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 impelement 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 recomended 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`:

```
Class labels: [0 1 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):

```
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:

```
>>> print('Labels counts in y:', np.bincount(y))
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
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**:

```
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:

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

```
from matplotlib.colors import ListedColormap
import numpy as np
import sklearn.linear_model as linear_models
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

#Plot Regions

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.2):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
```

```python
X = sc.fit_transform(X)

#Import Perceptron

clf = linear_models.Perceptron(eta0=0.1, random_state=1)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

#Fit model with training data

clf.fit(X_train, y_train)

#Plot regions

plot_decision_regions(X, y, clf, idx_test, 0.2)
plt.xlabel(iris.feature_names[2] + " (standarized)")
plt.ylabel(iris.feature_names[3] + " (standarized)")
plt.legend()
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:

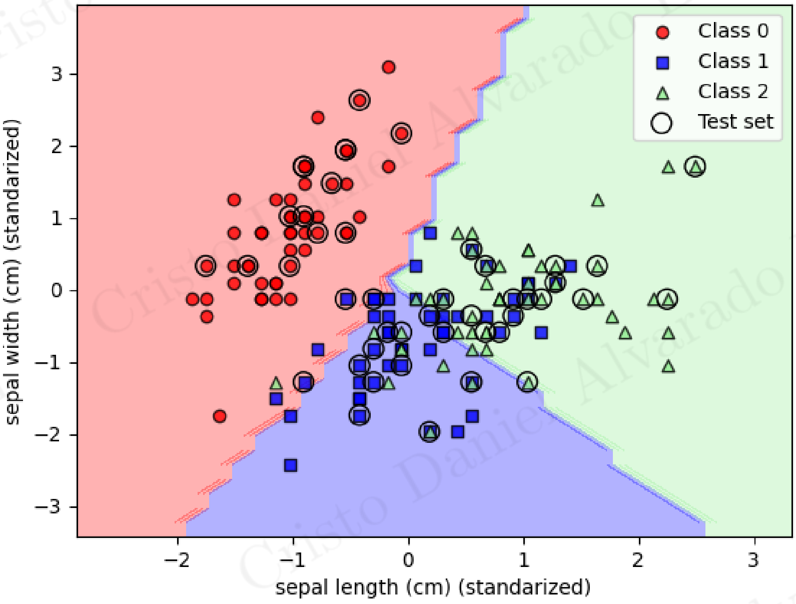


Figure 1.1: Perceptron Trained with Iris Dataset using Scikit-Learn.

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.

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 poch.

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 $\boldsymbol{x}$. Hence, for brevity, we can define the probability p as $p := p(y = 1|\boldsymbol{x})$, the conditional probability that a particular example belongs to class 1 given its features, $\boldsymbol{x}$.

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

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

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**:

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

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*:

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

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

$$z = \boldsymbol{w}^T\boldsymbol{x} + b \tag{1.4}$$

Example 1.2.2 (**Graph of Sigmoid Function**)
Graph of Sigmoid Function:

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

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
sigma_z = sigmoid(z)
plt.plot(z, sigma_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\\sigma (z)$')
# y-axis ticks and gridline
plt.yticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.yaxis.grid(True)
plt.tight_layout()
plt.show()
```

Code 1.6: 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.

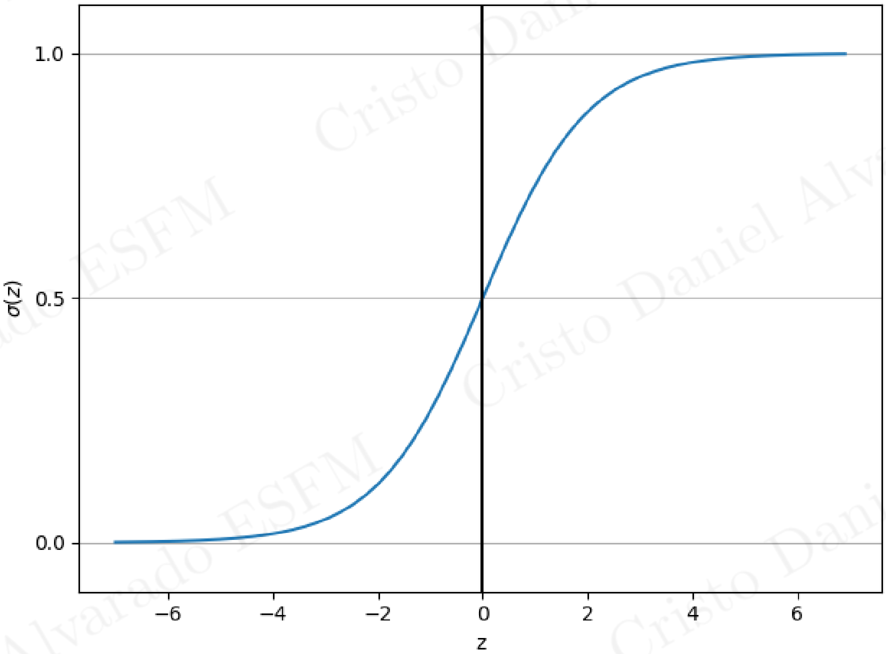


Figure 1.2: Sigmoid Function Graph.

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, $\sigma(z) = p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b)$, given its features $\boldsymbol{x}$, and parameterized by the weights and bias, $\boldsymbol{w}$ 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 $p(y = 0|\boldsymbol{x}; \boldsymbol{w}, b) = 1 - p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b) = 0.2$, 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, $\boldsymbol{w}$ and b.* Previously, we defined the mean squared error loss function as follows:

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

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:

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

This is the **likelihood function** $L(w, b \mid x)$ in *terms of a product over n terms*. It is expressed as the product of probabilities $p(y^{(i)} \mid x^{(i)}; w, b)$ 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:

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

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, $\mathcal{L}(w, b \mid x)$, 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:

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

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

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

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

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:

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

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:

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

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 $y=1$:

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

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:

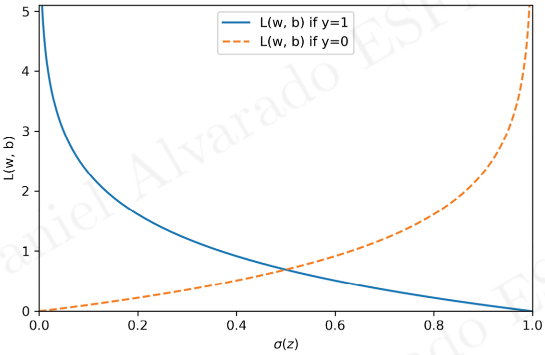


Figure 1.3: Loss Function.

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:

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

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):

```python
import numpy as np

class LogisticRegressionGD:
    """Gradient descent-based logistic regression classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after training.
    b_ : Scalar
      Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the
          number of examples and n_features is the
          number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : Instance of LogisticRegressionGD
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (-y.dot(np.log(output))
- ((1 - y).dot(np.log(1 - output)))
                    / X.shape[0])
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X)) >= 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:

```
from sklearn.linear_model import LogisticRegression
import numpy as np
import matplotlib.pyplot as plt
```

```python
from matplotlib.colors import ListedColormap
from sklearn import datasets #To import Iris Dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.2):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

# Split training data and test data
```

```
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

lr = LogisticRegression(C=100.0, solver=’lbfgs’,
                        multi_class=’ovr’)
lr.fit(X_train, y_train)
plot_decision_regions(X, y, lr, idx_test, 0.01)

plt.xlabel(iris.feature_names[2] + " (standarized)")
plt.ylabel(iris.feature_names[3] + " (standarized)")
plt.legend(loc=’upper left’)
plt.tight_layout()
plt.show()
```

Code 1.8: Logistic Regression Iris Dataset Python.

Which produces an output like this:

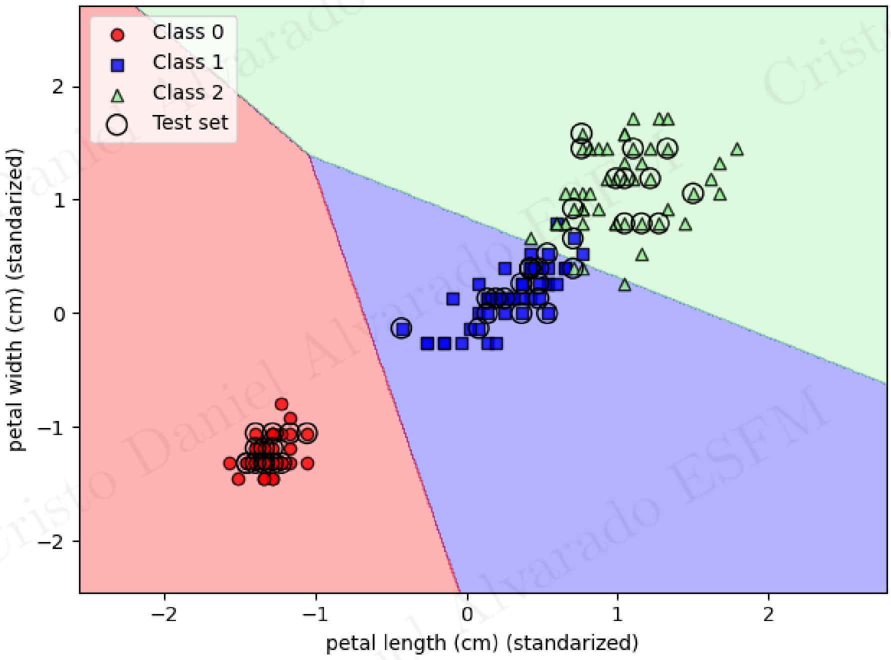


Figure 1.4: Logistic Regression on the Iris Dataset.

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:

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

Code 1.9: Predicting Probabilities.

This code snippet returns the following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [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:

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

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

