Part I

Supervised Learning

1
Linear Regression: An Introduction

This chapter introduces *least squares linear regression*, one of the simplest and and most popular model in data science. Several of you may have seen it in high school. In particular, we focus on understanding linear regression in the context of machine learning. Using linear regression as an example, we will introduce the terminologies and ideas (some of them were mentioned in the Preface) that are widely applicable to more complicated models in ML.

1.1 A Warm-up Example

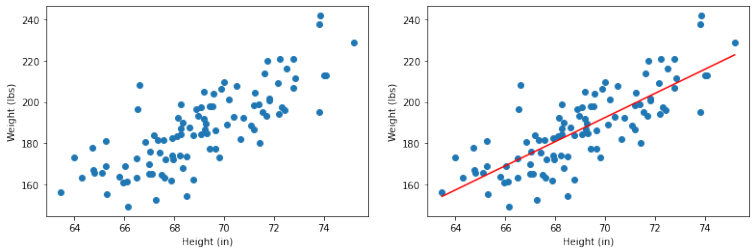

Figure 1.1: A dataset of heights and weights of some male adults. The figure on the right shows the least squares regression line that fits the data. Data from https://gist.github.com/nstokoe/7d4717e96c21b8ad04ec91f361b000cb

Suppose we have a dataset of heights and weights of some male individuals randomly chosen from the population. We wish to determine a relationship between heights and weights. The simplest relationship would be a linear relationship; namely:

$$w = a_0 + a_1 h \tag{1.1}$$

where w is the weight, h is the height, and a_0, a_1 are constant coefficients. We can think of this as a *predictor* that maps height h to a predicted weight $a_0 + a_1 h$, and we want this value to be similar to the actual weight w. Obviously, a linear relationship won't describe the data exactly but we hope it is a reasonable fit to data. [1] In a ML setting, this relationship between h and w is called a *model* — a linear model to be more specific.

[1] Similar linear models are used in many disciplines. For instance, the famous Philips model in economics suggests a linear relationship between inflation rate and unemployment rate, at least when inflation rate is high.

Based on the values of a_0 and a_1, there are infinitely many different choices of this linear model. Therefore, it is natural that we want to find the values of a_0, a_1 that yield the "best" model. In a ML setting, finding these optimal values of a_0, a_1 is known as *fitting* the model. One can posit different criteria for defining "goodness" of the model.

Here we use classic *least squares fit*, invented by Gauss. Given a dataset $\{(h_1, w_1), (h_2, w_2), \ldots, (h_n, w_n)\}$ of n pairs of heights and weights, the "goodness" of the model in (1.1) is

$$\frac{1}{n}\sum_{i=1}^{n}(w_i - a_0 - a_1 h_i)^2 \tag{1.2}$$

Notice that $w_i - a_0 - a_1 h_i$ is the difference between the actual weight w_i and the predicted weight $a_0 + a_1 h_i$. This difference is called the *residual* for the data point (h_i, w_i), and the full term in (1.2) is called the *average squared residuals*, or equivalently the *mean squared error* (MSE), of the dataset. The smaller the MSE, the closer the model's predictions are to actual weights, and the more accurate the model is. Therefore, the "best" model according to the least squares method would be the one defined by the values of a_0, a_1 that minimize (1.2). In a ML setting, a mathematical expression like (1.2) that captures the "goodness" of the model is called a *loss function*. In general, we find the "best" model by minimizing the loss function.

Example 1.1.1. *If the data points (h, w) are given as $\{(65, 130), (58, 120), (73, 160)\}$, the least squares fit will find the values of a_0, a_1 that minimize*

$$\frac{1}{3}((130 - a_0 - 65a_1)^2 + (120 - a_0 - 58a_1)^2 + (160 - a_0 - 73a_1)^2)$$

which are $a_0 = -\frac{510}{13}$, $a_1 = \frac{35}{13}$.

Problem 1.1.2. *Between the two lines in Figure 1.2, which is more preferred by the least squares regression method?*

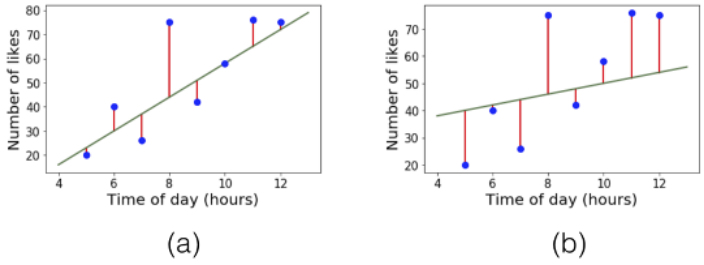

Figure 1.2: Two lines that describe the relationship of the same dataset.

Problem 1.1.3. *Using calculus, give an exact expression for a_0, a_1 that minimize (1.2). (Hint: (1.2) is quadratic in both a_0 and a_1. Fix the value of*

a_1 and minimize for a_0. Then minimize for a_1. Completing the square may be useful.) [2]

[2] A more general calculus based approach will be introduced in a later chapter.

1.1.1 Multivariate Linear Regression

One can generalize the above example to multi-variable settings. In general, we have k *predictor variables* and one *effect* variable. [3] The data points consist of $k+1$ coordinates, where the last coordinate is the value y of the effect variable and the first k coordinates contain values of the predictor variables $x_1, x_2, \ldots, x_k$. Then the relationship we are trying to fit has the form

[3] In the above example $k = 1$. The predictor variable was height and effect variable was weight.

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_k x_k \tag{1.3}$$

and the least squares fit method will find the values of $a_0, a_1, \cdots, a_k$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - a_0 - a_1 x_1^i - a_2 x_2^i - \cdots - a_k x_k^i)^2 \tag{1.4}$$

where $(x_1^i, x_2^i, \ldots x_k^i, y^i)$ is the i-th data point.

We can simplify the notation by rewriting everything above in a vectorized notation. If we set $\vec{\mathbf{x}} = (1, x_1, x_2, \cdots, x_k)$ [4] and $\vec{\mathbf{a}} = (a_0, a_1, \cdots, a_k)$, then the relationship we are trying to fit has the form

[4] The 1 in the first coordinate is a dummy variable to naturally include the constant term into the vectorized notation.

$$y = \vec{\mathbf{a}} \cdot \vec{\mathbf{x}} \tag{1.5}$$

and the least squares fit method will find $\vec{\mathbf{a}} \in \mathbb{R}^{k+1}$ that minimize

$$\frac{1}{n} \sum_{i=1}^{n} (y^i - \vec{\mathbf{a}} \cdot \vec{\mathbf{x}}^i)^2 \tag{1.6}$$

where $(\vec{\mathbf{x}}^i, y^i)$ is the i-th data point. We discuss how to find the best values of $a_0, a_1, \ldots, a_k$ later in Chapter 3; for now just assume that the solution can be found.

1.1.2 Testing a model (Held-out Data)

A crucial step in machine learning is to *test* the trained/fitted model on *newly seen data*, or *held-out data*, that was not used during training. If we were to test the model in the above example, we would hold out a portion of the data points (say 20%) — *i.e.*, not use them during training — and check the average squared residual of the model on the held-out data points.

We can think of the average squared residual of the held-out data as an *estimate* of the average squared residual of the fitted model on the entire population of male adults. [5] The reason is that if the training data points were a random sample of the adult male population,

[5] If later in life you ever write up the results of a regression study, be sure to report the *RMSE error*, which is the square root of the average square residual on held-out data. Also report the R^2 value, which is closely related.

then so is the set of held-out data points. This is quite analogous to opinion polls, where the opinions of a few thousand randomly sampled individuals can be a reasonable estimate for the average opinion across the US. The math for such sampling estimates is covered in Chapter 18.

1.1.3 *More about Linear Regression*

In the above example, we used the least squares method, which uses the average squared residual to assess the model. The least squares fit is very common but other notions of fit may also be used. For instance, instead of taking the sum of squares of residuals, one could consider the sum of absolute values, or expressions using logarithms, etc. We see some of these examples in Chapter 4.

It is important to note that the relationship learnt via regression — and machine learning in general — is (a) approximate and (b) only holds for the population that the data was drawn from. Therefore, we cannot use the relationship to predict the output of a data that is not from the same distribution. Additionally, if the distribution of the data is shifted, the relationship no longer holds. We will discuss more about this in depth in Chapter 2.

1.2 *Using Linear Regression for Sentiment Prediction*

1.2.1 *Introduction*

While you might have seen linear regression as early as in high school, you probably did not see this cool application. In *sentiment classification*, we are given a piece of text and have to label it with $+1$ if it expresses positive sentiment and -1 otherwise.

Example 1.2.1. *Consider the following dataset, collected by showing snippets of text to humans and asking them to label them as positive ($+1$) or negative (-1)*

The film's performances are thrilling.	$+1$
It's not a great monster movie.	-1
It is definitely worth seeing.	$+1$
Unflinchingly bleak and desperate.	-1

Table 1.1: Data from Stanford Sentiment Treebank (SST). https://nlp.stanford.edu/sentiment/treebank.html

How can we train a model to label text snippets with the correct sentiment value, given a dataset of training examples? Here is an idea to try to solve it using a linear regression model. We first enumerate all English words and assign a real-valued score to each word, where the score for the i-th word is denoted by w_i. These scores will

be the *parameters* of the model. The output of the model, given a training example, is defined as $\sum_{j \in S} w_j$ where S is the multiset of words in the text. [6] Then the least squares method needs to solve the following optimization problem for a dataset of (text, sentiment) pairs

$$\text{minimize} \sum_i \left(y^i - \sum_{j \in S^i} w_j \right)^2 \tag{1.7}$$

where S^i is the multiset of words in the i-th piece of text. Each of the values $\left(y^i - \sum_{j \in S^i} w_j \right)^2$ is called a *least squares error* or more generally the *loss* for that particular training example. The full summation is called a *training loss* of the dataset.

[6] Unlike in a set, an element can appear multiple times in a multiset. For example, if the word *good* appears twice in a text, then S contains two copies of *good*.

Example 1.2.2. *Assume we are training a sentiment prediction model on a dataset. Table 1.1 shows some of the model parameter values. Then the output of the model on the sentence "I like this movie" from the training data will be* $0.15 + 0.55 + 0.03 - 0.07 = 0.66$. *The output for "I dislike this movie" from the training data will be* $0.15 - 0.74 + 0.03 - 0.07 = -0.63$

i	word	w_i
1	I	0.15
2	like	0.55
3	dislike	−0.74
4	this	0.03
5	movie	−0.07
6	a	0

Table 1.2: Some of the parameter values of a sentiment prediction model.

We can also cast this in the standard formulation of linear regression as follows. The *bag of words (BoW)* representation of a piece of text is a vector in $\mathbb{R}^N$ where N is the number of dictionary words. The i-th coordinate is the number of times the i-th word appears in the piece of text. This represents the text as a very long vector, one coordinate per one English word in the dictionary. The vector usually contains a lot of zeros, since most words probably do not appear in this piece of text. If we denote the BoW vector as $\vec{\mathbf{x}}$, the output of the model is seen to be

$$\sum_{j \in S} w_j = \sum_i w_i x_i$$

which shows that the linear model we have proposed for sentiment prediction is just a subcase of linear regression (see (1.5)).

Example 1.2.3. *Consider the same model in Example 1.2.2. The BoW representation for the sentence "I like this movie" is* $(1, 1, 0, 1, 1, 0 \cdots)$. *The BoW representation for the sentence "I dislike this movie" is* $(1, 0, 1, 1, 1, 0 \cdots)$.

1.2.2 *Testing the Model*

Here we use the model from Example 1.2.2 to illustrate the training and testing process of a model. Assume that the following four sentences were a part of the training dataset.

I like this movie.	$+1$
I dislike this movie.	-1
I like this.	$+1$
I dislike this.	-1

Table 1.3: A portion of the training data for a sentiment prediction model.

Assuming that the model parameters are the same as reported in Table 1.2, we can calculate the training loss of the sentence "I like this movie" as $(+1 - 0.66)^2 \simeq 0.12$. Similarly, the squared residual for each of the four training sentences in Table 1.3 can be calculated as

I like this movie.	0.12
I dislike this movie.	0.14
I like this.	0.07
I dislike this.	0.19

Table 1.4: The squared residual for four training examples.

Now it is time to test the model. Assume that the sentence "I like a movie" is provided to the model as a test data. The *test loss* can be calculated in a way similar to the training loss as $(+1 - 0.63)^2 \simeq 0.14$. But to actually test if the model produces the correct sentiment label for this newly seen data, we now wish the model to output either $+1$ or -1, the only two labels that exist in the population. An easy fix is to change the output of the model at test time to be $\text{sign}(\sum_{j \in S} w_j)$. For this test data, the model will output $\text{sign}(0.63) = +1$.

