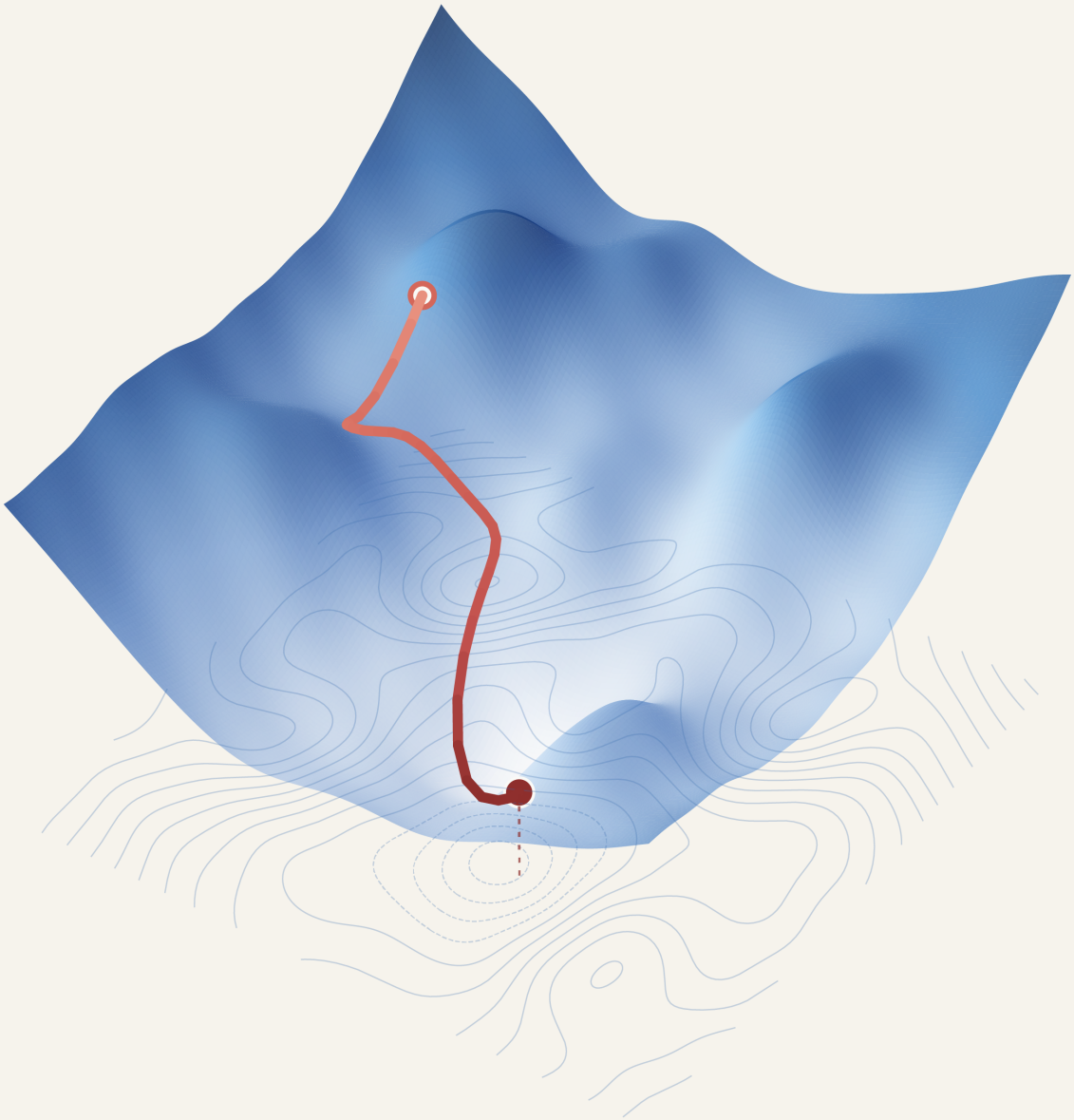

바닥부터 시작하는 머신러닝

직관에서 수식으로
수식에서 코드로

박성식

바닥부터 시작하는 **머신러닝**

직관에서 수식으로,
수식에서 코드로

박성식

Contents

Preface

잠이 오지 않던 2026년 6월 11일 새벽, 여러 가지 생각을 하다 문득 예전부터 해 보고 싶었던 일을 무턱대고 시작해 봤습니다. 코로나가 한창이던 시기에 조교수로 부임해, 첫 학기부터 여섯 학기를 내리 「머신러닝과데이터사이언스」를, 마지막 두 학기에는 「인공지능기초수학」도 함께 진행했습니다. 그 시간을 거치며 다듬어 온 기록들을 책으로 정리해 보자는 것이 그날 새벽의 결심이었습니다. 수강생들과 강의실에서 함께했던 시간들, 영상과 강의 자료와 판서로 남은 기록들, 그리고 종강 후에 남겨 준 강의 평가들이 저를 이끌었습니다.

그 수업들은, 그리고 이 책은 무얼 하고 싶었던 것일까요? 이렇게 표현해 보지요. 많은 사람들이 `sklearn`을 씁니다. `fit`을 부르고 `predict`를 부르면 답이 나옵니다. 그런데 그 사이에서 무슨 일이 일어나는지 물으면, 우리는 갑자기 말문이 막힙니다. 이 책은 그 침묵을 위한 책입니다.

시중에는 이미 훌륭한 머신러닝 책이 많습니다. 코드를 따라 하다 보면 어느새 모델이 돌아가는 책도 있고, 수백 쪽의 수식으로 이론을 쌓아 올리는 책도 있습니다. 그런데 그 사이가 비어 있었습니다. 코드는 돌릴 줄 알지만 그 밑의 수학이 궁금한 사람, 수식을 읽을 수는 있지만 그 수식이 왜 거기 있어야 하는지 납득하고 싶은 사람을 위한 책 말입니다. 강의실에서, 그리고 화면 너머에서 매 학기 만난 학생들이 가려워하던 곳이 바로 거기였습니다. 그 가려움을 긁어 주고 싶어서, 이 책은 처음부터 강의 없이 혼자 읽어도 진도가 나가는 책이 되도록 썼습니다.

무엇이든 물으면 답이 나오는 시대입니다. 아무것도 없어도 "딸깍" 하면 답이 나오는 시대에, 책이 무슨 소용일까. 이 책을 쓰는 내내 스스로에게 물었습니다. 그런데 딸깍해서 나오는 것은 답이지, 이해가 아닙니다. 그래서 저는 이렇게 답하기로 했습니다. 제대로 "이해"해 보자. 그렇다면 무엇을 이해할 것인가. 머신러닝은 곧 직관과 아이디어가 수학으로 기술되고, 그것이 코드로 구현되는 과정입니다. 그 과정을 바닥부터 이해해 보자.

그래서 이 책은 한 가지 원칙을 고집합니다. 동기로부터 직관을 얻고, 직관을 수식으로 옮기고, 수식을 코드로 구현하는 것입니다. 이 책의 거의 모든 개념이 이 순서로 흐릅니다. 제목의 "바닥부터"는 이 사슬의 어느 고리도 건너뛰지 않겠다는 뜻입니다.

Bishop의 Pattern Recognition and Machine Learning, Géron의 Hands-on Machine Learning, Hastie 등의 The Elements of Statistical Learning 같은 쟁쟁한 책들로 공부했습니다. 다만 그 책들은 위대한 만큼 두껍고 어렵습니다. 이 책은 그 책들에서 공부하고 그 내용들로 여러

학기에 걸쳐 진행한 수업을 통해 제 방식으로 한 번 더 씹어, 삼키기 좋게 다져 내놓은 결과라고 보시면 됩니다.

이 책은 한국어 용어를 먼저 쓰되, 처음 등장할 때 영어 용어를 괄호로 함께 적습니다. 한국어로 된 자료에는 한계가 있기 때문입니다. 영어 용어를 알고 있으면 책과 논문은 물론 강의 영상, 슬라이드, 포럼까지 훨씬 넓은 자료의 세계가 열립니다. 이 책이 끝이 아니라 그 세계로 나가는 통로가 되기를 바라는 마음입니다.

이 책을 시작하는 데 필요한 것은 많지 않습니다. Python을 다뤄 본 경험, 그리고 벡터, 행렬, 확률을 배운 기억이면 충분합니다. 그 기억이 흐릿해도 괜찮습니다. 필요한 수학은 2장에 정리해 두었고, 그마저도 이후의 장에서 실제로 쓰이는 것들만 골라 담았습니다. 2장이 낯설면 일단 건너뛰었다가, 본문에서 막히는 지점이 생겼을 때 돌아오셔도 됩니다.

완벽한 책이 아니라는 것을 압니다. 발견된 오류는 홈페이지(https://mlfromscratch.io/ko/)에 정오표로 정리해 두겠습니다. 앞서 말씀드린 강의 영상과 자료도 여기서 확인하실 수 있습니다. 새로운 오류나 의견은 홈페이지나 메일(spark88@knou.ac.kr)로 알려 주시기 바랍니다.

인생에는 때가 있는 듯합니다. 강의실에서 학생들과 얼굴을 맞대던 시절은, 수업에 대한 애정과 열정이 가장 뜨겁던 때였습니다. 지금은 화면 너머로 강의를 하며, 그때의 애정과 열정을 이 책 한 권에 갈무리하는 때를 지나고 있습니다.

아무쪼록 이 책이 여러분이 하시려는 일에 도움이 되었으면 좋겠습니다.

2026년 여름
박성식

Part II

Learning a Hyperplane

Chapter 3

Linear Regression

어떤 집의 면적과 방의 개수를 알고 있을 때, 그 집의 가격을 예측할 수 있을까요? 경험 많은 부동산 중개인이라면 감으로 짐작하겠지만, 우리에게는 감 대신 데이터가 있습니다. 과거에 거래된 집들의 면적과 방 개수, 그리고 실제로 팔린 가격 말입니다. 이 데이터로부터 아직 팔리지 않은 집의 가격을 예측하는 것, 이것이 이 장에서 풀려는 회귀 문제입니다.

이 문제를 풀기 위해 우리는 첫 번째 머신러닝 모델인 선형 회귀를 만납니다. 앞 장에서 준비한 수학이라는 도구가 드디어 실전에 투입되는 순간입니다. 선형 회귀는 가장 단순한 모델이지만, 이 책 전체를 관통하는 흐름이 이 장에 모두 담겨 있습니다. 문제를 정의하고, 모델을 가정하고, 비용 함수를 세우고, 그 비용을 최소로 만드는 파라미터를 찾는 것입니다. 이후에 등장하는 대다수의 모델은 이 틀 안에서 모델과 비용 함수를 바꿔 가는 변주입니다. 그런 의미에서 이 장은 하나의 모델을 배우는 장이 아니라, 머신러닝이라는 사고방식 자체를 배우는 장이라고 할 수 있습니다.

이론적으로는 선형 회귀 문제에 대한 두 가지 해법을 배웁니다. 먼저 최소 제곱법을 통해 비용을 최소화하는 파라미터를 수식 한 번으로 구하는 닫힌 해를 유도합니다. 그런데 답을 한 번에 구할 수 있는데도, 우리는 조금씩 답에 다가가는 두 번째 방법인 경사 하강법을 또 배웁니다. 왜 굳이 그런 근사적인 방법을 사용하는지 알아가봅시다. 마지막에는 이렇게 배운 방법들을 Python으로 직접 구현하여 실제 집값 데이터로 예측을 해 봅니다. 이 장을 시작하며 던진 질문에 우리 손으로 답하게 되는 것이지요.

3.1 Model Learning

모델(model). 모델의 목적은 주어진 입력들과 그에 대한 출력들 사이에 숨겨진 관계를 찾아 기술하고, 새로 주어진 입력값에 대한 그럴듯한 예측값을 내놓는 것입니다. 그 숨겨진 관계는 어떤 함수의 형태로 주어질 겁니다. 모델이라 함은 일반적으로 입력과 출력 사이에 적절한 가정을 두어 함수의 모양을 정한 것입니다. 그리고 그 함수에는 조절 가능한 모델의 파라미터(parameter, 매개변수)가

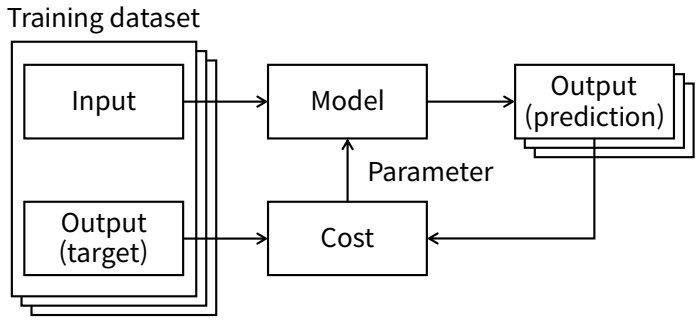

Figure 3.1: 모델 학습 전체 흐름

있어서, 이 파라미터를 통해 입력과 출력 사이의 관계를 더 정확하게 표현할 수 있도록 모델을 조정할 수 있습니다. 우리의 목적은 주어진 데이터를 잘 기술해주는 적절한 모델과 그 파라미터를 찾아내는 것이라고 할 수 있습니다.

예측(prediction) 및 목표(target). 모델은 주어진 입력(input)에 대해 출력(output)을 내게 됩니다. 출력이라고 하는 이유는 입력으로 들어온 값을 바탕으로 모델에 정해진 어떤 절차나 계산을 통해 결과 값을 출력해내기 때문입니다. 이렇게 계산되어 얻어진 출력을 우리는 "예측(prediction)"이라고 부릅니다.[3.1] 학습 데이터셋은 입력과 출력의 쌍으로 이루어져 있어서, 우리는 하나의 입력에 대해 두 종류의 출력을 갖게 됩니다. 하나는 학습 데이터셋에 들어있는 입력과 쌍을 이루고 있던 정답으로서의 출력과, 다른 하나는 동일한 입력을 모델에 넣어서 얻어진 예측으로서의 출력입니다. 이 때, 정답으로서의 출력을 "목표(target)"라 부르기도 합니다. 그래서 입력에 대한 정답으로서의 목표값을 y라고 쓰고, 모델에 동일한 입력을 넣어서 얻어진 예측값으로서의 출력값은 y에 대한 예측이라는 뜻에서 모자를 씌워서 $\hat{y}$라고 씁니다. 그러니 우리의 목표는 $\hat{y}$를 최대한 y에 가깝도록 만드는 것입니다.

