
CHAPTER 1

A TOUR OF MACHINE LEARNING CLASSIFIERS USING SCIKIT-LEARN

1.1 INTRODUCTION

The goal of this chapter is to use Scikit-Learn to implement some classifiers in order to learn how to use them, his advantages, disadvantages and different use scenarios.

Specifically, we will take a look of 4 popular machine learning models commonly used in academia and in industry. In addition, we will take a look at the Scikit-Learn library, which offers a user-friendly and consistent interface for using those algorithms efficiently and productively.

1.1.1 CHOOSING A CLASSIFICATION ALGORITHM

Each algorithm has its own quirks and relies on certain assumptions. no single classifier works best across all possible scenarios (The Lack of A Priori Distinctions Between Learning Algorithms, Wolpert, David H, Neural Computation 8.7 (1996): 1341-1390).

Observation 1.1.1 (Comparasion)

In practice, it is always recommended to compare the behaviour of different algorithms in order to find the best model suitable for a particular problem; these may differ in the number of features or examples, the amount of noise in a dataset, and whether the classes are linearly separable.

Idea 1.1.1

The performance of a classifier relies upon the data that is available for learning. The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples
2. Choosing a performance metric
3. Choosing a learning algorithm and training a model
4. Evaluating the performance of the model
5. Changing the settings of the algorithm and tuning the model.

We will mainly focus on the main concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in the book.

1.1.2 FIRST STEPS WITH SCIKIT-LEARN TRAINING A PERCEPTRON

Before we learn about two related learning algorithms: the perceptron and adaline, both implemented in Python using NumPy and other libraries by ourselves.

Now we will take a look at the **scikit-learn API**.

Observation 1.1.2

One of the advantages of using scikit-learn is that combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms. Also, this library offers a not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models.

1.1.3 TRAINING A MODEL USING SCIKIT-LEARN

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in Chapter 2. For simplicity, we will use the already familiar Iris dataset throughout the following sections.

Observation 1.1.3 (Iris Dataset and Its Uses)

Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Similar to the previous chapter, we will only use two features from the Iris dataset for visualization purposes.

We will assign the petal length and petal width of the 150 flower examples to the feature matrix, X , and the corresponding class labels of the flower species to the vector array, y :

```
1 Class labels: [0 1 2]
2 Class labels: [0 1 2]
```

Code 1.1: Class Labels of Matrix X

The `np.unique(y)` function returned the three unique class labels stored in `iris.target`, and as we can see, the Iris flower class names, `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`, are already stored as integers (here: 0, 1, 2).

Observation 1.1.4 (Scikit-Learn with Class Labels and String Formats)

Although many scikit-learn functions and class methods also work with *class labels in string format*, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will *further split the dataset into separate training and test datasets*.

Idea 1.1.2 (More Info About Best Practices around Model Evaluation)

In Chapter 6, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss the best practices around model evaluation in more detail.

Using the `train_test_split` function from scikit-learn's `model_selection` module, we randomly split the X and y arrays into 30 percent test data (45 examples) and 70 percent training data (105 examples):

```
1 train_test_split(X, y, test_size=0.3, random_state=1, stratify=y
  )
```

Code 1.2: Train Test Split Function

Observation 1.1.5

Note that the `train_test_split` function already shuffles the training datasets internally before splitting; otherwise, all examples from class 0 and class 1 would have ended up in the training datasets, and the test dataset would consist of 45 examples from class 2. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, *stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset.* We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
1 >>> print('Labels counts in y:', np.bincount(y))
2 Labels counts in y: [50 50 50]
3 >>> print('Labels counts in y_train:', np.bincount(y_train))
4 Labels counts in y_train: [35 35 35]
5 >>> print('Labels counts in y_test:', np.bincount(y_test))
6 Labels counts in y_test: [15 15 15]
```

Code 1.3: Use of `bincount` Method in Python.

Idea 1.1.3 (Feature Scaling)

Many machine learning and optimization algorithms also require **feature scaling for optimal performance**, as we saw in the gradient descent example in Chapter 2. Here, we will **standardize the features using the `StandardScaler` class from scikit-learn's preprocessing module**:

```
1 sc = StandardScaler()
```

Code 1.4: `StandardScaler` Method in Python.

Using the preceding code, we loaded the `StandardScaler` class from the `preprocessing` module and initialized a new `StandardScaler` object that we assigned to the `sc` variable.

Using the `fit` method, `StandardScaler` estimated the parameters, μ (sample mean) and σ (standard deviation), for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters, μ and σ . Note that we used the same scaling parameters to standardize the test dataset so that both the values in the training and test dataset are comparable with one another.

Example 1.1.1

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the one-versus-rest (OvR) method, which allows us to feed the three flower classes to the perceptron all at once.

Finally, we have the code for the model:

```
1 from sklearn import datasets #To import Iris Dataset
2 import matplotlib.pyplot as plt
```