On the Stanford Standard Treebank, this approach of training a least squares model yields a success rate of 78% [7]. By contrast, the state-of-the-art deep learning methods yield success rate exceeding 96%!

[7] To be more exact, this result is from a model called *ridge regression* model, which is linear regression model augmented by an ℓ_2 regularizer, which will be explained in Chapter 3

One thing to note is that while the training loss is calculated and explicitly used in the training process, the test loss is only a statistics that is generated after the training is over. It is a metric to assess if the model fitted on the training data also performs well for a more general data.

1.2.3 *Test Loss, Generalization, and Test accuracy*

As mentioned already, the goal of training a model is that it should make good predictions on new, previously-unseen data. Most models will exhibit a low training loss, but not all of them show a low test loss. This observation motivates the following definition:

$$\boxed{\text{Generalization Error} = |\text{training loss} - \text{test loss}|}$$

A trained model is said to *generalize well* if the generalization error is small. In our case, the loss is the average squared residual. Thus good generalization means that the average squared residual on test data points is similar to that on the training data.

Let us see what happens on our sentiment model when it is fitted and tested on the SST dataset.

Train MSE	0.0727
Test MSE	0.7523
Training accuracy	99.55%
Test accuracy	78.09%

Table 1.5: *Accuracy* refers to the classification accuracy when we make the model to output only ± 1 labels.

Example 1.2.4. *The **generalization error** above is the difference between MSE on test points and the MSE on training points, namely* $0.75 - 0.07 = 0.68$.

Let's try to understand the relationship between low test loss (the squared residual) and high test accuracy (for what fraction of test data points the sentiment was correct). Heuristically, the test loss (average squared residual) being 0.75 means that the the absolute value of the residual on a typical point is $\sqrt{0.75} \approx 0.87$. This means that for a data point with an actual positive sentiment (*i.e.*, label $+1$), the output of the model is roughly expected to lie in the interval $[1 - 0.87, 1 + 0.87]$, and similarly, for a data point with an actual negative sentiment (*i.e.*, label -1), the output of the model is roughly expected to lie in the interval $[-1 - 0.87, -1 + 0.87]$. Once we take the sign $\text{sign}(\sum_{j \in S} w_j)$ of the output of the model, the output is thus likely to be rounded off to the correct label. We also note that the training accuracy is almost 100%. This usually happens in settings where the number of parameters (*i.e.*, number of predictor variables) exceeds the number of training data points (or is close to it). The following problem explores this.

Problem 1.2.5. *An expert on TV claims to have a formula to predict outcome of presidential election. It uses* 31 *measurements of various economic and societal quantities (inflation, divorce rate, etc). The formula correctly predicts the winner of all elections* 1928-2020. *Should you believe the formula's prediction for the* 2024 *election? (Hint: Under fairly general conditions,* $T + 1$ *completely nonsense variables — i.e., having nothing to do with presidential politics — can be used to perfectly fit (via linear regression) the outcomes for* T *past presidential elections.* [8]*)*

[8] If a model does not generalize well, then it is said to *overfit* the training data.

1.2.4 Interpreting the Model

In many settings (*e.g.*, medicine), an important purpose of regression modeling is to understand the data or the phenomenon a bit

better. In this case, the phenomenon is "sentiment" and we are naturally curious about what positive or negative sentiment amounts to. Specifically, what caused the model's output to be $+1$ or -1 given a specific sentence?

Figure 1.3 shows a histogram of the values of w_i, the parameters of a sentiment prediction model that was trained on the Stanford Sentiment Treebank. Positive values of w_i imply that the words carry a positive sentiment, while negative values of w_i imply that the words carry a negative sentiment. Also, the greater the absolute value of w_i is, the stronger the sentiment. Notice that most words have a value of w_i close to zero, meaning the model views most words as neutral. The model "pays attention" to only a tiny set of words.

Words with high positive w_i values (*i.e.*, positive words) include *enjoyable*, *fun*, and *remarkable*. Words with high negative values (*i.e.*, negative words) include *suffers*, *dull*, and *worst*. Words with w_i values close to 0 (*i.e.*, neutral words) include *duty* and *desire*.

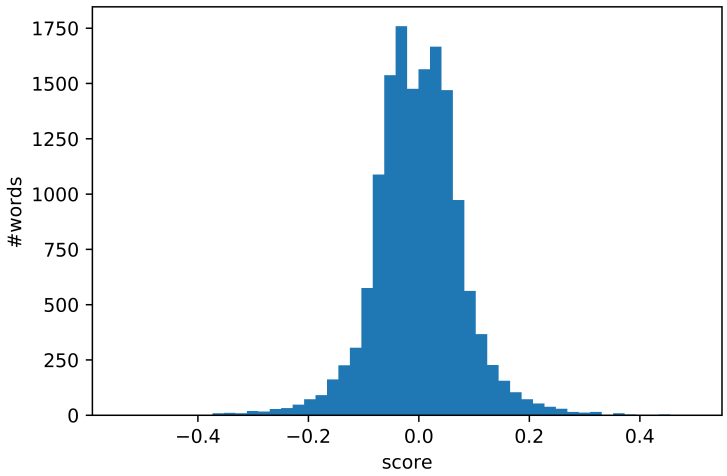

Figure 1.3: A histogram of the learned parameters w_i of a sentiment prediction model trained on the Stanford Sentiment Treebank.

1.3 *Importance of Featurization*

In the sentiment model, we chose a particular method to represent a piece of text with a vector. The coordinates of this vector are often referred to as *features* and this process of converting data into vectors is called *featurization*. One can conceive of other choices for featurizing text. For example, *bigram* featurization consists of the following: the coordinates of the vector correspond to *pairs* of words and the coordinate contains the number of times this pair of words appeared consecutively in the piece of text. In contrast, the choice of featurization from the earlier example matches each coordinate with a single word, and is called a *unigram* featuraization.

Bigram features allow the model to access information about phrases that were present in the text. For instance, in isolation "pretty" is a positive word and "bad" is a negative word. If they both occur in text one would imagine that they cancel each other out as far as overall sentiment is concerned. But the phrase "pretty bad" is more negative than "bad." Thus bigram features can improve the model's ability to capture sentiment.

The required number of dimension for bigram representations can get rather large. If the number of words is N, then the number of coordinates is $N(N-1)$. Realize that the number of model parameters in linear regression is the same as the number of coordinates. Thus if N is $30,000$ then the number of coordinates in bigram feature vector (and hence number of model parameters) is close to a billion, which a rather large number. In practice one might throw away information for all pairs except say the $10,000$ most common ones in the dataset. Usually models that incorporate bigram features do better than unigram-only models.

If one is trying to do studies of medical treatment with regression, there can be many potential featurizations of patient data. Doctors' annotations, test results, x-ray scans, etc. all have to be converted somehow into real-valued features, and the experimenter uses their prior knowledge and intuitions while featurizing the data.

Example 1.3.1. *Patients' raw data might include height and weight. If we use linear regression, the effect variable can only depend upon a linear combination of height and weight. But it is known that several health outcomes are better modeled using* Body Mass Index*, defined as weight/height*^{2}*. Thus the experimenter may include a separate coordinate for BMI, even though it duplicates information already present in the other coordinates.*

Example 1.3.2. *In the above dataset, the weight in pounds may span a range of* $[90, 350]$*, whereas cholesterol ratio may span a range of* $[1, 10]$*. It is often a good idea to* normalize *the coordinate, which means to replace* x *with* $(x-\mu)/\sigma$ *where* μ *is the mean of the coordinate values in the dataset and* σ *is the standard deviation.*

Thus the same raw dataset can have multiple featurizations, with different number of coordinates. Problem 1.2.5 may make us wary of using featurizations with too many coordinates. We will learn a technique called *regularization* in Chapter 3, which helps mitigate the issue identified in Problem 1.2.5.

1.4 *Linear Regression in Programming*

In this section, we briefly discuss how to write the Python code to perform linear regression (*e.g.*, sentiment prediction). Python is often the language of choice for many machine learning applications due to its relative ease of use and the large variety of external packages available to automate the process. Here, we introduce a few of these packages:

- *numpy*: This package is ubiquitous throughout the machine learning community. It provides access to specialized array data structures which are implemented in highly optimized C code. Linear algebra computations and array restructuring operations are significantly faster with *numpy* compared to using Python directly. [9]

- *matplotlib*: This package enables Python programmers to create high quality plots and graphs. Visualizations are highly configurable and interoperable with several other Python packages. [10]

- *sklearn*: This package provides a potpourri of machine learning and data science models through an easy to use object-oriented API. In addition to linear regression, *sklearn* makes it possible to implement SVMs, clustering, neural networks, and much more; you will learn about a some of these models later in the course. [11]

[9] Documentation is available at https://numpy.org/

[10] Documentation is available at https://matplotlib.org/

[11] Documentation is available at https://scikit-learn.org/stable/index.html

Throughout this course, you will be asked to make use of functions defined in some of these external packages. You may not always be familiar with the usage of these functions. It is important to check the official documentation to learn about the usage and the signature of the functions.

The code snippet below uses these the three aforementioned packages to perform linear regression on any given dataset.

```
# import necessary packages
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt

# prepare train, test data
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# perform linear regression on train data
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = np.sign(linreg.predict(X_test))
```

```
# print train results
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
print('Test Acc: ', '{0:.2f}'.format(100*(pred_test==y_test).mean()))

# plot gold vs predicted value
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

For readers who are not familiar with Python, we discuss some key details. In the first section of the code, we import the relevant packages

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as mse
import matplotlib.pyplot as plt
```

As seen in this example, there are two ways to load a package. The first option is to import the full package with the *import* keyword

```
import numpy as np
```

Notice that the we can assign the imported package a customized name with the *as* keyword. In this case, we decided refer to the package *numpy* with the name *np* throughout the rest of the code. This is indeed the case when we call

```
np.sign()
```

Here we refer to the method *sign()* of the *numpy* package with the customized name *np*. Alternatively, we can selectively import particular methods or classes with the *from* keyword

```
from sklearn.model_selection import train_test_split
```

The next part of the code is preparing the train, test data.

```
X = ...
y = ...
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

X will have to be an array of arrays, and y will have to be an array of values, with the same length as X. These arrays can be defined directly by specifying each of their entries, or they could be read from some external data (most commonly a csv file). Here, we present an example dataset where $\vec{\mathbf{x}} \in \mathbb{R}^2$:

```
X = [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9]]
y = [1, 1, 1, -1, -1]
```

Then we call the *train_test_split()* method to split the dataset into data for model training and testing. Alternatively, we can split the dataset by manually slicing the data arrays. [12] In general, slicing a Python array involves the *:* operator along with start and end indices. For instance, consider an arbitrary array *a*. Then, the output of *a[i:j]* will be a subarray of *a* from the index *i* (inclusive) to the index *j* (exclusive). In the following code sample, we slice the data by specifying the number of training data points

[12] In Python, the term *slicing* refers to the process of creating a subarray of an array.

```
train_size = ...
X_train = X[:train_size]
X_test = X[train_size:]
y_train = y[:train_size]
y_test = y[train_size:]
```

Note that we have omitted some of the bounding indices. If the start index is omitted, Python assumes it to be 0 (so that the subarray is from the start of the array); for example, *X[:train_size]* is the first *train_size* entries of *X*. If the end index is omitted, Python assumes it to be *n*, the length of the array (so that the subarray ends at the end of the array); for instance, *X[train_size:]* is the remaining entries of *X*, once we remove the first *train_size* entries. Another way to slice the arrays is by specifying the number of test data points

```
test_size = ...
X_train = X[:-test_size]
X_test = X[-test_size:]
y_train = y[:-test_size]
y_test = y[-test_size:]
```

Here, notice that the index *-test_size* is a negative number. In this case, Python interprets this as *n - test_size*, where *n* is the size of the array. In other words, it is the index of the *test_size*-th element from the back of the array.

The third part of the code is fitting the linear regression model.

```
linreg = LinearRegression().fit(X_train, y_train)
pred_train = linreg.predict(X_train)
pred_test = np.sign(linreg.predict(X_test))
```

The first line will generate the least squares fit model based on the train data. Then we can have the model make predictions on the train, test data. Notice that we changed the output of the model to be the sign of the predicted values, so that we can compare them with the gold values.

Next, we print out the mean squared loss and the accuracy for the train, test data.

```
print('Train MSE: ', '{0:.4f}'.format(mse(y_train, pred_train)))
print('Test MSE: ', '{0:.4f}'.format(mse(y_test, pred_test)))
print('Train Acc: ', '{0:.2f}'.format(100*(np.sign(pred_train)==y_train).
                                      mean()))
```

```
print('Test Acc: ', '{0:.2f}'.format(100*(pred_test==y_test).mean()))
```

Notice that we use the *mse()* method that we imported from the *sklearn* package. In many cases, there are packages that perform these elementary operations for machine learning.

Finally, we plot the actual and predicted values using the *matplotlib* package.