학습 데이터셋(training dataset). 학습 데이터셋에는 보통 여러 개의 입력-출력 쌍이 들어있습니다. 그 중 n번째 입력과 출력을 $\mathbf{x}^{(n)}, y^{(n)}$라고 합시다. 이 둘은 하나의 쌍 $(\mathbf{x}^{(n)}, y^{(n)})$을 이룹니다. 입력은 자기 나름의 차원을 가지고 있는 반면에 출력은 스칼라로 주어진다고 하겠습니다. 그러면 학습 데이터셋은 다음과 같은 입력-출력 쌍의 집합일 것입니다.

$$\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)}) : n = 1, 2, \ldots, N\} \tag{3.1}$$

보통 데이터셋은 집합이므로 $\mathcal{D}$로 씁니다. 또한 여기서 N은 전체 학습 데이터셋에 들어있는 입력-출력 쌍의 개수입니다.

[3.1] 통계학에서는 추정(estimation)이라는 표현도 사용합니다.

수학적 기술(mathematical formulation). 좀 더 수학적으로 기술해봅시다. 모델은 결국 입력에 대한 함수이며 동시에 모델의 파라미터에 대한 함수이기도 합니다. 보통은 아래와 같이 씁니다.

$$\hat{y} = f(\mathbf{x}; \boldsymbol{\theta}) \tag{3.2}$$

입력 $\mathbf{x}$가 주어졌을 때, 우리는 우리의 모델에 대한 가정 f와 이를 정의하는데 필요한 파라미터 $\boldsymbol{\theta}$를 이용하여 예측 $\hat{y}$를 계산합니다.

입력은 고정되어 있어도 파라미터가 바뀌면 모델의 출력도 달라집니다. 파라미터를 조정하면 동일한 가정과 동일한 입력에 대해서도 다른 예측값을 얻어낼 수 있습니다. 즉, 파라미터는 조정 패널에 달려있는 다이얼과 같아서 우리는 이 다이얼을 돌려 모델을 튜닝(tuning)할 수 있습니다. 그럼 우리의 문제는 학습 데이터셋에 들어있는 입력과 출력 관계를 잘 맞출 수 있는 적절한 모델과 파라미터를 찾는 문제로 귀결될 것입니다.

비용(cost). 우리는 주어진 목표값에 최대한 유사하도록 예측값을 만들 수 있는 파라미터를 찾고자 합니다. 그러기 위해서 우리는 하나의 비용(cost)이라는 것을 정의합니다. 이 비용은 주어진 전체 학습 데이터셋의 목표값과 예측값을 비교하여, 두 출력값의 차이가 크면 클수록 비용 역시 커지도록 설계합니다. 우리는 파라미터를 조정할 때마다 예측값이 바뀌게 되므로 비용 역시도 영향을 받게 되는 것을 알 수 있습니다. 이러한 관계로부터 우리는 비용을 최소로 만드는 어떤 파라미터를 찾고자 하는 것입니다. 보통 $J(\boldsymbol{\theta})$로 나타냅니다. 예를 들어 예측값과 목표값의 차이를 제곱하여 모두 더하는 방식이 대표적인 비용 함수 중 하나입니다.

이러한 전체적인 틀은 이 장에만 국한되는 것이 아니라, 학습을 하게 되는 머신러닝 전반에 일반적으로 적용할 수 있는 이해의 틀입니다. 모델의 형태와 비용을 정의하는 방식이 알고리즘마다 달라질 뿐, "모델 $\rightarrow$ 예측 $\rightarrow$ 비용 $\rightarrow$ 파라미터 조정"이라는 순환 구조 자체는 이 책 전반에 걸쳐 계속 등장하게 됩니다.

Problem Definition

주어진 학습 데이터셋

$$\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)}) : n = 1, 2, \ldots, N\} \tag{3.3}$$

모델 (가정)

$$\hat{y} = f(\mathbf{x}; \boldsymbol{\theta}) \tag{3.4}$$

비용 함수

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}; \mathcal{D}) \tag{3.5}$$

최적해

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{3.6}$$

예측

$$\hat{y} = f(\mathbf{x}; \hat{\boldsymbol{\theta}}) \tag{3.7}$$

이렇게 추정한 파라미터를 토대로 새로이 주어진 입력에 대한 모델의 예측을 하게 됩니다. 이 예측은 학습 데이터셋에서 본 적이 없는 샘플이지만, 학습 데이터셋에서 얻어진 입력과 출력 간의 관계를 토대로 새로운 예측을 수행하는 단계입니다.

3.2 Linear Regression

3.2.1 Linear Model

회귀(regression). 회귀(regression)란 입력으로부터 연속적인 값을 예측하는 문제입니다. 이 장을 열며 던진 집값 예측이 바로 그러한 예입니다. 면적과 방 개수라는 입력으로부터 가격이라는 연속적인 값을 알아내려 했으니까요. 가격뿐 아니라 광고비로부터 매출을, 기온으로부터 전력 수요를 예측하는 일이 모두 회귀에 속합니다. 뒤에서 다룰 분류가 "이것은 어느 범주에 속하는가"를 묻는다면, 회귀는 "이것은 얼마인가"를 묻습니다. 답이 정해진 몇 개의 선택지가 아니라 연속적인 수직선 위 어딘가의 값이라는 점이 회귀를 분류와 가르는 지점입니다. 그렇다면 입력으로부터 출력을 예측하는 이 관계를 어떻게 함수로 세울 수 있을까요? 여기에는 하나의 가정이 필요합니다.

선형 모델(linear model). 앞서 언급한 것처럼 모델을 구축하기 위해서는 커다란 "가정"이 필요합니다. 이 가정은 입력과 출력 사이를 어떤 관계로 표현할 것인지에 대한 내용입니다. 수학적으로는 예측하는 함수의 모양을 결정하는 것입니다.

선형 모델은 입력과 출력 사이의 관계를 선형 조합으로 모델링하겠다는 얘기입니다. 자세히 쓰면 아래와 같습니다.

$$\hat{y} = f(x_1, x_2, \ldots, x_D; \theta_0, \theta_1, \theta_2, \ldots, \theta_D) \tag{3.8}$$

$$= \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_D x_D \qquad (x_0 = 1) \tag{3.9}$$

각각의 입력 x_d에 가중치 θ_d를 곱해서 더한 형태로 모델링을 했습니다. 여기에 θ_0는 편향(bias)이 됩니다.

예제 3.1. 어떤 부동산 가격을 예측하는 모델을 생각해봅시다. 입력으로 전체 면적 x_1 (단위: 제곱미터)과 방의 개수 x_2를 사용하여 가격 $\hat{y}$ (단위: 백만 원)를 예측한다고 합시다.

$$\hat{y} = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 \tag{3.10}$$

파라미터가 $\theta_0 = 50, \theta_1 = 3, \theta_2 = 10$으로 주어졌다고 합시다. 즉, θ_0는 기본 가격 5천만원, θ_1은 제곱미터당 3백만원, 그리고 θ_2는 방 한 칸 당 1천만원을 부가하여 예측하는 모델입니다.

그러면 주어진 모델에 대해서 입력값으로 주어진 면적이 $x_1 = 80$, 방이 $x_2 = 3$개인 집의 예측 가격(출력)을 구해봅시다.

$$\hat{y} = 50 \times 1 + 3 \times 80 + 10 \times 3 = 50 + 240 + 30 = 320 \tag{3.11}$$

즉, 이 모델은 해당 집의 가격을 3억 2천만 원으로 예측합니다. ■

벡터를 이용해 선형 모델 (3.9)을 간결하게 나타내면 다음과 같습니다. 여기에도 편향 θ_0를 추가하여 모델링 하겠습니다.

$$\begin{aligned}
\hat{y} &= f(\mathbf{x}; \boldsymbol{\theta}) && (3.12)\\
&= \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_D x_D \quad (x_0 = 1) && (3.13)\\
&= \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & \cdots & \theta_D \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} && (3.14)\\
&= \begin{bmatrix} — & \boldsymbol{\theta}^T & — \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix} && (3.15)\\
&= \boldsymbol{\theta}^T \mathbf{x} && (3.16)
\end{aligned}$$

즉, 입력값을 하나의 벡터로 모으고 마찬가지로 모델 파라미터도 하나의 벡터로 모으게 되면, 선형 모델은 벡터의 내적을 이용해 손쉽게 표현이 가능합니다.

편향을 포함하는 차원. 여기서 한 가지 짚고 넘어가야 할 부분이 있습니다. 원래 주어진 입력 $\mathbf{x}$가 D차원이었다면, 편향(bias) θ_0을 함께 표현하기 위해 $x_0 = 1$이라는 항을 추가로 끼워 넣었으므로, 실제로 위 식에 쓰인 벡터 $\mathbf{x}$와 $\boldsymbol{\theta}$는 D차원이 아니라 $D + 1$차원입니다. 그러나 이렇게 매번 $D + 1$차원이라고 쓰는 것은 번거로우므로, 이 책에서는 편의상 $x_0 = 1$이 포함된 입력 벡터의 차원도 그냥 D차원이라고 부르겠습니다. 즉, 별도의 언급이 없는 한 이 책에서는 편향을 입력 벡터 $\mathbf{x}$와 파라미터 벡터 $\boldsymbol{\theta}$에 흡수시켜 표기합니다. $x_0 = 1$ 항이 $\mathbf{x}$에, 그에 대응하는 편향 θ_0이 $\boldsymbol{\theta}$에 포함되어, 두 벡터의 내적 $\boldsymbol{\theta}^T \mathbf{x}$ 하나로 편향까지 함께 계산됩니다.

3.2.2 Cost Function

오차(error). 비용 함수에서 중요한 점은 목표값과 예측값을 비교하여 이 두 값이 다르면 다를수록 큰 값을 갖는 함수를 설계해야 한다는 점입니다. 이를 설계하기 위해 먼저 목표값과 예측값 사이의 차이를 오차(error)라고 정의합시다. 때로는 잔차(residual)라는 이름으로 불리기도 합니다. 아래와 같이 표현할 수 있습니다.

$$e = y - \hat{y} = y - \boldsymbol{\theta}^T \mathbf{x} \tag{3.17}$$

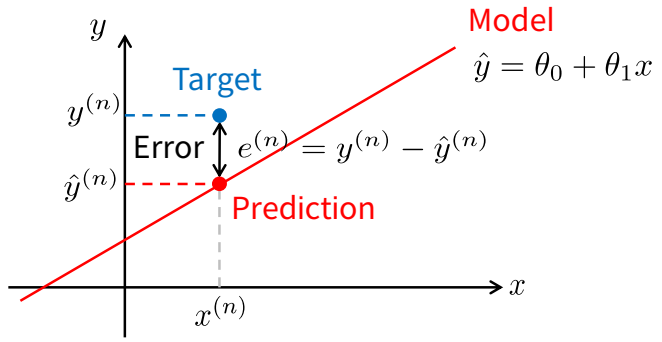

Figure 3.2: 모델이 주어졌을 때 하나의 샘플에 대한 목표값과 예측값 사이의 오차를 나타냅니다.

오차를 정의할 때 순서를 바꾸어 $\hat{y} - y$로 정의하기도 합니다. 어느 쪽으로 정의하든 오차를 제곱하여 더하는 비용 함수는 동일하고, 따라서 이를 최소화하여 얻는 최종 결과도 동일합니다.

대개 오차는 학습 데이터셋 안에 들어있는 샘플에 대해 계산되는 경우가 많습니다. 그렇기 때문에 n 번째 데이터에 대한 오차 $e^{(n)}$는 다음과 같이 정의될 것입니다.

$$e^{(n)} = y^{(n)} - \hat{y}^{(n)} = y^{(n)} - \boldsymbol{\theta}^T \mathbf{x}^{(n)} \tag{3.18}$$

그림 3.2에서는 1차원 입력에 대한 예측 모델을 볼 수 있습니다. 오차 $e^{(n)}$는 같은 입력 $x^{(n)}$에 대해 데이터가 갖고 있는 목표값 $y^{(n)}$과 모델이 내놓은 예측값 $\hat{y}^{(n)}$ 사이의 출력 방향(세로 방향) 거리로 이해할 수 있습니다. 모델이 데이터를 잘 설명할수록 이 거리는 짧아지고, 모델이 데이터를 잘못 설명할수록 거리는 길어질 것입니다.

비용 함수(cost function). 비용 함수는 전체 학습 데이터셋에 대하여 계산되는 함수입니다. 그렇기 때문에 비용 함수는 총 N개의 입력-출력 쌍에 대한 함수입니다. 그리고 또한 비용 함수는 모델 파라미터의 함수이기도 합니다. 모델 파라미터가 달라지면 전체 입력에 대한 전체 예측값이 영향을 받기 때문입니다.

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}\,; \mathcal{D}) \tag{3.19}$$

$$= J(\boldsymbol{\theta}\,; \mathbf{x}^{(1)}, y^{(1)}, \ldots, \mathbf{x}^{(N)}, y^{(N)}) \tag{3.20}$$

그렇지만 우리는 보통 비용 함수를 모델 파라미터에 대한 함수로 생각합니다. 주어진 학습 데이터셋은 변동 없이 유지되는 반면, 우리는 주어진 학습 데이터셋에 대해 모델 파라미터가 비용 함수에 주는 영향을 살펴보고자 하기 때문입니다.

