**Lab 01 Python环境配置与机器学习基础实验指导**

**一、实验内容：**

1. **Python环境配置**

**1.1 vscode安装**

**1.2 anaconda安装**

**1.3虚拟环境搭建**

1. **机器学习基础**
   1. **Python科学计算包**
   2. **感知器算法**

**二、实验步骤：**

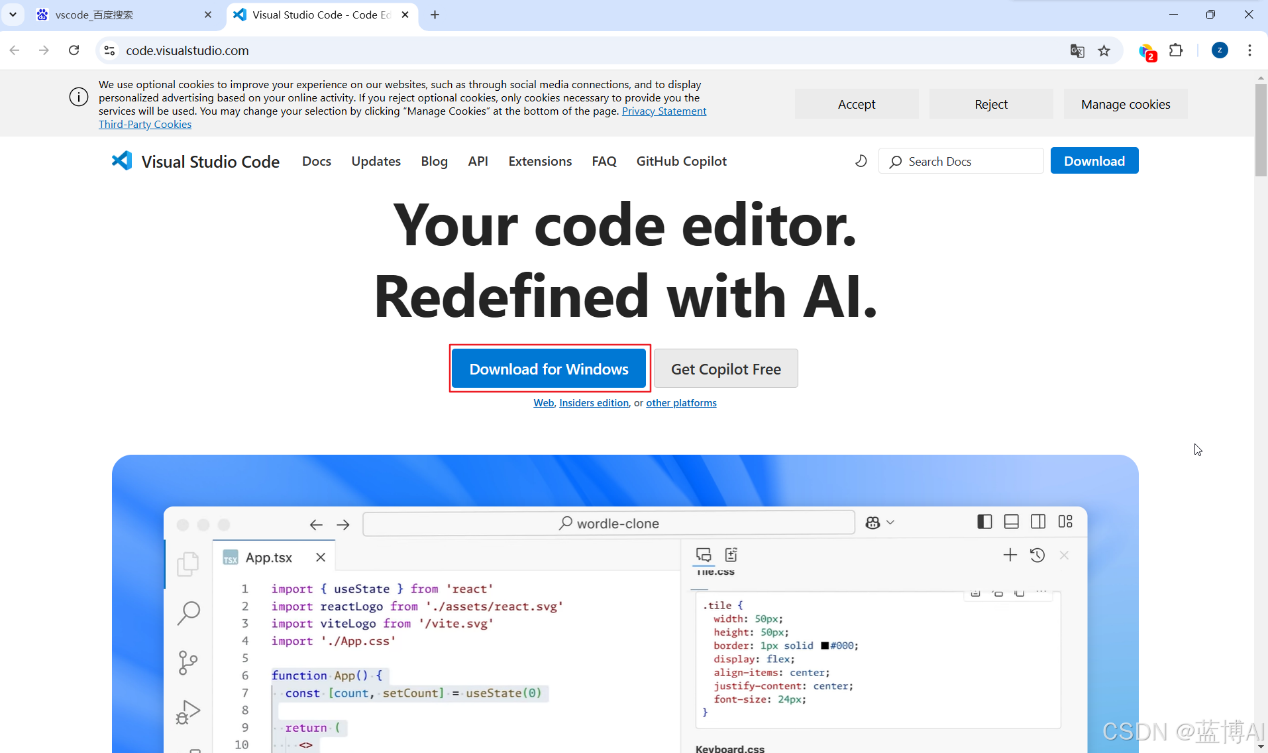
**1. Python环境配置**

**1.1 vscode安装**

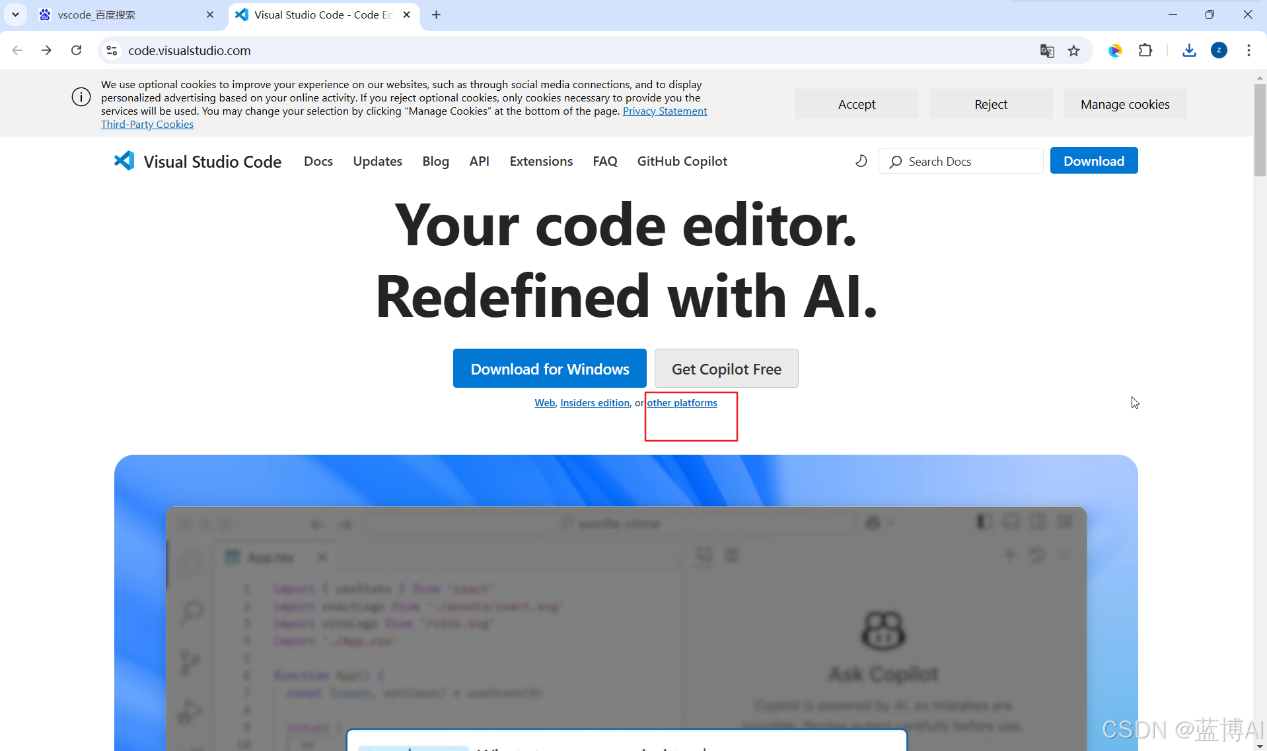
**1.1.1 下载**

从官网下载  [Visual Studio Code - Code Editing. Redefined](https://code.visualstudio.com/)

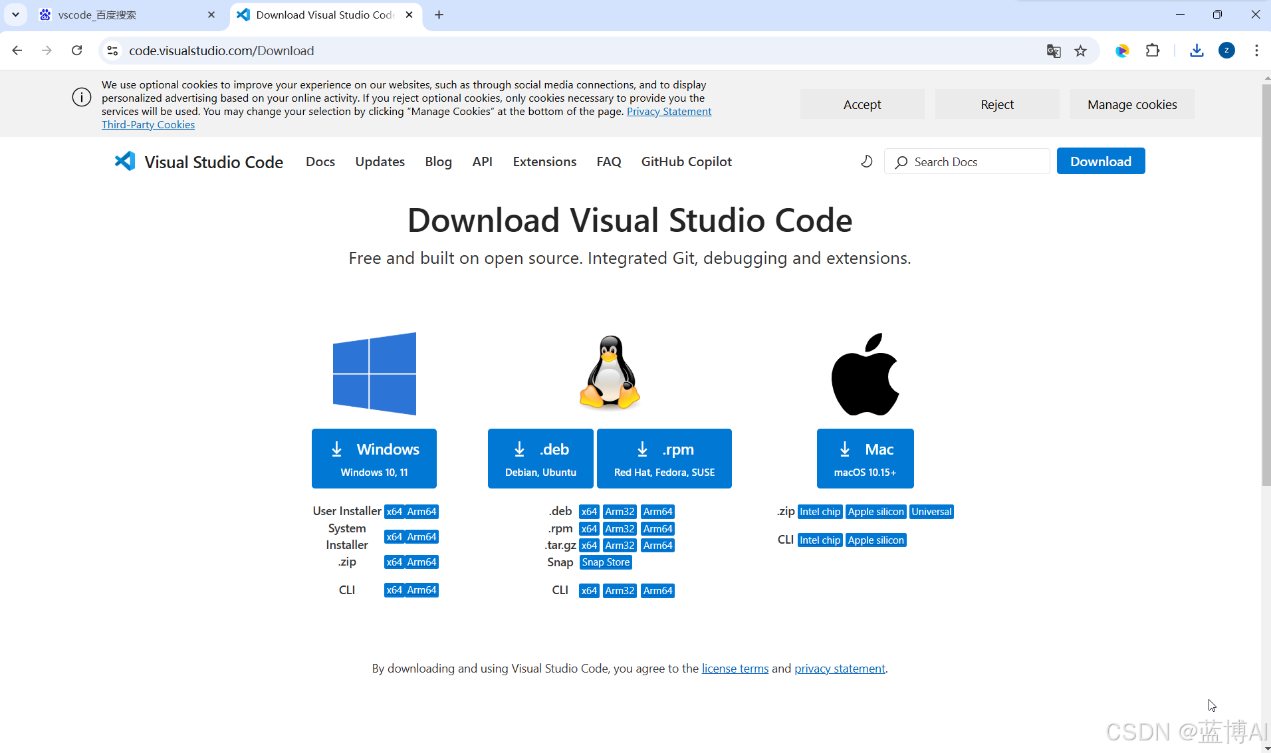
如果是windows 系统，则直接点击下面的 Download for Windows



如果是其他系统的，则点击下面的 other platforms



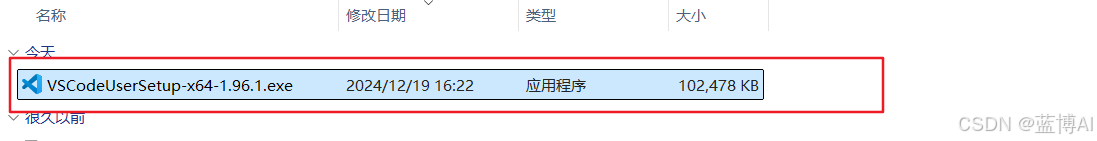
接着选择对应的系统的vscode

​

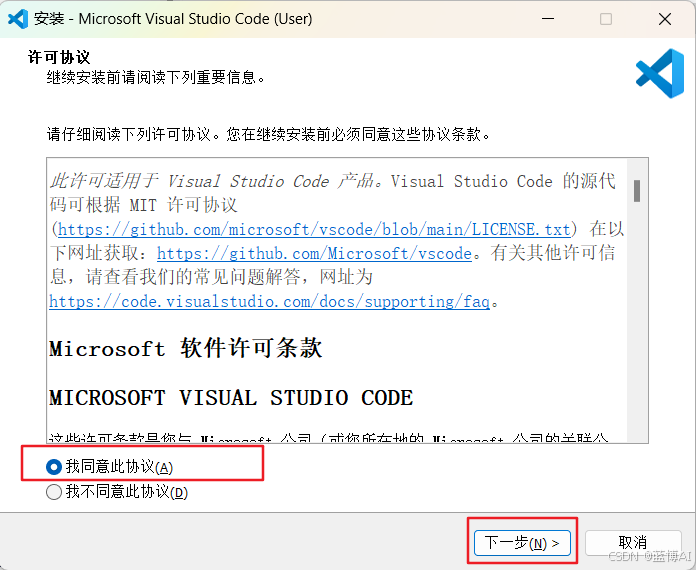
**1.1.2 安装**

系统是windows的，下面演示的是[windows系统](https://so.csdn.net/so/search?q=windows%E7%B3%BB%E7%BB%9F&spm=1001.2101.3001.7020)的vscode的安装

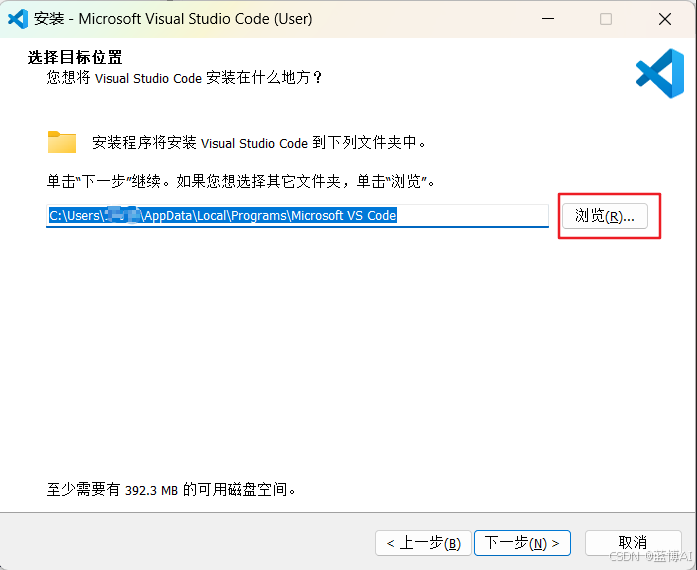
（1）双击下载好的[vscode安装包](https://so.csdn.net/so/search?q=vscode%E5%AE%89%E8%A3%85%E5%8C%85&spm=1001.2101.3001.7020)

