Samuel J. Cavazos

Differential Geometry & Machine Learning

With Python for Parseltongues

November 29, 2024

Springer Nature

To Adalyn & Luka

Preface

Differential Geometry & Machine Learning explores the fascinating intersection of two powerful fields: differential geometry and machine learning. While machine learning has revolutionized industries ranging from healthcare to finance, differential geometry has long been a cornerstone of advanced mathematics, providing the tools to understand the curvature and structure of spaces. The synergy between these domains offers profound insights into the mathematical foundations of machine learning and equips practitioners with powerful techniques to build more robust and explainable models.

The motivation for this book stems from the desire to bridge the gap between theory and practice. As machine learning algorithms grow increasingly complex, understanding their underlying mechanics becomes not just an academic exercise but a necessity for developing effective, interpretable, and scalable solutions. Differential geometry provides a rigorous framework to address questions about curvature, optimization, and structure within the high-dimensional spaces where these models operate.

This book begins with the basics of machine learning, using linear regression as a gateway to understanding the fundamental principles of modeling and optimization. Through accessible explanations and hands-on examples, we build a foundation that extends naturally to more complex architectures, including neural networks. From there, we delve into the tools of differential geometry, showing how concepts such as gradients, manifolds, and geodesics inform and enhance machine learning algorithms.

Key features of this book include:

- **Practical Examples**: Python-based implementations and visualizations to solidify theoretical concepts.
- **Mathematical Rigor**: Detailed derivations and explanations that connect machine learning practices to their geometric and mathematical underpinnings.
- **Interdisciplinary Approach**: Insights from both machine learning and differential geometry, fostering a holistic understanding of modern AI techniques.

This book is intended for a diverse audience:

- **Mathematicians** intrigued by the applications of differential geometry in contemporary AI.
- **Machine Learning Engineers** seeking a deeper understanding of the mathematical principles behind their tools.
- **Students and Educators** looking for an accessible yet rigorous resource to explore the intersection of these fields.

As we journey through this book, we will not only develop a deeper appreciation for the beauty of differential geometry but also see how it empowers us to design better, more interpretable machine learning models. Whether you are a practitioner, researcher, or student, this book invites you to explore a rich and rewarding mathematical landscape that underpins some of the most transformative technologies of our time.

The Rio Grande Valley
2025

Samuel J. Cavazos
DHR Health

Acknowledgements

This project would not have been possible without the inspiration and support of my colleagues at DHR Health, whose insights and discussions have enriched the ideas presented here. I am especially grateful to the readers and learners who engage with this material—your curiosity and passion continue to drive this work forward.

Contents

Acronyms

List of abbreviations and symbols used in the book.

$\mathbb{C}$	The field of complex numbers
$\mathbb{Q}$	The field of rational numbers
$\mathbb{R}$	The field of real numbers
$\mathbb{Z}$	The ring of integers
$\mathbb{Z}/n\mathbb{Z}$	The ring of integers modulo n
ML	Acronym for Machine-Learning

Part I
Regression Models

Chapter 1
Linear Regression

Abstract This chapter introduces the principles of linear regression as a foundation for understanding the connection between differential geometry and machine learning. A simple linear model $M(x) = W \cdot x + b$ is constructed, and a loss function is used to quantify prediction errors. The chapter details the derivation of gradients for the loss with respect to the model parameters W and b, providing insights into how these gradients guide the optimization process.

1.1 Linear Models

Linear regressions are a fundamental tool in statistics and machine learning for modeling the relationship between a dependent variable y and one or more independent variables x. The simplest form of linear regression is a univariate linear model, which assumes a linear relationship between y and x of the form $y = W \cdot x + b$, where $W, b \in \mathbb{R}$ are real numbers. The model parameters W and b are learned from a dataset of input-output pairs $\{(x_i, y_i)\}_{i=1}^{N}$ by minimizing a loss function that quantifies the prediction errors of the model.

A more general form of the linear model is:

$$M(x) = W \cdot x + b,$$

where:

- W is a **weight matrix** of shape $m \times d$,
- x is a d-dimensional input vector,
- b is an m-dimensional **bias vector**.