```
plt.scatter(y_test, pred_test, c="red")
plt.xlabel("actual y value (y)")
plt.ylabel("predicted y value (y hat)")
plt.title("y vs y hat")
```

The first line draws a scatter plot with the *y_test* in the x-axis and *pred_test* in the y-axis. Notice that you can specify the color of the data points by specifying the value of the parameter *c*. In general, *parameters* are optional values you can provide to Python functions. If the values to parameters are omitted, the function will use their default values. The second and third lines specify the labels that will be written next to the axes. The final line specifies the title of the plot.

2
Statistical Learning: What It Means to Learn

Students often get confused about the meaning and significance of a relationship learnt via fitting a model to data. Some of them think such relationships are analogous to, say, a law of nature like $F = ma$, which applies every time force is applied to a mass anywhere in the universe. The main goal of this chapter is to explain the statistical nature of machine learning — models are fitted on a *particular* distribution of data points, and its predictions are valid only for data points from the same distribution. (See Chapter 18.)

2.1 *A Warm-up Example*

We work through a concrete example [1] before enunciating the general properties of statistical learning. Suppose we are studying the relationship between the following quantities for the population of Princeton: *height (H), number of exercise hours per week (E), amount of calories consumed per week (C), and weight (W)*. After collecting information from 200 randomly sampled residents, and using a $80:20$ train/test split, we perform a linear regression on the training dataset to come up with the following relationship:

$$W = 50 + H + 0.1C - 4E \tag{2.1}$$

Let's also say that the average squared residual on train and test data were both 100. This means that the relationship (2.1) holds with an error of 10 lbs on a typical test data point. [2]

Question 2.1.1. *Alice was one of the Princeton residents in the study, but the prediction of the model is very off of her actual value (squared residual is 300). Does this prove the model wrong?*

The answer is no. The least squares linear regression finds the model that minimizes the *average* squared residual across all training data points. The residual could be large for a particular individual.

[1] This example is purely hypothetical, and all numbers in this section are made up.

[2] Also, the trained model exhibits *perfect* generalization: test loss is the same as training loss!

Question 2.1.2. *There was a follow-up research for every Princeton resident who is taller than 7 feet. All of them reported squared residual of* 500*. Does this prove the model wrong?*

The answer is still no. People who are taller than 7 feet make up a tiny fraction of the entire population. Their residuals have very small effect on the *average* squared residual. The residual could be large for a small subset of the population.

Question 2.1.3. *There was a follow-up survey that tested the model on every single Princeton resident. Is it possible that the average squared residue is* 200 *for the entire population?*

The answer is yes, although it is unlikely. Consider the distribution of 4-tuples (H, E, C, W) over the entire Princeton population. This is some distribution over a finite set of 4-dimensional vectors. [3] The 200 residents we surveyed were randomly drawn from this distribution. Out of these 200 data points, 40 were randomly chosen to be held-out as test data, while the remaining 160 were used as training data. We can also say that these 40 data points were chosen at random from the distribution over the entire population of Princeton. Thus when we test the model in (2.1) on held-out data, we're testing this relationship over a random sample of 40 data points drawn from the population. 40 is a large enough number to give us some confidence that the average squared residual of the test data is a good estimate of the squared residual in the population, but just as polling errors happen during elections, there is some chance that this estimate is off. In this case, we would say that the 40 test samples were *unrepresentative* of the full population.

[3] $31,000$ vectors to be more exact. The population of Princeton is $31,000$.

It is important to remember that the training and test data are sampled from the same distribution as the population. Therefore, the average squared residual of the training and test data are only a good estimate of the squared residual of the distribution they were sampled from. This also means that the relationship found from the training data only holds (with small residue) for that *particular* distribution. If the population is different, or if the distribution shifts within the same population, the relationship is not guaranteed to hold. For example, the relationship in (2.1) is not expected to hold for people from Timbuktu, Mali (a different population), or for residents of Princeton who are taller than 7 feet (a tiny subpopulation that is likely unrepresentative of the population). Now consider the following situation:

Question 2.1.4. *It becomes fashionable in Princeton to try to gain weight. Based on the relationship in* (2.1)*, everyone decides to increase their value of C and reduce their value of E. Does the model predict that many of them will gain weight?*

The answer is no. The model was fitted to and tested on the distribution obtained before everyone tried to gain weight. It has not been fitted on the distribution of data points from people who changed their values of C and E. In particular, note that if everyone reduces their E and increases their C, then the distribution has definitely changed — the average value of the E coordinate in this distribution has decreased, whereas the average value of the C coordinate has increased.

In general, a relationship learned from a fitted model illustrates *correlation* and need not imply *causation*. The values of H, C, E in (2.1) do not *cause* W to take a specific value. The equation only shows that the values are connected via this linear relationship on average (with some bounded square residuals).

2.2 *Summary of Statistical Learning*

The above discussion leads us to summarize properties of Statistical Learning. Note that these apply to most methods of machine learning, not just linear regression.

Training/test data points are sampled from some distribution $\mathcal{D}$: In the above example, 200 residents were randomly sampled from the entire population of Princeton residents.

The learnt relationship holds only for the distribution $\mathcal{D}$ that the data was sampled from. The performance of the model on test data is an estimate of the performance of the model on the full distribution $\mathcal{D}$.

There is a small probability that the estimate using test data is off. This is analogous to polling errors in opinion polls. The computation of "confidence bounds" is discussed in Chapter 18.

2.3 *Implications for Applications of Machine Learning*

The above framework and its limitations have real-life implications.

1. *Results of medical studies may not apply to minority populations.* This can happen if the minority population is genetically distinct and constitutes only a small fraction of the population. Then test error could be large on the minority population even if it is small on average. In fact, there have been classic studies about heart disease in the 1960s whose conclusions and recommendations fail to apply well to even a group that is half of the population: females! In those days heart disease was thought to largely strike males (which subsequently turned out to be quite false) and so

the studies were done primarily on males. It turns out that heart diseases in female patients behave differently. Many practices that came out of those studies turned out to be harmful to female patients. [4]

2. *Classifiers released by tech companies in the recent past were found to have high error rates on certain minority populations.* It was quickly recognized that relying on test error alone can lead to adverse outcomes on subpopulations. [5]

3. *Creating interactive agents is difficult.* In an interactive setting (*e.g.*, an online game), a decision-making program is often called an *agent*. When an agent has to enter an extended number of interactions [6] with a human (or another agent designed by a different group of researchers, as happens in Robocup soccer [7]), then statistical learning requires that the agent to have been exposed to similar situations/interactions during training (*i.e.*, from a fixed distribution). It is quite unclear if this is true.

[4] See https://www.theatlantic.com/health/archive/2015/10/heart-disease-women/412495/.

[5] See https://time.com/5520558/artificial-intelligence-racial-gender-bias/.

[6] Later in the book we encounter Reinforcement Learning, which deals with such settings.

[7] See https://www.robocup.org/.

3
Optimization via Gradient Descent

This chapter discusses how to train model parameters through *optimization* techniques that help find the best (or fairly good) model that has low training loss. We assume that you have seen simple root-finding techniques in high school or in calculus. Optimization in machine learning often uses a procedure called *gradient descent*. This chapter assumes your knowledge of basis multivariable calculus. If you have not taken a course in multivariable calculus, read Chapter 19 to familiarize yourself with the basic definitions.

3.1 Gradient Descent

In general, a ML model has an associated loss function. The "best" model is the one that minimizes the training loss. In most cases, it is impossible or difficult to find the minimum analytically; instead, we use a numerical method called the *gradient descent algorithm* to find the (approximate) optimum.

3.1.1 Univariate Example

Let's start with an univariate example to motivate the topic. Let $f(w) = 4w^2 - 6w - 9$ be a quadratic function. Figure 3.1 shows the graph of this function.

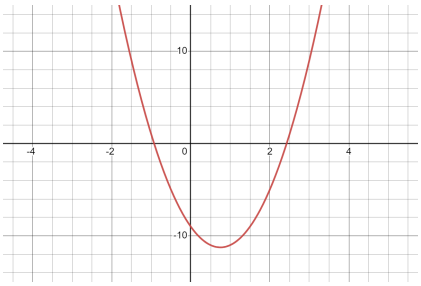

Figure 3.1: The graph of $f(w) = 4w^2 - 6w - 9$

Let's say that f attains its minimum at some point $w = w^*$. How

should we find the value of w^*? Here is an idea. Let's start from some random point on the curve and "walk down" the curve.

Notice from the graph that $f'(w^*) = 0$. Also, f is decreasing (*i.e.*, $f'(w) < 0$) when $w < w^*$ and increasing (*i.e.*, $f'(w) > 0$) when $w > w^*$. So if we examine a point w and find that $f'(w) = 0$, then we have arrived at our minimum. If $f'(w) > 0$, then we are currently on the right side of the minimum, so we need to *decrease* w. On the other hand, if $f'(w) < 0$, then we need to *increase* w.

For example, we start with the point $w = 0$. Since $f'(w) = -6 < 0$, we know that we are on the left side of the minimum, so we update $w \leftarrow 1$. Since $f'(w) = 2 > 0$, we are now on the right side of the minimum, so we update $w \leftarrow \frac{1}{2}$. When we iterate this process, we hope that we eventually slide down to the bottom of the curve. Observe that the change of value of w has the opposite sign from $f'(w)$ at that point. That is, for each step of this iteration, we can always find a $\eta > 0$ such that

$$w \leftarrow w - \eta f'(w)$$

This is not a mere coincidence — a similar result holds for a multivariate function.

3.1.2 *Gradient Descent (GD)*

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a multivariate function. If we want to "walk down" the curve of f as in the univariate case, we need to find a direction from the current point $\vec{\mathbf{w}}$ that *decreases* f.

A generalization of the Taylor expansion in the multivariable setting shows that the value of f in a small neighborhood around $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_d)$ can be approximated as a linear function in terms of the gradient.

$$f(\vec{\mathbf{w}} + \vec{\mathbf{h}}) \approx f(\vec{\mathbf{w}}) + \nabla f(\vec{\mathbf{w}}) \cdot \vec{\mathbf{h}}$$

where $\vec{\mathbf{h}} \in \mathbb{R}^d$ is small enough (*i.e.*, $\left\|\vec{\mathbf{h}}\right\| \approx 0$).

If ∇f is nonzero and we choose $\vec{\mathbf{h}} = -\eta \nabla f$ where η is a sufficiently small positive number, then

$$f(\vec{\mathbf{w}} - \eta \nabla f) \approx f(\vec{\mathbf{w}}) - \eta \|\nabla f\|_2^2$$

Since $\|\nabla f\|_2^2$ is positive, being the squared length of the vector ∇f, we conclude that the update $\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f$ causes a decrease in value of f. [1] This discussion motivates the *gradient descent algorithm*, which iteratively decreases the value of f until $\nabla f = 0$.

[1] In fact, the gradient ∇f is known as the direction of steepest increase of f. Hence, the opposite direction $-\nabla f$ is the direction of steepest decrease of f.

Definition 3.1.1 (Gradient Descent)**.** *Gradient descent is an iterative algorithm that updates the weight vector* $\vec{\mathbf{w}}$ *with the following rule:*

$$\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f(\vec{\mathbf{w}}) \tag{3.1}$$

where $\eta > 0$ is a sufficiently small positive constant, called the ***learning rate*** *or* ***step size****.*

We illustrate with an example.

Example 3.1.2. *Let $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. From Figure 3.2, we see that it attains a global minimum at $(0, 0)$. The partial derivatives of f can be calculated as:*

$$\frac{\partial f}{\partial w_1} = 2w_1(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$
$$\frac{\partial f}{\partial w_2} = 2w_2(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$

Now imagine initiating the gradient descent algorithm from the point $(0.5, 1)$ where the gradient vector is $(7.5, 15)$. One iteration of gradient descent with $\eta = 0.01$ would move from $(0.5, 1)$ to $(0.425, 0.85)$. The gradient vector at $(0.425, 0.85)$ is $(7.90, 15.81)$ and the next iteration of GD will move the point from $(0.425, 0.85)$ to $(0.35, 0.69)$. After 200 iterations, the algorithm moves the point to $(0.03, 0.06)$, which is very close to the global minimum of f.

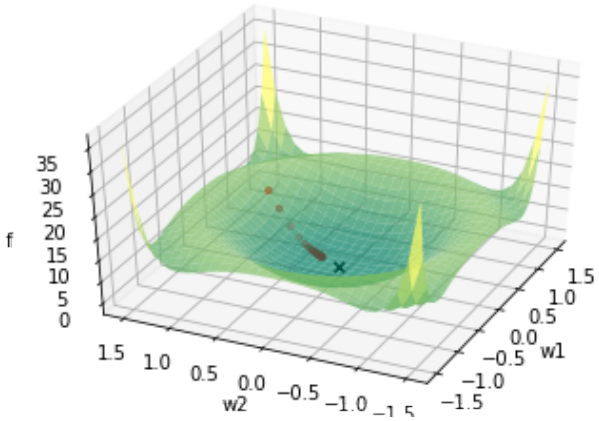

Figure 3.2: The graph of $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. The function attains a global minimum at $(0, 0)$.

3.1.3 *Learning Rate (LR)*

Choosing an appropriate learning rate is crucial for GD. Figure 3.3 shows the result of two iterations of gradient descent with a different learning rate. On the left, we see the result when λ is too small. The change of w is too small, and the loss function converges to the minimum very slowly. On the right, we see the result when λ is too big. The change of w is too large that the algorithm "shoots past" the minimum. If λ is even larger, the algorithm may even fail to converge to the minimum.

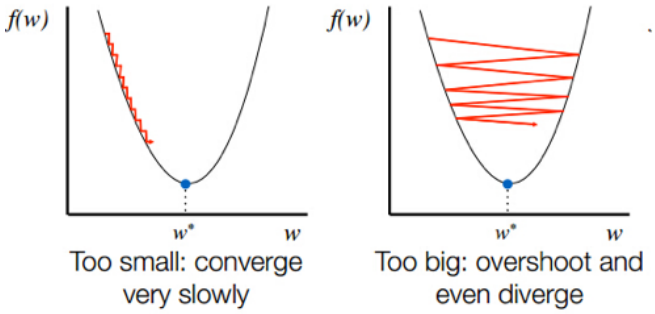

Figure 3.3: Two iterations of gradient descent with a different learning rate.

The natural question to ask is: what is the appropriate learning rate? There is some theory, and the best setting is known in some cases. But in general, it has to be set by trial and error, especially for non-convex loss functions. For instance, we start with some learning rate, say 0.5 and decrease η by $\frac{1}{2}$ if we do not observe a steady decrease in the training loss. Such heuristics are called *training schedules* and they are derived via trial and error on that *particular* dataset and model. [2]