​

（2） 选择--我同意此协议A，然后点击--下一步

​

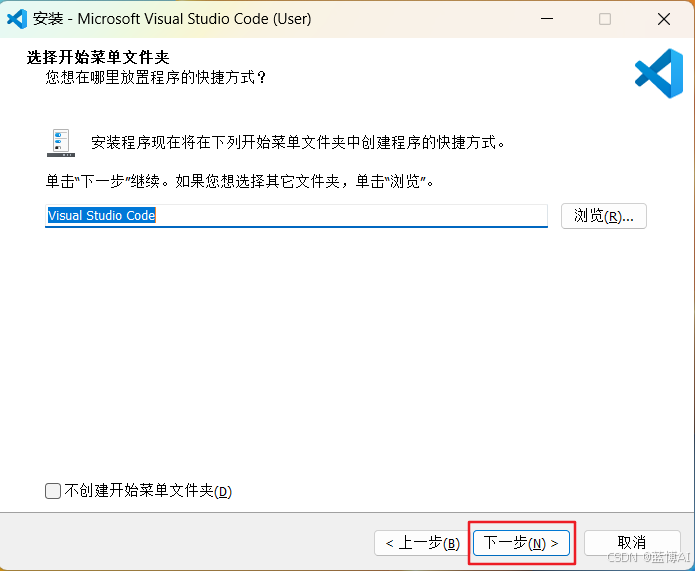
（3）点击 浏览，选择安装的位置

​

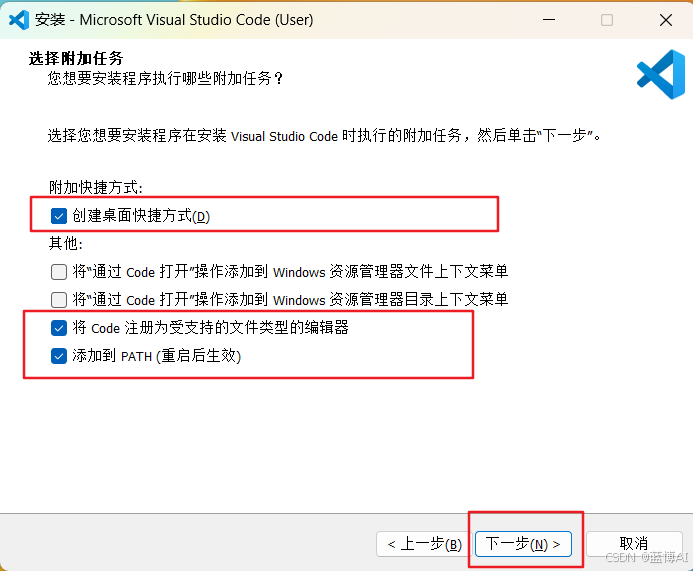
（4）点击 下一步

​

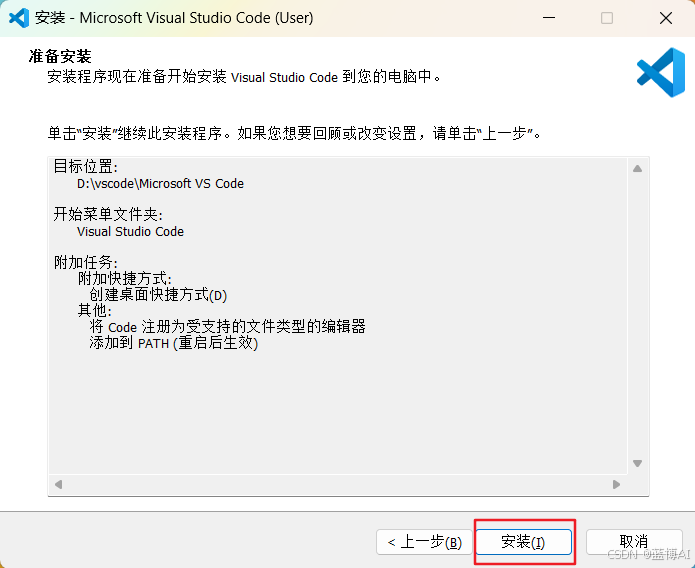
（5）点击 下一步

​

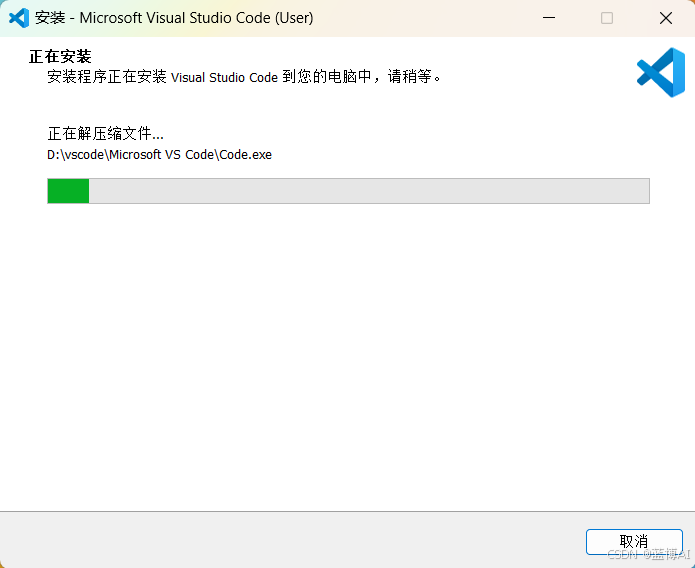
 （6）勾选下面的选项，然后点击 下一步

​

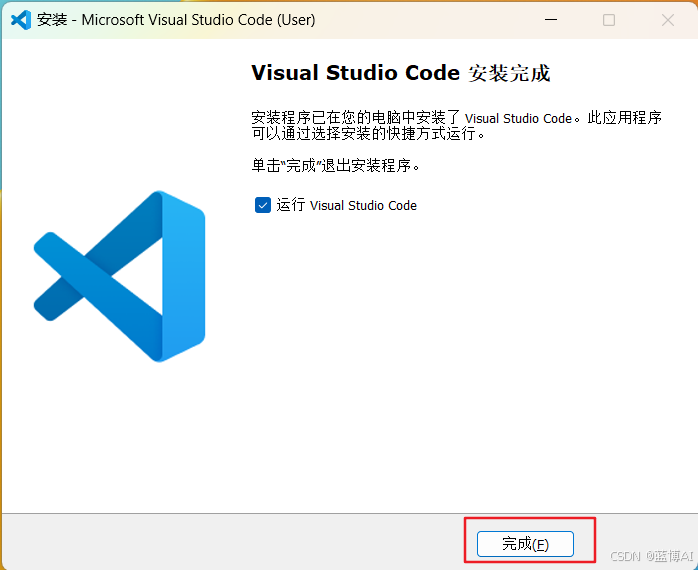
（7）点击 安装

​

 （8）正在安装，等待一会儿...

​

（9）点击 完成，此时vscode就会自动打开

​

**1.2 anaconda安装**

**1.2.1下载**

我的系统为windows系统，这里选择 Anaconda3-2021.11-Windows-x86\_64.exe ，也可以选择其他的版本

[Index of /anaconda/archive/ | 清华大学开源软件镜像站 | Tsinghua Open Source MirrorIndex of /anaconda/archive/ | 清华大学开源软件镜像站，致力于为国内和校内用户提供高质量的开源软件镜像、Linux 镜像源服务，帮助用户更方便地获取开源软件。本镜像站由清华大学 TUNA 协会负责运行维护。https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/](https://mirrors.tuna.tsinghua.edu.cn/anaconda/archive/)



**1.2.2 安装**

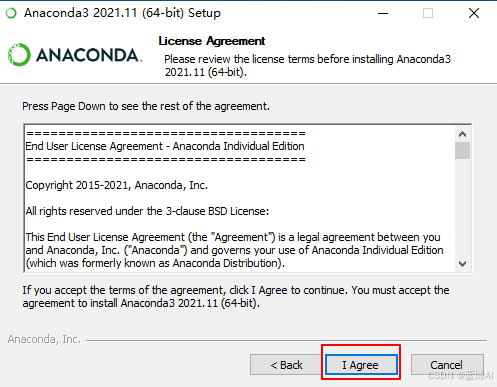
（1）双击下载好的安装包



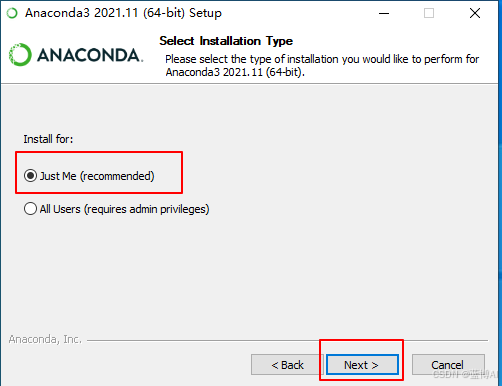
 （2）点击 Next



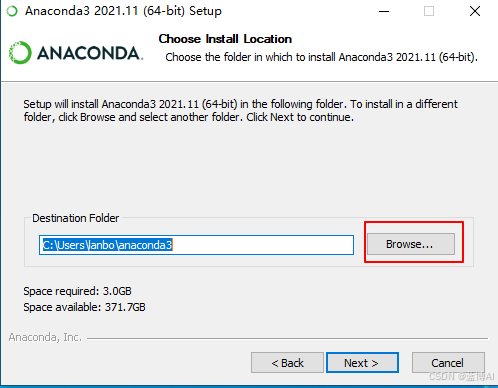
（3）点击 I Agree



（4）勾选下面的选项，然后点击 Next

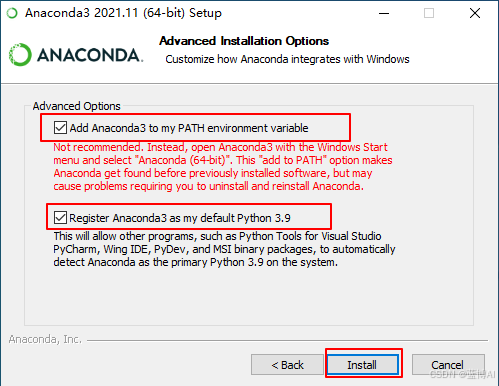


（5）点击 Browse，选择安装的位置，我安装在 D:\anaconda\ 目录下，然后点击 Next

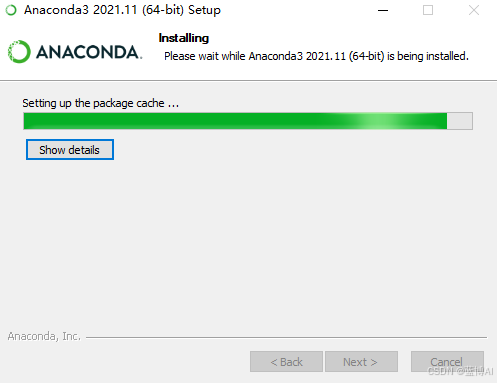




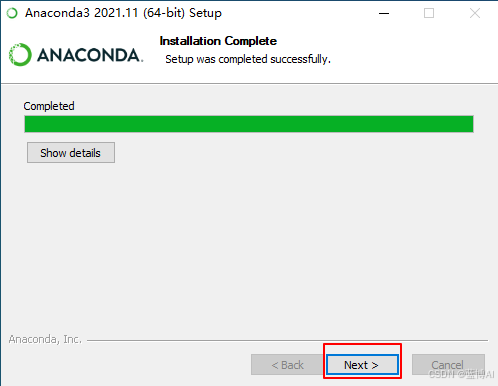
（6）勾选下面的选项，然后点击 Install



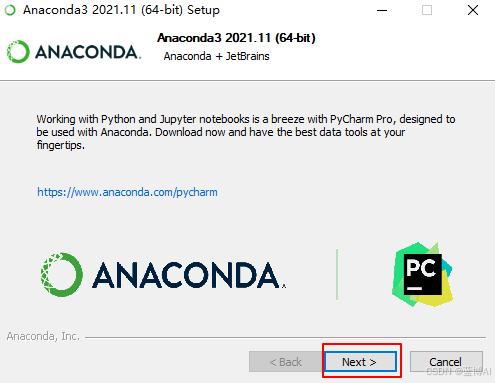
（7）安装进行中...



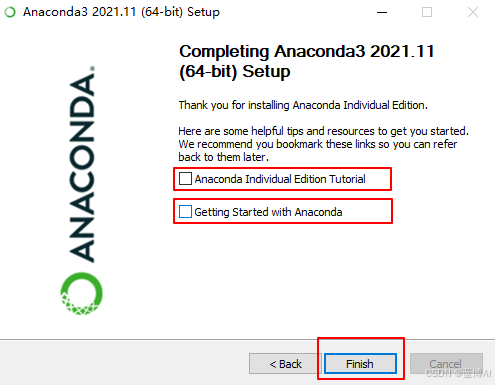
（8）点击 Next



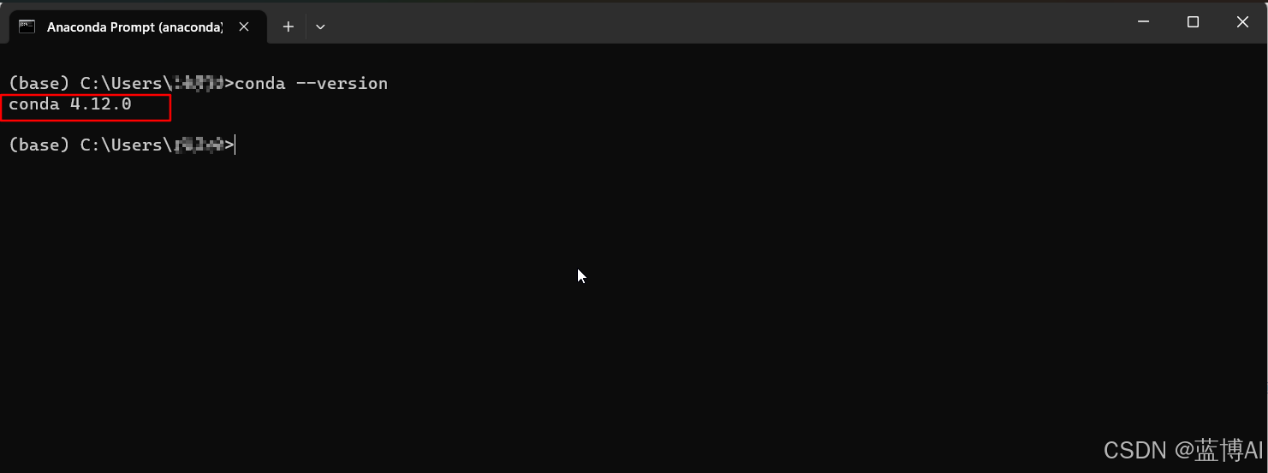
（9）点击 Next



（10）取消勾选，然后点击 Finish



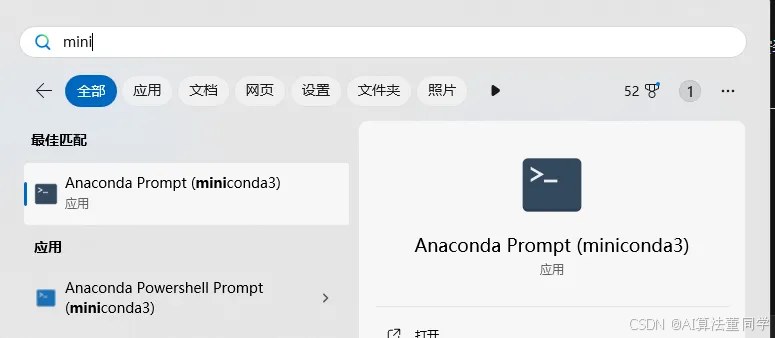
（11）打开终端（cmd），输入conda --version，按回车健，如果出现conda的版本号，则证明anaconda 安装成功