Here, the model $M(x)$ outputs an m-dimensional vector prediction. This flexibility allows linear regression to handle scenarios where the model predicts multiple outputs simultaneously, making it applicable to a wide range of machine learning tasks. We begin by studying the univariate case.

1.2 Univariate Linear Models

Let's construct some data to work with that follows a somewhat linear trend and build a machine-learning model from scratch. We'll take the function $f(x) = x^2 + 2 \cdot x + 1$ over a random sample of points in $[0, 10]$ and add some uniform noise.

```python
def f(x):
    return x**2 + 2*x + 1

# Plot using matplotlib
import matplotlib.pyplot as plt
import numpy as np
import random

# Define the true function
def f(x):
    return x**2 + 2*x + 1

# Generate data
np.random.seed(42)
x = np.linspace(0, 10, 200)
y_true = f(x)
y_data = y_true + np.random.uniform(-10, 10, size=x.shape)

# Plot from 0 to 10
plt.xlim(0, 10)
plt.plot(x, y_data, 'o')
plt.plot(x, y_true)
plt.show()
```

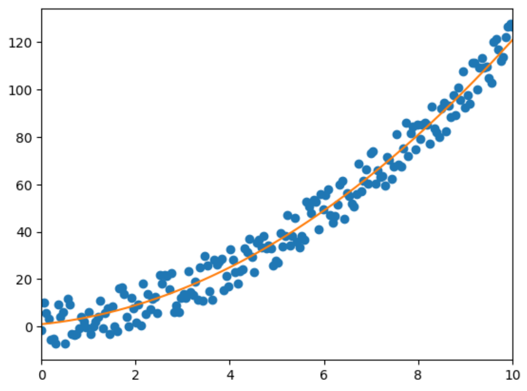

Fig. 1.1 Data generated from the function $f(x) = x^2 + 2 \cdot x + 1$ with added noise.

In Figure 1.1, the *best fit* for this data is the function we used to construct it. Of course, we usually don't know the equation for the best fit beforehand, but our goal is to create a model to approximate this line as closely as possible.

Let us start by constructing a simple machine-learning model for linear regression with no hidden layers, which essentially means there are no intermediate computations between the input and the output in our model.

Our goal is to build a machine-learning model $M : [0, 10] \to \mathbb{R}$ of the form

$$M(x) = W \cdot x + b,$$

where $W \in \mathbb{R}$ and $b \in \mathbb{R}$. Here, W is called the **weight** and b is called the **bias**.

Here, we define our linear model:

```
# Linear model
def linear_model(x, w, b):
    return w * x + b
```

In machine-learning, a model is initialized with random weights and biases, which are then corrected during training by minimizing a **loss function**. Let's start by choosing some random W and b.

```
# Initialize parameters
w = random.uniform(-1, 1)   # Random initial weight
b = random.uniform(-1, 1)   # Random initial bias


print(f'Initial weight: {w}')
print(f'Initial bias: {b}')
```

```
Initial weight: 0.918163880835746
Initial bias: 0.6882870252000235
```

Given that the weight and bias was chosen at random, we don't expect it to perform very well on our data, and indeed that is the case, as shown in the image below.

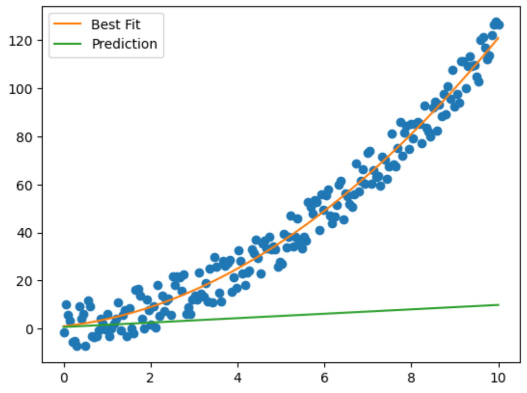

Fig. 1.2 Initial model prediction with random weight and bias.

Let's work on improving the model. Improving our model will involve tweaking W and b to better fit the model using a process called **gradient descent**.

