learning rate in a neural network. Typical approaches include grid searching ranges of hyperparameters or randomly sampling hyperparameter combinations. Bayesian optimization is a more efficient hyperparameter optimization that involves a directed search of the space of possible configurations based on those configurations that are most likely to result in better performance. As its name suggests, the approach was devised from and harnesses Bayes Theorem when sampling the space of possible configurations.

## 3.7 Probabilistic Measures Are Used to Evaluate Model Skill

For those algorithms where a prediction of probabilities is made, evaluation measures are required to summarize the performance of the model. There are many measures used to summarize the performance of a model based on predicted probabilities. Common examples include aggregate measures like log loss and Brier score. For binary classification tasks where a single probability score is predicted, Receiver Operating Characteristic, or ROC, curves can be constructed to explore different cut-offs that can be used when interpreting the prediction that, in turn, result in different trade-offs. The area under the ROC curve, or ROC AUC, can also be calculated as an aggregate measure. Choice and interpretation of these scoring methods require a foundational understanding of probability theory.

## 3.8 One More Reason

If I could give one more reason, it would be: Because it is fun. Seriously. Learning probability, at least the way I teach it with practical examples and executable code, is a lot of fun. Once you can see how the operations work on real data, you can't help but to develop an intuition for a subject that is often quite abstract.

## 3.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 3.9.1 Books

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *Machine Learning*, 1997.
  https://amzn.to/2jWd51p

### 3.9.2 Articles

- Graphical model, Wikipedia.
  https://en.wikipedia.org/wiki/Graphical_model
- Maximum likelihood estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_likelihood_estimation
- Expectation-maximization algorithm, Wikipedia.
  https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm

- Cross entropy, Wikipedia.
  https://en.wikipedia.org/wiki/Cross_entropy
- Kullback-Leibler divergence, Wikipedia.
  https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Bayesian optimization, Wikipedia.
  https://en.wikipedia.org/wiki/Bayesian_optimization

## 3.10 Summary

In this tutorial, you discovered why, as a machine learning practitioner, you should deepen your understanding of probability. Specifically, you learned:

- Not everyone should learn probability; it depends where you are in your journey of learning machine learning.
- Many algorithms are designed using the tools and techniques from probability, such as Naive Bayes and Probabilistic Graphical Models.
- The maximum likelihood framework that underlies the training of many machine learning algorithms comes from the field of probability.

### 3.10.1 Next

This was the final tutorial in this Part. In the next Part, you will discover the different types of probability.

# Part III

# Foundations

# Chapter 4

# Joint, Marginal, and Conditional Probability

Probability quantifies the uncertainty of the outcomes of a random variable. It is relatively easy to understand and compute the probability for a single variable. Nevertheless, in machine learning, we often have many random variables that interact in complex and unknown ways. There are specific techniques that can be used to quantify the probability of multiple random variables, such as the joint, marginal, and conditional probability. These techniques provide the basis for a probabilistic understanding of fitting a predictive model to data. In this tutorial, you will discover a gentle introduction to joint, marginal, and conditional probability for multiple random variables. After reading this tutorial, you will know:

- Joint probability is the probability of two or more events occurring simultaneously.
- Marginal probability is the probability of an event irrespective of the outcome of other variables.
- Conditional probability is the probability of one event occurring in the presence of one or more other events.

Let's get started.

## 4.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Probability for One Random Variable
2. Probability for Multiple Random Variables
3. Probability for Independence and Exclusivity

## 4.2 Probability for One Random Variable

Probability quantifies the likelihood of an event. Specifically, it quantifies how likely a specific outcome is for a random variable, such as the flip of a coin, the roll of a dice, or drawing a playing card from a deck.

> Probability gives a measure of how likely it is for something to happen.
>
> — Page 57, *Probability: For the Enthusiastic Beginner*, 2016.

For a random variable $x$, $P(x)$ is a function that assigns a probability to all possible values of $x$.

$$\text{Probability Density of } x = P(x) \tag{4.1}$$

The probability of a specific event $A$ for a random variable $x$ is denoted as $P(x = A)$, or simply as $P(A)$.

$$\text{Probability of Event } A = P(A) \tag{4.2}$$

Probability is calculated as the number of desired outcomes divided by the total possible outcomes, in the case where all outcomes are equally likely.

$$\text{Probability} = \frac{\text{number of desired outcomes}}{\text{total number of possible outcomes}} \tag{4.3}$$

This is intuitive if we think about a discrete random variable such as the roll of a die. For example, the probability of a die rolling a 5 is calculated as one outcome of rolling a 5 (1) divided by the total number of discrete outcomes (6) or $\frac{1}{6}$ or about 0.1666 or about 16.666%. The sum of the probabilities of all outcomes must equal one. If not, we do not have valid probabilities.

$$\text{Sum of the Probabilities for All Outcomes} = 1.0. \tag{4.4}$$

The probability of an impossible outcome is zero. For example, it is impossible to roll a 7 with a standard six-sided die.

$$\text{Probability of Impossible Outcome} = 0.0 \tag{4.5}$$

The probability of a certain outcome is one. For example, it is certain that a value between 1 and 6 will occur when rolling a six-sided die.

$$\text{Probability of Certain Outcome} = 1.0 \tag{4.6}$$

The probability of an event not occurring, called the complement. This can be calculated by one minus the probability of the event, or $1 - P(A)$. For example, the probability of not rolling a 5 would be $1 - P(5)$ or $1 - 0.166$ or about 0.833 or about 83.333%.

$$P(\text{not } A) = 1 - P(A) \tag{4.7}$$

Now that we are familiar with the probability of one random variable, let's consider probability for multiple random variables.

## 4.3 Probability for Multiple Random Variables

In machine learning, we are likely to work with many random variables. For example, given a table of data, such as in excel, each row represents a separate observation or event, and each column represents a separate random variable. Variables may be either discrete, meaning that they take on a finite set of values, or continuous, meaning they take on a real or numerical value. As such, we are interested in the probability across two or more random variables.

This is complicated as there are many ways that random variables can interact, which, in turn, impacts their probabilities. This can be simplified by reducing the discussion to just two random variables $(X, Y)$, although the principles generalize to multiple variables. And further, to discuss the probability of just two events, one for each variable $(X = A, Y = B)$, although we could just as easily be discussing groups of events for each variable.

Therefore, we will introduce the probability of multiple random variables as the probability of event $A$ and event $B$, which in shorthand is $X = A$ and $Y = B$. We assume that the two variables are related or dependent in some way. As such, there are three main types of probability we might want to consider; they are:

- **Joint Probability**: Probability of events $A$ and $B$.
- **Marginal Probability**: Probability of event $A$ given variable $Y$.
- **Conditional Probability**: Probability of event $A$ given event $B$.

These types of probability form the basis of much of predictive modeling with problems such as classification and regression. For example:

- The probability of a row of data is the joint probability across each input variable.
- The probability of a specific value of one input variable is the marginal probability across the values of the other input variables.
- The predictive model itself is an estimate of the conditional probability of an output given an input example.

Joint, marginal, and conditional probability are foundational in machine learning. Let's take a closer look at each in turn.

### 4.3.1 Joint Probability for Two Variables

We may be interested in the probability of two simultaneous events, e.g. the outcomes of two different random variables. The probability of two (or more) events is called the joint probability. The joint probability of two or more random variables is referred to as the joint probability distribution. For example, the joint probability of event $A$ and event $B$ is written formally as:

$$P(A \text{ and } B) \tag{4.8}$$

The *and* or conjunction is denoted using the upside down capital $U$ operator ($\cap$) or sometimes a comma (,).

$$P(A \text{ and } B) = P(A \cap B) = P(A, B) \tag{4.9}$$

The joint probability for events $A$ and $B$ is calculated as the probability of event $A$ given event $B$ multiplied by the probability of event $B$. This can be stated formally as follows:

$$P(A \cap B) = P(A \text{ given } B) \times P(B) \tag{4.10}$$

The calculation of the joint probability is sometimes called the fundamental rule of probability or the *product rule* of probability. Here, $P(A \text{ given } B)$ is the probability of event $A$ given that event $B$ has occurred, called the conditional probability, described below. The joint probability is symmetrical, meaning that $P(A \cap B)$ is the same as $P(B \cap A)$.

### 4.3.2 Marginal Probability

We may be interested in the probability of an event for one random variable, irrespective of the outcome of another random variable. For example, the probability of $X = A$ for all outcomes of $Y$. The probability of one event in the presence of all (or a subset of) outcomes of the other random variable is called the marginal probability or the marginal distribution. The marginal probability of one random variable in the presence of additional random variables is referred to as the marginal probability distribution.

It is called the marginal probability because if all outcomes and probabilities for the two variables were laid out together in a table ($X$ as columns, $Y$ as rows), then the marginal probability of one variable ($X$) would be the sum of probabilities for the other variable ($Y$ rows) on the margin of the table. There is no special notation for the marginal probability; it is just the sum or union over all the probabilities of all events for the second variable for a given fixed event for the first variable.

$$P(X = A) = \sum^{y \in Y} P(X = A, Y = y) \tag{4.11}$$

This is another important foundational rule in probability, referred to as the *sum rule*. The marginal probability is different from the conditional probability (described next) because it considers the union of all events for the second variable rather than the probability of a single event.

### 4.3.3 Conditional Probability

We may be interested in the probability of an event given the occurrence of another event. The probability of one event given the occurrence of another event is called the conditional probability. The conditional probability of one to one or more random variables is referred to as the conditional probability distribution. For example, the conditional probability of event $A$ given event $B$ is written formally as:

$$P(A \text{ given } B) \tag{4.12}$$

The *given* is denoted using the pipe (|) operator; for example:

$$P(A \text{ given } B) = P(A|B) \tag{4.13}$$

The conditional probability for events $A$ given event $B$ is calculated as follows:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{4.14}$$

This calculation assumes that the probability of event $B$ is not zero, e.g. is not impossible. The notion of event $A$ given event $B$ does not mean that event $B$ has occurred (e.g. is certain); instead, it is the probability of event $A$ occurring after or in the presence of event $B$ for a given trial.

## 4.4 Probability for Independence and Exclusivity

When considering multiple random variables, it is possible that they do not interact. We may know or assume that two variables are not dependent upon each other instead are independent. Alternately, the variables may interact but their events may not occur simultaneously, referred to as exclusivity. We will take a closer look at the probability of multiple random variables under these circumstances in this section.

### 4.4.1 Independence

If one variable is not dependent on a second variable, this is called independence or statistical independence. This has an impact on calculating the probabilities of the two variables. For example, we may be interested in the joint probability of independent events $A$ and $B$, which is the same as the probability of $A$ and the probability of $B$. Probabilities are combined using multiplication, therefore the joint probability of independent events is calculated as the probability of event A multiplied by the probability of event $B$. This can be stated formally as follows:

$$\text{Joint Probability} : P(A \cap B) = P(A) \times P(B) \tag{4.15}$$

As we might intuit, the marginal probability for an event for an independent random variable is simply the probability of the event. It is the idea of probability of a single random variable that are familiar with:

$$\text{Marginal Probability} : P(A) \tag{4.16}$$

We refer to the marginal probability of an independent probability as simply the probability. Similarly, the conditional probability of $A$ given $B$ when the variables are independent is simply the probability of $A$ as the probability of $B$ has no effect. For example:

$$\text{Conditional Probability} : P(A|B) = P(A) \tag{4.17}$$

We may be familiar with the notion of statistical independence from sampling. This assumes that one sample is unaffected by prior samples and does not affect future samples. Many machine learning algorithms assume that samples from a domain are independent to each other and come from the same probability distribution, referred to as independent and identically distributed, or i.i.d. for short.

### 4.4.2 Exclusivity

If the occurrence of one event excludes the occurrence of other events, then the events are said to be mutually exclusive. The probability of the events are said to be disjoint, meaning that they cannot interact, are strictly independent. If the probability of event $A$ is mutually exclusive with event $B$, then the joint probability of event $A$ and event $B$ is zero.

$$P(A \cap B) = 0.0 \tag{4.18}$$

Instead, the probability of an outcome can be described as event $A$ or event $B$, stated formally as follows:

$$P(A \text{ or } B) = P(A) + P(B) \tag{4.19}$$

The *or* is also called a union and is denoted as a capital $U$ letter ($\cup$); for example:

$$P(A \text{ or } B) = P(A \cup B) \tag{4.20}$$

If the events are not mutually exclusive, we may be interested in the outcome of either event. The probability of non-mutually exclusive events is calculated as the probability of event $A$ and the probability of event $B$ minus the probability of both events occurring simultaneously. This can be stated formally as follows:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{4.21}$$

## 4.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 4.5.1 Books

- *Probability: For the Enthusiastic Beginner*, 2016.
  https://amzn.to/2jULJsu
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 4.5.2 Articles

- Probability, Wikipedia.
  https://en.wikipedia.org/wiki/Probability
- Notation in probability and statistics, Wikipedia.
  https://en.wikipedia.org/wiki/Notation_in_probability_and_statistics
- Independence (probability theory), Wikipedia.
  https://en.wikipedia.org/wiki/Independence_(probability_theory)

- Independent and identically distributed random variables, Wikipedia.
https://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables

- Mutual exclusivity, Wikipedia.
https://en.wikipedia.org/wiki/Mutual_exclusivity

- Marginal distribution, Wikipedia.
https://en.wikipedia.org/wiki/Marginal_distribution

- Joint probability distribution, Wikipedia.
https://en.wikipedia.org/wiki/Joint_probability_distribution

- Conditional probability, Wikipedia.
https://en.wikipedia.org/wiki/Conditional_probability

## 4.6 Summary

In this tutorial, you discovered a gentle introduction to joint, marginal, and conditional probability for multiple random variables. Specifically, you learned:

- Joint probability is the probability of two or more events occurring simultaneously.
- Marginal probability is the probability of an event irrespective of the outcome of other variables.
- Conditional probability is the probability of one event occurring in the presence of one or more other events.

### 4.6.1 Next

In the next tutorial, you will discover how to develop an intuition for the different types of probability with worked examples.

# Chapter 5

# Intuition for Joint, Marginal, and Conditional Probability

Probability for a single random variable is straightforward, although it can become complicated when considering two or more variables. With just two variables, we may be interested in the probability of two simultaneous events (joint probability), the probability of one event given the occurrence of another event (conditional probability), or just the probability of an event regardless of other variables (marginal probability). These types of probability are easy to define but the intuition behind their meaning can take some time to sink in, requiring some worked examples that can be tinkered with. In this tutorial, you will discover the intuitions behind calculating joint, marginal, and conditional probability. After completing this tutorial, you will know:

- How to calculate joint, marginal, and conditional probability for independent random variables.
- How to collect observations from joint random variables and construct a joint probability table.
- How to calculate joint, marginal, and conditional probability from a joint probability table.

Let's get started.

## 5.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Joint, Marginal, and Conditional Probabilities
2. Probabilities of Rolling Two Dice
3. Probabilities of Weather in Two Cities

## 5.2 Joint, Marginal, and Conditional Probabilities

Calculating probability is relatively straightforward when working with a single random variable. It gets more interesting when considering two or more random variables, as we often do in many real world circumstances. There are three main types of probabilities that we may be interested in calculating when working with two (or more) random variables. Briefly, they are:

- **Joint Probability**. The probability of simultaneous events.
- **Marginal Probability**. The probability of an event irrespective of the other variables.
- **Conditional Probability**. The probability of events given the presence of other events.

The meaning and calculation of these different types of probabilities vary depending on whether the two random variables are independent (simpler) or dependent (more complicated). We will explore how to calculate and interpret these three types of probability with worked examples. In the next section, we will look at the independent rolls of two dice, and in the following section, we will look at the occurrence of weather events of two geographically close cities.

## 5.3 Probabilities of Rolling Two Dice

A good starting point for exploring joint and marginal probabilities is to consider independent random variables as the calculations are very simple. The roll of a fair die gives a one in six ($\frac{1}{6}$) or 0.166 (16.666%) probability of a particular number between 1 and 6 coming up.

$$\begin{aligned} P(\text{dice1=1}) &= \frac{1}{6} \\ P(\text{dice1=2}) &= \frac{1}{6} \\ P(\text{dice1=3}) &= \frac{1}{6} \\ P(\text{dice1=4}) &= \frac{1}{6} \\ P(\text{dice1=5}) &= \frac{1}{6} \\ P(\text{dice1=6}) &= \frac{1}{6} \end{aligned} \tag{5.1}$$

If we roll a second die, we get the same probability of each value on that die. Each event for a die has an equal probability and the rolls of dice1 and dice2 do not affect each other.

$$\begin{aligned} P(\text{dice1} \in \{1,2,3,4,5,6\}) &= 1.0 \\ P(\text{dice2} \in \{1,2,3,4,5,6\}) &= 1.0 \end{aligned} \tag{5.2}$$

First, using exclusivity, we can calculate the probability of rolling an even number for dice1 as the sum of the probabilities of rolling a 2, 4, or 6, for example:

$$\begin{aligned} P(\text{dice1} \in \{2,4,6\}) &= P(\text{dice1} = 2) + P(\text{dice1} = 4) + P(\text{dice1} = 6) \\ &= \frac{1}{6} + \frac{1}{6} + \frac{1}{6} \end{aligned} \tag{5.3}$$

This is 0.5 or 50% as we might intuitively expect. Now, we might consider the joint probability of rolling an even number with both dice simultaneously. The joint probability for independent random variables is calculated as follows:

$$P(A \cap B) = P(A) \times P(B) \tag{5.4}$$

This is calculated as the probability of rolling an even number for dice1 multiplied by the probability of rolling an even number for dice2. The probability of the first event constrains the probability of the second event.

$$P(\text{dice1} \in \{2, 4, 6\} \cap \text{dice2} \in \{2, 4, 6\}) = P(\text{dice1} \in \{2, 4, 6\}) \times P(\text{dice2} \in \{2, 4, 6\}) \tag{5.5}$$

We know that the probability of rolling an even number of each die is 0.5, therefore the probability of rolling an even number is $\frac{3}{6}$ or 0.5. Plugging that in, we get: $0.5 \times 0.5$ (0.25) or 25%. Another way to look at this is to consider that rolling one die gives 6 combinations. Rolling two dice together gives 6 combinations for dice2 for each of the 6 combinations of dice1 or $(6 \times 6)$ 36 combinations. A total of 3 of the 6 combinations of dice1 will be even, and 3 of the 6 combinations of those will be even. That gives $(3 \times 3)$ 9 out of the 36 combinations as an even number of each die or $(\frac{9}{36} = 0.25)$ 25%.

If you are ever in doubt of your probability calculations when working with independent variables with discrete events, think in terms of combinations and things will make sense again. We can construct a table of the joint probabilities based on our knowledge of the domain. The complete table is listed below with dice1 across the top (x-axis) and dice2 along the side (y-axis). The joint probabilities of each event for a given cell are calculated using the joint probability formula, e.g. $0.166 \times 0.166$ or 0.027 or about 2.777%.

```
   1     2     3     4     5     6
1  0.027 0.027 0.027 0.027 0.027 0.027
2  0.027 0.027 0.027 0.027 0.027 0.027
3  0.027 0.027 0.027 0.027 0.027 0.027
4  0.027 0.027 0.027 0.027 0.027 0.027
5  0.027 0.027 0.027 0.027 0.027 0.027
6  0.027 0.027 0.027 0.027 0.027 0.027
```

Listing 5.1: Example of the joint probability table for rolling two die.

This table captures the joint probability distribution of the events of the two random variables, dice1 and dice2. It is pretty boring, but we can use it to sharpen our understanding of joint and marginal probability of independent variables. For example, the joint probability of rolling a 2 with dice1 and a 2 with dice2 can be read from the table directly as 2.777%. We can explore more elaborate cases, such as rolling a 2 with dice1 and rolling an odd number with dice2. This can be read off as summing the values in the second column for rolling a 2 with dice1 and the first, third, and fifth rows for rolling an odd number with dice2.

$$P(\text{dice1} = 2 \cap \text{dice2} \in \{1, 3, 5\}) = 0.027 + 0.027 + 0.027 \tag{5.6}$$

This comes out to be about 0.083, or about 8.333%. We can also use this table to calculate the marginal probability of dice1 rolling a 2 in the presence of dice2. This is calculated as the sum of an entire column of probabilities for dice1 or a row of probabilities for dice2. For example, we can calculate the marginal probability of rolling a 6 with dice2 as the sum of probabilities

across the final row of the table. This comes out to be about 0.166 or 16.666% as we may intuitively expect.

Importantly, if we sum the probabilities for all cells in the table, it must equal 1.0. Additionally, if we sum the probabilities for each row, then the sum of these sums must equal 1.0. The same if we sum the probabilities in each column, then the sum of these sums too must equal 1.0. This is a requirement for a table of joint probabilities. Because the events are independent, there is nothing special needed to calculate conditional probabilities.

$$P(A|B) = P(A) \tag{5.7}$$

For example, the probability of rolling a 2 with dice1 is the same regardless of what was rolled with dice2.

$$P(\text{dice1} = 2|\text{dice2} = 6) = P(\text{dice1} = 2) \tag{5.8}$$

In this way, conditional probability does not have a useful meaning for independent random variables. Developing a table of joint probabilities is a helpful tool for better understanding how to calculate and explore the joint and marginal probabilities. In the next section, let's look at a more complicated example with dependent random variables.

## 5.4 Probabilities of Weather in Two Cities

We can develop an intuition for joint and marginal probabilities using a table of joint probabilities of events for two dependent random variables. Consider the situation where there are two cities, city1 and city2. The cities are close enough that they are generally affected by the same weather, yet they are far enough apart that they do not get identical weather. We can consider discrete weather classifications for these cities on a given day, such as sunny, cloudy, and rainy. When it is sunny in city1, it is usually sunny in city2, but not always. As such, there is a dependency between the weather in the two cities. Now, let's explore the different types of probability.

### 5.4.1 Data Collection

First, we can record the observed weather in each city over twenty days. For example, on day 1, what was the weather in each, day 2, and so on.

```
Day | City1 | City2
1     Sunny   Sunny
2     Sunny   Cloudy
3
...
```

Listing 5.2: Example of collected data for the weather in two cities.

The complete table of results is omitted for brevity, and we will make up totals later. We can then calculate the sum of the total number of paired events that were observed. For example, the total number of times it was sunny in city1 and sunny in city2, the total number of times it was sunny in city1 and cloudy in city2, and so on.

```
City 1 | City 2 | Total
sunny    sunny    6/20
```

```
sunny   cloudy  1/20
sunny   rainy   0/20
...
```

Listing 5.3: Example of the frequency of weather in two cities.

Again, the complete table is omitted for brevity, and we will make up totals later. This data provides the basis for exploring the probability of weather events in the two cities.

### 5.4.2 Joint Probabilities

First, we might be interested in the probability of weather events in each city. We can create a table that contains the probabilities of the paired or joint weather events. The table below summarizes the probability of each discrete weather for the two cities, with city1 defined across the top (columns) and city2 defined along the side (rows).

```
         Sunny | Cloudy | Rainy
Sunny    6/20   2/20     0/20
Cloudy   1/20   5/20     2/20
Rainy    0/20   1/20     3/20
```

Listing 5.4: Example of the joint probabilities for weather in two cities.

A cell in the table describes the joint probability of an event in each city, and together, the probabilities in the table summarize the joint probability distribution of weather events for the two cities. The sum of the joint probabilities for all cells in the table must equal 1.0. We can calculate the joint probability for the weather in two cities. For example, we would expect the joint probability of it being sunny in both cities at the same time as being high. This can be stated formally as:

$$P(\text{city1=sunny} \cap \text{city2=sunny}) \tag{5.9}$$

Or more compactly:

$$P(\text{sunny} \cap \text{sunny}) \tag{5.10}$$

We can read this off the table directly as $\frac{6}{20}$ or 0.3 or 30%. A relatively high probability. We can take this a step further and consider the probability of it not being rainy in the first city but having rain in the second city. We could state this as:

$$P(\text{city1} \in \{\text{sunny} \cup \text{cloudy}\} \cap \text{city2=rainy}) \tag{5.11}$$

Again, we can calculate this directly from the table. Firstly, $P(\text{sunny}, \text{rainy})$ is $\frac{0}{20}$ and $P(\text{cloudy}, \text{rainy})$ is $\frac{1}{20}$. We can then add these probabilities together to give $\frac{1}{20}$ or 0.05 or 5%. It can happen, but it is not likely. The table also gives an idea of the marginal distribution of events. For example, we might be interested in the probability of a sunny day in city1, regardless of what happens in city2. This can be read from the table by summing the probabilities for city1 for sunny, e.g the first column of probabilities:

$$\begin{aligned} P(\text{city1=sunny}) = & P(\text{city1=sunny} \cap \text{city2=sunny}) + \\ & P(\text{city1=sunny} \cap \text{city2=cloudy}) + \\ & P(\text{city1=sunny} \cap \text{city2=rainy}) \end{aligned} \tag{5.12}$$

Or:

$$\begin{aligned} P(\text{city1=sunny}) &= \frac{6}{20} + \frac{1}{20} + \frac{0}{20} \\ &= \frac{7}{20} \end{aligned} \tag{5.13}$$

Therefore, the marginal probability of a sunny day in city1 is 0.35 or 35%. We can do the same thing for city2 by calculating the marginal probability of an event across some or all probabilities in a row. For example, the probability of a rainy day in city2 would be calculated as the sum of probabilities along the bottom row of the table:

$$\begin{aligned} P(\text{city2=rainy}) &= \frac{0}{20} + \frac{1}{20} + \frac{3}{20} \\ &= \frac{4}{20} \end{aligned} \tag{5.14}$$

Therefore, the marginal probability of a rainy day in city2 is 0.2 or 20%. The marginal probabilities are often interesting and useful, and it is a good idea to update the table of joint probabilities to include them; for example:

```
          Sunny | Cloudy | Rainy | Marginal
Sunny     6/20    2/20     0/20    8/20
Cloudy    1/20    5/20     2/20    8/20
Rainy     0/20    1/20     3/20    4/20
Marginal  7/20    8/20     5/20    20/20
```

Listing 5.5: Example of the joint and marginal probabilities for weather in two cities.

### 5.4.3 Conditional Probabilities

We might be interested in the probability of a weather event given the occurrence of a weather event in another city. This is called the conditional probability and can be calculated using the joint and marginal probabilities.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{5.15}$$

For example, we might be interested in the probability of it being sunny in city1, given that it is sunny in city2. This can be stated formally as:

$$P(\text{city1=sunny}|\text{city2=sunny}) = \frac{P(\text{city1=sunny} \cap \text{city2=sunny})}{P(\text{city2=sunny})} \tag{5.16}$$

We can fill in the joint and marginal probabilities from the table in the previous section; for example:

$$\begin{aligned} P(\text{city1=sunny}|\text{city2=sunny}) &= \frac{\frac{6}{20}}{\frac{8}{20}} \\ &= \frac{0.3}{0.4} \end{aligned} \tag{5.17}$$

This comes out to be 0.75 or 75%, which is intuitive. We would expect that if it is sunny in city2 that city1 should also be sunny most of the time. This is different from the joint probability of it being sunny in both cities on a given day which has a lower probability of 30%. It makes more sense if we consider it from the perspective of the number of combinations. We have more information in this conditional case, therefore we don't have to calculate the probability across all 20 days. Specifically, we are assuming we know that it is sunny in city2, which dramatically reduces the number of days from 20 to 8. A total of 6 of the days that were sunny in city2 were also sunny in city1, giving the fraction $\frac{6}{8}$ or 0.75 or 75%. All of this can be read from the table of joint probabilities. An important aspect of conditional probability that is often misunderstood is that it is not reversible.

$$P(A|B) \neq P(B|A) \tag{5.18}$$

That is the probability of it being sunny in city1 given that it is sunny in city2 is not the same as the probability of it being sunny in city2 given that it is sunny in city1.

$$P(\text{city1=sunny}|\text{city2=sunny}) \neq P(\text{city2=sunny}|\text{city1=sunny}) \tag{5.19}$$

In this case, the probability of it being sunny in city2 given that it is sunny in city1 is calculated as follows:

$$\begin{aligned} P(\text{city2=sunny}|\text{city1=sunny}) &= \frac{P(\text{city2=sunny} \cap \text{city1=sunny})}{P(\text{city1=sunny})} \\ &= \frac{\frac{6}{20}}{\frac{7}{20}} \\ &= \frac{0.3}{0.35} \\ &= 0.857 \end{aligned} \tag{5.20}$$

In this case, it is higher, at about 85.714%. We can also use the conditional probability to calculate the joint probability.

$$P(A \cap B) = P(A|B) \times P(B) \tag{5.21}$$

For example, if all we know is the conditional probability of sunny in city2 given that it is sunny in city1 and the marginal probability of city2, we can calculate the joint probability as:

$$\begin{aligned} P(\text{city1=sunny} \cap \text{city2=sunny}) &= P(\text{city2=sunny}|\text{city1=sunny}) \times P(\text{city1=sunny}) \\ &= 0.857 \times 0.35 \\ &= 0.3 \end{aligned} \tag{5.22}$$

This gives 0.3 or 30% as we expected.

## 5.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 5.5.1 Books

- *Probability: For the Enthusiastic Beginner*, 2016.
  https://amzn.to/2jULJsu
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 5.5.2 Articles

- Probability, Wikipedia.
  https://en.wikipedia.org/wiki/Probability
- Notation in probability and statistics, Wikipedia.
  https://en.wikipedia.org/wiki/Notation_in_probability_and_statistics
- Independence (probability theory), Wikipedia.
  https://en.wikipedia.org/wiki/Independence_(probability_theory)
- Independent and identically distributed random variables, Wikipedia.
  https://en.wikipedia.org/wiki/Independent_and_identically_distributed_random_variables
- Mutual exclusivity, Wikipedia.
  https://en.wikipedia.org/wiki/Mutual_exclusivity
- Marginal distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Marginal_distribution
- Joint probability distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Joint_probability_distribution
- Conditional probability, Wikipedia.
  https://en.wikipedia.org/wiki/Conditional_probability

## 5.6 Summary

In this tutorial, you discovered the intuitions behind calculating joint, marginal, and conditional probability. Specifically, you learned:

- How to calculate joint, marginal, and conditional probability for independent random variables.
- How to collect observations from joint random variables and construct a joint probability table.
- How to calculate joint, marginal, and conditional probability from a joint probability table.

### 5.6.1 Next

In the next tutorial, you will discover how to further develop an intuition for the different types of probability with advanced worked examples.

# Chapter 6

# Advanced Examples of Calculating Probability

Probability calculations are frustratingly unintuitive. Our brains are too eager to take shortcuts and get the wrong answer, instead of thinking through a problem and calculating the probability correctly. To make this issue obvious and aid in developing intuition, it can be useful to work through classical problems from applied probability. These problems, such as the birthday problem, boy or girl problem, and the Monty Hall problem trick us with the incorrect intuitive answer and require a careful application of the rules of marginal, conditional, and joint probability in order to arrive at the correct solution. In this tutorial, you will discover how to develop an intuition for probability by working through classical thought-provoking problems. After reading this tutorial, you will know:

- How to solve the birthday problem by multiplying probabilities together.
- How to solve the boy or girl problem using conditional probability.
- How to solve the Monty Hall problem using joint probability.

Let's get started.

## 6.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Birthday Problem
2. Boy or Girl Problem
3. Monty Hall Problem

## 6.2 Birthday Problem

A classic example of applied probability involves calculating the probability of two people having the same birthday. It is a classic example because the result does not match our intuition. As such, it is sometimes called the birthday paradox. The problem can be generally stated as:

- **Problem**: How many people are required so that any two people in the group have the same birthday with at least a 50-50 chance?

There are no tricks to this problem; it involves simply calculating the marginal probability. It is assumed that the probability of a randomly selected person having a birthday on any given day of the year (excluding leap years) is uniformly distributed across the days of the year, e.g. $\frac{1}{365}$ or about 0.273%. Our intuition might leap to an answer and assume that we might need at least as many people as there are days in the year, e.g. 365. Our intuition likely fails because we are thinking about ourselves and other people matching our own birthday. That is, we are thinking about how many people are needed for another person born on the same day as you. That is a different question.

Instead, to calculate the solution, we can think about comparing pairs of people within a group and the probability of a given pair being born on the same day. This unlocks the calculation required. The number of pairwise comparisons within a group (excluding comparing each person with themselves) is calculated as follows:

$$comparisons = n \times \frac{(n-1)}{2} \tag{6.1}$$

For example, if we have a group of five people, we would be doing 10 pairwise comparisons among the group to check if they have the same birthday, which is more opportunity for a hit than we might expect. Importantly, the number of comparisons within the group increases exponentially with the size of the group. One more step is required. It is easier to calculate the inverse of the problem. That is, the probability that two people in a group do not have the same birthday. We can then invert the final result to give the desired probability, for example:

$$P(\text{2 in n same birthday}) = 1 - P(\text{2 in n not same birthday}) \tag{6.2}$$

We can see why calculating the probability of non-matching birthdays is easy with an example with a small group, in this case, three people. People can be added to the group one-by-one. Each time a person is added to the group, it decreases the number of available days where there is no matching birthday in the year, decreasing the number of available days by one. For example 365 days, 364 days, etc. Additionally, the probability of a non-match for a given additional person added to the group must be combined with the prior calculated probabilities before it. For example $P(n = 2) \times P(n = 3)$, etc. This gives the following, calculating the probability of no matching birthdays with a group size of three:

$$\begin{aligned} P(n = 3) &= \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \\ &= 99.18 \end{aligned} \tag{6.3}$$

Inverting this gives about 0.820% of a matching birthday among a group of three people. Stepping through this, the first person has a birthday, which reduces the number of candidate days for the rest of the group from 365 to 364 unused days (i.e. days without a birthday). For the second person, we calculate the probability of a conflicting birthday as 364 safe days from 365 days in the year or about a ($\frac{364}{365}$) 99.72% probability of not having the same birthday. We now subtract the second person's birthday from the number of available days to give 363. The probability of the third person of not having a matching birthday is then given as $\frac{363}{365}$ multiplied by the prior probability to give about 99.18%. This calculation can get tedious for large groups,

therefore we might want to automate it. The example below calculates the probabilities for group sizes from two to 30.

```
# example of the birthday problem
# define maximum group size
n = 30
# number of days in the year
days = 365
# calculate probability for different group sizes
p = 1.0
for i in range(1, n):
  av = days - i
  p *= av / days
  print('n=%d, %d/%d, p=%.3f 1-p=%.3f' % (i+1, av, days, p*100, (1-p)*100))
```

Listing 6.1: Example of calculating the probability of two birthdays for different group sizes.

Running the example first prints the group size, then the available days divided by the total days in the year, then the probability of no matching birthdays in the group followed by the complement or the probability of two people having a birthday in the group.

```
n=2, 364/365, p=99.726 1-p=0.274
n=3, 363/365, p=99.180 1-p=0.820
n=4, 362/365, p=98.364 1-p=1.636
n=5, 361/365, p=97.286 1-p=2.714
n=6, 360/365, p=95.954 1-p=4.046
n=7, 359/365, p=94.376 1-p=5.624
n=8, 358/365, p=92.566 1-p=7.434
n=9, 357/365, p=90.538 1-p=9.462
n=10, 356/365, p=88.305 1-p=11.695
n=11, 355/365, p=85.886 1-p=14.114
n=12, 354/365, p=83.298 1-p=16.702
n=13, 353/365, p=80.559 1-p=19.441
n=14, 352/365, p=77.690 1-p=22.310
n=15, 351/365, p=74.710 1-p=25.290
n=16, 350/365, p=71.640 1-p=28.360
n=17, 349/365, p=68.499 1-p=31.501
n=18, 348/365, p=65.309 1-p=34.691
n=19, 347/365, p=62.088 1-p=37.912
n=20, 346/365, p=58.856 1-p=41.144
n=21, 345/365, p=55.631 1-p=44.369
n=22, 344/365, p=52.430 1-p=47.570
n=23, 343/365, p=49.270 1-p=50.730
n=24, 342/365, p=46.166 1-p=53.834
n=25, 341/365, p=43.130 1-p=56.870
n=26, 340/365, p=40.176 1-p=59.824
n=27, 339/365, p=37.314 1-p=62.686
n=28, 338/365, p=34.554 1-p=65.446
n=29, 337/365, p=31.903 1-p=68.097
n=30, 336/365, p=29.368 1-p=70.632
```

Listing 6.2: Example output from calculating the probability of two birthdays for different group sizes.

The result is surprising, showing that only 23 people are required to give more than a 50% chance of two people having a birthday on the same day. More surprising is that with 30 people,

this increases to a 70% probability. It's surprising because 20 to 30 people is about the average class size in school, a number of people for which we all have an intuition (if we attended school). If the group size is increased to around 60 people, then the probability of two people in the group having the same birthday is above 99%!

## 6.3 Boy or Girl Problem

Another classic example of applied probability is the case of calculating the probability of whether a child is a boy or girl. The probability of whether a given child is a boy or a girl with no additional information is 50%. This may or may not be true in reality, but let's assume it for the case of this useful illustration of probability. As soon as more information is included, the probability calculation changes, and this trips up even people versed in math and probability. A popular example is called the *two-child problem* that involves being given information about a family with two children and estimating the sex of one child. If the problem is not stated precisely, it can lead to misunderstanding, and in turn, two different ways of calculating the probability. This is the challenge of using natural language instead of notation, and in this case is referred to as the *boy or girl paradox*. Let's look at two precisely stated examples.

- **Case 1**: A woman has two children and the oldest is a boy. What is the probability of this woman having two sons?

Consider a table of the unconditional probabilities.

```
Younger Child | Older Child | Unconditional Probability
Girl          Boy           1/4
Boy           Boy           1/4
Girl          Girl          1/4
Boy           Girl          1/4
```

Listing 6.3: Example of combinations for boys and girls with no additional information.

Our intuition suggests that the probability that the other child is a boy is 0.5 or 50%. Alternately, our intuition might suggest the probability of a family with two boys is $\frac{1}{4}$ (e.g. a probability of 0.25) for the four possible combinations of boys and girls for a two-child family. We can explore this by enumerating all possible combinations that include the information given:

```
Younger Child | Older Child | Conditional Probability
Girl          Boy           1/2
Boy           Boy           1/2 (*)
Girl          Girl          0 (impossible)
Boy           Girl          0 (impossible)
```

Listing 6.4: Example of combinations for boys and girls in case 1.

The intuition is that the additional information makes some cases impossible. There would be four outcomes, but the information given reduces the domain to 2 possible outcomes (older child is a boy). Indeed, only one of the two outcomes can be boy-boy, therefore the probability is $\frac{1}{2}$ or (0.5) 50%. Let's look at a second very similar case.

- **Case 2**: A woman has two children and one of them is a boy. What is the probability of this woman having two sons?

Our intuition leaps to the same conclusion. At least mine did. **And this would be incorrect**. For example, $\frac{1}{2}$ for a boy as the second child being a boy. Another leap might be $\frac{1}{4}$ for the case of boy-boy out of all possible cases of having two children. To find out why, again, let's enumerate all possible combinations:

```
Younger Child | Older Child | Conditional Probability
Girl          Boy           1/3
Boy           Boy           1/3 (*)
Boy           Girl          1/3
Girl          Girl          0 (impossible)
```

Listing 6.5: Example of combinations for boys and girls in case 2.

Again, the intuition is that the additional information makes some cases impossible. There would be four outcomes, but the information given reduces the domain to three possible outcomes (one child is a boy). One of the three cases is boy-boy, therefore the probability $\frac{1}{3}$ or about 33.33%. We have more information in Case 1, which allows us to narrow down the domain of possible outcomes and give a result that matches our intuition. Case 2 looks very similar, but in fact, it includes less information. We have no idea as to whether the older or younger child is a boy, therefore the domain of possible outcomes is larger, resulting in a non-intuitive answer.

These are both problems in conditional probability and we can solve them using the conditional probability formula, rather than enumerating examples.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{6.4}$$

The trick is in how the problem is stated. The outcomes that we are interested in are a sequence, not a single birth event. We are interested in a boy-boy outcome given some information. We can calculate the conditional probabilities using the table of unconditional probabilities listed above. In case 1, we know that the oldest child, or second part of the outcome, is a boy, therefore we can state the problem as follows:

$$P(\text{boy-boy}|\{\text{boy-boy} \cup \text{girl-boy}\}) \tag{6.5}$$

We can calculate the conditional probability as follows:

$$\begin{aligned}
&= \frac{P(\text{boy-boy} \cap \{\text{boy-boy} \cup \text{girl-boy}\})}{P(\text{boy-boy} \cup \text{girl-boy})} \\
&= \frac{P(\text{boy-boy})}{P(\text{boy-boy} \cup \text{girl-boy})} \\
&= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{4}} \\
&= \frac{\frac{1}{4}}{\frac{2}{4}} \\
&= \frac{0.25}{0.5} \\
&= 0.5
\end{aligned} \tag{6.6}$$

Note the simplification of the numerator from $P(\text{boy-boy} \cap \{\text{ boy-boy} \cup \text{ girl-boy}\})$ to $P(\text{boy-boy})$. This is possible because the first event boy-boy is a subset of the second event

{boy-boy ∪ girl-boy}. This is a special case in conditional probability where we say that the occurrence of the first event implies the occurrence of the second event. It is certain therefore we don't have to state it.

$$P(\text{boy-boy} \cap \{\text{boy-boy} \cup \text{girl-boy}\}) = \frac{1}{4} \times 1.0 \tag{6.7}$$

We can summarize this rule for when event $A$ is a subset of event $B$ as follows:

$$P(A \cup B) = P(A), \text{if } A \text{ is a subset of } B \tag{6.8}$$

In case 2, we know one child is a boy, but not whether it is the older or younger child; therefore, we can state the problem as follows:

$$P(\text{boy-boy}|\{\text{boy-boy} \cup \text{girl-boy} \cup \text{boy-girl}\}) \tag{6.9}$$

We can calculate the conditional probability as follows:

$$\begin{aligned}
&= \frac{P(\text{boy-boy} \cap \{\text{boy-boy} \cup \text{girl-boy} \cup \text{boy-girl}\})}{P(\{\text{boy-boy} \cup \text{girl-boy} \cup \text{boy-girl}\})} \\
&= \frac{P(\text{boy-boy})}{P(\{\text{boy-boy} \cup \text{girl-boy} \cup \text{boy-girl}\})} \\
&= \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{4} + \frac{1}{4}} \\
&= \frac{\frac{1}{4}}{\frac{3}{4}} \\
&= \frac{0.25}{0.75} \\
&= 0.333
\end{aligned} \tag{6.10}$$

Again, note we simplified the numerator because the first event is a subset and therefore implies the second event. This is a useful illustration of how we might overcome our incorrect intuitions and achieve the correct answer by first enumerating the possible cases, and second by calculating the conditional probability directly.

## 6.4 Monty Hall Problem

A final classical problem in applied probability is called the game show problem, or the Monty Hall problem. It is based on a real game show called *Let's Make a Deal* and named for the host of the show. The problem can be described generally as follows:

- **Problem**: The contestant is given a choice of three doors. Behind one is a car, behind the other two are goats. Once a door is chosen, the host, who knows where the car is and is not, opens another door, which has a goat, and asks the contestant if they wish to keep their choice or change to the other unopened door.

It is another classical problem because the solution is not intuitive and in the past has caused great confusion and debate. Intuition for the problem says that there is a 1 in 3 or 33%

chance of picking the car initially, and this becomes $\frac{1}{2}$ or 50% once the host opens a door to reveal a goat. **This is incorrect**. We can start by enumerating all combinations and listing the unconditional probabilities. Assume the three doors and the user randomly selects a door, e.g. door 1.

```
Door 1 | Door 2 | Door 3 | Unconditional Probability
Goat    Goat    Car     1/3
Goat    Car     Goat    1/3
Car     Goat    Goat    1/3
```

Listing 6.6: Example of unconditional probability Monty Hall problem.

At this stage, there is a $\frac{1}{3}$ probability of a car, matching our intuition so far. Then, the host opens another door with a goat, in this case, door 2. The opened door was not selected randomly; instead, it was selected with information about where the car is not. Our intuition suggests we remove the second case from the table and update the probability to $\frac{1}{2}$ for each remaining case. **This is incorrect and is the cause of the error**. We can summarize our intuitive conditional probabilities for this scenario as follows:

```
Door 1 | Door 2 | Door 3 | Uncon. | Cond.
Goat    Goat    Car     1/3     1/2
Goat    Car     Goat    1/3     0
Car     Goat    Goat    1/3     1/2
```

Listing 6.7: Example of conditional probability Monty Hall problem.

This would be correct if the contestant did not make a choice before the host opened a door, e.g. if the host opening a door was independent. The trick comes because the contestant made a choice before the host opened a door and this is useful information. It means the host could not open the chosen door (door1) or open a door with a car behind it. The host's choice was dependent upon the first choice of the contestant and then constrained. Instead, we must calculate the probability of switching or not switching, regardless of which door the host opens. Let's look at a table of outcomes given the choice of door 1 and staying or switching.

```
Door 1 | Door 2 | Door 3 | Stay | Switch
Goat    Goat    Car     Goat  Car
Goat    Car     Goat    Goat  Car
Car     Goat    Goat    Car   Goat
```

Listing 6.8: Example of combinations of switching and staying for Monty Hall problem.

We can see that $\frac{2}{3}$ cases of switching result in winning a car (first two rows), and that $\frac{1}{3}$ gives the car if we stay (final row). The contestant has a $\frac{2}{3}$ or 66.66% probability of winning the car if they switch. **They should always switch**. We have solved it by enumerating and counting. Another approach to solving this problem is to calculate the joint probability of the host opening doors to test the stay-versus-switch decision under both cases, in order to maximize the probability of the desired outcome. For example, given that the contestant has chosen door 1, we can calculate the probability of the host opening door 3 if door 1 has the car as follows:

$$\begin{aligned} P(\text{door1=car} \cap \text{door3=open}) &= \frac{1}{3} \times \frac{1}{2} \\ &= 0.333 \times 0.5 \\ &= 0.166 \end{aligned} \tag{6.11}$$

We can then calculate the joint probability of door 2 having the car and the host opening door 3. This is different because if door 2 contains the car, the host can only open door 3; it has a probability of 1.0, a certainty.

$$\begin{aligned} P(\text{door2=car} \cap \text{door3=open}) &= \frac{1}{3} \times 1 \\ &= 0.333 \times 1.0 \\ &= 0.333 \end{aligned} \tag{6.12}$$

Having chosen door 1 and the host opening door 3, the probability is higher that the car is behind door 2 (about 33%) than door 1 (about 16%). We should switch. In this case, we should switch to door 2. Alternately, we can model the choice of the host opening door 2, which has the same structure of probabilities:

$$\begin{aligned} P(\text{door1=car} \cap \text{door2=open}) &= 0.166 \\ P(\text{door3=car} \cap \text{door2=open}) &= 0.333 \end{aligned} \tag{6.13}$$

Again, having chosen door 1 and the host opening door 2, the probability is higher that the car is behind door 3 (about 33%) than door 1 (about 16%). We should switch. If we are seeking to maximize these probabilities, then the best strategy is to switch. Again, in this example, we have seen how we can overcome our faulty intuitions and solve the problem both by enumerating the cases and by using conditional probability.

## 6.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 6.5.1 Books

- *Probability: For the Enthusiastic Beginner*, 2016.
  https://amzn.to/2jULJsu
- *Probability Theory: The Logic of Science*, 2003.
  https://amzn.to/2lnW2pp

### 6.5.2 Articles

- Conditional probability, Wikipedia.
  https://en.wikipedia.org/wiki/Conditional_probability
- Boy or Girl paradox, Wikipedia.
  https://en.wikipedia.org/wiki/Boy_or_Girl_paradox
- Birthday problem, Wikipedia.
  https://en.wikipedia.org/wiki/Birthday_problem
- Monty Hall problem, Wikipedia.
  https://en.wikipedia.org/wiki/Monty_Hall_problem

## 6.6 Summary

In this tutorial, you discovered how to develop an intuition for probability by working through classical thought-provoking problems. Specifically, you learned:

- How to solve the birthday problem by multiplying probabilities together.
- How to solve the boy or girl problem using conditional probability.
- How to solve the Monty Hall problem using joint probability.

### 6.6.1 Next

This was the final tutorial in this Part. In the next Part, you will discover probability distributions.

# Part IV

# Distributions

# Chapter 7

# Probability Distributions

Probability can be used for more than calculating the likelihood of one event; it can summarize the likelihood of all possible outcomes. A thing of interest in probability is called a random variable, and the relationship between each possible outcome for a random variable and their probabilities is called a probability distribution. Probability distributions are an important foundational concept in probability and the names and shapes of common probability distributions will be familiar. The structure and type of the probability distribution varies based on the properties of the random variable, such as continuous or discrete, and this, in turn, impacts how the distribution might be summarized or how to calculate the most likely outcome and its probability. In this tutorial, you will discover a gentle introduction to probability distributions. After reading this tutorial, you will know:

- Random variables in probability have a defined domain and can be continuous or discrete.
- Probability distributions summarize the relationship between possible values and their probability for a random variable.
- Probability density or mass functions map values to probabilities and cumulative distribution functions map outcomes less than or equal to a value to a probability.

Let's get started.

## 7.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Random Variables
2. Probability Distribution
3. Discrete Probability Distributions
4. Continuous Probability Distributions

## 7.2 Random Variables

A random variable is a quantity that is produced by a random process. In probability, a random variable can take on one of many possible values, e.g. events from the state space. A specific value or set of values for a random variable can be assigned a probability.

> In probability modeling, example data or instances are often thought of as being events, observations, or realizations of underlying random variables.

— Page 336, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition. 2016.

A random variable is often denoted as a capital letter, e.g. $X$, and values of the random variable are denoted as a lowercase letter and an index, e.g. $x_1$, $x_2$, $x_3$.

> Upper-case letters like $X$ denote a random variable, while lower-case letters like $x$ denote the value that the random variable takes.

— Page viii, *Probability: For the Enthusiastic Beginner*, 2016.

The values that a random variable can take is called its domain, and the domain of a random variable may be discrete or continuous.

> Variables in probability theory are called random variables and their names begin with an uppercase letter. [...] Every random variable has a domain — the set of possible values it can take on.

— Page 486, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

A discrete random variable has a finite set of states: for example, colors of a car. A random variable that has values true or false is discrete and is referred to as a Boolean random variable: for example, a coin toss. A continuous random variable has a range of numerical values: for example, the height of humans.

- **Discrete Random Variable**. Values are drawn from a finite set of states.
- **Boolean Random Variable**. Values are drawn from the set of {true, false}.
- **Continuous Random Variable**. Values are drawn from a range of real-valued numerical values.

A value of a random variable can be specified via an equals operator: for example, $X = True$. The probability of a random variable is denoted as a function using the upper case $P$ or $Pr$; for example, $P(X)$ is the probability of all values for the random variable $X$. The probability of a value of a random variable can be denoted $P(X = True)$, in this case indicating the probability of the $X$ random variable having the value True.

## 7.3 Probability Distribution

A probability distribution is a summary of probabilities for the possible values of a random variable. As a distribution, the mapping of the values of a random variable to a probability has a shape when all values of the random variable are lined up. The distribution also has general properties that can be measured. Two important properties of a probability distribution are the expected value and the variance. Mathematically, these are referred to as the first and second moments of the distribution. Other moments include the skewness (3rd moment) and the kurtosis (4th moment).

You may be familiar with the mean and variance from statistics, where the concepts are generalized to random variable distributions other than probability distributions. The expected value is the average or mean value of a random variable $X$. This is the most likely value or the outcome with the highest probability. It is typically denoted as a function of the uppercase letter $E$ with square brackets: for example, $E[X]$ for the expected value of $X$ or $E[f(x)]$ where the function $f()$ is used to sample a value from the domain of $X$.

> The expectation value (or the mean) of a random variable $X$ is denoted by $E(X)$.
>
> — Page 134, *Probability: For the Enthusiastic Beginner*, 2016.

The variance is the spread of the values of a random variable from the mean. This is typically denoted as a function $Var$; for example, $Var(X)$ is the variance of the random variable $X$ or $Var(f(x))$ for the variance of values drawn from the domain of $X$ using the function $f()$. The square root of the variance normalizes the value and is referred to as the standard deviation. The variance between two variables is called the covariance and summarizes the linear relationship for how two random variables change together.

- **Expected Value**. The average value of a random variable.
- **Variance**. The average spread of values around the expected value.

Each random variable has its own probability distribution, although the probability distribution of many different random variables may have the same shape. Most common probability distributions can be defined using a few parameters and provide procedures for calculating the expected value and the variance. The structure of the probability distribution will differ depending on whether the random variable is discrete or continuous.

## 7.4 Discrete Probability Distributions

A discrete probability distribution summarizes the probabilities for a discrete random variable. The probability mass function, or PMF, defines the probability distribution for a discrete random variable. It is a function that assigns a probability for specific discrete values. A discrete probability distribution has a cumulative distribution function, or CDF. This is a function that assigns a probability that a discrete random variable will have a value of less than or equal to a specific discrete value.

- **Probability Mass Function**. Probability for a value for a discrete random variable.

- **Cumulative Distribution Function**. Probability less than or equal to a value for a random variable.

The values of the random variable may or may not be ordinal, meaning they may or may not be ordered on a number line, e.g. counts can, car color cannot. In this case, the structure of the PMF and CDF may be discontinuous, or may not form a neat or clean transition in relative probabilities across values. The expected value for a discrete random variable can be calculated from a sample using the mode, e.g. finding the most common value. The sum of probabilities in the PMF equals to one. Some examples of well known discrete probability distributions include:

- Bernoulli and binomial distributions.
- Multinoulli and multinomial distributions.
- Poisson distribution.

Some examples of common domains with well-known discrete probability distributions include:

- The probabilities of dice rolls form a discrete uniform distribution.
- The probabilities of coin flips form a Bernoulli distribution.
- The probabilities car colors form a multinomial distribution.

We will take a closer look at discrete probability distributions in Chapter 8.

## 7.5 Continuous Probability Distributions

A continuous probability distribution summarizes the probability for a continuous random variable. The probability distribution function, or PDF, defines the probability distribution for a continuous random variable. Note the difference in the name from the discrete random variable that has a probability mass function, or PMF. Like a discrete probability distribution, the continuous probability distribution also has a cumulative distribution function, or CDF, that defines the probability of a value less than or equal to a specific numerical value from the domain.

- **Probability Distribution Function**. Probability for a value for a continuous random variable.
- **Cumulative Distribution Function**. Probability less than or equal to a value for a random variable.

As a continuous function, the structure forms a smooth curve. Some examples of well-known continuous probability distributions include:

- Normal or Gaussian distribution.
- Exponential distribution.

- Pareto distribution.

Some examples of domains with well-known continuous probability distributions include:

- The probabilities of the heights of humans form a Normal distribution.
- The probabilities of movies being a hit form a Power-law distribution.
- The probabilities of income levels form a Pareto distribution.

We will take a closer look at continuous probability distributions in Chapter 9.

## 7.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 7.6.1 Books

- *Probability Theory: The Logic of Science*, 2003.
  https://amzn.to/2lnW2pp
- *Introduction to Probability*, 2nd edition, 2019.
  https://amzn.to/2xPvobK
- *Probability: For the Enthusiastic Beginner*, 2016.
  https://amzn.to/2jULJsu

### 7.6.2 Articles

- Random variable, Wikipedia.
  https://en.wikipedia.org/wiki/Random_variable
- Moment (mathematics), Wikipedia.
  https://en.wikipedia.org/wiki/Moment_(mathematics)
- Probability distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Probability_distribution
- List of probability distributions, Wikipedia.
  https://en.wikipedia.org/wiki/List_of_probability_distributions

## 7.7 Summary

In this tutorial, you discovered a gentle introduction to probability distributions. Specifically, you learned:

- Random variables in probability have a defined domain and can be continuous or discrete.

- Probability distributions summarize the relationship between possible values and their probability for a random variable.
- Probability density or mass functions map values to probabilities and cumulative distribution functions map outcomes less than or equal to a value to a probability.

### 7.7.1 Next

In the next tutorial, you will discover how to sample and use discrete probability distributions.

# Chapter 8

# Discrete Probability Distributions

The probability for a discrete random variable can be summarized with a discrete probability distribution. Discrete probability distributions are used in machine learning, most notably in the modeling of binary and multiclass classification problems, but also in evaluating the performance for binary classification models, such as the calculation of confidence intervals, and in the modeling of the distribution of words in text for natural language processing. Knowledge of discrete probability distributions is also required in the choice of activation functions in the output layer of deep learning neural networks for classification tasks and selecting an appropriate loss function. Discrete probability distributions play an important role in applied machine learning and there are a few distributions that a practitioner must know about. In this tutorial, you will discover discrete probability distributions used in machine learning. After completing this tutorial, you will know:

- The probability of outcomes for discrete random variables can be summarized using discrete probability distributions.
- A single binary outcome has a Bernoulli distribution, and a sequence of binary outcomes has a Binomial distribution.
- A single categorical outcome has a Multinoulli distribution, and a sequence of categorical outcomes has a Multinomial distribution.

Let's get started.

## 8.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Discrete Probability Distributions
2. Bernoulli Distribution
3. Binomial Distribution
4. Multinoulli Distribution
5. Multinomial Distribution

## 8.2 Discrete Probability Distributions

A random variable is the quantity produced by a random process. A discrete random variable is a random variable that can have one of a finite set of specific outcomes. The two types of discrete random variables most commonly used in machine learning are binary and categorical.

- **Binary Random Variable**: $x \in \{0, 1\}$
- **Categorical Random Variable**: $x \in \{1, 2, \cdots, K\}$.

A binary random variable is a discrete random variable where the finite set of outcomes is in $\{0, 1\}$. A categorical random variable is a discrete random variable where the finite set of outcomes is in $\{1, 2, \cdots, K\}$, where $K$ is the total number of unique outcomes. Each outcome or event for a discrete random variable has a probability. The relationship between the events for a discrete random variable and their probabilities is called the discrete probability distribution and is summarized by a probability mass function, or PMF for short. For outcomes that can be ordered, the probability of an event equal to or less than a given value is defined by the cumulative distribution function, or CDF for short. The inverse of the CDF is called the percentage-point function and will give the discrete outcome that is less than or equal to a probability.

- **PMF: Probability Mass Function**, returns the probability of a given outcome.
- **CDF: Cumulative Distribution Function**, returns the probability of a value less than or equal to a given outcome.
- **PPF: Percent-Point Function**, returns a discrete value that is less than or equal to the given probability.

There are many common discrete probability distributions. The most common are the Bernoulli and Multinoulli distributions for binary and categorical discrete random variables respectively, and the Binomial and Multinomial distributions that generalize each to multiple independent trials.

- **Binary Random Variable**: Bernoulli Distribution.
- **Sequence of a Binary Random Variable**: Binomial Distribution.
- **Categorical Random Variable**: Multinoulli Distribution.
- **Sequence of a Categorical Random Variable**: Multinomial Distribution.

In the following sections, we will take a closer look at each of these distributions in turn. There are additional discrete probability distributions that you may want to explore, including the Poisson Distribution and the Discrete Uniform Distribution.

## 8.3 Bernoulli Distribution

The Bernoulli distribution is a discrete probability distribution that covers a case where an event will have a binary outcome as either a 0 or 1.

$$x \in \{0, 1\} \tag{8.1}$$

A *Bernoulli trial* is an experiment or case where the outcome follows a Bernoulli distribution. The distribution and the trial are named after the Swiss mathematician Jacob Bernoulli. Some common examples of Bernoulli trials include:

- The single flip of a coin that may have a heads (0) or a tails (1) outcome.
- A single birth of either a boy (0) or a girl (1).

A common example of a Bernoulli trial in machine learning might be a binary classification of a single example as the first class (0) or the second class (1). The distribution can be summarized by a single variable $p$ that defines the probability of an outcome 1. Given this parameter, the probability for each event can be calculated as follows:

$$\begin{aligned} P(x = 1) &= p \\ P(x = 0) &= 1 - p \end{aligned} \tag{8.2}$$

In the case of flipping a fair coin, the value of $p$ would be 0.5, giving a 50% probability of each outcome.

## 8.4 Binomial Distribution

The repetition of multiple independent Bernoulli trials is called a Bernoulli process. The outcomes of a Bernoulli process will follow a Binomial distribution. As such, the Bernoulli distribution would be a Binomial distribution with a single trial. Some common examples of Bernoulli processes include:

- A sequence of independent coin flips.
- A sequence of independent births.

The performance of a machine learning algorithm on a binary classification problem can be analyzed as a Bernoulli process, where the prediction by the model on an example from a test set is a Bernoulli trial (correct or incorrect). The Binomial distribution summarizes the number of successes in a given number of Bernoulli trials $k$, with a given probability of success for each trial $p$. We can demonstrate this with a Bernoulli process where the probability of success is 30% or $P(x = 1) = 0.3$ and the total number of trials is 100 ($k = 100$).

We can simulate the Bernoulli process with randomly generated cases and count the number of successes over the given number of trials. This can be achieved via the `binomial()` NumPy function. This function takes the total number of trials and probability of success as arguments and returns the number of successful outcomes across the trials for one simulation.

```
# example of simulating a binomial process and counting success
from numpy.random import binomial
# define the parameters of the distribution
p = 0.3
k = 100
# run a single simulation
success = binomial(k, p)
print('Total Success: %d' % success)
```

Listing 8.1: Example of simulating a binomial process.

We would expect that 30 cases out of 100 would be successful given the chosen parameters ($k \times p$ or $100 \times 0.3$). A different random sequence of 100 trials will result each time the code is run, so your specific results will differ. Try running the example a few times. In this case, we can see that we get slightly less than the expected 30 successful trials.

```
Total Successes: 28
```

Listing 8.2: Example output from simulating a binomial process.

We can calculate the moments of this distribution, specifically the expected value or mean and the variance using the `binom.stats()` SciPy function.

```
# calculate moments of a binomial distribution
from scipy.stats import binom
# define the parameters of the distribution
p = 0.3
k = 100
# calculate moments
mean, var, _, _ = binom.stats(k, p, moments='mvsk')
print('Mean=%.3f, Variance=%.3f' % (mean, var))
```

Listing 8.3: Example of calculating the moments of a binomial distribution.

Running the example reports the expected value of the distribution, which is 30, as we would expect, as well as the variance of 21, which if we calculate the square root, gives us the standard deviation of about 4.5.

```
Mean=30.000, Variance=21.000
```

Listing 8.4: Example output from calculating the moments of a binomial distribution.

We can use the probability mass function to calculate the likelihood of different numbers of successful outcomes for a sequence of trials, such as 10, 20, 30, to 100. We would expect 30 successful outcomes to have the highest probability.

```
# example of using the pmf for the binomial distribution
from scipy.stats import binom
# define the parameters of the distribution
p = 0.3
k = 100
# define the distribution
dist = binom(k, p)
# calculate the probability of n successes
for n in range(10, 110, 10):
  print('P of %d success: %.3f%%' % (n, dist.pmf(n)*100))
```

Listing 8.5: Example of calculating probabilities for values from the binomial mass function.

Running the example defines the binomial distribution and calculates the probability for each number of successful outcomes in [10, 100] in groups of 10. The probabilities are multiplied by 100 to give percentages, and we can see that 30 successful outcomes has the highest probability at about 8.6%.

```
P of 10 success: 0.000%
P of 20 success: 0.758%
P of 30 success: 8.678%
P of 40 success: 0.849%
P of 50 success: 0.001%
P of 60 success: 0.000%
P of 70 success: 0.000%
P of 80 success: 0.000%
P of 90 success: 0.000%
P of 100 success: 0.000%
```

Listing 8.6: Example output from calculating probabilities for values from the binomial mass function.

Given the probability of success is 30% for one trial, we would expect that a probability of 50 or fewer successes out of 100 trials to be close to 100%. We can calculate this with the cumulative distribution function, demonstrated below.

```
# example of using the cdf for the binomial distribution
from scipy.stats import binom
# define the parameters of the distribution
p = 0.3
k = 100
# define the distribution
dist = binom(k, p)
# calculate the probability of <=n successes
for n in range(10, 110, 10):
  print('P of %d success: %.3f%%' % (n, dist.cdf(n)*100))
```

Listing 8.7: Example of calculating probabilities for values from the binomial cumulative function.

Running the example prints each number of successes in [10, 100] in groups of 10 and the probability of achieving that many success or less over 100 trials. As expected, after 50 successes or less covers 99.999% of the successes expected to happen in this distribution.

```
P of 10 success: 0.000%
P of 20 success: 1.646%
P of 30 success: 54.912%
P of 40 success: 98.750%
P of 50 success: 99.999%
P of 60 success: 100.000%
P of 70 success: 100.000%
P of 80 success: 100.000%
P of 90 success: 100.000%
P of 100 success: 100.000%
```

Listing 8.8: Example output from calculating probabilities for values from the binomial cumulative function.