```

3 from matplotlib.colors import ListedColormap
4 import numpy as np
5 import sklearn.linear_model as linear_models
6 from sklearn.preprocessing import StandardScaler
7 from sklearn.model_selection import train_test_split
8
9 #Plot Regions
10
11 def plot_decision_regions(X, y, classifier, test_idx=None,
12                           resolution=0.2):
13     markers = ('o', 's', '^', 'v', '<')
14     colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
15     cmap = ListedColormap(colors[:len(np.unique(y))])
16
17     x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
18     x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
19     xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution)
20                             ,
21                             np.arange(x2_min, x2_max, resolution)
22                             )
23     lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()
24                                       ]).T)
25     lab = lab.reshape(xx1.shape)
26     plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
27     plt.xlim(xx1.min(), xx1.max())
28     plt.ylim(xx2.min(), xx2.max())
29
30     for idx, cl in enumerate(np.unique(y)):
31         plt.scatter(x=X[y == cl, 0],
32                     y=X[y == cl, 1],
33                     alpha=0.8,
34                     c=colors[idx],
35                     marker=markers[idx],
36                     label=f'Class {cl}',
37                     edgecolor='black')
38
39     if test_idx is not None:
40         X_test, y_test = X[test_idx, :], y[test_idx]
41         plt.scatter(X_test[:, 0], X_test[:, 1],
42                     c='none', edgecolor='black', alpha=1.0,
43                     linewidth=1, marker='o',
44                     s=100, label='Test set')
45
46 #Import Data
47
48 iris = datasets.load_iris()
49 X = iris.data[:,2:4]
50 y = iris.target
51
52 #Scale Standard Data
53
54 sc = StandardScaler()

```

