Machine Learning for Engineering Problem Solving

A Practical Example-driven Guide to Classical Techniques

Austin R.J. Downey
Department of Mechanical Engineering
Department of Civil and Environmental Engineering
University of South Carolina, Columbia SC, USA

July 5, 2026

DOI:10.31224/4909

Contents

Preface

This text is a free, open-source textbook that introduces traditional machine learning through practical engineering examples. Designed for undergraduate engineering students and readers with limited programming experience, the text uses Python and the scikit-learn library to connect core concepts with hands-on implementation. The book is intended to serve as a self-contained introduction for a college-level machine learning course in an engineering curriculum. This preface collects the essential housekeeping information for using this text.

Accompanying Video Lectures

Videos of lectures associated with this text are available as a playlist here.

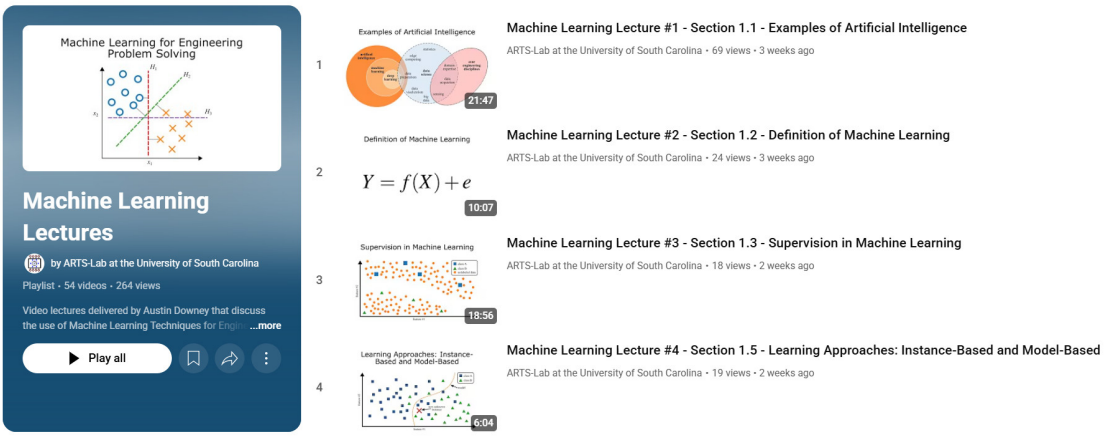


Figure 1: Playlist of videos associated with this text.

Programming in Python

This text uses Python programmed though the Spyder IDE managed through the Anaconda platform for the examples, leveraging the `scikit-learn` library to explain the topics discussed. To assist readers of the text, a six part video series that walks the practitioner though this combination of IDE and distribution manager is provided as a playlist here.

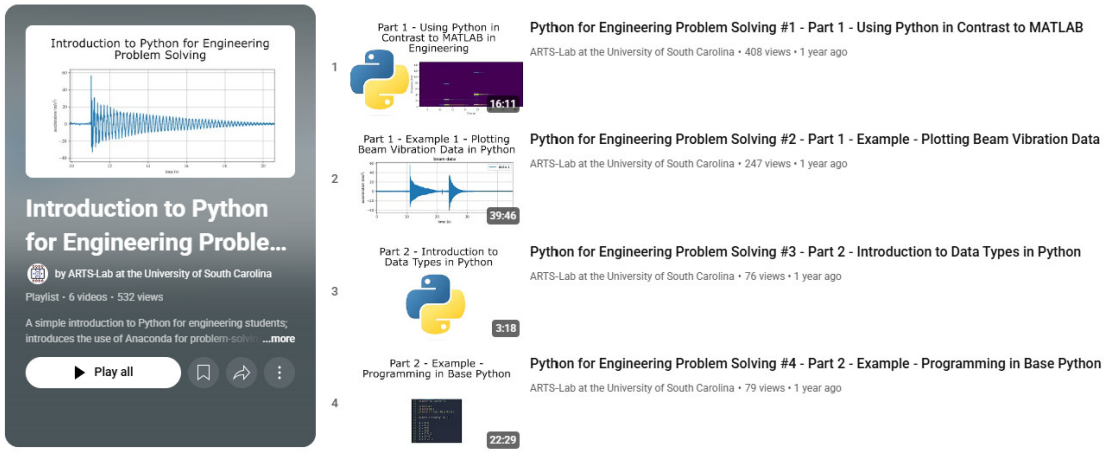


Figure 2: Playlist of videos for leaning how to program in Python using Spyder and Anaconda.

Cover Art

The cover image, is a mid-1930s International Harvester C-series truck. Built between 1934 and 1936 at the company's Springfield Works in Springfield, Ohio. Roughly 80,000 C-series trucks where made during this short time. The C-series was International's first line to feature an all-steel cab and a host of mechanical upgrades over the prior W-series. This truck was photographed by Ethan McNeese on Washington Island in Door County, Wisconsin in 2021.

The truck on the cover underscores the simple idea that a machine's purpose is to turn raw input into useful work. The same principle drives machine learning, where digital "machines" transform data into insight and data-informed actions.

License

Writing and Figure Development Tools

Generative AI and automated grammar/editing tools were used in selected aspects of this work to support writing, editing, and figure preparation.

Source Code

The source code for this text is available here.

Questions and Contact Information

For questions, corrections, or requests, contact Austin Downey at austindowney@sc.edu.

1 Basic Concepts in Machine Learning

Machine Learning (ML) is a method of data analysis that automates analytical model building. ML is closely coupled to data science, as shown in figure 1.1, which is a multidisciplinary field that utilizes scientific methods, processes, algorithms, and systems to extract knowledge and insights from structured and unstructured data. It is based on the idea that systems can learn from data, identify patterns, and make decisions with minimal human intervention. This section introduces some foundational concepts in machine learning and its applications.

- **ML is not creating robots**: A common misconception is that machine learning is about building robots. In reality, ML focuses on developing algorithms that can learn from and make predictions or decisions based on data.
- **The SPAM filter is one of the initial ML uses**: One of the earliest and most common applications of ML is the spam filter, which classifies emails as spam or not spam. This has been followed by numerous other applications such as:
  - **Speech to text technology**: Converting spoken language into written text, which is used in virtual assistants and transcription services.
  - **Medical diagnostics**: Assisting doctors by predicting diseases from medical images and patient data.
- **ML has lots of fundamental concepts (jargon)**: To effectively understand and apply machine learning, it's essential to grasp several key concepts and terminologies, including:
  - **Supervised vs unsupervised learning**: Supervised learning involves training a model on labeled data, while unsupervised learning deals with data that has no labels and tries to find hidden patterns.

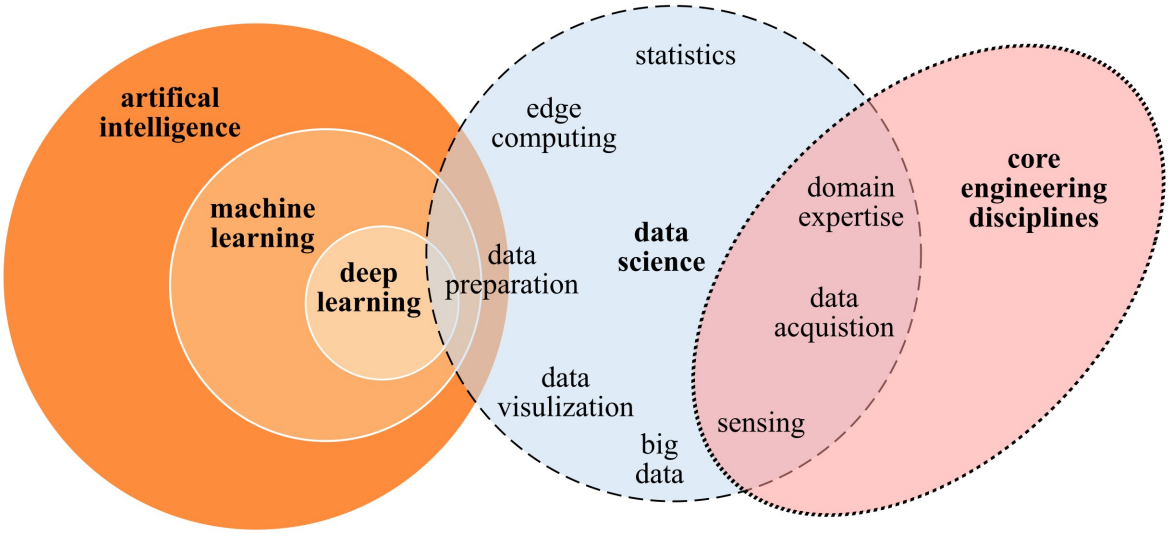


Figure 1.1: Overlap between the fields of Artificial Intelligence (AI), Data science (DS), and the core engineering disciplines of Civil, Mechanical, Electrical, and Chemical Engineering.

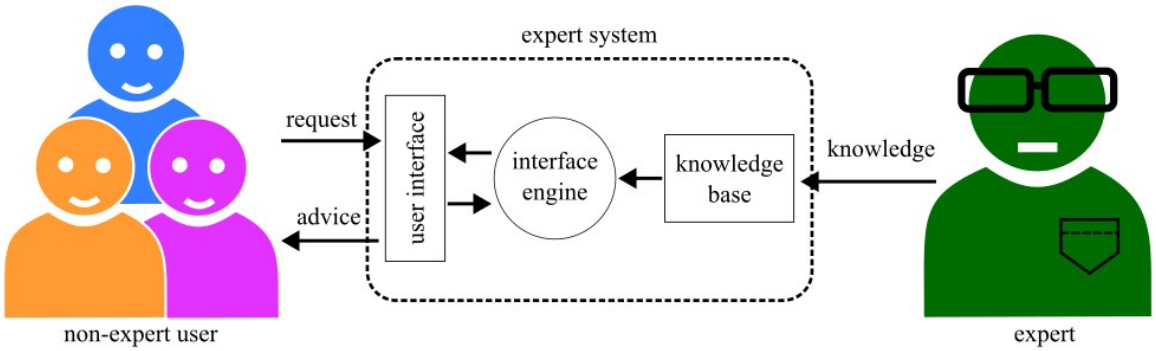


Figure 1.2: Diagram of an "expert system" in AI, which is a computer program that simulates the decision-making ability of a human expert by using a knowledge base and inference rules.

- **Online versus batch learning**: Online learning algorithms update the model incrementally as new data arrives, whereas batch learning algorithms train the model using the entire dataset at once.
- **Instance-based vs model-based learning**: Instance-based learning algorithms, such as k-nearest neighbors, use specific instances to make predictions, whereas model-based algorithms, like linear regression, build a model from the training data and use it to make predictions.

1.1 Examples of Artificial Intelligence

The field of Artificial Intelligence (AI) encompasses a diverse array of technologies and methodologies aimed at enabling machines to perform tasks that typically require human intelligence. This section explores various examples of AI, Machine Learning, and Deep Learning technologies, highlighting their distinct characteristics and applications.

1.1.1 Examples of Artificial Intelligence

Artificial Intelligence (AI) encompasses a wide range of technologies aimed at making machines simulate human intelligence. Some examples include:

- **Expert systems**: Computer programs that simulate the decision-making ability of a human expert by using a knowledge base and inference rules, as diagrammed in figure 1.2.
- **Chatbots**: Programs designed to simulate conversation with human users, especially over the internet.

Figure 1.3: A Symbolics Lisp Machine, a specialized hardware platform designed to run expert systems which are a version of AI focusing on answering questions to challenging problems.[a]

1.1.2 Examples of Machine Learning

Machine Learning, as a subset of AI, involves algorithms that improve automatically through experience. Some common examples include:

- **Linear regression**: A statistical method for modeling the relationship between a dependent variable and one or more independent variables.
- **Classification**: Techniques such as decision trees and support vector machines (SVMs) that categorize data into predefined classes.
- **Simple image/speech recognition**: Algorithms that identify objects in images or convert spoken language into text, fundamental in applications like facial recognition and virtual assistants. An example is the Symbolics Lisp Machine shown in figure 1.3.

[a]Michael L. Umbricht and Carl R. Friend (Retro-Computing Society of RI), CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

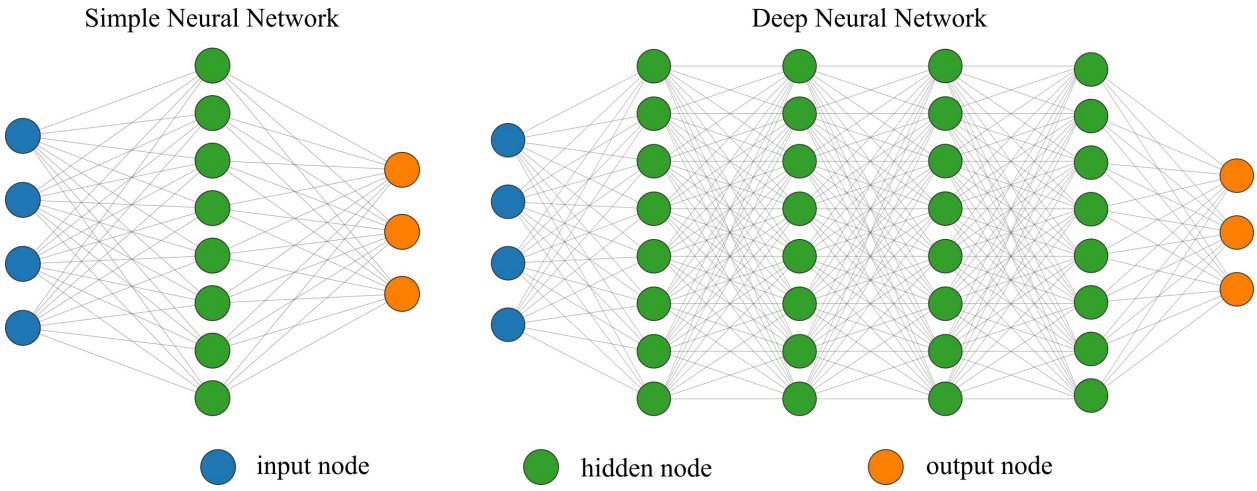
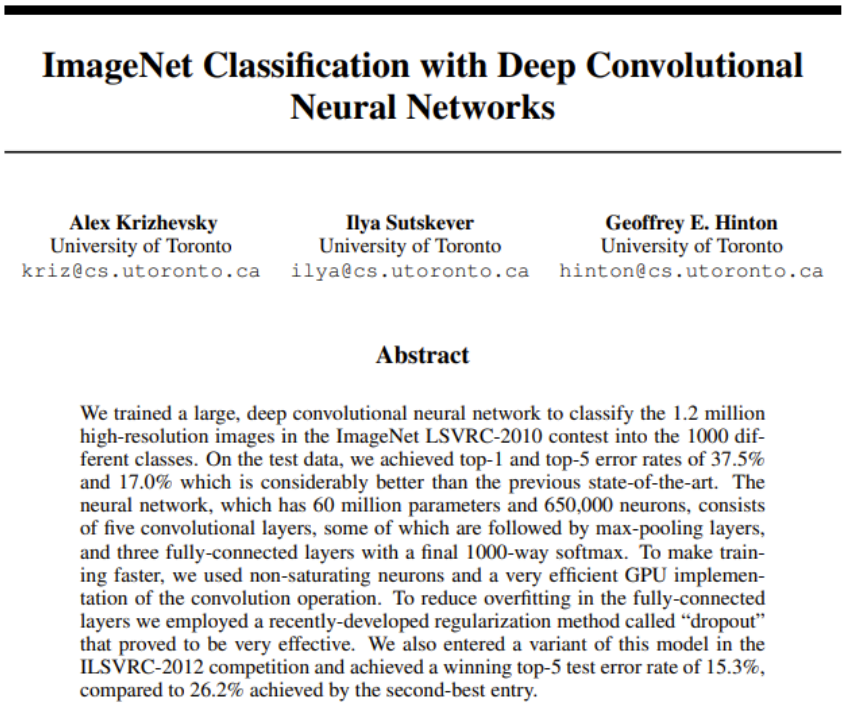


Figure 1.4: Simple neural network vs deep learning.

1.1.3 Examples of Deep Learning

Deep learning, an enhancement of machine learning, utilizes "deep" neural networks to construct knowledge graphs, as shown in figure 1.4. Though conceptualized in the 1970s, it was initially unfeasible due to the problem of vanishing gradients within the networks. In 2012, Geoffrey E. Hinton's team demonstrated that a network with 60 million parameters and 650,000 neurons could effectively perform image classification across a dataset containing 1,000 categories (paper shown in figure 1.5). This breakthrough was facilitated by the use of GPUs and a novel regularization technique known as "dropout." The team's modified model participated in the ILSVRC-2012 competition, securing a first-place top-5 test error rate of 15.3%, a significant improvement over the 26.2% recorded by the runner-up and a major leap when compared to the previous year as diagrammed in figure 1.6.

Review 1.1 Imagenet in 2012 represented a significant step forward in machine learning by introducing the first practical example of deep learning, famously known as AlexNet (Figure 1.5) This model, developed by Geoffrey Hinton and his team, utilized deep convolutional neural networks to dramatically improve the accuracy of image classification, which was a longstanding challenge in the field.



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

Figure 1.5: Geoffrey's 2012 paper "ImageNet Classification with Deep Convolutional Neural Networks".[a]

The results of AlexNet were rather monumental, reducing the top-5 test error rate to 15.3% compared to 26.2% by the next best entry, as shown in Figure 1.6. This was clear evidence of deep learning's superior capability over traditional machine learning methods, effectively revolutionizing the approach towards machine learning in the broader scientific community.

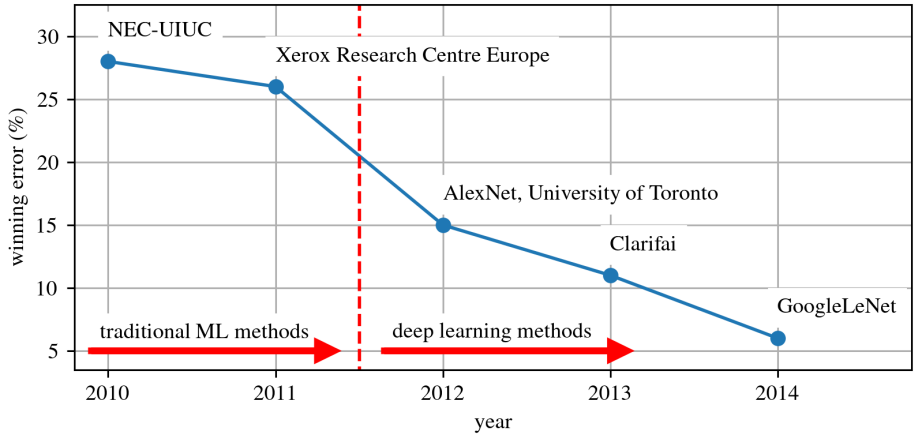
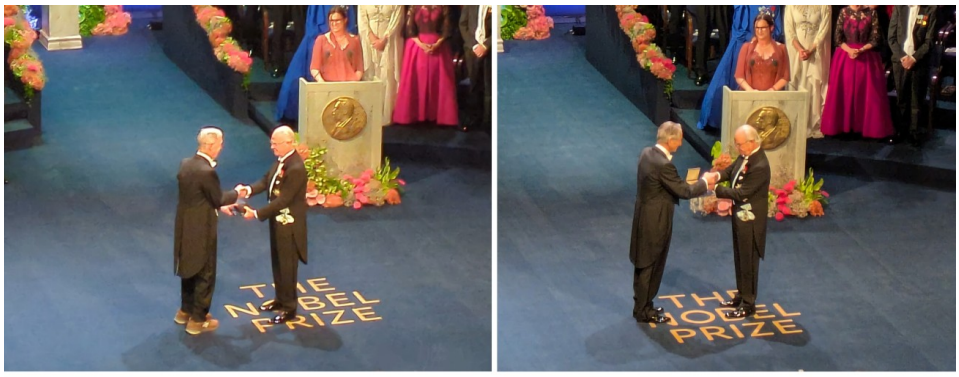


Figure 1.6: ImageNet Competition Results showing the impact of deep learning methods on image classification.

The implications of this leap forward led to broad applications of deep learning that permeate numerous aspects of technology and science today. The impact of AlexNet and subsequent deep learning developments culminated in the awarding of the 2024 Nobel Prize in Physics to Geoffrey Hinton, recognizing his contributions to the field of artificial neural networks. Figure 1.7 shows the 2024 Nobel Award Ceremony in Stockholm Sweden with John Hopfield and Geoffrey Hinton receiving their awards from the King of Sweden.



(a) (b)

Figure 1.7: The 2024 Nobel Prize in Physics "for foundational discoveries and inventions that enable machine learning with artificial neural networks" was awarded to: (a) John Hopfield and (b) Geoffrey Hinton. [a]

[a] Copyright held by Authors and Neural Information Processing Systems Foundation, Inc., used under fair use. <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>

[a] The author of this text was lucky enough to attend the 2024 Nobel Prize Ceremony in Stockholm and took these pictures.

1.2 Definition of Machine Learning

The definitions and conventions that follow provide a common language and point of reference for all subsequent material:

- Machine learning is the discipline of creating computer programs that improve automatically by analyzing data. Here a broader and a more engineering-focused definition of Machine learning is provided5:
  - [Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed. Arthur Samuel, 1959.
  - A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E. Tom Mitchell, 1997.
- The use of X and Y for variables.
  - For the i^{th} sample, $x^{(i)}$ contains all its input features (excluding the target), while $y^{(i)}$ denotes the corresponding target value.
  - General definition: "Machine learning algorithms are described as learning a target function (f) that best maps input variables (X) to an output variable (Y)."

$$Y = f(X) \tag{1.1}$$

This is described as a standard learning challenge where the goal is to predict future values Y using new samples of input variables X. The function f that relates inputs to outputs is not known. If it were, direct application would be possible, eliminating the necessity for learning it via machine learning methods. This process is more complex than it may initially seem. Furthermore, there is an error e associated with this task that is independent of the input data X.

$$Y = f(X) + e \tag{1.2}$$

This yields two primary phases in the machine-learning workflow:

- Training: Creating the model is a compute-intensive process often run in a data center
- Inference: Using the model can be computationally cheap and even performed "at the edge"

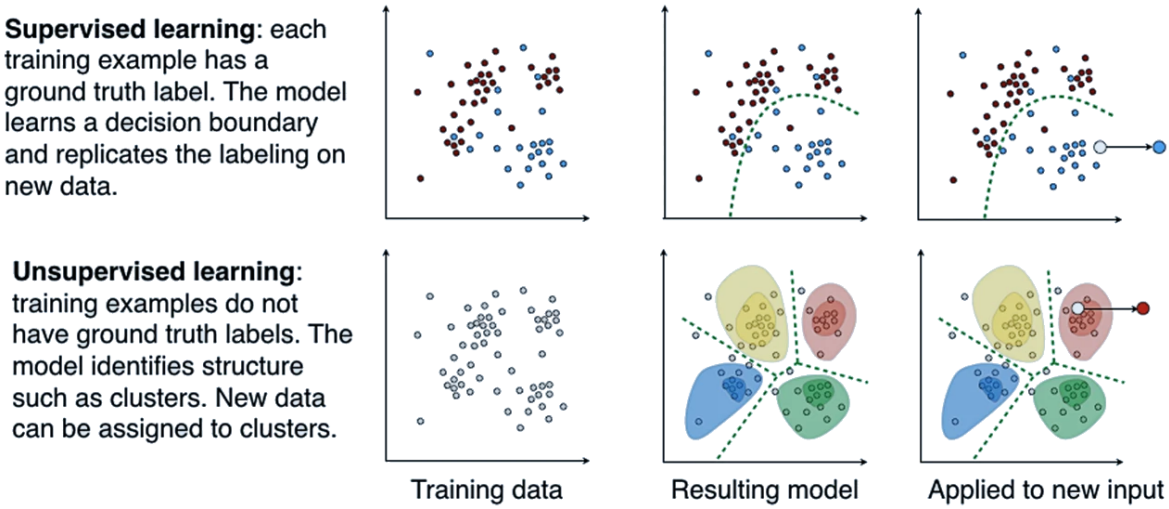


Figure 1.8: Supervised vs unsupervised unsupervised machine learning methods[a].

1.3 Supervision in Machine Learning

The general domains of ML can be classified into supervised and unsupervised learning, as shown in figure 1.8.

1.3.1 Supervised Learning

In supervised learning, the data is pre-categorized with labels

- **Classification**, The process of categorizing a given set of data into classes
- **Regression**, The process of estimating the relationships between a dependent variables (i.e. output) and one or more independent variables (i.e. input).

In supervised learning, the training data provided to the algorithm includes the desired solutions, known as labels. A common task within this type of learning is classification. An example of classification is a spam filter, which is trained using numerous emails that are each labeled as either spam or ham. The objective for the spam filter is to learn how to accurately classify new emails based on this training. Regression is another typical task is to predict a target numeric value, such as the price of a house, given a set of features (size, proximity to railroad or highways, age of house, style, etc.) called predictors. To train a system you need to build a model and train in on many house sale examples, including both predictors (house features) and their labels (i.e., prices). Some of the most important supervised learning algorithms:

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests

[a]modified from: Langs, G., Röhrich, S., Hofmanninger, J. et al., CC BY 4.0 <https://creativecommons.org/licenses/by/4.0>, via Wikimedia Commons

Figure 1.9: A semantic word cloud of Barack Obama's First Inaugural Address [a].

1.3.2 Unsupervised Learning

In unsupervised learning, the training data is unlabeled, and the system endeavors to learn independently without explicit guidance. For instance, consider a situation where you have extensive data on what people watch on YouTube. You might use a clustering algorithm to identify groups of similar users. However, at no point do you instruct the algorithm on which group a user belongs to; it discovers these connections autonomously. For example, it might identify that the types of videos people watch are closely linked to specific age and income metrics. Older viewers may prefer watching videos about vacations, while younger viewers might watch a lot of educational content, such as Khan Academy.

Visualization algorithms such as that in Figure 1.9) serve as an important instance of unsupervised learning algorithms: they take in extensive, complex, and unlabeled data and produce a 2D or 3D representation that can be conveniently visualized. These algorithms strive to maintain as much of the original data structure as possible (for example, ensuring that distinct clusters in the input space do not merge in the visual output). This helps in comprehending the organizational structure of the data and potentially uncovering hidden patterns.

[a]Modified from: GuoYongzhi, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

Other Important concepts in unsupervised learning include

- **Clustering**, The process of identifying and grouping similar data points in larger datasets without concern for the specific outcome
- **Association**, The process learning a rule-based method for discovering relations between variables data data
- **Dimension Reduction**, The process of reducing the number of input variables in training data.

Here are some of the most important unsupervised learning algorithms:

- Clustering
- k-Means
- Hierarchical Cluster Analysis (HCA)
- Expectation Maximization

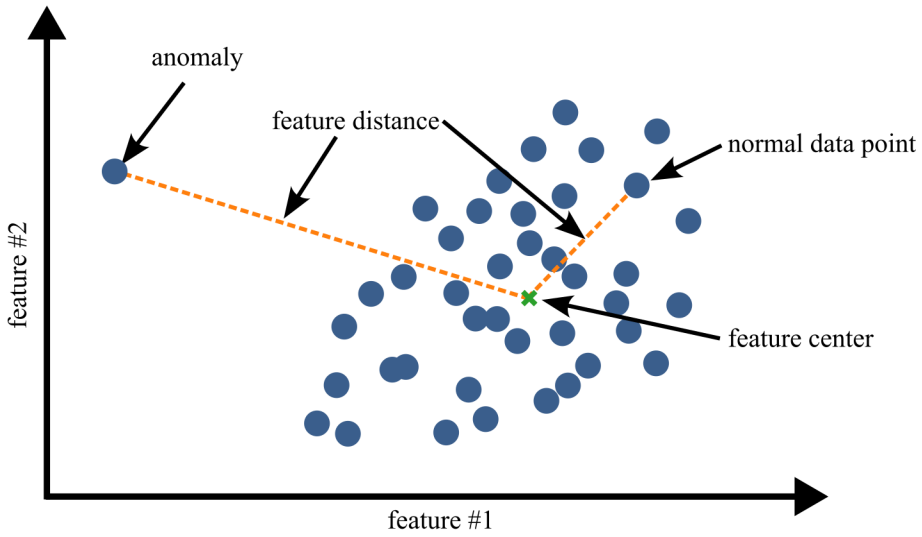


Figure 1.10: Anomaly Detection for a given set of data.

Another key application of unsupervised learning is anomaly detection as shown in figure 1.10. The goal is to flag unusual credit-card transactions that may signal fraud, spot defects on a production line, or filter out outliers before passing a dataset to another learning algorithm. The model is first exposed to many examples of normal behavior so it can build a reference profile. When a new observation arrives it checks how closely the example matches that profile and labels it normal or anomalous accordingly.

1.3.3 Semisupervised Learning

Certain algorithms are designed to manage partially labeled training data, typically characterized by a substantial amount of unlabeled data with a minimal amount of labeled data as shown in figure 1.11. Many semi-supervised learning algorithms are hybrid forms, integrating features of both unsupervised and supervised learning approaches.

An example is seen in image hosting services like Google Photos. Upon uploading your family images, the system automatically recognizes and groups repeated appearances of the same individuals across different photos, such as person 1 in photos A, C, and D and person 2 in photos A, B, and E. This clustering is the unsupervised component of the algorithm. The system then only needs a single label for each person to subsequently identify and tag everyone in all images, enhancing the ease of searching through photos.

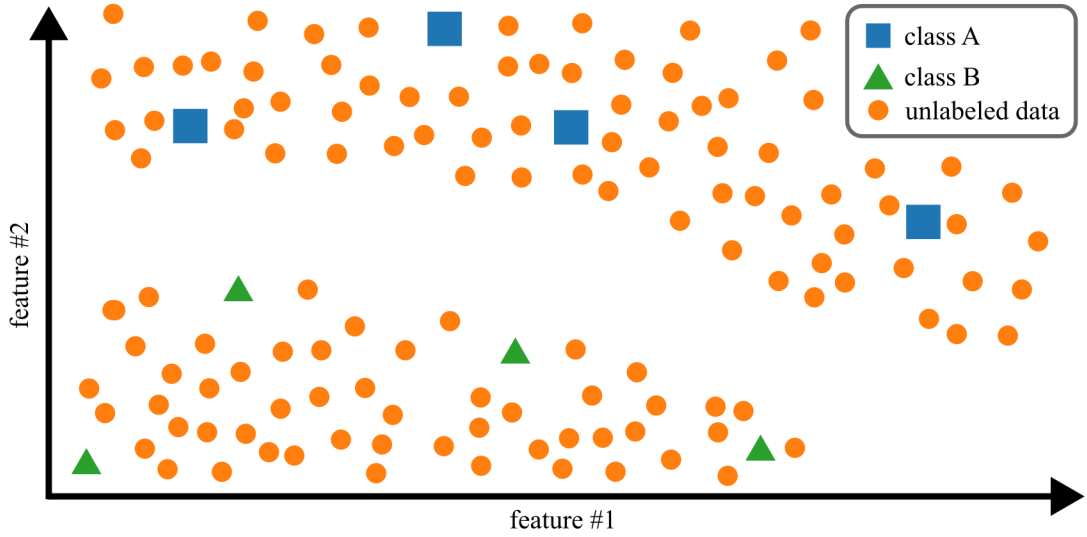


Figure 1.11: Semisupervised learning enabling the use of a limited set of labeled data to infer the labels of larger unlabeled datasets.

1.3.4 Reinforcement Learning

Reinforcement learning follows a distinct paradigm in which an autonomous agent repeatedly interacts with its environment. At each step the agent observes the current state, chooses and executes an action, and then receives feedback in the form of a reward or penalty; as shown in figure 1.12. By sampling many such cycles the agent gradually discovers a strategy, known as a policy, that maps perceived situations to actions so as to maximise the total reward accumulated over time.

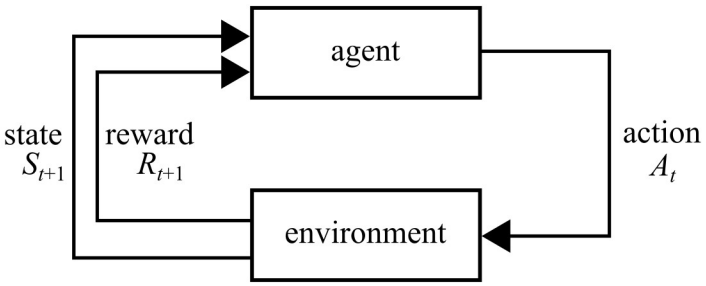


Figure 1.12: Reinforcement learning diagram.

1.4 Types of learning (Batch and Online)

Machine learning systems can be broadly categorized based on how they process and learn from data.

1.4.1 Batch Learning

In batch learning, the system is fully trained using the entirety of the available data, which often demands significant time and computational resources, and is thus usually performed offline. Initially, the system undergoes training; after which, it is deployed into production where it no longer learns but merely applies the previously acquired knowledge. This method is referred to as offline learning. The deployment phase, known as inference, is typically executed swiftly.

To update a batch learning system with new information, such as recognizing a novel form of spam, it is necessary to develop a completely new version of the system. This version must be trained from the ground up using the entire dataset, both old and new data, before replacing the older system in production.

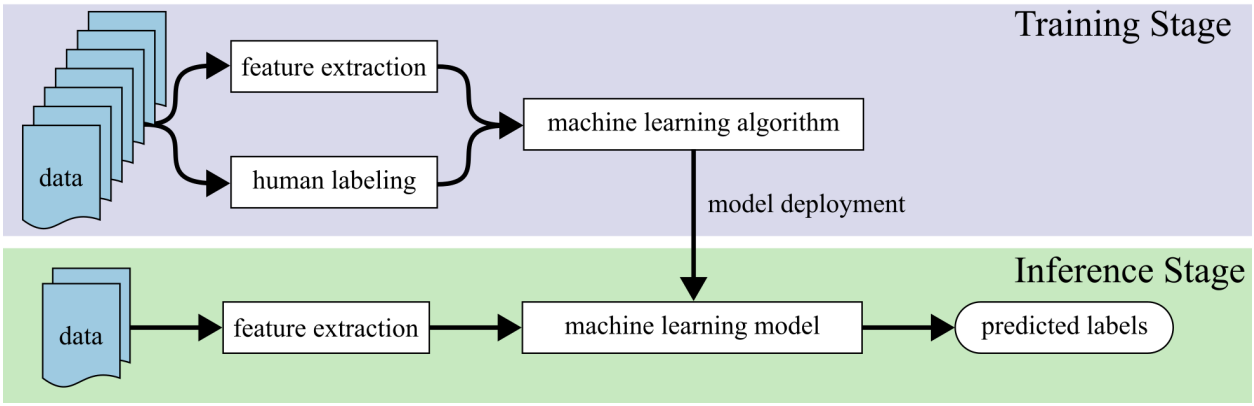


Figure 1.13: Batch learning framework with separate training and inference stages.

Despite these challenges, the training, evaluation, and deployment processes of a machine learning system can be automated, allowing even batch learning systems to adapt to changes. This approach is straightforward and usually effective; however, training on a full dataset can be time-consuming - often taking many hours - hence, systems are usually updated no more frequently than

daily or weekly. Moreover, utilizing the full dataset requires extensive computing resources, such as CPU power, memory, disk space, and network bandwidth. For organizations with vast amounts of data, the costs of daily retraining from scratch can be prohibitively expensive.

If the dataset is exceptionally large, employing a batch learning algorithm may become impractical. Additionally, in situations where autonomous learning is essential and computational resources are limited, such as with smartphone applications or extraterrestrial rovers, the need to manage large datasets and conduct lengthy training sessions daily poses significant challenges.

1.4.2 Online Learning

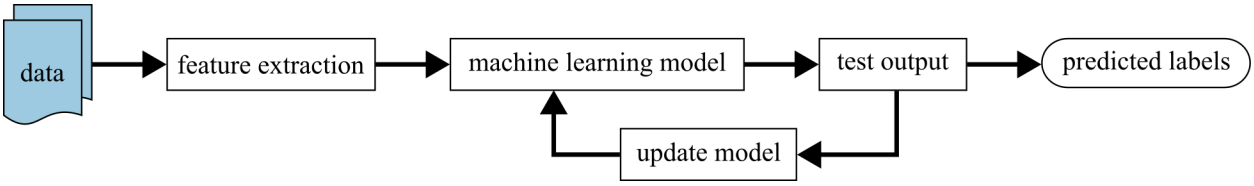


Figure 1.14: Online learning framework where data is used to continuously training (or update / fine-tune) the model.

Online learning trains a model incrementally by presenting new data points one at a time or in small mini-batches. Each update is computationally light and fast, allowing the model to refresh its knowledge continuously as fresh data streams in. This approach is particularly appropriate for systems that:

- receive data continuously, such as stock prices,
- need to quickly or autonomously adapt to changes,
- have limited computing resources: once an online learning system processes new data instances, they can be discarded to save space, unless there is a need to revert to a previous state and "replay" the data.

Online learning is also useful for managing large datasets that exceed the memory capacity of a single machine, known as out-of-core learning. The algorithm processes parts of the data, conducts a training step, and repeats this until all the data has been processed.

A critical parameter in online learning systems is the learning rate, which dictates how rapidly the system adapts to changing data. A high learning rate allows for rapid adaptation but may also lead to quick forgetting of old data and training on noise. A low learning rate results in slower learning and reduced sensitivity to variations in new data.

1.4.3 Bad Data in Machine Learning

A significant challenge in online learning is the system's susceptibility to performance degradation when exposed to poor quality data. This is particularly problematic in live systems where clients might quickly notice issues. For instance, bad data could originate from a malfunctioning robot sensor or an attempt to manipulate search engine rankings through spamming. To mitigate these

risks, it is crucial to vigilantly monitor the system and quickly disable learning or revert to a previously effective state if a decline in performance is observed. Additionally, monitoring the input data for anomalies and employing anomaly detection algorithms can help identify and respond to aberrant data.

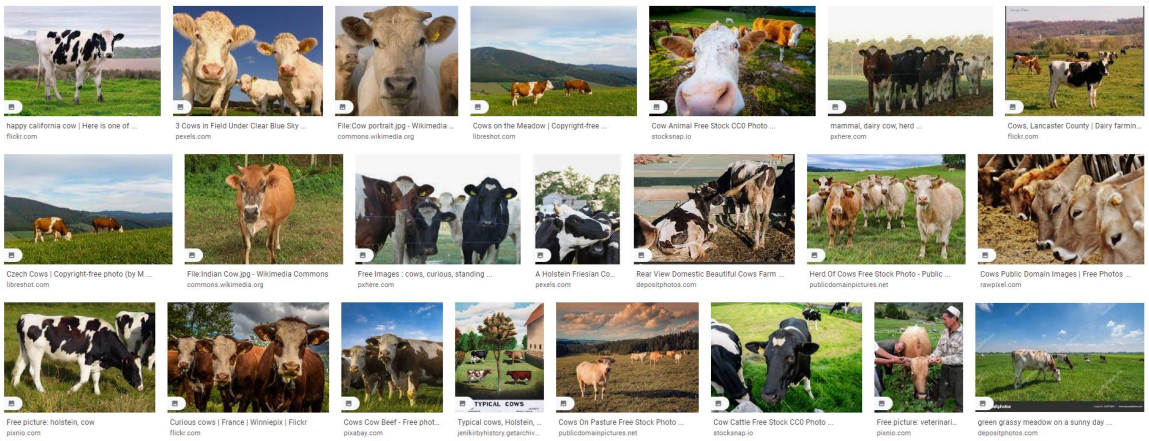


Figure 1.15: Training a machine learning algorithm to recognize cows may end up just learning to recognize grass. humorously called "short-cut learning"[a].