## 8.5 Multinoulli Distribution

The Multinoulli distribution, also called the categorical distribution, covers the case where an event will have one of $K$ possible outcomes.

$$x \in \{1, 2, 3, \cdots, K\} \tag{8.3}$$

It is a generalization of the Bernoulli distribution from a binary variable to a categorical variable, where the number of cases $K$ for the Bernoulli distribution is set to 2, $K = 2$. A common example that follows a Multinoulli distribution is:

- A single roll of a die that will have an outcome in $\{1, 2, 3, 4, 5, 6\}$, e.g. $K = 6$.

A common example of a Multinoulli distribution in machine learning might be a multiclass classification of a single example into one of $K$ classes, e.g. one of three different species of the iris flower. The distribution can be summarized with $p$ variables from $p_1$ to $p_K$, each defining the probability of a given categorical outcome from 1 to $K$, and where all probabilities sum to 1.0.

$$\begin{aligned} P(x = 1) &= p_1 \\ P(x = 2) &= p_1 \\ P(x = 3) &= p_3 \\ &\cdots \\ P(x = K) &= p_K \end{aligned} \tag{8.4}$$

In the case of a single roll of a die, the probabilities for each value would be $\frac{1}{6}$, or about 0.166 or about 16.6%.

## 8.6 Multinomial Distribution

The repetition of multiple independent Multinoulli trials will follow a multinomial distribution. The multinomial distribution is a generalization of the binomial distribution for a discrete variable with $K$ outcomes. An example of a multinomial process includes a sequence of independent dice rolls. A common example of the multinomial distribution is the occurrence counts of words in a text document, from the field of natural language processing. A multinomial distribution is summarized by a discrete random variable with $K$ outcomes, a probability for each outcome from $p_1$ to $p_K$, and $n$ successive trials.

We can demonstrate this with a small example with 3 categories ($K = 3$) with equal probability ($p$=33.33%) and 100 trials. Firstly, we can use the `multinomial()` NumPy function to simulate 100 independent trials and summarize the number of times that the event resulted in each of the given categories. The function takes both the number of trials and the probabilities for each category as a list. The complete example is listed below.

```
# example of simulating a multinomial process
from numpy.random import multinomial
# define the parameters of the distribution
p = [1.0/3.0, 1.0/3.0, 1.0/3.0]
k = 100
```

```
# run a single simulation
cases = multinomial(k, p)
# summarize cases
for i in range(len(cases)):
  print('Case %d: %d' % (i+1, cases[i]))
```

Listing 8.9: Example of simulating a multinomial process.

We would expect each category to have about 33 events. Running the example reports each case and the number of events. A different random sequence of 100 trials will result each time the code is run, so your specific results will differ. Try running the example a few times. In this case, we see a spread of cases as high as 37 and as low as 30.

```
Case 1: 37
Case 2: 33
Case 3: 30
```

Listing 8.10: Example output from simulating a multinomial process.

We might expect the idealized case of 100 trials to result in 33, 33, and 34 cases for events 1, 2 and 3 respectively. We can calculate the probability of this specific combination occurring in practice using the probability mass function or `multinomial.pmf()` SciPy function. The complete example is listed below.

```
# calculate the probability for a given number of events of each type
from scipy.stats import multinomial
# define the parameters of the distribution
p = [1.0/3.0, 1.0/3.0, 1.0/3.0]
k = 100
# define the distribution
dist = multinomial(k, p)
# define a specific number of outcomes from 100 trials
cases = [33, 33, 34]
# calculate the probability for the case
pr = dist.pmf(cases)
# print as a percentage
print('Case=%s, Probability: %.3f%%' % (cases, pr*100))
```

Listing 8.11: Example of calculating probabilities for values using the multinomial mass function.

Running the example reports the probability of less than 1% for the idealized number of cases of [33, 33, 34] for each event type.

```
Case=[33, 33, 34], Probability: 0.813%
```

Listing 8.12: Example output from calculating probabilities for values using the multinomial mass function.

## 8.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 8.7.1 Books

- Chapter 2: Probability Distributions, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Section 3.9: Common Probability Distributions, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Section 2.3: Some common discrete distributions, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 8.7.2 API

- Discrete Statistical Distributions, SciPy.
  https://docs.scipy.org/doc/scipy/reference/tutorial/stats/discrete.html
- `scipy.stats.bernoulli` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bernoulli.html
- `scipy.stats.binom` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.binom.html
- `scipy.stats.multinomial` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multinomial.html

### 8.7.3 Articles

- Bernoulli distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Bernoulli_distribution
- Bernoulli process, Wikipedia.
  https://en.wikipedia.org/wiki/Bernoulli_process
- Bernoulli trial, Wikipedia.
  https://en.wikipedia.org/wiki/Bernoulli_trial
- Binomial distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Binomial_distribution
- Categorical distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Categorical_distribution
- Multinomial distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Multinomial_distribution

## 8.8 Summary

In this tutorial, you discovered discrete probability distributions used in machine learning. Specifically, you learned:

- The probability of outcomes for discrete random variables can be summarized using discrete probability distributions.
- A single binary outcome has a Bernoulli distribution, and a sequence of binary outcomes has a Binomial distribution.
- A single categorical outcome has a Multinoulli distribution, and a sequence of categorical outcomes has a Multinomial distribution.

### 8.8.1 Next

In the next tutorial, you will discover how to sample and use continuous probability distributions.

# Chapter 9

# Continuous Probability Distributions

The probability for a continuous random variable can be summarized with a continuous probability distribution. Continuous probability distributions are encountered in machine learning, most notably in the distribution of numerical input and output variables for models and in the distribution of errors made by models. Knowledge of the normal continuous probability distribution is also required more generally in the density and parameter estimation performed by many machine learning models. As such, continuous probability distributions play an important role in applied machine learning and there are a few distributions that a practitioner must know about. In this tutorial, you will discover continuous probability distributions used in machine learning. After completing this tutorial, you will know:

- The probability of outcomes for continuous random variables can be summarized using continuous probability distributions.
- How to parameterize, define, and randomly sample from common continuous probability distributions.
- How to create probability density and cumulative density plots for common continuous probability distributions.

Let's get started.

## 9.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Continuous Probability Distributions
2. Normal Distribution
3. Exponential Distribution
4. Pareto Distribution

## 9.2 Continuous Probability Distributions

A random variable is a quantity produced by a random process. A continuous random variable is a random variable that has a real numerical value. Each numerical outcome of a continuous random variable can be assigned a probability. The relationship between the events for a continuous random variable and their probabilities is called the continuous probability distribution and is summarized by a probability density function, or PDF for short.

Unlike a discrete random variable, the probability for a given continuous random variable cannot be specified directly; instead, it is calculated as an integral (area under the curve) for a tiny interval around the specific outcome. The probability of an event equal to or less than a given value is defined by the cumulative distribution function, or CDF for short. The inverse of the CDF is called the percentage-point function and will give the discrete outcome that is less than or equal to a probability.

- **PDF: Probability Density Function**, returns the probability of a given continuous outcome.
- **CDF: Cumulative Distribution Function**, returns the probability of a value less than or equal to a given outcome.
- **PPF: Percent-Point Function**, returns a discrete value that is less than or equal to the given probability.

There are many common continuous probability distributions. The most common is the normal probability distribution. Practically all continuous probability distributions of interest belong to the so-called exponential family of distributions, which are just a collection of parameterized probability distributions (e.g. distributions that change based on the values of parameters).

Continuous probability distributions play an important role in machine learning from the distribution of input variables to the models, the distribution of errors made by models, and in the models themselves when estimating the mapping between inputs and outputs. In the following sections, will take a closer look at some of the more common continuous probability distributions.

## 9.3 Normal Distribution

The normal distribution is also called the Gaussian distribution (named for Carl Friedrich Gauss) or the bell curve distribution. The distribution covers the probability of real-valued events from many different problem domains, making it a common and well-known distribution, hence the name *normal*. A continuous random variable that has a normal distribution is said to be *normal* or *normally distributed*. Some examples of domains that have normally distributed events include:

- The heights of people.
- The weights of babies.
- The scores on a test.

The distribution can be defined using two parameters:

- **Mean** (mu or $\mu$): The expected value.
- **Variance** (sigma$^2$ or $\sigma^2$): The spread from the mean.

Often, the standard deviation is used instead of the variance, which is calculated as the square root of the variance, e.g. normalized.

- **Standard Deviation** (sigma or $\sigma$): The average spread from the mean.

A normal distribution with a mean of zero and a standard deviation of 1 is called a standard normal distribution, and often data is reduced or *standardized* to this for analysis for ease of interpretation and comparison. We can define a distribution with a mean of 50 and a standard deviation of 5 and sample random numbers from this distribution. We can achieve this using the `normal()` NumPy function. The example below samples and prints 10 numbers from this distribution.

```
# sample a normal distribution
from numpy.random import normal
# define the distribution
mu = 50
sigma = 5
n = 10
# generate the sample
sample = normal(mu, sigma, n)
print(sample)
```

Listing 9.1: Example of sampling from a normal distribution.

Running the example prints 10 numbers randomly sampled from the defined normal distribution.

```
[48.71009029 49.36970461 45.58247748 51.96846616 46.05793544 40.3903483
 48.39189421 50.08693721 46.85896352 44.83757824]
```

Listing 9.2: Example output from sampling from a normal distribution.

A sample of data can be checked to see if it is normal by plotting it and checking for the familiar normal shape, or by using statistical tests. If the samples of observations of a random variable are normally distributed, then they can be summarized by just the mean and variance, calculated directly on the samples. We can calculate the probability of each observation using the probability density function. A plot of these values would give us the tell-tale bell shape. We can define a normal distribution using the `norm()` SciPy function and then calculate properties such as the moments, PDF, CDF, and more. The example below calculates the probability for integer values between 30 and 70 in our distribution and plots the result, then does the same for the cumulative probability.

```
# pdf and cdf for a normal distribution
from scipy.stats import norm
from matplotlib import pyplot
# define distribution parameters
mu = 50
sigma = 5
```

```
# create distribution
dist = norm(mu, sigma)
# plot pdf
values = [value for value in range(30, 70)]
probabilities = [dist.pdf(value) for value in values]
pyplot.plot(values, probabilities)
pyplot.show()
# plot cdf
cprobs = [dist.cdf(value) for value in values]
pyplot.plot(values, cprobs)
pyplot.show()
```

Listing 9.3: Example of plotting the PDF and CDF for the normal distribution.

Running the example first calculates the probability for integers in the range [30, 70] and creates a line plot of values and probabilities. The plot shows the Gaussian or bell-shape with the peak of highest probability around the expected value or mean of 50 with a probability of about 8%.

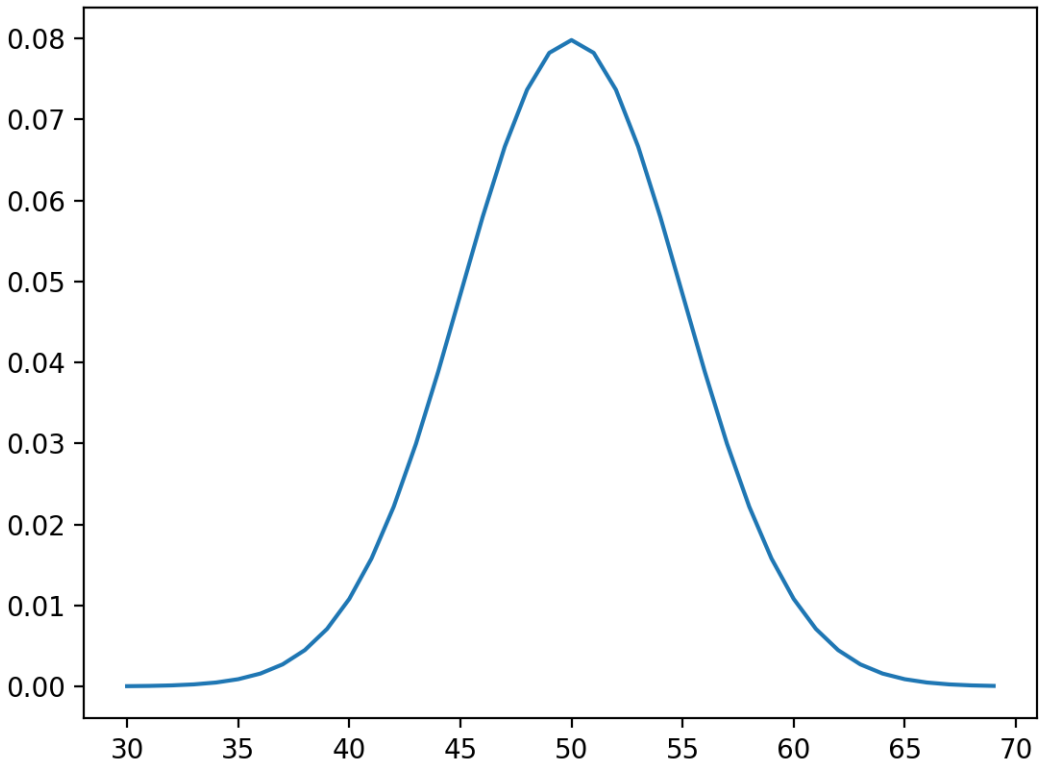


Figure 9.1: Line Plot of Events vs Probability or the Probability Density Function for the Normal Distribution.

The cumulative probabilities are then calculated for observations over the same range, showing that at the mean, we have covered about 50% of the expected values and very close to 100% after the value of about 65 or 3 standard deviations from the mean $(50 + (3 \times 5))$.

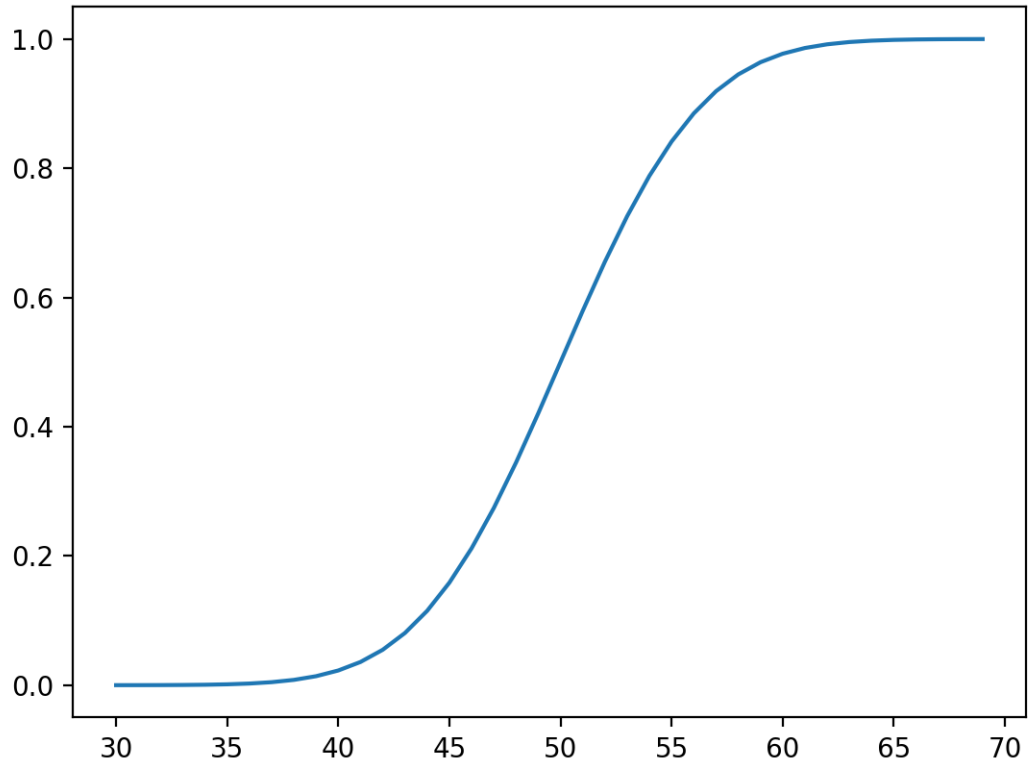


Figure 9.2: Line Plot of Events vs. Cumulative Probability or the Cumulative Density Function for the Normal Distribution.

In fact, the normal distribution has a heuristic or rule of thumb that defines the percentage of data covered by a given range by the number of standard deviations from the mean. It is called the 68-95-99.7 rule, which is the approximate percentage of the data covered by ranges defined by 1, 2, and 3 standard deviations from the mean. For example, in our distribution with a mean of 50 and standard deviation of 5, we would expect 95% of the data to be covered by values that are 2 standard deviations from the mean, or $50 - (2 \times 5)$ and $50 + (2 \times 5)$ or between 40 and 60. We can confirm this by calculating the exact values using the percentage-point function. The middle 95% would be defined by the percentage point function value for 2.5% at the low end and 97.5% at the high end, where 97.5 to 2.5 gives the middle 95%. The complete example is listed below.

```
# calculate the values that define the middle 95%
from scipy.stats import norm
# define distribution parameters
mu = 50
sigma = 5
# create distribution
dist = norm(mu, sigma)
low_end = dist.ppf(0.025)
high_end = dist.ppf(0.975)
print('Middle 95%% between %.1f and %.1f' % (low_end, high_end))
```

Listing 9.4: Example of calculating the values that define the middle of the distribution.

Running the example gives the exact outcomes that define the middle 95% of expected outcomes that are very close to our standard-deviation-based heuristics of 40 and 60.

```
Middle 95% between 40.2 and 59.8
```

Listing 9.5: Example output from calculating the values that define the middle of the distribution.

An important related distribution is the Log-Normal probability distribution.

## 9.4 Exponential Distribution

The exponential distribution is a continuous probability distribution where a few outcomes are the most likely with a rapid decrease in probability to all other outcomes. It is the continuous random variable equivalent to the geometric probability distribution for discrete random variables. Some examples of domains that have exponential distribution events include:

- The time between clicks on a Geiger counter.
- The time until the failure of a part.
- The time until the default of a loan.

The distribution can be defined using one parameter:

- **Scale** (Beta or $\beta$): The mean and standard deviation of the distribution.

Sometimes the distribution is defined more formally with a parameter lambda or rate. The beta parameter is defined as the reciprocal of the lambda parameter ($\beta = \frac{1}{\lambda}$)

- **Rate** (lambda or $\lambda$) = Rate of change in the distribution.

We can define a distribution with a mean of 50 and sample random numbers from this distribution. We can achieve this using the `exponential()` NumPy function. The example below samples and prints 10 numbers from this distribution.

```
# sample an exponential distribution
from numpy.random import exponential
# define the distribution
beta = 50
n = 10
# generate the sample
sample = exponential(beta, n)
print(sample)
```

Listing 9.6: Example of sampling from an exponential distribution.

Running the example prints 10 numbers randomly sampled from the defined distribution.

```
[ 3.32742946 39.10165624 41.86856606 85.0030387 28.18425491
  68.20434637 106.34826579 19.63637359 17.13805423 15.91135881]
```

Listing 9.7: Example output from sampling from a exponential distribution.

We can define an exponential distribution using the `expon()` SciPy function and then calculate properties such as the moments, PDF, CDF, and more. The example below defines a range of observations between 50 and 70 and calculates the probability and cumulative probability for each and plots the result.

```
# pdf and cdf for an exponential distribution
from scipy.stats import expon
from matplotlib import pyplot
# define distribution parameter
beta = 50
# create distribution
dist = expon(beta)
# plot pdf
values = [value for value in range(50, 70)]
probabilities = [dist.pdf(value) for value in values]
pyplot.plot(values, probabilities)
pyplot.show()
# plot cdf
cprobs = [dist.cdf(value) for value in values]
pyplot.plot(values, cprobs)
pyplot.show()
```

Listing 9.8: Example of plotting the PDF and CDF for the exponential distribution.

Running the example first creates a line plot of outcomes versus probabilities, showing a familiar exponential probability distribution shape.

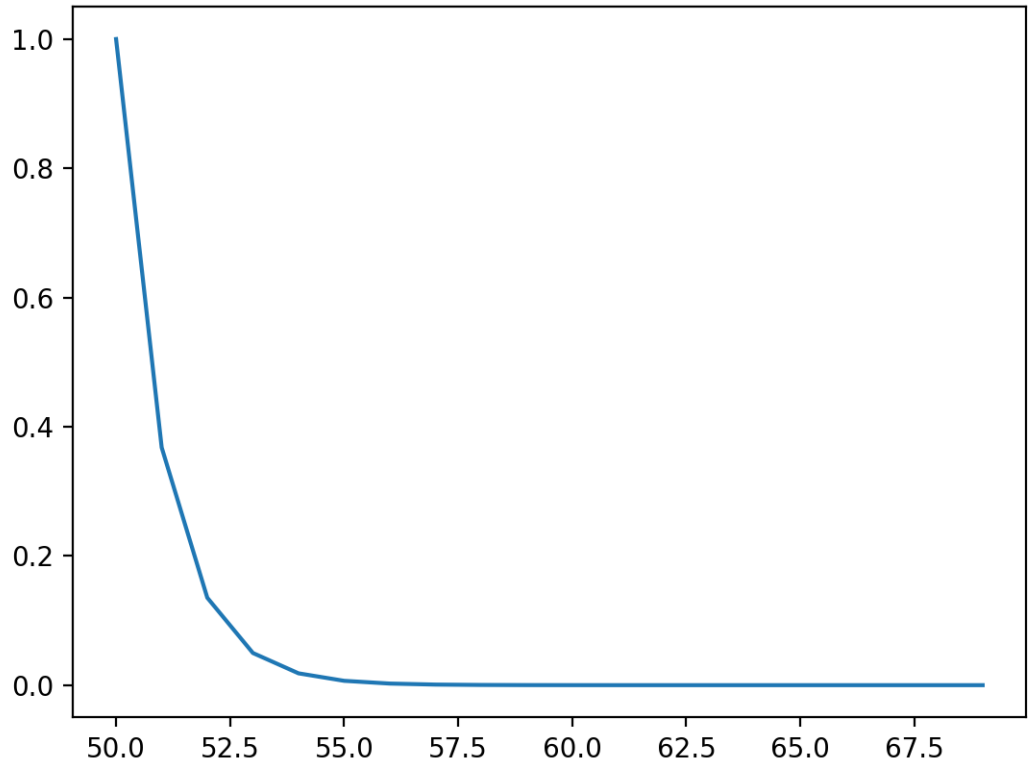


Figure 9.3: Line Plot of Events vs. Probability or the Probability Density Function for the Exponential Distribution.

Next, the cumulative probabilities for each outcome are calculated and graphed as a line plot, showing that after perhaps a value of 55 that almost 100% of the expected values will be observed.

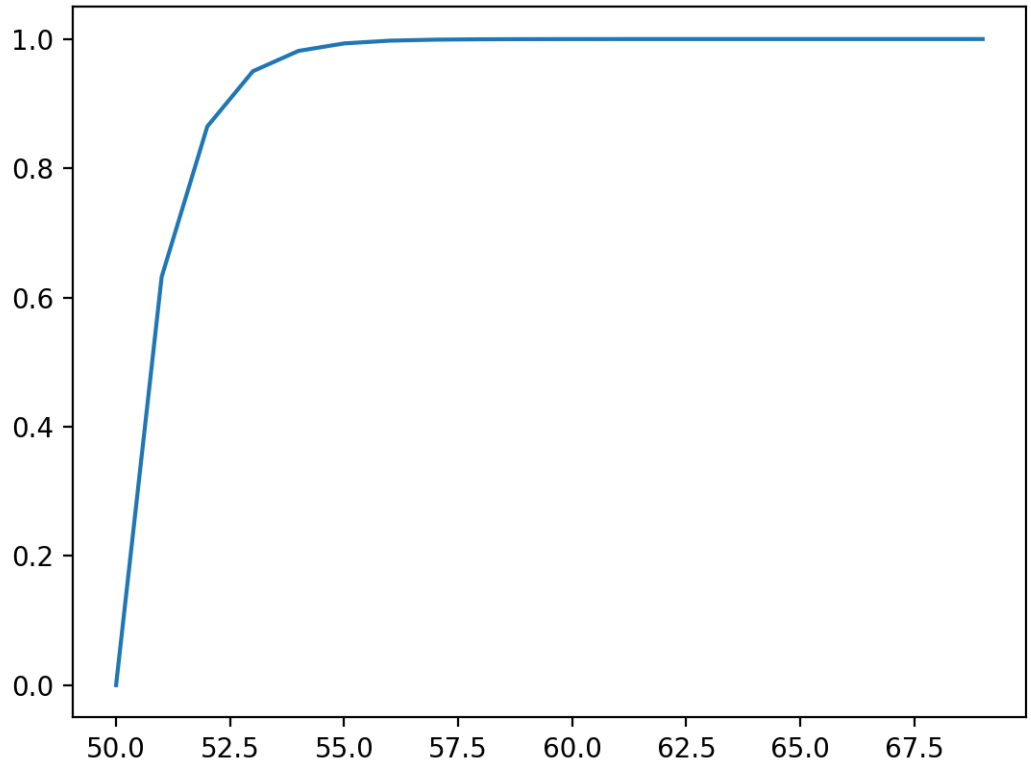


Figure 9.4: Line Plot of Events vs. Cumulative Probability or the Cumulative Density Function for the Exponential Distribution.

An important related distribution is the double exponential distribution, also called the Laplace distribution.

## 9.5 Pareto Distribution

A Pareto distribution is named after Vilfredo Pareto and is may be referred to as a power-law distribution. It is also related to the Pareto principle (or 80/20 rule) which is a heuristic for continuous random variables that follow a Pareto distribution, where 80% of the events are covered by 20% of the range of outcomes, e.g. most events are drawn from just 20% of the range of the continuous variable. The Pareto principle is just a heuristic for a specific Pareto distribution, specifically the Pareto Type II distribution, that is perhaps most interesting and on which we will focus. Some examples of domains that have Pareto distributed events include:

- The income of households in a country.
- The total sales of books.
- The scores by players on a sports team.

The distribution can be defined using one parameter:

- **Shape** (alpha or $\alpha$): The steepness of the decease in probability.

Values for the shape parameter are often small, such as between 1 and 3, with the Pareto principle given when alpha is set to 1.161. We can define a distribution with a shape of 1.1 and sample random numbers from this distribution. We can achieve this using the `pareto()` NumPy function.

```
# sample a pareto distribution
from numpy.random import pareto
# define the distribution
alpha = 1.1
n = 10
# generate the sample
sample = pareto(alpha, n)
print(sample)
```

Listing 9.9: Example of sampling from an Pareto distribution.

Running the example prints 10 numbers randomly sampled from the defined distribution.

```
[0.5049704 0.0140647 2.13105224 3.10991217 2.87575892 1.06602639
 0.22776379 0.37405415 0.96618778 3.94789299]
```

Listing 9.10: Example output from sampling from a Pareto distribution.

We can define a Pareto distribution using the `pareto()` SciPy function and then calculate properties, such as the moments, PDF, CDF, and more. The example below defines a range of observations between 1 and about 10 and calculates the probability and cumulative probability for each and plots the result.

```
# pdf and cdf for a pareto distribution
from scipy.stats import pareto
from matplotlib import pyplot
# define distribution parameter
alpha = 1.5
# create distribution
dist = pareto(alpha)
# plot pdf
values = [value/10.0 for value in range(10, 100)]
probabilities = [dist.pdf(value) for value in values]
pyplot.plot(values, probabilities)
pyplot.show()
# plot cdf
cprobs = [dist.cdf(value) for value in values]
pyplot.plot(values, cprobs)
pyplot.show()
```

Listing 9.11: Example of plotting the PDF and CDF for the Pareto distribution.

Running the example first creates a line plot of outcomes versus probabilities, showing a familiar Pareto probability distribution shape.

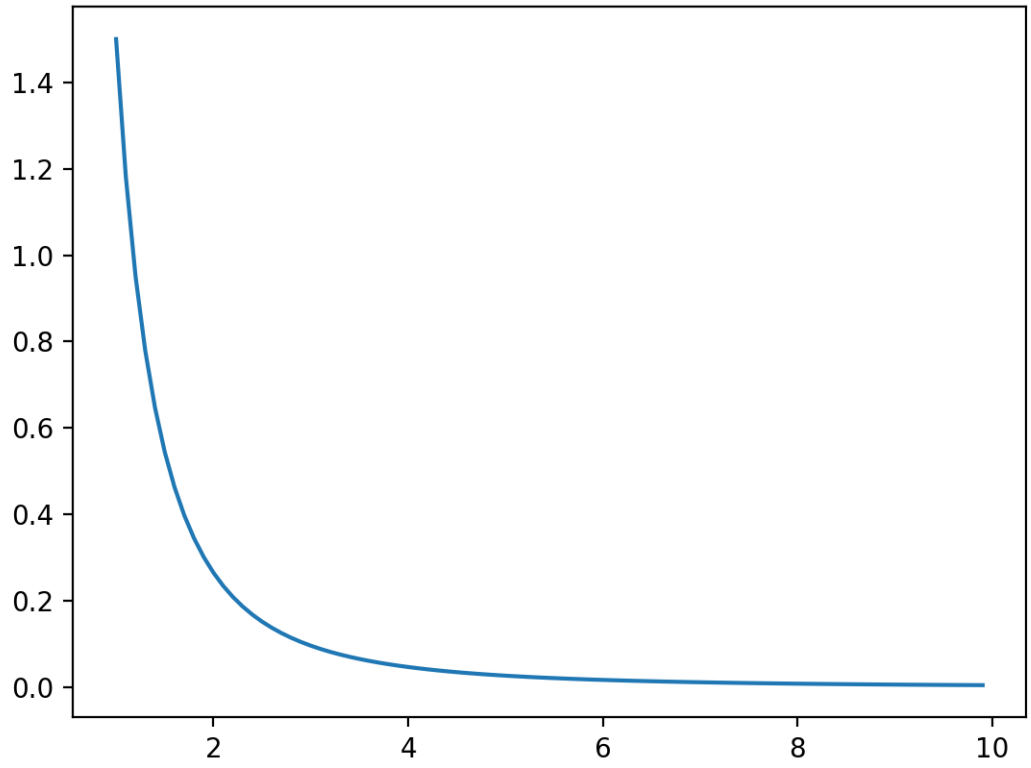


Figure 9.5: Line Plot of Events vs. Probability or the Probability Density Function for the Pareto Distribution.

Next, the cumulative probabilities for each outcome are calculated and graphed as a line plot, showing a rise that is less steep than the exponential distribution seen in the previous section.

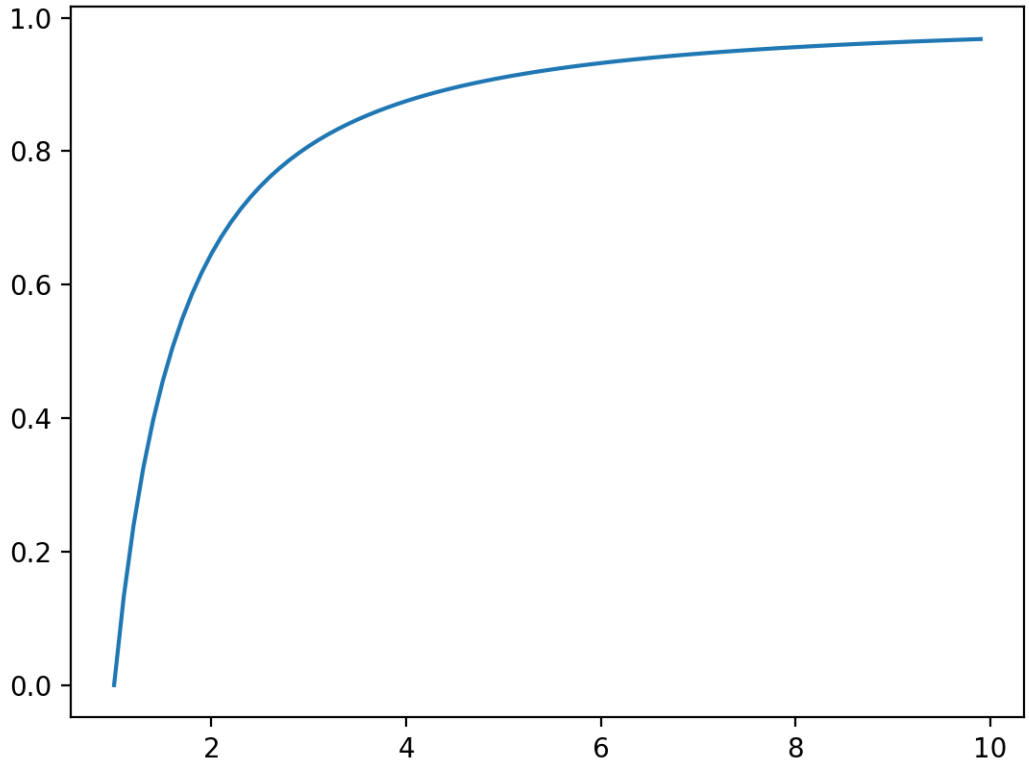


Figure 9.6: Line Plot of Events vs. Cumulative Probability or the Cumulative Density Function for the Pareto Distribution.

## 9.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 9.6.1 Books

- Chapter 2: Probability Distributions, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Section 3.9: Common Probability Distributions, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Section 2.3: Some common discrete distributions, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 9.6.2 API

- Continuous Statistical Distributions, SciPy.
  https://docs.scipy.org/doc/scipy/reference/tutorial/stats/continuous.html
- Random sampling (`numpy.random`), NumPy.
  https://docs.scipy.org/doc/numpy/reference/routines.random.html

### 9.6.3 Articles

- Normal distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Normal_distribution
- 68-95-99.7 rule, Wikipedia.
  https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule
- Exponential distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Exponential_distribution
- Pareto distribution, Wikipedia.
  https://en.wikipedia.org/wiki/Pareto_distribution

## 9.7 Summary

In this tutorial, you discovered continuous probability distributions used in machine learning. Specifically, you learned:

- The probability of outcomes for continuous random variables can be summarized using continuous probability distributions.
- How to parameterize, define, and randomly sample from common continuous probability distributions.
- How to create probability density and cumulative density plots for common continuous probability distributions.

### 9.7.1 Next

In the next tutorial, you will discover the challenge of density estimation and how to fit density estimation models.

# Chapter 10

# Probability Density Estimation

Probability density is the relationship between observations and their probability. Some outcomes of a random variable will have low probability density and other outcomes will have a high probability density. The overall shape of the probability density is referred to as a probability distribution, and the calculation of probabilities for specific outcomes of a random variable is performed by a probability density function, or PDF for short. It is useful to know the probability density function for a sample of data in order to know whether a given observation is unlikely, or so unlikely as to be considered an outlier or anomaly and whether it should be removed. It is also helpful in order to choose appropriate learning methods that require input data to have a specific probability distribution. It is unlikely that the probability density function for a random sample of data is known. As such, the probability density must be approximated using a process known as probability density estimation. In this tutorial, you will discover a gentle introduction to probability density estimation. After completing this tutorial, you will know:

- Histogram plots provide a fast and reliable way to visualize the probability density of a data sample.
- Parametric probability density estimation involves selecting a common distribution and estimating the parameters for the density function from a data sample.
- Nonparametric probability density estimation involves using a technique to fit a model to the arbitrary distribution of the data, like kernel density estimation.

Let's get started.

## 10.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Probability Density
2. Summarize Density With a Histogram
3. Parametric Density Estimation
4. Nonparametric Density Estimation

## 10.2 Probability Density

A random variable $x$ has a probability distribution $p(x)$. The relationship between the outcomes of a random variable and its probability is referred to as the probability density, or simply the *density*. If a random variable is continuous, then the probability can be calculated via probability density function, or PDF for short. The shape of the probability density function across the domain for a random variable is referred to as the probability distribution and common probability distributions have names, such as uniform, normal, exponential, and so on.

Given a random variable, we are interested in the density of its probabilities. For example, given a random sample of a variable, we might want to know things like the shape of the probability distribution, the most likely value, the spread of values, and other properties. Knowing the probability distribution for a random variable can help to calculate moments of the distribution, like the mean and variance, but can also be useful for other more general considerations, like determining whether an observation is unlikely or very unlikely and might be an outlier or anomaly. The problem is, we may not know the probability distribution for a random variable. We rarely do know the distribution because we don't have access to all possible outcomes for a random variable. In fact, all we have access to is a sample of observations. As such, we must select a probability distribution.

This problem is referred to as probability density estimation, or simply *density estimation*, as we are using the observations in a random sample to estimate the general density of probabilities beyond just the sample of data we have available. There are a few steps in the process of density estimation for a random variable. The first step is to review the density of observations in the random sample with a simple histogram. From the histogram, we might be able to identify a common and well-understood probability distribution that can be used, such as a normal distribution. If not, we may have to fit a model to estimate the distribution.

In the following sections, we will take a closer look at each one of these steps in turn. We will focus on univariate data, e.g. one random variable, in this tutorial for simplicity. Although the steps are applicable for multivariate data, they can become more challenging if the number of variables increases.

## 10.3 Summarize Density With a Histogram

The first step in density estimation is to create a histogram of the observations in the random sample. A histogram is a plot that involves first grouping the observations into bins and counting the number of events that fall into each bin. The counts, or frequencies of observations, in each bin are then plotted as a bar graph with the bins on the x-axis and the frequency on the y-axis. The choice of the number of bins is important as it controls the coarseness of the distribution (number of bars) and, in turn, how well the density of the observations is plotted. It is a good idea to experiment with different bin sizes for a given data sample to get multiple perspectives or views on the same data.

For example, observations between 1 and 100 could be split into 3 bins (1-33, 34-66, 67-100), which might be too coarse, or 10 bins (1-10, 11-20, ..., 91-100), which might better capture the density. A histogram can be created using the Matplotlib library and the `hist()` function. The data is provided as the first argument, and the number of bins is specified via the `bins` argument either as an integer (e.g. 10) or as a sequence of the boundaries of each bin (e.g. [1, 34, 67, 100]). The snippet below creates a histogram with 10 bins for a data sample.

```
...
# plot a histogram of the sample
pyplot.hist(sample, bins=10)
pyplot.show()
```

Listing 10.1: Example of plotting a histogram.

We can create a random sample drawn from a normal distribution and pretend we don't know the distribution, then create a histogram of the data. The `normal()` NumPy function will achieve this and we will generate 1,000 samples with a mean of 0 and a standard deviation of 1, e.g. a standard Gaussian. The complete example is listed below.

```
# example of plotting a histogram of a random sample
from matplotlib import pyplot
from numpy.random import normal
# generate a sample
sample = normal(size=1000)
# plot a histogram of the sample
pyplot.hist(sample, bins=10)
pyplot.show()
```

Listing 10.2: Example of plotting a histogram with 10 bins.

Running the example draws a sample of random observations and creates the histogram with 10 bins. We can clearly see the shape of the normal distribution. Note that your results will differ given the random nature of the data sample. Try running the example a few times.

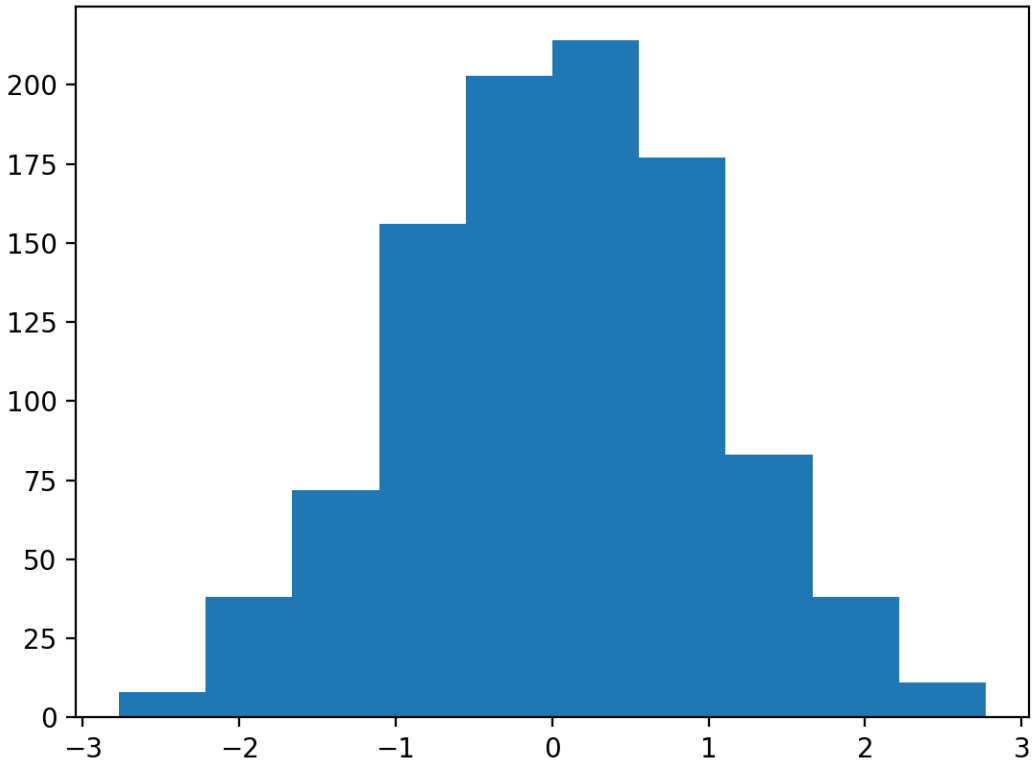


Figure 10.1: Histogram Plot With 10 Bins of a Random Data Sample.

Running the example with bins set to 3 makes the normal distribution less obvious.

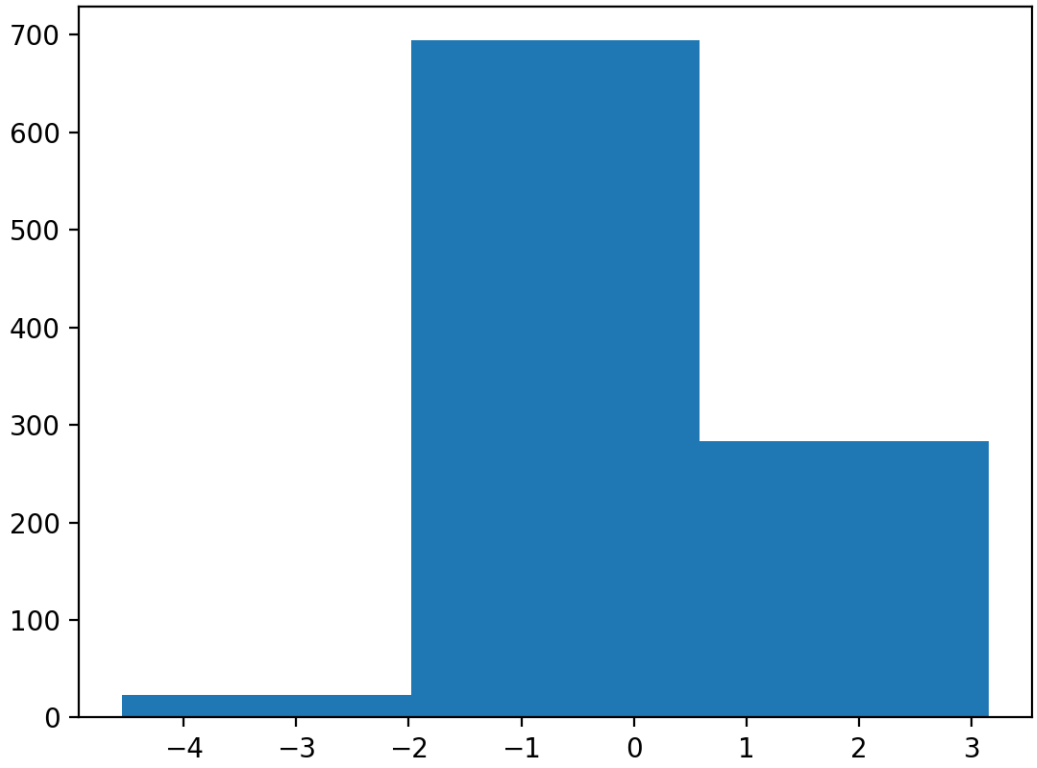


Figure 10.2: Histogram Plot With 3 Bins of a Random Data Sample.

Reviewing a histogram of a data sample with a range of different numbers of bins will help to identify whether the density looks like a common probability distribution or not. In most cases, you will see a unimodal distribution, such as the familiar bell shape of the normal, the flat shape of the uniform, or the descending or ascending shape of an exponential or Pareto distribution. You might also see complex distributions, such as two peaks that don't disappear with different numbers of bins, referred to as a bimodal distribution, or multiple peaks, referred to as a multimodal distribution. You might also see a large spike in density for a given value or small range of values indicating outliers, often occurring on the tail of a distribution far away from the rest of the density.

## 10.4 Parametric Density Estimation

The shape of a histogram of most random samples will match a well-known probability distribution. The common distributions are common because they occur again and again in different and sometimes unexpected domains. Get familiar with the common probability distributions as it will help you to identify a given distribution from a histogram. Once identified, you can attempt to estimate the density of the random variable with a chosen probability distribution. This can be achieved by estimating the parameters of the distribution from a random sample of data.

For example, the normal distribution has two parameters: the mean and the standard deviation. Given these two parameters, we now know the probability distribution function. These parameters can be estimated from data by calculating the sample mean and sample standard deviation. We refer to this process as parametric density estimation. The reason is that we are using predefined functions to summarize the relationship between observations and their probability that can be controlled or configured with parameters, hence *parametric*. Once we have estimated the density, we can check if it is a good fit. This can be done in many ways, such as:

- Plotting the density function and comparing the shape to the histogram.
- Sampling the density function and comparing the generated sample to the real sample.
- Using a statistical test to confirm the data fits the distribution.

We can demonstrate this with an example. We can generate a random sample of 1,000 observations from a normal distribution with a mean of 50 and a standard deviation of 5.

```
...
# generate a sample
sample = normal(loc=50, scale=5, size=1000)
```

Listing 10.3: Example of randomly sampling a normal probability distribution.

We can then pretend that we don't know the probability distribution and maybe look at a histogram and guess that it is normal. Assuming that it is normal, we can then calculate the parameters of the distribution, specifically the mean and standard deviation. We would not expect the mean and standard deviation to be 50 and 5 exactly given the small sample size and noise in the sampling process.

```
...
# calculate parameters
sample_mean = mean(sample)
sample_std = std(sample)
print('Mean=%.3f, Standard Deviation=%.3f' % (sample_mean, sample_std))
```

Listing 10.4: Example of calculating the moments of a data sample.

Then fit the distribution with these parameters, so-called parametric density estimation of our data sample. In this case, we can use the `norm()` SciPy function.

```
...
# define the distribution
dist = norm(sample_mean, sample_std)
```

Listing 10.5: Example of defining a normal probability distribution.

We can then sample the probabilities from this distribution for a range of values in our domain, in this case between 30 and 70.

```
...
# sample probabilities for a range of outcomes
values = [value for value in range(30, 70)]
probabilities = [dist.pdf(value) for value in values]
```

Listing 10.6: Example of calculating the expected probability of specific values.

Finally, we can plot a histogram of the data sample and overlay a line plot of the probabilities calculated for the range of values from the PDF. Importantly, we can convert the counts or frequencies in each bin of the histogram to a normalized probability to ensure the y-axis of the histogram matches the y-axis of the line plot. This can be achieved by setting the `density` argument to `True` in the call to `hist()`.

```
...
# plot the histogram and pdf
pyplot.hist(sample, bins=10, density=True)
pyplot.plot(values, probabilities)
```

Listing 10.7: Example of plotting observed values versus expected probabilities.

Tying these snippets together, the complete example of parametric density estimation is listed below.

```
# example of parametric probability density estimation
from matplotlib import pyplot
from numpy.random import normal
from numpy import mean
from numpy import std
from scipy.stats import norm
# generate a sample
sample = normal(loc=50, scale=5, size=1000)
# calculate parameters
sample_mean = mean(sample)
sample_std = std(sample)
print('Mean=%.3f, Standard Deviation=%.3f' % (sample_mean, sample_std))
# define the distribution
dist = norm(sample_mean, sample_std)
# sample probabilities for a range of outcomes
values = [value for value in range(30, 70)]
probabilities = [dist.pdf(value) for value in values]
# plot the histogram and pdf
pyplot.hist(sample, bins=10, density=True)
pyplot.plot(values, probabilities)
pyplot.show()
```

Listing 10.8: Example of parametric probability density estimation.

Running the example first generates the data sample, then estimates the parameters of the normal probability distribution. Note that your results will differ given the random nature of the data sample. Try running the example a few times. In this case, we can see that the mean and standard deviation have some noise and are slightly different from the expected values of 50 and 5 respectively. The noise is minor and the distribution is expected to still be a good fit.

```
Mean=49.852, Standard Deviation=5.023
```

Listing 10.9: Example output from parametric probability density estimation.

Next, the PDF is fit using the estimated parameters and the histogram of the data with 10 bins is compared to probabilities for a range of values sampled from the PDF. We can see that the PDF is a good match for our data.

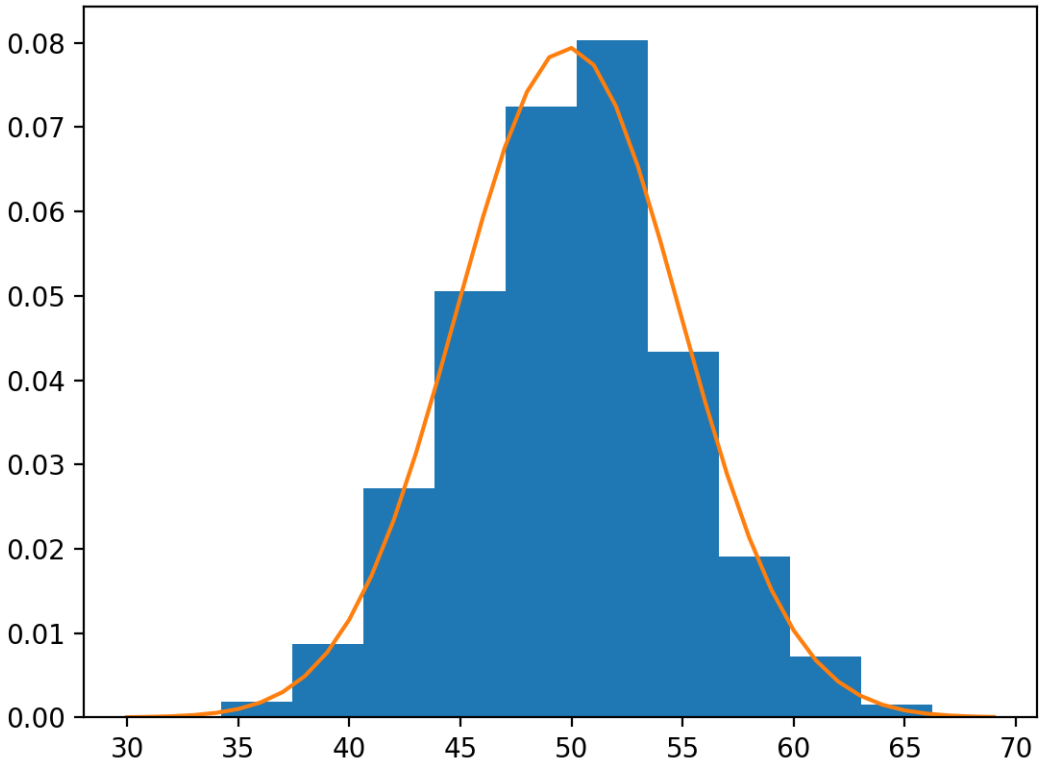


Figure 10.3: Data Sample Histogram With Probability Density Function Overlay for the Normal Distribution.

It is possible that the data does match a common probability distribution, but requires a transformation before parametric density estimation. For example, you may have outlier values that are far from the mean or center of mass of the distribution. This may have the effect of giving incorrect estimates of the distribution parameters and, in turn, causing a poor fit to the data. These outliers should be removed prior to estimating the distribution parameters. Another example is the data may have a skew or be shifted left or right. In this case, you might need to transform the data prior to estimating the parameters, such as taking the log or square root, or more generally, using a power transform like the Box-Cox transform. These types of modifications to the data may not be obvious and effective parametric density estimation may require an iterative process of:

- Loop Until Fit of Distribution to Data is Good Enough:
    1. Estimating distribution parameters
    2. Reviewing the resulting PDF against the data
    3. Transforming the data to better fit the distribution

## 10.5 Nonparametric Density Estimation

In some cases, a data sample may not resemble a common probability distribution or cannot be easily made to fit the distribution. This is often the case when the data has two peaks (bimodal distribution) or many peaks (multimodal distribution). In this case, parametric density estimation is not feasible and alternative methods can be used that do not use a common distribution. Instead, an algorithm is used to approximate the probability distribution of the data without a pre-defined distribution, referred to as a nonparametric method.

The distributions will still have parameters but are not directly controllable in the same way as simple probability distributions. For example, a nonparametric method might estimate the density using all observations in a random sample, in effect making all observations in the sample *parameters*. Perhaps the most common nonparametric approach for estimating the probability density function of a continuous random variable is called kernel smoothing, or kernel density estimation, KDE for short.

- **Kernel Density Estimation**: Nonparametric method for using a dataset to estimating probabilities for new points.

In this case, a kernel is a mathematical function that returns a probability for a given value of a random variable. The kernel effectively smooths or interpolates the probabilities across the range of outcomes for a random variable such that the sum of probabilities equals one, a requirement of well-behaved probabilities. The kernel function weights the contribution of observations from a data sample based on their relationship or distance to a given query sample for which the probability is requested. A parameter, called the smoothing parameter or the bandwidth, controls the scope, or window of observations, from the data sample that contributes to estimating the probability for a given sample. As such, kernel density estimation is sometimes referred to as a Parzen-Rosenblatt window, or simply a Parzen window, after the developers of the method.

- **Smoothing Parameter** (*bandwidth*): Parameter that controls the number of samples or window of samples used to estimate the probability for a new point.

A large window may result in a coarse density with little details, whereas a small window may have too much detail and not be smooth or general enough to correctly cover new or unseen examples. The contribution of samples within the window can be shaped using different functions, sometimes referred to as basis functions, e.g. uniform normal, etc., with different effects on the smoothness of the resulting density function.

- **Basis Function** (*kernel*): The function chosen used to control the contribution of samples in the dataset toward estimating the probability of a new point.

As such, it may be useful to experiment with different window sizes and different contribution functions and evaluate the results against histograms of the data. We can demonstrate this with an example. First, we can construct a bimodal distribution by combining samples from two different normal distributions. Specifically, 300 examples with a mean of 20 and a standard deviation of 5 (the smaller peak), and 700 examples with a mean of 40 and a standard deviation of 5 (the larger peak). The means were chosen close together to ensure the distributions overlap in the combined sample. The complete example of creating this sample with a bimodal probability distribution and plotting the histogram is listed below.

```
# example of a bimodal data sample
from matplotlib import pyplot
from numpy.random import normal
from numpy import hstack
# generate a sample
sample1 = normal(loc=20, scale=5, size=300)
sample2 = normal(loc=40, scale=5, size=700)
sample = hstack((sample1, sample2))
# plot the histogram
pyplot.hist(sample, bins=50)
pyplot.show()
```

Listing 10.10: Example of generating and plotting a bimodal data sample.

Running the example creates the data sample and plots the histogram. Note that your results will differ given the random nature of the data sample. Try running the example a few times. We have fewer samples with a mean of 20 than samples with a mean of 40, which we can see reflected in the histogram with a larger density of samples around 40 than around 20. Data with this distribution does not nicely fit into a common probability distribution, by design. It is a good case for using a nonparametric kernel density estimation method.

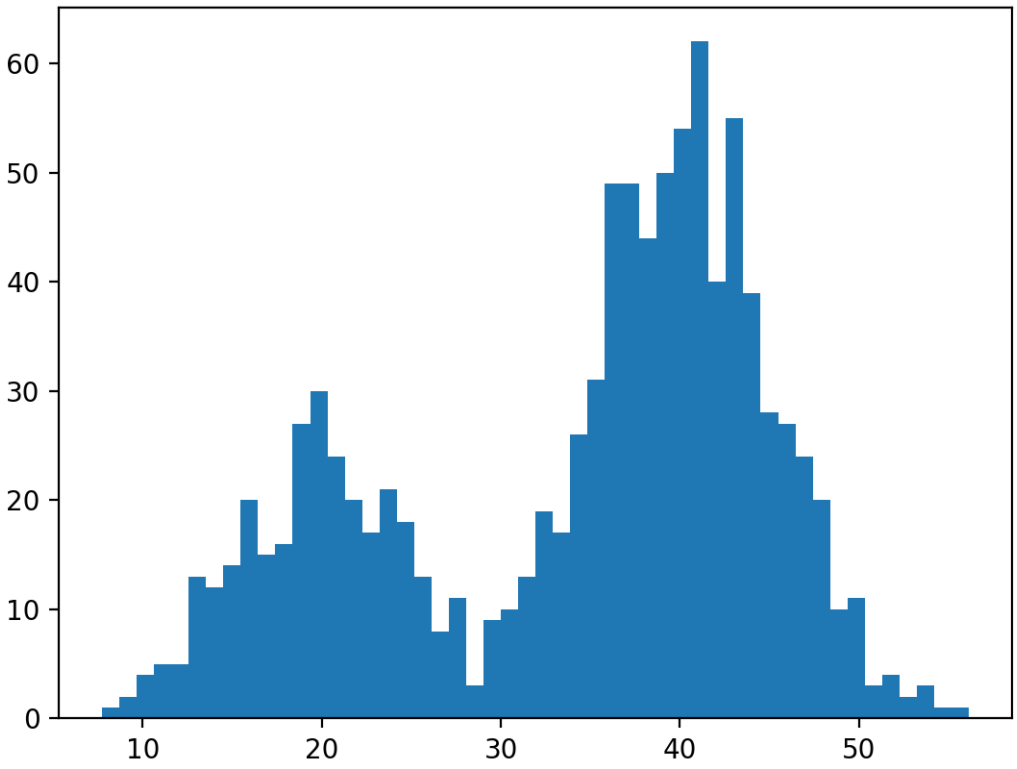


Figure 10.4: Histogram Plot of Data Sample With a Bimodal Probability Distribution.

The scikit-learn machine learning library provides the `KernelDensity` class that implements

kernel density estimation. First, the class is constructed with the desired bandwidth (window size) and kernel (basis function) arguments. It is a good idea to test different configurations on your data. In this case, we will try a bandwidth of 2 and a Gaussian kernel. The class is then fit on a data sample via the `fit()` function. The function expects the data to have a 2D shape with the form [rows, columns], therefore we can reshape our data sample to have 1,000 rows and 1 column.

```
...
# fit density
model = KernelDensity(bandwidth=2, kernel='gaussian')
sample = sample.reshape((len(sample), 1))
model.fit(sample)
```

Listing 10.11: Example defining a `KernelDensity` model on the data sample.

We can then evaluate how well the density estimate matches our data by calculating the probabilities for a range of observations and comparing the shape to the histogram, just like we did for the parametric case in the prior section. The `score_samples()` function on the `KernelDensity` will calculate the log probability for an array of samples. We can create a range of samples from 1 to 60, about the range of our domain, calculate the log probabilities, then invert the log operation by calculating the exponent or `exp()` to return the values to the range 0-1 for normal probabilities.

```
...
# sample probabilities for a range of outcomes
values = asarray([value for value in range(1, 60)])
values = values.reshape((len(values), 1))
probabilities = model.score_samples(values)
probabilities = exp(probabilities)
```

Listing 10.12: Example estimating probabilities for observations using the fit model.

Finally, we can create a histogram with normalized frequencies and an overlay line plot of values to estimated probabilities.

```
...
# plot the histogram and pdf
pyplot.hist(sample, bins=50, density=True)
pyplot.plot(values[:], probabilities)
pyplot.show()
```

Listing 10.13: Example of plotting the histogram and estimated probability density of the data sample.

Tying this together, the complete example of kernel density estimation for a bimodal data sample is listed below.

```
# example of kernel density estimation for a bimodal data sample
from matplotlib import pyplot
from numpy.random import normal
from numpy import hstack
from numpy import asarray
from numpy import exp
from sklearn.neighbors import KernelDensity
# generate a sample
sample1 = normal(loc=20, scale=5, size=300)
```

```
sample2 = normal(loc=40, scale=5, size=700)
sample = hstack((sample1, sample2))
# fit density
model = KernelDensity(bandwidth=2, kernel='gaussian')
sample = sample.reshape((len(sample), 1))
model.fit(sample)
# sample probabilities for a range of outcomes
values = asarray([value for value in range(1, 60)])
values = values.reshape((len(values), 1))
probabilities = model.score_samples(values)
probabilities = exp(probabilities)
# plot the histogram and pdf
pyplot.hist(sample, bins=50, density=True)
pyplot.plot(values[:], probabilities)
pyplot.show()
```

Listing 10.14: Example of kernel density estimation for a bimodal data sample.

Running the example creates the data distribution, fits the kernel density estimation model, then plots the histogram of the data sample and the PDF from the KDE model. Note that your results will differ given the random nature of the data sample. Try running the example a few times. In this case, we can see that the PDF is a good fit for the histogram. It is not very smooth and could be made more so by setting the `bandwidth` argument to 3 samples or higher. Experiment with different values of the bandwidth and the kernel function.

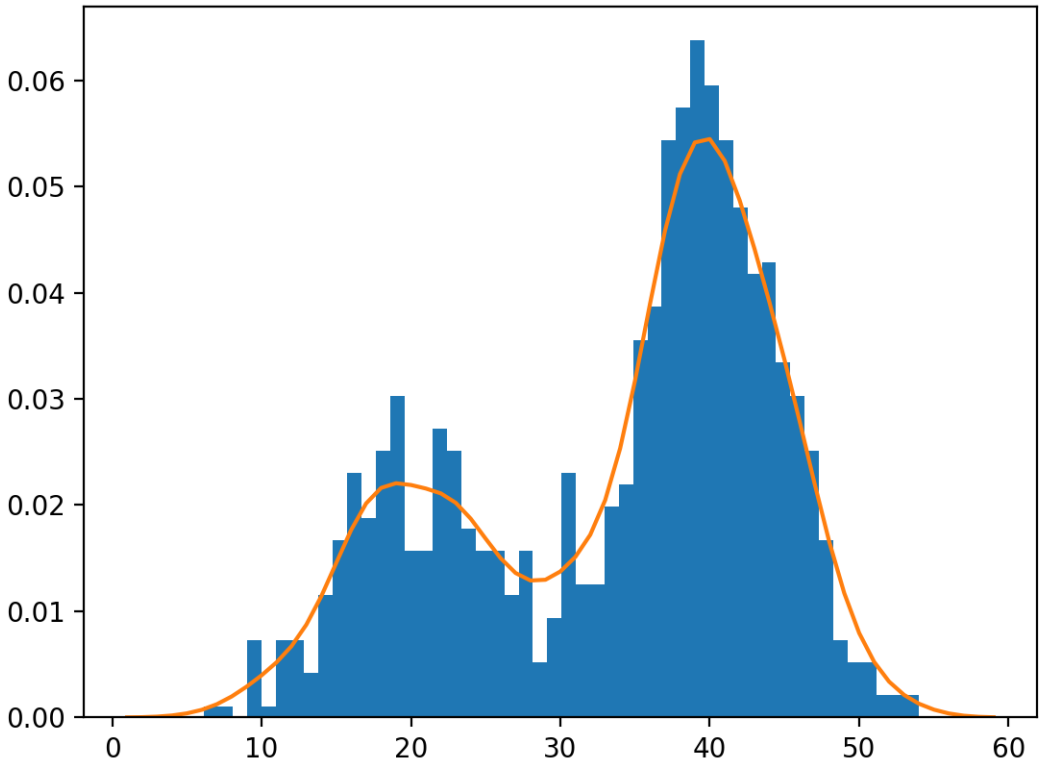


Figure 10.5: Histogram and Probability Density Function Plot Estimated via Kernel Density Estimation for a Bimodal Data Sample.

The `KernelDensity` class is powerful and does support estimating the PDF for multidimensional data.