1.2.1 Gradient Descent for Univariate Linear Models

We first define a **loss function** to measure how our model is performing. This function will quantify the difference between the model's predictions and the actual data. A common loss function for linear regression is the *Mean Squared Error* (MSE), which is defined as:

$$\mathcal{L} = MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_{\text{i,pred}} - y_{\text{i,true}}\right)^2 .$$

```python

# Mean Squared Error Loss
def mse_loss(y_pred, y_true):
    return np.mean((y_pred - y_true)**2)

print(f'50th sample target: {y_data[50]}')
print(f'50th prediction: {y_pred[50]}')
print(f'Loss at 50th sample: {mse_loss(y_pred[50], y_data[50])}')

print('Total Loss over all samples:', mse_loss(np.array(y_pred),
    np.array(y_data)))
```

```
50th sample target: 21.729790078081002
50th prediction: 2.995231449410441
Loss at 50th sample: 350.9836870110946
Total Loss over all samples: 2660.1490298370586
```

Our goal is to minimize this loss function.

One thing to note about this loss function is that it is a differentiable function. Recal from vector calculus that the **gradient** of a differentiable funtion f is a vector field ∇f whose value at point p is a vector that points towards the direction of steepest ascent.

Understanding the gradients of the loss function with respect to the model parameters—specifically, the weight W and bias b—is crucial in machine learning, particularly when employing optimization techniques like gradient descent. Our goal is to minimize the loss function.

The gradients $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$ indicate how sensitive the loss function $\mathcal{L}$ is to changes in the parameters W and b. In essence, they provide the direction and rate at which $\mathcal{L}$ increases or decreases as we adjust these parameters.

By computing these gradients, we can iteratively update W and b to minimize the loss function, thereby improving the model's performance. This process is the foundation of the gradient descent optimization algorithm.

1.2.1.1 Gradients

1. *Gradient with Respect to Weight W:*
   The partial derivative $\frac{\partial \mathcal{L}}{\partial W}$ measures how the loss changes concerning the weight W. A positive derivative suggests that increasing W will increase the loss, while a negative derivative indicates that increasing W will decrease the loss. By moving W in the direction opposite to the gradient, we can reduce the loss.
   If x_i is the i-th data point, let $y_{i,\text{pred}} = W \cdot x_i + b$ be the predicted value for the i-th data point while $y_{i,\text{true}}$ denotes the true value. Mathematically, this gradient is computed as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial}{\partial W}\left(\frac{1}{N}\sum_{i=1}^{N}(y_{i,\text{pred}} - y_{i,\text{true}})^2\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial W}(y_{i,\text{pred}} - y_{i,\text{true}})^2 \qquad \text{(Additive property of derivatives)} \\
&= \frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial W}((W \cdot x_i + b) - y_{i,\text{true}})^2 \\
&= \frac{1}{N}\sum_{i=1}^{N} 2 \cdot ((W \cdot x_i + b) - y_{i,\text{true}}) \cdot (x_i) \qquad \text{(Chain rule)} \\
&= \frac{2}{N}\sum_{i=1}^{N}(y_{i,\text{pred}} - y_{i,\text{true}}) \cdot x_i.
\end{aligned}$$

   Thus, we find that

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{2}{N}\sum_{i=1}^{N}(y_{i,\text{pred}} - y_{i,\text{true}}) \cdot x_i. \tag{1.1}$$

2. *Gradient with Respect to Bias b:*
   Similarly, the partial derivative $\frac{\partial \mathcal{L}}{\partial b}$ measures how the loss changes concerning the bias b. Adjusting b in the direction opposite to this gradient will also help in minimizing the loss.
   This gradient is computed as:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{N}\sum_{i=1}^{N}(y_{i,\text{pred}} - y_{i,\text{true}}). \tag{1.2}$$

   Proof of this equation is left as an exercise to the reader.