For example, imagine training a convolutional neural network to recognize cows (Figure 1.15). Every image in the training set shows black-and-white cattle standing on lush green pasture, so the easiest statistical cue for the model to latch onto is the dominant green background rather than the animals themselves. At inference time, the network confidently labels any scene filled with grass as "cow," yet fails when presented with a cow on snow or asphalt. In other words, it has learned "grass recognition," not "cow recognition".

1.5 Learning Approaches: Instance-Based and Model-Based

Another method of classifying machine learning systems is based on their generalization capabilities. Typically, the main objective in machine learning is to make predictions, which requires the system to generalize from its training examples to new, unseen instances. Achieving high performance on training data is useful but not the ultimate goal; the system must also excel when confronted with new data. There are primarily two strategies for generalization: instance-based learning and model-based learning.

[a]Screen shoot of a Google image search for cows taken by the Austin Downey, all images stated to be under a creative commons license per Google search tools; also assumed to be fair use under given the educational purpose of this text, via https://www.google.com/

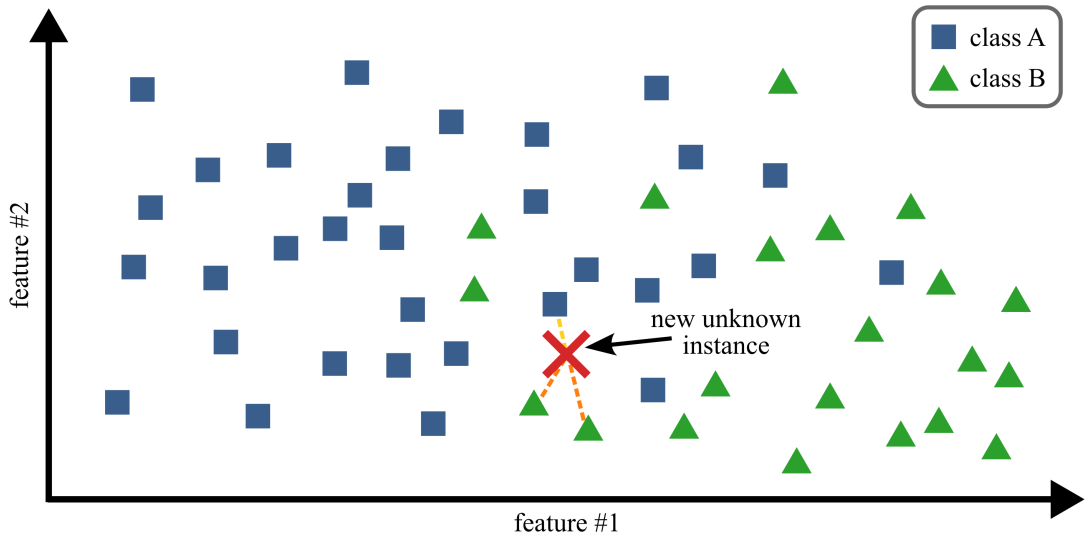


Figure 1.16: Instance-based learning where the class of a unknown instance is inferred from the its distance to data points with known labels.

1.5.1 Instance-based learning

One of the simplest learning strategies is rote memorization. For instance, a spam filter built on this principle would only mark emails as spam if they match previously flagged emails exactly. While this approach is straightforward, it's hardly the most effective. A more sophisticated spam filter could extend its detection capabilities to include emails that closely resemble known spam messages. This approach necessitates a metric for measuring the similarity between two emails. A basic method might involve counting the shared words between emails. Under this system, an email would be classified as spam if it shares a significant number of words with an email already identified as spam. This method exemplifies instance-based learning (Figure 1.16), where the system memorizes specific examples and uses a similarity metric to generalize to new instances.

1.5.2 Model-based learning

Another approach to generalization involves constructing a model based on a set of examples and then using this model to make predictions as shown in figure 1.17. This method is known as model-based learning.

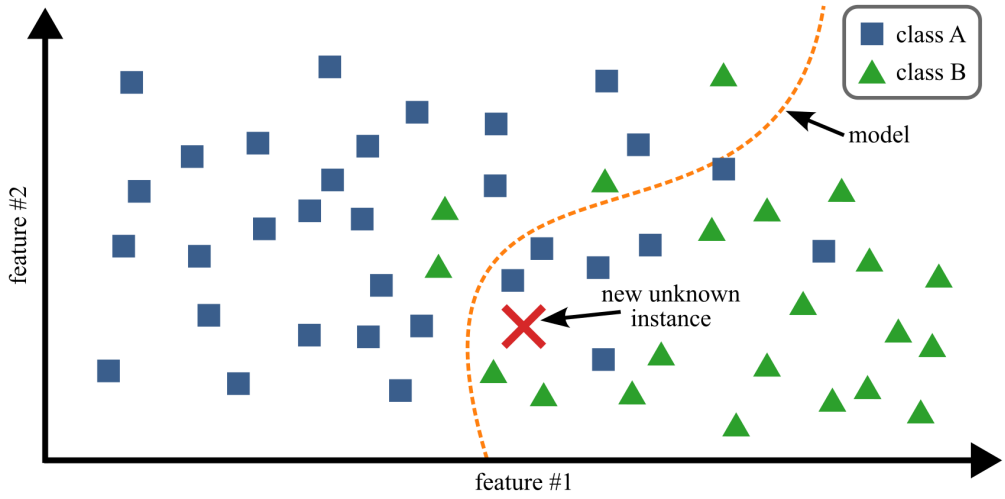


Figure 1.17: Model-based learning where the class of a unknown instance is inferred from its location in reference to a model trained on the data with known labels.

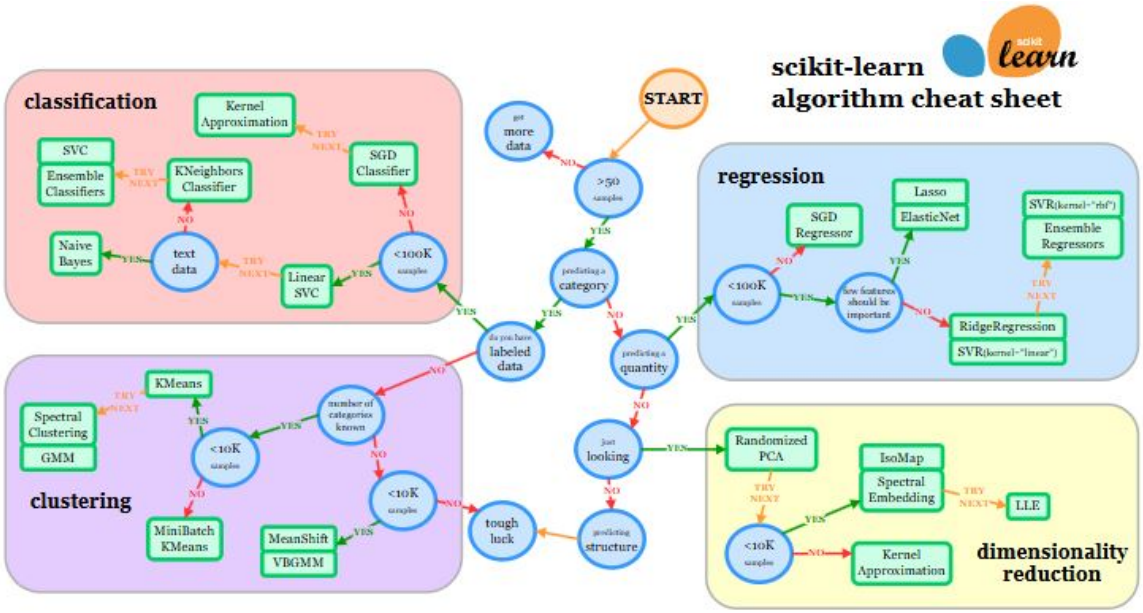


Figure 1.18: Flowchart of estimators used in the scikit learn library that intends guide users for what algorithms to use for a given case[a].

[a]Scikit-learn algorithms cheat sheet, permissive simplified BSD license and assumed to be fair use under given its nature as documentation and the educational purpose of this text, via https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

1.5.3 Selecting methods:

One of the initial challenges faced by newcomers to machine learning is selecting the appropriate algorithms (or estimators) for specific tasks. This is due to the fact that various algorithms excel with different kinds of data and problems. The flowchart depicted in Fig. 1.18, which originates from the scikit-learn library documentation, is intended to guide users in choosing the most suitable algorithms for their particular datasets.

1.6 The Unreasonable Effectiveness of Data

In a landmark study released in 2001, Microsoft researchers Michele Banko and Eric Brill demonstrated that a variety of Machine Learning algorithms, even relatively straightforward ones, achieved similar levels of performance on a complex task of natural language disambiguation when provided with sufficient data. This finding is illustrated in figure 1.19 and further discussed in their paper[a].

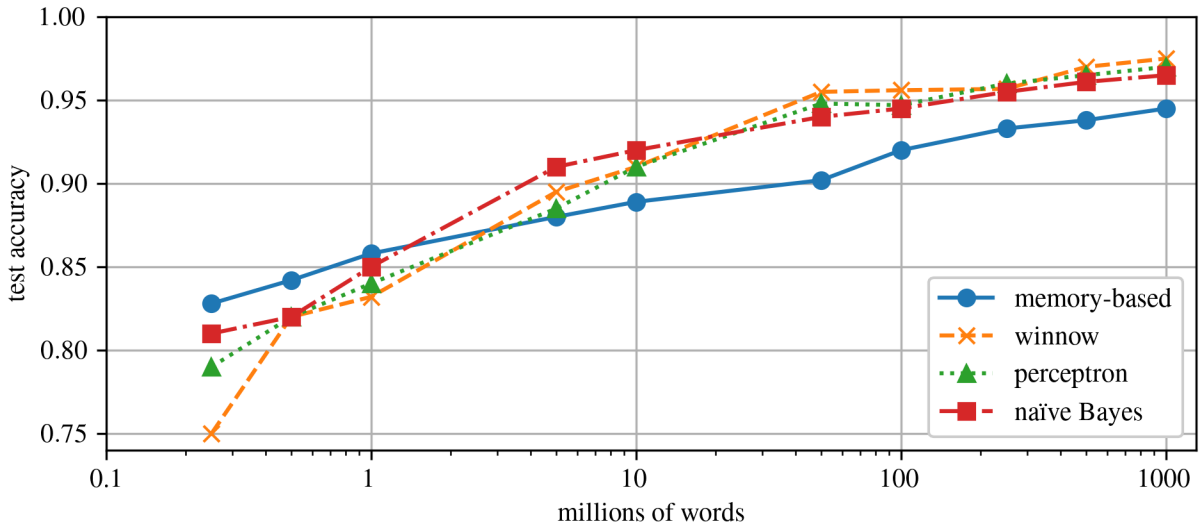


Figure 1.19: Learning curves for four algorithms studied in Banko and Brill showing the importance of the amount of data when compared to algorithm selection.

As the authors put it: "these results suggest that we may want to reconsider the tradeoff between spending time and money on algorithm development versus spending it on corpus development.". Or put more directly "The Unreasonable Effectiveness of Data"[b]. It's important to recognize, however, that small and medium-sized datasets remain prevalent, and acquiring additional training data is not always straightforward or economical. Therefore, the significance of algorithm selection should not be overlooked.

[a] Michele Banko and Eric Brill. "Scaling to very very large corpora for natural language disambiguation." Proceedings of the 39th annual meeting of the Association for Computational Linguistics. 2001.

[b] Halevy, Alon, Peter Norvig, and Fernando Pereira. "The unreasonable effectiveness of data." IEEE intelligent systems 24.2 (2009): 8-12.

2 Regression

Regression is a fundamental tool in machine learning used to model the relationship between an independent variable (called data and denoted x) and a dependent variable (called target and denoted y). The goal is to learn a function that best predicts the target value from the input data. This relationship is shown in figure 2.1 .

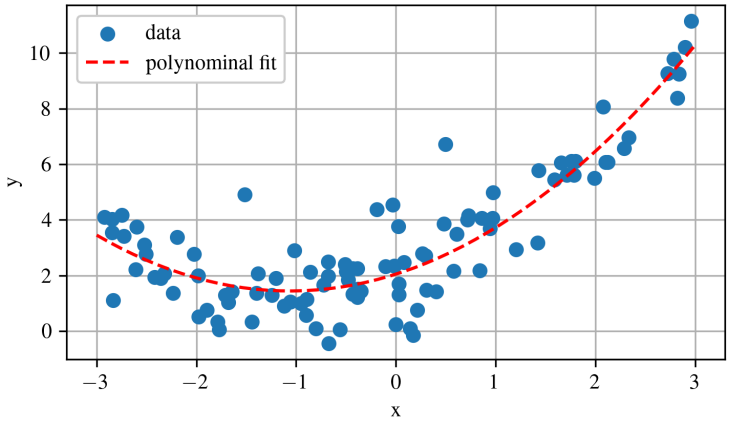


Figure 2.1: Polynomial model fit via regression to a generic dataset.

In this chapter, we first examine Linear Regression and contrast two ways to estimate its parameters:

1. **Closed-form solution.** Solve for the parameter vector in one step by minimizing the cost function on the entire training set.

2. **Gradient Descent .** Apply an iterative optimizer that repeatedly updates the parameters in small steps that lower the cost until it reaches the same optimum found by the closed-form method. We will look at three common Gradient Descent variants: Batch Gradient Descent, Mini-batch Gradient Descent, and Stochastic Gradient Descent.

The first approach gives an exact answer immediately, whereas the second reaches that answer through successive refinements.

Review 2.1 Ames Housing Dataset

The Ames housing dataset provides details on individual residential property sales in Ames, Iowa, from 2006 to 2010 [a]. It includes 2,930 entries and numerous variables (23 nominal, 23 ordinal, 14 discrete, and 20 continuous) that are used to evaluate home values [b].

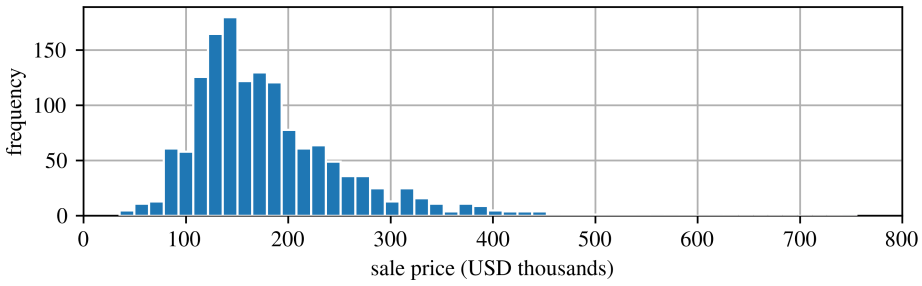


Figure 2.2: Histogram of sale prices from 2006 to 2010 for the 2,930 entries.

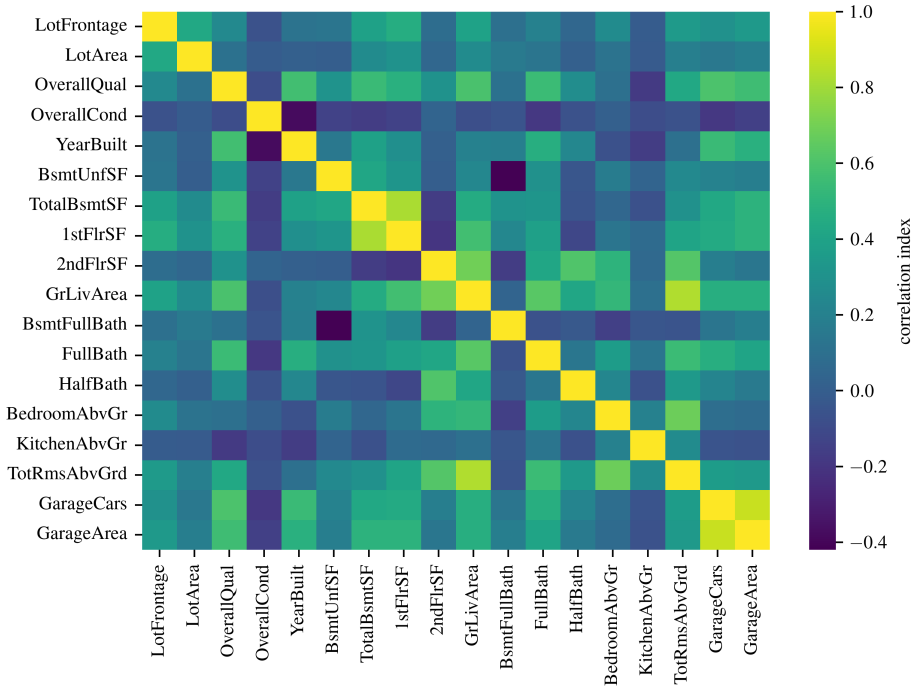


Figure 2.3: Correlation matrix for the 13 input parameters.

[a] The author of this text bought a home in Ames Iowa during this time right as the 2008 housing crash unfolded and on the last day they allowed home loans with no down payment.

[b] De Cock, Dean. "Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project." Journal of Statistics Education 19.3 (2011).

2.1 Linear Regression

Consider the scenario where you wish to determine the value of a house in Ames using the Ames dataset included in sklearn. To illustrate this, let us plot the relationship between the above ground living area and the housing price.

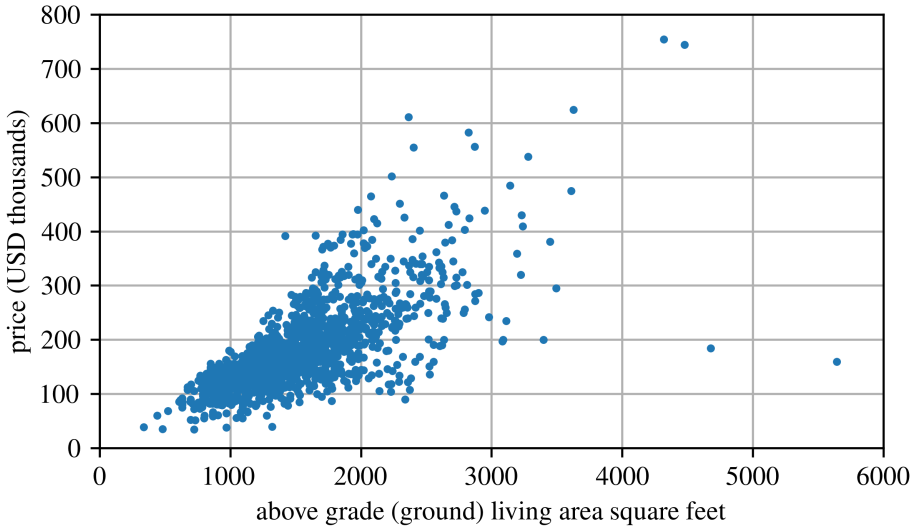


Figure 2.4: Ames housing data.

A clear trend is observable, despite the data's noise (i.e., partial randomness): the value of a house increases with the above ground living area. Therefore, the house value can be modeled as a linear function of the above ground living area:

$$\text{price\_model} = \theta_1 + \theta_2 \times \text{above\_ground\_living\_area} \tag{2.1}$$

This model comprises two parameters, θ_1 and θ_2, which can be adjusted to represent any linear relationship.

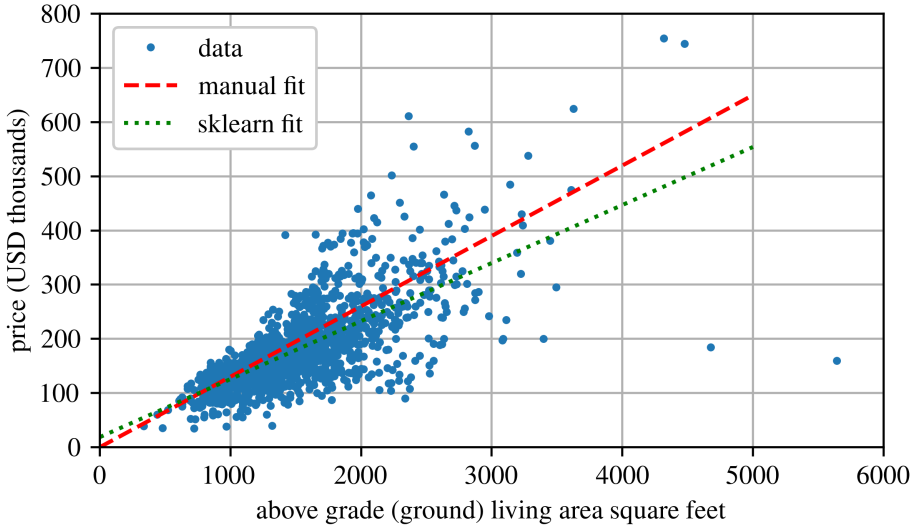


Figure 2.5: Ames housing data with trend lines.

Notations

In terms of linear algebra notation; we use:

- $\mathbf{x}$: lowercase bold symbols denote vectors.
- $\mathbf{X}$: uppercase bold symbols denote matrices.

In machine learning, we commonly use:

- $\mathbf{X}$: the feature matrix (input data).
- $\mathbf{y}$: the target vector (labels or outputs).
- $\mathbf{x}^{(i)}$: the i^{th} instance in the dataset.
- h: the hypothesis or prediction function, $\hat{\mathbf{y}} = h(\mathbf{X})$.
- n: the number of features.
- m: the number of training instances.
- $\hat{x}$: an estimated value ("x-hat").

2.1.1 Closed Form Solution

Before you can tune a model's parameters, you need a clear way to judge how well it fits the data. The standard approach is to define a cost function that assigns a numerical penalty to poor fits; lower values indicate better performance. For a linear model, this cost function measures the difference between the model's predictions and the actual training targets, and you minimize that difference using Ordinary Least Squares linear regression. Practically, you feed your training data into the algorithm, which then solves for the parameter values that align the linear relationship most closely to the data. In Python, this procedure is implemented in `scikit-learn` as `sklearn.linear_model.LinearRegression`.

In the context of the Ames housing data, a straightforward cost function used is the Mean Squared Error (MSE), defined as

$$\text{MSE} = J = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \tag{2.2}$$

where m is the number of dataset instances, and J serves as a general representation of the cost function in machine learning contexts.

The MSE is also applied to minimize the error in a linear regression model, with the hypothesis h_θ, trained on dataset $\mathbf{X}$. This is expressed as:

$$\text{MSE}(\mathbf{X}, h_\theta) = J = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)})^2 \tag{2.3}$$

where the objective is to minimize the cost function by iteratively refining the values of $\boldsymbol{\theta}$.

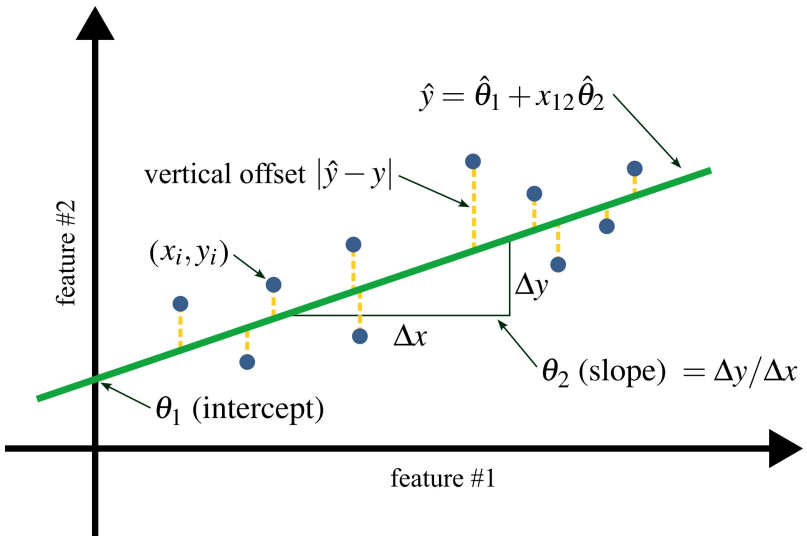


Figure 2.6: Linear regression of a two-feature dataset obtained with least squares.

Consider a dataset containing n observations, denoted as $(y_i, x_i)_{i=1}^{n}$. Each observation i consists of a scalar response y_i and a column vector x_i that holds the values for p predictors (regressors), represented as x_{ij} where $j = 1, \ldots, p$. In the context of a linear regression model, the response variable y_i is modeled as a linear combination of the regressors, influenced by an error term ε_i:

$$y_i = \theta_1 x_{i1} + \theta_2 x_{i2} + \ldots + \theta_y x_{im} + \varepsilon_i \tag{2.4}$$

This model can also be written in matrix notation as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\varepsilon} \tag{2.5}$$

Where:

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1m} \\ x_{21} & x_{22} & \ldots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nm} \end{bmatrix}, \boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2.6}$$

However, the best-fit model $\hat{y}$ will not be able to account for the unmodeled error, therefore:

$$\hat{y}_i = \hat{\theta}_1 x_{i1} + \hat{\theta}_2 x_{i2} + \ldots + \hat{\theta}_m x_{im} \tag{2.7}$$

or

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} \tag{2.8}$$

This goal is to find $\hat{\theta}$ such that the error is minimized. Mathmatically, this is simple enough as the $\hat{\theta}$ Is the minimization of the lease-squares hyperplane, or:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{2.9}$$

Equation 2.9 is referred to as the normal equation. Consider a simple linear model with $p = 2$, where $x_{11} = 1$. In this scenario, x_{11} acts as the bias term of the equation, whereas the remaining

elements of $\mathbf{X}$ represent the data. Without this bias term, the solution would only address the slope of the line. Subsequently, solving for $\hat{\theta}_1$ and $\hat{\theta}_2$ yields:

$$[x_{11}x_{12}][\hat{\theta}_1\hat{\theta}_2] = \hat{y} \tag{2.10}$$

As $x_{11} = 1$ (again, called the bias term) this becomes:

$$\hat{y} = \hat{\theta}_1 + x_{12}\hat{\theta}_2 \tag{2.11}$$

This is simply another expression for the equation of a liner line:

$$y = mx + b \tag{2.12}$$

Solving the model output for a input or a series of inputs can than be done simply enough.

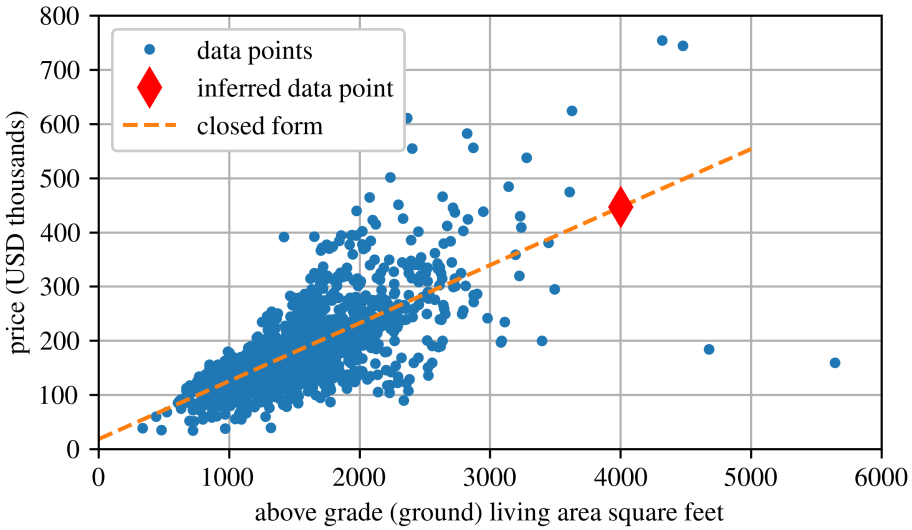


Figure 2.7: Ames housing data inferred.

2.1.2 Computational Complexity

The Normal Equation calculates the inverse of $\mathbf{X}^\top\mathbf{X}$, an $n \times n$ matrix, where n represents the number of features. The computational complexity of matrix inversion generally ranges from $O(n^{2.4})$ to $O(n^3)$, depending on the specific algorithm used. Consequently, doubling the number of features would increase the computational effort by approximately $2^{2.4} \approx 5.3$ to $2^3 = 8$ times.

> WARNING
>
> When the number of features becomes very large, for example around $100,000$, solving the Normal Equation becomes extremely slow.

On the positive side, the computational complexity of this equation is linear with respect to the number of instances in the training set, denoted as $O(m)$. This allows it to efficiently handle large training sets, as long as they fit within memory.

Furthermore, once your Linear Regression model is trained (whether through the Normal Equation or another method), making predictions is very quick. The computational effort is linear in relation to both the number of instances you wish to predict and the number of features. Essentially, predicting for twice as many instances or features will approximately double the computation time.

Next, we will explore different methods for training a Linear Regression model that are more appropriate for scenarios with a large number of features, or when the training data exceeds memory capacity.

Example 2.1 Linear Regression - Ames Housing Dataset
This example uses the Ames housing dataset to model the relationship between above-ground living area and house price using linear regression. It compares a manual fit, a closed-form solution, and gradient descent, alongside Scikit-Learn's `LinearRegression` and `SGDRegressor`, to illustrate how each method fits a line to the data.

2.2 Gradient Descent

Gradient Descent is a versatile optimization algorithm capable of finding optimal solutions across a broad spectrum of problems. The core concept of Gradient Descent involves iteratively adjusting parameters to minimize a cost function [a]. Consider the analogy of being lost in a dense fog in the mountains and seeking to descend:

1. You sense the ground's slope under your feet (the local gradient).
2. You descend in the direction of the steepest descent.
3. Reaching a point where the local gradient is zero indicates that you have found a minimum.

The Gradient Descent process starts with random initialization of the parameter θ, followed by gradual improvements. Each incremental step aims to reduce the cost function until the algorithm converges to a minimum.

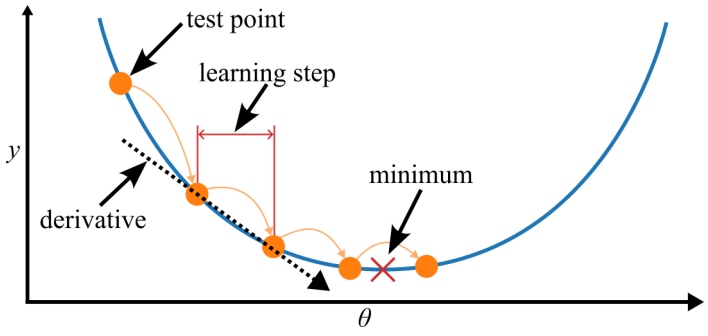


Figure 2.8: Illustration of gradient descent.

A critical aspect of Gradient Descent is the step size, controlled by the learning rate hyperparameter. A small learning rate leads to a slow convergence, requiring many iterations. Conversely,

[a] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

a high learning rate might cause the algorithm to overshoot the minimum, potentially causing divergence and failing to find an optimal solution.

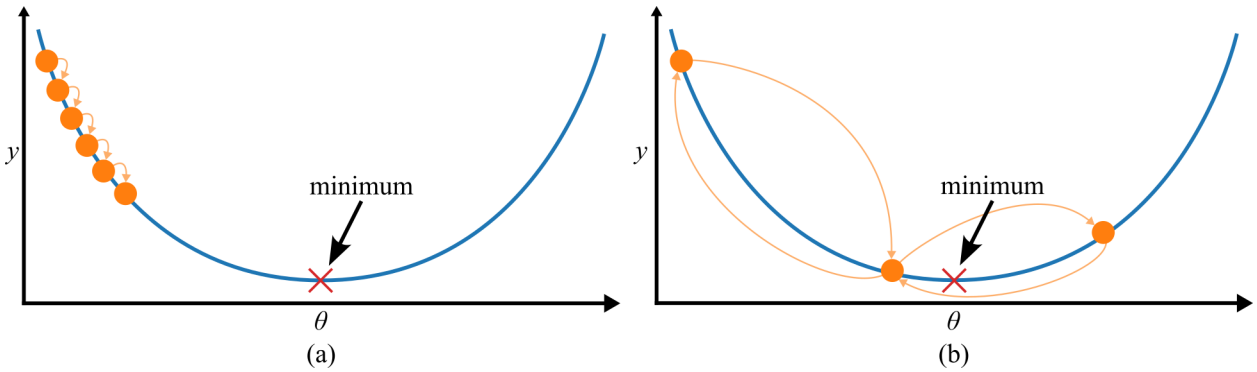


Figure 2.9: Effect of learning rate in gradient descent, showing: (a) a slow search with a low step size, and (b) a search with a large step size that oscillates around the minimum without ever finding the minimum.

Cost functions do not always present themselves as neat, concave shapes. They can feature obstacles such as holes, ridges, plateaus, and various complex topographies that complicate the convergence to the minimum. The challenges associated with Gradient Descent include:

- Starting the algorithm from a random point on the left may lead to convergence at a local minimum rather than the more optimal global minimum.
- Initiating on the right might result in a prolonged journey across a plateau, and terminating the process too soon could prevent reaching the global minimum.

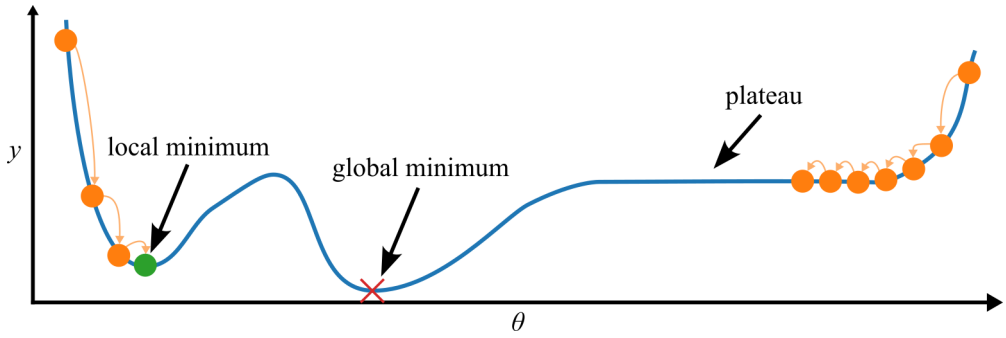


Figure 2.10: Gradient features and descent challenges.

Fortunately, the Mean Squared Error (MSE) cost function for Linear Regression models is convex. This characteristic ensures that for any two points on the curve, the line segment joining them does not intersect the curve itself. Consequently:

- There are no local minima, only one global minimum exists.
- It is a continuous function, and its slope changes smoothly.

These properties significantly benefit the optimization process: Gradient Descent can reliably approximate the global minimum given sufficient time and an appropriate learning rate.

2.2.1 Comparison of Gradient Descent Methods

Before looking into the mathematical details, let us examine how various gradient descent methods achieve the optimal solution, as illustrated in Figure 2.11. Additionally, Table 1 provides a comparative analysis of different gradient descent methods, highlighting their respective strengths and weaknesses.

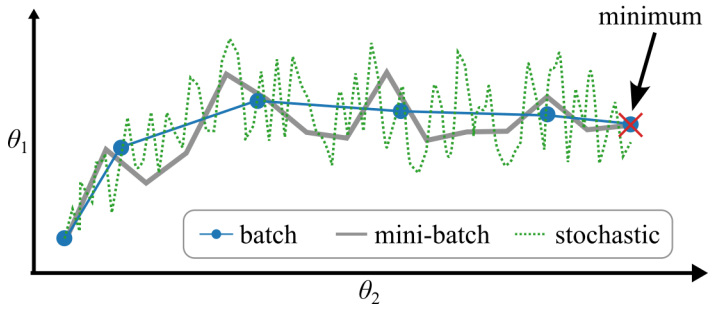


Figure 2.11: Comparing gradient descent methods.

Table 1: Comparison of various linear regression solutions.

algorithm	large number of instances (m)	data must fit in memory	large number of features (n)	hyperparams	scaling required	Scikit-Learn
normal equation	fast	yes	slow	0	no	`LinearRegression`
batch GD	slow	yes	fast	2	yes	n/a
stochastic GD	fast	no	fast	≥ 2	yes	`SGDRegressor`
mini-batch GD	fast	no	fast	≥ 2	yes	n/a

NOTE

After training, there is minimal difference between the models: these algorithms converge on similar solutions and make predictions in a nearly identical manner.

2.2.2 Batch Gradient Descent

Batch Gradient Descent (shown in figure 2.12) processes the entire batch of training data $\mathbf{X}$ at every step, which is the origin of its name, "batch." Consider again our Mean Squared Error (MSE) cost function:

$$J = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \tag{2.13}$$

For a hypothesis h_θ, the MSE for the dataset $\mathbf{X}$ can be computed for each instance $\mathbf{x}^{(i)}$ as follows:

$$J(\mathbf{X}, h_\theta) = J = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)})^2 \tag{2.14}$$

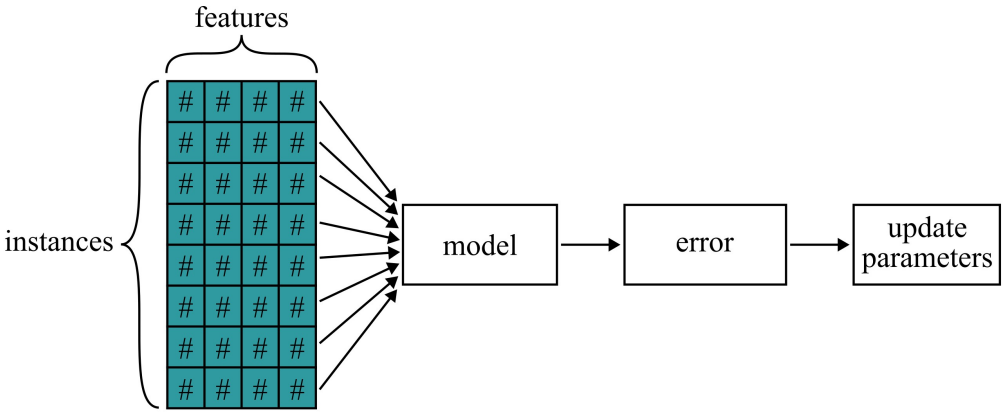


Figure 2.12: Flowchart of Batch Gradient Descent.

With this foundation, we can implement gradient descent by computing the gradient of the cost function with respect to each parameter θ_j. Specifically, this involves calculating how much the cost function changes when θ_j is altered slightly. This calculation is known as a partial derivative. Conceptually, it is akin to determining "the slope of the mountain under my feet if I face east," then repeating the question facing north, and similarly for any other direction in a higher-dimensional space. The partial derivative of the cost function with respect to parameter θ_j is computed as:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{2}{m} \sum_{i=1}^{m} (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)} \tag{2.15}$$

Rather than computing each partial derivative individually, all partial derivatives can be calculated simultaneously using the gradient vector, $\nabla_\theta J(\boldsymbol{\theta})$, which encompasses all partial derivatives of the cost function for each model parameter:

$$\nabla_\theta J(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} J(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\top \cdot (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) \tag{2.16}$$

WARNING

Be aware that this method computes using the entire training set $\mathbf{X}$ at every step of Gradient Descent, which is why the approach is named Batch Gradient Descent. Although this method can be exceedingly slow with large training sets, it performs well with numerous features, outpacing the Normal Equation in training Linear Regression models with extensive features.