## 10.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 10.6.1 Books

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2009.
  https://amzn.to/2LDuLLE

### 10.6.2 API

- `scipy.stats.gaussian_kde` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.gaussian_kde.html
- Nonparametric Methods nonparametric, Statsmodels API.
  https://www.statsmodels.org/stable/nonparametric.html
- Kernel Density Estimation Statsmodels Example.
  https://www.statsmodels.org/stable/examples/notebooks/generated/kernel_density.html
- Density Estimation, Scikit-Learn API.
  https://scikit-learn.org/stable/modules/density.html

### 10.6.3 Articles

- Density estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Density_estimation
- Histogram, Wikipedia.
  https://en.wikipedia.org/wiki/Histogram
- Kernel density estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Kernel_density_estimation
- Multivariate kernel density estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Multivariate_kernel_density_estimation

## 10.7 Summary

In this tutorial, you discovered a gentle introduction to probability density estimation. Specifically, you learned:

- Histogram plots provide a fast and reliable way to visualize the probability density of a data sample.
- Parametric probability density estimation involves selecting a common distribution and estimating the parameters for the density function from a data sample.
- Nonparametric probability density estimation involves using a technique to fit a model to the arbitrary distribution of the data, like kernel density estimation.

### 10.7.1 Next

This was the final tutorial in this Part. In the next Part, you will discover the maximum likelihood estimation probabilistic framework.

# Part V

# Maximum Likelihood

# Chapter 11

# Maximum Likelihood Estimation

Density estimation is the problem of estimating the probability distribution for a sample of observations from a problem domain. There are many techniques for solving density estimation, although a common framework used throughout the field of machine learning is maximum likelihood estimation. Maximum likelihood estimation involves defining a likelihood function for calculating the conditional probability of observing the data sample given a probability distribution and distribution parameters. This approach can be used to search a space of possible distributions and parameters. This flexible probabilistic framework also provides the foundation for many machine learning algorithms, including important methods such as linear regression and logistic regression for predicting numeric values and class labels respectively, but also more generally for deep learning artificial neural networks. In this tutorial, you will discover a gentle introduction to maximum likelihood estimation. After reading this tutorial, you will know:

- Maximum Likelihood Estimation is a probabilistic framework for solving the problem of density estimation.
- It involves maximizing a likelihood function in order to find the probability distribution and parameters that best explain the observed data.
- It provides a framework for predictive modeling in machine learning where finding model parameters can be framed as an optimization problem.

Let's get started.

## 11.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Problem of Probability Density Estimation
2. Maximum Likelihood Estimation
3. Relationship to Machine Learning

## 11.2 Problem of Probability Density Estimation

A common modeling problem involves how to estimate a joint probability distribution for a dataset. For example, given a sample of observation ($X$) from a domain ($x_1, x_2, x_3, \cdots, x_n$), where each observation is drawn independently from the domain with the same probability distribution (so-called independent and identically distributed, i.i.d., or close to it). Density estimation involves selecting a probability distribution function and the parameters of that distribution that best explain the joint probability distribution of the observed data ($X$).

- How do you choose the probability distribution function?
- How do you choose the parameters for the probability distribution function?

This problem is made more challenging if the sample ($X$) drawn from the population is small and has noise, meaning that any evaluation of an estimated probability density function and its parameters will have some error. There are many techniques for solving this problem, although two common approaches are:

- Maximum a Posteriori (MAP), a Bayesian method.
- Maximum Likelihood Estimation (MLE), frequentist method.

The main difference is that MLE assumes that all solutions are equally likely beforehand, whereas MAP allows prior information about the form of the solution to be harnessed. In this tutorial, we will take a closer look at the MLE method and its relationship to applied machine learning.

## 11.3 Maximum Likelihood Estimation

One solution to probability density estimation is referred to as Maximum Likelihood Estimation, or MLE for short. Maximum Likelihood Estimation involves treating the problem as an optimization or search problem, where we seek a set of parameters that results in the best fit for the joint probability of the data sample ($X$). First, it involves defining a parameter called theta ($\theta$) that defines both the choice of the probability density function and the parameters of that distribution. It may be a vector of numerical values whose values change smoothly and map to different probability distributions and their parameters. In Maximum Likelihood Estimation, we wish to maximize the probability of observing the data from the joint probability distribution given a specific probability distribution and its parameters, stated formally as:

$$P(X|\theta) \tag{11.1}$$

This conditional probability is often stated using the semicolon (;) notation instead of the bar notation (|) because $\theta$ is not a random variable, but instead an unknown parameter. For example:

$$P(X;\theta) \tag{11.2}$$

Or:

$$P(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{11.3}$$

This resulting conditional probability is referred to as the likelihood of observing the data given the model parameters and written using the notation $L()$ to denote the likelihood function. For example:

$$L(X; \theta) \tag{11.4}$$

The objective of Maximum Likelihood Estimation is to find the set of parameters ($\theta$) that maximize the likelihood function, e.g. result in the largest likelihood value.

$$\max L(X; \theta) \tag{11.5}$$

We can unpack the conditional probability calculated by the likelihood function. Given that the sample is comprised of $n$ examples, we can frame this as the joint probability of the observed data samples $x_1, x_2, x_3, \cdots, x_n$ in $X$ given the probability distribution parameters ($\theta$).

$$L(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{11.6}$$

The joint probability distribution can be restated as the multiplication of the conditional probability for observing each example given the distribution parameters.

$$\prod_{i=1}^{n} P(x_i; \theta) \tag{11.7}$$

Multiplying many small probabilities together can be numerically unstable in practice, therefore, it is common to restate this problem as the sum of the log conditional probabilities of observing each example given the model parameters.

$$\sum_{i=1}^{n} \log P(x_i; \theta) \tag{11.8}$$

Where log with base-e called the natural logarithm is commonly used.

> This product over many probabilities can be inconvenient [...] it is prone to numerical underflow. To obtain a more convenient but equivalent optimization problem, we observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product into a sum
>
> — Page 132, *Deep Learning*, 2016.

Given the frequent use of log in the likelihood function, it is commonly referred to as a log-likelihood function. It is common in optimization problems to prefer to minimize the cost function, rather than to maximize it. Therefore, the negative of the log-likelihood function is used, referred to generally as a Negative Log-Likelihood (NLL) function.

$$\min -\sum_{i=1}^{n} \log P(x_i; \theta) \tag{11.9}$$

> In software, we often phrase both as minimizing a cost function. Maximum likelihood thus becomes minimization of the negative log-likelihood (NLL) ...
>
> — Page 133, *Deep Learning*, 2016.

## 11.4 Relationship to Machine Learning

This problem of density estimation is directly related to applied machine learning. We can frame the problem of fitting a machine learning model as the problem of probability density estimation. Specifically, the choice of model and model parameters is referred to as a modeling hypothesis $h$, and the problem involves finding $h$ that best explains the data $X$.

$$P(X; h) \tag{11.10}$$

We can, therefore, find the modeling hypothesis that maximizes the likelihood function.

$$\max L(X; h) \tag{11.11}$$

Or, more fully:

$$\max \sum_{i=1}^{n} \log P(x_i; h) \tag{11.12}$$

This provides the basis for estimating the probability density of a dataset, typically used in unsupervised machine learning algorithms; for example:

- Clustering algorithms.

> Using the expected log joint probability as a key quantity for learning in a probability model with hidden variables is better known in the context of the celebrated *expectation maximization* or EM algorithm.

— Page 365, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.

The Maximum Likelihood Estimation framework is also a useful tool for supervised machine learning. This applies to data where we have input and output variables, where the output variable may be a numerical value or a class label in the case of regression and classification predictive modeling retrospectively. We can state this as the conditional probability of the output $y$ given the input ($X$) given the modeling hypothesis ($h$).

$$\max L(y|X; h) \tag{11.13}$$

Or, more fully:

$$\max \sum_{i=1}^{n} \log P(y_i|x_i; h) \tag{11.14}$$

> The maximum likelihood estimator can readily be generalized to the case where our goal is to estimate a conditional probability $P(y|x;\theta)$ in order to predict $y$ given $x$. This is actually the most common situation because it forms the basis for most supervised learning.
>
> — Page 133, *Deep Learning*, 2016.

This means that the same Maximum Likelihood Estimation framework that is generally used for density estimation can be used to find a supervised learning model and parameters. This provides the basis for foundational linear modeling techniques, such as:

- Linear Regression, for predicting a numerical value.
- Logistic Regression, for binary classification.

In the case of linear regression, the model is constrained to a line and involves finding a set of coefficients for the line that best fits the observed data. Fortunately, this problem can be solved analytically (e.g. directly using linear algebra). In the case of logistic regression, the model defines a line and involves finding a set of coefficients for the line that best separates the classes. This cannot be solved analytically and is often solved by searching the space of possible coefficient values using an efficient optimization algorithm such as the BFGS algorithm or variants.

Both methods can also be solved less efficiently using a more general optimization algorithm such as stochastic gradient descent. In fact, most machine learning models can be framed under the maximum likelihood estimation framework, providing a useful and consistent way to approach predictive modeling as an optimization problem. An important benefit of the maximum likelihood estimator in machine learning is that as the size of the dataset increases, the quality of the estimator continues to improve.

## 11.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 11.5.1 Books

- Chapter 5 Machine Learning Basics, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Chapter 2 Probability Distributions, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Chapter 8 Model Inference and Averaging, *The Elements of Statistical Learning*, 2016.
  https://amzn.to/2YVqu8s
- Chapter 9 Probabilistic methods, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2lnW5S7

- Chapter 22 Maximum Likelihood and Clustering, *Information Theory, Inference and Learning Algorithms*, 2003.
https://amzn.to/31q6fBo

- Chapter 8 Learning distributions, *Bayesian Reasoning and Machine Learning*, 2011.
https://amzn.to/31D2VTD

### 11.5.2 Articles

- Maximum likelihood estimation, Wikipedia.
https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

- Maximum Likelihood, Wolfram MathWorld.
http://mathworld.wolfram.com/MaximumLikelihood.html

- Likelihood function, Wikipedia.
https://en.wikipedia.org/wiki/Likelihood_function

## 11.6 Summary

In this tutorial, you discovered a gentle introduction to maximum likelihood estimation. Specifically, you learned:

- Maximum Likelihood Estimation is a probabilistic framework for solving the problem of density estimation.

- It involves maximizing a likelihood function in order to find the probability distribution and parameters that best explain the observed data.

- It provides a framework for predictive modeling in machine learning where finding model parameters can be framed as an optimization problem.

### 11.6.1 Next

In the next tutorial, you will discover how linear regression can be solved using maximum likelihood estimation.

# Chapter 12

# Linear Regression With Maximum Likelihood Estimation

Linear regression is a classical model for predicting a numerical quantity. The parameters of a linear regression model can be estimated using a least squares procedure or by a maximum likelihood estimation procedure. Maximum likelihood estimation is a probabilistic framework for automatically finding the probability distribution and parameters that best describe the observed data. Supervised learning can be framed as a conditional probability problem, and maximum likelihood estimation can be used to fit the parameters of a model that best summarizes the conditional probability distribution, so-called conditional maximum likelihood estimation. A linear regression model can be fit under this framework and can be shown to derive an identical solution to a least squares approach. In this tutorial, you will discover linear regression with maximum likelihood estimation. After reading this tutorial, you will know:

- Linear regression is a model for predicting a numerical quantity and maximum likelihood estimation is a probabilistic framework for estimating model parameters.
- Coefficients of a linear regression model can be estimated using a negative log-likelihood function from maximum likelihood estimation.
- The negative log-likelihood function can be used to derive the least squares solution to linear regression.

Let's get started.

## 12.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Linear Regression
2. Maximum Likelihood Estimation
3. Linear Regression as Maximum Likelihood
4. Least Squares and Maximum Likelihood

## 12.2 Linear Regression

Linear regression is a standard modeling method from statistics and machine learning.

> Linear regression is the "work horse" of statistics and (supervised) machine learning.
>
> — Page 217, *Machine Learning: A Probabilistic Perspective*, 2012.

Generally, it is a model that maps one or more numerical inputs to a numerical output. In terms of predictive modeling, it is suited to regression type problems: that is, the prediction of a real-valued quantity. The input data is denoted as $X$ with $n$ examples and the output is denoted $y$ with one output for each input. The prediction of the model for a given input is denoted as $yhat$.

$$yhat = model(X) \tag{12.1}$$

The model is defined in terms of parameters called coefficients (beta or $\beta$), where there is one coefficient per input and an additional coefficient that provides the intercept or bias. For example, a problem with inputs $X$ with $m$ variables $x_1, x_2, \cdots, x_m$ will have coefficients $\beta_1, \beta_2, \cdots, \beta_m$ and $\beta_0$. A given input is predicted as the weighted sum of the inputs for the example and the coefficients.

$$yhat = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \cdots + \beta_m \times x_m \tag{12.2}$$

The model can also be described using linear algebra, with a vector for the coefficients ($\beta$) and a matrix for the input data ($X$) and a vector for the output ($y$).

$$y = X \times \beta \tag{12.3}$$

The examples are drawn from a broader population and as such, the sample is known to be incomplete. Additionally, there is expected to be measurement error or statistical noise in the observations. The parameters of the model ($\beta$) must be estimated from the sample of observations drawn from the domain. There are many ways to estimate the parameters given the study of the model for more than 100 years; nevertheless, there are two frameworks that are the most common. They are:

- Least Squares Optimization.
- Maximum Likelihood Estimation.

Both are optimization procedures that involve searching for different model parameters. Least squares optimization is an approach to estimating the parameters of a model by seeking a set of parameters that results in the smallest squared error between the predictions of the model ($yhat$) and the actual outputs ($y$), averaged over all examples in the dataset, so-called mean squared error. Maximum Likelihood Estimation is a frequentist probabilistic framework that seeks a set of parameters for the model that maximize a likelihood function. We will take a closer look at this second approach.

Under both frameworks, different optimization algorithms may be used, such as local search methods like the BFGS algorithm (or variants), and general optimization methods like stochastic gradient descent. The linear regression model is special in that an analytical solution also exists, meaning that the coefficients can be calculated directly using linear algebra, a topic that is out of the scope of this tutorial.

## 12.3 Maximum Likelihood Estimation

Maximum Likelihood Estimation, or MLE for short, is a probabilistic framework for estimating the parameters of a model (introduced in Chapter 11). In Maximum Likelihood Estimation, we wish to maximize the conditional probability of observing the data ($X$) given a specific probability distribution and its parameters ($\theta$), stated formally as:

$$P(X; \theta) \tag{12.4}$$

Where $X$ is, in fact, the joint probability distribution of all observations from the problem domain from 1 to $n$.

$$P(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{12.5}$$

This resulting conditional probability is referred to as the likelihood of observing the data given the model parameters and written using the notation $L()$ to denote the likelihood function. For example:

$$L(X; \theta) \tag{12.6}$$

The joint probability distribution can be restated as the multiplication of the conditional probability for observing each example given the distribution parameters. Multiplying many small probabilities together can be unstable; as such, it is common to restate this problem as the sum of the natural log conditional probability.

$$\sum_{i=1}^{n} \log P(x_i; \theta) \tag{12.7}$$

Given the common use of log in the likelihood function, it is referred to as a log-likelihood function. It is also common in optimization problems to prefer to minimize the cost function rather than to maximize it. Therefore, the negative of the log-likelihood function is used, referred to generally as a Negative Log-Likelihood (NLL) function.

$$\min - \sum_{i=1}^{n} \log P(x_i; \theta) \tag{12.8}$$

The Maximum Likelihood Estimation framework can be used as a basis for estimating the parameters of many different machine learning models for regression and classification predictive modeling. This includes the linear regression model.

## 12.4 Linear Regression as Maximum Likelihood

We can frame the problem of fitting a machine learning model as the problem of probability density estimation. Specifically, the choice of model and model parameters is referred to as a modeling hypothesis $h$, and the problem involves finding $h$ that best explains the data $X$. We can, therefore, find the modeling hypothesis that maximizes the likelihood function.

$$\max \sum_{i=1}^{n} \log P(x_i; h) \tag{12.9}$$

Supervised learning can be framed as a conditional probability problem of predicting the probability of the output given the input:

$$P(y|X) \tag{12.10}$$

As such, we can define conditional maximum likelihood estimation for supervised machine learning as follows:

$$\max \sum_{i=1}^{n} \log P(y_i|x_i; h) \tag{12.11}$$

Now we can replace $h$ with our linear regression model. We can make some reasonable assumptions, such as the observations in the dataset are independent and drawn from the same probability distribution (i.i.d.), and that the target variable ($y$) has statistical noise with a Gaussian distribution, zero mean, and the same variance for all examples. With these assumptions, we can frame the problem of estimating $y$ given $X$ as estimating the mean value for $y$ from a Gaussian probability distribution given $X$. The analytical form of the Gaussian function is as follows:

$$f(x) = \frac{1}{\sqrt{2 \times \pi \times \sigma^2}} \times exp(-\frac{1}{2 \times \sigma^2} \times (y - \mu)^2) \tag{12.12}$$

Where $\mu$ is the mean of the distribution and $\sigma^2$ is the variance where the units are squared. We can use this function as our likelihood function, where $\mu$ is defined as the prediction from the model with a given set of coefficients and sigma is a fixed constant. We can replace $\mu$ with our model stated as $h(x_i, \beta)$. First, we can state the problem as the maximization of the product of the probabilities for each example in the dataset:

$$\max \prod_{i=1}^{n} \frac{1}{\sqrt{2 \times \pi \times \sigma^2}} \times exp(-\frac{1}{2 \times \sigma^2} \times (y_i - h(x_i, \beta))^2) \tag{12.13}$$

Where $x_i$ is a given example and $\beta$ refers to the coefficients of the linear regression model. We can transform this to a log-likelihood model as follows:

$$\max \sum_{i=1}^{n} \log \frac{1}{\sqrt{2 \times \pi \times \sigma^2}} - \frac{1}{2 \times \sigma^2} \times (y_i - h(x_i, \beta))^2 \tag{12.14}$$

The calculation can be simplified further, but we will stop there for now. It's interesting that the prediction is the mean of a distribution. It suggests that we can very reasonably add a bound to the prediction to give a prediction interval based on the standard deviation of the distribution, which is indeed a common practice. Although the model assumes a Gaussian distribution in the prediction (i.e. Gaussian noise function or error function), there is no such expectation for the inputs to the model ($X$).

> [the model] considers noise only in the target value of the training example and does not consider noise in the attributes describing the instances themselves.
>
> — Page 167, *Machine Learning*, 1997.

We can apply a search procedure to maximize this log likelihood function, or invert it by adding a negative sign to the beginning and minimize the negative log-likelihood function (more common). This provides a solution to the linear regression model for a given dataset. This framework is also more general and can be used for curve fitting and provides the basis for fitting other regression models, such as artificial neural networks.

## 12.5 Least Squares and Maximum Likelihood

Interestingly, the maximum likelihood solution to linear regression presented in the previous section can be shown to be identical to the least squares solution. After derivation, the least squares equation to be minimized to fit a linear regression to a dataset looks as follows:

$$\min \sum_{i=1}^{n} (y_i - h(x_i, \beta))^2 \tag{12.15}$$

Where we are summing the squared errors between each target variable ($y_i$) and the prediction from the model for the associated input $h(x_i, \beta)$. This is often referred to as ordinary least squares. More generally, if the value is normalized by the number of examples in the dataset (averaged) rather than summed, then the quantity is referred to as the mean squared error.

$$mse = \frac{1}{n} \times \sum_{i=1}^{n} (y_i - yhat)^2 \tag{12.16}$$

Starting with the likelihood function defined in the previous section, we can show how we can remove constant elements to give the same equation as the least squares approach to solving linear regression[1].

$$\max \sum_{i=1}^{n} \log \frac{1}{\sqrt{2 \times \pi \times \sigma^2}} - \frac{1}{2 \times \sigma^2} \times (y_i - h(x_i, \beta))^2 \tag{12.17}$$

Key to removing constants is to focus on what does not change when different models are evaluated, e.g. when $h(x_i, \beta)$ is evaluated. The first term of the calculation is independent of the model and can be removed to give:

$$\max \sum_{i=1}^{n} -\frac{1}{2 \times \sigma^2} \times (y_i - h(x_i, \beta))^2 \tag{12.18}$$

We can then remove the negative sign to minimize the positive quantity rather than maximize the negative quantity:

$$\min \sum_{i=1}^{n} \frac{1}{2 \times \sigma^2} \times (y_i - h(x_i, \beta))^2 \tag{12.19}$$

[1]This derivation is based on the example given in Chapter 6 of Machine Learning by Tom Mitchell.

Finally, we can discard the remaining first term that is also independent of the model to give:

$$\min \sum_{i=1}^{n} (y_i - h(x_i, \beta))^2 \tag{12.20}$$

We can see that this is identical to the least squares solution. In fact, under reasonable assumptions, an algorithm that minimizes the squared error between the target variable and the model output also performs maximum likelihood estimation.

> ... under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis pre- dictions and the training data will output a maximum likelihood hypothesis.
>
> — Page 164, *Machine Learning*, 1997.

## 12.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 12.6.1 Books

- Section 15.1 Least Squares as a Maximum Likelihood Estimator, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd Edition, 1992.
  https://amzn.to/2YX0Obn
- Chapter 5 Machine Learning Basics, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Section 2.6.3 Function Approximation, *The Elements of Statistical Learning*, 2016.
  https://amzn.to/2YVqu8s
- Section 6.4 Maximum Likelihood and Least-Squares Error Hypotheses, *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- Section 3.1.1 Maximum likelihood and least squares, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Section 7.3 Maximum likelihood estimation (least squares), *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 12.6.2 Articles

- Maximum likelihood estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_likelihood_estimation
- Likelihood function, Wikipedia.
  https://en.wikipedia.org/wiki/Likelihood_function
- Linear regression, Wikipedia.
  https://en.wikipedia.org/wiki/Linear_regression

## 12.7 Summary

In this tutorial, you discovered linear regression with maximum likelihood estimation. Specifically, you learned:

- Linear regression is a model for predicting a numerical quantity and maximum likelihood estimation is a probabilistic framework for estimating model parameters.
- Coefficients of a linear regression model can be estimated using a negative log-likelihood function from maximum likelihood estimation.
- The negative log-likelihood function can be used to derive the least squares solution to linear regression.

### 12.7.1 Next

In the next tutorial, you will discover how logistic regression can be fit using maximum likelihood estimation.

# Chapter 13

# Logistic Regression With Maximum Likelihood Estimation

Logistic regression is a model for binary classification predictive modeling. The parameters of a logistic regression model can be estimated by the probabilistic framework called maximum likelihood estimation. Under this framework, a probability distribution for the target variable (class label) must be assumed and then a likelihood function defined that calculates the probability of observing the outcome given the input data and the model. This function can then be optimized to find the set of parameters that results in the largest sum likelihood over the training dataset. The maximum likelihood approach to fitting a logistic regression model both aids in better understanding the form of the logistic regression model and provides a template that can be used for fitting classification models more generally. This is particularly true as the negative of the log-likelihood function used in the procedure can be shown to be equivalent to the cross-entropy loss function. In this tutorial, you will discover logistic regression with maximum likelihood estimation. After reading this tutorial, you will know:

- Logistic regression is a linear model for binary classification predictive modeling.
- The linear part of the model predicts the log-odds of an example belonging to class 1, which is converted to a probability via the logistic function.
- The parameters of the model can be estimated by maximizing a likelihood function that predicts the mean of a Bernoulli distribution for each example.

Let's get started.

## 13.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Logistic Regression
2. Logistic Regression and Log-Odds
3. Maximum Likelihood Estimation
4. Logistic Regression as Maximum Likelihood

## 13.2 Logistic Regression

Logistic regression is a classical linear method for binary classification. Classification predictive modeling problems are those that require the prediction of a class label (e.g. *red*, *green*, *blue*) for a given set of input variables. Binary classification refers to those classification problems that have two class labels, e.g. true/false or 0/1. Logistic regression has a lot in common with linear regression, although linear regression is a technique for predicting a numerical value, not for classification problems. Both techniques model the target variable with a line (or hyperplane, depending on the number of dimensions of input. Linear regression fits the line to the data, which can be used to predict a new quantity, whereas logistic regression fits a line to best separate the two classes. The input data is denoted as $X$ with $n$ examples and the output is denoted $y$ with one output for each input. The prediction of the model for a given input is denoted as $yhat$.

$$yhat = model(X) \tag{13.1}$$

The model is defined in terms of parameters called coefficients ($\beta$), where there is one coefficient per input and an additional coefficient that provides the intercept or bias. For example, a problem with inputs $X$ with $m$ variables $x_1, x_2, \cdots, x_m$ will have coefficients $\beta_1, \beta_2, \cdots, \beta_m$, and $\beta_0$. A given input is predicted as the weighted sum of the inputs for the example and the coefficients.

$$yhat = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \cdots + \beta_m \times x_m \tag{13.2}$$

The model can also be described using linear algebra, with a vector for the coefficients ($\beta$) and a matrix for the input data ($X$) and a vector for the output ($y$).

$$y = X \times Beta \tag{13.3}$$

So far, this is identical to linear regression and is insufficient as the output will be a real value instead of a class label. Instead, the model squashes the output of this weighted sum using a nonlinear function to ensure the outputs are a value between 0 and 1. The logistic function (also called the sigmoid) is used, which is defined as:

$$f(x) = \frac{1}{1 + exp(-x)} \tag{13.4}$$

Where $x$ is the input value to the function. In the case of logistic regression, $x$ is replaced with the weighted sum. For example:

$$yhat = \frac{1}{1 + exp(-(X \times \beta))} \tag{13.5}$$

The output is interpreted as a probability from a Binomial probability distribution function for the class labeled 1, if the two classes in the problem are labeled 0 and 1.

> Notice that the output, being a number between 0 and 1, can be interpreted as a probability of belonging to the class labeled 1.
>
> — Page 726, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

The examples in the training dataset are drawn from a broader population and as such, this sample is known to be incomplete. Additionally, there is expected to be measurement error or statistical noise in the observations. The parameters of the model ($\beta$) must be estimated from the sample of observations drawn from the domain. There are many ways to estimate the parameters. There are two frameworks that are the most common; they are:

- Least Squares Optimization (iteratively reweighted least squares).
- Maximum Likelihood Estimation.

Both are optimization procedures that involve searching for different model parameters. Maximum Likelihood Estimation is a frequentist probabilistic framework that seeks a set of parameters for the model that maximizes a likelihood function. We will take a closer look at this second approach in the subsequent sections.

## 13.3 Logistic Regression and Log-Odds

Before we dive into how the parameters of the model are estimated from data, we need to understand what logistic regression is calculating exactly. This might be the most confusing part of logistic regression, so we will go over it slowly. The linear part of the model (the weighted sum of the inputs) calculates the log-odds of a successful event, specifically, the log-odds that a sample belongs to class 1.

$$\text{log-odds} = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \cdots + \beta_m \times x_m \tag{13.6}$$

In effect, the model estimates the log-odds for class 1 for the input variables at each level (all observed values). What are odds and log-odds? Odds may be familiar from the field of gambling. Odds are often stated as wins to losses (wins : losses), e.g. a one to ten chance or ratio of winning is stated as 1:10. Given the probability of success ($p$) predicted by the logistic regression model, we can convert it to odds of success as the probability of success divided by the probability of not success:

$$\text{odds of success} = \frac{p}{1-p} \tag{13.7}$$

The logarithm of the odds is calculated, specifically log base-e or the natural logarithm. This quantity is referred to as the log-odds and may be referred to as the logit (logistic unit), a unit of measure.

$$\text{log-odds} = log(\frac{p}{1-p}) \tag{13.8}$$

Recall that this is what the linear part of logistic regression is calculating:

$$\text{log-odds} = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \cdots + \beta_m \times x_m \tag{13.9}$$

The log-odds of success can be converted back into an odds of success by calculating the exponential of the log-odds.

$$\text{odds} = exp(log - odds) \tag{13.10}$$

Or

$$\text{odds} = exp(\beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \cdots + \beta_m \times x_m) \tag{13.11}$$

The odds of success can be converted back into a probability of success as follows:

$$p = \frac{odds}{odds + 1} \tag{13.12}$$

And this is close to the form of our logistic regression model, except we want to convert log-odds to odds as part of the calculation. We can do this and simplify the calculation as follows:

$$p = \frac{1}{1 + exp(-\text{log-odds})} \tag{13.13}$$

This shows how we go from log-odds to odds, to a probability of class 1 with the logistic regression model, and that this final functional form matches the logistic function, ensuring that the probability is between 0 and 1. We can make these calculations of converting between probability, odds and log-odds concrete with some small examples in Python. First, let's define the probability of success at 80%, or 0.8, and convert it to odds then back to a probability again. The complete example is listed below.

```
# example of converting between probability and odds
# define our probability of success
prob = 0.8
print('Probability %.1f' % prob)
# convert probability to odds
odds = prob / (1 - prob)
print('Odds %.1f' % odds)
# convert back to probability
prob = odds / (odds + 1)
print('Probability %.1f' % prob)
```

Listing 13.1: Example of converting between probability and odds.

Running the example shows that 0.8 is converted to the odds of success 4, and back to the correct probability again.

```
Probability 0.8
Odds 4.0
Probability 0.8
```

Listing 13.2: Example output from converting between probability and odds.

Let's extend this example and convert the odds to log-odds and then convert the log-odds back into the original probability. This final conversion is effectively the form of the logistic regression model, or the logistic function. The complete example is listed below.

```
# example of converting between probability and log-odds
from math import log
from math import exp
# define our probability of success
prob = 0.8
print('Probability %.1f' % prob)
# convert probability to odds
```

```
odds = prob / (1 - prob)
print('Odds %.1f' % odds)
# convert odds to log-odds
logodds = log(odds)
print('Log-Odds %.1f' % logodds)
# convert log-odds to a probability
prob = 1 / (1 + exp(-logodds))
print('Probability %.1f' % prob)
```

Listing 13.3: Example of converting between probability and log odds.

Running the example, we can see that our odds are converted into the log odds of about 1.4 and then correctly converted back into the 0.8 probability of success.

```
Probability 0.8
Odds 4.0
Log-Odds 1.4
Probability 0.8
```

Listing 13.4: Example output from converting between probability and log odds.

Now that we have a handle on the probability calculated by logistic regression, let's look at maximum likelihood estimation.

## 13.4 Maximum Likelihood Estimation

Maximum Likelihood Estimation, or MLE for short, is a probabilistic framework for estimating the parameters of a model (introduced in Chapter 11). In Maximum Likelihood Estimation, we wish to maximize the conditional probability of observing the data ($X$) given a specific probability distribution and its parameters ($\theta$), stated formally as:

$$P(X; \theta) \tag{13.14}$$

Where $X$ is, in fact, the joint probability distribution of all observations from the problem domain from 1 to $n$.

$$P(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{13.15}$$

This resulting conditional probability is referred to as the likelihood of observing the data given the model parameters and written using the notation $L()$ to denote the likelihood function. For example:

$$L(X; \theta) \tag{13.16}$$

The joint probability distribution can be restated as the multiplication of the conditional probability for observing each example given the distribution parameters. Multiplying many small probabilities together can be unstable; as such, it is common to restate this problem as the sum of the log conditional probability.

$$\sum_{i=1}^{n} \log P(x_i; \theta) \tag{13.17}$$

Given the frequent use of log in the likelihood function, it is referred to as a log-likelihood function. It is common in optimization problems to prefer to minimize the cost function rather than to maximize it. Therefore, the negative of the log-likelihood function is used, referred to generally as a Negative Log-Likelihood (NLL) function.

$$\min -\sum_{i=1}^{n} \log P(x_i; \theta) \tag{13.18}$$

The Maximum Likelihood Estimation framework can be used as a basis for estimating the parameters of many different machine learning models for regression and classification predictive modeling. This includes the logistic regression model.

## 13.5 Logistic Regression as Maximum Likelihood

We can frame the problem of fitting a machine learning model as the problem of probability density estimation. Specifically, the choice of model and model parameters is referred to as a modeling hypothesis $h$, and the problem involves finding $h$ that best explains the data $X$. We can, therefore, find the modeling hypothesis that maximizes the likelihood function.

$$\max \sum_{i=1}^{n} \log P(x_i; h) \tag{13.19}$$

Supervised learning can be framed as a conditional probability problem of predicting the probability of the output given the input:

$$P(y|X) \tag{13.20}$$

As such, we can define conditional maximum likelihood estimation for supervised machine learning as follows:

$$\max \sum_{i=1}^{n} \log P(y_i|x_i; h) \tag{13.21}$$

Now we can replace $h$ with our logistic regression model. In order to use maximum likelihood, we need to assume a probability distribution. In the case of logistic regression, a Binomial probability distribution is assumed for the data sample, where each example is one outcome of a Bernoulli trial. The Bernoulli distribution has a single parameter: the probability of a successful outcome ($p$).

$$\begin{aligned} P(y = 1) &= p \\ P(y = 0) &= 1 - p \end{aligned} \tag{13.22}$$

> The probability distribution that is most often used when there are two classes is the binomial distribution. This distribution has a single parameter, $p$, that is the probability of an event or a specific class.
>
> — Page 283, *Applied Predictive Modeling*, 2013.

The expected value (mean) of the Bernoulli distribution can be calculated as follows:

$$mean = P(y = 1) \times 1 + P(y = 0) \times 0 \tag{13.23}$$

Or, given p:

$$mean = p \times 1 + (1 - p) \times 0 \tag{13.24}$$

This calculation may seem redundant, but it provides the basis for the likelihood function for a specific input, where the probability is given by the model ($yhat$) and the actual label is given from the dataset.

$$likelihood = yhat \times y + (1 - yhat) \times (1 - y) \tag{13.25}$$

This function will always return a large probability when the model is close to the matching class value, and a small value when it is far away, for both $y = 0$ and $y = 1$ cases. We can demonstrate this with a small worked example for both outcomes and small and large probabilities predicted for each. The complete example is listed below.

```
# test of Bernoulli likelihood function

# likelihood function for Bernoulli distribution
def likelihood(y, yhat):
  return yhat * y + (1 - yhat) * (1 - y)

# test for y=1
y, yhat = 1, 0.9
print('y=%.1f, yhat=%.1f, likelihood: %.3f' % (y, yhat, likelihood(y, yhat)))
y, yhat = 1, 0.1
print('y=%.1f, yhat=%.1f, likelihood: %.3f' % (y, yhat, likelihood(y, yhat)))
# test for y=0
y, yhat = 0, 0.1
print('y=%.1f, yhat=%.1f, likelihood: %.3f' % (y, yhat, likelihood(y, yhat)))
y, yhat = 0, 0.9
print('y=%.1f, yhat=%.1f, likelihood: %.3f' % (y, yhat, likelihood(y, yhat)))
```

Listing 13.5: Example of logistic regression likelihood function.

Running the example prints the class labels ($y$) and predicted probabilities ($yhat$) for cases with close and far probabilities for each case. We can see that the likelihood function is consistent in returning a probability for how well the model achieves the desired outcome.

```
y=1.0, yhat=0.9, likelihood: 0.900
y=1.0, yhat=0.1, likelihood: 0.100
y=0.0, yhat=0.1, likelihood: 0.900
y=0.0, yhat=0.9, likelihood: 0.100
```

Listing 13.6: Example output from the logistic regression likelihood function.

We can update the likelihood function using the log to transform it into a log-likelihood function:

$$\text{log-likelihood} = \log(yhat) \times y + \log(1 - yhat) \times (1 - y) \tag{13.26}$$

Finally, we can sum the likelihood function across all examples in the dataset to maximize the likelihood:

$$\max \sum_{i=1}^{n} \log(yhat_i) \times y_i + \log(1 - yhat_i) \times (1 - y_i) \tag{13.27}$$

It is common practice to minimize a cost function for optimization problems; therefore, we can invert the function so that we minimize the negative log-likelihood:

$$\min \sum_{i=1}^{n} -(\log(yhat_i) \times y_i + \log(1 - yhat_i) \times (1 - y_i)) \tag{13.28}$$

Calculating the negative of the log-likelihood function for the Bernoulli distribution is equivalent to calculating the cross-entropy function for the Bernoulli distribution, where $p()$ represents the probability of class 0 or class 1, and $q()$ represents the estimation of the probability distribution, in this case by our logistic regression model (this is described further in Chapter 23).

$$\text{cross-entropy} = -(\log(q(class0)) \times p(class0) + log(q(class1)) \times p(class1)) \tag{13.29}$$

Unlike linear regression, there is not an analytical solution to solving this optimization problem. As such, an iterative optimization algorithm must be used.

> Unlike linear regression, we can no longer write down the MLE in closed form. Instead, we need to use an optimization algorithm to compute it. For this, we need to derive the gradient and Hessian.
>
> — Page 246, *Machine Learning: A Probabilistic Perspective*, 2012.

The function does provide some information to aid in the optimization (specifically a Hessian matrix can be calculated), meaning that efficient search procedures that exploit this information can be used, such as the BFGS algorithm (and variants).

## 13.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 13.6.1 Books

- Section 4.4.1 Fitting Logistic Regression Models, *The Elements of Statistical Learning*, 2016.
  https://amzn.to/2YVqu8s
- Section 4.3.2 Logistic regression, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Chapter 8 Logistic regression, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

- Chapter 4 Algorithms: the basic methods, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
https://amzn.to/2lnW5S7

- Section 18.6.4 Linear classification with logistic regression, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.
https://amzn.to/2Y7yCpO

- Section 12.2 Logistic Regression, *Applied Predictive Modeling*, 2013.
https://amzn.to/2yXgeBT

- Section 4.3 Logistic Regression, *An Introduction to Statistical Learning with Applications in R*, 2017.
https://amzn.to/31DTbbO

### 13.6.2 Articles

- Maximum likelihood estimation, Wikipedia.
https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

- Likelihood function, Wikipedia.
https://en.wikipedia.org/wiki/Likelihood_function

- Logistic regression, Wikipedia.
https://en.wikipedia.org/wiki/Logistic_regression

- Logistic function, Wikipedia.
https://en.wikipedia.org/wiki/Logistic_function

- Odds, Wikipedia.
https://en.wikipedia.org/wiki/Odds

## 13.7 Summary

In this tutorial, you discovered logistic regression with maximum likelihood estimation. Specifically, you learned:

- Logistic regression is a linear model for binary classification predictive modeling.

- The linear part of the model predicts the log-odds of an example belonging to class 1, which is converted to a probability via the logistic function.

- The parameters of the model can be estimated by maximizing a likelihood function that predicts the mean of a Bernoulli distribution for each example.

### 13.7.1 Next

In the next tutorial, you will discover the expectation maximization algorithm under the maximum likelihood estimation framework.

# Chapter 14

# Expectation Maximization (EM Algorithm)

Maximum likelihood estimation is an approach to density estimation for a dataset by searching across probability distributions and their parameters. It is a general and effective approach that underlies many machine learning algorithms, although it requires that the training dataset is complete, e.g. all relevant interacting random variables are present. Maximum likelihood becomes intractable if there are variables that interact with those in the dataset but were hidden or not observed, so-called latent variables. The expectation-maximization algorithm is an approach for performing maximum likelihood estimation in the presence of latent variables. It does this by first estimating the values for the latent variables, then optimizing the model, then repeating these two steps until convergence. It is an effective and general approach and is most commonly used for density estimation with missing data, such as clustering algorithms like the Gaussian Mixture Model. In this tutorial, you will discover the expectation-maximization algorithm. After reading this tutorial, you will know:

- Maximum likelihood estimation is challenging on data in the presence of latent variables.
- Expectation maximization provides an iterative solution to maximum likelihood estimation with latent variables.
- Gaussian mixture models are an approach to density estimation where the parameters of the distributions are fit using the expectation-maximization algorithm.

Let's get started.

## 14.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Problem of Latent Variables for Maximum Likelihood
2. Expectation-Maximization Algorithm
3. Gaussian Mixture Model and the EM Algorithm
4. Example of Gaussian Mixture Model

## 14.2 Problem of Latent Variables for Maximum Likelihood

A common modeling problem involves how to estimate a joint probability distribution for a dataset. Density estimation involves selecting a probability distribution function and the parameters of that distribution that best explain the joint probability distribution of the observed data. There are many techniques for solving this problem, although a common approach is called maximum likelihood estimation, or simply *maximum likelihood*. Maximum Likelihood Estimation involves treating the problem as an optimization or search problem, where we seek a set of parameters that results in the best fit for the joint probability of the data sample (introduced in Chapter 11).

A limitation of maximum likelihood estimation is that it assumes that the dataset is complete, or fully observed. This does not mean that the model has access to all data; instead, it assumes that all variables that are relevant to the problem are present. This is not always the case. There may be datasets where only some of the relevant variables can be observed, and some cannot, and although they influence other random variables in the dataset, they remain hidden. More generally, these unobserved or hidden variables are referred to as latent variables.

> Many real-world problems have hidden variables (sometimes called latent variables), which are not observable in the data that are available for learning.
>
> — Page 816, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

Conventional maximum likelihood estimation does not work well in the presence of latent variables.

> ... if we have missing data and/or latent variables, then computing the [maximum likelihood] estimate becomes hard.
>
> — Page 349, *Machine Learning: A Probabilistic Perspective*, 2012.

Instead, an alternate formulation of maximum likelihood is required for searching for the appropriate model parameters in the presence of latent variables. The Expectation-Maximization algorithm is one such approach.

## 14.3 Expectation-Maximization Algorithm

The Expectation-Maximization Algorithm, or EM algorithm for short, is an approach for maximum likelihood estimation in the presence of latent variables.

> A general technique for finding maximum likelihood estimators in latent variable models is the expectation-maximization (EM) algorithm.
>
> — Page 424, *Pattern Recognition and Machine Learning*, 2006.

The EM algorithm is an iterative approach that cycles between two modes. The first mode attempts to estimate the missing or latent variables, called the estimation-step or E-step. The second mode attempts to optimize the parameters of the model to best explain the data, called the maximization-step or M-step.

- **E-Step**. Estimate the missing variables in the dataset.
- **M-Step**. Maximize the parameters of the model in the presence of the data.

The EM algorithm can be applied quite widely, although is perhaps most well known in machine learning for use in unsupervised learning problems, such as density estimation and clustering. Perhaps the most discussed application of the EM algorithm is for clustering with a mixture model.

## 14.4 Gaussian Mixture Model and the EM Algorithm

A mixture model is a model comprised of an unspecified combination of multiple probability distribution functions. A statistical procedure or learning algorithm is used to estimate the parameters of the probability distributions to best fit the density of a given training dataset. The Gaussian Mixture Model, or GMM for short, is a mixture model that uses a combination of Gaussian (Normal) probability distributions and requires the estimation of the mean and standard deviation parameters for each.

There are many techniques for estimating the parameters for a GMM, although a maximum likelihood estimate is perhaps the most common. Consider the case where a dataset is comprised of many points that happen to be generated by two different processes. The points for each process have a Gaussian probability distribution, but the data is combined and the distributions are similar enough that it is not obvious to which distribution a given point may belong. The processes used to generate the data point represents a latent variable, e.g. process 0 and process 1. It influences the data but is not observable. As such, the EM algorithm is an appropriate approach to use to estimate the parameters of the distributions.

In the EM algorithm, the estimation-step would estimate a value for the process latent variable for each data point, and the maximization step would optimize the parameters of the probability distributions in an attempt to best capture the density of the data. The process is repeated until a good set of latent values and a maximum likelihood is achieved that fits the data.

- **E-Step**. Estimate the expected value for each latent variable.
- **M-Step**. Optimize the parameters of the distribution using maximum likelihood.

We can imagine how this optimization procedure could be constrained to just the distribution means, or generalized to a mixture of many different Gaussian distributions.

## 14.5 Example of Gaussian Mixture Model

We can make the application of the EM algorithm to a Gaussian Mixture Model concrete with a worked example. First, let's contrive a problem where we have a dataset where points are generated from one of two Gaussian processes. The points are one-dimensional, the mean of the first distribution is 20, the mean of the second distribution is 40, and both distributions have a standard deviation of 5. We will draw 3,000 points from the first process and 7,000 points from the second process and mix them together.

```
...
# generate a sample
X1 = normal(loc=20, scale=5, size=3000)
X2 = normal(loc=40, scale=5, size=7000)
X = hstack((X1, X2))
```

Listing 14.1: Example of generating a sample from a bimodal probability distribution.

We can then plot a histogram of the points to give an intuition for the dataset. We expect to see a bimodal distribution with a peak for each of the means of the two distributions. The complete example is listed below.

```
# example of a bimodal constructed from two gaussian processes
from numpy import hstack
from numpy.random import normal
from matplotlib import pyplot
# generate a sample
X1 = normal(loc=20, scale=5, size=3000)
X2 = normal(loc=40, scale=5, size=7000)
X = hstack((X1, X2))
# plot the histogram
pyplot.hist(X, bins=50, density=True)
pyplot.show()
```

Listing 14.2: Example of generating and plotting a bimodal data sample.

Running the example creates the dataset and then creates a histogram plot for the data points. The plot clearly shows the expected bimodal distribution with a peak for the first process around 20 and a peak for the second process around 40. We can see that for many of the points in the middle of the two peaks that it is ambiguous as to which distribution they were drawn from.

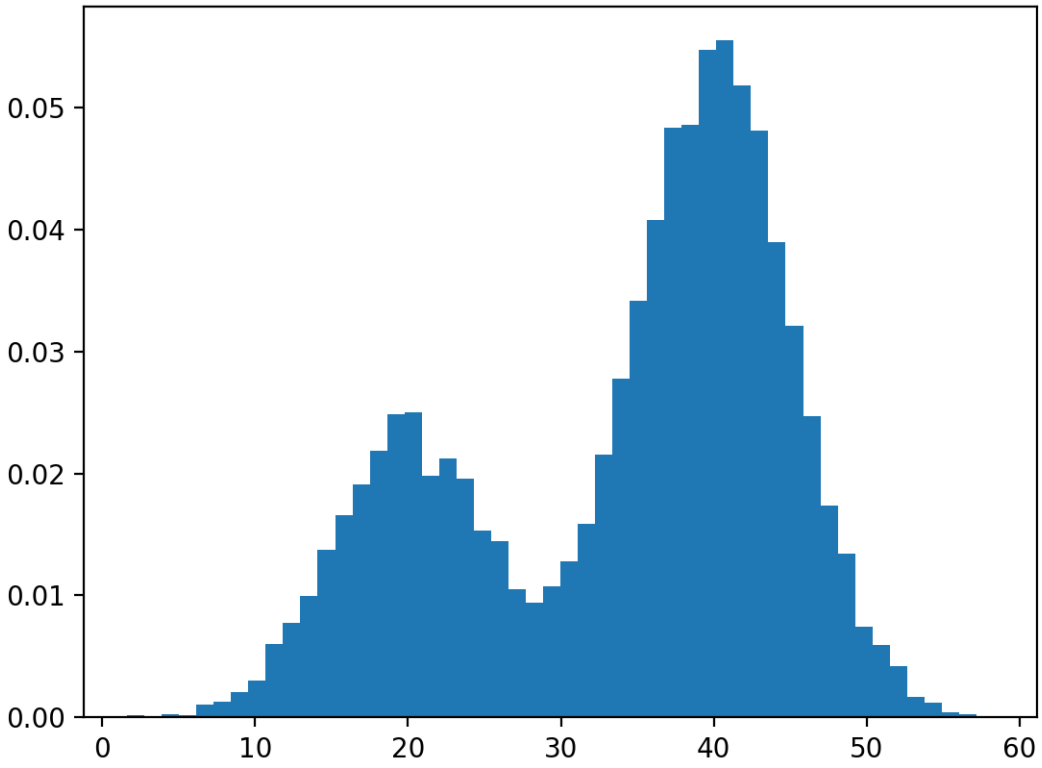


Figure 14.1: Histogram of Dataset Constructed From Two Different Gaussian Processes.

We can model the problem of estimating the density of this dataset using a Gaussian Mixture Model. The `GaussianMixture` scikit-learn class can be used to model this problem and estimate the parameters of the distributions using the expectation-maximization algorithm. The class allows us to specify the suspected number of underlying processes used to generate the data via the `n_components` argument when defining the model. We will set this to 2 for the two processes or distributions.

If the number of processes was not known, a range of different numbers of components could be tested and the model with the best fit could be chosen, where models could be evaluated using scores such as Akaike or Bayesian Information Criterion (AIC or BIC) (covered in Chapter 15). There are also many ways we can configure the model to incorporate other information we may know about the data, such as how to estimate initial values for the distributions. In this case, we will randomly guess the initial parameters, by setting the `init_params` argument to `'random'`.

```
...
# fit model
model = GaussianMixture(n_components=2, init_params='random')
model.fit(X)
```

Listing 14.3: Example of fitting the `GaussianMixture` model.

Once the model is fit, we can access the learned parameters via arguments on the model, such as the means, covariances, mixing weights, and more. More usefully, we can use the fit

model to estimate the latent parameters for existing and new data points. For example, we can estimate the latent variable for the points in the training dataset and we would expect the first 3,000 points to belong to one process (e.g. value=1) and the next 7,000 data points to belong to another process (e.g. value=0).

```
...
# predict latent values
yhat = model.predict(X)
# check latent value for first few points
print(yhat[:100])
# check latent value for last few points
print(yhat[-100:])
```

Listing 14.4: Example of using the fit `GaussianMixture` model to make predictions.

Tying all of this together, the complete example is listed below.

```
# example of fitting a gaussian mixture model with expectation maximization
from numpy import hstack
from numpy.random import normal
from sklearn.mixture import GaussianMixture
# generate a sample
X1 = normal(loc=20, scale=5, size=3000)
X2 = normal(loc=40, scale=5, size=7000)
X = hstack((X1, X2))
# reshape into a table with one column
X = X.reshape((len(X), 1))
# fit model
model = GaussianMixture(n_components=2, init_params='random')
model.fit(X)
# predict latent values
yhat = model.predict(X)
# check latent value for first few points
print(yhat[:100])
# check latent value for last few points
print(yhat[-100:])
```

Listing 14.5: Example of fitting a Gaussian Mixture Model using the EM algorithm.

Running the example fits the Gaussian mixture model on the prepared dataset using the EM algorithm. Once fit, the model is used to predict the latent variable values for the examples in the training dataset. Your specific results may vary given the stochastic nature of the learning algorithm. In this case, we can see that at least for the first few and last few examples in the dataset, that the model mostly predicts the correct value for the latent variable. It's a generally challenging problem and it is expected that the points between the peaks of the distribution will remain ambiguous and assigned to one process or another holistically.

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1
 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Listing 14.6: Example output from fitting a Gaussian Mixture Model using the EM algorithm.

## 14.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 14.6.1 Books

- Section 8.5 The EM Algorithm, *The Elements of Statistical Learning*, 2016.
  https://amzn.to/2YVqu8s
- Chapter 9 Mixture Models and EM, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Section 6.12 The EM Algorithm, *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- Chapter 11 Mixture models and the EM algorithm, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- Section 9.3 Clustering And Probability Density Estimation, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2lnW5S7
- Section 20.3 Learning With Hidden Variables: The EM Algorithm, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.
  https://amzn.to/2Y7yCpO

### 14.6.2 API

- Gaussian mixture models, scikit-learn API.
  https://scikit-learn.org/stable/modules/mixture.html
- `sklearn.mixture.GaussianMixture` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html

### 14.6.3 Articles

- Maximum likelihood estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_likelihood_estimation
- Expectation-maximization algorithm, Wikipedia.
  https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm
- Mixture model, Wikipedia.
  https://en.wikipedia.org/wiki/Mixture_model

## 14.7 Summary

In this tutorial, you discovered the expectation-maximization algorithm. Specifically, you learned:

- Maximum likelihood estimation is challenging on data in the presence of latent variables.
- Expectation maximization provides an iterative solution to maximum likelihood estimation with latent variables.
- Gaussian mixture models are an approach to density estimation where the parameters of the distributions are fit using the expectation-maximization algorithm.

### 14.7.1 Next

In the next tutorial, you will discover probabilistic measures for model selection that operate under the maximum likelihood estimation framework.

# Chapter 15

# Probabilistic Model Selection with AIC, BIC, and MDL

Model selection is the problem of choosing one from among a set of candidate models. It is common to choose a model that performs the best on a hold-out test dataset or to estimate model performance using a resampling technique, such as $k$-fold cross-validation. An alternative approach to model selection involves using probabilistic statistical measures that attempt to quantify both the model performance on the training dataset and the complexity of the model. Examples include the Akaike and Bayesian Information Criterion and the Minimum Description Length. The benefit of these information criterion statistics is that they do not require a hold-out test set, although a limitation is that they do not take the uncertainty of the models into account and may end-up selecting models that are too simple. In this tutorial, you will discover probabilistic statistics for machine learning model selection. After reading this tutorial, you will know:

- Model selection is the challenge of choosing one among a set of candidate models.
- Akaike and Bayesian Information Criterion are two ways of scoring a model based on its log-likelihood and complexity.
- Minimum Description Length provides another scoring method from information theory that can be shown to be equivalent to BIC.

Let's get started.

## 15.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. The Challenge of Model Selection
2. Probabilistic Model Selection
3. Akaike Information Criterion
4. Bayesian Information Criterion
5. Minimum Description Length

## 15.2 The Challenge of Model Selection

Model selection is the process of fitting multiple models on a given dataset and choosing one over all others.

> Model selection: estimating the performance of different models in order to choose the best one.
>
> — Page 222, *The Elements of Statistical Learning*, 2016.

This may apply in unsupervised learning, e.g. choosing a clustering model, or supervised learning, e.g. choosing a predictive model for a regression or classification task. It may also be a sub-task of modeling, such as feature selection for a given model. There are many common approaches that may be used for model selection. For example, in the case of supervised learning, the three most common approaches are:

- Train, Validation, and Test datasets.
- Resampling Methods.
- Probabilistic Statistics.

The simplest reliable method of model selection involves fitting candidate models on a training set, tuning them on the validation dataset, and selecting a model that performs the best on the test dataset according to a chosen metric, such as accuracy or error. A problem with this approach is that it requires a lot of data. Resampling techniques attempt to achieve the same as the train/val/test approach to model selection, although using a small dataset. An example is $k$-fold cross-validation where a training set is split into many train/test pairs and a model is fit and evaluated on each. This is repeated for each model and a model is selected with the best average score across the $k$-folds. A problem with this and the prior approach is that only model performance is assessed, regardless of model complexity.

A third approach to model selection attempts to combine the complexity of the model with the performance of the model into a score, then select the model that minimizes or maximizes the score. We can refer to this approach as statistical or probabilistic model selection as the scoring method uses a probabilistic framework.

## 15.3 Probabilistic Model Selection

Probabilistic model selection (or *information criteria*) provides an analytical technique for scoring and choosing among candidate models. Models are scored both on their performance on the training dataset and based on the complexity of the model.

- **Model Performance**. How well a candidate model has performed on the training dataset.
- **Model Complexity**. How complicated the trained candidate model is after training.

Model performance may be evaluated using a probabilistic framework, such as log-likelihood under the framework of maximum likelihood estimation. Model complexity may be evaluated as the number of degrees of freedom or parameters in the model.

> Historically various 'information criteria' have been proposed that attempt to correct for the bias of maximum likelihood by the addition of a penalty term to compensate for the over-fitting of more complex models.
>
> — Page 33, *Pattern Recognition and Machine Learning*, 2006.

A benefit of probabilistic model selection methods is that a test dataset is not required, meaning that all of the data can be used to fit the model, and the final model that will be used for prediction in the domain can be scored directly. A limitation of probabilistic model selection methods is that the same general statistic cannot be calculated across a range of different types of models. Instead, the metric must be carefully derived for each model.

> It should be noted that the AIC statistic is designed for preplanned comparisons between models (as opposed to comparisons of many models during automated searches).
>
> — Page 493, *Applied Predictive Modeling*, 2013.

A further limitation of these selection methods is that they do not take the uncertainty of the model into account.

> Such criteria do not take account of the uncertainty in the model parameters, however, and in practice they tend to favour overly simple models.
>
> — Page 33, *Pattern Recognition and Machine Learning*, 2006.

There are three statistical approaches to estimating how well a given model fits a dataset and how complex the model is. And each can be shown to be equivalent or proportional to each other, although each was derived from a different framing or field of study.
They are:

- Akaike Information Criterion (AIC). Derived from frequentist probability.
- Bayesian Information Criterion (BIC). Derived from Bayesian probability.
- Minimum Description Length (MDL). Derived from information theory.

Each statistic can be calculated using the log-likelihood for a model and the data. Log-likelihood comes from Maximum Likelihood Estimation (introduced in Chapter 11), a technique for finding or optimizing the parameters of a model in response to a training dataset. In Maximum Likelihood Estimation, we wish to maximize the conditional probability of observing the data ($X$) given a specific probability distribution and its parameters ($\theta$), stated formally as:

$$P(X; \theta) \tag{15.1}$$

Where $X$ is, in fact, the joint probability distribution of all observations from the problem domain from 1 to $n$.

$$P(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{15.2}$$

The joint probability distribution can be restated as the multiplication of the conditional probability for observing each example given the distribution parameters. Multiplying many small probabilities together can be unstable; as such, it is common to restate this problem as the sum of the natural log conditional probability.

$$\sum_{i=1}^{n} \log P(x_i; \theta) \tag{15.3}$$

Given the frequent use of log in the likelihood function, it is commonly referred to as a log-likelihood function. The log-likelihood function for common predictive modeling problems include the mean squared error for regression (e.g. linear regression) and log loss (binary cross-entropy) for binary classification (e.g. logistic regression). We will take a closer look at each of the three statistics, AIC, BIC, and MDL, in the following sections.

## 15.4 Akaike Information Criterion

The Akaike Information Criterion, or AIC for short, is a method for scoring and selecting a model. It is named for the developer of the method, Hirotugu Akaike, and may be shown to have a basis in information theory and frequentist-based inference.

> This is derived from a frequentist framework, and cannot be interpreted as an approximation to the marginal likelihood.
>
> — Page 162, *Machine Learning: A Probabilistic Perspective*, 2012.

The AIC statistic is defined generally for logistic regression as follows[1]:

$$AIC = -\frac{2}{N} \times LL + 2 \times \frac{k}{N} \tag{15.4}$$

Where $N$ is the number of examples in the training dataset, $LL$ is the log-likelihood of the model on the training dataset, and $k$ is the number of parameters in the model. The score, as defined above, is minimized, e.g. the model with the lowest AIC is selected.

> To use AIC for model selection, we simply choose the model giving smallest AIC over the set of models considered.
>
> — Page 231, *The Elements of Statistical Learning*, 2016.

Compared to the BIC method (below), the AIC statistic penalizes complex models less, meaning that it may put more emphasis on model performance on the training dataset, and, in turn, select more complex models.

> We see that the penalty for AIC is less than for BIC. This causes AIC to pick more complex models.
>
> — Page 162, *Machine Learning: A Probabilistic Perspective*, 2012.

[1] Taken from *The Elements of Statistical Learning*

## 15.5 Bayesian Information Criterion

The Bayesian Information Criterion, or BIC for short, is a method for scoring and selecting a model. It is named for the field of study from which it was derived: Bayesian probability and inference. Like AIC, it is appropriate for models fit under the maximum likelihood estimation framework. The BIC statistic is calculated for logistic regression as follows[2]:

$$BIC = -2 \times LL + \log(N) \times k \tag{15.5}$$

Where $log()$ has the base-e called the natural logarithm, $LL$ is the log-likelihood of the model, $N$ is the number of examples in the training dataset, and $k$ is the number of parameters in the model. The score as defined above is minimized, e.g. the model with the lowest BIC is selected. The quantity calculated is different from AIC, although can be shown to be proportional to the AIC. Unlike the AIC, the BIC penalizes the model more for its complexity, meaning that more complex models will have a worse (larger) score and will, in turn, be less likely to be selected.

> Note that, compared to AIC [...], this penalizes model complexity more heavily.
>
> — Page 217, *Pattern Recognition and Machine Learning*, 2006.

Importantly, the derivation of BIC under the Bayesian probability framework means that if a selection of candidate models includes a true model for the dataset, then the probability that BIC will select the true model increases with the size of the training dataset. This cannot be said for the AIC score.

> ... given a family of models, including the true model, the probability that BIC will select the correct model approaches one as the sample size $N \rightarrow$ infinity.
>
> — Page 235, *The Elements of Statistical Learning*, 2016.

A downside of BIC is that for smaller, less representative training datasets, it is more likely to choose models that are too simple.

## 15.6 Minimum Description Length

The Minimum Description Length, or MDL for short, is a method for scoring and selecting a model. It is named for the field of study from which it was derived, namely information theory. Information theory is concerned with the representation and transmission of information on a noisy channel, and as such, measures quantities like entropy, which is the average number of bits required to represent an event from a random variable or probability distribution.

From an information theory perspective, we may want to transmit both the predictions (or more precisely, their probability distributions) and the model used to generate them. Both the predicted target variable and the model can be described in terms of the number of bits required to transmit them on a noisy channel. The Minimum Description Length is the minimum number of bits, or the minimum of the sum of the number of bits required to represent the data and the model.

[2]Taken from *The Elements of Statistical Learning*

> The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths.
>
> — Page 173, *Machine Learning*, 1997.

The MDL statistic is calculated as follows[3]:

$$MDL = L(h) + L(D|h) \tag{15.6}$$

Where $h$ is the model, $D$ is the predictions made by the model, $L(h)$ is the number of bits required to represent the model, and $L(D|h)$ is the number of bits required to represent the predictions from the model on the training dataset. The score as defined above is minimized, e.g. the model with the lowest MDL is selected. The number of bits required to encode $(D|h)$ and the number of bits required to encode $(h)$ can be calculated as the negative log-likelihood; for example[4]:

$$MDL = -log(P(theta)) - \log(P(y|X, theta)) \tag{15.7}$$

Or the negative log-likelihood of the model parameters $(\theta)$ and the negative log-likelihood of the target values $(y)$ given the input values $(X)$ and the model parameters $(\theta)$. This desire to minimize the encoding of the model and its predictions is related to the notion of Occam's Razor that seeks the simplest (least complex) explanation: in this context, the least complex model that predicts the target variable.

> The MDL principle takes the stance that the best theory for a body of data is one that minimizes the size of the theory plus the amount of information necessary to specify the exceptions relative to the theory ...
>
> — Page 198, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.

The MDL calculation is very similar to BIC and can be shown to be equivalent in some situations.

> Hence the BIC criterion, derived as approximation to log-posterior probability, can also be viewed as a device for (approximate) model choice by minimum description length.
>
> — Page 236, *The Elements of Statistical Learning*, 2016.

## 15.7 Worked Example for Linear Regression

We can make the calculation of AIC and BIC concrete with a worked example. In this section, we will use a test problem and fit a linear regression model, then evaluate the model using the AIC and BIC metrics. Importantly, the specific functional form of AIC and BIC for a linear regression model has previously been derived, making the example relatively straightforward. In adapting these examples for your own algorithms, it is important to either find an appropriate

[3]Taken from *Machine Learning* by Tom Mitchell
[4]Taken from *The Elements of Statistical Learning*

derivation of the calculation for your model and prediction problem or look into deriving the calculation yourself. A good starting point would be to review the relevant literature for an existing example.

In this example, we will use a test regression problem provided by the `make_regression()` scikit-learn function. The problem will have two input variables and require the prediction of a target numerical value.

```
...
# generate dataset
X, y = make_regression(n_samples=100, n_features=2, noise=0.1)
# define and fit the model on all data
```

Listing 15.1: Example of preparing a test dataset.

We will fit a `LinearRegression` model on the entire dataset directly.

```
...
# define and fit the model on all data
model = LinearRegression()
model.fit(X, y)
```

Listing 15.2: Example of fitting a `LinearRegression` model.

Once fit, we can report the number of parameters in the model, which, given the definition of the problem, we would expect to be three (two coefficients and one intercept).

```
...
# number of parameters
num_params = len(model.coef_) + 1
print('Number of parameters: %d' % (num_params))
```

Listing 15.3: Example of summarizing the number of parameters in the model.

The likelihood function for a linear regression model can be shown to be identical to the least squares function; therefore, we can estimate the maximum likelihood of the model via the mean squared error metric. First, the model can be used to estimate an outcome for each example in the training dataset, then the `mean_squared_error()` scikit-learn function can be used to calculate the mean squared error for the model.

```
...
# predict the training set
yhat = model.predict(X)
# calculate the error
mse = mean_squared_error(y, yhat)
print('MSE: %.3f' % mse)
```

Listing 15.4: Example of estimating model error on the training dataset.

Tying this all together, the complete example of defining the dataset, fitting the model, and reporting the number of parameters and maximum likelihood estimate of the model is listed below.

```
# generate a test dataset and fit a linear regression model
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
# generate dataset
```

```
X, y = make_regression(n_samples=100, n_features=2, noise=0.1)
# define and fit the model on all data
model = LinearRegression()
model.fit(X, y)
# number of parameters
num_params = len(model.coef_) + 1
print('Number of parameters: %d' % (num_params))
# predict the training set
yhat = model.predict(X)
# calculate the error
mse = mean_squared_error(y, yhat)
print('MSE: %.3f' % mse)
```

Listing 15.5: Example of fitting a linear regression model on a test dataset.