```
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:

```
lr.predict(X_test_std[:3, :])
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:

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

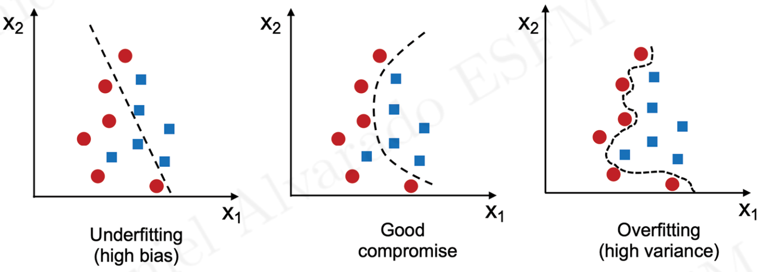


Figure 1.5: Examples of Underfitting, Good Compromise and Overfitting.

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 d*escribe 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 2 in the denominator is merely a scaling factor, such that cancels when computing the loss gradient. The sample size n added to scale the regularization term similarly to the loss.

Observation 1.7.4 (**Importance of Feature Scaling**)
Regularization is another reason why feature scaling such as standaraization is important. For regularization to work properly, all features must be on comparable scales.

The loss function for logistic regression can be regularized by adding a regularization term, which shrinks the weighs during model training:

$$\frac{\partial L}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left(w^T x^{(i)} - y^{(i)}\right) x_j^{(i)}\right)$$

by adding the regularization term to the loss changes the partial derivative to the following form:

$$\frac{\partial L}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left(w^T x^{(i)} - y^{(i)}\right) x_j^{(i)}\right) + \frac{\lambda}{n} w_j$$

Observation 1.7.5 (**Information About the Regularization Parameter** λ)
Vía the regularization parameter λ we can control how closely we fit the training data while keeping the weights small. By increasing λ, we increase the regularization strength. Note that the bias unit, essentially an intercept term or negative threshold, is usually not regularized.

1.7.2 Implementation of Regularization

In the following example, the paramenter `c` implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines. The term `c` is inversely proportional to the regularization parameter λ. Consecuently, decreasing `c` increases the value of λ and therefore, increases the regularization strength.

Observation 1.7.6
We can visualize this behaviour by plotting the $L2$ regularization path for the two weight coefficients.

```python
from sklearn.linear_model import LogisticRegression
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets #To import Iris Dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(
        C=10.**c,
        solver=’lbfgs’,
        max_iter=1000
    )
    lr.fit(X_train, y_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)
weights = np.array(weights)
plt.plot(params, weights[:, 0],
         label=’Petal length’)
plt.plot(params, weights[:, 1], linestyle=’--’,
         label=’Petal width’)
plt.ylabel(’Weight coefficient’)
plt.xlabel(’C’)
plt.legend(loc=’upper left’)
plt.xscale(’log’)
plt.show()
```

Code 1.10: Logistic Regression Regularized with different values of `c` on the Iris Dataset Python.

Executing this code fits 10 logistic regression models with different values for the inverse regularization parameter `c`. For illustration purposes, we only collected the weight coefficients of class 1 (the second class on the iris dataset, Iris-versicolor) versus all classifiers.

Observation 1.7.7
Here, we are using OvR technique for multiclass classification.

As shown in the resulting plot, the wright coefficients shrink when `c` decreases-that is, when the regularization strength increases:

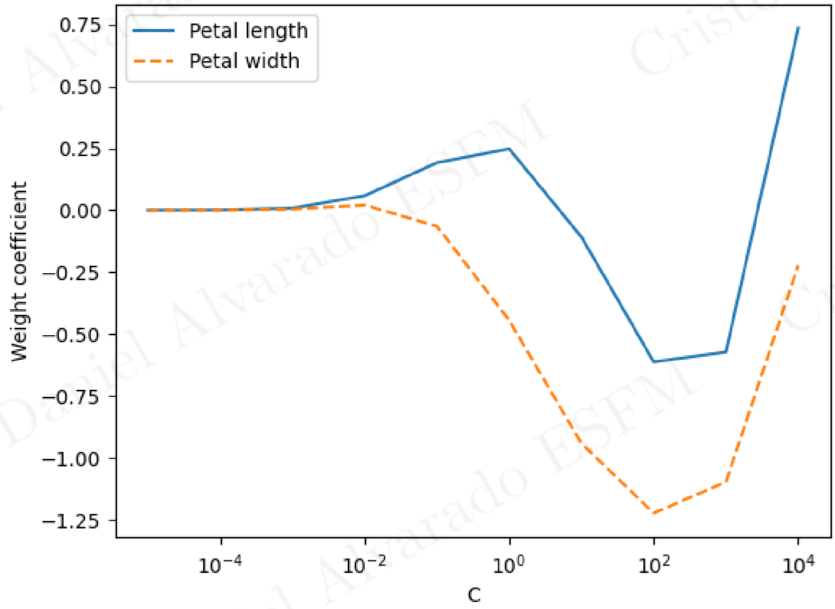


Figure 1.6: Weights for Different Values of `c`.

We have to keep in mind that:

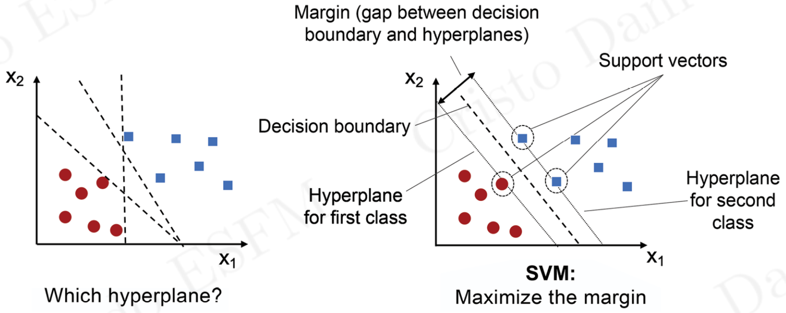
Increasing the regularization strength **can reduce overfitting**, but if *the strength is too high and the weight coefficients approach zero*, the **model can perform very poorly due to underfitting**.

1.8 Maximum Margin Classification with Support Vector Machines

Another powerful and widely used learning algorithm is the **support vector machine (SVM)**, which can be considered an extension of the perceptron.

Observation 1.8.1 (**Key Idea of Support Vector Machines**)
Using the perceptron algorithm, we minimized misclassification errors. In SMVs, our optimization objective is to **maximize** the margin.

Definition 1.8.1 (Margin)
The **margin** is defined as *the distance between the separating hyperplane (called desicion boundary) and the training examples that are closest to this hyperplane*, which are the so called **support vectors**.



x2
x1
Which hyperplane?
Margin (gap between decision boundary and hyperplanes)
Support vectors
Decision boundary
Hyperplane for first class
Hyperplane for second class
x2
x1
SVM:
Maximize the margin

Figure 1.7: SVM and Perceptron Various Boundaries.

Idea 1.8.1 (Reduce Overfitting)
The key idea behind having decision bouyndaries with large margins is that they tend to have a lower generalization error, whereas models with small margines are more prone to overfitting.

While the main intuition behind SVMs is relatively simple, the mathematics behind them is advanced and requieres sound knowledge of constrained optimization.

Observation 1.8.2
The following resources are recomended if interested in learning more:

- Chris J.C. Burges' explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data Mining and Knowledge Discovery, 2(2): 121-167, 1998)
- Vladimir Vapnik's book *The Nature of Statistical Learning Theory*, Springer Science+Business Media, Vladimir Vapnik, 2000
- Andrew Ng's lecture notes.

1.8.1 Dealing with a Nonlinearly Separable Case Using Slack Variables

Let's mention the slack variable introduced by Vladimir Vapnik in 1995 and led to the so-called **soft-margin classification**.

Idea 1.8.2
The motivation for introducing the slack variable was that the *linear constraints in the SVM optimization are objective need to be relaxed for nonlinearly separable data to allow the converegence of the optimization in the presence of misclassifications*, under appropiate loss penalization.

The use of slack variable introduces a parameter commonly referred to as C in SVM. We can consider C as a hyperparameter controlling the penalty for misclassification.

Large values of C correspond to *large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for* C. We can then use the C parameter to *control the width of the margin and therefore tune the bias-variance trade-off.*

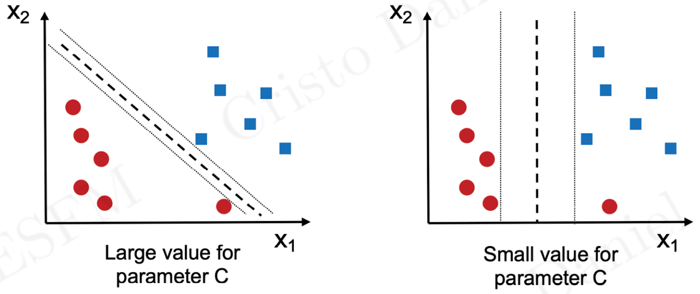


Figure 1.8: Parameter C in the Bias-Variance Trade-off.

Idea 1.8.3
This concept is related to regularization, which we discussed in the previous section in the context of regularized regression, where *decreasing the value of C increases the bias (underfitting) and lowers the variance (overfitting) of the model.*

1.8.2 Training a Model with a Support Vector Machine

Now that we have learned the basic concepts behind a linear SVM, let's train an SVM model to classify the different flowers in our Iris dataset:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets #To import Iris Dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.01):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train, y_train)
plot_decision_regions(X,
                      y,
                      classifier=svm,
                      test_idx=idx_test)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Code 1.11: Implementation of SVM on Python and Graph of Regions.

1.8.3 Alternative Implementations in Scikit-learn

The scikit-learn library's LogisticRegression class, which we used in the previous sections, can make use of the LIBLINEAR library by setting solver='liblinear'. LIBLINEAR is a highly optimized C/C++ library developed at the National Taiwan University (Liblinear).

Similarly, the SVC class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (Libsvm).

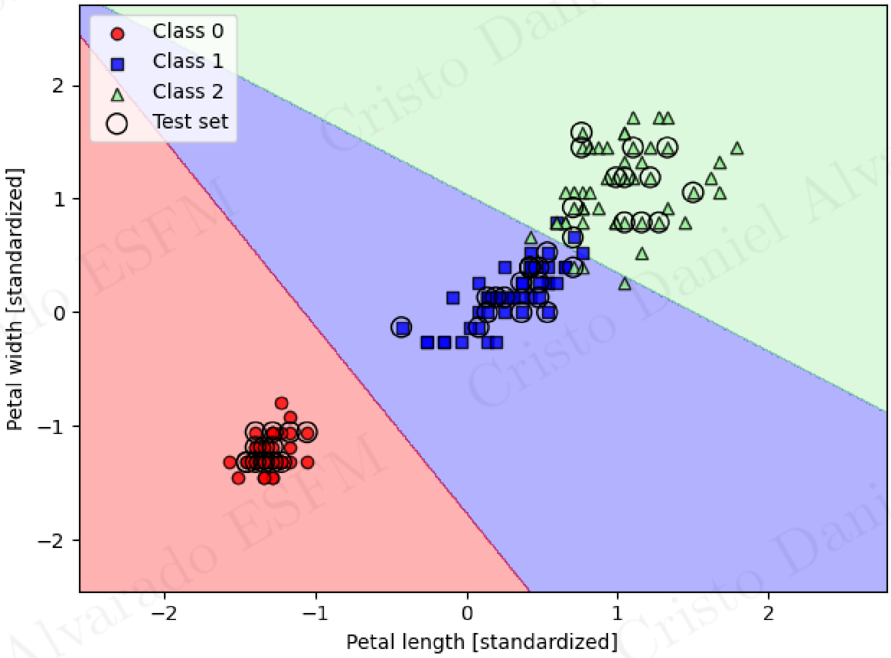


Figure 1.9: Implementation of SVM and Plot of Regions.

The advantage of using LIBLINEAR and LIBSVM over, for example, native Python implementations is that they allow the **extremely quick training of large amounts of linear classifiers**. However, *sometimes our datasets are too large to fit into computer memory.* Thus, scikit-learn also offers alternative implementations via the SGDClassifier class, which also supports online learning via the partial_fit method. The concept behind the SGDClassifier class is *similar to the stochastic gradient algorithm previously implemented for Adaline.*

Observation 1.8.3
We could initialize the SGD version of the perceptron (loss='perceptron'), logistic regression (loss='log'), and an SVM with default parameters (loss='hinge') as follows:

```
from sklearn.linear_model import SGDClassifier

ppn = SGDClassifier(loss='perceptron')
lr = SGDClassifier(loss='log')
svm = SGDClassifier(loss='hinge')
```

Code 1.12: caption

Definition 1.8.2 (SGDClassifier)
SGDClassifier es una *clase de scikit-learn que implementa clasificadores lineales entrenados usando Stochastic Gradient Descent (SGD).*

1.9 Solving Nonlinear Problems Using a Kernel SVM

Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily **kernelized** to solve nonlinear classification problems.

Before we discuss the main concept behind the so-called kernel SVM, the most common variant of SVMs, let's first create a synthetic dataset to see what such a nonlinear classification problem may look like.

1.9.1 Kernel Methods for Linearly Inseparable Data

Using the following code, we will create a simple dataset that has the form of an XOR gate using the logical_xor function from NumPy, where 100 examples will be assigned the class label 1, and 100 examples will be assigned the class label -1:

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0,
                       X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, 0)
plt.scatter(X_xor[y_xor == 1, 0],
            X_xor[y_xor == 1, 1],
            c=’royalblue’, marker=’s’,
            label=’Class 1’)
plt.scatter(X_xor[y_xor == 0, 0],
            X_xor[y_xor == 0, 1],
            c=’tomato’, marker=’o’,
            label=’Class 0’)
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.xlabel(’Feature 1’)
plt.ylabel(’Feature 2’)
plt.legend(loc=’best’)
plt.tight_layout()
plt.show()
```

Code 1.13: XOR in Python.

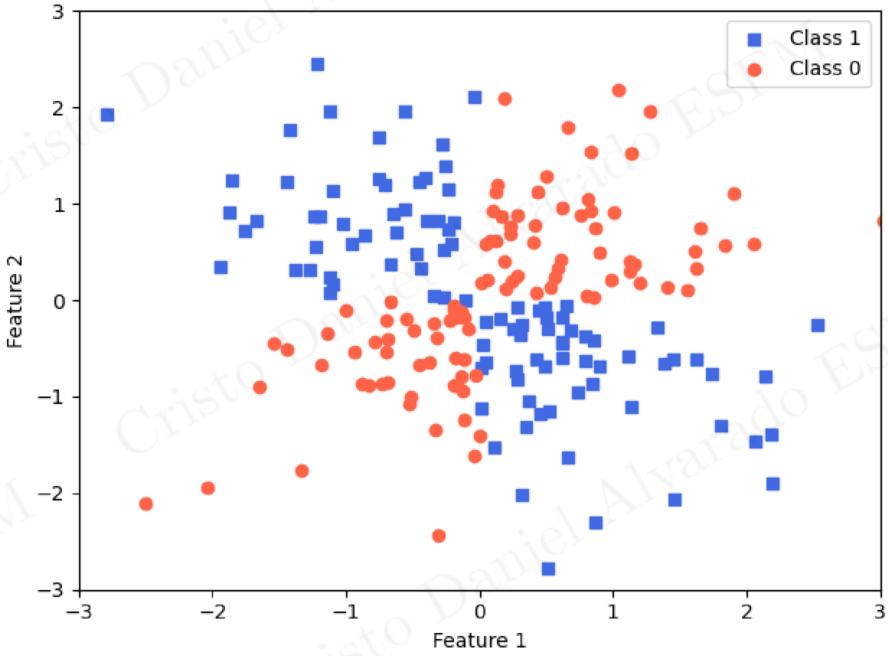


Figure 1.10: XOR Graph in Python.

Obviously, we would not be able to separate the examples from the positive and negative class very well using a linear hyperplane as a decision boundary via the linear logistic regression or linear SVM model that we discussed earlier.

