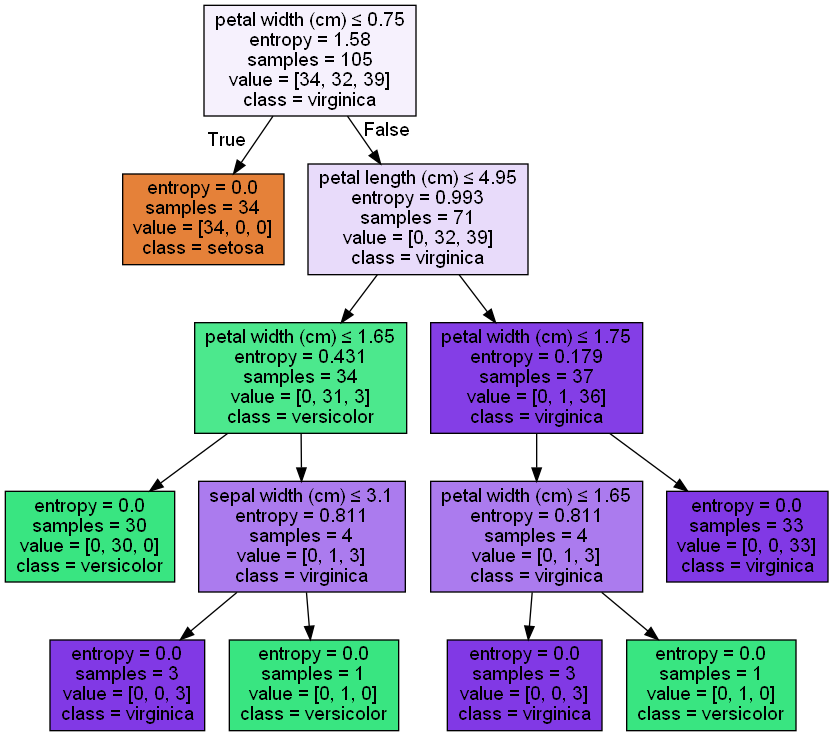
[1.数据分析](#数学基础)

**7.分类**

**决策树的图片展示——IO流存储dot**

from sklearn import datasets  
from sklearn import tree  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
from sklearn.tree import export\_graphviz  
from six import StringIO  
import pydotplus  
  
Iris = datasets.load\_iris() # 导入鸢尾花数据集  
X = Iris.data # 获得样本特征向量  
Y = Iris.target # 获得样本label  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=0)  
print("target的名称:%s" % Iris.target\_names)  
  
tree = tree.DecisionTreeClassifier(criterion='entropy') # 使用的划分标准是信息熵entropy  
tree.fit(x\_train, y\_train)  
print("决策树模型的训练集准确率:%.3f" % tree.score(x\_train, y\_train))  
print("决策树模型的测试集准确率:%.3f" % tree.score(x\_test, y\_test))  
y\_hat = tree.predict(x\_test)  
print(classification\_report(y\_test, y\_hat, target\_names=Iris.target\_names))  
  
# 生成决策树可视化的DOT文件  
dot\_data = StringIO()  
export\_graphviz(tree, out\_file=dot\_data, filled=True, feature\_names=Iris.feature\_names, class\_names=Iris.target\_names,  
 fontname='Arial', special\_characters=True)  
dot\_data = dot\_data.getvalue()  
dot\_data = dot\_data.replace('\n', '') # 去除右上角黑块  
# 使用pydotplus生成图像  
graph = pydotplus.graph\_from\_dot\_data(dot\_data)  
graph.write\_png("output/iris\_decision\_tree.png") # 将决策树保存为PNG文件

输出



target的名称:['setosa' 'versicolor' 'virginica']

决策树模型的训练集准确率:1.000

决策树模型的测试集准确率:0.978

precision recall f1-score support

setosa 1.00 1.00 1.00 16

versicolor 1.00 0.94 0.97 18

virginica 0.92 1.00 0.96 11

accuracy 0.98 45

macro avg 0.97 0.98 0.98 45

weighted avg 0.98 0.98 0.98 45

在训练集上准确率为100%，表示模型能完美地对训练数据分类，但也暗示可能过拟合。

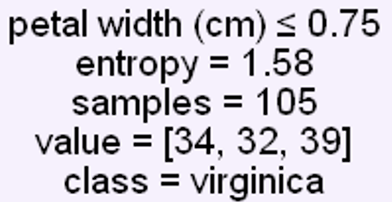
在测试集上准确率为97.8%。表明模型在未见过的数据上表现很好，具有较高泛化能力。

第一个矩阵代表三个target的**查准率**、**查全率**、**F1度量**、**支持的样本数量**。

第二个矩阵Accuracy**精度**：整个模型在所有类别上的正确分类比例。

Macro average**宏平均**：对每个类别的评估指标取平均值，不考虑类别的样本数量，以precision为例，。

Weighted average**加权平均**：对每个类别的评估指标取加权平均值，考虑不同类别的样本数量。以precision为例，。

**决策树图形的****参数讲解：**

**第一行**：划分依据。**第二行**：熵。

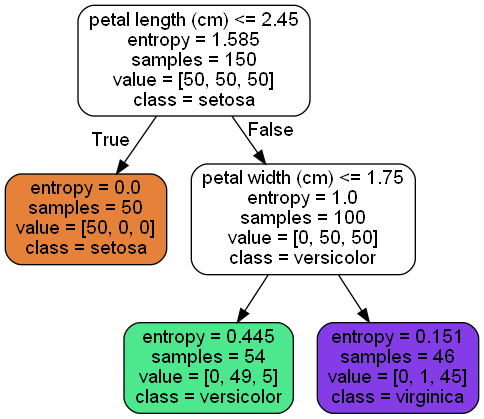
**第三行**：本次划分前有105个样本。

**第四行**：样本标签有三个，数量依次为34,32,39。

**第五行**：代表类别(叶子结点的class才代表最终标签)。

**决策树的图片展示——本地存储dot**

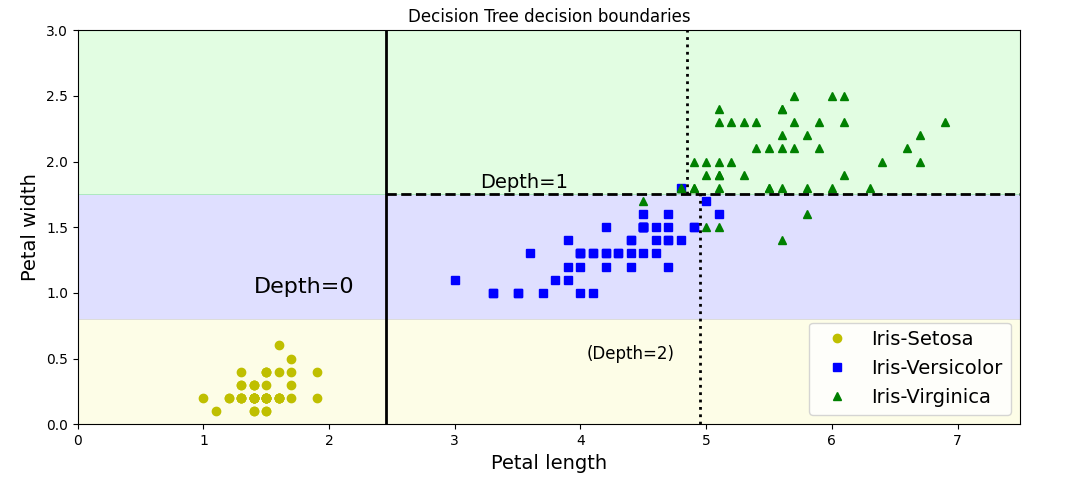
import pydotplus  
import os  
from sklearn.datasets import load\_iris  
from sklearn.datasets import make\_moons  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.tree import export\_graphviz  
from matplotlib.colors import ListedColormap  
import numpy as np  
import matplotlib.pyplot as plt  
  
  
Iris = load\_iris()  
X = Iris.data[:, 2:] # petal length and width  
y = Iris.target  
  
tree\_clf = DecisionTreeClassifier(max\_depth=2, criterion='entropy')  
tree\_clf.fit(X, y)  
  
os.makedirs("output/决策树/", exist\_ok=True) # 创建目录，如果不存在则创建  
export\_graphviz(  
 tree\_clf,  
 out\_file="output/决策树/Iris\_tree.dot",  
 feature\_names=Iris.feature\_names[2:],  
 class\_names=Iris.target\_names,  
 rounded=True,  
 filled=True  
)  
graph = pydotplus.graph\_from\_dot\_file("output/决策树/Iris\_tree.dot")  
dot\_data = graph.to\_string()  
dot\_data = dot\_data.replace('"\\r\\n";', '').replace('\n', '') # 去除黑块  
graph = pydotplus.graph\_from\_dot\_data(dot\_data)  
graph.write\_png("output/决策树/Iris\_tree.png")



注意两种方式去除黑块的不同，因为存到本地后多了一个"\\r\\n";，所以不一样。

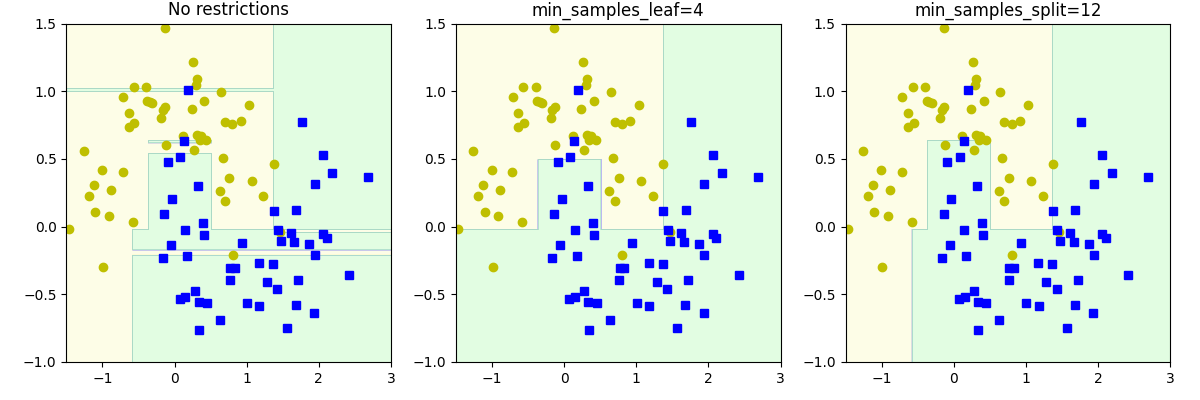
**决策树的决策边界展示**

def plot\_decision\_boundary(clf, X, y, plot\_boundary, draw\_boundary=False, label="other"):  
 x1s = np.linspace(plot\_boundary[0], plot\_boundary[1], 1000)  
 x2s = np.linspace(plot\_boundary[2], plot\_boundary[3], 1000)  
 x1, x2 = np.meshgrid(x1s, x2s)  
 X\_new = np.c\_[x1.ravel(), x2.ravel()]  
 y\_pred = clf.predict(X\_new).reshape(x1.shape)  
 custom\_cmap = ListedColormap(['#fafab0', '#9898ff', '#a0faa0'])  
 plt.contourf(x1, x2, y\_pred, alpha=0.3, cmap=custom\_cmap)  
  
 plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], "yo", label="Iris-Setosa")  
 plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], "bs", label="Iris-Versicolor")  
 plt.plot(X[:, 0][y == 2], X[:, 1][y == 2], "g^", label="Iris-Virginica")  
  
 if label == "Iris":  
 plt.xlabel("Petal length", fontsize=14)  
 plt.ylabel("Petal width", fontsize=14)  
 plt.title('Decision Tree decision boundaries')  
 plt.legend(loc="lower right", fontsize=14)  
  
 if draw\_boundary:  
 plt.plot([2.45, 2.45], [0, 3], "k-", linewidth=2) # 数据由决策树的树图像给出  
 plt.plot([2.45, 7.5], [1.75, 1.75], "k--", linewidth=2)  
 plt.plot([4.95, 4.95], [0, 1.75], "k:", linewidth=2)  
 plt.plot([4.85, 4.85], [1.75, 3], "k:", linewidth=2)  
 plt.text(1.40, 1.0, "Depth=0", fontsize=16)  
 plt.text(3.2, 1.80, "Depth=1", fontsize=14)  
 plt.text(4.05, 0.5, "(Depth=2)", fontsize=12) # 加括号代表这其实是下一层决策树才能给出具体的分割数字  
  
plt.figure()  
plot\_decision\_boundary(tree\_clf, X, y, draw\_boundary=True, plot\_boundary=[0, 7.5, 0, 3], label="Iris")  
plt.show()



**决策树的“正则化”——限制决策树形状，避免过拟合**

X, y = make\_moons(n\_samples=100, noise=0.25, random\_state=42)  
tree\_clf1 = DecisionTreeClassifier(random\_state=42)  
tree\_clf2 = DecisionTreeClassifier(min\_samples\_leaf=4, random\_state=42)  
tree\_clf3 = DecisionTreeClassifier(min\_samples\_split=12, random\_state=42)  
tree\_clf1.fit(X, y)  
tree\_clf2.fit(X, y)  
tree\_clf3.fit(X, y)  
plt.figure(figsize=(12, 4))  
plt.subplot(131)  
plot\_decision\_boundary(tree\_clf1, X, y, plot\_boundary=[-1.5, 3, -1, 1.5])  
plt.title('No restrictions')  
plt.subplot(132)  
plot\_decision\_boundary(tree\_clf2, X, y, plot\_boundary=[-1.5, 3, -1, 1.5])  
plt.title('min\_samples\_leaf=4')  
plt.subplot(133)  
plot\_decision\_boundary(tree\_clf3, X, y, plot\_boundary=[-1.5, 3, -1, 1.5])  
plt.title('min\_samples\_split=12')  
plt.show()



**常见参数：**

max\_depth(树最大的深度)

min\_samples\_split（节点在分割之前必须具有的最小样本数）

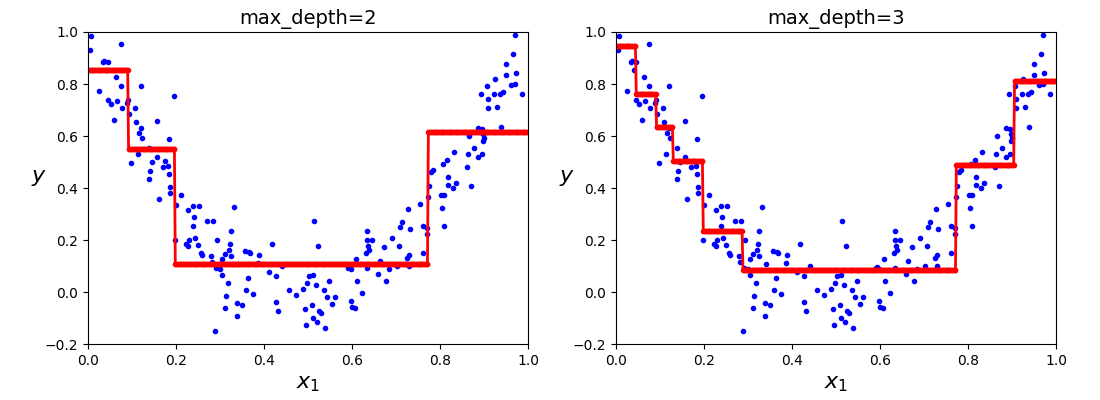
min\_samples\_leaf（叶子节点必须具有的最小样本数）

max\_leaf\_nodes（叶子节点的最大数量）

max\_features（在每个节点处评估用于拆分的最大特征数）

**决策树解决回归问题**

np.random.seed(42)  
m = 200  
X = np.random.rand(m, 1)  
y = 4 \* (X - 0.5) \*\* 2  
y = y + np.random.randn(m, 1) / 10  
tree\_reg1 = DecisionTreeRegressor(random\_state=42, max\_depth=2)  
tree\_reg2 = DecisionTreeRegressor(random\_state=42, max\_depth=3)  
tree\_reg1.fit(X, y)  
tree\_reg2.fit(X, y)  
  
def plot\_regression\_predictions(tree\_reg, X, y, axes=(0, 1, -0.2, 1), ylabel="$y$"):  
 # 因为默认参数在函数定义时只被计算一次，使用元组避免axes被函数内部改变。  
 # 当默认参数是可变对象(比如列表或字典)时，如果在函数中修改了这个对象，那么这个修改会在函数的后续调用中被保留。  
 x1 = np.linspace(axes[0], axes[1], 500).reshape(-1, 1)  
 y\_pred = tree\_reg.predict(x1)  
 plt.axis(axes)  
 plt.xlabel("$x\_1$", fontsize=16)  
 plt.ylabel(ylabel, fontsize=16, rotation=0)  
 plt.plot(X, y, "b.")  
 plt.plot(x1, y\_pred, "r.-", linewidth=2, label=r"$\hat{y}$")  
  
plt.figure(figsize=(11, 4))  
plt.subplot(121)  
plot\_regression\_predictions(tree\_reg1, X, y)  
plt.title("max\_depth=2", fontsize=14)  
plt.subplot(122)  
plot\_regression\_predictions(tree\_reg2, X, y)  
plt.title("max\_depth=3", fontsize=14)  
plt.show()



**KNN**

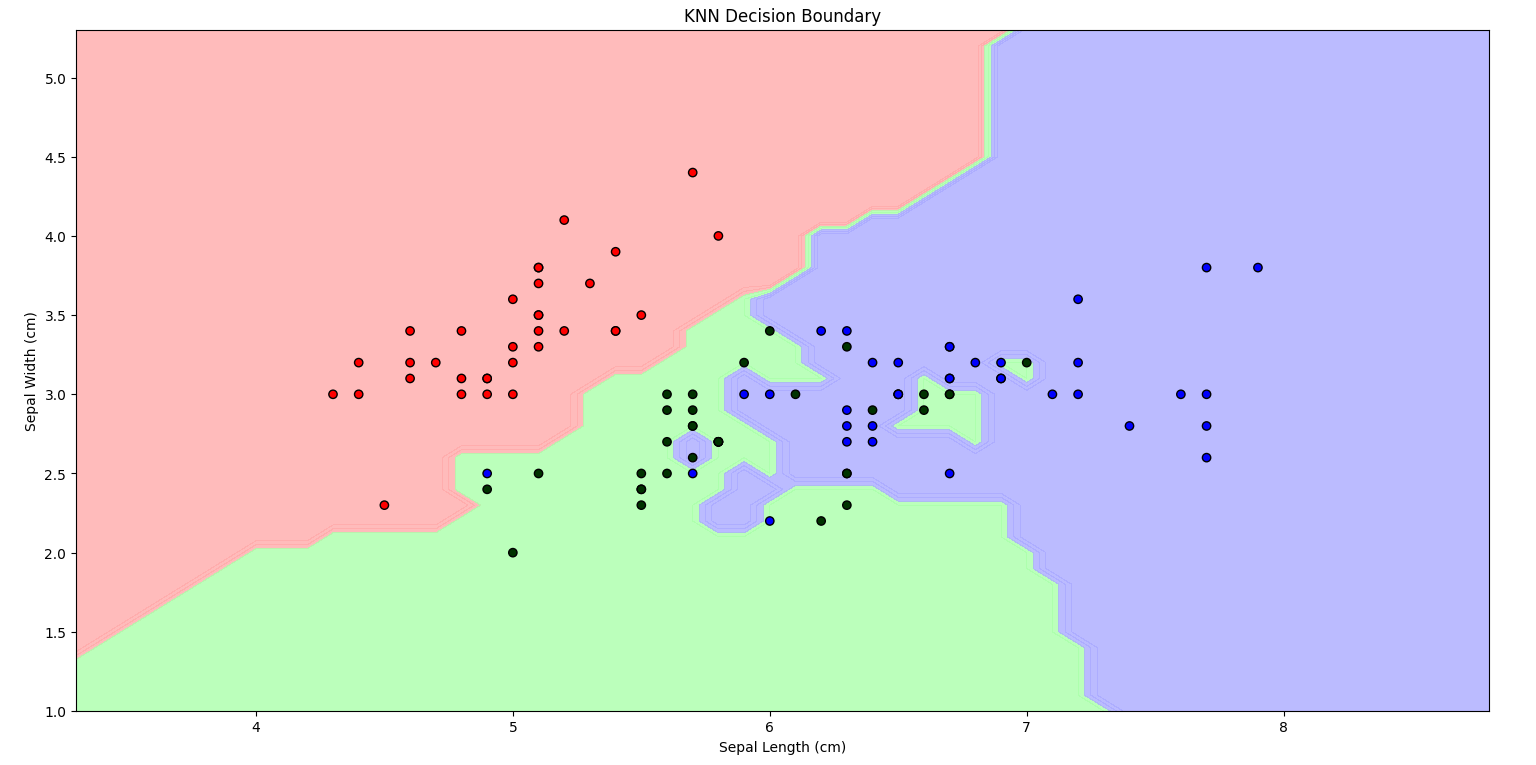
**距离度量-欧氏距离的计算**

import numpy as np  
  
vector1 = np.mat([1, 2, 3]) # [[1 2 3]]  
vector2 = np.mat([4, 5, 6]) # [[4 5 6]]  
distance = np.sqrt((vector1-vector2) \* (vector1-vector2).T) # [[-3 -3 -3]] \* [[-3] [-3] [-3]] = [[5.19615242]]  
print(distance)

**KNN-模型拟合、样本预测、二维绘图**

from sklearn import datasets  
from sklearn import neighbors  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.colors import ListedColormap  
  
Iris = datasets.load\_iris() # 导入鸢尾花数据集  
X = Iris.data # 获得样本特征向量  
Y = Iris.target # 获得样本label  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=0)  
print("target的名称:%s" % Iris.target\_names)  
  
KNN = neighbors.KNeighborsClassifier(n\_neighbors=3)  
KNN.fit(x\_train, y\_train)  
print("KNN模型的训练集准确率:%.3f" % KNN.score(x\_train, y\_train))  
print("KNN模型的测试集准确率:%.3f" % KNN.score(x\_test, y\_test))  
y\_hat = KNN.predict(x\_test)  
print(classification\_report(y\_test, y\_hat, target\_names=Iris.target\_names))  
  
new\_sample = np.array([5.1, 3.5, 1.4, 0.2]).reshape(1, -1) # 新样本的特征值(其实是Iris数据集的第一个数据)  
predicted\_class = KNN.predict(new\_sample) # 运行后得知predicted\_class的值为[0]，因此下一行代码里的predicted\_class[0]就是0。  
predicted\_class\_name = Iris.target\_names[predicted\_class[0]]  
print("样本[5.1, 3.5, 1.4, 0.2]预测的类别是:", predicted\_class\_name)  
  
# Create color maps  
cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF']) # 给不同区域赋以颜色  
cmap\_bold = ListedColormap(['#FF0000', '#003300', '#0000FF']) # 给不同属性的点赋以颜色  
# 定义绘图边界  
feature1\_min, feature1\_max = x\_train[:, 0].min() - 1, x\_train[:, 0].max() + 1 # 第一个属性sepal length花萼长度  
feature2\_min, feature2\_max = x\_train[:, 1].min() - 1, x\_train[:, 1].max() + 1 # 第二个属性sepal width花萼宽度  
xx, yy = np.meshgrid(np.arange(feature1\_min, feature1\_max, 0.1), np.arange(feature2\_min, feature2\_max, 0.1)) # 生成网格点  
KNN = neighbors.KNeighborsClassifier(n\_neighbors=3)  
KNN.fit(x\_train[:, :2], y\_train) # 以二维重新预测，属性变为原先训练集的一二个属性，target不变  
print("二维KNN模型的训练集准确率:%.3f" % KNN.score(x\_train[:, :2], y\_train))  
print("二维KNN模型的测试集准确率:%.3f" % KNN.score(x\_test[:, :2], y\_test))  
# 将预测的结果在平面坐标中画出其类别区域  
Z = KNN.predict(np.c\_[xx.ravel(), yy.ravel()])  
Z = Z.reshape(xx.shape)  
# 绘制决策边界和训练样本  
plt.contourf(xx, yy, Z, cmap=cmap\_light, alpha=0.8)  
plt.scatter(x\_train[:, 0], x\_train[:, 1], c=y\_train, marker='o', edgecolors='k', cmap=cmap\_bold)  
plt.xlim(xx.min(), xx.max())  
plt.ylim(yy.min(), yy.max())  
plt.xlabel('Sepal Length (cm)')  
plt.ylabel('Sepal Width (cm)')  
plt.title('KNN Decision Boundary')  
plt.show()

输出



target的名称:['setosa' 'versicolor' 'virginica']

KNN模型的训练集准确率:0.962

KNN模型的测试集准确率:0.978

precision recall f1-score support

setosa 1.00 1.00 1.00 16

versicolor 1.00 0.94 0.97 18

virginica 0.92 1.00 0.96 11

accuracy 0.98 45

macro avg 0.97 0.98 0.98 45

weighted avg 0.98 0.98 0.98 45

样本[5.1, 3.5, 1.4, 0.2]预测的类别是: setosa

二维KNN模型的训练集准确率:0.857

二维KNN模型的测试集准确率:0.733

试验可知n\_neighbor取5的时候比3在train上好一些。

绘制KNN边界的时候有一个问题，边界应当是连续的，但数据集是非常离散的。因此需要使用一个间隔(这里是0.1，也可以取得更小，但运行时间会显著上升)来近似连续。但是对于不存在于数据集的点，如何我们选定的二维空间对应到四维空间里呢？或者说，对于给定的某个点(x,y)，已知的只有花萼长宽，没有花瓣长宽，缺失的数据没办法填补。并且，花瓣长宽不完全取决于花萼长宽，因此可能相同的花萼长宽对应不同的花瓣长宽时，会使得花的target发生变化。所以在二维图像里没办法给出全部信息，使得KNN只能重新拟合二维，因为剩下两维没办法确定。

[补充：如何可视化4个特征？见Visualize-ML\_Book3\_Ch21.2 成对特征散点图]

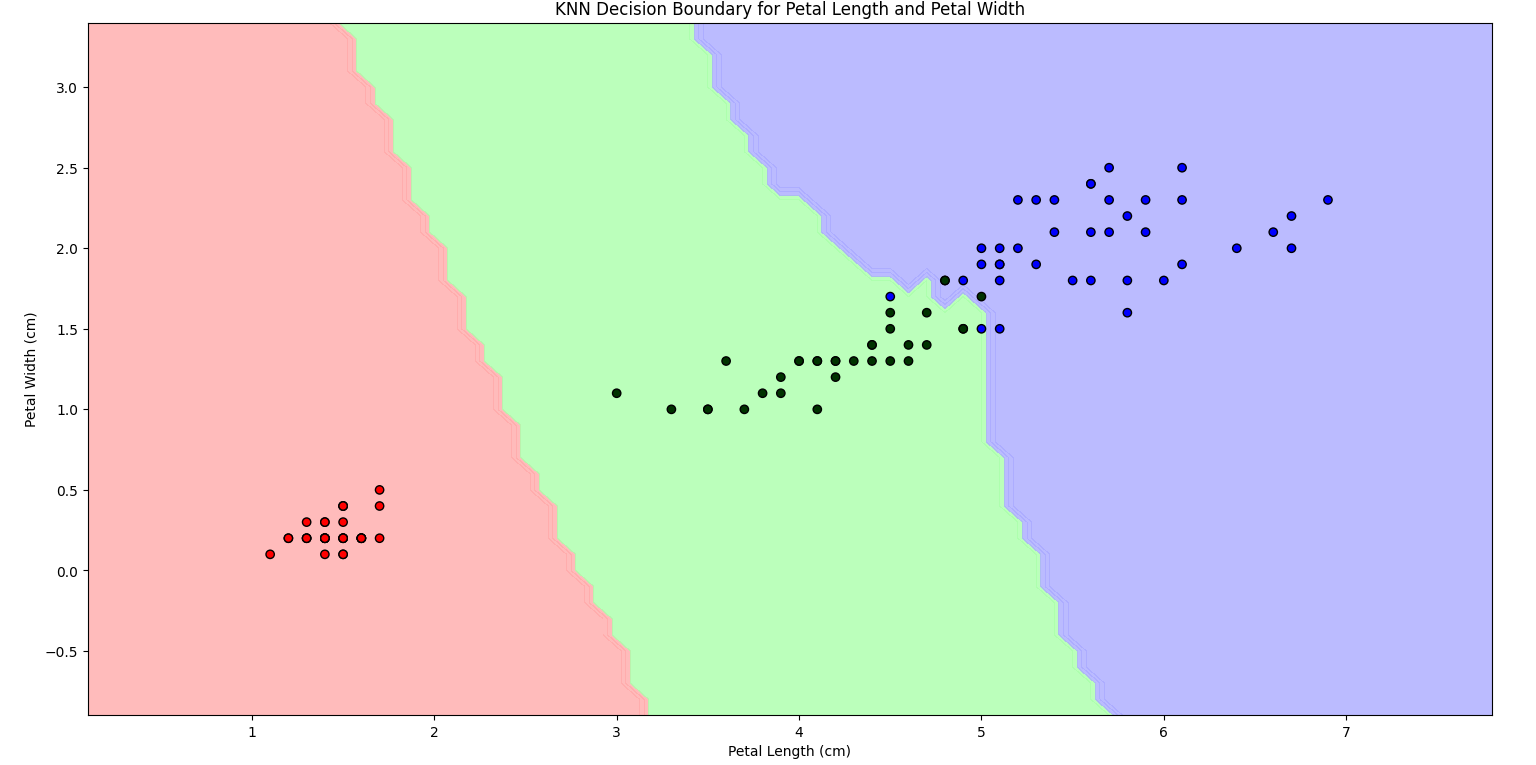
可以看到使用花萼属性的拟合其实不好，但使用花瓣拟合就很不错（可以预先查看散点图，就能发现花瓣很明显有更好的分类效果）：

feature3\_min, feature3\_max = x\_train[:, 2].min() - 1, x\_train[:, 2].max() + 1 # 第三个属性petal length花瓣长度  
feature4\_min, feature4\_max = x\_train[:, 3].min() - 1, x\_train[:, 3].max() + 1 # 第四个属性petal width花瓣宽度  
xx, yy = np.meshgrid(np.arange(feature3\_min, feature3\_max, 0.1), np.arange(feature4\_min, feature4\_max, 0.1))  
KNN = neighbors.KNeighborsClassifier(n\_neighbors=3)  
KNN.fit(x\_train[:, 2:4], y\_train)  
print("后两个属性的KNN模型的训练集准确率:%.3f" % KNN.score(x\_train[:, 2:4], y\_train))  
print("后两个属性的KNN模型的测试集准确率:%.3f" % KNN.score(x\_test[:, 2:4], y\_test))  
Z = KNN.predict(np.c\_[xx.ravel(), yy.ravel()])  
Z = Z.reshape(xx.shape)  
plt.contourf(xx, yy, Z, cmap=cmap\_light, alpha=0.8)  
plt.scatter(x\_train[:, 2], x\_train[:, 3], c=y\_train, marker='o', edgecolors='k', cmap=cmap\_bold)  
plt.xlim(xx.min(), xx.max())  
plt.ylim(yy.min(), yy.max())  
plt.xlabel('Petal Length (cm)')  
plt.ylabel('Petal Width (cm)')  
plt.title('KNN Decision Boundary for Petal Length and Petal Width')  
plt.show()

输出

后两个属性的KNN模型的训练集准确率:0.962

后两个属性的KNN模型的测试集准确率:0.978



另外，KNN要求所有的属性必须经过标准化处理，以免属性之间的数量级相差过大而导致偏差。KNN可以自动忽略缺失值。对异常数据不敏感，具有较好的抗噪性。

from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler() # 创建一个Z-Score标准化处理器  
x\_train\_scaled = scaler.fit\_transform(x\_train) # 对训练数据进行标准化处理  
x\_test\_scaled = scaler.transform(x\_test) # 对测试数据进行相同的标准化处理

其余与之前的基本一致。

**朴素贝叶斯**

from sklearn import datasets  
from sklearn import naive\_bayes  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
  
Iris = datasets.load\_iris()  
X = Iris.data  
Y = Iris.target  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=0)  
  
bayes = naive\_bayes.GaussianNB()  
bayes.fit(x\_train, y\_train)  
print("贝叶斯模型训练集的准确率:%.3f" % bayes.score(x\_train, y\_train))  
print("贝叶斯模型测试集的准确率:%.3f" % bayes.score(x\_test, y\_test))  
y\_hat = bayes.predict(x\_test)  
print(classification\_report(y\_test, y\_hat, target\_names=Iris.target\_names))

输出

贝叶斯模型训练集的准确率:0.943

贝叶斯模型测试集的准确率:1.000

precision recall f1-score support

setosa 1.00 1.00 1.00 16

versicolor 1.00 1.00 1.00 18

virginica 1.00 1.00 1.00 11

accuracy 1.00 45

macro avg 1.00 1.00 1.00 45

weighted avg 1.00 1.00 1.00 45

**SVM**

from sklearn import datasets  
from sklearn import neighbors  
from sklearn import svm  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
  
Iris = datasets.load\_iris()  
X = Iris.data  
Y = Iris.target  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=0)  
  
SVM = svm.SVC() # Support Vector Classification是用于分类问题的SVM实现  
SVM.fit(x\_train, y\_train)  
print("SVM模型训练集的准确率:%.3f" % SVM.score(x\_train, y\_train))  
print("SVM模型测试集的准确率:%.3f" % SVM.score(x\_test, y\_test))  
y\_hat = SVM.predict(x\_test)  
print(classification\_report(y\_test, y\_hat, target\_names=Iris.target\_names))

输出

SVM模型训练集的准确率:0.971

SVM模型测试集的准确率:0.978

precision recall f1-score support

setosa 1.00 1.00 1.00 16

versicolor 1.00 0.94 0.97 18

virginica 0.92 1.00 0.96 11

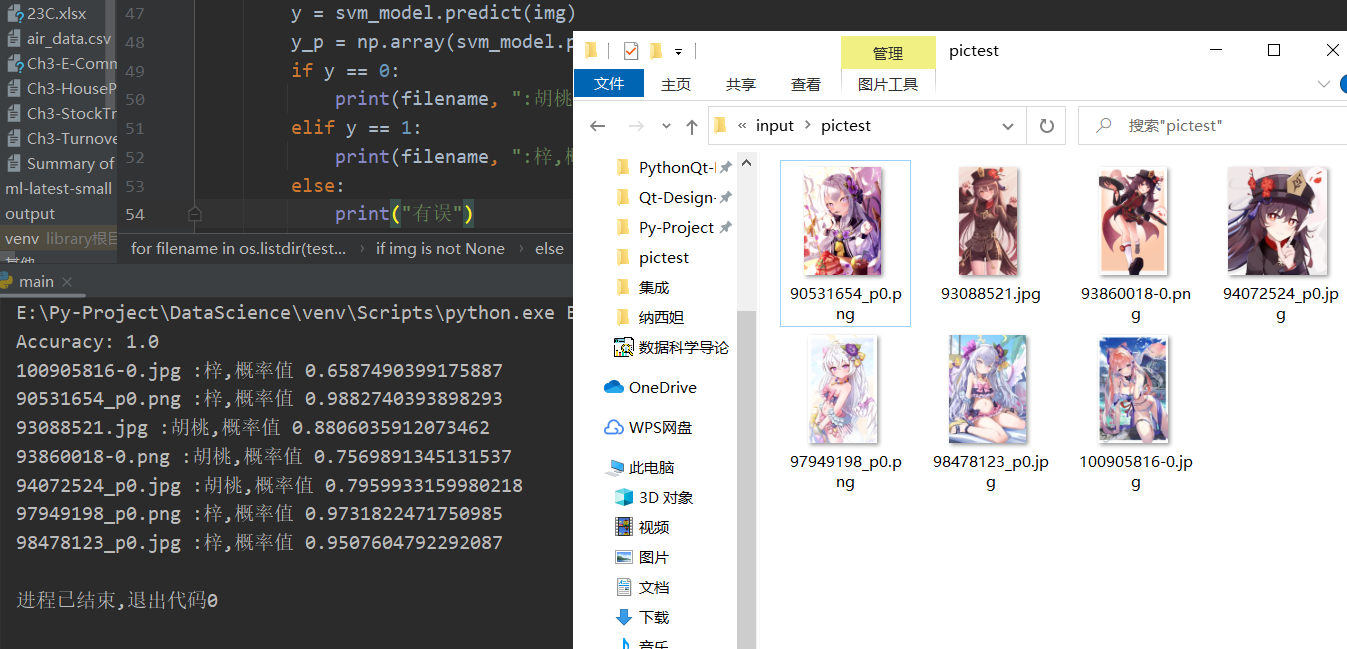
accuracy 0.98 45

macro avg 0.97 0.98 0.98 45

weighted avg 0.98 0.98 0.98 45

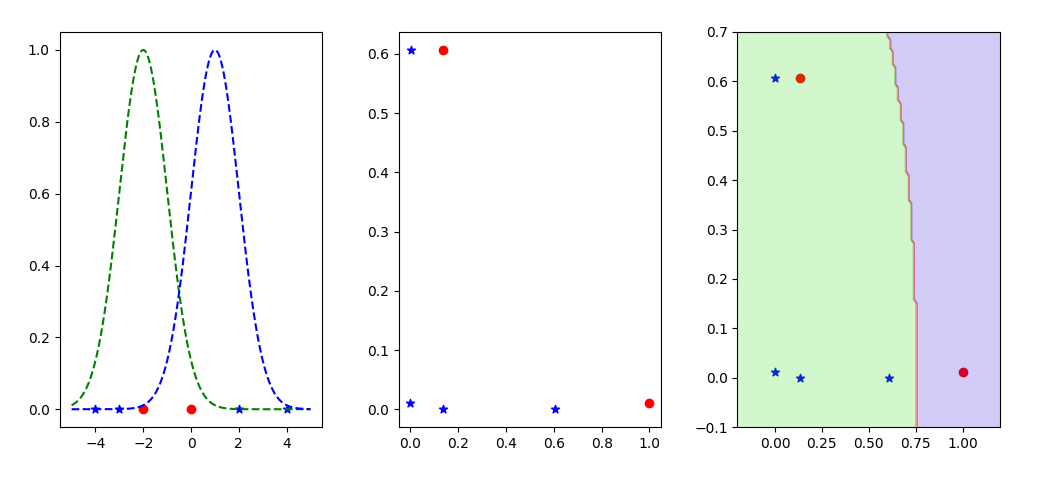
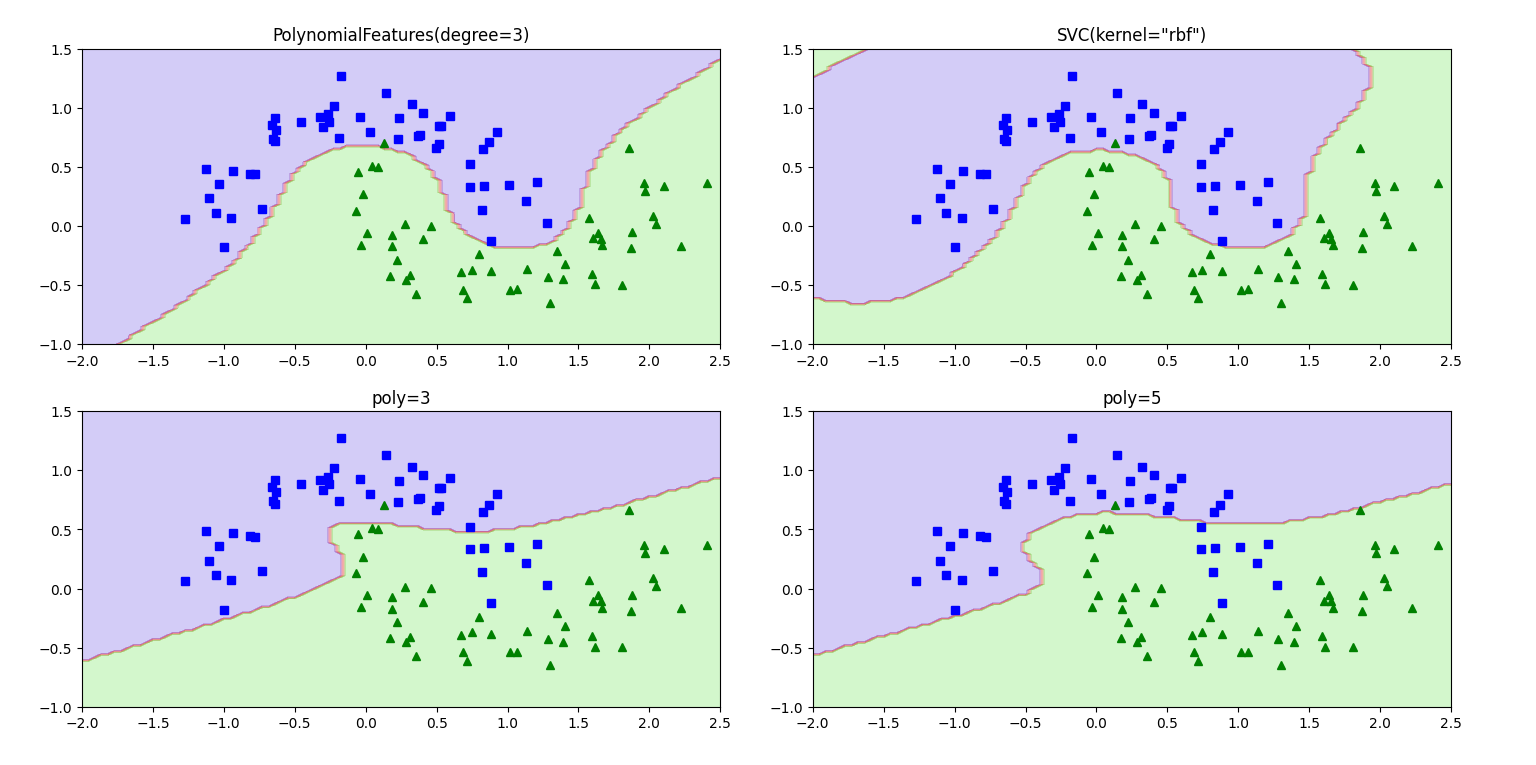
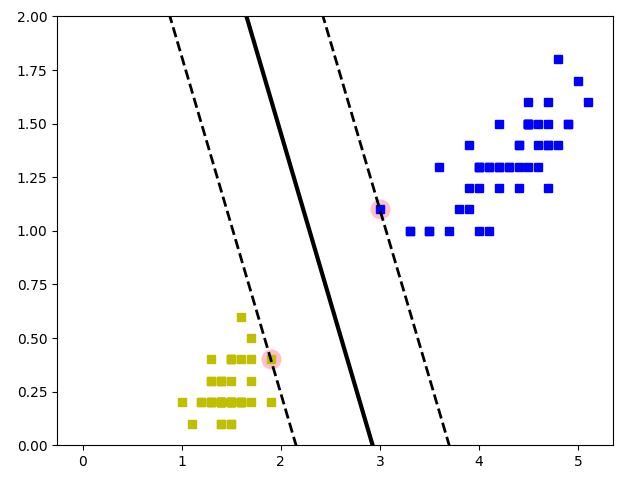
**SVM实现图片分类**

import cv2  
import os  
from sklearn import svm  
from sklearn.metrics import accuracy\_score  
from sklearn.model\_selection import train\_test\_split  
import numpy as np  
  
def load\_images\_from\_folder(folder):  
 images = []  
 for filename in os.listdir(folder):  
 img\_path = os.path.join(folder, filename)  
 img = cv2.imread(img\_path)  
 if img is not None:  
 img = cv2.resize(img, (700, 1000))  
 img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 images.append(img)  
 return images  
  
hutao\_folder = "input/hutao"  
azusa\_folder = "input/azusa"  
test\_folder = "input/pictest"  
  
hutao\_images = np.array(load\_images\_from\_folder(hutao\_folder)) # 大小(12, 1000, 700)  
azusa\_images = np.array(load\_images\_from\_folder(azusa\_folder))  
hutao\_images = hutao\_images.reshape((len(hutao\_images), -1)) # 大小(12, 700000)  
azusa\_images = azusa\_images.reshape((len(azusa\_images), -1))  
hutao\_labels = np.zeros(len(hutao\_images)) # 标签0:胡桃，大小(12,)  
azusa\_labels = np.ones(len(azusa\_images)) # 标签1:梓  
X = np.concatenate((hutao\_images, azusa\_images), axis=0) # 大小(24, 700000)  
y = np.concatenate((hutao\_labels, azusa\_labels), axis=0) # 大小(24,)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
svm\_model = svm.SVC(kernel='rbf', random\_state=42, probability=True)  
svm\_model.fit(X\_train, y\_train)  
y\_pred = svm\_model.predict(X\_test)  
accuracy = accuracy\_score(y\_test, y\_pred)  
print("Accuracy:", accuracy)  
  
for filename in os.listdir(test\_folder):  
 img\_path = os.path.join(test\_folder, filename)  
 img = cv2.imread(img\_path)  
 if img is not None:  
 img = cv2.resize(img, (700, 1000))  
 img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  
 img = np.array(img).reshape(1, -1)  
 y = svm\_model.predict(img)  
 y\_p = np.array(svm\_model.predict\_proba(img))  
 if y == 0:  
 print(filename, ":胡桃,概率值", str(y\_p[0][0]))  
 elif y == 1:  
 print(filename, ":梓,概率值", str(y\_p[0][1]))  
 else:  
 print("有误")