Running the example first reports the number of parameters in the model as 3, as we expected, then reports the MSE as about 0.01. Your specific MSE value may vary given the stochastic nature of the learning algorithm.

```
Number of parameters: 3
MSE: 0.010
```

Listing 15.6: Example output from fitting a linear regression model on a test dataset.

Next, we can adapt the example to calculate the AIC for the model. Skipping the derivation, the AIC calculation for an ordinary least squares linear regression model can be calculated as follows:[5]

$$AIC = n \times LL + 2 \times k \tag{15.8}$$

Where $n$ is the number of examples in the training dataset, $LL$ is the log-likelihood for the model using the natural logarithm (e.g. the log of the mean squared error), and $k$ is the number of parameters in the model. The `calculate_aic()` function below implements this, taking $n$, the raw mean squared error (mse), and $k$ as arguments.

```
# calculate aic for regression
def calculate_aic(n, mse, num_params):
  aic = n * log(mse) + 2 * num_params
  return aic
```

Listing 15.7: Example of a function for calculating AIC for a linear regression model.

The example can then be updated to make use of this new function and calculate the AIC for the model. The complete example is listed below.

```
# calculate akaike information criterion for a linear regression model
from math import log
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# calculate aic for regression
def calculate_aic(n, mse, num_params):
  aic = n * log(mse) + 2 * num_params
  return aic
```

[5] Taken from *A New Look At The Statistical Identification Model*.

```
# generate dataset
X, y = make_regression(n_samples=100, n_features=2, noise=0.1)
# define and fit the model on all data
model = LinearRegression()
model.fit(X, y)
# number of parameters
num_params = len(model.coef_) + 1
print('Number of parameters: %d' % (num_params))
# predict the training set
yhat = model.predict(X)
# calculate the error
mse = mean_squared_error(y, yhat)
print('MSE: %.3f' % mse)
# calculate the aic
aic = calculate_aic(len(y), mse, num_params)
print('AIC: %.3f' % aic)
```

Listing 15.8: Example of evaluating a linear regression model using AIC.

Running the example reports the number of parameters and MSE as before and then reports the AIC.

**Note**: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the AIC is reported to be a value of about -451.616. This value can be minimized in order to choose better models.

```
Number of parameters: 3
MSE: 0.010
AIC: -451.616
```

Listing 15.9: Example output from evaluating a linear regression model using AIC.

We can also explore the same example with the calculation of BIC instead of AIC. Skipping the derivation, the BIC calculation for an ordinary least squares linear regression model can be calculated as follows[6]:

$$BIC = n \times LL + k \times log(n) \tag{15.9}$$

Where $n$ is the number of examples in the training dataset, $LL$ is the log-likelihood for the model using the natural logarithm, and $k$ is the number of parameters in the model, and $log()$ is the natural logarithm. The `calculate_bic()` function below implements this, taking $n$, the raw mean squared error (mse), and $k$ as arguments.

```
# calculate bic for regression
def calculate_bic(n, mse, num_params):
  bic = n * log(mse) + num_params * log(n)
  return bic
```

Listing 15.10: Example of a function for calculating BIC for a linear regression model.

[6]Taken from *Bayesian information criterion, Gaussian special case, Wikipedia.*

The example can then be updated to make use of this new function and calculate the BIC for the model. The complete example is listed below.

```
# calculate bayesian information criterion for a linear regression model
from math import log
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# calculate bic for regression
def calculate_bic(n, mse, num_params):
  bic = n * log(mse) + num_params * log(n)
  return bic

# generate dataset
X, y = make_regression(n_samples=100, n_features=2, noise=0.1)
# define and fit the model on all data
model = LinearRegression()
model.fit(X, y)
# number of parameters
num_params = len(model.coef_) + 1
print('Number of parameters: %d' % (num_params))
# predict the training set
yhat = model.predict(X)
# calculate the error
mse = mean_squared_error(y, yhat)
print('MSE: %.3f' % mse)
# calculate the bic
bic = calculate_bic(len(y), mse, num_params)
print('BIC: %.3f' % bic)
```

Listing 15.11: Example of evaluating a linear regression model using BIC.

Running the example reports the number of parameters and MSE as before and then reports the BIC.

**Note**: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, the BIC is reported to be a value of about -450.020, which is very close to the AIC value of -451.616. Again, this value can be minimized in order to choose better models.

```
Number of parameters: 3
MSE: 0.010
BIC: -450.020
```

Listing 15.12: Example output from evaluating a linear regression model using BIC.

## 15.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 15.8.1 Books

- Chapter 7 Model Assessment and Selection, *The Elements of Statistical Learning*, 2016.
https://amzn.to/2YVqu8s

- Section 1.3 Model Selection, *Pattern Recognition and Machine Learning*, 2006.
https://amzn.to/2JwHE7I

- Section 4.4.1 Model comparison and BIC, *Pattern Recognition and Machine Learning*, 2006.
https://amzn.to/2JwHE7I

- Section 6.6 Minimum Description Length Principle, *Machine Learning*, 1997.
https://amzn.to/2jWd51p

- Section 5.3.2.4 BIC approximation to log marginal likelihood, *Machine Learning: A Probabilistic Perspective*, 2012.
https://amzn.to/2xKSTCP

- Applied Predictive Modeling, 2013.
https://amzn.to/2Helnu5

- Section 28.3 Minimum description length (MDL), *Information Theory, Inference and Learning Algorithms*, 2003.
https://amzn.to/2ZvZlJx

- Section 5.10 The MDL Principle, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
https://amzn.to/2lnW5S7

### 15.8.2 Papers

- *A New Look At The Statistical Identification Model*, 1974.
https://ieeexplore.ieee.org/document/1100705

### 15.8.3 API

- `sklearn.datasets.make_regression` API.
https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_regression.html

- `sklearn.linear_model.LinearRegression` API.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

- `sklearn.metrics.mean_squared_error` API.
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

### 15.8.4 Articles

- Akaike information criterion, Wikipedia.
  https://en.wikipedia.org/wiki/Akaike_information_criterion
- Bayesian information criterion, Wikipedia.
  https://en.wikipedia.org/wiki/Bayesian_information_criterion
- Minimum description length, Wikipedia.
  https://en.wikipedia.org/wiki/Minimum_description_length

## 15.9 Summary

In this tutorial, you discovered probabilistic statistics for machine learning model selection. Specifically, you learned:

- Model selection is the challenge of choosing one among a set of candidate models.
- Akaike and Bayesian Information Criterion are two ways of scoring a model based on its log-likelihood and complexity.
- Minimum Description Length provides another scoring method from information theory that can be shown to be equivalent to BIC.

### 15.9.1 Next

This was the final tutorial in this Part. In the next Part, you will discover Bayes Theorem for conditional probability.

# Part VI

# Bayesian Probability

# Chapter 16

# Introduction to Bayes Theorem

Bayes Theorem provides a principled way for calculating a conditional probability. It is a deceptively simple calculation, providing a method that is easy to calculate for scenarios where our intuition often fails. The best way to develop an intuition for Bayes Theorem is to think about the meaning of the terms in the equation and to apply the calculation many times in a range of different real-world scenarios. This will provide the context for what is being calculated and examples that can be used as a starting point when applying the calculation in new scenarios in the future. In this tutorial, you will discover an intuition for calculating Bayes Theorem by working through multiple realistic scenarios. After completing this tutorial, you will know:

- Bayes Theorem is a technique for calculating a conditional probability.
- The common and helpful names used for the terms in the Bayes Theorem equation.
- How to work through realistic scenarios by calculating Bayes Theorem to find a solution.

Let's get started.

## 16.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What is Bayes Theorem?
2. Naming the Terms in the Theorem
3. Example: Elderly Fall and Death
4. Example: Email and Spam Detection
5. Example: Liars and Lie Detectors

## 16.2 What is Bayes Theorem?

Conditional probability is the probability of one event given the occurrence of another event, often described in terms of events $A$ and $B$ from two dependent random variables e.g. $X$ and $Y$.

- **Conditional Probability**: Probability of one event given the occurrence of one or more other events, e.g. $P(A|B)$.

The conditional probability can be calculated using the joint probability; for example:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{16.1}$$

Additionally, conditional probability is not symmetrical; for example:

$$P(A|B) \neq P(B|A) \tag{16.2}$$

Nevertheless, one conditional probability can be calculated using the other conditional probability. This is called Bayes Theorem, named for Reverend Thomas Bayes, and can be stated as follows:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{16.3}$$

Bayes Theorem provides a principled way for calculating a conditional probability and an alternative to using the joint probability. This alternate approach of calculating the conditional probability is useful either when the joint probability is challenging to calculate, or when the reverse conditional probability is available or easy to calculate.

- **Bayes Theorem**: Principled way of calculating a conditional probability without the joint probability.

It is often the case that we do not have access to the denominator directly, e.g. $P(B)$. We can calculate it an alternative way, for example:

$$P(B) = P(B|A) \times P(A) + P(B|\text{not } A) \times P(\text{not } A) \tag{16.4}$$

This gives a formulation of Bayes Theorem that uses the alternate calculation of $P(B)$, described below:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B|A) \times P(A) + P(B|\text{not } A) \times P(\text{not } A)} \tag{16.5}$$

Note the denominator is simply the expansion we gave above. As such, if we have $P(A)$, then we can calculate $P(\text{not } A)$ as its complement, for example:

$$P(\text{not } A) = 1 - P(A) \tag{16.6}$$

Additionally, if we have $P(\text{not } B|\text{not } A)$, then we can calculate $P(B|\text{not } A)$ as its complement, for example:

$$P(B|\text{not } A) = 1 - P(\text{not } B|\text{not } A) \tag{16.7}$$

Now that we are familiar with the calculation of Bayes Theorem, let's take a closer look at the meaning of the terms in the equation.

## 16.3 Naming the Terms in the Theorem

The terms in the Bayes Theorem equation are given names depending on the context where the equation is used. It can be helpful to think about the calculation from these different perspectives and help to map your problem onto the equation. Firstly, in general the result $P(A|B)$ is referred to as the posterior probability and $P(A)$ is referred to as the prior probability.

- $P(A|B)$: Posterior probability.
- $P(A)$: Prior probability.

Sometimes $P(B|A)$ is referred to as the likelihood and $P(B)$ is referred to as the evidence.

- $P(B|A)$: Likelihood.
- $P(B)$: Evidence.

This allows Bayes Theorem to be restated as:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}} \tag{16.8}$$

We can make this clear with a *Smoke* and *Fire* example. What is the probability that there is fire given that there is smoke, where $P(\text{Fire})$ is the Prior, $P(\text{Smoke}|\text{Fire})$ is the Likelihood, and $P(\text{Smoke})$ is the evidence:

$$P(\text{Fire}|\text{Smoke}) = \frac{P(\text{Smoke}|\text{Fire}) \times P(\text{Fire})}{P(\text{Smoke})} \tag{16.9}$$

We can also think about the calculation in the terms of a binary classifier. For example, $P(B|A)$ may be referred to as the True Positive Rate (TPR) or the **sensitivity**, $P(B|\text{not } A)$ may be referred to as the False Positive Rate (FPR), the complement $P(\text{not } B|\text{not } A)$ may be referred to as the True Negative Rate (TNR) or **specificity**, and the value we are calculating $P(A|B)$ may be referred to as the Positive Predictive Value (PPV) or the **precision**.

- $P(\text{not } B|\text{not } A)$: True Negative Rate or TNR (specificity).
- $P(B|\text{not } A)$: False Positive Rate or FPR.
- $P(\text{not } B|A)$: False Negative Rate or FNR.
- $P(B|A)$: True Positive Rate or TPR (sensitivity or recall).
- $P(A|B)$: Positive Predictive Value or PPV (precision).

For example, we may re-state the calculation using these terms as follows:

$$PPV = \frac{TPR \times P(A)}{TPR \times P(A) + FPR \times P(\text{not } A)} \tag{16.10}$$

Now that we are familiar with what Bayes Theorem and the meaning of the terms, let's look at some scenarios where we can calculate it. Note that all of the following examples are contrived, they are not based on real-world probabilities.

## 16.4 Example: Elderly Fall and Death

Consider the case where an elderly person (over 80 years of age) falls, what is the probability that they will die from the fall? Let's assume that the base rate of someone elderly dying $P(A)$ is 10%, and the base rate for elderly people falling $P(B)$ is 5%, and from all elderly people, 7% of those that die had a fall $P(B|A)$. Let's plug what we know into the theorem:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$
$$P(\text{Die}|\text{Fall}) = \frac{P(\text{Fall}|\text{Die}) \times P(\text{Die})}{P(\text{Fall})} \tag{16.11}$$

Or:

$$P(A|B) = \frac{0.07 \times 0.10}{0.05}$$
$$P(\text{Die}|\text{Fall}) = 0.14 \tag{16.12}$$

That is if an elderly person falls then there is a 14% probability that they will die from the fall. To make this concrete, we can perform the calculation in Python, first defining what we know then using Bayes Theorem to calculate the outcome. The complete example is listed below.

```
# calculate the probability of an elderly person dying from a fall

# calculate P(A|B) given P(B|A), P(A) and P(B)
def bayes_theorem(p_a, p_b, p_b_given_a):
  # calculate P(A|B) = P(B|A) * P(A) / P(B)
  p_a_given_b = (p_b_given_a * p_a) / p_b
  return p_a_given_b

# P(A)
p_a = 0.10
# P(B)
p_b = 0.05
# P(B|A)
p_b_given_a = 0.07
# calculate P(A|B)
result = bayes_theorem(p_a, p_b, p_b_given_a)
# summarize
print('P(A|B) = %.3f%%' % (result * 100))
```

Listing 16.1: Example of calculating the probability of dieing from a fall.

Running the example confirms the value that we calculated manually.

```
P(A|B) = 14.000%
```

Listing 16.2: Example output from calculating the probability of dieing from a fall.

## 16.5 Example: Email and Spam Detection

Consider the case where we receive an email and the spam detector puts it in the spam folder, what is the probability it was spam? Let's assume some details such as 2% of the email we

receive is spam $P(A)$. Let's assume that the spam detector is really good and when an email is spam that it detects it $P(B|A)$ with an accuracy of 99%, and when an email is not spam, it will mark it as spam with a very low rate of 0.1% $P(B|\text{not } A)$. Let's plug what we know into the theorem:

$$\begin{aligned} P(A|B) &= \frac{P(B|A) \times P(A)}{P(B)} \\ P(\text{Spam}|\text{Detected}) &= \frac{P(\text{Detected}|\text{Spam}) \times P(\text{Spam})}{P(\text{Detected})} \end{aligned} \tag{16.13}$$

Or:

$$P(\text{Spam}|\text{Detected}) = \frac{0.99 \times 0.02}{P(\text{Detected})} \tag{16.14}$$

We don't know $P(B)$, that is $P(\text{Detected})$, but we can calculate it using:

$$\begin{aligned} P(B) &= P(B|A) \times P(A) + P(B|\text{not } A) \times P(\text{not } A) \\ P(\text{Detected}) &= P(\text{Detected}|\text{Spam}) \times P(\text{Spam}) + P(\text{Detected}|\text{not Spam}) \times P(\text{not Spam}) \end{aligned} \tag{16.15}$$

We know $P(\text{Detected}|\text{not Spam})$ which is 0.1% and we can calculate $P(\text{not Spam})$ as $1 - P(\text{Spam})$, for example:

$$\begin{aligned} P(\text{not Spam}) &= 1 - P(\text{Spam}) \\ &= 1 - 0.02 \\ &= 0.98 \end{aligned} \tag{16.16}$$

Therefore, we can calculate $P(\text{Detected})$ as:

$$\begin{aligned} P(\text{Detected}) &= 0.99 \times 0.02 + 0.001 \times 0.98 \\ &= 0.0198 + 0.00098 \\ &= 0.02078 \end{aligned} \tag{16.17}$$

That is, about 2% of all email is detected as spam, regardless of whether it is spam or not. Now we can calculate the answer to our original question as:

$$\begin{aligned} P(\text{Spam}|\text{Detected}) &= \frac{0.99 \times 0.02}{0.02078} \\ &= \frac{0.0198}{0.02078} \\ &= 0.95283926852743 \end{aligned} \tag{16.18}$$

That is, if an email is in the spam folder, there is a 95.2% probability that it is in fact spam. Again, let's confirm this result by calculating it in Python. The complete example is listed below.

```
# calculate the probability of an email in the spam folder being spam

# calculate P(A|B) given P(A), P(B|A), P(B|not A)
def bayes_theorem(p_a, p_b_given_a, p_b_given_not_a):
```

```
  # calculate P(not A)
  not_a = 1 - p_a
  # calculate P(B)
  p_b = p_b_given_a * p_a + p_b_given_not_a * not_a
  # calculate P(A|B)
  p_a_given_b = (p_b_given_a * p_a) / p_b
  return p_a_given_b

# P(A)
p_a = 0.02
# P(B|A)
p_b_given_a = 0.99
# P(B|not A)
p_b_given_not_a = 0.001
# calculate P(A|B)
result = bayes_theorem(p_a, p_b_given_a, p_b_given_not_a)
# summarize
print('P(A|B) = %.3f%%' % (result * 100))
```

Listing 16.3: Example of calculating the probability of detecting spam.

Running the example gives the same result, confirming our manual calculation.

```
P(A|B) = 95.284%
```

Listing 16.4: Example output from calculating the probability of detecting spam.

## 16.6 Example: Liars and Lie Detectors

Consider the case where a person is tested with a lie detector and gets a positive result suggesting that they are lying, what is the probability that the person is indeed lying? Let's assume some details, such as most people that are tested are telling the truth, such as 98%, meaning $(1 - 0.98)$ or 2% are liars $P(A)$. Let's also assume that when someone is lying that the test can detect them well, but not great, such as 72% of the time $P(B|A)$. Let's also assume that when the machine says they are not lying, this is true 97% of the time $P(\text{not } B|\text{ not } A)$. Let's plug what we know into the theorem:

$$\begin{aligned} P(A|B) &= \frac{P(B|A) \times P(A)}{P(B)} \\ P(\text{Lying}|\text{Positive}) &= \frac{P(\text{Positive}|\text{Lying}) \times P(\text{Lying})}{P(\text{Positive})} \end{aligned} \tag{16.19}$$

Or:

$$P(\text{Lying}|\text{Positive}) = \frac{0.72 \times 0.02}{P(\text{Positive})} \tag{16.20}$$

Again, we don't know $P(B)$, or in this case how often the detector returns a positive result

in general. We can calculate this using the formula:

$$\begin{aligned} P(B) &= P(B|A) \times P(A) + P(B|\text{not } A) \times P(\text{not } A) \\ P(\text{Positive}) &= P(\text{Positive}|\text{Lying}) \times P(\text{Lying}) + P(\text{Positive}|\text{not Lying}) \times P(\text{not Lying}) \\ &= 0.72 \times 0.02 + P(\text{Positive}|\text{not Lying}) \times (1 - 0.02) \\ &= 0.72 \times 0.02 + P(\text{Positive}|\text{not Lying}) \times 0.98 \end{aligned} \tag{16.21}$$

In this case, we don't know the probability of a positive detection result given that the person was not lying, that is we don't know the false positive rate or the false alarm rate. This can be calculated as follows:

$$\begin{aligned} P(B|\text{not } A) &= 1 - P(\text{not } B|\text{not } A) \\ P(\text{Positive}|\text{not Lying}) &= 1 - P(\text{not Positive}|\text{not Lying}) \\ &= 1 - 0.97 \\ &= 0.03 \end{aligned} \tag{16.22}$$

Therefore, we can calculate $P(\text{Positive})$ as:

$$\begin{aligned} P(\text{Positive}) &= 0.72 \times 0.02 + 0.03 \times 0.98 \\ &= 0.0144 + 0.0294 \\ &= 0.0438 \end{aligned} \tag{16.23}$$

That is, the test returns a positive result about 4% of the time, regardless of whether the person is lying or not. We can now calculate Bayes Theorem for this scenario:

$$\begin{aligned} P(\text{Lying}|\text{Positive}) &= \frac{0.72 \times 0.02}{0.0438} \\ &= \frac{0.0144}{0.0438} \\ &= 0.328767123287671 \end{aligned} \tag{16.24}$$

That is, if the lie detector test comes back with a positive result, then there is a 32.8% probability that they are in fact lying. It's a poor test! Finally, let's confirm this calculation in Python. The complete example is listed below.

```
# calculate the probability of a person lying given a positive lie detector result

# calculate P(A|B) given P(A), P(B|A), P(not B|not A)
def bayes_theorem(p_a, p_b_given_a, p_not_b_given_not_a):
  # calculate P(not A)
  not_a = 1 - p_a
  # calculate P(B|not A)
  p_b_given_not_a = 1 - p_not_b_given_not_a
  # calculate P(B)
  p_b = p_b_given_a * p_a + p_b_given_not_a * not_a
  # calculate P(A|B)
  p_a_given_b = (p_b_given_a * p_a) / p_b
  return p_a_given_b

# P(A), base rate
```

```
p_a = 0.02
# P(B|A)
p_b_given_a = 0.72
# P(not B| not A)
p_not_b_given_not_a = 0.97
# calculate P(A|B)
result = bayes_theorem(p_a, p_b_given_a, p_not_b_given_not_a)
# summarize
print('P(A|B) = %.3f%%' % (result * 100))
```

Listing 16.5: Example of calculating the probability of lying given a positive lie detector result.

Running the example gives the same result, confirming our manual calculation.

```
P(A|B) = 32.877%
```

Listing 16.6: Example output from calculating the probability of lying given a positive lie detector result.

We have now seen a few different cases of how simple problems can be mapped onto Bayes Theorem and how to work through their solution both manually and with Python code examples. This provides the basis for applying Bayes Theorem on your own similar problems.

## 16.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 16.7.1 Books

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- *Pattern Classification*, 2nd Edition, 2001.
  https://amzn.to/2xVBI1n
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

### 16.7.2 Articles

- Conditional probability, Wikipedia.
  https://en.wikipedia.org/wiki/Conditional_probability
- Bayes' theorem, Wikipedia.
  https://en.wikipedia.org/wiki/Bayes%27_theorem
- Maximum a posteriori estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation

- False positives and false negatives, Wikipedia.
  https://en.wikipedia.org/wiki/False_positives_and_false_negatives
- Base rate fallacy, Wikipedia.
  https://en.wikipedia.org/wiki/Base_rate_fallacy
- Sensitivity and specificity, Wikipedia.
  https://en.wikipedia.org/wiki/Sensitivity_and_specificity

## 16.8 Summary

In this tutorial, you discovered an intuition for calculating Bayes Theorem by working through multiple realistic scenarios. Specifically, you learned:

- Bayes Theorem is a technique for calculating a conditional probability.
- The common and helpful names used for the terms in the Bayes Theorem equation.
- How to work through realistic scenarios by calculating Bayes Theorem to find a solution.

### 16.8.1 Next

In the next tutorial, you will discover the Bayesian perspective of machine learning.

# Chapter 17

# Bayes Theorem and Machine Learning

Machine learning can be understood using Bayes Theorem as selecting a model that best describes the observed data. This perspective provides the basis for the Maximum a Posteriori or MAP probabilistic framework that can be shown to underlie many machine learning models such as linear regression and logistic regression. This framing of machine learning also gives rise to the notion of how to make optimal predictions for a new example using a Bayes Optimal Classifier, that in turn exposes Bayes Error, the minimum error expected when making predictions on a dataset. In this tutorial, you will discover a Bayesian perspective of machine learning modeling and prediction.

- How selecting a machine learning model can be framed as solving Bayes Theorem.
- How the Maximum a Posteriori probabilistic framework can be used to optimize a model for a dataset.
- How the Bayes Optimal Classifier defines the best possible prediction that can be made for a set of models on a dataset.

Let's get started.

## 17.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Bayes Theorem of Modeling Hypotheses
2. Density Estimation
3. Maximum a Posteriori
4. MAP and Machine Learning
5. Bayes Optimal Classifier

## 17.2 Bayes Theorem of Modeling Hypotheses

Bayes Theorem is a useful tool in applied machine learning. It provides a way of thinking about the relationship between data and a model. A machine learning algorithm or model is a specific way of thinking about the structured relationships in the data. In this way, a model can be thought of as a hypothesis about the relationships in the data, such as the relationship between input ($X$) and output ($y$). The practice of applied machine learning is the testing and analysis of different hypotheses (models) on a given dataset. Bayes Theorem provides a probabilistic model to describe the proportional relationship between data ($D$) and a hypothesis ($H$); for example:

$$P(h|D) = \frac{P(D|h) \times P(h)}{P(D)} \tag{17.1}$$

Breaking this down, it says that the probability of a given hypothesis holding or being true given some observed data can be calculated as the probability of observing the data given the hypothesis multiplied by the probability of the hypothesis being true regardless of the data, divided by the probability of observing the data regardless of the hypothesis.

> Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.
>
> — Page 156, *Machine Learning*, 1997.

Under this framework, each piece of the calculation has a specific name; for example:

- $P(h|D)$: Posterior probability of the hypothesis (the thing we want to calculate).
- $P(h)$: Prior probability of the hypothesis.

This gives a useful framework for thinking about and modeling a machine learning problem. If we have some prior domain knowledge about the hypothesis, this is captured in the prior probability. If we don't, then all hypotheses may have the same prior probability. If the probability of observing the data $P(D)$ increases, then the probability of the hypothesis holding given the data $P(h|D)$ decreases. Conversely, if the probability of the hypothesis $P(h)$ and the probability of observing the data given the hypothesis increases, the probability of the hypothesis holding given the data $P(h|D)$ increases.

The notion of testing different models on a dataset in applied machine learning can be thought of as estimating the probability of each hypothesis ($h_1, h_2, h_3, \cdots, \in H$) being true given the observed data. Under this framework, the probability of the data ($D$) is constant as it is used in the assessment of each hypothesis. Therefore, it can be removed from the calculation to give the simplified unnormalized estimate as follows:

$$\max h \in H P(h|D) = P(D|h) \times P(h) \tag{17.2}$$

If we do not have any prior information about the hypothesis being tested, they can be assigned a uniform probability, and this term too will be a constant and can be removed from the calculation to give the following:

$$\max h \in H P(h|D) = P(D|h) \tag{17.3}$$

That is, the goal is to locate a hypothesis that best explains the observed data. This simplification provides the basis for an optimization procedure for searching for a model and set of parameters that best fits the data, referred to generally as density estimation.

## 17.3 Density Estimation

A common modeling problem involves how to estimate a joint probability distribution for a dataset. For example, given a sample of observations ($X$) from a domain ($x_1, x_2, x_3, \cdots, x_n$), where each observation is drawn independently from the domain with the same probability distribution (so-called independent and identically distributed, i.i.d., or close to it). Density estimation involves selecting a probability distribution function and the parameters of that distribution that best explains the joint probability distribution of the observed data ($X$). Often estimating the density is too challenging; instead, we are happy with a point estimate from the target distribution, such as the mean. There are many techniques for solving this problem, although two common approaches are:

- Maximum a Posteriori (MAP), a Bayesian method.
- Maximum Likelihood Estimation (MLE), a frequentist method.

Both approaches frame the problem as optimization and involve searching for a distribution and set of parameters for the distribution that best describes the observed data. In Maximum Likelihood Estimation (introduced in Chapter 11), we wish to maximize the probability of observing the data from the joint probability distribution given a specific probability distribution and its parameters, stated formally as:

$$P(X; \theta) \tag{17.4}$$

or

$$P(x_1, x_2, x_3, \cdots, x_n; \theta) \tag{17.5}$$

This resulting conditional probability is referred to as the likelihood of observing the data given the model parameters. The objective of Maximum Likelihood Estimation is to find the set of parameters ($\theta$) that maximize the likelihood function, e.g. result in the largest likelihood value.

$$\max P(X; \theta) \tag{17.6}$$

An alternative and closely related approach is to consider the optimization problem from the perspective of Bayesian probability.

> A popular replacement for maximizing the likelihood is maximizing the Bayesian posterior probability density of the parameters instead.
>
> — Page 306, *Information Theory, Inference and Learning Algorithms*, 2003.

Let's take a closer look at the Bayesian approach to solving this optimization problem.

## 17.4 Maximum a Posteriori

Recall that the Bayes theorem provides a principled way of calculating a conditional probability. It involves calculating the conditional probability of one outcome given another outcome without using the joint probability of the outcomes, stated as follows:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{17.7}$$

The quantity that we are calculating is typically referred to as the posterior probability of $A$ given $B$ and $P(A)$ is referred to as the prior probability of $A$. The normalizing constant of $P(B)$ can be removed, and the posterior can be shown to be proportional to ($\propto$) the probability of $B$ given $A$ multiplied by the prior.

$$P(A|B) \propto P(B|A) \times P(A) \tag{17.8}$$

Therefore, for our purposes, we can simplify the calculation as:

$$P(A|B) = P(B|A) \times P(A) \tag{17.9}$$

This is a helpful simplification as we are not interested in estimating a probability, but instead in optimizing a quantity. A proportional quantity is good enough for this purpose. We can now relate this calculation to our desire to estimate a distribution and parameters ($\theta$) that best explains our dataset ($X$), as we described in the previous section. This can be stated as:

$$P(\theta|X) = P(X|\theta) \times P(\theta) \tag{17.10}$$

Maximizing this quantity over a range of $\theta$ solves an optimization problem for estimating the central tendency of the posterior probability (e.g. the model of the distribution). As such, this technique is referred to as *maximum a posteriori estimation*, or MAP estimation for short, and sometimes simply *maximum posterior estimation*.

$$\max P(X|\theta) \times P(\theta) \tag{17.11}$$

This is equivalent to $\max h \in H P(h|D) = P(D|h)$ that we derived in the first section, where here we refer to our hypothesis $h$ as $\theta$. We are typically not calculating the full posterior probability distribution, and in fact, this may not be tractable for many problems of interest.

> ... finding MAP hypotheses is often much easier than Bayesian learning, because it requires solving an optimization problem instead of a large summation (or integration) problem.
>
> — Page 804, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

Instead, we are calculating a point estimate such as a moment of the distribution, like the mode, the most common value, which is the same as the mean for the normal distribution.

> One common reason for desiring a point estimate is that most operations involving the Bayesian posterior for most interesting models are intractable, and a point estimate offers a tractable approximation.

— Page 139, *Deep Learning*, 2016.

Note that this is very similar to Maximum Likelihood Estimation, with the addition of the prior probability over the distribution and parameters. In fact, if we assume that all values of $\theta$ are equally likely because we don't have any prior information (e.g. a uniform prior), then both calculations are equivalent. Because of this equivalence, both MLE and MAP often converge to the same optimization problem for many machine learning algorithms. This is not always the case; if the calculation of the MLE and MAP optimization problem differ, the MLE and MAP solution found for an algorithm may also differ.

> ... the maximum likelihood hypothesis might not be the MAP hypothesis, but if one assumes uniform prior probabilities over the hypotheses then it is.

— Page 167, *Machine Learning*, 1997.

## 17.5 MAP and Machine Learning

In machine learning, Maximum a Posteriori optimization provides a Bayesian probability framework for fitting model parameters to training data and an alternative and sibling to the perhaps more common Maximum Likelihood Estimation framework.

> Maximum a posteriori (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.

— Page 825, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.

One framework is not better than another, and as mentioned, in many cases, both frameworks frame the same optimization problem from different perspectives. Instead, MAP is appropriate for those problems where there is some prior information, e.g. where a meaningful prior can be set to weight the choice of different distributions and parameters or model parameters. MLE is more appropriate where there is no such prior.

> Bayesian methods can be used to determine the most probable hypothesis given the data-the maximum a posteriori (MAP) hypothesis. This is the optimal hypothesis in the sense that no other hypothesis is more likely.

— Page 197, *Machine Learning*, 1997.

In fact, the addition of the prior to the MLE can be thought of as a type of regularization of the MLE calculation. This insight allows other regularization methods (e.g. L2 norm in models that use a weighted sum of inputs) to be interpreted under a framework of MAP Bayesian inference. For example, L2 is a bias or prior that assumes that a set of coefficients or weights have a small sum squared value.

> ... in particular, L2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on the weights.

— Page 236, *Deep Learning*, 2016.

Like MLE, solving the optimization problem depends on the choice of model. For simpler models, like linear regression, there are analytical solutions. For more complex models like logistic regression, numerical optimization is required that makes use of first- and second-order derivatives. For the more prickly problems, stochastic optimization algorithms may be required.

Now that we are familiar with how searching across candidate models can be framed using Bayes Theorem, let's look at one more important application of Bayes Theorem in machine learning, called the Bayes Classifier.

## 17.6 Bayes Optimal Classifier

The Bayes optimal classifier is a probabilistic model that makes the most probable prediction for a new example, given the training dataset. This model is also referred to as the Bayes optimal learner, the Bayes classifier, Bayes optimal decision boundary or the Bayes optimal discriminant function.

- **Bayes Classifier**: Probabilistic model that makes the most probable prediction for new examples.

Specifically, the Bayes optimal classifier answers the question: *What is the most probable classification of the new instance given the training data?* This is different from the MAP framework that seeks the most probable hypothesis (model). Instead, we are interested in making a specific prediction.

> In general, the most probable classification of the new instance is obtained by combining the predictions of all hypotheses, weighted by their posterior probabilities.

— Page 175, *Machine Learning*, 1997.

The equation below demonstrates how to calculate the conditional probability for a new instance ($v_i$) given the training data ($D$), given a space of hypotheses ($H$).

$$P(v_j|D) = \sum^{h \in H} P(v_j|h_i) \times P(h_i|D) \tag{17.12}$$

Where $v_j$ is a new instance to be classified, $H$ are the set of hypotheses for classifying the instance, $h_i$ is a given hypothesis, $P(v_j|h_i)$ is the posterior probability for $v_i$ given hypothesis $h_i$, and $P(h_i|D)$ is the posterior probability of the hypothesis $h_i$ given the data $D$. Selecting the outcome with the maximum probability is an example of a Bayes optimal classification.

$$\max \sum^{h \in H} P(v_j|h_i) \times P(h_i|D) \tag{17.13}$$

Any model that classifies examples using this equation is a Bayes optimal classifier and no other model can outperform this technique on average.

> Any system that classifies new instances according to [the equation] is called a Bayes optimal classifier, or Bayes optimal learner. No other classification method using the same hypothesis space and same prior knowledge can outperform this method on average.
>
> — Page 175, *Machine Learning*, 1997.

We have to let that sink in. It is a big deal. It means that any other algorithm that operates on the same data, the same set of hypotheses and same prior probabilities cannot outperform this approach, on average. Hence the name *optimal classifier*. Although the classifier makes optimal predictions, it is not perfect, given the uncertainty in the training data and incomplete coverage of the problem domain and hypothesis space. As such the model will make errors. This error are often referred to as Bayes error.

> The Bayes classifier produces the lowest possible test error rate, called the Bayes error rate. [...] The Bayes error rate is analogous to the irreducible error ...
>
> — Page 38, *An Introduction to Statistical Learning with Applications in R*, 2017.

Because the Bayes classifier is optimal, the Bayes error is the minimum possible error that can be made on a dataset.

- **Bayes Error**: The minimum possible error that can be made when making predictions.

Further, the model is often described in terms of classification, e.g. the *Bayes Classifier*. Nevertheless, the principal applies just as well to regression, that is predictive modeling problems where a numerical value is predicted instead of a class label. It is a theoretical model, but it is held up as an ideal that we may wish to pursue.

> In theory we would always like to predict qualitative responses using the Bayes classifier. But for real data, we do not know the conditional distribution of $Y$ given $X$, and so computing the Bayes classifier is impossible. Therefore, the Bayes classifier serves as an unattainable gold standard against which to compare other methods.
>
> — Page 39, *An Introduction to Statistical Learning with Applications in R*, 2017.

Because of the computational cost of this optimal strategy, we instead can work with direct simplifications of the approach. Two of the most commonly used approaches are using a sampling algorithm for hypotheses such as Gibbs sampling, or to use the simplifying assumptions of the Naive Bayes classifier.

- **Gibbs Algorithm**. Randomly sample hypotheses biased on their posterior probability.
- **Naive Bayes**. Assume that variables in the input data are conditionally independent.

Nevertheless, many nonlinear machine learning algorithms are able to make predictions are that are close approximations of the Bayes classifier in practice.

> Despite the fact that it is a very simple approach, KNN can often produce classifiers that are surprisingly close to the optimal Bayes classifier.
>
> — Page 39, *An Introduction to Statistical Learning with Applications in R*, 2017.

## 17.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 17.7.1 Books

- Chapter 6 Bayesian Learning, *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- Chapter 12 Maximum Entropy Models, *Foundations of Machine Learning*, 2018.
  https://amzn.to/2Zp9f3A
- Chapter 9 Probabilistic methods, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2lnW5S7
- Chapter 5 Machine Learning Basics, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL
- Chapter 13 MAP Inference, *Probabilistic Graphical Models: Principles and Techniques*, 2009.
  https://amzn.to/324l0tT
- *An Introduction to Statistical Learning with Applications in R*, 2017.
  https://amzn.to/2O4gCHv

### 17.7.2 Articles

- Maximum a posteriori estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation
- Bayesian statistics, Wikipedia.
  https://en.wikipedia.org/wiki/Bayesian_statistics

## 17.8 Summary

In this tutorial, you discovered a Bayesian perspective of machine learning modeling and prediction.

- How selecting a machine learning model can be framed as solving Bayes Theorem.
- How the Maximum a Posteriori probabilistic framework can be used to optimize a model for a dataset.
- How the Bayes Optimal Classifier defines the best possible prediction that can be made for a set of models on a dataset.

### 17.8.1 Next

In the next tutorial, you will discover the Naive Bayes classification algorithm.

# Chapter 18

# How to Develop a Naive Bayes Classifier

Classification is a predictive modeling problem that involves assigning a label to a given input data sample. The problem of classification predictive modeling can be framed as calculating the conditional probability of a class label given a data sample. Bayes Theorem provides a principled way for calculating this conditional probability, although in practice requires an enormous number of samples (very large-sized dataset) and is computationally expensive. Instead, the calculation of Bayes Theorem can be simplified by making some assumptions, such as each input variable is independent of all other input variables. Although a dramatic and unrealistic assumption, this has the effect of making the calculations of the conditional probability tractable and results in an effective classification model referred to as Naive Bayes. In this tutorial, you will discover the Naive Bayes algorithm for classification predictive modeling. After completing this tutorial, you will know:

- How to frame classification predictive modeling as a conditional probability model.
- How to use Bayes Theorem to solve the conditional probability model of classification.
- How to implement simplified Bayes Theorem for classification, called the Naive Bayes algorithm.

Let's get started.

## 18.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Conditional Probability Model of Classification
2. Simplified or Naive Bayes
3. How to Calculate the Prior and Conditional Probabilities
4. Worked Example of Naive Bayes
5. 5 Tips When Using Naive Bayes

## 18.2 Conditional Probability Model of Classification

In machine learning, we are often interested in a predictive modeling problem where we want to predict a class label for a given observation. For example, classifying the species of plant based on measurements of the flower. Problems of this type are referred to as classification predictive modeling problems, as opposed to regression problems that involve predicting a numerical value. The observation or input to the model is referred to as $X$ and the class label or output of the model is referred to as $y$. Together, $X$ and $y$ represent observations collected from the domain, i.e. a table or matrix (columns and rows or features and samples) of training data used to fit a model. The model must learn how to map specific examples to class labels or $y = f(X)$ that minimized the error of misclassification.

One approach to solving this problem is to develop a probabilistic model. From a probabilistic perspective, we are interested in estimating the conditional probability of the class label, given the observation. For example, a classification problem may have $k$ class labels $y_1, y_2, \cdots, y_k$ and $n$ input variables, $X_1, X_2, \cdots, X_n$. We can calculate the conditional probability for a class label with a given instance or set of input values for each column $x_1, x_2, \cdots, x_n$ as follows:

$$P(y_i|x_1, x_2, \cdots, x_n) \tag{18.1}$$

The conditional probability can then be calculated for each class label in the problem and the label with the highest probability can be returned as the most likely classification. The conditional probability can be calculated using the joint probability, although it would be intractable. Bayes Theorem provides a principled way for calculating the conditional probability. The simple form of the calculation for Bayes Theorem is as follows:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{18.2}$$

Where the probability that we are interested in calculating $P(A|B)$ is called the posterior probability and the marginal probability of the event $P(A)$ is called the prior. We can frame classification as a conditional classification problem with Bayes Theorem as follows:

$$P(y_i|x_1, x_2, \cdots, x_n) = \frac{P(x_1, x_2, \cdots, x_n|y_i) \times P(y_i)}{P(x_1, x_2, \cdots, x_n)} \tag{18.3}$$

The prior $P(y_i)$ is easy to estimate from a dataset, but the conditional probability of the observation based on the class $P(x_1, x_2, \cdots, x_n|y_i)$ is not feasible unless the number of examples is extraordinarily large, e.g. large enough to effectively estimate the probability distribution for all different possible combinations of values. As such, the direct application of Bayes Theorem also becomes intractable, especially as the number of variables or features ($n$) increases.

## 18.3 Simplified or Naive Bayes

The solution to using Bayes Theorem for a conditional probability classification model is to simplify the calculation. The Bayes Theorem assumes that each input variable is dependent upon all other variables. This is a cause of complexity in the calculation. We can remove this assumption and consider each input variable as being independent from each other. This

changes the model from a dependent conditional probability model to an independent conditional probability model and dramatically simplifies the calculation.

First, the denominator is removed from the calculation of $P(y_i|x_1, x_2, \cdots, x_n)$ as it is a constant used in calculating the conditional probability of each class for a given instance and has the effect of normalizing the result.

$$P(y_i|x_1, x_2, \cdots, x_n) = P(x_1, x_2, \cdots, x_n|y_i) \times P(y_i) \tag{18.4}$$

Next, the conditional probability of all variables given the class label is changed into separate conditional probabilities of each variable value given the class label. These independent conditional variables are then multiplied together. For example:

$$P(y_i|x_1, x_2, \cdots, x_n) = P(x_1|y_i) \times P(x_2|y_i) \times \cdots \times P(x_n|y_i) \times P(y_i) \tag{18.5}$$

This calculation can be performed for each of the class labels, and the label with the largest proportional score can be selected as the classification for the given instance. This decision rule is referred to as the maximum a posteriori, or MAP, decision rule. This simplification of Bayes Theorem is common and widely used for classification predictive modeling problems and is generally referred to as Naive Bayes. The word *naive* is French and typically has a diaeresis (umlaut) over the $i$ (e.g naïve), which is commonly left out for simplicity, and *Bayes* is capitalized as it is named for Reverend Thomas Bayes.

## 18.4 How to Calculate the Prior and Conditional Probabilities

Now that we know what Naive Bayes is, we can take a closer look at how to calculate the elements of the equation. The calculation of the prior $P(y_i)$ is straightforward. It can be estimated by dividing the frequency of observations in the training dataset that have the class label by the total number of examples (rows) in the training dataset. For example:

$$P(y_i) = \frac{\text{examples with } y_i}{\text{total examples}} \tag{18.6}$$

The conditional probability for a feature value given the class label can also be estimated from the data. Specifically, those data examples that belong to a given class, and one data distribution per variable. This means that if there are $K$ classes and $n$ variables, that $k \times n$ different probability distributions must be created and maintained. A different approach is required depending on the data type of each feature. Specifically, the data is used to estimate the parameters of one of three standard probability distributions. In the case of categorical variables, such as counts or labels, a multinomial distribution can be used. If the variables are binary, such as yes/no or true/false, a binomial distribution can be used. If a variable is numerical, such as a measurement, often a Gaussian distribution is used.

- **Binary**: Binomial distribution.
- **Categorical**: Multinomial distribution.
- **Numeric**: Gaussian distribution.

These three distributions are so common that the Naive Bayes implementation is often named after the distribution. For example:

- **Binomial Naive Bayes**: Naive Bayes that uses a binomial distribution.
- **Multinomial Naive Bayes**: Naive Bayes that uses a multinomial distribution.
- **Gaussian Naive Bayes**: Naive Bayes that uses a Gaussian distribution.

A dataset with mixed data types for the input variables may require the selection of different types of data distributions for each variable. Using one of the three common distributions is not mandatory; for example, if a real-valued variable is known to have a different specific distribution, such as exponential, then that specific distribution may be used instead. If a real-valued variable does not have a well-defined distribution, such as bimodal or multimodal, then a kernel density estimator can be used to estimate the probability distribution instead. The Naive Bayes algorithm has proven effective and therefore is popular for text classification tasks. The words in a document may be encoded as binary (word present), count (word occurrence), or frequency (tf/idf) input vectors and binary, multinomial, or Gaussian probability distributions used respectively.

## 18.5 Worked Example of Naive Bayes

In this section, we will make the Naive Bayes calculation concrete with a small example on a machine learning dataset. We can generate a small contrived binary (2 class) classification problem using the `make_blobs()` function from the scikit-learn API. The example below generates 100 examples with two numerical input variables, each assigned one of two classes.

```
# example of generating a small classification dataset
from sklearn.datasets import make_blobs
# generate 2d classification dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# summarize
print(X.shape, y.shape)
print(X[:5])
print(y[:5])
```

Listing 18.1: Example of preparing a test dataset.

Running the example generates the dataset and summarizes the size, confirming the dataset was generated as expected.

**Note**: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

The input and output elements of the first five examples are also printed, showing that indeed, the two input variables are numeric and the class labels are either 0 or 1 for each example.

```
(100, 2) (100,)
[[-10.6105446   4.11045368]
 [  9.05798365  0.99701708]
```

```
 [  8.705727    1.36332954]
 [ -8.29324753  2.35371596]
 [  6.5954554   2.4247682 ]]
[0 1 1 0 1]
```

Listing 18.2: Example output from preparing a test dataset.

We will model the numerical input variables using a Gaussian probability distribution. This can be achieved using the norm SciPy API. First, the distribution can be constructed by specifying the parameters of the distribution, e.g. the mean and standard deviation, then the probability density function can be sampled for specific values using the `norm.pdf()` function. We can estimate the parameters of the distribution from the dataset using the `mean()` and `std()` NumPy functions. The `fit_distribution()` function below takes a sample of data for one variable and fits a data distribution.

```
# fit a probability distribution to a univariate data sample
def fit_distribution(data):
  # estimate parameters
  mu = mean(data)
  sigma = std(data)
  print(mu, sigma)
  # fit distribution
  dist = norm(mu, sigma)
  return dist
```

Listing 18.3: Example of a function for fitting a normal distribution to a sample.

Recall that we are interested in the conditional probability of each input variable. This means we need one distribution for each of the input variables, and one set of distributions for each of the class labels, or four distributions in total. First, we must split the data into groups of samples for each of the class labels.

```
...
# sort data into classes
Xy0 = X[y == 0]
Xy1 = X[y == 1]
print(Xy0.shape, Xy1.shape)
```

Listing 18.4: Example of splitting the training dataset by class value.

We can then use these groups to calculate the prior probabilities for a data sample belonging to each group. This will be 50% exactly given that we have created the same number of examples in each of the two classes; nevertheless, we will calculate these priors for completeness.

```
...
# calculate priors
priory0 = len(Xy0) / len(X)
priory1 = len(Xy1) / len(X)
print(priory0, priory1)
```

Listing 18.5: Example of calculating the prior for each class label.

Finally, we can call the `fit_distribution()` function that we defined to prepare a probability distribution for each variable, for each class label.

```
...
# create PDFs for y==0
```

```
X1y0 = fit_distribution(Xy0[:, 0])
X2y0 = fit_distribution(Xy0[:, 1])
# create PDFs for y==1
X1y1 = fit_distribution(Xy1[:, 0])
X2y1 = fit_distribution(Xy1[:, 1])
```

Listing 18.6: Example of fitting a normal distribution for each variable, split by class label.

Tying this all together, the complete probabilistic model of the dataset is listed below.

```
# summarize probability distributions of the dataset
from sklearn.datasets import make_blobs
from scipy.stats import norm
from numpy import mean
from numpy import std

# fit a probability distribution to a univariate data sample
def fit_distribution(data):
  # estimate parameters
  mu = mean(data)
  sigma = std(data)
  print(mu, sigma)
  # fit distribution
  dist = norm(mu, sigma)
  return dist

# generate 2d classification dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# sort data into classes
Xy0 = X[y == 0]
Xy1 = X[y == 1]
print(Xy0.shape, Xy1.shape)
# calculate priors
priory0 = len(Xy0) / len(X)
priory1 = len(Xy1) / len(X)
print(priory0, priory1)
# create PDFs for y==0
X1y0 = fit_distribution(Xy0[:, 0])
X2y0 = fit_distribution(Xy0[:, 1])
# create PDFs for y==1
X1y1 = fit_distribution(Xy1[:, 0])
X2y1 = fit_distribution(Xy1[:, 1])
```

Listing 18.7: Example of fitting distributions on the test dataset.

Running the example first splits the dataset into two groups for the two class labels and confirms the size of each group is even and the priors are 50%. Probability distributions are then prepared for each variable for each class label and the mean and standard deviation parameters of each distribution are reported, confirming that the distributions differ.

```
(50, 2) (50, 2)
0.5 0.5
-1.5632888906409914 0.787444265443213
4.426680361487157 0.958296071258367
-9.681177100524485 0.8943078901048118
-3.9713794295185845 0.9308177595208521
```

Listing 18.8: Example output from fitting distributions on the test dataset.

Next, we can use the prepared probabilistic model to make a prediction. The independent conditional probability for each class label can be calculated using the prior for the class (50%) and the conditional probability of the value for each variable. The `probability()` function below performs this calculation for one input example (array of two values) given the prior and conditional probability distribution for each variable. The value returned is a score rather than a probability as the quantity is not normalized, a simplification often performed when implementing Naive Bayes.

```
# calculate the independent conditional probability
def probability(X, prior, dist1, dist2):
  return prior * dist1.pdf(X[0]) * dist2.pdf(X[1])
```

Listing 18.9: Example of a function for calculating the probability of a sample belonging to a class.

We can use this function to calculate the probability for an example belonging to each class. First, we can select an example to be classified; in this case, the first example in the dataset.

```
...
py0 = probability(Xsample, priory0, distX1y0, distX2y0)
py1 = probability(Xsample, priory1, distX1y1, distX2y1)
print('P(y=0 | %s) = %.3f' % (Xsample, py0*100))
print('P(y=1 | %s) = %.3f' % (Xsample, py1*100))
```

Listing 18.10: Example of calculating the membership of a sample to each class.

The class with the largest score will be the resulting classification. Tying this together, the complete example of fitting the Naive Bayes model and using it to make one prediction is listed below.

```
# example of preparing and making a prediction with a naive bayes model
from sklearn.datasets import make_blobs
from scipy.stats import norm
from numpy import mean
from numpy import std

# fit a probability distribution to a univariate data sample
def fit_distribution(data):
  # estimate parameters
  mu = mean(data)
  sigma = std(data)
  print(mu, sigma)
  # fit distribution
  dist = norm(mu, sigma)
  return dist

# calculate the independent conditional probability
def probability(X, prior, dist1, dist2):
  return prior * dist1.pdf(X[0]) * dist2.pdf(X[1])

# generate 2d classification dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# sort data into classes
```

```
Xy0 = X[y == 0]
Xy1 = X[y == 1]
# calculate priors
priory0 = len(Xy0) / len(X)
priory1 = len(Xy1) / len(X)
# create PDFs for y==0
distX1y0 = fit_distribution(Xy0[:, 0])
distX2y0 = fit_distribution(Xy0[:, 1])
# create PDFs for y==1
distX1y1 = fit_distribution(Xy1[:, 0])
distX2y1 = fit_distribution(Xy1[:, 1])
# classify one example
Xsample, ysample = X[0], y[0]
py0 = probability(Xsample, priory0, distX1y0, distX2y0)
py1 = probability(Xsample, priory1, distX1y1, distX2y1)
print('P(y=0 | %s) = %.3f' % (Xsample, py0*100))
print('P(y=1 | %s) = %.3f' % (Xsample, py1*100))
print('Truth: y=%d' % ysample)
```

Listing 18.11: Example of the Naive Bayes algorithm from scratch.

Running the example first prepares the prior and conditional probabilities as before, then uses them to make a prediction for one example. The score of the example belonging to $y = 0$ is about 0.3 (recall this is an unnormalized probability), whereas the score of the example belonging to $y = 1$ is 0.0. Therefore, we would classify the example as belonging to $y = 0$. In this case, the true or actual outcome is known, $y = 0$, which matches the prediction by our Naive Bayes model.

```
P(y=0 | [-0.79415228 2.10495117]) = 0.348
P(y=1 | [-0.79415228 2.10495117]) = 0.000
Truth: y=0
```

Listing 18.12: Example output from evaluating the Naive Bayes algorithm from scratch.

In practice, it is a good idea to use optimized implementations of the Naive Bayes algorithm. The scikit-learn library provides three implementations, one for each of the three main probability distributions; for example, `BernoulliNB`, `MultinomialNB`, and `GaussianNB` for binomial, multinomial and Gaussian distributed input variables respectively. To use a scikit-learn Naive Bayes model, first the model is defined, then it is fit on the training dataset. Once fit, probabilities can be predicted via the `predict_proba()` function and class labels can be predicted directly via the `predict()` function. The complete example of fitting a Gaussian Naive Bayes model (`GaussianNB`) to the same test dataset is listed below.

```
# example of gaussian naive bayes
from sklearn.datasets import make_blobs
from sklearn.naive_bayes import GaussianNB
# generate 2d classification dataset
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
# define the model
model = GaussianNB()
# fit the model
model.fit(X, y)
# select a single sample
Xsample, ysample = [X[0]], y[0]
# make a probabilistic prediction
```

```
yhat_prob = model.predict_proba(Xsample)
print('Predicted Probabilities: ', yhat_prob)
# make a classification prediction
yhat_class = model.predict(Xsample)
print('Predicted Class: ', yhat_class)
print('Truth: y=%d' % ysample)
```

Listing 18.13: Example of the Naive Bayes using the scikit-learn library.

Running the example fits the model on the training dataset, then makes predictions for the same first example that we used in the prior example. In this case, the probability of the example belonging to $y = 0$ is 1.0 or a certainty. The probability of $y = 1$ is a very small value close to 0.0. Note that these results will differ from our example as this model will return probabilities rather than proportional scores (unnormalized probabilities). Finally, the class label is predicted directly, again matching the ground truth for the example.

```
Predicted Probabilities: [[1.00000000e+00 5.52387327e-30]]
Predicted Class: [0]
Truth: y=0
```

Listing 18.14: Example output from the Naive Bayes using the scikit-learn library.

## 18.6 5 Tips When Using Naive Bayes

This section lists some practical tips when working with Naive Bayes models.

### 18.6.1 Use a KDE for Complex Distributions

If the probability distribution for a variable is complex or unknown, it can be a good idea to use a kernel density estimator or KDE to approximate the distribution directly from the data samples. A good example would be the Gaussian KDE.

### 18.6.2 Decreased Performance With Increasing Variable Dependence

By definition, Naive Bayes assumes the input variables are independent of each other. This works well most of the time, even when some or most of the variables are in fact dependent. Nevertheless, the performance of the algorithm degrades the more dependent the input variables happen to be.

### 18.6.3 Avoid Numerical Underflow with Log

The calculation of the independent conditional probability for one example for one class label involves multiplying many probabilities together, one for the class and one for each input variable. As such, the multiplication of many small numbers together can become numerically unstable, especially as the number of input variables increases. To overcome this problem, it is common to change the calculation from the product of probabilities to the sum of log probabilities. For example:

$$P(y_i|x_1, x_2, \cdots, x_n) = \log(P(x_1|y_i)) + \log(P(x_2|y_i)) + \cdots + \log(P(x_n|y_i)) + \log(P(y_i)) \quad (18.7)$$

Calculating the natural logarithm of probabilities has the effect of creating larger (negative) numbers and adding the numbers together will mean that larger probabilities will be closer to zero. The resulting values can still be compared and maximized to give the most likely class label. This is often called the log-trick when multiplying probabilities.

### 18.6.4 Update Probability Distributions

As new data becomes available, it can be relatively straightforward to use this new data with the old data to update the estimates of the parameters for each variable's probability distribution. This allows the model to easily make use of new data or the changing distributions of data over time.

### 18.6.5 Use as a Generative Model

The probability distributions will summarize the conditional probability of each input variable value for each class label. These probability distributions can be useful more generally beyond use in a classification model. For example, the prepared probability distributions can be randomly sampled in order to create new plausible data instances. The conditional independence assumption may mean that the examples are more or less plausible based on how much actual interdependence exists between the input variables in the dataset.

## 18.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 18.7.1 Books

- *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2lnW5S7

### 18.7.2 API

- `sklearn.datasets.make_blobs` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html
- `scipy.stats.norm` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html

- Naive Bayes, scikit-learn documentation.
  https://scikit-learn.org/stable/modules/naive_bayes.html
- sklearn.naive_bayes.GaussianNB API
  https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

### 18.7.3 Articles

- Bayes' theorem, Wikipedia.
  https://en.wikipedia.org/wiki/Bayes%27_theorem
- Naive Bayes classifier, Wikipedia.
  https://en.wikipedia.org/wiki/Naive_Bayes_classifier
- Maximum a posteriori estimation, Wikipedia.
  https://en.wikipedia.org/wiki/Maximum_a_posteriori_estimation

## 18.8 Summary

In this tutorial, you discovered the Naive Bayes algorithm for classification predictive modeling. Specifically, you learned:

- How to frame classification predictive modeling as a conditional probability model.
- How to use Bayes Theorem to solve the conditional probability model of classification.
- How to implement simplified Bayes Theorem for classification called the Naive Bayes algorithm.

### 18.8.1 Next

In the next tutorial, you will discover the Bayesian Optimization algorithm.

# Chapter 19

# How to Implement Bayesian Optimization

Global optimization is a challenging problem of finding an input vector that results in the minimum or maximum cost of a given objective function. Typically, the form of the objective function is complex and intractable to analyze and is often non-convex, nonlinear, high dimensional, noisy, and computationally expensive to evaluate. Bayesian Optimization provides a principled technique based on Bayes Theorem to direct a search of a global optimization problem that is efficient and effective. It works by building a probabilistic model of the objective function, called the surrogate function, that is then searched efficiently with an acquisition function before candidate samples are chosen for evaluation on the real objective function. Bayesian Optimization is often used in applied machine learning to tune the hyperparameters of a given well-performing model on a validation dataset. In this tutorial, you will discover Bayesian Optimization for directed search of complex optimization problems. After completing this tutorial, you will know:

- Global optimization is a challenging problem that involves black box and often non-convex, nonlinear, noisy, and computationally expensive objective functions.
- Bayesian Optimization provides a probabilistically principled method for global optimization.
- How to implement Bayesian Optimization from scratch and how to use open-source implementations.

Let's get started.

## 19.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Challenge of Function Optimization
2. What Is Bayesian Optimization
3. How to Perform Bayesian Optimization
4. Hyperparameter Tuning With Bayesian Optimization

## 19.2 Challenge of Function Optimization

Global function optimization, or function optimization for short, involves finding the minimum or maximum of an objective function. Samples are drawn from the domain and evaluated by the objective function to give a score or cost. Let's define some common terms:

- **Samples**. One example from the domain, represented as a vector.
- **Search Space**: Extent of the domain from which samples can be drawn.
- **Objective Function**. Function that takes a sample and returns a cost.
- **Cost**. Numeric score for a sample calculated via the objective function.

Samples are comprised of one or more variables generally easy to devise or create. One sample is often defined as a vector of variables with a predefined range in an n-dimensional space. This space must be sampled and explored in order to find the specific combination of variable values that result in the best cost. The cost often has units that are specific to a given domain. Optimization is often described in terms of minimizing cost, as a maximization problem can easily be transformed into a minimization problem by inverting the calculated cost. Together, the minimum and maximum of a function are referred to as the extremes of the function (or the plural extrema).

The objective function is often easy to specify but can be computationally challenging to calculate or result in a noisy calculation of cost over time. The form of the objective function is unknown and is often highly nonlinear, and highly multi-dimensional defined by the number of input variables. The function is also probably non-convex. This means that local extrema may or may not be the global extrema (e.g. could be misleading and result in premature convergence), hence the name of the task as global rather than local optimization. Although little is known about the objective function, (it is known whether the minimum or the maximum cost from the function is sought), and as such, it is often referred to as a black box function and the search process as black box optimization. Further, the objective function is sometimes called an oracle given the ability to only give answers.

Function optimization is a fundamental part of machine learning. Most machine learning algorithms involve the optimization of parameters (weights, coefficients, etc.) in response to training data. Optimization also refers to the process of finding the best set of hyperparameters that configure the training of a machine learning algorithm. Taking one step higher again, the selection of training data, data preparation, and machine learning algorithms themselves is also a problem of function optimization. Summary of optimization in machine learning:

- **Algorithm Training**. Optimization of model parameters.
- **Algorithm Tuning**. Optimization of model hyperparameters.
- **Predictive Modeling**. Optimization of data, data preparation, and algorithm selection.

Many methods exist for function optimization, such as randomly sampling the variable search space, called random search, or systematically evaluating samples in a grid across the search space, called grid search. More principled methods are able to learn from sampling the space so that future samples are directed toward the parts of the search space that are most likely to contain the extrema. A directed approach to global optimization that uses probability is called Bayesian Optimization.

## 19.3 What Is Bayesian Optimization

Bayesian Optimization is an approach that uses Bayes Theorem to direct the search in order to find the minimum or maximum of an objective function. It is an approach that is most useful for objective functions that are complex, noisy, and/or expensive to evaluate.

> Bayesian optimization is a powerful strategy for finding the extrema of objective functions that are expensive to evaluate. [...] It is particularly useful when these evaluations are costly, when one does not have access to derivatives, or when the problem at hand is non-convex.
>
> — *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*, 2010.

Recall that Bayes Theorem is an approach for calculating the conditional probability of an event:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)} \tag{19.1}$$

We can simplify this calculation by removing the normalizing value of $P(B)$ and describe the conditional probability as a proportional quantity. This is useful as we are not interested in calculating a specific conditional probability, but instead in optimizing a quantity.

$$P(A|B) = P(B|A) \times P(A) \tag{19.2}$$

The conditional probability that we are calculating is referred to generally as the posterior probability; the reverse conditional probability is sometimes referred to as the likelihood, and the marginal probability is referred to as the prior probability; for example:

$$\text{posterior} = \text{likelihood} \times \text{prior} \tag{19.3}$$

This provides a framework that can be used to quantify the beliefs about an unknown objective function given samples from the domain and their evaluation via the objective function. We can devise specific samples $(x_1, x_2, \cdots, x_n)$ and evaluate them using the objective function $f(x_i)$ that returns the cost or outcome for the sample $x_i$. Samples and their outcome are collected sequentially and define our data $D$, e.g. $D = \{x_i, f(x_i), ..., x_n, f(x_n)\}$ and is used to define the prior. The likelihood function is defined as the probability of observing the data given the function $P(D|f)$. This likelihood function will change as more observations are collected.

$$P(f|D) = P(D|f) \times P(f) \tag{19.4}$$

The posterior represents everything we know about the objective function. It is an approximation of the objective function and can be used to estimate the cost of different candidate samples that we may want to evaluate. In this way, the posterior probability is a surrogate objective function.

> The posterior captures the updated beliefs about the unknown objective function. One may also interpret this step of Bayesian optimization as estimating the objective function with a surrogate function (also called a response surface).

> — *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*, 2010.

- **Surrogate Function**: Bayesian approximation of the objective function that can be sampled efficiently.

The surrogate function gives us an estimate of the objective function, which can be used to direct future sampling. Sampling involves careful use of the posterior in a function known as the *acquisition* function, e.g. for acquiring more samples. We want to use our belief about the objective function to sample the area of the search space that is most likely to pay off, therefore the acquisition will optimize the conditional probability of locations in the search to generate the next sample.

- **Acquisition Function**: Technique by which the posterior is used to select the next sample from the search space.

Once additional samples and their evaluation via the objective function $f()$ have been collected, they are added to data $D$ and the posterior is then updated. This process is repeated until the extrema of the objective function is located, a good enough result is located, or resources are exhausted. The Bayesian Optimization algorithm can be summarized as follows:

1. Select a Sample by Optimizing the Acquisition Function.
2. Evaluate the Sample With the Objective Function.
3. Update the Data and, in turn, the Surrogate Function.
4. Go To 1.