Once you obtain the gradient vector, which indicates the direction of ascent, the next step involves moving in the opposite direction, effectively going downhill. This is achieved by subtracting $\nabla_\theta J(\theta)$ from $\boldsymbol{\theta}$, incorporating the learning rate η to scale the step size:

$$\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_\theta J(\boldsymbol{\theta}) \tag{2.17}$$

Interestingly, this process aligns perfectly with the results from the Normal Equation. However, variations in the learning rate η can significantly influence the outcomes. Figure 2.13 illustrates the first 10 steps of Gradient Descent with three different learning rates, with the dashed line marking the starting point.

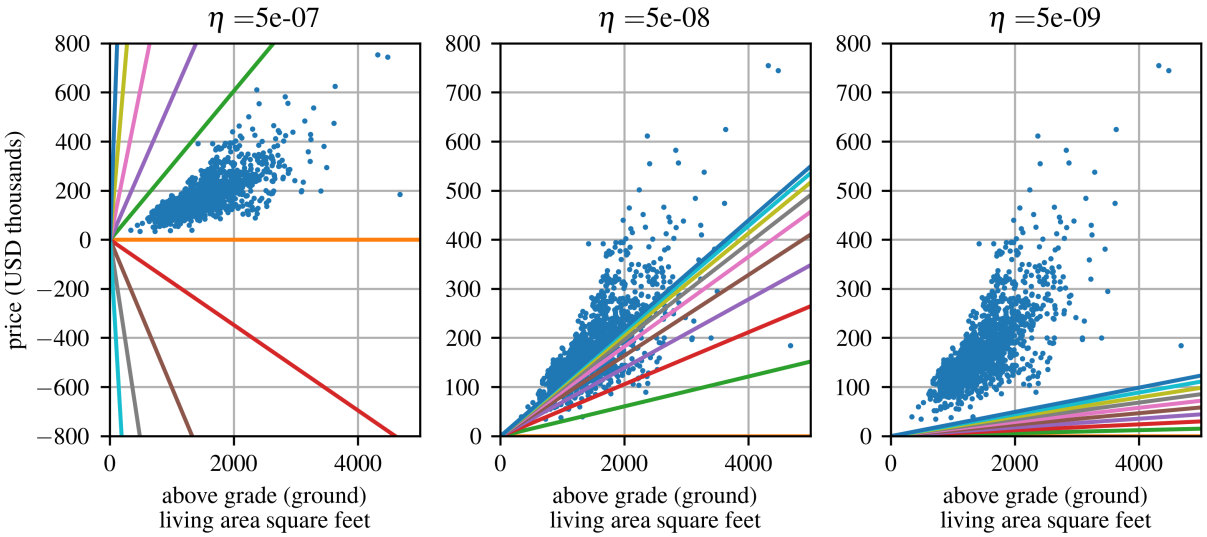


Figure 2.13: The effects of different learning rates on the gradient descent algorithms, showing: (a) too high ($\eta = 5e-07$); (b) about right ($\eta = 5e-08$), and; (c) too low ($\eta = 5e-09$).

Figure 2.13 highlights the impact of the learning rate on gradient descent. Figure 2.13(a) With a very large step size, $\eta = 5 \times 10^{-7}$, each update overshoots the minimum so the algorithm diverges rather than converging. Figure 2.13(b) Using $\eta = 5 \times 10^{-8}$ provides a balanced step size; the cost decreases just right [a] and the algorithm reaches the optimum in only a few iterations. Figure 2.13(c) A small step size, $\eta = 5 \times 10^{-9}$, keeps the algorithm moving toward the minimum yet progress is slow, requiring many more iterations to achieve the same result.

To identify an appropriate learning rate, a grid search can be employed. It is advisable to limit the number of iterations during grid search to avoid models that are too slow to converge.

Determining the correct number of iterations is also crucial. If set too low, the algorithm may stop far from the optimal solution. Conversely, overly high settings lead to unnecessary computations after convergence. A practical approach is to allow a large number of iterations but to halt the algorithm when the gradient vector's norm shrinks to below a small threshold ε (known as the tolerance), indicating proximity to the minimum.

[a] Pyle, K. "Goldilocks and the three bears." Mother's Nursery Tales (1918): 207-213.

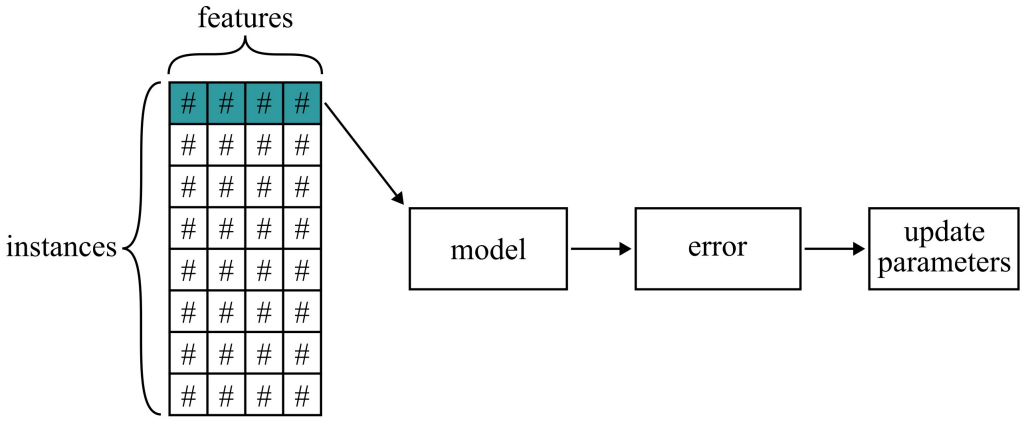


Figure 2.14: Flowchart of Stochastic Gradient Descent.

2.2.3 Stochastic Gradient Descent

Stochastic Gradient Descent (shown in figure 2.14) updates the model parameters after evaluating each individual training sample $x^{(i)}$ and its corresponding label $y^{(i)}$, which stands in contrast to batch gradient descent that uses the entire dataset for each update. The update rule for SGD can be expressed as

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)}). \tag{2.18}$$

While batch gradient descent tends to perform unnecessary recalculations for large datasets by reevaluating gradients for similar examples with each update, SGD eliminates this inefficiency by updating parameters incrementally. Consequently, SGD is generally faster and adaptable for online learning. Due to its frequent and individual updates, SGD exhibits high variance in the objective function, causing significant fluctuations as illustrated in Fig. 2.15(b).

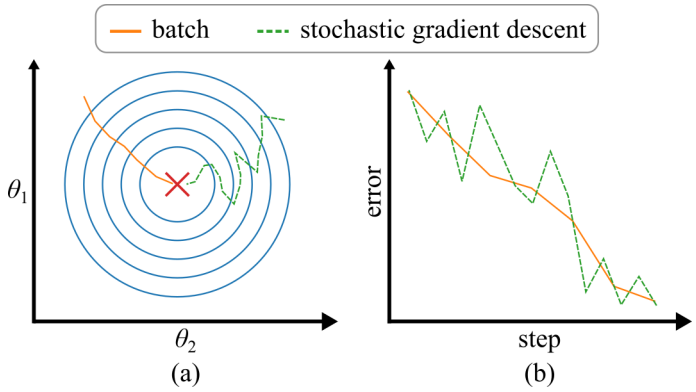


Figure 2.15: Comparison of the iterative performance of Batch and Stochastic Gradient Descent, showing: (a) how they move towards their target, and (b) the error at comparable steps.

2.2.4 Mini-batch Gradient Descent

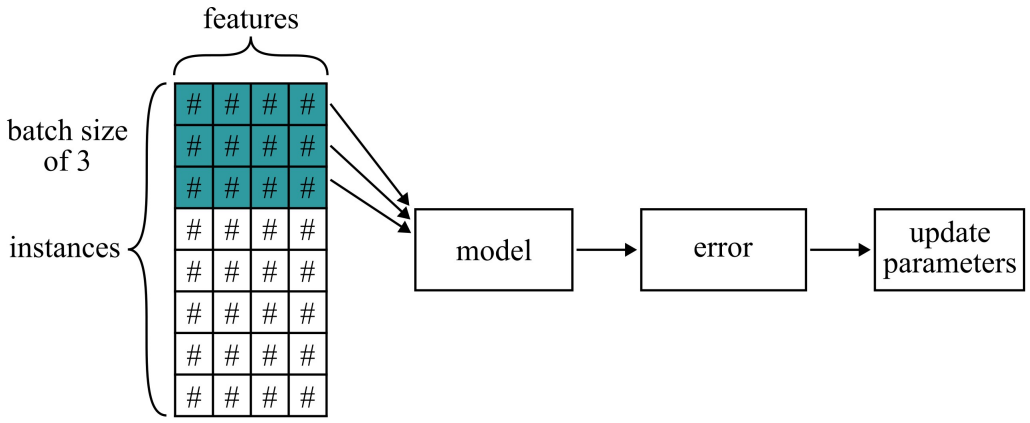


Figure 2.16: Flowchart of Mini-batch Gradient Descent.

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.19}$$

The final Gradient Descent algorithm we will discuss is Mini-batch Gradient Descent. This method combines elements of both Batch and Stochastic Gradient Descent: rather than calculating gradients using the entire training set (as in Batch GD) or a single instance (as in Stochastic GD), Mini-batch GD computes gradients using small, randomly selected subsets of instances known as minibatches. One significant benefit of Mini-batch GD over Stochastic GD is the improved performance from hardware-optimized matrix operations, particularly on GPUs.

Mini-batch GD's trajectory through parameter space tends to be more stable compared to the often erratic path of SGD, especially with larger minibatches. This stability typically brings Mini-batch GD closer to the minimum than SGD. However, Mini-batch GD might struggle more with escaping local minima in scenarios prone to such issues, which is less of a concern in Linear Regression. Figure 2.11 illustrates the trajectories of these three Gradient Descent techniques during training. While Batch GD precisely reaches the minimum, Stochastic GD and Mini-batch GD tend to oscillate nearby. It is important to note, however, that while Batch GD is slow in taking steps, both Stochastic GD and Mini-batch GD could also effectively reach the minimum with an appropriate learning rate schedule.

2.3 Feature Scaling

The cost function can resemble an elongated bowl if the feature scales vary significantly. Figure 2.17 illustrates the effect of feature scaling on Gradient Descent. In figure 2.17(a), Gradient Descent progresses directly towards the minimum, reaching it swiftly. However, in figure 2.17(b) it initially moves almost orthogonally to the direction of the global minimum, followed by a lengthy traverse down a nearly flat valley, significantly delaying the convergence.

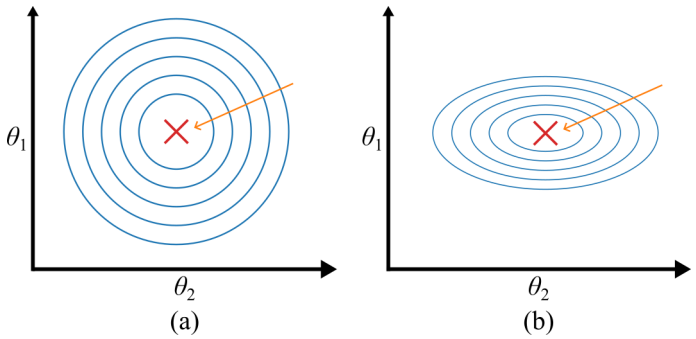


Figure 2.17: Gradient descent on a 2D surface for parameters that are: (a) equally scaled, and (b) unequally scaled.

Training a model can be viewed as a search through the parameter space for the combination that minimizes the cost function. When additional parameters are introduced, this space gains extra dimensions, making the optimization task more challenging. In Ordinary Least Squares Linear Regression, however, the cost surface is convex and bowl-shaped, so any descent method is guaranteed to reach the global minimum. To normalize feature scales, two common methods are employed. The are Min-max scaling and Standardization.

- **Min-max scaling**, often referred to as normalization, is straightforward: it rescales the data to a range of 0 to 1 by subtracting the minimum value and dividing by the range (max minus min).

$$\mathbf{X}' = \frac{\mathbf{X} - \mathbf{X}_{\text{min}}}{\mathbf{X}_{\text{max}} - \mathbf{X}_{\text{min}}} \tag{2.20}$$

  Scikit-Learn offers `sklearn.preprocessing.MinMaxScaler` for this purpose.

- **Standardization** differs significantly as it first subtracts the mean (resulting in a zero mean) and then divides by the standard deviation to achieve unit variance. This method does not limit values to a specific range, which can be problematic for certain algorithms, such as neural networks which often expect inputs between 0 and 1. However, standardization is less sensitive to outliers. For instance, if a median income is mistakenly recorded as 100, min-max scaling would compress all other values between 0 and 15 to between 0 and 0.15, whereas standardization would be minimally affected.

$$\mathbf{X}' = \frac{\mathbf{X} - \overline{\mathbf{X}}}{\sigma} \tag{2.21}$$

  `sklearn.preprocessing.StandardScaler` is provided by Scikit-Learn for standardization.

WARNING

It is crucial to apply scalers exclusively to the training data and not to the entire dataset, which includes the test set. This practice ensures that the model is not inadvertently exposed to test data during training. Once the scalers are fitted to the training data, they can then be used to transform the training set, the test set, and any new data subsequently encountered.

2.4 Polynomial Regression

What if the underlying pattern of your data is more complex than a simple linear relationship? Consider the dataset shown in figure 2.18 with non-linear characteristics (one feature across multiple instances), which typically can be modeled using a second-order polynomial

$$\hat{y} = ax^2 + bx + c. \tag{2.22}$$

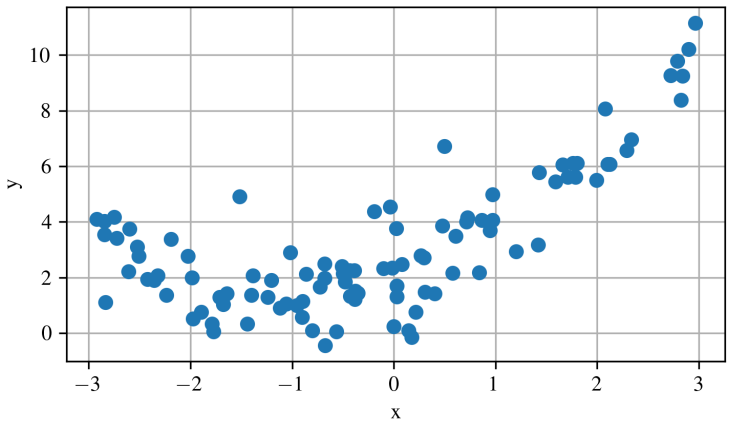


Figure 2.18: Dataset derived from a 2nd-order polynomial.

A simple linear fit will under-perform, so you first enrich the training set by adding the squared term of each feature as an extra column; shown in figure 2.19. Although the model you train is still linear with respect to its parameters, it now operates on an augmented feature space and can represent the desired quadratic curve. This approach is called Polynomial Regression, and it extends naturally by including higher powers whenever more complex nonlinearities are present.

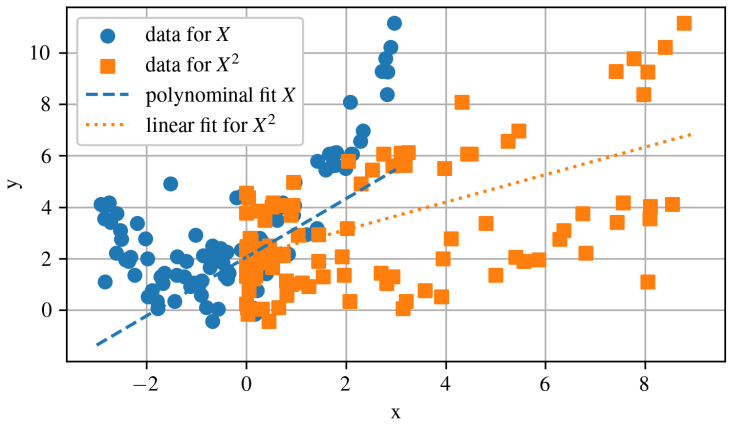


Figure 2.19: Linear models fit to the individual features of a polynomial dataset.

This data can then be incorporated into two linear models, where the slopes of the feature sets serve as the parameters for the base-line polynomial expression (a and b in Equation 2.22) and the bias term is the offset (c in Equation 2.22). The results of such a polynomial fit is shown in figure 2.20.

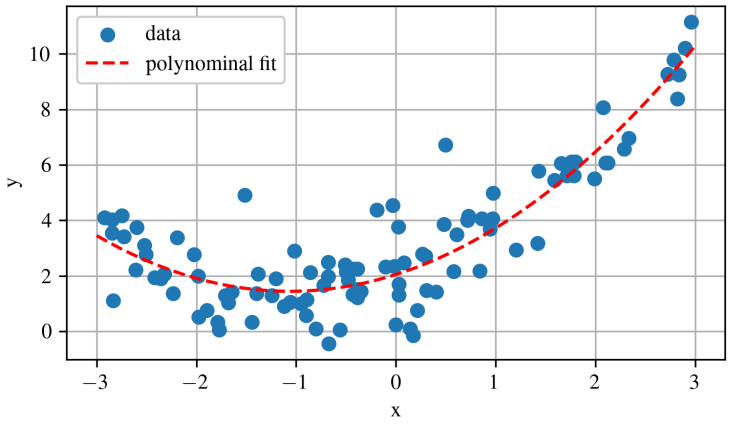


Figure 2.20: Polynomial model fit to a dataset.

It is important to note that Polynomial Regression can identify interrelationships between features in cases where multiple features exist, a capability beyond the scope of simple Linear Regression. This enhancement is enabled by `PolynomialFeatures`, which includes all possible combinations of features up to the specified degree. For instance, with two features a and b, and a degree of 3, `PolynomialFeatures` would add not only a^2, a^3, b^2, and b^3 but also the combined terms ab, a^2b, and ab^2.

WARNING

`PolynomialFeatures(degree=d)` expands an array with n original features into one that includes $\frac{(n+d)!}{d!n!}$ features, accounting for all combinations of features up to the d-th degree. Here, $n!$ represents the factorial of n, calculated as $1 \times 2 \times 3 \times \cdots \times n$. Be cautious of the rapid increase in the number of features, known as combinatorial explosion!

Example 2.2 Polynomial Regression

This example fits a non-linear dataset using polynomial regression by adding x^2 as an extra feature. It demonstrates how linear models can approximate curved relationships when polynomial terms are included, and shows the resulting fit compared to the original data.

2.5 Examples

Example 2.1

```
"""
Example 2.1 Linear Regression
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn.linear_model import LinearRegression

plt.close('all')

#%% load data

ames = sk.datasets.fetch_openml(name="house_prices", as_frame=True,parser='auto')
target = ames['target'].values
data = ames['data']
YrSold = data['YrSold'].values  # Year Sold (YYYY)
MoSold = data['MoSold'].values  # Month Sold (MM)
OverallCond = data['OverallCond'].values # OverallCond: Rates the overall condition of the house
GrLivArea = data['GrLivArea'].values # Above grade (ground) living area square feet
BedroomAbvGr = data['BedroomAbvGr'].values # Bedrooms above grade (does NOT include basement bedrooms)

# Ask a home buyer to describe their dream house, and they probably won't begin
# with the height of the basement ceiling or the proximity to an east-west railroad.
# But this playground competition's dataset proves that much more influences price
# negotiations than the number of bedrooms or a white-picket fence.

# With 79 explanatory variables describing (almost) every aspect of residential
# homes in Ames, Iowa, this competition challenges you to predict the final price
# of each home.

# Plot a few of the interesting features vs the target (price). In particular,
# let's plot the number of rooms vs. the price.
plt.figure()
plt.plot(GrLivArea,target,'o',markersize=2)
plt.xlabel('Above grade (ground) living area square feet')
plt.ylabel('price (USD)')
plt.grid(True)
plt.tight_layout()

#%% Build a model for the data
X = GrLivArea
Y = target
model_X = np.linspace(0,5000)

theta_1 = 0
theta_2 = 100
model_Y_manual = theta_1 + theta_2*model_X

plt.figure()
plt.plot(X,Y,'o',markersize=2,label='data')
plt.plot(model_X,model_Y_manual,'--',label='manual fit')
plt.xlabel('Above grade (ground) living area square feet')
plt.ylabel('price (USD)')
plt.grid(True)
#plt.xlim([3.5,9])
#plt.ylim([0,50000])
plt.legend(framealpha=1)
plt.tight_layout()

```

```
# add a dimension to the data as math is easier in 2d arrays and sk learn only
# takes 2d arrays
X = np.expand_dims(X,axis=1)
Y = np.expand_dims(Y,axis=1)
model_X = np.expand_dims(model_X,axis=1)

#%% compute the linear regression solution using the closed form solution

# compute
X_b = np.ones((X.shape[0],2))
X_b[:,1] = X.T # add x0 = 1 to each instance

theta_closed_form = np.linalg.inv(X_b.T@X_b)@X_b.T@Y

model_y_closed_form = theta_closed_form[0] + theta_closed_form[1]*3000
model_Y_closed_form = theta_closed_form[0] + theta_closed_form[1]*model_X

plt.figure()
plt.plot(X,Y,'o',markersize=3,label='data points')
plt.xlabel('Above grade (ground) living area square feet')
plt.ylabel('price (USD)')
plt.plot(3000,model_y_closed_form,'dr',markersize=10,zorder=10,
         label='inferred data point')
plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
plt.grid(True)
plt.legend()
plt.tight_layout()

#%% compute the linear regression solution using gradient descent

eta = 0.00000001 # learning rate
n_iterations = 100
m = X.shape[0]
theta_gradient_descent = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta_gradient_descent) - Y)
    theta_gradient_descent = theta_gradient_descent - eta * gradients

print(theta_gradient_descent)

model_Y_gradient_descent =  theta_gradient_descent[0] \
    + theta_gradient_descent[1]*model_X

plt.figure()
plt.plot(X,Y,'o',markersize=3,label='data points')
plt.xlabel('Above grade (ground) living area square feet')
plt.ylabel('price (USD)')
plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
plt.plot(model_X,model_Y_gradient_descent,':',label='gradient descent')
plt.grid(True)
plt.legend()
plt.tight_layout()

#%% compute the linear regression solution using sk-learn

# build and train a closed from linear regression model in sk-learn
model_LR = sk.linear_model.LinearRegression()
model_LR.fit(X,Y[:,0])
model_Y_sk_LR = model_LR.predict(model_X)

# build and train a Stochastic Gradient Descent linear regression model in sk-learn.
# Note that in running the model, the best way to do this would be to use a pipeline
# =with feature scaling. However, here we just set 'eta0' to a low value, this
# is done only for educational # purposes and is not the ideal methodology in
# terms of system robustness.
model_SGD = sk.linear_model.SGDRegressor(learning_rate='constant',eta0=0.00000001)
model_SGD.fit(X,Y[:,0])
```

```
133  model_Y_sk_SGD = model_SGD.predict(model_X)
134
135  # plot the modeled results
136  plt.figure()
137  plt.plot(X,Y,'o',markersize=2,label='data')
138  plt.plot(model_X,model_Y_closed_form,'-',label='normal equation')
139  plt.plot(model_X,model_Y_gradient_descent,'--',label='gradient descent')
140  plt.plot(model_X,model_Y_sk_LR,':',label='sklearn normal equation')
141  plt.plot(model_X,model_Y_sk_SGD,'-.',label='sklearn stochastic gradient descent')
142
143  plt.xlabel('Above grade (ground) living area square feet')
144  plt.ylabel('price (USD)')
145  plt.grid(True)
146  plt.legend(framealpha=1)
147  plt.tight_layout()
```

Example 2.2

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 2.2
Polynomial regression
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model

plt.close('all')

#%% build the data sets
np.random.seed(2) # 2 and 6 are pretty good
m = 100
X = 6 * np.random.rand(m,1) - 3
Y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)

# plot the data
plt.figure()
plt.grid(True)
plt.scatter(X,Y)
plt.xlabel('x')
plt.ylabel('y')

#%% perform  polynomial regression

# generate x^2 as we use the model y = a*x^2* + b*x + c
X_poly_manual = np.hstack((X,X**2))

# or use the code as this does lots of features for multi-feature data sets.
poly_features = sk.preprocessing.PolynomialFeatures(degree=2, include_bias=False)
X_poly_sk = poly_features.fit_transform(X)

# they do do same thing as shown below, so select one to carry forward.
print(X_poly_manual == X_poly_sk)
X_poly = X_poly_manual

# In essence, we now have two data sets. We can plot that here
plt.figure()
plt.grid(True)
plt.scatter(X_poly[:,0],Y,label = 'data for x')
plt.scatter(X_poly[:,1],Y,marker='s',label = 'data for $x^2$')
plt.legend()
plt.xlabel('x')
plt.ylabel('y')

# and fit linear models to these data sets
model = sk.linear_model.LinearRegression() # Select a linear model
model.fit(X_poly,Y) # Train the model
X_model_1 = np.linspace(-3,3)
X_model_2 = np.linspace(0,9)

# the model parameters are:
model_coefficients = model.coef_
model_intercept = model.intercept_
print(model_coefficients)
print(model_intercept)
```