With that, we can compute the gradients in Python:

```python
# Compute gradients
def compute_gradients(x, y_true, w, b):
    y_pred = linear_model(x, w, b)
    error = y_pred - y_true
    dw = 2 * np.mean(error * x)
    db = 2 * np.mean(error)
    return dw, db
```

Now that we have a way of computing the partial derivatives of $\mathcal{L}$ with respect to W and b, we can visualize the *gradient field*. For a given $p = (W, b) \in \mathbb{R}^2$, the gradient $\nabla\mathcal{L}$ at p is a vector that points towards the rate of fastest increase. In the following code, we compute these vectors on a grid. We also include a 2D contour plot of the loss function $\mathcal{L}$. Our initial weight W and bias b are marked on the plot by a red dot.

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

# Gradient Descent parameters
alpha = 0.001  # Learning rate
epochs = 1000  # Number of iterations

# Create a grid of w and b values for contour and quiver plotting
w_vals = np.linspace(-10, 20, 100)
b_vals = np.linspace(-20, 10, 100)
W, B = np.meshgrid(w_vals, b_vals)

# Compute the loss for each combination of w and b in the grid
Z = np.array([mse_loss(linear_model(x, w, b), y_data) for w, b in
     zip(np.ravel(W), np.ravel(B))])
Z = Z.reshape(W.shape)

# Compute the gradient field
dW = np.zeros(W.shape)
dB = np.zeros(B.shape)
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        dw, db = compute_gradients(x, y_data, W[i, j], B[i, j])
        dW[i, j] = dw
        dB[i, j] = db

# Plot the cost function contour, gradient field, and gradient
    descent path
plt.figure(figsize=(12, 8))

# Contour plot of the loss function
cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
    viridis')
plt.colorbar(cp)
plt.xlabel('Weight (w)')
plt.ylabel('Bias (b)')
plt.title('Cost Function Contour, Gradient Field, and Gradient
    Descent Path')