**1.3虚拟环境搭建**

**1.3.1 创建虚拟环境**

**Windows系统在开始菜单搜索 Anaconda Prompt 并打开：**

**  
打开 Anaconda Prompt，创建或激活你想要使用的 conda 环境。命令行输入：**

**# 创建新的 conda 环境，其中 py3.9 是自定义的虚拟环境名称**

|  |  |
| --- | --- |
| 1 | conda create --name py3.9 python=3.9 -y |

**# 激活环境**

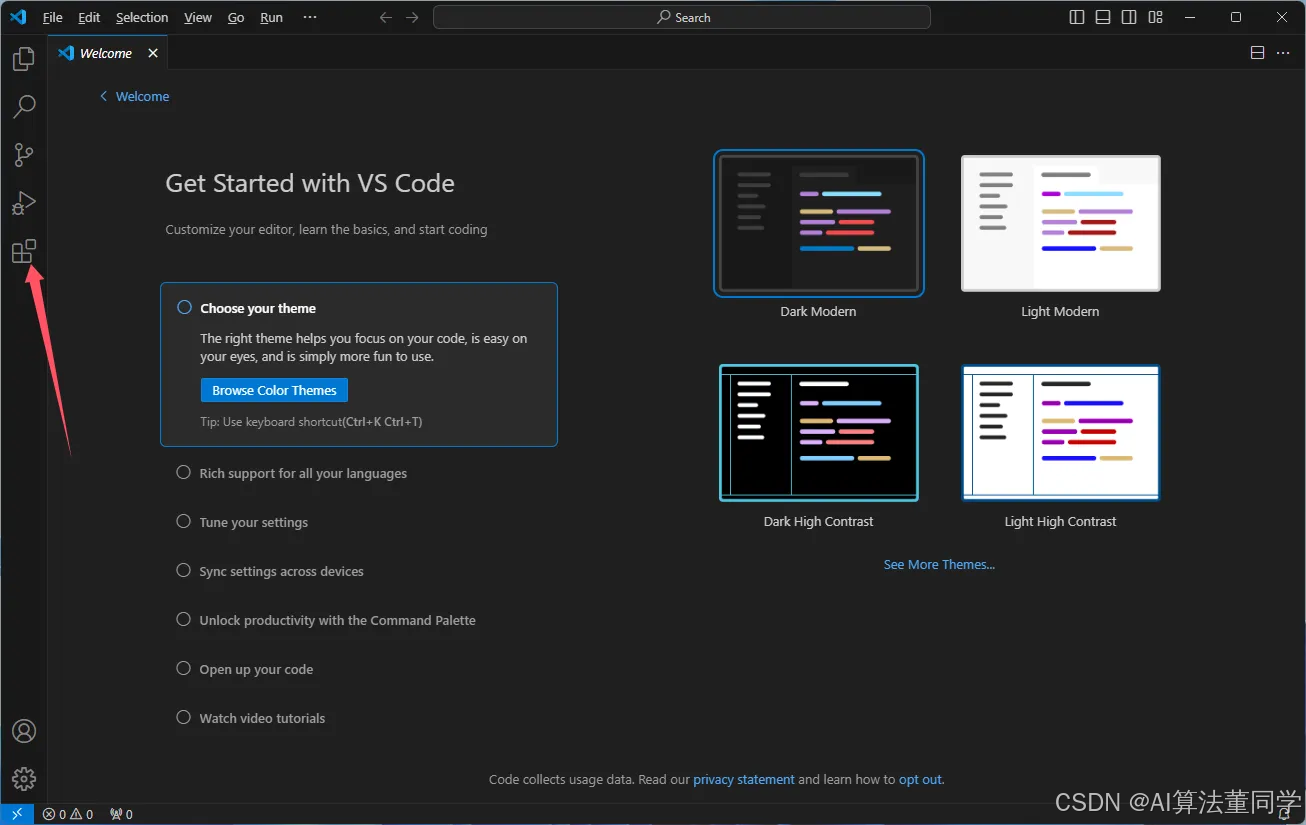
|  |  |
| --- | --- |
| 1 | **conda activate py3.9** |

**记住自定义的虚拟环境名称 py3.9 。**

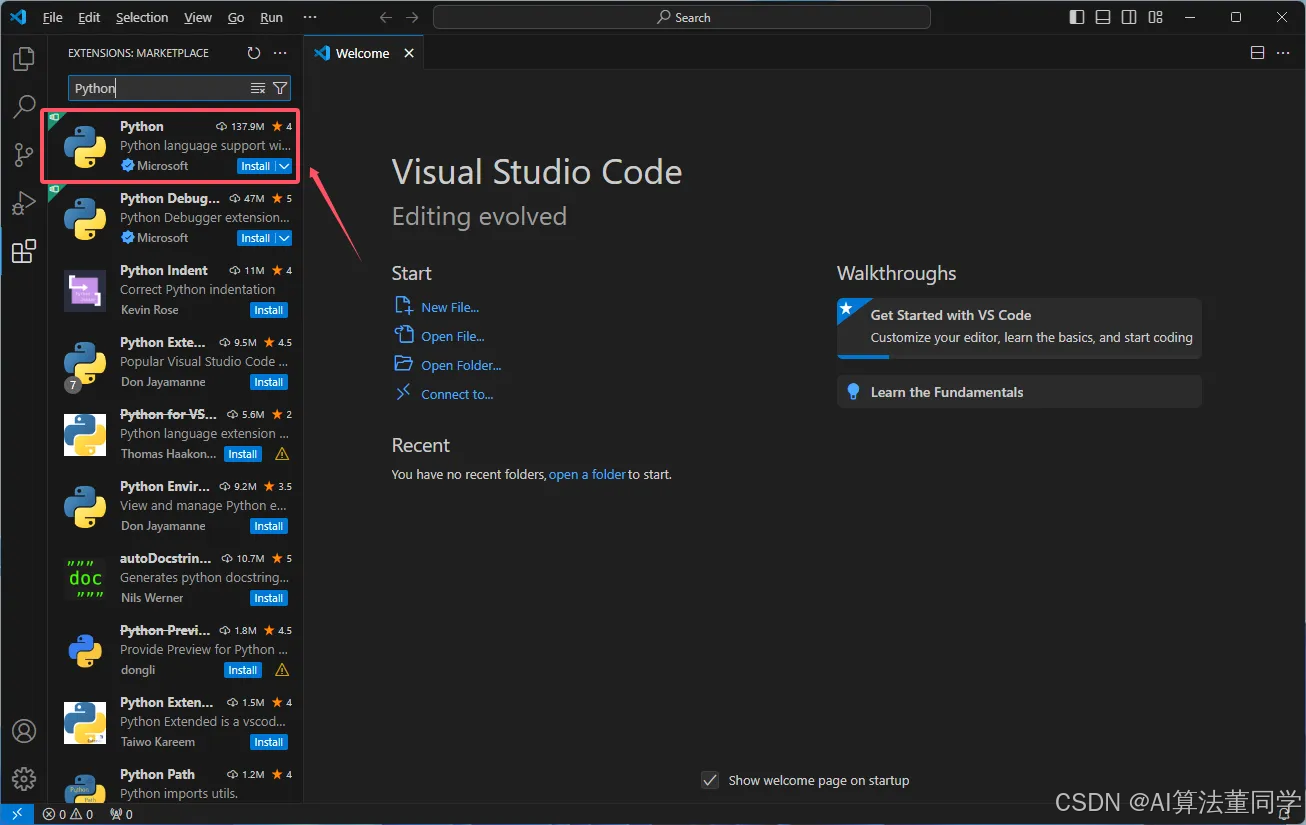
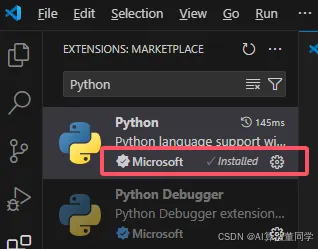
**1.3.2 VS Code 安装 Python 扩展**

**打开 VS Code，安装 Microsoft 提供的官方 Python 扩展：**

1. **打开 VS Code**
2. **点击左侧活动栏中的扩展图标（四个小方块的图标）**

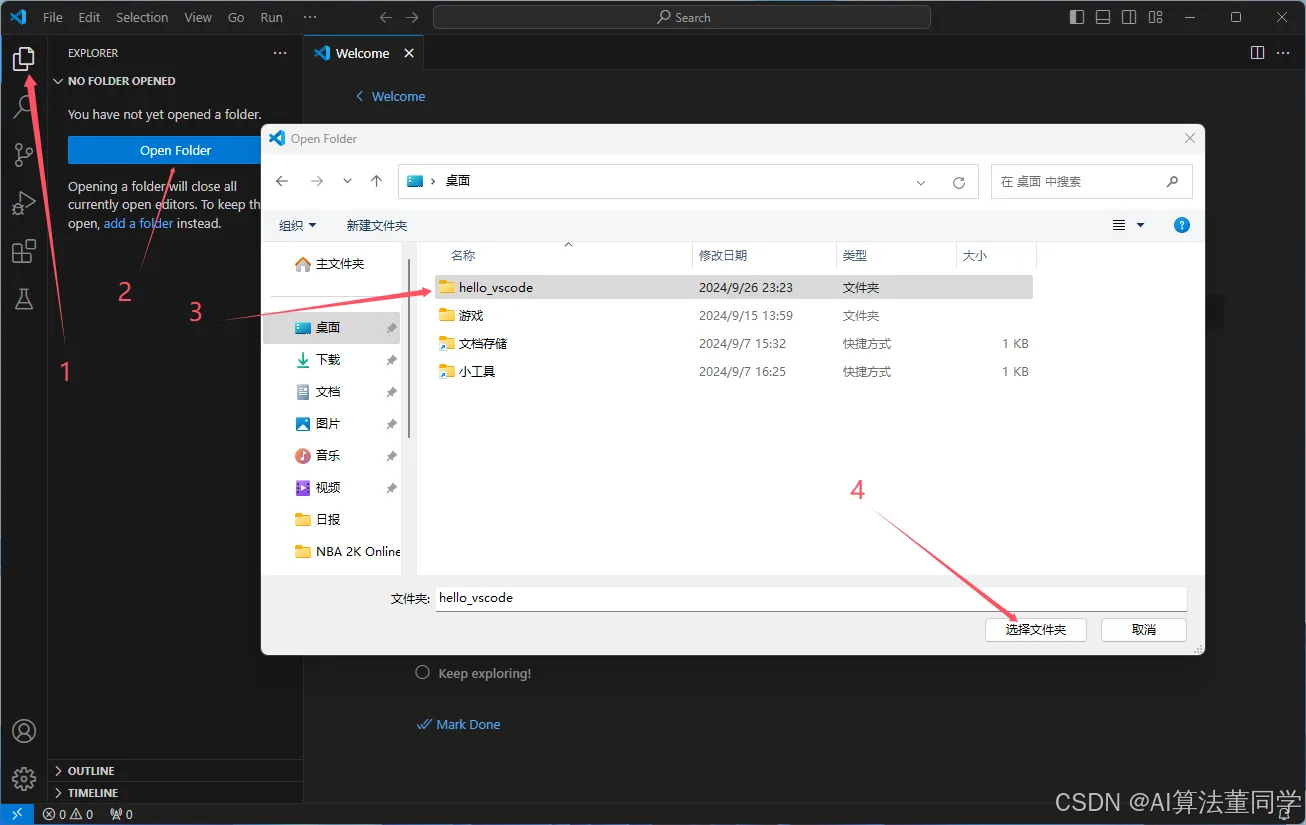
****

1. **在搜索栏中输入 Python**
2. **找到由 Microsoft 提供的 Python 扩展，并点击 Install 进行安装**

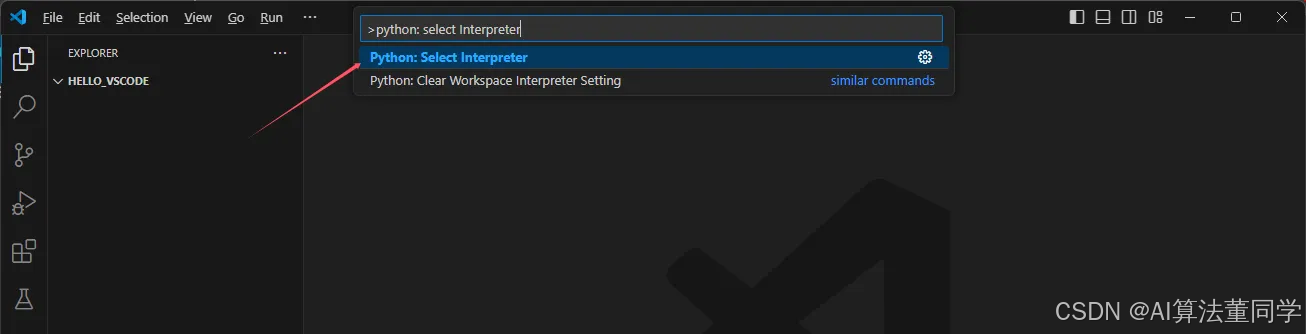
**  
等待安装结束，如下：  
**

* + 1. **配置 VS Code 使用 Anaconda 环境**

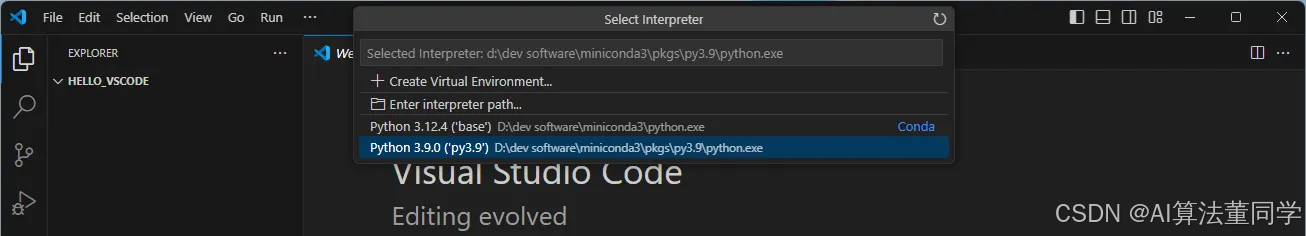
1. **打开 VS Code，并打开你要开发的工作区或文件夹，比如桌面新建一个 hello\_vscode 文件夹**