```
68
69  Y_X1 = model_coefficients[0][0]*X_model_1 + model_intercept
70  Y_X2 = model_coefficients[0][1]*X_model_2 + model_intercept
71
72  # now if we plot the linear models on the extended set of features.
73  plt.figure()
74  plt.grid(True)
75  plt.scatter(X_poly[:,0],Y,label = 'data for x')
76  plt.scatter(X_poly[:,1],Y,marker='s',label = 'data for $x^2$')
77  plt.plot(X_model_1,Y_X1,'--',label='linear fit $x$')
78  plt.plot(X_model_2,Y_X2,':',label='linear fit for $x^2$',)
79  plt.legend()
80  plt.xlabel('x')
81  plt.ylabel('y')
82  plt.savefig('example_6_fig_1',dpi=300)
83
84  # now that we have a parameter for x and x^2, these can be recombined into a single
85  # model, y = x^2*a + x*b + c.
86  Y_polynominal = model_coefficients[0][1]*X_model_1**2 + model_coefficients[0][0]*\
87      X_model_1 + model_intercept
88
89  plt.figure()
90  plt.grid(True)
91  plt.scatter(X,Y,label='data')
92  plt.plot(X_model_1,Y_polynominal,'r--',label='polynominal fit')
93  plt.xlabel('x')
94  plt.ylabel('y')
95  plt.legend()
96  plt.savefig('example_6_fig_2',dpi=300)
97
98
99
100
101
```

3 Machine Learning Workflows

Machine learning workflows describe the sequence of steps used to transform raw engineering data into a trained and evaluated predictive model. Although the learning algorithm is important, successful machine learning applications often depend just as much on the quality of the data, the choice of features, and the way the model is evaluated. A well-organized workflow helps ensure that the model learns meaningful patterns rather than noise or accidental structure in the data.

Figure 3.1 illustrates a common machine learning workflow. The process begins by collecting data from experiments, sensors, simulations, or existing records. The data are then inspected and cleaned to identify missing values, outliers, inconsistent units, or other quality issues. After this preparation step, feature engineering is used to convert the raw data into informative inputs for the model. The model is then trained using a portion of the available data and tested using data that were not used during training.

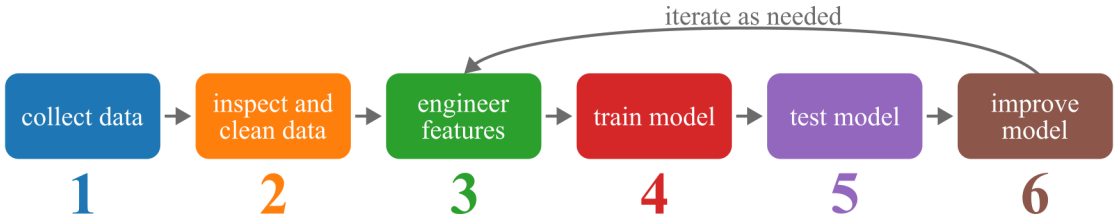


Figure 3.1: A typical machine learning workflow for engineering applications.

The final step is improvement. If the model does not perform well enough, the workflow is repeated by revisiting earlier decisions. This may involve collecting more data, improving the data-cleaning process, changing the engineered features, tuning model parameters, or selecting a different learning algorithm. For this reason, machine learning should be viewed as an iterative process rather than a single pass through the data.

A typical workflow can be summarized as follows:

1. **Collect data:** Gather measurements, simulations, experimental observations, or operational records.
2. **Inspect and clean data:** Check the data for missing values, outliers, incorrect units, noise, and obvious errors.
3. **Engineer features:** Transform the raw data into informative inputs that capture important patterns or physical behavior.
4. **Train model:** Fit a machine learning model using the training data.
5. **Test model:** Evaluate the trained model using data that were not used during training.
6. **Improve model:** Refine the data, features, model settings, or learning algorithm based on the test results.

3.1 Feature Engineering

Feature engineering is the process of transforming raw data into informative inputs that can be used by machine learning algorithms. Carefully designed features can significantly improve model performance and help capture important physical relationships in engineering data.

Figure 3.2 illustrates how raw engineering data can be transformed into a feature vector for machine learning. The features shown in this section are not the only features that can be extracted from data. Other useful inputs may include categorical features, countable features, indicator variables, domain-specific quantities, or features created from simulations and experiments. However, statistical, time-series, and frequency-domain features provide a useful starting point for building informative features from many engineering datasets.

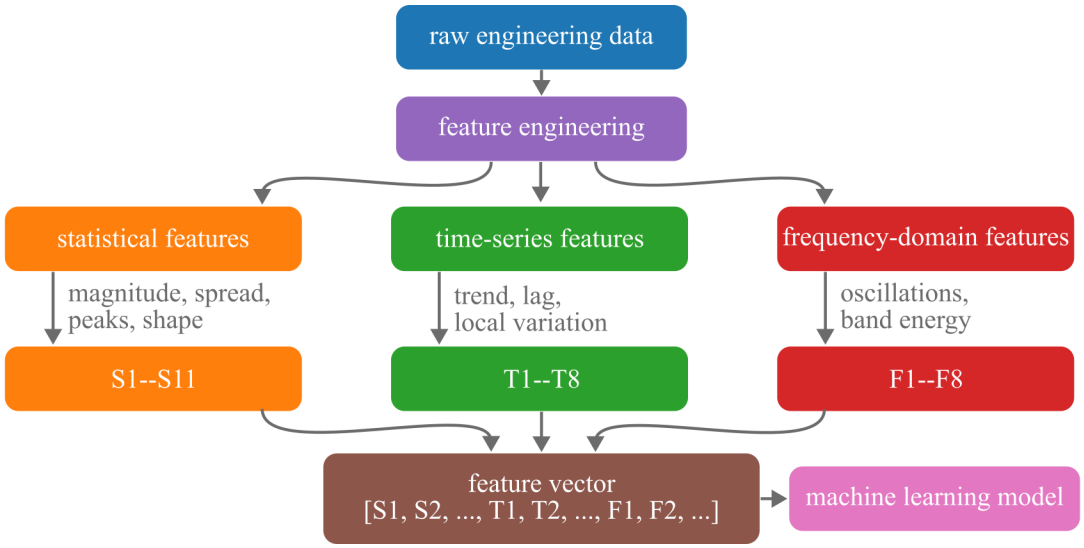


Figure 3.2: Feature engineering transforms raw engineering data into informative model inputs. Features may summarize the statistical properties, time-dependent behavior, or frequency content of the data.

3.1.1 Statistical Features

Statistical features summarize the distribution of values within a dataset, observation, or time window. These features are useful for describing the overall magnitude, variability, symmetry, and peak behavior of the data. Although statistical features are often computed from time-series measurements, they do not necessarily depend on the order of the samples in time.

Let x_i represent the ith value in a set of N measurements, let $\bar{x}$ represent the mean value, and let s represent the sample standard deviation. Table 1 lists a number of statistical features relevant in engineering.

Table 2: Common statistical features for engineering data.

No.	Feature	Equation	Physical Interpretation
S1	Maximum value	$\max(x_i)$	The maximum value captures the largest response in the signal.
S2	Minimum value	$\min(x_i)$	The minimum value captures the smallest response in the signal.
S3	Mean value	$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$	The mean value describes the average level of the signal.
S4	Absolute mean value	$\frac{1}{N}\sum_{i=1}^{N} \lvert x_i \rvert$	The absolute mean value describes the average magnitude of the signal.
S5	Root mean square value	$x_{\text{rms}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$	The root mean square value describes the effective signal magnitude.
S6	Standard deviation	$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N} (x_i - \bar{x})^2}$	The standard deviation describes how much the signal varies.
S7	Skewness	$\frac{1}{N}\sum_{i=1}^{N} \left(\frac{x_i - \bar{x}}{s}\right)^3$	Skewness describes whether the signal is biased toward one side.
S8	Kurtosis	$\frac{1}{N}\sum_{i=1}^{N} \left(\frac{x_i - \bar{x}}{s}\right)^4$	Kurtosis describes whether the signal contains sharp peaks or outliers.
S9	Shape factor	$\frac{x_{\text{rms}}}{\frac{1}{N}\sum_{i=1}^{N} \lvert x_i \rvert}$	The shape factor compares the effective magnitude to the average magnitude.
S10	Crest factor	$\frac{\max(\lvert x_i \rvert)}{x_{\text{rms}}}$	The crest factor compares the largest peak to the effective signal level.
S11	Impulse factor	$\frac{\max(\lvert x_i \rvert)}{\frac{1}{N}\sum_{i=1}^{N} \lvert x_i \rvert}$	The impulse factor compares the largest peak to the average magnitude.

3.1.2 Time-Series Features

Time-series features describe how a signal changes with time. Unlike statistical features, these features depend on the order of the samples. They are useful when the timing, trend, persistence, or local variation of the signal contains important information.

Table 2 lists a number of time-series features relevant in engineering. Let x_i represent the signal value at time index i, let Δt represent the sampling interval, let k represent a lag, and let w represent the number of samples in a moving window. The local mean over a window is denoted as $\bar{x}_{i,w}$.

Table 3: Common time-series features for engineering data.

No.	Feature	Equation	Physical Interpretation
T1	Lagged value	x_{i-k}	A lagged value captures the recent history of the signal.
T2	First difference	$x_i - x_{i-1}$	The first difference captures the step-to-step change in the signal.
T3	Rate of change	$\frac{x_i - x_{i-1}}{\Delta t}$	The rate of change describes how quickly the signal changes with time.
T4	Moving average	$\frac{1}{w}\sum_{j=i-w+1}^{i} x_j$	The moving average captures the local trend of the signal.
T5	Rolling standard deviation	$\sqrt{\frac{1}{w-1}\sum_{j=i-w+1}^{i}(x_j - \bar{x}_{i,w})^2}$	The rolling standard deviation captures local variability.
T6	Rolling maximum	$\max(x_{i-w+1}, \ldots, x_i)$	The rolling maximum captures recent peak behavior.
T7	Rolling minimum	$\min(x_{i-w+1}, \ldots, x_i)$	The rolling minimum captures recent low-response behavior.
T8	Zero-crossing rate	$\frac{1}{w-1}\sum_{j=i-w+2}^{i} \mathbb{I}(x_j x_{j-1} < 0)$	The zero-crossing rate captures how often the signal changes sign [a].

3.1.3 Frequency-Domain Features

Frequency-domain features describe how the content of a signal is distributed across frequency. These features are commonly extracted by applying a Fourier transform to a measured signal and then summarizing the resulting spectrum. Table 3 lists a number of frequency-domain features relevant in engineering. They are useful when the important behavior in the data is related to oscillations, repeating patterns, or changes in the balance between slow and rapid variations.

Let $P(f)$ represent the power spectrum of the signal at frequency f, and let B_L, B_M, and B_H represent the low-, middle-, and high-frequency bands. The total spectral energy is defined as

$$E_{\text{total}} = \int P(f)\, df. \tag{3.1}$$

In a specific engineering application, the frequency bands should be selected based on the system being studied. When no application-specific bands are known, the usable frequency range can be divided into low-, middle-, and high-frequency regions as a starting point.

[a] $\mathbb{I}(\cdot)$ is an indicator function that equals 1 when the condition is true and 0 otherwise.

Table 4: Common frequency-domain features for engineering time-series data.

No.	Feature	Equation	Physical Interpretation
F1	Dominant frequency	$f_{\text{dom}} = \arg\max_f P(f)$	The dominant frequency captures the strongest periodic behavior.
F2	Spectral centroid	$f_c = \frac{\int f P(f)\, df}{\int P(f)\, df}$	The spectral centroid describes where the spectrum is centered.
F3	Spectral bandwidth	$\sqrt{\frac{\int (f - f_c)^2 P(f)\, df}{\int P(f)\, df}}$	The spectral bandwidth describes how widely the energy is spread.
F4	Low-frequency energy ratio	$\frac{\int_{B_{\text{L}}} P(f)\, df}{E_{\text{total}}}$	The low-frequency energy ratio measures the share of energy in slow behavior.
F5	Middle-frequency energy ratio	$\frac{\int_{B_{\text{M}}} P(f)\, df}{E_{\text{total}}}$	The middle-frequency energy ratio measures the share of energy in mid-range behavior.
F6	High-frequency energy ratio	$\frac{\int_{B_{\text{H}}} P(f)\, df}{E_{\text{total}}}$	The high-frequency energy ratio measures the share of energy in rapid behavior.
F7	High-to-low frequency energy ratio	$\frac{\int_{B_{\text{H}}} P(f)\, df}{\int_{B_{\text{L}}} P(f)\, df}$	The high-to-low frequency energy ratio compares rapid behavior to slow behavior.
F8	Peak-to-total spectral energy ratio	$\frac{P(f_{\text{dom}})}{E_{\text{total}}}$	The peak-to-total spectral energy ratio measures how dominant the largest frequency peak is.

3.2 Training and Testing Data

Training a predictive model on a dataset and then testing it with the same data is a fundamental error in methodology. Such a model could simply memorize the labels of the training samples, achieving perfect performance during training but failing to make any useful predictions on new, unseen data. This phenomenon is known as overfitting. To prevent this, it is standard practice in supervised machine learning to reserve a portion of the available data as a test set, denoted as $\mathbf{X}_{\text{test}}$ and $\mathbf{y}_{\text{test}}$. The function `sklearn.model_selection.train_test_split` in scikit-learn randomly divides arrays or matrices into training and testing subsets.

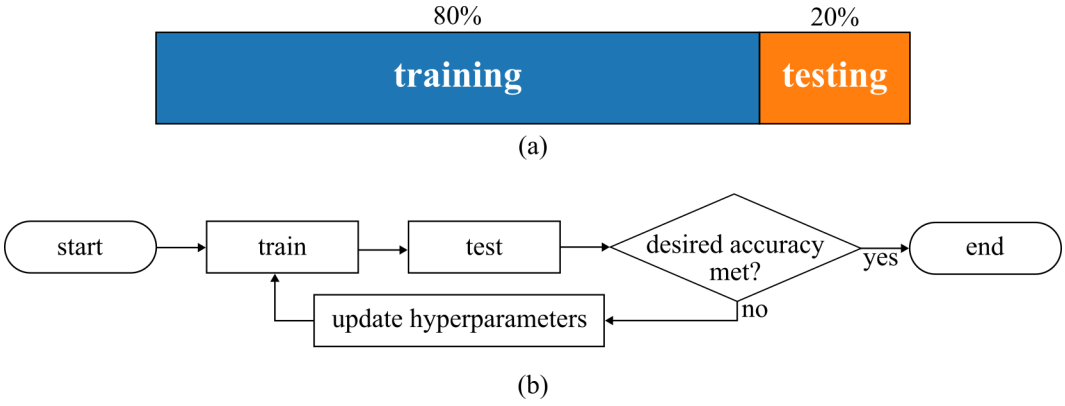


Figure 3.3: Splitting data up into training and testing subsets.

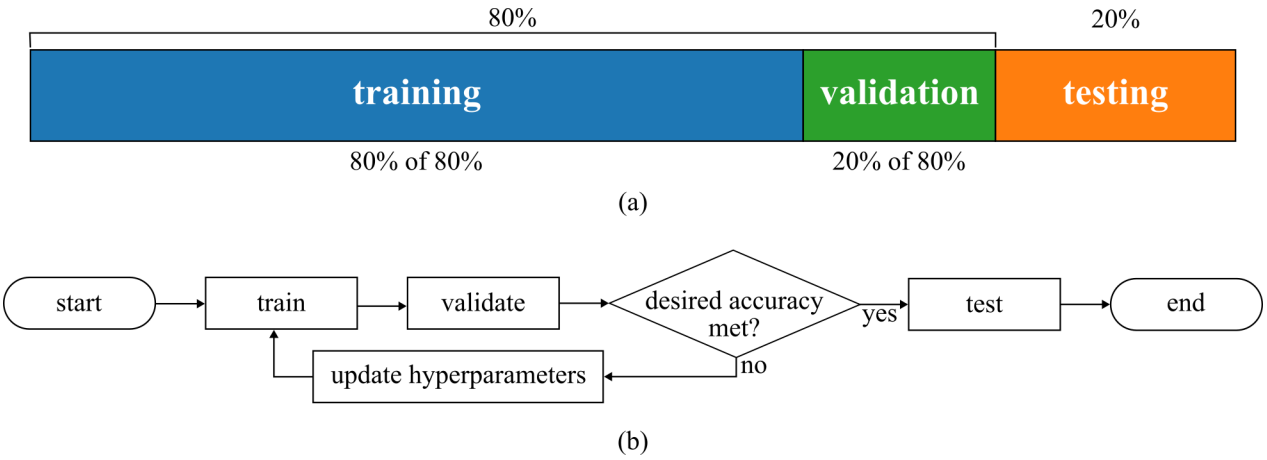


Figure 3.4: Splitting data up into training, validation, and testing subsets.

3.3 Pipelines

Pipelines in scikit-learn streamline the process by sequentially applying a list of transformations followed by a final estimator to a dataset. Employing pipelines allows for the integration of multiple processing steps, which can then be cross-validated together while experimenting with various parameters. Fig. 3.5 illustrates a typical pipeline configuration in scikit-learn.

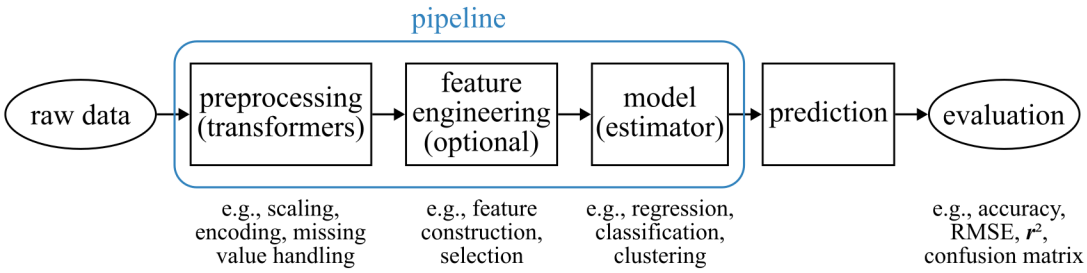


Figure 3.5: Pipeline setup for the automated deployment of pre-processing and modeling steps.

3.4 Learning Curves

Performing Polynomial Regression with a high degree typically allows for a closer fit to the training data compared to simple Linear Regression. For instance, Figure 3.6 demonstrates the application of a 30-degree polynomial model to a set of training data, contrasting it with both a linear model and a quadratic model (2^{nd}-degree polynomial). Observe how the 30-degree polynomial model contorts to closely match the training instances.

- The linear model exhibits underfitting.
- The high-degree Polynomial Regression model drastically overfits the training data.
- Among these, the quadratic model is likely to generalize best.

In Figure 3.6, the appropriateness of the quadratic model is clear since the data was initially generated using such a model. However, in real-world scenarios where the underlying function of the data is unknown, determining the optimal complexity for your model can be challenging. How can you ascertain whether your model is overfitting or underfitting the data?

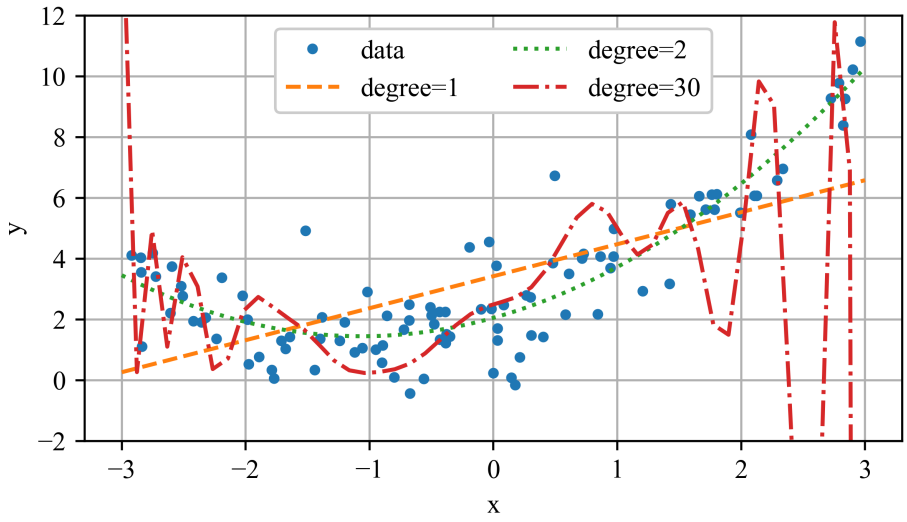


Figure 3.6: Polynomial regression showing underfitting (degree=1), a respectable model fit (degree=2), and overfitting (degree=30).

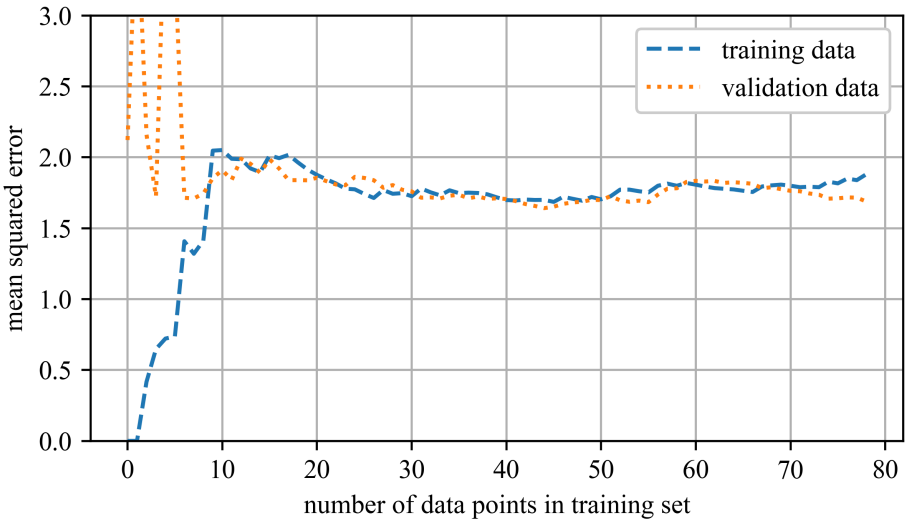


Figure 3.7: Learning curves for the underfitting linear model (degree=1) where underfitting is evident because both curves have plateaued; they are close together and relatively high.

Examining the model's behavior in figures 3.7 and 3.8 on the training set shows a clear trend. With only one or two samples the fit is perfect, so the error curve begins at zero. As additional observations are introduced the model can no longer capture every point exactly, partly because of measurement noise and partly because the underlying relationship is nonlinear, and consequently the training error rises. After a sufficient number of samples the curve flattens out; beyond this plateau adding further data neither markedly lowers nor raises the average error.

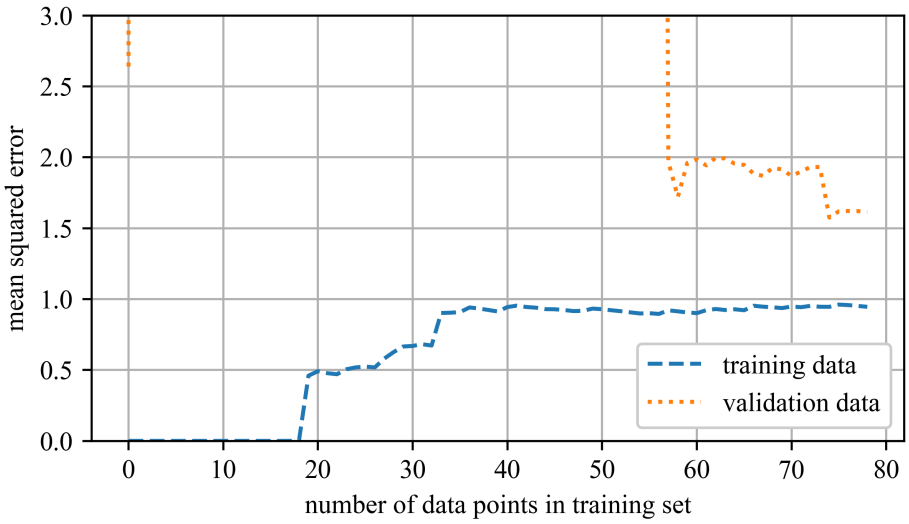


Figure 3.8: Learning curves for the overfitting polynomial model (degree=20) showing a significant gap between the curves, which indicates better performance on the training data than on the validation data.

Inspection of the validation curve in figure 3.8 tells a complementary story. With only a handful of training samples the model cannot generalize, so the validation error starts high. As more examples become available it learns progressively better patterns and the error falls. Eventually, though, the simple linear hypothesis is unable to capture the data's full complexity, causing the decline to flatten out; the validation curve plateaus at nearly the same level as the training curve. Their close proximity and uniformly large values are characteristic of an underfitting model.

Figure 3.8 displays the learning curves for a model fitted with a 20^{th}-degree polynomial. The overall shape resembles the curves seen earlier, yet two key differences stand out. First, the error on the training set is far lower than that achieved by the Linear Regression model, demonstrating the polynomial's extra flexibility. Second, a pronounced gap separates the training and validation curves, which means the model performs much better on the data it has already seen; this divergence is the hallmark of overfitting. Expanding the training set could reduce the gap, but without additional regularization the risk of overfitting would remain.

Next, let's apply the code using a 2^{nd} order polynomial, which accurately captures the essence of the data without underfitting or overfitting. These results are shown in Figure 3.9. Here it can be seen that the training and validation curves again meet after converging, showing a well-fit model.

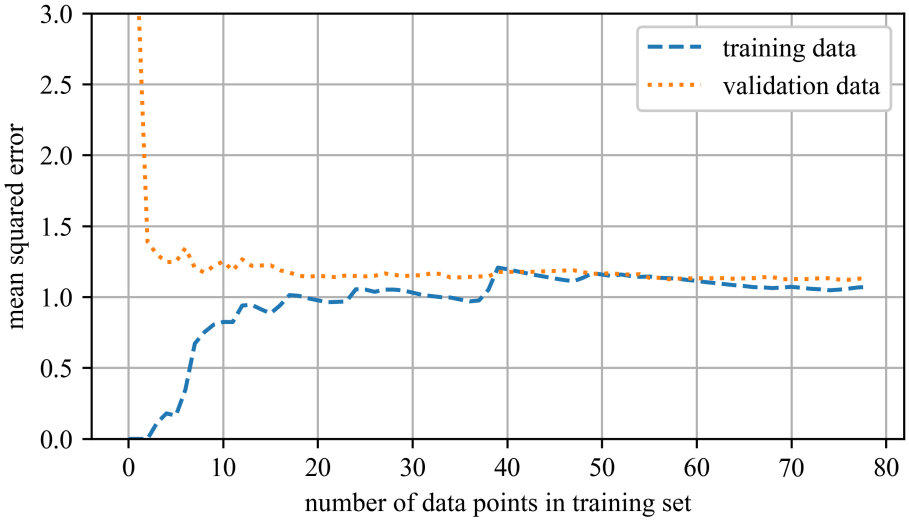


Figure 3.9: Learning curves for the optimal polynomial model (degree=2) showing well-aligned curves that plateau at a relatively low error level.

Example 3.1 Learning Curves
This example compares learning curves for linear and polynomial regression models. By plotting training and validation errors as the training set grows, it illustrates how model complexity impacts bias and variance, and helps identify underfitting and overfitting behavior.

3.5 Regularized Linear Models

Overfitting is frequently curbed by regularizing the model, meaning you introduce constraints that reduce its effective degrees of freedom and make it harder to memories noise. In a polynomial setting you can achieve this simply by choosing a lower-degree polynomial, while in a linear model you typically add a penalty that discourages large coefficient values. This naturally raises the question: how is regularization expressed mathematically?

3.5.1 Ridge Regression

Ridge Regression modifies Linear Regression by adding a regularization term to the cost function, which compels the learning algorithm to fit the data while keeping the model weights as small as possible. The updated cost-function is

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2. \tag{3.2}$$

The hyperparameter α dictates the degree of regularization. If $\alpha = 0$, Ridge Regression reduces to plain Linear Regression. If α is very high, the model weights are significantly shrunk, resulting in a nearly flat line through the mean of the data. Equation 3.2 outlines the cost function for Ridge Regression. The bias term θ_0 is not regularized, as indicated by the summation starting from $i = 1$.

NOTE

The regularization term is only included during the training phase. After training, the model's performance should be evaluated based on the unregularized performance measure.

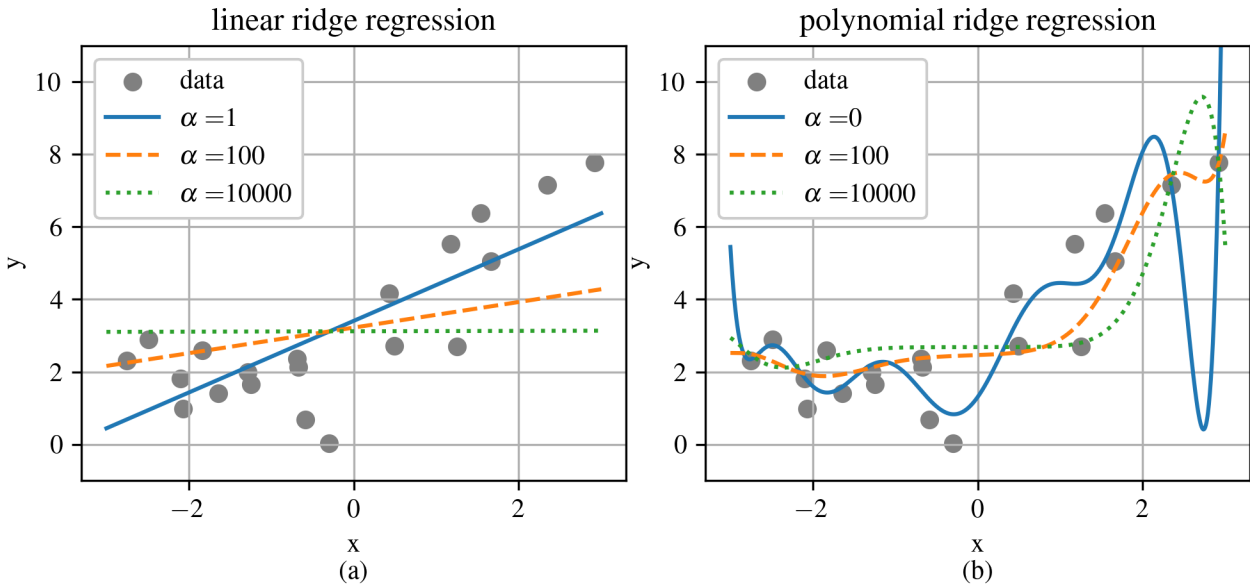


Figure 3.10: Visualization of Ridge Regression models with varying α values.

Figure 3.10 illustrates various Ridge models trained on linear data, showcasing different α value. In Figure 3.10(a), models are purely Ridge Regression, leading to linear predictions. In Figure 3.10(b), the data undergoes Polynomial Regression with Ridge regularization. To do this, the data is first expanded using `PolynomialFeatures(degree=10)`, then scaled with `StandardScaler`, and finally, Ridge models are applied to these transformed features.

Increasing α results in flatter (i.e., more reasonable and less extreme) predictions, thereby reducing the model's variance but increasing its bias. Ridge Regression can be performed using a closed-form solution or through Gradient Descent, similar to Linear Regression. The pros and cons for each method mirror those discussed previously. The closed-form solution, an extension of the normal equation, is given by:

$$\hat{\theta} = (X^\top \cdot X + \alpha A)^{-1} \cdot X^\top \cdot y \tag{3.3}$$

where α influences the regularization and A is an $n \times n$ identity matrix with the top-left value replaced by 0 to exclude the bias term. When using Gradient Descent, the derivative of the cost function guides the adjustment toward optimal model parameters.

Example 3.2 Ridge Regression

This example demonstrates Ridge Regression as a regularized alternative to linear regression. A high-degree polynomial model is fit to noisy data using a pipeline with Ridge regularization, showing how the regularization term reduces overfitting and stabilizes the model.

3.6 Early Stopping

An alternative approach to regularizing iterative learning methods (such as Gradient Descent) is to halt training as soon as the validation error hits its lowest point, a strategy known as early stopping. Figure 3.11 illustrates a complex model, specifically a high-degree Polynomial Regression model, being refined through Batch Gradient Descent. Over the course of training, as epochs progress, the model's prediction error (RMSE) on both the training and validation sets decreases. However, after some time, the validation error ceases to decline and begins to increase, signaling that the model is beginning to overfit the training data. By implementing early stopping, training is terminated the moment the validation error reaches its minimum. Geoffrey Hinton praised this method stating, "Early stopping (is) beautiful free lunch" due to its simplicity [a].

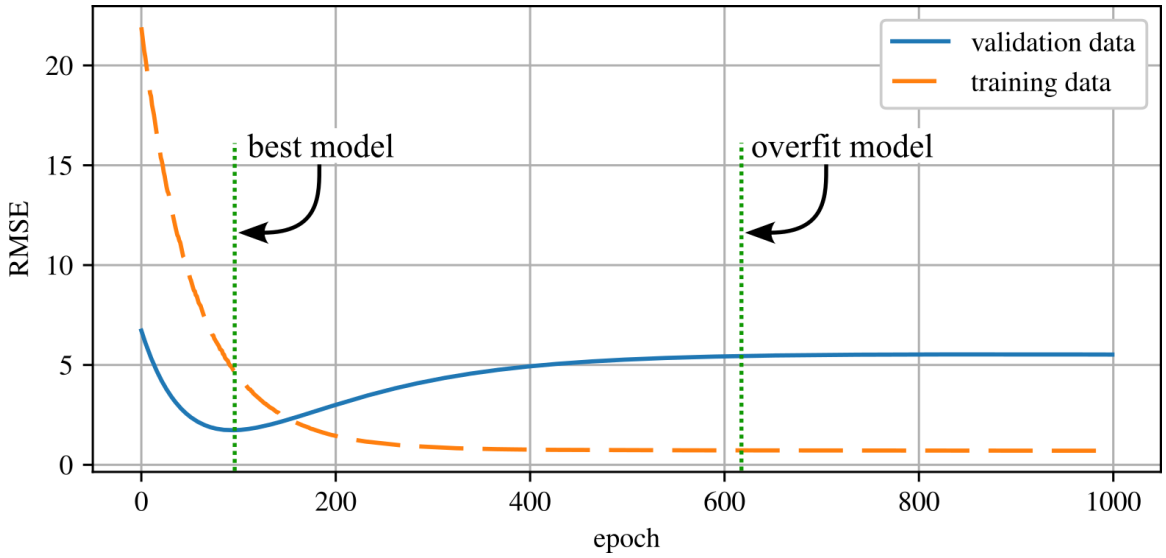


Figure 3.11: An example showing how early stopping can result in a better model as it does not allow for the over-fitting of the training set.

Example 3.3 Early Stopping
This example uses a high-degree polynomial model trained with stochastic gradient descent to demonstrate early stopping. By tracking training and validation errors across epochs, it highlights how halting training early can prevent overfitting and improve generalization.

[a] Pennington, Jeffrey, Richard Socher, and Christopher D. Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.

3.7 Examples

Example 3.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 3.1
Learning curves
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import pipeline

plt.close('all')

#%% build the data sets
np.random.seed(2) # 2 and 6 are pretty good
m = 100
X = 6 * np.random.rand(m,1) - 3
Y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)
X_model = np.linspace(-3,3)

# plot the data
plt.figure()
plt.grid(True)
plt.scatter(X,Y)
plt.xlabel('x')
plt.ylabel('y')

#%% generate learing curves for a linear model

# build the linear model in SK learn
model = sk.linear_model.LinearRegression()

# split the data into training and validation data sets
# Split arrays or matrices into random train and test subsets
X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)

train_errors, val_errors = [], []
for i in range(1, len(X_train)):
    model.fit(X_train[:i], y_train[:i])
    y_train_predict = model.predict(X_train[:i])
    y_val_predict = model.predict(X_val)

    # compute the error for the trained model
    mse_train = sk.metrics.mean_squared_error(y_train[:i],y_train_predict)
    train_errors.append(mse_train)

    # compute the error for the validation model
    mse_val = sk.metrics.mean_squared_error(y_val,y_val_predict)
    val_errors.append(mse_val)

    # predict model
    y_model = model.predict(np.expand_dims(X_model,axis=1))

    plt.figure('test model')
    plt.scatter(X,Y,s=2, label='data')
    plt.scatter(X_train[:i],y_train[:i], label='data in training set')
    plt.scatter(X_val,y_val, marker='s', label='validation data')
    plt.plot(X_model,y_model,'r--',label='model')
    plt.xlabel('x')
    plt.ylabel('y')
```

```
68      plt.legend(loc=2)
69      plt.grid(True)
70      plt.savefig('test_plots/linear_model_'+str(i))
71      plt.close('test model')
72
73  plt.figure()
74  plt.grid(True)
75  plt.plot(train_errors, "--",label="train")
76  plt.plot(val_errors, ":", label="val")
77  plt.xlabel('number of data points in training set')
78  plt.ylabel('mean squared error')
79  plt.legend(framealpha=1)
80  plt.ylim(0,6)
81  #%% generate learning curves for a polynomial model
82
83  model = sk.pipeline.Pipeline((
84  ("poly_features", sk.preprocessing.PolynomialFeatures(degree=20, include_bias=False)),
85  ("lin_reg", sk.linear_model.LinearRegression()),
86  ))
87
88  # split the data into training and validation data sets
89  # Split arrays or matrices into random train and test subsets
90  X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)
91
92  train_errors = []
93  val_errors = []
94  for i in range(1, len(X_train)):
95      model.fit(X_train[:i], y_train[:i])
96      y_train_predict = model.predict(X_train[:i])
97      y_val_predict = model.predict(X_val)
98
99      # compute the error for the trained model
100     mse_train = sk.metrics.mean_squared_error(y_train[:i],y_train_predict)
101     train_errors.append(mse_train)
102
103     # compute the error for the validation model
104     mse_val = sk.metrics.mean_squared_error(y_val,y_val_predict)
105     val_errors.append(mse_val)
106
107     plt.figure('test model')
108     plt.scatter(X,Y,s=2, label='data')
109     plt.scatter(X_train[:i],y_train[:i], label='data in training set')
110     plt.scatter(X_val,y_val, marker='s', label='validation data')
111     y_model = model.predict(np.expand_dims(X_model,axis=1))
112     plt.plot(X_model,y_model,'r--',label='model')
113     plt.xlabel('x')
114     plt.ylabel('y')
115     plt.legend(loc=2)
116     plt.grid(True)
117     plt.savefig('test_plots/polynominal_model_'+str(i))
118     plt.close('test model')
119
120 plt.figure()
121 plt.grid(True)
122 plt.plot(train_errors, "--",label="train")
123 plt.plot(val_errors, ":", label="val")
124 plt.xlabel('number of data points in training set')
125 plt.ylabel('mean squared error')
126 plt.legend(framealpha=1)
127 plt.ylim(0,6)
128
129
130
131
```

Example 3.2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 2.4
Ridge Regression
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import pipeline

plt.close('all')

#%% build the data sets
m = 20
X = 6 * np.random.rand(m, 1) - 3
Y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

X_model = np.linspace(-3,3,num=1000)
X_model = np.expand_dims(X_model,axis=1)


#%% Perform Ridge Regression

# plot the data
plt.figure()
plt.grid(True)
plt.scatter(X,Y,color='gray')
plt.xlabel('x')
plt.ylabel('y')

# build and plot a linear model
model_linear = sk.linear_model.Ridge(alpha=100, solver="cholesky")
model_linear.fit(X, Y)
y_model_linear = model_linear.predict(X_model)
plt.plot(X_model,y_model_linear,'-',label='linear model')

# build and plot a polynomial model
model_poly = sk.pipeline.make_pipeline(sk.preprocessing.PolynomialFeatures(10),
                                       sk.linear_model.Ridge(alpha=100, solver="cholesky"))
model_poly.fit(X, Y)
y_model_poly = model_poly.predict(X_model)
plt.plot(X_model,y_model_poly,'-',label='polynomial model')

plt.legend()
```

Example 3.3

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 3.3
Early Stopping
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk

plt.close('all')

#%% build the data sets

# use 6 to help give a smooth curve that makes the case for early stopping
np.random.seed(6)

m = 20
X = 6 * np.random.rand(m, 1) - 3
Y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)

# plot the data
plt.figure()
plt.grid(True)
plt.scatter(X,Y,color='gray')
plt.xlabel('x')
plt.ylabel('y')

X_model = np.linspace(-3,3,num=1000)
X_model = np.expand_dims(X_model,axis=1)

X_train, X_val, y_train, y_val = sk.model_selection.train_test_split(X, Y, test_size=0.2)

#%% perform early stopping


# prepare the data
poly_scaler = sk.pipeline.Pipeline([("poly_features", sk.preprocessing.PolynomialFeatures
(
    degree=90, include_bias=False)), ("std_scaler", sk.preprocessing.StandardScaler())])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.fit_transform(X_val)

# set up the model, not that by setting max_iter=1 it will only train one epoch
model = sk.linear_model.SGDRegressor(max_iter=1, tol=0,learning_rate="constant"
    ,eta0=0.0005,penalty=None,warm_start=True)


# Train the model in a loop to build the data set to investigate the benefit of early
stopping
val_errors = []
train_errors = []
for epoch in range(1000):
    model.fit(X_train_poly_scaled, y_train.ravel()) # continues where it left off
    y_val_predict = model.predict(X_val_poly_scaled) # Predict the target values
    y_train_predict = model.predict(X_train_poly_scaled) # Predict the target values
    val_error = sk.metrics.mean_squared_error(y_val, y_val_predict)  # Calculate error
    train_error = sk.metrics.mean_squared_error(y_train.ravel(), y_train_predict)  #
    Calculate error
    val_errors.append(val_error)
    train_errors.append(train_error)
```

```
65
66  # plot the early learning curves, you may have to plot this a few times to get
67  # a set of curves that shows strong results
68  plt.figure()
69  plt.grid(True)
70  plt.plot(val_errors,label='validation data')
71  plt.plot(train_errors,'--',label='training data')
72  plt.ylabel('RMSE')
73  plt.xlabel('epoch')
74  plt.legend()
75
76
77
78
79
80
```

4 Classification

In Chapter 1, we examined various tasks that machine learning excels at, such as regression (predicting values) and classification (identifying classes). Having explored linear and polynomial regression in Chapter 2, we now turn our attention to classification in this section.

4.1 Binary Classifier

A binary classifier is a model that assigns inputs to one of two distinct classes. Linear binary classification is a simple but powerful method used to separate data into two distinct classes by drawing a straight decision boundary—such as a line (in 2D) or hyperplane (in higher dimensions)—between them. It works by computing a weighted sum of the input features and applying a threshold to determine the output class. Mathematically, it learns a weight vector θ and bias term b such that the decision rule can be written as:

$$\hat{y} = \begin{cases} 1 & \text{if } \theta^\top \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where $\mathbf{x}$ is the input feature vector. During training, the weights are adjusted to minimize classification error, often using gradient descent on a suitable loss functions (e.g., hinge loss or logistic loss).

Despite its simplicity, the linear classifier is often surprisingly effective, especially when the data is linearly separable or nearly so. An example of such a case is shown in figure 4.1 where the three models (H_1, H_2, and H_3) are linear models defined by the hyperparameters θ that each correctly classify the data. The challenge then becomes how does one obtain the hyperparameters θ. To start, we will show that the hyperparameters θ can be obtained through gradient descent.

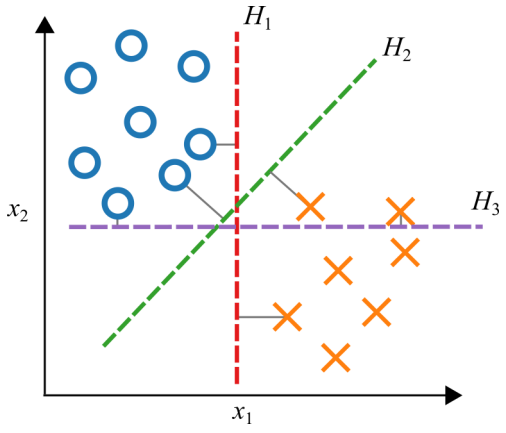


Figure 4.1: Linear classifier that can be implemented with gradient descent.

Review 4.1 MNIST (Modified National Institute of Standards and Technology) Dataset The MNIST (Modified National Institute of Standards and Technology) dataset is a collection of 70,000 small images of handwritten digits (figure 4.2). This dataset is known as "modified" because it combines two earlier sets of images: one created by U.S. high school students and another by US Census Bureau employees. First published in 1998, the dataset is particularly suitable for machine learning tasks as each image is labeled with the digit it represents. Each digit has been normalized and centered in a 28x28 pixel frame and anti-aliased to introduce grayscale levels. Each instance (i.e. picture) has 784 features that range from 0 to 256 and each represent an 8-bit gray scale pixel as shown in figure 4.3.

Typically, the dataset is split into two parts: the first 60,000 images are used for training, and the remaining 10,000 serve as validation data. It is standard practice to train and test models on the first 60,000 images, reserving the last 10,000 for final validation at the end of the project when the classifier is ready for deployment. Scikit-Learn includes a helper function that facilitates the downloading of the MNIST dataset.

Figure 4.2: A collection of the 10 digits (0-9) that make up the MNIST data set.

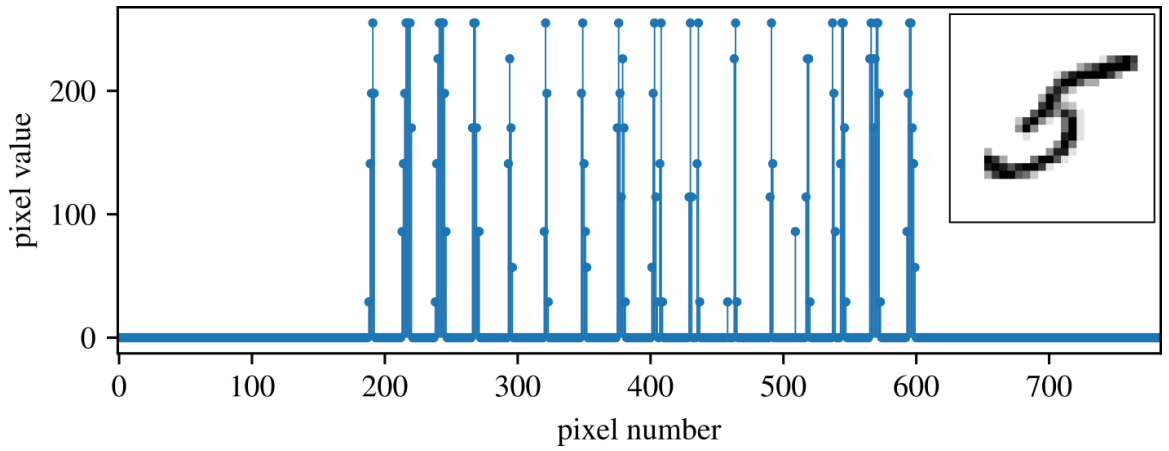


Figure 4.3: The 784 features that range from 0 to 256 and each represent an 8-bit gray scale pixel for a "5 digit" from the MNIST data set.

To begin, we simplify our task by focusing on identifying all the "5"s in the MNIST dataset. This task involves using a binary classifier that checks whether each digit is a 5. The number 5 is notoriously difficult to classify correctly within the MNIST dataset, making it a sensible starting point for binary classification.

Example 4.1 MNIST Dataset
This example demonstrates how to use `sklearn.datasets.fetch_openml` to load the MNIST dataset. It also includes visualization steps to display sample handwritten digits.

4.1.1 Regularized Linear Classifier

A practical algorithm for building this "5-detector" is the Regularized Linear Classifier solved with Stochastic Gradient Descent (SGD). The Regularized Linear Classifier offers an efficient and straightforward approach for discriminative learning of linear classifiers under convex loss functions, scaling well with large datasets and maintaining simplicity in implementation as it processes one training example at a time. The main advantages of the Regularized Linear Classifier include:

- Efficiency.
- Ease of implementation, providing many opportunities for optimization.

However, the Stochastic Gradient Descent classifier also presents certain disadvantages:

- It requires careful tuning of several hyperparameters, including the regularization parameter and the number of iterations.
- It is sensitive to feature scaling.

NOTE

The Regularized Linear Classifier solved with Stochastic Gradient Descent is commonly referred to simply called a Stochastic Gradient Descent classifier, despite Stochastic Gradient Descent only being the method used to trail the linear model. This is exemplified by the fact that scikit-learn uses the name `SGDClassifier` for their model that implements a "Regularized linear models with stochastic gradient descent (SGD) learning".

Given a labeled training set $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{m}$ with $y^{(i)} \in 1,0$ (1 marks the digit "5"), the regularised linear "5-detector" learns a weight vector θ (bias in the first entry θ_1) by minimising

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \max\left(0, 1 - y^{(i)}\theta^{\mathrm{T}}\mathbf{x}^{(i)}\right) + \frac{\lambda}{2}\left\|\theta_{1:}\right\|_2^2, \tag{4.5}$$

where the first term is the hinge loss enforcing a unit margin around the decision boundary. The second term applies ℓ_2-regularisation of strength $\lambda > 0$ to all weights except the bias (θ_1). Minimising (4.5) with stochastic-gradient descent yields a sparse, margin-maximising classifier that generalises well to unseen handwritten digits.

Example 4.2 Regularized Linear Classifier
This example trains a regularized linear classifier using `SGDClassifier` to distinguish the digit "5" from other digits in the MNIST dataset. It highlights how regularization helps improve generalization performance by penalizing large weights, and demonstrates basic prediction on a selected digit using a learned linear decision boundary.

4.2 Performance Measures for Binary Classification

Assessing the performance of a classifier can be more intricate than evaluating a regressor. Nonetheless, there are numerous performance metrics available that facilitate the comprehensive evaluation of a classifier's effectiveness.

4.2.1 Confusion Matrix

First, a brief explanation of Type I and Type II errors. These terms are not exclusive to classification problems in machine learning; they are also crucial in statistical hypothesis testing:

Type I Error: False positive (incorrectly rejecting a true null hypothesis)

Type II Error: False negative (incorrectly failing to reject a false null hypothesis)

These four cases from statistical testing can be combined into a single matrix known as the confusion matrix. In the context of our 5-detector, consider the simplified confusion matrix shown in figure 4.4.

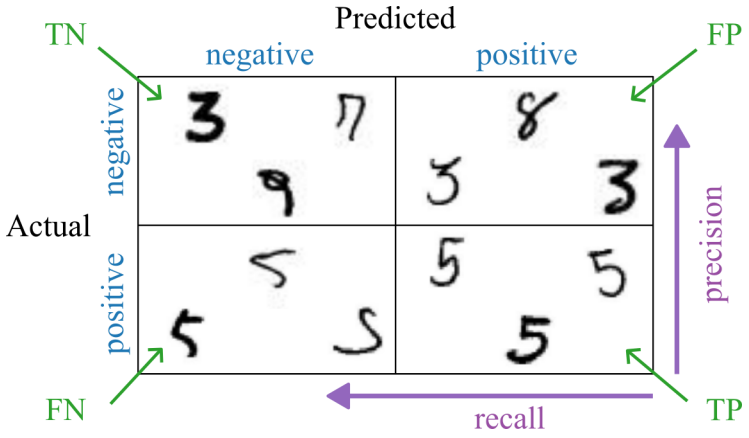


Figure 4.4: Confusion matrix binary.

A effective approach to evaluating classifier performance is to examine the confusion matrix. The essence is to tally the instances where class A is incorrectly classified as class B. A well-performing classifier will exhibit high values along the diagonal and lower values elsewhere in the matrix. Each row represents the actual class, while each column represents the predicted class. For instance, to determine how often the classifier misidentified images of 5s as 3s, inspect the intersection of the 5th row and 3rd column in figure 4.5.

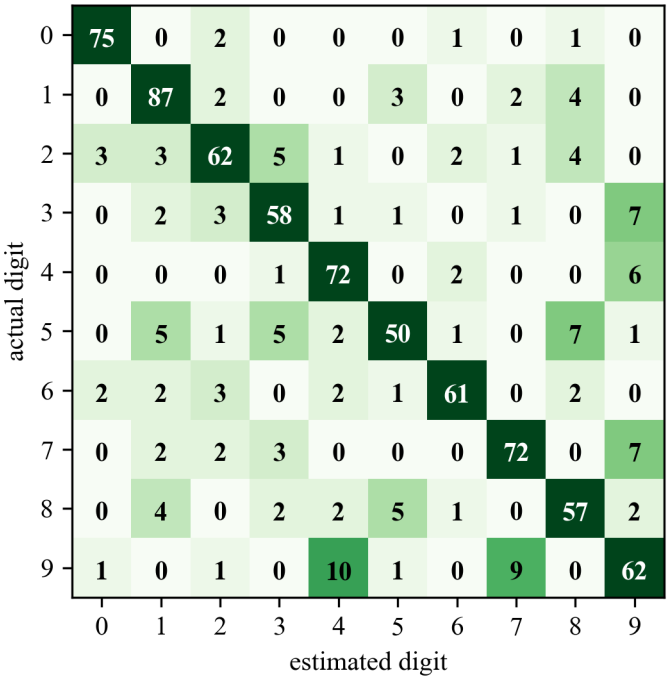


Figure 4.5: Confusion matrix for the first 1,000 data points from the MNIST dataset.

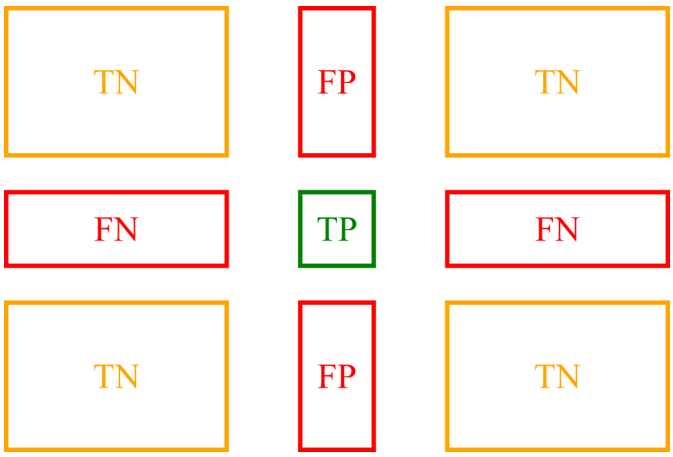


Figure 4.6: Confusion matrix breakout.