# Quiver plot of the gradient field
plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=1,
    color='blue', alpha=0.5)
# plot initial weight, bias
plt.plot(w, b, 'ro', label='Initial (weight, bias)')
plt.legend()
plt.grid(True)
plt.show()
```

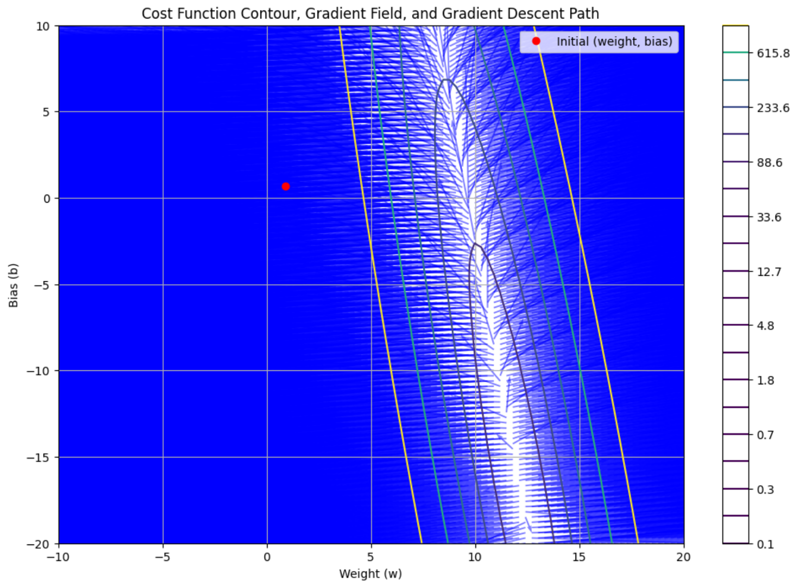

Fig. 1.3 Contour plot of the loss function, gradient field, and gradient descent path.

Since our goal is to minimize the loss function and these vectors are pointing towards the steepest ascent of the loss function with respect to W and b, we minimize by moving in the opposite direction of the gradients. This process is fundamental to optimization algorithms like gradient descent and is refered to as **backward propogation** within machine-learning.

1.2.1.2 Gradient Descent & Backward Propagation

Gradient descent is an optimization algorithm that iteratively updates the model parameters in the direction opposite to the gradients of the loss function. This process continues until the loss is minimized. **Backward propagation** is the process of computing these gradients and updating the model parameters.

The parameter updates are performed iteratively using the following rules:

1. Weight update:
$$W \leftarrow W - \alpha \frac{\partial \mathcal{L}}{\partial W}$$
Here, α is the learning rate, a hyperparameter that controls the step size of each update. The term $\frac{\partial \mathcal{L}}{\partial W}$ represents the gradient of the loss function with respect to the weight. By subtracting this scaled gradient from the current weight, we move W in the direction that decreases the loss.
2. Bias update:
$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$
Similarly, $\frac{\partial \mathcal{L}}{\partial b}$ is the gradient of the loss function with respect to the bias. Updating b in this manner adjusts the model's predictions to better fit the data.

The **learning rate** determines how large a step we take in the direction of the negative gradient. A small α leads to slow convergence, while a large α might cause overshooting the minimum, leading to divergence. Choosing an appropriate learning rate is crucial for effective training.

The gradients $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$ indicate the direction in which the loss function increases most rapidly. By moving in the opposite direction (hence the subtraction), we aim to find the parameters that minimize the loss.

We repeat this process over and over again. Each time we do it is referred to as an **epoch**.

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

# Assuming x and y_data are your input features and target values
    respectively

# Define the linear model
def linear_model(x, w, b):
    return w * x + b

# Define the loss function (Mean Squared Error)
def mse_loss(y_pred, y_true):
    return np.mean((y_pred - y_true) ** 2)

# Compute gradients
def compute_gradients(x, y_true, w, b):
    y_pred = linear_model(x, w, b)
    error = y_pred - y_true
    dw = 2 * np.mean(error * x)
    db = 2 * np.mean(error)
    return dw, db

# Gradient Descent parameters
```

```python
alpha = 0.001  # Learning rate
epochs = 2000  # Number of iterations

# Store parameters for plotting
w_history = [w]
b_history = [b]
loss_history = [mse_loss(linear_model(x, w, b), y_data)]

# Gradient Descent loop
for epoch in range(epochs):
    dw, db = compute_gradients(x, y_data, w, b)
    w = w - alpha * dw # Update the weight
    b = b - alpha * db # Update the bias

    w_history.append(w) # Add to weight tracker
    b_history.append(b) # Add to bias tracker
    loss_history.append(mse_loss(linear_model(x, w, b), y_data))
    # Add overall loss to loss tracker

# Convert history lists to numpy arrays for easier slicing
w_history = np.array(w_history)
b_history = np.array(b_history)

# Create a grid of w and b values for contour and quiver plotting
w_vals = np.linspace(-10, 20, 100)
b_vals = np.linspace(-20, 10, 100)
W, B = np.meshgrid(w_vals, b_vals)

# Compute the loss for each combination of w and b in the grid
Z = np.array([mse_loss(linear_model(x, w, b), y_data) for w, b in
     zip(np.ravel(W), np.ravel(B))])
Z = Z.reshape(W.shape)

# Compute the gradient field
dW = np.zeros(W.shape)
dB = np.zeros(B.shape)
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        dw, db = compute_gradients(x, y_data, W[i, j], B[i, j])
        dW[i, j] = dw
        dB[i, j] = db

# Print initial (weight, bias)
print(f'Initial (weight, bias): ({w_history[0]}, {b_history[0]})'
    )
# Print final (weight, bias)
print(f'Final (weight, bias): ({w_history[-1]}, {b_history[-1]})'
    )

# Plot the cost function contour, gradient field, and gradient
    descent path
plt.figure(figsize=(12, 8))

# Contour plot of the loss function
```

```python
cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
    viridis')
plt.colorbar(cp)
plt.xlabel('Weight (w)')
plt.ylabel('Bias (b)')
plt.title('Cost Function Contour, Gradient Field, and Gradient
    Descent Path')

# Quiver plot of the gradient field
plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=1,
    color='blue', alpha=0.5)

# Plot the gradient descent path
plt.plot(w_history, b_history, 'ro-', markersize=3, linewidth=1,
    label='Gradient Descent Path')
# Plot the initial weight, bias
plt.plot(w_history[0], b_history[0], 'ro', label='Initial (weight
    , bias)')

# Add arrows to indicate direction of descent
for i in range(1, len(w_history)):
    plt.arrow(w_history[i-1], b_history[i-1],
                w_history[i] - w_history[i-1],
                b_history[i] - b_history[i-1],
                head_width=0.05, head_length=0.1, fc='red', ec='
    red')

plt.legend()
plt.grid(True)
plt.show()
```

```
  Initial (weight, bias): (0.918163880835746, 0.6882870252000235)
Final (weight, bias): (11.090173700216313, -9.650829435811533)
```