Idea 1.9.1
The basic idea behind kernel methods for dealing with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, φ, where the data becomes linearly separable. We can transform a two-dimensional dataset into a new three-dimensional feature space, where the classes become separable via the following projection:

$$\begin{array}{rccl} \phi: & \mathbb{R}^2 & \to & \mathbb{R}^3 \\ & (x_1, x_2) & \mapsto & (x_1, x_2, x_1^2 + x_2^2) = (z_1, z_2, z_3) \end{array}$$

This idea allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space, as illustrated with this dataset:

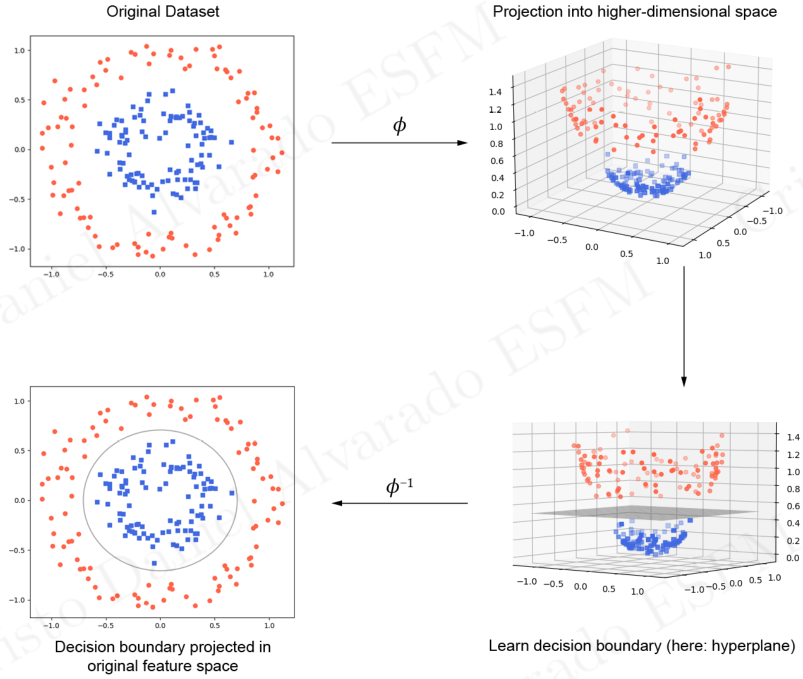


Figure 1.11: Dataset and Projection of Decision Boundary.

1.9.2 Using the Kernel Trick to Find Separating Hyperplanes in a High-Dimensional Space

To solve a nonlinear problem using an SVM, we would *transform the training data into a higher-dimensional feature space via a mapping function, ϕ, and train a linear SVM model to classify the data in this new feature space.* Then, we could use the same mapping function, ϕ, to transform new, unseen data to classify it using the linear SVM model.

Observation 1.9.1 (Problem with Construction of New Features)
However, one problem with this mapping approach is that *the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data.*

This is where the so-called **kernel trick** comes into play. Although we did not go into much detail about how to solve the quadratic programming task to train an SVM, in practice, we just need to replace the dot product $x^{(i)T}x^{(j)}$ by $\phi(x^{(i)})^T\phi(x^{(j)})$. To save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function:

$$k(x^i, x^j) = \phi(x^i)^T\phi(x^j)$$

One of the most widely used kernels is the radial basis function (RBF) kernel, which can simply be called the Gaussian kernel:

$$k(x^i, x^j) = \exp\left(-\frac{\|x^i - x^j\|^2}{2\sigma^2}\right)$$

This is often simplified to:

$$k(x^i, x^j) = \exp\left(-\gamma\|x^i - x^j\|^2\right)$$

Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter to be optimized.

Observation 1.9.2
Roughly speaking, the term "kernel" can be interpreted as a *similarity function between a pair of examples.*

The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, *the resulting similarity score will fall into a range between 1* (for exactly similar examples) *and 0* (for very dissimilar examples).

Now that we have covered the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the SVC class from scikit-learn that we imported earlier and replace the `kernel='linear'` parameter with `kernel='rbf'`.

Note: Fit the SVM model and plot decision regions.

As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:

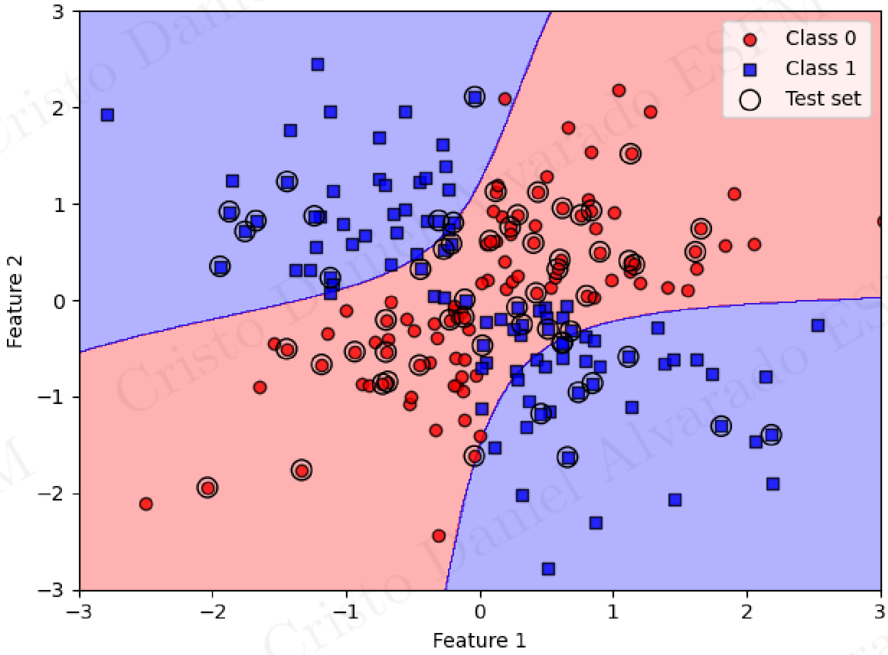


Figure 1.12: XOR Separation Using RBF.

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.svm import SVC
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.01):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

np.random.seed(1)

X = np.random.randn(200, 2)
y = np.logical_xor(X[:, 0] > 0,
                       X[:, 1] > 0)
y = np.where(y, 1, 0)

"""

plt.scatter(X_xor[y_xor == 1, 0],
            X_xor[y_xor == 1, 1],
            c='royalblue', marker='s',
            label='Class 1')

plt.scatter(X_xor[y_xor == 0, 0],
            X_xor[y_xor == 0, 1],
            c='tomato', marker='o',
            label='Class 0')
"""

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

#Train model
svm = svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
svm.fit(X_train, y_train)

#plot info
plot_decision_regions(X, y, classifier=svm, test_idx=idx_test)

plt.xlim([-3, 3])
plt.ylim([-3, 3])

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

Code 1.14: Decision Boundary using RBF Kernel on the XOR Dataset.

The γ parameter, which we set to gamma $= 0.1$, can be understood as a *cut-off parameter for the Gaussian sphere*; increasing γ *increases the influence of the training examples*, leading to a tighter and bumpier decision boundary. To get a better understanding of γ, let's apply an RBF kernel SVM to our Iris flower dataset:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import datasets #To import Iris Dataset
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.01):
    markers = ('o', 's', '^', 'v', '<')
```

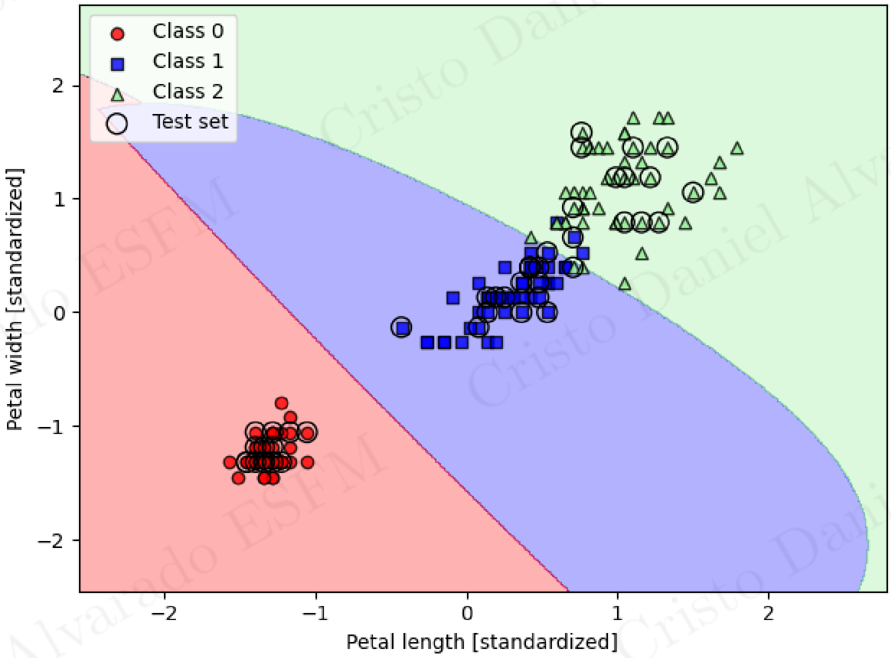


Figure 1.13: Iris SVC using RBF.