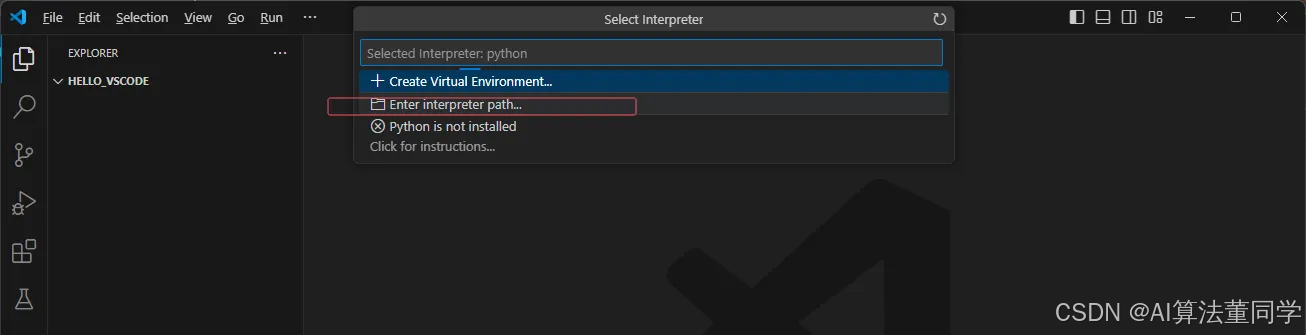
1. **按快捷键 Ctrl+Shift+P 打开命令面板，输入并选择 Python: Select Interpreter**



1. **在弹出的列表中，选择你的 Anaconda 环境（例如 py3.9）**

 **（如果第3步可以选择已创建的虚拟环境，则可以跳过4、5、6步骤。）**

1. **如果没有看到虚拟环境，点击 Enter interpreter path 并浏览到 Anaconda 环境中的 Python 可执行文件路径，通常是：**
   1. **Windows系统: C:\Users\用户\Anaconda3\envs\py3.9\python.exe**
   2. **MacOS/Linux: /Users/用户/anaconda3/envs/py3.9/bin/python**
2. **如果安装的是 miniconda，虚拟环境的 python 路径可能不同：**
3. **点击 Enter interpreter path 并浏览到 Anaconda 环境中的 Python 可执行文件路径：**

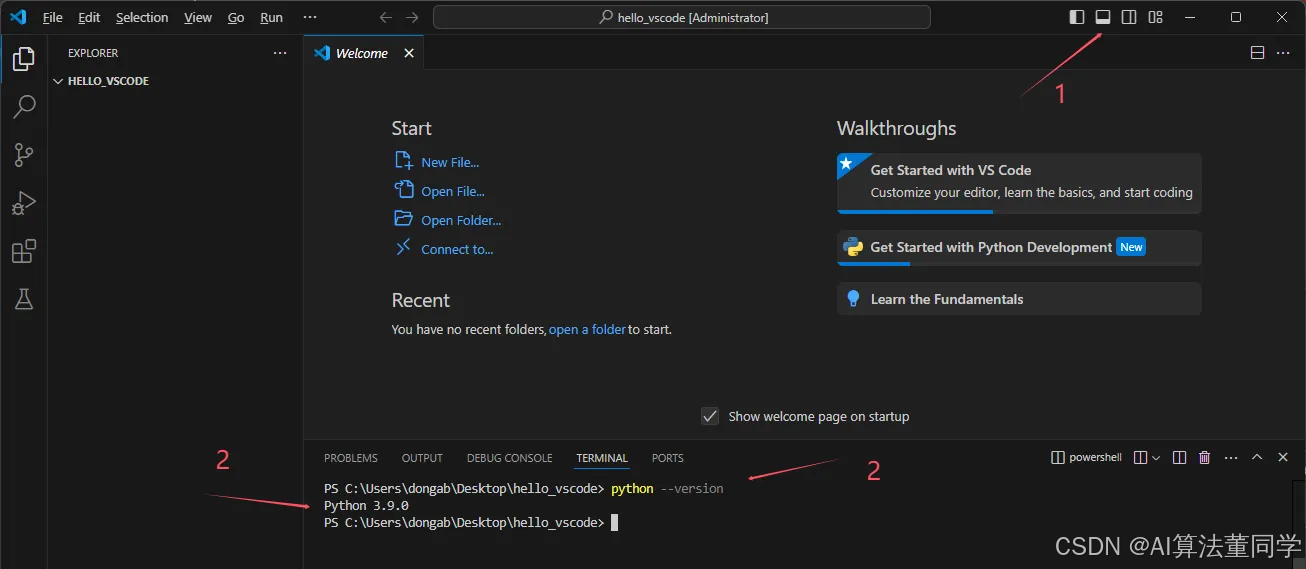


1. **选中 Python 可执行文件**

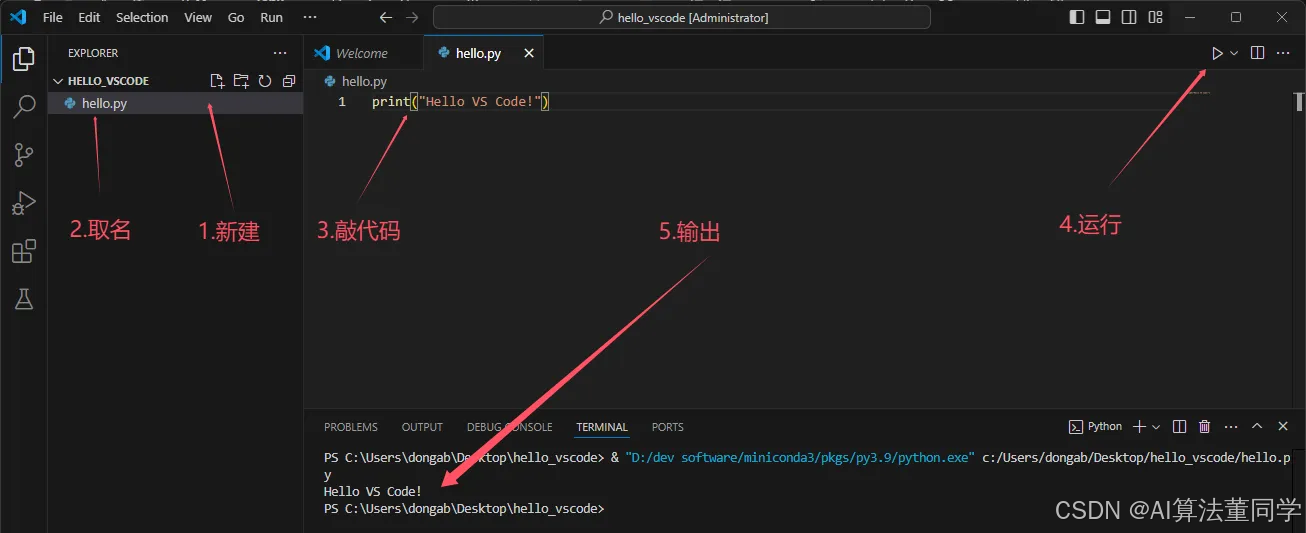
 **确认后就配置好了想要的虚拟环境。**

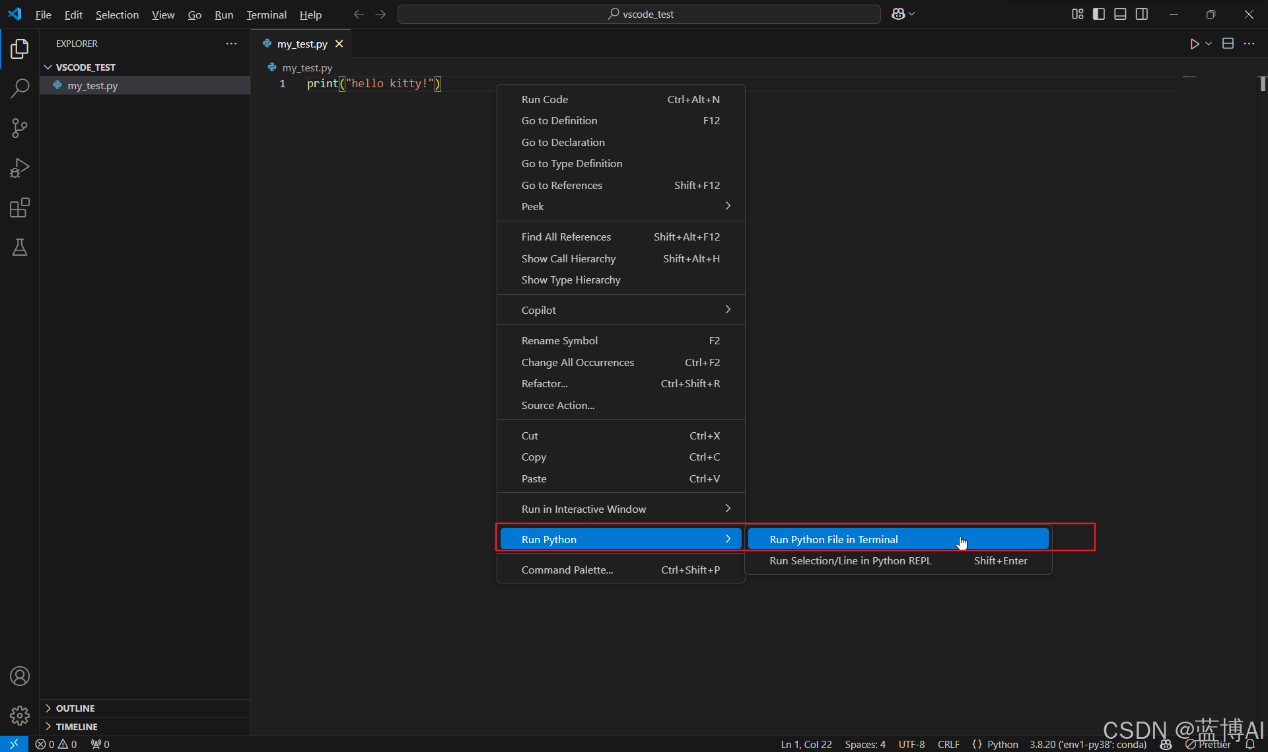
1. **验证环境配置**

**在 VS Code 中打开一个新的终端窗口，输入 python 版本检查命令：**

**确保终端使用的是你选择的 Anaconda 环境：** **正确显示虚拟环境对应的 Python 版本号，即配置成功。**

**1.3.4 运行第一个 python 程序**

****

**或者鼠标右键，点击下面的选项运行python脚本**

**2. 使用Python来进行机器学习**

**2.1 Python机器学习包**

Python是数据科学领域最著名的编程语言之一，借助非常活跃的开发者及开源社区，开发出了大量有用的科学计算和机器学习库。

虽然Python这类解释性语言的性能，在执行计算密集型任务时要逊于其它更底层的编程语言，但开发出了使用底层的Fortran和C所实现的NumPy和SciPy之类的扩展库，可对多维数组执行快速的向量化运算。

对于机器学习编程任务，我们多会使用scikit-learn库，它是当前最流行、易用的开源机器学习库之一。在后续的章节中，在聚焦于机器学习的一个分支深度学习时，我们会使用PyTorch库的最新版本，它通过使用显卡专门用于训练所谓的**深度神经网络**模型。

**2.1.1 安装Python及通过PyPI安装Python包**

三大操作系统，Microsoft Windows、macOS和Linux，均可使用Python ，可通过Python官方网站[https://www.python.org](https://www.python.org/)下载安装包及文档。

本书中的代码示例使用Python 3.9编写和测试，通常建议使用Python 3的最新稳定版本。有些代码同时兼容Python 2.7，但官方对Python 2.7的支持已于2019年终结，大部分开源库都不再支持Python 2.7 ([https://python3statement.org](https://python3statement.org/))，我们强烈建议使用Python 3.9或更新版本。

可以通过在终端（Windows下使用PowerShell ）执行如下命令来检测Python版本：

|  |  |
| --- | --- |
| 1 | python --version |

或

|  |  |
| --- | --- |
| 1 | python3 --version |

本书中所使用的其它包可通过pip安装程序来安装，pip从Python 3.3开始就集成在于标准库中，无需额外安装。有关pip的更多信息请参见<https://docs.python.org/3/installing/index.html>。

在成功安装Python之后，可以在终端中执行pip 来安装其它Python包：

|  |  |
| --- | --- |
| 1 | pip install SomePackage |

已安装的包可通过--upgrade标记来进行升级：

|  |  |
| --- | --- |
| 1 | pip install SomePackage --upgrade |

**2.1.2 使用Anaconda Python发布和包管理器**

针对科学计算强烈推荐安装来自Continuum Analytics的开源包管理系统conda。conda免费并且采用了许可式的开源证书。其目标是帮助在各操作系统中安装数据科学、数学和工程的Python以及其版本管理。如果想使用conda，有好几种选择，分别是Anaconda、Miniconda和Miniforge：

* Anaconda中预装了很多科学计算包。可通过<https://docs.anaconda.com/anaconda/install/>下载安装包，Anaconda的快速开始指南位于<https://docs.anaconda.com/anaconda/user-guide/getting-started/>。
* Miniconda是Anaconda的简化版(<https://docs.conda.io/en/latest/miniconda.html>)。和Anaconda基本一致，只是没有预装包，这深受很多人（包括作者在内）的喜爱。
* Miniforge类似于Miniconda，但它是社区维护的，使用了与Miniconda及Anaconda不同的包仓库 (conda-forge)。我们觉得Miniforge是Miniconda的一个很好的替代吕。下载及安装指导位于GitHub仓库<https://github.com/conda-forge/miniforge>。

在成功通过 Anaconda、Miniconda或Miniforge安装好了conda后，可执行如下命令安装新的Python包：

|  |  |
| --- | --- |
| 1 | conda install SomePackage |

可使用以下命令升级已有的包：

|  |  |
| --- | --- |
| 1 | conda update SomePackage |

官方conda通道中所没有的包可通过社区支持的conda-forge项目([https://conda-forge.org](https://conda-forge.org/))来获取，通过--channel conda-forge参数指定即可。例如：

|  |  |
| --- | --- |
| 1 | conda install SomePackage --channel conda-forge |

默认conda通道和conda-forge都没有的包可通过前面讲过的pip 进行安装。比如：

|  |  |
| --- | --- |
| 1 | pip install SomePackage |

**2.1.3 科学计算、数据科学和机器学习包**

使用NumPy的多维数组来存储和操作数据。有时会用到pandas，这是一个构建于NumPy之上的库，提供了一些更高阶的操作工具可以更方便地处理表格数据。为提升学习体验以及可视化量化数据，这通常有利于进行理解，我们会使用自定义的Matplotlib库。