```

51 X = sc.fit_transform(X)
52
53 #Import Perceptron
54
55 clf = linear_models.Perceptron(eta0=0.1, random_state=1)
56
57 # Split training data and test data
58 indices = np.arange(len(X))
59 X_train, X_test, idx_train, idx_test = train_test_split(
60     X, indices, test_size=0.3, random_state=1
61 )
62 y_train = y[idx_train]
63 y_test = y[idx_test]
64
65 #Fit model with training data
66
67 clf.fit(X_train, y_train)
68
69 #Plot regions
70
71 plot_decision_regions(X, y, clf, idx_test, 0.2)
72 plt.xlabel(iris.feature_names[2] + " (standardized)")
73 plt.ylabel(iris.feature_names[3] + " (standardized)")
74 plt.legend()
75 plt.show()

```

Code 1.5: Perceptron Trained Model using Sklearn

With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the examples that we want to mark on the resulting plots.

As we can see in the resulting plot, the three flower classes can't be perfectly separated by a linear decision boundary:

However, remember from our discussion in Chapter 2 that **the perceptron algorithm never converges on datasets that aren't perfectly linearly separable**, which is *why the use of the perceptron algorithm is typically not recommended in practice*. In the following sections, we will look at more powerful linear classifiers that converge to a loss minimum even if the classes are not perfectly linearly separable.

Observation 1.1.6 (Note)

The Perceptron, as well as other scikit-learn functions and classes, *often has additional parameters that we omit for clarity. You can read more about those parameters using the help function in Python (for instance, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation [here](#).*

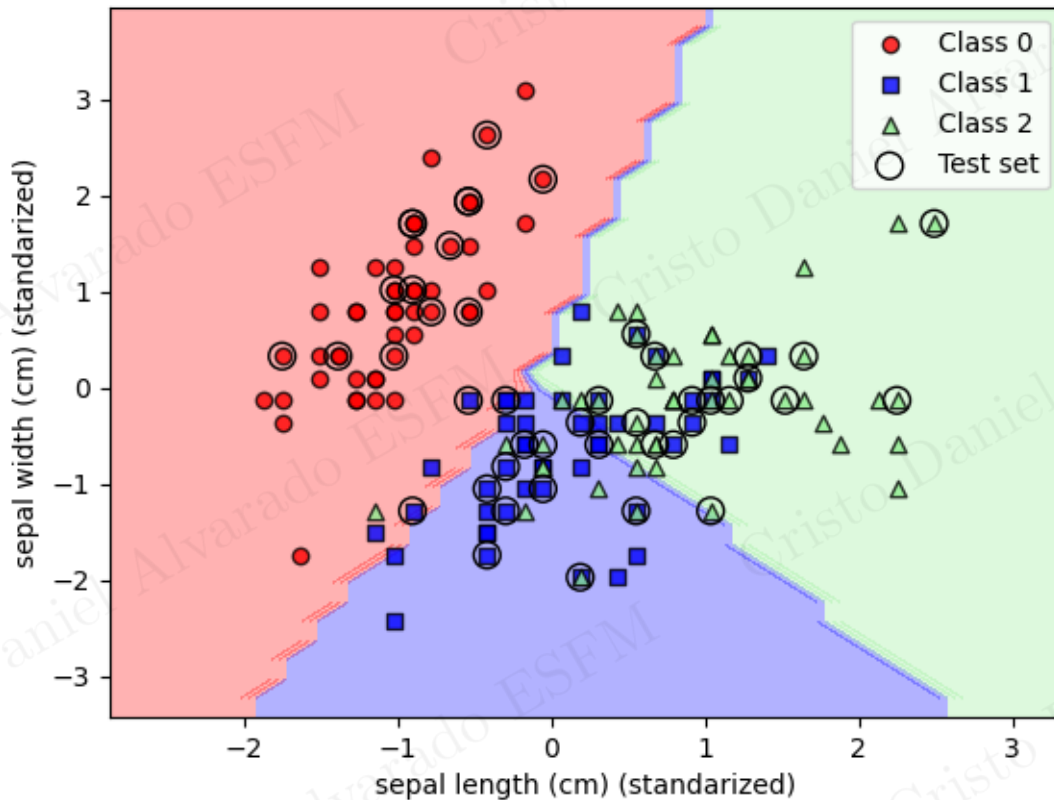


Figure 1.1: Perceptron Trained with Iris Dataset using Scikit-Learn.

1.2 MODELING CLASS PROBABILITIES VIA LOGISTIC REGRESSION

The biggest disadvantage of the Perceptron is that it never converges if the classes aren't linearly separable. The reason for this is that the weights are continuously updated because there is always at least one misclassified training example present in each epoch.

To make better use of our time, we will now look at another simple, yet more powerful, algorithm for linear and binary classification problems: **logistic regression**. Note that, despite its name, *logistic regression is a model for classification, not regression*.

1.2.1 LOGISTIC REGRESSION AND CONDITIONAL PROBABILITIES

Definition 1.2.1 (Logistic Regression)

Logistic regression is a *classification model that is easy to implement and performs very well on linearly separable classes*. It is one of the most widely used algorithms for classification in industry.

Similar to the perceptron and Adaline, *the logistic regression model in this section is also a linear model for binary classification*.

Idea 1.2.1 (Note)

Logistic regression can be generalized to multinomial logistic regression.

To explain the main mechanics behind logistic regression as a probabilistic model for binary classification, let's first introduce the odds: the odds in favor of a particular event. The odds can be

written as $\frac{p}{1-p}$, where p stands for the probability of the positive event. The term "positive event" does not necessarily mean "good," but refers to the event that we want to predict.

Example 1.2.1 (Conditional Probability)

Consider the **probability that a patient has a certain disease given certain symptoms**; we can think of the positive event as class label $y = 1$ and the symptoms as features \mathbf{x} . Hence, for brevity, we can define the probability p as $p := p(y = 1|\mathbf{x})$, the conditional probability that a particular example belongs to class 1 given its features, \mathbf{x} .

We can then further define the logit function, which is simply the logarithm of the odds (log-odds):

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) \quad (1.1)$$

Note that \log refers to the natural logarithm, as it is the common convention in computer science. The logit function takes input values in the range 0 to 1 and transforms them into values over the entire real-number range.

Under the logistic model, we assume that there is a linear relationship between the weighted inputs and the log-odds:

$$\text{logit}(p) = w_1x_1 + \cdots + w_nx_n + b = \sum_{i=1}^n w_ix_i + b = \mathbf{w}^T\mathbf{x} + b \quad (1.2)$$

Observation 1.2.1

While the preceding describes an assumption we make about the linear relationship between the log-odds and the net inputs, what **we are actually interested in is the probability p** , the *class-membership probability of an example given its features*. While the logit function maps the probability to a real-number range, *we can consider the inverse of this function to map the real-number range back to a $[0, 1)$ range for the probability p* .

This inverse of the logit function is typically called the **logistic sigmoid function**, which is sometimes simply abbreviated to *sigmoid function* due to its characteristic S-shape:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad \forall z \in \mathbb{R} \quad (1.3)$$

Here, z is the *net input, the linear combination of weights and the inputs* (that is, the features associated with the training examples):

$$z = \mathbf{w}^T\mathbf{x} + b \quad (1.4)$$

Example 1.2.2 (Graph of Sigmoid Function)

Graph of Sigmoid Function:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def sigmoid(z):
5     return 1.0 / (1.0 + np.exp(-z))
6
7 z = np.arange(-7, 7, 0.1)
8 sigma_z = sigmoid(z)
9 plt.plot(z, sigma_z)
```

```

10 plt.axvline(0.0, color='k')
11 plt.ylim(-0.1, 1.1)
12 plt.xlabel('z')
13 plt.ylabel('$\\sigma(z)$')
14 # y-axis ticks and gridline
15 plt.yticks([0.0, 0.5, 1.0])
16 ax = plt.gca()
17 ax.yaxis.grid(True)
18 plt.tight_layout()
19 plt.show()

```

Code 1.6: Sigmoid Function Graph.