[2] Constants whose values are decided by trial and error based on dataset and model are called *hyperparameters*. Modern ML models have several hyperparameters. Often optimization packages will suggest a default value and a fine-tuning method.

3.1.4 *Non-convex Functions*

For convex functions that are "bowl shaped," gradient descent with a small enough learning rate provably converges to the minimum solution. But for non-convex functions, the best we can hope for is converging to a point where $\nabla f = 0$. [3] Finding the global minimum of a non-convex function is NP-hard in the worst case.

In practice, loss functions are non-convex and have multiple local minima. Then, the gradient descent algorithm may converge to a different local minimum based on the initialization of the parameter vector $\vec{\mathbf{w}}$.

[3] Points where the gradient is zero are called *stationary points*, which include local minima, local maxima, and saddle points. It is possible for a GD algorithm to terminate at a saddle point, instead of the intended local minimum. There is advanced theory on how to escape saddle points, which will not be covered in this course.

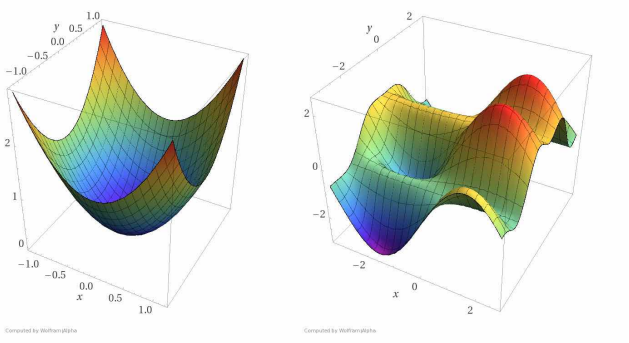

Figure 3.4: An example of a convex and a non-convex function in two variables. For non-convex functions, GD will reach a stationary point, where gradient is zero. Figure from https://www.kdnuggets.com/2016/12/hard-thing-about-deep-learning.html.

Example 3.1.3. *Consider the function* $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$*, which has two local minima at* $(-1, -1)$ *and* $(2, -1)$*. As seen in Figure 3.5, the*

local minimum that the gradient descent algorithm outputs depends on the initial point.

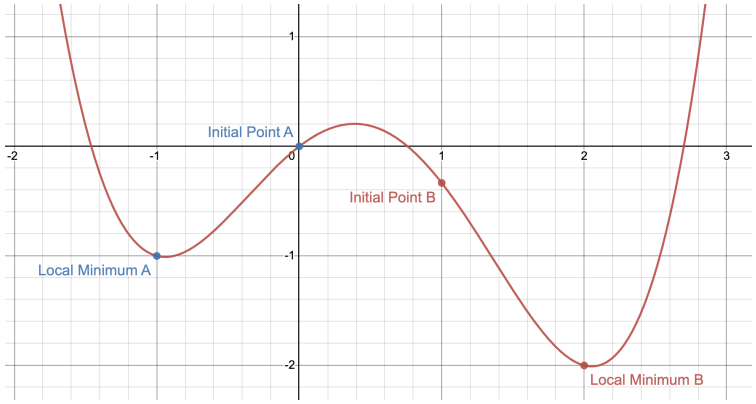

Figure 3.5: The graph of $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$ with two local minima.

3.2 *Implications of Linearity of Gradient*

The fact that gradient is a linear operator (*i.e.*, $\nabla(f_1 + f_2) = \nabla f_1 + \nabla f_2$) has great practical importance in machine learning.

Just like in (1.4), the training loss of a machine learning model is usually defined as the average (or the sum) of the loss on individual training data point. By the linearity of gradient, the gradient of the entire loss can be found by taking the sum of the gradient of the loss on individual data points.

3.2.1 *Stochastic Gradient Descent*

Since computing the gradient of the loss involves some computation on each of the data points, the computation can be quite slow for today's large data sets, which can contain millions of data points. A simple workaround is to estimate the gradient at each step by randomly sampling k data points and averaging the corresponding loss gradients. This is very analogous to opinion polls, which can also be seen as sampling from a distribution on vectors and using the average of the sample as a substitute for the population average. This algorithm is called *Stochastic Gradient Descent (SGD)*. [4] This technique works for two reasons: (1) all training data points are assumed to be sampled from the same distribution; (2) the overall training loss is just the sum/average of loss for individual data points.

[4] Some authors call this the *Batch SGD* and use the name *SGD* only for the case where $k = 1$.

3.2.2 *Mini-batch Stochastic Gradient Descent*

Today, large scale machine learning is done using special-purpose processors called *Graphical Processing Units (GPUs)*. [5] These highly

[5] As the name suggests, these were originally developed for computer graphics operations needed in computer games. Around 2012, deep learning experts realized their usefulness for deep learning. At the time writing code for GPUs was extremely difficult, but today's environments have made this much easier.

specialized architectures have the ability to perform fast parallel computations. To exploit these special capabilities, a special variant of SGD — Mini-batch SGD — can be used. Here the dataset is randomly partitioned into mini batches whose size is dictated by the degree of parallelism available in the GPU, usually a power of 2, such as 256. The members of each batch are loaded onto a different processor. Together the processors compute the gradient for one mini-batch in one go, add up the gradients to perform a single iteration for the gradient descent. Then they move on to the next batch, perform another update step, and so on.

3.2.3 *Federated Learning*

This is a conceptual framework for training a ML model on data belonging to different parties, who do not wish to hand the data over to a central server. Consider the following two examples:

1. Hospitals who wish to train an ML model on their pooled data, but who are forbidden by privacy laws to hand the data to other organizations.

2. Owners of Internet of Things (IoT) devices, who wish to benefit from training on their data but do not wish to submit the data.

In Federated Learning, the model is trained at a central server, whereas data remains with the data owners, who actively participate in the training. Users retrieve the current model parameters from the server and calculate the gradients *locally*. They send only the gradients, but not the data, to the server, and the overall gradient is calculated at the server as the weighted sum (or average) of the user gradients.

3.3 *Regularizers*

This section describes *regularization*, a useful idea that often improves generalization of the model. The main idea is that instead of minimizing the training loss function $\ell(\vec{\mathbf{w}})$, we minimize the function

$$\ell(\vec{\mathbf{w}}) + \lambda R(\vec{\mathbf{w}}) \tag{3.2}$$

where $\lambda > 0$ is a constant and $R(\vec{\mathbf{w}})$ is some non-negative function. $R(\vec{\mathbf{w}})$ is called a *regularizer* or sometimes *penalty*. We refer to (3.2) as the *regularized loss function*.

The most commonly used regularizer is the ℓ_2 regularizer where the squared ℓ_2 norm $R(\vec{\mathbf{w}}) = \|\vec{\mathbf{w}}\|_2^2$ of the weight vector is used.

Example 3.3.1. *Recall the sentiment prediction model using least squares loss. Suppose the training data consists of two data points:* $(\vec{\mathbf{x}}^1, y^1) = ((1,0,1),-1)$ *and* $(\vec{\mathbf{x}}^2, y^2) = ((1,1,0),+1)$. *Then the least squares loss, without any regularizer, can be written as*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2) \tag{3.3}$$

A little thought suggests that the minimum value of this loss is 0 *provided there exists* (w_0, w_1, w_2) *such that*

$$(-1-(w_0+w_2))^2 = 0 = (1-(w_0+w_1))^2.$$

You can verify that infinitely many solutions exist: all $\vec{\mathbf{w}}^* = (w_0, w_1, w_2)$ *that lie on the line* $(0,1,-1)+t(1,-1,-1)$ *where* $t \in \mathbb{R}$. *In other words, the loss has infinitely many minimizers.*

Now if impose an ℓ_2 *regularizer, the loss becomes*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2)+\lambda(w_0^2+w_1^2+w_2^2) \tag{3.4}$$

Any minimizer of this loss must make the gradient zero. In other words, the minimizer will satisfy the following system of linear equations:

$$\begin{cases} (2+2\lambda)w_0 + w_1 + w_2 = 0 \\ w_0 + (1+2\lambda)w_1 = 1 \\ w_0 + (1+2\lambda)w_2 = -1 \end{cases}$$

You can verify that $\vec{\mathbf{w}}^{**} = \left(0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda}\right)$ *is the unique minimizer for any* $\lambda > 0$. *For a sufficiently small value of* λ, *the corresponding* $\vec{\mathbf{w}}^{**}$ *is close enough to the line* $(0,1,-1)+t(1,-1,-1)$. *That is, it has a non-zero training loss, but the value is very close to zero. Combined with the fact it has a small norm,* $\vec{\mathbf{w}}^{**}$ *becomes the minimizer for the regularized loss.*

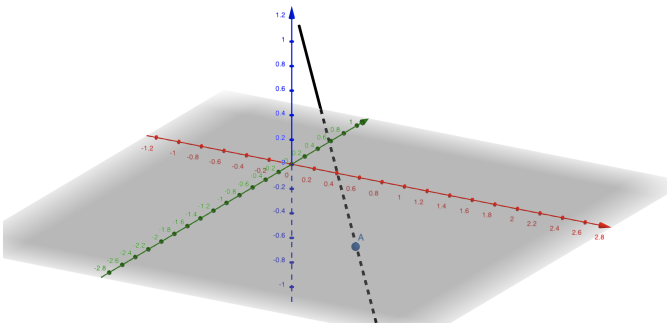

Figure 3.6: The graph of the line $(0,1,-1)+t(1,-1,-1)$ and the point $\vec{\mathbf{w}}^{**} = (0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda})$ when $\lambda = 0.01$

Note that if $\vec{\mathbf{w}}^*$ is the minimizer of $\ell(\vec{\mathbf{w}})$ and $\vec{\mathbf{w}}^{**}$ the minimizer of the regularized loss, then by definition of a minimizer, it always

holds that $\ell(\vec{\mathbf{w}}^*) \leq \ell(\vec{\mathbf{w}}^{**})$. In general, regularization ends up leading to training models with a *higher* value of $\ell(\vec{\mathbf{w}})$. This is considered acceptable because the models often generalize better. In other words, a slightly higher training loss is considered a price worth paying for a significantly lower test loss. This is illustrated by the example of sentiment prediction from Chapter 1. As hinted there, the results shown used a model trained with an ℓ_2 regularizer. The dataset involves 15k distinct words, so that is the number of model variables. There are 8k data points. Recall from Problem 1.2.5 that in such settings, there usually will exist a linear model that perfectly fits the data points. Indeed, we see in Table 3.1 that this is the case when we don't use a regularizer. However, using a regularizer prevents the model from perfectly fitting the training data. But the test loss drops tenfold with regularization.

	No regularizer	With ℓ_2-regularizer
Train MSE	0.0000	0.0727
Test MSE	7.9469	0.7523
Training accuracy	100.00%	99.55%
Test accuracy	61.67%	78.07%

Table 3.1: Training sentiment model on the SST with and without ℓ_2 regularizer.

3.3.1 *Effects of Regularization*

Here we briefly list some benefits of regularization.

1. Regularizers often help improve generalization. Above we saw a concrete example with the sentiment prediction model.

2. Adding a scalar multiple of $\|\vec{\mathbf{w}}\|_2^2$ to a function can speed up optimization by slightly reshaping the optimization landscape. The mathematical treatment of this is beyond the scope of this course.

3. Without a regularizer term, models such as logistic regression and soft-margin SVMs begin to lose their power. This will be explained when we discuss these models in Chapter 4.

3.3.2 *Why Does Regularization Help?*

The simplest answer is that we do not fully understand this concept yet. In this section, we present some intuitions derived from simple models, but keep in mind that these ideas might be misleading in more complicated models.

The usual explanation given is that the norm of the parameter vector controls the *expressiveness* or *complexity* of the model. Here "complexity" is being used in the sense of "complicatedness". By

trying to minimize loss as well as the norm of the parameter vector, the learned model tends to stay simple. [6] Whereas this discussion can be made fairly rigorous for linear models, it does not seem to apply to more complicated models: for instance regularization often helps a lot in deep learning, but the rigorous explanation appear to be at best incomplete and at worst incorrect there. [7]

Another explanation [8] is that a regularizer serves as a penalty for large weights and forces the model to choose smaller absolute values of parameters. According to this explanation, adding regularizers to a model penalizes higher-order terms or unnecessary variables and is able to avoid overfitting. Indeed, Figure 3.7 shows that the weights of the parameters in our sentiment model is significantly smaller when trained with a regularizer. But one lingering question with this explanation is: *How come attaching the **same** penalty to all variables forces the model to identify variables that are needed, and those that are not? What causes this disparate treatment of the variables?*

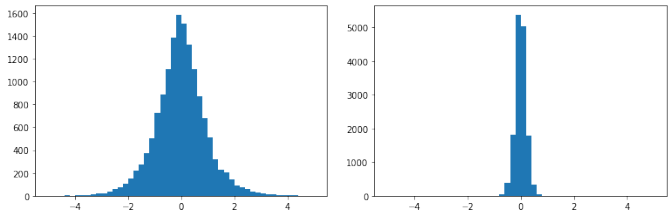

Figure 3.7: The histogram of weights of the parameters in the sentiment prediction model with (right) or without (left) an ℓ_2 regularizer.

Now consider this explanation — ℓ_2 regularization introduces a new dynamic to gradient descent, whereby gradient updates have to constantly battle against a rescaling that is always trying to whittle *all* variables down to zero. The effort succeeds only for variables where gradient updates are pushing hardest to make them nonzero. Therefore, the weights for "necessary" variables survive, while "unnecessary" variables are thrown away. To say this more precisely, consider the regularized loss $\ell(\vec{\mathbf{w}}) + \lambda \|\vec{\mathbf{w}}\|_2^2$ whose gradient is

$$\nabla \ell + 2\lambda \vec{\mathbf{w}}$$

Thus the update rule in gradient descent can be written as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t - \eta(\nabla \ell + 2\lambda \vec{\mathbf{w}}^t)$$

where $\vec{\mathbf{w}}^t$ denotes the weight vector at the t-th time step. This update rule can be rewritten as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t(1 - 2\eta\lambda) - \eta \nabla \ell \tag{3.5}$$

The first term is *down-scaling*: if for example $\eta = \lambda = 0.1$, this amounts to multiplying the current vector by 0.98, and this of course will make $\vec{\mathbf{w}}$ very small in a few hundred iterations.

[6] Recall the famous *Occam's Razor* for judging goodness of scientific theories: The simpler the theory that explains the known facts, the more likely it is to be correct. An ML model can be seen as a "theory" about relationships in the data, and thus the simplest theory is to be preferred.

[7] See the blog https://www.offconvex.org for posts about generalization and deep learning. They also discuss how other ideas such as VC dimension, which we did not cover in this course, also do not apply in deep learning.

[8] See the online lecture video by Andrew Ng. https://www.youtube.com/watch?v=KvtGD37Rm5I&ab_channel=ArtificialIntelligence-AllinOne

The second term is the gradient update. It can counteract the down-scaling by making the variables larger. But notice that the amount of change is based on how much each of the coordinates contribute to reducing the loss. Variables that are not useful will tend not to get increased by the gradient update and thus will keep getting down-scaled to low values. [9] The choice of λ mediates between these two processes.

[9] It is one of those "use it or lose it" situations!

3.4 *Gradient Descent in Programming*

In this section, we briefly discuss how to implement the Gradient Descent algorithm in Python. It is customary to use the *numpy* package to speed up computation and the *matplotlib* package for visualization.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

# initialize variables
num_iter = ...
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
x[0], y[0], eta = ...

# define functions to calculate f and grad_f
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f

# run Gradient Descent
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y

# plot the surface
xmin, xmax, ymin, ymax, n = ...
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)

fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)

# plot the trajectory of Gradient Descent
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

We first start off by importing necessary packages and initializing variables. The following code initializes *numpy* arrays of length *num_iter* + 1, with all entries initialized to 0