**线性/非线性/RBF的SVM**

import matplotlib.pyplot as plt  
import numpy as np  
from sklearn.svm import SVC  
from sklearn.datasets import load\_iris, make\_moons  
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import PolynomialFeatures,StandardScaler  
  
**# 线性**  
Iris = load\_iris()  
X = Iris['data'][:, (2, 3)]  
y = Iris['target']  
setosa\_and\_versicolor = (y == 0) | (y == 1)  
X = X[setosa\_and\_versicolor]  
y = y[setosa\_and\_versicolor]  
svm\_clf = SVC(kernel='linear', C=100)  
svm\_clf.fit(X, y)  
  
def draw\_decision\_boundary(clf, xlim=(0,5)):  
 w = clf.coef\_  
 b = clf.intercept\_  
 x0 = np.linspace(xlim[0], xlim[1], 100)  
 decision\_boundary = -(x0 \* w[0][0] + b) / w[0][1] # 因为预测x0w0+x1w1+b=0，因此x1=-(x0w0+b)/w1  
 up = decision\_boundary + 1 / w[0][1] # 因为间隔d是2/||w||，因此margin就是1/||w||  
 down = decision\_boundary - 1 / w[0][1]  
 print(w[0], b) # w是[[w0,w1...]]这样的数据，因此先要执行w[0]  
 plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs')  
 plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'ys')  
 plt.ylim(0, 2)  
 plt.plot(x0, decision\_boundary, 'k-', linewidth=3)  
 plt.plot(x0, up, 'k--', linewidth=2)  
 plt.plot(x0, down, 'k--', linewidth=2)  
 plt.scatter(clf.support\_vectors\_[:, 0], clf.support\_vectors\_[:, 1], s=180, facecolors='#FFC0CB')  
  
plt.figure()  
draw\_decision\_boundary(svm\_clf)  
plt.show()  
  
**# 非线性**  
X, y = make\_moons(n\_samples=100, noise=0.15, random\_state=42)  
poly\_svm\_clf = Pipeline([  
 ('poly', PolynomialFeatures(degree=3)),  
 ('scaler', StandardScaler()),  
 ('svm\_clf', SVC(kernel='linear', C=100))  
])  
rbf\_svm\_clf = Pipeline([  
 ('scaler', StandardScaler()),  
 ('svm\_clf', SVC(kernel='rbf', C=100))  
])  
poly3\_svm\_clf = Pipeline([  
 ('scaler', StandardScaler()),  
 ('svm\_clf', SVC(kernel='poly', degree=3, C=100))  
])  
poly5\_svm\_clf = Pipeline([  
 ('scaler', StandardScaler()),  
 ('svm\_clf', SVC(kernel='poly', degree=5, C=100))  
])  
poly\_svm\_clf.fit(X, y)  
rbf\_svm\_clf.fit(X, y)  
poly3\_svm\_clf.fit(X, y)  
poly5\_svm\_clf.fit(X, y)  
  
def draw\_decision\_boundary(clf, axes=(-2, 2.5, -1, 1.5)):  
 x0s = np.linspace(axes[0], axes[1], 100)  
 x1s = np.linspace(axes[2], axes[3], 100)  
 x0, x1 = np.meshgrid(x0s, x1s)  
 x = np.c\_[x0.ravel(), x1.ravel()]  
 y\_pred = clf.predict(x).reshape(x0.shape)  
 plt.contourf(x0, x1, y\_pred, cmap='brg', alpha=0.2)  
 plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'bs')  
 plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'g^')  
  
plt.subplot(221)  
plt.title('PolynomialFeatures(degree=3)')  
draw\_decision\_boundary(clf=poly\_svm\_clf)  
plt.subplot(222)  
plt.title('SVC(kernel="rbf")')  
draw\_decision\_boundary(clf=rbf\_svm\_clf)  
plt.subplot(223)  
plt.title('poly=3')  
draw\_decision\_boundary(clf=poly3\_svm\_clf)  
plt.subplot(224)  
plt.title('poly=5')  
draw\_decision\_boundary(clf=poly5\_svm\_clf)  
plt.show()  
  
**# rbf**  
def RBF\_cal(x, gamma=0.5, landmark=0):  
 *"""计算高斯核函数，传入列向量，这里默认γ=0.5"""* return np.exp(-gamma \* np.linalg.norm(x - landmark, axis=1) \*\* 2) # 按行计算范数(欧氏距离)  
  
X = np.array((-4, -3, -2, 0, 2, 4)).reshape(-1, 1)  
y = np.array((0, 0, 1, 1, 0, 0))  
  
x1 = np.linspace(-5, 5, 1000).reshape(-1, 1)  
x2 = RBF\_cal(x1, landmark=-2)  
x3 = RBF\_cal(x1, landmark=1)  
X\_rbf = np.c\_[RBF\_cal(X, landmark=-2), RBF\_cal(X, landmark=1)]  
  
X\_SVC = np.array(((-4, 0), (-3, 0), (-2, 0), (0, 0), (2, 0), (4, 0)))  
X\_SVC\_rbf = SVC(kernel='rbf', gamma=0.5)  
X\_SVC\_rbf.fit(X\_SVC, y)  
  
plt.subplot(131)  
plt.scatter(X\_SVC[:, 0][y == 0], X\_SVC[:, 1][y == 0], c='blue', marker="\*")  
plt.scatter(X\_SVC[:, 0][y == 1], X\_SVC[:, 1][y == 1], c='red', marker="o")  
plt.plot(x1, x2, 'g--')  
plt.plot(x1, x3, 'b--')  
plt.subplot(132)  
plt.scatter(X\_rbf[:, 0][y == 0], X\_rbf[:, 1][y == 0], c='blue', marker="\*")  
plt.scatter(X\_rbf[:, 0][y == 1], X\_rbf[:, 1][y == 1], c='red', marker="o")  
plt.subplot(133)  
x0s = np.linspace(-0.2, 1.2, 100)  
x1s = np.linspace(-0.1, 0.7, 100)  
x0, x1 = np.meshgrid(x0s, x1s)  
x = np.c\_[x0.ravel(), x1.ravel()]  
y\_pred = X\_SVC\_rbf.predict(x).reshape(x0.shape)  
plt.scatter(X\_rbf[:, 0][y == 0], X\_rbf[:, 1][y == 0], c='blue', marker="\*")  
plt.scatter(X\_rbf[:, 0][y == 1], X\_rbf[:, 1][y == 1], c='red', marker="o")  
plt.contourf(x0, x1, y\_pred, cmap='brg', alpha=0.2)  
plt.show()

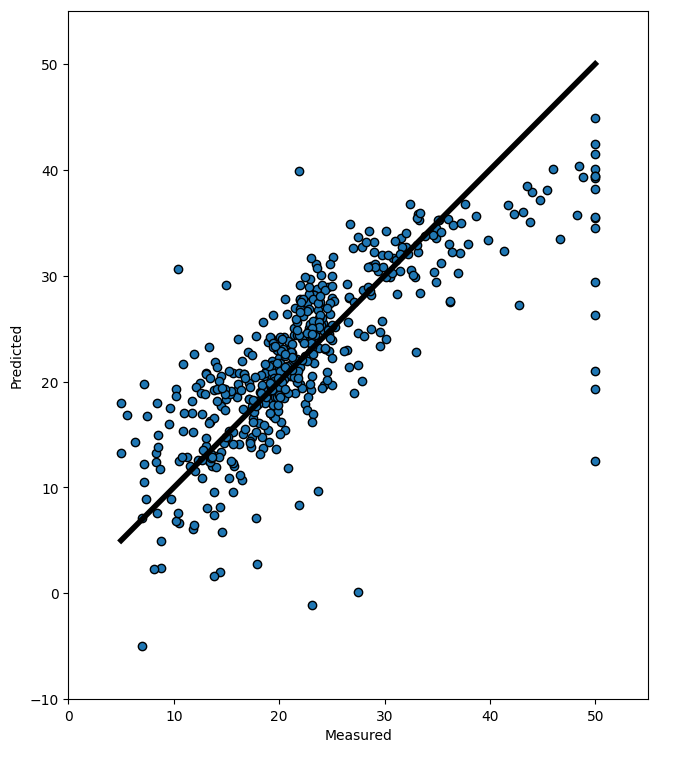


**8.回归**

**线性回归**

from sklearn.model\_selection import cross\_val\_predict  
from sklearn import linear\_model  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
  
data\_url = "http://lib.stat.cmu.edu/datasets/boston"  
raw\_df = pd.read\_csv(data\_url, sep=r"\s+", skiprows=22, header=None)  
data = np.hstack([raw\_df.values[::2, :], raw\_df.values[1::2, :2]]) # 属性  
target = raw\_df.values[1::2, 2] # 预测值  
print(data.shape, target.shape) # 规格(506, 13) (506,)  
  
Ir = linear\_model.LinearRegression() # 线性回归  
# 使用交叉验证。将数据分成10折(由cv=10指定)，每次使用9折数据训练模型，然后用模型预测第10折数据，并将预测结果保存下来。重复10次。  
predicted = cross\_val\_predict(Ir, data, target, cv=10)  
  
# 查看 target 的最小和最大值以便绘图  
print("Target - Min:", min(target)) # 5.0  
print("Target - Max:", max(target)) # 50.0  
# 查看 predicted 的最小和最大值  
print("Predicted - Min:", min(predicted)) # -4.9536551238824345  
print("Predicted - Max:", max(predicted)) # 44.912287884174994  
  
fig, ax = plt.subplots()  
ax.scatter(target, predicted, edgecolors=(0, 0, 0))  
# 绘制线条（从(5,5)到(50,50)）。这条直线表示理想情况下，如果模型的预测完全准确，所有的点都应该落在这条直线上。  
ax.plot([target.min(), target.max()], [target.min(), target.max()], 'k-', lw=4) # k:黑色，-:实线，lw:line width。  
ax.set\_xlim(0, 55) # 设置 x 轴范围  
ax.set\_ylim(-10, 55) # 设置 y 轴范围  
ax.set\_xlabel('Measured')  
ax.set\_ylabel('Predicted')  
plt.show() # 不要问我为什么target有那么多50的数据点，我也很奇怪啊

输出



(506, 13) (506,)

Target - Min: 5.0

Target - Max: 50.0

Predicted - Min: -4.9536551238824345

Predicted - Max: 44.912287884174994

右图，横轴是准确值，纵轴是预测值。直线代表预测准确的点。预测值在直线之上/下代表预测过高/低。

**线性回归的三种梯度下降方法**

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.linear\_model import LinearRegression  
  
np.random.seed(42) # 设置固定种子便于实验  
  
  
**""" ---线性回归--- """  
""" 生成数据集 & 绘图 """**  
X = 2 \* np.random.rand(727, 1) # 727个[0, 1)之间均匀分布的随机数 \* 2  
y = 4 + 3 \* X + np.random.randn(727, 1) # 4 + 3x + 标准正态分布误差项  
plt.plot(X, y, 'b.')  
plt.axis([0, 2, 1, 14])  
plt.show()  
  
**""" 直接使用公式计算 & 绘图 """**  
X\_b = np.c\_[np.ones((727, 1)), X] # 拼接数据，形如[[1. 0.32950847] [1. 0.62878135]...[1. 0.53402584]]  
theta\_best = np.linalg.inv(X\_b.T.dot(X\_b)).dot(X\_b.T).dot(y) # (X^T \* X)^(-1) \* X^T \* y  
print("公式计算的θ：", theta\_best) # [[4.02650858] [2.96474178]]很接近我们希望的系数4和3  
X\_HeadAndTail = np.array([[0], [2]]) # [[0] [2]]  
X\_HeadAndTail\_b = np.c\_[np.ones((2, 1)), X\_HeadAndTail] # [[1. 0.] [1. 2.]]  
y\_predicted = X\_HeadAndTail\_b.dot(theta\_best) # [[4.02650858] [9.95599214]]  
plt.plot(X\_HeadAndTail, y\_predicted, 'r--')  
plt.plot(X, y, 'b.')  
plt.axis([0, 2, 1, 14])  
plt.show()  
  
**""" LinearRegression """**  
lin\_reg = LinearRegression()  
lin\_reg.fit(X, y)  
print("偏置参数：", lin\_reg.intercept\_) # 偏置参数 [4.02650858]  
print("权重参数：", lin\_reg.coef\_) # 权重参数 [[2.96474178]]  
  
**""" 使用MSE的批量梯度下降 """**  
eta = 0.01 # 学习率learning rate  
n\_iterations = 10000 # 迭代次数  
n = len(X\_b) # 样本数量  
theta = np.random.randn(2, 1) # 形状为(2,1)的数组  
for iteration in range(n\_iterations):  
 gradients = 2 / n \* X\_b.T.dot(X\_b.dot(theta) - y) # 梯度(使用均方误差损失函数MSE)  
 theta = theta - eta \* gradients # 参数更新  
print("MSE批量梯度下降迭代10000次的θ：", theta)  
  
**""" 三种梯度下降的对比"""**  
**# 批量梯度下降BGD(学习率对结果的影响)**  
theta\_path\_bgd = [] # bgd：Batch Gradient Descent批量梯度下降  
def plot\_gradient\_descent(theta, eta, store\_bgd = None):  
 n = len(X\_b)  
 plt.plot(X, y, 'b.')  
 n\_iterations = 200 # 迭代次数  
 for iteration in range(n\_iterations):  
 gradients = 2 / n \* X\_b.T.dot(X\_b.dot(theta) - y) # 梯度(使用均方误差损失函数MSE)  
 theta = theta - eta \* gradients # 参数更新  
 if store\_bgd is not None: # 因为BGD有三组对比实验，我们只需要存入一个bgd的值与后续sgd,mgd比较  
 store\_bgd.append(theta) # 通过传入store\_bgd=theta\_path\_bgd实现向theta\_path\_bgd里传值  
 y\_predicted = X\_HeadAndTail\_b.dot(theta)  
 plt.plot(X\_HeadAndTail, y\_predicted, color=(0.1, 1, 0.5, min(8 \* iteration / n\_iterations, 0.2222 \* iteration / n\_iterations + 0.7778))) # color的参数依次是红绿蓝和透明度alpha, alpha是分段函数 y=8\*x(x<=0.1),y=0.2222x+0.7778(x>0.1)  
 # print("x:", iteration / n\_iterations, "\ty:", min(40\*iteration/n\_iterations, 0.204\*iteration/n\_iterations+0.796)) # 查看color中的函数对应的x,y值  
 plt.text(X\_HeadAndTail[-1], y\_predicted[-1], f'{iteration}', color='red') # 在当前迭代的最后一个点上添加文本标签  
 plt.xlabel('X')  
 plt.ylabel('y')  
 plt.axis([0, 2, 1, 14])  
 plt.title('eta={}'.format(eta))  
theta = np.random.randn(2, 1)  
plt.figure(figsize=(10, 4))  
plt.subplot(131)  
plot\_gradient\_descent(theta, eta=0.01, store\_bgd=theta\_path\_bgd)  
plt.subplot(132)  
plot\_gradient\_descent(theta, eta=0.1)  
plt.subplot(133)  
plot\_gradient\_descent(theta, eta=0.3)  
plt.show()  
  
**# 随机梯度下降SGD**  
theta\_path\_sgd = []  
n = len(X\_b) # 数据个数  
n\_epochs = 50 # 迭代的轮数，即整个数据集被遍历的次数  
t0 = 5  
t1 = 50 # t0,t1共同控制学习率的衰减  
theta = np.random.randn(2, 1)  
for epoch in range(n\_epochs):  
 for i in range(n):  
 if epoch < 10 and i < n and i % 20 == 0: # 绘制前十个epoch对应的拟合直线，每个epoch绘制间隔为20的直线(便于显示出明显的间隔)  
 y\_predicted = X\_HeadAndTail\_b.dot(theta)  
 plt.plot(X\_HeadAndTail, y\_predicted, color=(0.1, 1, 0.5, i/n))  
 if i == 0: # 仅当每个epoch的第一个i的时候绘制出epoch的值  
 plt.text(X\_HeadAndTail[-1], y\_predicted[-1], f'{epoch}', color='red')  
 random\_index = np.random.randint(n) # 随机选择一个样本  
 xi = X\_b[random\_index:random\_index + 1]  
 yi = y[random\_index:random\_index + 1]  
 gradients = 2 \* xi.T.dot(xi.dot(theta) - yi)  
 eta = t0 / (t1+(n\_epochs\*n+i)) # 当迭代次数(n\_epochs\*n+i)增加，学习率eta呈降低趋势  
 theta = theta - eta \* gradients  
 theta\_path\_sgd.append(theta)  
plt.plot(X, y, 'b.')  
plt.axis([0, 2, 1, 14])  
plt.show()  
  
**# 小批量梯度下降Mini-batch Gradient Descent**  
theta\_path\_mgd = []  
n\_epochs = 800 # 为什么这里迭代次数远高于上面的才能收敛？因为每次epoch只迭代了n/minibatch次，虽然每次迭代有minibatch个数据样本点，但minibatch个数据样本点真的比1个数据样本点强很多吗？  
minibatch = 16  
theta = np.random.randn(2, 1)  
for epoch in range(n\_epochs):  
 shuffled\_index = np.random.permutation(n) # 样本洗牌  
 X\_b\_shuffled = X\_b[shuffled\_index]  
 y\_shuffled = y[shuffled\_index]  
 for i in range(0, n, minibatch): # 从0开始，以步长minibatch迭代，直到n-1结束。  
 if epoch < 160 and epoch % 10 == 0 and i % (20\*minibatch) == 0: # 只绘制前面的epoch。对每个epoch依然只绘制部分迭代过程  
 y\_predicted = X\_HeadAndTail\_b.dot(theta)  
 plt.plot(X\_HeadAndTail, y\_predicted, color=(0.1, 1, 0.5, i / n))  
 if i == 0: # 仅当每个epoch的第一个i的时候绘制出epoch的值  
 plt.text(X\_HeadAndTail[-1], y\_predicted[-1], f'{epoch}', color='red')  
 xi = X\_b\_shuffled[i: i+minibatch]  
 yi = y\_shuffled[i: i+minibatch]  
 gradients = 2/minibatch \* xi.T.dot(xi.dot(theta) - yi)  
 eta = t0 / (t1 + (n\_epochs\*n/minibatch + i/minibatch))  
 theta = theta - eta \* gradients  
 theta\_path\_mgd.append(theta)  
plt.plot(X, y, 'b.')  
plt.axis([0, 2, 1, 14])  
plt.show()  
  
**# BGD,SGD,MGD三者对比**  
theta\_path\_bgd = np.array(theta\_path\_bgd)  
theta\_path\_sgd = np.array(theta\_path\_sgd)  
theta\_path\_mgd = np.array(theta\_path\_mgd)  
print("BGD:", theta\_path\_bgd[-1], "\nSGD:", theta\_path\_sgd[-1], "\nMGD:", theta\_path\_mgd[-1])  
plt.plot(theta\_path\_bgd[:, 0], theta\_path\_bgd[:, 1], 'b-o', label='BGD')  
plt.plot(theta\_path\_sgd[:, 0], theta\_path\_sgd[:, 1], 'r-s', label='SGD')  
plt.plot(theta\_path\_mgd[:, 0], theta\_path\_mgd[:, 1], 'g-+', label='MGD')  
plt.legend()  
plt.show() # 全局图像  
plt.plot(theta\_path\_bgd[::10, 0], theta\_path\_bgd[::10, 1], 'b-o', label='BGD') # ::n表示以步长为n进行切片  
plt.plot(theta\_path\_sgd[::100, 0], theta\_path\_sgd[::100, 1], 'r-s', label='SGD')  
plt.plot(theta\_path\_mgd[::80, 0], theta\_path\_mgd[::80, 1], 'g-+', label='MGD')  
plt.axis([3.2, 4.5, 2.3, 4.5])  
plt.legend()  
plt.show() # 部分数据点的局部图像  
**# 理论上BGD会是笔直到达。SGD是大范围的浮动,很容易出现偏离。MGD与BGD隔的比较近,在BGD的周围浮动。**

输出

公式计算的θ： [[4.02650858]

[2.96474178]]

偏置参数： [4.02650858]

权重参数： [[2.96474178]]

MSE批量梯度下降迭代10000次的θ： [[4.02650858]

[2.96474178]]

BGD: [[3.66635721]

[3.2697089 ]]

SGD: [[3.93608917]

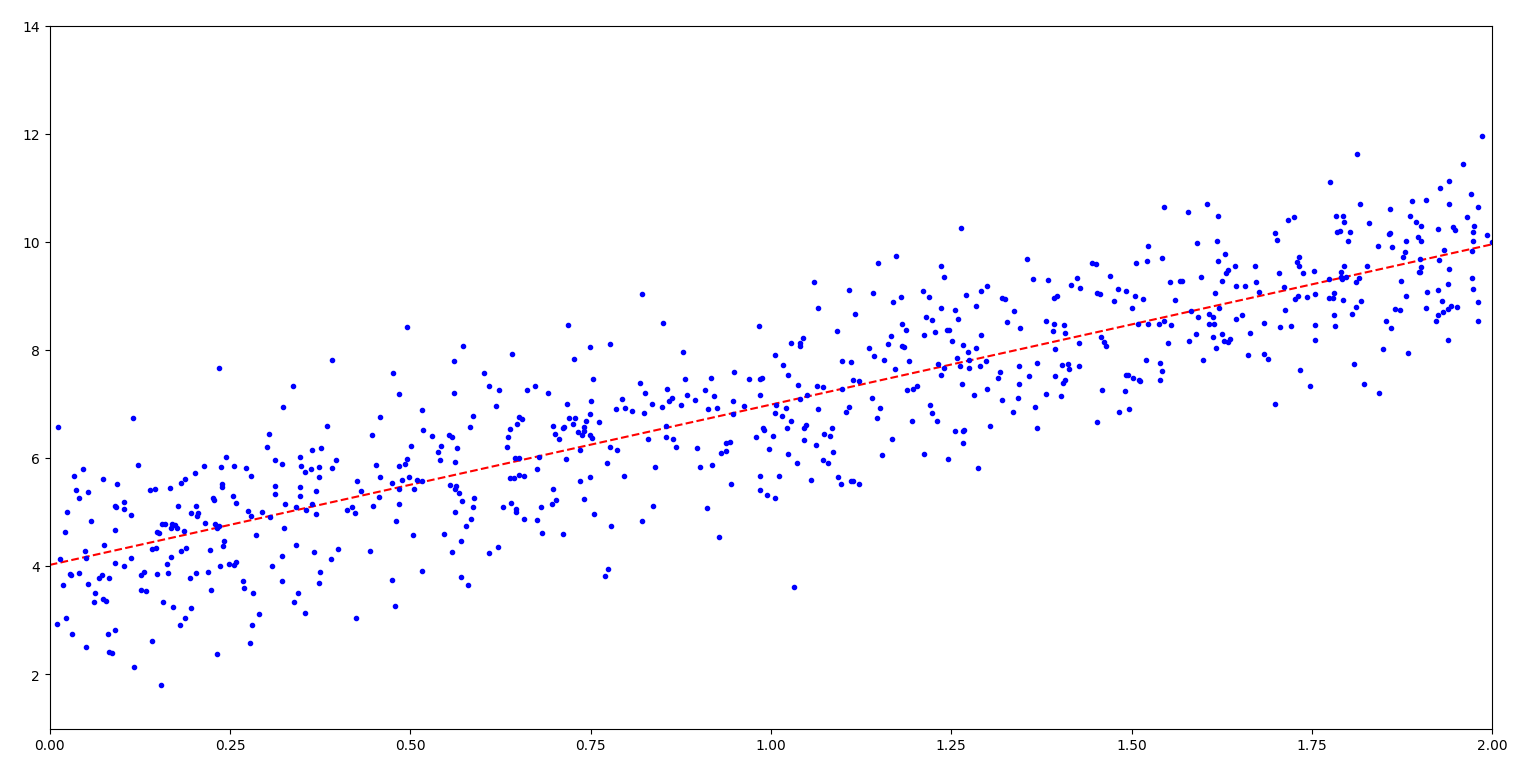
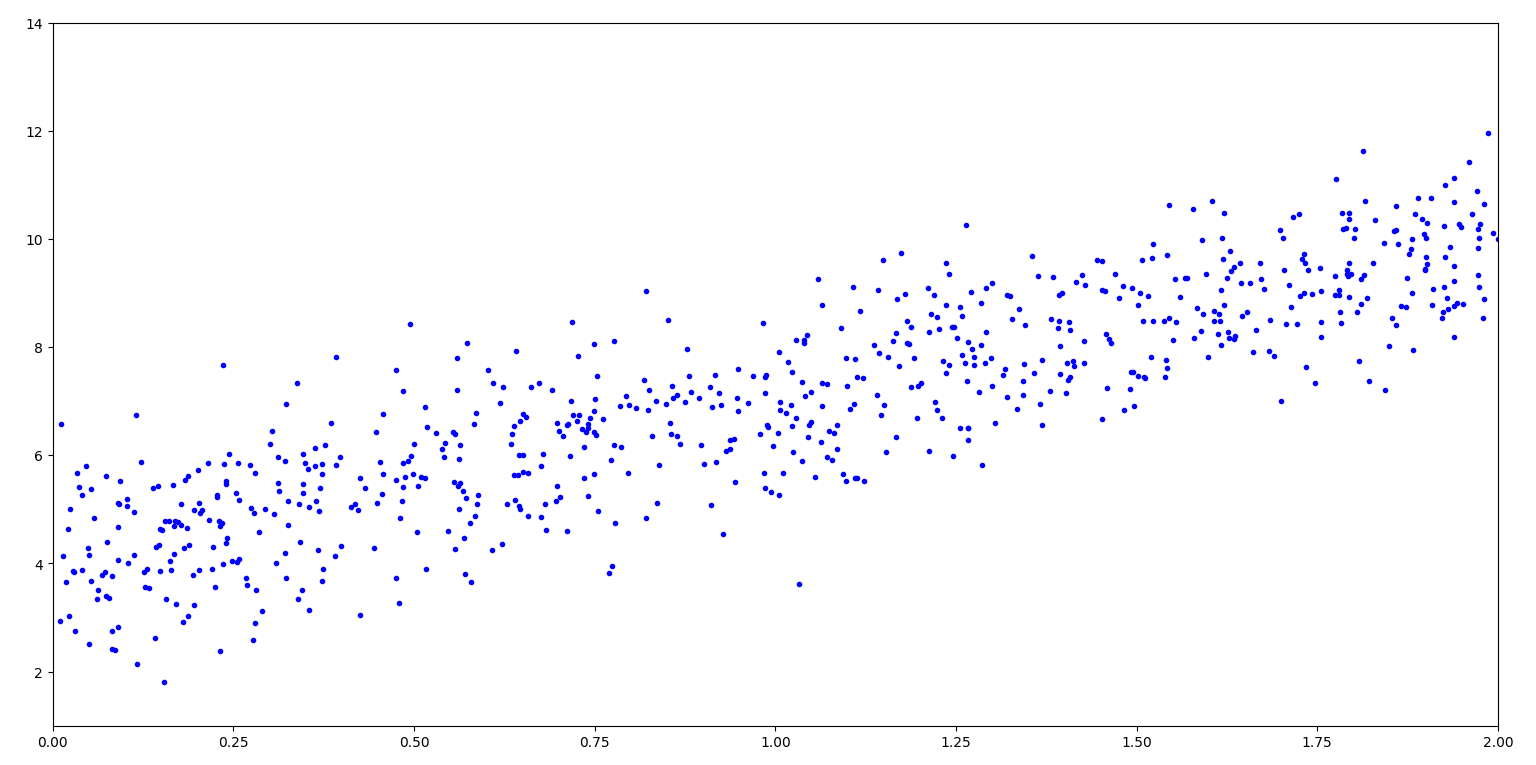
[3.03300464]]

MGD: [[4.09271048]

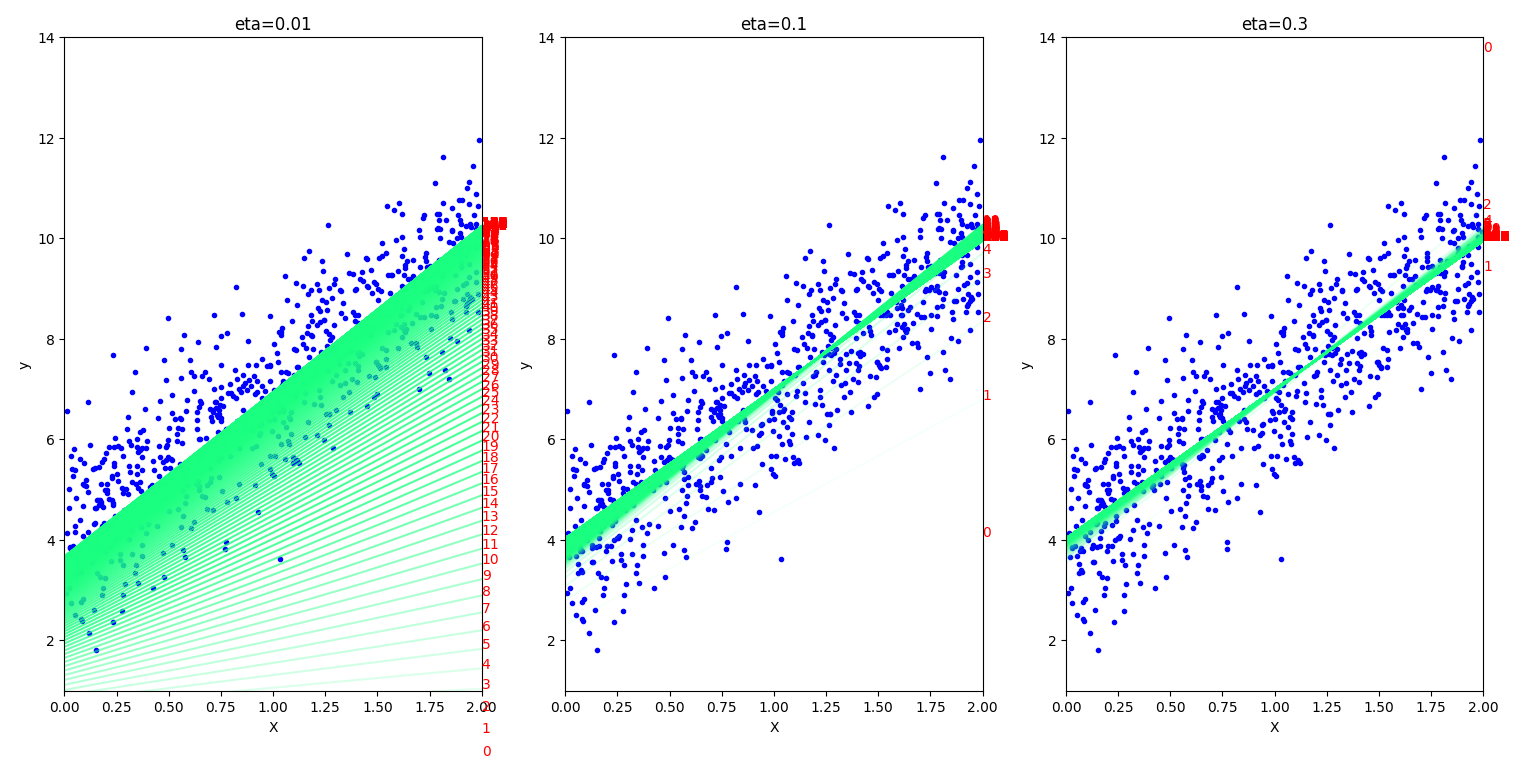
[2.90851744]]

图片输出如下：

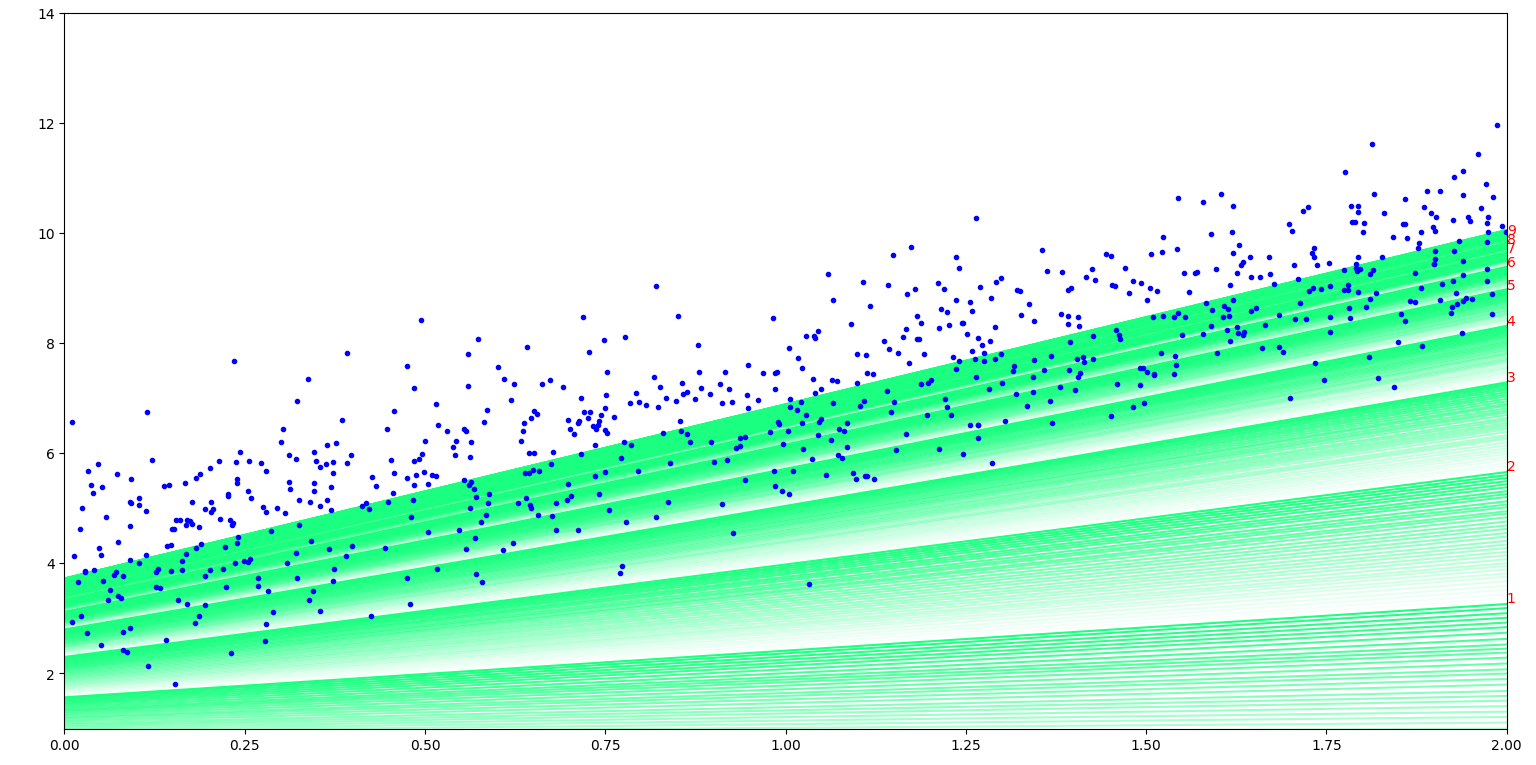
原始数据 ＆ LR拟合



MSE-eta不同导致的结果不同：



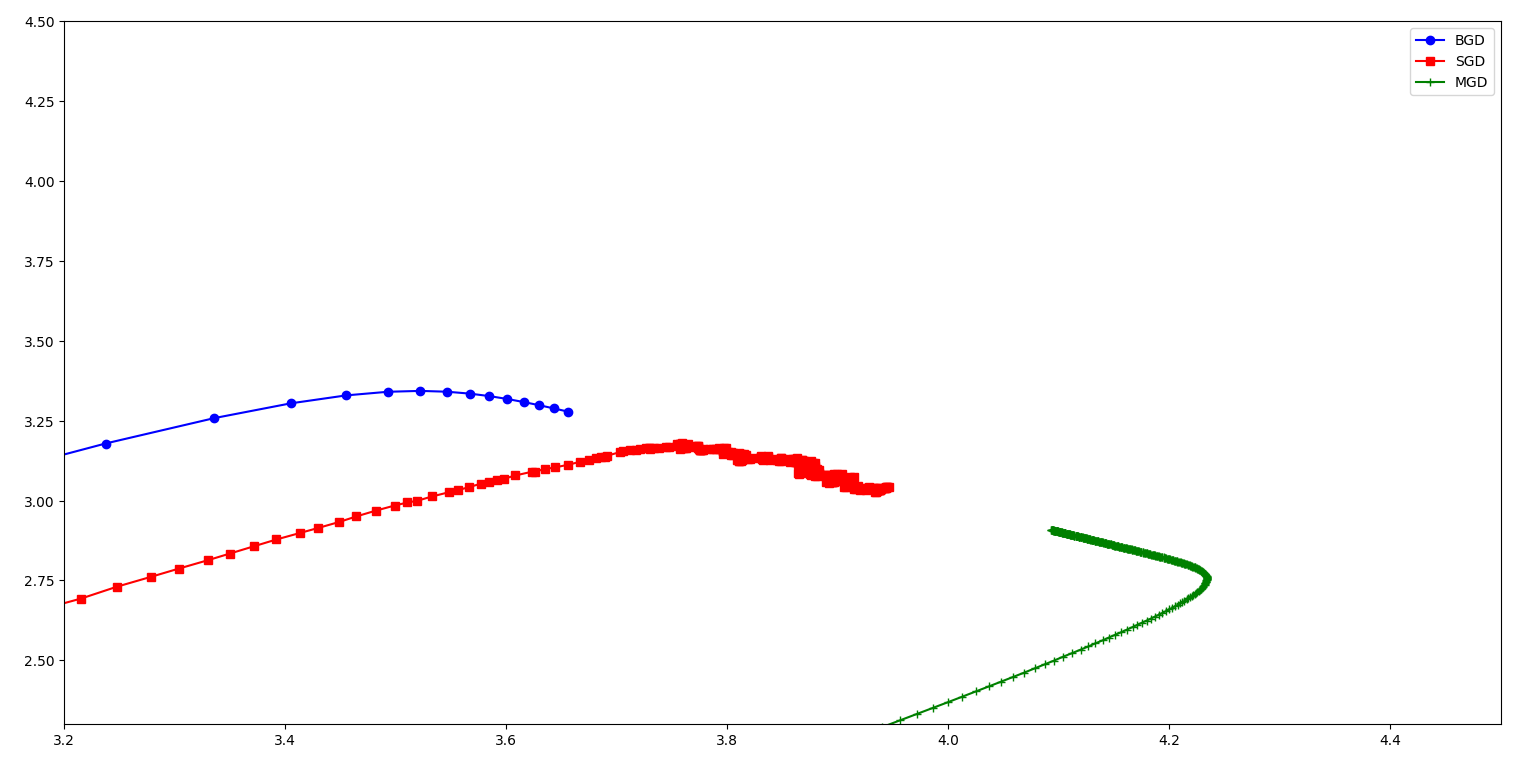
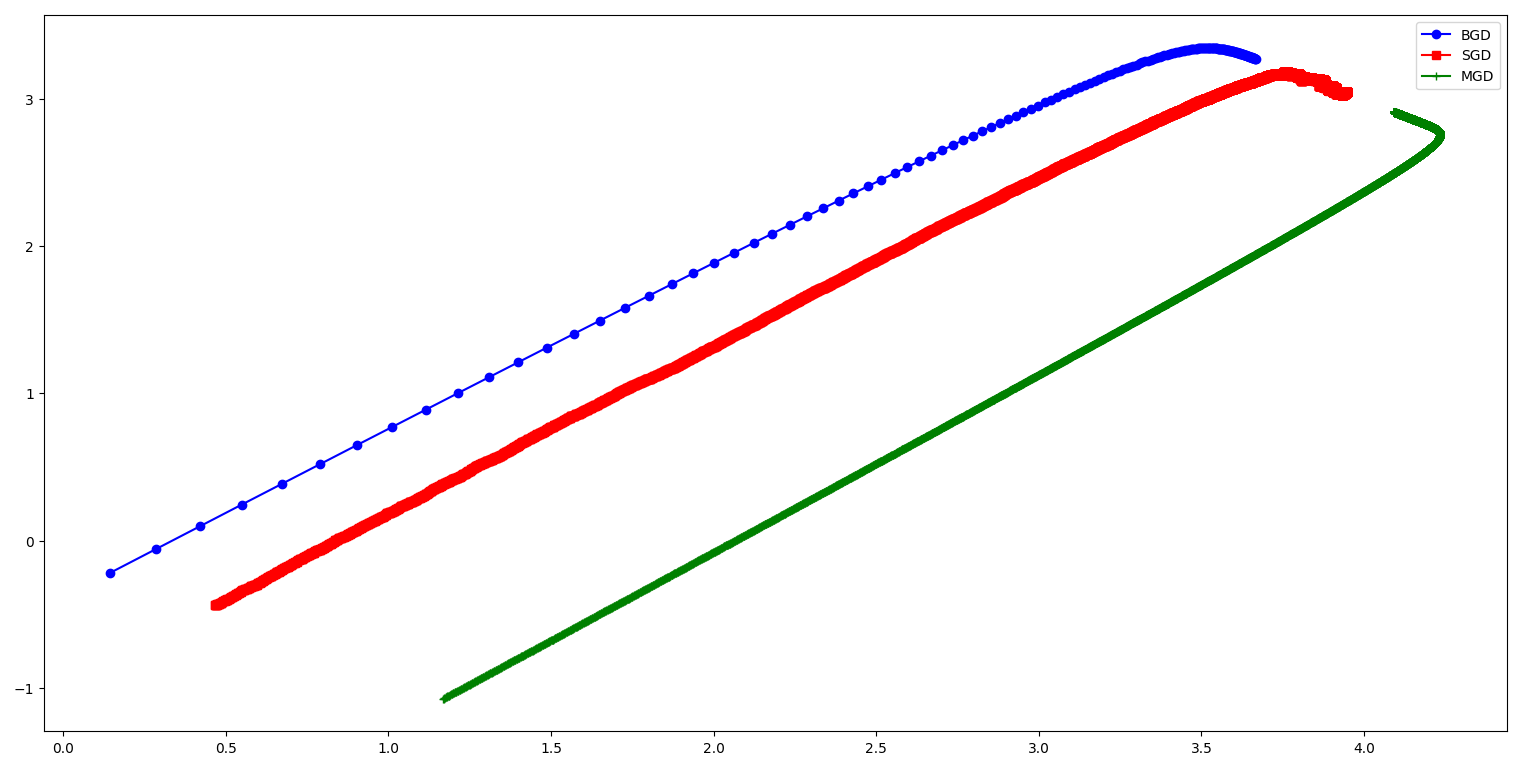
随机梯度下降：