本课程主要使用到的机器学习库有 scikit-learn和PyTorch。

主要Python包的版本号参见下方的列表。理想情况下你所安装的包的版本应该要保持一致，这样可保障示例代码能正确运行：

* NumPy 1.21.2
* SciPy 1.7.0
* Scikit-learn 1.0
* Matplotlib 3.4.3
* pandas 1.3.2

在安装好这些包之后，可以通过在Python 中导入这些包并文章其\_\_version\_\_属性来确认版本，比如：

|  |  |
| --- | --- |
| 1  2  3 | >>> import numpy  >>> numpy.\_\_version\_\_  '1.21.2' |

**使用 pip查看包的版本**

bash

# 通用格式：检查单个包版本

pip show <包名>

# 示例：检查numpy版本

pip show numpy

# 简洁输出版本号（仅显示版本）

pip list | grep <包名> # Linux/Mac（grep筛选）

pip list | findstr <包名> # Windows（findstr筛选）

# 示例：简洁查看pandas版本

pip list | grep pandas # Linux/Mac

pip list | findstr pandas # Windows

**使用 conda（适用于 conda 环境）查看包的版本**

bash

# 检查单个包版本

conda list <包名>

# 示例：检查scikit-learn版本

conda list scikit-learn

为方便起见，附属代码仓库<https://github.com/rasbt/machine-learning-book>中添加了一个python-environment-check.py脚本，可通过执行脚本来确认Python的版本和包的版本。运行**python\_environment\_check.py**查看多个包的版本。

如果代码与章节中的代码完全一致时仍然出现错误，建议首先检查底层包的版本，再进行调试。有时，新版本会引入向后不兼容的修改，可能导致错误。

如果不希望修改主Python安装中包的版本，推荐使用虚拟环境来安装包。如果使用Python却没有安装conda管理器，可以使用venv 库来新建虚拟环境。例如，可以通过如下两条命令创建及激活虚拟环境：

|  |  |
| --- | --- |
| 1  2 | python3 -m venv /Users/sebastian/Desktop/pyml-book  source /Users/sebastian/Desktop/pyml-book/bin/activate |

注意在每次打开新的终端或PowerShell时都需要激活虚拟环境。可在<https://docs.python.org/3/library/venv.html>中了解到更多有关venv 的信息。

如果使用Anaconda，可以通过如下命令创建和激活虚拟环境：

|  |  |
| --- | --- |
| 1  2 | conda create -n pyml python=3.9  conda activate pyml |

**2.2 使用Python实现感知机学习算法**

本节我们使用Python进行实现并使用[第1章 赋予计算机学习数据的能力](https://alanhou.org/giving-computers-the-ability-to-learn-from-data/)中介绍的鸢尾花数据集进行训练。

**2.2.1 面向对象的感知机API**

采用面向对象的方法将感知机接口定义为一个Python类，这样可初始化新的Perceptron对象，来通过fit方法学习数据并通过单独的predict方法完成预测。按照约定，将衍生属性（不在对象初始化期间创建，而是在后续调用对象的其它方法时动态创建的属性）命名后加上一个下划线(\_) ，，比如self.w\_。

**Python科学计算的其它资源**

如果不熟悉Python的机器学习库或是需要复习一下，可以参考如下资源：

* **NumPy**: <https://sebastianraschka.com/blog/2020/numpy-intro.html>
* **pandas**: <https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html>
* **Matplotlib**: <https://matplotlib.org/stable/tutorials/introductory/usage.html>

以下为使用Python实现的感知机：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | import numpy as np  class Perceptron:      """Perceptron 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 fitting.      b\_ : Scalar        Bias unit after fitting.      errors\_ : list        Number of misclassifications (updates) 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 : object            """          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.errors\_ = []            for \_ in range(self.n\_iter):              errors = 0              for xi, target in zip(X, y):                  update = self.eta \* (target - self.predict(xi))                  self.w\_ += update \* xi                  self.b\_ += update                  errors += int(update != 0.0)              self.errors\_.append(errors)          return self        def net\_input(self, X):          """Calculate net input"""          return np.dot(X, self.w\_) + self.b\_        def predict(self, X):          """Return class label after unit step"""          return np.where(self.net\_input(X) >= 0.0, 1, 0) |

使用这一感知机实现，可以按给定的学习率eta(η)、迭代次数n\_iter（通过训练数据集传递）来初始化新的Perceptron对象，借助于fit方法，我们将偏置self.b\_初始化为0，并将self.w\_中的权重初始化为向量第2章 为分类训练简单机器学习算法，其中的*m*表示数据集中的维数（特征数）。

注意初始权重向量为一些小随机数，通过rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])得到0.01标准差（standard deviation ）的正态分布中提取，其中rgen是NumPy的随机数字生成器，我们使用了用户指定的随机种子，这样在需要时可以复现之前的结果。

技术上讲，我们应当将权重初始化为零（事实上在原始的感知机算法中就是这么做的）。但是，如果那样的话，学习率第2章 为分类训练简单机器学习算法 (eta)就不会对决策边界产生任何效果。如果所有权重都初始化为零，学习率参数，eta，只能影响权重向量的大小，而影响不到方向。

修改self.w\_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])为self.w\_ = np.zeros(X.shape[1])，使用不同的eta值来运行下一节中的感知机训练代码。读者会看到决策边界不会变化。

在初始化权重之后，fit方法遍历训练数据集中的所有样式，并根据前面小节中讨论的感知机学习率更新权重。

类标签由predict方法预测，该方法在训练期间由fit方法调用以获取更新权重后的类标签；但predict也可在我们拟合好模型后用于预测新数据的类标签。此外，我们还在self.errors\_列表中收集每次迭代所产生的错误分类数，这样稍后可分析出训练期间感知机的表现。net\_input方法中使用的np.dot函数只是用于计算向量的点乘，**w**T**x** + *b*。

**2.2.2 对鸢尾花数据集训练感知机模型**

为测试我们的感知机实现，在剩下部分中的分析和示例中会限定为两个特征变量（维度）。虽然感知机规则不只限于两个维度，只考虑两个特征，萼片长度和花瓣长度，让我们可以将训练模型的决策区域可视化为散点图方便学习。

注意我们也只会考虑鸢尾花数据集中的两个类别setosa和versicolor，原因也很实际：感知机是一个二元分类器。但感知机算法也可扩展为多类分类，比如一**对剩余(OvA)** 技术

**用于多类分类的OvA方法**

OvA（one-versus-all）有时也称为**one-versus-rest** (**OvR**)，是一种将二元分类扩展为多类问题的技术。使用OvA，我们可以为每个类训练一个分类器，其中特定的类视为正类，其它类的样本则被划为反类。如果对新的未打标签的数据实例进行分类，我们可以使用*n*分类器，其中的*n*是类标签数，对要分类的具体实例打上最确定的分类标签。在感知机示例中，我们使用OvA选择最大绝对净输入值所关联的类标签。

**1）数据检查**

首先我们使用pandas库直接从*UCI*机器学习仓库加载鸢尾花数据集，放到DataFrame对象中并通过tail方法打印最后五行来检查所加载数据是否正确：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | >>> import os  >>> import pandas as pd  >>> s = 'https://archive.ics.uci.edu/ml/'\  ...     'machine-learning-databases/iris/iris.data'  >>> print('From URL:', s)  From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data  >>> df = pd.read\_csv(s,  ...                  header=None,  ...                  encoding='utf-8')  >>> df.tail() |

执行上述代码后，会看到显示鸢尾花数据集最后五行的如下输出：

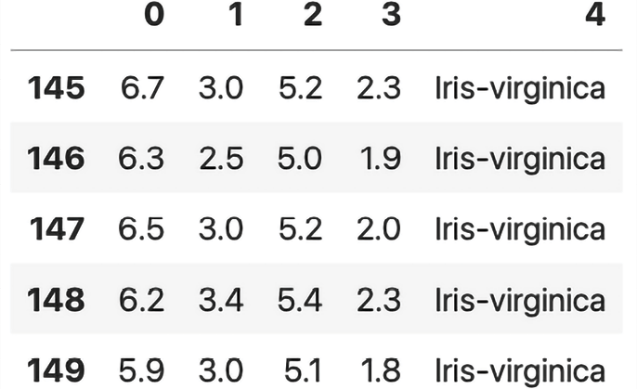


图2.5：鸢尾花数据集的最后五行

**2）加载鸢尾花数据集**

UCI服务器<https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>临时宕机。通过本地目录加载数据集时可以将如下行

|  |  |
| --- | --- |
| 1  2  3  4 | df = pd.read\_csv(    'https://archive.ics.uci.edu/ml/'    'machine-learning-databases/iris/iris.data',    header=None, encoding='utf-8') |

替换为：

|  |  |
| --- | --- |
| 1  2  3 | df = pd.read\_csv(    'your/local/path/to/iris.data',    header=None, encoding='utf-8') |

接下来我们提取与前50个Iris-setosa和50个Iris-versicolor相对应的100个类标签，并将类标签转化为两个整型类标签1 (versicolor)和0 (setosa)，赋值给向量y，其中pandas库DataFrame的values方法与NumPy中的相对应。

类似地，我们提取这100个训练样本中每一个特征列（花萼长度）和第三个特征列（花瓣长度），将其赋值给特征矩阵X，可使用二维散点图对其进行可视化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>> import matplotlib.pyplot as plt  >>> import numpy as np  >>> # select setosa and versicolor  >>> y = df.iloc[0:100, 4].values  >>> y = np.where(y == 'Iris-setosa', 0, 1)  >>> # extract sepal length and petal length  >>> X = df.iloc[0:100, [0, 2]].values  >>> # plot data  >>> plt.scatter(X[:50, 0], X[:50, 1],  ...             color='red', marker='o', label='Setosa')  >>> plt.scatter(X[50:100, 0], X[50:100, 1],  ...             color='blue', marker='s', label='Versicolor')  >>> plt.xlabel('Sepal length [cm]')  >>> plt.ylabel('Petal length [cm]')  >>> plt.legend(loc='upper left')  >>> plt.show() |

执行完上述示例代码后，会生成如下的散点图：

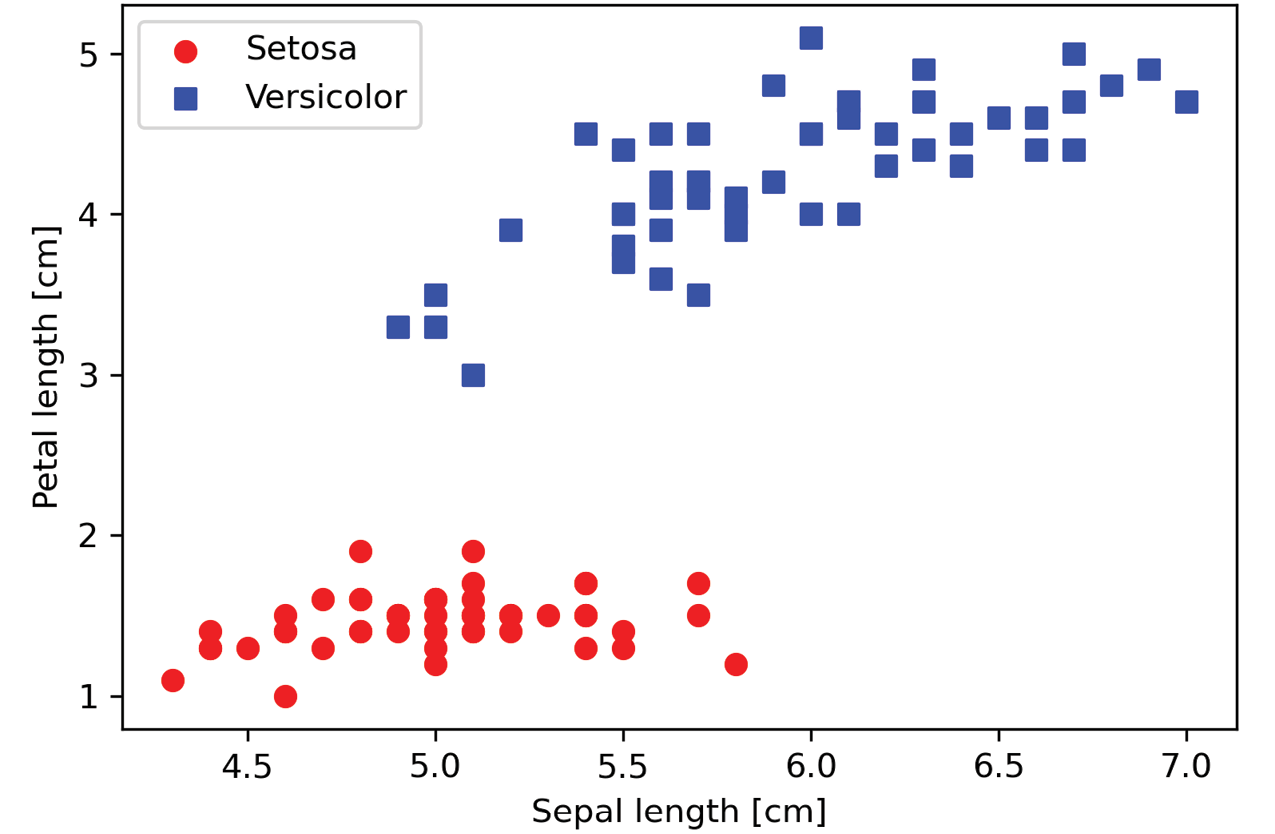


图2.6：setosa和versicolor鸢尾花的花萼长度及花瓣长度散点图

图2.6沿纵轴和横轴展示了鸢尾花数据集中样本的分布：纵轴为花瓣长度，横轴为花萼长度（单位厘米）。在这个二维特征子空间中，我们可以看出线性决策边界足够区分出setosa和versicolor花了。因此可以使用感知机这样的线性分类器来对数据集中的花进行完美分类。

下面该对所提取的鸢尾花数据子集训练感知机算法了。绘制出每次迭代的错误分类，以检查算法是否收敛并找到区分两种鸢尾花类别的决策边界：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> ppn = Perceptron(eta=0.1, n\_iter=10)  >>> ppn.fit(X, y)  >>> plt.plot(range(1, len(ppn.errors\_) + 1),  ...          ppn.errors\_, marker='o')  >>> plt.xlabel('Epochs')  >>> plt.ylabel('Number of updates')  >>> plt.show() |