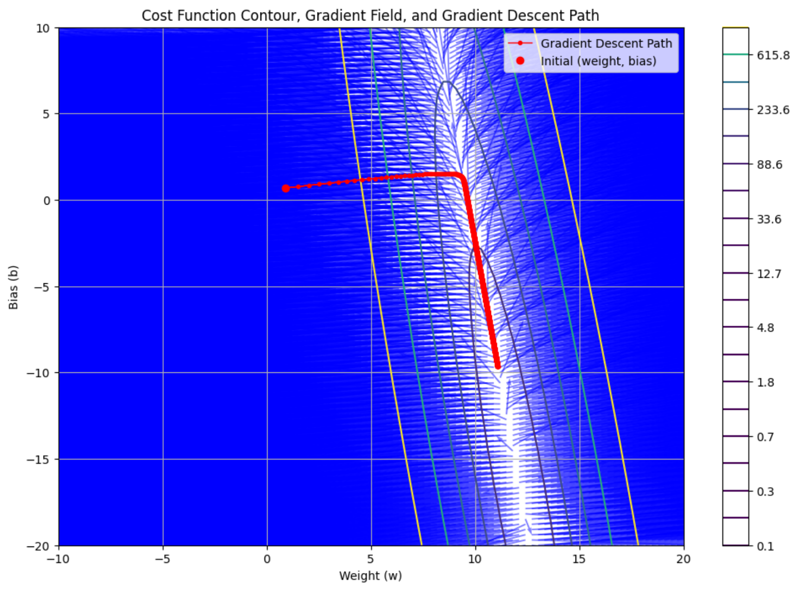

Fig. 1.4 Contour plot of the loss function, gradient field, and gradient descent path.

Since our weight W and bias b together form a point $(W, b) \in \mathbb{R}^2$, the loss function $\mathcal{L}$ forms a 3-dimensional surface. The visualization below shows the path taken during gradient descent on the surface of the loss function $\mathcal{L}$. The initial point (W, b) is in green. The path moves towards $\mathcal{L}$'s minimum.