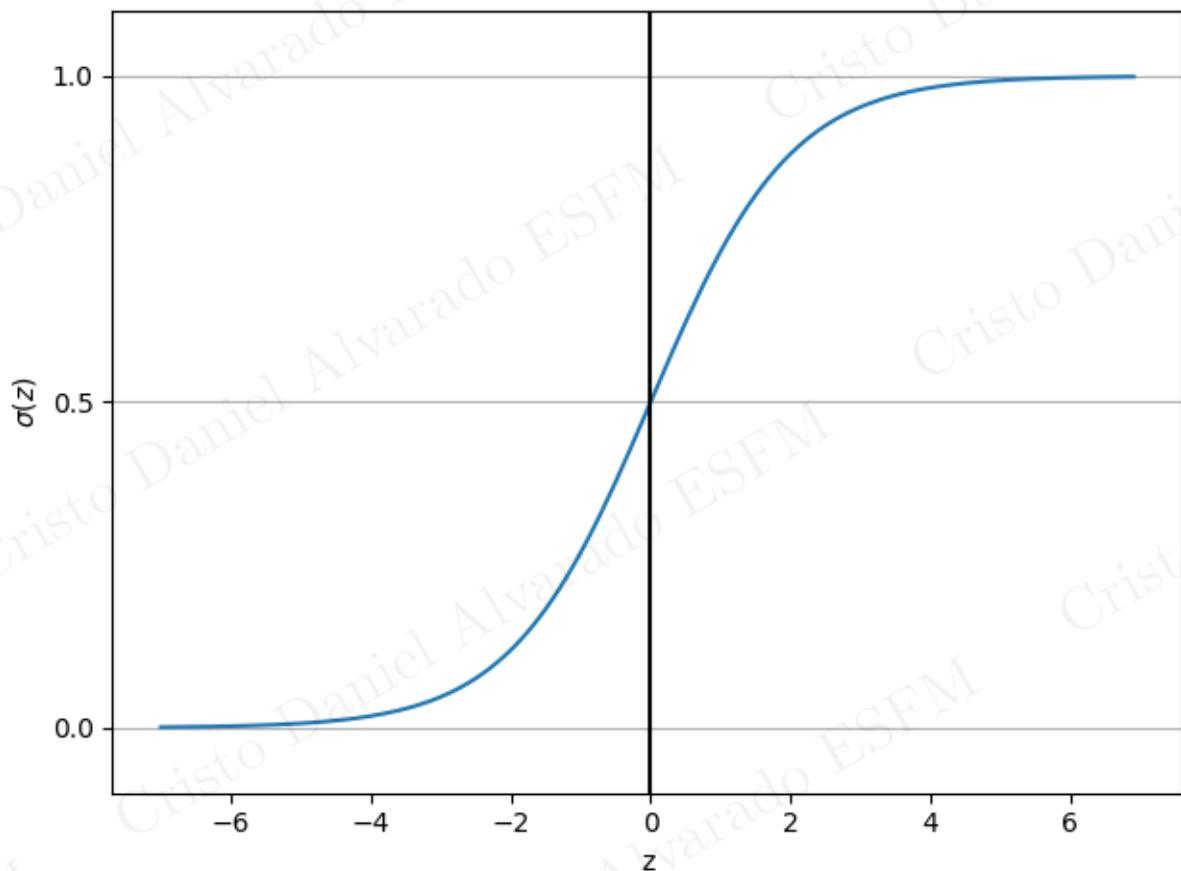


Figure 1.2: Sigmoid Function Graph.

As a result of executing the previous code, you should now see the S -shaped (sigmoidal) curve.

Observation 1.2.2 (Comparasion with Adaline)

To build some understanding of the logistic regression model, we can relate it to Adaline. In Adaline, we used the identity function, $\sigma(z) = z$ as the activation function. In logistic regression, *this activation function simply becomes the sigmoid function defined earlier.*

The only difference between Adaline and logistic regression is the activation function.

Definition 1.2.2 (Purpose of an Activation Function)

In simple terms, the purpose of an **activation function** is to decide whether a neuron in a neural network should be "activated" or not, and to what degree. It's what allows neural networks to learn and model complex, non-linear relationships.

The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\sigma(z) = p(y = 1|\mathbf{x}; \mathbf{w}, b)$, given its features \mathbf{x} , and parameterized by the weights and bias, \mathbf{w} and b .

For example, if we compute $\sigma(z) = 0.8$ for a particular flower, it means that the chance that this example is an **Iris-versicolor** flower is 80 percent. Therefore, the probability that this flower is an **Iris-setosa** can be calculated as $p(y = 0|\mathbf{x}; \mathbf{w}, b) = 1 - p(y = 1|\mathbf{x}; \mathbf{w}, b) = 0.2$, or 20 percent.

The predicted probability can then be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding plot of the sigmoid function, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Observation 1.2.3 (Interest in Predicted Class Labels)

In many applications, we are not only interested in the predicted class labels but also in the class-membership probabilities produced by the sigmoid function before applying the threshold.

Logistic regression is used in weather forecasting, for example, not only to predict whether it will rain on a particular day but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in the field of medicine.

1.3 LEARNING THE MODEL WEIGHTS VIA THE LOGISTIC LOSS FUNCTION

You have learned how we can use the logistic regression model to predict probabilities and class labels; now, let's briefly talk about how we fit the parameters of the model, for instance, the weights and bias unit, \mathbf{w} and b . Previously, we defined the mean squared error loss function as follows:

$$L(\mathbf{w}, b|\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \sigma(z^{(i)}))^2$$

We minimized this function in order to learn the parameters for our Adaline classification model. To explain how we can derive the loss function for logistic regression, let's first define the likelihood, \mathcal{L} , that we **want to maximize when we build a logistic regression model**, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$L(w, b | x) = p(y|x; w, b) = \prod_{i=1}^n p(y^{(i)} | x^{(i)}; w, b) = \prod_{i=1}^n (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{(1-y^{(i)})} \quad (1.5)$$

This is the **likelihood function** $L(w, b | x)$ in terms of a product over n terms. It is expressed as the product of probabilities $p(y^{(i)} | x^{(i)}; w, b)$ for i from 1 to n , which is further expanded to the product of $(\sigma(z^{(i)}))^{y^{(i)}}$ times $(1 - \sigma(z^{(i)}))^{(1-y^{(i)})}$, where σ is the sigmoid function.