Each row in the confusion matrix corresponds to an actual class, while each column corresponds to a predicted class This is diagrammed in figure 4.6. The first row represents non-5 images (the negative class), and the second row represents images classified as 5s (the positive class). The first column of the first row indicates true negatives (TN), while the second column of the first row shows false positives (FP). The second row denotes images of 5s (the positive class): the first column represents false negatives (FN), and the second column displays true positives (TP). A perfect classifier would only have nonzero values along the main diagonal (from top left to bottom right).

Example 4.3 Binary Confusion Matrix
This example evaluates a binary classifier trained to detect the digit "5" using a confusion matrix. The matrix summarizes the number of true positives, true negatives, false positives, and false negatives. It also visualizes specific misclassifications to help interpret where the model fails, such as incorrectly labeling a "5" as not a "5" (false negative) or misidentifying another digit as a "5" (false positive).

4.2.2 Accuracy, Precision, and Recall

The confusion matrix provides detailed information, but sometimes a more concise metric is preferred. Before looking into accuracy, let's define it as

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total number of Predictions}} \tag{4.6}$$

or

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}. \tag{4.7}$$

While accuracy offers valuable insights, it's essential to consider the number of falsely classified positive and negative predictions, especially within the context of the problem being addressed. For instance, a 99% accuracy rate might seem satisfactory for predicting credit card fraud. However, what if a false negative indicates a severe virus or cancer? This is why we need to look deeper into the accuracy formula.

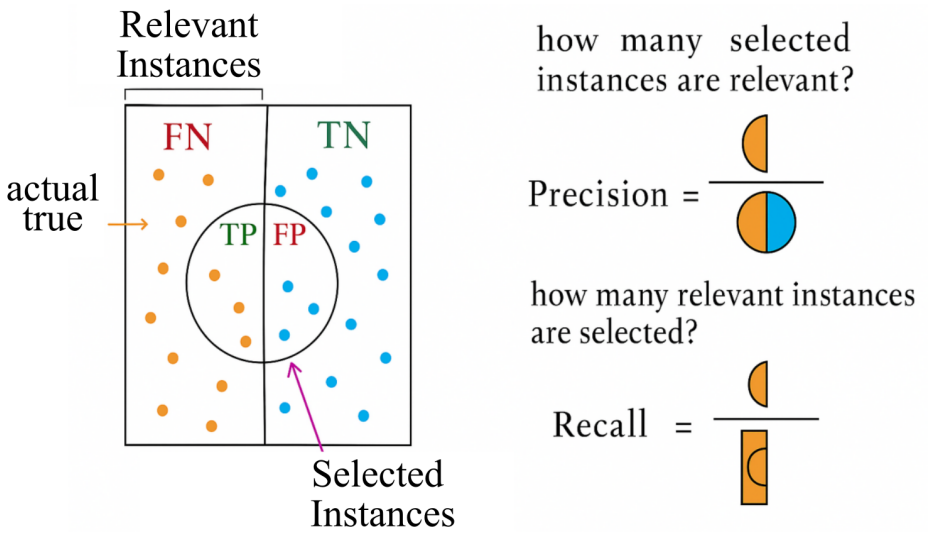


Figure 4.7: Visual representation of relevant instances, showing their relation to True Positives (TP) and False Positives (FP).

This is where precision and recall come into play. For this, we will need a definitions of instances

$$\text{Selected instances} = TP + FP \tag{4.8}$$

and

$$\text{Relevant instances} = TP + FN \tag{4.9}$$

which are visualized in figure 4.7. With these definitions in mind in mind, let's define precision as

$$\text{Precision} = \frac{\text{selected relevant instances}}{\text{selected instances}}, \tag{4.10}$$

or,

$$\text{Precision} = \frac{TP}{TP + FP}. \tag{4.11}$$

One way to achieve perfect precision is to make only one positive prediction and ensure it's correct (precision = 1/1 = 100%). However, this approach would be impractical as it would ignore most positive instances. Recall complements precision by considering all relevant results, both true positives and false negatives. Recall, also known as sensitivity or true positive rate (TPR), is the ratio of positive instances correctly detected by the classifier. We can define recall as

$$\text{Recall} = \frac{\text{selected relevant instances}}{\text{relevant instances}}, \tag{4.12}$$

or,

$$\text{Recall} = \frac{TP}{TP + FN}. \tag{4.13}$$

While the definition of accuracy, precision, and recall may not be immediately intuitive, a visualizations such as that shown in figure 4.7 can help clarify them.

To balance precision and recall, we often use the F1 score, which is the harmonic mean of precision and recall. The F1 score favors classifiers with similar precision and recall, computed as

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}. \tag{4.14}$$

However, optimizing both precision and recall simultaneously is challenging due to the precision/recall tradeoff.

Figure 4.8 illustrates the tradeoffs when adjusting the decision threshold in a Regularized Linear Classifier. By adjusting the classification threshold, we can control the balance between precision and recall. Lowering the threshold increases recall but reduces precision, and vice versa.

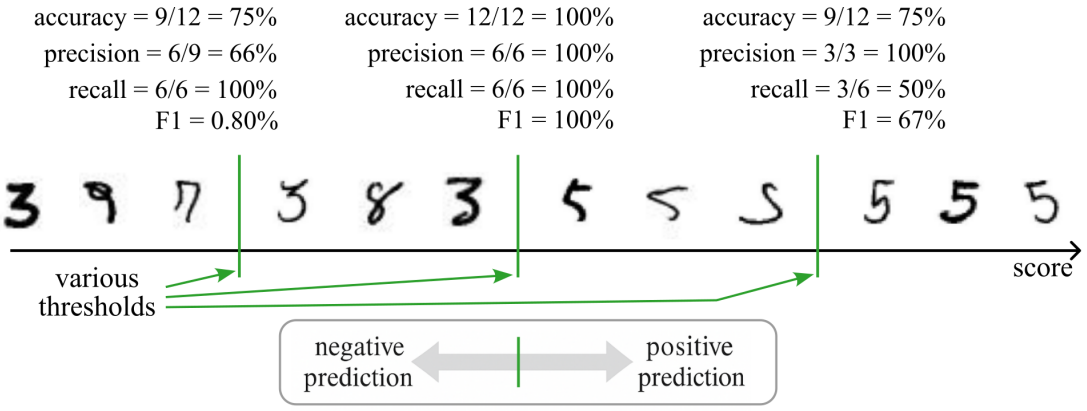


Figure 4.8: Visualization changing thresholds on the classification of digits in a "5-detector" and the resulting change in precision and recall.

To determine the optimal threshold, it's helpful to plot precision and recall against the threshold values. By default, the classifier in scikit-learn uses a threshold of zero, but this can be adjusted as needed. As illustrated in Figure 4.9, it's relatively straightforward to develop a classifier with nearly any desired precision: simply adjust the threshold to a sufficiently high value. However, it's important to bear in mind that a high-precision classifier may not be very practical if its recall is too low.

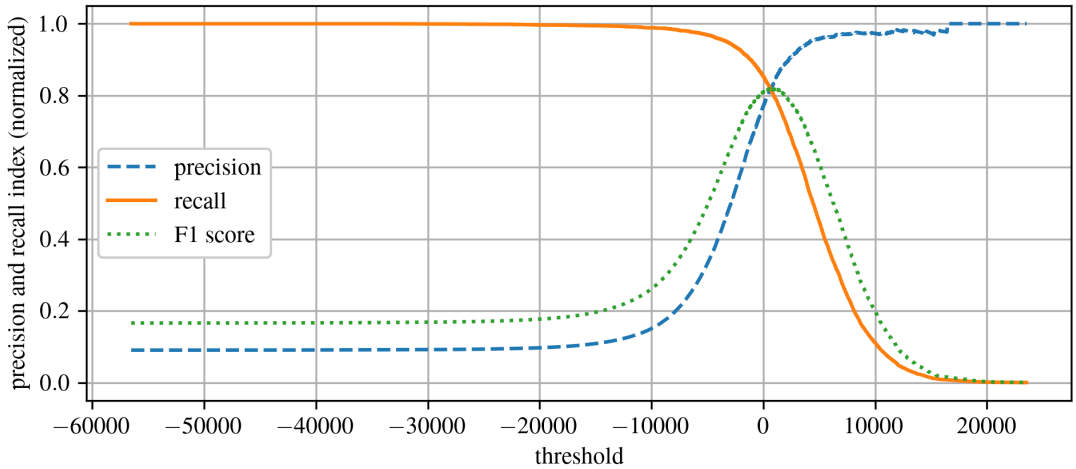


Figure 4.9: Visualization of the precision, recall, and F1 score as a function of changing a threshold value.

Figure 4.10 reports a precision vs recall curve that is obtained by changing the threshold of the trained model. At the leftmost part of the precision-recall curve (low recall), precision is highest because the model makes very few positive predictions, and those are likely true positives. As recall increases, the model starts predicting more positives, including more false positives, which reduces precision. The shape of the curve is also important; a steep drop-off indicates that the addition of false positives happens rapidly with increasing recall, whereas a more gradual decline suggests a better balance between precision and recall.

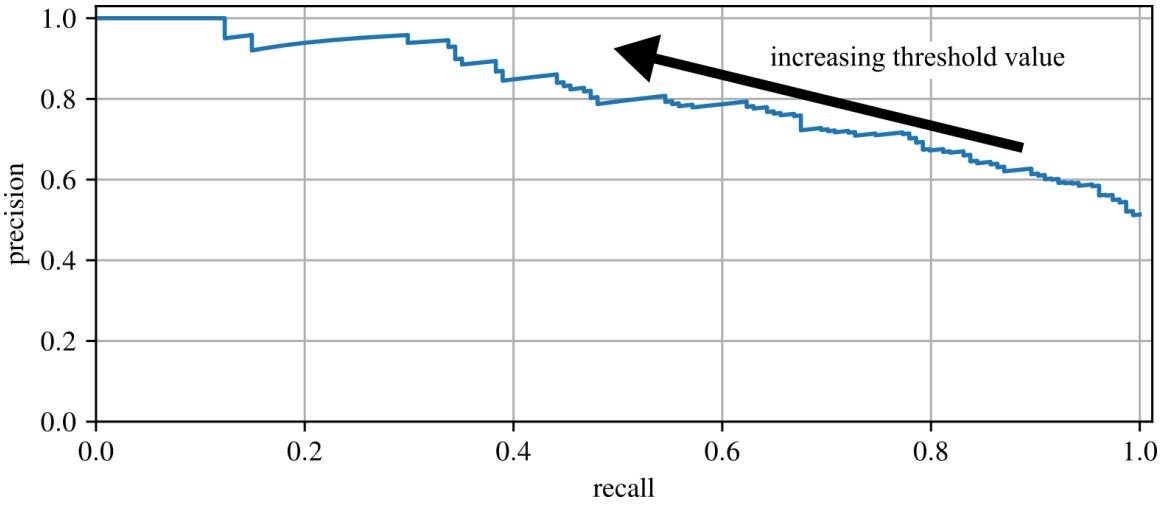


Figure 4.10: The relationship between precision vs recall for a changing threshold value.

By analyzing the curve, one can choose a threshold that balances precision and recall according to the specific needs of the application. For instance, in medical diagnosis, high recall is crucial to ensure no cases are missed, even if precision is lower. Additionally, the area under the precision-recall curve (AUC-PR) can be used to compare models. A higher area indicates a model with better performance across all thresholds.

Example 4.4 Accuracy, Precision, and Recall
This example computes and compares three key classification metrics (accuracy, precision, and recall) for a linear classifier trained to detect the digit "5" in the MNIST dataset. It shows both manual and `sklearn`-based calculations, introduces the F1 score, and visualizes how precision and recall vary with the classification threshold using a precision-recall curve.

4.3 k-fold Cross-validation

As you've likely observed, training your SGD classifier doesn't always yield consistent results in terms of model performance; hence the "stochastic" in stochastic gradient descent. One approach to obtaining a solution closer to the global minimum is to allow the gradient descent algorithm to iterate over more epochs. However, this poses a challenge as it's highly time-consuming and may lead to overfitting the model. Moreover, the variability in model performance makes it difficult to gauge the model's effectiveness.

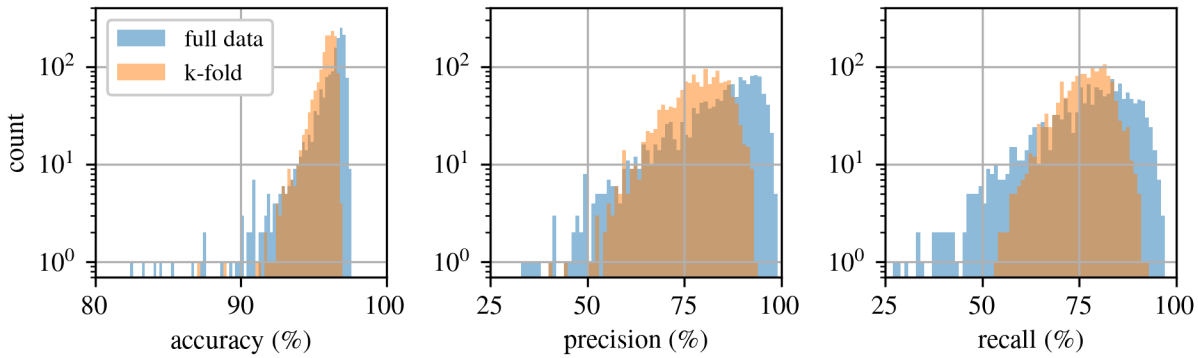


Figure 4.11: Performance metrics for 500 5-detectors trained either with full data or through k-fold cross-validation with 3 folds.

k-fold cross-validation offers a more rigorous method for evaluating your algorithm's performance. For instance, consider Figure 4.11, which depicts the performance metrics for 500 classification models trained as 5-detectors. Noticeably, the results obtained using k-fold cross-validation exhibit significantly less variability compared to training the classifier on the full dataset. In k-fold cross-validation, the training set is divided into smaller subsets for training and validation. The model is trained on these subsets and evaluated on the validation sets. Figure 4.12 provides a graphical representation of this technique.

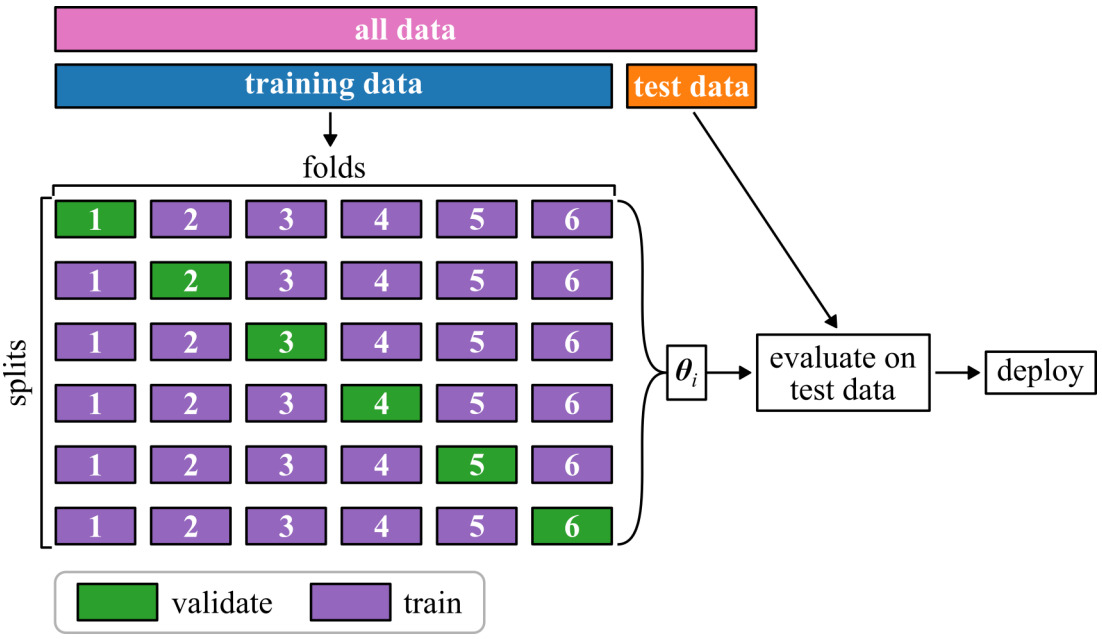


Figure 4.12: Illustration of how k-fold cross-validation operates.

Of course, Scikit-Learn provides a cross-validation feature, known as `sk.model_selection.cross_val_score`. This function conducts k-fold cross-validation by randomly partitioning the training set into distinct folds. Each fold is used for validation while the rest are used for training the SGD classifier. The reported performance metrics are averaged across all folds. It's important to note that by partitioning the data into folds, the number of samples available for learning is limited, which may result in peculiar classifiers with unusually high or low error rates. Despite being more computationally expensive, this approach effectively utilizes all the data for training and validation.

Example 4.5 ***k*-fold Cross-Validation**
This example trains a binary classifier to detect the digit "5" from the MNIST dataset using Stochastic Gradient Descent (SGD). It first demonstrates how model performance can vary when trained multiple times on the same dataset. Then, by applying k-fold cross-validation using `sk.model_selection.cross_val_predict`, it shows how splitting the data into k subsets helps reduce this variation and provides a more reliable estimate of accuracy, precision, and recall.

4.4 Multiclass Classification

Previously, we explored binary classifiers, which distinguish between two classes. Now, we'll explore the use of multiclass classifiers (also known as multinomial classifiers), which can discern between more than two classes. The simplest approach to multiclass classification involves multiple iterations of a binary classifier. There are two primary strategies for achieving this, both of which are considered in the context of the MNIST database challenge:

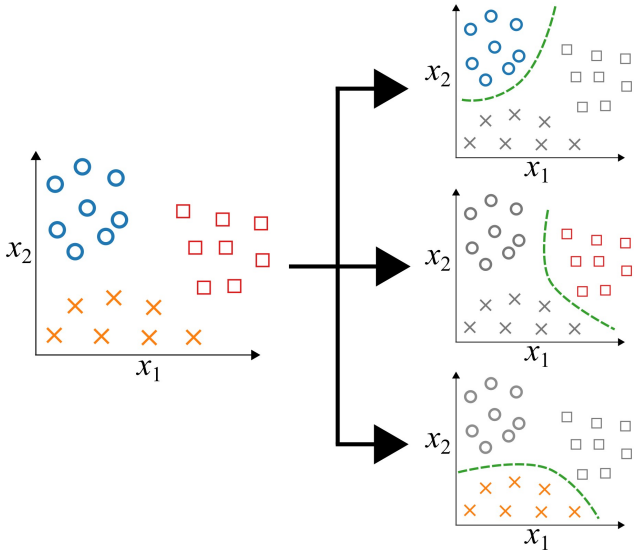


Figure 4.13: One-versus-Rest (OvR) classifier for a multi-class dataset.

- **One-versus-Rest (OvR)**: This strategy involves creating a system capable of classifying digit images into 10 classes (from 0 to 9) by training 10 binary classifiers, one for each digit (e.g., a 0-detector, a 1-detector, etc.). When classifying an image, the decision score from each classifier is obtained, and the class with the highest score is selected. While sometimes referred to as One-Versus-All (OvA), this term is slightly misleading as it does not involve comparisons against the same class.

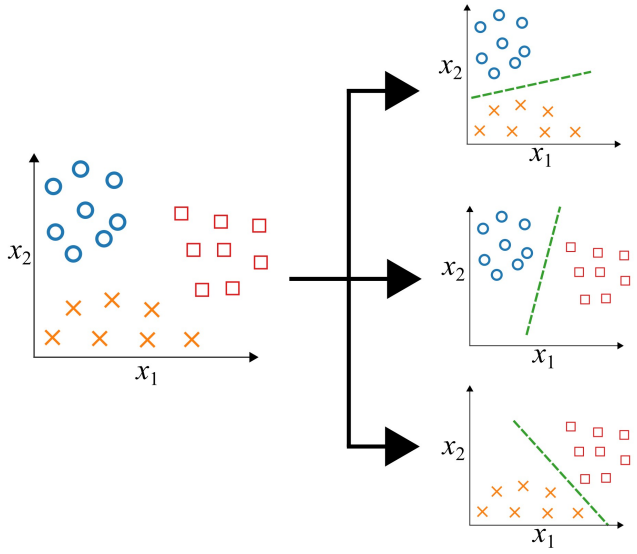


Figure 4.14: One-versus-one (OvO) classifier for a multi-class dataset.

- **One-versus-one (OvO)**: This approach trains a binary classifier for every pair of digits, such as one for distinguishing between 0s and 1s, another for distinguishing between 0s and 5s, and so forth. If there are N classes, $N \times (N-1)/2$ classifiers need to be trained. For

the MNIST problem, this translates to training 45 binary classifiers. During classification, the image is evaluated against all 45 classifiers, and the class with the most victories is selected. The primary advantage of OvO is that each classifier only needs to be trained on the relevant portion of the training set for the two classes it distinguishes, which is beneficial for algorithms that scale poorly.

Certain algorithms, such as Support Vector Machine classifiers, suffer from scalability issues with larger training sets. Algorithms that do not scale well with large datasets often resort to the OvO approach because training many small classifiers is computationally cheaper than fitting a handful of models on the full data. For most binary classifiers the OvR strategy is the norm. Some methods (including Random Forests and naïve Bayes) support multiclass problems natively and therefore require neither reduction strategy.

Example 4.6 Multiclass Regularized Linear Classifier
This example extends the use of `SGDClassifier` to multiclass classification on the MNIST dataset. It compares three approaches: One-vs-Rest (OvR), One-versus-One (OvO), and Scikit-Learn's default automatic strategy. Each method trains multiple binary classifiers to distinguish between the 10 digit classes.

4.5 Performance Measures for Multiclass Classification

Assessing a multiclass classifier often poses more challenges compared to evaluating a binary classifier. Nonetheless, many of the principles and methodologies utilized in evaluating binary classifiers can be seamlessly applied to multiclass classifiers as well.

4.5.1 Confusion Matrix

First, examine the confusion matrix. To do this, generate predictions using the `cross_val_predict()` function, followed by calling the `confusion_matrix()` function, similar to your previous steps. The confusion matrix in figure 4.15 appears satisfactory, with most images located on the main diagonal, indicating correct classification. However, the 5s appear slightly darker than other digits, suggesting fewer instances of 5s in the dataset or inferior performance of the classifier on 5s. Further investigation confirms both scenarios.

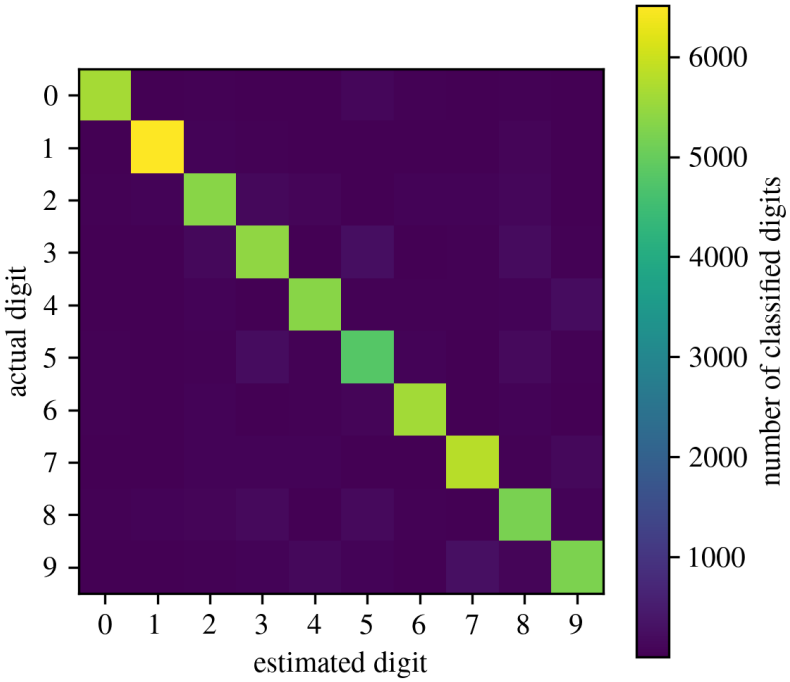


Figure 4.15: Confusion matrix for the MNIST data set solved using a one-versus-one classifier with Stochastic Gradient Descent and a k-fold of 3.

To turn figure 4.15 into a figure that highlights the mistakes, first normalize each entry of the confusion matrix by the number of images in its true class so you compare error rates rather than raw counts that would overweight frequent classes. Then set the diagonal cells to `NaN` so only the off-diagonal errors remain visible, and plot the resulting matrix. The resulting figure 4.16 more clearly highlights the errors in the confusion matrix.

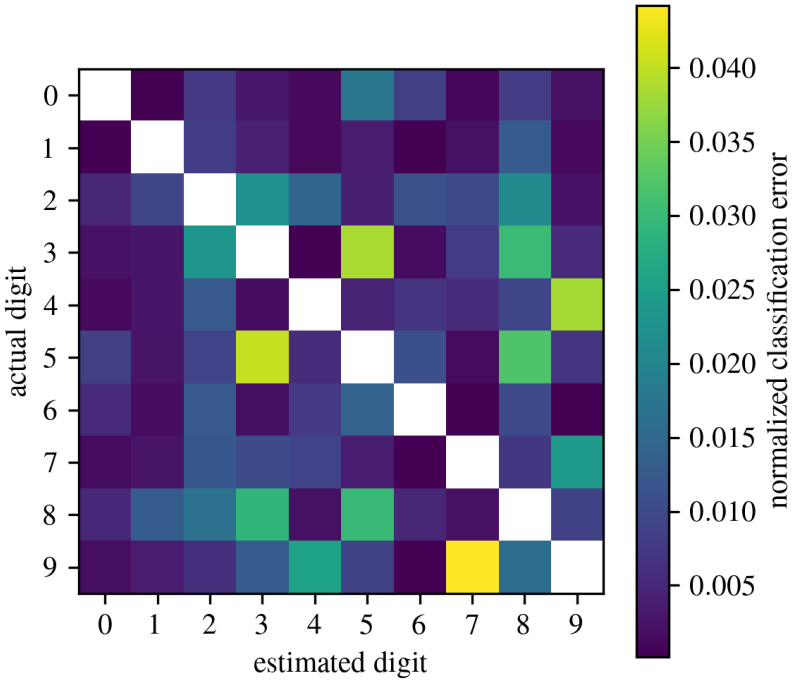


Figure 4.16: Normalized confusion matrix (similar to figure 4.15) with the diagonal removed to emphasize errors.

The refined plot facilitates a clear understanding of the classifier's error patterns. Rows represent actual classes, while columns depict predicted classes. Columns corresponding to classes 8 and 9 exhibit brightness, indicating numerous misclassifications as 8s or 9s. Likewise, rows for classes 8 and 9 also appear bright, signifying frequent confusion between 8s, 9s, and other digits. Conversely, some rows, like row 1, display darkness, indicating accurate classification of most 1s (with few exceptions confused with 8s). Notably, errors are asymmetric; for instance, more 5s are misclassified as 8s than vice versa.

Analyzing the confusion matrix provides valuable insights for classifier enhancement. In this case, efforts should concentrate on improving the classification of 8s and 9s, along with addressing the specific 3/5 confusion. Potential strategies include gathering more training data for these digits, devising new features (e.g., counting closed loops), or preprocessing images to emphasize distinguishing patterns (e.g., using Scikit-Image, Pillow, or OpenCV).

4.5.2 Analyzing Individual Errors

Analyzing individual errors provides valuable insights into the classifier's behavior and reasons for its failures.

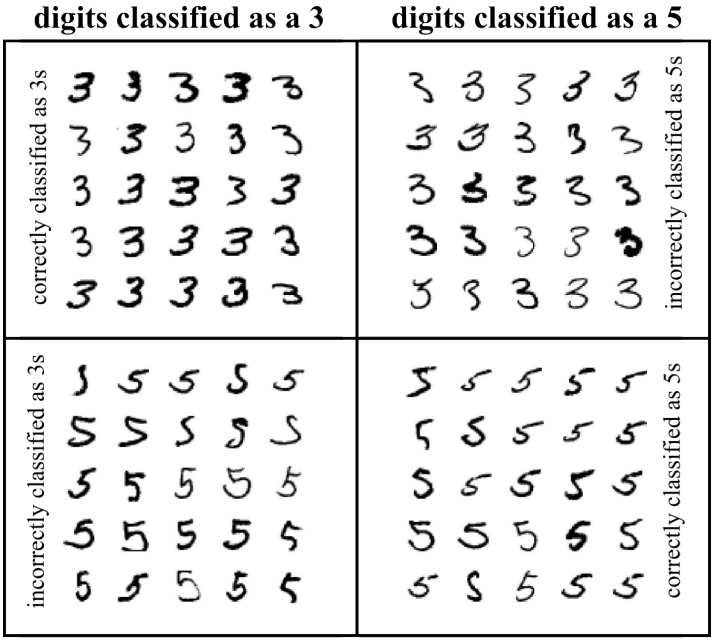


Figure 4.17: Confusion matrix showing digits classified as 3s and 5s for the same classifier used to obtain the results shown in figure 4.15.

The left half of figure 4.17 contains the blocks for images the classifier labelled as 3, while the right half shows those it labelled as 5. A few misclassified digits are so poorly written that even a human might hesitate, yet many errors look quite clear. Remember that a regularised linear classifier assigns a single weight to every pixel for each class and then sums the weighted pixel intensities to score each class.

Because just a handful of pixels separate a 3 from a 5, a linear classifier often confuses the two. Their main distinction is the small stroke that joins the top bar to the bottom curve; even a slight shift or rotation of this junction can flip the prediction. The linear model is therefore very sensitive to translations and rotations. Pre-processing the images to center the digits and correct their orientation should lessen the 3-5 confusion and improve performance across the board. 5

A non-linear model can learn more complex patterns by applying transformations (such as kernel mappings), effectively giving different relative importance to various parts of the image, and would therefore be better equipped to capture more intricate pixel patterns and more effectively separate these digits.

Example 4.7 Multiclass Confusion Matrix

This example uses a One-versus-One classifier trained with Stochastic Gradient Descent to evaluate multiclass classification performance on the MNIST dataset. It constructs a confusion matrix using 3-fold cross-validation and visualizes both the raw classification counts and the normalized error rates, helping identify which digits are most commonly misclassified.

4.6 Examples

Example 4.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 4.1 Load the MNIST data set
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import datasets

plt.close('all')


#%% Load your data

# this fetches "a" MNIST dataset  from openml and loads it into your environment
# as a Bunch, a Dictionary-like object that exposes its keys as attributes.
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')

# calling the DESCR key will return a description of the dataset
print(mnist['DESCR'])

# calling the data key will return an array with one row per instance and one
# column per feature where each features is a pixel, as defined in the key feature_names
X = mnist['data']


# calling the target key will return an array with the labels
Y = np.asarray(mnist['target'],dtype=int)

# Each image is 784 features or 28×28 pixels, however, the features must be reshaped
# into a 29x29 grid to make them into a digit, where the values represents one
# the intensity of one pixel, from 0 (white) to 255 (black).

digit_id = 35 # An OK 5
# digit_id = 0 # An odd 5
# digit_id = 100 # A bad 5


test_digit = X[digit_id,:]
digit_reshaped = np.reshape(test_digit,(28,28))

# plot an image of the random pixel you picked above.
plt.figure()
plt.imshow(digit_reshaped,cmap = mpl.cm.binary,interpolation="nearest")
plt .title('A "'+str(Y[digit_id])+'" digit from the MNIST dataset')
plt.xlabel('pixel column number')
plt.ylabel('pixel row number')


```

Example 4.2

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 4.2 Stochastic Gradient Descent (SDG) for the MINST data set
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import datasets

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')

#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')
X = mnist['data']    # load the data
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Train a Stochastic Gradient Descent classifier

# Extract a subset for our "5-detector".
Y_train_5 = (Y_train == 5)
Y_test_5 = (Y_test == 5)

# build and train the classifier
sgd_clf = sk.linear_model.SGDClassifier()
sgd_clf.fit(X_train, Y_train_5)

# get a digit from the dataset to test the classifier on
digit_id = 35 # An OK 5
# digit_id = 0 # An odd 5
# digit_id = 100 # A bad 5
test_digit = X[digit_id,:]
digit_reshaped = np.reshape(test_digit,(28,28))

# plot an image of the random pixel you picked above.
plt.figure()
plt.imshow(digit_reshaped,cmap = mpl.cm.binary,interpolation="nearest")
plt .title('A "'+str(Y[digit_id])+'" digit from the MNIST dataset')
plt.xlabel('pixel column number')
plt.ylabel('pixel row number')
plt.savefig('MNIST_digit')


# we can now test this for the "5" that we plotted earlier.
print(sgd_clf.predict([test_digit])) # a True case
print(sgd_clf.predict([X[2,:]])) # a False case


```

Example 4.3

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 4.3 Confusion matirx for the MINST dataset
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import pipeline
from sklearn import datasets
from sklearn import metrics

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')
X = mnist['data']    # load the data
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Train a Stochastic Gradient Descent classifier

# Extract a subset for our "5-dector".
Y_train_5 = (Y_train == 5)
Y_test_5 = (Y_test == 5)

# build and train the classifier
sgd_clf = sk.linear_model.SGDClassifier()
sgd_clf.fit(X_train, Y_train_5)

# we can now test this for the "5" that we plotted earlier.
digit_id = 35
test_digit = X[digit_id,:]
print(sgd_clf.predict([test_digit]))

#%% Build the Confusion Matrices

# Return the predictions made on each test fold
X_train_pred = sgd_clf.predict(X_train)


# build the confusion Matrix
print(sk.metrics.confusion_matrix(Y_train_5, X_train_pred))

# Now let's find all the False positive and false negative
confusion_booleans = np.vstack((Y_train_5, X_train_pred)).T
FN_index = np.where((confusion_booleans == [True,False]).all(axis=1))[0]
FP_index = np.where((confusion_booleans == [False,True]).all(axis=1))[0]

```

```
# We built a 5-detector, so:
# True and True is true positive (TP)
# False and False is True Negative (TN)
# True and False is false negative (FN)
# False and True is false positive (FP)


# from this, we see #0 is a false negative (FN), i.e. its is actually a 5 but
# the classifier said it was not. We can plot this digit below
digit_id = 0
test_digit = X[digit_id,:]
digit_reshaped = np.reshape(test_digit,(28,28))
plt.figure()
plt.imshow(digit_reshaped,cmap = mpl.cm.binary,interpolation="nearest")
plt .title('A "'+str(Y[digit_id])+'" digit from the MNIST dataset')
plt.xlabel('pixel column number')
plt.ylabel('pixel row number')

# Lastly, we see that #68 is classified as an false positive (FP), again, we can plot
this.
digit_id = 161
test_digit = X[digit_id,:]
digit_reshaped = np.reshape(test_digit,(28,28))
plt.figure()
plt.imshow(digit_reshaped,cmap = mpl.cm.binary,interpolation="nearest")
plt .title('A "'+str(Y[digit_id])+'" digit from the MNIST dataset')
plt.xlabel('pixel column number')
plt.ylabel('pixel row number')


```

Example 4.4

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 3.4 Precision and recall accuracy for the MNIST dataset
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import datasets
from sklearn import metrics

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')
X = np.asarray(mnist['data'])     # load the data
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Train a Stochastic Gradient Descent classifier

# Extract a subset for our "5-detector".
Y_train_5 = (Y_train == 5)
Y_test_5 = (Y_test == 5)

# build and train the classifier
sgd_clf = sk.linear_model.SGDClassifier()
sgd_clf.fit(X_train, Y_train_5)

#%% Build the Confusion Matrices

# Return the predictions made with the trained model
X_train_pred = sgd_clf.predict(X_train)

# build the confusion Matrix
confusion_matrix = sk.metrics.confusion_matrix(Y_train_5, X_train_pred)

TN = confusion_matrix[0,0]
FP = confusion_matrix[0,1]
FN = confusion_matrix[1,0]
TP = confusion_matrix[1,1]

#%% Calculate Precision and Recall

# calculate the Precision and Recall values using the commands discussed in class
accuracy = (TP + TN)/(TP + TN + FP + FN)
precision = TP/(TP+FP)
recall = TP/(TP+FN)

# of course, SK learn has built-in functions for this.
```

```
accuracy_SK = sk.metrics.accuracy_score(Y_train_5, X_train_pred)
precision_SK = sk.metrics.precision_score(Y_train_5, X_train_pred)
recall_SK = sk.metrics.recall_score(Y_train_5, X_train_pred)

# compute the F1 score for the data set
F1 = sk.metrics.f1_score(Y_train_5, X_train_pred)

#%% plot the precision and recall over the threshold domain.

# first, compute the scores for the predictions.
Y_scores= sgd_clf.decision_function(X_train)

#Y_scores= Y3
# now, for the scores and the target training set, calculate the precisions, recalls,
threshold values.
precisions, recalls, thresholds = sk.metrics.precision_recall_curve(Y_train_5, Y_scores)

# now, compute the F1 score over the entire threshold range
F1s = 2/(1/precisions + 1/recalls)

# plot the Precision and Recall vs threshold.
plt.figure()
plt.plot(thresholds, precisions[:-1], "--", label="Precision")
plt.plot(thresholds, recalls[:-1], "-", label="Recall")
plt.plot(thresholds, F1s[:-1], ":", label="F1 score")
plt.xlabel("Threshold")
plt.ylabel("normalized precision\nand recall index")
plt.legend(loc=6,framealpha=1)
plt.ylim([-0.05, 1.05])
plt.grid(True)
plt.tight_layout()
```

Example 4.5

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 4.5 k-fold cross-validationStochastic for Gradient Descent (SDG) using the MINST
data set
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().magic('reset -sf')

import numpy as np
import scipy as sp
import pandas as pd
from scipy import fftpack, signal # have to add
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import linear_model
from sklearn import pipeline
from sklearn import datasets

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False)
X = mnist['data']    # load the data
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Train a Stochastic Gradient Descent classifier

# Extract a subset for our "5-detector".
Y_train_5 = (Y_train == 5)
Y_test_5 = (Y_test == 5)

sgd_clf = sk.linear_model.SGDClassifier()

#%% Build a 5-detector several times so see the variation in metrics returned by SGD

# Solve the model multiple times to see the variations
for i in range(5):

    # Train the model and eturn the predictions made by SGD
    sgd_clf.fit(X_train, Y_train_5)
    Y_train_pred = sgd_clf.predict(X_train)

    # Use SK learn model to retunr metrics
    accuracy = sk.metrics.accuracy_score(Y_train_5, Y_train_pred)
    precision = sk.metrics.precision_score(Y_train_5, Y_train_pred)
    recall = sk.metrics.recall_score(Y_train_5, Y_train_pred)
    print('accuracy is '+str(np.round(accuracy,4))+'; precision is '+str(np.round(
    precision,4))+
          '; recall is '+str(np.round(recall,4)))


#%% Build a 5-detector using k-fold cross-validation
```

```
66
67  # From our example we can see quite a variation in results for a model, making it
68  # hard to select the proper model. However, k-fold cross-validation can help with this.
69
70
71  # make a prediction using the k-fold method to split up the data set. Again,
72  # solve the model multiple times to see the variations
73  for i in range(5):
74
75      # Train the model and eturn the predictions made by SGD
76      Y_train_pred = sk.model_selection.cross_val_predict(sgd_clf, X_train, Y_train_5, cv=3
        )
77
78      # Use SK learn model to retunr metrics
79      accuracy = sk.metrics.accuracy_score(Y_train_5, Y_train_pred)
80      precision = sk.metrics.precision_score(Y_train_5, Y_train_pred)
81      recall = sk.metrics.recall_score(Y_train_5, Y_train_pred)
82      print('accuracy is '+str(np.round(accuracy,4))+'; precision is '+str(np.round(
        precision,4))+
83            '; recall is '+str(np.round(recall,4)))
84
85
```

Example 4.6

```
"""
Example 4.6 Multiclass Stochastic Gradient Descent (SDG) for the MINST data set
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
import time as time
from sklearn import linear_model
from sklearn import pipeline
from sklearn import datasets
from sklearn import multiclass

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')

#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')
X = np.asarray(mnist['data'])   # load the data
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Train a Multiclass Stochastic Gradient Descent classifiers

# SK learn has a  Multiclass and multilabel module as sk.multiclass. You can use
# this module to do one-vs-the-rest or one-vs-one classification.