批量梯度下降：



三者对比的完整图 ＆ 部分图：



**多项式回归**

from sklearn.preprocessing import PolynomialFeatures  
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import mean\_squared\_error  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import Ridge  
from sklearn.linear\_model import Lasso  
  
np.random.seed(42) # 设置固定种子便于实验

""" ---多项式回归--- """  
""" 生成数据集 & 绘图 """  
X = 6 \* np.random.rand(727, 1) - 3 # 727个[0, 1)之间均匀分布的随机数 \* 6 - 3  
y = 0.5\*X\*\*2 + X + np.random.randn(727, 1) # 0.5x^2 + x + 标准正态分布误差项  
plt.plot(X, y, 'b.')  
plt.axis([-3.2, 3.2, -4, 11])  
plt.show()  
  
""" 将多项式化为LinearRegression """  
poly\_features = PolynomialFeatures(degree=2, include\_bias=False)  
X\_poly = poly\_features.fit\_transform(X) # 将X变为[[X1,X1^2], [X2,X2^2], ... , [Xn,Xn^2]]，相当于将X^2变成了一个新的特征  
print("X[0]:", X[0], "X\_poly[0]", X\_poly[0]) # X\_poly[i] = (X[i])^2  
lin\_reg = LinearRegression()  
lin\_reg.fit(X\_poly, y)  
print("偏置参数：", lin\_reg.intercept\_) # 偏置参数 [0.01279778]。即常数项  
print("权重参数：", lin\_reg.coef\_) # 权重参数 [[0.98806406 0.4931436 ]]。因为X\_poly相当于X和X^2，因此系数分别对应一次项和二次项  
# 得到拟合曲线为0.4931436x^2 + 0.98806406x + 0.01279778  
X\_curve = np.linspace(-3, 3, 100).reshape(100, 1)  
X\_curve\_poly = poly\_features.transform(X\_curve) # 从-3到3的100个点经过变换(变为x与x^2)后的值  
y\_curve = lin\_reg.predict(X\_curve\_poly)  
plt.plot(X, y, 'b.')  
plt.plot(X\_curve, y\_curve, 'r-')  
plt.axis([-3.2, 3.2, -4, 11])  
plt.show()  
  
""" degree不同对结果的影响 """  
for style, width, degree in (('g-', 3, 50), ('r-', 2, 2), ('y--', 2, 1)):  
 poly\_features = PolynomialFeatures(degree=degree, include\_bias=False)  
 std = StandardScaler()  
 lin\_reg = LinearRegression()  
 # 流水线车间Pipeline包含三个流水线：poly\_features,StandardScaler,LinearRegression  
 polynomial\_reg = Pipeline([('poly\_features', poly\_features), # 多项式特征生成  
 ('StandardScaler', std), # 标准化  
 ('LinearRegression', lin\_reg)]) # 线性回归  
 polynomial\_reg.fit(X, y)  
 y\_curve\_ComparativeExperiments = polynomial\_reg.predict(X\_curve)  
 plt.plot(X\_curve, y\_curve\_ComparativeExperiments, style, label='degree:'+str(degree), linewidth=width)  
plt.plot(X, y, 'b.')  
plt.axis([-3.2, 3.2, -4, 11])  
plt.legend()  
plt.show()  
  
""" 数据样本数量对结果的影响 """  
def plot\_learning\_curves(model, X, y):  
 X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42) # val:validation验证集  
 train\_errors, val\_errors = [], []  
 for size in range(1, len(X\_train)): # 数据量从1到len(X\_train)  
 model.fit(X\_train[:size], y\_train[:size])  
 y\_train\_predicted = model.predict(X\_train[:size])  
 y\_val\_predict = model.predict(X\_val)  
 train\_errors.append(mean\_squared\_error(y\_train[:size], y\_train\_predicted[:size]))  
 val\_errors.append(mean\_squared\_error(y\_val, y\_val\_predict))  
 plt.plot(np.sqrt(train\_errors), 'r--', linewidth=2, label='train\_errors')  
 plt.plot(np.sqrt(val\_errors), 'b-', linewidth=3, label='val\_errors')  
 plt.xlabel("Training Size")  
 plt.ylabel("RMSE")  
 plt.legend()  
polynomial\_reg = Pipeline([('poly\_features', PolynomialFeatures(degree=2, include\_bias=False)), # 多项式特征生成  
 ('LinearRegression', LinearRegression())]) # 线性回归  
plot\_learning\_curves(polynomial\_reg, X, y)  
plt.show() # error越小越好  
  
""" 正则化(效果就是将拟合曲线拉平) """  
def plot\_model(model\_choice, alphas, poly\_degree):  
 for alpha, style in zip(alphas, ('y-', 'm--', 'r-.')):  
 ridge\_model = model\_choice(alpha=alpha)  
 model = Pipeline([('poly\_features', PolynomialFeatures(degree=poly\_degree, include\_bias=False)), # 多项式特征生成  
 ('StandardScaler', std), # 标准化  
 ('LinearRegression', ridge\_model)]) # 线性回归  
 model.fit(X, y)  
 y\_curve\_RidgeExperiments = model.predict(X\_curve)  
 plt.plot(X\_curve, y\_curve\_RidgeExperiments, style, label='alpha={}'.format(alpha), linewidth=3)  
 plt.plot(X, y, 'b.')  
 plt.legend()  
plot\_model(Ridge, alphas=(0, 10\*\*-25, 1), poly\_degree=200) # 学习率在10^-25处就能比较明显地与alpha=0区分(alpha=10^-30与alpha=0就差不多了)  
plt.axis([-3.2, 3.2, -4, 11])  
plt.show()  
  
plot\_model(Lasso, alphas=(0, 0.5, 1), poly\_degree=2)  
plt.axis([-3.2, 3.2, -4, 11])  
plt.show()

文字输出：

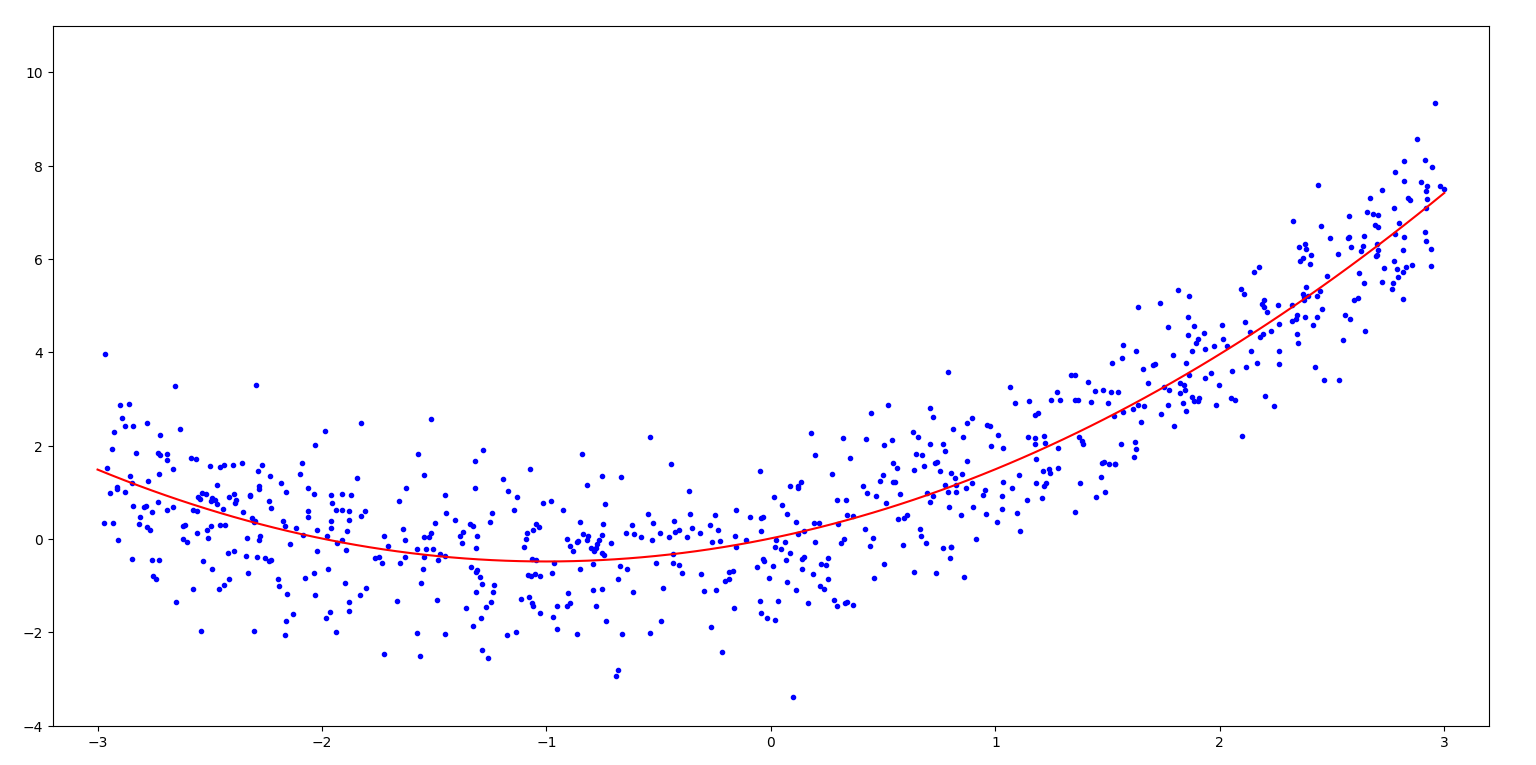
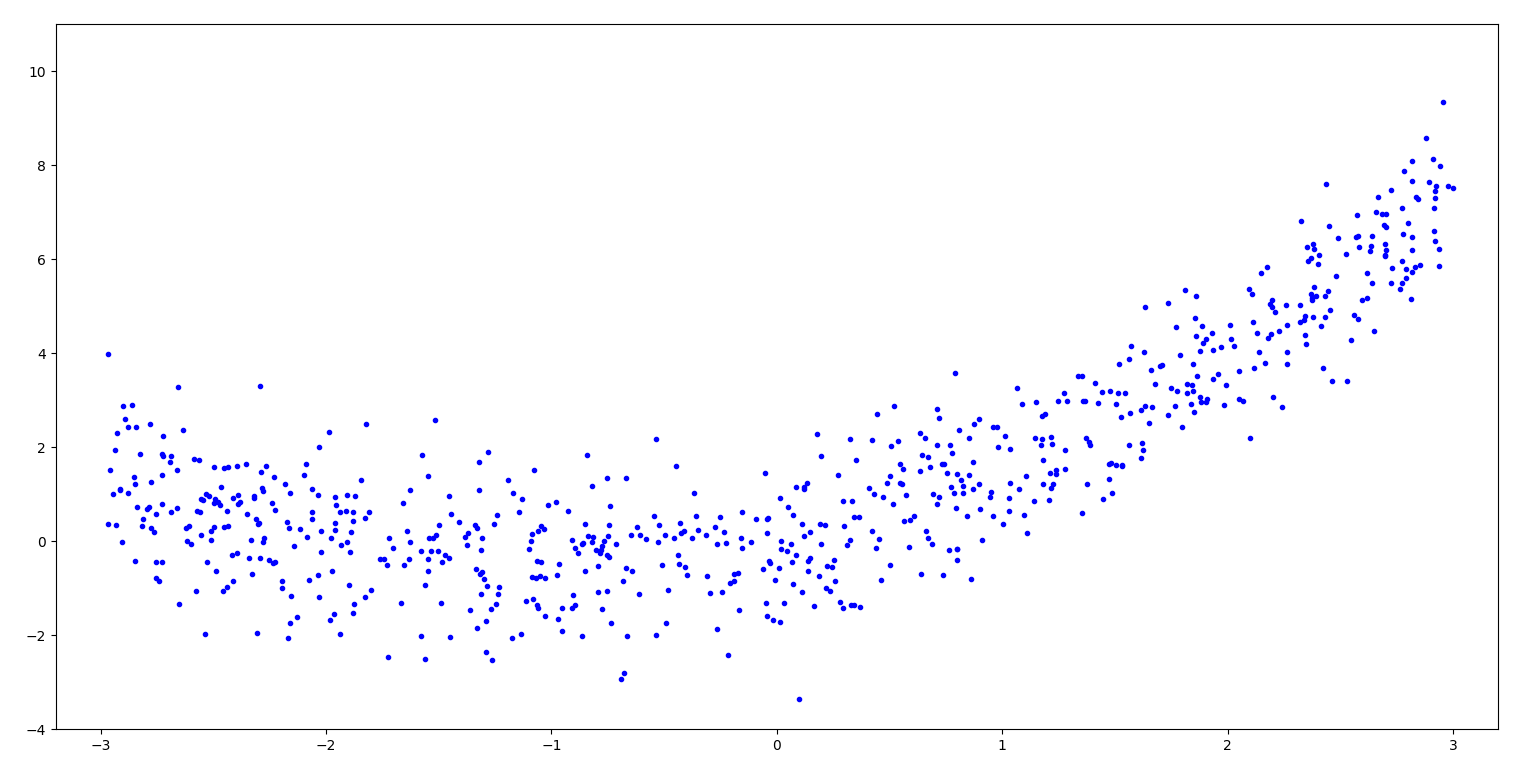
X[0]: [-0.75275929] X\_poly[0] [-0.75275929 0.56664654]

偏置参数： [0.01279778]

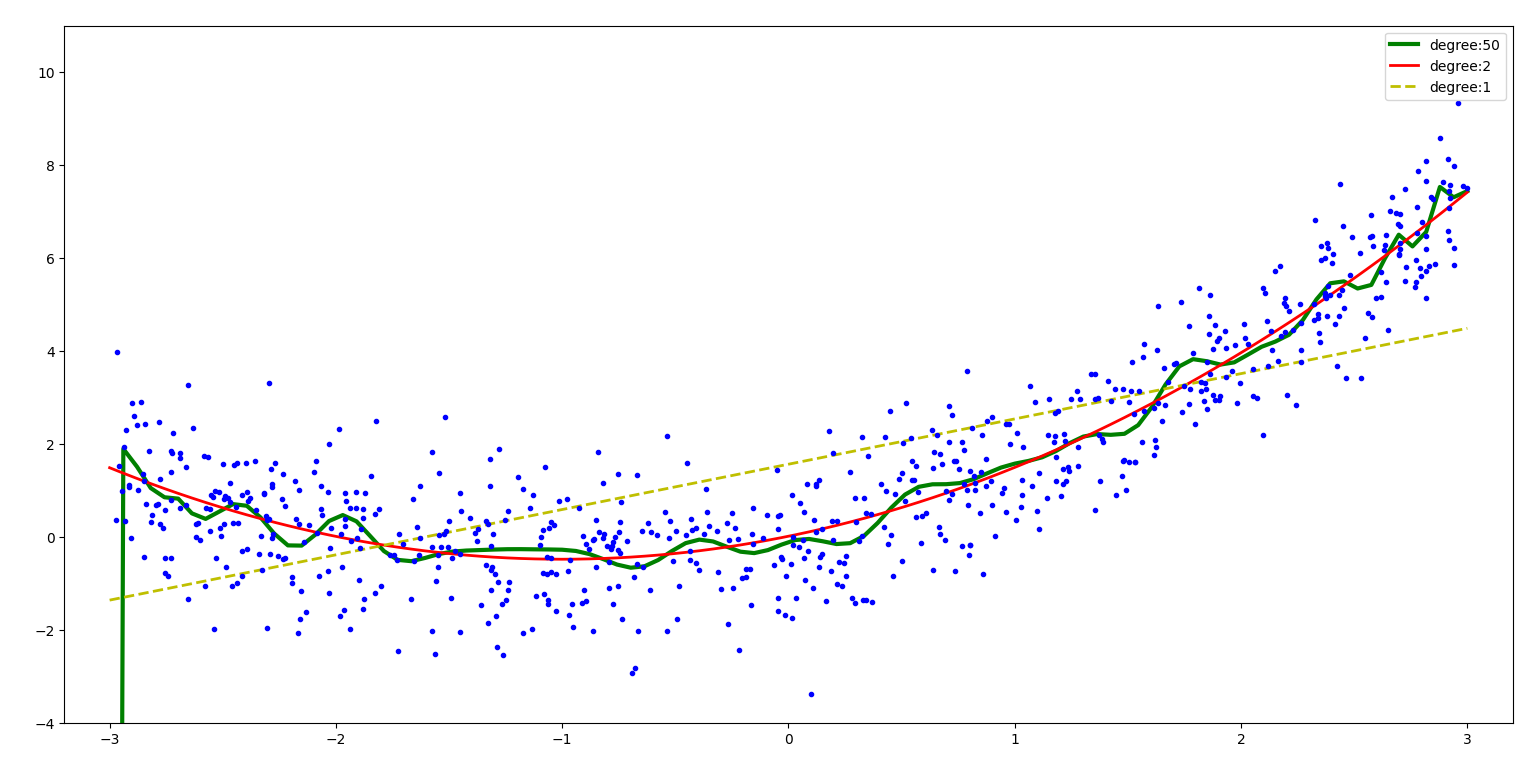
权重参数： [[0.98806406 0.4931436 ]]

图片输出：

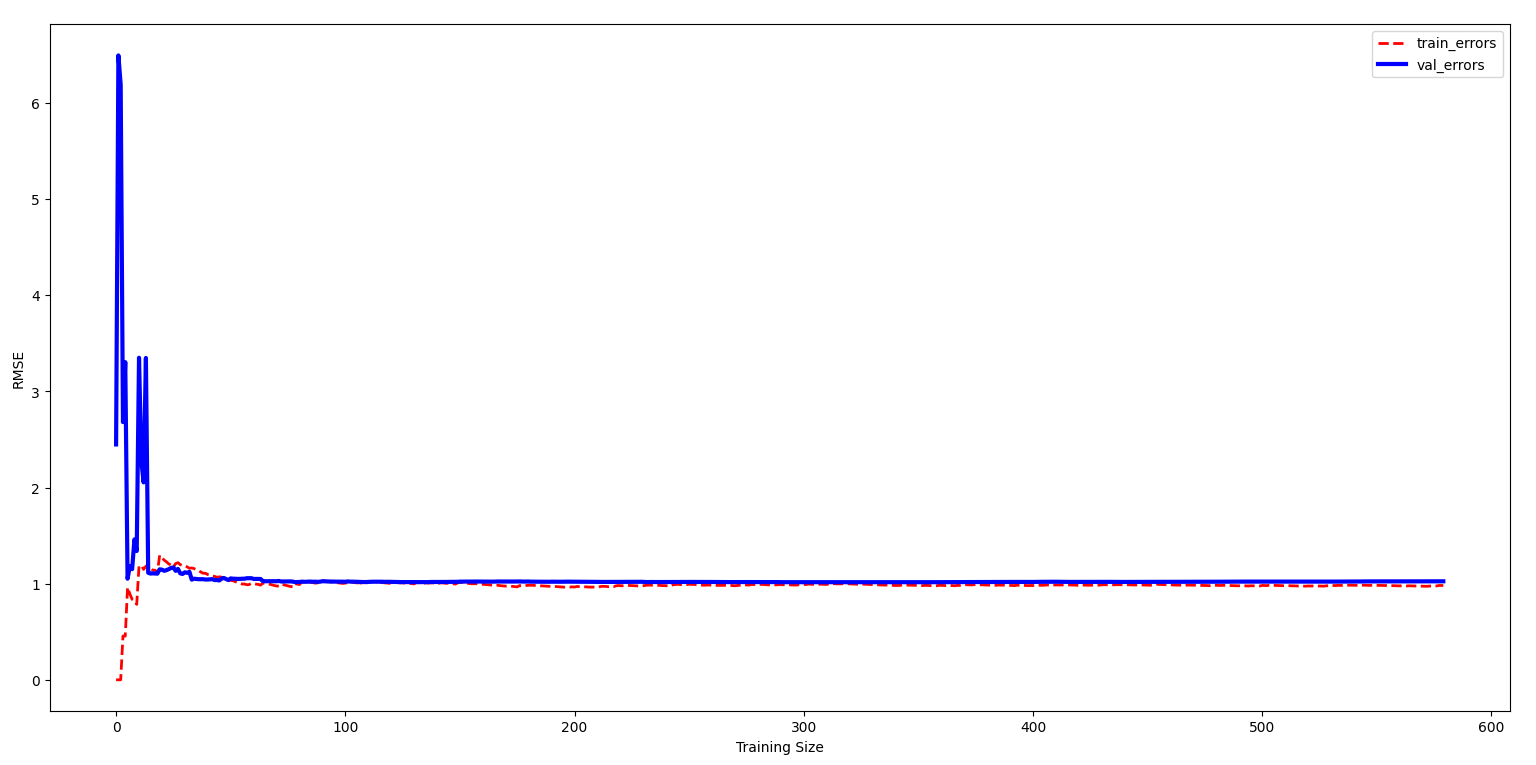
原始数据 & 二次拟合：



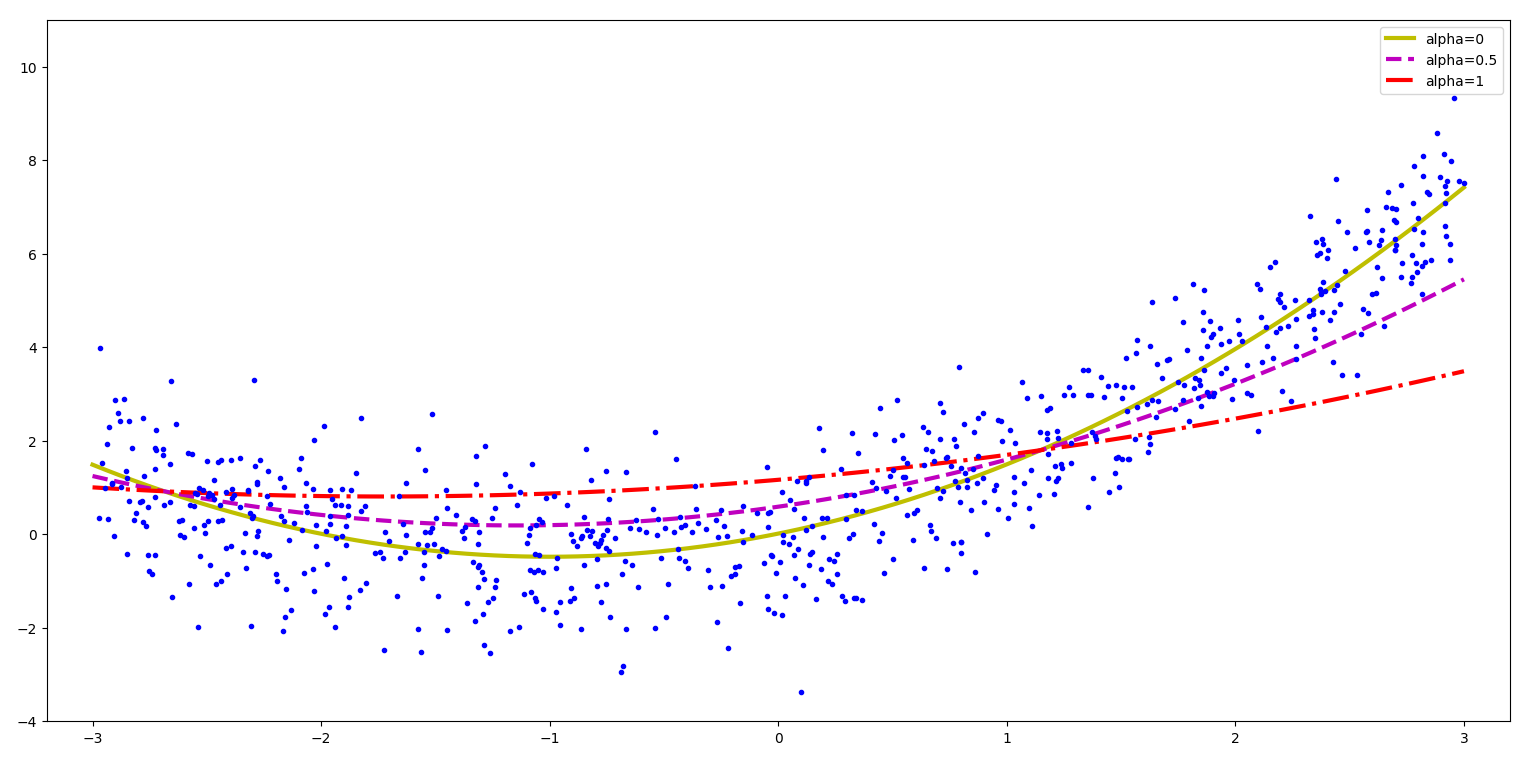
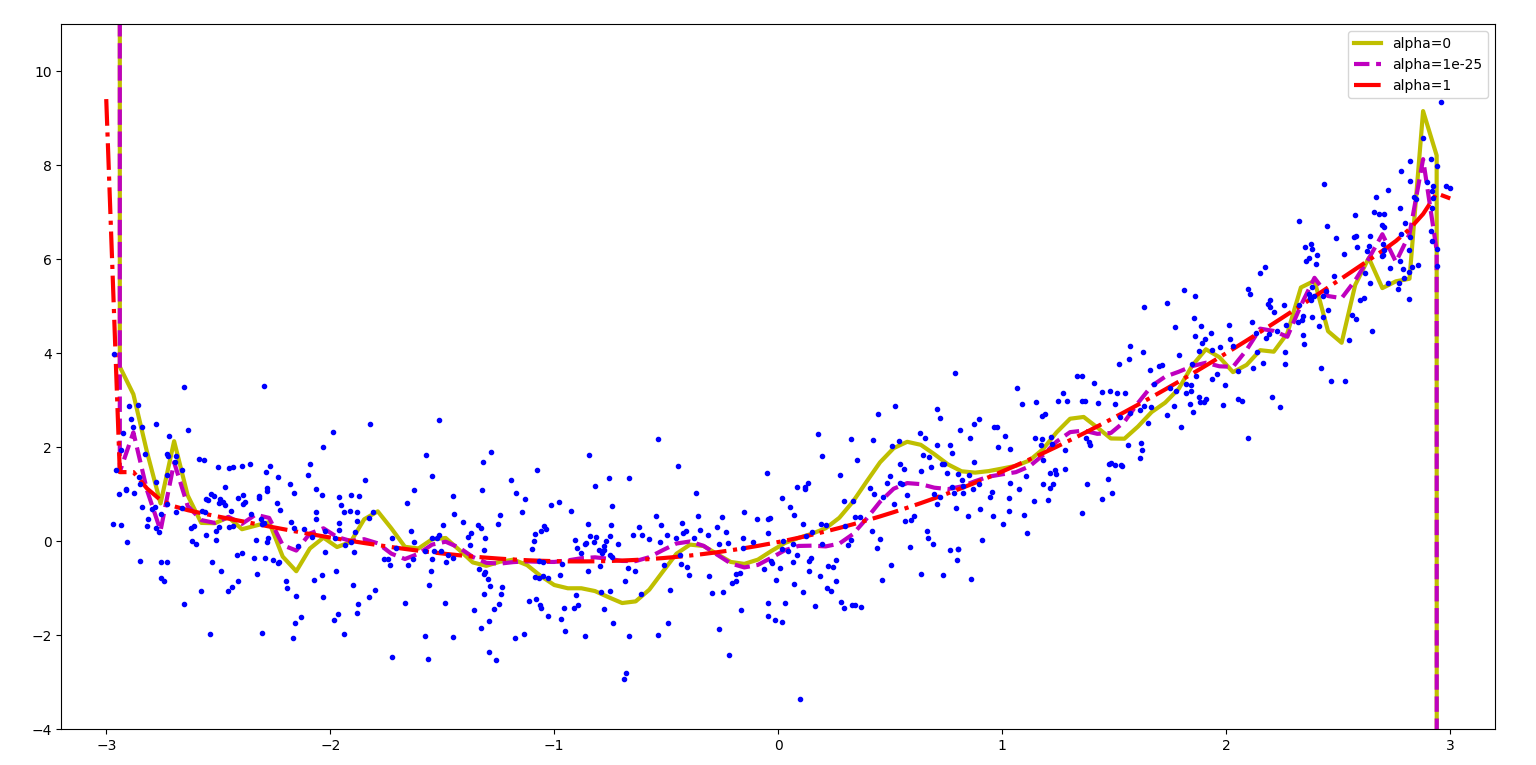
不同幂次的对比：



不同样本个数的对比：



L2 & L1 正则化：



**Logistic回归**

from sklearn.datasets import load\_iris  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import LogisticRegression  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=0)  
LR = LogisticRegression(penalty='l2', solver='newton-cg', multi\_class='multinomial')  
LR.fit(x\_train,y\_train)  
print("Logistic Regression模型训练集的准确率：%.3f" % LR.score(x\_train, y\_train))  
print("Logistic Regression模型测试集的准确率：%.3f" % LR.score(x\_test, y\_test))  
y\_hat = LR.predict(x\_test)  
print(classification\_report(y\_test, y\_hat, target\_names=Iris.target\_names))

输出

Logistic Regression模型训练集的准确率：0.981

Logistic Regression模型测试集的准确率：0.978

precision recall f1-score support

setosa 1.00 1.00 1.00 16

versicolor 1.00 0.94 0.97 18

virginica 0.92 1.00 0.96 11

accuracy 0.98 45

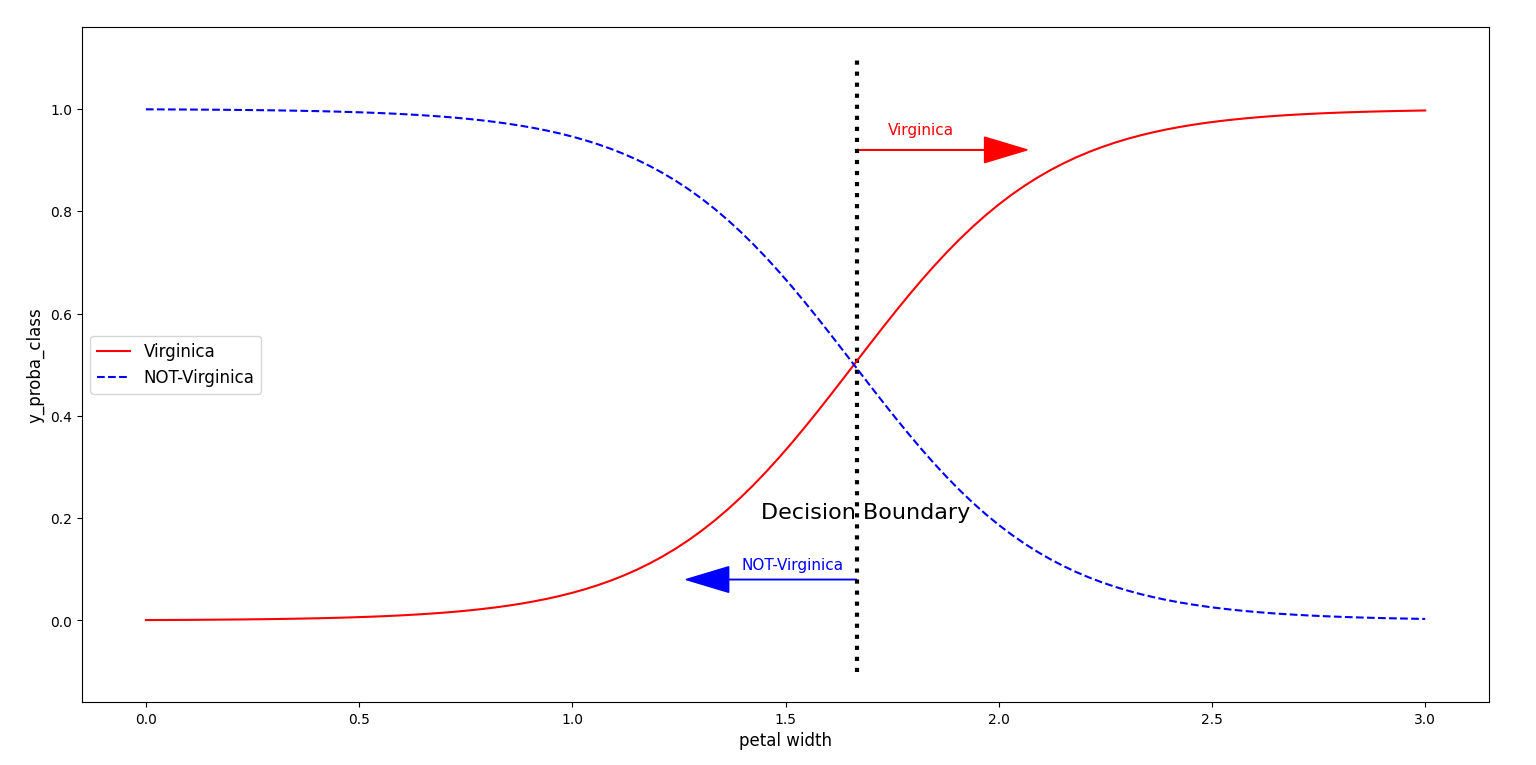
macro avg 0.97 0.98 0.98 45

weighted avg 0.98 0.98 0.98 45

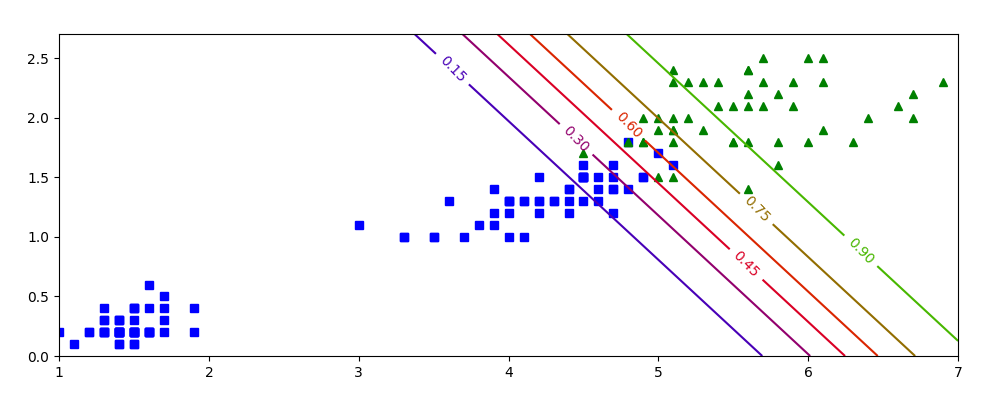
**Logistic回归绘制决策边界**

Iris = datasets.load\_iris() # 导入鸢尾花数据集  
print("属性列表：", list(Iris.keys())) # 查看Iris有什么属性可以供我们调用  
# print(Iris.DESCR) # 查看数据集描述  
X = Iris.data[:, 3:] # 选取一个特征：花瓣宽度petal width (cm)  
y = (Iris['target'] == 2).astype(int) # 将virginica设为1  
log\_res = LogisticRegression()  
log\_res.fit(X, y)  
X\_curve = np.linspace(0, 3, 100).reshape(-1, 1)  
y\_proba = log\_res.predict\_proba(X\_curve)  
print(y\_proba) # 默认第一列是样本被预测为负类别0的概率，第二列是被预测为正类别1的概率  
# 概率结果随所选特征的变化的可视化  
decision\_boundary = X\_curve[y\_proba[:, 1] >= 0.5][0] # [1.66666667]  
plt.plot([decision\_boundary, decision\_boundary], [-0.1, 1.1], 'k:', linewidth=3)  
plt.plot(X\_curve, y\_proba[:, 1], 'r-', label='Virginica')  
plt.plot(X\_curve, y\_proba[:, 0], 'b--', label='NOT-Virginica')  
plt.arrow(decision\_boundary[0], 0.08, -0.3, 0, head\_width=0.05, head\_length=0.1, fc='b', ec='b')  
plt.arrow(decision\_boundary[0], 0.92, 0.3, 0, head\_width=0.05, head\_length=0.1, fc='r', ec='r')  
plt.text(decision\_boundary+0.02, 0.2, 'Decision Boundary', fontsize=16, color='k', ha='center')  
plt.text(decision\_boundary-0.15, 0.1, 'NOT-Virginica', fontsize=11, color='b', ha='center')  
plt.text(decision\_boundary+0.15, 0.95, 'Virginica', fontsize=11, color='r', ha='center')  
plt.xlabel('petal width', fontsize=12)  
plt.ylabel('y\_proba\_class', fontsize=12)  
plt.legend(loc='center left', fontsize=12)  
plt.show()  
  