```
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')
```

```

#Import Data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

#Scale Standard Data

sc = StandardScaler()
X = sc.fit_transform(X)

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

svm = SVC(kernel='rbf', gamma=1, C=1.0, random_state=1)
svm.fit(X_train, y_train)
plot_decision_regions(X,
                      y,
                      classifier=svm,
                      test_idx=idx_test)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Code 1.15: Decision Boundary on the Iris Dataset using RBF.

If we increase the value of gamma, the following happens:

Observation 1.9.3
Although the model fits the training dataset very well, *such a classifier will likely have a high generalization error on unseen data.* This illustrates that the γ *parameter also plays an important role in controlling overfitting or variance* when the algorithm is too sensitive to fluctuations in the training dataset.

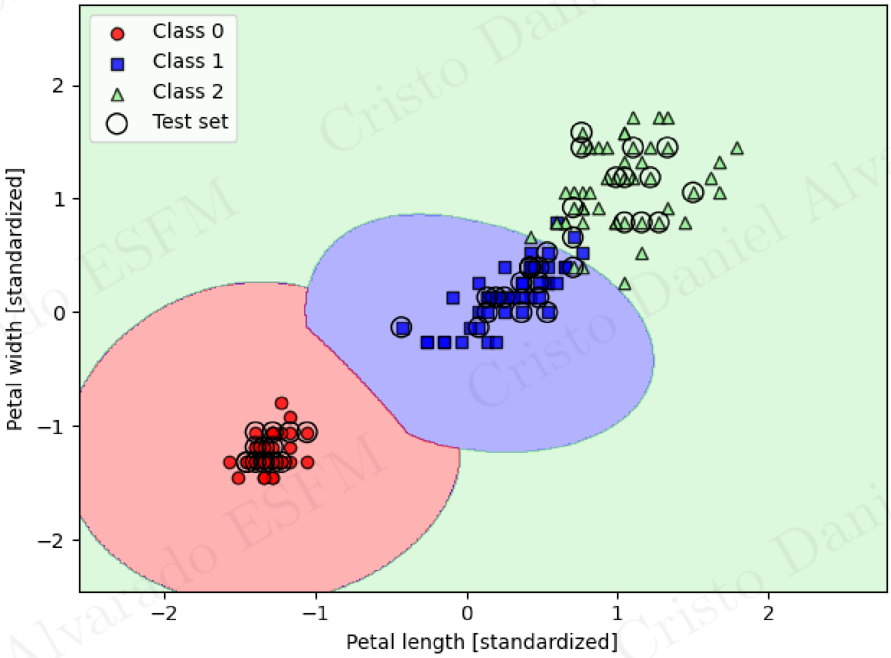


Figure 1.14: Iris Dataset using RBF Kernel SVM with $\gamma = 1$.

1.10 Decision Tree Learning

Definition 1.10.1 (Decision Tree)
Decision tree classifiers are attractive models if we care about interpretability. As the name **decision tree** suggests, we can think of this model as *breaking down our data by making a decision based on asking a series of questions.*

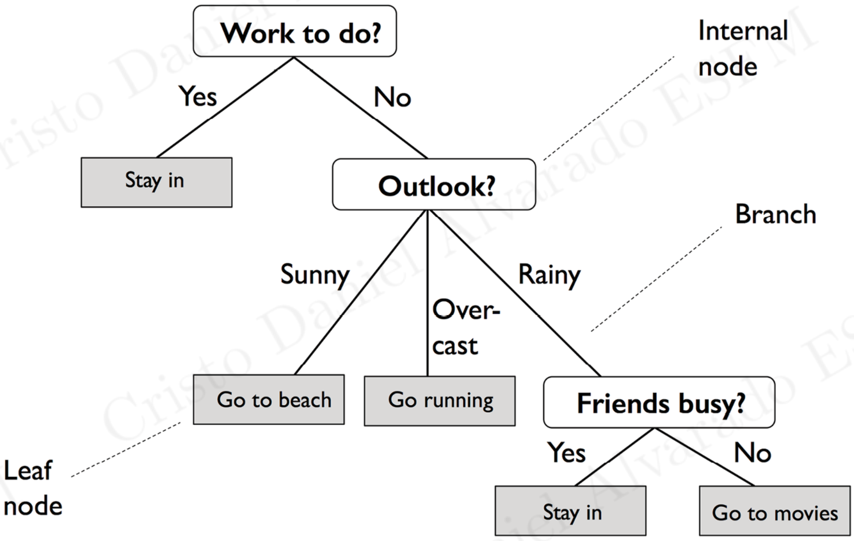


Figure 1.15: Decision Tree

Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples.

Observation 1.10.1
Although the concept of a decision tree based on categorical variables is illustrated, the same concept applies if our features are real numbers, like in the Iris dataset.

For example, we could simply define a cut-off value along the sepal width feature axis and ask a binary question: *Is the sepal width* $\geq$ *2.8 cm?*

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest information gain (IG), which will be explained in more detail in the following section.

In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the training examples at each node all belong to the same class.

Idea 1.10.1
In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to prune the tree by setting a limit for the maximum depth of the tree.

1.10.1 Maximizing IG Getting the Most Bang for Your Buck

To split the nodes at the most informative features, we need to define an objective function to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here:

- f is the feature to perform the split.
- D_p and D_j are the datasets of the parent and j-th child node.
- I is our **impurity measure**.
- N_p is the total number of training examples at the parent node.
- N_j is the number of examples in the j-th node.

The information gain (function IG) is simply the difference between the impurity of the parent node and the sum of the child node impurities — the lower the impurities of the child nodes, the larger the information gain.

Observation 1.10.2
However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes: D_{left} and D_{right}.

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

The three impurity measures (I) or splitting criteria that are commonly used in binary decision trees are:

- **Gini impurity** (I_G).

- **Entropy** (I_H).
- **Classification error** (I_E).

Definition 1.10.2 (Entropy I_H)
The definition of **entropy** for all non-empty classes is given by:

$$\begin{cases} p(i|t) \neq 0 \\ I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \end{cases}$$

Here:

- $p(i|t)$ is the proportion of examples that belong to class i for a particular node t.

Observation 1.10.3
The entropy is therefore zero if all examples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution.

Example 1.10.1 (Objective of the Entropy Criterion)
In a binary class setting, the entropy is 0 is $p(i=0|t) = 1$ or $p(i=0|t) = 0$. If the classes are distributed uniformly with $p(i=0|t) = \frac{1}{2}$ or $p(i=0|t) = \frac{1}{2}$, the entropy is 1. Therefore, we can say that **the entropy criterion attempts to maximize tue mutual information in the tree**.

To provide a visual intuition, let us visualize the entropy values for different class distributions via the following code:

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

def entropy(p):
    return - p * np.log2(p) - (1 - p) * np.log2((1 - p))