SSE 비용. 오차의 부호는 목표값과 예측값 중 어느 쪽이 큰지에 따라 양수나 음수가 모두 가능합니다. 그렇기 때문에 우리는 오차에 제곱을 취해 항상 양의 크기를 갖도록 만들어서 비용 함수를 정의할

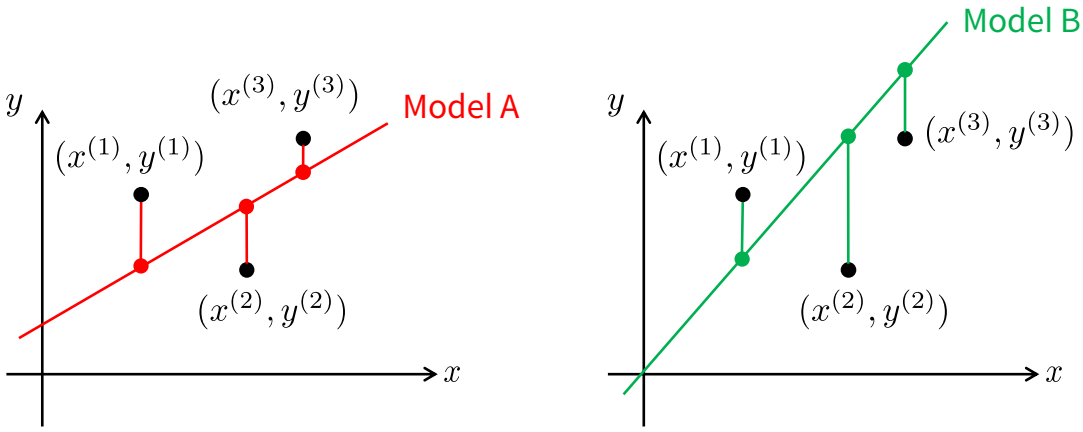

Figure 3.3: 서로 다른 두 모델이 같은 데이터셋에 대해 만들어내는 오차와 비용을 비교해봅니다.

것입니다. 둘의 차이가 클수록 큰 값으로 정의가 될 수 있도록 말입니다. 마치 분산을 정의할 때와 유사하죠. 아래와 같이 전체 학습 데이터셋에서 오차의 제곱을 합하여 구한 값을 정의해봅시다.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \left(e^{(n)}\right)^2 \tag{3.21}$$

$$= \sum_{n=1}^{N} (y^{(n)} - \hat{y}^{(n)})^2 \tag{3.22}$$

이렇게 정의한 비용 함수 $J(\boldsymbol{\theta})$를 SSE(sum of squared errors)라고 합니다.

그렇다면 파라미터를 다르게 선택하면 비용은 어떻게 달라질까요? 같은 데이터셋이라도 어떤 모델을 선택하느냐에 따라 오차의 크기가 달라지고, 그에 따라 비용도 달라집니다. 그림 3.3은 동일한 학습 데이터셋 $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)})$에 대해 서로 다른 두 선형 모델(Model A, Model B)을 적합시킨 결과를 보여줍니다. 각 모델이 동일한 입력에 대해 예측하는 결과들이 다르기 때문에, 같은 입력 데이터에 대해서도 오차의 크기(빨간 선과 초록 선의 길이)가 서로 다르게 나타납니다. 그럼 두 모델 중 어떤 모델이 주어진 데이터셋을 더 잘 설명한다고 볼 수 있을지 생각해봅시다. 이 오차들을 제곱하여 합한 값이 곧 각 모델의 SSE 비용이므로, 두 모델 중 오차가 전반적으로 더 작은 모델일수록 비용도 더 작을 것입니다.

한 가지 유념할 것은, 이 그림은 1차원 입력을 가정했기 때문에 입력과 출력을 각 축으로 하는 좌표 평면 위에 모델의 예측이나 오차를 표현할 수 있었다는 점입니다. 입력이 2차원인 경우까지는 출력을 포함하여 3차원 공간에 그려서 눈으로 확인할 수 있습니다. 이 때 모델의 예측은 3차원 공간 상에서 평면으로 나타날 것입니다. 그러나 입력이 3차원 이상으로 늘어나면 더 이상 그림으로 표현할 방법이 없습니다. 그럼에도 입력의 차원이 얼마가 되든 지금까지 다룬 오차와 비용의 정의는 그대로 유지됩니다. 우리는 비록 그림으로 직접 확인할 수는 없더라도, 다차원 입력에 대해서도 동일한 논리가 적용된다는 점을 염두에 두어야 합니다.

예제 3.2. 앞서 살펴본 부동산 가격 예측 모델을 다시 생각해봅시다. 입력은 면적 x_1(제곱미터)과 방의 개수 x_2이고, 파라미터는 $\theta_0 = 50, \theta_1 = 3, \theta_2 = 10$으로 동일합니다. 이제 실제로 거래된 집 세 채에 대한 학습 데이터셋이 다음과 같이 주어졌다고 합시다.

n	면적 $x_1^{(n)}$	방 개수 $x_2^{(n)}$	실제 거래가 $y^{(n)}$
1	80	3	320
2	60	2	250
3	100	4	410

각 데이터에 대한 예측값 $\hat{y}^{(n)} = \theta_0 x_0^{(n)} + \theta_1 x_1^{(n)} + \theta_2 x_2^{(n)}$을 먼저 구해봅시다.

$$\hat{y}^{(1)} = 50 \times 1 + 3 \times 80 + 10 \times 3 = 320 \tag{3.23}$$

$$\hat{y}^{(2)} = 50 \times 1 + 3 \times 60 + 10 \times 2 = 250 \tag{3.24}$$

$$\hat{y}^{(3)} = 50 \times 1 + 3 \times 100 + 10 \times 4 = 390 \tag{3.25}$$

이를 바탕으로 각 데이터에 대한 오차 $e^{(n)} = y^{(n)} - \hat{y}^{(n)}$를 구하면

$$e^{(1)} = 320 - 320 = 0 \tag{3.26}$$

$$e^{(2)} = 250 - 250 = 0 \tag{3.27}$$

$$e^{(3)} = 410 - 390 = 20 \tag{3.28}$$

입니다. 그러므로 이 모델과 이 학습 데이터셋에 대한 SSE 비용은

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{3} \left(e^{(n)}\right)^2 \tag{3.29}$$

$$= 0^2 + 0^2 + 20^2 = 400 \tag{3.30}$$

입니다. 첫 번째와 두 번째 집은 모델이 정확하게 예측했지만, 세 번째 집은 실제보다 2천만원 낮게 예측하여 SSE 비용이 발생하였습니다.

그렇다면 파라미터를 다르게 선택하면 비용은 어떻게 달라질까요? 이번에는 $\theta_0 = 50, \theta_1 = 3, \theta_2 = 15$로 방의 개수에 대한 가중치만 살짝 키운 모델을 생각해봅시다. 동일한 학습 데이터셋에 대해 예측값을 다시 구하면

$$\hat{y}^{(1)} = 50 \times 1 + 3 \times 80 + 15 \times 3 = 335 \tag{3.31}$$

$$\hat{y}^{(2)} = 50 \times 1 + 3 \times 60 + 15 \times 2 = 260 \tag{3.32}$$

$$\hat{y}^{(3)} = 50 \times 1 + 3 \times 100 + 15 \times 4 = 410 \tag{3.33}$$

오차는

$$e^{(1)} = 320 - 335 = -15 \tag{3.34}$$

$$e^{(2)} = 250 - 260 = -10 \tag{3.35}$$

$$e^{(3)} = 410 - 410 = 0 \tag{3.36}$$

이고, 비용은

$$J(\boldsymbol{\theta}) = (-15)^2 + (-10)^2 + 0^2 = 225 + 100 + 0 = 325 \tag{3.37}$$

입니다. 처음 모델의 비용 400보다 두 번째 모델의 비용 325가 더 작습니다. 즉, 두 번째 파라미터가 이 학습 데이터셋을 좀 더 잘 설명하고 있다는 뜻입니다. 이처럼 파라미터를 어떻게 선택하느냐에 따라 비용이 달라지며, 우리의 목표는 이 비용을 가장 작게 만드는 파라미터를 찾는 것입니다. ■

MSE 및 RMSE 비용. 지금까지 비용 함수로는 오차 제곱의 합인 SSE를 사용했습니다. 그런데 SSE는 비슷한 수준의 오차이더라도 샘플 수가 많아질수록 값이 커지기 때문에, 서로 다른 데이터셋 사이에서 모델의 성능을 비교하는 지표로는 적절하지 않습니다. 그래서 성능 평가에는 SSE를 샘플 수 N으로 나눈 평균 제곱 오차(MSE, mean squared error)를 주로 사용합니다.

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2 \tag{3.38}$$

그런데 MSE에는 한 가지 불편한 점이 있습니다. 오차를 제곱했기 때문에 스케일도 제곱이 됩니다. 가격 예측이라면 MSE의 단위는 "달러의 제곱"이 되어 직관적으로 와닿지 않습니다. 그래서 MSE에 제곱근을 취해 단위를 원래대로 되돌린 RMSE(root mean squared error)를 사용하기도 합니다.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2} \tag{3.39}$$

RMSE는 "평균적으로 예측이 실제값에서 얼마나 벗어나는가"를 원래 단위로 나타내는 값이라고 해석할 수 있습니다. 가격 예측 문제에서 RMSE가 40000이라면, 예측이 평균적으로 약 4만 달러 정도 벗어난다는 뜻입니다.

한 가지 짚어둘 점은, SSE를 최소화하는 파라미터는 MSE와 RMSE도 함께 최소화하기 때문에 어떤 비용을 선택하더라도 최소화 해는 동일하다는 점입니다.

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2 \tag{3.40}$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{N}\sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2 \tag{3.41}$$

$$= \arg\min_{\boldsymbol{\theta}} \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y^{(n)} - \hat{y}^{(n)})^2} \tag{3.42}$$

문제 정의(problem setup). 선형 회귀 문제를 다음과 같이 정리할 수 있습니다. 주어진 입력-출력 쌍으로 이루어진 학습 데이터셋에 대해, 선형 회귀 모델을 가정하여 SSE 비용을 최소로 만드는 모델 파라미터를 찾으세요.

Linear Regression: Problem Definition

주어진 학습 데이터셋

$$\{(\mathbf{x}^{(n)}, y^{(n)}) : n = 1, 2, \ldots, N\} \tag{3.43}$$

비용 함수의 최소화

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \sum_{n=1}^{N} (y^{(n)} - \boldsymbol{\theta}^T \mathbf{x}^{(n)})^2 \tag{3.44}$$

3.3 Closed-Form Solution

3.3.1 Least Squares

디자인 행렬(design matrix). 우리는 지금까지 하나의 샘플 $(\mathbf{x}^{(n)}, y^{(n)})$에 대한 모델의 예측과 오차를 살펴보았습니다. 그러나 비용 함수 $J(\boldsymbol{\theta})$는 학습 데이터셋에 들어있는 N개의 데이터 전체에 대해 계산되는 값입니다. 그렇기 때문에 N개의 데이터를 하나씩 별개로 다루는 것이 아니라, 이들을 하나의 행렬로 모아서 한꺼번에 다루는 표기법을 도입하면 식이 훨씬 간결해집니다. 이렇게 N개의 입력 벡터를 모아 만든 행렬을 디자인 행렬(design matrix) $\mathbf{X} \in \mathbb{R}^{N \times D}$라고 부르며, N개의 입력 벡터를 행벡터로 만들어 세로로 쌓아서 만든 행렬입니다.

$$\mathbf{X} = \begin{bmatrix} — & \mathbf{x}^{(1),T} & — \\ — & \mathbf{x}^{(2),T} & — \\ & \vdots & \\ — & \mathbf{x}^{(N),T} & — \end{bmatrix} \in \mathbb{R}^{N \times D} \tag{3.45}$$

즉, 디자인 행렬 $\mathbf{X}$의 n번째 행은 n번째 데이터의 입력 벡터 $\mathbf{x}^{(n)} \in \mathbb{R}^D$가 됩니다. 행렬의 행의 개수는 데이터의 개수 N과 같고, 열의 개수는 입력의 차원 D와 같습니다.[3.2]

마찬가지로 목표값도 개별 스칼라들을 하나의 벡터로 모아서 표기할 수 있습니다. N개의 목표값 $y^{(1)}, y^{(2)}, \ldots, y^{(N)}$을 모은 벡터를 $\mathbf{y}$라고 합시다.

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} \in \mathbb{R}^N \tag{3.46}$$

이렇게 디자인 행렬 $\mathbf{X}$와 목표값 벡터 $\mathbf{y}$를 정의하고 나면, 전체 학습 데이터셋에 대한 모델의 예측을 한 번에 표현할 수 있습니다. n번째 데이터에 대한 예측은 $\hat{y}^{(n)} = \boldsymbol{\theta}^T \mathbf{x}^{(n)}$였습니다. 이를 모든 샘플에

[3.2] 앞서 약속한 대로, 여기서 D는 편향을 위한 $x_0 = 1$이 포함된 차원입니다.

대해 한꺼번에 행렬 곱으로 표현하면 다음과 같습니다.

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta}^T \mathbf{x}^{(1)} \\ \boldsymbol{\theta}^T \mathbf{x}^{(2)} \\ \vdots \\ \boldsymbol{\theta}^T \mathbf{x}^{(N)} \end{bmatrix} \tag{3.47}$$

$$= \begin{bmatrix} — & \mathbf{x}^{(1),T} & — \\ — & \mathbf{x}^{(2),T} & — \\ & \vdots & \\ — & \mathbf{x}^{(N),T} & — \end{bmatrix} \begin{bmatrix} | \\ \boldsymbol{\theta} \\ | \end{bmatrix} \tag{3.48}$$

$$= \mathbf{X}\boldsymbol{\theta} \tag{3.49}$$

여기서 $\hat{\mathbf{y}} \in \mathbb{R}^N$은 N개의 예측값 $\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(N)}$을 모은 벡터입니다. 디자인 행렬 $\mathbf{X}$의 n번째 행벡터 $\mathbf{x}^{(n),T}$와 파라미터 벡터 $\boldsymbol{\theta}$의 행렬곱이 정확히 n번째 행의 결과로 $\hat{y}^{(n)}$을 만들어내기 때문입니다.