X = Iris.data[:, (2, 3)] # 提取了第2、3列petal length (cm) petal width (cm)  
y = (Iris.target == 2).astype(int) # 是不是virginica  
log\_res.fit(X, y)  
x0, x1 = np.meshgrid(np.linspace(1, 7, 500).reshape(-1, 1), np.linspace(0, 2.7, 500).reshape((-1, 1))) # 笛卡尔积  
X\_new = np.c\_[x0.ravel(), x1.ravel()] # shape为(250000, 2)  
y\_proba = log\_res.predict\_proba(X\_new)  
plt.figure(figsize=(10, 4))  
plt.plot(X[y == 0, 0], X[y == 0, 1], 'bs') # 类别为0的样本用蓝色方块  
plt.plot(X[y == 1, 0], X[y == 1, 1], "g^") # 类别为1的样本用绿色三角  
zz = y\_proba[:, 1].reshape(x0.shape)  
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg) # 绘制等高线图  
plt.clabel(contour, inline=1) # 在等高线上标注概率值  
plt.show()  
  
X = Iris.data[:, (2, 3)] # 提取了第2、3列petal length (cm) petal width (cm)  
y = Iris.target  
softmax\_log = LogisticRegression(multi\_class='multinomial', solver='lbfgs')  
softmax\_log.fit(X, y)  
x0, x1 = np.meshgrid(np.linspace(1, 7, 500).reshape(-1, 1), np.linspace(0, 2.7, 500).reshape((-1, 1))) # 笛卡尔积  
X\_new = np.c\_[x0.ravel(), x1.ravel()] # shape为(250000, 2)  
y\_proba = softmax\_log.predict\_proba(X\_new)  
y\_predict = softmax\_log.predict(X\_new)  
zzl = y\_proba[:, 1].reshape(x0.shape)  
zz = y\_predict.reshape(x0.shape)  
plt.figure(figsize=(10, 4))  
plt.plot(X[y == 0, 0], X[y == 0, 1], "yo", label="setosa")  
plt.plot(X[y == 1, 0], X[y == 1, 1], "bs", label="versicolor")  
plt.plot(X[y == 2, 0], X[y == 2, 1], "g^", label="virginica")  
plt.contourf(x0, x1, zz, cmap=matplotlib.colors.ListedColormap(['#66ccff', "#39c5bb", '#ffc0cb'])) # 绘制类别区域的等高线图  
contour = plt.contour(x0, x1, zzl, cmap=plt.cm.brg) # 绘制决策边界的等高线图  
plt.clabel(contour, inline=1, fontsize=12)  
plt.show()

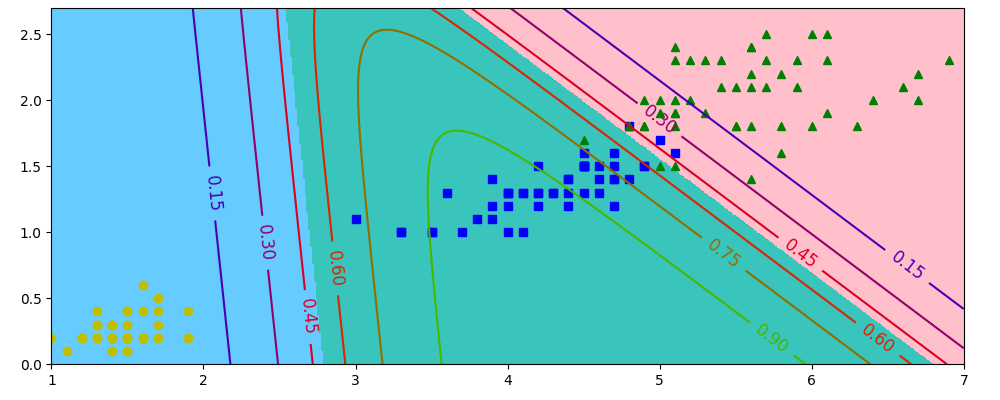
预测：



单分类边界：



多分类边界：

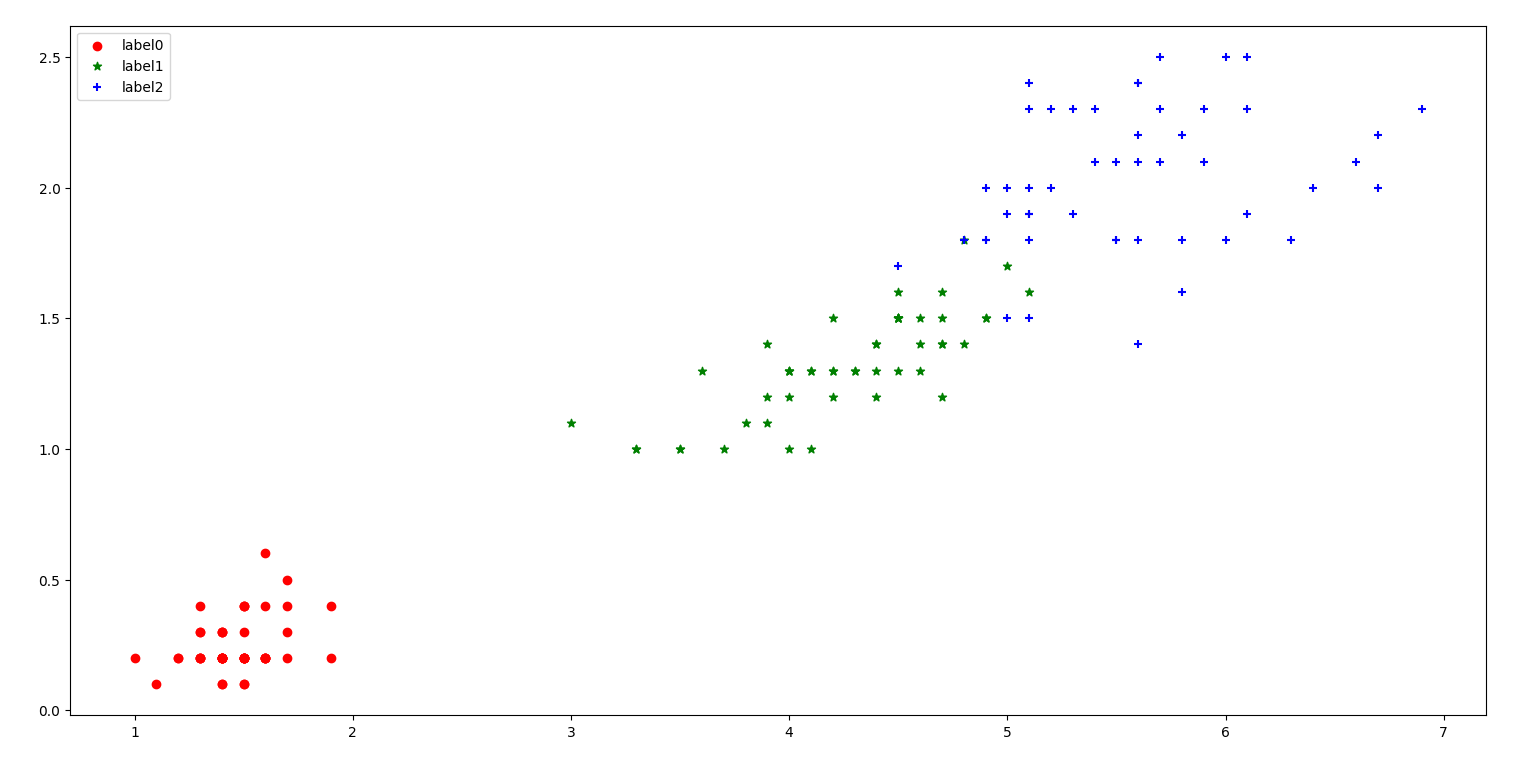
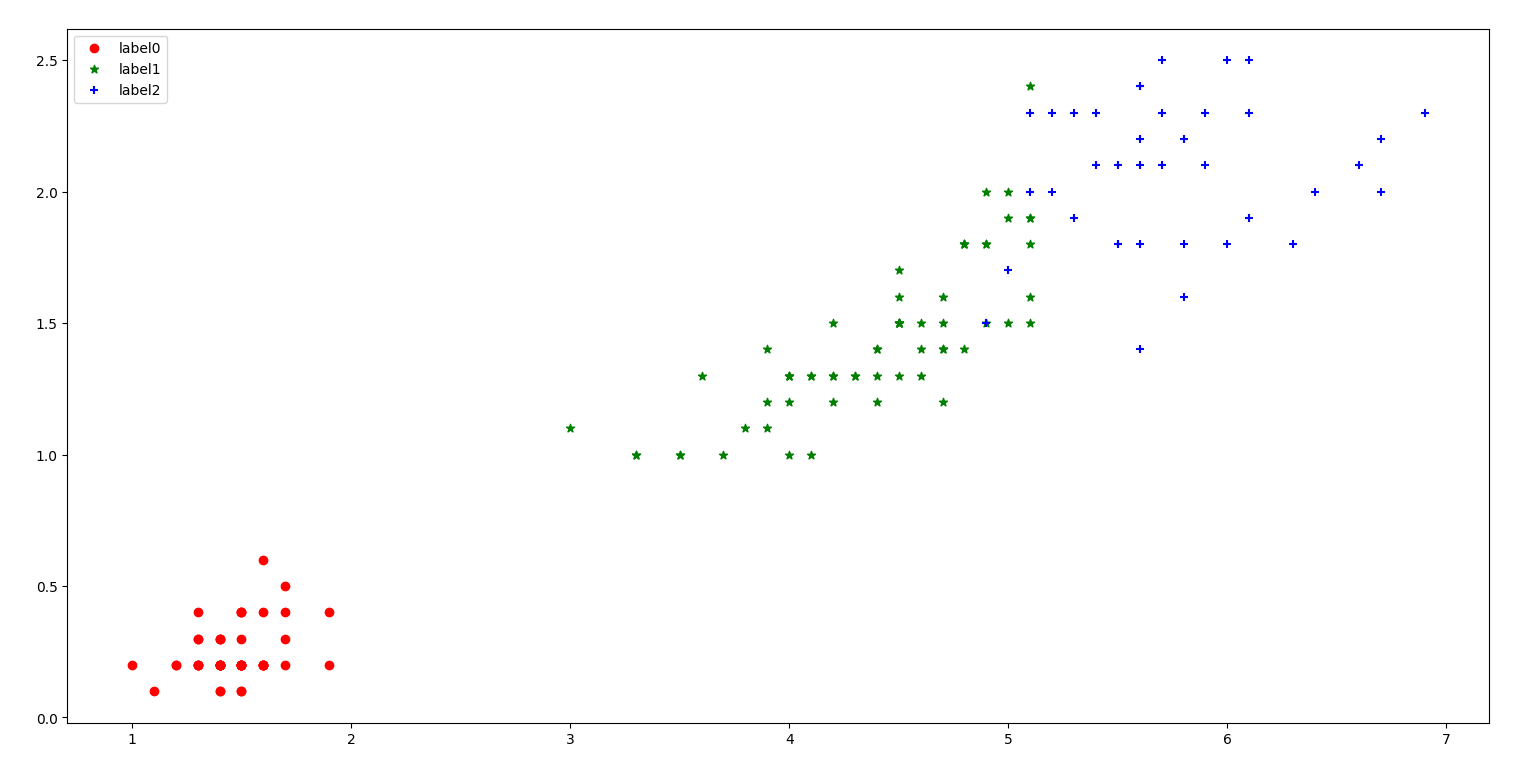


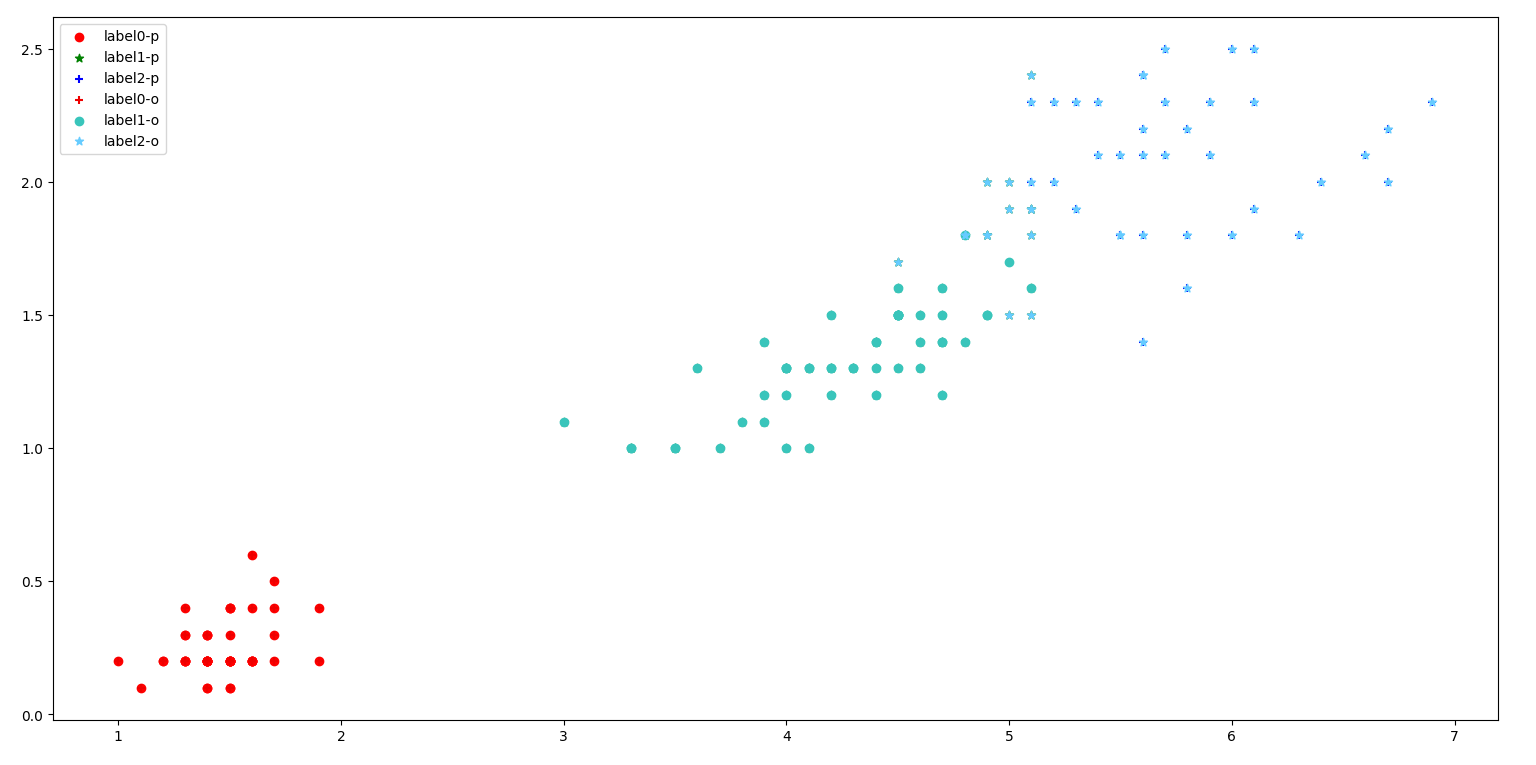
**9.聚类**

**K-means**

from sklearn.datasets import load\_iris  
from sklearn.cluster import KMeans  
import matplotlib.pyplot as plt  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
  
estimator = KMeans(n\_clusters=3, n\_init=10) # n\_init 是KMeans算法中用于指定随机初始化的次数的参数。  
estimator.fit(X)  
label\_predict = estimator.labels\_ # 是一个list，包含对应的聚类标签(0,1,2)  
  
# 绘制预测数据  
xp0 = X[label\_predict == 0] # xp0 代指predict的label是0的x  
xp1 = X[label\_predict == 1]  
xp2 = X[label\_predict == 2]  
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')  
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='\*', label='label1')  
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')  
plt.legend(loc=2) # 添加图例(loc=2代表左上角绘制)  
plt.show()  
  
# 绘制原始数据  
xo0 = X[Y == 0] # xo0 代指original label是0的x  
xo1 = X[Y == 1]  
xo2 = X[Y == 2]  
plt.scatter(xo0[:, 2], xo0[:, 3], c='red', marker='o', label='label0')  
plt.scatter(xo1[:, 2], xo1[:, 3], c='green', marker='\*', label='label1')  
plt.scatter(xo2[:, 2], xo2[:, 3], c='blue', marker='+', label='label2')  
plt.legend(loc=2)  
plt.show()  
  
# 比较绘制  
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0-p')  
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='\*', label='label1-p')  
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2-p')  
plt.scatter(xo0[:, 2], xo0[:, 3], c='#EE0000', marker='+', label='label0-o')  
plt.scatter(xo1[:, 2], xo1[:, 3], c='#39C5BB', marker='o', label='label1-o')  
plt.scatter(xo2[:, 2], xo2[:, 3], c='#66CCFF', marker='\*', label='label2-o')  
plt.legend(loc=2)  
plt.show()

输出（依次为预测、原始、预测和原始叠加）





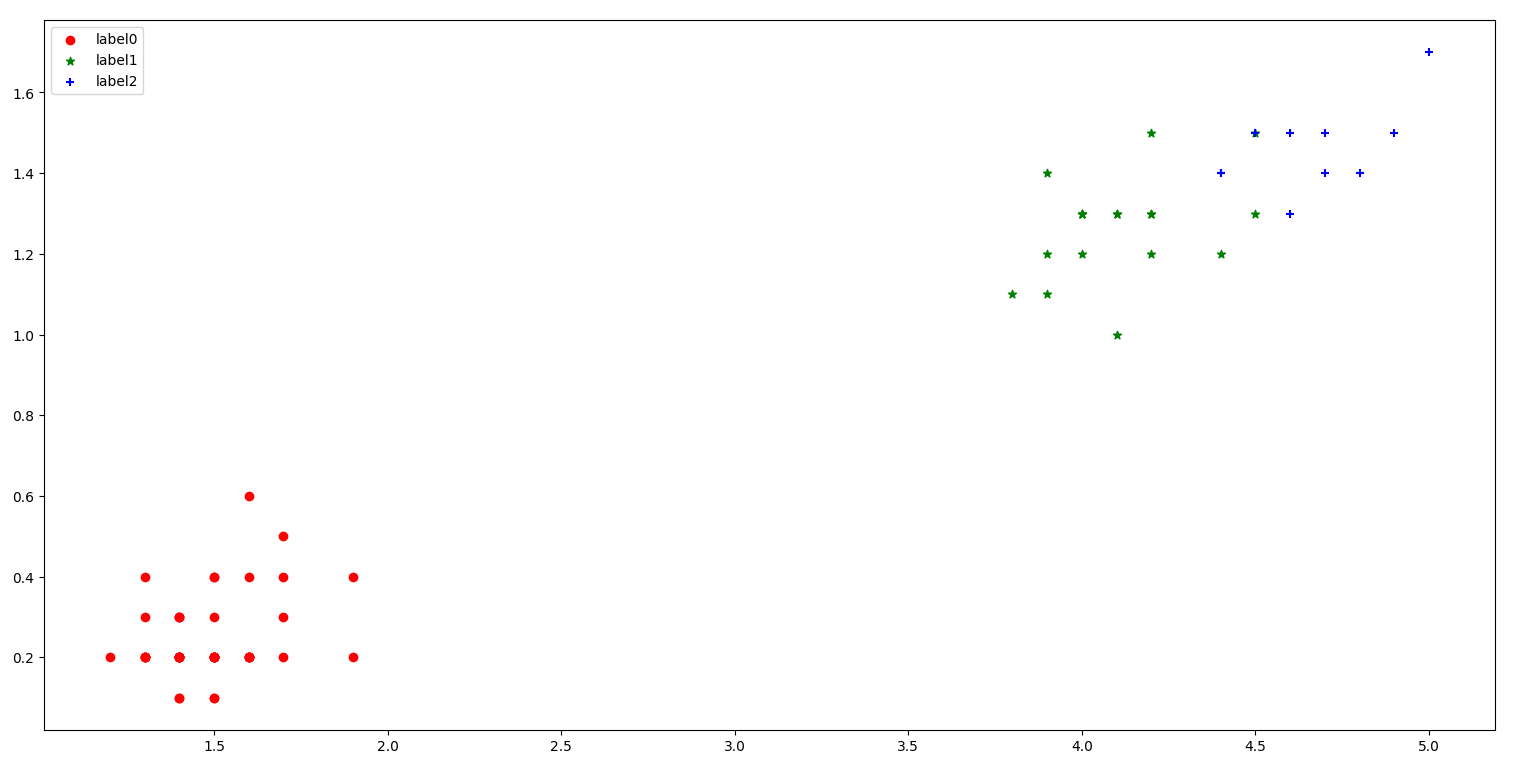
当聚类数量较多不便书写时可考虑以下两种方法:（以绘制原始数据的target值的图为例）

for target\_value in set(Y):  
 x = X[Y == target\_value]  
 if target\_value == 0:  
 plt.scatter(x[:, 2], x[:, 3], c='red', marker='o', label=f'Target {target\_value}')  
 elif target\_value == 1:  
 plt.scatter(x[:, 2], x[:, 3], c='green', marker='\*', label=f'Target {target\_value}')  
 else:  
 plt.scatter(x[:, 2], x[:, 3], c='blue', marker='+', label=f'Target {target\_value}')  
  
# 定义颜色和标记对应关系  
colors = ['red', 'green', 'blue']  
markers = ['o', '\*', '+']  
for target\_value in set(Y):  
 x = X[Y == target\_value]  
 color = colors[target\_value]  
 marker = markers[target\_value]  
 label = f'Target {target\_value}'  
 plt.scatter(x[:, 2], x[:, 3], c=color, marker=marker, label=label)

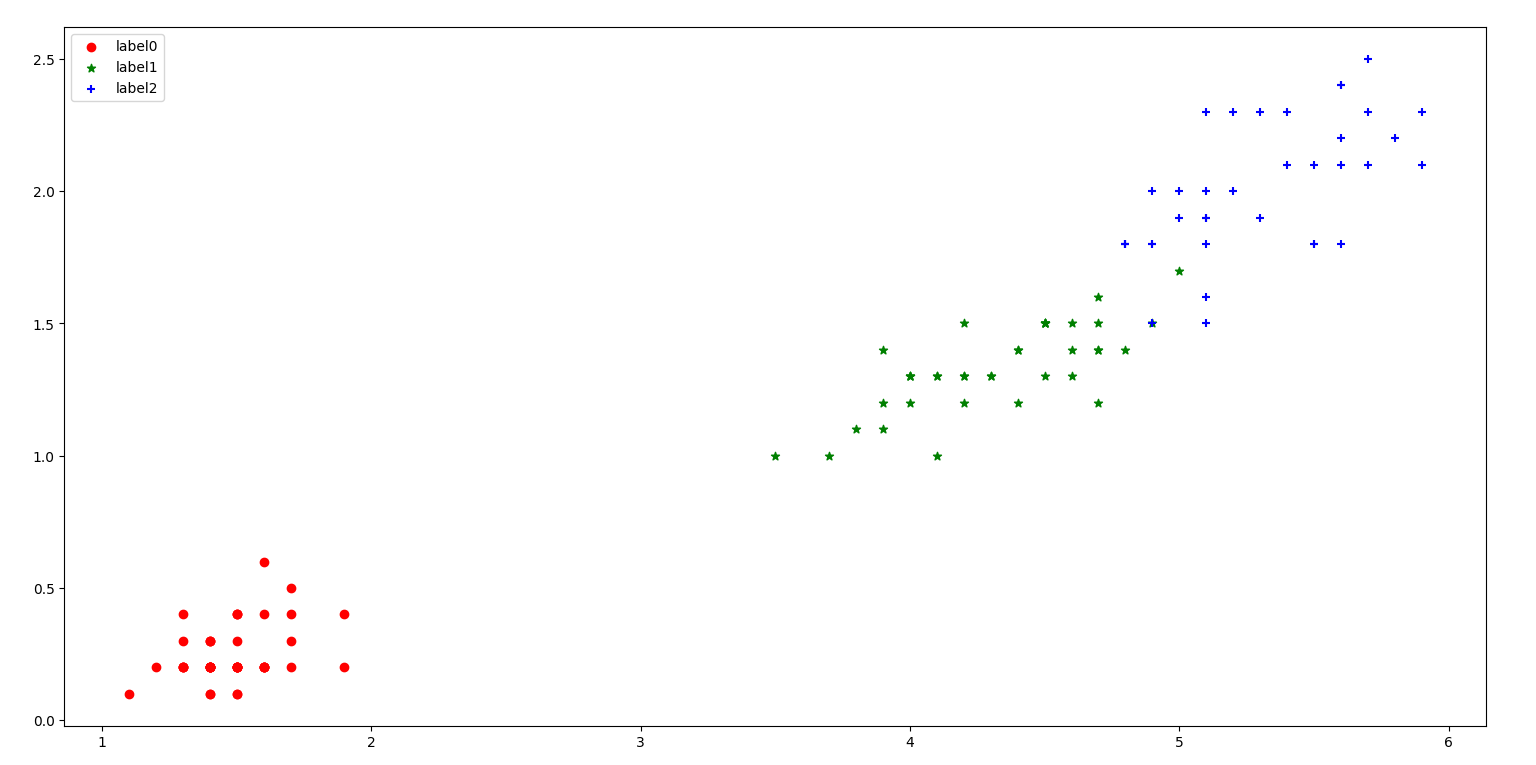
**DBSCAN**

from sklearn.datasets import load\_iris  
from sklearn.cluster import DBSCAN  
import matplotlib.pyplot as plt  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
  
dbscan = DBSCAN(eps=0.4, min\_samples=9) # 半径0.4内最少要有9个样本  
dbscan.fit(X)  
label\_predict = dbscan.labels\_ # 是一个list，包含对应的聚类标签(0,1,2)  
  
# 绘制预测数据  
xp0 = X[label\_predict == 0] # xp0 代指predict的label是0的x  
xp1 = X[label\_predict == 1]  
xp2 = X[label\_predict == 2]  
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')  
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='\*', label='label1')  
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')  
plt.legend(loc=2) # 添加图例(loc=2代表左上角绘制)  
plt.show()

输出

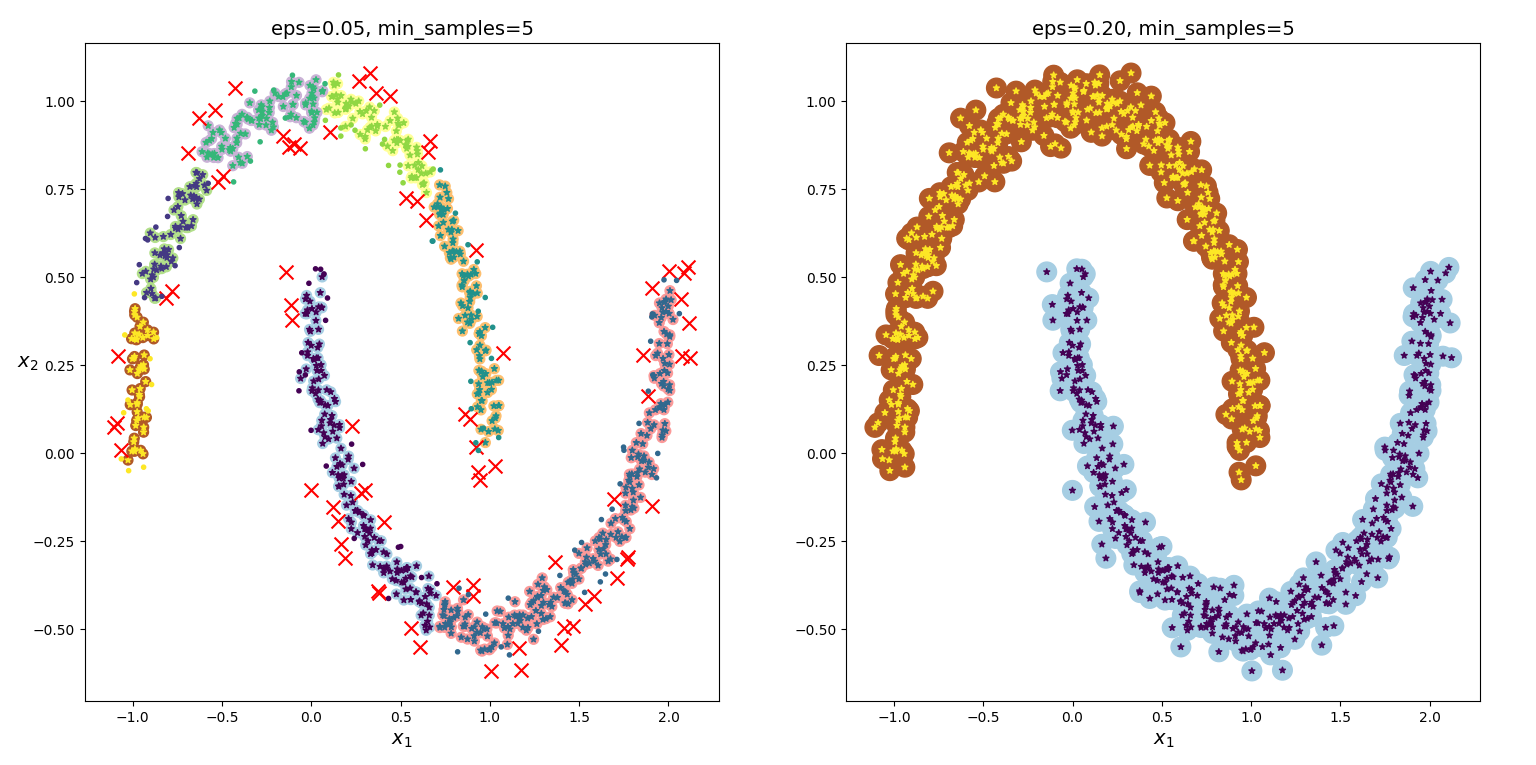


会发现和想要的结果不太一样。更改参数为eps=0.4, min\_samples=4，输出：



**DBSCAN的可视化展示**

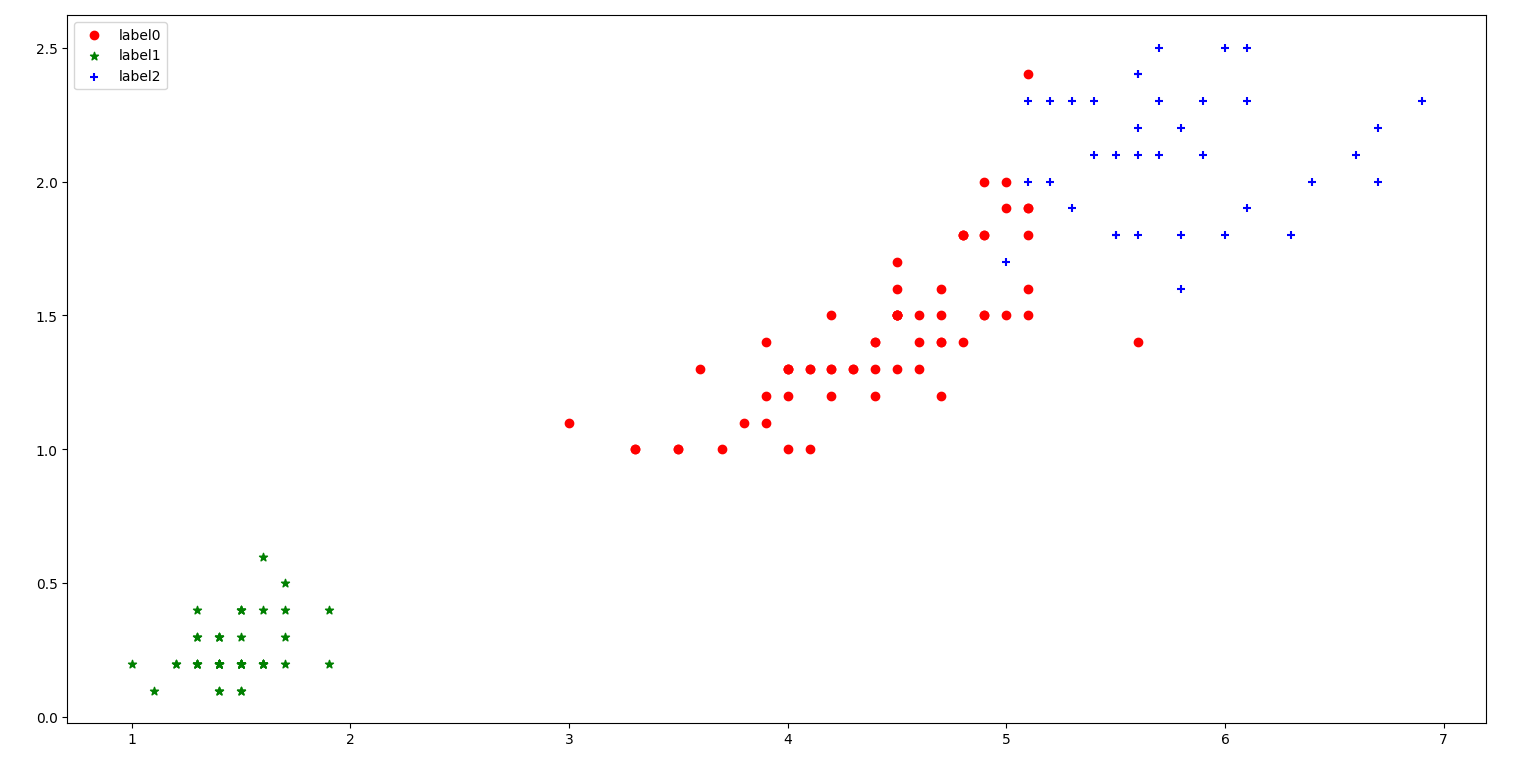
from sklearn.cluster import DBSCAN  
from sklearn.datasets import make\_moons  
import matplotlib.pyplot as plt  
import numpy as np  
  
X, y = make\_moons(n\_samples=1000, noise=0.05, random\_state=42)  
  
dbscan = DBSCAN(eps=0.05, min\_samples=5)  
dbscan.fit(X)  
dbscan2 = DBSCAN(eps=0.2, min\_samples=5)  
dbscan2.fit(X)  
  
  
def plot\_dbscan(dbscan, X, size, show\_ylabels=True):  
 core\_mask = np.zeros\_like(dbscan.labels\_, dtype=bool)  
 core\_mask[dbscan.core\_sample\_indices\_] = True  
 anomalies\_mask = dbscan.labels\_ == -1  
 non\_core\_mask = ~(core\_mask | anomalies\_mask)  
  
 cores = dbscan.components\_ # 核心点  
 anomalies = X[anomalies\_mask] # 异常点  
 non\_cores = X[non\_core\_mask] # 既不是核心点也不是异常点的数据点  
  
 plt.scatter(cores[:, 0], cores[:, 1], c=dbscan.labels\_[core\_mask], marker='o', s=size, cmap="Paired") # 填充核心点的周围区域  
 plt.scatter(cores[:, 0], cores[:, 1], marker='\*', s=20, c=dbscan.labels\_[core\_mask]) # 绘制核心点  
 plt.scatter(anomalies[:, 0], anomalies[:, 1], c="r", marker="x", s=100) # 绘制异常点  
 plt.scatter(non\_cores[:, 0], non\_cores[:, 1], c=dbscan.labels\_[non\_core\_mask], marker=".") # 绘制其他点  
 plt.xlabel("$x\_1$", fontsize=14)  
 if show\_ylabels:  
 plt.ylabel("$x\_2$", fontsize=14, rotation=0)  
 else:  
 plt.tick\_params(labelleft='off')  
 plt.title("eps={:.2f}, min\_samples={}".format(dbscan.eps, dbscan.min\_samples), fontsize=14)  
  
  
plt.subplot(121)  
plot\_dbscan(dbscan, X, size=50)  
plt.subplot(122)  
plot\_dbscan(dbscan2, X, size=200, show\_ylabels=False)  
plt.show()



**AGENS**

from sklearn.datasets import load\_iris  
from sklearn.cluster import AgglomerativeClustering  
import matplotlib.pyplot as plt  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
  
AGENS = AgglomerativeClustering(linkage='ward', n\_clusters=3) # 以ward方差最小化的方式来计算簇的距离，分为3个簇。  
AGENS.fit(X)  
label\_predict = AGENS.labels\_ # 是一个list，包含对应的聚类标签(0,1,2)  
  
# 绘制预测数据  
xp0 = X[label\_predict == 0] # xp0 代指predict的label是0的x  
xp1 = X[label\_predict == 1]  
xp2 = X[label\_predict == 2]  
plt.scatter(xp0[:, 2], xp0[:, 3], c='red', marker='o', label='label0')  
plt.scatter(xp1[:, 2], xp1[:, 3], c='green', marker='\*', label='label1')  
plt.scatter(xp2[:, 2], xp2[:, 3], c='blue', marker='+', label='label2')  
plt.legend(loc=2) # 添加图例(loc=2代表左上角绘制)  
plt.show()

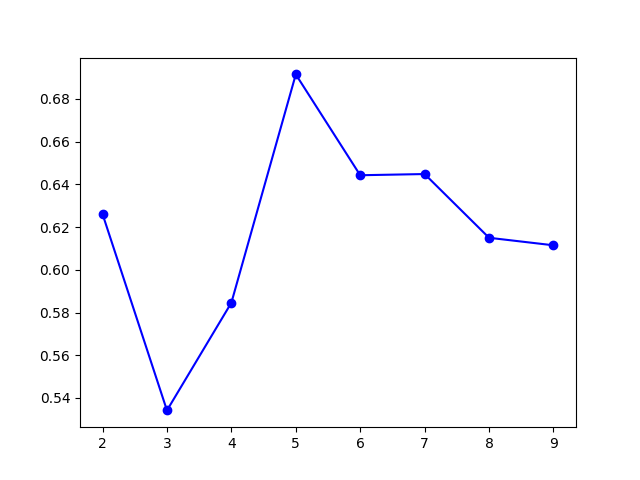
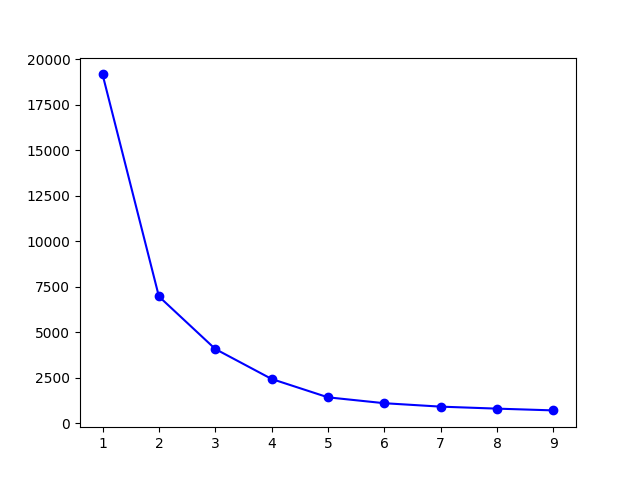
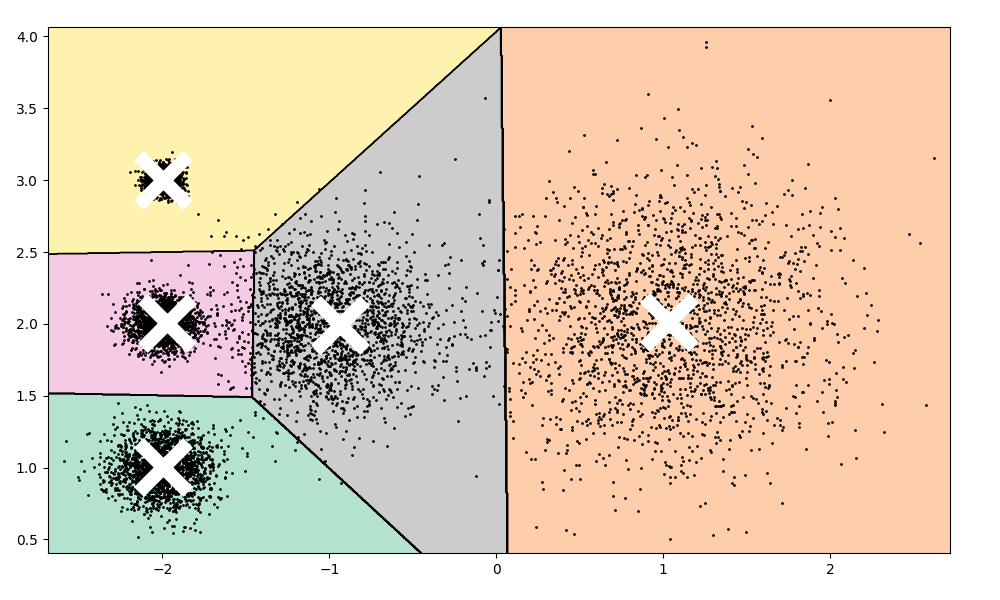
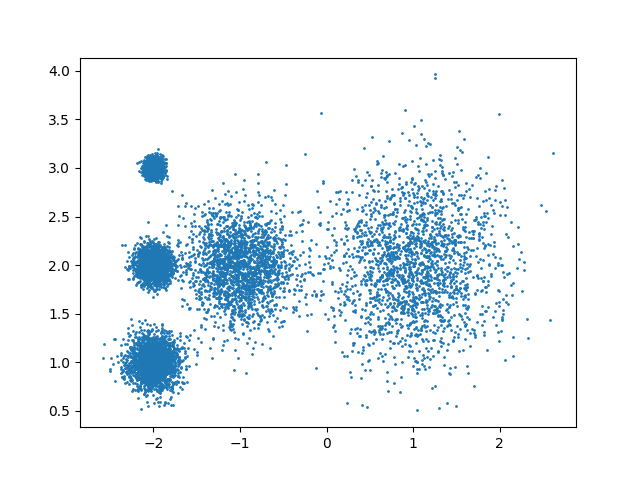
输出