x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
plt.ylabel('Entropy')
plt.xlabel('Class-membership probability p(i=1)')
plt.plot(x, ent)
plt.show()
```

Code 1.16: Entropy Computation.

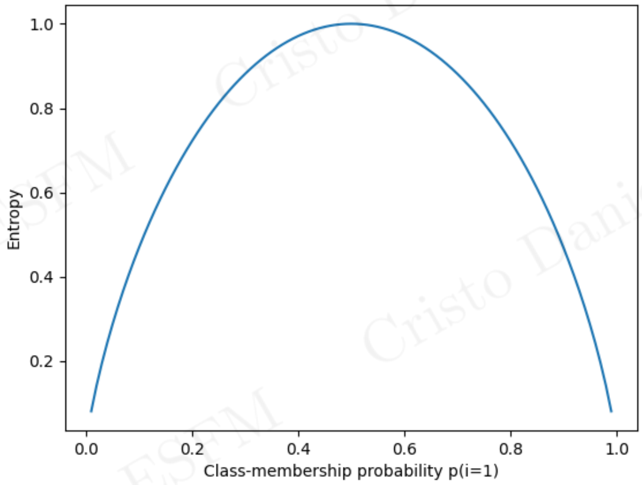


Figure 1.16: Entropy Visualization

Definition 1.10.3 (Gini Impurity)
The Gini impurity is given by:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\,(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Observation 1.10.4
That is, the Gini Impurity can be understood as a **criterion to minimize the probability of misclassification**.

Observation 1.10.5 (**Relation Between Gini Impurity and Entropy**)
Similar to entropy, Gini Impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting (where $c = 2$):

$$I_G(t) = 1 - \sum_{i=1}^{c} \left(\frac{1}{2}\right)^2 = \frac{1}{2}$$

In practice, **both the Gini impurity and entropy typically yield very similar results**, and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs.

Definition 1.10.4 (**Pruning**)
Pruning means *removing parts of a model that are unnecessary or harmful, with the goal of making the model simpler, faster, and better at generalizing.*

Definition 1.10.5 (**Classification Error**)

Another impurity measure is the **classification error**:

$$I_E(t) = 1 - \max\{p(i|t)\}$$

This is a useful criterion for pruning, but not recommended for growing a decision tree, since it **is less sensitive to changes in the class probabilities of the nodes**. We can illustrate this by looking at the two possible splitting scenarios shown below:

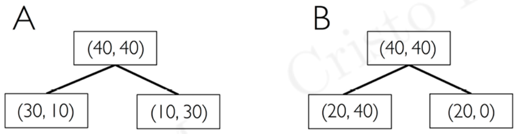


Figure 1.17: Decision Trees A and B.

Example 1.10.2
We start with a dataset D_p, at the parent node, which consists of 40 examples from class 1 and 40 examples from class 2, that we split into two datasets, D_{left} and D_{right}.

- The information gain using the **classification error** as a splitting criterion would be the same (since $IG = 0.25$) in both scenarios, A and B:

$$\begin{cases} I_E(D_p) = 1 - 0.5 = 0.5 \\ A: \ I_E(D_{\text{left}}) = 1 - \frac{3}{4} = 0.25 \\ A: \ I_E(D_{\text{right}}) = 1 - \frac{3}{4} = 0.25 \\ A: \ IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25 \\ B: \ I_E(D_{\text{left}}) = 1 - \frac{4}{6} = \frac{1}{3} \\ B: \ I_E(D_{\text{right}}) = 1 - 1 = 0 \\ B: \ IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25 \end{cases}$$

- However, the **Gini impurity** would favor the split in scenario B ($IG_G \approx 0.1667$) over scenario A ($IGG = 0.125$), which is indeed purer:

$$\begin{cases} I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5 \\ A: \ I_G(D_{\text{left}}) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375 \\ A: \ I_G(D_{\text{right}}) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375 \\ A: \ IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125 \\ B: \ I_G(D_{\text{left}}) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\overline{4} \\ B: \ I_G(D_{\text{right}}) = 1 - (1^2 + 0^2) = 0 \\ B: \ IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.1\overline{6} \end{cases}$$

- Similarly, the **entropy criterion** would also favor scenario B ($IG_H = 0.31$) over scenario

A $(IG_H = 0.19)$:

$$\begin{cases} I_H(D_p) = -\left(0.5\log_2(0.5) + 0.5\log_2(0.5)\right) = 1 \\ A: \ I_H(D_{\text{left}}) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.81 \\ A: \ I_H(D_{\text{right}}) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.81 \\ A: \ IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19 \\ B: \ I_H(D_{\text{left}}) = -\left(\frac{2}{6}\log_2\left(\frac{2}{6}\right) + \frac{4}{6}\log_2\left(\frac{4}{6}\right)\right) = 0.92 \\ B: \ I_H(D_{\text{right}}) = 0 \\ B: \ IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31 \end{cases}$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range $[0, 1]$ for class 1. Note that we will also add a scaled version of the entropy (entropy / 2) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

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

def gini(p):
    return p*(1 - p) + (1 - p)*(1 - (1-p))

def entropy(p):
    return - p*np.log2(p) - (1 - p)*np.log2((1 - p))

def error(p):
    return 1 - np.max([p, 1 - p])

x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e*0.5 if e else None for e in ent]
err = [error(i) for i in x]
fig = plt.figure()
ax = plt.subplot(111)
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
                          ['Entropy', 'Entropy (scaled)',
                           'Gini impurity',
                           'Misclassification error'],
                          ['-', '-', '--', '-.'],
                          ['black', 'lightgray',
                           'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab,
                  linestyle=ls, lw=2, color=c)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
          ncol=5, fancybox=True, shadow=False)
ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('impurity index')
plt.show()
```

Code 1.17: Entropy Computation.

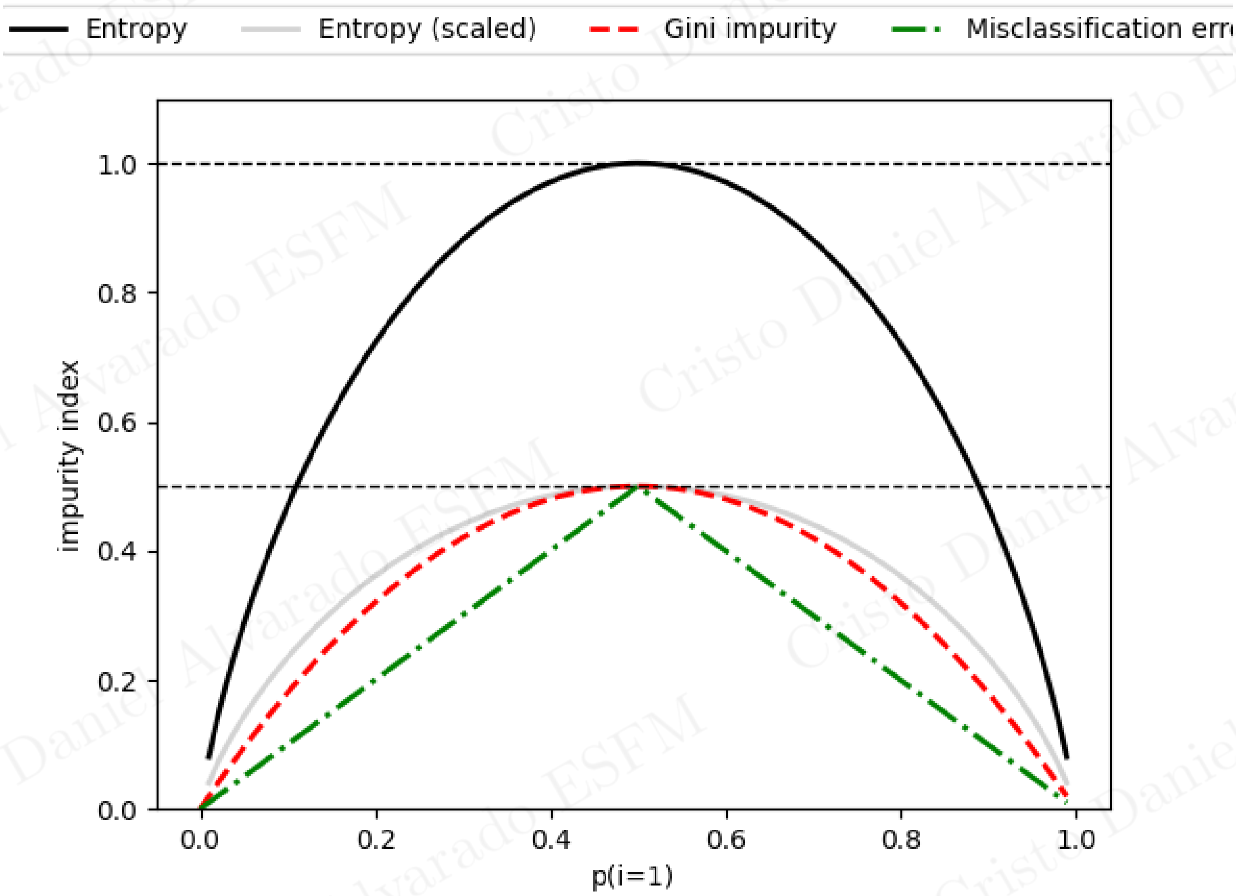


Figure 1.18: Caption

Idea 1.10.2
This example illustrates that, although two splits may appear equivalent under a coarse criterion such as classification error, more sensitive criteria like Gini impurity and entropy better capture the true purity of the resulting nodes, favoring splits that produce highly homogeneous subsets. In summary:

Key Point	Criterion	Sensitivity	Interpretation
(1) Informative power	Classification Error	Low	Insensitive to changes in class distribution; only considers the majority class and may assign the same information gain to qualitatively different splits.
(1) Informative power	Gini Impurity	Medium	Sensitive to class proportions and able to distinguish between moderately and highly pure nodes.
(1) Informative power	Entropy	High	Highly sensitive to uncertainty; strongly rewards splits that produce nodes with high class concentration.
(2) Preference for pure nodes	Gini / Entropy	High	Both criteria favor splits that generate highly pure (or perfectly pure) child nodes, even if the remaining child node becomes less pure.
(3) Relative position	Classification Error	Very Low	Nearly flat behavior as a function of class probability; provides a coarse measure of impurity.
(3) Relative position	Gini Impurity	Medium	Acts as an intermediate measure between entropy and classification error, balancing sensitivity and stability.
(3) Relative position	Entropy	Very High	Exhibits strong curvature near probabilities close to 0 or 1, heavily penalizing uncertainty and strongly favoring purity.

