**机器学习应用实践（实验一） ——线性回归（一）**

**工业智能2201班 刘天行 20225354**

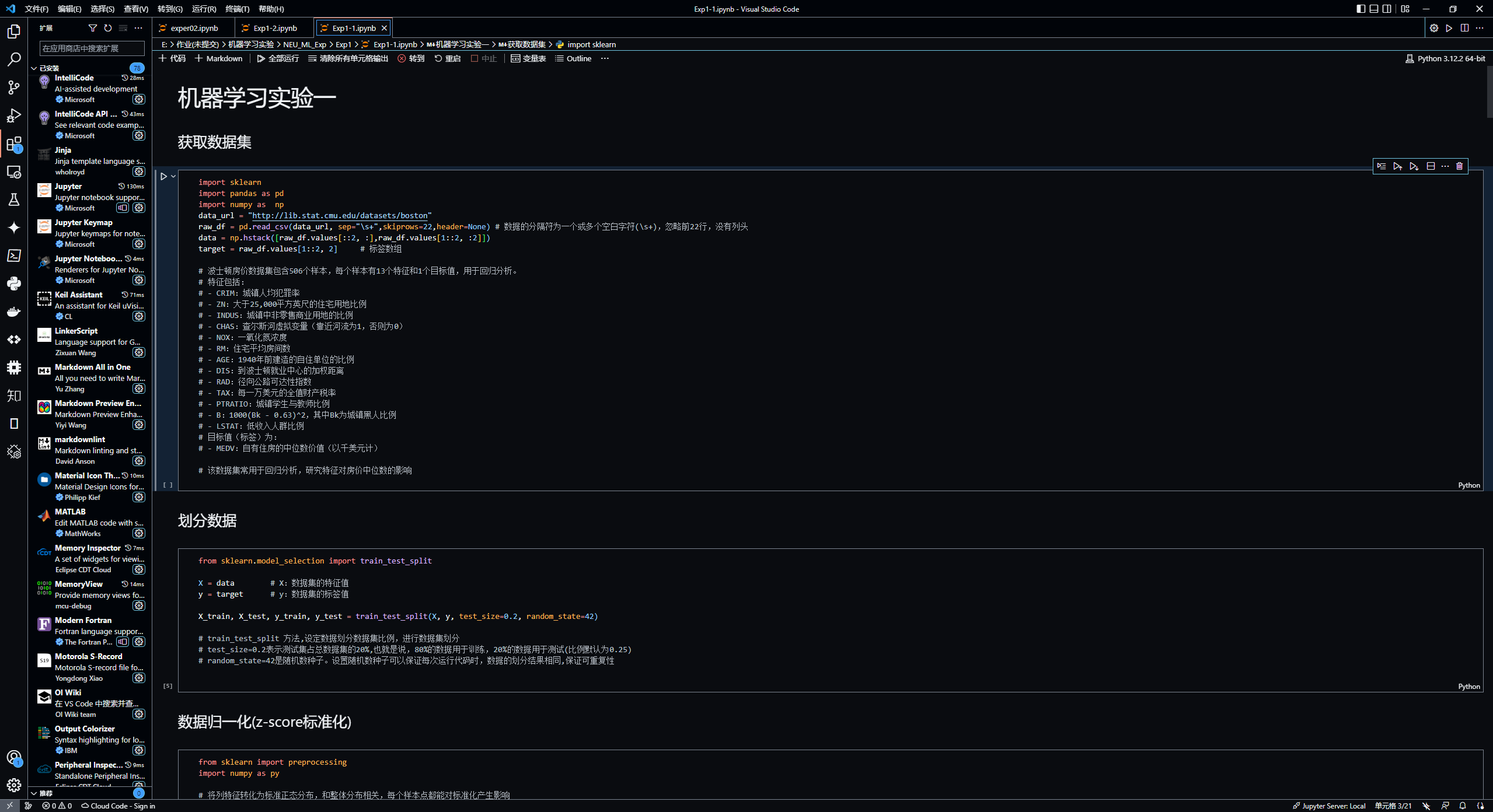
1. **实验目的**

1、掌握机器学习算法的开发流程。

2、掌握 Scikit-Learn 官方网站的查看、 学习和使用方法。

3、掌握 Scikit-Learn 中线性回归算法的使用， 解决波士顿房价预测问题。

1. **开发环境：****Visual Studio Code + Jupyter notebook拓展 + python 3.12.2**



1. **实验内容**

**题目一：采用 scikit-learn 中的 LinearRegression(最小二乘)线性回归模型对波士顿房价数据集进行预测，分别使用正则方程和随机梯度下降方法建模。**

**具体内容：**

1. 导入数据
2. 查看数据集的描述、特征名、标签名、数据样本量等信息。
3. 获取样本的特征数据和标签数据。
4. 划分数据（分成训练集和测试集）
5. 数据归一化
6. 训练模型
7. 使用 sklearn 中线性回归的正规方程（LinearRegression）优化方法建模。
8. 使用 sklearn 中线性回归的随机梯度下降（SGDRegressor）优化方法建模。
9. 模型评估（2 个模型）

评价指标： MSE 和 R2 值。

**根据这样的实验步骤，进行一系列实验，回答以下四个讨论题：**

**【讨论一】 梯度下降和正规方程两种算法有何不同？**

分析梯度下降和正规方程两种算法的差异（计算时间、 评价指标对比）与优劣点。提示： python 中计时器-timeit.default\_timer()方法。

**【讨论二】数据归一化对算法有什么影响？**

对比使用数据归一化和不使用数据归一化，正规方程和梯度下降算法性能是否有差异？分析原因。

**【讨论三】 梯度下降算法中的学习率如何影响其工作？**

尝试修改随机梯度下降算法(SGDRegressor)的学习率， 观察参数对模型性能的影响（通过数据分析即可）， 试分析学习率与模型性能之间的关系。

**【讨论四】（选做）模型的泛化能力如何？**

（1）分别计算模型在训练样本上性能和在测试样本上的性能，判断模型是过拟合还是欠拟合？

（2）数据集划分不同对模型性能是否有影响？可尝试修改方法(train\_test\_split)中的 test\_size 参数，观察数据集划分对模型性能的影响。