**聚类效果的评估**

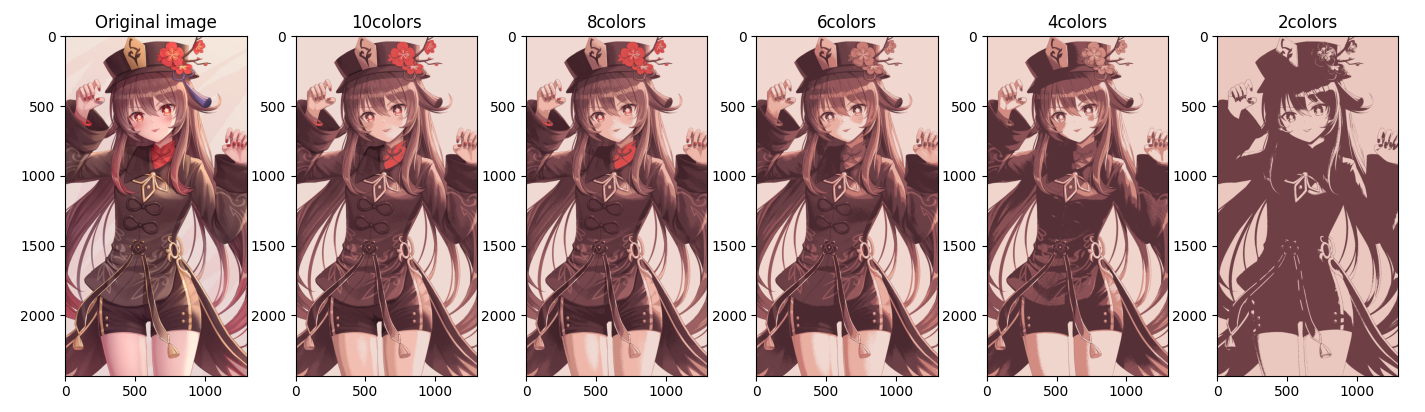
import matplotlib.pyplot as plt  
import numpy as np  
from sklearn.datasets import make\_blobs  
from sklearn.cluster import KMeans  
from sklearn.metrics import silhouette\_score  
  
blob\_centers = np.array([[1, 2], [-1, 2], [-2, 1], [-2, 2], [-2, 3]])  
blob\_std = np.array([0.5, 0.3, 0.15, 0.1, 0.05])  
X, y = make\_blobs(n\_samples=10000, centers=blob\_centers, cluster\_std=blob\_std, random\_state=42)  
  
  
def plot\_clusters(X, y=None):  
 plt.scatter(X[:, 0], X[:, 1], c=y, s=1)  
  
  
plot\_clusters(X)  
plt.show()  
  
kmeans = KMeans(n\_clusters=5, random\_state=42, n\_init=10)  
y\_predict = kmeans.fit\_predict(X)  
print(y\_predict) # [0 2 0 ... 0 1 3]  
  
  
def plot\_data(X):  
 plt.plot(X[:, 0], X[:, 1], "k.", markersize=2)  
  
  
def plot\_centroids(centroids, weights=None, circle\_color="w", cross\_color="k"):  
 if weights is not None:  
 centroids = centroids[weights > weights.max() / 10]  
 plt.scatter(centroids[:, 0], centroids[:, 1], marker="o", s=30, linewidths=8, color=circle\_color, zorder=10, alpha=0.9)  
 plt.scatter(centroids[:, 0], centroids[:, 1], marker="x", s=50, linewidths=50, color=circle\_color, zorder=11, alpha=1)  
  
  
def plot\_decision\_boundaries(cluster, X, resolution=1000):  
 X\_min = X.min(axis=0) - 0.1  
 X\_max = X.max(axis=0) + 0.1  
 xx, yy = np.meshgrid(np.linspace(X\_min[0], X\_max[0], resolution), np.linspace(X\_min[1], X\_max[1], resolution))  
 Z = cluster.predict(np.c\_[xx.ravel(), yy.ravel()])  
 Z = Z.reshape(xx.shape)  
 plt.contourf(Z, extent=(X\_min[0], X\_max[0], X\_min[1], X\_max[1]), cmap="Pastel2")  
 plt.contour(Z, extent=(X\_min[0], X\_max[0], X\_min[1], X\_max[1]), linewidths=1, colors="k")  
 plot\_data(X)  
 plot\_centroids(cluster.cluster\_centers\_)  
  
  
plt.figure(figsize=(10, 6))  
plot\_decision\_boundaries(kmeans, X)  
plt.show()  
  
# 评估方法：样本离最近聚类中心的总和:inertia。拐点附近的K更好

kmeans\_per\_k = [KMeans(n\_clusters=k, n\_init=10).fit(X) for k in range(1, 10)]  
inertias = [model.inertia\_ for model in kmeans\_per\_k]  
plt.plot(range(1, 10), inertias, "bo-")  
plt.show()  
  
# 评估方法：轮廓系数。越接近1越好  
silhouette\_scores = [silhouette\_score(X, model.labels\_) for model in kmeans\_per\_k[1:]] # 因为k=1无法计算si的值  
plt.plot(range(2, 10), silhouette\_scores, "bo-")  
plt.show()



**聚类在图像分割中的应用**

from matplotlib.image import imread  
from sklearn.cluster import KMeans  
import matplotlib.pyplot as plt  
  
image = imread("input/93088521.jpg")  
print(image)  
print(image.shape) # (2434, 1300, 3)  
X = image.reshape(-1, 3) # 将图像的每个像素点作为一个样本  
  
segmented\_imgs = []  
n\_colors = (10, 8, 6, 4, 2)  
for n\_cluster in n\_colors:  
 kmeans = KMeans(n\_clusters=n\_cluster, n\_init=10, random\_state=42)  
 kmeans.fit(X)  
 segmented\_img = kmeans.cluster\_centers\_[kmeans.labels\_].reshape(image.shape) / 255  
 segmented\_imgs.append(segmented\_img.reshape(image.shape))  
  
plt.figure(figsize=(10, 5))  
plt.subplot(161)  
plt.imshow(image)  
plt.title('Original image')  
  
for idx, n\_clusters in enumerate(n\_colors):  
 plt.subplot(162 + idx)  
 plt.imshow(segmented\_imgs[idx])  
 plt.title('{}colors'.format(n\_clusters))  
  
plt.show()



segmented\_img = kmeans.cluster\_centers\_[kmeans.labels\_].reshape(image.shape) / 255

这个代码的作用是：将图像像素分配给最近的簇中心，并用相应的中心值替换原始像素值，以生成分割后的图像。以n\_colors=10为例：

kmeans.cluster\_centers\_返回10个聚类中心，每个中心都是一个 RGB 值(红绿蓝三通道的值)的数组。

kmeans.labels\_是对每个样本点分配的簇标签，表示每个像素点属于哪个聚类中心。

利用kmeans.labels\_中的标签去索引kmeans.cluster\_centers\_，以找到每个像素点最接近的聚类中心。也就是，寻找图像中所有的像素点所对应的是哪个聚类中心。换句话说，就是将(2434, 1300, 3)个原始像素点映射到对应的(10, 3)的聚类中心。本次聚类中心是：

[[ 95.79026631 56.06818906 62.78858877]

[239.68942344 218.24742709 211.64805813]

[185.06630501 121.53136937 116.18169914]

[152.30885607 90.31655274 93.93635596]

[ 40.91918301 21.38834967 25.29352941]

[ 75.42078651 41.28879712 48.52008189]

[219.37994866 157.15874073 141.56587774]

[243.17167364 190.31619774 178.98773861]

[224.20816723 79.47032398 77.40751159]

[119.04504141 71.37239398 77.16417315]]

**采用半监督进行手写数字识别**

from sklearn.datasets import load\_digits  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import LogisticRegression  
from sklearn.cluster import KMeans  
import numpy as np  
import matplotlib.pyplot as plt  
  
# 查看数据集  
X\_digits, y\_digits = load\_digits(return\_X\_y=True)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_digits, y\_digits, random\_state=42)  
print(X\_train.shape) # (1347, 64)  
print(X\_test.shape) # (450, 64)  
  
# 选取50个完全随机的样本进行逻辑回归  
n\_labeled = 50  
log\_reg = LogisticRegression(max\_iter=10000, random\_state=42)  
np.random.seed(42)  
random\_indices = np.random.choice(X\_train.shape[0], n\_labeled, replace=False)  
log\_reg.fit(X\_train[random\_indices], y\_train[random\_indices])  
print("50个随机样本的得分：", log\_reg.score(X\_test, y\_test))  
  
# 先聚类成50个簇，然后自行打标签，再以这些有代表性的图像来训练  
k = n\_labeled # 之前的是随机选取的50个训练样本，现在尝试选择50个离簇中心最近的点  
kmeans = KMeans(n\_clusters=k, n\_init=10, random\_state=42)  
X\_digits\_dist = kmeans.fit\_transform(X\_train) # 距离矩阵X\_digits\_dist的每一行表示此数据点到每个簇中心的距离。  
representative\_digits\_idx = np.argmin(X\_digits\_dist, axis=0) # 50个离簇中心最近的样本的索引  
X\_representative\_digits = X\_train[representative\_digits\_idx]  
# 查看样本的样子，然后自行打标签  
plt.figure(figsize=(8, 2))  
for index, X\_representative\_digit in enumerate(X\_representative\_digits):  
 plt.subplot(k // 10, 10, index + 1)  
 plt.imshow(X\_representative\_digit.reshape(8, 8), cmap="binary", interpolation="bilinear")  
 plt.axis('off')  
plt.show()  
y\_representative\_digits = np.array([  
 3, 8, 5, 1, 0, 4, 6, 7, 1, 2,  
 4, 5, 9, 9, 6, 2, 1, 7, 2, 8,  
 6, 7, 9, 2, 0, 9, 2, 5, 4, 5,  
 0, 7, 1, 3, 7, 1, 2, 8, 4, 3,  
 7, 4, 5, 3, 3, 9, 1, 3, 6, 8])  
log\_reg = LogisticRegression(max\_iter=10000, random\_state=42)  
log\_reg.fit(X\_representative\_digits, y\_representative\_digits)  
print("50个有代表性样本的得分：", log\_reg.score(X\_test, y\_test))  
  
# 标签传播 认为在该簇内的样本的属性值就是簇中心点的属性值  
y\_train\_propagated = np.empty(len(X\_train), dtype=np.int32) # 创建一个未初始化的数组，数组的长度是X\_train的样本数量。  
for i in range(k):  
 y\_train\_propagated[kmeans.labels\_ == i] = y\_representative\_digits[i]  
log\_reg = LogisticRegression(max\_iter=10000, random\_state=42)  
log\_reg.fit(X\_train, y\_train\_propagated)  
print("标签传播到同一簇中的所有其他实例的得分：", log\_reg.score(X\_test, y\_test))  
  
# 不一定要选取簇中所有的样本，只考虑前20%试试  
percentile\_closest = 20  
X\_cluster\_dist = X\_digits\_dist[np.arange(len(X\_train)), kmeans.labels\_] # 记录每个样本到当前簇中心的距离  
for i in range(k):  
 cutoff\_distance = np.percentile(X\_cluster\_dist[kmeans.labels\_ == i], percentile\_closest) # 选择属于当前簇的所有样本，排序找到前20%  
 X\_cluster\_dist[(kmeans.labels\_ == i) & (X\_cluster\_dist > cutoff\_distance)] = -1 # &逐元素按位与，对于两个布尔数组，它返回一个新的布尔数组  
X\_train\_partially\_propagated = X\_train[X\_cluster\_dist != -1] # 使用布尔数组作为索引来选择数组中满足某些条件的元素  
y\_train\_partially\_propagated = y\_train\_propagated[X\_cluster\_dist != -1]  
log\_reg = LogisticRegression(max\_iter=10000, random\_state=42)  
log\_reg.fit(X\_train\_partially\_propagated, y\_train\_partially\_propagated)  
print("标签传播到同一簇中的前20%实例的得分：", log\_reg.score(X\_test, y\_test))

输出得分：

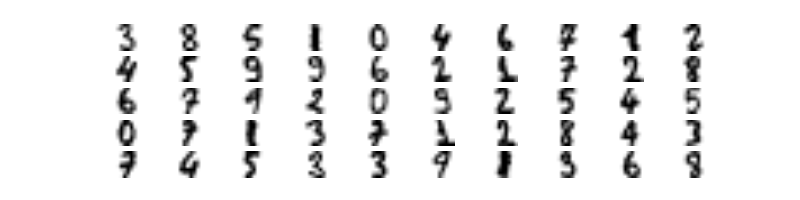
50个随机样本的得分： 0.8244444444444444

50个有代表性样本的得分： 0.9022222222222223

标签传播到同一簇中的所有其他实例的得分： 0.9155555555555556

标签传播到同一簇中的前20%实例的得分： 0.9244444444444444

手写数据采样的50个中心点是：

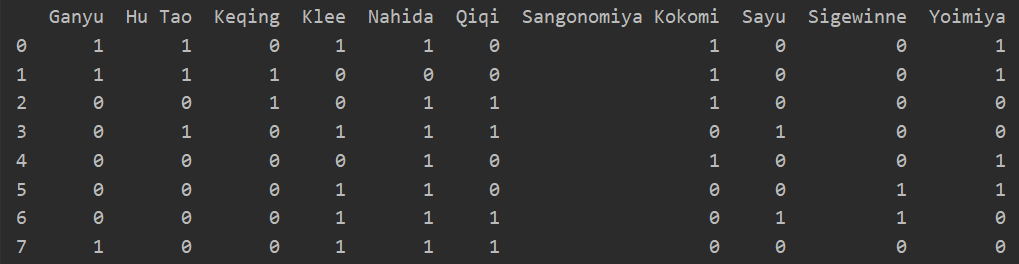


**10.关联规则**

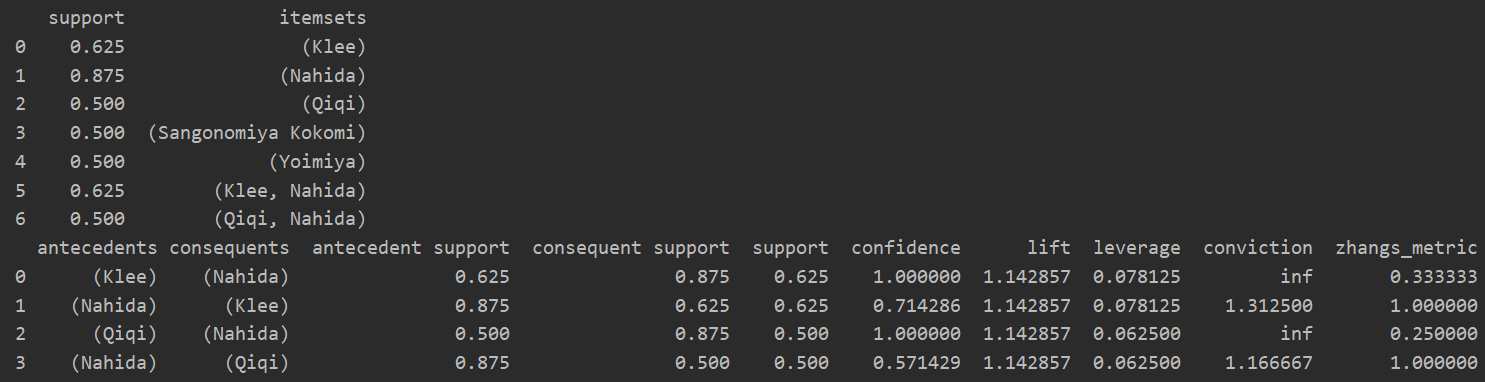
**自制数据集**

import pandas as pd  
from mlxtend.frequent\_patterns import apriori, association\_rules  
  
pd.set\_option('display.max\_rows', None)  
pd.set\_option('display.max\_columns', None)  
pd.set\_option('display.expand\_frame\_repr', False)  
  
likes\_raw\_data = {'ID': [1, 2, 3, 4, 5, 6, 7, 8],  
 'Likes': [['Klee', 'Hu Tao', 'Nahida', 'Sangonomiya Kokomi', 'Ganyu', 'Yoimiya'],  
 ['Keqing', 'Ganyu', 'Yoimiya', 'Hu Tao', 'Sangonomiya Kokomi'],  
 ['Sangonomiya Kokomi', 'Qiqi', 'Nahida', 'Keqing'],  
 ['Qiqi', 'Sayu', 'Klee', 'Nahida', 'Hu Tao'],  
 ['Yoimiya', 'Sangonomiya Kokomi', 'Nahida'],  
 ['Klee', 'Sigewinne', 'Yoimiya', 'Nahida'],  
 ['Klee', 'Nahida', 'Sigewinne', 'Sayu', 'Qiqi'],  
 ['Nahida', 'Ganyu', 'Klee', 'Qiqi'],  
 ]  
 }  
  
likes\_df = pd.DataFrame(likes\_raw\_data)  
print(likes\_df)  
# .str 是用于处理字符串的特殊属性，用于对Series中的每个元素进行字符串操作  
likes\_Likes = likes\_df['Likes'].str.join(',')  
print(likes\_Likes)  
# 将Series中的每个字符串按sep进行分割，并返回一个由虚拟(dummy)/指示(indicator)变量组成的DataFrame。  
# 即使分割时重复出现了某个字符串，也只记为1。  
likes\_Likes = likes\_Likes.str**.get\_dummies(',')**  
print(likes\_Likes)  
# 转换成bool类型，以避免DeprecationWarning。布尔类型的计算性能更好。  
likes\_Likes = likes\_Likes**.astype(bool)**# 设置最小支持度查找频繁项集  
likes\_frequent\_itemsets = **apriori**(likes\_Likes, min\_support=0.5, use\_colnames=True)  
print(likes\_frequent\_itemsets)  
# 设置其他衡量标准(metric)的最小阈值(min\_threshold)的关联规则  
likes\_association\_rules = **association\_rules**(likes\_frequent\_itemsets, metric='lift')  
print(likes\_association\_rules)  
print(likes\_association\_rules[(likes\_association\_rules['lift'] > 1) & (likes\_association\_rules['confidence'] > 0.7)])

先将[Nahida, Ganyu, Klee, Qiqi]变成这样的Nahida,Ganyu,Klee,Qiqi，得到指示矩阵：

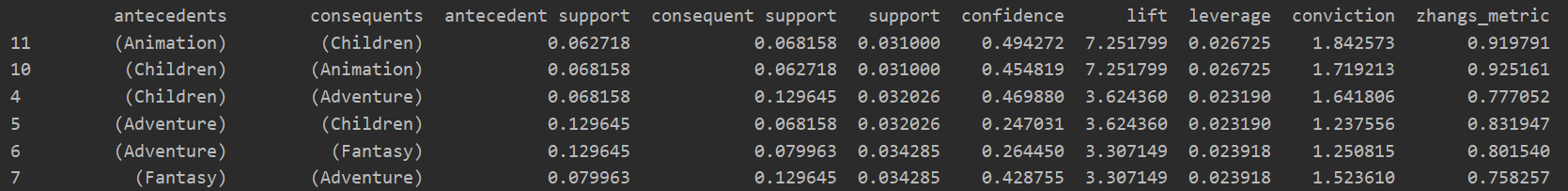


然后转为bool类型，得到频繁项集与关联规则：



**电影数据集**

import pandas as pd  
from mlxtend.frequent\_patterns import apriori, association\_rules  
  
pd.set\_option('display.max\_rows', None)  
pd.set\_option('display.max\_columns', None)  
pd.set\_option('display.expand\_frame\_repr', False)  
  
movies\_raw\_data = pd.read\_csv('ml-latest-small/movies.csv')  
movies\_genres = movies\_raw\_data['genres'].str.get\_dummies('|').astype(bool)  
print(movies\_genres.shape) # (9742, 20)  
genres\_freq = apriori(movies\_genres, use\_colnames=True, min\_support=0.03)  
print(genres\_freq.sort\_values(by='support', ascending=False))  
genres\_rules = association\_rules(genres\_freq, metric='lift', min\_threshold=2)  
print(genres\_rules.sort\_values(by='lift', ascending=False))



可以看出Animation和Children有强关联性。

**11.集成**

**集成算法**

from sklearn.model\_selection import cross\_val\_score  
from sklearn.datasets import load\_iris  
from sklearn.ensemble import AdaBoostClassifier  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.ensemble import BaggingClassifier  
from sklearn.ensemble import RandomForestClassifier  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
clf = AdaBoostClassifier(n\_estimators=100) # 迭代次数为100  
scores = cross\_val\_score(clf, X, Y) # 交叉验证  
print('AdaBoost准确率：', scores.mean())  
  
bagging = BaggingClassifier(KNeighborsClassifier(), max\_samples=0.5, max\_features=0.5) # 结合KNN分类器，每个基分类器抽样样本数量和特征数量的最大比例为0.5  
scores = cross\_val\_score(bagging, X, Y)  
print('bagging准确率：', scores.mean())  
  
clf = RandomForestClassifier(n\_estimators=10, max\_features=2) # 决策树的数量为10棵，每棵树最大的特征数量为2  
scores = cross\_val\_score(clf, X, Y)  
print('RF准确率：', scores.mean())

输出

AdaBoost准确率： 0.9466666666666665

bagging准确率： 0.9533333333333334

RF准确率： 0.9533333333333334

**硬投票&软投票**

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.model\_selection import train\_test\_split  
from sklearn.datasets import make\_moons  
from sklearn.ensemble import RandomForestClassifier, VotingClassifier, BaggingClassifier  
from sklearn.linear\_model import LogisticRegression  
from sklearn.svm import SVC  
from sklearn.metrics import accuracy\_score  
from sklearn.tree import DecisionTreeClassifier  
from matplotlib.colors import ListedColormap  
  
X, y = make\_moons(n\_samples=500, noise=0.30, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)  
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'yo', alpha=0.6)  
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs', alpha=0.6)  
plt.show()  
  
# 硬投票  
log\_clf = LogisticRegression(random\_state=42)  
rnd\_clf = RandomForestClassifier(random\_state=42)  
svm\_clf = SVC(random\_state=42)  
**voting\_clf = VotingClassifier(estimators=[('lr', log\_clf), ('rf', rnd\_clf), ('svc', svm\_clf)], voting='hard')**  
voting\_clf.fit(X\_train, y\_train)  
for clf in (log\_clf, rnd\_clf, svm\_clf, voting\_clf):  
 clf.fit(X\_train, y\_train)  
 y\_pred = clf.predict(X\_test)  
 print(clf.\_\_class\_\_.\_\_name\_\_, accuracy\_score(y\_test, y\_pred))  
  
# 软投票  
log\_clf = LogisticRegression(random\_state=42)  
rnd\_clf = RandomForestClassifier(random\_state=42)  
svm\_clf = SVC(probability=True, random\_state=42) # SVM默认不计算概率值，但软投票需要知道概率值。  
**voting\_clf = VotingClassifier(estimators=[('lr', log\_clf), ('rf', rnd\_clf), ('svc', svm\_clf)], voting='soft')**  
voting\_clf.fit(X\_train, y\_train)  
for clf in (log\_clf, rnd\_clf, svm\_clf, voting\_clf):  
 clf.fit(X\_train, y\_train)  
 y\_pred = clf.predict(X\_test)  
 print(clf.\_\_class\_\_.\_\_name\_\_, accuracy\_score(y\_test, y\_pred))

输出

LogisticRegression 0.864

RandomForestClassifier 0.896

SVC 0.896

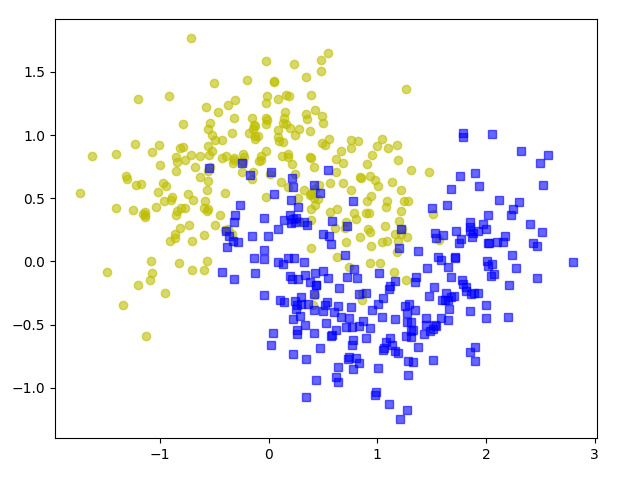
VotingClassifier 0.912

LogisticRegression 0.864

RandomForestClassifier 0.896

SVC 0.896

VotingClassifier 0.92



**bagging(多次采样，分别训练多个模型，预测时用所有模型结果进行集成)**

# bagging  
tree\_clf = DecisionTreeClassifier(random\_state=42)  
tree\_clf.fit(X\_train, y\_train)  
y\_pred\_tree = tree\_clf.predict(X\_test)  
print(accuracy\_score(y\_test, y\_pred\_tree))  
bag\_clf = BaggingClassifier(DecisionTreeClassifier(),  
 n\_estimators=500, # 集成的基础分类器的数量  
 max\_samples=100, # （有放回抽样）从训练集中抽取的样本数量  
 bootstrap=True, # 随机采样  
 # n\_jobs=-1, # 用全部CPU核心进行训练  
 random\_state=42  
 )  
bag\_clf.fit(X\_train, y\_train)  
y\_pred = bag\_clf.predict(X\_test)  
print(accuracy\_score(y\_test, y\_pred))  
  
def plot\_decision\_boundary(clf, X, y, axes=(-1.5, 2.5, -1, 1.5)):  
 x1s = np.linspace(axes[0], axes[1], 100)  
 x2s = np.linspace(axes[2], axes[3], 100)  
 x1, x2 = np.meshgrid(x1s, x2s)  
 X\_new = np.c\_[x1.ravel(), x2.ravel()]  
 y\_pred = clf.predict(X\_new).reshape(x1.shape)  
 custom\_cmap = ListedColormap(['#66ccff', '#ee0000', '#39c5bb'])  
 plt.contourf(x1, x2, y\_pred, cmap=custom\_cmap, alpha=0.3)  
 plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'yo', alpha=0.6)  
 plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs', alpha=0.6)  
 plt.axis(axes)  
 plt.xlabel('x1')  
 plt.xlabel('x2')  
  
plt.figure(figsize=(12, 5))  
plt.subplot(121)  
plot\_decision\_boundary(tree\_clf, X, y)  
plt.title('Decision Tree')  
plt.subplot(122)  
plot\_decision\_boundary(bag\_clf, X, y)  
plt.title('Decision Tree With Bagging')  
plt.show()  
  
bag\_clf = BaggingClassifier(DecisionTreeClassifier(),  
 n\_estimators=500,  
 max\_samples=100,  
 bootstrap=True,  
 random\_state=42,  
 **oob\_score=True**  
 )  
bag\_clf.fit(X\_train, y\_train)  
**print(bag\_clf.oob\_score\_)** # 在训练过程中用每次没被训练到的数据来获取模型的性能估计  
y\_pred = bag\_clf.predict(X\_test)  
print(accuracy\_score(y\_test, y\_pred)) # 使用独立的测试集来评估模型性能  
# print(bag\_clf.oob\_decision\_function\_) # 每个样本属于对应类别的概率值

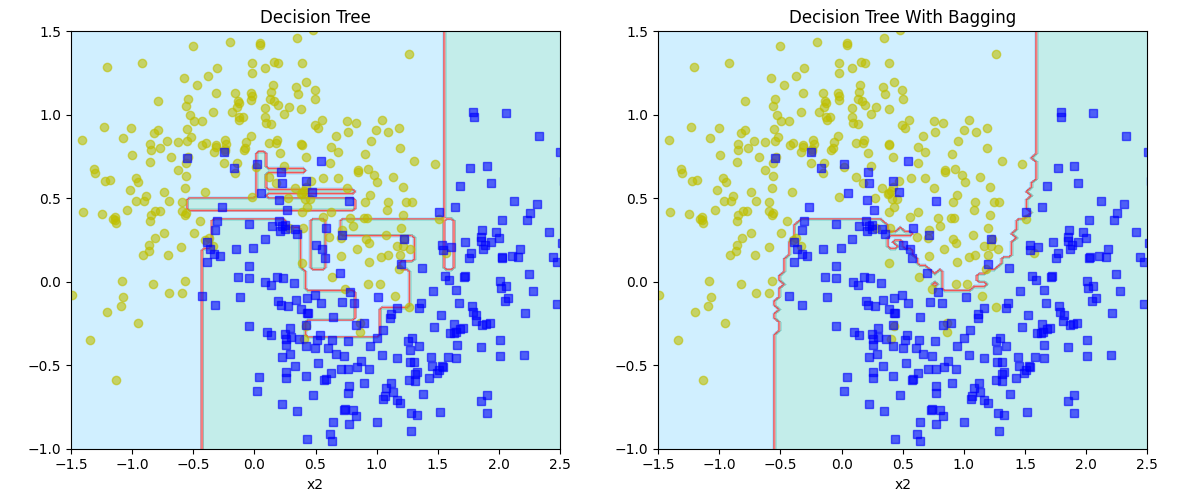
输出

0.856

0.904

0.9253333333333333

0.904



**用随机森林进行特征重要性判断**

import matplotlib.pyplot as plt  
from sklearn.ensemble import RandomForestClassifier  
from sklearn.datasets import load\_iris, fetch\_openml  
  
iris = load\_iris()  
rf\_clf = RandomForestClassifier(n\_estimators=500)  
rf\_clf.fit(iris['data'], iris['target'])  
for name, score in zip(iris['feature\_names'], rf\_clf.feature\_importances\_):  
 print(name, score)  
  
mnist = fetch\_openml('MNIST\_784', parser='auto')  
rf\_clf = RandomForestClassifier(n\_estimators=500, n\_jobs=-1) # 任务量大的时候才指定为-1，这时候能节省时间。任务量小的时候指定为-1反而增加开销。  
rf\_clf.fit(mnist['data'], mnist['target'])  
print(rf\_clf**.feature\_importances\_**.shape)  
image = rf\_clf.feature\_importances\_.reshape(28, 28)  
plt.imshow(image, cmap='hot')  
plt.axis('off')  
char = plt.colorbar(ticks=[rf\_clf.feature\_importances\_.min(), rf\_clf.feature\_importances\_.max()])  
# char.ax.set\_yticklabels(['Not important','Very important']) # 更改坐标标签文字  
plt.show()

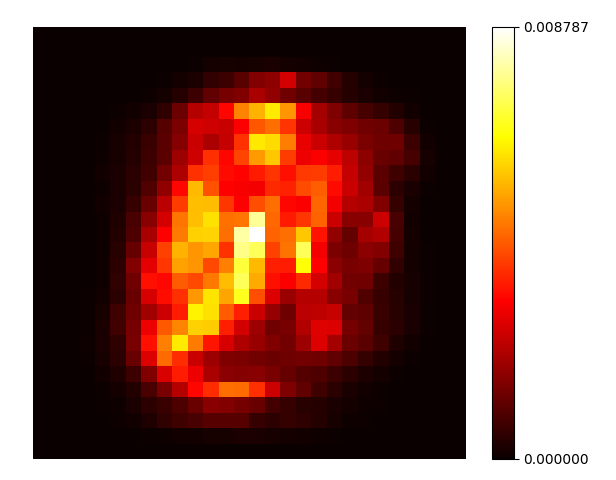
输出

sepal length (cm) 0.10156108390408021

sepal width (cm) 0.02390628173194181

petal length (cm) 0.4432523934459005

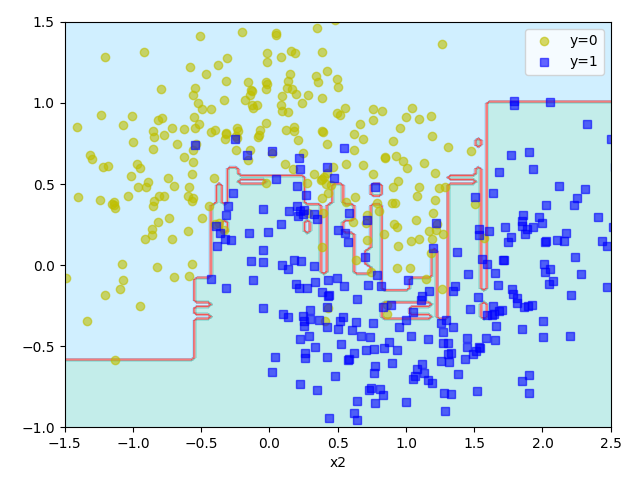
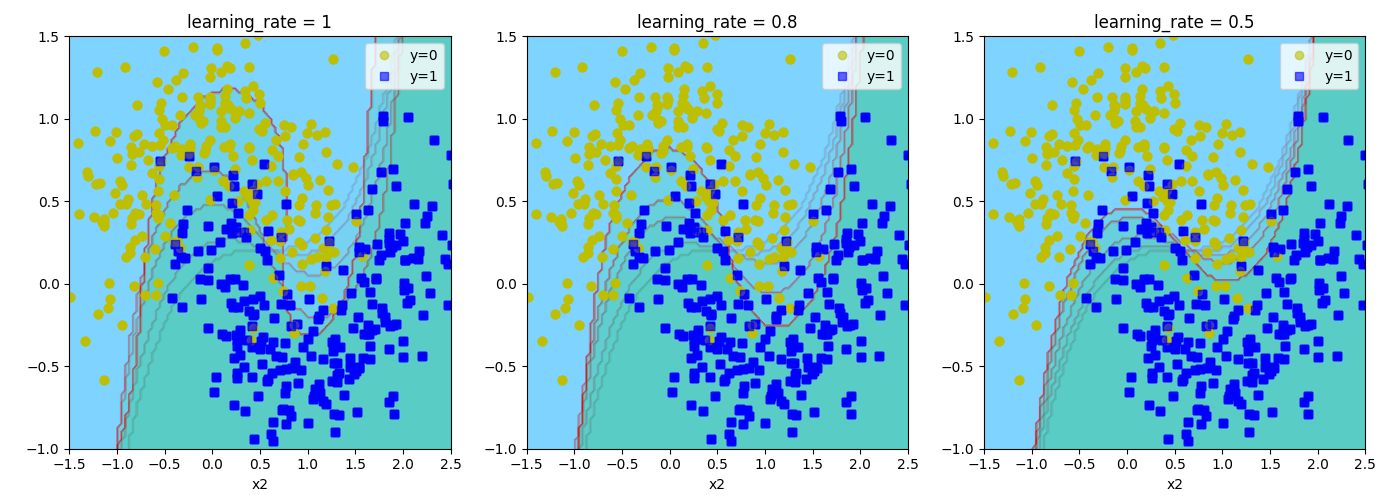
petal width (cm) 0.43128024091807743



MNIST\_784是70000个28\*28=784个像素点的手写数据集。

**模拟AdaBoost和AdaBoost**

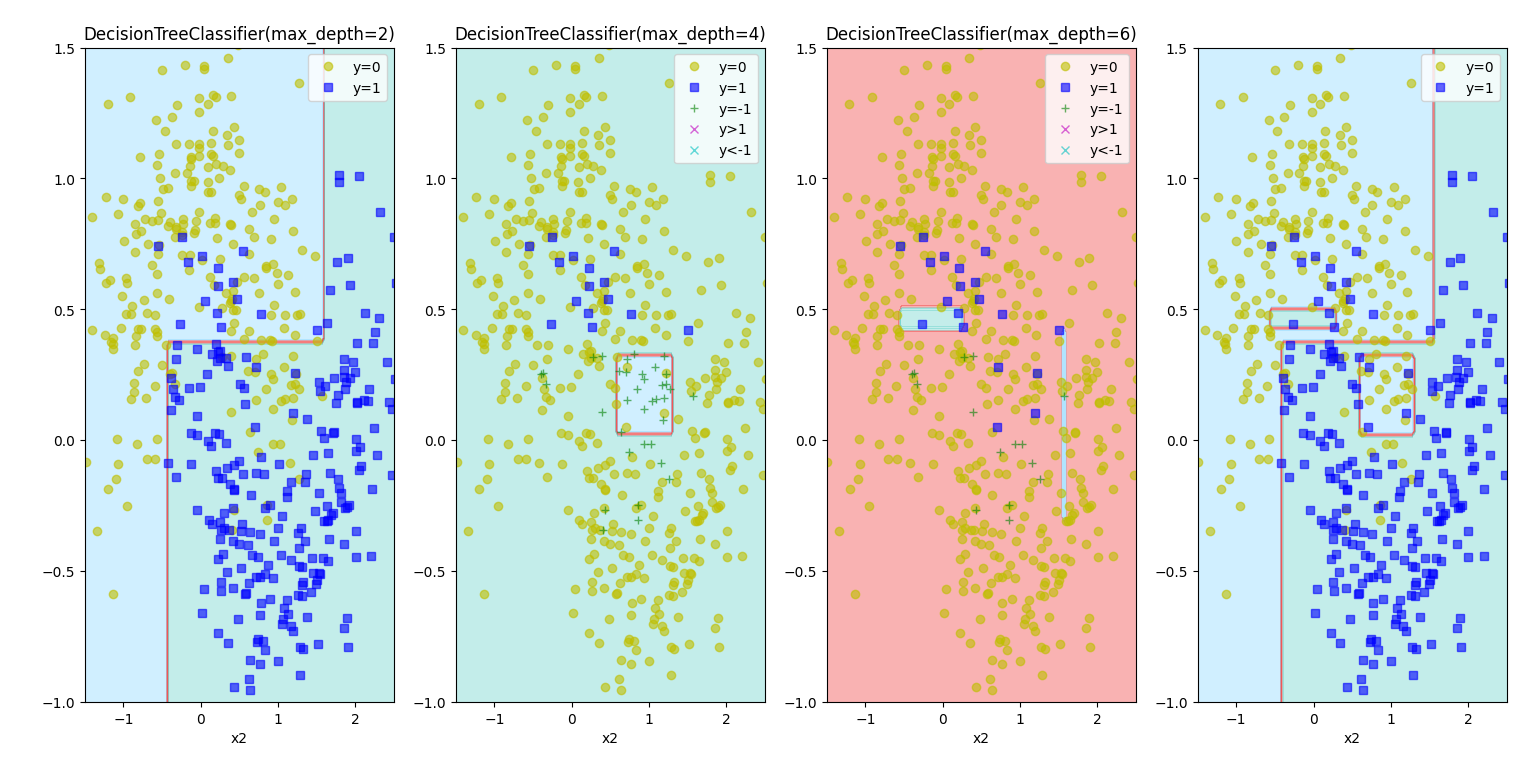
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.colors import ListedColormap  
from sklearn.model\_selection import train\_test\_split  
from sklearn.datasets import make\_moons  
from sklearn.metrics import mean\_squared\_error, accuracy\_score  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.svm import SVC  
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier  
  
X, y = make\_moons(n\_samples=500, noise=0.30, random\_state=42)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)  
def plot\_decision\_boundary(clf, X, y, axes=(-1.5, 2.5, -1, 1.5), show\_legend=True, show\_other\_label=False):  
 x1s = np.linspace(axes[0], axes[1], 100)  
 x2s = np.linspace(axes[2], axes[3], 100)  
 x1, x2 = np.meshgrid(x1s, x2s)  
 X\_new = np.c\_[x1.ravel(), x2.ravel()]  
 y\_pred = clf.predict(X\_new).reshape(x1.shape)  
 custom\_cmap = ListedColormap(['#66ccff', '#ee0000', '#39c5bb'])  
 plt.contourf(x1, x2, y\_pred, cmap=custom\_cmap, alpha=0.3)  
 plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'yo', alpha=0.6, label='y=0')  
 plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs', alpha=0.6, label='y=1')  
 if show\_other\_label:  
 plt.plot(X[:, 0][y == -1], X[:, 1][y == -1], 'g+', alpha=0.6, label='y=-1')  
 plt.plot(X[:, 0][y > 1], X[:, 1][y > 1], 'mx', alpha=0.6, label='y>1')  
 plt.plot(X[:, 0][y < -1], X[:, 1][y < -1], 'cx', alpha=0.6, label='y<-1')  
 if show\_legend:  
 plt.legend() # 显示图例  
 plt.axis(axes)  
 plt.xlabel('x1')  
 plt.xlabel('x2')  
  