注意分类错误的数量与更新数量相同，因为感知机的权重和偏置在每次错误归类样本时都会进行更新。在执行以上代码后，读者应该会看到如图2.7所示的错误归类数对迭代次数的折线图：

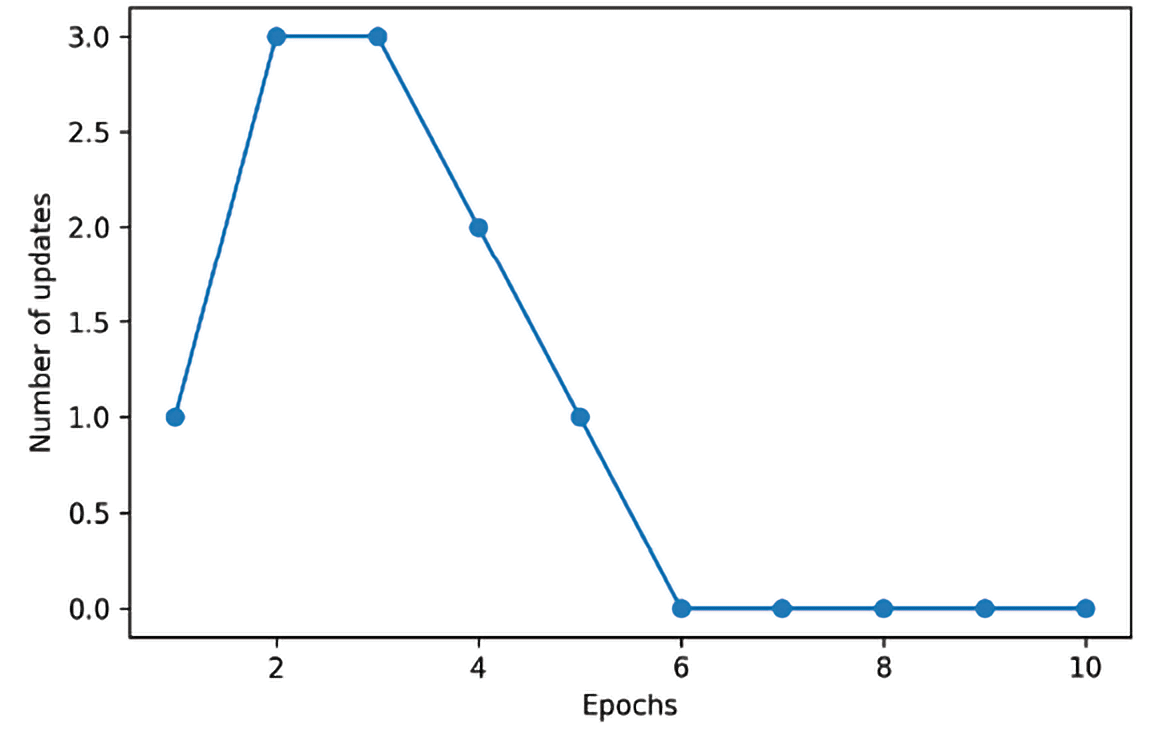


图2.7：错误归类数对迭代次数的折线图

从图2.7中可以看出，在第6次迭代后我们的感知机收敛了，此时应该可以完美地对训练样本进行分类了。我们来实现一个函数对二维数据进行决策边界的可视化：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | from matplotlib.colors import ListedColormap  def plot\_decision\_regions(X, y, classifier, resolution=0.02):      # setup marker generator and color map      markers = ('o', 's', '^', 'v', '<')      colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')      cmap = ListedColormap(colors[:len(np.unique(y))])        # plot the decision surface      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())        # plot class examples      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') |

首先我们定义了一些colors和markers并通过ListedColormap创建一个颜色列表。然后我们确定两个特征的最大、最小值，使用这些特征向量通过NumPy的meshgrid函数创建一对栅格数组xx1和xx2。因为是在两个维度上训练感知机分类器，我们需要展平栅格数组、创建相同列数的矩阵为鸢尾花子集，这样就能使用predict方法预测对应栅格点的类标签lab。

在改变预测类标签lab，以xx1和xx2放入相同维度的栅格后，我们就可以通过Matplotlib的contourf函数画一个等高线图，将不同决策树使用不同颜色来对应栅格数组中的每个预测类：

|  |  |
| --- | --- |
| 1  2  3  4  5 | >>> plot\_decision\_regions(X, y, classifier=ppn)  >>> plt.xlabel('Sepal length [cm]')  >>> plt.ylabel('Petal length [cm]')  >>> plt.legend(loc='upper left')  >>> plt.show() |

执行以上示例代码，应该会看到如图2.8所示的决策区域图：

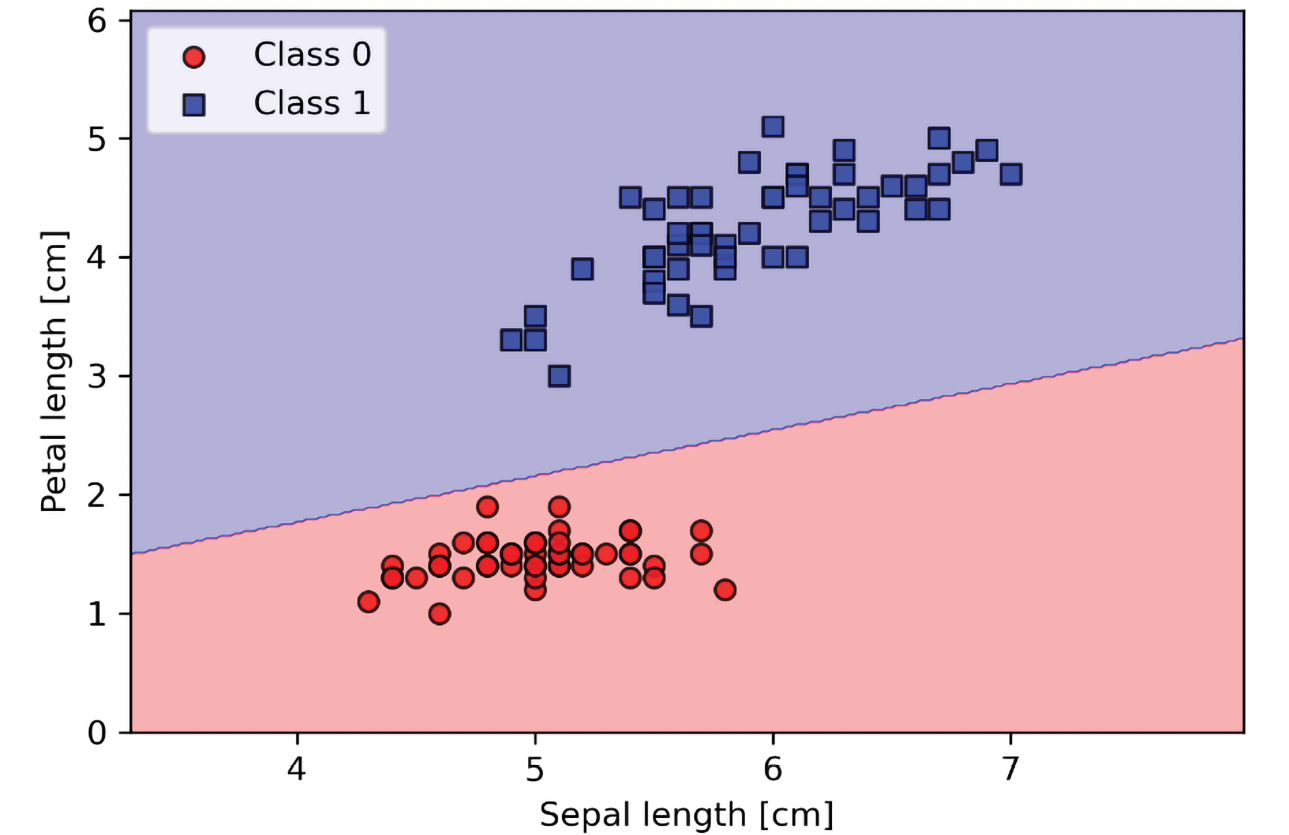


图2.8：感知机决策区域图

如图中所示，感知机学习了一个可对鸢尾花训练子集中样本进行完美分类的决策边界。

**2.2.3 感知机收敛性分析**

虽然感知机对两种鸢尾花进行了很好的分类，收敛仍是感知机的最大问题之一。Rosenblatt从数学上证明了如果两个类可通过线性超平面分割，感知机学习规则就会收敛。但是，如果类别不能由线性决策边界完全分开，权重在没有设置最大迭代次数时就会不停地进行更新。感兴趣的读者可以阅读作者在授课笔记中所做的证明总结<https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf>。

**1）自适应线性神经元及学习的收敛**

本小节中我们会学习另一类型的单层神经网络(**NN**):自适应线性神经元（**ADAptive LInear NEuron** (缩写为**Adaline**)）。Adaline由Bernard Widrow同其博士学生Tedd Hoff在Rosenblatt发表感知机算法几年后发布，可将其看成是对后者的改进(*An Adaptive “Adaline” Neuron Using Chemical “Memistors”*, *Technical Report Number 1553-2* by *B. Widrow and colleagues*, *Stanford Electron Labs*, Stanford, CA, *October*1960)。

自适应线性神经元算法有趣之处在于它描绘了定义和最小化连续损失函数的概念。对理解其它分类机器学习算法提供了基础，比如逻辑回归、支持向量机和多层神经网络，以及我们在后面章节中会讨论的线性回归模型。

Adaline学习规则（也称为**Widrow-Hoff学习规则**）与Rosenblatt的感知机主要的分别是权重根据线性激活函数而不是感知机这样的单位阶跃函数进行更新。在Adaline中，线性激活函数σ(z)只是净输入的一个恒等函数（identity function），因此σ(z)=z。

虽然线性激活函数用于学习权重，我们仍使用阈值函数来做最终决策，这与前面所讨论的单位阶跃函数相类似。

感知机和Adaline算法最主要的区别在图2.9中进行了标注：

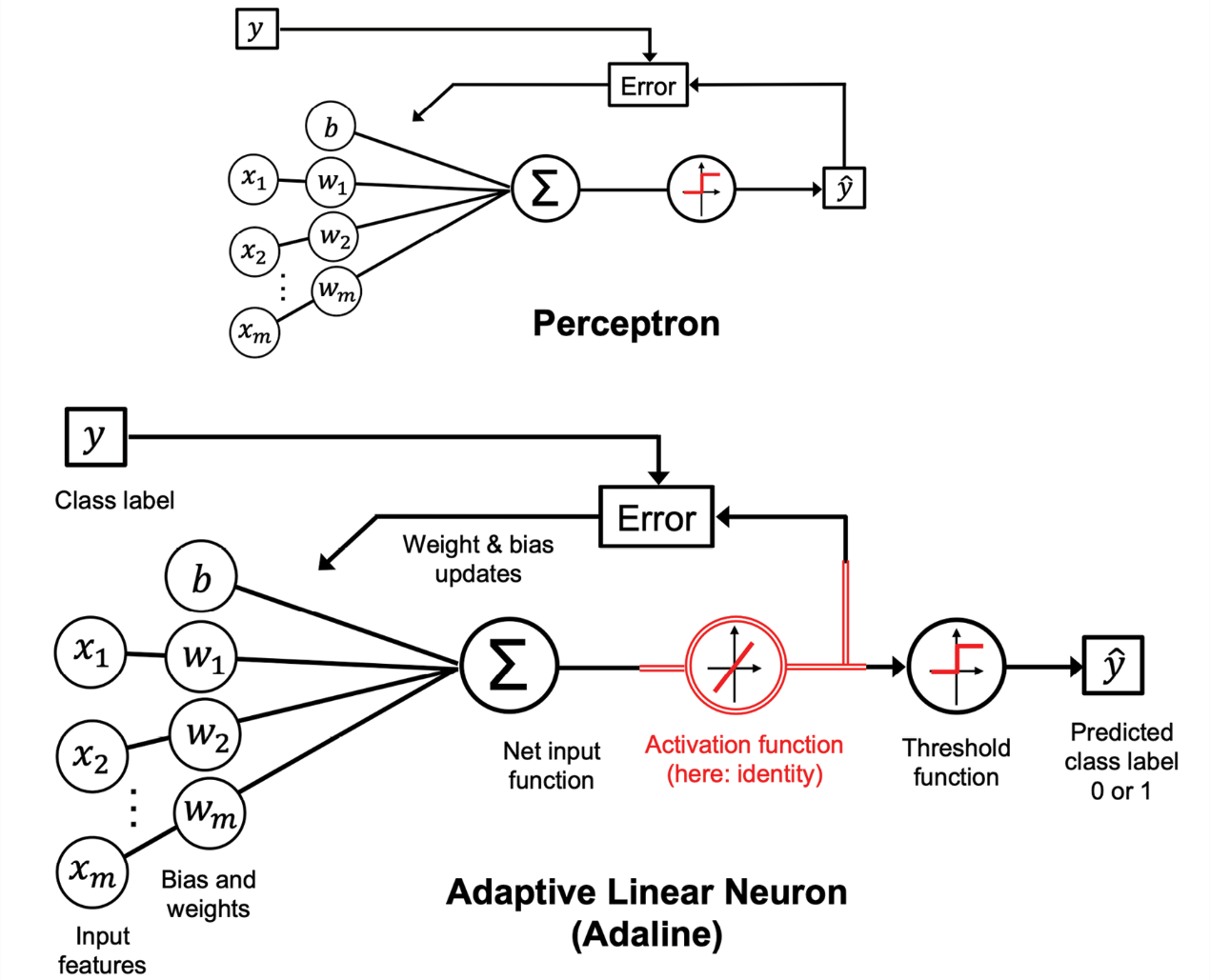
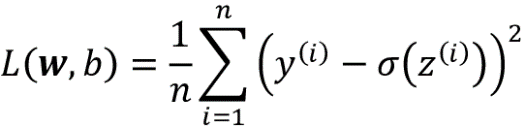


图2.9：感知机和自适应线性神经元算法的对比

在图2.9中可以看出，Adaline算法将真实类标签与线性激活函数的连续值输出对比计算模型错误并更新权重。而大型机使用真实类标签与预测的类标签进行对比。

**2）使用梯度下降最小化损失函数**

监督机器学习算法的一个主要组成是所定义的在学习过程中不断优化的目标函数。目标函数通常是我们希望最小化的损失函数或成本函数。对于Adaline算法而言，我们可以定义损失函数*L*，来以计算结果和真实类标签的平均方差(**MSE**)学习模型参数：