（3）尝试使用其他线性回归模型。线性回归模型中除了 LinearRegression，还有 Ridge(岭回归)、Lasso、 Polynomial regression（多项式回归）等模型，自学官网资料，使用不同模型进行建模，观察不同模型训练后的模型权重差异，试分析模型的使用场合。

**题目二：采用梯度下降法（BGD）优化线性回归模型，对波士顿房价进行预测。**

（ 1）导入数据（从.csv 文件中导入数据代码如下）

（ 2）划分数据（分成训练集和数据集）

（ 3）数据归一化

（ 4）训练模型 model(train\_x,train\_y)

1. 初始化参数 w。可使用 np.concatenate 数组拼接函数，将截距与权重参数合并在一起（也可以不拼接合并）。
2. 求 f(x)。
3. 求 J(w)。
4. 求梯度。
5. 更新参数 w。
6. 的过程经过 epochs 次迭代。

（5）画出损失函数虽迭代次数的变化曲线。（ 通过损失函数变化曲线来观察梯度下降执行情况）

（6）测试集数据进行预测，模型评估。

（7）可视化： 展示数据拟合的效果。

**题目三（选做） ：小批量梯度下降算法（ MBGD）的编程实现。**

1. **实验情况**

**题目一：**

获取数据集：

import pandas as pd

import numpy as np

# 设置数据文件的路径

data\_path = "boston\_house\_prices.csv"

# 使用逗号作为分隔符加载数据

raw\_df = pd.read\_csv(data\_path, sep=",")

# 将第一行设置为列头

raw\_df.columns = raw\_df.iloc[0]

raw\_df = raw\_df[1:]  # 删除重复的列名行

# 转换数据类型为数值型，方便处理

raw\_df = raw\_df.apply(pd.to\_numeric, errors='coerce')

# 提取特征值 X 和目标值 y

X = raw\_df.drop(columns=["MEDV"])  # 移除目标列以分离特征

y = raw\_df["MEDV"].values  # 将目标值列提取为数组

# 打印数据结构和一些样本以确认

print(raw\_df.head())

print(X.head())

print(y[:5])