## 19.4 How to Perform Bayesian Optimization

In this section, we will explore how Bayesian Optimization works by developing an implementation from scratch for a simple one-dimensional test function. First, we will define the test problem, then how to model the mapping of inputs to outputs with a surrogate function. Next, we will see how the surrogate function can be searched efficiently with an acquisition function before tying all of these elements together into the Bayesian Optimization procedure.

### 19.4.1 Test Problem

The first step is to define a test problem. We will use a multimodal problem with five peaks, calculated as:

$$y = x^2 \times \sin(5 \times \pi \times x)^6 \tag{19.5}$$

Where $x$ is a real value in the range [0,1] and $\pi$ is the value of pi. We will augment this function by adding Gaussian noise with a mean of zero and a standard deviation of 0.1. This will mean that the real evaluation will have a positive or negative random value added to it, making the function challenging to optimize. The `objective()` function below implements this.

```
# objective function
def objective(x, noise=0.1):
  noise = normal(loc=0, scale=noise)
  return (x**2 * sin(5 * pi * x)**6.0) + noise
```

Listing 19.1: Example of an objective function.

We can test this function by first defining a grid-based sample of inputs from 0 to 1 with a step size of 0.01 across the domain.

```
...
# grid-based sample of the domain [0,1]
X = arange(0, 1, 0.01)
```

Listing 19.2: Example of a grid across the input domain.

We can then evaluate these samples using the objective function without any noise to see what the real objective function looks like.

```
...
# sample the domain without noise
y = [objective(x, 0) for x in X]
```

Listing 19.3: Example of a evaluating samples across the input domain.

We can then evaluate these same points with noise to see what the objective function will look like when we are optimizing it.

```
...
# sample the domain with noise
ynoise = [objective(x) for x in X]
```

Listing 19.4: Example of a evaluating samples across the input domain with noise.

We can look at all of the non-noisy objective function values to find the input that resulted in the best score and report it. This will be the optima, in this case, maxima, as we are maximizing the output of the objective function. We would not know this in practice, but for our test problem, it is good to know the real best input and output of the function to see if the Bayesian Optimization algorithm can locate it.

```
...
# find best result
ix = argmax(y)
print('Optima: x=%.3f, y=%.3f' % (X[ix], y[ix]))
```

Listing 19.5: Example of a summarizing the function optima.

Finally, we can create a plot, first showing the noisy evaluation as a scatter plot with input on the x-axis and score on the y-axis, then a line plot of the scores without any noise.

```
...
# plot the points with noise
pyplot.scatter(X, ynoise)
# plot the points without noise
pyplot.plot(X, y)
# show the plot
pyplot.show()
```

Listing 19.6: Example of plotting the samples.

The complete example of reviewing the test function that we wish to optimize is listed below.

```
# example of the test problem
from math import sin
from math import pi
from numpy import arange
from numpy import argmax
from numpy.random import normal
from matplotlib import pyplot

# objective function
def objective(x, noise=0.1):
  noise = normal(loc=0, scale=noise)
  return (x**2 * sin(5 * pi * x)**6.0) + noise

# grid-based sample of the domain [0,1]
X = arange(0, 1, 0.01)
# sample the domain without noise
y = [objective(x, 0) for x in X]
# sample the domain with noise
ynoise = [objective(x) for x in X]
# find best result
ix = argmax(y)
print('Optima: x=%.3f, y=%.3f' % (X[ix], y[ix]))
# plot the points with noise
pyplot.scatter(X, ynoise)
# plot the points without noise
pyplot.plot(X, y)
# show the plot
pyplot.show()
```

Listing 19.7: Example plotting samples of the objective function.

Running the example first reports the global optima as an input with the value 0.9 that gives the score 0.81.

```
Optima: x=0.900, y=0.810
```

Listing 19.8: Example output from the optima from sampling the objective function.

A plot is then created showing the noisy evaluation of the samples (dots) and the non-noisy and true shape of the objective function (line). Your specific dots will differ given the stochastic nature of the noisy objective function.

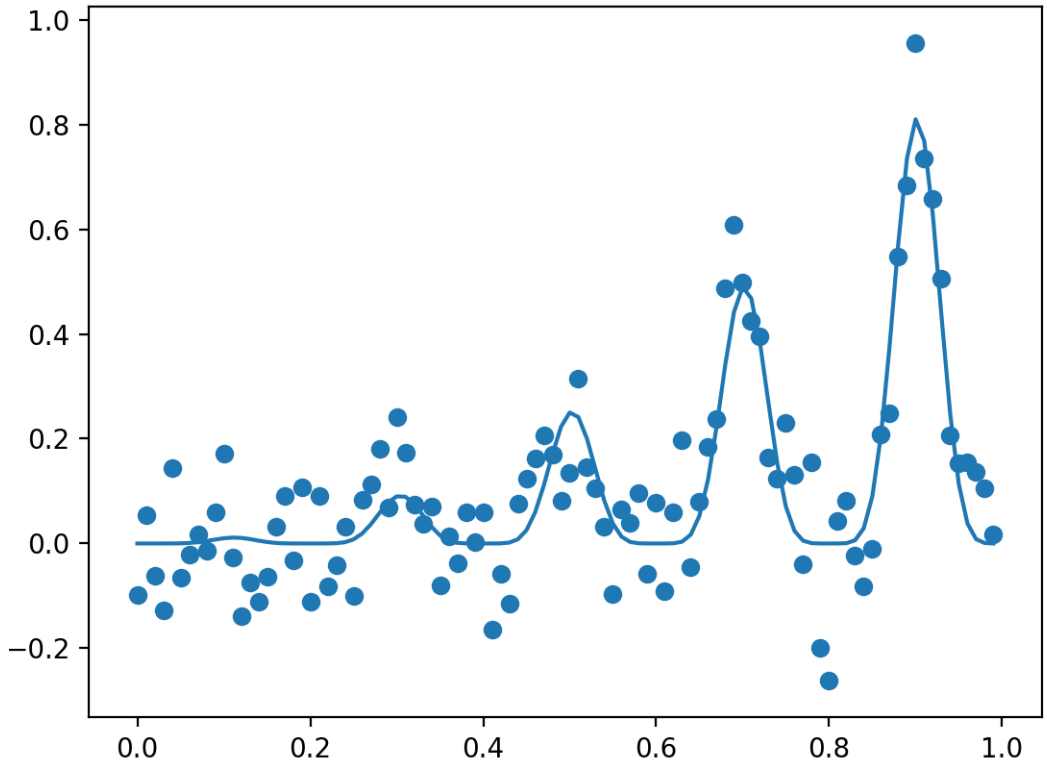


Figure 19.1: Plot of The Input Samples Evaluated with a Noisy (dots) and Non-Noisy (Line) Objective Function.

Now that we have a test problem, let's review how to train a surrogate function.

### 19.4.2 Surrogate Function

The surrogate function is a technique used to best approximate the mapping of input examples to an output score. Probabilistically, it summarizes the conditional probability of an objective function ($f$), given the available data ($D$) or $P(f|D)$. A number of techniques can be used for this, although the most popular is to treat the problem as a regression predictive modeling problem with the data representing the input and the score representing the output to the model. This is often best modeled using a random forest or a Gaussian Process. A Gaussian Process, or GP, is a model that constructs a joint probability distribution over the variables, assuming a multivariate Gaussian distribution. As such, it is capable of efficient and effective summarization of a large number of functions and smooth transition as more observations are made available to the model.

This smooth structure and smooth transition to new functions based on data are desirable properties as we sample the domain, and the multivariate Gaussian basis to the model means that an estimate from the model will be a mean of a distribution with a standard deviation; that will be helpful later in the acquisition function. As such, using a GP regression model is often preferred.

We can fit a GP regression model using the `GaussianProcessRegressor` scikit-learn implementation from a sample of inputs ($X$) and noisy evaluations from the objective function ($y$). First, the model must be defined. An important aspect in defining the GP model is the kernel. This controls the shape of the function at specific points based on distance measures between actual data observations. Many different kernel functions can be used, and some may offer better performance for specific datasets. By default, a Radial Basis Function, or RBF, is used that can work well.

```
...
# define the model
model = GaussianProcessRegressor()
```

Listing 19.9: Example defining the Gaussian Process model.

Once defined, the model can be fit on the training dataset directly by calling the `fit()` function. The defined model can be fit again at any time with updated data concatenated to the existing data by another call to `fit()`.

```
...
# fit the model
model.fit(X, y)
```

Listing 19.10: Example fitting the Gaussian Process model.

The model will estimate the cost for one or more samples provided to it. The model is used by calling the `predict()` function. The result for a given sample will be a mean of the distribution at that point. We can also get the standard deviation of the distribution at that point in the function by specifying the argument `return_std=True`; for example:

```
...
yhat = model.predict(X, return_std=True)
```

Listing 19.11: Example making predictions with the Gaussian Process model.

This function can result in warnings if the distribution is thin at a given point we are interested in sampling. Therefore, we can silence all of the warnings when making a prediction. The `surrogate()` function below takes the fit model and one or more samples and returns the mean and standard deviation estimated costs whilst not printing any warnings.

```
# surrogate or approximation for the objective function
def surrogate(model, X):
  # catch any warning generated when making a prediction
  with catch_warnings():
    # ignore generated warnings
    simplefilter("ignore")
    return model.predict(X, return_std=True)
```

Listing 19.12: Example of the surrogate function for approximating the objective function.

We can call this function any time to estimate the cost of one or more samples, such as when we want to optimize the acquisition function in the next section. For now, it is interesting to see what the surrogate function looks like across the domain after it is trained on a random sample. We can achieve this by first fitting the GP model on a random sample of 100 data points and their real objective function values with noise. We can then plot a scatter plot of these points. Next, we can perform a grid-based sample across the input domain and estimate

the cost at each point using the surrogate function and plot the result as a line. We would expect the surrogate function to have a crude approximation of the true non-noisy objective function. The `plot()` function below creates this plot, given the random data sample of the real noisy objective function and the fit model.

```
# plot real observations vs surrogate function
def plot(X, y, model):
  # scatter plot of inputs and real objective function
  pyplot.scatter(X, y)
  # line plot of surrogate function across domain
  Xsamples = asarray(arange(0, 1, 0.001))
  Xsamples = Xsamples.reshape(len(Xsamples), 1)
  ysamples, _ = surrogate(model, Xsamples)
  pyplot.plot(Xsamples, ysamples)
  # show the plot
  pyplot.show()
```

Listing 19.13: Example of a function that plots real vs surrogate evaluation of input samples.

Tying this together, the complete example of fitting a Gaussian Process regression model on noisy samples and plotting the sample vs. the surrogate function is listed below.

```
# example of a gaussian process surrogate function
from math import sin
from math import pi
from numpy import arange
from numpy import asarray
from numpy.random import normal
from numpy.random import random
from matplotlib import pyplot
from warnings import catch_warnings
from warnings import simplefilter
from sklearn.gaussian_process import GaussianProcessRegressor

# objective function
def objective(x, noise=0.1):
  noise = normal(loc=0, scale=noise)
  return (x**2 * sin(5 * pi * x)**6.0) + noise

# surrogate or approximation for the objective function
def surrogate(model, X):
  # catch any warning generated when making a prediction
  with catch_warnings():
    # ignore generated warnings
    simplefilter("ignore")
    return model.predict(X, return_std=True)

# plot real observations vs surrogate function
def plot(X, y, model):
  # scatter plot of inputs and real objective function
  pyplot.scatter(X, y)
  # line plot of surrogate function across domain
  Xsamples = asarray(arange(0, 1, 0.001))
  Xsamples = Xsamples.reshape(len(Xsamples), 1)
  ysamples, _ = surrogate(model, Xsamples)
  pyplot.plot(Xsamples, ysamples)
```

```
  # show the plot
  pyplot.show()

# sample the domain sparsely with noise
X = random(100)
y = asarray([objective(x) for x in X])
# reshape into rows and cols
X = X.reshape(len(X), 1)
y = y.reshape(len(y), 1)
# define the model
model = GaussianProcessRegressor()
# fit the model
model.fit(X, y)
# plot the surrogate function
plot(X, y, model)
```

Listing 19.14: Example of sampling the surrogate function and plotting the results.

Running the example first draws the random sample, evaluates it with the noisy objective function, then fits the GP model. The data sample and a grid of points across the domain evaluated via the surrogate function are then plotted as dots and a line respectively. Your specific results will vary given the stochastic nature of the data sample. Consider running the example a few times. In this case, as we expected, the plot resembles a crude version of the underlying non-noisy objective function, importantly with a peak around 0.9 where we know the true maxima is located.

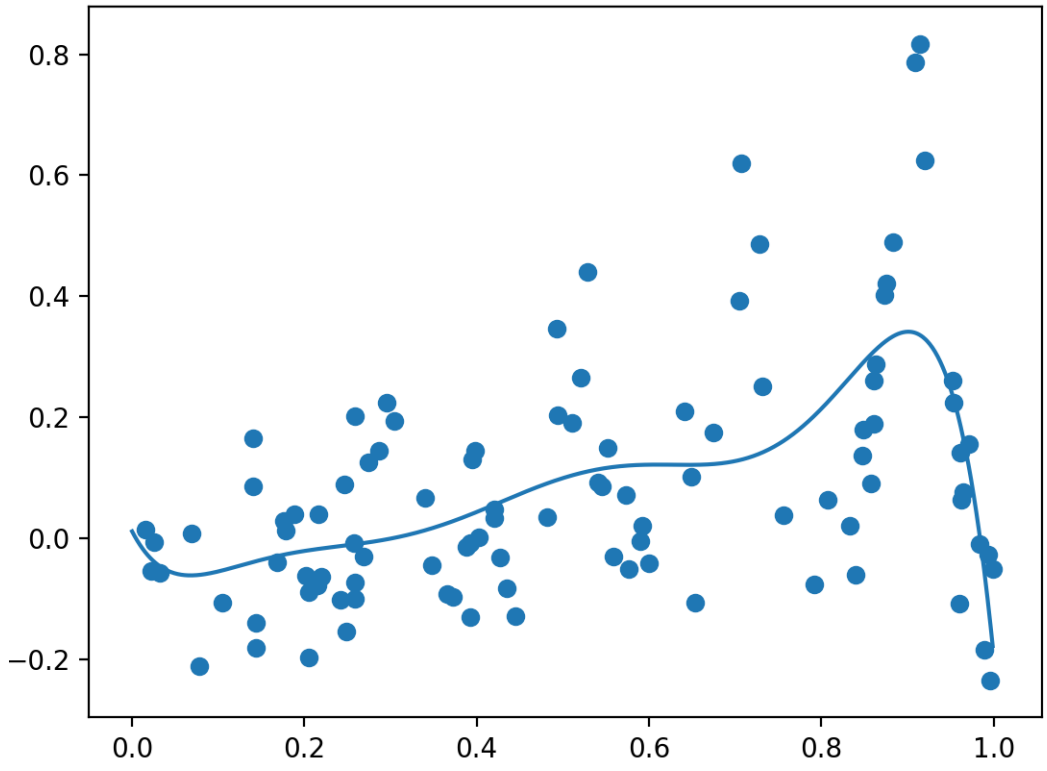


Figure 19.2: Plot Showing Random Sample With Noisy Evaluation (dots) and Surrogate Function Across the Domain (line).

Next, we must define a strategy for sampling the surrogate function.

### 19.4.3 Acquisition Function

The surrogate function is used to test a range of candidate samples in the domain. From these results, one or more candidates can be selected and evaluated with the real, and in normal practice, computationally expensive cost function. This involves two pieces: the search strategy used to navigate the domain in response to the surrogate function and the acquisition function that is used to interpret and score the response from the surrogate function. A simple search strategy, such as a random sample or grid-based sample, can be used, although it is more common to use a local search strategy, such as the popular BFGS algorithm. In this case, we will use a random search or random sample of the domain in order to keep the example simple. This involves first drawing a random sample of candidate samples from the domain, evaluating them with the acquisition function, then maximizing the acquisition function or choosing the candidate sample that gives the best score. The `opt_acquisition()` function below implements this.

```
# optimize the acquisition function
def opt_acquisition(X, y, model):
  # random search, generate random samples
```

```
  Xsamples = random(100)
  Xsamples = Xsamples.reshape(len(Xsamples), 1)
  # calculate the acquisition function for each sample
  scores = acquisition(X, Xsamples, model)
  # locate the index of the largest scores
  ix = argmax(scores)
  return Xsamples[ix, 0]
```

Listing 19.15: Example of a function for optimizing the acquisition function.

The acquisition function is responsible for scoring or estimating the likelihood that a given candidate sample (input) is worth evaluating with the real objective function. We could just use the surrogate score directly. Alternately, given that we have chosen a Gaussian Process model as the surrogate function, we can use the probabilistic information from this model in the acquisition function to calculate the probability that a given sample is worth evaluating. There are many different types of probabilistic acquisition functions that can be used, each providing a different trade-off for how exploitative (greedy) and explorative they are.

Three common examples include:

- Probability of Improvement (PI).
- Expected Improvement (EI).
- Lower Confidence Bound (LCB).

The Probability of Improvement method is the simplest, whereas the Expected Improvement method is the most commonly used. In this case, we will use the simpler Probability of Improvement method, which is calculated as the normal cumulative probability of the normalized expected improvement, calculated as follows:

$$PI = cdf(\frac{\mu - \mu_{best}}{\sigma}) \tag{19.6}$$

Where PI is the probability of improvement, $cdf()$ is the normal cumulative distribution function, $\mu$ is the mean of the surrogate function for a given sample $x$, $\sigma$ is the standard deviation of the surrogate function for a given sample $x$, and $\mu_{best}$ is the mean of the surrogate function for the best sample found so far. We can add a very small number to the standard deviation to avoid a divide by zero error. The `acquisition()` function below implements this given the current training dataset of input samples, an array of new candidate samples, and the fit GP model.

```
# probability of improvement acquisition function
def acquisition(X, Xsamples, model):
  # calculate the best surrogate score found so far
  yhat, _ = surrogate(model, X)
  best = max(yhat)
  # calculate mean and stdev via surrogate function
  mu, std = surrogate(model, Xsamples)
  mu = mu[:, 0]
  # calculate the probability of improvement
  probs = norm.cdf((mu - best) / (std+1E-9))
  return probs
```

Listing 19.16: Example of a probability of improvement acquisition function.

### 19.4.4 Complete Bayesian Optimization Algorithm

We can tie all of this together into the Bayesian Optimization algorithm. The main algorithm involves cycles of selecting candidate samples, evaluating them with the objective function, then updating the GP model.

```
...
# perform the optimization process
for i in range(100):
  # select the next point to sample
  x = opt_acquisition(X, y, model)
  # sample the point
  actual = objective(x)
  # summarize the finding for our own reporting
  est, _ = surrogate(model, [[x]])
  print('>x=%.3f, f()=%3f, actual=%.3f' % (x, est, actual))
  # add the data to the dataset
  X = vstack((X, [[x]]))
  y = vstack((y, [[actual]]))
  # update the model
  model.fit(X, y)
```

Listing 19.17: Example of the Bayesian Optimization algorithm loop.

The complete example is listed below.

```
# example of bayesian optimization for a 1d function from scratch
from math import sin
from math import pi
from numpy import arange
from numpy import vstack
from numpy import argmax
from numpy import asarray
from numpy.random import normal
from numpy.random import random
from scipy.stats import norm
from sklearn.gaussian_process import GaussianProcessRegressor
from warnings import catch_warnings
from warnings import simplefilter
from matplotlib import pyplot

# objective function
def objective(x, noise=0.1):
  noise = normal(loc=0, scale=noise)
  return (x**2 * sin(5 * pi * x)**6.0) + noise

# surrogate or approximation for the objective function
def surrogate(model, X):
  # catch any warning generated when making a prediction
  with catch_warnings():
    # ignore generated warnings
    simplefilter("ignore")
    return model.predict(X, return_std=True)

# probability of improvement acquisition function
def acquisition(X, Xsamples, model):
  # calculate the best surrogate score found so far
```

```
  yhat, _ = surrogate(model, X)
  best = max(yhat)
  # calculate mean and stdev via surrogate function
  mu, std = surrogate(model, Xsamples)
  mu = mu[:, 0]
  # calculate the probability of improvement
  probs = norm.cdf((mu - best) / (std+1E-9))
  return probs

# optimize the acquisition function
def opt_acquisition(X, y, model):
  # random search, generate random samples
  Xsamples = random(100)
  Xsamples = Xsamples.reshape(len(Xsamples), 1)
  # calculate the acquisition function for each sample
  scores = acquisition(X, Xsamples, model)
  # locate the index of the largest scores
  ix = argmax(scores)
  return Xsamples[ix, 0]

# plot real observations vs surrogate function
def plot(X, y, model):
  # scatter plot of inputs and real objective function
  pyplot.scatter(X, y)
  # line plot of surrogate function across domain
  Xsamples = asarray(arange(0, 1, 0.001))
  Xsamples = Xsamples.reshape(len(Xsamples), 1)
  ysamples, _ = surrogate(model, Xsamples)
  pyplot.plot(Xsamples, ysamples)
  # show the plot
  pyplot.show()

# sample the domain sparsely with noise
X = random(100)
y = asarray([objective(x) for x in X])
# reshape into rows and cols
X = X.reshape(len(X), 1)
y = y.reshape(len(y), 1)
# define the model
model = GaussianProcessRegressor()
# fit the model
model.fit(X, y)
# plot before hand
plot(X, y, model)
# perform the optimization process
for i in range(100):
  # select the next point to sample
  x = opt_acquisition(X, y, model)
  # sample the point
  actual = objective(x)
  # summarize the finding
  est, _ = surrogate(model, [[x]])
  print('>x=%.3f, f()=%3f, actual=%.3f' % (x, est, actual))
  # add the data to the dataset
  X = vstack((X, [[x]]))
  y = vstack((y, [[actual]]))
```

```
  # update the model
  model.fit(X, y)

# plot all samples and the final surrogate function
plot(X, y, model)
# best result
ix = argmax(y)
print('Best Result: x=%.3f, y=%.3f' % (X[ix], y[ix]))
```

Listing 19.18: Examples of Bayesian Optimization from scratch.

Running the example first creates an initial random sample of the search space and evaluation of the results. Then a GP model is fit on this data. Your specific results will vary given the stochastic nature of the sampling of the domain. Try running the example a few times. A plot is created showing the raw observations as dots and the surrogate function across the entire domain. In this case, the initial sample has a good spread across the domain and the surrogate function has a bias towards the part of the domain where we know the optima is located.

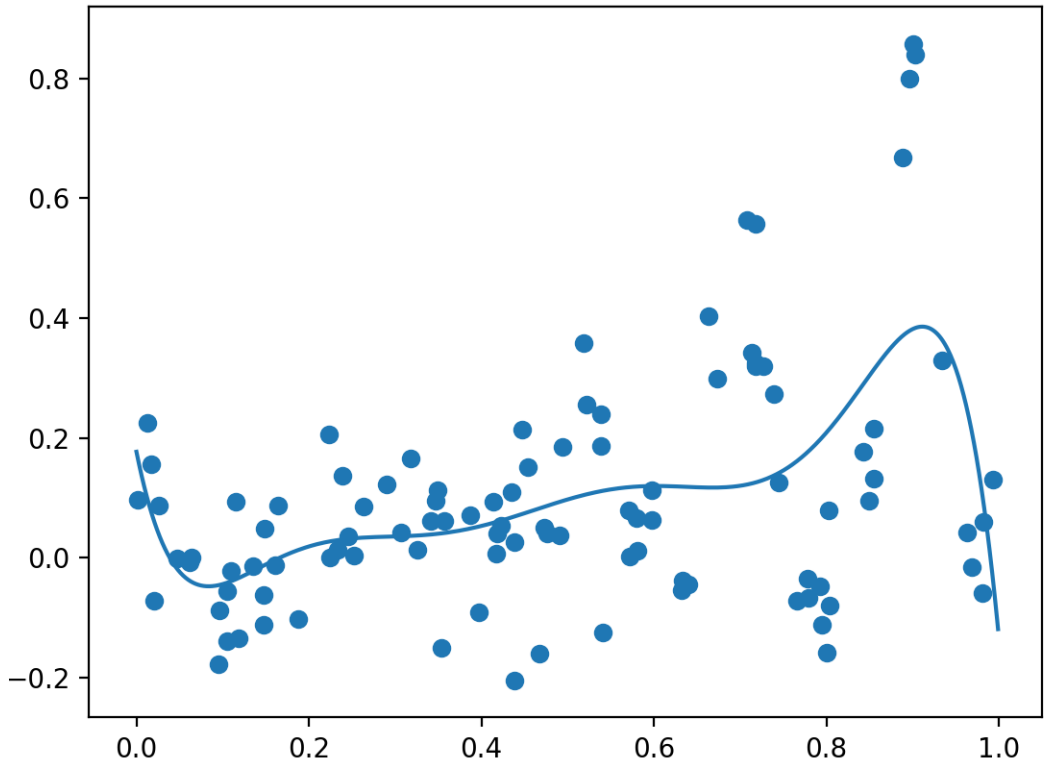


Figure 19.3: Plot of Initial Sample (dots) and Surrogate Function Across the Domain (line).

The algorithm then iterates for 100 cycles, selecting samples, evaluating them, and adding them to the dataset to update the surrogate function, and over again. Each cycle reports the selected input value, the estimated score from the surrogate function, and the actual score.

Ideally, these scores would get closer and closer as the algorithm converges on one area of the search space.

```
...
>x=0.922, f()=0.661501, actual=0.682
>x=0.895, f()=0.661668, actual=0.905
>x=0.928, f()=0.648008, actual=0.403
>x=0.908, f()=0.674864, actual=0.750
>x=0.436, f()=0.071377, actual=-0.115
```

Listing 19.19: Example output from running the Bayesian Optimization algorithm from scratch.

Next, a final plot is created with the same form as the prior plot. This time, all 200 samples evaluated during the optimization task are plotted. We would expect an overabundance of sampling around the known optima, and this is what we see, with many dots around 0.9. We also see that the surrogate function has a stronger representation of the underlying target domain.

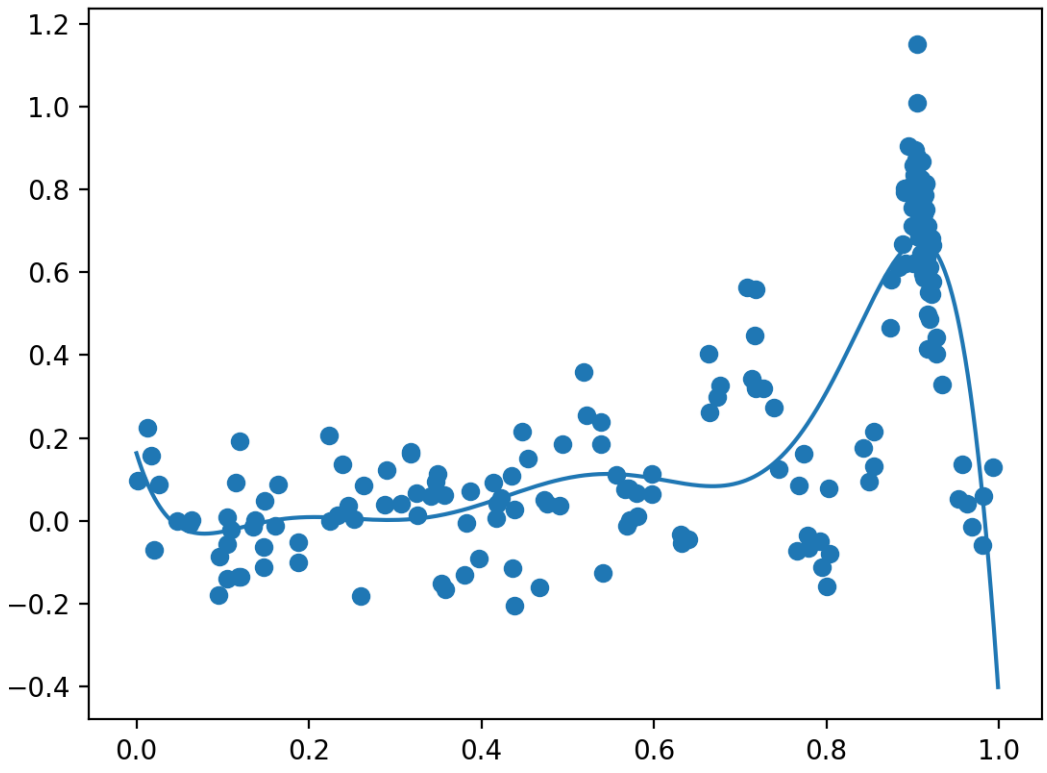


Figure 19.4: Plot of All Samples (dots) and Surrogate Function Across the Domain (line) after Bayesian Optimization.

Finally, the best input and its objective function score are reported. We know the optima has an input of 0.9 and an output of 0.810 if there was no sampling noise. Given the sampling noise, the optimization algorithm gets close in this case, suggesting an input of 0.905.

```
Best Result: x=0.905, y=1.150
```

Listing 19.20: Example final output from running the Bayesian Optimization algorithm from scratch.

## 19.5 Hyperparameter Tuning With Bayesian Optimization

It can be a useful exercise to implement Bayesian Optimization to learn how it works. In practice, when using Bayesian Optimization on a project, it is a good idea to use a standard implementation provided in an open-source library. This is to both avoid bugs and to leverage a wider range of configuration options and speed improvements. Two popular libraries for Bayesian Optimization include Scikit-Optimize and HyperOpt. In machine learning, these libraries are often used to tune the hyperparameters of algorithms. Hyperparameter tuning is a good fit for Bayesian Optimization because the evaluation function is computationally expensive (e.g. training models for each set of hyperparameters) and noisy (e.g. noise in training data and stochastic learning algorithms).

In this section, we will take a brief look at how to use the Scikit-Optimize library to optimize the hyperparameters of a $k$-nearest neighbor classifier for a simple test classification problem. This will provide a useful template that you can use on your own projects. The Scikit-Optimize project is designed to provide access to Bayesian Optimization for applications that use SciPy and NumPy, or applications that use scikit-learn machine learning algorithms. First, the library must be installed, which can be achieved easily using pip; for example:

```
sudo pip install scikit-optimize
```

Listing 19.21: Example of installing the scikit-optimize library.

It is also assumed that you have scikit-learn installed for this example. Once installed, there are two ways that scikit-optimize can be used to optimize the hyperparameters of a scikit-learn algorithm. The first is to perform the optimization directly on a search space, and the second is to use the BayesSearchCV class, a sibling of the scikit-learn native classes for random and grid searching. In this example, will use the simpler approach of optimizing the hyperparameters directly. The first step is to prepare the data and define the model. We will use a simple test classification problem via the `make_blobs()` function with 500 examples, each with two features and three class labels. We will then use a `KNeighborsClassifier` algorithm.

```
...
# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2)
# define the model
model = KNeighborsClassifier()
```

Listing 19.22: Example of defining the test problem and base classifier.

Next, we must define the search space. In this case, we will tune the number of neighbors (`n_neighbors`) and the shape of the neighborhood function (`p`). This requires ranges be defined for a given data type. In this case, they are Integers, defined with the min, max, and the name of the parameter to the scikit-learn model. For your algorithm, you can just as easily optimize `Real()` and `Categorical()` data types.

```
...
# define the space of hyperparameters to search
search_space = [Integer(1, 5, name='n_neighbors'), Integer(1, 2, name='p')]
```

Listing 19.23: Example of defining the hyperparameter search space.

Next, we need to define a function that will be used to evaluate a given set of hyperparameters. We want to minimize this function, therefore smaller values returned must indicate a better performing model. We can use the `use_named_args()` decorator from the scikit-optimize project on the function definition that allows the function to be called directly with a specific set of parameters from the search space. As such, our custom function will take the hyperparameter values as arguments, which can be provided to the model directly in order to configure it. We can define these arguments generically in Python using the `**params` argument to the function, then pass them to the model via the `set_params(**)` function.

Now that the model is configured, we can evaluate it. In this case, we will use 5-fold cross-validation on our dataset and evaluate the accuracy for each fold. We can then report the performance of the model as one minus the mean accuracy across these folds. This means that a perfect model with an accuracy of 1.0 will return a value of 0.0 (1.0 – mean accuracy). This function is defined after we have loaded the dataset and defined the model so that both the dataset and model are in scope and can be used directly.

```
# define the function used to evaluate a given configuration
@use_named_args(search_space)
def evaluate_model(**params):
  # something
  model.set_params(**params)
  # calculate 5-fold cross-validation
  result = cross_val_score(model, X, y, cv=5, n_jobs=-1, scoring='accuracy')
  # calculate the mean of the scores
  estimate = mean(result)
  return 1.0 - estimate
```

Listing 19.24: Example of a function for evaluating a set of model hyperparameters.

Next, we can perform the optimization. This is achieved by calling the `gp_minimize()` function with the name of the objective function and the defined search space. By default, this function will use a `gp_hedge` acquisition function that tries to figure out the best strategy, but this can be configured via the `acq_func` argument. The optimization will also run for 100 iterations by default, but this can be controlled via the `n_calls` argument.

```
...
# perform optimization
result = gp_minimize(evaluate_model, search_space)
```

Listing 19.25: Example of running the Bayesian Optimization.

Once run, we can access the best score via the `fun` property and the best set of hyperparameters via the `x` array property.

```
...
# summarizing finding:
print('Best Accuracy: %.3f' % (1.0 - result.fun))
print('Best Parameters: n_neighbors=%d, p=%d' % (result.x[0], result.x[1]))
```

Listing 19.26: Example of summarizing the results of the Bayesian Optimization.

Tying this all together, the complete example is listed below.

```
# example of bayesian optimization with scikit-optimize
from numpy import mean
from sklearn.datasets import make_blobs
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from skopt.space import Integer
from skopt.utils import use_named_args
from skopt import gp_minimize

# generate 2d classification dataset
X, y = make_blobs(n_samples=500, centers=3, n_features=2)
# define the model
model = KNeighborsClassifier()
# define the space of hyperparameters to search
search_space = [Integer(1, 5, name='n_neighbors'), Integer(1, 2, name='p')]

# define the function used to evaluate a given configuration
@use_named_args(search_space)
def evaluate_model(**params):
  # something
  model.set_params(**params)
  # calculate 5-fold cross validation
  result = cross_val_score(model, X, y, cv=5, n_jobs=-1, scoring='accuracy')
  # calculate the mean of the scores
  estimate = mean(result)
  return 1.0 - estimate

# perform optimization
result = gp_minimize(evaluate_model, search_space)
# summarizing finding:
print('Best Accuracy: %.3f' % (1.0 - result.fun))
print('Best Parameters: n_neighbors=%d, p=%d' % (result.x[0], result.x[1]))
```

Listing 19.27: Examples of Bayesian Optimization for model hyperparameters.

Running the example executes the hyperparameter tuning using Bayesian Optimization. The code may report many warning messages, such as:

```
UserWarning: The objective has been evaluated at this point before.
```

Listing 19.28: Example of a possible warning message.

This is to be expected and is caused by the same hyperparameter configuration being evaluated more than once. Your specific results will vary given the stochastic nature of the test problem. Try running the example a few times. In this case, the model achieved about 97% accuracy via mean 5-fold cross-validation with 3 neighbors and a `p` neighborhood shape value of 2.

```
Best Accuracy: 0.976
Best Parameters: n_neighbors=3, p=2
```

Listing 19.29: Example of final results from running a Bayesian Optimization on model hyperparameters.

## 19.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 19.6.1 Papers

- *A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning*, 2010.
  https://arxiv.org/abs/1012.2599
- *Practical Bayesian Optimization of Machine Learning Algorithms*, 2012.
  http://papers.nips.cc/paper/4522-practical-bayesian-optimization
- *A Tutorial on Bayesian Optimization*, 2018.
  https://arxiv.org/abs/1807.02811

### 19.6.2 API

- Gaussian Processes, Scikit-Learn API.
  https://scikit-learn.org/stable/modules/gaussian_process.html
- Hyperopt: Distributed Asynchronous Hyper-parameter Optimization.
  https://github.com/hyperopt/hyperopt
- Scikit-Optimize Project.
  https://github.com/scikit-optimize/scikit-optimize

### 19.6.3 Articles

- Global optimization, Wikipedia.
  https://en.wikipedia.org/wiki/Global_optimization
- Bayesian optimization, Wikipedia.
  https://en.wikipedia.org/wiki/Bayesian_optimization

## 19.7 Summary

In this tutorial, you discovered Bayesian Optimization for directed search of complex optimization problems. Specifically, you learned:

- Global optimization is a challenging problem that involves black box and often non-convex, nonlinear, noisy, and computationally expensive objective functions.
- Bayesian Optimization provides a probabilistically principled method for global optimization.
- How to implement Bayesian Optimization from scratch and how to use open-source implementations.

### 19.7.1 Next

In the next tutorial, you will discover Bayesian Belief Networks.

# Chapter 20

# Bayesian Belief Networks

Probabilistic models can define relationships between variables and be used to calculate probabilities. For example, fully conditional models may require an enormous amount of data to cover all possible cases, and probabilities may be intractable to calculate in practice. Simplifying assumptions such as the conditional independence of all random variables can be effective, such as in the case of Naive Bayes, although it is a drastically simplifying step. An alternative is to develop a model that preserves known conditional dependence between random variables and conditional independence in all other cases. Bayesian networks are a probabilistic graphical model that explicitly capture known conditional dependence with directed edges in a graph model. All missing connections define the conditional independencies in the model. As such Bayesian Networks provide a useful tool to visualize the probabilistic model for a domain, review all of the relationships between the random variables, and reason about causal probabilities for scenarios given available evidence. In this tutorial, you will discover a gentle introduction to Bayesian Networks. After reading this tutorial, you will know:

- Bayesian networks are a type of probabilistic graphical model comprised of nodes and directed edges.
- Bayesian network models capture both conditionally dependent and conditionally independent relationships between random variables.
- Models can be prepared by experts or learned from data, then used for inference to estimate the probabilities for causal or subsequent events.

Let's get started.

## 20.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Challenge of Probabilistic Modeling
2. Bayesian Belief Network as a Probabilistic Model
3. How to Develop and Use a Bayesian Network
4. Example of a Bayesian Network
5. Bayesian Networks in Python

## 20.2 Challenge of Probabilistic Modeling

Probabilistic models can be challenging to design and use. Most often, the problem is the lack of information about the domain required to fully specify the conditional dependence between random variables. If available, calculating the full conditional probability for an event can be impractical. A common approach to addressing this challenge is to add some simplifying assumptions, such as assuming that all random variables in the model are conditionally independent. This is a drastic assumption, although it proves useful in practice, providing the basis for the Naive Bayes classification algorithm.

An alternative approach is to develop a probabilistic model of a problem with some conditional independence assumptions. This provides an intermediate approach between a fully conditional model and a fully conditionally independent model. Bayesian belief networks are one example of a probabilistic model where some variables are conditionally independent.

> Bayesian belief networks provide an intermediate approach that is less constraining than the global assumption of conditional independence made by the naive Bayes classifier, but more tractable than avoiding conditional independence assumptions altogether.
>
> — Page 184, *Machine Learning*, 1997.

## 20.3 Bayesian Belief Network as a Probabilistic Model

A Bayesian belief network is a type of probabilistic graphical model.

### 20.3.1 Probabilistic Graphical Models

A probabilistic graphical model (PGM), or simply *graphical model* for short, is a way of representing a probabilistic model with a graph structure. The nodes in the graph represent random variables and the edges that connect the nodes represent the relationships between the random variables.

> A graph comprises nodes (also called vertices) connected by links (also known as edges or arcs). In a probabilistic graphical model, each node represents a random variable (or group of random variables), and the links express probabilistic relationships between these variables.
>
> — Page 360, *Pattern Recognition and Machine Learning*, 2006.

- **Nodes**: Random variables in a graphical model.
- **Edges**: Relationships between random variables in a graphical model.

There are many different types of graphical models, although the two most commonly described are the Hidden Markov Model and the Bayesian Network. The Hidden Markov Model (HMM) is a graphical model where the edges of the graph are undirected, meaning the graph contains cycles. Bayesian Networks are more restrictive, where the edges of the graph are directed, meaning they can only be navigated in one direction. This means that cycles are not possible, and the structure can be more generally referred to as a directed acyclic graph (DAG).

> Directed graphs are useful for expressing causal relationships between random variables, whereas undirected graphs are better suited to expressing soft constraints between random variables.
>
> — Page 360, *Pattern Recognition and Machine Learning*, 2006.

### 20.3.2 Bayesian Belief Networks

A Bayesian Belief Network, or simply *Bayesian Network*, provides a simple way of applying Bayes Theorem to complex problems. The networks are not exactly Bayesian by definition, although given that both the probability distributions for the random variables (nodes) and the relationships between the random variables (edges) are specified subjectively, the model can be thought to capture the *belief* about a complex domain. Bayesian probability is the study of subjective probabilities or belief in an outcome, compared to the frequentist approach where probabilities are based purely on the past occurrence of the event. A Bayesian Network captures the joint probabilities of the events represented by the model.

> A Bayesian belief network describes the joint probability distribution for a set of variables.
>
> — Page 185, *Machine Learning*, 1997.

Central to the Bayesian network is the notion of conditional independence. Independence refers to a random variable that is unaffected by all other variables. A dependent variable is a random variable whose probability is conditional on one or more other random variables. Conditional independence describes the relationship among multiple random variables, where a given variable may be conditionally independent of one or more other random variables. This does not mean that the variable is independent per se; instead, it is a clear definition that the variable is independent of specific other known random variables.

A probabilistic graphical model, such as a Bayesian Network, provides a way of defining a probabilistic model for a complex problem by stating all of the conditional independence assumptions for the known variables, whilst allowing the presence of unknown (latent) variables. As such, both the presence and the absence of edges in the graphical model are important in the interpretation of the model.

> A graphical model (GM) is a way to represent a joint distribution by making [Conditional Independence] CI assumptions. In particular, the nodes in the graph represent random variables, and the (lack of) edges represent CI assumptions. (A better name for these models would in fact be "independence diagrams" ...
>
> — Page 308, *Machine Learning: A Probabilistic Perspective*, 2012.

Bayesian networks provide useful benefits as a probabilistic model. For example:

- **Visualization**. The model provides a direct way to visualize the structure of the model and motivate the design of new models.
- **Relationships**. Provides insights into the presence and absence of the relationships between random variables.
- **Computations**. Provides a way to structure complex probability calculations.

## 20.4 How to Develop and Use a Bayesian Network

Designing a Bayesian Network requires defining at least three things: Random Variables. What are the random variables in the problem? Conditional Relationships. What are the conditional relationships between the variables? Probability Distributions. What are the probability distributions for each variable? It may be possible for an expert in the problem domain to specify some or all of these aspects in the design of the model. In many cases, the architecture or topology of the graphical model can be specified by an expert, but the probability distributions must be estimated from data from the domain.

Both the probability distributions and the graph structure itself can be estimated from data, although it can be a challenging process. As such, it is common to use learning algorithms for this purpose; for example, assuming a Gaussian distribution for continuous random variables and using gradient ascent for estimating the distribution parameters. Once a Bayesian Network has been prepared for a domain, it can be used for reasoning, e.g. making decisions. Reasoning is achieved via inference with the model for a given situation. For example, the outcome for some events is known and plugged into the random variables. The model can be used to estimate the probability of causes for the events or possible further outcomes.

> Reasoning (inference) is then performed by introducing evidence that sets variables in known states, and subsequently computing probabilities of interest, conditioned on this evidence.
>
> — Page 13, *Bayesian Reasoning and Machine Learning*, 2012.

Practical examples of using Bayesian Networks in practice include medicine (symptoms and diseases), bioinformatics (traits and genes), and speech recognition (utterances and time).

## 20.5 Example of a Bayesian Network

We can make Bayesian Networks concrete with a small example. Consider a problem with three random variables: $A$, $B$, and $C$. $A$ is dependent upon $B$, and $C$ is dependent upon $B$. We can state the conditional dependencies as follows:

- $A$ is conditionally dependent upon $B$, e.g. $P(A|B)$
- $C$ is conditionally dependent upon $B$, e.g. $P(C|B)$

We know that $C$ and $A$ have no effect on each other. We can also state the conditional independencies as follows:

- $A$ is conditionally independent from $C$: $P(A|B, C)$
- $C$ is conditionally independent from $A$: $P(C|B, A)$

Notice that the conditional dependence is stated in the presence of the conditional independence. That is, $A$ is conditionally independent of $C$, or $A$ is conditionally dependent upon $B$ in the presence of $C$. We might also state the conditional independence of $A$ given $C$ as the

conditional dependence of $A$ given $B$, as $A$ is unaffected by $C$ and can be calculated from $A$ given $B$ alone.

$$P(A|C, B) = P(A|B) \tag{20.1}$$

We can see that $B$ is unaffected by $A$ and $C$ and has no dependencies or parents; we can simply state the conditional independence of $B$ from $A$ and $C$ as $P(B, P(A|B), P(C|B))$ or $P(B)$. We can also write the joint probability of $A$ and $C$ given $B$ or conditioned on $B$ as the product of two conditional probabilities; for example:

$$P(A, C|B) = P(A|B) \times P(C|B) \tag{20.2}$$

The model summarizes the joint probability of $P(A, B, C)$, calculated as:

$$P(A, B, C) = P(A|B) \times P(C|B) \times P(B) \tag{20.3}$$

We can draw the graph as follows:

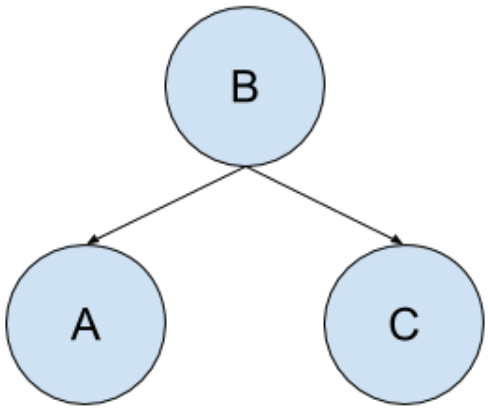


Figure 20.1: Example of a Simple Bayesian Network.

Notice that the random variables are each assigned a node, and the conditional probabilities are stated as directed connections between the nodes. Also notice that it is not possible to navigate the graph in a cycle, e.g. no loops are possible when navigating from node to node via the edges. Also notice that the graph is useful even at this point where we don't know the probability distributions for the variables. You might want to extend this example by using contrived probabilities for discrete events for each random variable and practice some simple inference for different scenarios.

## 20.6 Bayesian Networks in Python

Bayesian Networks can be developed and used for inference in Python. A popular library for this is called PyMC and provides a range of tools for Bayesian modeling, including graphical models like Bayesian Networks. The most recent version of the library is called PyMC3, named for Python version 3, and was developed on top of the Theano mathematical computation library that offers fast automatic differentiation.

> PyMC3 is a new open source probabilistic programming framework written in Python that uses Theano to compute gradients via automatic differentiation as well as compile probabilistic programs on-the-fly to C for increased speed.
>
> — *Probabilistic programming in Python using PyMC3*, 2016.

More generally, the use of probabilistic graphical models in computer software used for inference is referred to *probabilistic programming*.

> This type of programming is called probabilistic programming, [...] it is probabilistic in the sense that we create probability models using programming variables as the model's components. Model components are first-class primitives within the PyMC framework.
>
> — *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*, 2015.

## 20.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 20.7.1 Books

- *Bayesian Reasoning and Machine Learning*, 2012.
  https://amzn.to/2YoHbgV
- *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*, 2015.
  https://amzn.to/2Khk3bq

### 20.7.2 Book Chapters

- Chapter 6: Bayesian Learning, *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- Chapter 8: Graphical Models, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- Chapter 10: Directed graphical models (Bayes nets), *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- Chapter 14: Probabilistic Reasoning, *Artificial Intelligence: A Modern Approach*, 3rd edition, 2009.
  https://amzn.to/2Y7yCpO

### 20.7.3 Papers

- *Probabilistic programming in Python using PyMC3*, 2016.
  https://peerj.com/articles/cs-55/

### 20.7.4 Code

- PyMC3, Probabilistic Programming in Python.
  https://docs.pymc.io/
- Variational Inference: Bayesian Neural Networks
  https://docs.pymc.io/notebooks/bayesian_neural_network_advi.html

### 20.7.5 Articles

- Graphical model, Wikipedia.
  https://en.wikipedia.org/wiki/Graphical_model
- Hidden Markov model, Wikipedia.
  https://en.wikipedia.org/wiki/Hidden_Markov_model
- Bayesian network, Wikipedia.
  https://en.wikipedia.org/wiki/Bayesian_network
- Conditional independence, Wikipedia.
  https://en.wikipedia.org/wiki/Conditional_independence

## 20.8 Summary

In this tutorial, you discovered a gentle introduction to Bayesian Networks. Specifically, you learned:

- Bayesian networks are a type of probabilistic graphical model comprised of nodes and directed edges.
- Bayesian network models capture both conditionally dependent and conditionally independent relationships between random variables.
- Models can be prepared by experts or learned from data, then used for inference to estimate the probabilities for causal or subsequent events.

### 20.8.1 Next

This was the final tutorial in this Part. In the next Part, you will discover important probabilistic tools from the field of information theory.

# Part VII

# Information Theory

# Chapter 21

# Information Entropy

Information theory is a subfield of mathematics concerned with transmitting data across a noisy channel. A cornerstone of information theory is the idea of quantifying how much information there is in a message. More generally, this can be used to quantify the information in an event and a random variable, called entropy, and is calculated using probability. Calculating information and entropy is a useful tool in machine learning and is used as the basis for techniques such as feature selection, building decision trees, and, more generally, fitting classification models. As such, a machine learning practitioner requires a strong understanding and intuition for information and entropy. In this tutorial, you will discover a gentle introduction to information entropy. After reading this tutorial, you will know:

- Information theory is concerned with data compression and transmission and builds upon probability and supports machine learning.
- Information provides a way to quantify the amount of surprise for an event measured in bits.
- Entropy provides a measure of the average amount of information needed to represent an event drawn from a probability distribution for a random variable.

Let's get started.

## 21.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. What Is Information Theory?
2. Calculate the Information for an Event
3. Calculate the Information for a Random Variable

## 21.2 What Is Information Theory?

Information theory is a field of study concerned with quantifying information for communication. It is a subfield of mathematics and is concerned with topics like data compression and the limits of signal processing. The field was proposed and developed by Claude Shannon while working at the US telephone company Bell Labs.

> Information theory is concerned with representing data in a compact fashion (a task known as data compression or source coding), as well as with transmitting and storing it in a way that is robust to errors (a task known as error correction or channel coding).
>
> — Page 56, *Machine Learning: A Probabilistic Perspective*, 2012.

A foundational concept from information theory is the quantification of the amount of information in things like events, random variables, and distributions. Quantifying the amount of information requires the use of probabilities, hence the relationship of information theory to probability. Measurements of information are widely used in artificial intelligence and machine learning, such as in the construction of decision trees and the optimization of classifier models. As such, there is an important relationship between information theory and machine learning and a practitioner must be familiar with some of the basic concepts from the field.

> Why unify information theory and machine learning? Because they are two sides of the same coin. [...] Information theory and machine learning still belong together. Brains are the ultimate compression and communication systems. And the state-of-the-art algorithms for both data compression and error-correcting codes use the same tools as machine learning.
>
> — Page v, *Information Theory, Inference, and Learning Algorithms*, 2003.

## 21.3 Calculate the Information for an Event

Quantifying information is the foundation of the field of information theory. The intuition behind quantifying information is the idea of measuring how much surprise there is in an event, that is, how unlikely it is. Those events that are rare (low probability) are more surprising and therefore have more information those events that are common (high probability).

- **Low Probability Event**: High Information (surprising).
- **High Probability Event**: Low Information (unsurprising).

> The basic intuition behind information theory is that learning that an unlikely event has occurred is more informative than learning that a likely event has occurred.
>
> — Page 73, *Deep Learning*, 2016.

Rare events are more uncertain or more surprising and require more information to represent them than common events. We can calculate the amount of information there is in an event using the probability of the event. This is called *Shannon information*, *self-information*, or simply the *information*, and can be calculated for a discrete event $I(x)$ as follows:

$$I(x) = -\log(p(x)) \tag{21.1}$$

Where $I(x)$ is the information content (self-information) of $x$, $log()$ is the base-2 logarithm and $p(x)$ is the probability of the event $x$. The choice of the base-2 logarithm means that the units of the information measure is in bits (binary digits). This can be directly interpreted in the information processing sense as the number of bits required to represent and the event on a noisy communication channel. The calculation of information is often written as $h()$ to contrast it with entropy $H()$ (discussed later); for example:

$$h(x) = -\log(p(x)) \tag{21.2}$$

The negative sign ensures that the result is always positive or zero. Information will be zero when the probability of an event is 1.0 or a certainty, e.g. there is no surprise. Let's make this concrete with some examples. Consider a flip of a single fair coin. The probability of heads (and tails) is 0.5. We can calculate the information for flipping a head in Python using the `log2()` function.

```
# calculate the information for a coin flip
from math import log2
# probability of the event
p = 0.5
# calculate information for event
h = -log2(p)
# print the result
print('p(x)=%.3f, information: %.3f bits' % (p, h))
```

Listing 21.1: Example calculating the information of a fair coin flip.

Running the example prints the probability of the event as 50% and the information content for the event as 1 bit.

```
p(x)=0.500, information: 1.000 bits
```

Listing 21.2: Example output from calculating the information of fair a coin flip.

If the same coin was flipped $n$ times, then the information for this sequence of flips would be $n$ bits. If the coin was not fair and the probability of a head was instead 10% (0.1), then the event would be more rare and would require more than 3 bits of information.

```
p(x)=0.100, information: 3.322 bits
```

Listing 21.3: Example output from calculating the information of a biased coin flip.

We can also explore the information in a single roll of a fair six-sided dice, e.g. the information in rolling a 6. We know the probability of rolling any number is $\frac{1}{6}$, which is a smaller number than $\frac{1}{2}$ for a coin flip, therefore we would expect more surprise or a larger amount of information.

```
# calculate the information for a dice roll
from math import log2
# probability of the event
```

```
p = 1.0 / 6.0
# calculate information for event
h = -log2(p)
# print the result
print('p(x)=%.3f, information: %.3f bits' % (p, h))
```

Listing 21.4: Example calculating the information of a dice roll.

Running the example, we can see that our intuition is correct and that indeed, there is more than 2.5 bits of information in a single roll of a fair die.

```
p(x)=0.167, information: 2.585 bits
```

Listing 21.5: Example output from calculating the information of a dice roll.

Other logarithms can be used instead of the base-2. For example, it is also common to use the natural logarithm that uses base-e (Euler's number) in calculating the information, in which case the units are referred to as *nats*.

We can further develop the intuition that low probability events have more information. To make this clear, we can calculate the information for probabilities between 0 and 1 and plot the corresponding information for each. We can then create a plot of probability vs information. We would expect the plot to curve downward from low probabilities with high information to high probabilities with low information. The complete example is listed below.

```
# compare probability vs information entropy
from math import log2
from matplotlib import pyplot
# list of probabilities
probs = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
# calculate information
info = [-log2(p) for p in probs]
# plot probability vs information
pyplot.plot(probs, info, marker='.')
pyplot.title('Probability vs Information')
pyplot.xlabel('Probability')
pyplot.ylabel('Information')
pyplot.show()
```

Listing 21.6: Example of comparing probability and information.

Running the example creates the plot of probability vs information in bits. We can see the expected relationship where low probability events are more surprising and carry more information, and the complement of high probability events carry less information. We can also see that this relationship is not linear, it is in-fact slightly sub-linear. This makes sense given the use of the log function.

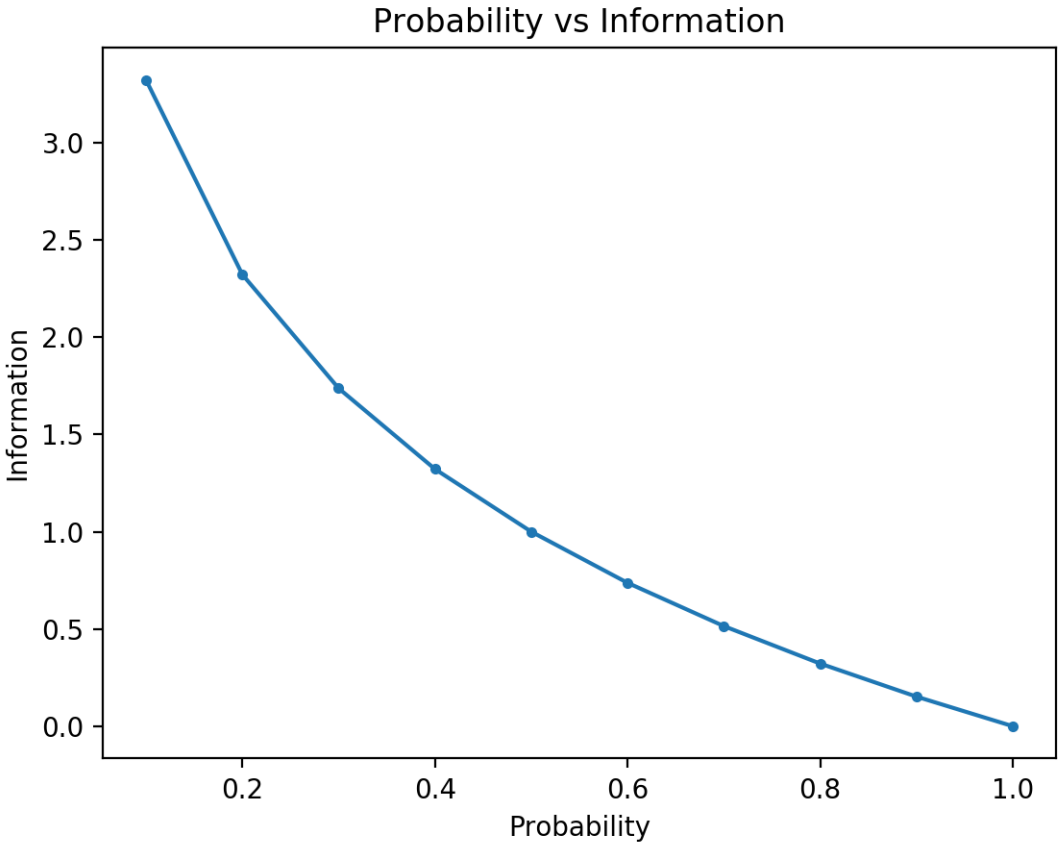


Figure 21.1: Plot of Probability vs Information.

## 21.4 Calculate the Information for a Random Variable

We can also quantify how much information there is in a random variable. For example, if we wanted to calculate the information for a random variable X with probability distribution p, this might be written as a function $H()$; for example:

$$H(X) \tag{21.3}$$

In effect, calculating the information for a random variable is the same as calculating the information for the probability distribution of the events for the random variable. Calculating the information for a random variable is called *information entropy*, *Shannon entropy*, or simply *entropy*. It is related to the idea of entropy from physics by analogy, in that both are concerned with uncertainty. The intuition for entropy is that it is the average number of bits required to represent or transmit an event drawn from the probability distribution for the random variable.

> ... the Shannon entropy of a distribution is the expected amount of information in an event drawn from that distribution. It gives a lower bound on the number of bits [...] needed on average to encode symbols drawn from a distribution $P$.
>
> — Page 74, *Deep Learning*, 2016.

Entropy can be calculated for a random variable $X$ with $K$ discrete states as follows:

$$H(X) = -\sum_{i=1}^{K} p(k_i) \times \log(p(k_i)) \tag{21.4}$$

That is the negative of the sum of the probability of each event multiplied by the log of the probability of each event. Like information, the $log()$ function uses base-2 and the units are bits. The natural logarithm can be used instead and the units will be nats. The lowest entropy is calculated for a random variable that has a single event with a probability of 1.0, a certainty. The largest entropy for a random variable will be if all events are equally likely. We can consider a roll of a fair die and calculate the entropy for the variable. Each outcome has the same probability of $\frac{1}{6}$, therefore it is a uniform probability distribution. We therefore would expect the average information to be the same information for a single event calculated in the previous section.

```
# calculate the entropy for a dice roll
from math import log2
# the number of events
n = 6
# probability of one event
p = 1.0 /n
# calculate entropy
entropy = -sum([p * log2(p) for _ in range(n)])
# print the result
print('entropy: %.3f bits' % entropy)
```

Listing 21.7: Example calculating the entropy of rolling a dice.

Running the example calculates the entropy as more than 2.5 bits, which is the same as the information for a single outcome. This makes sense, as the average information is the same as the lower bound on information as all outcomes are equally likely.

```
entropy: 2.585 bits
```

Listing 21.8: Example output from calculating the entropy of rolling a dice.

If we know the probability for each event, we can use the `entropy()` SciPy function to calculate the entropy directly. For example:

```
# calculate the entropy for a dice roll
from scipy.stats import entropy
# discrete probabilities
p = [1/6, 1/6, 1/6, 1/6, 1/6, 1/6]
# calculate entropy
e = entropy(p, base=2)
# print the result
print('entropy: %.3f bits' % e)
```

Listing 21.9: Example calculating the entropy of rolling a dice with SciPy.

Running the example reports the same result that we calculated manually.

```
entropy: 2.585 bits
```

Listing 21.10: Example output from calculating the entropy of rolling a dice with SciPy.

We can further develop the intuition for entropy of probability distributions. Recall that entropy is the number of bits required to represent a randomly drawn even from the distribution, e.g. an average event. We can explore this for a simple distribution with two events, like a coin flip, but explore different probabilities for these two events and calculate the entropy for each.

In the case where one event dominates, such as a skewed probability distribution, then there is less surprise and the distribution will have a lower entropy. In the case where no event dominates another, such as equal or approximately equal probability distribution, then we would expect larger or maximum entropy.

- **Skewed Probability Distribution** (*unsurprising*): Low entropy.
- **Balanced Probability Distribution** (*surprising*): High entropy.

If we transition from skewed to equal probability of events in the distribution we would expect entropy to start low and increase, specifically from the lowest entropy of 0.0 for events with impossibility/certainty (probability of 0 and 1 respectively) to the largest entropy of 1.0 for events with equal probability. The example below implements this, creating each probability distribution in this transition, calculating the entropy for each and plotting the result.

```
# compare probability distributions vs entropy
from math import log2
from matplotlib import pyplot

# calculate entropy
def entropy(events, ets=1e-15):
  return -sum([p * log2(p + ets) for p in events])

# define probabilities
probs = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
# create probability distribution
dists = [[p, 1.0 - p] for p in probs]
# calculate entropy for each distribution
ents = [entropy(d) for d in dists]
# plot probability distribution vs entropy
pyplot.plot(probs, ents, marker='.')
pyplot.title('Probability Distribution vs Entropy')
pyplot.xticks(probs, [str(d) for d in dists])
pyplot.xlabel('Probability Distribution')
pyplot.ylabel('Entropy (bits)')
pyplot.show()
```

Listing 21.11: Example calculating probability distributions vs entropy.

Running the example creates the 6 probability distributions with [0,1] probability through to [0.5,0.5] probabilities. As expected, we can see that as the distribution of events changes from skewed to balanced, the entropy increases from minimal to maximum values. That is, if the average event drawn from a probability distribution is not surprising we get a lower entropy, whereas if it is surprising, we get a larger entropy.

We can see that the transition is not linear, that it is super linear. We can also see that this curve is symmetrical if we continued the transition to [0.6, 0.4] and onward to [1.0, 0.0] for the two events, forming an inverted parabola-shape. Note we had to add a tiny value to the probability when calculating the entropy to avoid calculating the log of a zero value, which would result in an infinity on not a number.