예제 3.3. 앞서 다룬 부동산 가격 예측 문제로 돌아가봅시다. 입력은 면적 x_1과 방의 개수 x_2였고, 학습 데이터셋은 다음과 같았습니다.

n	면적 $x_1^{(n)}$	방 개수 $x_2^{(n)}$	실제 거래가 $y^{(n)}$
1	80	3	320
2	60	2	250
3	100	4	410

편향을 위한 더미 성분 $x_0 = 1$을 포함하면, n번째 데이터의 입력은 $\mathbf{x}^{(n)} = \begin{bmatrix} 1 & x_1^{(n)} & x_2^{(n)} \end{bmatrix}^T$가 됩니다. 이 세 데이터의 입력을 행벡터로 쌓아 디자인 행렬 $\mathbf{X}$를 만들면 다음과 같습니다.

$$\mathbf{X} = \begin{bmatrix} 1 & 80 & 3 \\ 1 & 60 & 2 \\ 1 & 100 & 4 \end{bmatrix} \in \mathbb{R}^{3 \times 3} \tag{3.50}$$

목표값들을 모은 벡터 $\mathbf{y}$는

$$\mathbf{y} = \begin{bmatrix} 320 \\ 250 \\ 410 \end{bmatrix} \in \mathbb{R}^3 \tag{3.51}$$

입니다. 이제 파라미터 $\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 \end{bmatrix}^T = \begin{bmatrix} 50 & 3 & 10 \end{bmatrix}^T$에 대한 예측값을 $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$로 한 번에

계산해봅시다.

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta} = \begin{bmatrix} 1 & 80 & 3 \\ 1 & 60 & 2 \\ 1 & 100 & 4 \end{bmatrix} \begin{bmatrix} 50 \\ 3 \\ 10 \end{bmatrix} \tag{3.52}$$

$$= \begin{bmatrix} 1 \times 50 + 80 \times 3 + 3 \times 10 \\ 1 \times 50 + 60 \times 3 + 2 \times 10 \\ 1 \times 50 + 100 \times 3 + 4 \times 10 \end{bmatrix} \tag{3.53}$$

$$= \begin{bmatrix} 320 \\ 250 \\ 390 \end{bmatrix} \tag{3.54}$$

이 결과는 앞선 예제에서 각 데이터에 대해 하나씩 계산했던 예측값 $\hat{y}^{(1)} = 320, \hat{y}^{(2)} = 250, \hat{y}^{(3)} = 390$과 정확히 일치합니다. 다만 이번에는 N개의 예측을 각각 따로 계산하지 않고, 행렬 곱 한 번으로 모두 구했다는 차이가 있습니다. ■

최소 제곱법(least squares). 그러면 우리가 최소로 만들고 싶은 SSE 비용 함수 $J(\boldsymbol{\theta})$를 앞서 정의한 디자인 행렬 $\mathbf{X}$와 예측값 벡터 $\hat{\mathbf{y}}$, 목표값 벡터 $\mathbf{y}$를 이용하여 다시 정리하여 써보겠습니다. 여기에는 앞서 다루었던 벡터의 크기를 이용해봅니다.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} (y^{(n)} - \hat{y}^{(n)})^2 \tag{3.55}$$

$$= \left\| \begin{bmatrix} y^{(1)} - \hat{y}^{(1)} \\ y^{(2)} - \hat{y}^{(2)} \\ \vdots \\ y^{(N)} - \hat{y}^{(N)} \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} - \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(N)} \end{bmatrix} \right\|^2 \tag{3.56}$$

$$= \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 \tag{3.57}$$

학습 데이터셋의 모든 예측값과 목표값을 하나의 벡터로 나타낼 수 있기 때문에 전체 데이터셋에 대한 합 $\sum_{n=1}^{N}(\cdot)^2$ 대신에 벡터의 크기 $\|\cdot\|^2$을 이용하여 표현할 수 있었습니다. 여기에 예측값 벡터를 디자인 행렬을 통해 표현하여 최종적인 식을 얻었습니다.

이제 우리의 목표를 다시 정리해봅시다. 주어진 학습 데이터셋을 선형 모델로 잘 표현해내는 모델 파라미터를 찾기 위해 SSE 비용 함수 $J(\boldsymbol{\theta})$를 정의하였고 이를 최소로 만드는 파라미터 $\boldsymbol{\theta}$를 찾고자 합니다. 이는 다음과 같이 표기할 수 있습니다.

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 \tag{3.58}$$

이제부터 일반적인 모델 파라미터를 쓸 때는 $\boldsymbol{\theta}$라 쓰고, 그 중에서 SSE 비용을 최소로 만들 때의 모델 파라미터는 $\hat{\boldsymbol{\theta}}$로 씁니다. 이렇게 목표값과 예측값 사이의 오차의 제곱을 최소화하여 파라미터를 구하는

방법을 최소 제곱법(least squares)이라고 합니다. 최소 제곱법으로 구한 파라미터 $\hat{\boldsymbol{\theta}}$는, 주어진 학습 데이터셋에 대해 SSE 비용을 가장 작게 만드는 파라미터, 즉 SSE 비용 관점에서 데이터를 가장 잘 설명하는 선형 모델의 파라미터가 됩니다.

이제 남은 문제는 이 $\hat{\boldsymbol{\theta}}$를 실제로 어떻게 구할 것인가입니다. 다음 소절에서는 기하학적 접근으로 이 문제를 풀어보겠습니다. 미분을 이용하는 해석학적 접근은 부록 A.3에 실어 두었습니다.

3.3.2 Geometric Approach

다음과 같이 주어진 SSE 비용 함수를 살펴봅시다.

$$J(\boldsymbol{\theta}) = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 \tag{3.59}$$

$$= \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 \tag{3.60}$$

$$\hat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{3.61}$$

$$= \arg\min_{\boldsymbol{\theta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 \tag{3.62}$$

이 문제는 이제 완전히 선형 대수 문제로 바뀌었습니다. 여기서 예측값 벡터 $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$는 디자인 행렬의 열벡터들의 선형 조합으로 표현됩니다. 선형 대수에서는 이를 $\mathbf{X}$의 열공간(column space)에 속한다고도 말하는 것을 앞서 살펴본 바 있습니다.

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta} \tag{3.63}$$

$$= \begin{bmatrix} | & | & | & & | \\ \mathbf{1} & \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_D \\ | & | & | & & | \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_D \end{bmatrix} \tag{3.64}$$

$$= \theta_0\mathbf{1} + \theta_1\mathbf{x}_1 + \theta_2\mathbf{x}_2 + \cdots + \theta_D\mathbf{x}_D \tag{3.65}$$

여기서 디자인 행렬 $\mathbf{X}$의 각 열벡터를 $\mathbf{x}_d$로 표현했습니다. 디자인 행렬은 전체 입력 벡터들을 행벡터로 넣어서 세로로 쌓았기 때문에, 열벡터 $\mathbf{x}_d$에는 모든 입력의 d번째 성분들만 모여있는 벡터가 될 것입니다. 편향 θ_0를 포함하였기 때문에, 디자인 행렬의 가장 첫 열벡터는 1로만 가득차있는 벡터, 즉 일벡터(one-vector)가 될 것입니다.

그러면 이와 같은 관계 속에서 $\mathbf{y} - \hat{\mathbf{y}}$의 크기를 최소화 한다는 의미는 곧 $\mathbf{X}$의 열공간 밖에 있는 목표값 벡터 $\mathbf{y}$에 대해, 열공간 내에서 $\mathbf{y}$까지의 거리가 가장 짧아지는 지점 $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$를 찾는 문제가 됩니다. 이는 곧 목표값 벡터 $\mathbf{y}$를 열공간에 수직으로 투영(projection)시킨 벡터가 예측값 벡터 $\hat{\mathbf{y}}$가 된다는 것을 말합니다. 그림으로 표현하면 그림 3.4와 같습니다.

그러므로 이를 정리하면 오차 벡터 $\mathbf{y} - \hat{\mathbf{y}}$는 디자인 행렬 $\mathbf{X}$의 열공간 $\mathrm{col}(\mathbf{X})$에 항상 수직이라는 얘기로 풀어낼 수 있습니다. 벡터 $\mathbf{y}$를 열공간 $\mathrm{col}(\mathbf{X})$에 투영시켰기 때문에, 벡터 $\mathbf{y}$와 투영된 지점

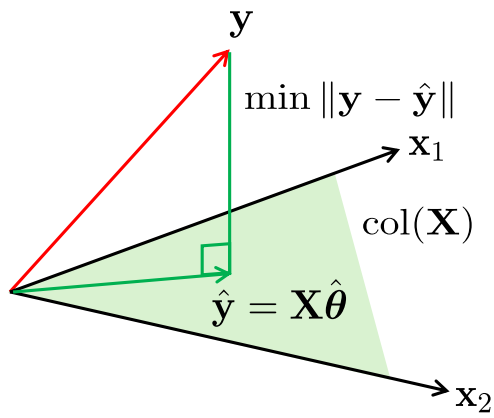

Figure 3.4: 목표값 벡터 $\mathbf{y}$를 디자인 행렬의 열공간 $\mathrm{col}(\mathbf{X})$에 수직으로 투영한 것이 예측값 벡터 $\hat{\mathbf{y}}$가 됩니다.

$\hat{\mathbf{y}}$을 연결하는 $\mathbf{y} - \hat{\mathbf{y}}$는 열공간 $\mathrm{col}(\mathbf{X})$에 직교합니다. 또한 이는 곧 열공간을 생성한 디자인 행렬의 열벡터들과도 직교해야 합니다.

$$\mathbf{x}_d \perp (\mathbf{y} - \hat{\mathbf{y}}) \Longleftrightarrow \mathbf{x}_d^T(\mathbf{y} - \hat{\mathbf{y}}) = 0 \tag{3.66}$$

$$(d = 1, 2, \ldots, D) \tag{3.67}$$

전체 열벡터에 대해 위의 관계가 성립하므로 $\hat{\boldsymbol{\theta}}$가 만족해야하는 조건을 아래와 같이 정리할 수 있습니다.

$$\mathbf{X}^T(\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{0} \tag{3.68}$$

대입하여 정리하면

$$\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\theta}}) = \mathbf{0} \tag{3.69}$$

$$\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\hat{\boldsymbol{\theta}} = \mathbf{0} \tag{3.70}$$

와 같습니다. 최종적으로 역행렬을 취해 정리하면

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.71}$$

입니다. 그러므로 우리는 학습 데이터셋으로부터 정의한 디자인 행렬 $\mathbf{X}$와 목표값 벡터 $\mathbf{y}$를 가지고 계산하여 SSE 비용을 최소로 만드는 $\hat{\boldsymbol{\theta}}$를 구할 수 있습니다. 우변에 있는 값들을 학습 데이터셋에서 가져올 수 있다는 점을 명심합시다.

예제 3.4. 광고비 x (단위: 백만 원)와 매출액 y (단위: 백만 원) 사이의 관계를 선형 모델로 학습하는 문제를 생각해봅시다. 학습 데이터셋은 다음과 같습니다.

n	광고비 $x^{(n)}$	매출액 $y^{(n)}$
1	1	16
2	2	19
3	3	24
4	4	31

편향을 포함한 디자인 행렬 $\mathbf{X}$와 목표값 벡터 $\mathbf{y}$는 다음과 같습니다.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \in \mathbb{R}^{4\times 2}, \qquad \mathbf{y} = \begin{bmatrix} 16 \\ 19 \\ 24 \\ 31 \end{bmatrix} \in \mathbb{R}^{4} \tag{3.72}$$

기하학적 접근에서 살펴보았듯이, 예측값 벡터 $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\theta}$는 $\mathbf{X}$의 열공간에 속합니다. 즉, $\hat{\mathbf{y}}$는 열벡터 $\mathbf{x}_0 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^T$와 $\mathbf{x}_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}^T$의 선형 조합으로 표현됩니다. 우리가 확인할 것은 목표값 벡터 $\mathbf{y}$가 이 열공간 밖에 있고, $\hat{\boldsymbol{\theta}}$로 계산된 예측값 $\hat{\mathbf{y}}$가 $\mathbf{y}$의 열공간으로의 투영이라는 점입니다.

먼저 $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$를 계산해봅시다.

$$\mathbf{X}^T\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 10 \\ 10 & 30 \end{bmatrix} \tag{3.73}$$

$$\mathbf{X}^T\mathbf{y} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} 16 \\ 19 \\ 24 \\ 31 \end{bmatrix} = \begin{bmatrix} 90 \\ 250 \end{bmatrix} \tag{3.74}$$

$$(\mathbf{X}^T\mathbf{X})^{-1} = \frac{1}{4\times 30 - 10\times 10} \begin{bmatrix} 30 & -10 \\ -10 & 4 \end{bmatrix} = \frac{1}{20} \begin{bmatrix} 30 & -10 \\ -10 & 4 \end{bmatrix} \tag{3.75}$$

$$\hat{\boldsymbol{\theta}} = \frac{1}{20} \begin{bmatrix} 30 & -10 \\ -10 & 4 \end{bmatrix} \begin{bmatrix} 90 \\ 250 \end{bmatrix} = \frac{1}{20} \begin{bmatrix} 200 \\ 100 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \end{bmatrix} \tag{3.76}$$

이제 예측값 벡터를 구해봅시다.

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\theta}} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix} \begin{bmatrix} 10 \\ 5 \end{bmatrix} = \begin{bmatrix} 15 \\ 20 \\ 25 \\ 30 \end{bmatrix} \tag{3.77}$$

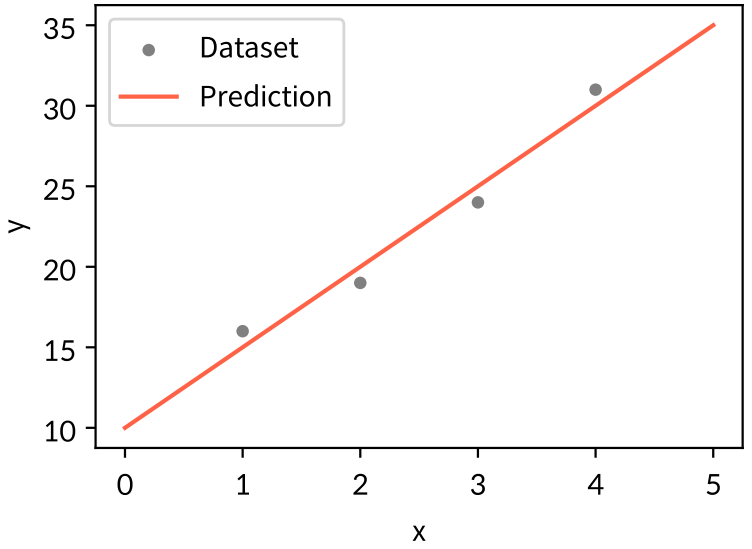

Figure 3.5: 광고비-매출액 데이터셋에 대한 최소 제곱법의 결과

마지막으로 오차 벡터 $\mathbf{y} - \hat{\mathbf{y}}$가 실제로 열공간에 수직인지 확인해봅시다.

$$\mathbf{y} - \hat{\mathbf{y}} = \begin{bmatrix} 16 \\ 19 \\ 24 \\ 31 \end{bmatrix} - \begin{bmatrix} 15 \\ 20 \\ 25 \\ 30 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \tag{3.78}$$

$$\mathbf{X}^T(\mathbf{y} - \hat{\mathbf{y}}) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1-1-1+1 \\ 1-2-3+4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.79}$$

$\mathbf{X}^T(\mathbf{y} - \hat{\mathbf{y}}) = \mathbf{0}$이 성립하므로, 오차 벡터는 $\mathbf{X}$의 열공간에 수직입니다. 즉, $\hat{\mathbf{y}}$는 목표값 벡터 $\mathbf{y}$를 열공간에 투영한 벡터임을 확인할 수 있습니다.