# 模拟AdaBoost  
plt.figure(figsize=(14, 5))  
for subplot, learning\_rate in ((131, 1), (132, 0.8), (133, 0.5)):  
 sample\_weights = np.ones(len(X\_train)) # 样本权重矩阵  
 plt.subplot(subplot)  
 for i in range(5):  
 # 以SVM分类器模拟5次AdaBoost  
 svm\_clf = SVC(kernel='rbf', C=0.05, random\_state=42) # 高斯核函数rbf，正则化参数0.05  
 svm\_clf.fit(X\_train, y\_train, sample\_weight=sample\_weights)  
 y\_pred = svm\_clf.predict(X\_train)  
 sample\_weights[y\_pred != y\_train] \*= (1 + learning\_rate) # 错分的权重升高  
 plot\_decision\_boundary(svm\_clf, X, y, show\_legend=(bool(i == 0)))  
 plt.title('learning\_rate = {}'.format(learning\_rate))  
plt.show()  
  
# AdaBoost  
ada\_clf = AdaBoostClassifier(DecisionTreeClassifier(max\_depth=1),  
 n\_estimators=500,  
 learning\_rate=0.5,  
 random\_state=42  
 )  
ada\_clf.fit(X\_train, y\_train)  
plot\_decision\_boundary(ada\_clf, X, y)  
plt.show()



**模拟Gradient Boosting（同理可用于回归问题，每次都去预测新的误差）**

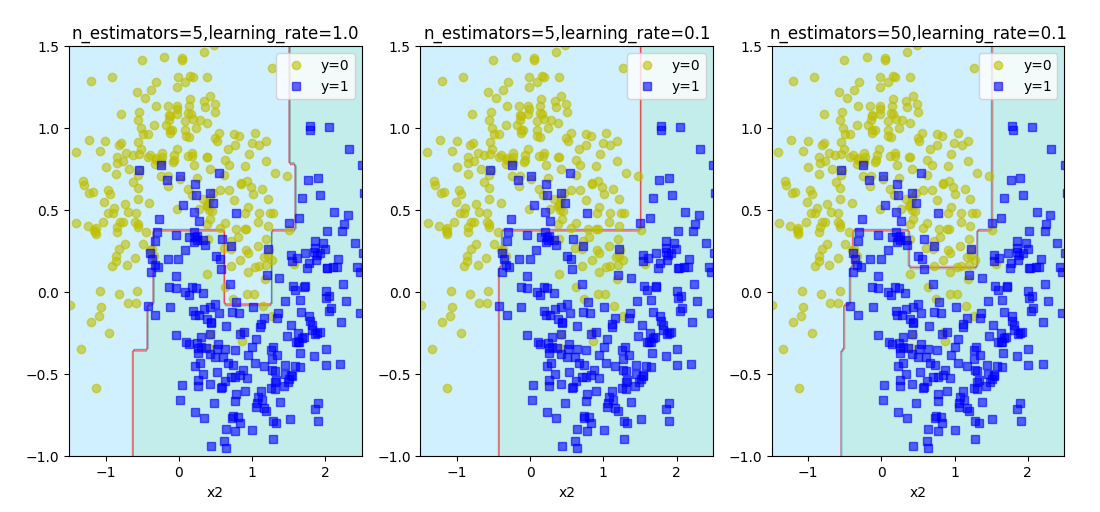
tree\_reg1 = DecisionTreeClassifier(max\_depth=2)  
tree\_reg1.fit(X, y)  
y2 = y - tree\_reg1.predict(X)  
tree\_reg2 = DecisionTreeClassifier(max\_depth=4)  
tree\_reg2.fit(X, y2)  
y3 = y2 - tree\_reg2.predict(X)  
tree\_reg3 = DecisionTreeClassifier(max\_depth=6)  
tree\_reg3.fit(X, y3)  
for subplot, tree\_reg, y\_n in ((141, tree\_reg1, y), (142, tree\_reg2, y2), (143, tree\_reg3, y3)):  
 plt.subplot(subplot)  
 plot\_decision\_boundary(tree\_reg, X, y\_n, show\_other\_label=bool(subplot != 141))  
 plt.title('{}'.format(tree\_reg))  
plt.subplot(144)  
x1s = np.linspace(-1.5, 3.5, 100)  
x2s = np.linspace(-1, 1.5, 100)  
x1, x2 = np.meshgrid(x1s, x2s)  
X\_new = np.c\_[x1.ravel(), x2.ravel()]  
y\_pred = sum(tree.predict(X\_new).reshape(x1.shape) for tree in (tree\_reg1, tree\_reg2, tree\_reg3))  
custom\_cmap = ListedColormap(['#66ccff', '#ee0000', '#39c5bb'])  
plt.contourf(x1, x2, y\_pred, cmap=custom\_cmap, alpha=0.3)  
plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], 'yo', alpha=0.6, label='y=0')  
plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], 'bs', alpha=0.6, label='y=1')  
plt.legend() # 显示图例  
plt.axis((-1.5, 2.5, -1, 1.5))  
plt.xlabel('x1')  
plt.xlabel('x2')  
plt.show()

每次去预测前一个样本的误差：



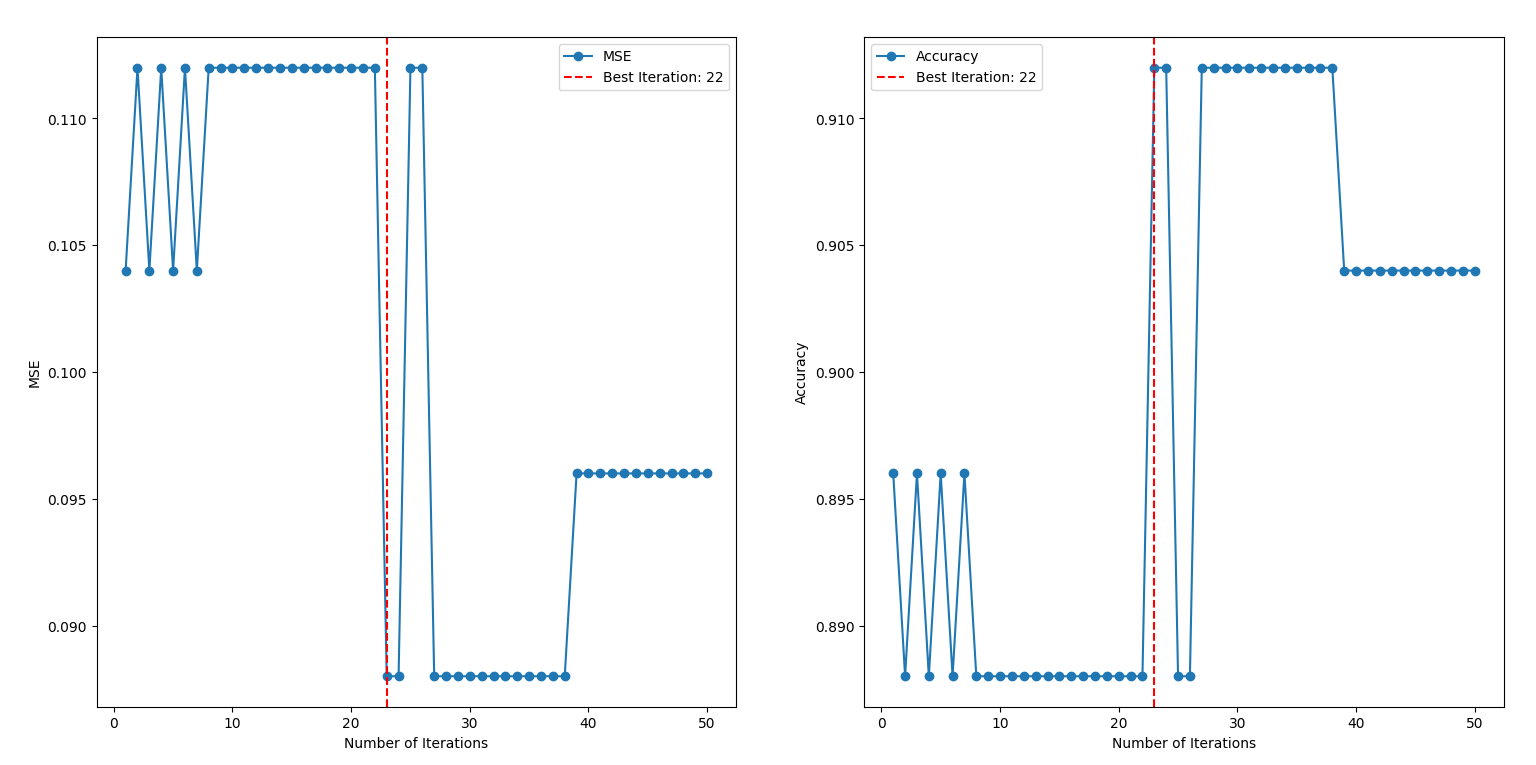
**Gradient Boost**

gbclf1 = GradientBoostingClassifier(max\_depth=2,  
 n\_estimators=5,  
 learning\_rate=1.0,  
 random\_state=42  
 )  
gbclf1.fit(X, y)  
gbclf2 = GradientBoostingClassifier(max\_depth=2,  
 n\_estimators=5,  
 learning\_rate=0.1,  
 random\_state=42  
 )  
gbclf2.fit(X, y)  
gbclf3 = GradientBoostingClassifier(max\_depth=2,  
 n\_estimators=50,  
 learning\_rate=0.1,  
 random\_state=42  
 )  
gbclf3.fit(X, y)  
plt.figure(figsize=(11, 4))  
for subplot, gbclf in ((131, gbclf1), (132, gbclf2), (133, gbclf3)):  
 plt.subplot(subplot)  
 plot\_decision\_boundary(gbclf, X, y)  
 plt.title("n\_estimators={},learning\_rate={}".format(gbclf.n\_estimators, gbclf.learning\_rate))  
plt.show()  
  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, random\_state=42)  
gbclf = GradientBoostingClassifier(max\_depth=2,  
 n\_estimators=50,  
 random\_state=42  
 )  
gbclf.fit(X\_train, y\_train)



**通过误差提前停止**

# 1.MSE  
# for i, y\_pred\_i in enumerate(gbclf.staged\_predict(X\_val)):  
# # 在这里，y\_pred\_i 是模型在第 i 次迭代后的预测结果  
# error = mean\_squared\_error(y\_val, y\_pred\_i)  
# print(f'第 {i+1} 次, MSE: {error}')  
errors = [mean\_squared\_error(y\_val, y\_pred) for y\_pred in gbclf.staged\_predict(X\_val)]  
print(np.argmin(errors)) # 22，也就是第23次  
# 2.Accuracy  
accuracies = [**accuracy\_score**(y\_val, y\_pred) for y\_pred in gbclf.staged\_predict(X\_val)]  
print(np.argmax(accuracies)) # 22，也就是第23次  
# 绘制误差/精度曲线  
plt.figure(figsize=(10, 6))  
plt.subplot(121)  
plt.plot(range(1, len(errors) + 1), errors, label='MSE', marker='o')  
plt.axvline(x=np.argmin(errors)+1, color='red', linestyle='--', label=f'Best Iteration: {np.argmin(errors)}')  
plt.xlabel('Number of Iterations')  
plt.ylabel('MSE')  
plt.legend()  
plt.subplot(122)  
plt.plot(range(1, len(accuracies) + 1), accuracies, label='Accuracy', marker='o')  
plt.axvline(x=np.argmax(accuracies)+1, color='red', linestyle='--', label=f'Best Iteration: {np.argmax(accuracies)}')  
plt.xlabel('Number of Iterations')  
plt.ylabel('Accuracy')  
plt.legend()  
plt.show()



**warm\_start**

gbclf = GradientBoostingClassifier(max\_depth=2,  
 random\_state=42,  
 warm\_start=True  
 )  
error\_going\_up = 0  
max\_accuracy = 0  
# min\_val\_error = float('inf')  
for n\_estimators in range(1, 120):  
 gbclf.n\_estimators = n\_estimators  
 gbclf.fit(X\_train, y\_train)  
 y\_pred = gbclf.predict(X\_val)  
 accuracy = accuracy\_score(y\_val, y\_pred)  
 if accuracy > max\_accuracy:  
 max\_accuracy = accuracy  
 error\_going\_up = 0  
 else:  
 error\_going\_up += 1  
 if error\_going\_up == 30:  
 # 至少有30次没有新的MAX(accuracy)才算到了max。但是30次这个间隔不一定好  
 break  
print(gbclf.n\_estimators-30) # 23

warm\_start能在每次fit时从上一次的训练开始继续训练。

**Stacking堆叠集成**

import numpy as np  
from sklearn.datasets import fetch\_openml  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier  
from sklearn.svm import LinearSVC  
from sklearn.neural\_network import MLPClassifier  
  
mnist = fetch\_openml('MNIST\_784', parser='auto')  
# 首先将MNIST数据集(包括图像和标签)分成训练验证集和测试集test。接着划分训练验证集得到训练集train和验证集val。  
X\_train\_val, X\_test, y\_train\_val, y\_test = train\_test\_split(mnist.data, mnist.target, test\_size=10000, random\_state=42)  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_train\_val, y\_train\_val, test\_size=10000, random\_state=42)  
  
random\_forest\_clf = RandomForestClassifier(random\_state=42)  
extra\_trees\_clf = ExtraTreesClassifier(random\_state=42)  
svm\_clf = LinearSVC(random\_state=42, dual=True, max\_iter=2000)  
mlp\_clf = MLPClassifier(random\_state=42)  
estimators = [random\_forest\_clf, extra\_trees\_clf, svm\_clf, mlp\_clf]  
  
for estimator in estimators:  
 print("Training:", estimator)  
 estimator.fit(X\_train, y\_train)  
  
X\_val\_pred = np.empty((len(X\_val), len(estimators)), dtype=np.float32)  
for index, estimator in enumerate(estimators):  
 X\_val\_pred[:, index] = estimator.predict(X\_val) # 对手写数字验证集的预测，每行是4个模型的对应预测  
  
rdf = RandomForestClassifier(n\_estimators=200, oob\_score=True, random\_state=42)  
rdf.fit(X\_val\_pred, y\_val)  
print(rdf.oob\_score\_) # 0.9663  
  
X\_test\_pred = np.empty((len(X\_test), len(estimators)), dtype=np.float32)  
for index, estimator in enumerate(estimators):  
 X\_test\_pred[:, index] = estimator.predict(X\_test)  
y\_test\_pred = rdf.predict(X\_test\_pred)  
stacking\_accuracy = accuracy\_score(y\_test, y\_test\_pred)  
print(f"Stacking Model Accuracy: {stacking\_accuracy}") # 0.9664

**性能比较**

from sklearn.datasets import fetch\_20newsgroups  
from sklearn.feature\_extraction.text import CountVectorizer  
from sklearn.feature\_extraction.text import TfidfTransformer  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.svm import SVC  
from sklearn.linear\_model import LogisticRegression  
from sklearn.ensemble import VotingClassifier  
  
categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']  
twenty\_train = fetch\_20newsgroups(subset='train', categories=categories, shuffle=True, random\_state=42)  
count\_vect = CountVectorizer()  
X\_train\_counts = count\_vect.fit\_transform(twenty\_train.data)  
tfidf\_transformer = TfidfTransformer()  
X\_train\_tfidf = tfidf\_transformer.fit\_transform(X\_train\_counts)  
twenty\_test = fetch\_20newsgroups(subset='test', categories=categories, shuffle=True, random\_state=42)  
X\_test\_counts = count\_vect.transform(twenty\_test.data)  
X\_test\_tfidf = tfidf\_transformer.transform(X\_test\_counts)  
  
  
def Vot\_clf(voting\_type):  
 log\_clf = LogisticRegression(penalty='l2', solver='newton-cg')  
 tree\_clf = DecisionTreeClassifier(criterion='entropy')  
 svm\_clf = SVC(kernel='linear', probability=True)   
 return VotingClassifier([('lr', log\_clf), ('tree', tree\_clf), ('svc', svm\_clf)], voting=voting\_type, weights=[2, 1, 3])  
  
  
def Acc\_of\_clf(voting\_type):  
 log\_clf = LogisticRegression()  
 tree\_clf = DecisionTreeClassifier(criterion='entropy')  
 svm\_clf = SVC(kernel='linear', probability=True)  
 vot\_clf = Vot\_clf(voting\_type)  
 for clf in (log\_clf, tree\_clf, svm\_clf, vot\_clf):  
 clf.fit(X\_train\_tfidf, twenty\_train.target)  
 print(clf.\_\_class\_\_.\_\_name\_\_, "准确率%.4f" % (clf.score(X\_test\_tfidf, twenty\_test.target)))  
  
  
Acc\_of\_clf('soft')

输出

LogisticRegression 准确率0.8975

DecisionTreeClassifier 准确率0.6897

SVC 准确率0.9208

VotingClassifier 准确率0.9288

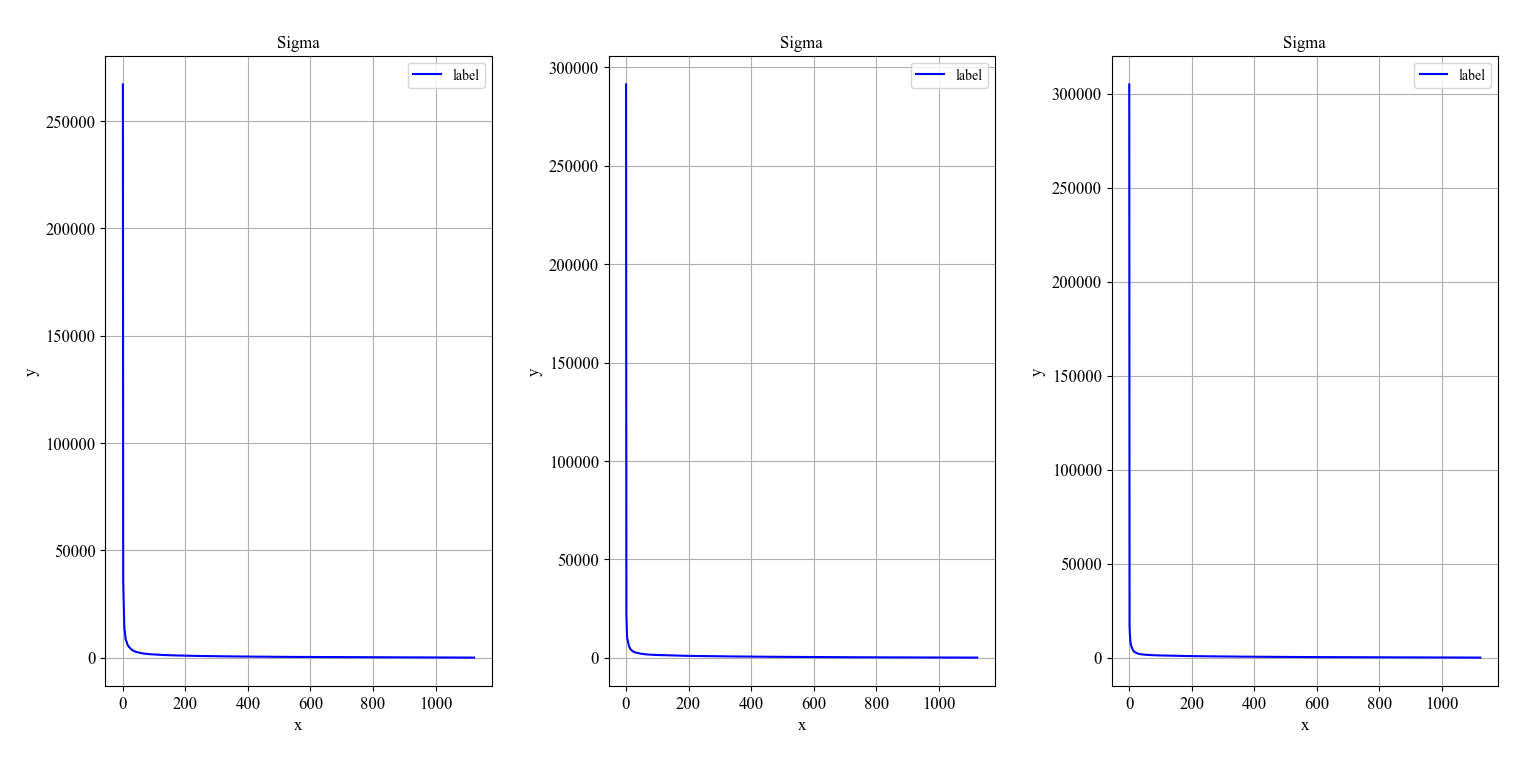
用于比较逻辑回归、决策树、支持向量机和集成投票分类器在20news文本数据集上的性能。

**12.线性代数**

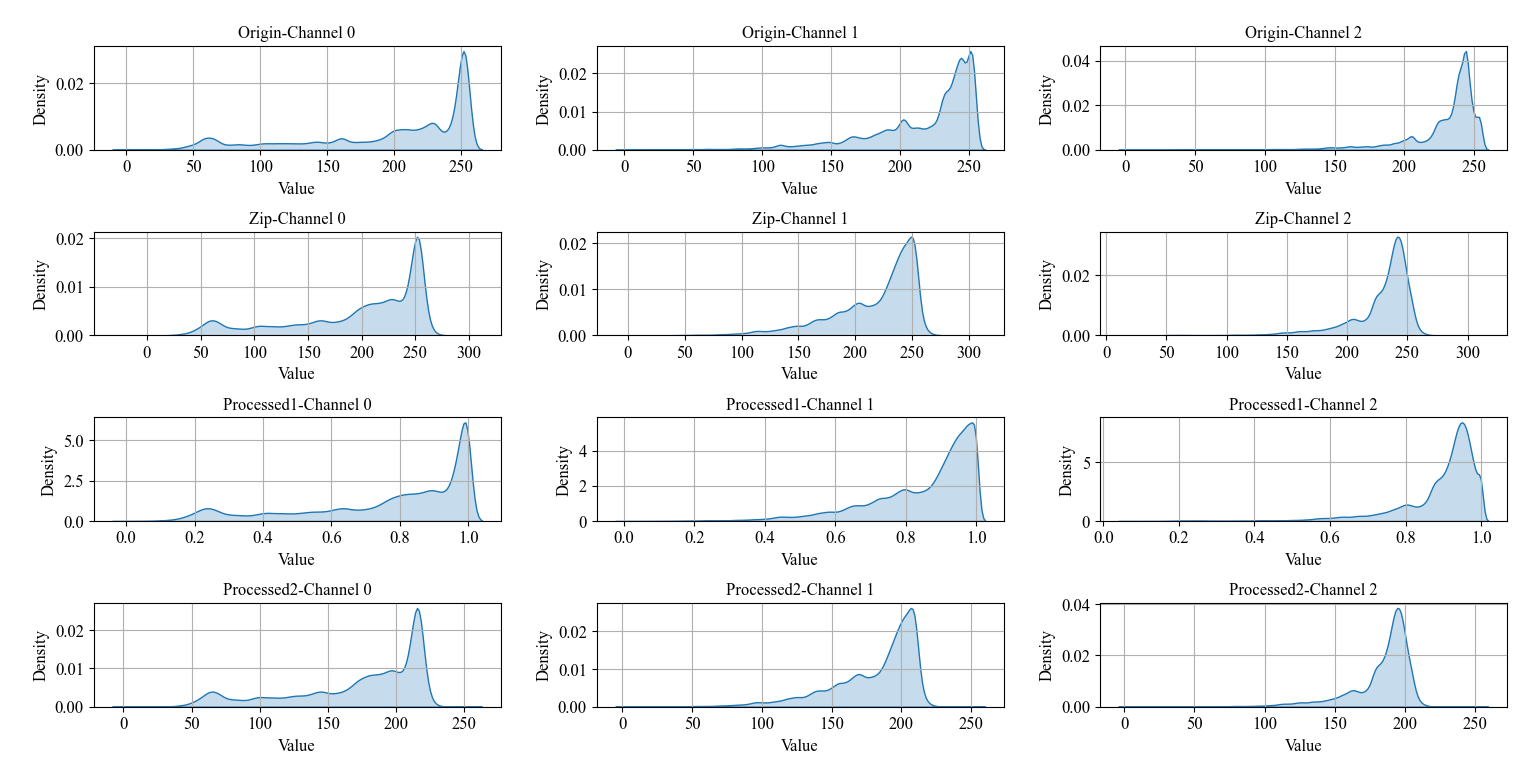
**SVD实现图片压缩里的颜色细节**

import cv2  
import numpy as np  
import matplotlib.pyplot as plt  
  
from easier\_excel.draw\_data import draw\_density, plot\_xy  
  
def read\_image(img\_path, gray\_pic=False, show\_details=False):  
 *"""  
 读取图片* ***:param*** *img\_path: 图像路径* ***:param*** *gray\_pic: 是否读取灰度图像* ***:param*** *show\_details: 是否输出图片的shape以及显示图片* ***:return****: 图像数组，类型为np.ndarray  
 """* if gray\_pic:  
 img = cv2.imread(path, cv2.IMREAD\_GRAYSCALE)  
 else:  
 img\_gbr = cv2.imread(img\_path)  
 img = cv2.cvtColor(img\_gbr, cv2.COLOR\_BGR2RGB)  
 if show\_details:  
 print(img.shape)  
 plt.imshow(img)  
 plt.show()  
 plt.close()  
 return img  
  
  
def zip\_image\_by\_svd(origin\_image, rate=0.8, channel=3, show\_img=True):  
 *"""  
 使用SVD对图像进行压缩* ***:param*** *origin\_image: 传入np.ndarray类型的图像数组* ***:param*** *rate: 保留率* ***:param*** *channel: 通道数* ***:param*** *show\_img: 是否显示压缩前后的图片* ***:return*** *zip\_img: 压缩后的图像  
 """* zip\_img = np.zeros(origin\_image.shape) # 用于存储压缩后的图像  
 u\_shape, s\_shape, vT\_shape, n\_sigmas = 0, 0, 0, 0  
  
 fig, axs = plt.subplots(1, 3)  
 for chan in range(channel):  
 # 对每层进行SVD分解  
 U, Sigma, VT = np.linalg.svd(origin\_image[:, :, chan])  
 # 计算达到保留率需要的奇异值数量  
 total\_Sigma = np.sum(Sigma)  
 cum\_Sigma = np.cumsum(Sigma)  
 n\_sigmas = np.argmax(cum\_Sigma >= rate \* total\_Sigma) + 1  
 Sigma\_n = np.diag(Sigma[:n\_sigmas])  
 zip\_img[:, :, chan] = np.dot(U[:, :n\_sigmas], np.dot(Sigma\_n, VT[:n\_sigmas, :]))  
 # 记录每个矩阵的shape  
 u\_shape = U[:, 0:n\_sigmas].shape  
 s\_shape = Sigma\_n.shape  
 vT\_shape = VT[0:n\_sigmas, :].shape  
 plot\_xy(x=np.arange(Sigma.shape[0]), y=Sigma, title='Sigma', show\_plt=False, ax=axs[chan], use\_ax=True,  
 font\_name='Times New Roman')  
 plt.show()  
 plt.close()  
  
 fig, axs = plt.subplots(4, 3)  
 for i in range(channel):  
 axs[0][i] = draw\_density(origin\_image[:, :, i].reshape(-1), ax=axs[0][i], show\_plt=False,  
 title=f'Origin-Channel {i}', show\_legend=False)  
 for i in range(channel):  
 axs[1][i] = draw\_density(zip\_img[:, :, i].reshape(-1), ax=axs[1][i], show\_plt=False,  
 title=f'Zip-Channel {i}', show\_legend=False)  
  
 # 这里暂时没想到更好的方法，应该是让zip\_img更接近origin\_image的值，使得颜色差异小  
 # 如果使用zip\_img就是归一化到[0, 1]，但这里使用origin\_image的最大最小值来进行归一化，是为了减少颜色差异  
 # 使用zip\_img[:, :, i] = (O\_MAX-O\_MIN) \* (zip\_img[:, :, i] - Z\_MIN) / (Z\_MAX - Z\_MIN) + O\_MIN  
 # 可能不如只使用O\_MAX和O\_MIN，猜测是Z\_MIN和Z\_MAX的值是离群点  
 zip\_img1 = np.zeros(zip\_img.shape)  
 zip\_img2 = np.zeros(zip\_img.shape)  
 for i in range(channel):  
 O\_MAX = np.max(origin\_image[:, :, i])  
 O\_MIN = np.min(origin\_image[:, :, i])  
 Z\_MIN = np.min(zip\_img[:, :, i])  
 Z\_MAX = np.max(zip\_img[:, :, i])  
 zip\_img1[:, :, i] = (zip\_img[:, :, i] - O\_MIN) / (O\_MAX - O\_MIN)  
 # print("min-max:", np.min(zip\_img[:, :, i]), np.max(zip\_img[:, :, i]))  
 zip\_img1[zip\_img1 < 0] = 0  
 zip\_img1[zip\_img1 > 1] = 1  
 for i in range(channel):  
 O\_MAX = np.max(origin\_image[:, :, i])  
 O\_MIN = np.min(origin\_image[:, :, i])  
 Z\_MIN = np.min(zip\_img[:, :, i])  
 Z\_MAX = np.max(zip\_img[:, :, i])  
 zip\_img2[:, :, i] = (O\_MAX-O\_MIN) \* (zip\_img[:, :, i] - Z\_MIN) / (Z\_MAX - Z\_MIN) + O\_MIN  
  
 for i in range(channel):  
 axs[2][i] = draw\_density(zip\_img1[:, :, i].reshape(-1), ax=axs[2][i], show\_plt=False,  
 title=f'Processed1-Channel {i}', show\_legend=False)  
 for i in range(channel):  
 axs[3][i] = draw\_density(zip\_img2[:, :, i].reshape(-1), ax=axs[3][i], show\_plt=False,  
 title=f'Processed2-Channel {i}', show\_legend=False)  
 plt.show()  
 plt.close()  
  
 # # 因为数据支持RGB data ([0..1] for floats or [0..255] for integers，所以不一定需要调整到[0, 255]  
 # zip\_img = np.round(zip\_img \* 255).astype('int')  
  
 # 计算压缩率  
 zip\_rate = (origin\_image.size - 3 \* (u\_shape[0] \* u\_shape[1] + s\_shape[0] \* s\_shape[1] + vT\_shape[0] \* vT\_shape[1])) \  
 / origin\_image.size  
  
 print(f"保留率： {rate \* 100:.1f}%")  
 print(f"选择的奇异值数量：{n\_sigmas}--->原来的奇异值数量: {Sigma.shape[0]}")  
 print(f"原图Shape： {origin\_image.shape}--->Size: {origin\_image.size}", )  
 print(f"压缩后的矩阵大小：{u\_shape} , {s\_shape} , {vT\_shape}")  
 print(f"压缩率为： {zip\_rate \* 100:.3f}%")  
 if show\_img:  
 fig, axes = plt.subplots(1, 2)  
 if channel == 1:  
 axes[0].imshow(origin\_image[:, :, 0], cmap='gray')  
 axes[1].imshow(zip\_img[:, :, 0], cmap='gray')  
 else:  
 axes[0].imshow(origin\_image)  
 axes[1].imshow(zip\_img)  
 axes[0].set\_title('Before SVD')  
 axes[1].set\_title(f'After SVD with rate={zip\_rate \* 100:.3f}% and n\_sigmas={n\_sigmas}')  
 plt.show()  
 return zip\_img  
  
  
path = '../input/arona.jpg'  
img\_gray = read\_image(path, gray\_pic=False, show\_details=False)  
# img\_gray = img\_gray.reshape((img\_gray.shape[0], img\_gray.shape[1], 1))  
zip\_image\_by\_svd(img\_gray, rate=0.6, channel=3)

图片的奇异值的大小趋势：



两种压缩后的处理方式是①映射到与原始图像差不多的值域，剩余部分取0或1②映射到[0,255]。因为SVD分解时存在离群值，因此①更优：



**13.神经网络**

**sklearn神经网络**

from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split  
from sklearn.neural\_network import MLPClassifier  
  
Iris = load\_iris()  
X = Iris.data  
Y = Iris.target  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.3)  
model = MLPClassifier(max\_iter=1000)  
model.fit(x\_train, y\_train)  
print("神经网络模型训练集的准确率：%.3f" % model.score(x\_train, y\_train))  
print("神经网络模型测试集的准确率：%.3f" % model.score(x\_test, y\_test))

输出

神经网络模型训练集的准确率：0.971

神经网络模型测试集的准确率：0.978

**torch神经网络**

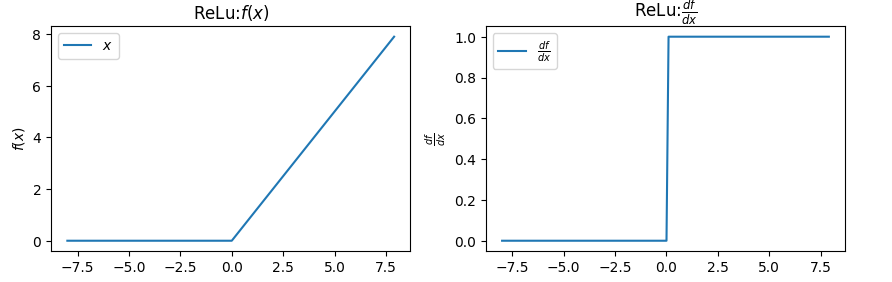
**绘制激活函数与其导数**

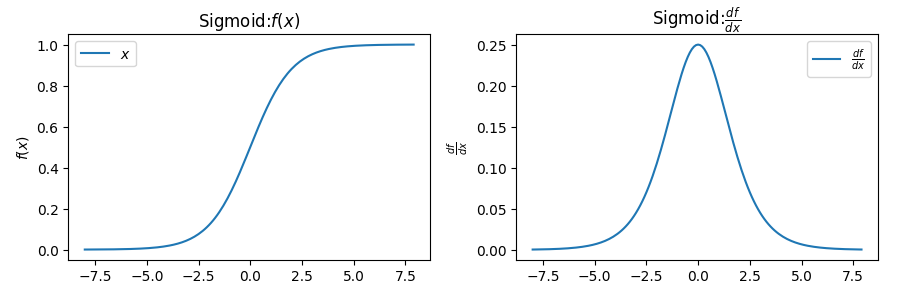
import torch  
import xm\_draw  
  
x = torch.arange(-8.0, 8.0, 0.1, requires\_grad=True)  
y1 = torch.relu(x)  
y2 = torch.sigmoid(x)  
y3 = torch.tanh(x)  
  
xm\_draw.draw\_f\_and\_df(x, y1, funcName\_in\_title="ReLu")  
xm\_draw.draw\_f\_and\_df(x, y2, funcName\_in\_title="Sigmoid")  
xm\_draw.draw\_f\_and\_df(x, y3, funcName\_in\_title="tanh")

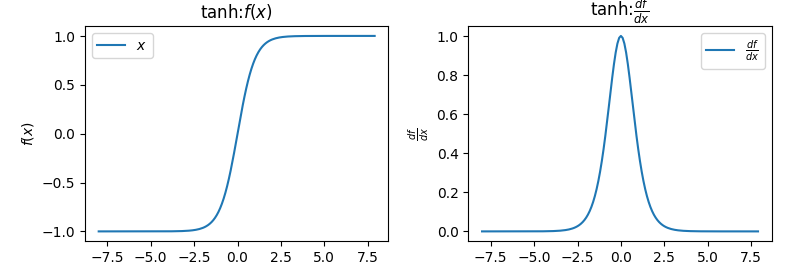
用到的函数如下：

def draw\_f\_and\_df(x, y, draw\_f=True, draw\_df=True, separate=True, funcName\_in\_title=""):  
 *"""绘制函数与其导数，要求输入的x,y的requires\_grad=True"""* x\_latex = r'$x$'  
 fx\_latex = r'$f(x)$'  
 dfx\_latex = r'$\frac{df}{dx}$'  
 if separate:  
 fig, ax = plt.subplots(1, 2)  
 if draw\_f:  
 ax[0].plot(x.detach(), y.detach(), label=x\_latex)  
 ax[0].set\_xlabel(x\_latex)  
 ax[0].set\_ylabel(fx\_latex)  
 ax[0].set\_title(f"{funcName\_in\_title + ':' if funcName\_in\_title else ''}{fx\_latex}")  
 ax[0].legend()  
 if draw\_df:  
 y.backward(torch.ones\_like(x), retain\_graph=True)  
 dy = x.grad  
 ax[1].plot(x.detach(), dy.detach(), label=dfx\_latex)  
 ax[1].set\_xlabel(x\_latex)  
 ax[1].set\_ylabel(dfx\_latex)  
 ax[1].set\_title(f"{funcName\_in\_title + ':' if funcName\_in\_title else ''}{dfx\_latex}")  
 ax[1].legend()  
 plt.show()  
 plt.close()  
 else:  
 fig, ax = plt.subplots(1, 1)  
 y\_latex\_temp, title\_latex\_temp = '', ''  
 if draw\_f:  
 ax.plot(x.detach(), y.detach(), label=fx\_latex)  
 y\_latex\_temp += fx\_latex  
 title\_latex\_temp += fx\_latex  
 if draw\_df:  
 y.backward(torch.ones\_like(x), retain\_graph=True)  
 dy = x.grad  
 y\_latex\_temp += dfx\_latex  
 title\_latex\_temp += "and" if draw\_f else "" + dfx\_latex  
 ax.plot(x.detach(), dy.detach(), label=dfx\_latex)  
 ax.set\_xlabel(x\_latex)  
 ax.set\_ylabel(y\_latex\_temp)  
 ax.set\_title(f"{funcName\_in\_title + ':' if funcName\_in\_title else ''}{title\_latex\_temp}")  
 ax.legend()  
 plt.show()  
 plt.close()  
 x.grad.data.zero\_() # 防止对同一个x多次调用此函数

函数图像如下：







**自己&api实现线性回归**

import torch  
import matplotlib.pyplot as plt  
import xm\_func  
  
  
def synthetic\_data(w, b, num\_examples):  
 *"""生成y=Xw+b+噪声"""* X = torch.normal(0, 1, (num\_examples, len(w))) # 大小(num\_examples, len(w))的张量 X，X的元素服从N(0, 1)  
 y = xm\_func.linreg(X, w, b)  
 y += torch.normal(0, 0.01, y.shape)  
 return X, y.reshape((-1, 1))  
  
true\_w = torch.tensor([2, -3.4])  
true\_b = 4.2  
features, labels = synthetic\_data(true\_w, true\_b, 1000)  
  
print('features:', features[0], '\nlabel:', labels[0]) # 比如features: tensor([-2.2002, 0.1260]) label: tensor([-0.6346])  
plt.scatter(features[:, 1].detach().numpy(), labels.detach().numpy(), 1) # .detach()创建张量副本。这里x轴选取features第二列，y选取labels  
plt.show()  
  
# 初始化  
w = torch.normal(0, 0.01, size=(2, 1), requires\_grad=True) # 比如[[-0.0193], [ 0.0127]]  
b = torch.zeros(1, requires\_grad=True) # [0.]  
print(w, b)  
  
# 超参数  
lr = 0.03 # 学习率  
num\_epochs = 30 # 迭代周期  
batch\_size = 10 # 每个小批量样本的数量  
net = xm\_func.linreg # 线性回归模型  
loss = xm\_func.squared\_loss # 均方损失函数  
  
for epoch in range(num\_epochs):  
 # 控制训练的轮数  
 for X, y in xm\_func.data\_iter(batch\_size, features, labels):  
 # 迭代获取小批量样本  
 l = loss(net(X, w, b), y) # 计算模型的损失。l的大小是(batch\_size,1)，而不是一个标量  
 l.sum().backward() # 计算关于[w,b]的梯度  
 xm\_func.sgd([w, b], lr, batch\_size) # 使用[w,b]的梯度更新[w,b]  
 with torch.no\_grad():  
 train\_l = loss(net(features, w, b), labels)  
 print(f'epoch {epoch + 1}, loss {float(train\_l.mean()):f}, w {w[0].item():.2f},{w[1].item():.2f}, b {b.item():.2f}')  
  
print(f'w的估计误差: {true\_w - w.reshape(true\_w.shape)}')  
print(f'b的估计误差: {true\_b - b}')

用到的xm\_func模块：

def linreg(X, w, b):# mm要求输入矩阵的维度必须匹配，matmul支持广播操作。  
 return torch.matmul(X, w) + b  
  
def squared\_loss(y\_hat, y):return (y\_hat - y.reshape(y\_hat.shape)) \*\* 2 / 2  
  
def data\_iter(batch\_size, features, labels):  
 num\_examples = len(features) # 数据集的长度  
 indices = list(range(num\_examples)) # 创建一个包含数据集索引的列表  
 random.shuffle(indices) # 将数据集索引随机打乱  
 for i in range(0, num\_examples, batch\_size):  
 batch\_indices = torch.tensor(indices[i: min(i + batch\_size, num\_examples)]) # 从打乱的索引列表中取出一个批次的索引  
 yield features[batch\_indices], labels[batch\_indices] # 取出对应的批量数据  
 # yield不会终止函数的执行，而是会暂时挂起函数的执行，并返回一个值，保持函数的状态。每次调用函数时，都会从上次挂起的地方继续执行，直到遇到下一个yield语句。这使得函数可以产生一个序列的值(让函数生成一个数据流)，而不是单个值。  
  
def sgd(params, lr, batch\_size):with torch.no\_grad():  
 # with torch.no\_grad()：在该模块下，所有计算得出的tensor的requires\_grad都自动设置为False。  
 for param in params:  
 param -= lr \* param.grad / batch\_size # 更新参数（使用-=运算符进行原地操作in-place，所以修改的是全局变量）  
 param.grad.zero\_() # 梯度清零(为下一个小批量样本做准备)