1.11 Building a Decision Tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles.

Observation 1.11.1
However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting.

Using scikit-learn, we will now train a decision tree with a maximum depth of 4, using the Gini impurity as a criterion for impurity.

Although *feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms.* The code is as follows:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn import datasets

```

```python
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.2):
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')

#Gather data

iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

#Train model

tree_model = DecisionTreeClassifier(criterion='gini',
                                     max_depth=4,
                                     random_state=1)
tree_model.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined,
                      y_combined,
                      classifier=tree_model,
                      test_idx=idx_test)
plt.xlabel('Petal length [cm]')
plt.ylabel('Petal width [cm]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Code 1.18: Code for Decision Tree Implementation.

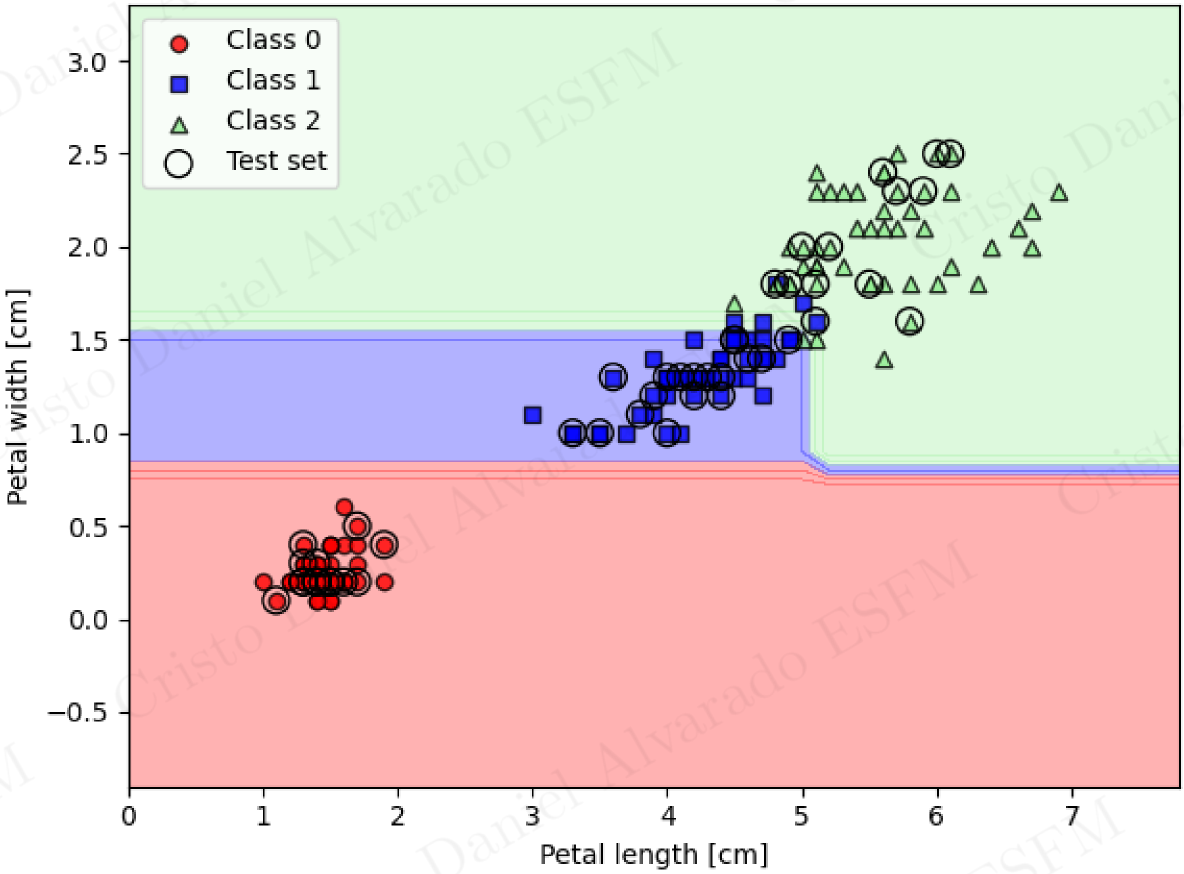


Figure 1.19: Decision Tree Boundary Visualized.

Observation 1.11.2

A nice feature in scikit-learn is that it allows us to readily visualize the decision tree model after training via the following code:

```
from sklearn import tree
feature_names = ['Sepal length', 'Sepal width',
                 'Petal length', 'Petal width']
```

```
tree.plot_tree(tree_model,
               feature_names=feature_names,
               filled=True)
plt.show()
```

Code 1.19: caption

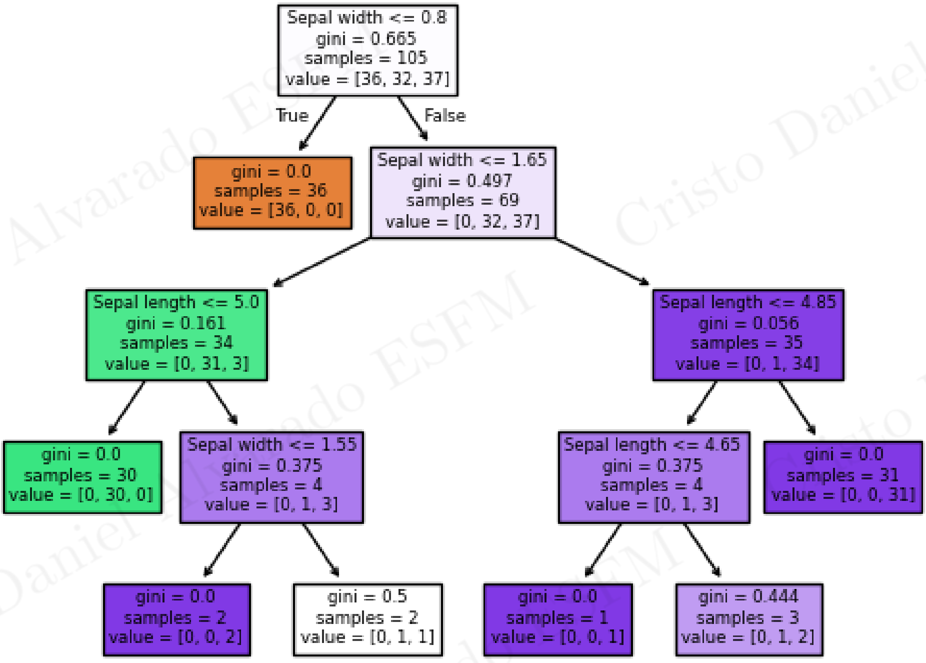


Figure 1.20: Tree Visualized.

Setting `filled=True` in the `plot_tree` function we called colors the nodes by the majority class label at that node.

Observation 1.11.3
There are many additional options available, which you can find in the documentation at Plot Tree.

Looking at the root node, it starts with `105` examples at the top. The first split uses a sepal width $cut-off \leq 0.75$ cm for splitting the root node into two child nodes with 35 examples (left child node) and `70` examples (right child node). After the first split, we can see that the left child node is already pure and only contains examples from the Iris-setosa class (`Gini impurity = 0`). The further splits on the right are then used to separate the examples from the Iris-versicolor and Iris-virginica class.

Idea 1.11.1
The decision tree does a very good job of separating the flower classes. Unfortunately, scikit-learn currently does not implement functionality to manually post-prune a decision tree. However, we

could go back to our previous code example, change the `max_depth` of our decision tree to 3, and compare it to our current model, but we leave this as an exercise for the interested reader.

Observation 1.11.4 (Cost Complexity Post-Pruning)
Alternatively, scikit-learn provides an automatic cost complexity post-pruning procedure for decision trees. Interested readers can find more information about this more advanced topic in the following tutorial: Plot Cost Complexity Pruning.

1.11.1 Combining Multiple Decision Trees via Random Forests

Ensemble methods have gained huge popularity in applications of machine learning during the last decade due to their good classification performance and robustness toward overfitting.

Let's discuss the decision tree-based random forest algorithm, which is known for its good scalability and ease of use.

Definition 1.11.1 (Random Forest)
A **random forest** can be considered as *an ensemble of decision trees.*

The idea behind a random forest is to *average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting.*

The random forest algorithm can be summarized in four simple steps:

1) Draw a random bootstrap sample of size n (randomly choose n examples from the training dataset with replacement).

2) Grow a decision tree from the bootstrap sample. At each node:

- Randomly select d features without replacement.
- Split the node using the feature that provides the best split according to the objective function, for instance, maximizing the information gain.

3) Repeat steps *1)-2)* k times.

