# 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 $\pm 1$. 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 $\pm 1$ 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 $\pm 1$ 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* $\sigma$ *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*

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

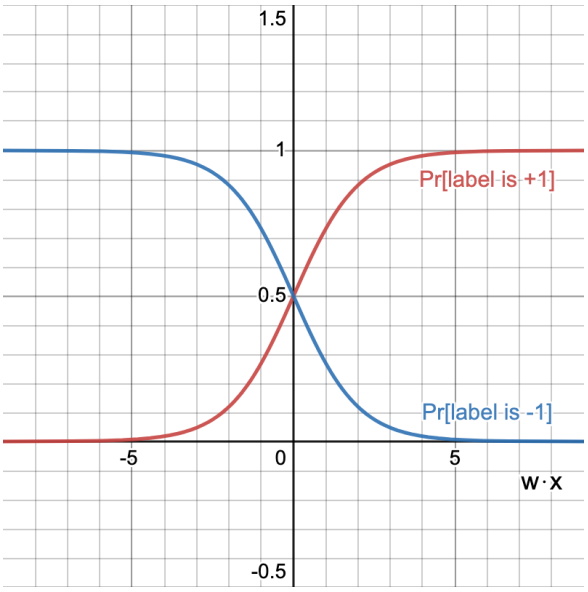


Figure 4.1: The graph of the probability that the output of a logistic model is +1 (red) or -1 (blue) given $\vec{w} \cdot \vec{x}$.

### 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 $\pm 1$ 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.

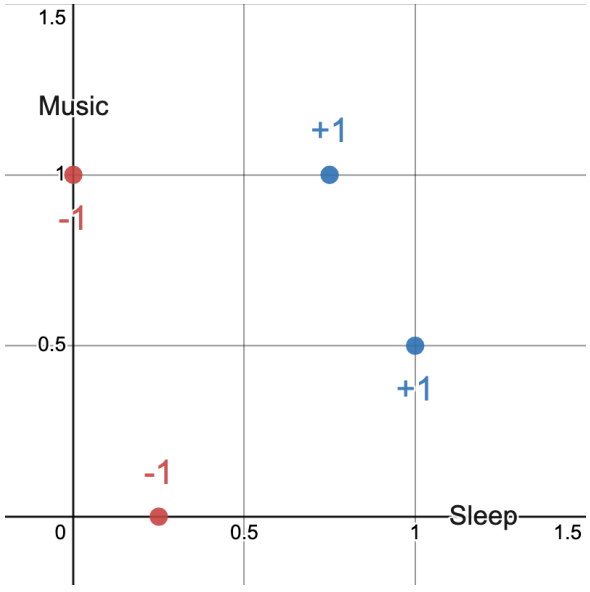


Figure 4.2: Graph representing the points in Table 4.2. The $x$-, $y$-axis in the graph correspond to the Sleep and Music compatibility scores, or the second and third coordinates in (4.7).

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 $\ell_2$ 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

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

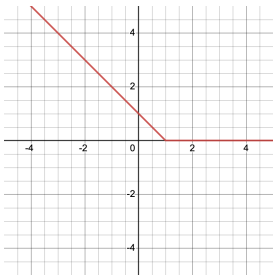

Figure 4.3: The graph of the hinge function.

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

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

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.*

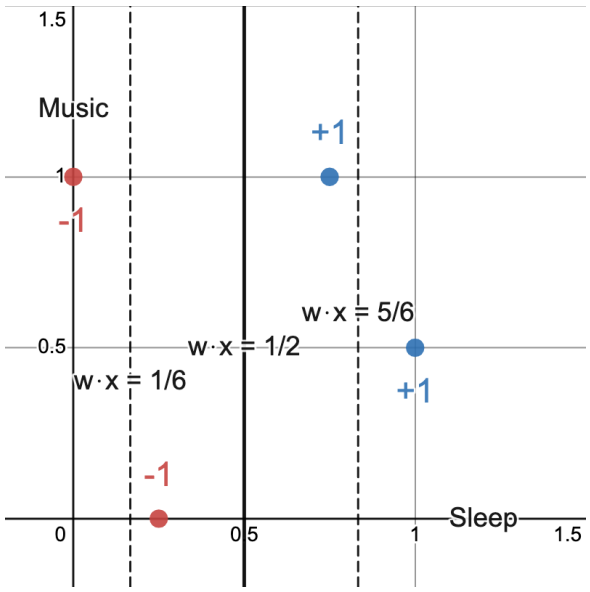


Figure 4.4: The decision boundary of a soft-margin SVM on the roommate matching example. The region to the left of the two dotted lines is where the model confidently classifies as $-1$; the region to the right is where it confidently classifies as $+1$; and the region between the two dotted lines is 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 $\pm 1$. 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 $\ell_2$, with logistic regression models and SVMs. When a $\ell_2$ 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 $\ell_2$ 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 $\ell_2$ 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])
```