```
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
```

Sometimes, it is useful to make use of *np.ones()*, which will generate arrays filled with entries equal to 1.

We then define functions that will calculate the values of f and ∇f given an array of data points (x, y).

```
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f
```

This allows us to run the Gradient Descent algorithm as in

```
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y
```

Here we iteratively update the value of (x, y) using $\nabla f(x, y)$ and store each of the points in the array x and y.

We next plot the surface of the function $f(x, y)$. To start, we first create a grid of (x, y) points to evaluate $f(x, y)$ at.

```
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)
```

The function call *np.linspace(min, max, n)* generates an array of n equally spaced values from *min* to *max*. For example, the code

```
np.linspace(-2, 2, 5)
```

will create an array $[-2, -1, 0, 1, 2]$. Then *np.meshgrid(x, y)* will create a grid from the array of x values and the array of y values. We can now perform the $3D$ plotting with the following code.

```
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)
```

Feel free to change the values of the optional parameters to understand their purpose. Unlike the code for plotting a scatter plot of

linear regression in Chapter 1, here we create an object of the *Axes* class with the function *plt.figure().gca()*[10]. Then we call its instance methods to add features to it (*e.g.*, *x*-, *y*-, *z*-labels).

[10] You can read more about the differences between these two *matplotlib* interfaces at https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

Finally, we can plot the trajectory of the Gradient Descent algorithm with the code

```
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

You can alternatively call

```
ax.scatter(x, y, f(x, y))
```

but the names of optional parameters might be slightly different.

3.4.1 *Using Machine Learning Packages*

When the function f is simple and it is possible to calculate ∇f by hand, we can implement the Gradient Descent algorithm by hand as in the previous subsection. However, in most ML programs, the loss function f is very high-dimensional, and it is difficult to write a single function to directly compute the gradient ∇f. Instead, we can make use of functions defined in popular ML packages. Here, we introduce one such package called PyTorch:

- *torch*: This is a popular package used for designing and training deep learning models. PyTorch uses an object-oriented interface for user convenience and provides access to optimized array data structures called *tensors* to make computations faster and more efficient. The package also provides support for GPU training. [11]

[11] Documentation is available at https://pytorch.org/docs/stable/index.html

Using PyTorch, Gradient Descent can be implemented in just a few lines:

```
import torch
model = ...
opt = torch.optim.SGD(model.parameters(), lr=0.1)
```

The code above will create an instance of the *Optimzer* class, which has pre-defined methods that will compute the gradients and automate the Gradient Descent process.

3.4.2 *Beyond Vanilla Gradient Descent*

If you visit the documentation for the *torch.optim*, [12] you may notice that there are other algorithms listed as an alternative to the Stochastic Gradient Descent. A lot of these algorithms are extensions of the GD algorithm we explained throughout this chapter, which have proven to be more effective than the vanilla GD algorithm in certain cases (*e.g.*, Adam, Adagrad, Nesterov momentum). For example,

[12] https://pytorch.org/docs/stable/optim.html

these algorithms may choose to add a *momentum* to the gradient, so that the rate of change of f will be accelerated if it has been updating in the same direction in the recent few steps. These algorithms may also choose to use a different learning rate for each of the model parameters. In particular, the appropriate learning rate can be computed based on the mean and the variance of the gradient values from the recent few steps.

4
Linear Classification

Multi-way Classification is a task of learning to predict a label on newly seen data out of k possible labels. In *binary classification*, there are only two possible labels, say ± 1. Sentiment prediction in Chapter 1 was an example of a binary classification task. In this chapter, we introduce two other linear models that perform binary classification: logistic regression and Support Vector Machines (SVMs). From these two models, we learn more about the thought process of designing loss functions that are appropriate to the task. [1]

In this chapter, we are interested in using linear models to perform classification. In a binary classification problem, the training dataset consists of (point, label) pairs $(\vec{\mathbf{x}}, y)$ where y can take two values (*e.g.*, $\{\pm 1\}$ or $\{0, 1\}$). In a more general multi-class classification problem, the data has one of k labels, drawn from $\{0, 1, \ldots, k-1\}$.

[1] All the linear models we will study fall under an all-encompassing framework called *Generalized Linear Models*. If you ever are faced with a new situation where none of the models below are an exact match, try looking up this general framework.

4.1 General Form of a Linear Model

You already encountered a linear model in Chapter 1 — the least squares regression model for sentiment prediction. Given an input $\vec{\mathbf{x}}$, we learned a parameter vector $\vec{\mathbf{w}}$ that minimizes the loss $\sum_i (y^i - \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)^2$. The model can be seen as mapping an input vector $\vec{\mathbf{x}}$ to a real value $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$. For sentiment classification, we changed this real-valued output at test time to ± 1 by outputting $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$.

You probably wondered there: *Why don't we simply use sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$ directly as the output of the model while training?* In other words, why not do training on the following loss:

$$\sum_i (y^i - \text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i))^2 \tag{4.1}$$

The answer is that using the $\text{sign}(z)$ function in the loss makes gradient-based optimization ill-behaved. The derivative of $\text{sign}(z)$ is 0 except at $z = 0$ (where the derivative is discontinuous) and thus the gradient is uninformative about how to update the weight vector.

So the work-around in Chapter 1 (primarily for ease of exposition) was to train the sentiment classification model using the least squares loss $\sum_i (y^i - \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)^2$, which in practice is used more often in settings where the desired output y^i is real-valued output as opposed to binary. This gave OK results, but in practice one would use either of the two linear models [2] introduced in this chapter: *Logistic Regression* and *Support Vector Machines*. These are similar in spirit to the linear regression model — (1) given an input $\vec{\mathbf{x}}$, the models learn a parameter vector $\vec{\mathbf{w}}$ that minimizes a loss, defined as a differentiable function on $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$; (2) at test-time, the model outputs $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$. [3] The main difference, however, is that the loss for the linear models introduced in this chapter is designed specifically for the binary classification task. Pay close attention to our "story" for why the loss makes sense. This will prepare you to understand any new loss functions you come across in your future explorations.

[2] They are called *linear* because they use the mapping $\vec{\mathbf{x}} \mapsto \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$.

[3] There are other ways to output a discrete ± 1 label, but using the sign function is the most canonical way. We will discuss the behavior of the models at test-time later in the chapter.

4.2 Logistic Regression

The logistic regression model arises from thinking of the answer as being probabilistic: the model assigns a "probability" to each of the two labels, with the sum of the two probabilities being 1. [4] This paradigm of a probabilistic answer is a popular way to design loss functions in a host of ML settings, including deep learning.

[4] This "probability" is what is called *subjective probability*, analogous to what we mean when say things like "I am 99 percent sure my friend X will like movie Y." There is only one person X and one movie Y and they are not drawn from some probability space. Instead we're expressing a subjective feeling of near-certainty based upon past observations of person X.

Definition 4.2.1 (Logistic model)**.** *Given the input* $\vec{\mathbf{x}}$, [5] *the model assigns the "Probability that the output is* $+1$*" to be*

$$\sigma(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) = \frac{1}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} \tag{4.2}$$

where σ *is the sigmoid function (see Chapter 19). This implies that "Probability that the output is* -1*" is given by*

$$1 - \frac{1}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} = \frac{\exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})}{1 + \exp(-\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} = \frac{1}{1 + \exp(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})} \tag{4.3}$$

[5] As in Chapter 1 we assume $\vec{\mathbf{x}}$ contains a dummy coordinate x_0 that is 1 at all points: this allows us to include a constant bias term when we take the dot product $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ with the weight vector.