위 결과를 그림으로 나타내면 다음과 같습니다. 샘플들이 정확히 직선 위에 놓여 있지 않기 때문에 오차는 피할 수 없지만, 최소 제곱법으로 구한 직선은 SSE 비용을 최소로 만드는 직선임을 알 수 있습니다. 실선은 $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$로 구한 선형 모델의 예측을 나타내며, 각 데이터 포인트와 직선 사이의 수직 거리가 오차 $e^{(n)}$에 해당합니다.

아래 코드는 위의 계산 과정을 Python으로 구현한 것입니다. `numpy`의 행렬 연산을 이용하여 $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$를 직접 계산하고, 그 결과를 그림으로 나타냅니다.

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

x = np.array([1, 2, 3, 4])
y = np.array([16, 19, 24, 31])

X = np.column_stack([np.ones(len(x)), x])
theta = np.linalg.inv(X.T @ X) @ X.T @ y

x_line = np.linspace(0, 5, 100)
y_line = theta[0] + theta[1] * x_line

plt.figure()
plt.scatter(x, y, color='gray', s=12, label='Dataset')
plt.plot(x_line, y_line, color='tomato', label='Prediction')
```

■

3.3.3 Normal Equation

정규방정식(normal equation). 앞서 기하학적 접근을 통해 얻어진 식

$$\mathbf{X}^T\mathbf{X}\hat{\boldsymbol{\theta}} = \mathbf{X}^T\mathbf{y} \tag{3.80}$$

을 정규방정식(normal equation)이라고 합니다. 오차 벡터가 열공간에 수직이어야 한다는 조건, 즉 "normal"하다는 조건에서 비롯된 이름입니다. 이 방정식은 $\mathbf{X}^T\mathbf{X}$가 역행렬을 가질 때, 해가 유일하게 정해지며 그 해가 곧 SSE 비용을 최소로 만드는 파라미터

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.81}$$

입니다.

닫힌 해(closed-form solution). 우리가 앞서 구한 $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$를 닫힌 해(closed-form solution)라고 부릅니다. 닫힌 해란 주어진 값들을 수식에 대입하는 것만으로 단번에 답을 얻을 수 있는 형태의 해를 말합니다. 즉, 반복적인 계산 없이 데이터로부터 $\mathbf{X}$와 $\mathbf{y}$를 구성하고 행렬 연산을 수행하면 곧바로 SSE 비용을 최소로 만드는 파라미터를 얻을 수 있습니다. 이는 뒤에서 다룰 경사 하강법처럼 초기값에서 출발하여 반복적으로 파라미터를 수정해가며 해에 접근하는 방식과 대비됩니다. 이 장을 닫힌 해와 반복적 해법의 두 절로 나눈 것도 바로 이러한 대비를 반영한 것입니다.

Linear Regression: Closed-Form Solution

주어진 학습 데이터셋

$$\mathbf{X} = \begin{bmatrix} — & \mathbf{x}^{(1),T} & — \\ — & \mathbf{x}^{(2),T} & — \\ & \vdots & \\ — & \mathbf{x}^{(N),T} & — \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} \in \mathbb{R}^N \tag{3.82}$$

비용 함수를 최소화하는 닫힌 해

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{3.83}$$

의사역행렬(pseudo-inverse). 여기서 $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$를 디자인 행렬 $\mathbf{X}$의 의사역행렬(pseudo-inverse)이라고 하며, $\mathbf{X}^+$로 표기합니다.

$$\mathbf{X}^+ = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T \tag{3.84}$$

의사역행렬은 정방 행렬이 아닌 행렬에 대해 역행렬의 역할을 하는 행렬입니다. 실제로 $\mathbf{X}^+\mathbf{X} = \mathbf{I}$가 성립합니다.[3.3] 그러므로 최소 제곱법의 해를 간결하게

$$\hat{\boldsymbol{\theta}} = \mathbf{X}^+\mathbf{y} \tag{3.85}$$

로 쓸 수 있습니다.

예제 3.5. 2차원 입력에 대한 선형 회귀 모델을 구해봅시다. 손으로 풀기는 어렵기 때문에 Python을 이용해서 데이터셋을 생성하고 모델 파라미터를 구해보겠습니다. 여기서 사용하는 선형 모델은 아래와 같습니다.

$$y = 1.0 + 2.0x_1 - 1.5x_2 \tag{3.86}$$

코드는 여기에 가우시안 노이즈를 섞어서 실제 학습 데이터셋의 샘플들을 생성합니다. 그 후, 디자인 행렬을 만들고 정규 방정식으로 최소 제곱법의 해를 구합니다.

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

rng = np.random.default_rng(42)

n = 200
```

[3.3] 단, $\mathbf{X}\mathbf{X}^+ = \mathbf{I}$는 일반적으로 성립하지 않습니다. 이는 일반 역행렬에서 $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$가 성립하는 것과 다릅니다.

```
7
8  # 2D input samples
9  x1 = rng.uniform(-3, 3, n)
10 x2 = rng.uniform(-3, 3, n)
11
12 # Output = linear model + Gaussian noise
13 w0, w1, w2 = 1.0, 2.0, -1.5
14 noise = rng.normal(0, 1.5, n)
15 y = w0 + w1 * x1 + w2 * x2 + noise
16
17 # Least squares
18 X = np.column_stack([np.ones(n), x1, x2])
19 theta = np.linalg.inv(X.T @ X) @ X.T @ y
20
21 fig = plt.figure()
22 ax = fig.add_subplot(111, projection='3d')
23
24 ax.scatter(x1, x2, y, color='gray', s=10, alpha=0.6, label='Dataset
       ')
25
26 # Hyperplane from theta
27 g1, g2 = np.meshgrid(np.linspace(-3, 3, 30), np.linspace(-3, 3, 30)
       )
28 g_y = theta[0] + theta[1] * g1 + theta[2] * g2
29 ax.plot_surface(g1, g2, g_y, alpha=0.4, color='tomato', edgecolor='
       none', label='Prediction')
```

■

3.4 Iterative Solution

3.4.1 Gradient Descent

앞 절에서 우리는 닫힌 해를 얻었습니다. 식 하나로 답이 단번에 나오는데, 왜 다른 방법이 더 필요할까요? 닫힌 해가 언제나 통하는 것은 아니기 때문입니다. 먼저 $(\mathbf{X}^T\mathbf{X})^{-1}$의 계산량이 문제가 됩니다. 역행렬을 구하는 연산은 저렴하지 않습니다. 역행렬의 연산량은 입력 차원 D의 세제곱에 비례하여 늘어나기 때문에, 특성이 수만 개에 이르는 문제에서는 역행렬 계산 자체가 병목이 됩니다. 또한 데이터 개수 N이 아주 커지면 디자인 행렬을 통째로 메모리에 올리는 것부터 부담이 됩니다. 더 근본적인 한계도 있습니다. 닫힌 해는 SSE 비용과 선형 모델의 조합이 만들어준 행운에 가깝습니다. 앞으로 만나게 될 대부분의 모델과 비용 함수에서는 닫힌 해가 아예 존재하지 않습니다. 그래서 우리는 단번에

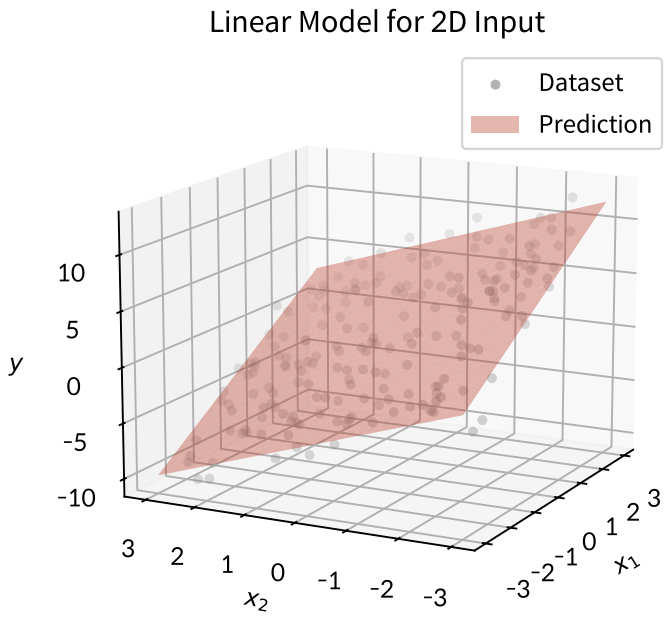

Figure 3.6: 입력이 2차원일 때의 선형 회귀 모델입니다. 입력이 하나일 때 모델이 직선이었다면, 입력이 둘이 되면 모델은 평면이 됩니다. 최소 제곱법은 각 샘플에서 이 평면까지의 세로 방향 거리를 제곱해 모두 더한 값을 가장 작게 만드는 평면을 찾습니다.

답을 구하는 대신, 임의의 위치에서 출발하여 근사적이고 반복적으로 조금씩 해에 다가가는 방법을 사용합니다.

경사 하강법(gradient descent). 경사 하강법이란 어떤 주어진 함수 $J(\boldsymbol{\theta})$의 최소 위치를 그래디언트를 이용해서 근사적으로 찾아가는 방법입니다. 경사 하강법을 설명하면 다음과 같습니다. 임의의 입력값 $\boldsymbol{\theta}^{(0)}$에서 출발합니다. 지금 위치에서의 그래디언트 $\nabla J(\boldsymbol{\theta}^{(0)})$를 구하여 그 반대 방향으로 η배 하여 이동합니다. 식으로 나타내면 다음과 같습니다.

$$\boldsymbol{\theta}^{(1)} = \boldsymbol{\theta}^{(0)} - \eta \nabla J(\boldsymbol{\theta}^{(0)}) \tag{3.87}$$

이제 현재 위치는 $\boldsymbol{\theta}^{(1)}$가 되었습니다. 다시 위 과정을 반복합니다. 이 자리에서 그래디언트 $\nabla J(\boldsymbol{\theta}^{(1)})$을 구하고 η배 하여 다음 위치 $\boldsymbol{\theta}^{(2)} = \boldsymbol{\theta}^{(1)} - \eta \nabla J(\boldsymbol{\theta}^{(1)})$로 이동합니다. 이를 매번 반복하며 한발짝 한발짝씩 함수 $J(\boldsymbol{\theta})$가 감소하는 방향으로 나아가는 겁니다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla J(\boldsymbol{\theta}^{(k)}) \tag{3.88}$$

이처럼 한 번에 답을 구하지 않고 갱신을 거듭하며 해에 다가가는 방법을 반복법(iterative method)이라고 부릅니다. 경사 하강법은 반복법의 대표적인 예입니다.

그래디언트는 항상 주어진 위치에서 함수가 가장 가파르게 증가하는 방향을 가리키고 있습니다. 그렇기 때문에 함수가 작아지는 방향인 그래디언트의 반대 방향으로 점차적으로 진행해 나가게 됩니다. 이 때, 그래디언트 앞에 곱한 스칼라 η를 학습률(learning rate)이라고 부릅니다. 학습률을 조절하여 한번에 내딛는 발자국의 너비를 조절할 수 있습니다. 학습률이 크면 최소 위치로 빠르게 수렴할 수 있지만 매 단계마다 변화가 매우 크게 나타나며 심지어 발산하는 경우도 발생할 수 있습니다. 반대로 학습률이 작으면 천천히 매끄럽게 수렴하지만 그만큼 많은 반복이 필요합니다. 또한 보폭이 작은 만큼, 도중에 지역 최솟값(local minimum)을 만나면 빠져나오기 어렵습니다.

여기서 눈여겨볼 점은, 경사 하강법의 어디에도 선형 모델이나 SSE 비용을 강제하는 부분이 없다는 것입니다. 필요한 것은 그래디언트를 구할 수 있느냐뿐입니다. 비용 함수가 파라미터에 대해 미분 가능하기만 하다면 똑같은 방법을 적용할 수 있습니다. 닫힌 해가 모델과 비용의 조합이 좋을 때만 주어지는 행운이었다면, 경사 하강법은 그런 조건을 따지지 않는 일종의 범용 도구인 셈입니다. 실제로 이후 장에서 만나게 될 로지스틱 회귀부터 신경망에 이르기까지, 이 책에 등장하는 대부분의 모델은 바로 이 방법으로 학습됩니다. 오늘날 거대한 신경망을 학습시키는 방법도 본질적으로는 지금 살펴본 절차와 다르지 않습니다.

그러나 경사 하강법은 학습률과 관계없이 함수의 전역 최솟값(global minimum)의 위치를 찾는 것을 보장하지는 못합니다. 그 원리를 생각했을 때 주변에 지역 최솟값이 존재한다면 그곳으로 수렴할 수 있으며, 바꾸어 말하면 지금 수렴한 어떤 위치가 지역 최솟값인지 전역 최솟값인지 알 수 없다는 말이 됩니다. 반면 앞서 다루었던 닫힌 해의 경우는 디자인 행렬의 의사역행렬이 존재한다면 전역 최솟값을 보장받을 수 있습니다.[3.4]

[3.4] 다행히 지금 다루는 SSE 비용은 지역 최솟값이 따로 없는 모양이어서, 경사 하강법도 이 문제에서는 전역 최솟값을 찾아갑니다. 지역 최솟값의 문제는 이후 장에서 더 복잡한 모델을 만날 때 실제로 마주하게 될 것입니다.