# 波士顿房价数据集是一个用于回归分析的数据集，其中包含506个观测值，每个观测值都有13个特征和1个目标值。

# 这13个特征值包括：

# CRIM：城镇人均犯罪率。

# ZN：大于25,000平方英尺的住宅土地的比例。

# INDUS：城镇非零售业务地区的比例。

# CHAS：查尔斯河虚拟变量（如果土地在河边，则为1；否则为0）。

# NOX：一氧化氮浓度。

# RM：住宅平均房间数。

# AGE：1940年以前建造的自住房屋的比例。

# DIS：到波士顿五个就业中心的加权距离。

# RAD：径向公路可达性指数。

# TAX：每10,000美元的全额财产税率。

# PTRATIO：城镇学生与教师比例。

# B：1000(Bk - 0.63)^2，其中Bk是城镇黑人的比例。

# LSTAT：人口中低收入者的比例。

# 目标值/标签值是：

# MEDV：自有住房的中位数价值（以1000美元计）。

# 波士顿房价数据集的特征值是这13个与房价相关的特征，而标签值是房价的中位数价值

X.shape, y.shape

划分数据：

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# train\_test\_split 方法,设定数据划分数据集比例，进行数据集划分

# test\_size=0.2表示测试集占总数据集的20%,也就是说，80%的数据用于训练，20%的数据用于测试(比例默认为0.25)

# random\_state=42是随机数种子。设置随机数种子可以保证每次运行代码时，数据的划分结果相同,保证可重复性

数据归一化(z-score标准化)：

from sklearn import preprocessing

import numpy as py

# 将列特征转化为标准正态分布，和整体分布相关，每个样本点都能对标准化产生影响

# 对于每个数值特征，算出它的均值（mean）和标准差（std）

# Z = (X - mean) / std

scaler = preprocessing.StandardScaler().fit(X\_train) # 统计训练集的均值、方差

X\_train\_std = scaler.transform(X\_train)

X\_test\_std  = scaler.transform(X\_test)

# 将每一个特征都转化为标准正态分布

# 标准化的主要目的是消除不同特征之间的尺度差异，让每个特征都处在同样的尺度上

# 用归一化的数据代替原始数据

X\_train = X\_train\_std

X\_test  = X\_test\_std

至此，完成全部数据预处理的部分

1. **梯度下降和正规方程两种算法有何不同？**

此处我们在上面对数据集进行观察的基础上进行如下的操作：

**训练模型：**

正规方程求解线性回归：

from sklearn.linear\_model import LinearRegression

import time

from sklearn.metrics import mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

reg = LinearRegression()# 初始化线性回归模型

start\_time = time.time()# 记录起始时间

reg.fit(X\_train, y\_train)# 拟合训练数据

print("Linear Regression fitting time:", time.time() - start\_time, "seconds")# 打印拟合过程所耗时间

y\_pred = reg.predict(X\_test)# 使用测试集进行预测

# 计算训练集和测试集的均方误差（MSE）

train\_mse = mean\_squared\_error(y\_train, reg.predict(X\_train))

test\_mse = mean\_squared\_error(y\_test, y\_pred)

# 打印训练集和测试集的MSE

print("Training MSE: ", train\_mse)

print("Testing MSE:  ", test\_mse)

# 计算训练集和测试集的R²（决定系数）

train\_r2 = r2\_score(y\_train, reg.predict(X\_train))

test\_r2 = r2\_score(y\_test, y\_pred)

# 打印训练集和测试集的R²

print("Training R^2: ", train\_r2)

print("Testing R^2: ", test\_r2)

# 绘制真实值和预测值的折线图

plt.figure(figsize=(12, 6))

plt.plot(y\_test, label='Actual Values', color='blue')

plt.plot(y\_pred, label='Predicted Values', color='orange')

plt.legend()

plt.title("Actual vs Predicted Values")

plt.xlabel("Sample Index")

plt.ylabel("Value")

plt.show()