Observation 1.3.1

In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(w, b | x) = \log(\mathcal{L}(w, b | x)) = \sum_{i=1}^n [y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))] \quad (1.6)$$

Applying the log function reduces the potential for numerical underflow if the likelihoods are very small. In addition, the product of factors becomes a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick.

1.3.1 DERIVING THE LIKELIHOOD FUNCTION

We can obtain the expression for the likelihood of the model given the data, $\mathcal{L}(w, b | x)$, as follows. Given that we have a binary classification problem with class labels 0 and 1, we can think of the label 1 as a Bernoulli variable—it can take on two values, 0 and 1, with the probability p of being 1: $Y \sim \text{Bernoulli}(p)$.

For a single data point, we can write this probability as:

$$P(Y = 1 | X = x^{(i)}) = \sigma(z^{(i)}) \quad (1.7)$$

$$P(Y = 0 | X = x^{(i)}) = 1 - \sigma(z^{(i)}) \quad (1.8)$$

Putting these two expressions together, and using the shorthand $P(Y = y^{(i)} | X = x^{(i)}) = p(y^{(i)} | x^{(i)})$, we get the probability mass function of the Bernoulli variable:

$$p(y^{(i)} | x^{(i)}) = (\sigma(z^{(i)}))^{(y^{(i)})} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}$$

Substituting the probability mass function of the Bernoulli variable, we arrive at the expression for the likelihood, which we attempt to maximize by changing the model parameters:

$$\mathcal{L}(w, b | x) = \prod_{i=1}^n (\sigma(z^{(i)}))^{(y^{(i)})} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}$$

Observation 1.3.2

We could use an optimization algorithm such as **gradient ascent** to maximize this log-likelihood function. (*Gradient ascent works the same way as gradient descent explained earlier, except that gradient ascent maximizes a function instead of minimizing it.*)

Alternatively, let's rewrite the log-likelihood as a loss function, L , that can be minimized using gradient descent:

$$L(w, b) = \sum_{i=1}^n [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))]$$

Example 1.3.1

For one single training example, looking at the equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$:

$$L(\sigma(z), y, w, b) = \begin{cases} -\log(\sigma(z)) & \text{if } y = 0 \\ -\log(1 - \sigma(z)) & \text{if } y = 1 \end{cases}$$

Let's write a short code snippet to create a plot that illustrates the loss of classifying a single

training example for different values of $\sigma(z)$. The resulting plot shows the sigmoid activation on the x axis in the range 0 to 1 (the inputs to the sigmoid function were z values in the range -10 to 10) and the associated logistic loss on the y axis:

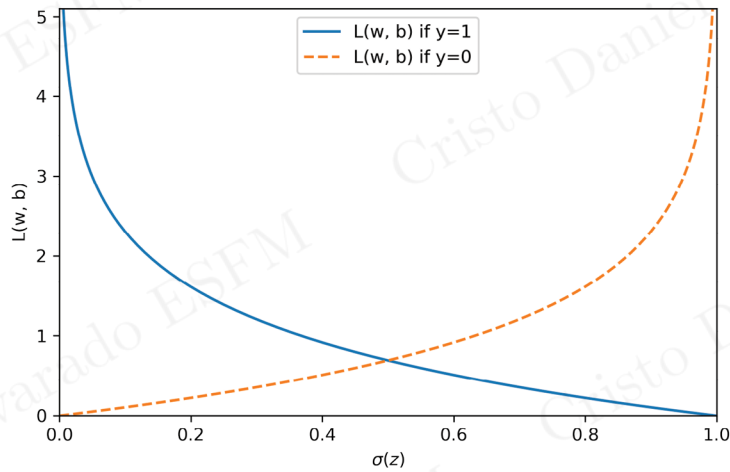


Figure 1.3: Loss Function.

We can see that the loss approaches 0 (continuous line) if we correctly predict that an example belongs to class 1. Similarly, the loss also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the loss goes toward infinity. The main point is that we penalize incorrect predictions with an increasingly larger loss.

1.4 CONVERTING AN ADALINE IMPLEMENTATION INTO AN ALGORITHM FOR LOGISTIC REGRESSION

If we implement logistic regression from scratch, we can substitute the loss function, L , in the earlier Adaline implementation with the new loss function:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))]$$

Observation 1.4.1

We use this to compute the loss of classifying all training examples per epoch. We also swap the linear activation function for the sigmoid. With those changes, the Adaline code becomes a working logistic regression implementation.