예제 3.6. 함수 $J(\theta_1, \theta_2) = \theta_1^2 + 2\theta_2^2$의 최솟값을 경사 하강법으로 찾아봅시다. 학습률은 $\eta = 0.3$으로 하고, 초기값은 $\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$에서 출발합니다.

먼저 그래디언트를 구합니다.

$$\nabla J(\boldsymbol{\theta}) = \begin{bmatrix} \dfrac{\partial J}{\partial \theta_1} \\ \dfrac{\partial J}{\partial \theta_2} \end{bmatrix} = \begin{bmatrix} 2\theta_1 \\ 4\theta_2 \end{bmatrix} \tag{3.89}$$

1단계: 현재 위치 $\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$에서 그래디언트를 구하고 이동합니다.

$$\nabla J(\boldsymbol{\theta}^{(0)}) = \begin{bmatrix} 2 \times 2 \\ 4 \times 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix} \tag{3.90}$$

$$\boldsymbol{\theta}^{(1)} = \begin{bmatrix} 2 \\ 2 \end{bmatrix} - 0.3 \begin{bmatrix} 4 \\ 8 \end{bmatrix} = \begin{bmatrix} 2 - 1.2 \\ 2 - 2.4 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.4 \end{bmatrix} \tag{3.91}$$

2단계: 현재 위치 $\boldsymbol{\theta}^{(1)} = \begin{bmatrix} 0.8 \\ -0.4 \end{bmatrix}$에서 그래디언트를 구하고 이동합니다.

$$\nabla J(\boldsymbol{\theta}^{(1)}) = \begin{bmatrix} 2 \times 0.8 \\ 4 \times (-0.4) \end{bmatrix} = \begin{bmatrix} 1.6 \\ -1.6 \end{bmatrix} \tag{3.92}$$

$$\boldsymbol{\theta}^{(2)} = \begin{bmatrix} 0.8 \\ -0.4 \end{bmatrix} - 0.3 \begin{bmatrix} 1.6 \\ -1.6 \end{bmatrix} = \begin{bmatrix} 0.8 - 0.48 \\ -0.4 + 0.48 \end{bmatrix} = \begin{bmatrix} 0.32 \\ 0.08 \end{bmatrix} \tag{3.93}$$

이 함수의 실제 최솟값은 $\nabla J = \mathbf{0}$을 풀면 $\boldsymbol{\theta} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ 임을 알 수 있습니다. 경사 하강법은 $\begin{bmatrix} 2 \\ 2 \end{bmatrix} \to \begin{bmatrix} 0.8 \\ -0.4 \end{bmatrix} \to \begin{bmatrix} 0.32 \\ 0.08 \end{bmatrix} \to \cdots$ 로 반복할수록 점점 원점에 가까워지고 있음을 확인할 수 있습니다. ■

SSE 비용의 경사 하강법. 먼저 SSE 비용의 그래디언트를 구해봅시다.

$$\begin{aligned} \nabla J(\boldsymbol{\theta}) &= \nabla \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 && (3.94) \\ &= \nabla (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T (\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) && (3.95) \\ &= \nabla (\mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\boldsymbol{\theta} + \boldsymbol{\theta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}) && (3.96) \\ &= 2(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}) && (3.97) \end{aligned}$$

셋째 줄에서 넷째 줄로 넘어가는 미분이 낯설 수 있습니다. 벡터 미분을 정식으로 다루지는 않겠지만, 스칼라에서의 미분과 견주어 보면 결과를 받아들이기 어렵지 않습니다. 스칼라 함수 $J(\theta) =$

$y^2 - 2yx\theta + x^2\theta^2$을 θ로 미분하면 $-2xy + 2x^2\theta = 2(x^2\theta - xy)$가 됩니다. 위 넷째 줄은 이 결과에서 x^2이 $\mathbf{X}^T\mathbf{X}$로, xy가 $\mathbf{X}^T\mathbf{y}$로 바뀐 것과 정확히 같은 모양입니다. 상수인 첫 항은 미분하면 사라지고, $\boldsymbol{\theta}$의 일차 항은 계수만 남고, 이차 항은 2가 앞으로 나온다는 익숙한 규칙이 벡터에서도 같은 모양으로 성립하는 것입니다.[3.5]

앞서 구한 SSE 비용의 그래디언트를 이용한 경사 하강법은 다음과 같습니다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla J(\boldsymbol{\theta}^{(k)}) \tag{3.98}$$

$$= \boldsymbol{\theta}^{(k)} - 2\eta(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}^{(k)} - \mathbf{X}^T\mathbf{y}) \tag{3.99}$$

$$= \boldsymbol{\theta}^{(k)} - \eta^*(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}^{(k)} - \mathbf{X}^T\mathbf{y}) \tag{3.100}$$

이 때, 앞에 붙는 계수에는 우리가 조절하는 학습률이 포함되어 있으므로, 계수를 $\eta^* = 2\eta$로 정의하여 하나로 묶어버릴 수 있습니다. 이후로는 표기를 간단히 하여 η^*를 다시 η로 쓰겠습니다.

예제 3.7. 다음 학습 데이터셋에 대한 선형 회귀 문제를 경사 하강법으로 풀어봅시다.

n	$x_1^{(n)}$	$x_2^{(n)}$	$y^{(n)}$
1	1	0	3
2	0	1	4
3	1	1	6
4	2	1	8

디자인 행렬과 목표값 벡터는 다음과 같습니다.

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 3 \\ 4 \\ 6 \\ 8 \end{bmatrix} \tag{3.101}$$

학습률 $\eta = 0.05$, 초기 파라미터 $\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$에서 출발합니다. 먼저 필요한 행렬을 계산해둡니다.

$$\mathbf{X}^T\mathbf{X} = \begin{bmatrix} 4 & 4 & 3 \\ 4 & 6 & 3 \\ 3 & 3 & 3 \end{bmatrix}, \qquad \mathbf{X}^T\mathbf{y} = \begin{bmatrix} 21 \\ 25 \\ 18 \end{bmatrix} \tag{3.102}$$

[3.5] 이 그래디언트를 $\mathbf{0}$으로 놓고 풀면 앞 절의 정규방정식 $\mathbf{X}^T\mathbf{X}\hat{\boldsymbol{\theta}} = \mathbf{X}^T\mathbf{y}$가 그대로 나타납니다. 기하학적 접근으로 얻었던 결과를 미분으로도 똑같이 얻을 수 있는 것입니다. 자세한 전개는 부록 A.3을 참고하세요.

1단계: $\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ 에서 그래디언트를 구하고 이동합니다.

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}^{(0)} - \mathbf{X}^T\mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 21 \\ 25 \\ 18 \end{bmatrix} = \begin{bmatrix} -21 \\ -25 \\ -18 \end{bmatrix} \tag{3.103}$$

$$\boldsymbol{\theta}^{(1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.05 \begin{bmatrix} -21 \\ -25 \\ -18 \end{bmatrix} = \begin{bmatrix} 1.05 \\ 1.25 \\ 0.9 \end{bmatrix} \tag{3.104}$$

이 시점의 SSE 비용은 $J(\boldsymbol{\theta}^{(1)}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}^{(1)}\|^2 \approx 25.14$입니다.

2단계: $\boldsymbol{\theta}^{(1)} = \begin{bmatrix} 1.05 \\ 1.25 \\ 0.9 \end{bmatrix}$ 에서 그래디언트를 구하고 이동합니다.

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}^{(1)} - \mathbf{X}^T\mathbf{y} = \begin{bmatrix} 4 & 4 & 3 \\ 4 & 6 & 3 \\ 3 & 3 & 3 \end{bmatrix} \begin{bmatrix} 1.05 \\ 1.25 \\ 0.9 \end{bmatrix} - \begin{bmatrix} 21 \\ 25 \\ 18 \end{bmatrix} = \begin{bmatrix} -9.1 \\ -10.6 \\ -8.4 \end{bmatrix} \tag{3.105}$$

$$\boldsymbol{\theta}^{(2)} = \begin{bmatrix} 1.05 \\ 1.25 \\ 0.9 \end{bmatrix} - 0.05 \begin{bmatrix} -9.1 \\ -10.6 \\ -8.4 \end{bmatrix} = \begin{bmatrix} 1.505 \\ 1.78 \\ 1.32 \end{bmatrix} \tag{3.106}$$

이 시점의 SSE 비용은 $J(\boldsymbol{\theta}^{(2)}) \approx 6.02$로, 1단계보다 더 줄어들었습니다. 이와 같은 과정을 계속 반복하면 파라미터는 점차 SSE 비용을 최소로 만드는 지점을 향해 나아갑니다. ■

3.4.2 Batch, Mini-Batch and Stochastic Gradient Descent

배치 경사 하강법(batch gradient descent). SSE 비용의 경사 하강법을 다시 봅시다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta(\mathbf{X}^T\mathbf{X}\boldsymbol{\theta}^{(k)} - \mathbf{X}^T\mathbf{y}) \tag{3.107}$$

여기서 그래디언트 항을 다시 풀어서 써보겠습니다.

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y} = \mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \mathbf{X}^T(\hat{\mathbf{y}} - \mathbf{y}) \tag{3.108}$$

$$= \begin{bmatrix} | & | & & | \\ \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(N)} \\ | & | & & | \end{bmatrix} \begin{bmatrix} \boldsymbol{\theta}^T\mathbf{x}^{(1)} - y^{(1)} \\ \boldsymbol{\theta}^T\mathbf{x}^{(2)} - y^{(2)} \\ \vdots \\ \boldsymbol{\theta}^T\mathbf{x}^{(N)} - y^{(N)} \end{bmatrix} \tag{3.109}$$

$$= \sum_{n=1}^{N} \mathbf{x}^{(n)}(\boldsymbol{\theta}^T\mathbf{x}^{(n)} - y^{(n)}) = \sum_{n=1}^{N} \mathbf{x}^{(n)}(\hat{y}^{(n)} - y^{(n)}) \tag{3.110}$$

즉 그래디언트는 각 입력 샘플 $\mathbf{x}^{(n)}$을 예측값과 목표값의 차이만큼 가중하여 더한 선형 조합입니다. 이와 같이 한번 반복될 때 데이터셋의 모든 샘플을 이용해서 그래디언트를 구하는 방법을 배치 경사 하강법이라고 합니다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \sum_{n=1}^{N} \mathbf{x}^{(n)}(\hat{y}^{(n)} - y^{(n)}) \tag{3.111}$$

Linear Regression: Batch Gradient Descent

배치 경사 하강법

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \sum_{n=1}^{N} \mathbf{x}^{(n)}(\hat{y}^{(n)} - y^{(n)}) \tag{3.112}$$

확률적 경사 하강법(stochastic gradient descent). 반면에 한번 반복할 때 하나의 샘플만 뽑아 사용하는 것을 확률적 경사 하강법이라고 합니다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \mathbf{x}^{(n)}(\hat{y}^{(n)} - y^{(n)}) \tag{3.113}$$

여기서 n은 매 반복마다 무작위로 새로 뽑는 샘플의 인덱스이며, $\hat{y}^{(n)}$은 현재 파라미터 $\boldsymbol{\theta}^{(k)}$로 계산한 그 샘플의 예측값입니다.

Linear Regression: Stochastic Gradient Descent

확률적 경사 하강법

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \mathbf{x}^{(n)}(\hat{y}^{(n)} - y^{(n)}) \tag{3.114}$$

위의 식을 한번 음미하는 시간을 가져봅시다. 예를 들어, 모델 파라미터 $\boldsymbol{\theta}$가 어떤 샘플 $\mathbf{x}'$에 대한 예측을 목표값보다 더 크게 $\hat{y}' = \boldsymbol{\theta}^T\mathbf{x}' > y'$ 했다고 합시다. 그러면 우리는 파라미터를 잘 조절해서 이 샘플에 대한 예측값을 조금 낮춰야 할 것입니다. 그러면 새로운 파라미터를 $\boldsymbol{\theta} - \eta\mathbf{x}'$로 살짝 조정하면 어떤 일이 일어날까요? 원래의 예측값 $\hat{y}' = \boldsymbol{\theta}^T\mathbf{x}'$과 비교하면 새로운 파라미터의 예측값은

$$(\boldsymbol{\theta} - \eta\mathbf{x}')^T\mathbf{x}' = \boldsymbol{\theta}^T\mathbf{x}' - \eta\|\mathbf{x}'\|^2 \tag{3.115}$$

$$= \hat{y}' - \eta\|\mathbf{x}'\|^2 \tag{3.116}$$

가 되면서 예측 결과가 원래 파라미터일 때보다 줄어들게 됩니다. 왜냐하면 파라미터에 샘플 성분을 조금 빼면서 생긴 뒤에 붙은 항은 항상 $\eta\|\mathbf{x}'\|^2 \geq 0$이기 때문입니다. 여기에 현재 예측이 어긋난 정도를 가중치로 두면 $\boldsymbol{\theta} - \eta\mathbf{x}'(\boldsymbol{\theta}^T\mathbf{x}' - y')$와 같은 꼴을 완성할 수 있습니다. 반대로 예측이 목표값보다 작았다면, 이번에는 괄호 안의 부호가 음수가 되어 파라미터에 샘플 성분을 더하는 방향으로 갱신이 일어나고, 예측값은 커지게 됩니다.

현재 상태	갱신 방향	예측값의 변화
$\hat{y}' > y'$ (과대 예측)	$\boldsymbol{\theta}$에서 $\mathbf{x}'$ 성분을 뺌	$\hat{y}'$ 감소
$\hat{y}' < y'$ (과소 예측)	$\boldsymbol{\theta}$에 $\mathbf{x}'$ 성분을 더함	$\hat{y}'$ 증가

예제 3.8. 선형 회귀 모델과 학습 데이터가 다음과 같이 주어졌다고 합시다. 입력에 편향을 위한 더미 성분 1을 포함하여 $\mathbf{x} = [\,1,\, x_1,\, x_2\,]^T$로 두고, 파라미터는 $\boldsymbol{\theta} = [\,\theta_0,\, \theta_1,\, \theta_2\,]^T$입니다.