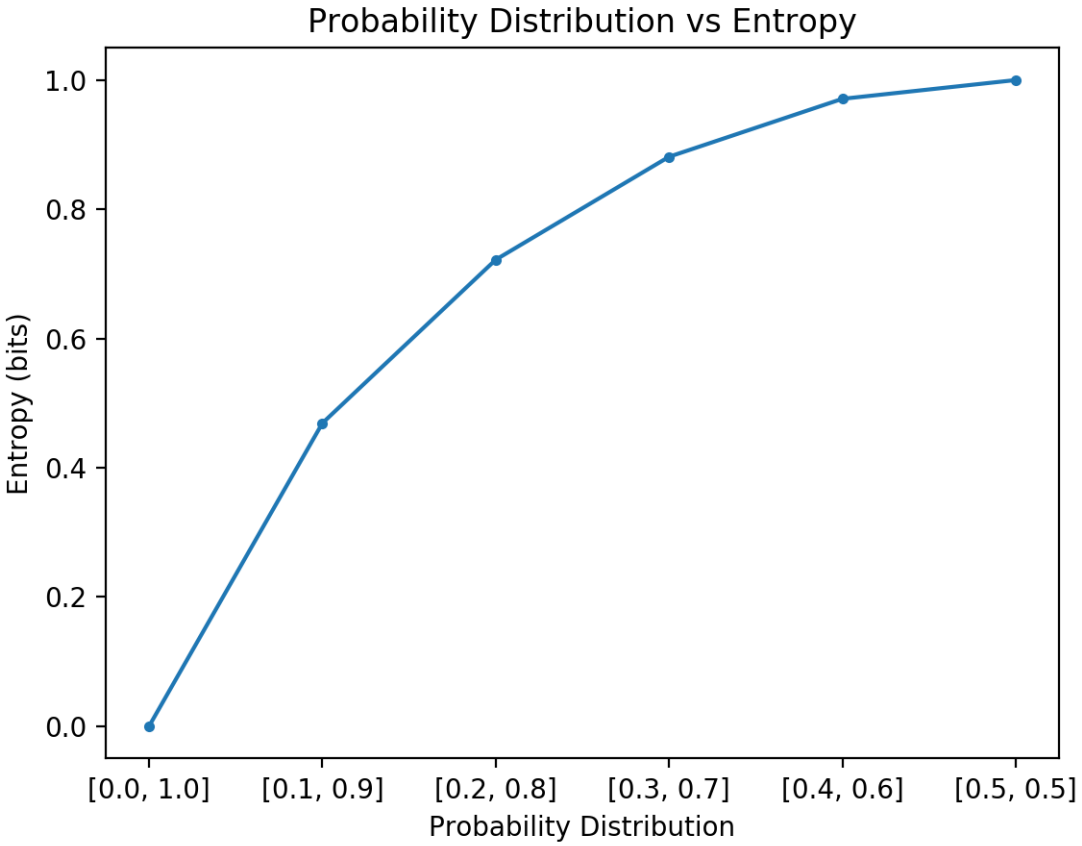


Figure 21.2: Plot of Probability Distribution vs Entropy.

Calculating the entropy for a random variable provides the basis for other measures such as mutual information (information gain). It also provides the basis for calculating the difference between two probability distributions with cross-entropy and the KL-divergence.

## 21.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 21.5.1 Books

- *Information Theory, Inference, and Learning Algorithms*, 2003.
  https://amzn.to/2KfDDF7

### 21.5.2 Chapters

- Section 2.8: Information theory, *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- Section 1.6: Information Theory, *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I

- Section 3.13 Information Theory, *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL

### 21.5.3 API

- `scipy.stats.entropy` API
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.entropy.html

### 21.5.4 Articles

- Entropy (information theory), Wikipedia.
  https://en.wikipedia.org/wiki/Entropy_(information_theory)
- Information gain in decision trees, Wikipedia.
  https://en.wikipedia.org/wiki/Information_gain_in_decision_trees
- Information gain ratio, Wikipedia.
  https://en.wikipedia.org/wiki/Information_gain_ratio

## 21.6 Summary

In this tutorial, you discovered a gentle introduction to information entropy. Specifically, you learned:

- Information theory is concerned with data compression and transmission and builds upon probability and supports machine learning.
- Information provides a way to quantify the amount of surprise for an event measured in bits.
- Entropy provides a measure of the average amount of information needed to represent an event drawn from a probability distribution for a random variable.

### 21.6.1 Next

In the next tutorial, you will discover the KL divergence measure calculated between two probability distributions.

# Chapter 22

# Divergence Between Probability Distributions

It is often desirable to quantify the difference between probability distributions for a given random variable. This occurs frequently in machine learning, when we may be interested in calculating the difference between an actual and observed probability distribution. This can be achieved using techniques from information theory, such as the Kullback-Leibler Divergence (KL divergence), or relative entropy, and the Jensen-Shannon Divergence that provides a normalized and symmetrical version of the KL divergence. These scoring methods can be used as shortcuts in the calculation of other widely used methods, such as mutual information for feature selection prior to modeling, and cross-entropy used as a loss function for many different classifier models. In this tutorial, you will discover how to calculate the divergence between probability distributions. After reading this tutorial, you will know:

- Statistical distance is the general idea of calculating the difference between statistical objects like different probability distributions for a random variable.
- Kullback-Leibler divergence calculates a score that measures the divergence of one probability distribution from another.
- Jensen-Shannon divergence extends KL divergence to calculate a symmetrical score and distance measure of one probability distribution from another.

Let's get started.

## 22.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Statistical Distance
2. Kullback-Leibler Divergence
3. Jensen-Shannon Divergence

## 22.2 Statistical Distance

There are many situations where we may want to compare two probability distributions. Specifically, we may have a single random variable and two different probability distributions for the variable, such as a true distribution and an approximation of that distribution. In situations like this, it can be useful to quantify the difference between the distributions. Generally, this is referred to as the problem of calculating the statistical distance between two statistical objects, e.g. probability distributions. One approach is to calculate a distance measure between the two distributions. This can be challenging as it can be difficult to interpret the measure.

Instead, it is more common to calculate a divergence between two probability distributions. A divergence is like a measure but is not symmetrical. This means that a divergence is a scoring of how one distribution differs from another, where calculating the divergence for distributions $P$ and $Q$ would give a different score from $Q$ and $P$. Divergence scores are an important foundation for many different calculations in information theory and more generally in machine learning. For example, they provide shortcuts for calculating scores such as mutual information (information gain) and cross-entropy used as a loss function for classification models.

Divergence scores are also used directly as tools for understanding complex modeling problems, such as approximating a target probability distribution when optimizing generative adversarial network (GAN) models. Two commonly used divergence scores from information theory are Kullback-Leibler Divergence and Jensen-Shannon Divergence. We will take a closer look at both of these scores in the following section.

## 22.3 Kullback-Leibler Divergence

The Kullback-Leibler Divergence score, or KL divergence score, quantifies how much one probability distribution differs from another probability distribution. The KL divergence between two distributions $Q$ and $P$ is often stated using the following notation:

$$KL(P||Q) \tag{22.1}$$

Where the $||$ operator indicates *divergence* or $P$s divergence from $Q$. KL divergence can be calculated as the negative sum of probability of each event in $P$ multiplied by the log of the probability of the event in $Q$ over the probability of the event in $P$.

$$KL(P||Q) = -\sum^{\{x \in X\}} P(x) \times \log(\frac{Q(x)}{P(x)}) \tag{22.2}$$

The value within the sum is the divergence for a given event. This is the same as the positive sum of probability of each event in $P$ multiplied by the log of the probability of the event in $P$ over the probability of the event in $Q$ (e.g. the terms in the fraction are flipped). This is the more common implementation used in practice.

$$KL(P||Q) = \sum^{\{x \in X\}} P(x) \times \log(\frac{P(x)}{Q(x)}) \tag{22.3}$$

The intuition for the KL divergence score is that when the probability for an event from $P$ is large, but the probability for the same event in $Q$ is small, there is a large divergence. When the

probability from $P$ is small and the probability from $Q$ is large, there is also a large divergence, but not as large as the first case. It can be used to measure the divergence between discrete or continuous probability distributions, where in the latter case the integral of the events is calculated instead of the sum of the probabilities of the discrete events.

> One way to measure the dissimilarity of two probability distributions, $p$ and $q$, is known as the Kullback-Leibler divergence (KL divergence) or relative entropy.
>
> — Page 57, *Machine Learning: A Probabilistic Perspective*, 2012.

The log can be base-2 to give units in *bits*, or the natural logarithm base-e with units in *nats*. When the score is 0, it suggests that both distributions are identical, otherwise the score is positive. Importantly, the KL divergence score is not symmetrical, for example:

$$KL(P||Q) \neq KL(Q||P) \tag{22.4}$$

It is named for the two authors of the method Solomon Kullback and Richard Leibler, and is sometimes referred to as *relative entropy*.

> This is known as the relative entropy or Kullback-Leibler divergence, or KL divergence, between the distributions $p(x)$ and $q(x)$.
>
> — Page 55, *Pattern Recognition and Machine Learning*, 2006.

If we are attempting to approximate an unknown probability distribution, then the target probability distribution from data is $P$ and $Q$ is our approximation of the distribution. In this case, the KL divergence summarizes the number of additional bits (i.e. calculated with the base-2 logarithm) required to represent an event from the random variable. The better our approximation, the less additional information is required.

> ... the KL divergence is the average number of extra bits needed to encode the data, due to the fact that we used distribution $q$ to encode the data instead of the true distribution $p$.
>
> — Page 58, *Machine Learning: A Probabilistic Perspective*, 2012.

We can make the KL divergence concrete with a worked example. Consider a random variable with three events as different colors. We may have two different probability distributions for this variable; for example:

```
...
# define distributions
events = ['red', 'green', 'blue']
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
```

Listing 22.1: Example of defining probability distributions.

We can plot a bar chart of these probabilities to compare them directly as probability histograms. The complete example is listed below.

```
# plot of distributions
from matplotlib import pyplot
# define distributions
events = ['red', 'green', 'blue']
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
print('P=%.3f Q=%.3f' % (sum(p), sum(q)))
# plot first distribution
pyplot.subplot(2,1,1)
pyplot.bar(events, p)
# plot second distribution
pyplot.subplot(2,1,2)
pyplot.bar(events, q)
# show the plot
pyplot.show()
```

Listing 22.2: Example of summarizing two probability distributions.

Running the example creates a histogram for each probability distribution, allowing the probabilities for each event to be directly compared. We can see that indeed the distributions are different.

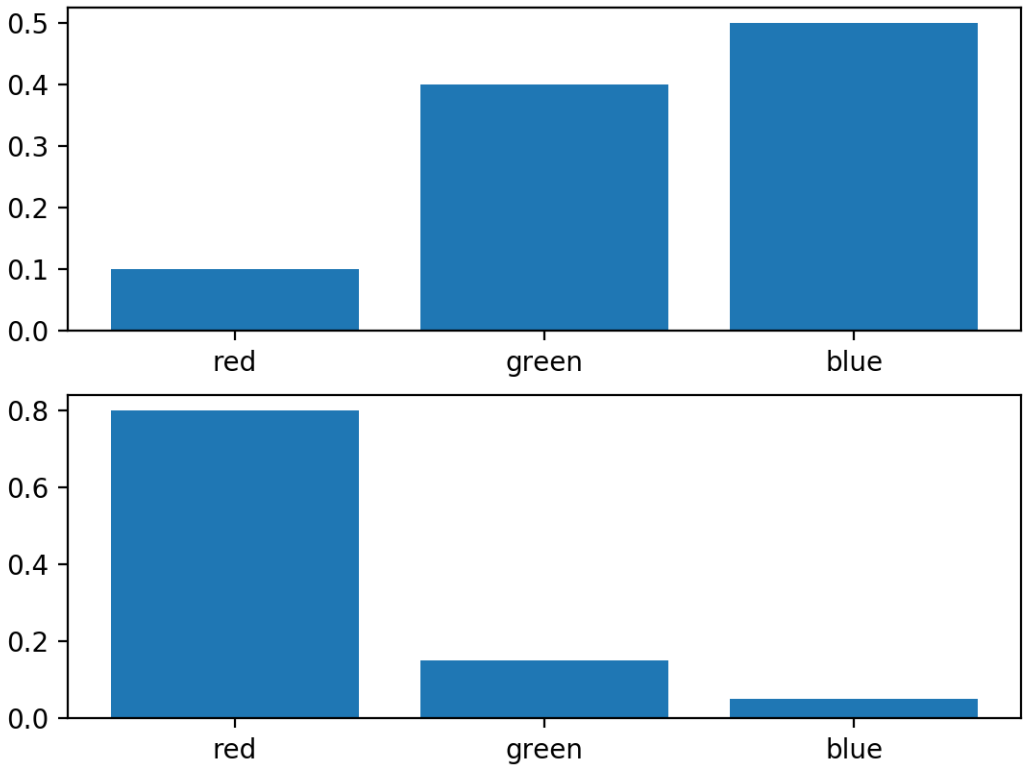


Figure 22.1: Histogram of Two Different Probability Distributions for the Same Random Variable.

Next, we can develop a function to calculate the KL divergence between the two distributions. We will use log base-2 to ensure the result has units in bits.

```
# calculate the kl divergence
def kl_divergence(p, q):
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))
```

Listing 22.3: Example of defining a function for calculating the KL divergence.

We can then use this function to calculate the KL divergence of $P$ from $Q$, as well as the reverse, $Q$ from $P$.

```
# calculate (P || Q)
kl_pq = kl_divergence(p, q)
print('KL(P || Q): %.3f bits' % kl_pq)
# calculate (Q || P)
kl_qp = kl_divergence(q, p)
print('KL(Q || P): %.3f bits' % kl_qp)
```

Listing 22.4: Example of calculating the KL divergence.

Tying this all together, the complete example is listed below.

```
# example of calculating the kl divergence between two probability mass functions
from math import log2

# calculate the kl divergence
def kl_divergence(p, q):
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))

# define distributions
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
# calculate (P || Q)
kl_pq = kl_divergence(p, q)
print('KL(P || Q): %.3f bits' % kl_pq)
# calculate (Q || P)
kl_qp = kl_divergence(q, p)
print('KL(Q || P): %.3f bits' % kl_qp)
```

Listing 22.5: Example of calculating the KL divergence between two probability distributions.

Running the example first calculates the divergence of $P$ from $Q$ as just under 2 bits, then $Q$ from $P$ as just over 2 bits. This is intuitive if we consider $P$ has large probabilities when $Q$ is small, giving $P$ less divergence than $Q$ from $P$ as $Q$ has more small probabilities when $P$ has large probabilities. There is more divergence in this second case.

```
KL(P || Q): 1.927 bits
KL(Q || P): 2.022 bits
```

Listing 22.6: Example output from calculating the KL divergence between two probability distributions in bits.

If we change `log2()` to the natural logarithm `log()` function, the result is in nats, as follows:

```
# KL(P || Q): 1.336 nats
# KL(Q || P): 1.401 nats
```

Listing 22.7: Example output from calculating the KL divergence between two probability distributions in nats.

The SciPy library provides the `kl_div()` function for calculating the KL divergence, although with a different definition as defined in this tutorial. It also provides the `rel_entr()` function for calculating the relative entropy, which matches our definition of KL divergence. This is odd as *relative entropy* is often used as a synonym for *KL divergence*. Nevertheless, we can calculate the KL divergence using the `rel_entr()` SciPy function and confirm that our manual calculation is correct. The `rel_entr()` function takes lists of probabilities across all events from each probability distribution as arguments and returns a list of divergences for each event. These can be summed to give the KL divergence. The calculation uses the natural logarithm instead of log base-2 so the units are in nats instead of bits. The complete example using SciPy to calculate $KL(P||Q)$ and $KL(Q||P)$ for the same probability distributions used above is listed below:

```
# example of calculating the kl divergence (relative entropy) with scipy
from scipy.special import rel_entr
# define distributions
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
# calculate (P || Q)
kl_pq = rel_entr(p, q)
print('KL(P || Q): %.3f nats' % sum(kl_pq))
# calculate (Q || P)
kl_qp = rel_entr(q, p)
print('KL(Q || P): %.3f nats' % sum(kl_qp))
```

Listing 22.8: Example of calculating the KL divergence between two probability distributions with SciPy.

Running the example, we can see that the calculated divergences match our manual calculation of about 1.3 nats and about 1.4 nats for $KL(P||Q)$ and $KL(Q||P)$ respectively.

```
KL(P || Q): 1.336 nats
KL(Q || P): 1.401 nats
```

Listing 22.9: Example output from calculating the KL divergence between two probability distributions with SciPy.

## 22.4 Jensen-Shannon Divergence

The Jensen-Shannon divergence, or JS divergence for short, is another way to quantify the difference (or similarity) between two probability distributions. It uses the KL divergence to calculate a normalized score that is symmetrical. This means that the divergence of $P$ from $Q$ is the same as $Q$ from $P$, or stated formally:

$$JS(P||Q) \equiv JS(Q||P) \tag{22.5}$$

The JS divergence can be calculated as follows:

$$JS(P||Q) = \frac{1}{2} \times KL(P||M) + \frac{1}{2} \times KL(Q||M) \tag{22.6}$$

Where $M$ is calculated as:

$$M = \frac{1}{2} \times (P + Q) \tag{22.7}$$

And $KL()$ function is calculated as the KL divergence described in the previous section. It is more useful as a measure as it provides a smoothed and normalized version of KL divergence, with scores between 0 (identical) and 1 (maximally different), when using the base-2 logarithm. The square root of the score gives a quantity referred to as the Jensen-Shannon distance, or JS distance for short. We can make the JS divergence concrete with a worked example. First, we can define a function to calculate the JS divergence that uses the `kl_divergence()` function prepared in the previous section.

```
# calculate the kl divergence
def kl_divergence(p, q):
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))

# calculate the js divergence
def js_divergence(p, q):
  m = 0.5 * (p + q)
  return 0.5 * kl_divergence(p, m) + 0.5 * kl_divergence(q, m)
```

Listing 22.10: Example of defining a function for calculating the JS divergence.

We can then test this function using the same probability distributions used in the previous section. First, we will calculate the JS divergence score for the distributions, then calculate the square root of the score to give the JS distance between the distributions. For example:

```
...
# calculate JS(P || Q)
js_pq = js_divergence(p, q)
print('JS(P || Q) divergence: %.3f bits' % js_pq)
print('JS(P || Q) distance: %.3f' % sqrt(js_pq))
```

Listing 22.11: Example of calculating the JS divergence and distance.

This can then be repeated for the reverse case to show that the divergence is symmetrical, unlike the KL divergence.

```
...
# calculate JS(Q || P)
js_qp = js_divergence(q, p)
print('JS(Q || P) divergence: %.3f bits' % js_qp)
print('JS(Q || P) distance: %.3f' % sqrt(js_qp))
```

Listing 22.12: Example of calculating the JS divergence and distance for the reverse case.

Tying this together, the complete example of calculating the JS divergence and JS distance is listed below.

```
# example of calculating the js divergence between two mass functions
from math import log2
from math import sqrt
from numpy import asarray

# calculate the kl divergence
def kl_divergence(p, q):
```

```
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))

# calculate the js divergence
def js_divergence(p, q):
  m = 0.5 * (p + q)
  return 0.5 * kl_divergence(p, m) + 0.5 * kl_divergence(q, m)

# define distributions
p = asarray([0.10, 0.40, 0.50])
q = asarray([0.80, 0.15, 0.05])
# calculate JS(P || Q)
js_pq = js_divergence(p, q)
print('JS(P || Q) divergence: %.3f bits' % js_pq)
print('JS(P || Q) distance: %.3f' % sqrt(js_pq))
# calculate JS(Q || P)
js_qp = js_divergence(q, p)
print('JS(Q || P) divergence: %.3f bits' % js_qp)
print('JS(Q || P) distance: %.3f' % sqrt(js_qp))
```

Listing 22.13: Example of calculating the JS divergence between two probability distributions.

Running the example shows that the JS divergence and distance between the distributions is about 0.4 bits and that the distance is about 0.6. We can see that the calculation is symmetrical, giving the same score and distance measure for $JS(P||Q)$ and $JS(Q||P)$.

```
JS(P || Q) divergence: 0.420 bits
JS(P || Q) distance: 0.648
JS(Q || P) divergence: 0.420 bits
JS(Q || P) distance: 0.648
```

Listing 22.14: Example output from calculating the JS divergence and distance between two probability distributions in bits.

The SciPy library provides an implementation of the JS distance via the `jensenshannon()` function. It takes arrays of probabilities across all events from each probability distribution as arguments and returns the JS distance score, not a divergence score. We can use this function to confirm our manual calculation of the JS distance. The complete example is listed below.

```
# calculate the jensen-shannon distance metric
from scipy.spatial.distance import jensenshannon
from numpy import asarray
# define distributions
p = asarray([0.10, 0.40, 0.50])
q = asarray([0.80, 0.15, 0.05])
# calculate JS(P || Q)
js_pq = jensenshannon(p, q, base=2)
print('JS(P || Q) Distance: %.3f' % js_pq)
# calculate JS(Q || P)
js_qp = jensenshannon(q, p, base=2)
print('JS(Q || P) Distance: %.3f' % js_qp)
```

Listing 22.15: Example of calculating the JS divergence between two probability distributions with SciPy.

Running the example, we can confirm the distance score matches our manual calculation of 0.648, and that the distance calculation is symmetrical as expected.

```
JS(P || Q) Distance: 0.648
JS(Q || P) Distance: 0.648
```

Listing 22.16: Example output from calculating the JS distance between two probability distributions with SciPy.

## 22.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 22.5.1 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I

### 22.5.2 APIs

- `scipy.stats.entropy` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.entropy.html
- `scipy.special.kl_div` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.kl_div.html
- `scipy.spatial.distance.jensenshannon` API.
  https://scipy.github.io/devdocs/generated/scipy.spatial.distance.jensenshannon.html

### 22.5.3 Articles

- Statistical distance, Wikipedia.
  https://en.wikipedia.org/wiki/Statistical_distance
- Divergence (statistics), Wikipedia.
  https://en.wikipedia.org/wiki/Divergence_(statistics)
- Kullback-Leibler divergence, Wikipedia.
  https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Jensen-Shannon divergence, Wikipedia.
  https://en.wikipedia.org/wiki/Jensen%E2%80%93Shannon_divergence

## 22.6 Summary

In this tutorial, you discovered how to calculate the divergence between probability distributions. Specifically, you learned:

- Statistical distance is the general idea of calculating the difference between statistical objects like different probability distributions for a random variable.
- Kullback-Leibler divergence calculates a score that measures the divergence of one probability distribution from another.
- Jensen-Shannon divergence extends KL divergence to calculate a symmetrical score and distance measure of one probability distribution from another.

### 22.6.1 Next

In the next tutorial, you will discover the cross-entropy measure calculated between two probability distributions.

# Chapter 23

# Cross-Entropy for Machine Learning

Cross-entropy is commonly used in machine learning as a loss function. Cross-entropy is a measure from the field of information theory, building upon entropy and generally calculating the difference between two probability distributions. It is closely related to but is different from KL divergence that calculates the relative entropy between two probability distributions, whereas cross-entropy can be thought to calculate the total entropy between the distributions. Cross-entropy is also related to and often confused with logistic loss, called log loss. Although the two measures are derived from a different source, when used as loss functions for classification models, both measures calculate the same quantity and can be used interchangeably. In this tutorial, you will discover cross-entropy for machine learning. After completing this tutorial, you will know:

- How to calculate cross-entropy from scratch and using standard machine learning libraries.
- Cross-entropy can be used as a loss function when optimizing classification models like logistic regression and artificial neural networks.
- Cross-entropy is different from KL divergence but can be calculated using KL divergence, and is different from log loss but calculates the same quantity when used as a loss function.

Let's get started.

## 23.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is Cross-Entropy?
2. Difference Between Cross-Entropy and KL Divergence
3. How to Calculate Cross-Entropy?
4. Cross-Entropy as a Loss Function
5. Difference Between Cross-Entropy and Log Loss

## 23.2 What Is Cross-Entropy?

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. Specifically, it builds upon the idea of entropy from information theory and calculates the average number of bits required to represent or transmit an event from one distribution compared to the other distribution.

> ... the cross-entropy is the average number of bits needed to encode data coming from a source with distribution $p$ when we use model $q$ ...
>
> — Page 57, *Machine Learning: A Probabilistic Perspective*, 2012.

The intuition for this definition comes if we consider a target or underlying probability distribution $P$ and an approximation of the target distribution $Q$, then the cross-entropy of $Q$ from $P$ is the number of additional bits to represent an event using $Q$ instead of $P$. The cross-entropy between two probability distributions, such as $Q$ from $P$, can be stated formally as:

$$H(P, Q) \tag{23.1}$$

Where $H()$ is the cross-entropy function, $P$ may be the target distribution and $Q$ is the approximation of the target distribution. Cross-entropy can be calculated using the probabilities of the events from $P$ and $Q$, as follows:

$$H(P, Q) = -\sum^{\{x \in X\}} P(x) \times \log(Q(x)) \tag{23.2}$$

Where $P(x)$ is the probability of the event $x$ in $P$, $Q(x)$ is the probability of event $x$ in $Q$ and $log()$ is the base-2 logarithm, meaning that the results are in bits. If the base-e or natural logarithm is used instead, the result will have the units called nats. This calculation is for discrete probability distributions, although a similar calculation can be used for continuous probability distributions using the integral across the events instead of the sum. The result will be a positive number measured in bits and will be equal to the entropy of the distribution if the two probability distributions are identical. Note that this notation looks a lot like the joint probability, or more specifically, the joint entropy between $P$ and $Q$. This is misleading as we are scoring the difference between probability distributions with cross-entropy. Whereas, joint entropy is a different concept that uses the same notation and instead calculates the uncertainty across two (or more) random variables.

## 23.3 Difference Between Cross-Entropy and KL Divergence

Cross-entropy is not KL Divergence. Cross-entropy is related to divergence measures, such as the Kullback-Leibler, or KL, Divergence that quantifies how much one distribution differs from another. Specifically, the KL divergence measures a very similar quantity to cross-entropy. It measures the average number of extra bits required to represent a message with $Q$ instead of $P$, not the total number of bits.

> In other words, the KL divergence is the average number of extra bits needed to encode the data, due to the fact that we used distribution $q$ to encode the data instead of the true distribution $p$.
>
> — Page 58, *Machine Learning: A Probabilistic Perspective*, 2012.

As such, the KL divergence is often referred to as the *relative entropy*.

- **Cross-Entropy**: Average number of total bits to represent an event from $Q$ instead of $P$.
- **Relative Entropy** (KL Divergence): Average number of extra bits to represent an event from $Q$ instead of $P$.

KL divergence can be calculated as the negative sum of probability of each event in $P$ multiplied by the log of the probability of the event in $Q$ over the probability of the event in $P$. Typically, log base-2 so that the result is measured in bits.

$$KL(P||Q) = -\sum^{\{x \in X\}} P(x) \times \log(\frac{Q(x)}{P(x)}) \tag{23.3}$$

The value within the sum is the divergence for a given event. As such, we can calculate the cross-entropy by adding the entropy of the distribution plus the additional entropy calculated by the KL divergence. This is intuitive, given the definition of both calculations; for example:

$$H(P, Q) = H(P) + KL(P||Q) \tag{23.4}$$

Where $H(P, Q)$ is the cross-entropy of $Q$ from $P$, $H(P)$ is the entropy of $P$ and $KL(P||Q)$ is the divergence of $Q$ from $P$. Entropy can be calculated for a probability distribution as the negative sum of the probability for each event multiplied by the log of the probability for the event, where log is base-2 to ensure the result is in bits.

$$H(P) = -\sum^{\{x \in X\}} p(x) \times \log(p(x)) \tag{23.5}$$

Like KL divergence, cross-entropy is not symmetrical, meaning that:

$$H(P, Q) \neq H(Q, P) \tag{23.6}$$

As we will see later, both cross-entropy and KL divergence calculate the same quantity when they are used as loss functions for optimizing a classification predictive model. It is under this context that you might sometimes see that cross-entropy and KL divergence are the same.

## 23.4 How to Calculate Cross-Entropy

In this section we will make the calculation of cross-entropy concrete with a small example.

### 23.4.1 Two Discrete Probability Distributions

Consider a random variable with three discrete events as different colors: *red*, *green*, and *blue*.

We may have two different probability distributions for this variable; for example:

```
...
# define distributions
events = ['red', 'green', 'blue']
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
```

Listing 23.1: Example of defining probability distributions.

We can plot a bar chart of these probabilities to compare them directly as probability histograms. The complete example is listed below.

```
# plot of distributions
from matplotlib import pyplot
# define distributions
events = ['red', 'green', 'blue']
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
print('P=%.3f Q=%.3f' % (sum(p), sum(q)))
# plot first distribution
pyplot.subplot(2,1,1)
pyplot.bar(events, p)
# plot second distribution
pyplot.subplot(2,1,2)
pyplot.bar(events, q)
# show the plot
pyplot.show()
```

Listing 23.2: Example of summarizing two probability distributions.

Running the example creates a histogram for each probability distribution, allowing the probabilities for each event to be directly compared. We can see that indeed the distributions are different.

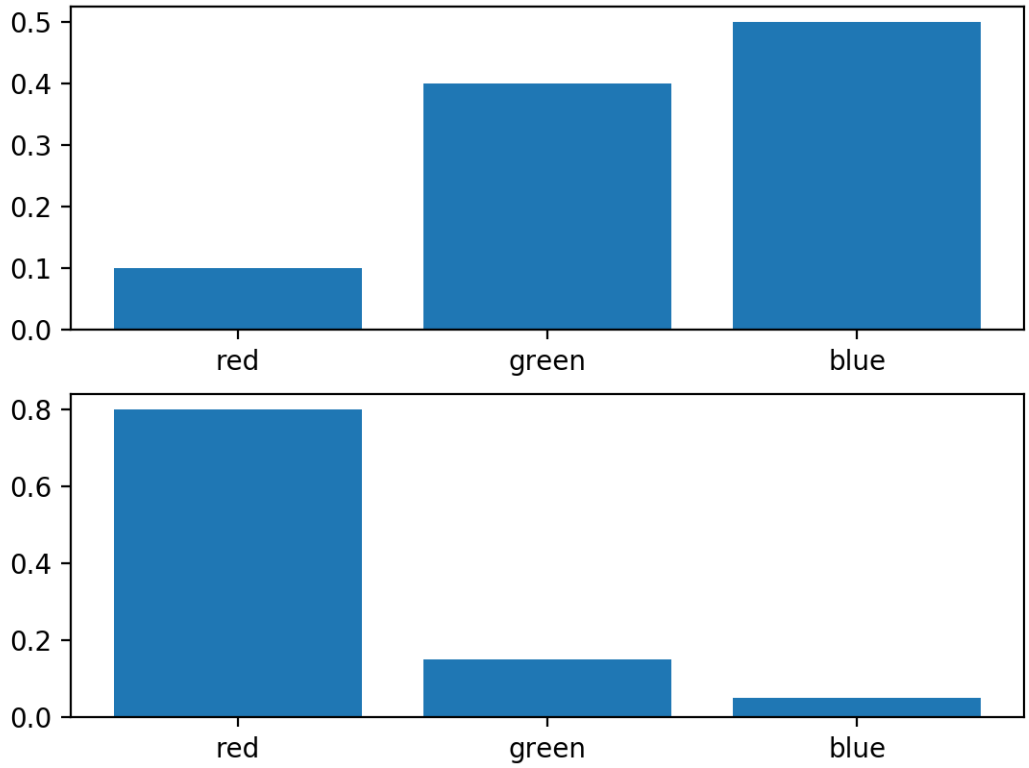


Figure 23.1: Histogram of Two Different Probability Distributions for the Same Random Variable.

### 23.4.2 Calculate Cross-Entropy Between Distributions

Next, we can develop a function to calculate the cross-entropy between the two distributions. We will use log base-2 to ensure the result has units in bits.

```
# calculate cross-entropy
def cross_entropy(p, q):
  return -sum([p[i]*log2(q[i]) for i in range(len(p))])
```

Listing 23.3: Example of defining a function for calculating the cross-entropy.

We can then use this function to calculate the cross-entropy of $P$ from $Q$, as well as the reverse, $Q$ from $P$.

```
...
# calculate cross-entropy H(P, Q)
ce_pq = cross_entropy(p, q)
print('H(P, Q): %.3f bits' % ce_pq)
# calculate cross-entropy H(Q, P)
ce_qp = cross_entropy(q, p)
print('H(Q, P): %.3f bits' % ce_qp)
```

Listing 23.4: Example of calculating the cross-entropy.

Tying this all together, the complete example is listed below.

```
# example of calculating cross-entropy
from math import log2

# calculate cross-entropy
def cross_entropy(p, q):
  return -sum([p[i]*log2(q[i]) for i in range(len(p))])

# define data
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
# calculate cross-entropy H(P, Q)
ce_pq = cross_entropy(p, q)
print('H(P, Q): %.3f bits' % ce_pq)
# calculate cross-entropy H(Q, P)
ce_qp = cross_entropy(q, p)
print('H(Q, P): %.3f bits' % ce_qp)
```

Listing 23.5: Example of calculating the cross-entropy between two probability distributions.

Running the example first calculates the cross-entropy of $Q$ from $P$ as just over 3 bits, then $P$ from $Q$ as just under 3 bits.

```
H(P, Q): 3.288 bits
H(Q, P): 2.906 bits
```

Listing 23.6: Example output from calculating the cross-entropy between two probability distributions in bits.

### 23.4.3 Calculate Cross-Entropy Between a Distribution and Itself

If two probability distributions are the same, then the cross-entropy between them will be the entropy of the distribution. We can demonstrate this by calculating the cross-entropy of $P$ vs $P$ and $Q$ vs $Q$. The complete example is listed below.

```
# example of calculating cross entropy for identical distributions
from math import log2

# calculate cross entropy
def cross_entropy(p, q):
  return -sum([p[i]*log2(q[i]) for i in range(len(p))])

# define data
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
# calculate cross entropy H(P, P)
ce_pp = cross_entropy(p, p)
print('H(P, P): %.3f bits' % ce_pp)
# calculate cross entropy H(Q, Q)
ce_qq = cross_entropy(q, q)
print('H(Q, Q): %.3f bits' % ce_qq)
```

Listing 23.7: Example of calculating the cross-entropy between a probability distribution and itself.

Running the example first calculates the cross-entropy of $Q$ vs $Q$ which is calculated as the entropy for $Q$, and $P$ vs $P$ which is calculated as the entropy for $P$.

```
H(P, P): 1.361 bits
H(Q, Q): 0.884 bits
```

Listing 23.8: Example output from calculating the cross-entropy between a probability distribution and itself.

### 23.4.4 Calculate Cross-Entropy Using KL Divergence

We can also calculate the cross-entropy using the KL divergence. The cross-entropy calculated with KL divergence should be identical to the method above, and it may be interesting to calculate the KL divergence between the distributions as well to see the relative entropy or additional bits required instead of the total bits calculated by the cross-entropy. First, we can define a function to calculate the KL divergence between the distributions using log base-2 to ensure the result is also in bits.

```
# calculate the kl divergence KL(P || Q)
def kl_divergence(p, q):
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))
```

Listing 23.9: Example of defining a function for calculating the KL divergence.

Next, we can define a function to calculate the entropy for a given probability distribution.

```
# calculate entropy H(P)
def entropy(p):
  return -sum([p[i] * log2(p[i]) for i in range(len(p))])
```

Listing 23.10: Example of defining a function for calculating the entropy.

Finally, we can calculate the cross-entropy using the `entropy()` and `kl_divergence()` functions.

```
# calculate cross-entropy H(P, Q)
def cross_entropy(p, q):
  return entropy(p) + kl_divergence(p, q)
```

Listing 23.11: Example of defining a function for calculating the cross-entropy using KL divergence.

To keep the example simple, we can compare the cross-entropy for $H(P, Q)$ to the KL divergence $KL(P||Q)$ and the entropy $H(P)$. The complete example is listed below.

```
# example of calculating cross-entropy with kl divergence
from math import log2

# calculate the kl divergence KL(P || Q)
def kl_divergence(p, q):
  return sum(p[i] * log2(p[i]/q[i]) for i in range(len(p)))

# calculate entropy H(P)
def entropy(p):
  return -sum([p[i] * log2(p[i]) for i in range(len(p))])
```

```
# calculate cross-entropy H(P, Q)
def cross_entropy(p, q):
  return entropy(p) + kl_divergence(p, q)

# define data
p = [0.10, 0.40, 0.50]
q = [0.80, 0.15, 0.05]
# calculate H(P)
en_p = entropy(p)
print('H(P): %.3f bits' % en_p)
# calculate kl divergence KL(P || Q)
kl_pq = kl_divergence(p, q)
print('KL(P || Q): %.3f bits' % kl_pq)
# calculate cross-entropy H(P, Q)
ce_pq = cross_entropy(p, q)
print('H(P, Q): %.3f bits' % ce_pq)
```

Listing 23.12: Example of calculating the cross-entropy between two probability distributions using the KL divergence.

Running the example, we can see that the cross-entropy score of 3.288 bits is comprised of the entropy of $P$ 1.361 and the additional 1.927 bits calculated by the KL divergence. This is a useful example that clearly illustrates the relationship between all three calculations.

```
H(P): 1.361 bits
KL(P || Q): 1.927 bits
H(P, Q): 3.288 bits
```

Listing 23.13: Example output from calculating the cross-entropy between two probability distributions using the KL divergence.

## 23.5 Cross-Entropy as a Loss Function

Cross-entropy is widely used as a loss function when optimizing classification models. Two examples that you may encounter include the logistic regression algorithm (a linear classification algorithm), and artificial neural networks that can be used for classification tasks.

> ... using the cross-entropy error function instead of the sum-of-squares for a classification problem leads to faster training as well as improved generalization.
>
> — Page 235, *Pattern Recognition and Machine Learning*, 2006.

Classification problems are those that involve one or more input variables and the prediction of a class label. Classification tasks that have just two labels for the output variable are referred to as binary classification problems, whereas those problems with more than two labels are referred to as categorical or multiclass classification problems.

- **Binary Classification**: Task of predicting one of two class labels for a given example.
- **Multiclass Classification**: Task of predicting one of more than two class labels for a given example.

We can see that the idea of cross-entropy may be useful for optimizing a classification model. Each example has a known class label with a probability of 1.0, and a probability of 0.0 for all other labels. A model can estimate the probability of an example belonging to each class label. Cross-entropy can then be used to calculate the difference between the two probability distributions. As such, we can map the classification of one example onto the idea of a random variable with a probability distribution as follows:

- **Random Variable**: The example for which we require a predicted class label.
- **Events**: Each class label that could be predicted.

In classification tasks, we know the target probability distribution $P$ for an input as the class label 0 or 1 interpreted as probabilities as *impossible* and *certain* respectively. These probabilities have no surprise at all, therefore they have no information content or zero entropy. Our model seeks to approximate the target probability distribution $Q$. In the language of classification, these are the actual and the predicted probabilities, or $y$ and $yhat$.

- **Expected Probability** ($y$): The known probability of each class label for an example in the dataset ($P$).
- **Predicted Probability** ($yhat$): The probability of each class label an example predicted by the model ($Q$).

We can, therefore, estimate the cross-entropy for a single prediction using the cross-entropy calculation described above; for example.

$$H(P, Q) = -\sum^{\{x \in X\}} P(x) \times \log(Q(x)) \tag{23.7}$$

Where each $x \in X$ is a class label that could be assigned to the example, and $P(x)$ will be 1 for the known label and 0 for all other labels. The cross-entropy for a single example in a binary classification task can be stated by unrolling the sum operation as follows:

$$H(P, Q) = -(P(class0) \times \log(Q(class0)) + P(class1) \times \log(Q(class1))) \tag{23.8}$$

You may see this form of calculating cross-entropy cited in textbooks. If there are just two class labels, the probability is modeled as the Bernoulli distribution for the positive class label. This means that the probability for class 1 is predicted by the model directly, and the probability for class 0 is given as one minus the predicted probability, for example:

- **Predicted** $P(class0) = 1 - yhat$
- **Predicted** $P(class1) = yhat$

When calculating cross-entropy for classification tasks, the base-e or natural logarithm is used. This means that the units are in nats, not bits. We are often interested in minimizing the cross-entropy for the model across the entire training dataset. This is calculated by calculating the average cross-entropy across all training examples.

### 23.5.1 Calculate Entropy for Class Labels

Recall that when two distributions are identical, the cross-entropy between them is equal to the entropy for the probability distribution. Class labels are encoded using the values 0 and 1 when preparing data for classification tasks. For example, if a classification problem has three classes, and an example has a label for the first class, then the probability distribution will be [1, 0, 0]. If an example has a label for the second class, it will have a probability distribution for the two events as [0, 1, 0]. This is called a one hot encoding.

This probability distribution has no information as the outcome is certain. We know the class. Therefore the entropy for this variable is zero. This is an important concept and we can demonstrate it with a worked example. Pretend with have a classification problem with 3 classes, and we have one example that belongs to each class. We can represent each example as a discrete probability distribution with a 1.0 probability for the class to which the example belongs and a 0.0 probability for all other classes. We can calculate the entropy of the probability distribution for each *variable* across the *events*. The complete example is listed below.

```
# entropy of examples from a classification task with 3 classes
from math import log2
from numpy import asarray

# calculate entropy
def entropy(p):
  return -sum([p[i] * log2(p[i]) for i in range(len(p))])

# class 1
p = asarray([1,0,0]) + 1e-15
print(entropy(p))
# class 2
p = asarray([0,1,0]) + 1e-15
print(entropy(p))
# class 3
p = asarray([0,0,1]) + 1e-15
print(entropy(p))
```

Listing 23.14: Example of calculating the entropy of class labels.

Running the example calculates the entropy for each random variable. We can see that in each case, the entropy is 0.0 (actually a number very close to zero). Note that we had to add a very small value to the 0.0 values to avoid the `log()` from blowing up, as we cannot calculate the log of 0.0.

```
9.805612959471341e-14
9.805612959471341e-14
9.805612959471341e-14
```

Listing 23.15: Example output from calculating the entropy of class labels.

As such, the entropy of a known class label is always 0.0. This means that the cross entropy of two distributions (real and predicted) that have the same probability distribution for a class label, will also always be 0.0. Recall that when evaluating a model using cross-entropy on a training dataset that we average the cross-entropy across all examples in the dataset. Therefore, a cross-entropy of 0.0 when training a model indicates that the predicted class probabilities are identical to the probabilities in the training dataset, e.g. zero loss.

We could just as easily minimize the KL divergence as a loss function instead of the cross-entropy. Recall that the KL divergence is the extra bits required to transmit one variable compared to another. It is the cross-entropy without the entropy of the class label, which we know would be zero anyway. As such, minimizing the KL divergence and the cross entropy for a classification task are identical. In practice, a cross-entropy loss of 0.0 often indicates that the model has overfit the training dataset, but that is another story.

### 23.5.2 Cross-Entropy Between Class Labels and Probabilities

The use of cross-entropy for classification often gives different specific names based on the number of classes, mirroring the name of the classification task; for example:

- **Binary Cross-Entropy**: Cross-entropy as a loss function for a binary classification task.
- **Categorical Cross-Entropy**: Cross-entropy as a loss function for a multiclass classification task.

We can make the use of cross-entropy as a loss function concrete with a worked example. Consider a binary classification task with the following 10 actual class labels ($P$) and predicted class labels ($Q$).

```
...
# define classification data
p = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
q = [0.8, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3]
```

Listing 23.16: Example of defining expected values and predicted probabilities.

We can enumerate these probabilities and calculate the cross-entropy for each using the cross-entropy function developed in the previous section using `log()` (natural logarithm) instead of `log2()`.

```
# calculate cross-entropy
def cross_entropy(p, q):
  return -sum([p[i]*log(q[i]) for i in range(len(p))])
```

Listing 23.17: Example of a function for calculating cross-entropy in nats.

For each actual and predicted probability, we must convert the prediction into a distribution of probabilities across each state, in this case, the states $\{0, 1\}$ as one minus the probability for class 0 and the probability for class 1. We can then calculate the cross-entropy and repeat the process for all examples.

```
...
# calculate cross entropy for each example
results = list()
for i in range(len(p)):
  # create the distribution for each event {0, 1}
  expected = [1.0 - p[i], p[i]]
  predicted = [1.0 - q[i], q[i]]
  # calculate cross entropy for the two events
  ce = cross_entropy(expected, predicted)
  print('>[y=%.1f, yhat=%.1f] ce: %.3f nats' % (p[i], q[i], ce))
  results.append(ce)
```

Listing 23.18: Example of calculating the cross-entropy for each examples.

Finally, we can calculate the average cross-entropy across the dataset and report it as the cross-entropy loss for the model on the dataset.

```
...
# calculate the average cross-entropy
mean_ce = mean(results)
print('Average Cross Entropy: %.3f nats' % mean_ce)
```

Listing 23.19: Example of calculating the average cross-entropy.

Tying this all together, the complete example is listed below.

```
# calculate cross entropy for classification problem
from math import log
from numpy import mean

# calculate cross entropy
def cross_entropy(p, q):
  return -sum([p[i]*log(q[i]) for i in range(len(p))])

# define classification data
p = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
q = [0.8, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3]
# calculate cross entropy for each example
results = list()
for i in range(len(p)):
  # create the distribution for each event {0, 1}
  expected = [1.0 - p[i], p[i]]
  predicted = [1.0 - q[i], q[i]]
  # calculate cross entropy for the two events
  ce = cross_entropy(expected, predicted)
  print('>[y=%.1f, yhat=%.1f] ce: %.3f nats' % (p[i], q[i], ce))
  results.append(ce)

# calculate the average cross entropy
mean_ce = mean(results)
print('Average Cross Entropy: %.3f nats' % mean_ce)
```

Listing 23.20: Example of calculating the average cross-entropy between expected and predicted probabilities.

Running the example prints the actual and predicted probabilities for each example and the cross-entropy in nats. The final average cross-entropy loss across all examples is reported, in this case, as 0.247 nats.

```
>[y=1.0, yhat=0.8] ce: 0.223 nats
>[y=1.0, yhat=0.9] ce: 0.105 nats
>[y=1.0, yhat=0.9] ce: 0.105 nats
>[y=1.0, yhat=0.6] ce: 0.511 nats
>[y=1.0, yhat=0.8] ce: 0.223 nats
>[y=0.0, yhat=0.1] ce: 0.105 nats
>[y=0.0, yhat=0.4] ce: 0.511 nats
>[y=0.0, yhat=0.2] ce: 0.223 nats
>[y=0.0, yhat=0.1] ce: 0.105 nats
```

```
>[y=0.0, yhat=0.3] ce: 0.357 nats
Average Cross Entropy: 0.247 nats
```

Listing 23.21: Example output from cross-entropy between expected and predicted probabilities.

This is how cross-entropy loss is calculated when optimizing a logistic regression model or a neural network model under a cross-entropy loss function.

### 23.5.3 Calculate Cross-Entropy Using Keras

We can confirm the same calculation by using the `binary_crossentropy()` function from the Keras deep learning API to calculate the cross-entropy loss for our small dataset. The complete example is listed below. Note that this example assumes that you have the Keras library installed and configured with a backend library such as TensorFlow. If not, you can skip running this example.

```
# calculate cross entropy with keras
from numpy import asarray
from keras import backend
from keras.losses import binary_crossentropy
# prepare classification data
p = asarray([1, 1, 1, 1, 1, 0, 0, 0, 0, 0])
q = asarray([0.8, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3])
# convert to keras variables
y_true = backend.variable(p)
y_pred = backend.variable(q)
# calculate the average cross-entropy
mean_ce = backend.eval(binary_crossentropy(y_true, y_pred))
print('Average Cross Entropy: %.3f nats' % mean_ce)
```

Listing 23.22: Example of calculating the average cross-entropy between expected and predicted probabilities with Keras.

Running the example, we can see that the same average cross-entropy loss of 0.247 nats is reported. This confirms the correct manual calculation of cross-entropy.

```
Average Cross Entropy: 0.247 nats
```

Listing 23.23: Example output from calculating the average cross-entropy between expected and predicted probabilities with Keras.

We can see that we could just as easily minimize the KL divergence as a loss function instead of the cross-entropy. The specific values would be different, but the effect would be the same as the two values are proportional to each other (e.g. minus the constant of the entropy for $P$).

> Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions.
>
> — Page 132, *Deep Learning*, 2016.

## 23.6 Difference Between Cross-Entropy and Log Loss

Cross-Entropy is not Log Loss, but they calculate the same quantity when used as loss functions for binary classification problems (introduced in Chapter 13).

### 23.6.1 Log Loss is the Negative Log Likelihood

Logistic loss refers to the loss function commonly used to optimize a logistic regression model. It may also be referred to as logarithmic loss (which is confusing) or simply log loss. Many models are optimized under a probabilistic framework called maximum likelihood estimation, or MLE, that involves finding a set of parameters that best explain the observed data.

This involves selecting a likelihood function that defines how likely a set of observations (data) are given model parameters. When a log likelihood function is used (which is common), it is often referred to as optimizing the log likelihood for the model. Because it is more common to minimize a function than to maximize it in practice, the log likelihood function is inverted by adding a negative sign to the front. This transforms it into a Negative Log Likelihood function or NLL for short. In deriving the log likelihood function under a framework of maximum likelihood estimation for a Bernoulli probability distribution functions (two classes), the calculation comes out to be:

$$\text{NLL}(P, Q) = -(P(class0) \times \log(Q(class0)) + P(class1) \times \log(Q(class1))) \tag{23.9}$$

This quantity can be averaged over all training examples by calculating the average of the log of the likelihood function. Negative log-likelihood for binary classification problems is often shortened to simply *log loss* as the loss function derived for logistic regression.

$$\text{log loss} = \text{negative log-likelihood, under a Bernoulli probability distribution} \tag{23.10}$$

We can see that the negative log-likelihood is the same calculation as is used for the cross-entropy for Bernoulli probability distribution functions (two events or classes). In fact, the negative log-likelihood for Multinoulli distributions (multiclass classification) also matches the calculation for cross-entropy.

### 23.6.2 Log Loss and Cross Entropy Calculate the Same Thing

For binary classification problems, *log loss*, *cross-entropy* and *negative log-likelihood* are used interchangeably. More generally, the terms *cross-entropy* and *negative log-likelihood* are used interchangeably in the context of loss functions for classification models.

> The negative log-likelihood for logistic regression is given by [...] This is also called the cross-entropy error function.
>
> — Page 246, *Machine Learning: A Probabilistic Perspective*, 2012.

Therefore, calculating log loss will give the same quantity as calculating the cross-entropy for Bernoulli probability distribution. We can confirm this by calculating the log loss using the `log_loss()` function from the scikit-learn API. Calculating the average log loss on the same set of actual and predicted probabilities from the previous section should give the same result as calculating the average cross-entropy. The complete example is listed below.

```
# calculate log loss for classification problem with scikit-learn
from sklearn.metrics import log_loss
from numpy import asarray
# define classification data
p = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
q = [0.8, 0.9, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.1, 0.3]
# define data as expected, e.g. probability for each event {0, 1}
y_true = asarray([[1-v, v] for v in p])
y_pred = asarray([[1-v, v] for v in q])
# calculate the average log loss
ll = log_loss(y_true, y_pred)
print('Average Log Loss: %.3f' % ll)
```

Listing 23.24: Example of calculating the average log loss between expected and predicted probabilities.

Running the example gives the expected result of 0.247 log loss, which matches 0.247 nats when calculated using the average cross-entropy.

```
Average Log Loss: 0.247
```

Listing 23.25: Example output from calculating the average log loss between expected and predicted probabilities.

This does not mean that log loss calculates cross-entropy or cross-entropy calculates log loss. Instead, they are different quantities, arrived at from different fields of study, that under the conditions of calculating a loss function for a classification task, result in an equivalent calculation and result. Specifically, a cross-entropy loss function is equivalent to a maximum likelihood function under a Bernoulli or Multinoulli probability distribution. This demonstrates a connection between the study of maximum likelihood estimation and information theory for discrete probability distributions.

It is not limited to discrete probability distributions, and this fact is surprising to many practitioners that hear it for the first time. Specifically, a linear regression optimized under the maximum likelihood estimation framework assumes a Gaussian continuous probability distribution for the target variable and involves minimizing the mean squared error function. This is equivalent to the cross-entropy for a random variable with a Gaussian probability distribution.

> Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.
>
> — Page 132, *Deep Learning*, 2016.

This is a little mind blowing, and comes from the field of differential entropy for continuous random variables. It means that if you calculate the mean squared error between two Gaussian random variables that cover the same events (have the same mean and standard deviation), then you are calculating the cross-entropy between the variables. It also means that if you are using mean squared error loss to optimize your neural network model for a regression problem, you are in effect using a cross entropy loss.

## 23.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 23.7.1 Books

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Deep Learning*, 2016.
  https://amzn.to/2lnc3vL

### 23.7.2 API

- Usage of loss functions, Keras API.
  https://keras.io/losses/
- `sklearn.metrics.log_loss` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

### 23.7.3 Articles

- Kullback-Leibler divergence, Wikipedia.
  https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
- Cross entropy, Wikipedia.
  https://en.wikipedia.org/wiki/Cross_entropy
- Joint Entropy, Wikipedia.
  https://en.wikipedia.org/wiki/Joint_entropy
- Loss functions for classification, Wikipedia.
  https://en.wikipedia.org/wiki/Loss_functions_for_classification

## 23.8 Summary

In this tutorial, you discovered cross-entropy for machine learning. Specifically, you learned:

- How to calculate cross-entropy from scratch and using standard machine learning libraries.
- Cross-entropy can be used as a loss function when optimizing classification models like logistic regression and artificial neural networks.
- Cross-entropy is different from KL divergence but can be calculated using KL divergence, and is different from log loss but calculates the same quantity when used as a loss function.

### 23.8.1 Next

In the next tutorial, you will discover information gain and mutual information between two probability distributions.

# Chapter 24

# Information Gain and Mutual Information

Information gain calculates the reduction in entropy or surprise from transforming a dataset in some way. It is commonly used in the construction of decision trees from a training dataset, by evaluating the information gain for each variable, and selecting the variable that maximizes the information gain, which in turn minimizes the entropy and best splits the dataset into groups for effective classification. Information gain can also be used for feature selection, by evaluating the gain of each variable in the context of the target variable. In this slightly different usage, the calculation is referred to as mutual information between the two random variables. In this tutorial, you will discover information gain and mutual information in machine learning. After reading this tutorial, you will know:

- Information gain is the reduction in entropy or surprise by transforming a dataset and is often used in training decision trees.
- Information gain is calculated by comparing the entropy of the dataset before and after a transformation.
- Mutual information calculates the statistical dependence between two variables and is the name given to information gain when applied to variable selection.

Let's get started.

## 24.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. What Is Information Gain?
2. Worked Example of Calculating Information Gain
3. Examples of Information Gain in Machine Learning
4. What Is Mutual Information?
5. How Are Information Gain and Mutual Information Related?

## 24.2 What Is Information Gain?

Information Gain, or IG for short, measures the reduction in entropy or surprise by splitting a dataset according to a given value of a random variable. A larger information gain suggests a lower entropy group or groups of samples, and hence less surprise. You might recall that information quantifies how surprising an event is in bits. Lower probability events have more information, higher probability events have less information. Entropy quantifies how much information there is in a random variable, or more specifically its probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy.

In information theory, we like to describe the *surprise* of an event. Low probability events are more surprising therefore have a larger amount of information. Whereas probability distributions where the events are equally likely are more surprising and have larger entropy.

- **Skewed Probability Distribution** (*unsurprising*): Low entropy.
- **Balanced Probability Distribution** (*surprising*): High entropy.

Now, let's consider the entropy of a dataset. We can think about the entropy of a dataset in terms of the probability distribution of observations in the dataset belonging to one class or another, e.g. two classes in the case of a binary classification dataset.

> One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of $S$ (i.e., a member of $S$ drawn at random with uniform probability).
>
> — Page 58, *Machine Learning*, 1997.

For example, in a binary classification problem (two classes), we can calculate the entropy of the data sample as follows:

$$Entropy = -(p(0) \times \log(P(0)) + p(1) \times \log(P(1))) \tag{24.1}$$

A dataset with a 50/50 split of samples for the two classes would have a maximum entropy (maximum surprise) of 1 bit, whereas an imbalanced dataset with a split of 10/90 would have a smaller entropy as there would be less surprise for a randomly drawn example from the dataset. We can demonstrate this with an example of calculating the entropy for this imbalanced dataset in Python. The complete example is listed below.

```
# calculate the entropy for a dataset
from math import log2
# proportion of examples in each class
class0 = 10/100
class1 = 90/100
# calculate entropy
entropy = -(class0 * log2(class0) + class1 * log2(class1))
# print the result
print('entropy: %.3f bits' % entropy)
```

Listing 24.1: Example of calculating entropy.

Running the example, we can see that entropy of the dataset for binary classification is less than 1 bit. That is, less than one bit of information is required to encode the class label for an arbitrary example from the dataset.

```
entropy: 0.469 bits
```

Listing 24.2: Example output from calculating entropy.

In this way, entropy can be used as a calculation of the purity of a dataset, e.g. how balanced the distribution of classes happens to be. An entropy of 0 bits indicates a dataset containing one class; an entropy of 1 or more bits suggests maximum entropy for a balanced dataset (depending on the number of classes), with values in between indicating levels between these extremes. Information gain provides a way to use entropy to calculate how a change to the dataset impacts the purity of the dataset, e.g. the distribution of classes. A smaller entropy suggests more purity or less surprise.

> ... information gain, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute.
>
> — Page 57, *Machine Learning*, 1997.

For example, we may wish to evaluate the impact on purity by splitting a dataset S by a random variable with a range of values. This can be calculated as follows:

$$IG(S, a) = H(S) - H(S|a) \tag{24.2}$$

Where $IG(S, a)$ is the information for the dataset $S$ for the variable a for a random variable, $H(S)$ is the entropy for the dataset before any change (described above) and $H(S|a)$ is the conditional entropy for the dataset given the variable a. This calculation describes the gain in the dataset $S$ for the variable $a$. It is the number of bits saved when transforming the dataset. The conditional entropy can be calculated by splitting the dataset into groups for each observed value of $a$ and calculating the sum of the ratio of examples in each group out of the entire dataset multiplied by the entropy of each group.

$$H(S|a) = \sum^{\{v \in a\}} \frac{Sa(v)}{S} \times H(Sa(v)) \tag{24.3}$$

Where $\frac{Sa(v)}{S}$ is the ratio of the number of examples in the dataset with variable $a$ has the value $v$, and $H(Sa(v))$ is the entropy of group of samples where variable a has the value $v$. This might sound a little confusing. We can make the calculation of information gain concrete with a worked example.

## 24.3 Worked Example of Calculating Information Gain

In this section, we will make the calculation of information gain concrete with a worked example. We can define a function to calculate the entropy of a group of samples based on the ratio of samples that belong to class 0 and class 1.

```
# calculate the entropy for the split in the dataset
def entropy(class0, class1):
  return -(class0 * log2(class0) + class1 * log2(class1))
```

Listing 24.3: Example of defining a function for calculating entropy.

Now, consider a dataset with 20 examples, 13 for class 0 and 7 for class 1. We can calculate the entropy for this dataset, which will have less than 1 bit.

```
...
# split of the main dataset
class0 = 13 / 20
class1 = 7 / 20
# calculate entropy before the change
s_entropy = entropy(class0, class1)
print('Dataset Entropy: %.3f bits' % s_entropy)
```

Listing 24.4: Example of calculating the entropy for the dataset.

Now consider that one of the variables in the dataset has two unique values, say `value1` and `value2`. We are interested in calculating the information gain of this variable. Let's assume that if we split the dataset by `value1`, we have a group of eight samples, seven for class 0 and one for class 1. We can then calculate the entropy of this group of samples.

```
...
# split 1 (split via value1)
s1_class0 = 7 / 8
s1_class1 = 1 / 8
# calculate the entropy of the first group
s1_entropy = entropy(s1_class0, s1_class1)
print('Group1 Entropy: %.3f bits' % s1_entropy)
```

Listing 24.5: Example of calculating the entropy for split 1.

Now, let's assume that we split the dataset by `value2`; we have a group of 12 samples with six in each group. We would expect this group to have an entropy of 1.

```
...
# split 2 (split via value2)
s2_class0 = 6 / 12
s2_class1 = 6 / 12
# calculate the entropy of the second group
s2_entropy = entropy(s2_class0, s2_class1)
print('Group2 Entropy: %.3f bits' % s2_entropy)
```

Listing 24.6: Example of calculating the entropy for split 2.

Finally, we can calculate the information gain for this variable based on the groups created for each value of the variable and the calculated entropy. The first variable resulted in a group of eight examples from the dataset, and the second group had the remaining 12 samples in the data set. Therefore, we have everything we need to calculate the information gain. In this case, information gain can be calculated as:

$$H(Dataset) - \frac{Count(Group1)}{Count(Dataset)} \times H(Group1) + \frac{Count(Group2)}{Count(Dataset)} \times H(Group2) \quad (24.4)$$

Or:

$$H(\frac{13}{20}, \frac{7}{20}) - \frac{8}{20} \times H(\frac{7}{8}, \frac{1}{8}) + \frac{12}{20} \times H(\frac{6}{12}, \frac{6}{12}) \tag{24.5}$$

Or in code:

```
...
# calculate the information gain
gain = s_entropy - (8/20 * s1_entropy + 12/20 * s2_entropy)
print('Information Gain: %.3f bits' % gain)
```

Listing 24.7: Example of calculating the information gain.

Tying this all together, the complete example is listed below.

```
# calculate the information gain
from math import log2

# calculate the entropy for the split in the dataset
def entropy(class0, class1):
  return -(class0 * log2(class0) + class1 * log2(class1))

# split of the main dataset
class0 = 13 / 20
class1 = 7 / 20
# calculate entropy before the change
s_entropy = entropy(class0, class1)
print('Dataset Entropy: %.3f bits' % s_entropy)

# split 1 (split via value1)
s1_class0 = 7 / 8
s1_class1 = 1 / 8
# calculate the entropy of the first group
s1_entropy = entropy(s1_class0, s1_class1)
print('Group1 Entropy: %.3f bits' % s1_entropy)

# split 2 (split via value2)
s2_class0 = 6 / 12
s2_class1 = 6 / 12
# calculate the entropy of the second group
s2_entropy = entropy(s2_class0, s2_class1)
print('Group2 Entropy: %.3f bits' % s2_entropy)

# calculate the information gain
gain = s_entropy - (8/20 * s1_entropy + 12/20 * s2_entropy)
print('Information Gain: %.3f bits' % gain)
```

Listing 24.8: Example of calculating information gain for two splits of a dataset.

First, the entropy of the dataset is calculated at just under 1 bit. Then the entropy for the first and second groups are calculated at about 0.5 and 1 bits respectively. Finally, the information gain for the variable is calculated as 0.117 bits. That is, the gain to the dataset by splitting it via the chosen variable is 0.117 bits.

```
Dataset Entropy: 0.934 bits
Group1 Entropy: 0.544 bits
Group2 Entropy: 1.000 bits
```

```
Information Gain: 0.117 bits
```

Listing 24.9: Example output from calculating information gain for two splits of a dataset.

## 24.4 Examples of Information Gain in Machine Learning

Perhaps the most popular use of information gain in machine learning is in decision trees. An example is the Iterative Dichotomiser 3 algorithm, or ID3 for short, used to construct a decision tree.

> Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree.
>
> — Page 58, *Machine Learning*, 1997.

The information gain is calculated for each variable in the dataset. The variable that has the largest information gain is selected to split the dataset. Generally, a larger gain indicates a smaller entropy or less surprise.

> Note that minimizing the entropy is equivalent to maximizing the information gain ...
>
> — Page 547, *Machine Learning: A Probabilistic Perspective*, 2012.

The process is then repeated on each created group, excluding the variable that was already chosen. This stops once a desired depth to the decision tree is reached or no more splits are possible.

> The process of selecting a new attribute and partitioning the training examples is now repeated for each non terminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree.
>
> — Page 60, *Machine Learning*, 1997.

Information gain can be used as a split criterion in most modern implementations of decision trees, such as the implementation of the Classification and Regression Tree (CART) algorithm in the scikit-learn Python machine learning library in the `DecisionTreeClassifier` class for classification. This can be achieved by setting the `criterion` argument to `'entropy'` when configuring the model; for example:

```
# example of a decision tree trained with information gain
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(criterion='entropy')
...
```

Listing 24.10: Example of defining a `DecisionTreeClassifier` classifier to use information gain.

Information gain can also be used for feature selection prior to modeling. It involves calculating the information gain between the target variable and each input variable in the training dataset. The Weka machine learning workbench provides an implementation of information gain for feature selection via the `InfoGainAttributeEval` class. In this context of feature selection, information gain may be referred to as *mutual information* and calculate the statistical dependence between two variables. An example of using information gain (mutual information) for feature selection is the `mutual_info_classif()` scikit-learn function.

## 24.5 What Is Mutual Information?

Mutual information is calculated between two variables and measures the reduction in uncertainty for one variable given a known value of the other variable.

> A quantity called mutual information measures the amount of information one can obtain from one random variable given another.

— Page 310, *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.

The mutual information between two random variables $X$ and $Y$ can be stated formally as follows:

$$I(X;Y) = H(X) - H(X|Y) \tag{24.6}$$

Where $I(X;Y)$ is the mutual information for $X$ and $Y$, $H(X)$ is the entropy for $X$ and $H(X|Y)$ is the conditional entropy for $X$ given $Y$. The result has the units of bits. Mutual information is a measure of dependence or *mutual dependence* between two random variables. As such, the measure is symmetrical, meaning that $I(X;Y) = I(Y;X)$.