The following example uses full-batch gradient descent (the same changes can be applied to the stochastic variant):

```
1 import numpy as np
2
3 class LogisticRegressionGD:
4     """Gradient descent-based logistic regression classifier.
5
6     Parameters
7     -----
```

```

8     eta : float
9         Learning rate (between 0.0 and 1.0)
10    n_iter : int
11        Passes over the training dataset.
12    random_state : int
13        Random number generator seed for random weight
14        initialization.
15
16    Attributes
17    -----
18    w_ : 1d-array
19        Weights after training.
20    b_ : Scalar
21        Bias unit after fitting.
22    losses_ : list
23        Mean squared error loss function values in each epoch.
24    """
25    def __init__(self, eta=0.01, n_iter=50, random_state=1):
26        self.eta = eta
27        self.n_iter = n_iter
28        self.random_state = random_state
29
30    def fit(self, X, y):
31        """ Fit training data.
32
33        Parameters
34        -----
35        X : {array-like}, shape = [n_examples, n_features]
36            Training vectors, where n_examples is the
37            number of examples and n_features is the
38            number of features.
39        y : array-like, shape = [n_examples]
40            Target values.
41
42        Returns
43        -----
44        self : Instance of LogisticRegressionGD
45        """
46        rgen = np.random.RandomState(self.random_state)
47        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape
48                               [1])
49        self.b_ = np.float_(0.)
50        self.losses_ = []
51        for i in range(self.n_iter):
52            net_input = self.net_input(X)
53            output = self.activation(net_input)
54            errors = (y - output)
55            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.
56                      shape[0]
57            self.b_ += self.eta * 2.0 * errors.mean()
58            loss = (-y.dot(np.log(output))
59                  - ((1 - y).dot(np.log(1 - output))))

```

```

58         / X.shape[0])
59         self.losses_.append(loss)
60     return self
61
62     def net_input(self, X):
63         """Calculate net input"""
64         return np.dot(X, self.w_) + self.b_
65
66     def activation(self, z):
67         """Compute logistic sigmoid activation"""
68         return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
69
70     def predict(self, X):
71         """Return class label after unit step"""
72         return np.where(self.activation(self.net_input(X)) >=
73                         0.5, 1, 0)

```

Code 1.7: Implementation of Logistic Regression in Python.

When fitting a logistic regression model, remember that it only works for binary classification tasks.

The resulting decision region plot looks as follows:

1.5 GRADIENT DESCENT

One observation we can make on the latter code is that the partial derivative of the loss function L is really simple to compute, this is due to the fact:

$$\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial z} \cdot \frac{\partial z}{\partial w_j}$$

where:

$$\begin{aligned} \frac{\partial L}{\partial \sigma} &= \frac{\sigma - y}{\sigma \cdot (1 - \sigma)} \\ \frac{\partial \sigma}{\partial z} &= \sigma(1 - \sigma) \\ \frac{\partial z}{\partial w_j} &= x_j \end{aligned}$$

So,

$$\frac{\partial L}{\partial w_j} = -(y - \sigma)x_j$$

Recall that gradient descent takes steps in the opposite direction of the gradient. Hence, we flip $\nabla L(w)$ and update the j -th weight as follows, including the learning rate η :

$$w_j = w_j - \eta \cdot \frac{\partial L}{\partial w_j}(w)$$

While the partial derivative of the loss function with respect to the bias unit is not shown, bias derivation follows the same concept.

Observation 1.5.1

We should compute the mean in all this process, but due we are working in this example with a single record for simplicity.

Idea 1.5.1 (Relation between Adaline and Logistical Regression)

Both the weight and bias updates are the same as for Adaline, but the change is in the activation function.

1.6 TRAINING A LOGISTIC REGRESSION MODEL WITH SCIKIT-LEARN

We just went through useful coding and math exercises in the previous subsection, which helped to illustrate the conceptual differences between Adaline and logistic regression. Now, let's learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off the shelf.

Observation 1.6.1

Note that in recent versions of scikit-learn, the technique used for multiclass classification—multinomial or OvR—is chosen automatically.

In the following code example, we will use the `sklearn.linear_model.LogisticRegression` class as well as the familiar fit method to train the model on all three classes in the standardized flower training dataset. Also, we set `multi_class='ovr'` for illustration purposes. As an exercise, you may want to compare the results with `multi_class='multinomial'`. Note that the `multinomial` setting is now the default choice in scikit-learn's `LogisticRegression` class and recommended in practice for mutually exclusive classes, such as those found in the Iris dataset.