该连续线性激活函数相对于单位阶跃函数的主要优势是损失函数可微分的。损失函数的另一个优秀属性是它是凸函数，因此我们可以非常简单但强大的优化算法梯度下降来找到最小化损失函数的权重，对鸢尾花数据集中的样式进行分类。

如图2.10中所示，我们可以将梯度下降的主体思想描述为下山，直至抵达本地或全局损失最小值。在每次迭代中，我们沿倾斜的相反方向走一步，步长由学习率的值以及梯度决定（为简化起见，下图只使用了一个权重*w*）：

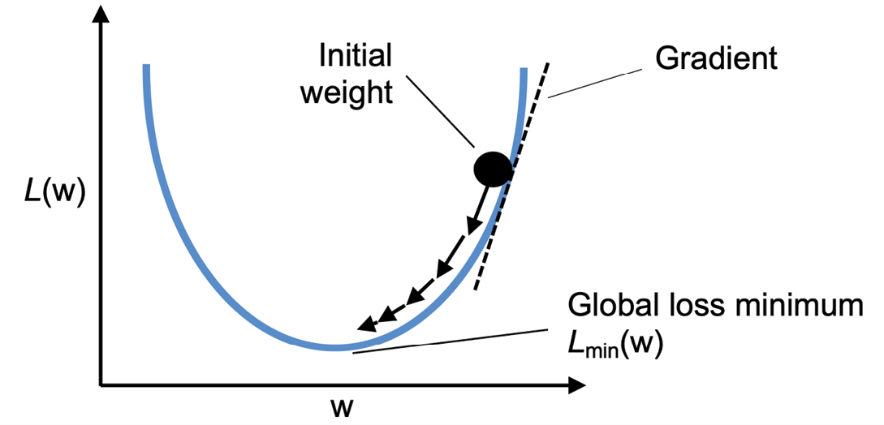


图2.10：梯度下降的原理

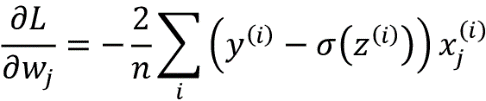
使用梯度下降，现在我们可以沿斜坡相反方向一步步更新模型参数第2章 为分类训练简单机器学习算法或损失函数*L*(**w**, *b*)：

第2章 为分类训练简单机器学习算法

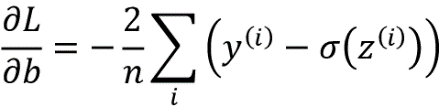
参数变化第2章 为分类训练简单机器学习算法和第2章 为分类训练简单机器学习算法由负梯度乘上学习率第2章 为分类训练简单机器学习算法：

第2章 为分类训练简单机器学习算法

要计算损失函数的梯度，我们需要计算损失函数与各个权重*w*j的偏导数：

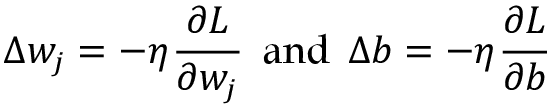


类似地，我们计算损失与偏置的偏导数：



请注意上面分母中的2中是一个恒定比例因子，我们可以省略掉也不影响算法。删除比例因子与按2倍数修改学习率等效。下面会讲解比例因子的来源。

我们可以将权重更新写为：

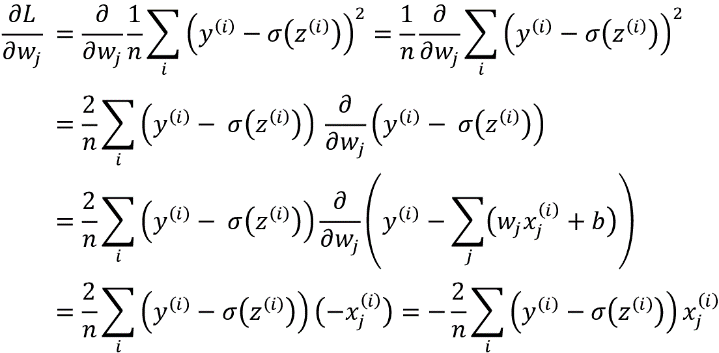


因为我们同步更新了所有参数，Adaline学习规则便变成了：

第2章 为分类训练简单机器学习算法

**均方根误差导数**

熟悉微积分的读者会知道，MSE（均方根）损失函数对第*j*个权重的偏导数可通过如下方式求取：



可使用同样的方式未取第2章 为分类训练简单机器学习算法的偏导数，只是第2章 为分类训练简单机器学习算法等于-1，因此最后一步可简化为第2章 为分类训练简单机器学习算法。

虽然Adaline学习规则和感知机规则看起来一样，应该注意是第2章 为分类训练简单机器学习算法一个真实数字而不是一个整型类标签，其中第2章 为分类训练简单机器学习算法。此外，权重更新根据训练数据集中的所有样本进行计算（而不是在每个训练样本后增量更新参数），这也是为什么这种方法也被称为批量梯度下降。为更加明确以及在本章和本书稍后讨论相关概念时避免混淆，我们会将这一过程称为全批量梯度下降。

**3）使用Python实现Adaline算法**

因为感知机学习规则和Adaline学习规则非常相似，我们会取之前定义的感知机实现，修改其中的fit方法以使用权重和偏置参数按梯度下降的最小化损失函数更新：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | class AdalineGD:      """ADAptive LInear NEuron 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 fitting.      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 : object            """          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 = (errors\*\*2).mean()              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, X):          """Compute linear activation"""          return X        def predict(self, X):          """Return class label after unit step"""          return np.where(self.activation(self.net\_input(X))                          >= 0.5, 1, 0) |

不像感知机中在每个训练样本评估后更新权重，这里我们根据整个训练数据集计算梯度。对于偏置单元，通过self.eta \* 2.0 \* errors.mean()计算，其中errors是包含偏导数值第2章 为分类训练简单机器学习算法的数组。类似地，我们会更新权重。但注意根据偏导数第2章 为分类训练简单机器学习算法更新权重时涉及到特征值*x*j，可通过将errors乘上每个权重的特征值来计算：

|  |  |
| --- | --- |
| 1  2  3 | for w\_j in range(self.w\_.shape[0]):                  self.w\_[w\_j] += self.eta \*                      (2.0 \* (X[:, w\_j]\*errors)).mean() |

要不通过for循环更高效地实现权重更新，我们可以使用特征矩阵和错误向量的矩阵-向量乘法：

|  |  |
| --- | --- |
| 1 | self.w\_ += self.eta \* 2.0 \* X.T.dot(errors) / X.shape[0] |

请注意activation方法对代码没有效果，因为它只是一个恒等函数。这里我们添加了激活函数（通过activation方法计算）来描述信息如何在单层神经网络中流动的整体概念：来自输入数据的特征、净输入、激活和输出。

现在，类似前面的感知机实现，我们将损失值收集到一个self.losses\_列表中用于检查在训练后算法是否收敛。

实践中，常常要求进行多次实验才能找到对应最佳收敛的学习率第2章 为分类训练简单机器学习算法。因此我们先选择两个学习率第2章 为分类训练简单机器学习算法和第2章 为分类训练简单机器学习算法，使用损失函数和迭代次数绘图查看Adaline实现对训练数据学习的效果。

**4）超参数影响**

学习率第2章 为分类训练简单机器学习算法(eta)以及迭代次数(n\_iter)，也被称作感知机和Adaline学习算法的超参数（或调优参数）。在[第6章 学习模型评估和超参数调优的最佳实践](https://alanhou.org/learning-best-practices-for-model-evaluation-and-hyperparameter-tuning)中，我们会学习各种技术自动查找产生分类模型最优表现的不同超参数值。

下面绘制两种学习率的损失对迭代次数的图像：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | >>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))  >>> ada1 = AdalineGD(n\_iter=15, eta=0.1).fit(X, y)  >>> ax[0].plot(range(1, len(ada1.losses\_) + 1),  ...            np.log10(ada1.losses\_), marker='o')  >>> ax[0].set\_xlabel('Epochs')  >>> ax[0].set\_ylabel('log(Mean squared error)')  >>> ax[0].set\_title('Adaline - Learning rate 0.1')  >>> ada2 = AdalineGD(n\_iter=15, eta=0.0001).fit(X, y)  >>> ax[1].plot(range(1, len(ada2.losses\_) + 1),  ...            ada2.losses\_, marker='o')  >>> ax[1].set\_xlabel('Epochs')  >>> ax[1].set\_ylabel('Mean squared error')  >>> ax[1].set\_title('Adaline - Learning rate 0.0001')  >>> plt.show() |

从最终的损失函数图中可以看出，我们遇到了两种类型的问题。左图显示了选择了过大的学习率时的情况。它没有最小化损失函数，而是每次迭代均方误差都在变大，因为超过全局最小值。而另一边，我们可以看到右图的损失在下降，但所选的学习率第2章 为分类训练简单机器学习算法太小了，算法需要经过大量的迭代才能收敛至全局最小损失：

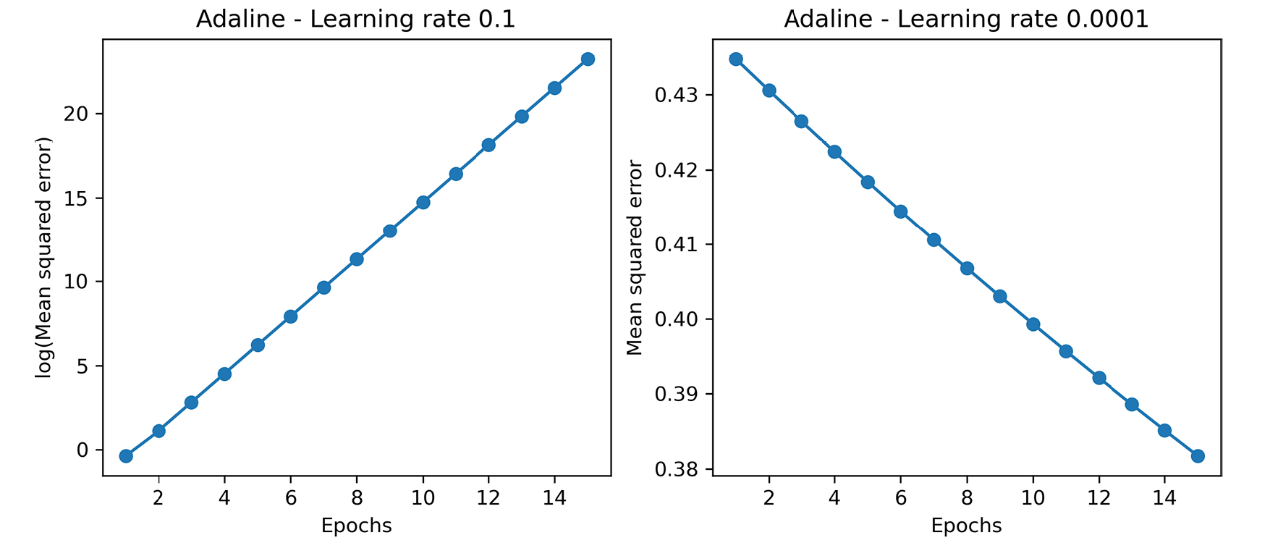


图2.11：次优学习率的误差图

图2.12描绘了如果修改具体的权重参数值来最小化损失函数*L*会发生什么。左图为精选学习率的示例，其中损失递减，沿全局最小值的方向移动。

而右图描绘了如果选择的学习率过大，会超过全局最小值：

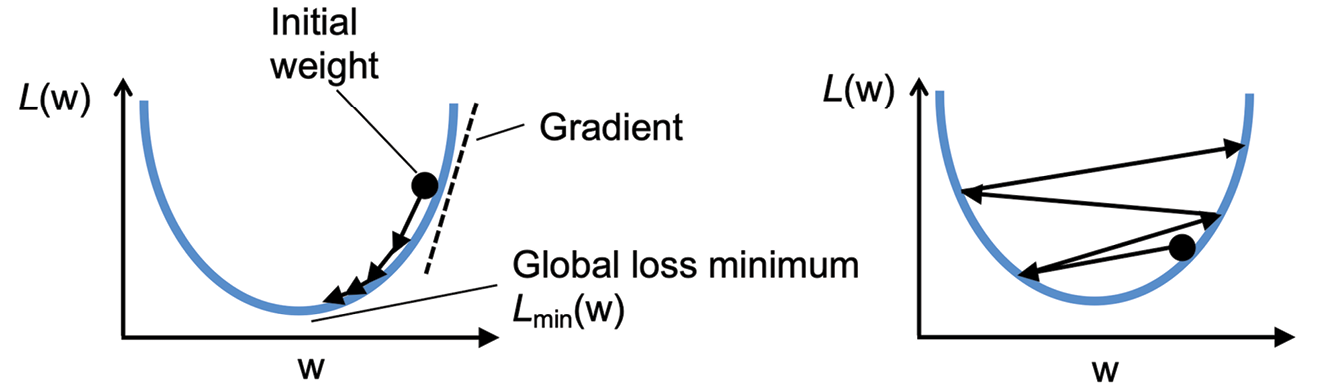
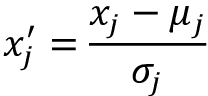


图2.12：精选学习率和学习率过大的对比

**5）通过特征缩放改进梯度下降**

很多机器学习算法会需要某种程度的特征缩放以实现最佳效果，这会在[第3章 使用Scikit-Learn的机器学习分类器之旅](https://alanhou.org/a-tour-of-machine-learning-classifiers-using-scikit-learn)和[第4章 构建优秀训练数据集 – 数据预处理](https://alanhou.org/building-good-training-datasets-data-preprocessing)中进行详细讨论。

梯度下降是受益于特征缩放的多种算法之一。本小节中，我们会使用称为标准化（**standardization**）的特征缩放方法。这种归一化处理有助于让梯度下降学习更快速地收敛，但不会让原数据集正态分布。归一化平衡每个特征的均值，让其中心点为零并且每个特征的标准差为1（单位方差）。例如，要归一化第*j*个特征，可以对每个训练样本减去样式均值第2章 为分类训练简单机器学习算法并除以标准差第2章 为分类训练简单机器学习算法：



这里*x*j是包含所有训练样式*n*第*j*个特征值的向量，这一标准化技术应用于数据集中的每个特征*j*。

标准化有助于梯度下降学习的原因之一是它更易于找到对所有权重（及偏置）可良好运作的学习率。如果特征规模很大，可很好更新一个权重的学习率可能对于更新另一个权重就会太大或太小。总的来说，使用标准化的特征可使用训练稳定，这样优化器可经过更少的步骤找到一个好的或最佳解法（全局损失最小）。图2.13描述了未缩放特征（左）和标准化特征（右）可能出现的梯度更新，其中的同心圆表示二维分类问题中两个模型权重函数的损失表面：