> It measures the average reduction in uncertainty about $x$ that results from learning the value of $y$; or vice versa, the average amount of information that $x$ conveys about $y$.

— Page 139, *Information Theory, Inference, and Learning Algorithms*, 2003.

Kullback-Leibler, or KL, divergence is a measure that calculates the difference between two probability distributions. The mutual information can also be calculated as the KL divergence between the joint probability distribution and the product of the marginal probabilities for each variable.

> If the variables are not independent, we can gain some idea of whether they are 'close' to being independent by considering the Kullback-Leibler divergence between the joint distribution and the product of the marginals [...] which is called the mutual information between the variables

— Page 57, *Pattern Recognition and Machine Learning*, 2006.

This can be stated formally as follows:

$$I(X;Y) = KL(p(X,Y)||p(X) \times p(Y)) \tag{24.7}$$

Mutual information is always larger than or equal to zero, where the larger the value, the greater the relationship between the two variables. If the calculated result is zero, then the variables are independent. Mutual information is often used as a general form of a correlation coefficient, e.g. a measure of the dependence between random variables. It is also used as an aspect in some machine learning algorithms. A common example is the Independent Component Analysis, or ICA for short, that provides a projection of statistically independent components of a dataset.

## 24.6 How Are Information Gain and Mutual Information Related?

Mutual Information and Information Gain are the same thing, although the context or usage of the measure often gives rise to the different names. For example:

- Effect of Transforms to a Dataset (*decision trees*): Information Gain.
- Dependence Between Variables (*feature selection*): Mutual Information.

Notice the similarity in the way that the mutual information is calculated and the way that information gain is calculated; they are equivalent:

$$I(X;Y) = H(X) - H(X|Y) \tag{24.8}$$

and

$$IG(S,a) = H(S) - H(S|a) \tag{24.9}$$

As such, mutual information is sometimes used as a synonym for information gain. Technically, they calculate the same quantity if applied to the same data. We can understand the relationship between the two as the more the difference in the joint and marginal probability distributions (mutual information), the larger the gain in information (information gain).

## 24.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 24.7.1 Books

- *Information Theory, Inference, and Learning Algorithms*, 2003.
  https://amzn.to/2KfDDF7
- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP

- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning*, 1997.
  https://amzn.to/2jWd51p
- *Data Mining: Practical Machine Learning Tools and Techniques*, 4th edition, 2016.
  https://amzn.to/2lnW5S7

### 24.7.2 API

- `scipy.stats.entropy` API.
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.entropy.html

### 24.7.3 Articles

- Entropy (information theory), Wikipedia.
  https://en.wikipedia.org/wiki/Entropy_(information_theory)
- Information gain in decision trees, Wikipedia.
  https://en.wikipedia.org/wiki/Information_gain_in_decision_trees
- ID3 algorithm, Wikipedia.
  https://en.wikipedia.org/wiki/ID3_algorithm
- Information gain ratio, Wikipedia.
  https://en.wikipedia.org/wiki/Information_gain_ratio
- Mutual Information, Wikipedia.
  https://en.wikipedia.org/wiki/Mutual_information

## 24.8 Summary

In this tutorial, you discovered information gain and mutual information in machine learning. Specifically, you learned:

- Information gain is the reduction in entropy or surprise by transforming a dataset and is often used in training decision trees.
- Information gain is calculated by comparing the entropy of the dataset before and after a transformation.
- Mutual information calculates the statistical dependence between two variables and is the name given to information gain when applied to variable selection.

### 24.8.1 Next

This was the final tutorial in this Part. In the next Part, you will discover useful probability tools when working with classification predictive modeling problems.

# Part VIII

# Classification

# Chapter 25

# How to Develop and Evaluate Naive Classifier Strategies

A Naive Classifier is a simple classification model that assumes little to nothing about the problem and the performance of which provides a baseline by which all other models evaluated on a dataset can be compared. There are different strategies that can be used for a naive classifier, and some are better than others, depending on the dataset and the choice of performance measures. The most common performance measure is classification accuracy and common naive classification strategies, including randomly guessing class labels, randomly choosing labels from a training dataset, and using a majority class label. It is useful to develop a small probability framework to calculate the expected performance of a given naive classification strategy and to perform experiments to confirm the theoretical expectations. These exercises provide an intuition both for the behavior of naive classification algorithms in general, and the importance of establishing a performance baseline for a classification task. In this tutorial, you will discover how to develop and evaluate naive classification strategies for machine learning. After completing this tutorial, you will know:

- The performance of naive classification models provides a baseline by which all other models can be deemed skillful or not.
- The majority class classifier achieves better accuracy than other naive classifier models such as random guessing and predicting a randomly selected observed class label.
- Naive classifier strategies can be used on predictive modeling projects via the `DummyClassifier` class in the scikit-learn library.

Let's get started.

## 25.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Naive Classifier
2. Predict a Random Guess

3. Predict a Randomly Selected Class
4. Predict the Majority Class
5. Naive Classifiers in scikit-learn

## 25.2 Naive Classifier

Classification predictive modeling problems involve predicting a class label given an input to the model. Classification models are fit on a training dataset and evaluated on a test dataset, and performance is often reported as a fraction of the number of correct predictions compared to the total number of predictions made, called accuracy. Given a classification model, how do you know if the model has skill or not? This is a common question on every classification predictive modeling project. The answer is to compare the results of a given classifier model to a baseline or naive classifier model.

A naive classifier model is one that does not use any sophistication in order to make a prediction, typically making a random or constant prediction. Such models are naive because they don't use any knowledge about the domain or any learning in order to make a prediction. The performance of a baseline classifier on a classification task provides a lower bound on the expected performance of all other models on the problem. For example, if a classification model performs better than a naive classifier, then it has some skill. If a classifier model performs worse than the naive classifier, it does not have any skill. What classifier should be used as the naive classifier? This is a common area of confusion for beginners, and different naive classifiers are adopted. Some common choices include:

- Predict a random class.
- Predict a randomly selected class from the training dataset.
- Predict the majority class from the training dataset.

The problem is, not all naive classifiers are created equal, and some perform better than others. As such, we should use the best-performing naive classifier on all of our classification predictive modeling projects. We can use simple probability to evaluate the performance of different naive classifier models and confirm the one strategy that should always be used as the native classifier. Before we start evaluating different strategies, let's define a contrived two-class classification problem. To make it interesting, we will assume that the number of observations is not equal for each class (e.g. the problem is imbalanced) with 25 examples for class-0 and 75 examples for class-1. We can make this concrete with a small example in Python, listed below.

```
# summarize a test dataset
# define dataset
class0 = [0 for _ in range(25)]
class1 = [1 for _ in range(75)]
y = class0 + class1
# summarize distribution
print('Class 0: %.3f' % (len(class0) / len(y) * 100))
print('Class 1: %.3f' % (len(class1) / len(y) * 100))
```

Listing 25.1: Example of defining and summarizing a test dataset.

Running the example creates the dataset and summarizes the fraction of examples that belong to each class, showing 25% and 75% for class-0 and class-1 as we might intuitively expect.

```
Class 0: 25.000
Class 1: 75.000
```

Listing 25.2: Example output from defining and summarizing a test dataset.

Finally, we can define a probabilistic model for evaluating naive classification strategies. In this case, we are interested in calculating the classification accuracy of a given binary classification model.

$$P(\text{yhat=y}) \tag{25.1}$$

This can be calculated as the probability of the model predicting each class value multiplied by the probability of observing each class occurrence.

$$P(\text{yhat=y}) = P(\text{yhat=0}) \times P(\text{y=0}) + P(\text{yhat=1}) \times P(\text{y=1}) \tag{25.2}$$

This calculates the expected performance of a model on a dataset. It provides a very simple probabilistic model that we can use to calculate the expected performance of a naive classifier model in general. Next, we will use this contrived prediction problem to explore different strategies for a naive classifier.

## 25.3 Predict a Random Guess

Perhaps the simplest strategy is to randomly guess one of the available classes for each prediction that is required. We will call this the random-guess strategy. Using our probabilistic model, we can calculate how well this model is expected to perform on average on our contrived dataset. A random guess for each class is a uniform probability distribution over each possible class label, or in the case of a two-class problem, a probability of 0.5 for each class. Also, we know the expected probability of the values for class-0 and class-1 for our dataset because we contrived the problem; they are 0.25 and 0.75 respectively. Therefore, we calculate the average performance of this strategy as follows:

$$\begin{aligned} P(\text{yhat=y}) &= P(\text{yhat=0}) \times P(\text{y=0}) + P(\text{yhat=1}) \times P(\text{y=1}) \\ &= 0.5 \times 0.25 + 0.5 \times 0.75 \\ &= 0.125 + 0.375 \\ &= 0.5 \end{aligned} \tag{25.3}$$

This calculation suggests that the performance of predicting a uniformly random class label on our contrived problem is 0.5 or 50% classification accuracy. This might be surprising, which is good as it highlights the benefit of systematically calculating the expected performance of a naive strategy. We can confirm that this estimation is correct with a small experiment. The strategy can be implemented as a function that randomly selects a 0 or 1 for each prediction required.

```
# guess random class
def random_guess():
  if random() < 0.5:
```

```
    return 0
  return 1
```

Listing 25.3: Example of a function for predicting a random guess.

This can then be called for each prediction required in the dataset and the accuracy can be evaluated.

```
...
yhat = [random_guess() for _ in range(len(y))]
acc = accuracy_score(y, yhat)
```

Listing 25.4: Example of using the strategy to make predictions.

That is a single trial, but the accuracy will be different each time the strategy is used. To counter this issue, we can repeat the experiment 1,000 times and report the average performance of the strategy. We would expect the average performance to match our expected performance calculated above. The complete example is listed below.

```
# example of a random guess naive classifier
from numpy import mean
from numpy.random import random
from sklearn.metrics import accuracy_score

# guess random class
def random_guess():
  if random() < 0.5:
    return 0
  return 1

# define dataset
class0 = [0 for _ in range(25)]
class1 = [1 for _ in range(75)]
y = class0 + class1
# average performance over many repeats
results = list()
for _ in range(1000):
  yhat = [random_guess() for _ in range(len(y))]
  acc = accuracy_score(y, yhat)
  results.append(acc)
print('Mean: %.3f' % mean(results))
```

Listing 25.5: Example of evaluating a random guess classification strategy.

Running the example performs 1,000 trials of our experiment and reports the mean accuracy of the strategy.

**Note**: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the expected performance very closely matches the calculated performance. Given the law of large numbers, the more trials of this experiment we perform, the closer our estimate will get to the theoretical value we calculated.

```
Mean: 0.499
```

Listing 25.6: Example output from evaluating a random guess classification strategy.

This is a good start, but what if we use some basic information about the composition of the training dataset in the strategy. We will explore that next.

## 25.4 Predict a Randomly Selected Class

Another naive classifier approach is to make use of the training dataset in some way. Perhaps the simplest approach would be to use the observations in the training dataset as predictions. Specifically, we can randomly select observations in the training set and return them for each requested prediction. This makes sense, and we may expect this primitive use of the training dataset would result in a slightly better naive accuracy than randomly guessing.

We can find out by calculating the expected performance of the approach using our probabilistic framework. If we select examples from the training dataset with a uniform probability distribution, we will draw examples from each class with the same probability of their occurrence in the training dataset. That is, we will draw examples of class-0 with a probability of 25% and class-1 with a probability of 75%. This too will be the probability of the independent predictions by the model.

With this knowledge, we can plug-in these values into the probabilistic model.

$$\begin{aligned} P(\text{yhat=y}) &= P(\text{yhat=0}) \times P(\text{y=0}) + P(\text{yhat=1}) \times P(\text{y=1}) \\ &= 0.25 \times 0.25 + 0.75 \times 0.7 \\ &= 0.0625 + 0.562 \\ &= 0.625 \end{aligned} \tag{25.4}$$

The result suggests that using a uniformly randomly selected class from the training dataset as a prediction results in a better naive classifier than simply predicting a uniformly random class on this dataset, showing 62.5% instead of 50%, or a 12.2% lift. Not bad! Let's confirm our calculations again with a small simulation. The `random_class()` function below implements this naive classifier strategy by selecting and returning a random class label from the training dataset.

```
# predict a randomly selected class
def random_class(y):
  return y[randint(len(y))]
```

Listing 25.7: Example of a function for predicting a randomly selected label.

We can then use the same framework from the previous section to evaluate the model 1,000 times and report the average classification accuracy across those trials. We would expect that this empirical estimate would match our expected value, or be very close to it. The complete example is listed below.

```
# example of selecting a random class naive classifier
from numpy import mean
from numpy.random import randint
from sklearn.metrics import accuracy_score

# predict a randomly selected class
def random_class(y):
  return y[randint(len(y))]
```

```
# define dataset
class0 = [0 for _ in range(25)]
class1 = [1 for _ in range(75)]
y = class0 + class1
# average over many repeats
results = list()
for _ in range(1000):
  yhat = [random_class(y) for _ in range(len(y))]
  acc = accuracy_score(y, yhat)
  results.append(acc)
print('Mean: %.3f' % mean(results))
```

Listing 25.8: Example of evaluating a randomly selected class classification strategy.

Running the example performs 1,000 trials of our experiment and reports the mean accuracy of the strategy.

**Note**: Your specific results may vary given the stochastic nature of the learning algorithm. Consider running the example a few times and compare the average performance.

In this case, we can see that the expected performance again very closely matches the calculated performance: 62.4% in the simulation vs. 62.5% that we calculated above.

```
Mean: 0.624
```

Listing 25.9: Example output from evaluating a randomly selected class classification strategy.

Perhaps we can do better than a uniform distribution when predicting a class label. We will explore this in the next section.

## 25.5 Predict the Majority Class

In the previous section, we explored a strategy that selected a class label based on a uniform probability distribution over the observed label in the training dataset. This allowed the predicted probability distribution to match the observed probability distribution for each class and an improvement over a uniform distribution of class labels. A downside to this imbalanced dataset, in particular, is one class is expected above the other to a greater degree and randomly predicting classes, even in a biased way, leads to too many incorrect predictions.

Instead, we can predict the majority class and be assured of achieving an accuracy that is at least as high as the composition of the majority class in the training dataset. That is, if 75% of the examples in the training set are class-1, and we predicted class-1 for all examples, then we know that we would at least achieve an accuracy of 75%, an improvement over randomly selecting a class as we did in the previous section. We can confirm this by calculating the expected performance of the approach using our probability model. The probability of this naive classification strategy predicting class-0 would be 0.0 (impossible), and the probability of predicting class-1 is 1.0 (certain). Therefore:

$$\begin{aligned} P(\text{yhat=y}) &= P(\text{yhat=0}) \times P(\text{y=0}) + P(\text{yhat=1}) \times P(\text{y=1}) \\ &= 0.0 \times 0.25 + 1.0 \times 0.75 \\ &= 0.0 + 0.75 \\ &= 0.75 \end{aligned} \tag{25.5}$$

This confirms our expectations and suggests that this strategy would give a further lift of 12.5% over the previous strategy on this specific dataset. Again, we can confirm this approach with a simulation. The majority class can be calculated statistically using the mode; that is, the most common observation in a distribution. The `mode()` SciPy function can be used. It returns two values, the first of which is the mode that we can return. The `majority_class()` function below implements this naive classifier.

```
# predict the majority class
def majority_class(y):
  return mode(y)[0]
```

Listing 25.10: Example of a function for predicting the majority class.

We can then evaluate the strategy on the contrived dataset. We do not need to repeat the experiment multiple times as there is no random component to the strategy, and the algorithm will give the same performance on the same dataset every time. The complete example is listed below.

```
# example of a majority class naive classifier
from scipy.stats import mode
from sklearn.metrics import accuracy_score

# predict the majority class
def majority_class(y):
  return mode(y)[0]

# define dataset
class0 = [0 for _ in range(25)]
class1 = [1 for _ in range(75)]
y = class0 + class1
# make predictions
yhat = [majority_class(y) for _ in range(len(y))]
# calculate accuracy
accuracy = accuracy_score(y, yhat)
print('Accuracy: %.3f' % accuracy)
```

Listing 25.11: Example of evaluating majority class classification strategy.

Running the example reports the accuracy of the majority class naive classifier on the dataset. The accuracy matches the expected value calculated by the probability framework of 75% and the composition of the training dataset.

```
Accuracy: 0.750
```

Listing 25.12: Example output from evaluating majority class classification strategy.

This majority class naive classifier is the method that should be used to calculate a baseline performance on your classification predictive modeling problems. It works just as well for those datasets with an equal number of class labels, and for problems with more than two class labels, e.g. multiclass classification problems. Now that we have discovered the best-performing naive classifier model, we can see how we might use it in our next project.

## 25.6 Naive Classifiers in scikit-learn

The scikit-learn machine learning library provides an implementation of the majority class naive classification algorithm that you can use on your next classification predictive modeling project. It is provided as part of the `DummyClassifier` class. To use the naive classifier, the class must be defined and the `strategy` argument set to `'most_frequent'` to ensure that the majority class is predicted. The class can then be fit on a training dataset and used to make predictions on a test dataset or other resampling model evaluation strategy.

```
...
# define model
model = DummyClassifier(strategy='most_frequent')
# fit model
model.fit(X, y)
# make predictions
yhat = model.predict(X)
```

Listing 25.13: Example of fitting a `DummyClassifier` classifier.

In fact, the `DummyClassifier` is flexible and allows the other two naive classifiers to be used. Specifically, setting `strategy` to `'uniform'` will perform the random guess strategy that we tested first, and setting `strategy` to `'stratified'` will perform the randomly selected class strategy that we tested second.

- **Random Guess**: Set the `strategy` argument to `'uniform'`.
- **Select Random Class**: Set the `strategy` argument to `'stratified'`.
- **Majority Class**: Set the `strategy` argument to `'most_frequent'`.

We can confirm that the `DummyClassifier` performs as expected with the majority class naive classification strategy by testing it on our contrived dataset. The complete example is listed below.

```
# example of the majority class naive classifier in scikit-learn
from numpy import asarray
from sklearn.dummy import DummyClassifier
from sklearn.metrics import accuracy_score
# define dataset
X = asarray([0 for _ in range(100)])
class0 = [0 for _ in range(25)]
class1 = [1 for _ in range(75)]
y = asarray(class0 + class1)
# reshape data for sklearn
X = X.reshape((len(X), 1))
# define model
model = DummyClassifier(strategy='most_frequent')
# fit model
model.fit(X, y)
# make predictions
yhat = model.predict(X)
# calculate accuracy
accuracy = accuracy_score(y, yhat)
print('Accuracy: %.3f' % accuracy)
```

Listing 25.14: Example of evaluating majority class classification strategy with scikit-learn.

Running the example prepares the dataset, then defines and fits the `DummyClassifier` on the dataset using the majority class strategy. Evaluating the classification accuracy of the predictions from the model confirms that the model performs as expected, achieving a score of 75%.

```
Accuracy: 0.750
```

Listing 25.15: Example output from evaluating majority class classification strategy with scikit-learn.

This example provides a starting point for calculating the naive classifier baseline performance on your own classification predictive modeling projects in the future.

## 25.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `sklearn.dummy.DummyClassifier` API.
  https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html

## 25.8 Summary

In this tutorial, you discovered how to develop and evaluate naive classification strategies for machine learning. Specifically, you learned:

- The performance of naive classification models provides a baseline by which all other models can be deemed skillful or not.
- The majority class classifier achieves better accuracy than other naive classifier models, such as random guessing and predicting a randomly selected observed class label.
- Naive classifier strategies can be used on predictive modeling projects via the `DummyClassifier` class in the scikit-learn library.

### 25.8.1 Next

In the next tutorial, you will discover how to evaluate probabilities predicted by a classification predictive model.

# Chapter 26

# Probability Scoring Metrics

Predicting probabilities instead of class labels for a classification problem can provide additional nuance and uncertainty for the predictions. The added nuance allows more sophisticated metrics to be used to interpret and evaluate the predicted probabilities. In general, methods for the evaluation of the accuracy of predicted probabilities are referred to as scoring rules or scoring functions. In this tutorial, you will discover three scoring methods that you can use to evaluate the predicted probabilities on your classification predictive modeling problem. After completing this tutorial, you will know:

- The log loss score that heavily penalizes predicted probabilities far away from their expected value.
- The Brier score that is gentler than log loss but still penalizes proportional to the distance from the expected value.
- The area under ROC curve that summarizes the likelihood of the model predicting a higher probability for true positive cases than true negative cases.

Let's get started.

## 26.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Log Loss Score
2. Brier Score
3. ROC AUC Score

## 26.2 Log Loss Score

Log loss, also called *logistic loss*, *logarithmic loss*, or *cross-entropy* can be used as a measure for evaluating predicted probabilities (introduced in Chapter 23). Each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the

probability based on the distance from the expected value. The penalty is logarithmic, offering a small score for small differences (0.1 or 0.2) and enormous score for a large difference (0.9 or 1.0). A model with perfect skill has a log loss score of 0.0. In order to summarize the skill of a model using log loss, the log loss is calculated for each predicted probability, and the average loss is reported.

The log loss can be implemented in Python using the `log_loss()` function in scikit-learn. In the binary classification case, the function takes a list of true outcome values and a list of probabilities as arguments and calculates the average log loss for the predictions. We can make a single log loss score concrete with an example. Given a specific known outcome of 0, we can predict values of 0.0 to 1.0 in 0.01 increments (101 predictions) and calculate the log loss for each. The result is a curve showing how much each prediction is penalized as the probability gets further away from the expected value. We can repeat this for a known outcome of 1 and see the same curve in reverse. The complete example is listed below.

```
# plot impact of log loss for single forecasts
from sklearn.metrics import log_loss
from matplotlib import pyplot
# predictions as 0 to 1 in 0.01 increments
yhat = [x*0.01 for x in range(0, 101)]
# evaluate predictions for a 0 true value
losses_0 = [log_loss([0], [x], labels=[0,1]) for x in yhat]
# evaluate predictions for a 1 true value
losses_1 = [log_loss([1], [x], labels=[0,1]) for x in yhat]
# plot input to loss
pyplot.plot(yhat, losses_0, label='true=0')
pyplot.plot(yhat, losses_1, label='true=1')
pyplot.legend()
pyplot.show()
```

Listing 26.1: Example of plotting the change in log loss for each class

Running the example creates a line plot showing the loss scores for probability predictions from 0.0 to 1.0 for both the case where the true label is 0 and 1. This helps to build an intuition for the effect that the loss score has when evaluating predictions.

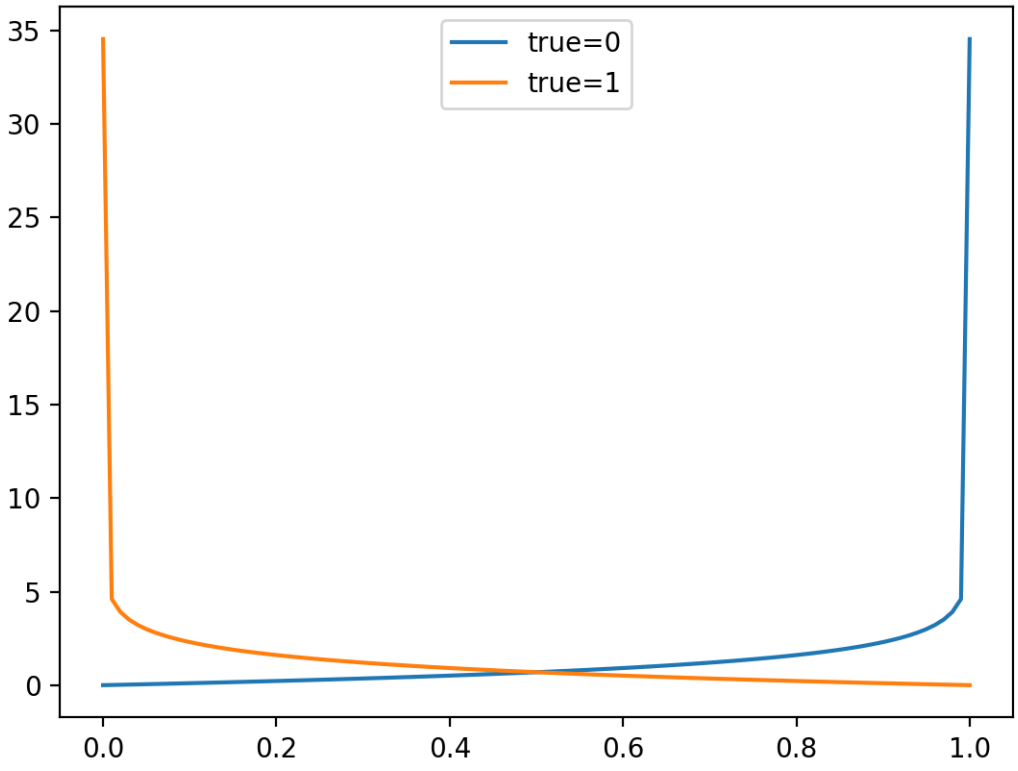


Figure 26.1: Line Plot of Evaluating Predictions with Log Loss.

Model skill is reported as the average log loss across the predictions in a test dataset. As an average, we can expect that the score will be suitable with a balanced dataset and misleading when there is a large imbalance between the two classes in the test set. This is because predicting 0 or small probabilities will result in a small loss. We can demonstrate this by comparing the distribution of loss values when predicting different constant probabilities for a balanced and an imbalanced dataset. First, the example below predicts values from 0.0 to 1.0 in 0.1 increments for a balanced dataset of 50 examples of class 0 and 1.

```
# plot impact of log loss with balanced datasets
from sklearn.metrics import log_loss
from matplotlib import pyplot
# define an imbalanced dataset
testy = [0 for x in range(50)] + [1 for x in range(50)]
# loss for predicting different fixed probability values
predictions = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
losses = [log_loss(testy, [y for x in range(len(testy))]) for y in predictions]
# plot predictions vs loss
pyplot.plot(predictions, losses)
pyplot.show()
```

Listing 26.2: Example of plotting the change in log loss for a balanced dataset.

Running the example, we can see that a model is better-off predicting probability values

that are not sharp (close to the edge) and are back towards the middle of the distribution. The penalty of being wrong with a sharp probability is very large.

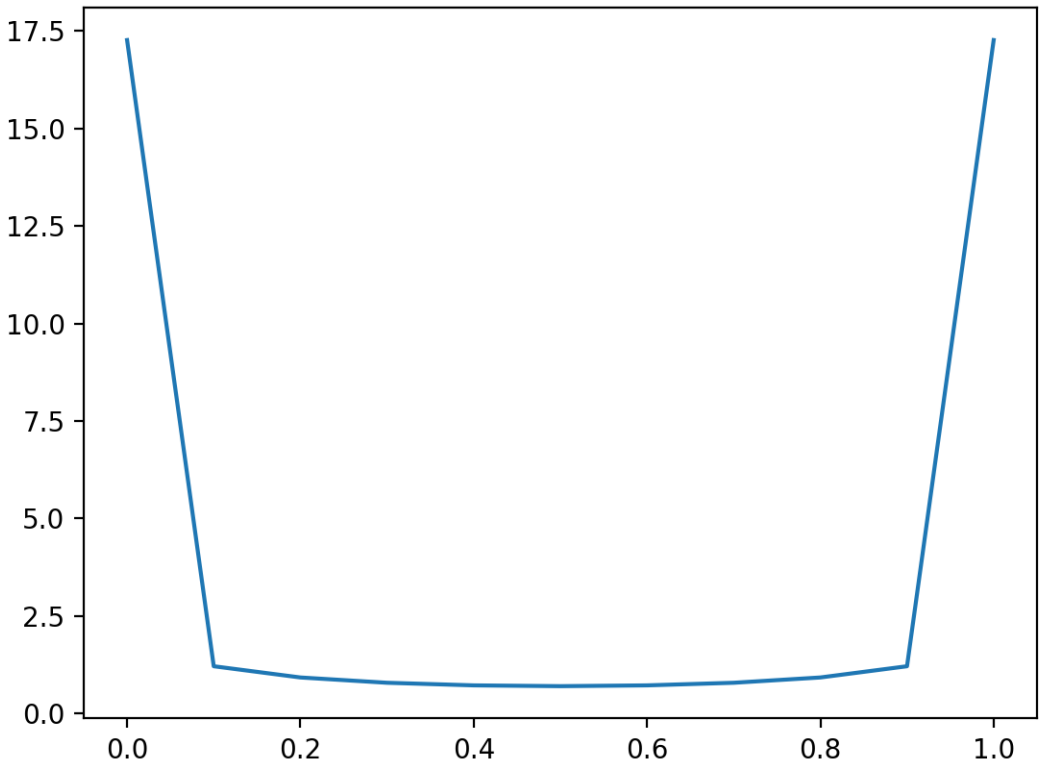


Figure 26.2: Line Plot of Predicting Log Loss for Balanced Dataset.

We can repeat this experiment with an imbalanced dataset with a 10:1 ratio of class 0 to class 1.

```
# plot impact of log loss with imbalanced datasets
from sklearn.metrics import log_loss
from matplotlib import pyplot
# define an imbalanced dataset
testy = [0 for x in range(100)] + [1 for x in range(10)]
# loss for predicting different fixed probability values
predictions = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
losses = [log_loss(testy, [y for x in range(len(testy))]) for y in predictions]
# plot predictions vs loss
pyplot.plot(predictions, losses)
pyplot.show()
```

Listing 26.3: Example of plotting the change in log loss for a imbalanced dataset.

Here, we can see that a model that is skewed towards predicting very small probabilities will perform well, optimistically so. The naive model that predicts a constant probability of 0.1 will be the baseline model to beat. The result suggests that model skill evaluated with log loss

should be interpreted carefully in the case of an imbalanced dataset, perhaps adjusted relative to the base rate for class 1 in the dataset.

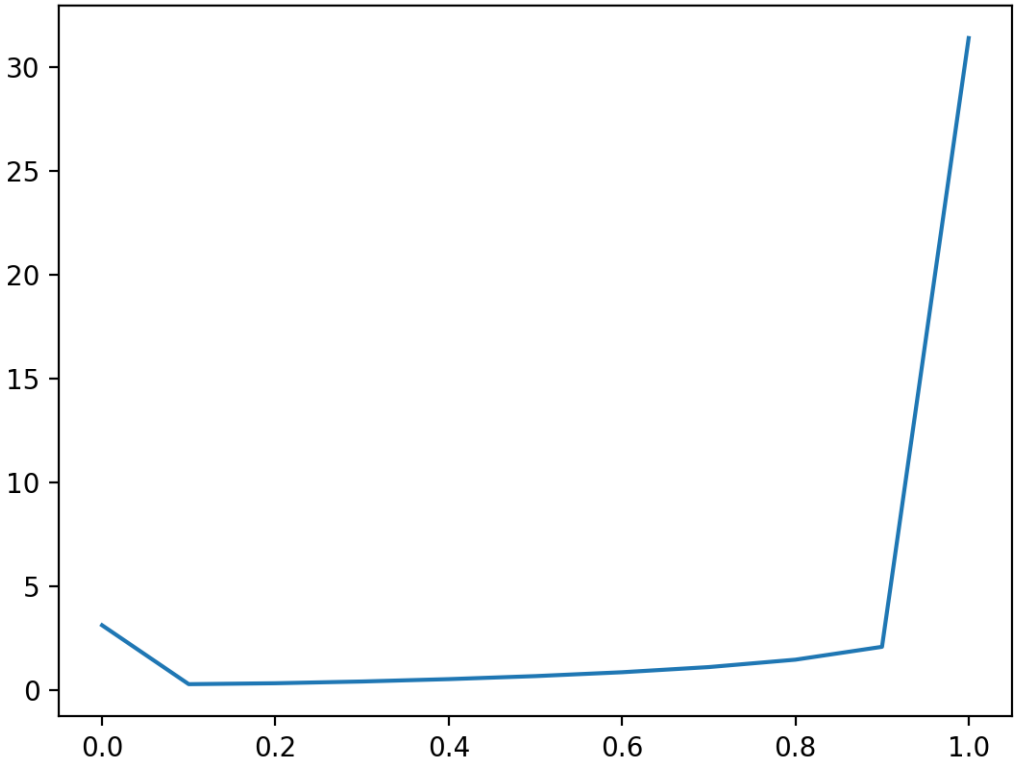


Figure 26.3: Line Plot of Predicting Log Loss for Imbalanced Dataset.

## 26.3 Brier Score

The Brier score, named for Glenn Brier, calculates the mean squared error between predicted probabilities and the expected values. The score summarizes the magnitude of the error in the probability predictions. The error score is always between 0.0 and 1.0, where a model with perfect skill has a score of 0.0. Predictions that are further away from the expected probability are penalized, but less severely as in the case of log loss. The skill of a model can be summarized as the average Brier score across all probabilities predicted for a test dataset.

The Brier score can be calculated in Python using the `brier_score_loss()` function in scikit-learn. It takes the true class values (0, 1) and the predicted probabilities for all examples in a test dataset as arguments and returns the average Brier score. We can evaluate the impact of prediction errors by comparing the Brier score for single probability predictions in increasing error from 0.0 to 1.0. The complete example is listed below.

```
# plot impact of brier for single forecasts
from sklearn.metrics import brier_score_loss
from matplotlib import pyplot
```

```
# predictions as 0 to 1 in 0.01 increments
yhat = [x*0.01 for x in range(0, 101)]
# evaluate predictions for a 1 true value
losses = [brier_score_loss([1], [x], pos_label=[1]) for x in yhat]
# plot input to loss
pyplot.plot(yhat, losses)
pyplot.show()
```

Listing 26.4: Example of plotting the change in brier score for increasing error.

Running the example creates a plot of the probability prediction error in absolute terms (x-axis) to the calculated Brier score (y axis). We can see a familiar quadratic curve, increasing from 0 to 1 with the squared error.

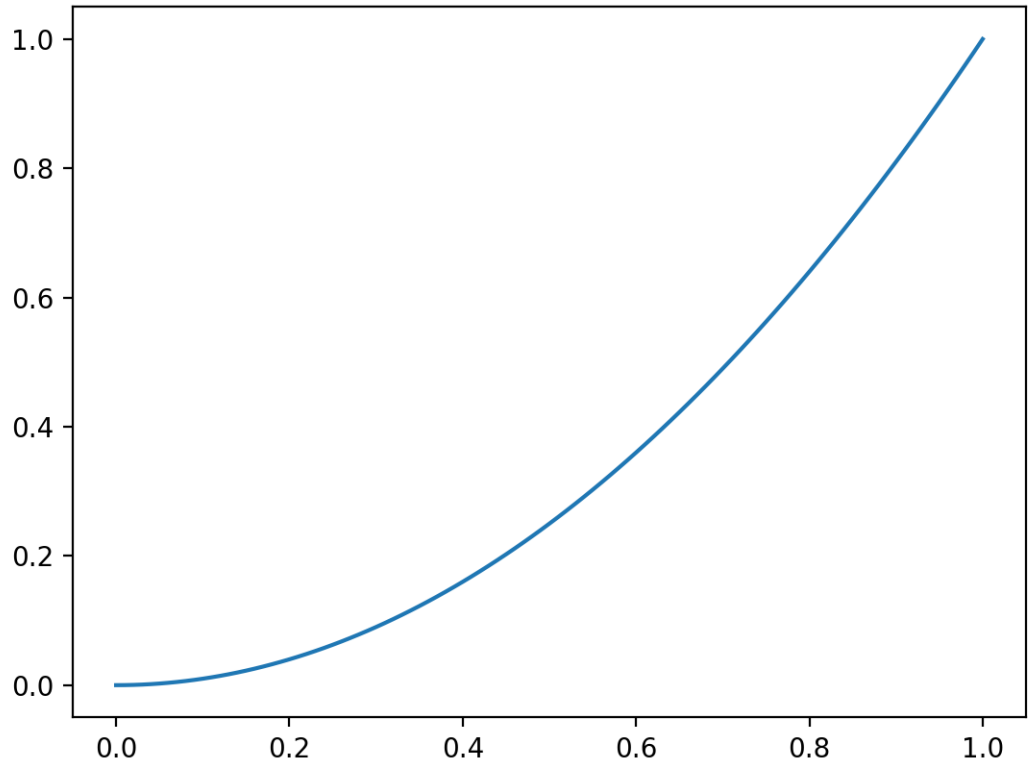


Figure 26.4: Line Plot of Evaluating Predictions with Brier Score.

Model skill is reported as the average Brier across the predictions in a test dataset. As with log loss, we can expect that the score will be suitable with a balanced dataset and misleading when there is a large imbalance between the two classes in the test set. We can demonstrate this by comparing the distribution of loss values when predicting different constant probabilities for a balanced and an imbalanced dataset. First, the example below predicts values from 0.0 to 1.0 in 0.1 increments for a balanced dataset of 50 examples of class 0 and 1.

```
# plot impact of brier score with balanced datasets
```

```
from sklearn.metrics import brier_score_loss
from matplotlib import pyplot
# define an imbalanced dataset
testy = [0 for x in range(50)] + [1 for x in range(50)]
# brier score for predicting different fixed probability values
predictions = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
losses = [brier_score_loss(testy, [y for x in range(len(testy))]) for y in predictions]
# plot predictions vs loss
pyplot.plot(predictions, losses)
pyplot.show()
```

Listing 26.5: Example of plotting the change in brier score for a balanced dataset.

Running the example, we can see that a model is better-off predicting middle of the road probabilities values like 0.5. Unlike log loss that is quite flat for close probabilities, the parabolic shape shows the clear quadratic increase in the score penalty as the error is increased.

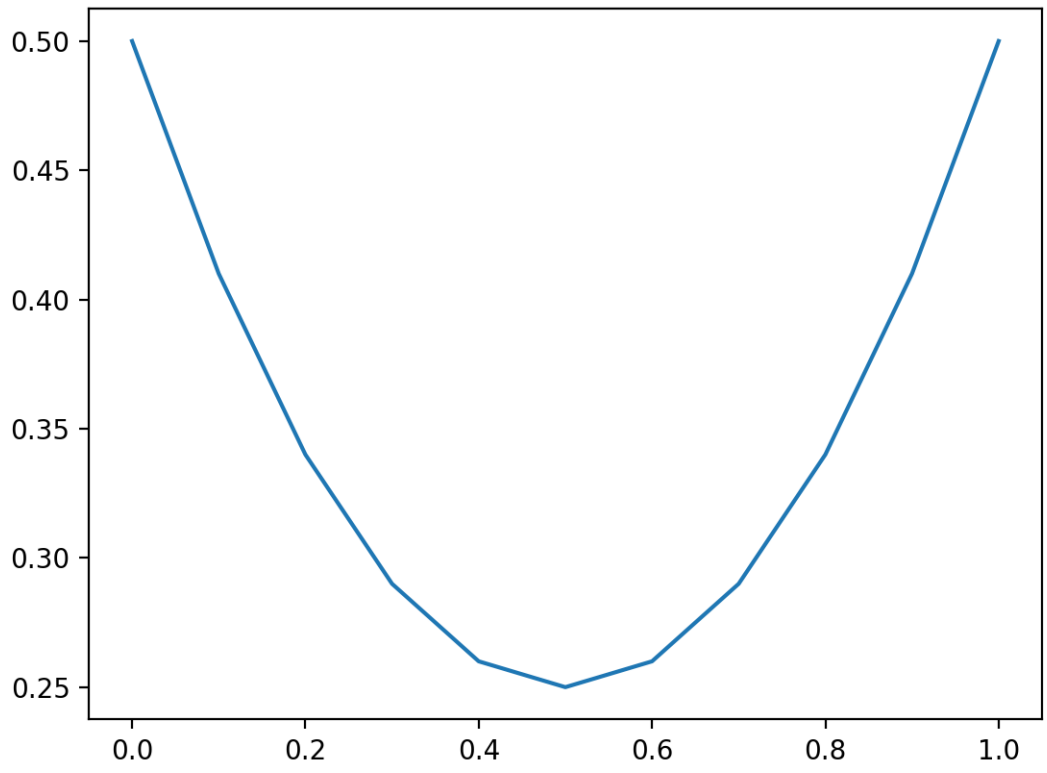


Figure 26.5: Line Plot of Predicting Brier Score for Balanced Dataset.

We can repeat this experiment with an imbalanced dataset with a 10:1 ratio of class 0 to class 1.

```
# plot impact of brier score with imbalanced datasets
from sklearn.metrics import brier_score_loss
from matplotlib import pyplot
```

```
# define an imbalanced dataset
testy = [0 for x in range(100)] + [1 for x in range(10)]
# brier score for predicting different fixed probability values
predictions = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
losses = [brier_score_loss(testy, [y for x in range(len(testy))]) for y in predictions]
# plot predictions vs loss
pyplot.plot(predictions, losses)
pyplot.show()
```

Listing 26.6: Example of plotting the change in brier score for a imbalanced dataset.

Running the example, we see a very different picture for the imbalanced dataset. Like the average log loss, the average Brier score will present optimistic scores on an imbalanced dataset, rewarding small prediction values that reduce error on the majority class. In these cases, the Brier score should be compared relative to the naive prediction (e.g. the base rate of the minority class or 0.1 in the above example) or normalized by the naive score. This latter example is common and is called the Brier Skill Score (BSS).

$$BSS = 1 - (\frac{BS}{BS_{ref}}) \tag{26.1}$$

Where $BS$ is the Brier skill of model, and $BS_{ref}$ is the Brier skill of the naive prediction. The Brier Skill Score reports the relative skill of the probability prediction over the naive forecast. A good update to the scikit-learn API would be to add a parameter to the `brier_score_loss()` to support the calculation of the Brier Skill Score.

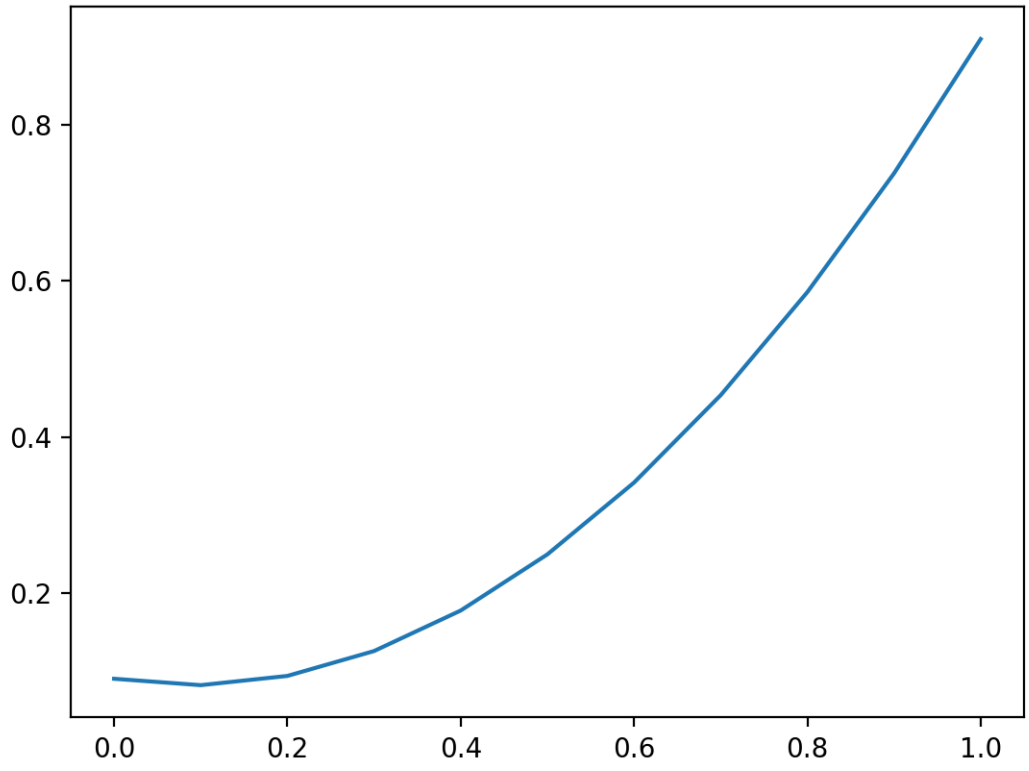


Figure 26.6: Line Plot of Predicting Brier Score for Imbalanced Dataset.

## 26.4 ROC AUC Score

A predicted probability for a binary (two-class) classification problem can be interpreted with a threshold. The threshold defines the point at which the probability is mapped to class 0 versus class 1, where the default threshold is 0.5. Alternate threshold values allow the model to be tuned for higher or lower false positives and false negatives. Tuning the threshold by the operator is particularly important on problems where one type of error is more or less important than another or when a model is makes disproportionately more or less of a specific type of error.

The Receiver Operating Characteristic, or ROC, curve is a plot of the true positive rate versus the false positive rate for the predictions of a model for multiple thresholds between 0.0 and 1.0. Predictions that have no skill for a given threshold are drawn on the diagonal of the plot from the bottom left to the top right. This line represents no-skill predictions for each threshold. Models that have skill have a curve above this diagonal line that bows towards the top left corner. Below is an example of fitting a logistic regression model on a binary classification problem and calculating and plotting the ROC curve for the predicted probabilities on a test set of 500 new data instances.

```
# roc curve
from sklearn.datasets import make_classification
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate roc curve
fpr, tpr, thresholds = roc_curve(testy, probs)
# plot no skill
pyplot.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
pyplot.plot(fpr, tpr)
# show the plot
pyplot.show()
```

Listing 26.7: Example of plotting a ROC curve for predictions.

Running the example creates an example of a ROC curve that can be compared to the no skill line on the main diagonal. ROC curves are covered in more detail in Chapter 27.

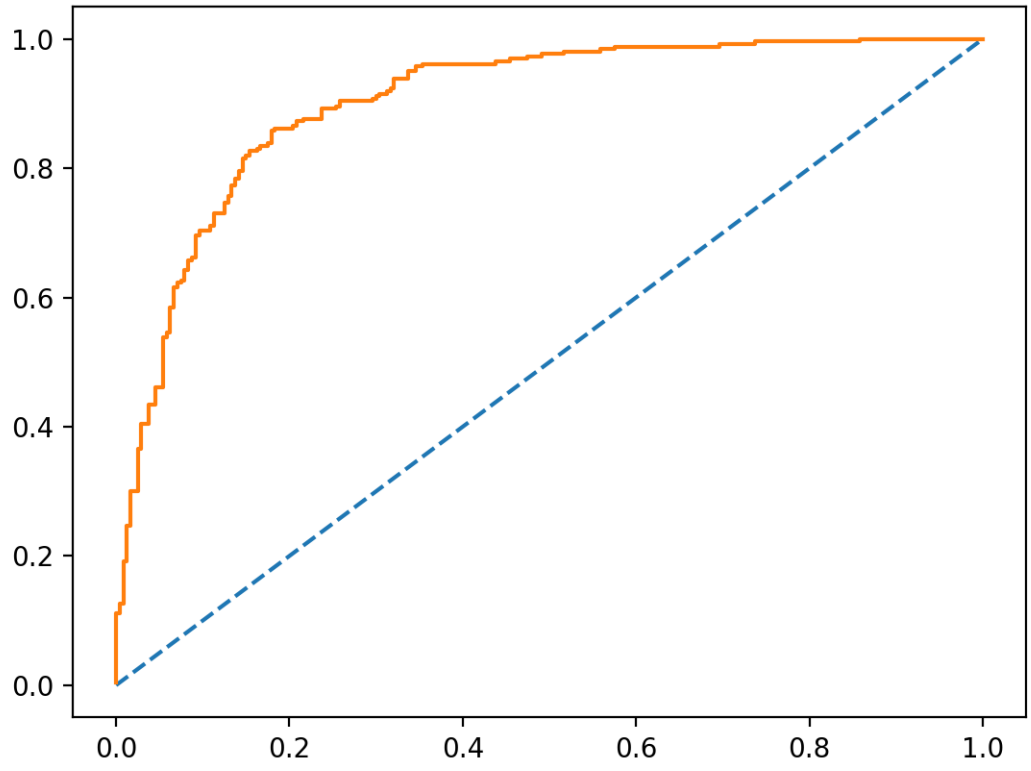


Figure 26.7: Example ROC Curve Plot for Binary Classification Predictions.

The integrated area under the ROC curve, called AUC or ROC AUC, provides a measure of the skill of the model across all evaluated thresholds. An AUC score of 0.5 suggests no skill, e.g. a curve along the diagonal, whereas an AUC of 1.0 suggests perfect skill, all points along the left y-axis and top x-axis toward the top left corner. An AUC of 0.0 suggests perfectly incorrect predictions. Predictions by models that have a larger area have better skill across the thresholds, although the specific shape of the curves between models will vary, potentially offering opportunity to optimize models by a pre-chosen threshold. Typically, the threshold is chosen by the operator after the model has been prepared.

The AUC can be calculated in Python using the `roc_auc_score()` function in scikit-learn. This function takes a list of true output values and predicted probabilities as arguments and returns the ROC AUC. An AUC score is a measure of the likelihood that the model that produced the predictions will rank a randomly chosen positive example above a randomly chosen negative example. Specifically, that the probability will be higher for a real event ($class = 1$) than a real non-event ($class = 0$). This is an instructive definition that offers two important intuitions:

- **Naive Prediction**. A naive prediction under ROC AUC is any constant probability. If the same probability is predicted for every example, there is no discrimination between positive and negative cases, therefore the model has no skill (AUC=0.5).

- **Insensitivity to Class Imbalance**. ROC AUC is a summary on the models ability to correctly discriminate a single example across different thresholds. As such, it is unconcerned with the base likelihood of each class.

Below, the example demonstrating the ROC curve is updated to calculate and display the AUC.

```
# roc auc
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
# calculate roc auc
auc = roc_auc_score(testy, probs)
print(auc)
```

Listing 26.8: Example of calculating the ROC AUC for predictions.

Running the example calculates and prints the ROC AUC for the logistic regression model evaluated on 500 new examples.

```
0.9028205128205128
```

Listing 26.9: Example output from calculating the ROC AUC for predictions.

An important consideration in choosing the ROC AUC is that it does not summarize the specific discriminative power of the model, rather the general discriminative power across all thresholds. It might be a better tool for model selection rather than in quantifying the practical skill of a model's predicted probabilities.

## 26.5 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 26.5.1 API

- `sklearn.metrics.log_loss` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html
- `sklearn.metrics.brier_score_loss` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.brier_score_loss.html

- `sklearn.metrics.roc_curve` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
- `sklearn.metrics.roc_auc_score` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html

### 26.5.2 Articles

- Scoring rule, Wikipedia.
  https://en.wikipedia.org/wiki/Scoring_rule
- Cross entropy, Wikipedia.
  https://en.wikipedia.org/wiki/Cross_entropy
- Brier score, Wikipedia.
  https://en.wikipedia.org/wiki/Brier_score
- Receiver operating characteristic, Wikipedia.
  https://en.wikipedia.org/wiki/Receiver_operating_characteristic

## 26.6 Summary

In this tutorial, you discovered three metrics that you can use to evaluate the predicted probabilities on your classification predictive modeling problem. Specifically, you learned:

- The log loss score that heavily penalizes predicted probabilities far away from their expected value.
- The Brier score that is gentler than log loss but still penalizes proportional to the distance from the expected value
- The area under ROC curve that summarizes the likelihood of the model predicting a higher probability for true positive cases than true negative cases.

### 26.6.1 Next

In the next tutorial, you will discover how to evaluate predicted probabilities using ROC and Precision-Recall curves.

# Chapter 27

# When to Use ROC Curves and Precision-Recall Curves

It can be more flexible to predict probabilities of an observation belonging to each class in a classification problem rather than predicting classes directly. This flexibility comes from the way that probabilities may be interpreted using different thresholds that allow the operator of the model to trade-off concerns in the errors made by the model, such as the number of false positives compared to the number of false negatives. This is required when using models where the cost of one error outweighs the cost of other types of errors. Two diagnostic tools that help in the interpretation of probabilistic forecast for binary (two-class) classification predictive modeling problems are ROC Curves and Precision-Recall curves. In this tutorial, you will discover ROC Curves, Precision-Recall Curves, and when to use each to interpret the prediction of probabilities for binary classification problems. After completing this tutorial, you will know:

- ROC Curves summarize the trade-off between the true positive rate and false positive rate for a predictive model using different probability thresholds.
- Precision-Recall curves summarize the trade-off between the true positive rate and the positive predictive value for a predictive model using different probability thresholds.
- ROC curves are appropriate when the observations are balanced between each class, whereas precision-recall curves are appropriate for imbalanced datasets.

Let's get started.

## 27.1 Tutorial Overview

This tutorial is divided into 6 parts; they are:

1. Predicting Probabilities
2. What Are ROC Curves?
3. ROC Curves and AUC in Python
4. What Are Precision-Recall Curves?

5. Precision-Recall Curves and AUC in Python
6. When to Use ROC vs. Precision-Recall Curves?

## 27.2 Predicting Probabilities

In a classification problem, we may decide to predict the class values directly. Alternately, it can be more flexible to predict the probabilities for each class instead. The reason for this is to provide the capability to choose and even calibrate the threshold for how to interpret the predicted probabilities. For example, a default might be to use a threshold of 0.5, meaning that a probability in [0.0, 0.49] is a negative outcome (0) and a probability in [0.5, 1.0] is a positive outcome (1). This threshold can be adjusted to tune the behavior of the model for a specific problem. An example would be to reduce more of one or another type of error. When making a prediction for a binary or two-class classification problem, there are two types of errors that we could make.

- **False Positive**. Predict an event when there was no event.
- **False Negative**. Predict no event when in fact there was an event.

By predicting probabilities and calibrating a threshold, a balance of these two concerns can be chosen by the operator of the model. For example, in a smog prediction system, we may be far more concerned with having low false negatives than low false positives. A false negative would mean not warning about a smog day when in fact it is a high smog day, leading to health issues in the public that are unable to take precautions. A false positive means the public would take precautionary measures when they didn't need to. A common way to compare models that predict probabilities for two-class problems is to use a ROC curve.

## 27.3 What Are ROC Curves?

A useful tool when predicting the probability of a binary outcome is the Receiver Operating Characteristic curve, or ROC curve. It is a plot of the false positive rate (x-axis) versus the true positive rate (y-axis) for a number of different candidate threshold values between 0.0 and 1.0. Put another way, it plots the false alarm rate versus the hit rate. The true positive rate is calculated as the number of true positives divided by the sum of the number of true positives and the number of false negatives. It describes how good the model is at predicting the positive class when the actual outcome is positive.

$$\text{True Positive Rate} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{27.1}$$

The true positive rate is also referred to as sensitivity.

$$\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{27.2}$$

The false positive rate is calculated as the number of false positives divided by the sum of the number of false positives and the number of true negatives. It is also called the false

alarm rate as it summarizes how often a positive class is predicted when the actual outcome is negative.

$$\text{False Positive Rate} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} \tag{27.3}$$

The false positive rate is also referred to as the inverted specificity where specificity is the total number of true negatives divided by the sum of the number of true negatives and false positives.

$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \tag{27.4}$$

Where:

$$\text{False Positive Rate} = 1 - \text{Specificity} \tag{27.5}$$

The ROC curve is a useful tool for a few reasons:

- The curves of different models can be compared directly in general or for different thresholds.
- The area under the curve (AUC) can be used as a summary of the model skill.

The shape of the curve contains a lot of information, including what we might care about most for a problem, the expected false positive rate, and the false negative rate. To make this clear:

- Smaller values on the x-axis of the plot indicate lower false positives and higher true negatives.
- Larger values on the y-axis of the plot indicate higher true positives and lower false negatives.

If you are confused, remember, when we predict a binary outcome, it is either a correct prediction (true positive) or not (false positive). There is a tension between these options, the same with true negative and false negative. A skillful model will assign a higher probability to a randomly chosen real positive occurrence than a negative occurrence on average. This is what we mean when we say that the model has skill. Generally, skillful models are represented by curves that bow up to the top left of the plot.

A model with no skill is represented at the point (0.5, 0.5). A model with no skill at each threshold is represented by a diagonal line from the bottom left of the plot to the top right and has an AUC of 0.5. A model with perfect skill is represented at a point (0,1). A model with perfect skill is represented by a line that travels from the bottom left of the plot to the top left and then across the top to the top right. An operator may plot the ROC curve for the final model and choose a threshold that gives a desirable balance between the false positives and false negatives.

## 27.4 ROC Curves and AUC in Python

We can plot a ROC curve for a model in Python using the `roc_curve()` scikit-learn function (introduced in Chapter 26). The function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the 1 class. The function returns the false positive and true positive rates for each threshold.

```
...
# calculate roc curve
fpr, tpr, thresholds = roc_curve(y, probs)
```

Listing 27.1: Example of calculating a ROC curve.

The AUC for the ROC can be calculated using the `roc_auc_score()` function. Like the `roc_curve()` function, the AUC function takes both the true outcomes (0,1) from the test set and the predicted probabilities for the 1 class. It returns the AUC score between 0.0 and 1.0 for no skill and perfect skill respectively.

```
...
# calculate ROC AUC
auc = roc_auc_score(y, probs)
print('AUC: %.3f' % auc)
```

Listing 27.2: Example of calculating a ROC AUC score.

A complete example of calculating the ROC curve and ROC AUC for a Logistic Regression model on a small test problem is listed below.

```
# roc curve and auc
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(testy))]
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# calculate scores
ns_auc = roc_auc_score(testy, ns_probs)
lr_auc = roc_auc_score(testy, lr_probs)
# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('Logistic: ROC AUC=%.3f' % (lr_auc))
# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(testy, ns_probs)
```

```
lr_fpr, lr_tpr, _ = roc_curve(testy, lr_probs)
# plot the roc curve for the model
pyplot.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
pyplot.plot(lr_fpr, lr_tpr, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('False Positive Rate')
pyplot.ylabel('True Positive Rate')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 27.3: Example of calculating and plotting a ROC curve for predicted probabilities.

Running the example prints the area under the ROC curve.

```
No Skill: ROC AUC=0.500
Logistic: ROC AUC=0.903
```

Listing 27.4: Example output from calculating the ROC AUC for predicted probabilities.

Running the example prints the ROC AUC for the logistic regression model and the no skill classifier that only predicts 0 for all examples.

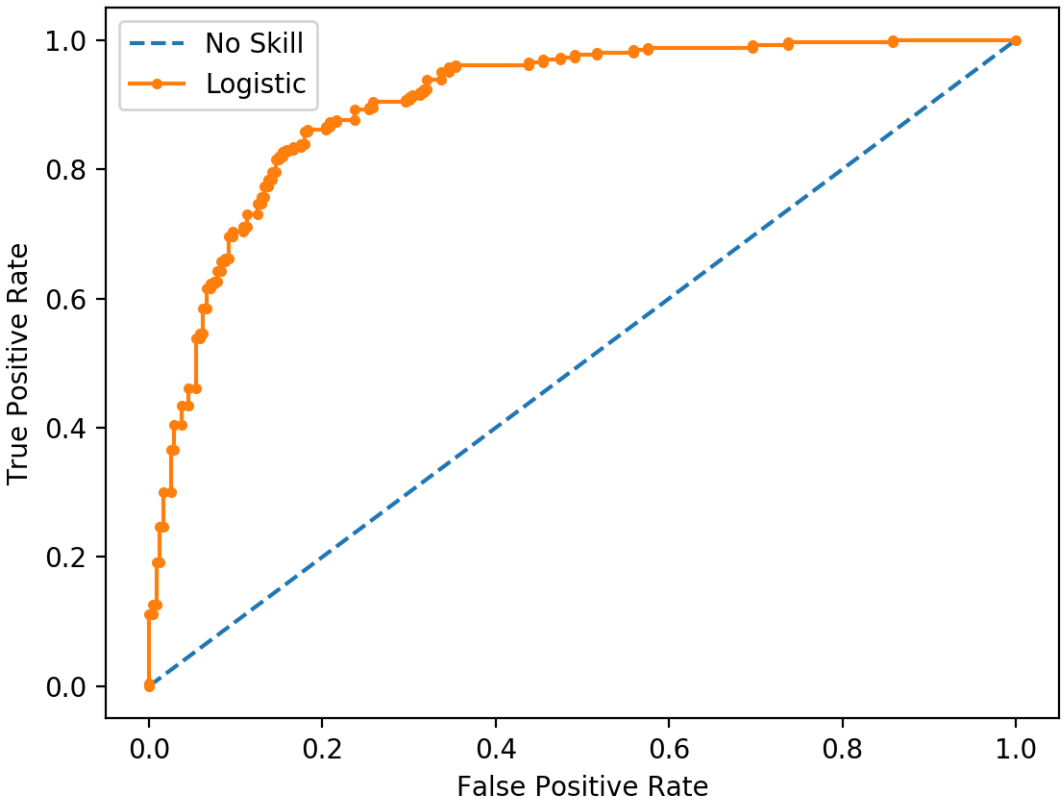


Figure 27.1: Line Plot of ROC Curve for Predicted Probabilities.

## 27.5 What Are Precision-Recall Curves?

There are many ways to evaluate the skill of a prediction model. An approach in the related field of information retrieval (finding documents based on queries) measures precision and recall. These measures are also useful in applied machine learning for evaluating binary classification models. Precision is a ratio of the number of true positives divided by the sum of the true positives and false positives. It describes how good a model is at predicting the positive class. Precision is referred to as the positive predictive value.

$$\text{Positive Predictive Power} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{27.6}$$

Or:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{27.7}$$

Recall is calculated as the ratio of the number of true positives divided by the sum of the true positives and the false negatives. Recall is the same as sensitivity, and the true-positive rate.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{27.8}$$

Or:

$$\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{27.9}$$

Where:

$$\text{Recall} \equiv \text{Sensitivity} \tag{27.10}$$

Reviewing both precision and recall is useful in cases where there is an imbalance in the observations between the two classes. Specifically, there are many examples of no event (class 0) and only a few examples of an event (class 1). The reason for this is that typically the large number of class 0 examples means we are less interested in the skill of the model at predicting class 0 correctly, e.g. high true negatives. Key to the calculation of precision and recall is that the calculations do not make use of the true negatives. It is only concerned with the correct prediction of the minority class, class 1.

A precision-recall curve is a plot of the precision (y-axis) and the recall (x-axis) for different thresholds, much like the ROC curve. A no-skill classifier is one that cannot discriminate between the classes and would predict a random class or a constant class in all cases. The no-skill line changes based on the distribution of the positive to negative classes. It is a horizontal line with the value of the ratio of positive cases in the dataset. For a balanced dataset, this is 0.5. A model with perfect skill is depicted as a point at (1,1). A skillful model is represented by a curve that bows towards (1,1) above the flat line of no skill. There are also composite scores that attempt to summarize the precision and recall; two examples include:

- **F-Measure or F1 score**: that calculates the harmonic mean of the precision and recall (harmonic mean because the precision and recall are ratios).

- **Area Under Curve**: like the ROC AUC, summarizes the integral or an approximation of the area under the precision-recall curve.

In terms of model selection, F1 summarizes model skill for a specific probability threshold (0.5), whereas the area under curve summarize the skill of a model across thresholds, like ROC AUC. This makes precision-recall and a plot of precision vs. recall and summary measures useful tools for binary classification problems that have an imbalance in the observations for each class.

## 27.6 Precision-Recall Curves in Python

Precision and recall can be calculated in scikit-learn via the `precision_score()` and `recall_score()` functions. The precision and recall can be calculated for thresholds using the `precision_recall_curve()` function that takes the true output values and the probabilities for the positive class as output and returns the precision, recall and threshold values.

```
...
# calculate precision-recall curve
precision, recall, thresholds = precision_recall_curve(testy, probs)
```

Listing 27.5: Example of calculating a precision-recall curve.

The F1 score can be calculated by calling the `f1_score()` function that takes the true class values and the predicted class values as arguments.

```
...
# calculate F1 score
f1 = f1_score(testy, yhat)
```

Listing 27.6: Example of calculating the F1 score.

The area under the precision-recall curve can be approximated by calling the `auc()` function and passing it the recall (x) and precision (y) values calculated for each threshold.

```
...
# calculate precision-recall AUC
auc = auc(recall, precision)
```

Listing 27.7: Example of calculating the area under the precision-recall curve.

When plotting precision and recall for each threshold as a curve, it is important that recall is provided as the x-axis and precision is provided as the y-axis. The complete example of calculating precision-recall curves for a Logistic Regression model is listed below.