Observation 1.6.2

Here, "mutually exclusive" means that each training example can belong to only a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Now, let's look at the code example:

```
1 from sklearn.linear_model import LogisticRegression
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from matplotlib.colors import ListedColormap
5 from sklearn import datasets #To import Iris Dataset
6 from sklearn.preprocessing import StandardScaler
7 from sklearn.model_selection import train_test_split
8
9 def plot_decision_regions(X, y, classifier, test_idx=None,
10                          resolution=0.2):
11     markers = ('o', 's', '^', 'v', '<')
12     colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
13     cmap = ListedColormap(colors[:len(np.unique(y))])
14
15     x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
16     x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
17     xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution)
```

```

17         np.arange(x2_min, x2_max, resolution)
18     )
19     lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()
20         ]).T)
21     lab = lab.reshape(xx1.shape)
22     plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
23     plt.xlim(xx1.min(), xx1.max())
24     plt.ylim(xx2.min(), xx2.max())
25
26     for idx, cl in enumerate(np.unique(y)):
27         plt.scatter(x=X[y == cl, 0],
28             y=X[y == cl, 1],
29             alpha=0.8,
30             c=colors[idx],
31             marker=markers[idx],
32             label=f'Class {cl}',
33             edgecolor='black')
34
35     if test_idx is not None:
36         X_test, y_test = X[test_idx, :], y[test_idx]
37         plt.scatter(X_test[:, 0], X_test[:, 1],
38             c='none', edgecolor='black', alpha=1.0,
39             linewidth=1, marker='o',
40             s=100, label='Test set')
41
42 #Import Data
43
44 iris = datasets.load_iris()
45 X = iris.data[:,2:4]
46 y = iris.target
47
48 #Scale Standard Data
49
50 sc = StandardScaler()
51 X = sc.fit_transform(X)
52
53 # Split training data and test data
54 indices = np.arange(len(X))
55 X_train, X_test, idx_train, idx_test = train_test_split(
56     X, indices, test_size=0.3, random_state=1
57 )
58 y_train = y[idx_train]
59 y_test = y[idx_test]
60
61 lr = LogisticRegression(C=100.0, solver='lbfgs',
62     multi_class='ovr')
63 lr.fit(X_train, y_train)
64 plot_decision_regions(X, y, lr, idx_test, 0.01)
65
66 plt.xlabel(iris.feature_names[2] + " (standarized)")
67 plt.ylabel(iris.feature_names[3] + " (standarized)")
68 plt.legend(loc='upper left')

```

```

67 plt.tight_layout()
68 plt.show()

```

Code 1.8: Logistic Regression Iris Dataset Python.

Which produces an output like this:

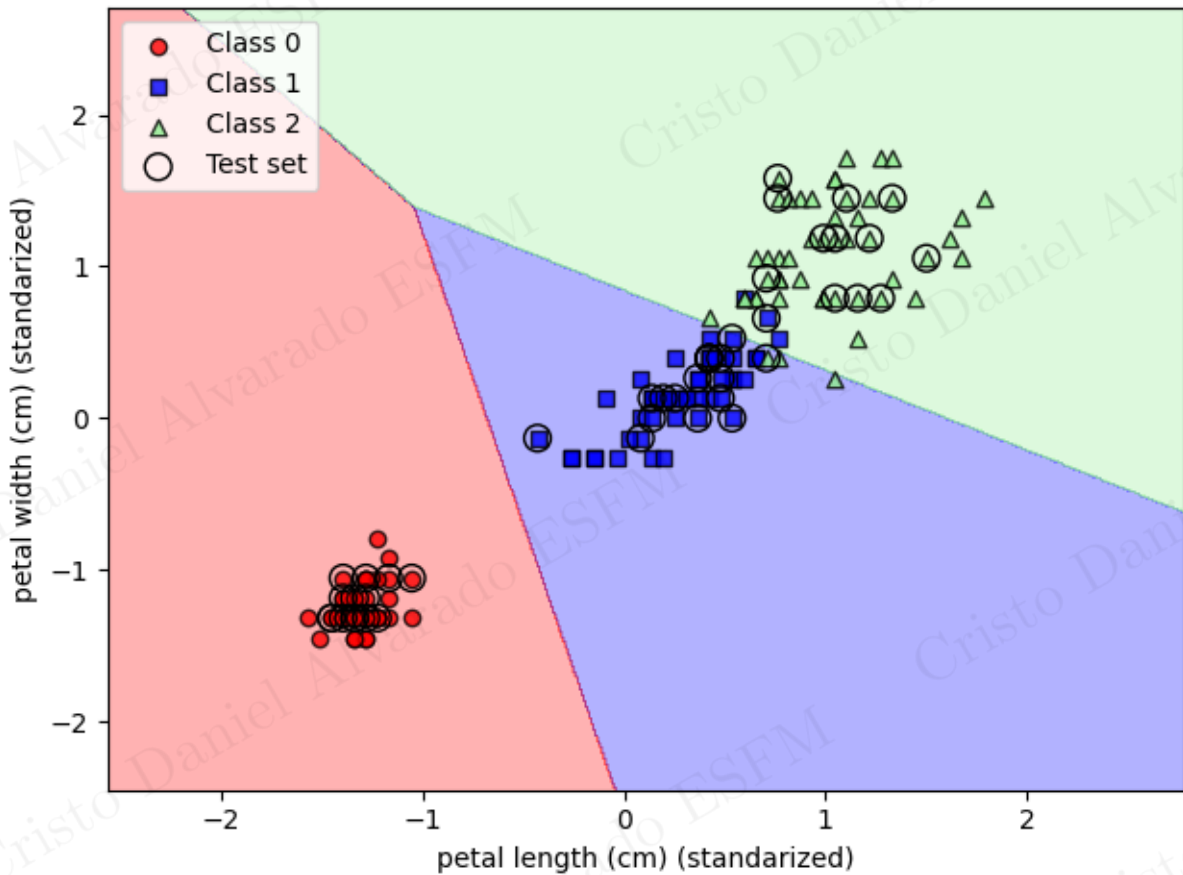


Figure 1.4: Logistic Regression on the Iris Dataset.

1.6.1 ALGORITHMS FOR CONVEX OPTIMIZATION

Many different algorithms exist for solving optimization problems. For minimizing convex loss functions, such as the logistic regression loss, it is recommended to use more advanced approaches than regular **stochastic gradient descent (SGD)**. Scikit-learn implements a range of such optimization algorithms, which can be specified via the solver parameter: `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, and `'saga'`.