4) Aggregate the prediction by each tree to assign the class label by majority vote.

Observation 1.11.5
We should note one slight modification in step *2)* when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.

1.12 K-Nearest Neighbours

The last supervised learning algorithm that we want to discuss in this section is the k-nearest neighbor (KNN) classifier, which is particularly interesting because it is fundamentally different from the learning algorithms discussed so far.

Observation 1.12.1
KNN is a typical example of a lazy learner. It is called *lazy* not because of its apparent simplicity, but because *it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.*

KNN belongs to a subcategory of non-parametric models described as instance-based learning. *Models based on instance-based learning memorize the training dataset, and lazy learning is a special case of instance-based learning that incurs no cost during the learning process.*

- **Parametric** models estimate parameters.
- **Non-parametric** models adapt to data.

The KNN algorithm itself is straightforward and can be summarized by the following steps:

1) Choose the number of k and a distance metric

2) Find the k-nearest neighbors of the data record to classify

3) Assign the class label by majority vote

Based on the chosen distance metric, the KNN algorithm finds the k examples in the training dataset that are closest (most similar) to the point that we want to classify.

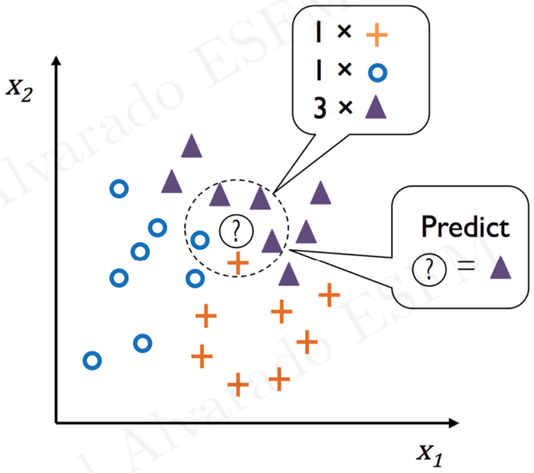


Figure 1.21: k-Nearest Neighbours Algorithm

Observation 1.12.2
The class label of the data point is then determined by a majority vote among its k nearest neighbors.

1.12.1 Advantages and disadvantages of memory-based approaches

- **Advantage** of a memory-based approach is that the classifier immediately adapts as new training data are collected.
- **Downside** is that the computational complexity for classifying new examples grows linearly with the number of examples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions and the algorithm uses efficient data structures for querying the training data.

Such data structures include $k-d$ tree and ball tree, which are both supported in scikit-learn.

In addition to computational costs, large datasets can be problematic in terms of storage requirements.

Idea 1.12.1
For relatively small to medium-sized datasets, memory-based methods can provide good predictive and computational performance and are therefore suitable for many real-world problems.

By executing the following code, we implement a KNN model in scikit-learn using a Euclidean distance metric:

```
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn import datasets

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.2):
    markers = (’o’, ’s’, ’ˆ’, ’v’, ’<’)
    colors = (’red’, ’blue’, ’lightgreen’, ’gray’, ’cyan’)
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f’Class {cl}’,
                    edgecolor=’black’)

    if test_idx is not None:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c=’none’, edgecolor=’black’, alpha=1.0,
                    linewidth=1, marker=’o’,
                    s=100, label=’Test set’)

#Gather data

```

```python
iris = datasets.load_iris()
X = iris.data[:,2:4]
y = iris.target

# Split training data and test data
indices = np.arange(len(X))
X_train, X_test, idx_train, idx_test = train_test_split(
    X, indices, test_size=0.3, random_state=1
)
y_train = y[idx_train]
y_test  = y[idx_test]

#Train model

knn = KNeighborsClassifier(n_neighbors=5, p=2,
                           metric='minkowski')
knn.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined,
                      y_combined,
                      classifier=knn,
                      test_idx=idx_test)
plt.xlabel('Petal length [standardized]')
plt.ylabel('Petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

Code 1.20: Code for k-Nearest Neighbours.

By specifying five neighbors, we obtain a relatively smooth decision boundary:

Observation 1.12.3
In the case of a tie, scikit-learn prefers neighbors with a closer distance. If the neighbors have identical distances, the algorithm chooses the class label that appears first in the training dataset.

Idea 1.12.2
Choosing k is crucial to finding a good balance between overfitting and underfitting. A suitable distance metric is also essential. A simple Euclidean distance measure is often used for real-valued features, such as the centimeters-based measurements in the Iris dataset.

When using Euclidean distance, standardizing the data is important so that each feature contributes equally to the distance. The minkowski distance used in the preceding code is a generalization of the Euclidean and Manhattan distances and can be written as follows:

$$d_p(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}, p \in \mathbb{N}$$

It becomes the Euclidean distance when `p=2` and the Manhattan distance when `p=1`. Many other distance metrics are available in scikit-learn and can be provided to the metric parameter.

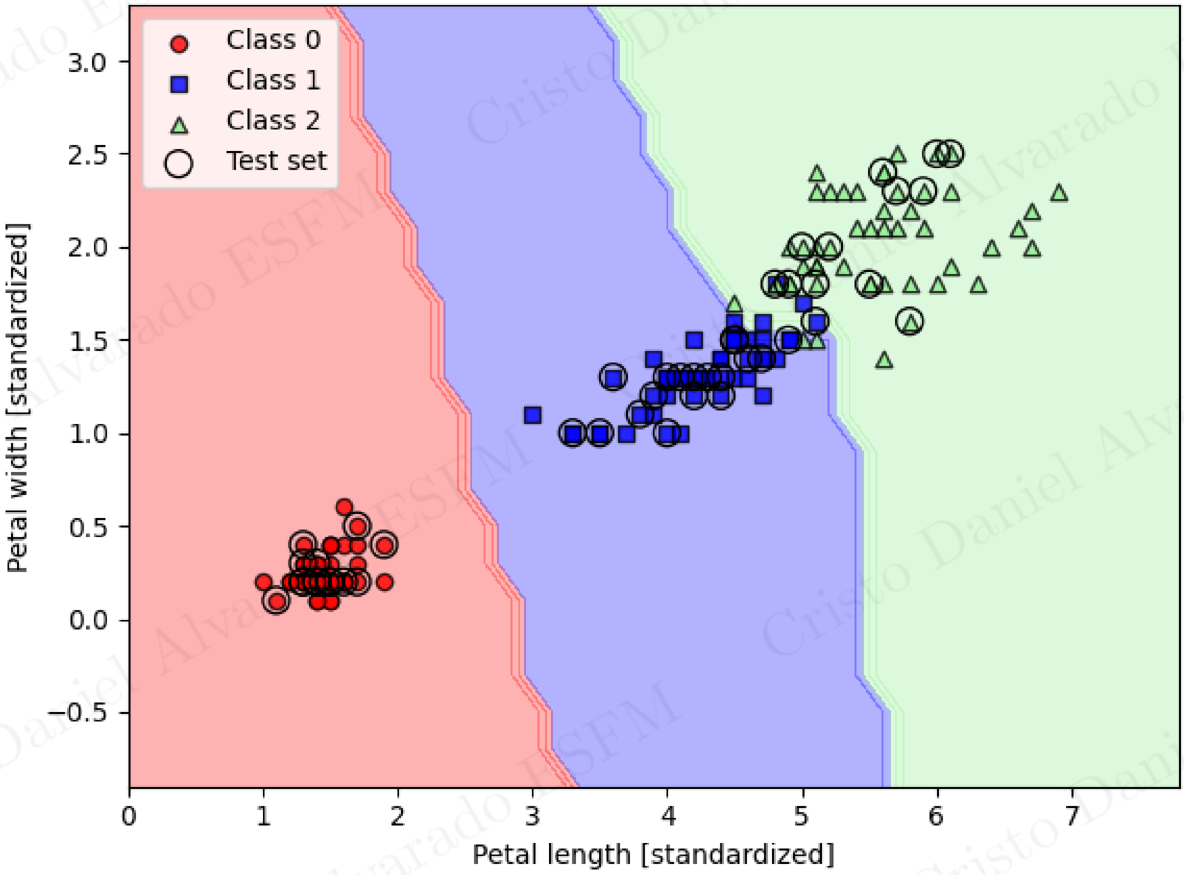


Figure 1.22: Caption

Observation 1.12.4
KNN is susceptible to overfitting due to the curse of dimensionality, where the feature space becomes increasingly sparse as dimensionality grows. Even the closest neighbors can be too far away in a high-dimensional space to provide a good estimate.

Regularization helps avoid overfitting in models such as logistic regression, but it is not applicable to decision trees and KNN. Instead, feature selection and dimensionality reduction techniques help mitigate the curse of dimensionality.

Observation 1.12.5
Note: Use RAPIDS' cuML for GPU-accelerated machine learning with large datasets.