上面只用了张量来进行数据存储和线性代数以及通过自动微分来计算梯度，下面用api来实现（feature、labels和超参数不变）：

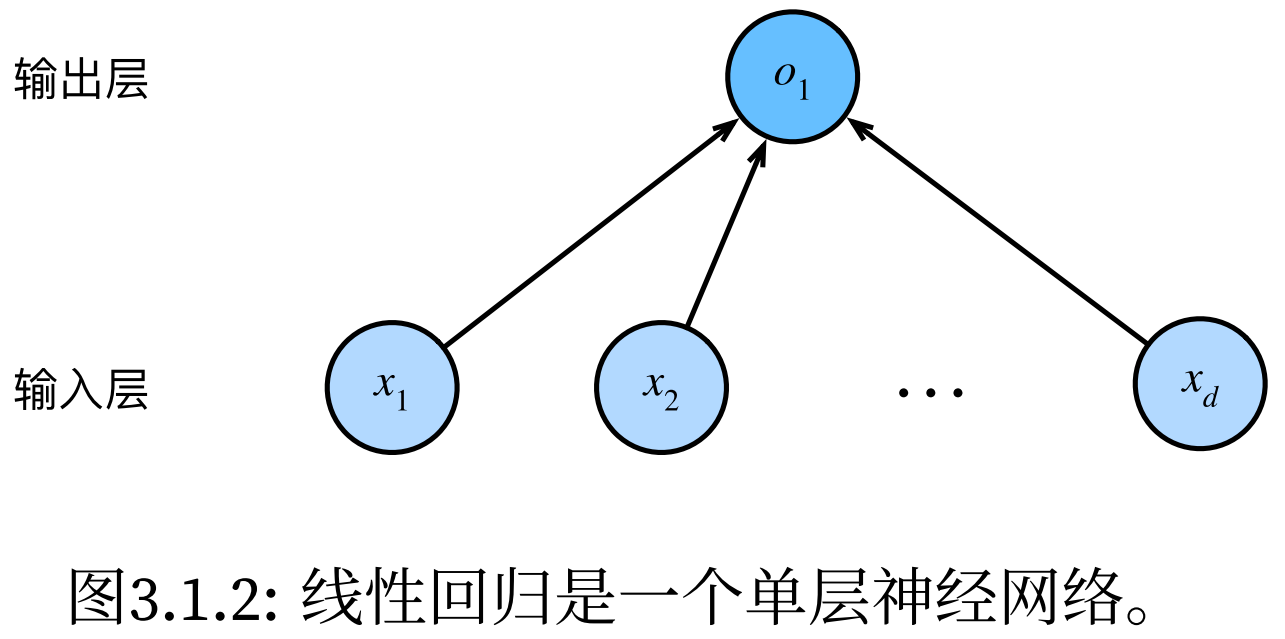
from torch import nn # 神经网络

net = nn.Sequential(nn.Linear(2, 1)) # 流水线里只有一个线性回归模型(输入特征有2个，输出为1个)  
net[0].weight.data.normal\_(0, 0.01) # 用N(0, 0.01^2)的正态分布来初始化权重。net[0]是第一个层(Linear)。weight是该层的权重，data表示获取权重的数据。  
net[0].bias.data.fill\_(0) # bias是该层的偏置，data表示获取偏置的数据。fill\_(0) 表示将偏置的值填充为0  
loss = nn.MSELoss() # 均方损失函数(如果指定reduction='sum'，会使得梯度值放大为原来的num\_example倍，容易在最优解附近震荡)。还可以换成nn.HuberLoss()等  
trainer = torch.optim.SGD(net.parameters(), lr=lr)  
data\_iter = xm\_func.load\_array((features, labels), batch\_size) # 可以用next(iter(data\_iter))来查看数据  
  
for epoch in range(num\_epochs):  
 for X, y in data\_iter:  
 l = loss(net(X), y) # 这里的net返回输入x经过定义的网络所计算出的值  
 trainer.zero\_grad() # 清除上一次的梯度值  
 l.backward() # 反向传播，求参数的梯度  
 trainer.step() # 步进 根据指定的优化算法进行参数的寻优  
 l = loss(net(features), labels)  
 print(f'epoch {epoch + 1}, loss {l:f}')  
w = net[0].weight.data  
print('w的估计误差：', true\_w - w.reshape(true\_w.shape))  
b = net[0].bias.data  
print('b的估计误差：', true\_b - b)

在反向传播后可以这样查看梯度值：

for param in net.parameters():  
 print(param.grad)

线性回归与softmax的比较：

**自己&api实现softmax回归**

**读入数据集**

import torch  
import torchvision  
import xm\_func  
from matplotlib import pyplot as plt  
from torch.utils import data  
from torchvision import transforms  
  
  
# ToTensor实例将图像数据从PIL类型变换成32位浮点数格式，并进行归一化处理(除以255使得所有像素的数值均在0～1之间)  
trans = transforms.ToTensor()  
mnist\_train = torchvision.datasets.FashionMNIST(root="../data", train=True, transform=trans, download=True)  
mnist\_test = torchvision.datasets.FashionMNIST(root="../data", train=False, transform=trans, download=True)  
print(len(mnist\_train), len(mnist\_test)) # 60000 10000  
print(mnist\_train[0][0].shape) # torch.Size([1, 28, 28])  
  
def get\_fashion\_mnist\_labels(labels):  
 *"""输入数值，返回文本标签：t‐shirt T恤、trouser裤子、pullover套衫、dress连衣裙、coat外套、sandal凉鞋、shirt衬衫、sneaker运动鞋、bag包、ankle boot短靴"""* text\_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']  
 return [text\_labels[i] for i in labels]  
  
def show\_images(imgs, num\_rows, num\_cols, titles=None):  
 *"""绘制图像列表"""* \_, axes = plt.subplots(num\_rows, num\_cols)  
 axes = axes.flatten() # 以num\_rows=3,num\_cols=6为例，flatten将(3, 6)的axes转换成(18,)，这样更方便使用单个索引来访问每个子图轴对象  
 for i, (ax, img) in enumerate(zip(axes, imgs)):  
 # 通过对当前的子图轴对象ax的操作直接影响了axes中相应位置的元素  
 if torch.is\_tensor(img):  
 # 图片张量  
 ax.imshow(img.numpy())  
 else:  
 # PIL图片  
 ax.imshow(img)  
 ax.axes.get\_xaxis().set\_visible(False)  
 ax.axes.get\_yaxis().set\_visible(False)  
 if titles:  
 ax.set\_title(titles[i])  
 return axes  
  
# 输出部分图片  
X, y = next(iter(data.DataLoader(mnist\_train, batch\_size=18)))  
show\_images(X.reshape(18, 28, 28), 3, 6, titles=get\_fashion\_mnist\_labels(y))  
plt.show()  
plt.close()  
  
# 测试读入数据  
batch\_size = 256  
train\_iter = data.DataLoader(mnist\_train, batch\_size, shuffle=True)  
timer = xm\_func.Timer()  
timer.start()  
for X, y in train\_iter:  
 continue  
print(f'{batch\_size}--{timer.stop():.2f} sec') # 5.20sec左右  
  
def load\_data\_fashion\_mnist(batch\_size, resize=None):  
 *"""使用已经下载了的Fashion-MNIST数据集(将其加载到内存中)"""* trans = [transforms.ToTensor()] # 加上[]是为了便于后续的列表操作  
 if resize:  
 trans.insert(0, transforms.Resize(resize)) # 将resize变换插入到变成tensor之前  
 trans = transforms.Compose(trans) # 把这个变换列表trans按顺序组合成一个整体的变换操作。  
 mnist\_train = torchvision.datasets.FashionMNIST(root="../data", train=True, transform=trans)  
 mnist\_test = torchvision.datasets.FashionMNIST(root="../data", train=False, transform=trans)  
 return data.DataLoader(mnist\_train, batch\_size, shuffle=True), data.DataLoader(mnist\_test, batch\_size, shuffle=False)  
  
train\_iter, test\_iter = load\_data\_fashion\_mnist(32, resize=64)  
for X, y in train\_iter:  
 print(X.shape, X.dtype, y.shape, y.dtype) # torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64  
 break

**手动(部分)**

def softmax(X):  
 *"""softmax"""* X\_exp = torch.exp(X) # 形状(n, D)，n为样本数量(如batch\_size),D即类别数量  
 partition = X\_exp.sum(1, keepdim=True) # 形状(n, 1)  
 return X\_exp / partition # 广播机制  
  
def softmax\_net(X):  
 *"""softmax回归模型"""* return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)  
  
def cross\_entropy(y\_hat, y):  
 *"""交叉熵损失"""* return - torch.log(y\_hat[range(len(y\_hat)), y]) # range(len(y\_hat))遍历每个样本，y是对应样本的正确类别

**API**

from torch import nn  
from xm\_draw import draw\_LossAndAccuracy  
from xm\_func import evaluate\_accuracy, bool\_accuracy  
  
batch\_size = 256  
train\_iter, test\_iter = load\_data\_fashion\_mnist(batch\_size)  
  
# PyTorch不会隐式地调整输入的形状。因此在线性层前定义了展平层（flatten），来调整网络输入的形状  
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))  
def init\_weights(m):  
 if type(m) == nn.Linear:  
 nn.init.normal\_(m.weight, mean=0, std=0.01)  
net.apply(init\_weights) # 对Flatten,Linear,Sequential依次init\_weights  
loss = nn.CrossEntropyLoss(reduction='none')  
trainer = torch.optim.SGD(net.parameters(), lr=0.1)  
num\_epochs = 10  
  
def train(net, train\_iter, test\_iter, loss, trainer, num\_epochs):  
 train\_loss\_values = []  
 train\_acc\_values = []  
 test\_acc\_values = []  
 for epoch in range(num\_epochs):  
 train\_loss\_sum, train\_acc\_sum, n = 0.0, 0.0, 0  
 for X, y in train\_iter:  
 trainer.zero\_grad()  
 y\_hat = net(X)  
 l = loss(y\_hat, y)  
 l.sum().backward()  
 trainer.step()  
 n += y.shape[0]  
 train\_loss\_sum += l.sum().item()  
 train\_acc\_sum += bool\_accuracy(y\_hat, y)  
 print(f"epoch:{epoch}, 样本数n:{n}")  
 train\_loss\_values.append(train\_loss\_sum / n)  
 train\_acc\_values.append(train\_acc\_sum / n)  
 test\_acc\_values.append(evaluate\_accuracy(net, test\_iter))  
 return train\_loss\_values, train\_acc\_values, test\_acc\_values  
  
train\_loss\_values, train\_acc\_values, test\_acc\_values = train(net, train\_iter, test\_iter, loss, trainer, num\_epochs)  
draw\_LossAndAccuracy(y=[train\_loss\_values, train\_acc\_values, test\_acc\_values], epochs=num\_epochs)

其中用到：

def draw\_LossAndAccuracy(y, epochs, x=None):  
 *"""传入y=[train\_loss\_values, train\_acc\_values, test\_acc\_values]和epochs的值，绘制Loss和Accuracy"""* epochs = range(1, epochs + 1)  
 fig, ax = plt.subplots(2, 1, figsize=(10, 7))  
  
 ax[0].plot(epochs, y[0], label='Train Loss')  
 ax[0].set\_xlabel('Epoch')  
 ax[0].set\_ylabel('Loss')  
 ax[0].set\_title('Training Loss')  
  
 ax[1].plot(epochs, y[1], label='Train Accuracy')  
 ax[1].plot(epochs, y[2], label='Test Accuracy')  
 ax[1].set\_xlabel('Epoch')  
 ax[1].set\_ylabel('Accuracy')  
 ax[1].set\_title('Training & Test Accuracy')  
 ax[1].legend()  
  
 plt.show()

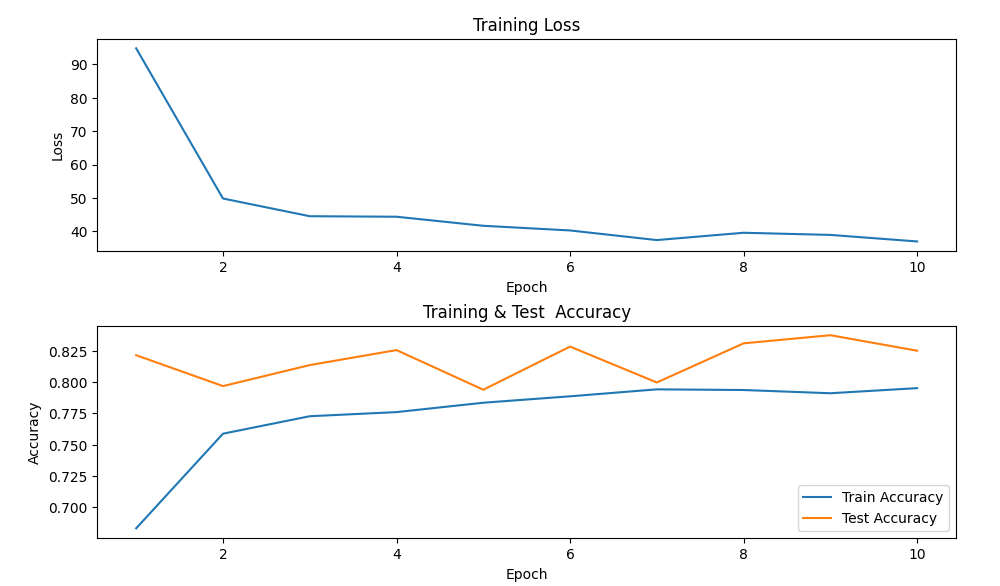
plt.close()

def evaluate\_accuracy(net, data\_iter):  
 *"""计算分类模型的准确率"""* if isinstance(net, torch.nn.Module):  
 net.eval() # 将模型设置为评估模式  
 accuracy\_sum, n = 0, 0  
 with torch.no\_grad():  
 for X, y in data\_iter:  
 accuracy\_sum += bool\_accuracy(net(X), y)  
 n += y.numel()  
 return accuracy\_sum / n  
  
  
def bool\_accuracy(y\_hat, y):  
 *"""计算分类正确的数量"""* if len(y\_hat.shape) > 1 and y\_hat.shape[1] > 1:  
 cmp = y\_hat.argmax(axis=1).type(y.dtype) == y  
 return float(cmp.sum())

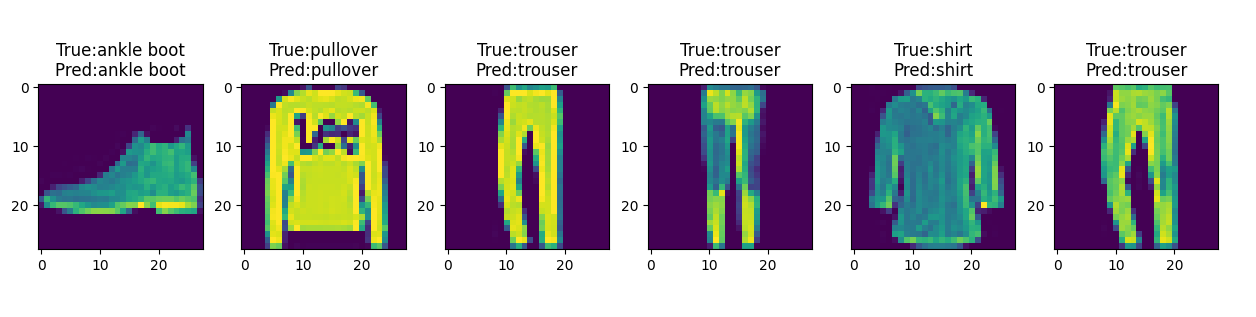
还可以进行预测：

def predict(net, test\_iter, n=6):  
 *"""对前n个验证集样本进行预测"""* X, y = next(iter(test\_iter))  
 trues = get\_fashion\_mnist\_labels(y)  
 preds = get\_fashion\_mnist\_labels(net(X).argmax(axis=1))  
 titles = ["True:" + true + '\n' + "Pred:" + pred for true, pred in zip(trues, preds)]  
 show\_images(X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])  
  
predict(net, test\_iter, n=6)  
plt.show()  
plt.close()

训练结果：



预测结果：



**多层感知机的实现：**

net = nn.Sequential(nn.Flatten(),  
 nn.Linear(784, 256),  
 nn.ReLU(),  
 nn.Linear(256, 10))  
def init\_weights(m):  
 if type(m) == nn.Linear:  
 nn.init.normal\_(m.weight, mean=0, std=0.01)  
net.apply(init\_weights)

其中net的使用方法可以参考softmax的API实现中的过程。

但建议用tanh作为激活函数，这里用relu会让loss值很小而准确率极低(11%)，原因不知。

另外nn.Linear的weight和bias会自动初始化，这里使用init\_weight只是不希望采用对weight的默认初始化，对bias则采用默认初始化。

**多项式回归的实现：**

import numpy as np  
import torch  
import math  
import matplotlib.pylab as plt  
from xm\_func import evaluate\_loss, load\_array  
from torch import nn  
from xm\_draw import draw\_LossAndAccuracy  
  
max\_degree = 20 # 多项式的最大阶数  
n\_train, n\_test = 100, 100 # 训练和测试数据集大小  
true\_w = np.zeros(max\_degree)  
true\_w[0:4] = np.array([5, 1.2, -3.4, 5.6])  
# 生成一个维度为(n\_train + n\_test, 1)的随机正态分布样本特征矩阵  
**features = np.random.normal(size=(n\_train + n\_test, 1))**  
np.random.shuffle(features)  
# 对于features的每一行的那个元素x，进行x^0, x, x^2,..., x^(max\_degree-1)，生成维度为(n\_train + n\_test, max\_degree)的多项式特征矩阵  
# .reshape(1, -1)让维度从(max\_degree,)变成(1, max\_degree)利于广播原则（不reshape其实也行）  
**poly\_features = np.power(features, np.arange(max\_degree).reshape(1, -1))**for i in range(max\_degree):  
 poly\_features[:, i] /= math.gamma(i + 1) # 通过gamma(n)=(n-1)!标准化，**避免非常大的梯度值**或损失值  
# labels维度(n\_train+n\_test,)  
**labels = np.dot(poly\_features, true\_w)**  
labels += np.random.normal(scale=0.1, size=labels.shape) # 自定义误差  
# ndarray转换为tensor  
true\_w, features, poly\_features, labels = [torch.tensor(x, dtype=torch.float32) for x in [true\_w, features, poly\_features, labels]]  
  
def train(train\_features, test\_features, train\_labels, test\_labels, num\_epochs=200):  
 loss = nn.MSELoss(reduction='mean')  
 input\_shape = train\_features.shape[-1] # 以x^0~x^3为例，train\_features大小是[100, 4]，[-1]取出的是4(也就是多少次幂)  
 net = nn.Sequential(nn.Linear(input\_shape, 1, bias=False)) # 不设置偏置，因为已经在多项式中实现了它  
 batch\_size = 10  
 train\_iter = load\_array((train\_features, train\_labels.reshape(-1, 1)), batch\_size)  
 test\_iter = load\_array((test\_features, test\_labels.reshape(-1, 1)), batch\_size, is\_train=False)  
 trainer = torch.optim.SGD(net.parameters(), lr=0.01)  
  
 train\_loss\_values = []  
 train\_acc\_values = []  
 test\_acc\_values = []  
 for epoch in range(num\_epochs):  
 train\_loss\_sum, train\_acc\_sum, n = 0.0, 0.0, 0  
 for X, y in train\_iter:  
 trainer.zero\_grad()  
 y\_hat = net(X)  
 l = loss(y\_hat, y)  
 l.sum().backward()  
 trainer.step()  
 n += y.shape[0]  
 with torch.no\_grad():  
 train\_loss\_sum += l.sum().item()  
 train\_acc\_sum += loss(y\_hat, y)  
 if epoch % 100 == 0:  
 print(f"epoch:{epoch}, 样本数n:{n}")  
 train\_loss\_values.append(train\_loss\_sum / n)  
 train\_acc\_values.append(train\_acc\_sum / n)  
 test\_acc\_values.append(evaluate\_loss(net, test\_iter, loss))  
 print('weight:', net[0].weight.data.numpy())  
 print('loss:', train\_loss\_values[-1])  
 return train\_loss\_values, train\_acc\_values, test\_acc\_values, net  
  
# 从多项式特征中选择前4个维度，即1,x,x^2/2!,x^3/3!  
train\_loss\_values, train\_acc\_values, test\_acc\_values, net1 = \  
 train(poly\_features[:n\_train, :4], poly\_features[n\_train:, :4], labels[:n\_train], labels[n\_train:])  
draw\_LossAndAccuracy([train\_loss\_values, train\_acc\_values, test\_acc\_values], epochs=200, loss\_log=True, Accuracy\_log=True)  
train\_loss\_values, train\_acc\_values, test\_acc\_values, net2 = \  
 train(poly\_features[:n\_train, :2], poly\_features[n\_train:, :2], labels[:n\_train], labels[n\_train:])  
draw\_LossAndAccuracy([train\_loss\_values, train\_acc\_values, test\_acc\_values], epochs=200, loss\_log=True, Accuracy\_log=True)  
train\_loss\_values, train\_acc\_values, test\_acc\_values, net3 = \  
 train(poly\_features[:n\_train, :], poly\_features[n\_train:, :], labels[:n\_train], labels[n\_train:], num\_epochs=1500)  
draw\_LossAndAccuracy([train\_loss\_values, train\_acc\_values, test\_acc\_values], epochs=1500, loss\_log=True, Accuracy\_log=True)

# 绘制y和y\_hat  
for X, net in zip([poly\_features[:, :4], poly\_features[:, :2], poly\_features[:, :]], [net1, net2, net3]):  
 with torch.no\_grad():  
 x = features  
 y = np.dot(poly\_features, true\_w)  
 y\_hat = net(X)  
 f, ax = plt.subplots()  
 ax.scatter(x, y, label='y', color='black')  
 ax.scatter(x, y\_hat, label='y\_hat', color='red')  
 plt.show()  
 plt.close()

用到的模块（其他的函数同之前的代码）：

def evaluate\_loss(net, data\_iter, loss):  
 *"""计算回归模型的损失"""* loss\_sum, n = 0, 0  
 with torch.no\_grad():  
 for X, y in data\_iter:  
 out = net(X)  
 y = y.reshape(out.shape)  
 l = loss(out, y)  
 loss\_sum += l.sum()  
 n += l.numel()  
 return loss\_sum / n

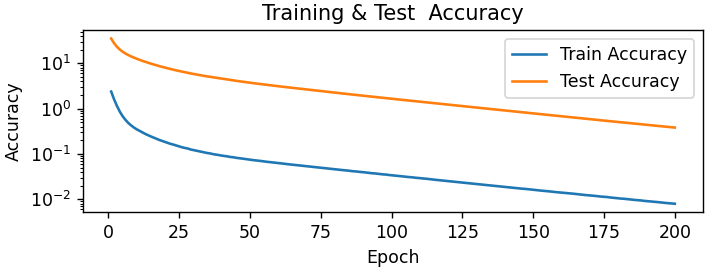
**训练数据：**

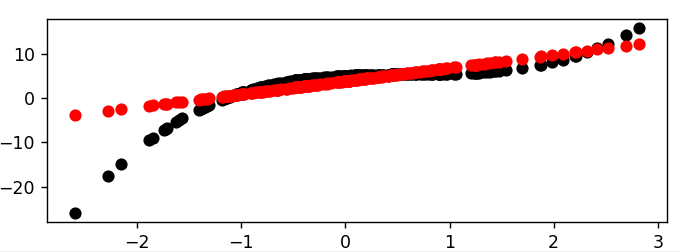
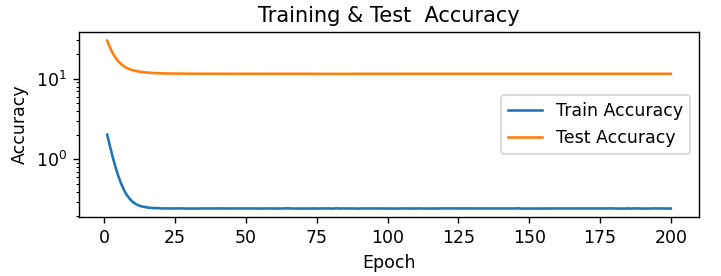
weight: [[ 4.859618 1.6995777 -2.9753466 4.3367476]] loss: 0.007932368274778128

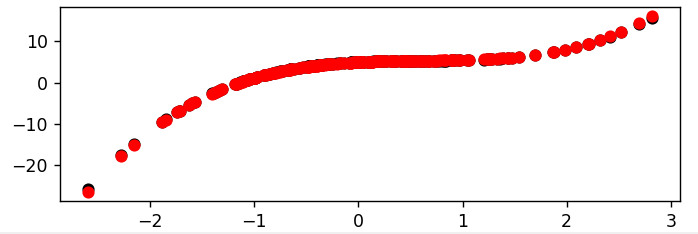
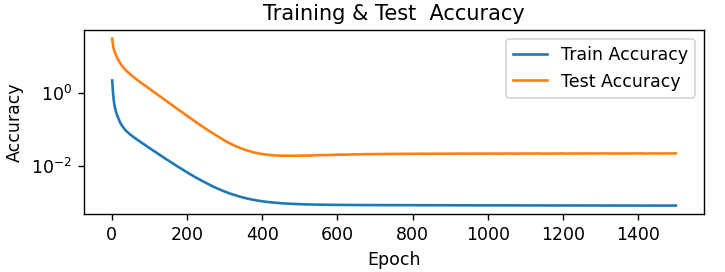
weight: [[3.7808843 2.954084 ]] loss: 0.24653663396835326

weight: [[ 4.976041 1.281252 -3.3253098 5.287895 -0.30464238 1.139414 0.2866924 0.3230021 0.14926858 -0.13491264 -0.06434472 0.01017338 -0.17069647 -0.17043759 -0.17870218 -0.08004908 -0.10076946 0.20569256 0.09011322 -0.19666234]] loss: 0.0007883880054578186

仅展示部分图片（这里的Accuracy用loss来代替）：







**为什么没有过拟合的感觉？因为大部分feature都在0附近，高阶项很小。**

生成数据集的时候还可以只用tensor：

max\_degree = 20  
n\_train, n\_test = 100, 100  
true\_w = torch.zeros(max\_degree)  
true\_w[:4] = torch.tensor([5, 1.2, -3.4, 5.6])  
**features = torch.randn((n\_train + n\_test, 1))**  
features = features[torch.randperm(len(features))]  
**poly\_features = torch.pow(features, torch.arange(max\_degree))**poly\_features /= torch.tensor([math.gamma(i + 1) for i in range(max\_degree)])  
**labels = poly\_features @ true\_w**  
labels += torch.randn(labels.shape) \* 0.1  
labels = labels.reshape(-1, 1)

**还可以直接算出解析解：**

w = torch.linalg.inv(poly\_features[:, :4].t() @ poly\_features[:, :4]) @ poly\_features[:, :4].t() @ labels.reshape(200, 1)

某次得到的是tensor([[ 5.0035], [ 1.1972], [-3.4066], [ 5.6050]])。原理是：对凑广义逆得。

**添加正则化(权重衰减)**只需要更改trainer：

trainer = torch.optim.SGD([  
 {"params": net[0].weight, 'weight\_decay': 3},  
 {"params": net[0].bias}], lr=0.01)

注意是给weight正则化，bias不需要正则化。（而且如果之前指定bias=False了，这里也不应该出现bias）

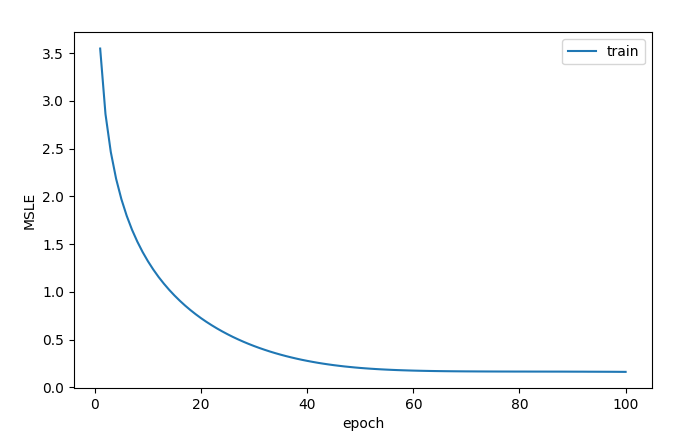
**暂退法的实现：**

dropout1, dropout2 = 0.2, 0.5  
net = nn.Sequential(nn.Flatten(),  
 nn.Linear(784, 256),  
 nn.ReLU(),  
 # 在第一个全连接层之后添加一个dropout层  
 nn.Dropout(dropout1),  
 nn.Linear(256, 256),  
 nn.ReLU(),  
 # 在第二个全连接层之后添加一个dropout层  
 nn.Dropout(dropout2),  
 nn.Linear(256, 10))  
def init\_weights(m):  
 if type(m) == nn.Linear:  
 nn.init.normal\_(m.weight)  
net.apply(init\_weights)

可以将暂退法应用于每个隐藏层的输出（**在激活函数之后**），并且可以为每一层分别设置暂退概率。常见的技巧是**在靠近输入层的地方设置较低的暂退概率**。

**kaggle-房价**

import pandas as pd  
import numpy as np  
import torch  
import xm\_func  
import matplotlib.pylab as plt  
from torch import nn  
  
# 1.读入  
train\_data = pd.read\_csv(r"..\data\kaggle\_house\_pred\_train.csv") # shape(1460, 81)  
test\_data = pd.read\_csv(r"..\data\kaggle\_house\_pred\_test.csv") # shape(1459, 80)  
all\_features = pd.concat((train\_data.iloc[:, 1:-1], test\_data.iloc[:, 1:])) # shape(2919, 79)，去除了ID和train里的SalePrice  
  
# 2.数据处理  
# .dtypes返回一个Series(索引是列名，值是对应列的数据类型)  
numeric\_features = **all\_features.dtypes[all\_features.dtypes != 'object'].index**  
# 对每一列x进行标准化处理  
all\_features[numeric\_features] = all\_features[numeric\_features]**.apply(lambda x: (x - x.mean()) / (x.std()))**# # 若无法获得测试数据，则可根据训练数据计算均值和标准差  
# train\_mean, train\_std = all\_features[numeric\_features].mean(), all\_features[numeric\_features].std()  
# all\_features = (all\_features - train\_mean) / train\_std  
# 在标准化数据之后，所有均值消失，因此我们可以将缺失值设置为0  
all\_features[numeric\_features] = all\_features[numeric\_features]**.fillna(0)**all\_features = **pd.get\_dummies(all\_features, dummy\_na=True)**  # 将object的数据转为指示性变量，此时shape为(2919, 330)  
all\_features = all\_features.astype(np.float32) # 将True,False变成float。也可以all\_features = all\_features \* 1  
# 转为tensor  
n\_train = train\_data.shape[0]  
train\_features = torch.tensor(all\_features[:n\_train].values, dtype=torch.float32)  
test\_features = torch.tensor(all\_features[n\_train:].values, dtype=torch.float32)  
train\_labels = torch.tensor(train\_data['SalePrice'].values.reshape(-1, 1), dtype=torch.float32)  
  
# 3.训练  
loss = nn.MSELoss()  
in\_features = train\_features.shape[1]  
  
def log\_rmse(net, features, labels):  
 # 为了在取对数时进一步稳定该值，将数值限定在1~inf之间，避免小于1时出现负数  
 clipped\_preds = torch**.clamp**(net(features), 1, float('inf'))  
 rmse = torch.sqrt**(**loss(torch.log(clipped\_preds), torch.log(labels))**)**  
 return rmse.item()  
  
def train(net, train\_features, train\_labels, test\_features, test\_labels, num\_epochs, learning\_rate, weight\_decay, batch\_size):  
 train\_ls, test\_ls = [], []  
 train\_iter = xm\_func.load\_array((train\_features, train\_labels), batch\_size)  
 # 这里使用的是Adam优化算法  
 optimizer = torch**.optim.Adam**(net.parameters(), lr=learning\_rate, weight\_decay=weight\_decay)  
 for epoch in range(num\_epochs):  
 for X, y in train\_iter:  
 optimizer.zero\_grad()  
 l = loss(net(X), y)  
 l.backward()  
 optimizer.step()  
 train\_ls.append(log\_rmse(net, train\_features, train\_labels))  
 if test\_labels is not None:  
 test\_ls.append(log\_rmse(net, test\_features, test\_labels))  
 return train\_ls, test\_ls  
  
def get\_k\_fold\_data(k, i, X, y):  
 assert k > 1  
 fold\_size = X.shape[0] // k  
 X\_train, y\_train, X\_valid, y\_valid = None, None, None, None  
 for j in range(k):  
 idx = slice(j \* fold\_size, (j + 1) \* fold\_size) # [j\*fold\_size, (j+1)\*fold\_size)  
 X\_part, y\_part = X[idx, :], y[idx]  
 if j == i:  
 X\_valid, y\_valid = X\_part, y\_part  
 **elif** X\_train is None:  
 X\_train, y\_train = X\_part, y\_part  
 else:  
 X\_train = torch.cat([X\_train, X\_part], 0) # 按行拼接  
 y\_train = torch.cat([y\_train, y\_part], 0)  
 return X\_train, y\_train, X\_valid, y\_valid  
  
def k\_fold(k, X\_train, y\_train, num\_epochs, learning\_rate, weight\_decay, batch\_size):  
 *"""K折交叉验证，有助于模型选择和超参数调整。如果训练误差非常低但K折交叉验证的误差要高得多，表明模型过拟合了"""* train\_l\_sum, valid\_l\_sum = 0, 0  
 for i in range(k):  
 data = get\_k\_fold\_data(k, i, X\_train, y\_train)  
 net = nn.Sequential(nn.Linear(in\_features, 1))  
 train\_ls, valid\_ls = train(net, \*data, num\_epochs, learning\_rate, weight\_decay, batch\_size)  
 train\_l\_sum += train\_ls[-1]  
 valid\_l\_sum += valid\_ls[-1]  
 if i == 0:  
 plt.plot(list(range(1, num\_epochs + 1)), train\_ls, label='train')  
 plt.plot(list(range(1, num\_epochs + 1)), valid\_ls, label='valid')  
 plt.gca().set\_xlabel('epoch') # gca:Get Current Axes 获取当前轴对象  
 plt.gca().set\_ylabel('MSLE')  
 plt.ylim([0, 0.5])  
 plt.legend()  
 plt.show()  
 plt.close()  
 print(f'折{i + 1}，训练MSLE{float(train\_ls[-1]):f}, 'f'验证MSLE{float(valid\_ls[-1]):f}')  
 return train\_l\_sum / k, valid\_l\_sum / k  
  
k, num\_epochs, lr, weight\_decay, batch\_size = 5, 100, 5, 0, 64  
# train\_l, valid\_l = k\_fold(k, train\_features, train\_labels, num\_epochs, lr, weight\_decay, batch\_size)  
# print(f'{k}-折验证: 平均训练MSLE: {float(train\_l):f}, 'f'平均验证MSLE: {float(valid\_l):f}')  
  
def train\_and\_pred(train\_features, test\_features, train\_labels, test\_data, num\_epochs, lr, weight\_decay, batch\_size):  
 *"""应该选择什么样的超参数后，再使用所有数据对其进行训练"""* net = nn.Sequential(nn.Linear(in\_features, 1))  
 train\_ls, \_ = train(net, train\_features, train\_labels, None, None, num\_epochs, lr, weight\_decay, batch\_size)  
 plt.plot(list(range(1, num\_epochs + 1)), train\_ls, label='train')  
 plt.gca().set\_xlabel('epoch')  
 plt.gca().set\_ylabel('MSLE')  
 plt.legend()  
 plt.show()  
 plt.close()  
 print(f'训练MSLE：{float(train\_ls[-1]):f}') # 某次是0.162501  
 # 将网络应用于测试集。  
 preds = net(test\_features).detach().numpy() # 大小(1459, 1)  
 # 将其重新格式化以导出到Kaggle  
 test\_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])  
 submission = pd.concat([test\_data['Id'], test\_data['SalePrice']], axis=1)  
 submission.to\_csv('submission.csv', index=False)  
  
train\_and\_pred(train\_features, test\_features, train\_labels, test\_data, num\_epochs, lr, weight\_decay, batch\_size)



另外，根据别人的经验，可以选用：

net = nn.Sequential(nn.Flatten(),

nn.Linear(in\_features, 1024),

nn.ReLU(),

nn.Linear(1024, 1))

k, num\_epochs, lr, weight\_decay, batch\_size = 5, 100, 0.1, 35, 256

**访问参数**

# 一般的Sequential的参数访问  
net = nn.Sequential(nn.Linear(4, 2), nn.ReLU(), nn.Linear(2, 1))  
X = torch.rand(size=(3, 4))  
print(X)  
print(net(X))  
print(net.state\_dict()) # 参数访问  
print(net[2].bias) # 提取偏置，提取后返回的是一个参数类实例  
print(net[2].bias.data) # 进一步访问该参数的值  
print(net[2].bias.grad) # 梯度值（因为这里并没有反向传播，所以是None）  
print(\*[(name, param.shape) for name, param in net[0].named\_parameters()]) # \*将整个列表解包成单独的元素(这里是name,shape构成的元组)  
print([(name, param.shape) **for name, param in net.named\_parameters()**])print(net.state\_dict()['2.bias'].data) # 另一种访问参数的方法  
net[2].bias.data = torch.tensor([7.27]) # 修改参数时data set to a tensor that requires gradients must be floating point or complex dtype  
  
# 嵌套net的参数访问  
def block\_small():  
 return nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 4), nn.ReLU())  
def block\_main():  
 net = nn.Sequential()  
 for i in range(4):  
 # 在这里嵌套  
 net.add\_module(f'block {i}', block\_small())  
 return net  
rgnet = nn.Sequential(block\_main(), nn.Linear(4, 1))  
rgnet(X)  
print(rgnet)  
print(rgnet[0][1][0].bias.data) # 获取Sequential里的第1个主块(block\_main)的第2个子块(block 1)的第1层(Linear(4, 8))的偏置项

从net.state\_dict()到[(name, param.shape) ……]的输出：

OrderedDict([('0.weight', tensor([[-0.3908, 0.1140, -0.1552, 0.3524],

[-0.2434, 0.4896, -0.4983, -0.1692]])), ('0.bias', tensor([-0.1386, 0.0155])), ('2.weight', tensor([[-0.1979, -0.5240]])), ('2.bias', tensor([-0.5439]))])

Parameter containing:

tensor([-0.5439], requires\_grad=True)

tensor([-0.5439])

None

('weight', torch.Size([2, 4])) ('bias', torch.Size([2]))

[('0.weight', torch.Size([2, 4])), ('0.bias', torch.Size([2])), ('2.weight', torch.Size([1, 2])), ('2.bias', torch.Size([1]))]

**参数初始化**

初始化权重矩阵还可以改为：torch.nn.init.xavier\_normal\_(tensor, gain=1)等。

xavier的思想是让，即，也就是从的高斯分布中采样权重。因为均匀分布的方差是，所以初始化值域是。这里gain参数表示。

# 内置初始化  
net = nn.Sequential(nn.Linear(4, 8), nn.Linear(8, 6), nn.Linear(6, 2))  
def init\_xavier(m):  
 if type(m) == nn.Linear:  
 nn.init.xavier\_uniform\_(m.weight)  
def init\_normal(m):  
 if type(m) == nn.Linear:  
 nn.init.normal\_(m.weight, mean=0, std=0.01)  
 nn.init.zeros\_(m.bias)  
def init\_constant(m, c):  
 if type(m) == nn.Linear:  
 nn.init.constant\_(m.weight, c)  
 nn.init.constant\_(m.bias, 0)  
net[0].apply(init\_xavier)  
net[1].apply(init\_normal) # 也可以net[1].weight.data.normal\_(0, 0.01) net[1].bias.data.fill\_(0)  
net[2].apply(lambda m: init\_constant(m, 0.5)) # 该匿名函数接受参数m，将c设置为0.5  
  
# 自定义初始化  
def my\_init(m):  
 if type(m) == nn.Linear:  
 print("Init", \*[(name, param.shape) for name, param in m.named\_parameters()][0])  
 nn.init.uniform\_(m.weight, -10, 10)  
 m.weight.data \*= m.weight.data.abs() >= 5  
net.apply(my\_init)  
print(net[0].weight)

**参数绑定**

shared = nn.Linear(8, 8) # 给共享层一个名称，以便可以引用它的参数  
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),  
 shared, nn.ReLU(),  
 shared, nn.ReLU(),  
 nn.Linear(8, 1))  
net(X)  
print(net[2].weight.data[0] == net[4].weight.data[0]) # 输出都是True。  
net[2].weight.data[0, 0] = 100  
print(net[2].weight.data[0] == net[4].weight.data[0]) # 输出都是True。它们实际上是同一个对象，而不只是有相同的值

参数绑定时，在反向传播期间net[2]和net[4]的梯度会加在一起(因为传播的梯度还没有清零)。

共享参数可以减少计算成本(共享的参数只需要计算一次梯度)。还可以用于处理变长输入序列(如nlp)。

**延后初始化**

net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))  
print(net[0].weight) # 尚未初始化，输出的是<UninitializedParameter>  
print(net) # LazyLinear(in\_features=**0**, out\_features=256, bias=True) 和 LazyLinear(in\_features=**0**, out\_features=10, bias=True)  
X = torch.rand(2, 20)  
net(X)  
print(net) # Linear(in\_features=**20**, out\_features=256, bias=True) 和 Linear(in\_features=**256**, out\_features=10, bias=True)  
print(net(X)) # 能够输出了

延后初始化defers initialization：直到数据第一次通过模型传递时，框架才会动态地推断出每个层的大小。

现在使用会提示：UserWarning: Lazy modules are a new feature under heavy development so changes to the API or functionality can happen at any moment.