While the logistic regression loss is convex, most optimization algorithms should converge to the global loss minimum with ease. However, there are certain advantages to using one algorithm over another. For example, in earlier versions (for instance, v0.21), scikit-learn used `'liblinear'` as the default, which cannot handle the multinomial loss and is limited to the OvR scheme for multiclass classification. In scikit-learn v0.22, the default solver was changed to `'lbfgs'`, which stands for the **limited-memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm** and is more flexible in this regard.

Observation 1.6.3

Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter `C`?" We will discuss this parameter in the next subsection, where we will introduce the concepts of overfitting and regularization. However,

before we move on to those topics, let's finish our discussion of class membership probabilities.

The probability that training examples belong to a certain class can be computed using the `predict_proba` method.

Example 1.6.1

For example, we can predict the probabilities of the first three examples in the test dataset as follows:

```
1 lr.predict_proba(X_test[:3, :])
```

Code 1.9: Predicting Probabilities.

This code snippet returns the following array:

```
1 array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
2        [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
3        [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

The first row corresponds to the class membership probabilities of the first flower, the second row corresponds to the second flower, and so forth. Notice that the column-wise sum in each row is 1, as expected (you can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`).

The highest value in the first row is approximately 0.85, which means that the first example belongs to class 3 (*Iris-virginica*) with a predicted probability of 85 percent.

Observation 1.6.4

As you may have noticed, we can obtain the predicted class labels by identifying the largest column in each row, for example, by using NumPy's `argmax` function:

```
1 lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

The returned class indices (corresponding to *Iris-virginica*, *Iris-setosa*, and *Iris-setosa*) are:

```
1 array([2, 0, 0])
```

In the preceding code example, we converted conditional probabilities into class labels manually by using NumPy's `argmax` function. In practice, the more convenient way of obtaining class labels when using scikit-learn is to call the `predict` method directly:

```
1 lr.predict(X_test_std[:3, :])
2 array([2, 0, 0])
```

Lastly, if you want to predict the class label of a single flower example, scikit-learn expects a two-dimensional array as input. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as shown here:

```
1 lr.predict(X_test[0, :].reshape(1, -1))
2 array([2])
```

1.7 TACKLING OVERFITTING VIA REGULARIZATION

Definition 1.7.1 (Overfitting)

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data).

If a model suffers from overfitting, we also say that the model has **high variance**, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data.

Observation 1.7.1 (Underfitting)

Similarly, a model can suffer from **underfitting (high bias)**, which means it is not complex enough to capture the pattern in the training data well and therefore also performs poorly on unseen data.

Although we have only encountered linear models for classification so far, the problems of overfitting and underfitting can be illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries.

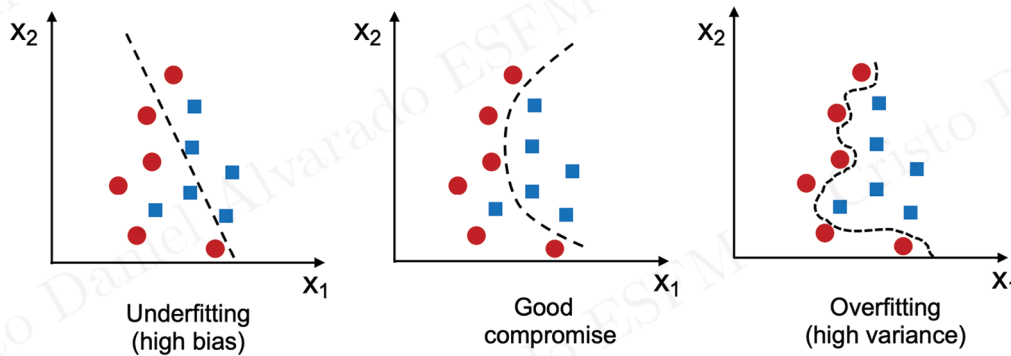


Figure 1.5: Examples of Underfitting, Good Compromise and Overfitting.

1.7.1 THE BIAS-VARIANCE TRADEOFF

Definition 1.7.2 (Tradeoff)

Tradeoff is a balance achieved between two desirable but incompatible features; a compromise.

Researchers often use the terms **bias** and **variance** or **bias-variance tradeoff** to describe model performance—that is, you may encounter talks, courses, or articles where people say that a model has high variance or high bias.

In general, high variance is proportional to overfitting and high bias is proportional to underfitting.

Observation 1.7.2 (Variance in Machine Learning)

In the context of machine learning models, *variance measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times on different subsets of the training dataset.*

In contrast, *bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.*

One way of finding a good **bias-variance tradeoff** is to **tune the complexity of the model via regularization**.

Definition 1.7.3 (Regularization)

Regularization is a *useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.*

The concept behind regularization is to *introduce additional information to penalize extreme parameter (weight) values.*

Observation 1.7.3 (L2 Regularization)

The most common form of regularization is **L2 regularization** (sometimes called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2n} \|w\|^2 = \frac{\lambda}{2n} \sum_{j=1}^m w_j^2$$

Here, λ is the **regularization parameter**. The $2n$ in the denominator is merely a scaling factor, such that cancels then