# here we test a one-vs-rest classifier that uses Stochastic Gradient Descent
tt_1 = time.time()
ovr_clf = sk.multiclass.OneVsRestClassifier(sk.linear_model.SGDClassifier())
ovr_clf.fit(X_train, Y_train)
print('One-vs-Rest took '+str(time.time()-tt_1 )+' seconds to train and execute')

# here we test a one-vs-one classifier that uses Stochastic Gradient Descent
tt_1 = time.time()
ovo_clf = sk.multiclass.OneVsOneClassifier(sk.linear_model.SGDClassifier())
ovo_clf.fit(X_train, Y_train)
print('One-vs-one took '+str(time.time()-tt_1 )+' seconds to train and execute')

# Moreover, Scikit-Learn detects when you try to use a binary classification algorithm
for
# a multiclass classification task, and it automatically runs OvA (except for SVM
classifiers for which it uses OvO).
tt_1 = time.time()
multi_sgd_clf = sk.linear_model.SGDClassifier()
multi_sgd_clf.fit(X_train, Y_train) # y_train, not y_train_5
print('SK learns automated selection (OvA) took '+str(time.time()-tt_1 )+' seconds to
train and execute')
```

Example 4.7

```
"""
Example 4.7 Multiclass confusion matrix for the MINST data set
Machine Learning for Engineering Problem Solving
@author: Austin Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')

#%% Load your data

# Fetch the MNIST dataset from openml
mnist = sk.datasets.fetch_openml('mnist_784',as_frame=False,parser='auto')
X = np.asarray(mnist['data'])    # load the data and convert to np array
Y = np.asarray(mnist['target'],dtype=int)  # load the target

# Split the data set up into a training and testing data set
X_train = X[0:60000,:]
X_test = X[60000:,:]
Y_train = Y[0:60000]
Y_test = Y[60000:]

#%% Confusion Matrix for a Multiclass classifier.

# Use the one-vs-one classifier that uses Stochastic Gradient Descent as this is
# faster for this specific data set
ovo_clf = sk.multiclass.OneVsOneClassifier(sk.linear_model.SGDClassifier())

# make a prediction for every case using the k-fold method.
Y_train_pred = sk.model_selection.cross_val_predict(ovo_clf, X_train, Y_train, cv=3)
conf_mx = sk.metrics.confusion_matrix(Y_train, Y_train_pred)
print(conf_mx)

# plot the results
fig = plt.figure(figsize=(4,4))
pos = plt.imshow(conf_mx) #, cmap=plt.cm.gray)
cbar = plt.colorbar(pos)
cbar.set_label('number of classified digits')
plt.ylabel('actual digit')
plt.xlabel('estimated digit')
plt.savefig('confusion_matrix',dpi=300)

# Normalize the confusion matrix by class size to compare error rates, not raw counts.
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums

# Next, we remove the high values along the diagonal. This is done by converting the
# confusion matrix to a float data type, and replacing everything on the diagonal with
NaNs.
conf_mx_noise = np.asarray(norm_conf_mx,dtype=np.float32)
np.fill_diagonal(conf_mx_noise, np.NaN)

# plot the results but only consider the noise
fig = plt.figure(figsize=(4,4))
pos = plt.imshow(conf_mx_noise) #, cmap=plt.cm.gray)
cbar = plt.colorbar(pos)
cbar.set_label('normalized classification error')
plt.ylabel('actual digit')
plt.xlabel('estimated digit')
plt.savefig('confusion_matrix_error',dpi=300)
```

5 Regression-Based Classification

Certain algorithms that were first developed for regression can be adapted for classification, and some classifiers can be modified to predict continuous values. Converting a linear regressor into a classifier by adding a logistic link, for instance, retains the original coefficient vector, which makes it easy to see how each input feature influences the decision. As these dual-purpose models expose many of the hyper-parameters found in their regression versions, they often provide more opportunities for fine-tuning and clearer interpretability than algorithms designed purely for classification.

5.1 Logistic Regression

Logistic Regression, sometimes called logit regression, models the probability that a sample belongs to a specific class, such as the chance that an email is spam. When this probability exceeds 50%, the model predicts the positive class (label "1"); otherwise, it predicts the negative class (label "0"). It therefore acts as a binary classifier.

The predicted probability is

$$\hat{p} = h_\theta(X) = \sigma(\theta^\top T \cdot X). \tag{5.1}$$

Here $\hat{p}$ is the estimated probability, and $\sigma(\cdot)$ is the sigmoid function, an S-shaped curve that maps any real number to the interval $(0,1)$. The logistic function is defined in Equation 5.2 and illustrated in Figure 5.1. As before, $h_\theta(X)$ denotes the hypothesis that the input matrix X, augmented with a bias term, belongs to the positive class under the parameters θ.

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{5.2}$$

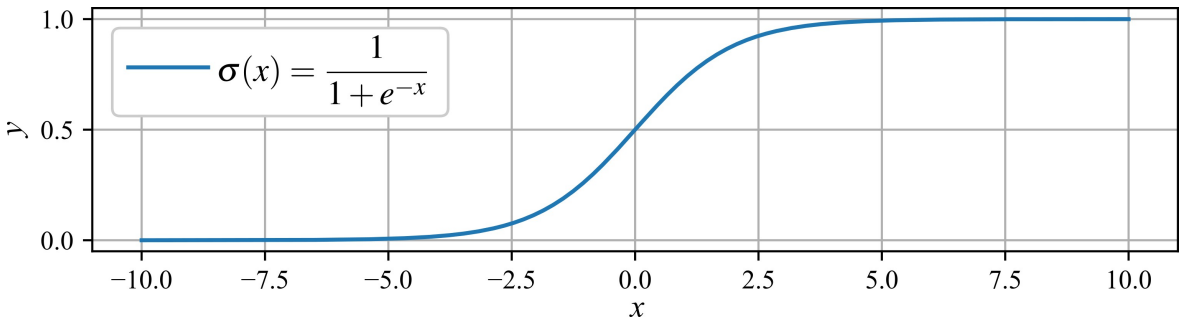


Figure 5.1: Sigmoid function that maps any real-valued input x to a value between 0 and 1.

Once the probability $\hat{p} = h_\theta(X)$ that an instance X belongs to the positive class has been estimated using Logistic Regression, the prediction ($\hat{y}$) can be made. $\hat{y}$ is calculated as

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases} \tag{5.3}$$

Note that $\sigma(x) < 0.5$ when $x < 0$, and $\sigma(x) \geq 0.5$ when $x \geq 0$. Therefore, the Logistic Regression model predicts 0 if $\theta^\top \cdot X$ is negative, and 0 if it is positive.

Now that we understand how logistic regression assigns probabilities and produces predictions, let's walk through a concise example that shows its training procedure and the associated cost function. Training seeks parameter values θ that give high predicted probabilities to positive examples ($y = 1$) and low probabilities to negative ones ($y = 0$). This aim is captured by the cost defined in equation 5.4, which evaluates a single training sample x. To achieve this, we require a cost function, such as

$$C(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1-\hat{p}) & \text{if } y = 0. \end{cases} \tag{5.4}$$

The considered cost function is plotted in figure 5.2.

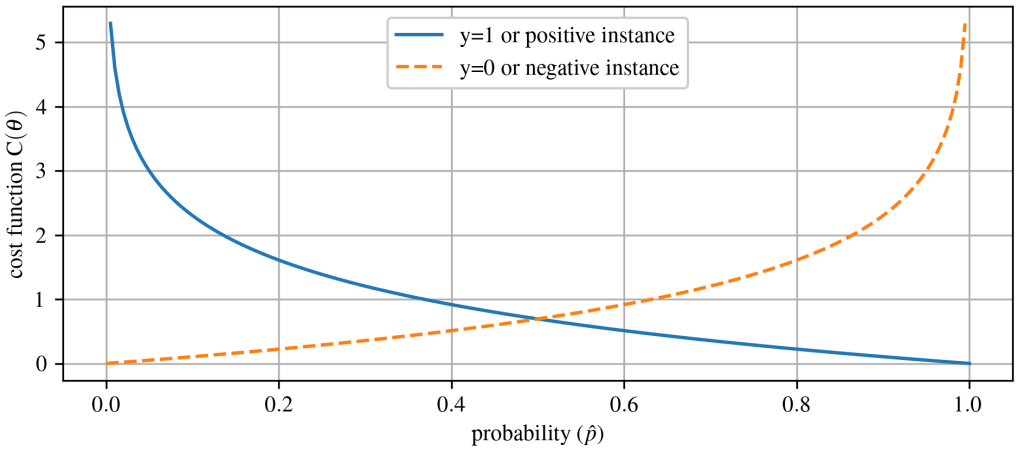


Figure 5.2: Cost function behavior for classification that heavily penalizes incorrect predictions.

The cost function in equation 5.4 behaves intuitively: $-\log(\hat{p})$ increases sharply as $\hat{p} \to 0$, so the loss is large when the model assigns a probability near0 to a positive example. It likewise produces a high loss when the model predicts a probability close to1 for a negative example. Conversely, because $-\log(\hat{p}) \to 0$ as $\hat{p} \to 1$, the loss becomes negligible when the predicted probability is near1 for a positive instance or near0 for a negative one, aligning with our expectations.

The overall cost, denoted $J(\theta)$, is the mean loss across all m training examples. This metric is commonly called the log-loss and written as

$$J(\theta);=;-\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right);+;\left(1-y^{(i)}\right)\log\left(1-\hat{p}^{(i)}\right)\right]. \tag{5.5}$$

Because there is no closed-form solution for the parameters θ (embedded within $\hat{p}$), the minimum must be found with an iterative optimizer. The cost surface is convex, so gradient descent or another suitable algorithm will reach the global minimum provided the learning rate is reasonable and enough iterations are allowed. The gradient of the cost with respect to the j^{th} parameter θ_j is given as

$$\frac{\partial J}{\partial \theta_j};=;\frac{1}{m}\sum_{i=1}^{m}\left[\sigma\left(\theta^\top X^{(i)}\right)-y^{(i)}\right],x_j^{(i)}. \tag{5.6}$$

This equation resembles the partial derivative used in gradient descent. For every training example, it finds the prediction error, multiplies it by the j^{th} feature value, and then averages these products over all m samples. With the resulting gradient vector of partial derivatives, you can update the parameters using the Batch Gradient Descent algorithm, completing the training of a Logistic Regression model. Stochastic Gradient Descent performs the update after each single example, while Mini-batch Gradient Descent updates the parameters after processing each mini-batch.

Review 5.1 Iris Flower Dataset

The Iris dataset, collected by botanist Edgar Anderson in 1935, lists sepal length, sepal width, petal length, and petal width for 150 flowers drawn equally from three species: Iris-Setosa, Iris-Versicolor, and Iris-Virginic (refer to Figure 5.3)a. Statistician Ronald Fisher analysed these measurements in a 1936 study on linear discriminant analysis, and the data have since become a standard benchmark in statistics and machine learning because they are small, clean, and perfectly balanced, making them ideal for testing classification algorithms and visualisation techniques.

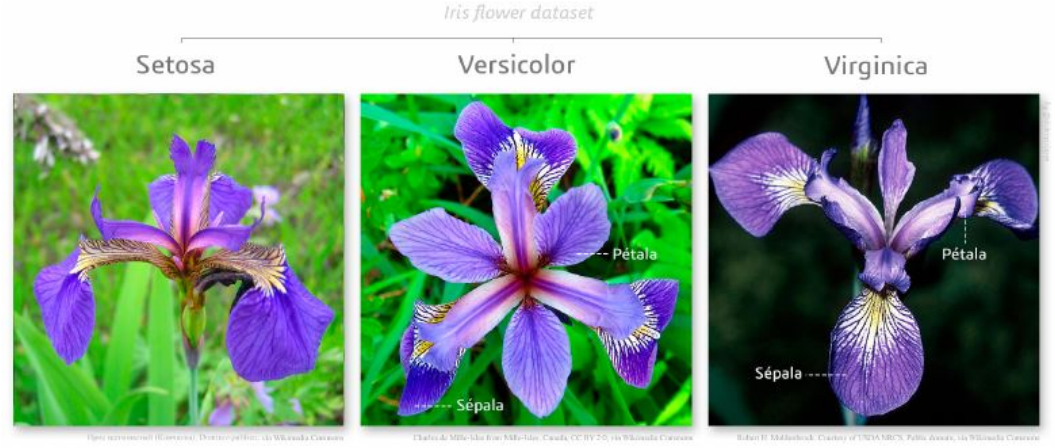


Figure 5.3: Flowers representing three species of iris plants [a]

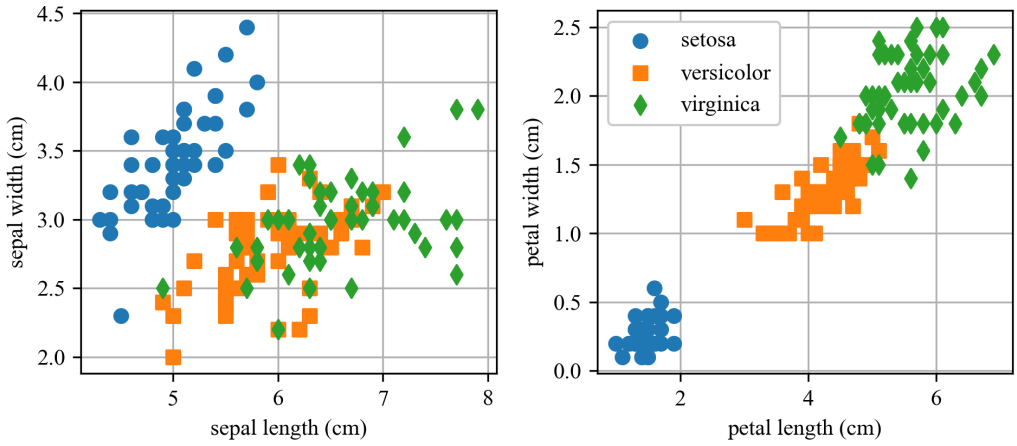


Figure 5.4: Iris dataset scatterplot showing sepal length vs. sepal width (left) and petal length vs. petal width (right).

[a]Diego Mariano, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

Example 5.1 Iris Dataset Exploration
This example introduces the Iris dataset (Figure 5.3), originally compiled by Ronald Fisher. It demonstrates how to load the dataset, access feature and label information, and visualize the relationships between sepal and petal measurements across the three iris species.

5.1.1 1-D Decision Boundaries

The petal width of Iris-virginica flowers usually lies between 1.4 cm and 2.5 cm, whereas the other two species range from 0.1 cm to 1.8 cm. Because these intervals overlap, the classifier's certainty changes across the scale. Above roughly 2 cm, the model is confident that any given sample is Iris-virginica. Moreover, below 1 cm it is equally sure the flower is not Iris-virginica (This means the model redicts high probability for the class "Not Iris-Virginica"). In between the limits of the classes, the model is unsure. When you call `predict()` instead of `predict_proba()`, it outputs the most likely class, creating a decision boundary around 1.6 cm where both class probabilities reach 50 percent. Thus, flowers with petal widths greater than 1.6 cm lead to a prediction of Iris-virginica, while flowers with smaller petal widths yield the opposite prediction even if confidence is low.

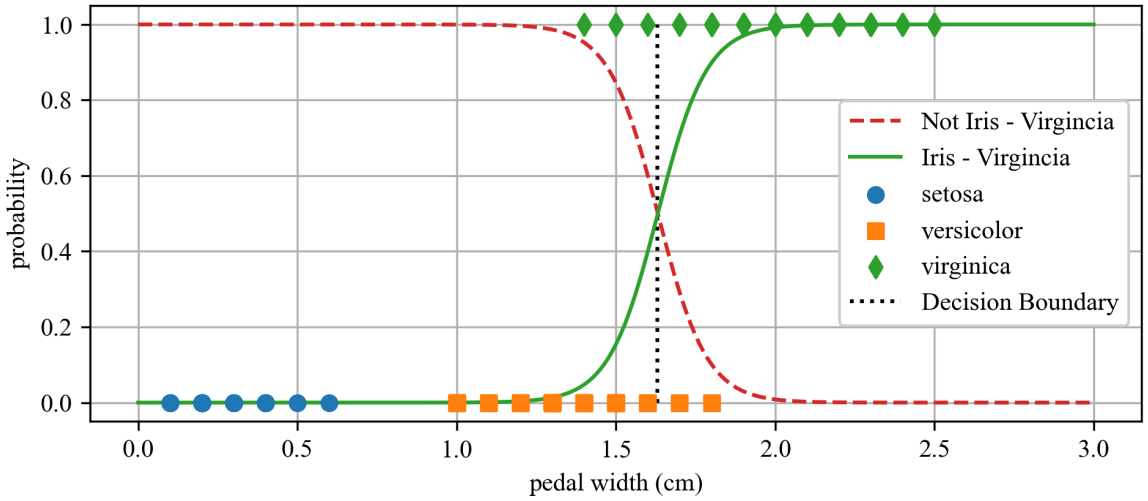


Figure 5.5: Decision boundary for the flowers of three Iris plant species with C set to $C = 10^{10}$.

Example 5.2 1D Logistic Regression
This example builds a logistic regression model to classify Iris-Virginica based on a single feature: petal width. It visualizes class probabilities and shows the decision boundary at the 50% threshold.

5.1.2 2-D Decision Boundaries

Figure 5.6 plots petal width against petal length for the Iris samples. After training, the logistic regression model assigns each point the probability that the flower is *Irisvirginica*. The dashed line marks where this probability equals50 %, forming the decision boundary, which is linear. The parallel contour lines show equal-probability levels from 10% to 90%. Points be beyond the top-right line have a greater than 90% probability of being classified as Iris-Virginica by the model.

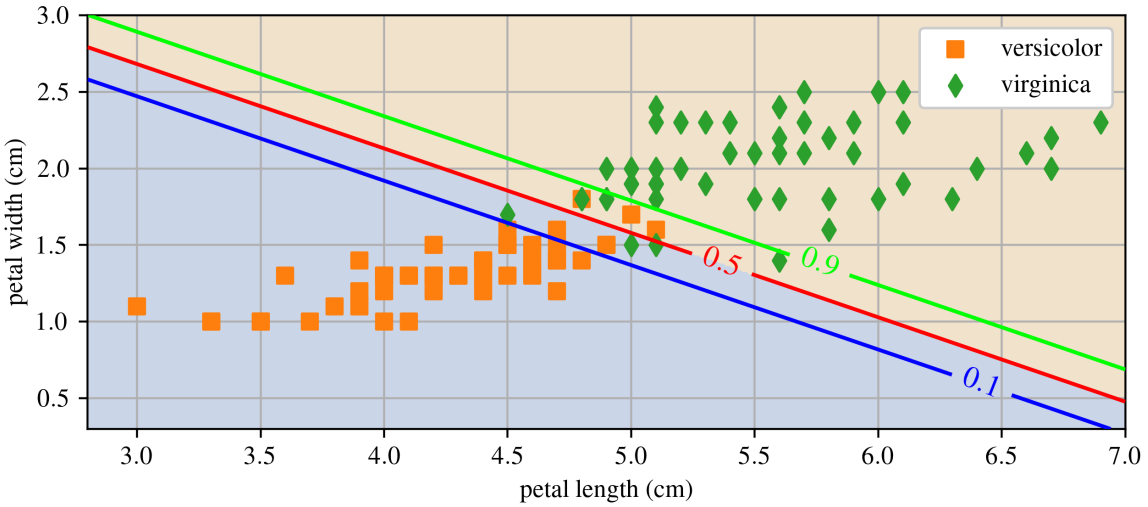


Figure 5.6: A 2D decision boundary for the Iris dataset.

Example 5.3 2D Decision Boundary
This example uses logistic regression to classify the Iris-Virginica species based on petal length and width. A 2D decision boundary is visualized in the petal feature space, with prediction regions and probability contours illustrating classifier confidence.

5.2 Softmax Regression

Logistic Regression can be generalised to handle many classes in a single model, so there is no need to train and merge several binary classifiers. This multiclass version is called Softmax Regression, or Multinomial Logistic Regression. Softmax Regression provides a straightforward way to perform regression-based classification when more than two classes are present. For an input vector $\mathbf{x}$, the model first computes a score $s_k(\mathbf{x})$ for every class k, then turns these scores into class probabilities with the softmax (normalised exponential) function. The score is obtained exactly as in linear regression:

$$s_k(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}^{(k)}. \tag{5.7}$$

Here each class has its own parameter vector $\theta^{(k)}$, and the collection of these vectors is usually stored as the rows of a parameter matrix Θ.

After the model has calculated a score for each class given an input $\mathbf{x}$, it converts these scores to probabilities with the softmax function. For class k the predicted probability is

$$\hat{p}_k = \sigma\big(s(\mathbf{x})\big)_k = \frac{\exp\big(s_k(\mathbf{x})\big)}{\sum_{j=1}^{K} \exp\big(s_j(\mathbf{x})\big)}. \tag{5.8}$$

In this expression, $\mathbf{s}(\mathbf{x})$ is the vector of class scores for the input, $\sigma\big(\mathbf{s}(\mathbf{x})\big)_k$ is the probability that $\mathbf{x}$ belongs to class k, and K is the total number of classes. In short, equation 5.8 exponentiates each score and then normalises the results by dividing by the sum of all exponentials so the probabilities sum to one.

Like logistic regression, a Softmax Regression model assigns an input to the class whose predicted probability is largest, which coincides with the class that has the highest score:

$$\hat{y} = \underset{k}{\operatorname{argmax}}\, \sigma\big(\mathbf{s}(\mathbf{x})\big)_k = \underset{k}{\operatorname{argmax}}\, s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\, \big((\theta^{(k)})^\top \cdot \mathbf{x}\big). \tag{5.9}$$

NOTE
The argmax operator returns the argument that maximises a function. In this setting, it yields the index k for which the estimated probability $\sigma\big(s(\mathbf{x})\big)_k$ attains its largest value.

With probability estimation and prediction established, we next consider training. The task is to learn parameters that place a large probability on the correct class and correspondingly small probabilities on all others. This objective is met by minimising the cost in equation 5.10, known as the cross entropy, which heavily penalises the model when it assigns a low probability to the true label. Cross entropy is widely used to measure how well predicted class probabilities agree with the actual classes. The cost function is

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)}). \tag{5.10}$$

NOTE

In this expression, $y_k^{(i)} = 1$ when the i^{th} sample's true label is class k, and $y_k^{(i)} = 0$ otherwise.

When considering only two classes ($K = 2$), it's important to highlight that this cost function aligns with the Logistic Regression's cost function, commonly referred to as log loss (refer to Equation 5.5).

The gradient of the cross-entropy cost with respect to $\theta^{(k)}$ is

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} (\hat{p}_k^{(i)} - y_k^{(i)} X^{(i)}). \tag{5.11}$$

By computing this vector for each class, you obtain the full gradient, which can then be fed to Gradient Descent or another optimizer to find the parameter matrix Θ that minimises the cost.

Applying Softmax Regression to the three-class iris problem in Scikit-Learn is straightforward. The `LogisticRegression` estimator normally uses one-versus-all when more than two classes are present, but setting `multi_class=''multinomial''` activates true Softmax learning. Choose a solver that supports this option, such as `lbfgs` (see the library documentation). The model applies ℓ_2 regularisation by default, governed by the hyperparameter C. After training, a flower with 5 cm long and 2 cm wide petals is classified as Iris-Virginica with probability 94.2%, while the probability of Iris versicolor is 5.8%.

NOTE

The Softmax Regression classifier is multiclass, not multioutput. As such, Softmax Regression can only predict one class at a time, so it works for problems where each input belongs to exactly one category-like classifying an email as spam, promotions, or updates. It can't be used for cases where multiple labels may apply, such as tagging a news article with topics like politics, economics, and technology all at once.

Figure 5.7 displays the decision regions, shaded with background colours to mark each class. The borders separating any two classes are straight lines. The plot also includes contour curves for the predicted probability of the Iris-Versicolor class. At the point where all three borders meet, every class receives a probability of 33%, so the selected class can have a confidence below 50%.

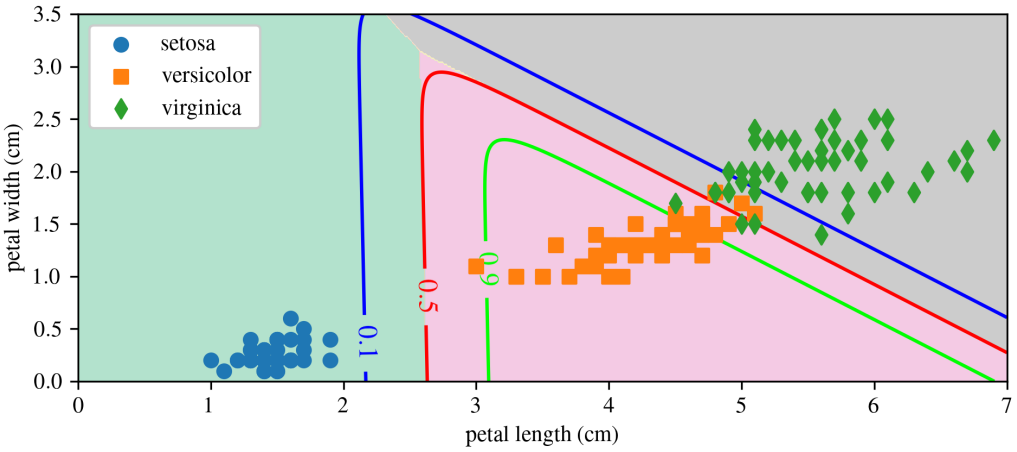


Figure 5.7: Softmax classification for the three iris plant species.

Example 5.4 Softmax Classification
This example applies softmax regression to classify all three iris species using petal length and width. It builds a multinomial logistic regression model and visualizes the decision boundaries along with confidence contours over the 2D petal feature space.

5.3 Examples

Example 5.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 5.1 Introduction to the IRIS data set
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import matplotlib.pyplot as plt
import sklearn as sk

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis

iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the pedals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o')
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s')
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d')
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])


plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0])
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1])
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2])
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()

```

Example 5.2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 5.2 1D Decision boundary for the Iris dataset
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk


cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis

iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the pedals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o')
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s')
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d')
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])


plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0])
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1])
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2])
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()


#%% Train a Logistic Regression model

# define the features (X) and the output (Y)
X_pedal = iris["data"][:, 3:] # consider just the petal width
y_pedal = iris["target"] == 2 # 1 if Iris-Virginica, else 0

# Build the logistic Regression model and train it.
log_reg = sk.linear_model.LogisticRegression( C=1)
log_reg.fit(X_pedal, y_pedal)
# Note: The hyper-parameter controlling the regularization strength of a
```

```
# Scikit-Learn LogisticRegression model is not alpha (as in other linear models),
# but its  inverse: C. The higher the value of C, the less the model is regularized.

# Build a range of the feature (X) to predict over. Here we just consider pedal width.
X_new = np.linspace(0, 3, 1000)
X_new = np.expand_dims(X_new, axis=1)

# Use the Logistic Regression Model to predict the pedal type based on pedal width
y_proba = log_reg.predict_proba(X_new)

# plot the probability plots
plt.figure(figsize=(6.5,3))
plt.grid(True)

# plot the data used for training, set at 0 and 1
plt.scatter(X[Y==0,3],np.zeros(50),marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,3],np.zeros(50),marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,3],np.ones(50),marker='d',label=Y_names[2],zorder=10)

# plot the probability
plt.plot(X_new,y_proba[:,0], '--',color=cc[3],label='Not Iris - Virgincia')
plt.plot(X_new,y_proba[:,1],color=cc[2], label='Iris - Virgincia')

# find and plot the 50% decision boundary
x_at_50 = X_new[np.argmin(np.abs(y_proba[:,0] - 0.5))]
plt.vlines(x_at_50,0,1,color='k',linestyles=':',label='Decision Boundary')

plt.xlabel('pedal width (cm)')
plt.ylabel('probability')
plt.legend(framealpha=1)
plt.tight_layout()

# make predictions on the model trained on the data.
log_reg.predict([[1.7]])
```

Example 5.3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 5.3 2D Decision boundary for the Iris dataset
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')


import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk


cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis

iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the petals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o',zorder=10)
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s',zorder=10)
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d',zorder=10)
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])


plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()


#%% plot the Linear decision boundary in 2D "Petal" space

# build the training and target set.
X_train = X[:, (2, 3)]  # petal length, petal width
y_train = Y == 2

# build the Logistic Regression model
log_reg = sk.linear_model.LogisticRegression(C=10**10)
# Note: The hyper-parameter controlling the regularization strength of a Scikit-Learn
```

```
# LogisticRegression model is not alpha (as in other linear models), but its
# inverse: C. The higher the value of C, the less the model is regularized.

# train the Logistic Regression model
log_reg.fit(X_train, y_train)

# build the x values for the predictions over the entire "petal space"
x_grid, y_grid = np.meshgrid(
        np.linspace(2.8, 7, 500),
        np.linspace(0.3, 3, 200),
    )
X_new = np.vstack((x_grid.reshape(-1), y_grid.reshape(-1))).T # build a vector format of
the mesh grid

# predict on the vectorized format
y_predict = log_reg.predict(X_new)
y_proba = log_reg.predict_proba(X_new)

# convert back to meshgrid shape for plotting
zz_predict = y_predict.reshape(x_grid.shape)
zz_proba = y_proba[:, 1].reshape(x_grid.shape)

# plot the 2D "petal space"
plt.figure(figsize=(6.5,3))
plt.grid(True)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',color=cc[1],label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',color=cc[2],label=Y_names[2],zorder=10)
plt.contourf(x_grid, y_grid, zz_predict, cmap='Pastel2')
contour = plt.contour(x_grid, y_grid, zz_proba, [0.100,0.5,0.900],cmap=plt.cm.brg)
plt.clabel(contour, inline=1) # add the labels to the plot
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend()
plt.tight_layout()
```

Example 5.4

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 5.4 Softmax decision boundary for the Iris dataset
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().magic('reset -sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk


cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis
iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the petals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o',zorder=10)
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s',zorder=10)
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d',zorder=10)
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])

plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()


#%% Softmax Regression

# build the training and target set.
X_train = X[:, (2, 3)]  # petal length, petal width
y_train = Y

# build and train the softmax model
softmax_reg = sk.linear_model.LogisticRegression(multi_class="multinomial",
        solver="lbfgs", C=10)
softmax_reg.fit(X_train, y_train)

# build the x values for the predictions over the entire "petal space"
```

```
x_grid, y_grid = np.meshgrid(
        np.linspace(0, 7, 500),
        np.linspace(0, 4, 200),
    )
X_new = np.vstack((x_grid.reshape(-1), y_grid.reshape(-1))).T # build a vector format of the mesh grid

# predict on the vectorized format
y_predict = softmax_reg.predict(X_new)
y_proba = softmax_reg.predict_proba(X_new)


# convert back to meshgrid shape for plotting
zz_predict = y_predict.reshape(x_grid.shape)
zz_proba = y_proba[:, 1].reshape(x_grid.shape) # the selected column selects the probability that the data falls within this class.