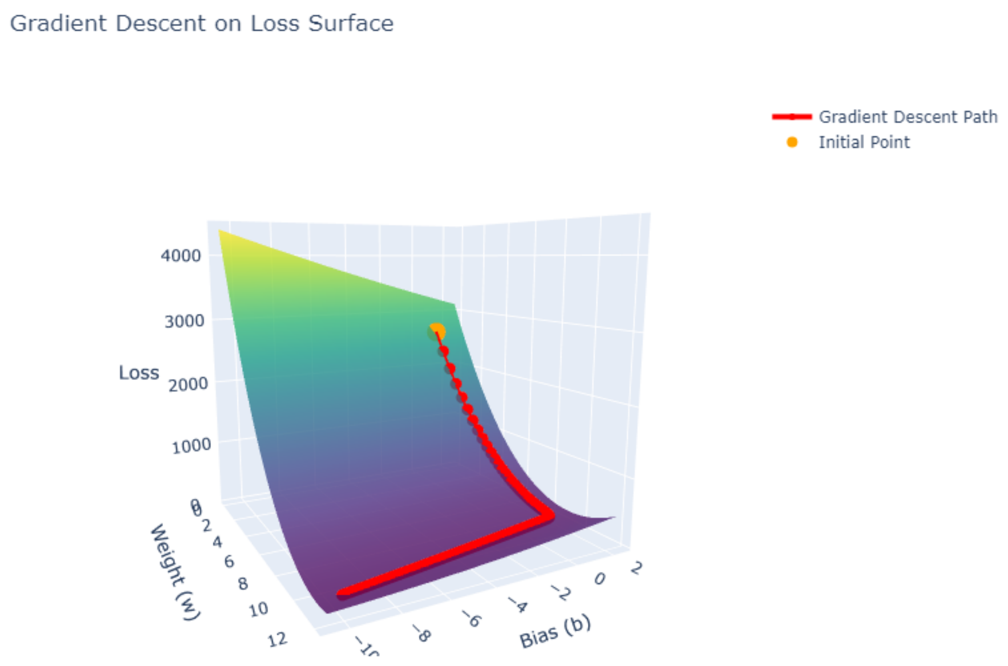

Fig. 1.5 Gradient descent path on the loss function surface.

Finally, we visualize our initial (green) and final (red) linear model on a graph, alongside the data and true line of best fit (orange).

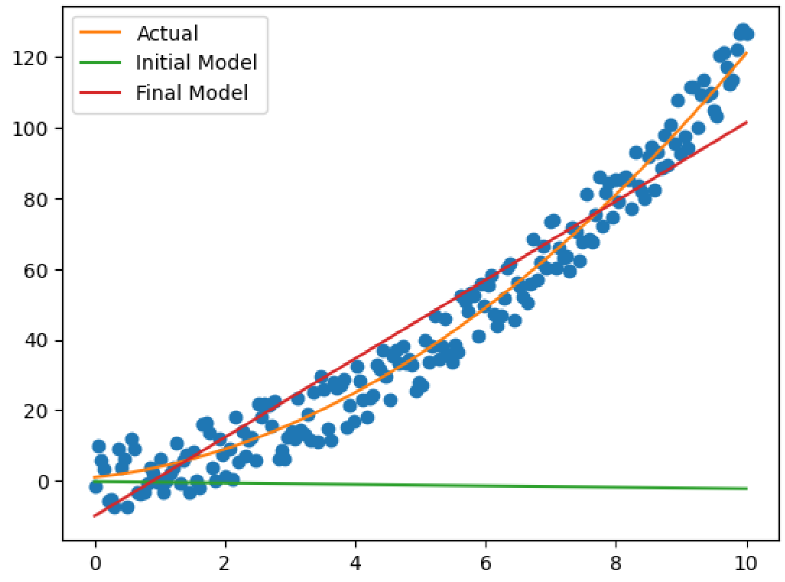

Fig. 1.6 Initial and final linear models compared to the true line of best fit.

Appendix A
Chapter Heading

All's well that ends well

Use the template *appendix.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style appendix of your book.

A.1 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations.

A.1.1 Subsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.

For multiline equations we recommend to use the `eqnarray` environment.

$$\begin{aligned}\mathbf{a}\times\mathbf{b}&=\mathbf{c}\\ \mathbf{a}\times\mathbf{b}&=\mathbf{c} \qquad (A.1)\end{aligned}$$

A.1.1.1 Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.1.

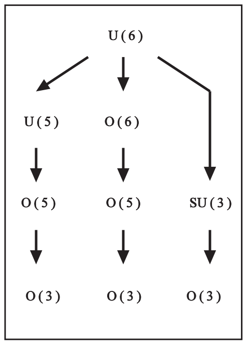

Fig. A.1 Please write your figure caption here

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

Table A.1 Please write your table caption here

Classes	Subclass	Length	Action Mechanism
Translation	mRNA^a	22 (19–25)	Translation repression, mRNA cleavage
Translation	mRNA cleavage	21	mRNA cleavage
Translation	mRNA	21–22	mRNA cleavage
Translation	mRNA	24–26	Histone and DNA Modification

^{a} Table foot note (with superscript)

Glossary

Use the template *glossary.tex* together with the Springer document class SVMono (monograph-type books) or SVMult (edited books) to style your glossary in the Springer layout.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

Solutions

Problems of Chapter 1

?? The solution is revealed here.

?? Problem Heading
(a) The solution of first part is revealed here.
(b) The solution of second part is revealed here.

Index