```
# precision-recall curve and f1
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import f1_score
from sklearn.metrics import auc
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
```

```
# fit a model
model = LogisticRegression(solver='lbfgs')
model.fit(trainX, trainy)
# predict probabilities
lr_probs = model.predict_proba(testX)
# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]
# predict class values
yhat = model.predict(testX)
lr_precision, lr_recall, _ = precision_recall_curve(testy, lr_probs)
lr_f1, lr_auc = f1_score(testy, yhat), auc(lr_recall, lr_precision)
# summarize scores
print('Logistic: f1=%.3f auc=%.3f' % (lr_f1, lr_auc))
# plot the precision-recall curves
no_skill = len(testy[testy==1]) / len(testy)
pyplot.plot([0, 1], [no_skill, no_skill], linestyle='--', label='No Skill')
pyplot.plot(lr_recall, lr_precision, marker='.', label='Logistic')
# axis labels
pyplot.xlabel('Recall')
pyplot.ylabel('Precision')
# show the legend
pyplot.legend()
# show the plot
pyplot.show()
```

Listing 27.8: Example of calculating and plotting a precision-recall curve for predicted probabilities.

Running the example first prints the F1, area under curve (AUC) for the logistic regression model.

```
Logistic: f1=0.841 auc=0.898
```

Listing 27.9: Example output from calculating the precision-recall AUC for predicted probabilities.

The precision-recall curve plot is then created showing the precision/recall for each threshold for a logistic regression model (orange) compared to a no skill model (blue).

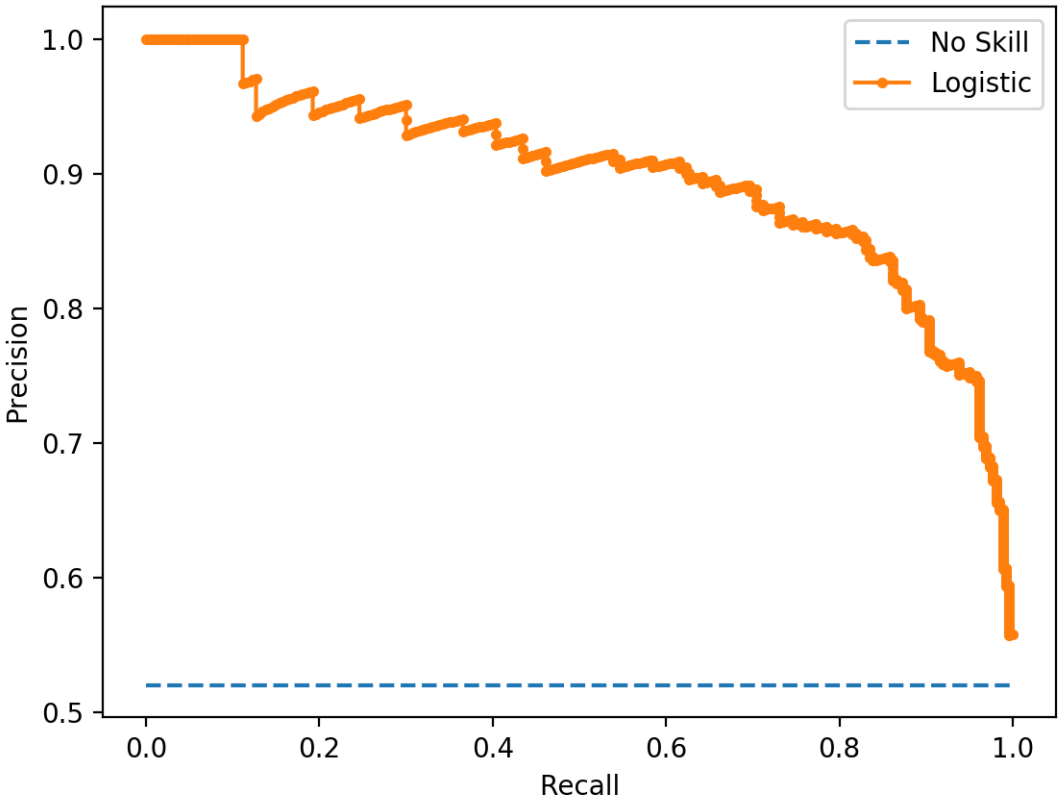


Figure 27.2: Line Plot of Precision-Recall Curve for Predicted Probabilities.

## 27.7 When to Use ROC vs. Precision-Recall Curves?

Generally, the use of ROC curves and precision-recall curves are as follows:

- ROC curves should be used when there are roughly equal numbers of observations for each class.
- Precision-Recall curves should be used when there is a moderate to large class imbalance.

The reason for this recommendation is that ROC curves present an optimistic picture of the model on datasets with a class imbalance.

> However, ROC curves can present an overly optimistic view of an algorithm's performance if there is a large skew in the class distribution. [...] Precision-Recall (PR) curves, often used in Information Retrieval , have been cited as an alternative to ROC curves for tasks with a large skew in the class distribution.
>
> — *The Relationship Between Precision-Recall and ROC Curves*, 2006.

Some go further and suggest that using a ROC curve with an imbalanced dataset might be deceptive and lead to incorrect interpretations of the model skill.

> ... the visual interpretability of ROC plots in the context of imbalanced datasets can be deceptive with respect to conclusions about the reliability of classification performance, owing to an intuitive but wrong interpretation of specificity. [Precision-recall curve] plots, on the other hand, can provide the viewer with an accurate prediction of future classification performance due to the fact that they evaluate the fraction of true positives among positive predictions

— *The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets*, 2015.

The main reason for this optimistic picture is because of the use of true negatives in the False Positive Rate in the ROC Curve and the careful avoidance of this rate in the Precision-Recall curve.

> If the proportion of positive to negative instances changes in a test set, the ROC curves will not change. Metrics such as accuracy, precision, lift and F scores use values from both columns of the confusion matrix. As a class distribution changes these measures will change as well, even if the fundamental classifier performance does not. ROC graphs are based upon TP rate and FP rate, in which each dimension is a strict columnar ratio, so do not depend on class distributions.

— *ROC Graphs: Notes and Practical Considerations for Data Mining Researchers*, 2003.

## 27.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 27.8.1 Papers

- *A Critical Investigation Of Recall And Precision As Measures Of Retrieval System Performance*, 1989.
  https://dl.acm.org/citation.cfm?id=65945
- *The Relationship Between Precision-Recall and ROC Curves*, 2006.
  https://dl.acm.org/citation.cfm?id=1143874
- *The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets*, 2015.
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4349800/
- *ROC Graphs: Notes and Practical Considerations for Data Mining Researchers*, 2003.
  http://www.blogspot.udec.ugto.saedsayad.com/docs/ROC101.pdf

### 27.8.2 API

- `sklearn.metrics.roc_curve` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
- `sklearn.metrics.roc_auc_score` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html
- `sklearn.metrics.precision_recall_curve` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html
- `sklearn.metrics.auc` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.auc.html
- `sklearn.metrics.average_precision_score` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html
- Precision-Recall, scikit-learn.
  http://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html
- Precision, recall and F-measures, scikit-learn.
  http://scikit-learn.org/stable/modules/model_evaluation.html

### 27.8.3 Articles

- Receiver operating characteristic on Wikipedia.
  https://en.wikipedia.org/wiki/Receiver_operating_characteristic
- Sensitivity and specificity on Wikipedia.
  https://en.wikipedia.org/wiki/Sensitivity_and_specificity
- Precision and recall on Wikipedia.
  https://en.wikipedia.org/wiki/Precision_and_recall
- Information retrieval on Wikipedia.
  https://en.wikipedia.org/wiki/Information_retrieval
- F1 score on Wikipedia.
  https://en.wikipedia.org/wiki/F1_score

## 27.9 Summary

In this tutorial, you discovered ROC Curves, Precision-Recall Curves, and when to use each to interpret the prediction of probabilities for binary classification problems. Specifically, you learned:

- ROC Curves summarize the trade-off between the true positive rate and false positive rate for a predictive model using different probability thresholds.
- Precision-Recall curves summarize the trade-off between the true positive rate and the positive predictive value for a predictive model using different probability thresholds.
- ROC curves are appropriate when the observations are balanced between each class, whereas precision-recall curves are appropriate for imbalanced datasets.

### 27.9.1 Next

In the next tutorial, you will discover how to calibrate probabilities predicted by a classification predictive model.

# Chapter 28

# How to Calibrate Predicted Probabilities

Instead of predicting class values directly for a classification problem, it can be convenient to predict the probability of an observation belonging to each possible class. Predicting probabilities allows some flexibility including deciding how to interpret the probabilities, presenting predictions with uncertainty, and providing more nuanced ways to evaluate the skill of the model.

Predicted probabilities that match the expected distribution of probabilities for each class are referred to as calibrated. The problem is, not all machine learning models are capable of predicting calibrated probabilities. There are methods to both diagnose how calibrated predicted probabilities are and to better calibrate the predicted probabilities with the observed distribution of each class. Often, this can lead to better quality predictions, depending on how the skill of the model is evaluated. In this tutorial, you will discover the importance of calibrating predicted probabilities and how to diagnose and improve the calibration of models used for probabilistic classification. After completing this tutorial, you will know:

- Nonlinear machine learning algorithms often predict uncalibrated class probabilities.
- Reliability diagrams can be used to diagnose the calibration of a model, and methods can be used to better calibrate predictions for a problem.
- How to develop reliability diagrams and calibrate classification models in Python with scikit-learn.

Let's get started.

## 28.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Predicting Probabilities
2. Calibration of Predictions
3. How to Calibrate Probabilities in Python
4. Worked Example of Calibrating SVM Probabilities

## 28.2 Predicting Probabilities

A classification predictive modeling problem requires predicting or forecasting a label for a given observation. An alternative to predicting the label directly, a model may predict the probability of an observation belonging to each possible class label. This provides some flexibility both in the way predictions are interpreted and presented (choice of threshold and prediction uncertainty) and in the way the model is evaluated. Although a model may be able to predict probabilities, the distribution and behavior of the probabilities may not match the expected distribution of observed probabilities in the training data.

This is especially common with complex nonlinear machine learning algorithms that do not directly make probabilistic predictions and instead use approximations. The distribution of the probabilities can be adjusted to better match the expected distribution observed in the data. This adjustment is referred to as calibration, as in the calibration of the model or the calibration of the distribution of class probabilities.

> ... we desire that the estimated class probabilities are reflective of the true underlying probability of the sample. That is, the predicted class probability (or probability-like value) needs to be well-calibrated. To be well-calibrated, the probabilities must effectively reflect the true likelihood of the event of interest.
>
> — Page 249, *Applied Predictive Modeling*, 2013.

## 28.3 Calibration of Predictions

There are two concerns in calibrating probabilities; they are diagnosing the calibration of predicted probabilities and the calibration process itself.

### 28.3.1 Reliability Diagrams (Calibration Curves)

A reliability diagram is a line plot of the relative frequency of what was observed (y-axis) versus the predicted probability frequency (x-axis).

> Reliability diagrams are common aids for illustrating the properties of probabilistic forecast systems. They consist of a plot of the observed relative frequency against the predicted probability, providing a quick visual intercomparison when tuning probabilistic forecast systems, as well as documenting the performance of the final product
>
> — *Increasing the Reliability of Reliability Diagrams*, 2007.

Specifically, the predicted probabilities are divided up into a fixed number of buckets along the x-axis. The number of events ($class = 1$) are then counted for each bin (e.g. the relative observed frequency). Finally, the counts are normalized. The results are then plotted as a line plot. These plots are commonly referred to as *reliability diagrams* in forecast literature, although may also be called *calibration plots* or curves as they summarize how well the forecast probabilities are calibrated. The better calibrated or more reliable a forecast, the closer the points will appear along the main diagonal from the bottom left to the top right of the plot. The position of the points or the curve relative to the diagonal can help to interpret the probabilities; for example:

- **Below the diagonal**: The model has over-forecast; the probabilities are too large.
- **Above the diagonal**: The model has under-forecast; the probabilities are too small.

Probabilities, by definition, are continuous, so we expect some separation from the line, often shown as an $S$-shaped curve showing pessimistic tendencies over-forecasting low probabilities and under-forecasting high probabilities.

> Reliability diagrams provide a diagnostic to check whether the forecast value Xi is reliable. Roughly speaking, a probability forecast is reliable if the event actually happens with an observed relative frequency consistent with the forecast value.
>
> — *Increasing the Reliability of Reliability Diagrams*, 2007.

The reliability diagram can help to understand the relative calibration of the forecasts from different predictive models.

### 28.3.2 Probability Calibration

The predictions made by a predictive model can be calibrated. Calibrated predictions may (or may not) result in an improved calibration on a reliability diagram. Some algorithms are fit in such a way that their predicted probabilities are already calibrated. Without going into details why, logistic regression is one such example. Other algorithms do not directly produce predictions of probabilities, and instead a prediction of probabilities must be approximated. Some examples include neural networks, support vector machines, and decision trees. The predicted probabilities from these methods will likely be uncalibrated and may benefit from being modified via calibration.

Calibration of prediction probabilities is a rescaling operation that is applied after the predictions have been made by a predictive model. There are two popular approaches to calibrating probabilities; they are the Platt Scaling and Isotonic Regression. Platt Scaling is simpler and is suitable for reliability diagrams with the $S$-shape. Isotonic Regression is more complex, requires a lot more data (otherwise it may overfit), but can support reliability diagrams with different shapes (is nonparametric).

> Platt Scaling is most effective when the distortion in the predicted probabilities is sigmoid-shaped. Isotonic Regression is a more powerful calibration method that can correct any monotonic distortion. Unfortunately, this extra power comes at a price. A learning curve analysis shows that Isotonic Regression is more prone to overfitting, and thus performs worse than Platt Scaling, when data is scarce.
>
> — *Predicting Good Probabilities With Supervised Learning*, 2005.

**Note, and this is really important**: better calibrated probabilities may or may not lead to better class-based or probability-based predictions. It really depends on the specific metric used to evaluate predictions. In fact, some empirical results suggest that the algorithms that can benefit the more from calibrating predicted probabilities include SVMs, bagged decision trees, and random forests.

> ... after calibration the best methods are boosted trees, random forests and SVMs.
>
> — *Predicting Good Probabilities With Supervised Learning*, 2005.

## 28.4 How to Calibrate Probabilities in Python

The scikit-learn machine learning library allows you to both diagnose the probability calibration of a classifier and calibrate a classifier that can predict probabilities.

### 28.4.1 Diagnose Calibration

You can diagnose the calibration of a classifier by creating a reliability diagram of the actual probabilities versus the predicted probabilities on a test set. In scikit-learn, this is called a calibration curve. This can be implemented by first calculating the `calibration_curve()` function. This function takes the true class values for a dataset and the predicted probabilities for the main class ($class = 1$). The function returns the true probabilities for each bin and the predicted probabilities for each bin. The number of bins can be specified via the `n_bins` argument and default to 5. For example, below is a code snippet showing the API usage:

```
...
# predict probabilities
probs = model.predic_proba(testX)[:,1]
# reliability diagram
fop, mpv = calibration_curve(testy, probs, n_bins=10)
# plot perfectly calibrated
pyplot.plot([0, 1], [0, 1], linestyle='--')
# plot model reliability
pyplot.plot(mpv, fop, marker='.')
pyplot.show()
```

Listing 28.1: Example of the API for plotting a calibration curve.

### 28.4.2 Calibrate Classifier

A classifier can be calibrated in scikit-learn using the `CalibratedClassifierCV` class. There are two ways to use this class: prefit and cross-validation. You can fit a model on a training dataset and calibrate this prefit model using a hold out validation dataset. For example, below is a code snippet showing the API usage:

```
...
# prepare data
trainX, trainy = ...
valX, valy = ...
testX, testy = ...
# fit base model on training dataset
model = ...
model.fit(trainX, trainy)
# calibrate model on validation data
calibrator = CalibratedClassifierCV(model, cv='prefit')
calibrator.fit(valX, valy)
# evaluate the model
yhat = calibrator.predict(testX)
```

Listing 28.2: Example of the API for calibrating probabilities.

Alternately, the `CalibratedClassifierCV` can fit multiple copies of the model using $k$-fold cross-validation and calibrate the probabilities predicted by these models using the hold out set.

Predictions are made using each of the trained models. For example, below is a code snippet showing the API usage:

```
...
# prepare data
trainX, trainy = ...
testX, testy = ...
# define base model
model = ...
# fit and calibrate model on training data
calibrator = CalibratedClassifierCV(model, cv=3)
calibrator.fit(trainX, trainy)
# evaluate the model
yhat = calibrator.predict(testX)
```

Listing 28.3: Example of the API for calibrating probabilities with cross-validation.

The `CalibratedClassifierCV` class supports two types of probability calibration; specifically, the parametric `'sigmoid'` method (Platt's method) and the nonparametric `'isotonic'` method which can be specified via the `method` argument.

## 28.5 Worked Example of Calibrating SVM Probabilities

We can make the discussion of calibration concrete with some worked examples. In these examples, we will fit a support vector machine (SVM) to a noisy binary classification problem and use the model to predict probabilities, then review the calibration using a reliability diagram and calibrate the classifier and review the result. SVM is a good candidate model to calibrate because it does not natively predict probabilities, meaning the probabilities are often uncalibrated.

**A note on SVM**: probabilities can be predicted by calling the `decision_function()` function on the fit model instead of the usual `predict_proba()` function. The probabilities are not normalized, but can be normalized when calling the `calibration_curve()` function by setting the `normalize` argument to `True`. The example below fits an SVM model on the test problem, predicted probabilities, and plots the calibration of the probabilities as a reliability diagram,

```
# SVM reliability diagram
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.calibration import calibration_curve
from matplotlib import pyplot
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[1,1], random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = SVC(gamma='scale')
model.fit(trainX, trainy)
# predict probabilities
probs = model.decision_function(testX)
# reliability diagram
fop, mpv = calibration_curve(testy, probs, n_bins=10, normalize=True)
```

```
# plot perfectly calibrated
pyplot.plot([0, 1], [0, 1], linestyle='--')
# plot model reliability
pyplot.plot(mpv, fop, marker='.')
pyplot.show()
```

Listing 28.4: Example of plotting a reliability diagram for predicted probabilities.

Running the example creates a reliability diagram showing the calibration of the SVMs predicted probabilities (solid line) compared to a perfectly calibrated model along the diagonal of the plot (dashed line.) We can see the expected $S$-shaped curve of a conservative forecast.

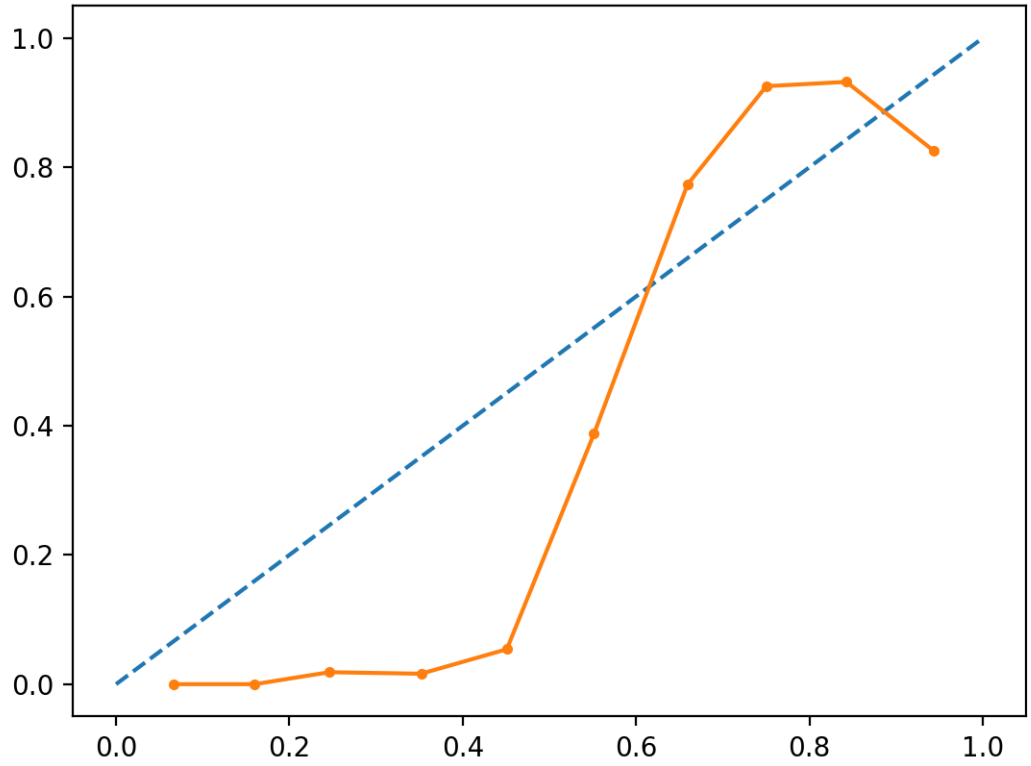


Figure 28.1: Plot of the Uncalibrated SVM Reliability Diagram.

We can update the example to fit the SVM via the `CalibratedClassifierCV` class using 5-fold cross-validation, using the holdout sets to calibrate the predicted probabilities. The complete example is listed below.

```
# SVM reliability diagram with calibration
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split
from sklearn.calibration import calibration_curve
from matplotlib import pyplot
```

```
# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[1,1], random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# fit a model
model = SVC(gamma='scale')
calibrated = CalibratedClassifierCV(model, method='sigmoid', cv=5)
calibrated.fit(trainX, trainy)
# predict probabilities
probs = calibrated.predict_proba(testX)[:, 1]
# reliability diagram
fop, mpv = calibration_curve(testy, probs, n_bins=10, normalize=True)
# plot perfectly calibrated
pyplot.plot([0, 1], [0, 1], linestyle='--')
# plot calibrated reliability
pyplot.plot(mpv, fop, marker='.')
pyplot.show()
```

Listing 28.5: Example of plotting a reliability diagram for the calibrated probabilities.

Running the example creates a reliability diagram for the calibrated probabilities. The shape of the calibrated probabilities is different, hugging the diagonal line much better, although still under-forecasting in the upper quadrant. Visually, the plot suggests a better calibrated model.

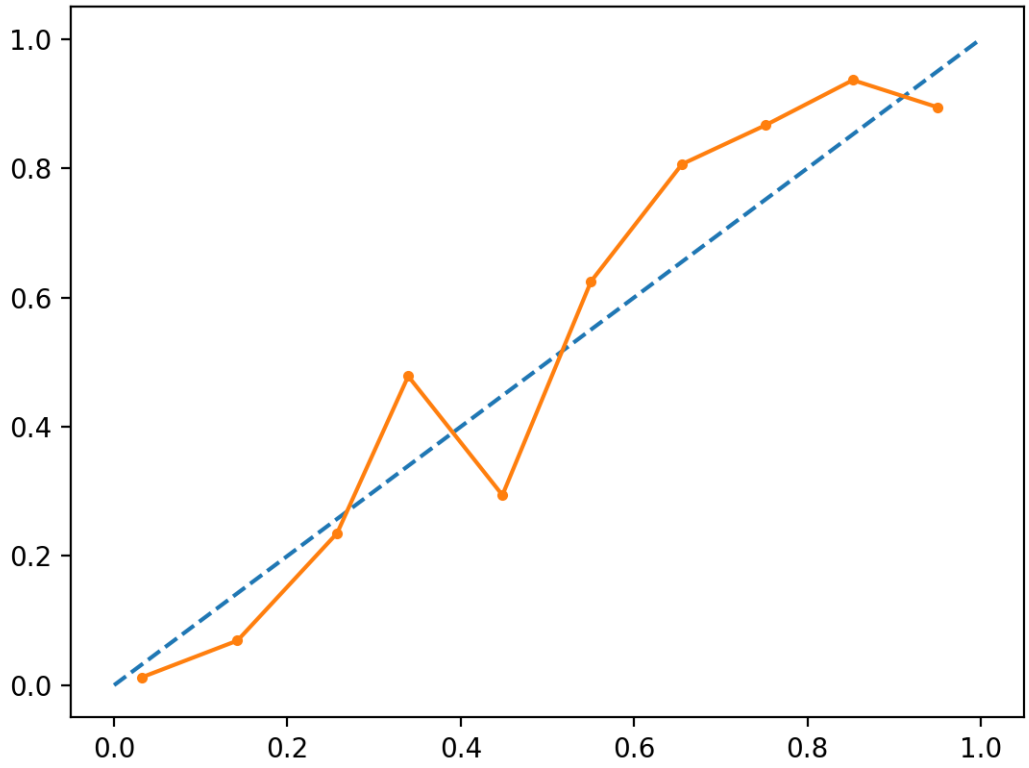


Figure 28.2: Plot of the Calibrated SVM Reliability Diagram.

We can make the contrast between the two models more obvious by including both reliability diagrams on the same plot. The complete example is listed below.

```
# SVM reliability diagrams with uncalibrated and calibrated probabilities
from sklearn.datasets import make_classification
from sklearn.svm import SVC
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import train_test_split
from sklearn.calibration import calibration_curve
from matplotlib import pyplot

# predict uncalibrated probabilities
def uncalibrated(trainX, testX, trainy):
  # fit a model
  model = SVC(gamma='scale')
  model.fit(trainX, trainy)
  # predict probabilities
  return model.decision_function(testX)

# predict calibrated probabilities
def calibrated(trainX, testX, trainy):
  # define model
  model = SVC(gamma='scale')
  # define and fit calibration model
  calibrated = CalibratedClassifierCV(model, method='sigmoid', cv=5)
  calibrated.fit(trainX, trainy)
  # predict probabilities
  return calibrated.predict_proba(testX)[:, 1]

# generate 2 class dataset
X, y = make_classification(n_samples=1000, n_classes=2, weights=[1,1], random_state=1)
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2)
# uncalibrated predictions
yhat_uncalibrated = uncalibrated(trainX, testX, trainy)
# calibrated predictions
yhat_calibrated = calibrated(trainX, testX, trainy)
# reliability diagrams
fop_uncalibrated, mpv_uncalibrated = calibration_curve(testy, yhat_uncalibrated, n_bins=10,
    normalize=True)
fop_calibrated, mpv_calibrated = calibration_curve(testy, yhat_calibrated, n_bins=10)
# plot perfectly calibrated
pyplot.plot([0, 1], [0, 1], linestyle='--', color='black')
# plot model reliabilities
pyplot.plot(mpv_uncalibrated, fop_uncalibrated, marker='.')
pyplot.plot(mpv_calibrated, fop_calibrated, marker='.')
pyplot.show()
```

Listing 28.6: Example of plotting a reliability diagram for the uncalibrated and calibrated probabilities.

Running the example creates a single reliability diagram showing both the calibrated (orange) and uncalibrated (blue) probabilities. It is not really an apples-to-apples comparison as the predictions made by the calibrated model are in fact a combination of five submodels. Nevertheless, we do see a marked difference in the reliability of the calibrated probabilities (very likely caused by the calibration process).

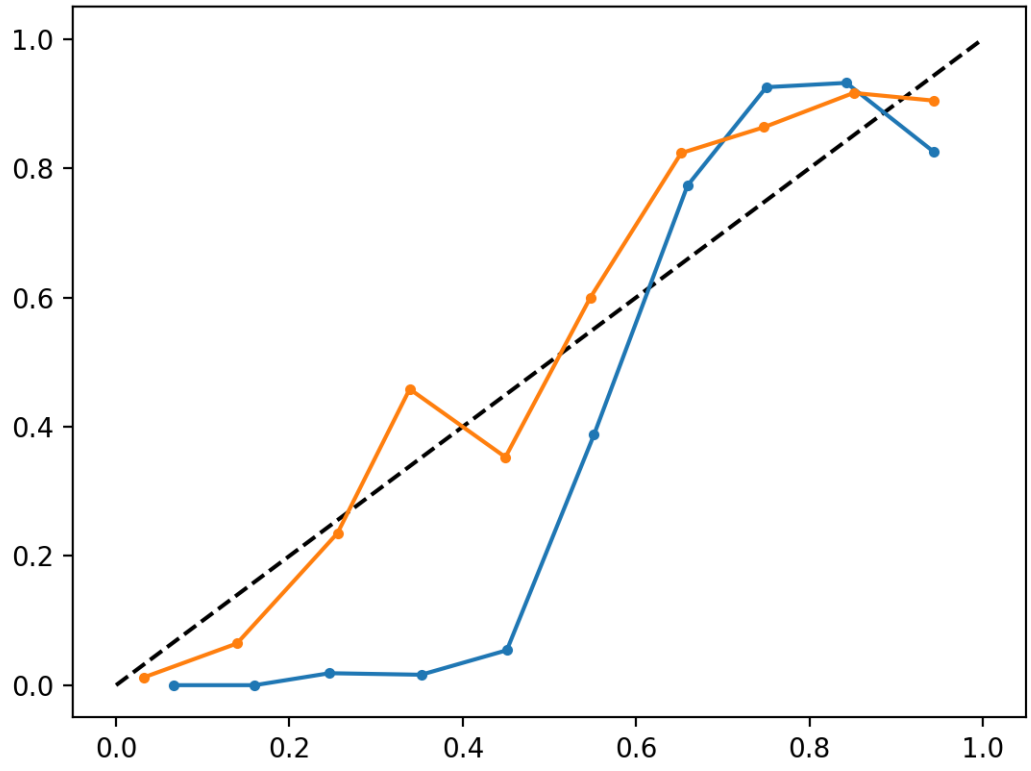


Figure 28.3: Plot of the Calibrated and Uncalibrated SVM Reliability Diagram.

## 28.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

### 28.6.1 Books and Papers

- *Applied Predictive Modeling*, 2013.
  https://amzn.to/2kXE35G
- *Predicting Good Probabilities With Supervised Learning*, 2005.
  https://www.cs.cornell.edu/~alexn/papers/calibration.icml05.crc.rev3.pdf
- *Obtaining Calibrated Probability Estimates From Decision Trees And Naive Bayesian Classifiers*, 2001.
  http://cseweb.ucsd.edu/~elkan/calibrated.pdf
- *Increasing the Reliability of Reliability Diagrams*, 2007.
  https://journals.ametsoc.org/doi/abs/10.1175/WAF993.1

### 28.6.2 API

- `sklearn.calibration.CalibratedClassifierCV` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.calibration.CalibratedClassifier
  html
- `sklearn.calibration.calibration_curve` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.calibration.calibration_
  curve.html#sklearn.calibration.calibration_curve
- Probability calibration, scikit-learn User Guide.
  http://scikit-learn.org/stable/modules/calibration.html
- Probability Calibration curves, scikit-learn.
  http://scikit-learn.org/stable/auto_examples/calibration/plot_calibration_
  curve.html
- Comparison of Calibration of Classifiers, scikit-learn.
  http://scikit-learn.org/stable/auto_examples/calibration/plot_compare_calibration.
  html

### 28.6.3 Articles

- Calibration (statistics) on Wikipedia.
  https://en.wikipedia.org/wiki/Calibration_(statistics)
- Probabilistic classification on Wikipedia.
  https://en.wikipedia.org/wiki/Probabilistic_classification

## 28.7 Summary

In this tutorial, you discovered the importance of calibrating predicted probabilities and how to diagnose and improve the calibration of models used for probabilistic classification. Specifically, you learned:

- Nonlinear machine learning algorithms often predict uncalibrated class probabilities.
- Reliability diagrams can be used to diagnose the calibration of a model, and methods can be used to better calibrate predictions for a problem.
- How to develop reliability diagrams and calibrate classification models in Python with scikit-learn.

### 28.7.1 Next

This was the final tutorial in this Part. In the next Part, you will discover additional resources.

# Part IX

# Appendix

# Appendix A

# Getting Help

This is just the beginning of your journey with probability. As you start to work on projects and expand your existing knowledge of the techniques, you may need help. This appendix points out some of the best sources of help.

## A.1 Probability on Wikipedia

Wikipedia is a great place to start. All of the important topics are covered, the descriptions are concise, and the equations are consistent and readable. What is missing is the more human level descriptions such as analogies and intuitions. Nevertheless, when you have questions about probability, I recommend stopping by Wikipedia first. Some good high-level pages to start on include:

- Probability, Wikipedia.
  https://en.wikipedia.org/wiki/Probability
- Probability theory, Wikipedia.
  https://en.wikipedia.org/wiki/Probability_theory
- List of probability topics, Wikipedia.
  https://en.wikipedia.org/wiki/List_of_probability_topics
- Catalog of articles in probability theory, Wikipedia.
  https://en.wikipedia.org/wiki/Catalog_of_articles_in_probability_theory
- Notation in probability and statistics, Wikipedia.
  https://en.wikipedia.org/wiki/Notation_in_probability_and_statistics

## A.2 Probability Textbooks

I strongly recommend getting a good textbook on the topic of probability and using it as a reference. The benefit of a good textbook is that the explanations of the various operations you require will be consistent (or should be). The downside of textbooks is that they can be very expensive. A good textbook is often easy to spot because it will be the basis for a range of undergraduate or postgraduate courses at top universities.

Some introductory textbooks on probability I recommend include:

- *Probability Theory: The Logic of Science*, 2003.
  https://amzn.to/2lnW2pp
- *Introduction to Probability*, 2nd Edition, 2019.
  https://amzn.to/2xPvobK
- *Introduction to Probability*, 2nd Edition, 2008.
  https://amzn.to/2llA3PR

I'd also recommend reading popular science books on probability and statistics. They provide good context and intuitions for the methods. Some recommendations include:

- *Naked Statistics: Stripping the Dread from the Data*, 2014.
  http://amzn.to/2t7yXdV
- *The Drunkard's Walk: How Randomness Rules Our Lives*, 2009.
  http://amzn.to/2CT4rUD
- *The Signal and the Noise: Why So Many Predictions Fail - but Some Don't*, 2015.
  http://amzn.to/2FQdYyX

## A.3 Probability and Machine Learning

There are a number of textbooks that introduce machine learning from a probabilistic perspective. These can be helpful, but do assume you have some background in probability and linear algebra. A few probabilistic perspectives on machine learning I would recommend include:

- *Machine Learning: A Probabilistic Perspective*, 2012.
  https://amzn.to/2xKSTCP
- *Pattern Recognition and Machine Learning*, 2006.
  https://amzn.to/2JwHE7I
- *Machine Learning*, 1997.
  https://amzn.to/2jWd51p

## A.4 Ask Questions About Probability

There are a lot of places that you can ask questions about probability online given the current abundance of question-and-answer platforms. Below is a list of the top places I recommend posting a question. Remember to search for your question before posting in case it has been asked and answered before.

- Probability tag on the Mathematics Stack Exchange.
  https://math.stackexchange.com/?tags=probability
- Cross Validated.
  https://stats.stackexchange.com

- Probability tag on Stack Overflow.
  https://stackoverflow.com/questions/tagged/probability
- Probability on Quora.
  https://www.quora.com/topic/Probability-statistics-1
- Probability Theory Subreddit.
  https://www.reddit.com/r/probabilitytheory/

## A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like "*my model does not work*" or "*how does x work*".
- Search for answers before asking questions.
- Provide complete code and error messages.
- Boil your code down to the smallest possible working example that demonstrates the issue.

## A.6 Contact the Author

You are not alone. If you ever have any questions about probability, or the tutorials in this book, please contact me directly. I will do my best to help.

**Jason Brownlee**
Jason@MachineLearningMastery.com

# Appendix B

# How to Setup Python on Your Workstation

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

## B.1 Tutorial Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda

Note: The specific versions may differ as the software and libraries are updated frequently.

## B.2 Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  https://www.continuum.io/
- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
  https://www.continuum.io/downloads

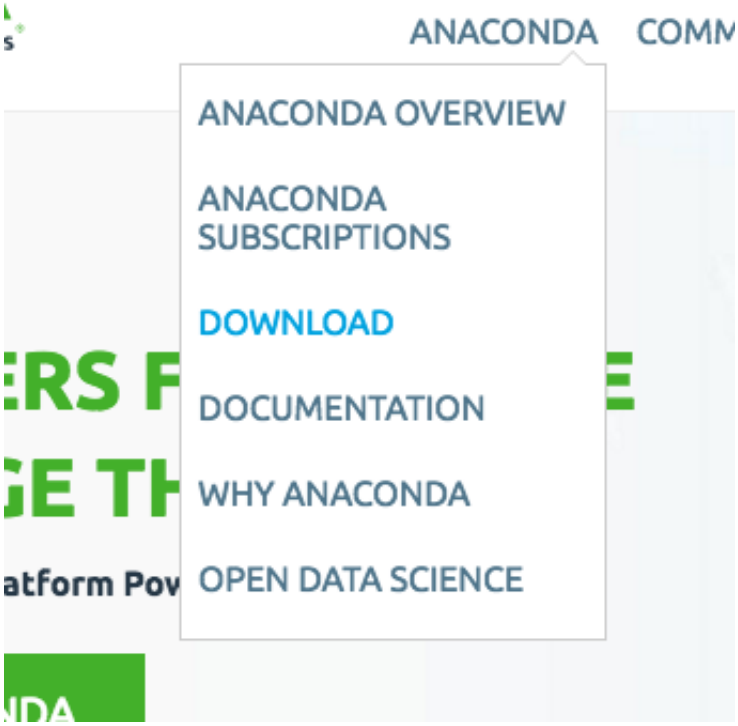


Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):
  - Choose Python 3.6
  - Choose the Graphical Installer

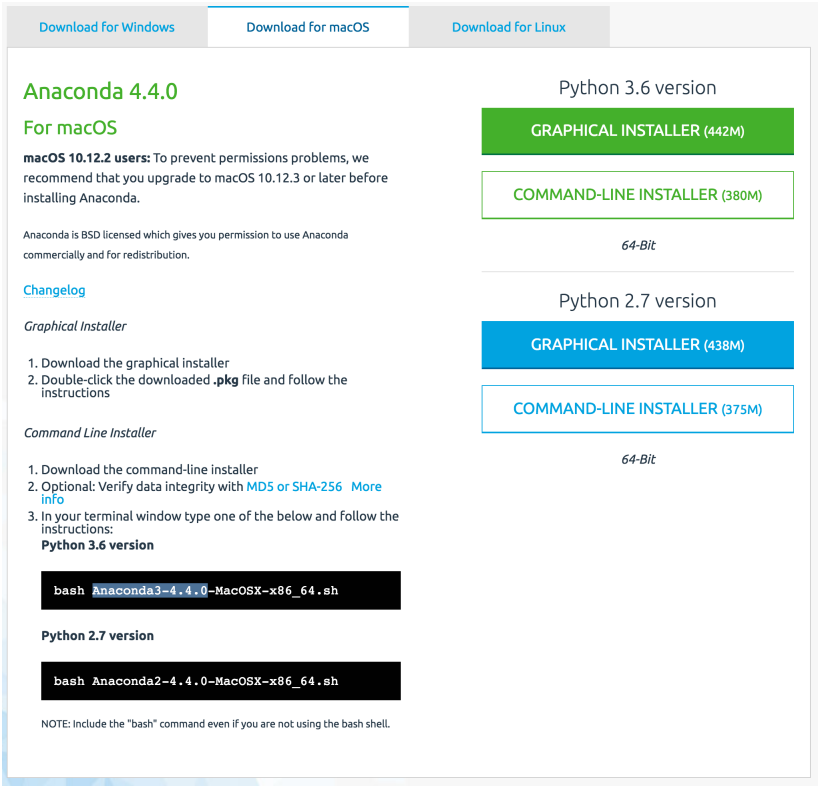


Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

## B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
- 2. Follow the installation wizard.

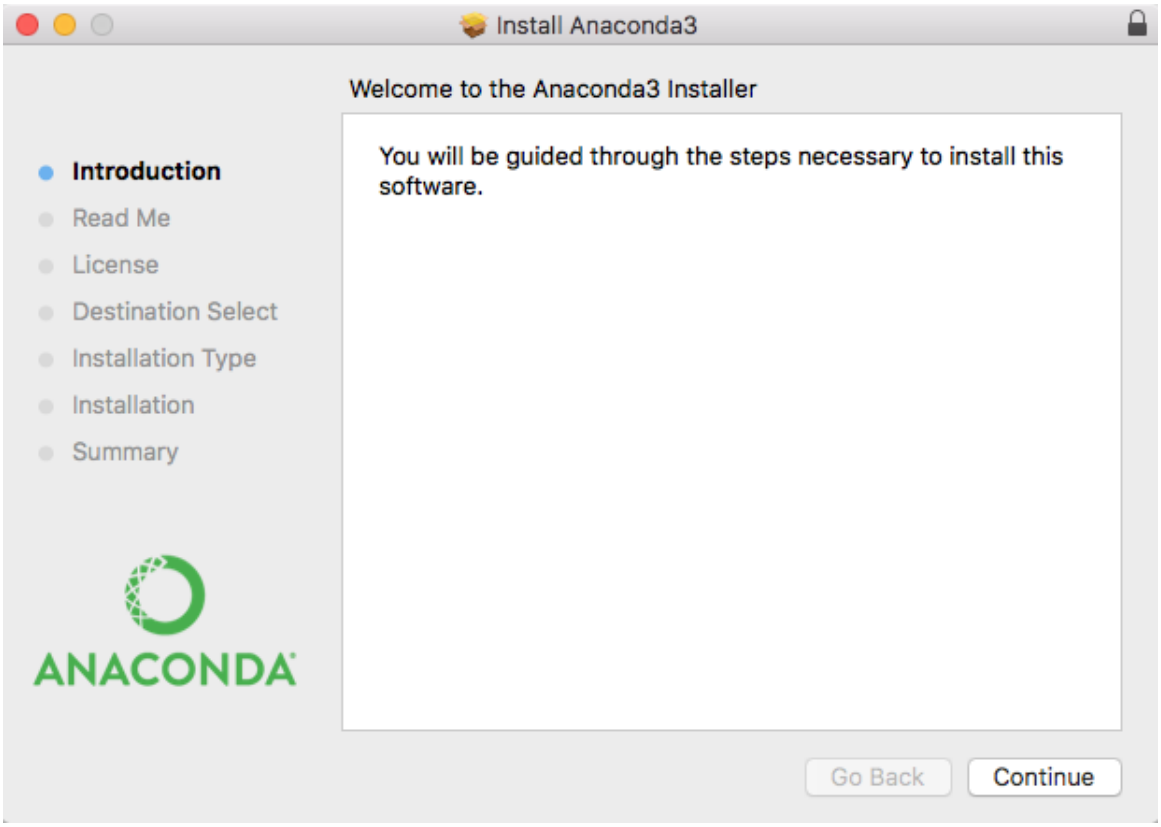


Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.

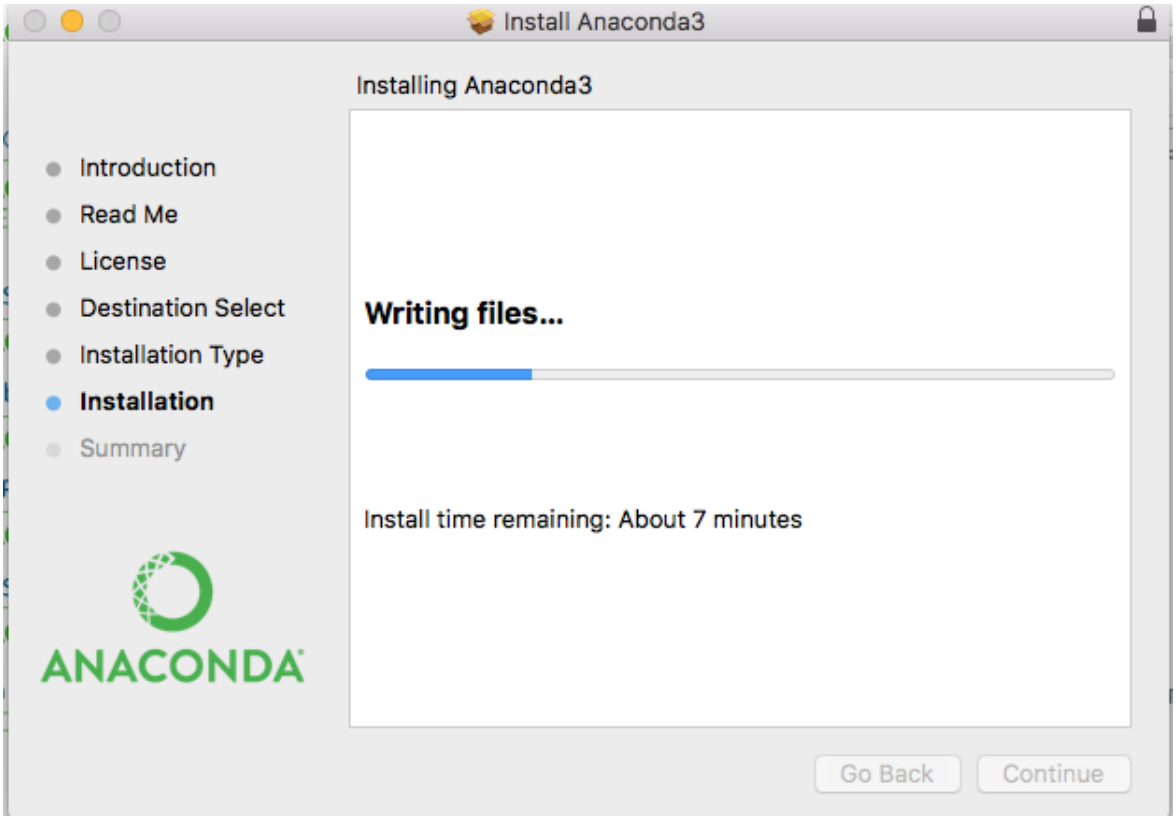


Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

## B.4 Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.

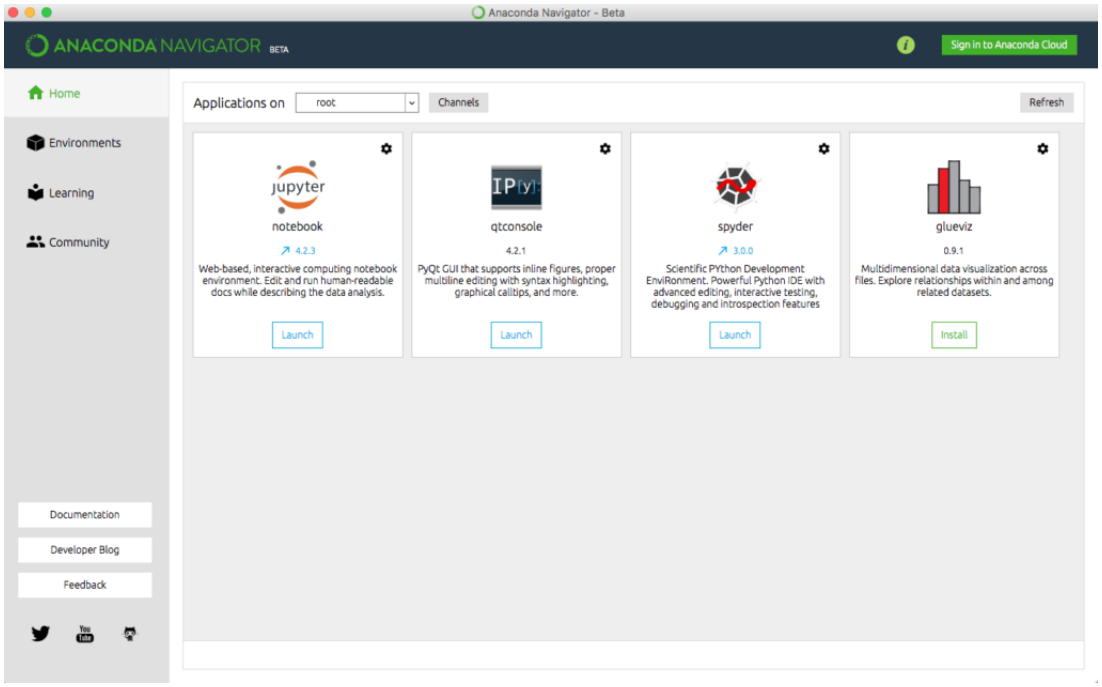


Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).
- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.3.1
numpy: 1.17.1
matplotlib: 3.1.1
pandas: 0.25.1
statsmodels: 0.10.1
sklearn: 0.21.3
```

Listing B.9: Sample output of versions script.

## B.5 Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/
- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html
- The conda command line tool.
  http://conda.pydata.org/docs/index.html

## B.6 Summary

Congratulations, you now have a working Python development environment for machine learning. You can now learn and practice machine learning on your workstation.

# Appendix C

# Basic Math Notation

You cannot avoid mathematical notation when reading the descriptions of machine learning methods. Often, all it takes is one term or one fragment of notation in an equation to completely derail your understanding of the entire procedure. This can be extremely frustrating, especially for machine learning beginners coming from the world of development. You can make great progress if you know a few basic areas of mathematical notation and some tricks for working through the description of machine learning methods in papers and books. In this tutorial, you will discover the basics of mathematical notation that you may come across when reading descriptions of techniques in machine learning. After completing this tutorial, you will know:

- Notation for arithmetic including variations of multiplication, exponents, roots and logarithms.
- Notation for sequences and sets including indexing, summation and set membership.
- 5 Techniques you can use to get help if you are struggling with mathematical notation.

Let's get started.

## C.1 Tutorial Overview

This tutorial is divided into 7 parts; they are:

1. The Frustration with Math Notation
2. Arithmetic Notation
3. Greek Alphabet
4. Sequence Notation
5. Set Notation
6. Other Notation
7. Tips for Getting More Help

## C.2 The Frustration with Math Notation

You will encounter mathematical notation when reading about machine learning algorithms. For example, notation may be used to:

- Describe an algorithm.
- Describe data preparation.
- Describe results.
- Describe a test harness.
- Describe implications.

These descriptions may be in research papers, textbooks, blog posts and elsewhere. Often the terms are well defined, but there are also mathematical notation norms that you may not be familiar with. All it takes is one term or one equation that you do not understand and your understanding of the entire method will be lost. I've suffered this problem myself many times and it is incredibly frustrating! In this tutorial we will review some basic mathematical notation that will help you when reading descriptions of machine learning methods.

## C.3 Arithmetic Notation

In this section we will go over some less obvious notations for basic arithmetic as well as a few concepts you may have forgotten since school.

### C.3.1 Simple Arithmetic

The notation for basic arithmetic is as you would write it. For example:

- Addition: $1 + 1 = 2$
- Subtraction: $2 - 1 = 1$
- Multiplication: $2 \times 2 = 4$
- Division: $\frac{2}{2} = 1$

Most mathematical operations have a sister operation that performs the inverse operation, for example subtraction is the inverse of addition and division is the inverse of multiplication.

### C.3.2 Algebra

We often want to describe operations abstractly to separate them from specific data or specific implementations. For this reason we see heavy use of algebra, that is uppercase and/or lowercase letters or words to represents terms or concepts in mathematical notation. It is also common to use letters from the Greek alphabet. Each sub-field of math may have reserved letters, that is terms or letters that always mean the same thing. Nevertheless, algebraic terms should be defined as part of the description and if they are not, it may just be a poor description, not your fault.

### C.3.3 Multiplication Notation

Multiplication is a common notation and has a few short hands. Often a little "x" ($\times$) or an asterisk "*" is used to represent multiplication:

$$c = a \times b \tag{C.1}$$

Or

$$c = a * b \tag{C.2}$$

You may see a dot notation used, for example:

$$c = a \cdot b \tag{C.3}$$

Alternately, you may see no operation and no white space separation between previously defined terms, for example:

$$c = ab \tag{C.4}$$

Which again is the same thing.

### C.3.4 Exponents and Square Roots

An exponent is a number raised to a power. The notation is written as the original number or the base with a second number or the exponent shown as a superscript, for example:

$$2^3 \tag{C.5}$$

Which would be calculated as 2 multiplied by itself 3 times or cubing:

$$2 \times 2 \times 2 = 8 \tag{C.6}$$

A number raised to the power 2 to is said to be it's square.

$$2^2 = 2 \times 2 = 4 \tag{C.7}$$

The square of a number can be inverted by calculating the square root. This is shown using the notation of a number and with a tick above $\sqrt{x}$.

$$\sqrt{4} = 2 \tag{C.8}$$

Here, we know the result and the exponent and we wish to find the base. In fact, the root operation can be used to inverse any exponent, it just so happens that the default square root assumes an exponent of 2, represented by a subscript 2 in front of the square root tick. For example, we can invert the cubing of a number by taking the cube root:

$$2^3 = 8 \tag{C.9}$$

$$\sqrt[3]{8} = 2 \tag{C.10}$$

### C.3.5 Logarithms and $e$

When we raise 10 to an integer exponent we often call this an order of magnitude.

$$10^2 = 10 \times 10 \tag{C.11}$$

Another way to reverse this operation is by calculating the logarithm of the result 100 assuming a base of 10, in notation this is written as `log10()`.

$$log10(100) = 2 \tag{C.12}$$

Here, we know the result and the base and wish to find the exponent. This allows us to move up and down orders of magnitude very easily. Taking the logarithm assuming the base of 2 is also commonly used, given the use of binary arithmetic used in computers. For example:

$$2^6 = 64 \tag{C.13}$$

$$log2(64) = 6 \tag{C.14}$$

Another popular logarithm is to assume the natural base called $e$. The $e$ is reserved and is a special number or a constant called Euler's number (pronounced *oy-ler*) that refers to a value with practically infinite precision.

$$e = 2.71828\cdots \tag{C.15}$$

Raising $e$ to a power is called a natural exponential function:

$$e^2 = 7.38905\cdots \tag{C.16}$$

It can be inverted using the natural logarithm which is denoted as `ln()`:

$$ln(7.38905\cdots) = 2 \tag{C.17}$$

Without going into detail, the natural exponent and natural logarithm prove useful throughout mathematics to abstractly describe the continuous growth of some systems, e.g. systems that grow exponentially such as compound interest.

## C.4 Greek Alphabet

Greek letters are used throughout mathematical notation for variables, constants, functions and more. For example in statistics we talk about the mean using the lowercase Greek letter mu ($\mu$), and the standard deviation as the lowercase Greek letter sigma ($\sigma$). In linear regression we talk about the coefficients as the lowercase letter beta ($\beta$). And so on. It is useful to know all of the uppercase and lowercase Greek letters and how to pronounce them. When I was a grad student, I printed the Greek alphabet and stuck it on my computer monitor so that I could memorize it. A useful trick! Below is the full Greek alphabet.

| Greek letters | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | TeX | HTML | Name | TeX | HTML | Name | TeX | HTML | Name | TeX | HTML | Name | TeX | HTML |
| Alpha | $A\,\alpha$ | Α α | Digamma | $F\,\digamma$ | Ϝ ϝ | Kappa | $K\,\kappa\,\varkappa$ | Κ κ ϰ | Omicron | $O\,o$ | Ο ο | Upsilon | $\Upsilon\,\upsilon$ | Υ υ |
| Beta | $B\,\beta$ | Β β | Zeta | $Z\,\zeta$ | Ζ ζ | Lambda | $\Lambda\,\lambda$ | Λ λ | Pi | $\Pi\,\pi\,\varpi$ | Π π ϖ | Phi | $\Phi\,\phi\,\varphi$ | Φ ϕ φ |
| Gamma | $\Gamma\,\gamma$ | Γ γ | Eta | $H\,\eta$ | Η η | Mu | $M\,\mu$ | Μ μ | Rho | $P\,\rho\,\varrho$ | Ρ ρ ϱ | Chi | $X\,\chi$ | Χ χ |
| Delta | $\Delta\,\delta$ | Δ δ | Theta | $\Theta\,\theta\,\vartheta$ | Θ θ ϑ | Nu | $N\,\nu$ | Ν ν | Sigma | $\Sigma\,\sigma\,\varsigma$ | Σ σ ς | Psi | $\Psi\,\psi$ | Ψ ψ |
| Epsilon | $E\,\epsilon\,\varepsilon$ | Ε ε ε | Iota | $I\,\iota$ | Ι ι | Xi | $\Xi\,\xi$ | Ξ ξ | Tau | $T\,\tau$ | Τ τ | Omega | $\Omega\,\omega$ | Ω ω |

Figure C.1: Greek Alphabet, Taken from Wikipedia.

The Wikipedia page titled *Greek letters used in mathematics, science, and engineering*[1] is also a useful guide as it lists common uses for each Greek letter in different sub-fields of math and science.

## C.5 Sequence Notation

Machine learning notation often describes an operation on a sequence. A sequence may be an array of data or a list of terms.

### C.5.1 Indexing

A key to reading notation for sequences is the notation of indexing elements in the sequence. Often the notation will specify the beginning and end of the sequence, such as 1 to $n$, where $n$ will be the extent or length of the sequence. Items in the sequence are index by a variable such as $i$, $j$, $k$ as a subscript. This is just like array notation. For example $a_i$ is the $i^{th}$ element of the sequence $a$. If the sequence is two dimensional, two indices may be used, for example: $b_{i,j}$ is the $(i,j)^{th}$ element of the sequence $b$.

### C.5.2 Sequence Operations

Mathematical operations can be performed over a sequence. Two operations are performed on sequences so often that they have their own shorthand, the sum and the multiplication.

#### Sequence Summation

The sum over a sequence is denoted as the uppercase Greek letter sigma ($\Sigma$). It is specified with the variable and start of the sequence summation below the sigma (e.g. $i = 1$) and the index of the end of the summation above the sigma (e.g. $n$).

$$\sum_{i=1}^{n} a_i \tag{C.18}$$

This is the sum of the sequence a starting at element 1 to element $n$.

[1] https://en.wikipedia.org/wiki/Greek_letters_used_in_mathematics,_science,_and_engineering

### Sequence Multiplication

The multiplication over a sequence is denoted as the uppercase Greek letter pi (Π). It is specified in the same way as the sequence summation with the beginning and end of the operation below and above the letter respectively.

$$\prod_{i=1}^{n} a_i \tag{C.19}$$

This is the product of the sequence a starting at element 1 to element $n$.

## C.6 Set Notation

A set is a group of unique items. We may see set notation used when defining terms in machine learning.

### C.6.1 Set of Numbers

A common set you may see is a set of numbers, such as a term defined as being within the set of integers or the set of real numbers. Some common sets of numbers you may see include:

- Set of all natural numbers: $\mathbb{N}$
- Set of all integers: $\mathbb{Z}$
- Set of all real numbers: $\mathbb{R}$

There are other sets, see *Special sets on Wikipedia*[2]. We often talk about real-values or real numbers when defining terms rather than floating point values, which are really discrete creations for operations in computers.

### C.6.2 Set Membership

It is common to see set membership in definitions of terms. Set membership is denoted as a symbol that looks like an uppercase "E" ($\in$).

$$a \in \mathbb{R} \tag{C.20}$$

Which means $a$ is defined as being a member of the set $\mathbb{R}$ or the set of real numbers. There is also a host of set operations, two common set operations include:

- Union, or aggregation: $A \cup B$
- Intersection, or overlap: $A \cap B$

Learn more about sets on Wikipedia[3].

[2] https://en.wikipedia.org/wiki/Set_(mathematics)#Special_sets
[3] https://en.wikipedia.org/wiki/Set_(mathematics)

## C.7 Other Notation

There is other notation that you may come across. I try to lay some of it out in this section. It is common to define a method in the abstract and then define it again as a specific implementation with separate notation. For example, if we are estimating a variable $x$ we may represent it using a notation that modifies the $x$, for example:

- x-bar ($\bar{x}$)
- x-prime ($\grave{x}$)
- x-hat ($\hat{x}$)
- x-tilde ($\tilde{x}$)

The same notation may have different meaning in a different context, such as use on different objects or sub-fields of mathematics. For example, a common point of confusion is $|x|$, which, depending on context can mean:

- $|x|$: The absolute or positive value of $x$.
- $|x|$: The length of the vector $x$.
- $|x|$: The cardinality of the set $x$.

This tutorial only covered the basics of mathematical notation. There are some subfields of mathematics that are more relevant to machine learning and should be reviewed in more detail. They are:

- Linear Algebra.
- Statistics.
- Probability.
- Calculus.

And perhaps a little bit of multivariate analysis and information theory.

## C.8 Tips for Getting More Help

This section lists some tips that you can use when you are struggling with mathematical notation in machine learning.

### C.8.1 Think About the Author

People wrote the paper or book you are reading. People that can make mistakes, make omissions, and even make things confusing because they don't fully understand what they are writing. Relax the constraints of the notation you are reading slightly and think about the intent of the author. What are they trying to get across? Perhaps you can even contact the author via email, Twitter, Facebook, Linked-in, etc, and seek clarification. Remember that academics want other people to understand and use their work (mostly).

### C.8.2 Check Wikipedia

Wikipedia has lists of notation which can help narrow down on the meaning or intent of the notation you are reading. Two places I recommend you start are:

- List of mathematical symbols on Wikipedia.
  https://en.wikipedia.org/wiki/List_of_mathematical_symbols
- Greek letters used in mathematics, science, and engineering on Wikipedia.
  https://en.wikipedia.org/wiki/Greek_letters_used_in_mathematics,_science,_and_engineering

### C.8.3 Sketch in Code

Mathematical operations are just functions on data. Map everything you're reading to pseudocode with variables, for-loops and more. You might want to use a scripting language as you go along with small arrays of contrived data or even an Excel spreadsheet. As your reading and understanding of the technique improves, your code-sketch of the technique will make more sense and at the end you will have a mini prototype to play with.

I never used to take much stock in this approach until I saw an academic sketch out a very complex paper in a few lines of Matlab with some contrived data. It knocked my socks off because I believed the system had to be coded completely and run with a *real* dataset and that the only option was to get the original code and data. I was very wrong. Also, looking back, the guy was gifted. I now use this method all the time and sketch techniques in Python.

### C.8.4 Seek Alternatives

There is a trick I use when I'm trying to understand a new technique. I find and read all the papers that reference the paper I'm reading with the new technique. Reading other academics interpretation and re-explanation of the technique can often clarify my misunderstandings in the original description. Not always though. Sometimes it can muddy the waters and introduce misleading explanations or new notation. But more often than not, it helps. After circling back to the original paper and re-reading it, I can often find cases where subsequent papers have actually made errors and misinterpretations of the original method.

### C.8.5 Post a Question

There are places online where people love to explain math to others. Seriously! Consider taking a screen shot of the notation you are struggling with, write out the full reference or link to it and put it and your area of misunderstanding to a question and answer site. Two great places to start are:

- Mathematics Stack Exchange.
  https://math.stackexchange.com/
- Cross Validated.
  https://stats.stackexchange.com/

## C.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Section 0.1. Reading Mathematics, *Vector Calculus, Linear Algebra, and Differential Forms*, 2009.
  http://amzn.to/2qarp8L
- The Language and Grammar of Mathematics, Timothy Gowers.
  http://assets.press.princeton.edu/chapters/gowers/gowers_I_2.pdf
- Understanding Mathematics, a guide, Peter Alfeld.
  http://www.math.utah.edu/~pa/math.html

## C.10 Summary

In this tutorial, you discovered the basics of mathematical notation that you may come across when reading descriptions of techniques in machine learning. Specifically, you learned:

- Notation for arithmetic including variations of multiplication, exponents, roots and logarithms.
- Notation for sequences and sets including indexing, summation and set membership.
- 5 Techniques you can use to get help if you are struggling with mathematical notation.

# Part X

# Conclusions

# How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- About the field of probability, how it relates to machine learning, and how to harness probabilistic thinking on a machine learning project.
- How to calculate different types of probability, such as joint, marginal, and conditional probability.
- How to consider data in terms of random variables and how to recognize and sample common discrete and continuous probability distribution functions.
- How to frame learning as maximum likelihood estimation and how this important probabilistic framework is used for regression, classification, and clustering machine learning algorithms.
- How to use probabilistic methods to evaluate machine learning models directly without evaluating their performance on a test dataset.
- How to calculate and consider probability from the Bayesian perspective and to calculate conditional probability with Bayes theorem for common scenarios.
- How to use Bayes theorem for classification with Naive Bayes, optimization with Bayesian Optimization, and graphical models with Bayesian Networks.
- How to quantify uncertainty using measures of information and entropy from the field of information theory and calculate quantities such as cross-entropy and mutual information.
- How to develop and evaluate naive classifiers using a probabilistic framework.
- How to evaluate classification models that predict probabilities and calibrate probability predictions.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable foundational skills in probability. You can now confidently:

- Confidently calculate and wield both frequentist probability (counts) and Bayesian probability (beliefs) generally and within the context of machine learning datasets.

- Confidently select and use loss functions and performance measures when training machine learning algorithms, backed by a knowledge of the underlying probabilistic framework (e.g. maximum likelihood estimation) and the relationships between metrics (e.g. cross-entropy and negative log-likelihood).
- Confidently evaluate classification predictive models, including establishing a robust baseline in performance, probabilistic performance measures, and calibrated predicted probabilities.

The sky's the limit.

## Thank You!

I want to take a moment and sincerely thank you for letting me help you start your journey with probability for machine learning. I hope you keep learning and have fun as you continue to master machine learning.

Jason Brownlee
2020