# plot the 2D "petal space"
plt.figure(figsize=(6.5, 4))
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.contourf(x_grid, y_grid, zz_predict, cmap='Pastel2')
contour = plt.contour(x_grid, y_grid, zz_proba, [0.100,0.5,0.900], cmap=plt.cm.brg)
plt.clabel(contour, inline=1)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend()
plt.tight_layout()
```

6 Decision Trees

Decision trees are flexible learners that work well for classification, regression, and multi-output problems. A single tree can capture intricate relationships in the data with minimal preprocessing. Each tree also acts as the core building block of ensemble methods such as Random Forests, which are among the most reliable models in modern practice. One limitation is size: a fully grown tree can become quite large, so pruning or depth limits are often applied to control memory use and reduce overfitting.

In this chapter, we will explore the essentials of Decision Trees, starting with their training, visualization, and prediction processes. We will then look into the CART (Classification and Regression Trees) training algorithm, which Scikit-Learn utilizes for constructing Decision Trees. Additionally, we will examine how to regulate the complexity of Decision Trees and adapt them for regression tasks. The chapter concludes by addressing some inherent limitations of Decision Trees.

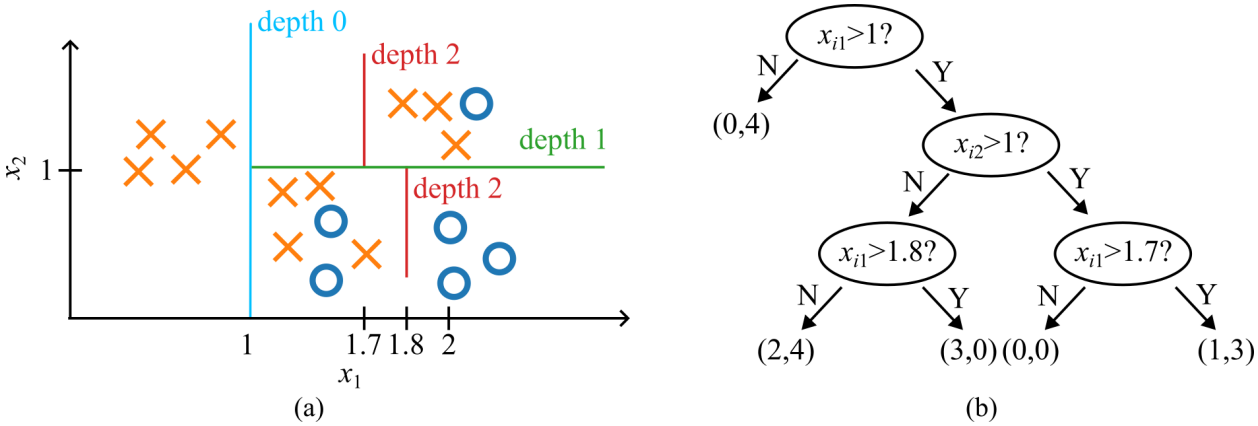


Figure 6.1: The basic connect of a decision tree, showing (a) how a decision tree is built, and (b) the developed decision tree.

6.1 Decision Tree Classification

Figure 6.2 shows the regions formed by the decision tree. The solid vertical line marks the root split at depth 0, where petal length equals 2.45 cm. All samples on the left of this line belong solely to Iris setosa, so no further division is needed there. The right half still mixes species, so the depth-1 right child splits again at petal width 1.75 cm, indicated by the dashed line. With `max_depth=2` the tree stops at this point. If `max_depth` were increased to 3, the two depth-2 leaves would introduce additional boundaries, drawn as dotted lines.

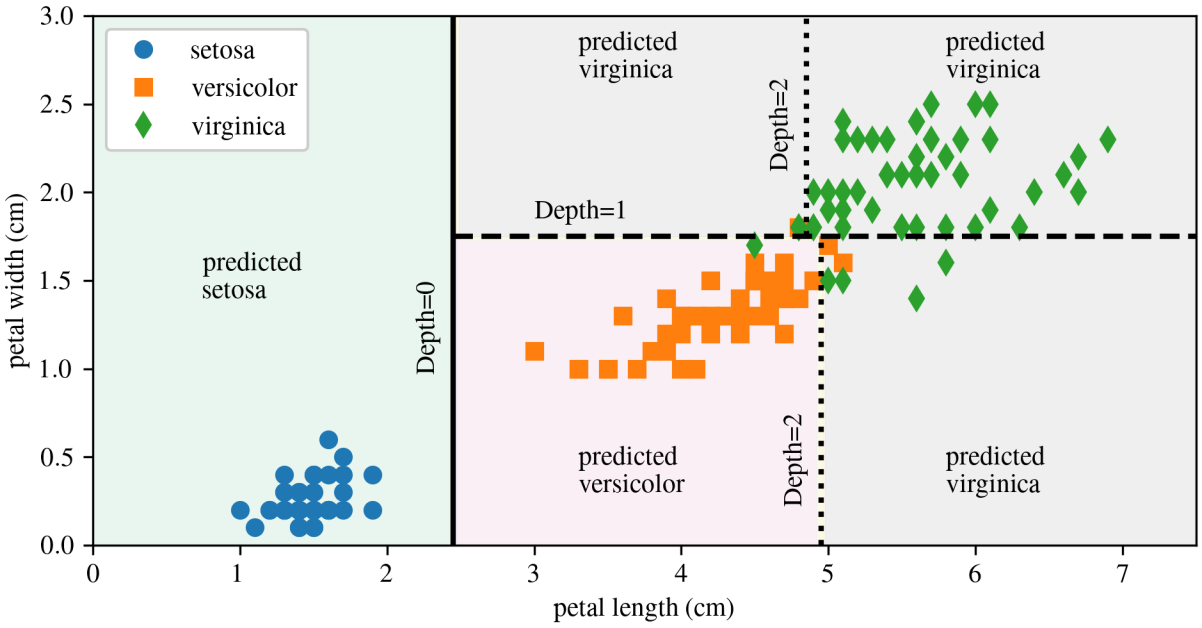


Figure 6.2: Decision Tree decision boundaries.

A decision tree for the Iris Dataset looks like Figure 6.3. Let us examine how the Decision Tree processes predictions. Suppose you come across an iris and need to classify it. Start at the root node (depth=0), which asks whether the flower's petal length is less than 2.45 cm. If the answer is yes, move to the root's left child (depth=1). This child is a leaf, so no further questions follow. The class stored in that leaf is Iris-setosa, and the tree therefore labels the sample as setosa.

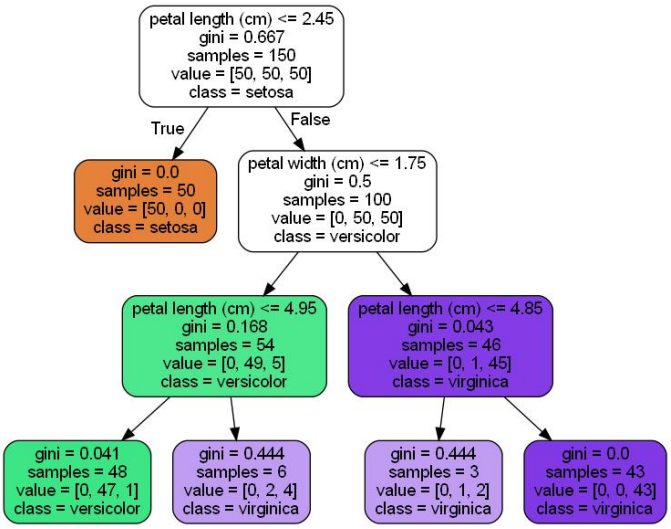


Figure 6.3: Decision tree for the Iris Dataset trained using the CART algorithms.

In Figure 6.3, each node reports the split criterion, Gini impurity, sample count, and class distribution. Fill colors denote the predicted species; orange for Setosa, green for Versicolor, and

purple for Virginica. Shade intensity conveys confidence: darker hues indicate purer (more certain) leaves, whereas white represents complete uncertainty. Consider a second iris whose petal length is greater than 2.45 cm. From the root you move to its right child (depth 1). This internal node asks a new question: is the petal width less than 1.75 cm? If yes, the flower is classified as Iris-versicolor (depth=2, left leaf). If no, it is labeled Iris-virginica (depth=2, right leaf). The decision path is clear and easy to follow.

The attributes of a node include:

- *class*: the class label stored in the leaf.
- *samples*: the number of training instances that reach the node. For example, 100 flowers have petal length >2.45 cm (depth1, right); among them, 54 also have petal width $<$ 1.75 cm (depth2, left).
- *value*: a three-element vector giving the count of instances from each species at the node. The bottom-right leaf, for instance, contains 0 Iris-setosa, 1 Iris-versicolor, and 45 Iris-virginica.
- *gini*: the Gini impurity. A node is *pure* when $gini = 0$, meaning all samples belong to the same class. The depth-1 left node, which holds only Iris-setosa, is pure.

The Gini impurity G_i for the i^{th} node is computed using

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2, \tag{6.1}$$

where $p_{i,k}$ is the proportion of class k instances among the training instances at the i^{th} node. For instance, the Gini score for the depth-2 left node is calculated as follows: $1-(0/54)^2-(49/54)^2-(5/54)^2 \approx 0.168$. We will discuss an alternative impurity measure later.

6.1.1 Class Probability Estimation

A Decision Tree can estimate the likelihood that a sample belongs to class k. The algorithm traces the sample's path to its leaf node and then reports the fraction of training instances of class k found in that leaf. Consider a flower with petals 5cm long and 1.5cm wide. This sample reaches the depth-2 left leaf, which contains 54 training flowers: 0 *Iris*-setosa, 49 *Iris*-versicolor, and 5 *Iris*-virginica. The tree therefore assigns probabilities of 0%, 90.7%, and 9.3% to the three classes, respectively. When asked for a class label, it selects *Iris*-versicolor, the class with the highest probability.

Interestingly, the same estimated probabilities apply throughout the bottom-right rectangle of Figure 6.2, even in a scenario where the petal dimensions are 6 cm by 1.5 cm where one might intuitively expect an Iris-Virginica classification.

Example 6.1 Decision Tree Classifier
This example uses Scikit-Learn's `DecisionTreeClassifier` to classify iris species based on petal features. The decision process is visualized using Graphviz, showing the structure of learned rules and how decisions are made based on feature thresholds. This online viewer is the easiest way to do that https://dreampuf.github.io/GraphvizOnline/

6.2 The CART Training Algorithm

Scikit-Learn trains decision trees with the Classification and Regression Tree (CART) algorithm[a], a process often called "growing" the tree. At each node the algorithm splits the local training set into two parts by selecting a feature k and a threshold t_k, for example "petal length ≤ 2.45 cm". It evaluates every candidate pair (k, t_k) and chooses the one that yields the purest child nodes, judging purity by a weighted combination of each child's impurity measure and sample count.

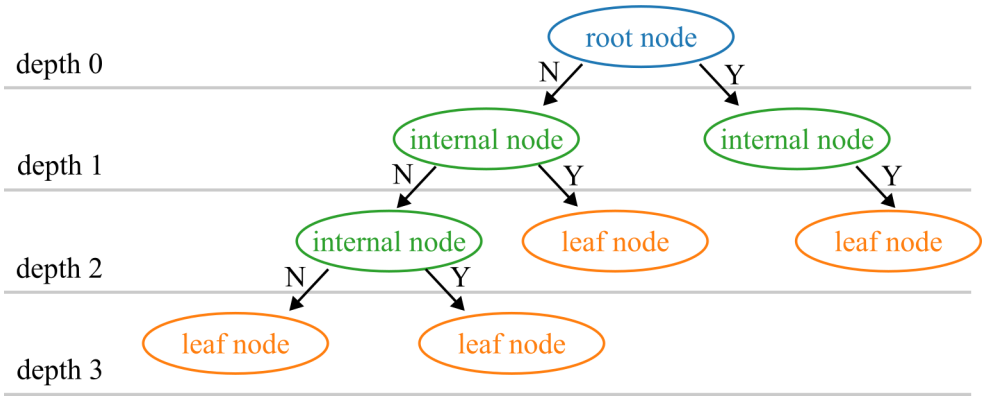


Figure 6.4: Illustration of the CART algorithm process.

The objective function it aims to minimize during this process is

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}. \tag{6.2}$$

In this equation, $G_{\text{left/right}}$ measures the impurity of the left/right subset, and $m_{\text{left/right}}$ is the number of instances in the left/right subset. After the initial split, the algorithm continues to divide the resulting subsets and their subsequent divisions recursively. This recursive process halts when it reaches the predefined maximum depth (`max_depth`) set through a hyperparameter, or if no further impurity-reducing splits can be found.

Unlike linear models that assume a specific data structure, Decision Trees impose few assumptions on their initial data. If not appropriately constrained, a Decision Tree can intricately conform to the training data, leading to overfitting. This model type is described as nonparametric, not due to a lack of parameters, which it can have in abundance, but because the parameters are not predetermined before training, allowing the model's structure to freely mirror the data intricacies. Conversely, a parametric model, like a linear model, has a fixed number of parameters, limiting its flexibility but also minimizing overfitting risks while potentially increasing underfitting risks.

[a]Breiman, Leo. Classification and regression trees. Routledge, 2017

To mitigate overfitting in Decision Trees, it is essential to control the model's freedom during training through regularization. The regularization hyperparameters vary by the algorithm, but typically, the tree's maximum depth can be restricted. In Scikit-Learn, this is managed by the `max_depth` hyperparameter, which is unlimited by default. Lowering `max_depth` helps regularize the model, thereby reducing overfitting likelihood.

Other parameters in Scikit-Learn's DecisionTreeClassifier also influence the tree's structure:

- `min_samples_split`: The minimum number of samples required to split a node.
- `min_samples_leaf`: The minimum number of samples a leaf node must have.
- `min_weight_fraction_leaf`: Similar to `min_samples_leaf`, but expressed as a fraction of total weighted instances.
- `max_leaf_nodes`: The maximum number of leaf nodes.
- `max_features`: The maximum number of features evaluated for splitting at each node.

In general, Decision Trees can be regularized by restricting how freely they grow. This is done by increasing hyperparameters that set minimum requirements, such as `min_samples_split` or `min_samples_leaf`, and by decreasing hyperparameters that set maximum limits, such as `max_depth`, `max_leaf_nodes`, or `max_features`.

Figure 6.5 illustrates two Decision Trees trained on the moons dataset: one on the left with default hyperparameters (unrestricted) and another on the right with `min_samples_leaf=4`. The left model appears to be overfitting, whereas the right model, with its restrictions, likely offers better generalization.

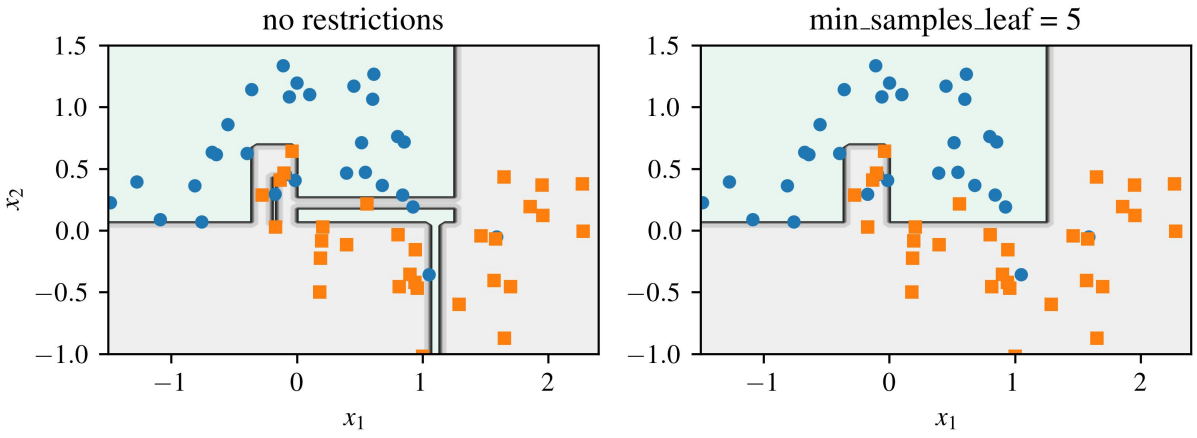


Figure 6.5: Regularization effects using `min_samples_leaf`

6.2.1 Computational Complexity

To predict with a Decision Tree the algorithm follows a path from the root to a leaf. Because most trained trees are roughly balanced, this path crosses about

$$T_{\text{predict}} = O\big(\log_2 m\big), \tag{6.3}$$

nodes, where m is the number of training samples. Each node tests a single feature, so the prediction cost in equation 6.3 is unaffected by the total feature count and is therefore very fast.

Training is costlier. At every split the algorithm scans each candidate feature (or the subset limited by `max_features`) over all samples that reach the node. For n input features the total work across all levels is

$$T_{\text{train}} = O\big(n \times m \log m\big), \tag{6.4}$$

because the tree has roughly $\log_2 m$ layers and each layer processes a shrinking fraction of the m samples.

When the dataset contains only a few thousand samples, enabling `presort=True` in Scikit-Learn can shorten training time, but for larger datasets the presorting step turns into a bottleneck and the standard (unsorted) approach is quicker.

6.2.2 Entropy verse Gini Impurity

Decision Trees in sklearn use the Gini impurity as the default criterion for node purity, entropy can also be utilized by setting the criterion hyperparameter to “entropy”. Originally a thermodynamic concept representing molecular disorder, entropy reaches zero when molecules are in a complete state of rest and order. This concept was later adopted in information theory, introduced by Shannon, to quantify the average informational content in messages entropy is zero when all messages are identical. In the realm of Machine Learning, entropy serves as a measure of impurity: it is zero in a dataset if all its elements belong to a single class. The entropy for the i^{th} node is defined as

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2(p_{i,k}). \tag{6.5}$$

The choice between Gini impurity and entropy often results in negligible differences, producing similar trees. Gini impurity has a slight computational advantage and is thus the default choice. However, it tends to separate the most frequent class into a distinct branch, whereas entropy generally yields more balanced trees.

6.3 Decision Tree Regression

Decision Trees are not limited to classification tasks; they can also be adapted for regression. To illustrate, we use Scikit-Learn’s `DecisionTreeRegressor` to train a regression tree on a noisy linear dataset with `max_depth=2`. The structure of this tree is displayed in Figure 6.6. In contract to the color scheme for classification, the change in color shades here relate to the predicted value of y.

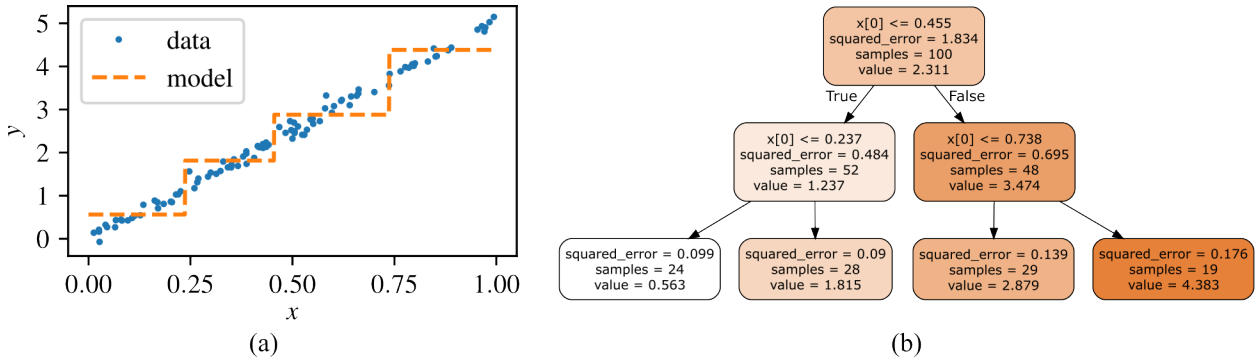


Figure 6.6: A regression model developed using a Decision Tree, showing the: (a) model superimposed over noisy data, and (b) the decision tree developed for the task.

The workings of a regression tree mirror those of a classification tree, except that each leaf holds a continuous prediction instead of a class label. To obtain a prediction for a sample with $x_1 = 0.6$, trace the path from the root to its leaf; that leaf outputs 0.1106, which is the average target value among the 110 training instances that reach it. For those instances the mean squared error is 0.0151.

The predictions from this model are visualized in Figure 6.7, with results shown for trees of depth 2 and 3. The deeper tree partitions the input space into more regions, with each region's predicted value being the average target value of the instances it encompasses. The model attempts to organize the regions such that the instances within each are as close as possible to their predicted value.

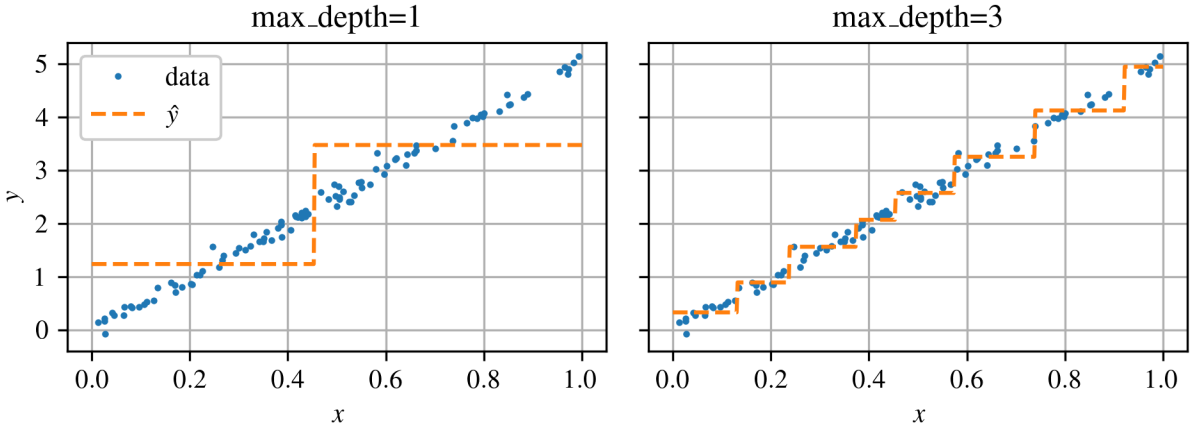


Figure 6.7: Comparison of predictions from two Decision Tree regression models with varying depths.

The CART algorithm for regression trees aims to minimize the MSE when splitting the training set, similar to how it minimizes impurity in classification tasks. The cost function minimized by the algorithm is represented as

$$J(k,t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}}. \tag{6.6}$$

Knowing that,

$$\text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \tag{6.7}$$

and

$$\hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)}. \tag{6.8}$$

Similar to classification, Decision Trees for regression can overfit if not properly regularized. Without regularization, the predictions, as depicted on the left of Figure 6.8, can fit the training data excessively. By setting `min_samples_leaf` to 15, a more generalized model is achieved, as shown on the right in the same figure.

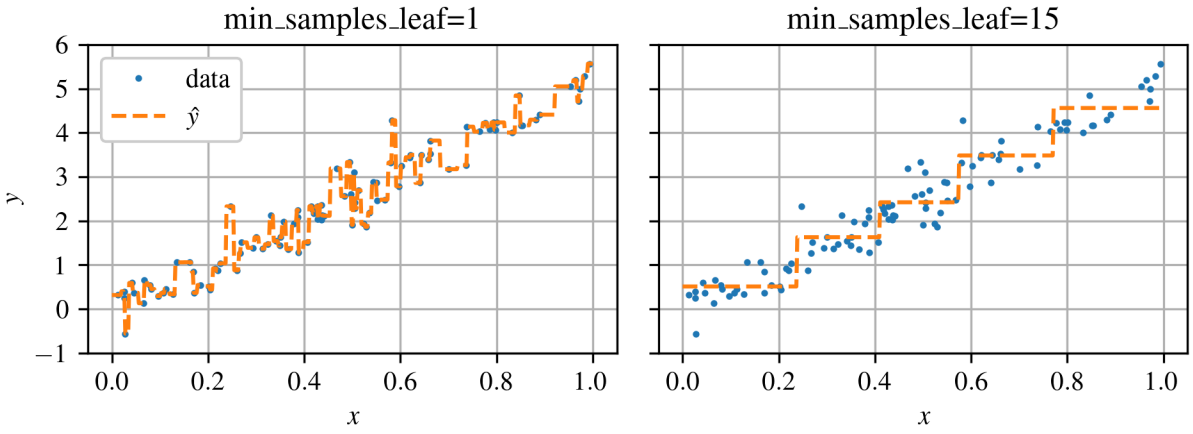


Figure 6.8: Impact of regularization through setting the minimum number of samples in a leaf on a Decision Tree regression model.

Example 6.2 Decision Tree Regression
This example uses Scikit-Learn's `DecisionTreeRegressor` to fit a non-linear relationship between input and output data. A tree of depth 4 is trained on noisy linear data and visualized using Graphviz to show how the regression model partitions the input space.

6.4 Random Forest

If one Decision Tree is useful, a natural question is whether many trees can be combined to produce better models. Random Forests answer this question by averaging the predictions of many decorrelated trees.

6.4.1 Instability of Individual Trees

While Decision Trees offer simplicity, interpretability, versatility, and power, they come with certain drawbacks. A notable limitation is their preference for orthogonal decision boundaries, which makes them highly sensitive to the orientation of the data. For instance, Figure 6.9 illustrates how a simple linearly separable dataset can be perfectly split by a Decision Tree in its original alignment, whereas a 45° rotation results in a convoluted decision boundary. Despite perfect fits to their respective training sets, the rotated tree's model is likely to perform poorly on unseen data. Utilizing Principal Component Analysis (PCA) can often mitigate this issue by reorienting the data more suitably.

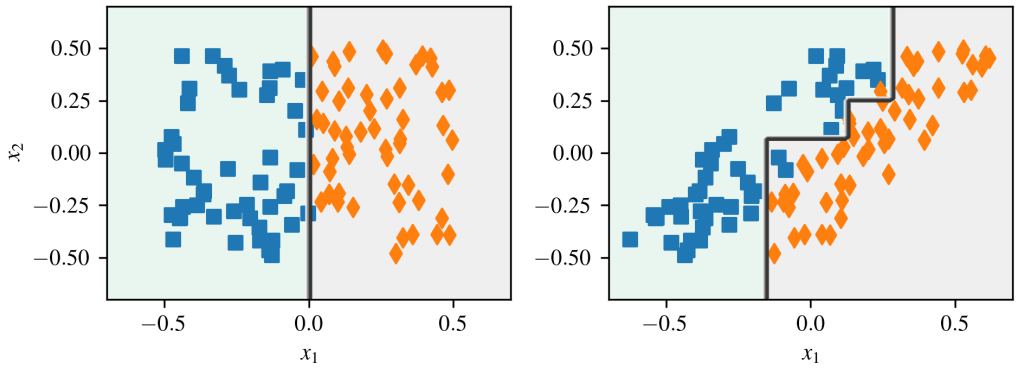


Figure 6.9: Sensitivity to training set rotation. Decision trees create a single clean split on the original data (left), but after a 45° rotation they must build a jagged, multi-step boundary; illustrating their sensitivity to feature orientation.

More broadly, Decision Trees are sensitive to small variations in training data. For instance, changing the seed of the random number generator can lead to a substantially different model, as shown in Figure 6.10. This variability is partly due to the stochastic nature of the Scikit-Learn's training algorithms; different models may result from the same data unless the `random_state` hyperparameter is fixed.

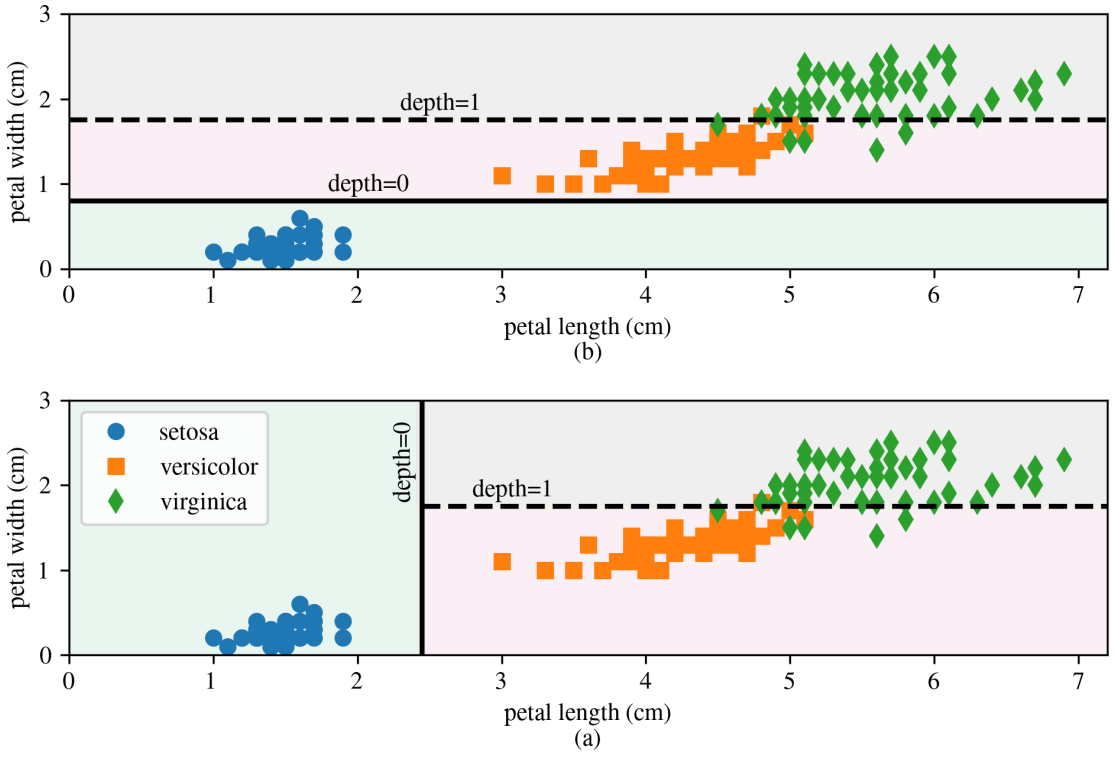


Figure 6.10: Decision Tree Sensitivity to initial conditions, showing: (a) random number generator seeded with "1", and: (b) random number generator seeded with "2". [a]

[a] "2" is the author's preferred random number seed.

6.4.2 Ensembling Decision Trees

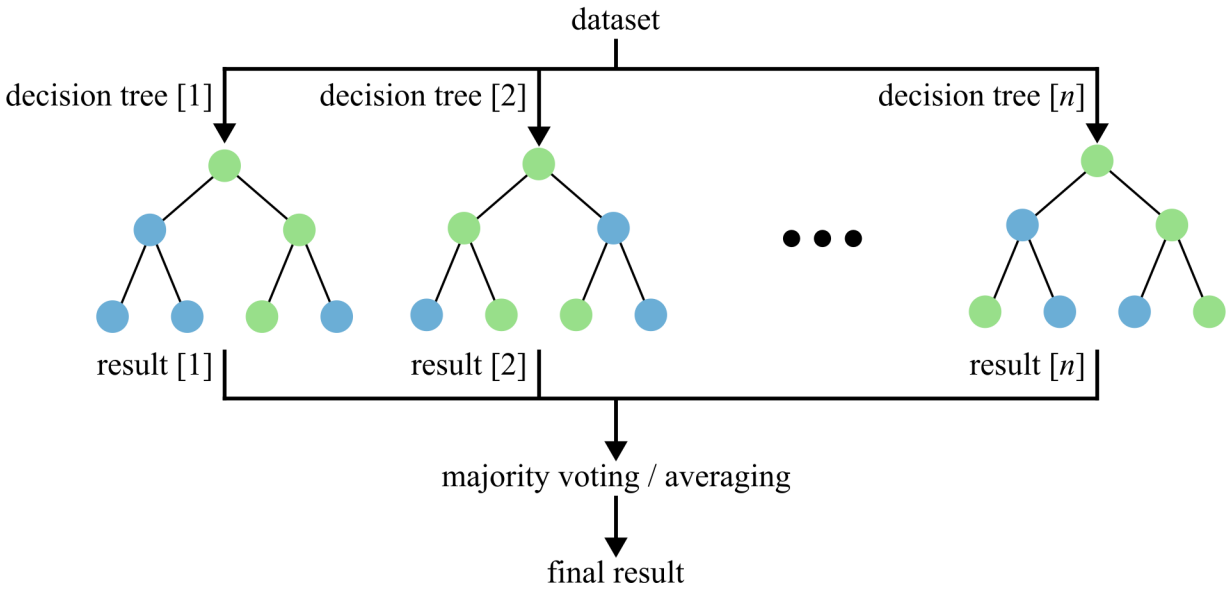


Figure 6.11: Random forest with majority voting.

Random Forests reduce the high variance of individual decision trees by building many trees on bootstrap samples of the training data and then averaging their outputs. Each tree is grown with a random subset of input features, so the ensemble decorrelates the trees and improves generalisation. For classification problems the forest predicts the class that receives the majority vote, while for regression it returns the mean of the trees' numeric predictions. This bagging strategy limits overfitting and usually yields better performance than a single tree, although it often falls short of the accuracy achieved by gradient-boosted ensembles. Model quality still depends on the data's size, noise level, and feature interactions, and on hyperparameters such as the number of trees, the maximum depth, and the number of features considered at each split.

6.5 Examples

Example 6.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 6.1 Decision Tree Classifier
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn.datasets import load_iris
from sklearn.tree import export_graphviz
import graphviz as graphviz

plt.close('all')


#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis
iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the petals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

#%% Build the model

# train the decision tree
tree_clf = sk.tree.DecisionTreeClassifier(max_depth=3)
X_petal = X[:,2:]
tree_clf.fit(X_petal, Y)


#%% Visualize the decision tree

#create the export file for graphviz and export it. The file is exported as a
#.DOT file and can be viewed in an online viewer
https://dreampuf.github.io/GraphvizOnline/
export_graphviz(
        tree_clf,
        out_file="tree_clf.dot",
        feature_names=feature_names[2:],
        class_names=Y_names,
        rounded=True,
        filled=True
    )

# We can load the file back in
s = graphviz.Source.from_file('tree_clf.dot')

# look at what is inside it. Also, just typing s in the console will diplay the image
print(s)

# export the image to a jpg
s.render('tree_clf', format='jpg',view=True)

#%% Predict the class for any petal size
```

```
66
67  size = [[7, 2.5]]
68  print(tree_clf.predict_proba(size))
69  print(iris.target_names)
70
71
72  # plot the new data point over the Iris dataset
73  plt.figure()
74  plt.grid(True)
75  plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=2)
76  plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=2)
77  plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=2)
78  plt.scatter(size[0][0],size[0][1],s=300,marker='*',label='new data point',zorder=2)
79  plt.xlabel(feature_names[2])
80  plt.ylabel(feature_names[3])
81  plt.legend(framealpha=1)
82  plt.tight_layout()
83
84
85
86
87
```

Example 6.2

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 6.2 Decision Tree Regression
Machine Learning for Engineering Problem Solving
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
import graphviz as graphviz
from sklearn.tree import export_graphviz

plt.close('all')


#%% Train and plot a decision tree regression model

# build the data
m = 200
X = np.random.rand(m, 1)
y = 5 * X
y = y + np.random.randn(m, 1) / 10

# train the model
tree_reg = sk.tree.DecisionTreeRegressor(max_depth=3)
tree_reg.fit(X, y)

x_model = np.linspace(0, 1, 100).reshape(-1, 1)
y_model = tree_reg.predict(x_model)

plt.figure()
plt.plot(X, y, ".",label='data')
plt.plot(x_model, y_model, "-", label="model")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.legend()

#%% Plot the regression tree

# create the export file for graphviz and export it. The file is exported as a
# .DOT file and can be viewed in an online viewer
https://dreampuf.github.io/GraphvizOnline/
export_graphviz(
        tree_reg,
        out_file="tree_reg",
        rounded=True,
        filled=True
    )