See Figure 4.1. Note that "the probability that the output is $+1$" is greater than $\frac{1}{2}$ precisely if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 0$. Furthermore, increasing the value of $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ causes the probability to rise towards 1. Conversely, if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < 0$, then "the probability of label -1" is greater than $\frac{1}{2}$. When $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$, the probability of label $+1$ and -1 are both equal to $\frac{1}{2}$. In this way, the logistic model can be seen as a continuous version of the $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$.

Example 4.2.2. *If* $\vec{\mathbf{x}} = (1, -3)$ *and* $\vec{\mathbf{w}} = (0.2, -0.1)$*, then the probability of label* $+1$ *is*

$$\frac{1}{1 + \exp(-0.2 - 0.3)} = \frac{1}{1 + e^{-0.5}} \simeq 0.62$$

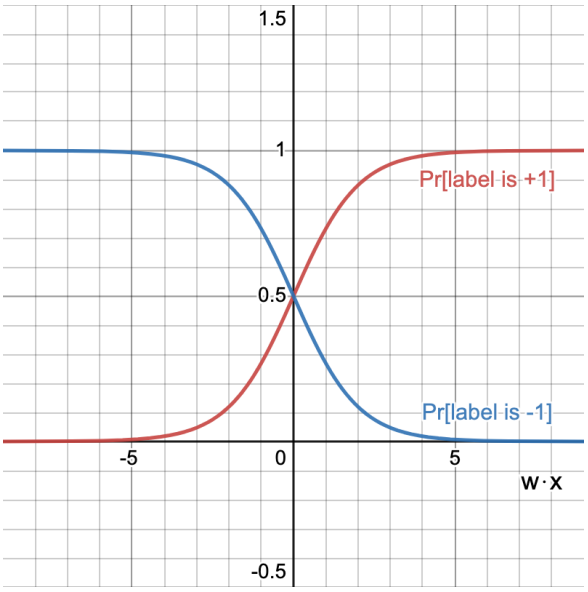

Figure 4.1: The graph of the probability that the output of a logistic model is +1 (red) or -1 (blue) given $\vec{w} \cdot \vec{x}$.

4.2.1 *Defining Goodness of Probabilistic Predictions*

Thus far, we explained how the logistic model generates its output given an input vector $\vec{x}$ and the current weight vector $\vec{w}$. But we have not yet talked about how to train the model. To define a loss function, we need to decide what are the "good" values for $\vec{w}$. Specifically, we formulate a definition of "quality" of probabilistic predictions.

Definition 4.2.3 (Maximum Likelihood Principle)**.** *Given a set of observed events, the* ***goodness*** *of a probabilistic prediction model* [6] *is the probability it assigned to the observed events.*

[6] This is a *definition* of goodness, not the consequence of some theory.

We illustrate with an example.

Example 4.2.4. *You often see weather predictions that include an estimate of the probability of rain. Table 4.1 shows the predictions by two models at the start of each day of the week. After the week is over, we have observed if it actually rained on each of the days. Based on these observations, which model made better predictions this week?*

	M	T	W	Th	F
Model 1	60%	20%	90%	50%	40%
Model 2	70%	50%	80%	20%	60%
Rained?	Y	N	Y	N	N

Table 4.1: Weather predictions by Model 1 and Model 2.

We can answer this question by seeing which model assigns higher likelihood to the events that were actually observed (i.e., whether or not it rained). For instance, the likelihood of the observed sequence according to Model 1 *is*

$$0.6 \times (1 - 0.2) \times 0.9 \times (1 - 0.5) \times (1 - 0.4) = 0.1296$$

The corresponding number for Model 2 *is* 0.0896 *(check this!). So Model* 1 *was a "better" model for this week.*

4.2.2 Loss Function for Logistic Regression

We employ the Maximum Likelihood Principle from the previous part to define the loss function for the logistic model. Suppose we are provided the labeled dataset $\{(\vec{\mathbf{x}}^1, y^1), (\vec{\mathbf{x}}^2, y^2), \ldots, (\vec{\mathbf{x}}^N, y^N)\}$ for training where y^i is a ± 1 label for the input $\vec{\mathbf{x}}^i$. By the description given in Definition 4.2.1, the probability assigned by the model with the weights $\vec{\mathbf{w}}$ to the i-th labeled data point is

$$\frac{1}{1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)}$$

which means that the total probability ("likelihood") assigned to the dataset is

$$P = \prod_{i=1}^{N} \frac{1}{1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)} \tag{4.4}$$

We desire the model $\vec{\mathbf{w}}$ that maximizes P. Since $\log(x)$ is an increasing function, the best model is also the one that maximizes $\log P$, hence the one that minimizes $-\log P = \log \frac{1}{P}$. This leads to the logistic loss function:

$$\log\left(\prod_{i=1}^{N} (1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i))\right) = \sum_{i=1}^{N} \log(1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)) \tag{4.5}$$

Note that this expression involves a sum over training data points, which as discussed in Section 3.2, is a very desirable and practical property of loss in machine learning.

Problem 4.2.5. *Verify that the gradient for the logistic loss function is*

$$\nabla \ell = \sum_{i=1}^{N} \frac{-y^i \vec{\mathbf{x}}^i}{1+\exp(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)} \tag{4.6}$$

4.2.3 Using Logistic Regression for Roommate Matching

In this part, we use the following example to illustrate some of the material covered in the previous parts.

Example 4.2.6. *Suppose Princeton University decides to pair up newly admitted undergraduate students as roommates. All students are asked to fill a questionnaire about their sleep schedule and their music taste. The questionnaire is used to generate a compatibility score in* $[0, 1]$ *for each of the two attributes, for each pair of students. Table* 4.2 *shows the calculated*

Sleep (S)	Music (M)	Compatible?
1	0.5	+1
0.75	1	+1
0.25	0	−1
0	1	−1

Table 4.2: Sample data of compatibility scores for four pairs of students.

compatibility scores for four pairs of roommates from previous years and whether or not they turned out to be compatible (+1 for compatible, −1 for incompatible).

We wish to train a logistic model to predict if a pair of students will be compatible based on their sleep and music compatibility scores. To do this, we first convert the data in Table 4.2 into a vector form.

$$\begin{aligned} \vec{\mathbf{x}}^1 &= (1, 1, 0.5) \quad & y^1 &= +1 \\ \vec{\mathbf{x}}^2 &= (1, 0.75, 1) \quad & y^2 &= +1 \\ \vec{\mathbf{x}}^3 &= (1, 0.25, 0) \quad & y^3 &= -1 \\ \vec{\mathbf{x}}^4 &= (1, 0, 1) \quad & y^4 &= -1 \end{aligned} \tag{4.7}$$

where the first coordinate x_0^i of $\vec{\mathbf{x}}^i$ is a dummy variable to introduce a constant bias term, and the second and third coordinates are respectively for sleep and music compatibility scores.

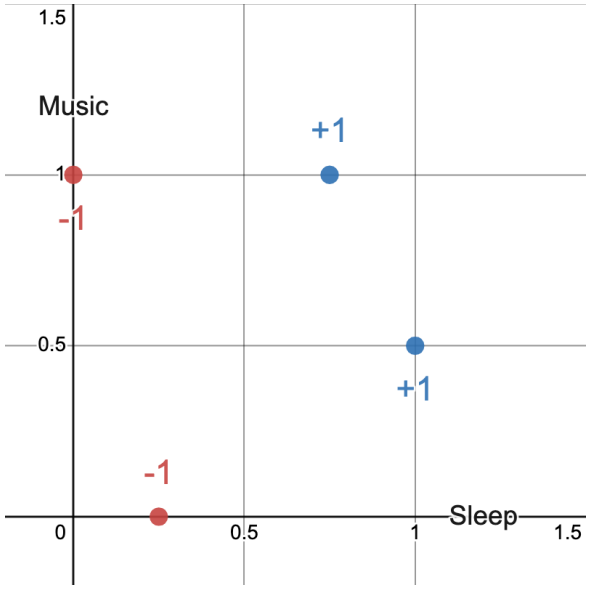

Figure 4.2: Graph representing the points in Table 4.2. The x-, y-axis in the graph correspond to the Sleep and Music compatibility scores, or the second and third coordinates in (4.7).

Consider two models — Model 1 with the weight vector $\vec{\mathbf{w}}^1 = (0, 1, 0)$ and Model 2 with the weight vector $\vec{\mathbf{w}}^2 = (0, 0, 1)$. Model 1 only looks at the sleep compatibility score to calculate the probability that a pair of students will be compatible as roommates, whereas

Model 2 only uses the music compatibility score. For example, Model 1 assigns the probability that the first pair of students are compatible as

$$\sigma(\vec{\mathbf{w}}^1 \cdot \vec{\mathbf{x}}^1) = \frac{1}{1+\exp(-1)} \simeq 0.73$$

We can calculate the probability for the other pairs and for Model 2 and fill out the following Table 4.3:

	Pair 1	Pair 2	Pair 3	Pair 4
Model 1	0.73	0.68	0.56	0.50
Model 2	0.62	0.73	0.50	0.73
Compatible?	Y	Y	N	N

Table 4.3: Roommate compatibility predictions by Model 1 and Model 2.

Then the likelihood of the observations (YYNN) according to Model 1 can be calculated as

$$0.73 \times 0.68 \times (1-0.56) \times (1-0.50) \simeq 0.11$$

where as the likelihood of the observations according to Model 2 is

$$0.62 \times 0.73 \times (1-0.50) \times (1-0.73) \simeq 0.06$$

Therefore, the Maximum Likelihood Principle tells us that Model 1 is a "better" model than Model 2.

The full logistic loss for this training data can be written as

$$\begin{aligned}\sum_{i=1}^{4} \log(1+\exp(-y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i)) = &\log(1+\exp(-(w_0 \cdot 1 + w_1 \cdot 1 + w_2 \cdot 0.5))+ \\ &\log(1+\exp(-(w_0 \cdot 1 + w_1 \cdot 0.75 + w_2 \cdot 1))+ \\ &\log(1+\exp(w_0 \cdot 1 + w_1 \cdot 0.25 + w_2 \cdot 0)+ \\ &\log(1+\exp(w_0 \cdot 1 + w_1 \cdot 0 + w_2 \cdot 1)\end{aligned}$$

and the values that minimize this loss can be found as $w_0 = -21, w_1 = 32, w_2 = 8.9$.

4.2.4 *Testing the Model*

After training the model on the training data, we can use it to define label probabilities on any new data point. However, the probabilities do not explicitly tell us what label to output on a new data point. There are two options:

1. (Probabilistic) If p is the probability of the label $+1$ according to (4.2), then use a random number generator to output $+1$ with probability p and -1 with probability $1-p$.
2. (Deterministic) Output the label with a higher probability.

Recall from an earlier discussion that $\Pr[+1] \geq \Pr[-1]$ if and only if $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} \geq 0$. In other words, the second deterministic option is equivalent to the sign(z) function: sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$!

We conclude that logistic regression is quite analogous to what we did in Chapter 1, except instead of least squares loss, we are using logistic loss to train the model. The logistic loss is explicitly designed with binary classification in mind. [7]

[7] Using logistic loss (and ℓ_2 regularizer) instead of least squares in our sentiment dataset boosts test accuracy from 78.1% to 80.7%.

4.3 *Support Vector Machines*

A *Support Vector Machine (SVM)* [8] is also a linear model. It comes in several variants, including a more powerful *kernel SVM* that we will not study here. But this rich set of variants made it an interesting family of models, and it is fair to say that in the 1990s its popularity was somewhat analogous to the popularity of deep nets today. It remains a very useful model for your toolkit. The version we are describing is a so-called *soft margin SVM*.

[8] From *An optimal algorithm for training maximum margin classifiers.* by Boser, Guyon, and Vapnik in *COLT* 1992. The name *Support Vector Machine* comes from a theorem that characterizes the optimum model in terms of "support vectors." We will not cover that theorem here.

As in the least squares regression, the main idea in designing the loss is that the label should be $+1$ or -1 according to sign$(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$. But we want to design a loss with a well-behaved gradient that provides a clearer direction of improvement. To be more specific, we want the model to have more "confident" answers, and we will penalize the model if it comes up with a correct answer but with a low degree of "confidence."

For $z \in \mathbb{R}$, let us define

$$\text{Hinge}(z) = \max\{0, 1 - z\} \tag{4.8}$$

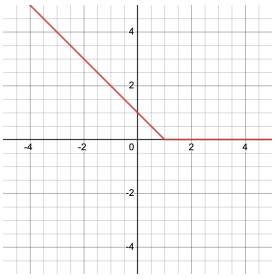

Figure 4.3: The graph of the hinge function.

Note that this function is always at least zero, and strictly positive for $z < 1$. When z decreases to negative infinity, there is no finite upper bound to the value. The derivative is zero for $z > 1$ and 1 for $z < 1$. The derivative is currently undefined at $z = 1$, but we can arbitrarily choose between 0 or 1 as the newly defined value.