**卷积核的使用与训练(图像中的目标边缘检测)**

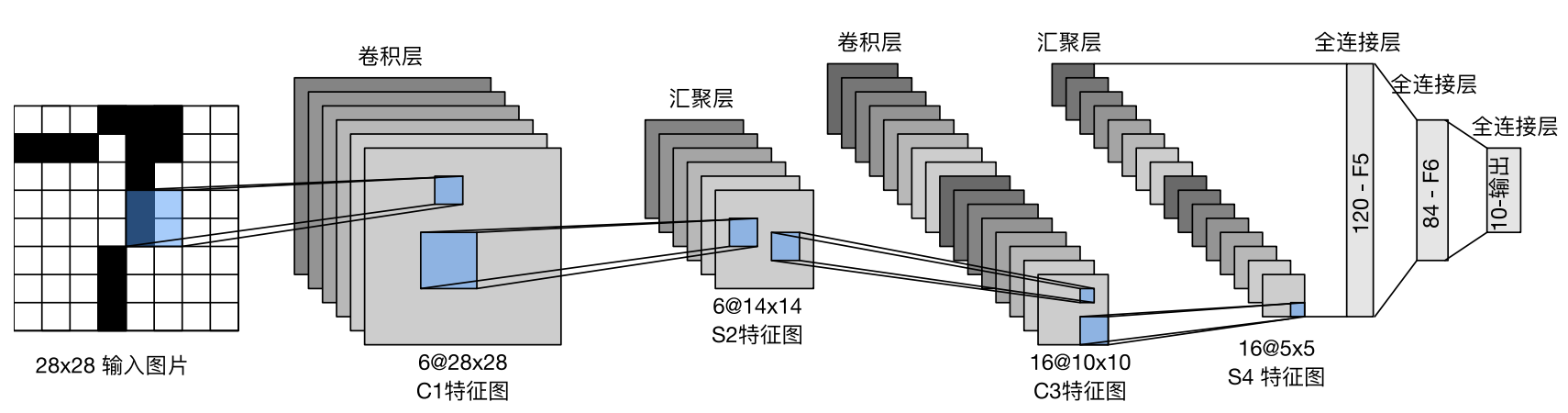
def corr2d(X, K):  
 *"""计算二维互相关运算cross‐correlation"""* h, w = K.shape  
 Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))  
 for i in range(Y.shape[0]):  
 for j in range(Y.shape[1]):  
 Y[i, j] = (X[i:i + h, j:j + w] \* K).sum()  
 return Y  
  
X = torch.ones((6, 8))  
X[:, 2:6] = 0 # 3~6列是1，其他是0  
K = torch.tensor([[1.0, -1.0]]) # 当进行互相关运算时，如果水平相邻的两元素相同，则输出为零，否则输出为非零（这个卷积核K只可以检测垂直边缘，无法检测水平边缘）  
Y = corr2d(X, K) # Y中的1代表从白色到黑色的边缘，‐1代表从黑色到白色的边缘，其他情况的输出为0  
  
# 1个输入通道,1个输出通道，形状为(1, 2)的卷积核  
conv2d = nn.Conv2d(1, 1, kernel\_size=(1, 2), bias=False)  
# 这个二维卷积层使用四维输入和输出格式(batch\_size批量大小, channels通道数, height, width)  
X = X.reshape((1, 1, 6, 8))  
Y = Y.reshape((1, 1, 6, 7))  
lr = 0.01 # 学习率  
for i in range(50):  
 Y\_hat = conv2d(X)  
 l = (Y\_hat - Y) \*\* 2  
 conv2d.zero\_grad()  
 l.sum().backward()  
 conv2d.weight.data[:] -= lr \* conv2d.weight.grad # 迭代卷积核  
 print(f'epoch {i+1}, loss {l.sum():.6f}')  
print(conv2d.weight.data.reshape((1, 2))) # 将[1, 1, 1, 2]大小的weight变成[1, 2]

最后得到的weight是tensor([[ 0.9980, -0.9980]])与垂直边缘检测核K=[1,-1]很接近了。

**各种CNN网络**

**LeNet** (代码实现时对网络有修改，没有全连接层，sigmoid改为了relu)

net = nn.Sequential(  
 nn.Conv2d(1, 6, kernel\_size=5, padding=2), nn.ReLU(),  
 nn.AvgPool2d(kernel\_size=2, stride=2),  
 nn.Conv2d(6, 16, kernel\_size=5), nn.ReLU(),  
 nn.AvgPool2d(kernel\_size=2, stride=2),  
 nn.Flatten(),  
 nn.Linear(16 \* 5 \* 5, 120), nn.ReLU(),  
 nn.Linear(120, 84), nn.ReLU(),  
 nn.Linear(84, 10))  
def init\_weights(m):  
 if type(m) == nn.Linear or type(m) == nn.Conv2d:  
 nn.init.xavier\_uniform\_(m.weight)  
net.apply(init\_weights)



**AlexNet** (对部分数据更改以适应本数据集，更改的标注在了注释里)

net = nn.Sequential(  
 nn.Conv2d(1, 96, kernel\_size=3, stride=2, padding=1), nn.ReLU(), # 11->3, 4->2  
 nn.MaxPool2d(kernel\_size=3, stride=2),  
 nn.Conv2d(96, 256, kernel\_size=5, padding=2), nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=1), # 2->1  
 nn.Conv2d(256, 384, kernel\_size=3, padding=1), nn.ReLU(),  
 nn.Conv2d(384, 384, kernel\_size=3, padding=1), nn.ReLU(),  
 nn.Conv2d(384, 256, kernel\_size=3, padding=1), nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=2),  
 nn.Flatten(),  
 nn.Linear(256, 4096), nn.ReLU(), # 6400->256  
 nn.Dropout(p=0.5),  
 nn.Linear(4096, 4096), nn.ReLU(),  
 nn.Dropout(p=0.5),  
 nn.Linear(4096, 10)) # 1000->10

该net需要的GPU空间太大了，我运行不出来…

**VGG** (请注意这里的输入图像大小改为224\*224了，下同)

batch\_size, channels, height, width = 64, 1, 224, 224  
X = torch.randn(batch\_size, channels, height, width)  
  
def vgg\_block(num\_convs, in\_channels, out\_channels):  
 *"""卷积层的数量、输入通道的数量、输出通道的数量*

*3×3卷积核、填充为1的卷积层->保持高度和宽度，*

*2×2汇聚窗口、步幅为2的最大汇聚层->每个块后的分辨率减半"""* layers = []  
 for \_ in range(num\_convs):  
 layers.append(nn.Conv2d(in\_channels, out\_channels, kernel\_size=3, padding=1))  
 layers.append(nn.ReLU())  
 in\_channels = out\_channels  
 layers.append(nn.MaxPool2d(kernel\_size=2, stride=2))  
 return nn.Sequential(\*layers)  
def vgg(conv\_arch):  
 *"""conv\_arch指定了每个VGG块里卷积层个数和输出通道数"""* conv\_blks = []  
 in\_channels = 1  
 # 卷积层部分  
 for (num\_convs, out\_channels) in conv\_arch:  
 conv\_blks.append(**vgg\_block**(num\_convs, in\_channels, out\_channels))  
 in\_channels = out\_channels  
 return nn.Sequential(  
 \*conv\_blks, nn.Flatten(),  
 # 全连接层部分  
 nn.Linear(out\_channels \* 7 \* 7, 4096), nn.ReLU(), nn.Dropout(0.5),  
 nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),  
 nn.Linear(4096, 10))

conv\_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))  
net = vgg(conv\_arch)

其中7是根据输入得出来的，可以将nn.AdaptiveMaxPool2d(output\_size=(7, 7))应用于最后一个卷积层的输出，然后将其Flatten作为全连接层的输入。这样就不需要手动计算特征图大小，而是根据输出的固定大小来自适应地处理输入。

**NiN**

def nin\_block(in\_channels, out\_channels, kernel\_size, strides, padding):  
 return nn.Sequential(  
 nn.Conv2d(in\_channels, out\_channels, kernel\_size, strides, padding),  
 nn.ReLU(),  
 nn.Conv2d(out\_channels, out\_channels, kernel\_size=1), nn.ReLU(),  
 nn.Conv2d(out\_channels, out\_channels, kernel\_size=1), nn.ReLU())  
  
  
net = nn.Sequential(  
 nin\_block(1, 96, kernel\_size=11, strides=4, padding=0),  
 nn.MaxPool2d(3, stride=2),  
 nin\_block(96, 256, kernel\_size=5, strides=1, padding=2),  
 nn.MaxPool2d(3, stride=2),  
 nin\_block(256, 384, kernel\_size=3, strides=1, padding=1),  
 nn.MaxPool2d(3, stride=2),  
 nn.Dropout(0.5),  
 # 标签类别数是10  
 nin\_block(384, 10, kernel\_size=3, strides=1, padding=1),  
 nn.AdaptiveAvgPool2d((1, 1)),  
 # 将四维的输出(N,C,1,1)转成二维的输出(N,C)，其形状为(N=批量大小,C=10)  
 nn.Flatten())

将空间维度中的每个像素视为单个样本，将通道维度视为不同特征。也就是在每个像素位置(对每个高度和宽度)应用一个全连接层，也就是用1×1卷积层将权重连接到每个空间位置。

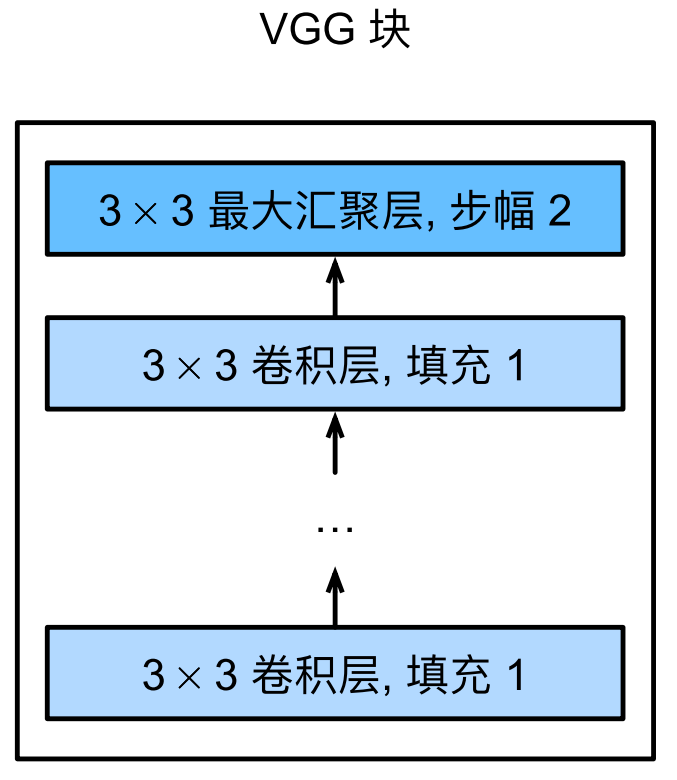
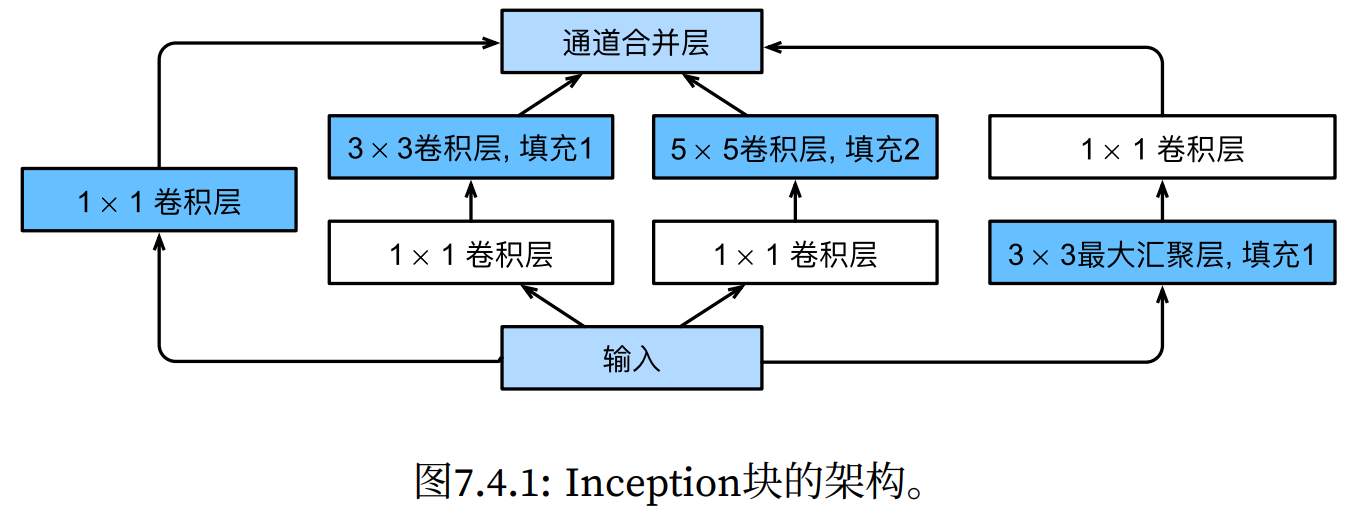
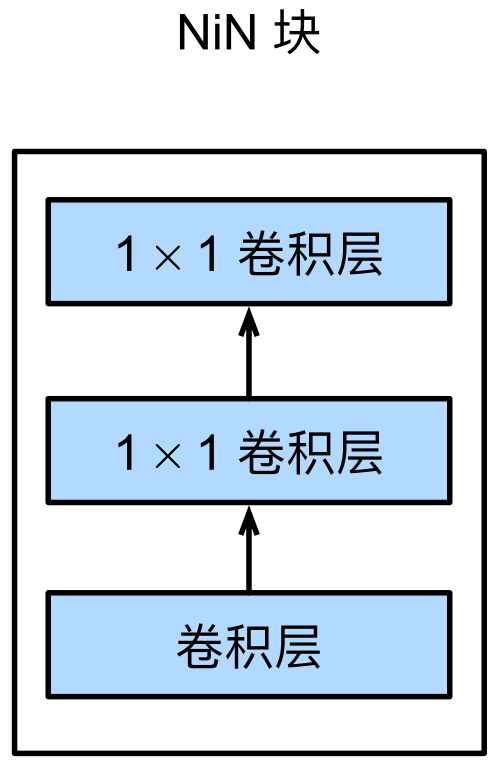
**GoogLeNet的Inception块**

class Inception(nn.Module):  
 def \_\_init\_\_(self, in\_channels, c1, c2, c3, c4, \*\*kwargs):  
 *"""c1--c4是每条路径的输出通道数"""* super(Inception, self).\_\_init\_\_(\*\*kwargs)  
 # 线路1，单1x1卷积层  
 self.p1\_1 = nn.Conv2d(in\_channels, c1, kernel\_size=1)  
 # 线路2，1x1卷积层后接3x3卷积层  
 self.p2\_1 = nn.Conv2d(in\_channels, c2[0], kernel\_size=1)  
 self.p2\_2 = nn.Conv2d(c2[0], c2[1], kernel\_size=3, padding=1)  
 # 线路3，1x1卷积层后接5x5卷积层  
 self.p3\_1 = nn.Conv2d(in\_channels, c3[0], kernel\_size=1)  
 self.p3\_2 = nn.Conv2d(c3[0], c3[1], kernel\_size=5, padding=2)  
 # 线路4，3x3最大汇聚层后接1x1卷积层  
 self.p4\_1 = nn.MaxPool2d(kernel\_size=3, stride=1, padding=1)  
 self.p4\_2 = nn.Conv2d(in\_channels, c4, kernel\_size=1)  
  
 def forward(self, x):  
 p1 = F.relu(self.p1\_1(x))  
 p2 = F.relu(self.p2\_2(F.relu(self.p2\_1(x))))  
 p3 = F.relu(self.p3\_2(F.relu(self.p3\_1(x))))  
 p4 = F.relu(self.p4\_2(self.p4\_1(x)))  
 # 在通道维度上连结输出  
 return torch.cat((p1, p2, p3, p4), dim=1)  
  
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel\_size=7, stride=2, padding=3),  
 nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel\_size=1),  
 nn.ReLU(),  
 nn.Conv2d(64, 192, kernel\_size=3, padding=1),  
 nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),  
 Inception(256, 128, (128, 192), (32, 96), 64),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),  
 Inception(512, 160, (112, 224), (24, 64), 64),  
 Inception(512, 128, (128, 256), (24, 64), 64),  
 Inception(512, 112, (144, 288), (32, 64), 64),  
 Inception(528, 256, (160, 320), (32, 128), 128),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),  
 Inception(832, 384, (192, 384), (48, 128), 128),  
 nn.AdaptiveAvgPool2d((1, 1)),  
 nn.Flatten())  
net = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))

Inception块通过填充使得输出的shape能够在dim=1上联结，也就是shape均满足[N, \*, H, W]，仅在shape[1]上不同。

GoogLeNet就像用不同大小的滤波器识别不同范围的图像细节，还能为不同的滤波器分配不同数量的参数，因此效果好。

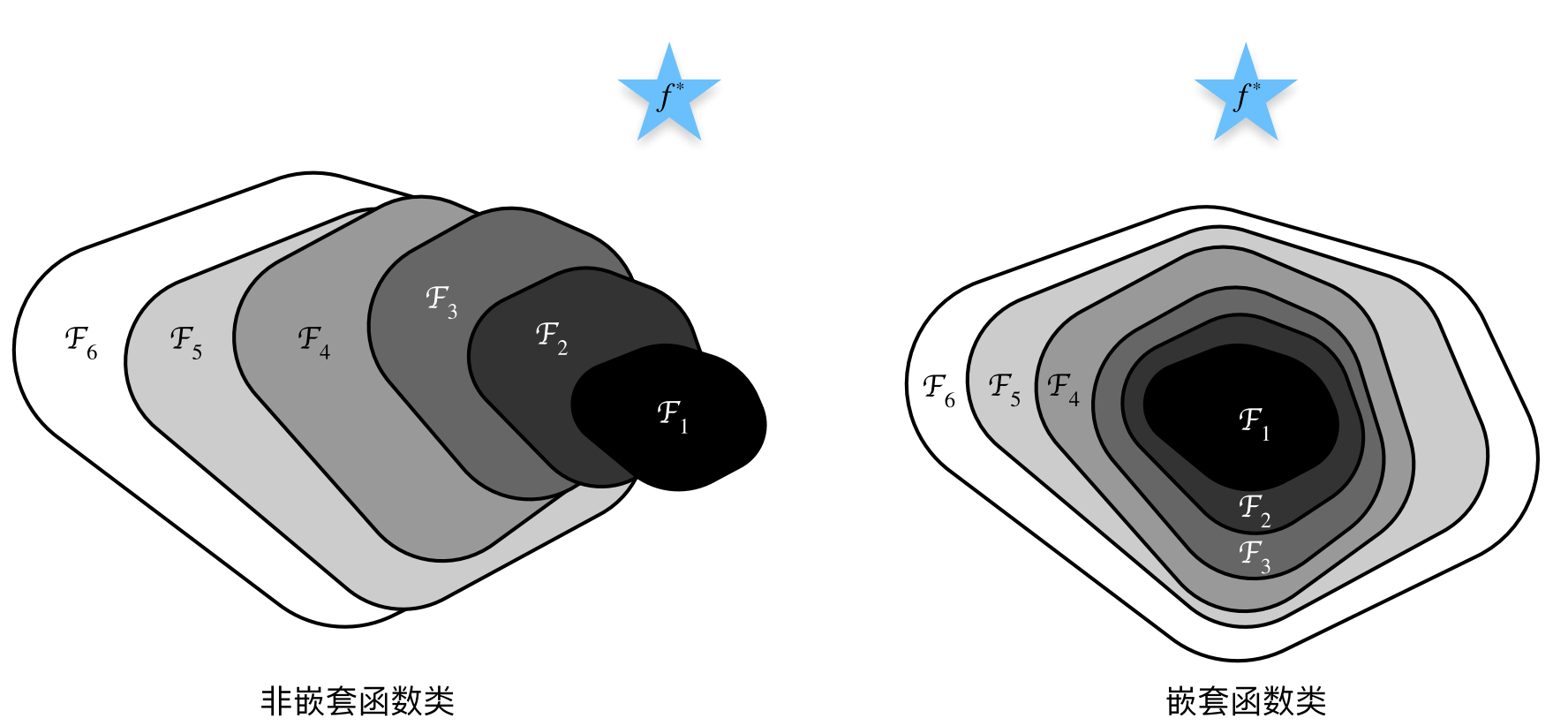
三种块的对比如下：

**ResNet残差网络**

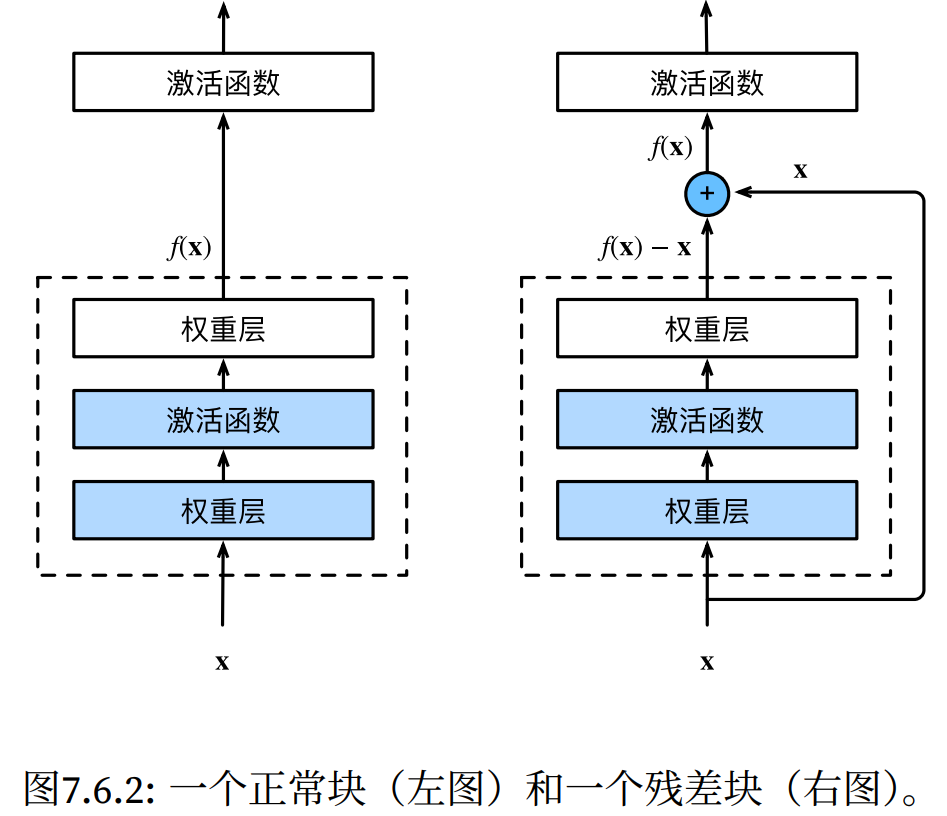
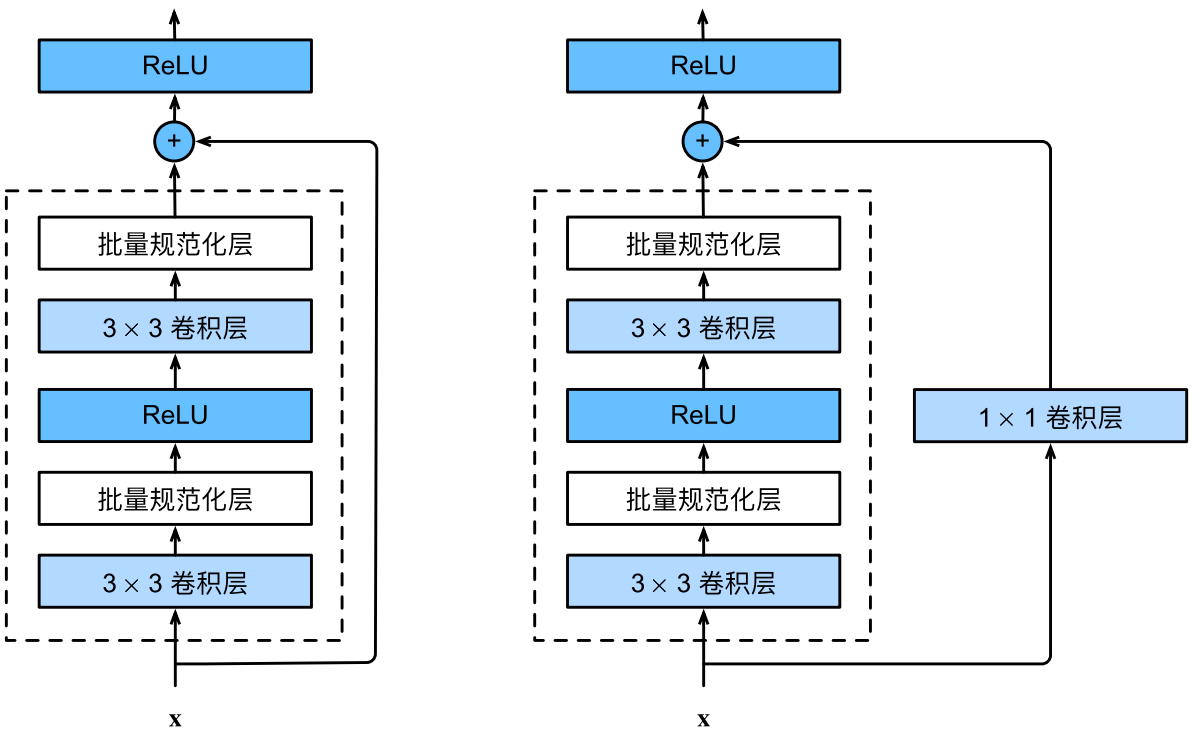
class Residual(nn.Module):  
 def \_\_init\_\_(self, input\_channels, num\_channels, use\_1x1conv=False, strides=1):  
 super().\_\_init\_\_()  
 self.conv1 = nn.Conv2d(input\_channels, num\_channels, kernel\_size=3, padding=1, stride=strides)  
 self.conv2 = nn.Conv2d(num\_channels, num\_channels, kernel\_size=3, padding=1)  
 if use\_1x1conv:  
 self.conv3 = nn.Conv2d(input\_channels, num\_channels, kernel\_size=1, stride=strides)  
 else:  
 self.conv3 = None  
 self.bn1 = nn.BatchNorm2d(num\_channels)  
 self.bn2 = nn.BatchNorm2d(num\_channels)  
  
 def forward(self, X):  
 Y = F.relu(self.bn1(self.conv1(X)))  
 Y = self.bn2(self.conv2(Y))  
 if self.conv3:  
 X = self.conv3(X)  
 Y += X  
 return F.relu(Y)  
  
def resnet\_block(input\_channels, num\_channels, num\_residuals,  
 first\_block=False):  
 blk = []  
 for i in range(num\_residuals):  
 if i == 0 and not first\_block:  
 blk.append(Residual(input\_channels, num\_channels,  
 use\_1x1conv=True, strides=2))  
 else:  
 blk.append(Residual(num\_channels, num\_channels))  
 return blk  
  
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel\_size=7, stride=2, padding=3),  
 nn.BatchNorm2d(64), nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
b2 = nn.Sequential(\*resnet\_block(64, 64, 2, first\_block=True))  
b3 = nn.Sequential(\*resnet\_block(64, 128, 2))  
b4 = nn.Sequential(\*resnet\_block(128, 256, 2))  
b5 = nn.Sequential(\*resnet\_block(256, 512, 2))  
net = nn.Sequential(b1, b2, b3, b4, b5,  
 nn.AdaptiveAvgPool2d((1, 1)),  
 nn.Flatten(), nn.Linear(512, 10))

当较复杂的函数类包含较小的函数类时，我们才能确保提高它们的性能，如下图，非嵌套函数不能保证越来越接近目标函数(因为显然是不如的)。也就是当我们预测更强大的架构时，应该保证原有的函数。



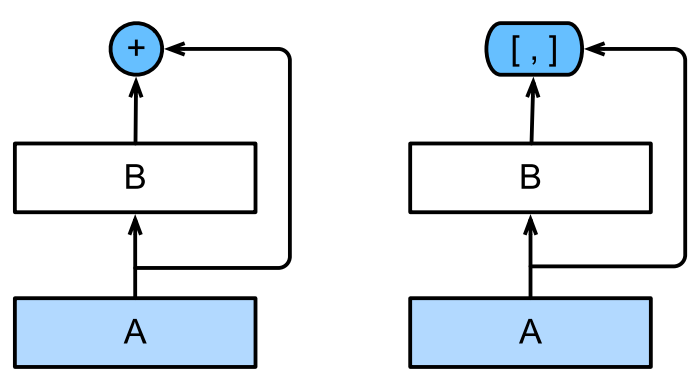
什么是残差，如下左图：我们定义当前块的最终输出是，也就是当前层的输出与上一块的输出的并集。当时，，称为恒等映射。

当use\_1x1conv=True时，会添加通过1×1卷积调整通道和分辨率，如下右图：

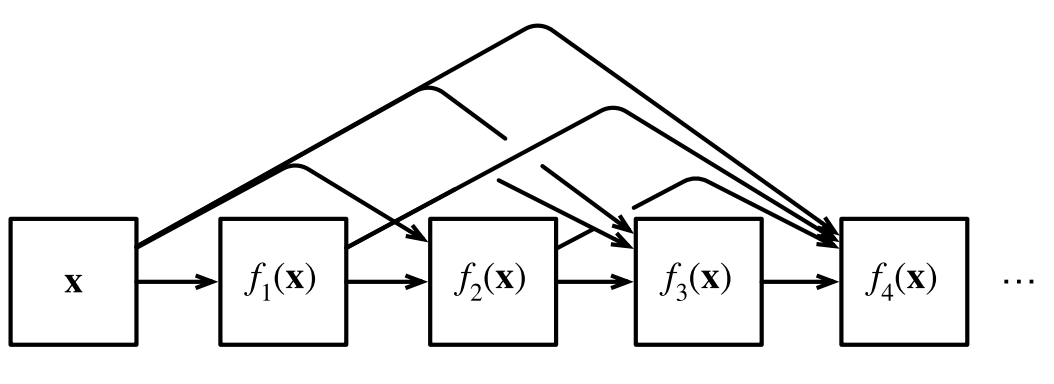
 

**DenseNet稠密连接网络**

def conv\_block(input\_channels, num\_channels):  
 return nn.Sequential(  
 nn.BatchNorm2d(input\_channels), nn.ReLU(),  
 nn.Conv2d(input\_channels, num\_channels, kernel\_size=3, padding=1))  
# 稠密块体  
class DenseBlock(nn.Module):  
 def \_\_init\_\_(self, num\_convs, input\_channels, num\_channels):  
 super(DenseBlock, self).\_\_init\_\_()  
 layer = []  
 for i in range(num\_convs):  
 layer.append(conv\_block(num\_channels \* i + input\_channels, num\_channels))  
 self.net = nn.Sequential(\*layer)  
 def forward(self, X):  
 for blk in self.net:  
 Y = blk(X)  
 # 连接通道维度上每个块的输入和输出  
 X = torch.cat((X, Y), dim=1)  
 return X  
# 过渡层  
def transition\_block(input\_channels, num\_channels):  
 return nn.Sequential(  
 nn.BatchNorm2d(input\_channels), nn.ReLU(),  
 nn.Conv2d(input\_channels, num\_channels, kernel\_size=1),  
 nn.AvgPool2d(kernel\_size=2, stride=2))  
# DenseNet模型  
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel\_size=7, stride=2, padding=3),  
 nn.BatchNorm2d(64), nn.ReLU(),  
 nn.MaxPool2d(kernel\_size=3, stride=2, padding=1))  
num\_channels, growth\_rate = 64, 32 # num\_channels为当前的通道数  
num\_convs\_in\_dense\_blocks = [4, 4, 4, 4]  
blks = []  
for i, num\_convs in enumerate(num\_convs\_in\_dense\_blocks):  
 blks.append(DenseBlock(num\_convs, num\_channels, growth\_rate))  
 # 上一个稠密块的输出通道数  
 num\_channels += num\_convs \* growth\_rate  
 # 在稠密块之间添加一个转换层，使通道数量减半  
 if i != len(num\_convs\_in\_dense\_blocks) - 1:  
 blks.append(transition\_block(num\_channels, num\_channels // 2))  
 num\_channels = num\_channels // 2  
net = nn.Sequential(  
 b1, \*blks,  
 nn.BatchNorm2d(num\_channels), nn.ReLU(),  
 nn.AdaptiveAvgPool2d((1, 1)),  
 nn.Flatten(),  
 nn.Linear(num\_channels, 10))

 相对于残差网络，这里将+变成×，也就是使用连接：

如右图，ResNet(左)与 DenseNet(右)在跨层连接上的主要区别是使用相加和使用连结。

 于是定义：稠密块dense block(如右图)和过渡层transition layer。前者定义如何连接输入和输出，而后者则控制通道数量，使其不会太复杂。

**为什么需要RNN**

这里以模拟的数据集y=sinx+噪声为例：

from easier\_nn.classic\_dataset import VirtualDataset as VD

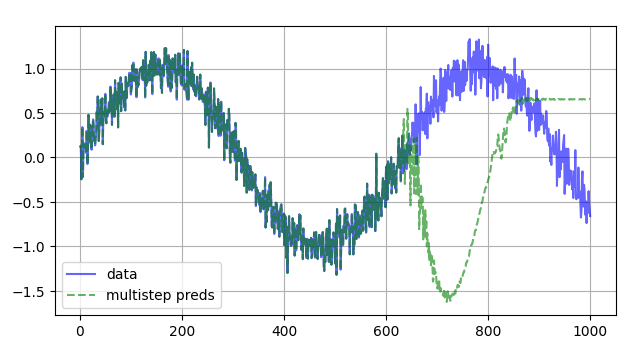
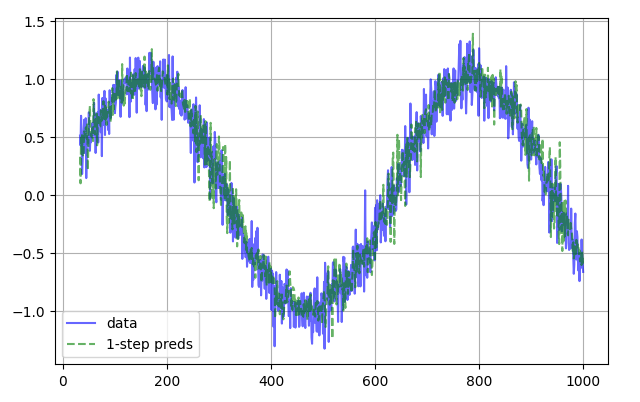
vd = VD(end=1000)  
vd.sinx(noise\_sigma=0.15, show\_plt=False)  
  
def get\_train\_iter(train\_class, tau=4, batch\_size=16, n\_train=600):  
 features = torch.zeros((train\_class.num\_points - tau, tau))  
 for i in range(tau):  
 features[:, i] = train\_class.y[i: train\_class.num\_points - tau + i]  
 labels = train\_class.y[tau:].reshape((-1, 1))  
 train\_iter = load\_array((features[:n\_train], labels[:n\_train]), batch\_size, if\_shuffle=True)  
 return train\_iter, features, labels  
  
tau = 32 # 书上是tau=4，但预测的结果不够好  
n\_train = 600  
train\_iter, features, labels = get\_train\_iter(vd, tau=tau, n\_train=n\_train) # 大小分别是[968, 32])，[968, 1]

训练网络（网络是输入是tau个数，输出是预测的一个数）：

net = nn.Sequential(nn.Linear(tau, 20), nn.ReLU(), nn.Linear(20, 8), nn.ReLU(), nn.Linear(8, 1))  
def init\_weights(m):  
 if type(m) == nn.Linear:  
 nn.init.xavier\_uniform\_(m.weight)  
net.apply(init\_weights)  
loss = nn.MSELoss(reduction='none')  
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)  
net = train\_net(features, labels, data\_iter=train\_iter, net=net, loss=loss, optimizer=optimizer, num\_epochs=300)  
# 保存net模型  
torch.save(net, '../model/test/rnn\_predict\_net\_tau=32.pth')  
# 加载net模型  
net = torch.load('../model/test/rnn\_predict\_net\_tau=32.pth')

单步预测和多步预测：

onestep\_preds = net(features)  
plot\_xys(x=vd.x.detach().numpy()[tau:], y\_list=[vd.y.detach().numpy()[tau:], onestep\_preds.detach().numpy()],  
 labels=['data', '1-step preds'], alpha=0.6)  
multistep\_preds = torch.zeros(vd.num\_points)  
multistep\_preds[: n\_train + tau] = vd.y[: n\_train + tau]  
for i in range(n\_train + tau, vd.num\_points):  
 multistep\_preds[i] = net(multistep\_preds[i - tau:i].reshape((1, -1)))  
plot\_xys(x=vd.x.detach().numpy(), y\_list=[vd.y.detach().numpy(), multistep\_preds.detach().numpy()],  
 labels=['data', 'multistep preds'], alpha=0.6)



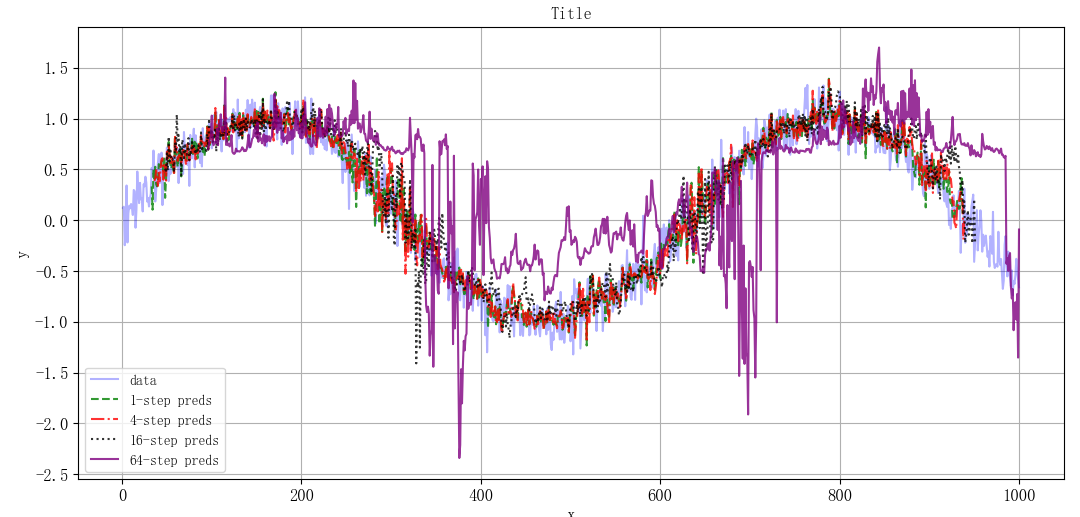
对于直到时间步t的观测序列，其在时间步t+k的预测输出是k步预测。

单步预测比多步更好，因为多步预测在600+tau之后的预测依赖于之前的预测，一旦出现误差，误差就会被累积。

对比步数：

max\_steps = 64  
features = torch.zeros((vd.num\_points - tau - max\_steps + 1, tau + max\_steps))  
# 列i（i<tau）是来自x的观测，其时间步从（i）到（i+T-tau-max\_steps+1）  
for i in range(tau):  
 features[:, i] = vd.y[i: i + vd.num\_points - tau - max\_steps + 1]  
# 列i（i>=tau）是来自（i-tau+1）步的预测，其时间步从（i）到（i+T-tau-max\_steps+1）  
for i in range(tau, tau + max\_steps):  
 features[:, i] = net(features[:, i - tau:i]).reshape(-1)

steps = (1, 4, 16, 64)  
colors = ['blue', 'green', 'red', 'black', 'purple', 'pink', 'orange', 'cyan']  
linstyles = ['-', '--', '-.', ':', 'solid', 'dashed', 'dashdot', 'dotted']  
# y\_list的列表的第1个元素是vd.y，后面的2~i+1个元素是features的第i列  
y\_list = [vd.y.detach().numpy()] + [features[:, tau + i - 1].detach().numpy() for i in steps]  
fig, ax = plt.subplots(figsize=(10, 6))  
ax = plot\_xy(x=vd.x.detach().numpy(), y=y\_list[0], label='data', use\_ax=True, ax=ax, show\_plt=False, alpha=0.3)  
for i, n in enumerate(steps):  
 ax = plot\_xy(x=vd.x.detach().numpy()[tau + n - 1: vd.num\_points - max\_steps + n], y=y\_list[i+1], alpha=0.8,  
 label=f'{n}-step preds', use\_ax=True, ax=ax, show\_plt=False,  
 color=colors[i+1], linestyle=linstyles[i+1])



步数越大预测越不准确，因为误差的累积更大。