# We can load the file back in
s = graphviz.Source.from_file('tree_reg')
s.render('tree_reg', format='jpg',view=True)
```

7 Support Vector Machines

The core idea of Support Vector Machines becomes clear when viewed geometrically. Figure 7.1 shows a slice of the iris dataset containing two linearly separable species. In the left panel two conventional linear classifiers are drawn; although both label every training point correctly, their separating lines sit very close to several observations, so even a small perturbation could lead to misclassifications on new data. The right panel plots the boundary produced by an SVM. The solid line still splits the classes cleanly, but it is positioned to maximise the distance to the nearest points, leaving the widest possible "street" (bounded by the dashed parallels). The samples that lie on these margins are the *support vectors*. This strategy, called large margin classification, usually delivers models that generalise better than those that merely separate the training set.

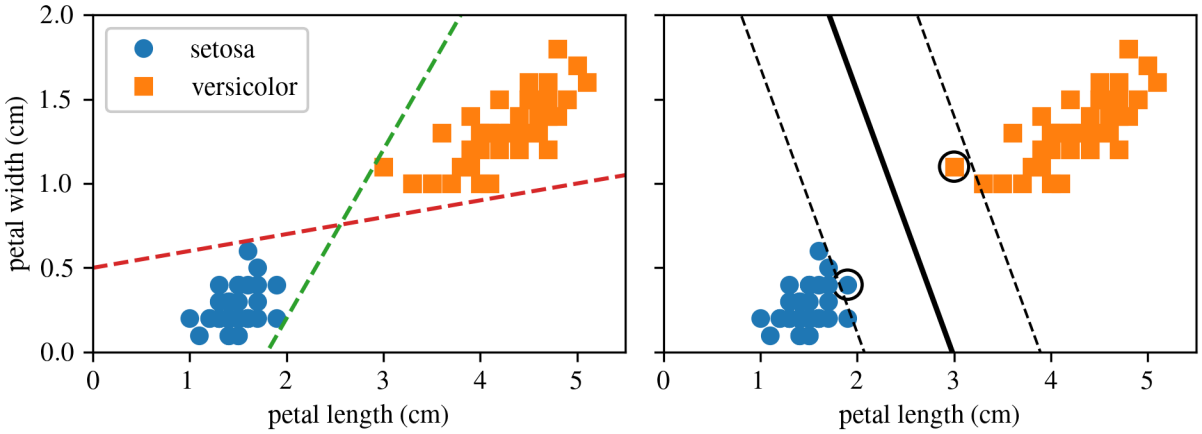


Figure 7.1: Large margin classification.

Observe that the addition of further training instances outside the "street" does not influence the decision boundary; it is entirely shaped by the instances situated on the boundary's edge. These pivotal instances are termed support vectors and are highlighted with circles in Figure 7.1.

NOTE

The sensitivity of SVMs to feature scales is evident in Figure 7.2. In the left plot, the vertical dimension greatly outweighs the horizontal dimension, resulting in a nearly horizontal "street." However, after applying feature scaling such as using Scikit-Learn's `StandardScaler` the decision boundary becomes more appropriate, as illustrated in the right plot.

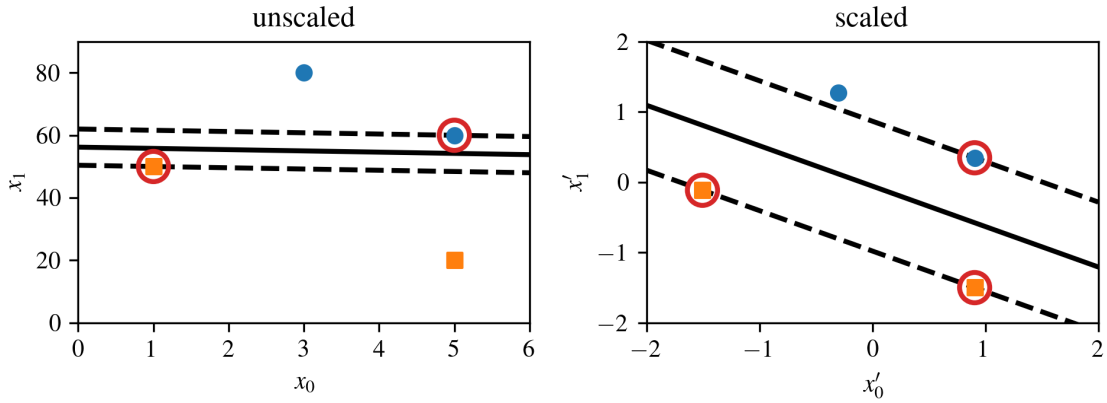


Figure 7.2: Sensitivity of support vectors to feature scaling.

7.1 Linear SVM Classification

Hard margin classification demands that all instances be correctly classified without any margin violations. This strict approach faces two significant challenges:

- It is only feasible when the data is linearly separable.
- It is highly sensitive to outliers.

Figure 7.3 illustrates these challenges using the iris dataset with an added outlier. On the left, achieving a hard margin is impossible due to the outlier. On the right, although a decision boundary is found, it deviates substantially from the optimal boundary shown in Figure 7.1 and is less likely to perform well on new data.

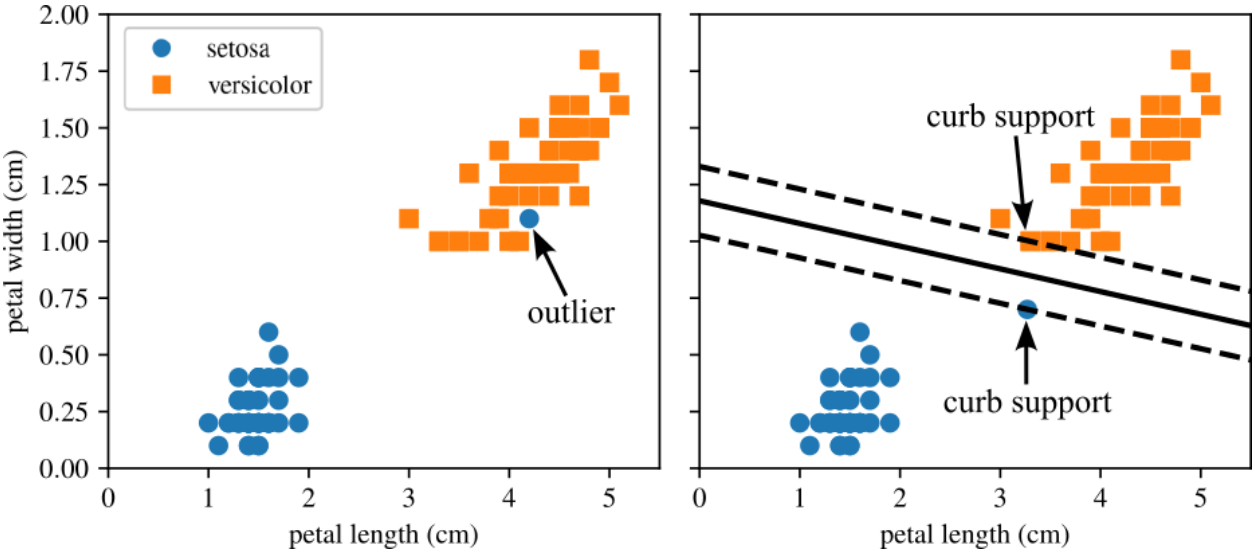


Figure 7.3: Support vector machines showing (left) an un-separable case, and (right) a separable case with two data points supporting the curbs of the support vector machine.

To mitigate the limitations of hard margin classification, a more adaptable model, known as soft margin classification, is often employed. The goal here is to achieve an optimal balance between maximizing the margin width and minimizing margin violations, where instances might fall into the margin or on the incorrect side.

Scikit-Learn's SVM implementations facilitate this balance through the hyperparameter C. A smaller value of C results in a wider margin but allows more margin violations, which is beneficial for model flexibility. Conversely, a larger C value tightens the margin, reducing margin violations but at the risk of a less flexible model. Figure 7.4 demonstrates this trade-off: the left plot with a low C value

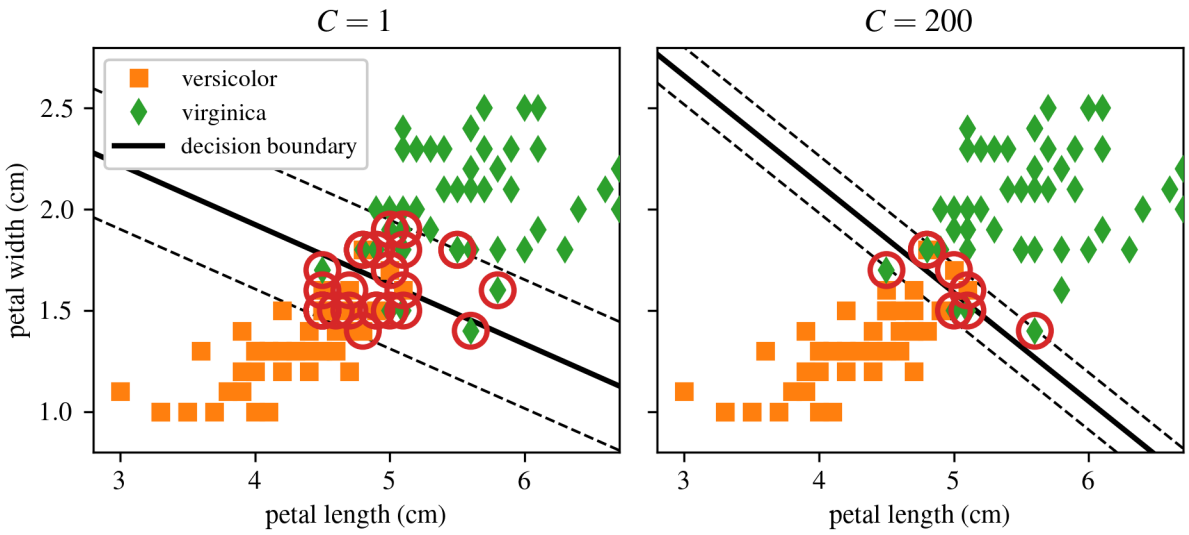


Figure 7.4: SVM margin sizes for different C values.

NOTE

Overfitting in an SVM model can often be addressed by reducing the C value, which increases regularization.

In earlier chapters we placed every model parameter in a single vector θ: the first entry θ_0 acted as the bias, while $\theta_1, \ldots, \theta_n$ were the feature weights, and each input was augmented with a constant bias feature $x_0 = 1$. In this chapter we adopt the notation most common for SVMs. The bias is written as b, the weight vector as $\mathbf{w}$, and no extra bias feature is appended to the input vectors.

7.1.1 Decision Function and Predictions

A linear SVM classifier determines the class of a new instance x by calculating the decision function

$$\mathbf{w}^\top \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b. \tag{7.1}$$

If the outcome is positive, the predicted class $\hat{y}$ is the positive class (1); otherwise, it is the negative class (0). This is written as

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^\top \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0. \end{cases} \tag{7.2}$$

Figure 7.5 plots the decision function for a model with two features, so the surface is a plane in $\mathbb{R}^2$. The thick solid line marks the decision boundary where the function equals zero. The dashed lines show the loci where the function equals 1 and -1; they run parallel to the boundary and sit at equal distance from it, outlining the margin. Training a linear SVM adjusts $\mathbf{w}$ and b to make this margin as wide as possible while either prohibiting margin violations (hard margin) or keeping them small through a penalty term (soft margin).

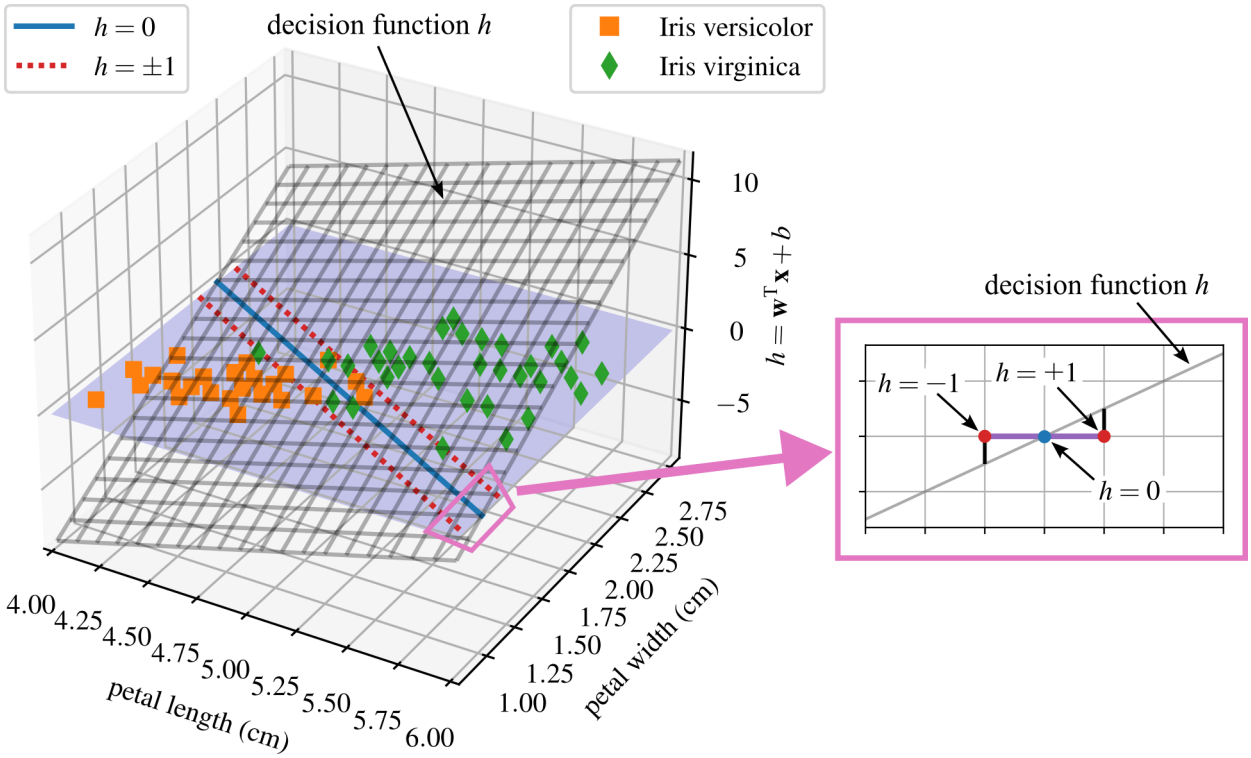


Figure 7.5: Decision function for the Iris Dataset showing how the decision function h cuts through the feature space.

7.1.2 Training Objective

The slope of the decision function corresponds to the norm of the weight vector, $\|\mathbf{w}\|$. Halving this slope causes the decision boundary margins, where the decision function equals ± 1, to double in distance from the decision boundary. Effectively, reducing the norm of $\mathbf{w}$ by half doubles the margin. This geometric interpretation is perhaps simpler to visualize in two dimensions, as shown in Figure 7.6. Thus, minimizing $\|\mathbf{w}\|$ maximizes the margin.

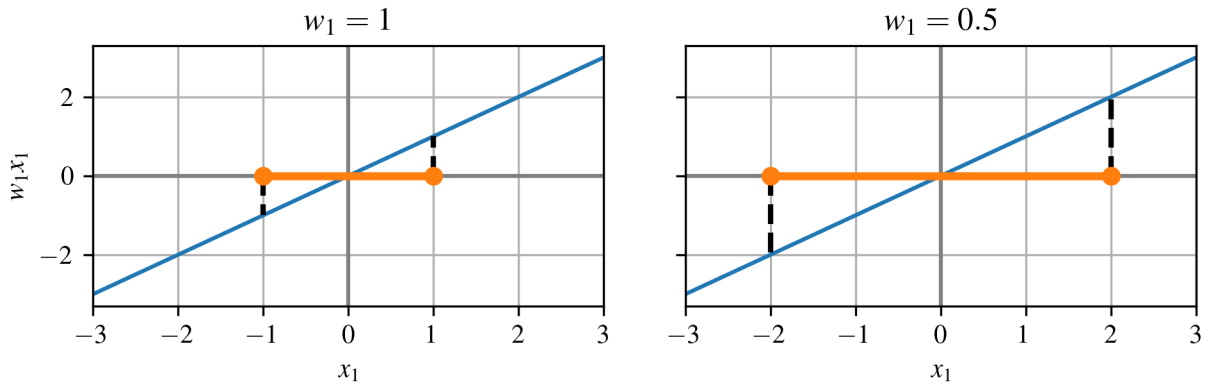


Figure 7.6: The margin is dependent on the value of the weight vector where a smaller weight vector results in a larger margin and vise versa.

To achieve a large margin while enforcing that no data points fall within the margin (hard margin), we ensure the decision function exceeds +1 for all positive training instances and is less than -1 for all negative instances. Let $t^{(i)}$ equal -1 for negative instances ($y^{(i)} = 0$) and +1 for positive ones ($y^{(i)} = 1$). The constraints then require

$$t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \tag{7.3}$$

for all training instances. This forms the basis of the hard margin linear SVM classifier optimization problem:

$$\begin{aligned} \underset{\mathbf{w},b}{\text{minimize}} \quad & \frac{1}{2}\mathbf{w}^\top \mathbf{w} \\ \text{subject to} \quad & t^{(i)}(\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 \text{ for } i = 1, 2, \ldots, m \end{aligned} \tag{7.4}$$

NOTE

The objective function minimized is $\frac{1}{2}\mathbf{w}^\top \mathbf{w}$, equivalent to $\frac{1}{2}\|\mathbf{w}\|^2$. This formulation is chosen over minimizing $\|\mathbf{w}\|$ directly because $\frac{1}{2}\|\mathbf{w}\|^2$ offers a straightforward derivative, simply $\mathbf{w}$, facilitating gradient calculations. In contrast, $\|\mathbf{w}\|$ lacks differentiability at $\mathbf{w} = 0$, posing challenges for optimization algorithms, which typically require smooth, differentiable functions to ensure effective optimization.

To formulate the soft margin objective, it is necessary to introduce a slack variable $\zeta^{(i)} \geq 0$ for each instance. This variable, $\zeta^{(i)}$, quantifies the permissible margin violation for the i$^{\text{th}}$ instance. Consequently, we face dual objectives: minimizing the slack variables to reduce margin violations and minimizing $\frac{1}{2}\mathbf{w}^\top\mathbf{w}$ to maximize the margin. The hyperparameter C plays a crucial role here, enabling a balance between these competing objectives. The introduction of C transforms our task into a constrained optimization problem.

$$\begin{aligned} \underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad & \frac{1}{2}\mathbf{w}^\top\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)} \\ \text{subject to} \quad & t^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \text{ and } \zeta^{(i)} \geq 0 \text{ for } i = 1,\ 2,\ \cdots,\ m \end{aligned} \tag{7.5}$$

7.1.3 Quadratic Programming

Both hard margin and soft margin problems are examples of convex quadratic optimization problems with linear constraints, commonly referred to as Quadratic Programming (QP) problems. QP involves solving optimization problems where the objective is a quadratic function and the constraints are linear. This form of programming, established in the 1940s, predates and is distinct from "computer programming," and is sometimes more descriptively termed "quadratic optimization" to avoid confusion.

A variety of techniques available through off-the-shelf solvers can address these QP problems, though they extend beyond the scope of this text. The general formulation of a QP problem is as follows:

$$\begin{aligned} \underset{\mathbf{p}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{p}^\top\mathbf{H}\mathbf{p} + \mathbf{f}^\top\mathbf{p} \\ \text{subject to} \quad & \mathbf{A}\mathbf{p} \leq \mathbf{b} \end{aligned} \tag{7.6}$$

Here, $\mathbf{p}$ is an n_p-dimensional vector (where n_p is the number of parameters), $\mathbf{H}$ is an $n_p \times n_p$ matrix, $\mathbf{f}$ is an n_p-dimensional vector, $\mathbf{A}$ is an $n_c \times n_p$ matrix (with n_c being the number of constraints), and $\mathbf{b}$ is an n_c-dimensional vector.

Equation 7.6 defines a standard quadratic program with constraints of the form $\mathbf{A}\mathbf{p} \leq \mathbf{b}$. For training a hard margin linear SVM, this setup can be used by choosing parameters that encode the SVM objective and constraints. The solution vector $\mathbf{p}$ contains both the bias term and the feature weights. Using this knowledge, a standard QP solver can be applied directly to find the optimal SVM classifier.

Example 7.1 Support Vector Machine Classification

This example applies a linear Support Vector Machine (SVM) classifier to distinguish Iris-Virginity flowers based on petal length and width. The decision boundary is derived after scaling the data, and margins are visualized. Misclassified instances within the margin are identified and marked. The model's performance is assessed using a confusion matrix and F1 score.

7.2 Nonlinear SVM Classification

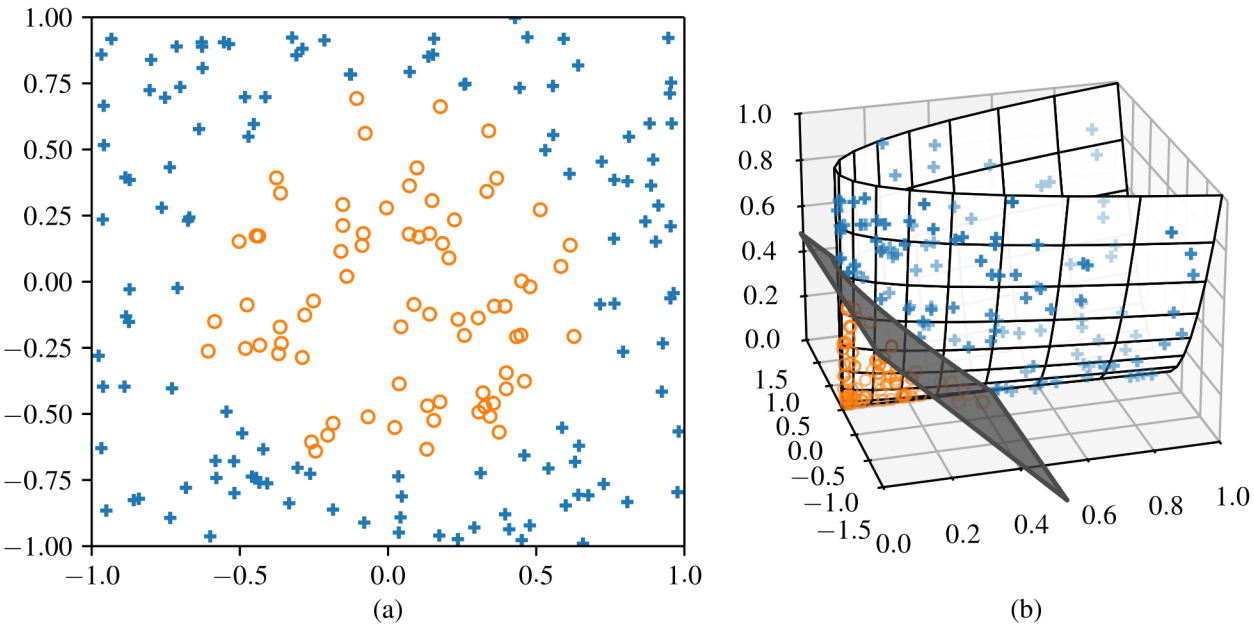


Figure 7.7: Nonlinear SVM example and illustration that shows: (a) 2D data that is not linearly separable , and (b) the same data plotted in a transformed feature space such that is is now linearly separable.

While linear SVM classifiers are quite effective and perform exceptionally well in various scenarios, many datasets are far from being linearly separable. One strategy to address non-linear datasets is to introduce additional features, such as polynomial feature. Adding features can sometimes transform the dataset into one that is linearly separable. A representation of this technique is shown in 7.7.

A simple example of converting non-linearly separable variables is shown in figure 7.8 where the left plot displays a simple dataset with a single feature x_1. Clearly, this dataset is not linearly separable. However, by adding another feature $x_2 = (x_1)^2$, the dataset becomes perfectly linearly separable in two dimensions.

[a]Machine Learner, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons

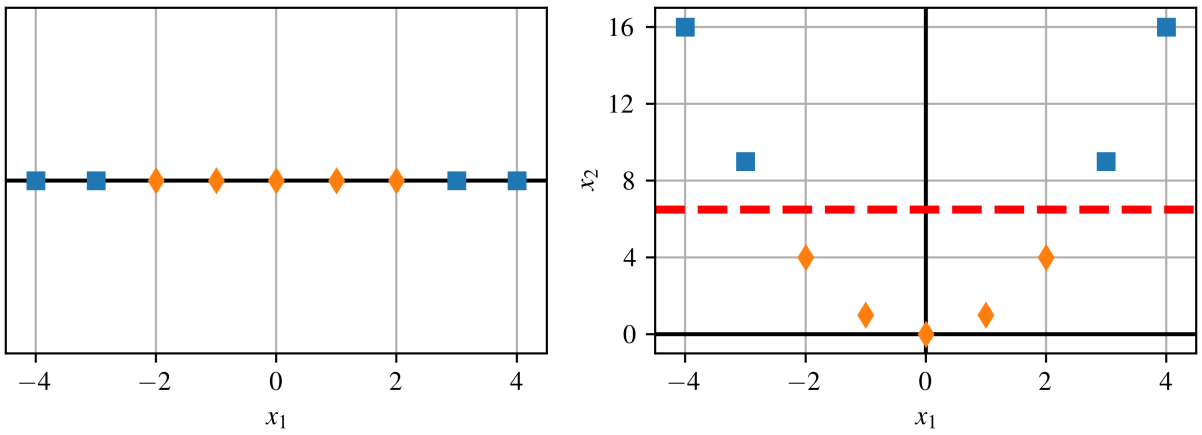


Figure 7.8: Illustration of SVM in higher dimensions

You can implement this idea in Scikit-Learn by creating a pipeline that applies a Polynomial Features transformer (introduced in the Regression Chapter), followed by a `StandardScaler` and a `LinearSVC`. The approach works nicely on the moons dataset, a toy binary-classification problem in which the samples trace two interleaving half-circles, as illustrated in Figure 7.9. You can generate this dataset with the function `make_moons()`.

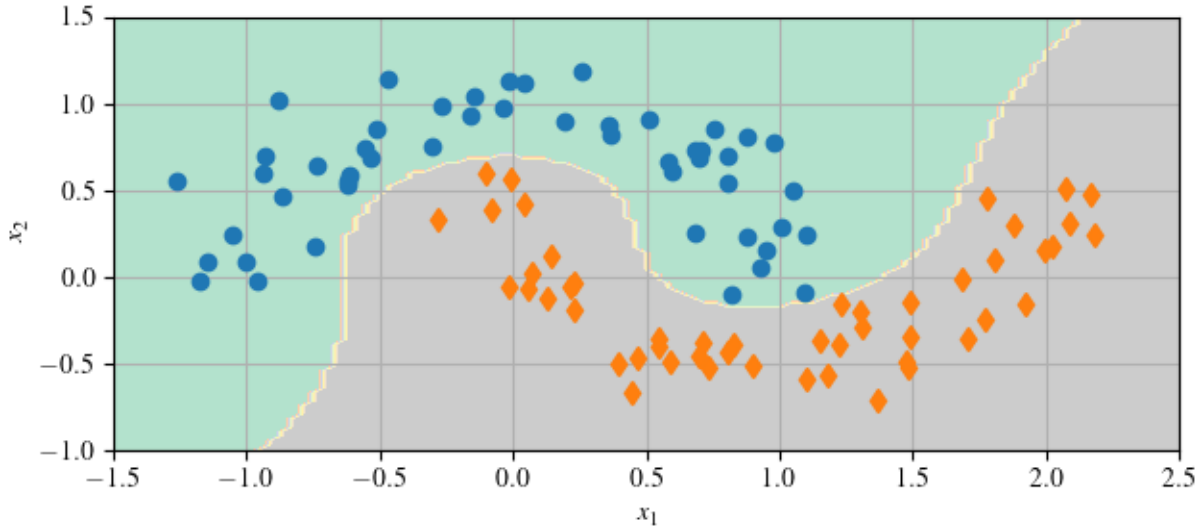


Figure 7.9: Demonstration of a SVM classifier with polynomial features.

Example 7.2 Nonlinear Classification
This example demonstrates nonlinear classification leveraging `LinearSVC` in a pipeline with `preprocessing.PolynomialFeatures` using the `make_moons` dataset. A polynomial feature transformation is combined with a linear SVM to classify the data, and the resulting decision boundaries are visualized.

7.2.1 Kernel trick

While adding polynomial features is straightforward and enhances the performance of various Machine Learning algorithms (not limited to SVMs), it presents limitations. Specifically, lower-degree polynomials may not adequately handle complex datasets, and higher-degree polynomials significantly increase the feature count, slowing down the model.

SVMs offer a unique solution through a remarkable mathematical technique known as the kernel trick, which allows for the benefits of high-degree polynomial features without actually expanding the feature space, thereby avoiding a rapid increase in computation. This kernel trick is incorporated within the SVC class.

In the dual form of an SVM the explicit dot product of two input vectors

$$\mathbf{x}^\top \mathbf{z} \tag{7.7}$$

is replaced by a kernel function

$$k(\mathbf{x}, \mathbf{z}), \tag{7.8}$$

where $\mathbf{x}$ and $\mathbf{z}$ are n-dimensional feature vectors representing any two data points in the training set. This substitution lets the algorithm construct a linear separator in a (possibly infinite-dimensional) feature space while all calculations still occur in the original input coordinates.

Figure 7.10 shows configured SVM classifiers using a 3^{rd} and 3^{th} degree polynomial kernel; on the left and right respectively. Adjusting the polynomial degree can help manage model fit: reducing the degree may prevent overfitting, whereas increasing it may be necessary for underfitting scenarios. The ‘coef0‘ hyperparameter is crucial as it determines the influence of high versus low-degree polynomials in the model.

NOTE

A typical method for determining optimal hyperparameter settings involves utilizing grid search techniques. Starting with a broad, coarse grid search to quickly narrow down potential candidates, followed by a more detailed, finer grid search centered on these promising values often yields faster results. Additionally, understanding the function and influence of each hyperparameter aids in efficiently targeting the most relevant areas of the hyperparameter space.

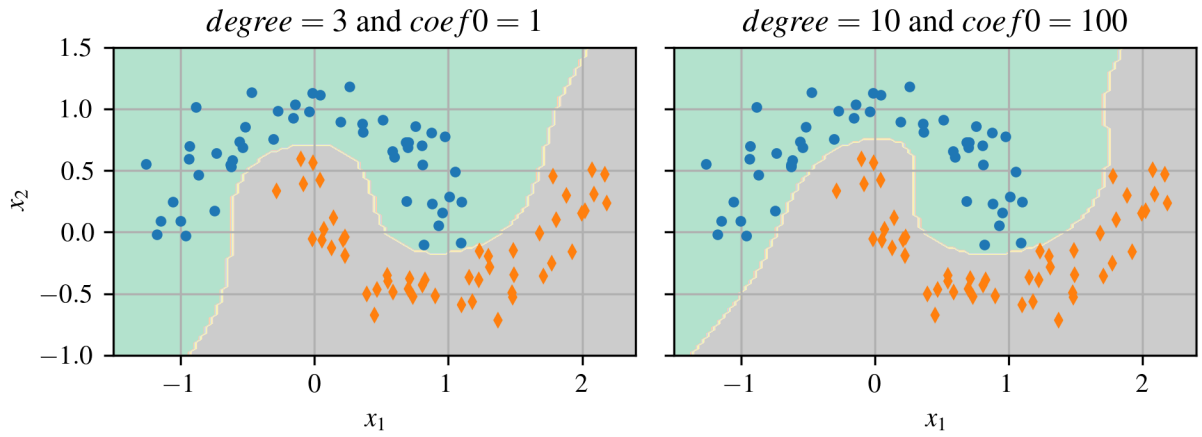


Figure 7.10: SVM polynomial kernel

7.2.2 Standard Kernels

Three kernels cover the vast majority of practical cases.

- The polynomial kernel captures interactions between input features up to a chosen degree d:

$$k_{\text{poly}}(\mathbf{x},\mathbf{z}) = \left(\mathbf{x}^{\mathrm{T}}\mathbf{z} + c\right)^{d} \tag{7.9}$$

  where $c \geq 0$ is a constant offset. It is effective when domain knowledge suggests that a low-order combination of variables explains the target. Keep d modest (typically $d \leq 5$) and standardise inputs to avoid exploding feature dimensions and overfitting.

- The radial basis function (RBF) kernel builds smooth, highly flexible decision boundaries:

$$k_{\text{rbf}}(\mathbf{x},\mathbf{z}) = \exp\left(-\gamma\|\mathbf{x}-\mathbf{z}\|^{2}\right) \tag{7.10}$$

  with width parameter $\gamma > 0$. A large γ makes the surface too flat (underfitting), while a small γ lets every point carve its own pocket (overfitting). Tune C and γ jointly, typically on a logarithmic grid, after x-score standardising the features.

- The sigmoid kernel adds an S-shaped non-linearity to the inner product:

$$k_{\text{sig}}(\mathbf{x},\mathbf{z}) = \tanh\left(\kappa\,\mathbf{x}^{\mathrm{T}}\mathbf{z} + \theta\right) \tag{7.11}$$

  where κ controls the slope and θ the offset. Although useful for some sparse or text data, this kernel is not always positive-semidefinite, so ensure your software handles the resulting optimisation safely.

Example 7.3 Polynomial Kernel Trick
This example uses the kernel trick to enable an SVM to classify non-linearly separable data using `SVC` (not `LinearSVC`). A third-degree polynomial kernel is applied to the moons dataset using Scikit-Learn's `Pipeline` and `SVC` tools, producing a curved decision boundary that cleanly separates the classes.

NOTE

Begin with the RBF kernel as a strong default, explore polynomial kernels when you expect specific interaction orders, and treat the sigmoid option as experimental unless you have evidence it helps.

7.3 Computational Complexity

Scikit-Learn provides two main SVM classifiers that trade accuracy for speed in different ways. `LinearSVC` is optimized for large, high-dimensional data when a linear decision boundary is adequate; `SVC` sacrifices runtime for the expressive power of kernels and thus handles complex, nonlinear patterns. Table 1 highlights how these priorities translate into computational costs and practical constraints and compares them to `SGDClassifier` for reference.

Table 5: Key characteristics of three Scikit-Learn SVM classifier implementations.

Trule class	time complexity	out-of-core support	scaling required	kernel trick
`LinearSVC`	$O(m \times n)$	no	yes	no
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	no	yes	yes
`SGDClassifier`	$O(m \times n)$	yes	yes	no

`LinearSVC` builds on the `liblinear` solver and is limited to *linear* decision boundaries. Because it does not apply the kernel trick, its training cost grows almost linearly with both the number of samples m and features n, i.e. $O(mn)$. Convergence is controlled by the tolerance parameter `tol` (denoted ε in the literature); the default value is usually sufficient, but smaller tolerances can be specified when higher accuracy is critical.

`SVC`, in contrast, relies on `libsvm` and *does* support kernel functions. Its computational burden is markedly heavier, between $O(m^2 n)$ and $O(m^3 n)$ in practice, so training becomes prohibitive once the dataset reaches the hundreds-of-thousands range. Nevertheless, `SVC` excels on smaller or medium-sized problems that demand nonlinear decision surfaces. Runtime also scales with the average count of non-zero features per instance, meaning sparse high-dimensional inputs remain tractable.

7.4 SVM Regression

Support Vector Regression (SVR) adapts the SVM idea to regression by surrounding the prediction curve with a tube of width ε; points outside this tube incur a penalty and the optimizer keeps the tube as flat as possible while allowing a limited number of violations governed by the regularization constant C. Conceptually this reverses the classification goal: instead of widening a margin to separate two classes, SVR encourages the data to lie inside the margin, making ε the principal knob that controls how loose the fit may be for both linear and kernel-based models.

7.4.1 Linear SVR

The presence of additional training instances within the margin does not influence the predictions of the model, rendering it ε-insensitive. For linear SVM Regression tasks, the LinearSVR class from Scikit-Learn can be utilized. For instance, Figure 7.11 demonstrates two linear SVM Regression models trained on random linear data; one features a wide margin ($\varepsilon = 1.5$), and the other a narrower margin ($\varepsilon = 0.5$).

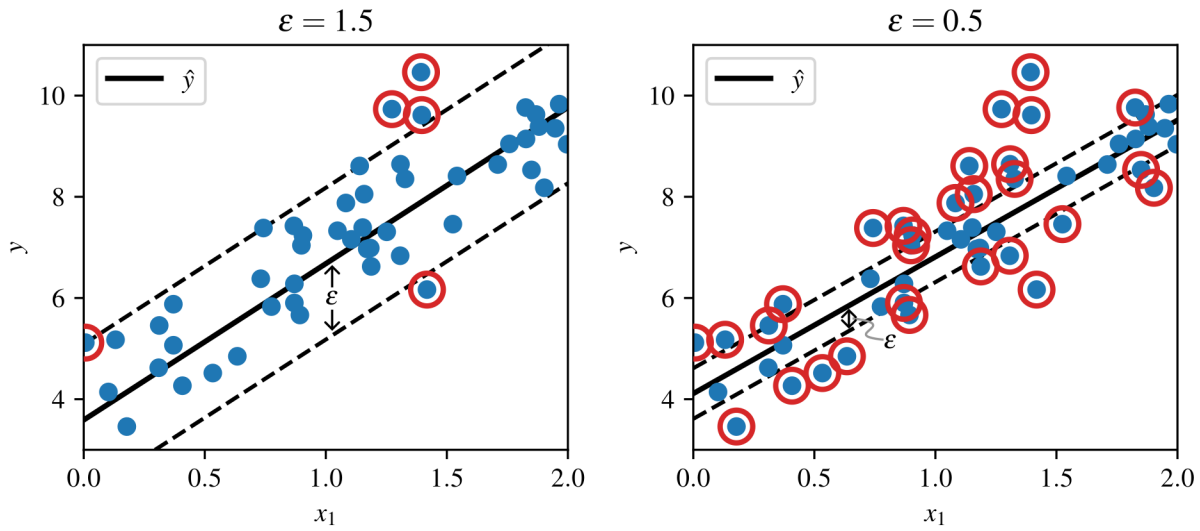


Figure 7.11: SVM Regression models with different ε values.

For data that follow an approximately linear trend, the `LinearSVR` class in Scikit-Learn solves the primal problem directly. It scales in $O(mn)$ time with m samples and n features, making it a practical choice for large data sets where a simple linear fit is adequate.

7.4.2 Kernel SVR

When the relationship is non-linear, the `SVR` class employs the kernel trick:

$$f(x) = \sum_{i=1}^{m} (\alpha_i - \alpha_i^*)\, k(x_i, x) + b, \tag{7.12}$$

where $k(\cdot,\cdot)$ is typically RBF or polynomial. Figure 7.12 shows a 2^{nd}-degree polynomial kernel capturing quadratic structure under various regularisation levels.

The Scikit-lern SVR class, supporting the kernel trick and acting as the regression counterpart to the SVC class, performs well with small to medium-sized datasets but slows considerably as dataset size increases. In contrast, the LinearSVR class, akin to the LinearSVC class, scales linearly with the size of the training set.

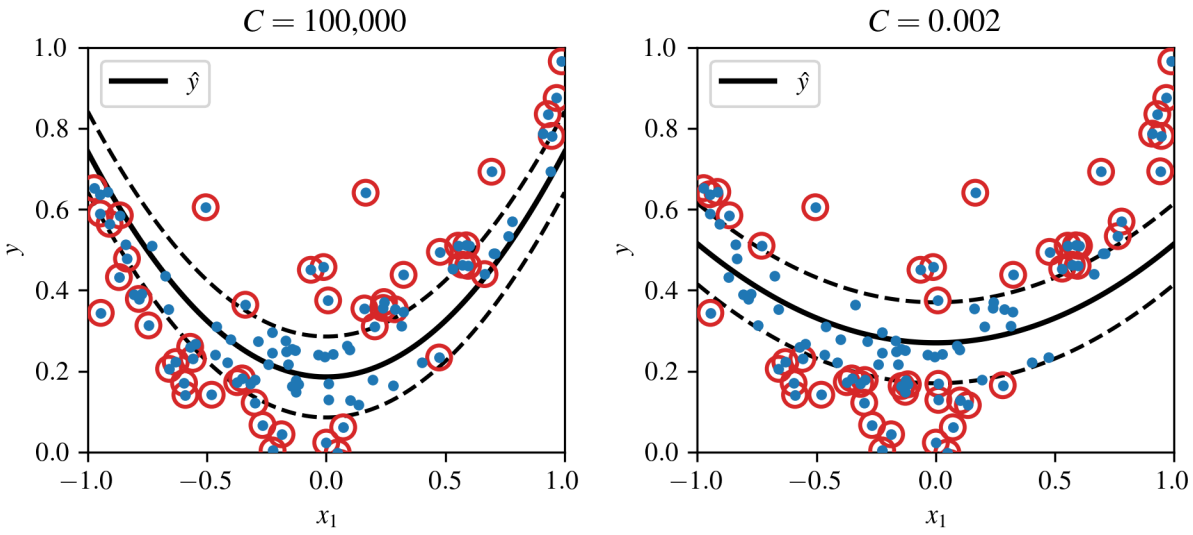


Figure 7.12: SVM regression with a 2nd-degree polynomial kernel, showcasing different regularization levels.

Example 7.4 SVM Polynomial Regression

This example fits a non-linear regression curve to noisy quadratic data using Support Vector Regression (SVR) with a polynomial kernel. It highlights how SVR models can handle non-linear relationships by introducing a margin of tolerance.

7.5 Examples

Example 7.1

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 7.1 Support Vector Machine Classification
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import svm
from sklearn import pipeline

cc = plt.rcParams['axes.prop_cycle'].by_key()['color']
plt.close('all')


# This code will build and train a support vector machine classifier with soft
# for the iris flower data set.

#%% Load your data

# We will use the Iris data set.  This dataset was created by biologist Ronald
# Fisher in his 1936 paper "The use of multiple measurements in taxonomic
# problems" as an example of linear discriminant analysis
iris = sk.datasets.load_iris()

# for simplicity, extract some of the data sets
X = iris['data'] # this contains the length of the petals and sepals
Y = iris['target'] # contains what type of flower it is
Y_names = iris['target_names'] # contains the name that aligns with the type of the
flower
feature_names = iris['feature_names'] # the names of the features

# plot the Sepal data
plt.figure(figsize=(6.5,3))
plt.subplot(121)
plt.grid(True)
plt.scatter(X[Y==0,0],X[Y==0,1],marker='o',zorder=10)
plt.scatter(X[Y==1,0],X[Y==1,1],marker='s',zorder=10)
plt.scatter(X[Y==2,0],X[Y==2,1],marker='d',zorder=10)
plt.xlabel(feature_names[0])
plt.ylabel(feature_names[1])

plt.subplot(122)
plt.grid(True)
plt.scatter(X[Y==0,2],X[Y==0,3],marker='o',label=Y_names[0],zorder=10)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()

#%% Extract just the petal space of the code

X_petal = X[50:, (2, 3)]  # petal length, petal width
y_petal = Y[50:] == 2

#%% Build and train the SVM classifier

# build handles to regularize the model data and a Linear Support Vector Classification.
scaler = sk.preprocessing.StandardScaler()
```

```
svm_clf = sk.svm.LinearSVC(C=1000000,max_iter=10000)

# build the model pipeline of regularization and a Linear Support Vector Classification.
scaled_svm_clf = sk.pipeline.Pipeline([
        ("scaler", scaler),
        ("linear_svc", svm_clf),
    ])

# train the data
scaled_svm_clf.fit(X_petal, y_petal)


#%% Build and plot the decision boundary along with the curbs

# Convert to unscaled parameters as the SVM is solved in a scaled space.
w = svm_clf.coef_[0] / scaler.scale_
b = svm_clf.decision_function([-scaler.mean_ / scaler.scale_])

# At the decision boundary, w0*x0 + w1*x1 + b = 0
# => x1 = -w0/w1 * x0 - b/w1
x0 = np.linspace(4, 5.9, 200)
decision_boundary = -w[0]/w[1] * x0 - b/w[1]

margin = 1/w[1]
curbs_up = decision_boundary + margin
curbs_down = decision_boundary - margin

#%% Plot the data and the classifier

plt.figure()
plt.grid(True)
plt.scatter(X[Y==1,2],X[Y==1,3],marker='s',label=Y_names[1],zorder=10)
plt.scatter(X[Y==2,2],X[Y==2,3],marker='d',label=Y_names[2],zorder=10)
plt.xlabel(feature_names[2])
plt.ylabel(feature_names[3])
plt.legend(framealpha=1)
plt.tight_layout()

# plot the decision boundy and margins
plt.plot(x0, decision_boundary, "k-", linewidth=2)
plt.plot(x0, curbs_up, "k--", linewidth=2)
plt.plot(x0, curbs_down, "k--", linewidth=2)


#%% Find the misclassified instancances and add a circle to mark them

# Find support vectors (LinearSVC does not do this automatically) and add them
# to the SVM handle
t = y_petal * 2 - 1 # convert 0 and 1 to -1 and 1
support_vectors_idx = (t * (X_petal.dot(w) + b) < 1) # find the locations
# of the miss classifed data points that fall withing the vectors
svs = X_petal[support_vectors_idx]

plt.scatter(svs[:, 0], svs[:, 1], s=180,marker='o', facecolors='none',edgecolors='k')

#%% compute the confusion matirx and F1 score

y_predicted = scaled_svm_clf.predict(X_petal)
confusion_matrix = sk.metrics.confusion_matrix(y_predicted, y_petal)
f1_score = sk.metrics.f1_score(y_predicted, y_petal)

print(f1_score)
```

Example 7.2

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 7.2 Polynomial Features
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import pipeline
from sklearn import svm

plt.close('all')

#%% Build and plot the data

# build the data
X, y = sk.datasets.make_moons(n_samples=100, noise=0.25, random_state=2)

plt.figure()
plt.plot(X[:,0][y==0],X[:,1][y==0],'s')
plt.plot(X[:,0][y==1],X[:,1][y==1],'d')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

#%% SVM polynominal features
svm_clf = sk.pipeline.Pipeline([
        ("poly_features", sk.preprocessing.PolynomialFeatures(degree=3)),
        ("scaler", sk.preprocessing.StandardScaler()),
        ("svm_clf", sk.svm.LinearSVC(C=10))
    ])
svm_clf.fit(X, y)

# make the 2d space for the color
x1 = np.linspace(-2, 3, 200)
x2 = np.linspace(-2, 2, 100)
x1_grid, x2_grid = np.meshgrid(x1, x2)

# calculate the binary decions and predection values
X2 = np.vstack((x1_grid.ravel(), x2_grid.ravel())).T
y_pred = svm_clf.predict(X2).reshape(x1_grid.shape)
y_decision = svm_clf.decision_function(X2).reshape(x1_grid.shape)

con_lines = [-30,-20,-10,-5,-2,-1,0,1,2,5,10,20,30]


# plot the figure
plt.figure()
# provide the solid background color for classification
plt.contourf(x1_grid, x2_grid, y_pred, cmap=plt.cm.brg, alpha=0.2)
# add the contour colors for the threshold
plt.contourf(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg, alpha=0.1)
# add the contour lines
contour = plt.contour(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "s")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "d")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
```

Example 7.3

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Example 7.2 Polynomial Features
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import datasets
from sklearn import pipeline
from sklearn import svm

plt.close('all')

#%% Build and plot the data

# build the data
X, y = sk.datasets.make_moons(n_samples=100, noise=0.25, random_state=2)

plt.figure()
plt.plot(X[:,0][y==0],X[:,1][y==0],'s')
plt.plot(X[:,0][y==1],X[:,1][y==1],'d')
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")

#%% SVM polynominal features
svm_clf = sk.pipeline.Pipeline([
        ("poly_features", sk.preprocessing.PolynomialFeatures(degree=3)),
        ("scaler", sk.preprocessing.StandardScaler()),
        ("svm_clf", sk.svm.LinearSVC(C=10))
    ])
svm_clf.fit(X, y)

# make the 2d space for the color
x1 = np.linspace(-2, 3, 200)
x2 = np.linspace(-2, 2, 100)
x1_grid, x2_grid = np.meshgrid(x1, x2)

# calculate the binary decions and predection values
X2 = np.vstack((x1_grid.ravel(), x2_grid.ravel())).T
y_pred = svm_clf.predict(X2).reshape(x1_grid.shape)
y_decision = svm_clf.decision_function(X2).reshape(x1_grid.shape)

con_lines = [-30,-20,-10,-5,-2,-1,0,1,2,5,10,20,30]


# plot the figure
plt.figure()
# provide the solid background color for classification
plt.contourf(x1_grid, x2_grid, y_pred, cmap=plt.cm.brg, alpha=0.2)
# add the contour colors for the threshold
plt.contourf(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg, alpha=0.1)
# add the contour lines
contour = plt.contour(x1_grid, x2_grid, y_decision, con_lines, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "s")
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "d")
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
```

Example 7.4

```
"""
Example 7.4 SVM Regression
@author: Austin R.J. Downey
"""

import IPython as IP
IP.get_ipython().run_line_magic('reset', '-sf')

import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
from sklearn import svm


plt.close('all')

#%% build the data sets
np.random.seed(2) # 2 and 6 are pretty good
m = 100
X = 6 * np.random.rand(m,1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m,1)
y = y.ravel()

# plot the data
plt.figure()
plt.grid(True)
plt.plot(X,y,'o')
plt.xlabel('x')
plt.ylabel('y')


#%% SVM regression

svm_reg = sk.svm.SVR(kernel="rbf", degree=3, C=1, epsilon=0.8, gamma="scale")
# Try poly kernel, and different degree, C, and epsilon values
svm_reg.fit(X, y)
x1 = np.linspace(-3, 3, 100).reshape(100, 1)
y_pred = svm_reg.predict(x1)


# plot the SVR model on top of the existing data
plt.plot(x1, y_pred, "-", linewidth=2, label=r"$\hat{y}$")
plt.plot(x1, y_pred + svm_reg.epsilon, "g--",label='curb')
plt.plot(x1, y_pred - svm_reg.epsilon, "g--")
plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=100,marker='o', facecolor='none',
 edgecolors='gray')
plt.legend(loc="upper left")
```