For a single labeled data point $(\vec{\mathbf{x}}, y)$ where $y \in \{-1, 1\}$, the SVM loss is defined as

$$\ell = \text{Hinge}(y\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}) \tag{4.9}$$

and its gradient is

$$\nabla \ell = \begin{cases} -y\vec{\mathbf{x}} & y\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < 1 \\ 0 & y\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1 \end{cases}$$

The SVM loss for the entire training dataset can be defined as

$$\sum_i \text{Hinge}(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i) \tag{4.10}$$

that is, the sum of the SVM loss on each of the training data points.

Suppose $y = +1$. Then this loss is 0 only when $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$. In other words, making loss zero not only requires $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ to be positive, but also be comfortably above 0. If $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}}$ dips below 1, the loss is positive and increases towards $+\infty$ as $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} \to -\infty$. (Likewise if the label $y = -1$, then the loss is 0 only when $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} < -1$.)

Recall that the goal of a gradient-based optimization algorithm is to minimize the loss. Therefore, the loss gives a clear indication of the direction of improvement until the data point has been classified correctly with a comfortable *margin* away from 0, out of the *zone of confusion*.

Example 4.3.1. *Recall the roommate compatibility data from Table 4.2. Consider the soft-margin SVM with the weight vector* $\vec{\mathbf{w}} = (-1.5, 3, 0)$. *This means the decision boundary — the set of points where* $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 0$ *— is drawn at Sleep* $= \frac{1}{2}$, *and the margins — the set of points where* $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = \pm 1$ *— are drawn at Sleep* $= \frac{5}{6}$ *and Sleep* $= \frac{1}{6}$. *Figure 4.4 shows the decision boundary and the two margin lines of the model. The SVM loss is zero for the point* $(1, 0.5)$ *because it is labeled +1 and away from the decision boundary with enough margin. Similarly, the loss is zero for the point* $(0, 1)$. *The loss for the point* $(0.75, 1)$, *however, can be calculated as*

$$Hinge(1 \cdot (-1.5 \cdot 1 + 3 \cdot 0.75)) = 0.25$$

and similarly, the loss for the point $(0.25, 0)$ *is 0.25.*

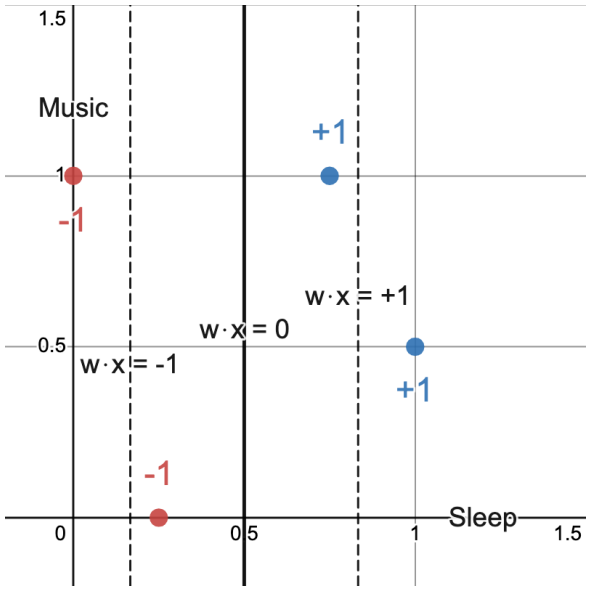

Figure 4.4: The decision boundary of a soft-margin SVM on the roommate matching example. The region to the left of the two dotted lines is where the model confidently classifies as -1; the region to the right is where it confidently classifies as $+1$; and the region between the two dotted lines is the zone of confusion.

The gradient of the loss at the point $(0.75, 1)$ *is*

$$-y\vec{\mathbf{x}} = (-1, -0.75, -1)$$

and the update rule for a gradient descent algorithm will be written as

$$\vec{\mathbf{w}} \leftarrow (-1.5, 3, 0) - 0.1(-1, -0.75, -1) = (-1.4, 3.075, 0.1)$$

where $\eta = 0.1$*, and the new SVM loss will be*

$$Hinge(1 \cdot (-1.4, 3.075, 0.1) \cdot (1, 0.75, 1)) = 0$$

which is now lower than the SVM loss before the update.

4.4 Multi-class Classification (Multinomial Regression)

So far, we have only seen problems where the model has to classify using two labels ± 1. In many settings there are k possible labels for each data point [9] and the model has to assign one of them. The conceptual framework is similar to logistic regression, except the model defines a nonzero probability for each label as follows. The notation assumes data is d-dimensional and the model parameters consist of k vectors $\vec{\theta}^1, \vec{\theta}^2, \ldots, \vec{\theta}^k \in \mathbb{R}^d$. We define a new vector $\vec{\mathbf{z}} \in \mathbb{R}^k$ where each coordinate z_i satisfies $z_i = \vec{\theta}^i \cdot \vec{\mathbf{x}}$. Then the probability of a particular label is defined through the *softmax function* (see Chapter 19):

[9] This is the case in most settings in modern machine learning. For instance in the famous ImageNet challenge, each image belongs to one of 1000 classes.

$$\begin{aligned} \Pr[\text{label } i \text{ on input } \vec{\mathbf{x}}] &= softmax(\vec{\mathbf{z}}) \\ &= \frac{\exp(\vec{\theta}^i \cdot \vec{\mathbf{x}})}{\sum_{j=1}^{k} \exp(\vec{\theta}^j \cdot \vec{\mathbf{x}})} \end{aligned} \tag{4.11}$$

This distribution can be understood as assigning a probability to label i such that it is *exponentially proportional* to the value of $\vec{\theta}^i \cdot \vec{\mathbf{x}}$.

Problem 4.4.1. *Using the result of Problem 19.2.4, verify that the definition of logistic regression as in (4.2), (4.3) are equivalent to the definition of multi-class regression as in (4.11).*

Problem 4.4.2. *Reasoning analogously as in logistic regression, derive a training loss for this model using Maximum Likelihood Principle.*

Since $\exp(z) > 0$ for every real number z the model above assigns a nonzero probability to every label. In some settings that may be appropriate. But as in case of logistic regression, at test time we also have the option of extracting a deterministic answer out of the model: the $i \in \{1, 2, \ldots, k\}$ that has the largest value of $\vec{\theta}^j \cdot \vec{\mathbf{x}}$.

4.5 Regularization with SVM

It is customary to use a regularizer, typically ℓ_2, with logistic regression models and SVMs. When a ℓ_2 regularizer is applied, the full

SVM loss is rewritten as

$$\sum_i \text{Hinge}(y^i \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^i) + \lambda \|\vec{\mathbf{w}}\|_2^2 \tag{4.12}$$

Let's see why regularization is sensible for SVMs, and even needed. The Hinge function (4.8) treats the point $z = 1$ as special. In terms of the SVM loss, this translates to the thought that having $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$ is a more "confident" classification of $\vec{\mathbf{x}}$ than just having $\text{sign}(\vec{\mathbf{w}} \cdot \vec{\mathbf{x}})$ to be correct (*i.e.*, $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 0$). But this choice is arbitrary because we have not specified the scale of $\vec{\mathbf{w}}$. If $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} = 1/10$ then scaling $\vec{\mathbf{w}}$ by a factor 10 ensures $\vec{\mathbf{w}} \cdot \vec{\mathbf{x}} > 1$. Thus the training algorithm has cheap and meaningless ways of reducing the training loss. By applying an ℓ_2 regularizer, we are able to prevent this easy route for the model, and instead, force the training to find optimal weights $\vec{\mathbf{w}}$ with a small norm.

Problem 4.5.1. *Write a justification for why it makes sense to limit the ℓ_2 norm of the classifier during logistic regression. How can large norm lead to false confidence (i.e., unrealistically low training loss)?*

4.6 Linear Classification in Programming

In this section, we briefly discuss how to implement the logistic regression model in Python. It is customary to use the *numpy* package to speed up computation and the *matplotlib* package for visualization.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt

# prepare dataset
X = ... # array of shape (n, d), each row is a d-dimensional data point
y = ... # array of shape (n), each value = -1 or +1
w = ... # array of shape (d), each value is a weight for each dimension
X_train, X_test, y_train, y_test, eta = ...

# define functions
def loss(X, y, w):
    # returns the logistic loss
    # return sum log(1 + exp(-y*w*x))
    ...

def grad_loss(X, y, w):
    # returns the gradient of the logistic loss
    # return sum (-y*x)/(1 + exp(y*w*x))
    ...

def gradient_descent(X, y, w0, eta)
    ...
    return w

# run Gradient Descent
w = gradient_descent(X_train, y_train, w, eta)
```

```
# plot the learned classifier
# assuming data is 2-dimensional
colors = {1: 'blue', -1: 'red'}
xmin, xmax, ymin, ymax = ...
plt.scatter(X[:,0], X[:,1], c=np.array([colors[y_i] for y_i in y]))
plt.plot([xmin, xmax], [ymin, ymax], c='black')
```

We have already discussed how to implement the majority of the code sample above in previous chapters. The only parts that are new are the functions to calculate the logistic loss and its gradient. This is consistent with the theme of this chapter — to discuss how to design loss functions that are appropriate for the task. Nevertheless, while the content of this code sample is familiar, some sections of the code introduce new Python functionality and syntax. We first consider the logistic loss and gradient functions:

```
def loss(X, y, w):
    # returns the logistic loss
    # return sum log(1 + exp(-y*w*x))
    ...

def grad_loss(X, y, w):
    # returns the gradient of the logistic loss
    # return sum (-y*x)/(1 + exp(y*w*x))
    ...
```

In Java, the programming language you learned in earlier programming classes, you would have to rely on a *for* loop to account for the array inputs in the *loss()* and *grad_loss()* functions. However, Python and *numpy* support many *vectorized operations*, including matrix multiplication and element-wise multiplication. These operations are far more concise to read and will also improve the runtime of the program by a great margin. Note that the code snippet above does not contain these operations; it is simply pseudo-code for your intuition. You will be introduced to these vectorized operations during the precept, and you will be expected to implement the loss function with these new tools in your programming assignments.

Next, we use a Python *dictionary* to store information corresponding to the plot's coloring scheme:

```
colors = {1: 'blue', -1: 'red'}
```

This is equivalent to a hash table from Java. Here, 1 and -1 are the *keys* and "blue" and "red" are respectively their *values*.

We will now discuss multi-dimensional arrays in Python. There are multiple ways to perform array indexing. For example, if X is a 2-dimensional array, both *X[i][j]* and *X[i, j]* can be used to extract the entry at the i-th row, j-th column. It is also possible to provide a set of rows or a set of columns to extract. The following code snippet generates an array of shape (2, 2), where each entry is from the row 0

or 1 and column 0 or 2:

```
X[[0, 1], [0, 2]]
```

Note that similar to the $1D$ case, the *:* operator is used to perform array slicing. Bounding indices can be omitted as shown in the following code snippet:

```
X[:,0]
```

This extracts the full set of rows and the column 0, or in other words, the first column of X.

Finally, we use a *list comprehension* to specify the plotting color for each data point:

```
[colors[y_i] for y_i in y]
```

This is Python syntactic sugar that allows the user to create an array while iterating over the elements of an iterator. The code snippet here is equivalent to the following code.