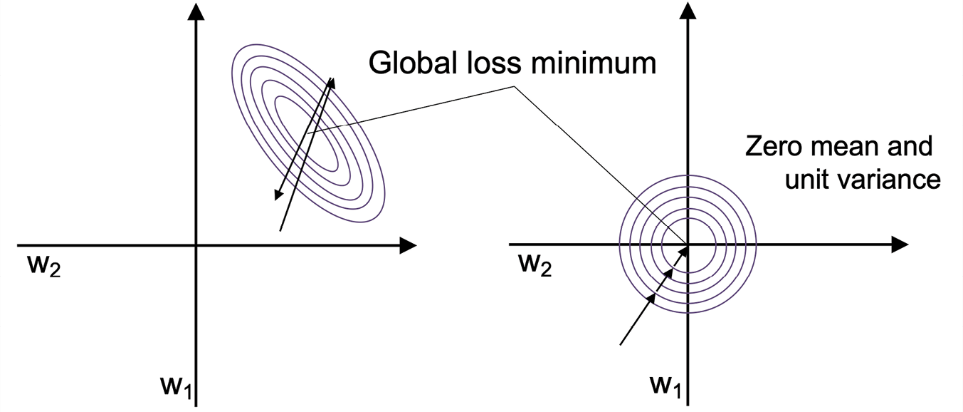


图2.13：对比未缩放和标准化特征的梯度更新

标准化可通过NumPy内置的mean和std方法轻松实现：

|  |  |
| --- | --- |
| 1  2  3 | >>> X\_std = np.copy(X)  >>> X\_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()  >>> X\_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std() |

在进行标准化后，我们会再次训练Adaline并看到使用学习率第2章 为分类训练简单机器学习算法它会经过很少的迭代就收敛：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> ada\_gd = AdalineGD(n\_iter=20, eta=0.5)  >>> ada\_gd.fit(X\_std, y)  >>> plot\_decision\_regions(X\_std, y, classifier=ada\_gd)  >>> plt.title('Adaline - Gradient descent')  >>> plt.xlabel('Sepal length [standardized]')  >>> plt.ylabel('Petal length [standardized]')  >>> plt.legend(loc='upper left')  >>> plt.tight\_layout()  >>> plt.show()  >>> plt.plot(range(1, len(ada\_gd.losses\_) + 1),  ...          ada\_gd.losses\_, marker='o')  >>> plt.xlabel('Epochs')  >>> plt.ylabel('Mean squared error')  >>> plt.tight\_layout()  >>> plt.show() |

执行这段代码，应该会看到一个决策区域图，以及一个损失下降图，如图2.14：

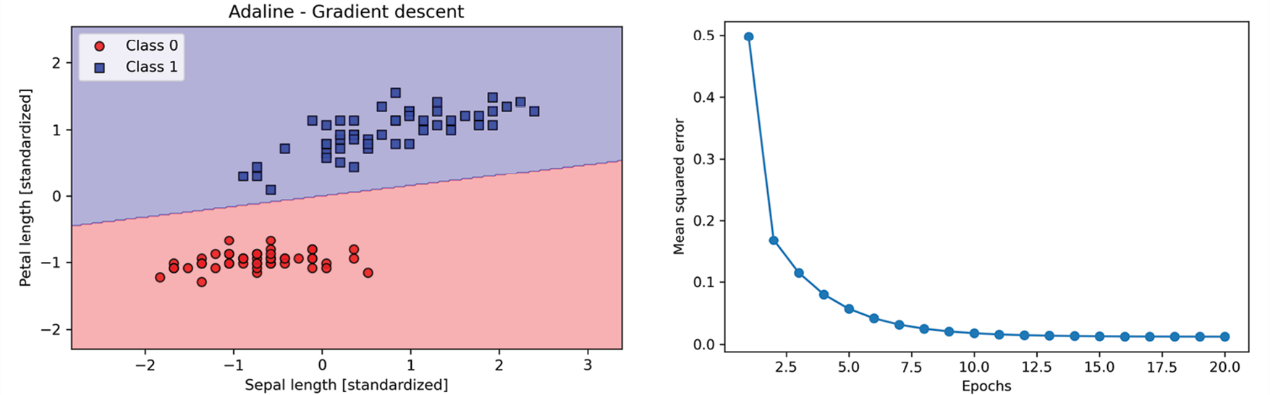


图2.14：Adaline的决策区域及均方差对迭代次数图

从图中可以看出，在通过标准化特征训练后现在Adaline收敛了。但注意虽然所有的样本花分类正确均方差仍不是零。

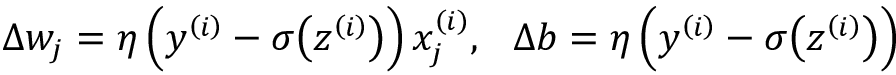
**2.2.4 随机梯度下降算法**

在前一小节中，我们学习了如何通过整体训练数据集计算损失梯度相反方向的步骤最小化损失函数，这也是这种方法有时也称作全批量梯度下降的原因。现在想象下我们有几百万数据点的超大数据集 ，在机器学习应用中这并不罕见。运行全批量梯度下降在这种场景下计算开销很大，因为每朝全局最小走一步就需要重新运算整个训练数据集。

全批量梯度下降一个著名的替代是随机梯度下降(**SGD**)，有时也称为迭代或在线梯度下降。它不是根据全部训练样本**x**(i)的累积误差总和更新权重：



而是对每个训练样本增量更新参数，比如：



虽然可将SGD看作与梯度下降类似，但通常因更高频次的权重更新它更可以更快速地收敛。因为每个梯度按单个样式计算，其误差面比梯度下降中的噪音更大，在使用非线性损失函数时也就让SGD提前具备了逃脱浅层全局最小值的优势，这在[第11章 从零实现多层人工神经网络](https://alanhou.org/implementing-a-multilayer-artificial-neural-network-from-scratch)中会学习到。为通过随机梯度下降获取满意的结果，以随机顺序提供训练数据很重要，同时，我们最好在每次迭代时打乱训练数据集以避免循环。

**1）在训练期间调整学习率**

在SGD中，固定的学习率第2章 为分类训练简单机器学习算法，通常由随时间下降的自适应学习率替换，例如：



其中*c*1和*c*2是常量。注意SGD没有到达全局损失最小值，而是一个非常接近它的区域。并且使用自适应学习率，我们会更接近损失最小值。

SGD的另一个优势是可用于在线学习。在线学习时，新训练数据一到达就实时训练模型。这对于累积大量数据时尤为有用，比如web应用的客户数据。使用在线数据，系统可立即应对变化，并且在存储空间不足时更新完模型就可以删除训练数据。

**2）小批量梯度下降**

在全批量梯度下降和SGD之间的一个折中称为**mini-batch梯度下降**。mini-batch梯度下降理解为将全批量梯度下降应用于更小的训练数据子集，比如每次32个训练样本。mini-batch相对全批量梯度下降的优势是收敛更快，因为权重更新的更频繁。此外mini-batch学习让我们可以将随机梯度下降中的for循环换成使用线性代数概念的向量化运算（比如通过点乘实现加权和），这可以进一步提升学习算法的计算效率。

我们已经使用梯度下降实现了Adaline学习规则，只需要做很少的就修改就可以通过SGD更新权重。在fit方法中，现在我们在每个训练样本后都会更新权重。此外，我们还会实现一个partial\_fit方法，对于在线学习它不会重新初始化权重。为了检查训练后该算法是否收敛，我们会在每次迭代中以训练样本的平均损失计算损失。并且我们会添加一个在每次迭代前打乱训练数据的选项，以避免优化损失函数时的反复循环；通过random\_state参数，可以指定用于保障可复现随机种子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110 | class AdalineSGD:      """ADAptive LInear NEuron classifier.        Parameters      ------------      eta : float          Learning rate (between 0.0 and 1.0)      n\_iter : int          Passes over the training dataset.      shuffle : bool (default: True)          Shuffles training data every epoch if True to prevent          cycles.      random\_state : int          Random number generator seed for random weight          initialization.          Attributes      -----------      w\_ : 1d-array          Weights after fitting.      b\_ : Scalar          Bias unit after fitting.      losses\_ : list          Mean squared error loss function value averaged over all          training examples in each epoch.          """      def \_\_init\_\_(self, eta=0.01, n\_iter=10,                   shuffle=True, random\_state=None):          self.eta = eta          self.n\_iter = n\_iter          self.w\_initialized = False          self.shuffle = shuffle          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 : object            """          self.\_initialize\_weights(X.shape[1])          self.losses\_ = []          for i in range(self.n\_iter):              if self.shuffle:                  X, y = self.\_shuffle(X, y)              losses = []              for xi, target in zip(X, y):                  losses.append(self.\_update\_weights(xi, target))              avg\_loss = np.mean(losses)              self.losses\_.append(avg\_loss)          return self        def partial\_fit(self, X, y):          """Fit training data without reinitializing the weights"""          if not self.w\_initialized:              self.\_initialize\_weights(X.shape[1])          if y.ravel().shape[0] > 1:              for xi, target in zip(X, y):                  self.\_update\_weights(xi, target)          else:              self.\_update\_weights(X, y)          return self        def \_shuffle(self, X, y):          """Shuffle training data"""          r = self.rgen.permutation(len(y))          return X[r], y[r]        def \_initialize\_weights(self, m):          """Initialize weights to small random numbers"""          self.rgen = np.random.RandomState(self.random\_state)          self.w\_ = self.rgen.normal(loc=0.0, scale=0.01,                                     size=m)          self.b\_ = np.float\_(0.)          self.w\_initialized = True        def \_update\_weights(self, xi, target):          """Apply Adaline learning rule to update the weights"""          output = self.activation(self.net\_input(xi))          error = (target - output)          self.w\_ += self.eta \* 2.0 \* xi \* (error)          self.b\_ += self.eta \* 2.0 \* error          loss = error\*\*2          return loss        def net\_input(self, X):          """Calculate net input"""          return np.dot(X, self.w\_) + self.b\_        def activation(self, X):          """Compute linear activation"""          return X        def predict(self, X):          """Return class label after unit step"""          return np.where(self.activation(self.net\_input(X))                          >= 0.5, 1, 0) |

现在我们在AdalineSGD分类器中使用的\_shuffle方法运行方式如下：通过np.random中的permutation函数，我们生成了0到100范围内唯一数的随机序列。然后可使用这些数字作为索引打乱特征矩阵和类标签向量。

之后我们可以使用fit方法训练AdalineSGD分类器，并使用plot\_decision\_regions来绘训练结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | >>> ada\_sgd = AdalineSGD(n\_iter=15, eta=0.01, random\_state=1)  >>> ada\_sgd.fit(X\_std, y)  >>> plot\_decision\_regions(X\_std, y, classifier=ada\_sgd)  >>> plt.title('Adaline - Stochastic gradient descent')  >>> plt.xlabel('Sepal length [standardized]')  >>> plt.ylabel('Petal length [standardized]')  >>> plt.legend(loc='upper left')  >>> plt.tight\_layout()  >>> plt.show()  >>> plt.plot(range(1, len(ada\_sgd.losses\_) + 1), ada\_sgd.losses\_,  ...          marker='o')  >>> plt.xlabel('Epochs')  >>> plt.ylabel('Average loss')  >>> plt.tight\_layout()  >>> plt.show() |

执行示例代码后我们获取的两张图如图2.15：

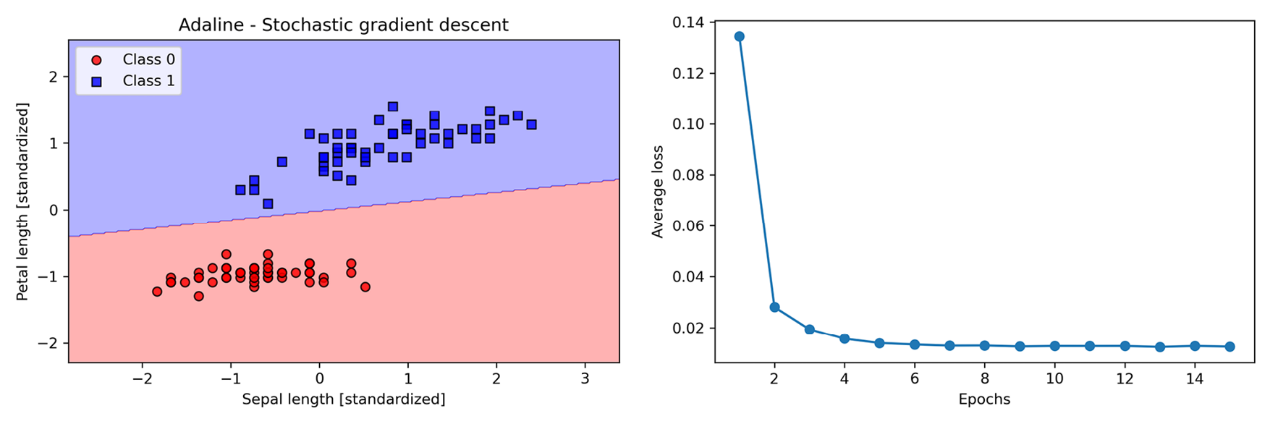


图2.15：使用SGD训练Adaline模型后的决策区域和平均损失图

可以看到，平均损失下降很快，15次迭代后的最终决策边界与批量梯度下降Adaline很类似。举个例子，如果希望使用在线学习场景的流数据更新我们的模型，只需对单独的训练样本调用partial\_fit方法，如ada\_sgd.partial\_fit(X\_std[0, :], y[0])。

**小结**

本章中，我们很好地掌握了监督学习线性分类器的基本概念。在实现了感知机后，我们学习了如何通过梯度下降的向量化实现有效实现自适应线性神经元以及通过SGD实现在线学习。

下一章，我们会使用Python的scikit-learn机器学习库来实现更高级、更强大的机器学习分类器。

我们用于实现感知机和Adaline算法的面向对象方法也会有助于对scikit-learn API的学习，其实现同样基于本章中所使用的核心概念：fit和predict方法。根据这些核心概念，我们会学习类概率的逻辑回归建模以及可用于非线性决策边界的支持向量机。此外，我们还会介绍另一种监督学习算法，基于树的算法，常常并入健壮的集成分类器（ensemble classifiers）。