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = \boldsymbol{\theta}^T \mathbf{x} \tag{3.117}$$

현재 파라미터는 $\boldsymbol{\theta} = [\,1,\, 1,\, 3\,]^T$입니다. 학습 데이터는 다음과 같습니다.

n	$x_1^{(n)}$	$x_2^{(n)}$	$y^{(n)}$
1	1	1	6
2	2	1	8
3	1	2	9
4	3	2	13
5	2	3	14

다음 물음에 답해 봅시다. 학습률은 $\eta = 0.01$입니다.

1. 현재 파라미터에서 SSE 비용을 계산하세요.
2. 배치 경사 하강법을 한 번 적용하여 갱신된 파라미터를 구하세요.
3. 현재 파라미터에서 다시 시작하여, 확률적 경사 하강법을 한 번 적용합니다. 선택된 샘플은 4번입니다. 갱신된 파라미터를 구하세요.

풀이. (1) 먼저 각 샘플의 예측값 $\hat{y}^{(n)} = \boldsymbol{\theta}^T \mathbf{x}^{(n)}$을 구합니다. 예컨대 1번 샘플은 $\hat{y}^{(1)} = 1 + 1 \cdot 1 + 3 \cdot 1 = 5$입니다. 모든 샘플에 대해 예측값과 목표값의 차이 $\hat{y}^{(n)} - y^{(n)}$을 구하면 다음과 같습니다.

n	$x_1^{(n)}$	$x_2^{(n)}$	$y^{(n)}$	$\hat{y}^{(n)}$	$\hat{y}^{(n)} - y^{(n)}$
1	1	1	6	5	-1
2	2	1	8	6	-2
3	1	2	9	8	-1
4	3	2	13	10	-3
5	2	3	14	12	-2

SSE 비용은 이 차이의 제곱을 모두 더한 것입니다.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{5} \left(\hat{y}^{(n)} - y^{(n)} \right)^2 = (-1)^2 + (-2)^2 + (-1)^2 + (-3)^2 + (-2)^2 = 19 \tag{3.118}$$

(2) 배치 경사 하강법은 다섯 샘플 전체를 사용합니다. 각 샘플의 입력에 예측값과 목표값의 차이를

곱해 모두 더하면

$$\sum_{n=1}^{5} \mathbf{x}^{(n)} \left(\hat{y}^{(n)} - y^{(n)} \right) = (-1) \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + (-2) \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} + (-1) \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix} + (-3) \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} + (-2) \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -9 \\ -19 \\ -17 \end{bmatrix} \tag{3.119}$$

입니다. 갱신식에 대입하면

$$\boldsymbol{\theta} \leftarrow \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} - 0.01 \begin{bmatrix} -9 \\ -19 \\ -17 \end{bmatrix} = \begin{bmatrix} 1.09 \\ 1.19 \\ 3.17 \end{bmatrix} \tag{3.120}$$

와 같이 갱신됩니다. (3) 확률적 경사 하강법은 뽑힌 4번 샘플 하나만 사용합니다. 4번 샘플의 차이는 -3, 입력은 $\mathbf{x}^{(4)} = [\,1,\ 3,\ 2\,]^T$ 이므로

$$\boldsymbol{\theta} \leftarrow \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} - 0.01 \times (-3) \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 1.03 \\ 1.09 \\ 3.06 \end{bmatrix} \tag{3.121}$$

와 같이 갱신됩니다. 두 방법 모두 차이가 음수인, 즉 예측이 목표값보다 작은 상황이므로 파라미터를 키워 예측을 목표값 쪽으로 끌어올리는 방향으로 움직입니다. 다만 배치 경사 하강법은 다섯 샘플의 정보를 모두 반영해 움직인 반면, 확률적 경사 하강법은 4번 샘플이 시키는 방향으로만 움직였다는 차이가 있습니다. ■

배치 경사 하강법과 확률적 경사 하강법의 비교. 배치 경사 하강법은 한번 움직일 때 학습 데이터셋의 모든 데이터를 종합하여 움직이기 때문에 비교적 안정적으로 수렴하지만 지역 최솟값 탈출이 어렵다는 단점이 있습니다. 반면 확률적 경사 하강법은 한번에 하나의 샘플을 사용하기 때문에 수렴이 매우 불규칙합니다. 그러나 지역 최솟값에서 벗어나기에는 유리합니다. 아래 표와 같이 그 특징을 비교할 수 있습니다.

	배치 경사 하강법	확률적 경사 하강법
갱신 기준	전체 샘플	단일 샘플
계산 비용	높음	낮음
수렴 안정성	안정	불안정
지역 최솟값 탈출	어려움	가능
대용량 데이터	느림	빠름

미니배치 경사 하강법(mini-batch gradient descent). 배치 경사 하강법과 확률적 경사 하강법의 중간 형태로, 한 번 반복할 때 전체 샘플도 단일 샘플도 아닌 B개의 샘플을 묶어서 사용하는 방법을 미니배치 경사 하강법이라고 합니다. 배치가 전체 묶음을 뜻하기 때문에, 작은 단위의 묶음들을 미니배

치(mini-batch)라고 하고, 이 때 미니배치 하나당 들어있는 샘플의 개수 B를 배치 크기(batch size)라고 합니다.

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \sum_{n \in \mathcal{B}_k} \mathbf{x}^{(n)} (\boldsymbol{\theta}^{(k),T} \mathbf{x}^{(n)} - y^{(n)}) \tag{3.122}$$

여기서 $\mathcal{B}_k$는 k번째 반복에서 무작위로 선택된 B개의 샘플 인덱스 집합입니다. 실제로는 매번 독립적으로 뽑기보다, 에포크마다 전체 샘플을 무작위로 섞은 뒤에 앞에서부터 B개씩 잘라 미니배치들을 만드는 방식을 주로 사용합니다. $B = N$으로 두면 배치 경사 하강법, $B = 1$로 두면 확률적 경사 하강법이 되므로, 미니배치 경사 하강법은 두 방법을 모두 포괄하는 좀 더 일반화된 형태입니다.

Linear Regression: Mini-Batch Gradient Descent

미니배치 경사 하강법

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \sum_{n \in \mathcal{B}_k} \mathbf{x}^{(n)} (\hat{y}^{(n)} - y^{(n)}) \tag{3.123}$$

실제 머신러닝에서는 미니배치 경사 하강법이 가장 널리 사용됩니다. GPU의 병렬 연산을 효율적으로 활용할 수 있고, 배치 경사 하강법보다 빠르면서 확률적 경사 하강법보다 안정적이기 때문입니다. 배치 크기는 보통 $32, 64, 128$처럼 2의 거듭제곱으로 설정합니다.

학습 데이터셋 전체를 한 번 모두 사용하는 것을 에포크(epoch)라고 합니다. 학습은 보통 수십에서 수백 에포크에 걸쳐 진행됩니다. 미니배치 경사 하강법에서는 전체 N개의 샘플을 배치 크기 B로 나누어 순서대로 사용하므로, 한 에포크에 $\lceil N/B \rceil$번의 파라미터 갱신이 이루어집니다. 여기서 $\lceil \cdot \rceil$은 올림 함수입니다.

예제 3.9. $N = 6$개의 샘플과 배치 크기 $B = 2$로 미니배치 경사 하강법을 수행한다고 합시다. 학습 데이터셋은 다음과 같습니다.

n	$x_1^{(n)}$	$x_2^{(n)}$	$y^{(n)}$
1	1	0	2
2	0	1	3
3	1	1	4
4	2	0	3
5	0	2	5
6	2	1	5

한 에포크에 $\lceil 6/2 \rceil = 3$번의 갱신이 이루어집니다. 샘플을 무작위로 섞은 후 순서대로 미니배치를 구성하면 다음과 같습니다.

갱신	미니배치 $\mathcal{B}_k$	사용되는 샘플
$k = 0$	$\mathcal{B}_0 = \{3, 1\}$	$\mathbf{x}^{(3)}, \mathbf{x}^{(1)}$
$k = 1$	$\mathcal{B}_1 = \{5, 2\}$	$\mathbf{x}^{(5)}, \mathbf{x}^{(2)}$
$k = 2$	$\mathcal{B}_2 = \{4, 6\}$	$\mathbf{x}^{(4)}, \mathbf{x}^{(6)}$

초기 파라미터 $\boldsymbol{\theta}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$, 학습률 $\eta = 0.1$로 첫 번째 갱신을 계산해봅시다.

$$\boldsymbol{\theta}^{(1)} = \boldsymbol{\theta}^{(0)} - \eta \sum_{n \in \mathcal{B}_0} \mathbf{x}^{(n)} (\boldsymbol{\theta}^{(0),T} \mathbf{x}^{(n)} - y^{(n)}) \tag{3.124}$$

$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.1 \left[\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} (0 - 4) + \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} (0 - 2) \right] \tag{3.125}$$

$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.1 \left[\begin{bmatrix} -4 \\ -4 \\ -4 \end{bmatrix} + \begin{bmatrix} -2 \\ -2 \\ 0 \end{bmatrix} \right] \tag{3.126}$$

$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.1 \begin{bmatrix} -6 \\ -6 \\ -4 \end{bmatrix} = \begin{bmatrix} 0.6 \\ 0.6 \\ 0.4 \end{bmatrix} \tag{3.127}$$

이렇게 한 에포크 안에서 미니배치 단위로 파라미터를 갱신해나갑니다. 두 번째와 세 번째 갱신도 동일한 방식으로 계산하면 다음과 같습니다.

$$\boldsymbol{\theta}^{(2)} = \boldsymbol{\theta}^{(1)} - \eta \sum_{n \in \mathcal{B}_1} \mathbf{x}^{(n)} (\boldsymbol{\theta}^{(1),T} \mathbf{x}^{(n)} - y^{(n)}) \tag{3.128}$$

$$= \begin{bmatrix} 0.6 \\ 0.6 \\ 0.4 \end{bmatrix} - 0.1 \begin{bmatrix} -5.6 \\ 0 \\ -9.2 \end{bmatrix} = \begin{bmatrix} 1.16 \\ 0.6 \\ 1.32 \end{bmatrix} \tag{3.129}$$

$$\boldsymbol{\theta}^{(3)} = \boldsymbol{\theta}^{(2)} - \eta \sum_{n \in \mathcal{B}_2} \mathbf{x}^{(n)} (\boldsymbol{\theta}^{(2),T} \mathbf{x}^{(n)} - y^{(n)}) \tag{3.130}$$

$$= \begin{bmatrix} 1.16 \\ 0.6 \\ 1.32 \end{bmatrix} - 0.1 \begin{bmatrix} -1.96 \\ -3.92 \\ -1.32 \end{bmatrix} = \begin{bmatrix} 1.356 \\ 0.992 \\ 1.452 \end{bmatrix} \tag{3.131}$$

한 에포크가 끝난 후 파라미터는 $\boldsymbol{\theta}^{(3)} \approx \begin{bmatrix} 1.36 \\ 0.99 \\ 1.45 \end{bmatrix}$ 입니다. ■

3.5 Practice with Implementation

3.5.1 Python Class

여기에서는 이 장에서 배운 내용을 직접 파이썬 클래스로 구현하고, 실제 데이터셋에 적용해봅니다. 먼저 선형 회귀 모델을 하나의 클래스로 만들어보겠습니다. 클래스로 구현해두면 서로 다른 데이터셋

에 대해 같은 코드를 재사용할 수 있고, 뒤에서 다룰 sklearn 라이브러리의 사용 방식과도 자연스럽게 연결됩니다.

```python
import numpy as np

class LinearRegression:
    def __init__(self):
        self.theta = None

    def _add_bias(self, X):
        # prepend a column of ones for the bias term
        return np.column_stack([np.ones(len(X)), X])

    def fit_closed_form(self, X, y):
        # closed-form solution: theta = (X^T X)^{-1} X^T y
        X = self._add_bias(X)
        self.theta = np.linalg.inv(X.T @ X) @ X.T @ y
        return self

    def fit_gd(self, X, y, eta=0.01, n_iters=1000):
        # batch gradient descent
        X = self._add_bias(X)
        N, D = X.shape
        self.theta = np.zeros(D)
        for _ in range(n_iters):
            grad = (X.T @ X @ self.theta - X.T @ y)
            self.theta = self.theta - eta * grad
        return self

    def predict(self, X):
        X = self._add_bias(X)
        return X @ self.theta
```

클래스의 구조를 하나씩 살펴봅시다. `_add_bias`는 입력 행렬의 첫 열에 1을 추가하여 편향을 포함한 디자인 행렬을 구성합니다. `fit_closed_form`은 정규방정식의 닫힌 해 $\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$를 그대로 코드로 옮긴 것입니다. `fit_gd`는 배치 경사 하강법으로, 매 반복마다 SSE 비용의 그래디언트 $\nabla J = \mathbf{X}^T\mathbf{X}\boldsymbol{\theta} - \mathbf{X}^T\mathbf{y}$를 계산하여 파라미터를 갱신합니다.[3.6] 마지막으로 `predict`는 학습된 파라미터로 새로운 입력에 대한 예측값을 계산합니다.

앞서 다뤘던 예제 데이터로 두 방법이 같은 해에 도달하는지 확인해봅시다.

[3.6] 그래디언트에서 앞에 곱해지는 계수를 생략한 이유는 앞선 이론 설명에서 다루고 있습니다.

```
X = np.array([[1, 0], [0, 1], [1, 1], [2, 1]], dtype=float)
y = np.array([3, 4, 6, 8], dtype=float)

model1 = LinearRegression().fit_closed_form(X, y)
print(model1.theta)  # [1. 2. 3.]

model2 = LinearRegression().fit_gd(X, y, eta=0.05, n_iters=1000)
print(model2.theta)  # [1. 2. 3.]
```

닫힌 해와 경사 하강법 모두 $\hat{\boldsymbol{\theta}} = [1, 2, 3]^T$ 라는 동일한 해에 도달합니다. 경사 하강법 예제에서 다뤘던 그 데이터셋이므로, 손으로 계산했던 반복 과정이 코드 안에서 1000번 수행된 셈입니다.