```
list = []
for y_i in y:
    list.append(colors[y_i])
```

5
Exploring "Data Science" via Linear Regression

So far, our treatment of machine learning has been from the perspective of a computer scientist. It is important to note, however, that models such as linear regression are useful in a variety of other fields including the physical sciences, social sciences, etc. In this chapter, we present case studies from different fields. Here, the inputs x_i are considered to be *explanatory variables*, the output y is considered to be the *effect variable*, and the weights w_i quantify the causal significance of the associated inputs x_i on the output y. The interpretation of weights as a type of causality is crucial; often, the ideal method of determining causality through a set of rigorous randomized control trials is too expensive.

5.1 Boston Housing: Machine Learning in Economics

Our first case study comes from the field of economics. In 1978, Harrison and Rubinfeld released a classic study on the willingness to pay for clean air in the Boston metropolitan area. Their methodology involved an economic model called *hedonic pricing*, [1] which essentially estimates the value of a good by breaking it down into "constituent characteristics." It turns out we can use linear regression can help determine which of these attributes are most important. Specifically, suppose we have a dataset of house sales where y represents the price of the house and $\vec{\mathbf{x}} \in \mathbb{R}^{15}$ represents a set of house attributes. [2] Then, we aim to find an optimum set of weights $\vec{\mathbf{w}}$ for the linear model:

$$y \approx \sum_{i=0}^{14} w_i x_i \tag{5.1}$$

Table 5.1 lists all 14 attributes that were used in the linear regression model. Before fitting the model with these attributes, it is useful to intuitively reason about some of the attributes. For instance, we expect the weight w_5 corresponding to RM, the number of bedrooms,

[1] This definition is paraphrased from the following Wikipedia article: https://en.wikipedia.org/wiki/Hedonic_regression

[2] x_0 is a dummy variable, and the remaining 14 coordinates $x_1, \ldots, x_{14}$ each correspond to an attribute.

Index	Code	Description
1	ZN	proportion of residential land zoned for lots over $25,000$ ft^2
2	INDUS	proportion of non-retail business acres per town
3	CHAS	Charles River dummy variable (1 if tract bounds river; 0 otherwise)
4	NOX	nitric oxides concenteration (parts per 10 million)
5	RM	average number of rooms per dwelling
6	AGE	proportion of owner-occupied units built prior to 1940
7	DIS	weighted distances to five Boston employment centres
8	RAD	index of accessibility to radial highways
9	TAX	full-value property-tax rate per \$10,000
10	MEDV	Median value of owner-occupied homes (in \$1,000s)
11	CRIM	per capita crime rate in town
12	PTRATIO	pupil-teacher ratio by town
13	LSTAT	% lower status of the population
14	B	$1000(Bk - 0.63)^2$ where Bk is the proportion of black population in town

Table 5.1: 14 attributes used in the Boston housing regression model. The attributes are presented in a different order from the paper.

to be positive because larger houses typically sell for more. Conversely, we expect the weight w_4 corresponding to NOX, the amount of air pollution, to be negative as people would prefer not to live in a polluted environment. After running the regression, it indeed turns out that these intuitions are correct. [3] In general, it can be useful to double-check that the calculated weights align with intuition: if they do not, it could be a sign that a modeling assumption is incorrect.

[3] The regression weights can be found on page 100 of the original paper. https://deepblue.lib.umich.edu/bitstream/handle/2027.42/22636/0000186.pdf?sequence=1&isAllowed=y.

5.1.1 *The Strange Math of Feature B*

The headline result of the paper is that the willingness to pay for cleaner air increases both when income level is higher and when the current pollution level is higher. However, if you read the paper closely, you may notice the presence of a curious parameter B, which is defined in terms of Bk, the proportion of black population in the neighborhood. This parameter is meant to represent a social segregation effect present within the Boston housing market. The authors of the paper speculated that (1) at a lower level of Bk, the housing price will decrease as Bk increases since the white population tend to avoid black population, but (2) at a very high level of Bk, the housing price will increase as Bk increases because black population prefer predom-

inantly black neighborhoods. To capture this intuition, they defined the attribute B in the parabolic expression $B = 1000(Bk - 0.63)^2$. It indeed turns out that the weight w_{14} corresponding to B is positive as shown in Figure 5.1.

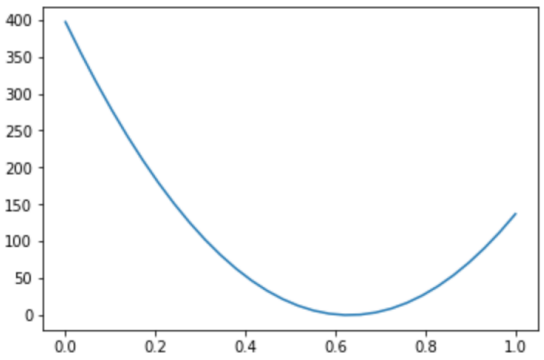

Figure 5.1: The graph of $B = 1000(Bk - 0.63)^2$. This is an example of featurization as discussed in Chapter 1. It encodes prevailing discrimination of that period. The term "black" is not favored today either.

5.1.2 *Ethnic Concerns Behind a Model*

It seems strange to have such a sensitive attribute B have an influence on the model. We might wonder about the social harm that could arise if the model was used by real-life sellers or buyers (*e.g.*, the buyers could demand a house for a lower price based on the proportion of black population in the neighborhood). On the other hand, the fitted model confirms that there is an underlying segregation effect already present in the society. Also, we cannot guarantee that the model would be race-neutral even if we eliminated the parameter B. For instance, maybe one or more of the other variables (*e.g.*, air quality variables) is highly correlated with B. [4]

[4] We will revisit such issues of bias in Chapter 16.

Ultimately, the primary takeaway from this case study is that implementing machine learning models in real life is a challenge itself. At a technical level, the model may make sense and make good predictions of house prices. But one has to consider the social effects of an ML model on the phenomenon being studied: in particular, whether it supports or extends prevailing inequities. The following are some important pointers to keep in mind:

1. If the world has a problem, the data will reflect it and *so will our models*
2. If a problematic model later gets used in real life, it can *exacerbate the existing problem*
3. The choices of attributes when making a model might *bias the outcome*

4. Carelessly using data can later *lead to modeling issues*

5.2 *fMRI Analysis: Machine Learning in Neuroscience*

We next consider an application of ML in a vastly different field. One of the most important tools in contemporary neuroscience is Functional Magnetic Resonance Imaging (fMRI). fMRI has been used successfully to map human functionality (*e.g.*, speech, memory) to brain regions. In a more active role, it can assist with tumor surgery or "decoding" thoughts and emotions.

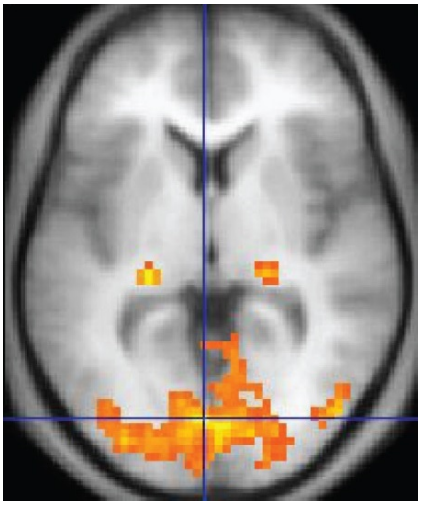

Figure 5.2: A sample image of a fMRI reading. Source: https://en.wikipedia.org/wiki/Functional_magnetic_resonance_imaging

fMRI experiments often involve presenting a set of stimuli (*e.g.*, images of human face) to the subject in order to elicit a neurological response, which is then captured through a fMRI reading. Each reading reveals the concentration of oxygen in the blood stream throughout the brain, which is used as a proxy for brain activity. [5] Through the result of the reading, we are able to conclude if a particular voxel responds to a particular stimulus. The naive way of conducting these experiments is to present one stimulus at a time and wait until we get a reading of the brain response before we move on to the next stimulus.

[5] Formally, this is referred to as the blood-oxygen-level-dependent (BOLD) signal

But if you have previously taken a course in neuroscience, you may recall that fMRI is unfortunately a double-edged sword. It features excellent spatial resolution, with each voxel as small as 1 mm^3. However, it has poor temporal resolution: often, readings require several seconds for blood flow to stabilize! Coupled with the fact that regulations limit the amount of time human subjects can spend in the scanner, it becomes clear that methodologies based on sequential presentation of stimuli are too inefficient. In this section, we explore

how to leverage techniques from linear regression in order to solve this problem.

5.2.1 *Linear Superposition*

The key intuition involves a concept called *linear superposition*: if a subject is shown multiple stimuli in quick succession, the strength of the voxel's response is the sum of the strength of its response to each of the individual stimuli. [6] Instead of waiting until we have the image of one stimulus to move on, considering showing a new stimulus every 1 or 2 seconds. Each fMRI reading will now capture the *composite* brain response to the stimuli from the past few seconds. We will use linear regression to *disentangle* the information, and extract which voxel responded to which stimulus. [7]

Consider the following example.

Example 5.2.1. *See Figure 5.3. The graph on the top left represents a voxel's response when the subject is shown the image of a face. The graph on the top right represents the response when the subject is shown the image of a flower. The bottom graph represents the response when the subject is shown the image of a flower* 1 *second after the image of a face. Notice that the first two graphs have been* ***superposed*** *to create the third graph. In practice, we are interested in the problem of extracting the individual graphs when given the superposed graph.*

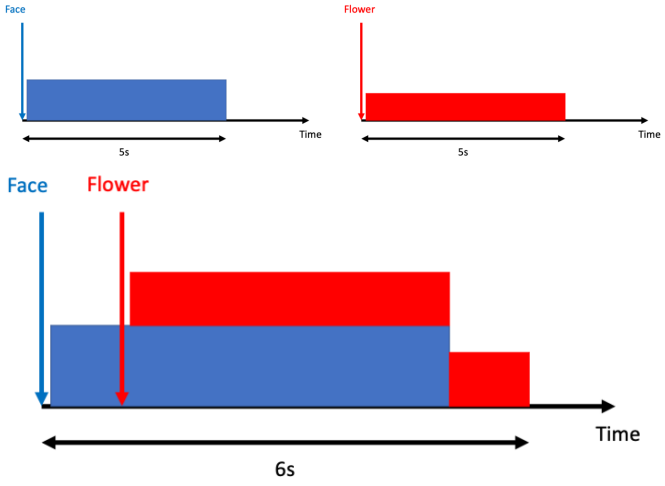

Figure 5.3: Three graphs explaining the effect of linear superposition.

[6] This is exactly like the linear superposition of wave functions in physics.

[7] Note that this is a very simplified version. The actual process is much more complicated.

5.2.2 *Linear Regression*

Now let us describe how to formulate this problem in terms of linear regression. First assume that the subject is shown one of k types of stimuli at each time step t where $t \in \{1, 2, \ldots, T\}$. Let y_t be the

response of a particular voxel at step t. The main assumption is that y_t is the linear superposition of the responses to stimuli from the steps in $[t-10, t]$. We also define a $T \times k$ matrix X with 0/1 entries, where $X_{ts} = 1$ if stimulus type s is shown during $[t-10, t]$ and 0 otherwise. Then we can set up the following linear regression model:

$$y_t \approx \sum_{s=1}^{k} w_s X_{ts}$$

When we find the optimal values of w_s via least squares, $w_s = 1$ means that the particular voxel responds to the stimulus type s.

5.2.3 *Neural Correlates of Thought*

Now we know how to find the values of w_s for a specific voxel. That is, we can test if a particular voxel responds to a particular stimulus. Combining this method with a *spatial smoothing* (*i.e.*, applying the principle that nearby voxels behave similarly), [8] we are able to identify which region of a brain is associated to which stimulus. So far, more than 1,000 regions of the brain have been identified and mapped.

[8] The simplest smoothing method is to take the w_s values for one voxel and replace them with the average of the neighboring voxels.

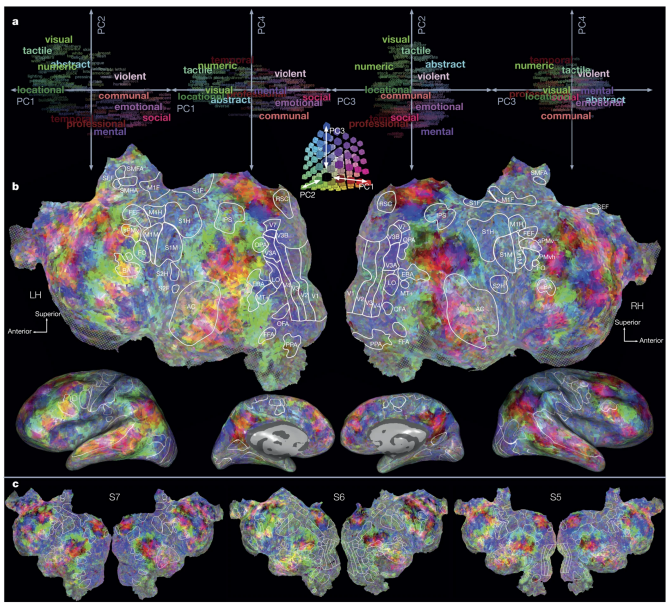

Figure 5.4: A detailed map labeling areas of the brain with corresponding stimuli. https://www.nature.com/articles/nature17637

5.2.4 *Brain-Computer Interface (BCI)*

We finish off with a tangible example of how our studies can help people. Patients who are suffering from Locked-in Syndrome (LIS) are aware of their surroundings and have normal reasoning capacities

but have *no way* of communicating with others through speech or facial movements. Using a combination of a technology called Brain-Computer Interface and a linear regression model, we are able to communicate with these patients.

Brain-Computer Interface is an electode sensor implanted near the motor cortex that can detect the electric signal that LIS patients are trying to send to the motor cortex. We can teach the patients to *visualize* writing with their dominant hand if they want to answer "no" and visualize writing with their non-dominant hand if they want to answer "yes." Since the neural correlates of the two movements are very different, BCI will pick up essentially disjoint signals, and we can use linear regression model to distinguish between them. [9]

[9] Note: training also requires labeled data, which can be produced by asking the patient questions about known facts (*e.g.*, birth date, marital status, etc.). This technique has been used to communicate with patients in deep coma and presumed to be in a vegetative state. See *Science of Mind Reading*, New Yorker, December 6 2021, which also profiles several Princeton researchers.