3.5.2 Kaggle: House Prices Dataset

클래스 적용. 이제 실제 데이터에 우리의 클래스를 적용해봅시다. 사용할 데이터는 Kaggle의 House Prices 데이터셋[3.7]으로, 미국 아이오와주 Ames 지역의 주택 1460채에 대한 실거래 데이터입니다. 79개의 특성 중에서 다섯 개의 수치형 특성을 사용하겠습니다. 지상 생활 면적(`GrLivArea`), 전반적 품질 등급(`OverallQual`), 지하실 면적(`TotalBsmtSF`), 차고 수용 대수(`GarageCars`), 건축 연도(`YearBuilt`)로 판매 가격(`SalePrice`)을 예측하는 문제입니다. 우리가 다뤘던 "면적과 방 개수로 집값 예측하기"가 실제 데이터로 확장된 셈입니다.

```
import pandas as pd

df = pd.read_csv('train.csv')
features = ['GrLivArea', 'OverallQual', 'TotalBsmtSF',
            'GarageCars', 'YearBuilt']
X = df[features].to_numpy(dtype=float)
y = df['SalePrice'].to_numpy(dtype=float)
print(X.shape, y.shape)  # (1460, 5) (1460,)
```

앞서 구현한 클래스를 활용하여 먼저 닫힌 해로 학습해봅시다.

```
model = LinearRegression().fit_closed_form(X, y)
print(model.theta)
```

문제없이 파라미터가 구해집니다. 그런데 경사 하강법은 어떨까요? 같은 데이터로 `fit_gd`를 호출해 봅시다.

```
model = LinearRegression().fit_gd(X, y, eta=0.0000001)
print(model.theta)  # [nan nan nan nan nan nan]
```

[3.7] https://www.kaggle.com/c/house-prices-advanced-regression-techniques

웬만큼 작은 학습률로는 파라미터가 발산하여 nan이 됩니다. 수렴하는 학습률이 존재하기는 하지만 너무 작아서, 그 학습률로는 사실상 학습이 진행되지 않습니다. 앞선 토이 예제에서는 잘 동작하던 코드가 실제 데이터에서는 실패합니다. 무엇이 문제일까요?

원인은 특성들의 스케일 차이에 있습니다. GrLivArea와 TotalBsmtSF는 수백에서 수천의 값을 갖는 반면, OverallQual은 1부터 10, GarageCars는 0부터 4 사이의 값입니다. 이렇게 스케일이 다른 특성이 섞여 있으면 비용 함수의 등고선이 극단적으로 길쭉한 타원이 되어, 경사 하강법이 안정적으로 수렴할 수 있는 학습률의 범위가 매우 좁아집니다. 게다가 목표값인 가격은 수십만 단위이므로 그래디언트 값 자체도 매우 커집니다.

표준화(standardization). 이를 해결하기 위해 학습 전에 각 특성의 스케일을 맞춰줄 수 있습니다. 각 특성에서 평균 μ를 빼고 표준편차 σ로 나누는 표준화(standardization)가 대표적인 방법입니다.

$$x' = \frac{x - \mu}{\sigma} \tag{3.132}$$

이를 사용하면 각각의 특성은 평균이 0이고 분산이 1을 갖는 값들이 됩니다.

```
X_mean, X_std = X.mean(axis=0), X.std(axis=0)
y_mean, y_std = y.mean(), y.std()
X_scaled = (X - X_mean) / X_std
y_scaled = (y - y_mean) / y_std

model_gd = LinearRegression().fit_gd(X_scaled, y_scaled,
                                     eta=0.0001, n_iters=5000)
model_cf = LinearRegression().fit_closed_form(X_scaled, y_scaled)
print(model_gd.theta)
print(model_cf.theta)
```

표준화된 데이터에서는 경사 하강법이 닫힌 해와 정확히 같은 파라미터로 수렴합니다. 예측할 때는 표준화된 입력으로 예측한 뒤 결과를 원래 스케일로 되돌리면 됩니다.

```
pred = model_gd.predict(X_scaled) * y_std + y_mean
rmse = np.sqrt(np.mean((pred - y)**2))
print(rmse)
```

sklearn으로 구현. 같은 문제를 sklearn으로 풀어봅시다. sklearn에서 선형 회귀 모델은 sklearn.linear_model의 LinearRegression 클래스로 제공됩니다. 이름이 같으므로 이 import 이후로 LinearRegression은 sklearn의 클래스를 가리키게 됩니다.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

```

```
model = LinearRegression()
model.fit(X, y)
print(model.intercept_)  # bias term
print(model.coef_)       # feature coefficients

pred = model.predict(X)
rmse = np.sqrt(mean_squared_error(y, pred))
print(rmse)
```

우리가 만든 클래스와 사용 방식이 거의 같습니다. `fit`으로 학습하고 `predict`로 예측하며, 학습된 파라미터는 편향 `intercept_`와 계수 `coef_`로 나뉘어 저장됩니다. 실행해보면 `fit_closed_form`으로 구한 파라미터와 정확히 일치하는 것을 확인할 수 있습니다. sklearn의 `LinearRegression`은 내부적으로 닫힌 해 방식으로 해를 구하기 때문에, 표준화 없이 원본 데이터를 넣어도 문제없이 동작합니다.

표준화를 함께 적용하고 싶다면 `StandardScaler`를 사용합니다. sklearn에서는 전처리와 모델을 하나로 묶는 파이프라인(pipeline) 기능을 제공합니다.

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

pipe = make_pipeline(StandardScaler(), LinearRegression())
pipe.fit(X, y)
pred = pipe.predict(X)
print(np.sqrt(mean_squared_error(y, pred)))
```

파이프라인은 `fit`이 호출되면 먼저 `StandardScaler`가 특성을 표준화하고, 변환된 데이터가 `LinearRegression`으로 전달되는 방식으로 동작합니다. `predict` 시에도 같은 변환이 자동으로 적용되므로, 우리가 직접 했던 것처럼 평균과 표준편차를 따로 저장해두고 되돌리는 수고를 할 필요가 없습니다. 예측 성능은 표준화 여부와 관계없이 동일합니다. 선형 회귀의 닫힌 해는 특성의 스케일에 영향을 받지 않기 때문입니다. 표준화가 필수였던 것은 경사 하강법이었다는 점을 다시 떠올려보시길 바랍니다.

예측 결과 분석. 학습이 잘 되었는지 예측 결과를 여러 각도에서 분석해봅시다. 먼저 모든 학습 샘플에 대해 예측 가격과 실제 가격을 산점도로 비교해보겠습니다.

```
import matplotlib.pyplot as plt

pred = model.predict(X)

fig, ax = plt.subplots(figsize=(5, 5))
ax.scatter(y/1000, pred/1000, s=8, alpha=0.4)
```

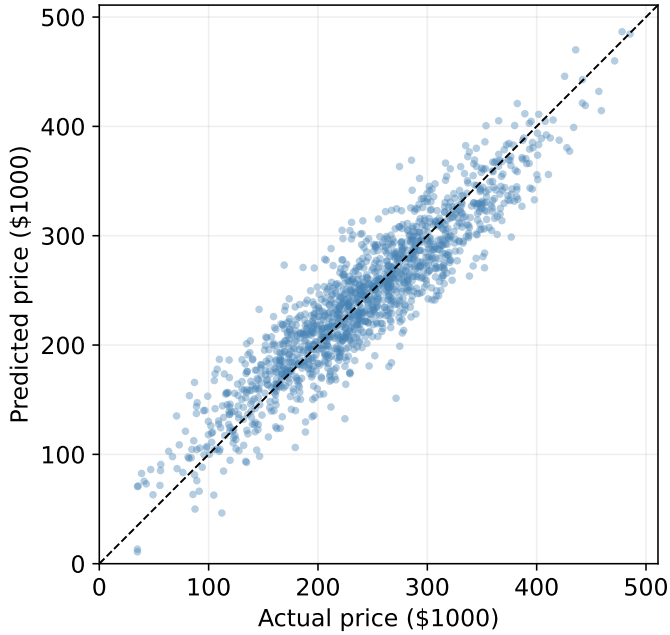

Figure 3.7: 실제 가격과 예측 가격의 비교. 점선은 예측이 실제와 정확히 일치하는 $y = \hat{y}$ 직선입니다.

```
lims = [0, max(y.max(), pred.max())/1000 * 1.05]
ax.plot(lims, lims, 'k--')  # perfect prediction line
ax.set_xlabel('Actual price ($1000)')
ax.set_ylabel('Predicted price ($1000)')
ax.set_aspect('equal')
plt.show()
```

그림 3.7에서 점선은 예측이 완벽하게 맞는 경우를 나타냅니다. 항상 그래프를 볼땐 각 축이 무엇을 나타내는지 잘 살펴봅시다. 가로축이 실제값 y, 세로축이 예측된 값 $\hat{y}$입니다. 가운데 점선은 실제값과 예측값이 같은 경우를 그려놓은 것입니다. 그러므로 점들이 점선 주변에 모여 있을수록 예측이 정확하다고 볼 수 있습니다. 전반적으로 점들이 점선을 따라 분포하고 있어 모델이 가격의 큰 흐름을 잡아내고 있음을 알 수 있습니다. 동시에 어느정도 그 오차의 폭도 함께 시각적으로 확인할 수 있습니다.

RMSE의 해석. 앞서 계산한 RMSE를 해석해봅시다. RMSE는 대략 "평균적으로 예측이 실제 가격에서 얼마나 벗어나는가"를 나타냅니다. 이 데이터셋의 평균 주택 가격과 비교해보면 오차의 상대적인 크기를 가늠할 수 있습니다.

```
print(rmse)              # rmse
print(rmse / y.mean())   # relative error
```

오차를 평균으로 나눈 값을 통해서 평균 가격 대비 오차가 어느 정도 비율인지 확인해보시길 바랍니다. 다섯 개의 특성만 사용한 단순한 선형 모델임을 감안하면 나쁘지 않은 수준이지만, 실제 부동산 감정에 쓰기에는 오차가 큽니다. 이 오차를 줄이기 위해 더 많은 특성을 사용하거나 더 복잡한 모델을 사용하는 등의 방법을 고려해볼 수 있습니다.

새로운 데이터 예측. 마지막으로 학습에 사용하지 않은 새로운 데이터를 예측해봅시다. Kaggle에서 내려받은 파일 중 `test.csv`에는 정답 가격이 없는 주택 1459채의 정보가 들어 있습니다. 이 집들의 가격을 우리 모델로 예측해보겠습니다.

```
df_test = pd.read_csv('test.csv')
X_test = df_test[features].to_numpy(dtype=float)
print(np.isnan(X_test).sum(axis=0))
```

출력을 확인해보면 일부 특성에 결측치(missing value)가 있습니다. 실제 데이터에서는 이렇게 값이 비어 있는 경우가 흔합니다. 여기서는 간단하게 학습 데이터에서 계산한 각 특성의 평균으로 결측치를 채우겠습니다.

```
# fill missing values with training means
for j in range(X_test.shape[1]):
    mask = np.isnan(X_test[:, j])
    X_test[mask, j] = X_mean[j]
```

이제 학습된 모델로 예측합니다. 경사 하강법 모델을 쓴다면 학습 때와 동일한 표준화를 적용해야 한다는 점에 주의합시다.

```
X_test_scaled = (X_test - X_mean) / X_std
pred_test = model_gd.predict(X_test_scaled) * y_std + y_mean
print(pred_test[:5])
```

예측 결과를 Kaggle이 요구하는 제출 형식으로 저장하면 실제로 채점을 받아볼 수도 있습니다.

```
submission = pd.DataFrame({'Id': df_test['Id'],
                           'SalePrice': pred_test})
submission.to_csv('submission.csv', index=False)
```

이 파일을 Kaggle에 제출하면 숨겨진 정답과 비교한 점수를 확인할 수 있습니다. 데이터로부터 입력과 출력 사이의 관계를 학습하고, 그 관계를 한 번도 본 적 없는 입력에 적용한다는 머신러닝의 기본 흐름이 하나의 완결된 사이클로 마무리된 것입니다. 직접 제출해보고, 이 장에서 만든 단순한 선형 모델이 어느 정도의 성적을 받는지 확인해보시길 바랍니다.

3.5.3 Remarks

이 장은 선형 회귀라는 하나의 모델을 배우는 자리였지만, 실은 그보다 큰 것을 배우는 자리였습니다. 장을 열며 말씀드렸듯, 이 장에 담긴 흐름은 이 책 전체를 관통합니다. 문제를 정의하고, 모델을

가정하고, 비용 함수를 세우고, 그 비용을 최소로 만드는 파라미터를 찾는 것. 이 네 걸음을 되짚어 보면 선형 회귀에서 우리가 한 일이 전부 여기에 들어맞습니다. 입력과 출력의 쌍으로 문제를 정의했고, 둘 사이를 선형 조합으로 잇는 모델을 가정했으며, 오차의 제곱합인 SSE로 비용을 세웠고, 그 비용을 최소로 만드는 파라미터 $\hat{\theta}$를 구하고자 했습니다.

같은 비용을 최소화하는데도 우리는 두 갈래 길을 걸었습니다. 하나는 정규방정식으로 답을 단번에 얻는 닫힌 해였고, 다른 하나는 조금씩 답에 다가가는 경사 하강법이었습니다. 답이 한 줄 수식으로 떨어지는데 왜 굳이 반복법을 배우는가 하는 물음도 던졌었지요. 그 답의 절반은 실습에서 드러났습니다. 특성의 스케일이 제각각인 실제 데이터에서 경사 하강법이 발산하고, 표준화를 거치고 나서야 닫힌 해와 같은 자리에 도달하는 과정을 우리는 직접 목격했습니다. 나머지 절반은 앞으로 드러납니다. 닫힌 해는 선형 회귀라는 특별한 경우에만 허락된 사치이고, 다음 장에서 만날 분류 모델부터는 비용을 최소로 만드는 파라미터를 한 줄 수식으로 내어주는 닫힌 해가 더이상 존재하지 않습니다. 그때 우리를 기다리고 있는 것이 바로 이 장에서 미리 손에 익혀 둔 경사 하강법입니다.

그러니 다음 장으로 넘어가며 바뀌는 것은 많지 않습니다. 모델의 모양이 달라지고, 비용 함수의 생김새가 달라질 뿐, 문제를 바라보는 네 걸음의 틀과 그 비용을 최소화하는 도구는 그대로입니다. 앞 장에서 준비해 둔 연장을 손에 쥔 채로, 이제 회귀를 떠나 분류의 세계로 들어가 봅시다.