# 获取模型系数和截距

coefficients = reg.coef\_

intercept = reg.intercept\_

# 打印模型系数和截距

print("Coefficients:\n", coefficients)

print("Intercept:", intercept)

# 绘制线性回归模型的图像

plt.figure(figsize=(10, 6))

plt.scatter(X\_test[:, 0], y\_test, color='blue', label='Test Data')

plt.plot(X\_test[:, 0], coefficients[0] \* X\_test[:, 0] + intercept, color='red', label='Regression Line')

plt.xlabel("Feature")

plt.ylabel("Target")

plt.title("Linear Regression Model")

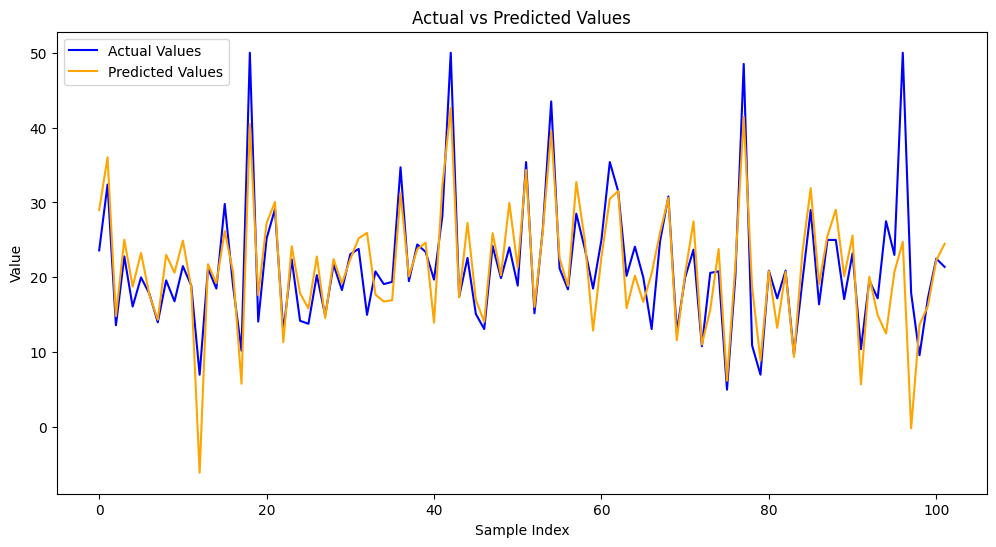
plt.legend()

plt.show()

运行结果：

Linear Regression fitting time: 0.0019974708557128906 seconds Training MSE: 21.641412753226312 Testing MSE: 24.291119474973517 Training R^2: 0.7508856358979673 Testing R^2: 0.668759493535632

Coefficients: [-1.00213533 0.69626862 0.27806485 0.7187384 -2.0223194 3.14523956 -0.17604788 -3.0819076 2.25140666 -1.76701378 -2.03775151 1.12956831 -3.61165842] Intercept: 22.796534653465343



图表, 散点图

描述已自动生成

从这个图像中可以看出，正规方程得到的的线性回归模型并不能很好地拟合数据，因为数据的分布并不是线性的，所以这个方法并不适合这个数据集

梯度下降法求解线性回归问题：

from sklearn.linear\_model import SGDRegressor

import time

from sklearn.metrics import mean\_squared\_error

import matplotlib.pyplot as plt

# 随机梯度下降算法在每次更新中根据一个样本计算损失函数的梯度

# 其优点是运行速度快，适用于大批量数据训练。

# SGDRegressor 可以支持不同的 loss函数和正则化惩罚项来拟合线性回归模型。

# 设置字体

plt.rcParams['font.family'] = 'SimHei'

plt.rcParams['axes.unicode\_minus'] = False

# 设置多次循环次数和初始学习率

iterations = 10

initial\_eta = 0.0005

# 存储预测结果

y\_predict\_list = []

# 对不同学习率进行模型训练和评估

for i in range(iterations):

    eta = initial\_eta \* (i + 4) # 设置当前学习率

    reg = SGDRegressor(eta0=eta, random\_state=42)# 创建SGDRegressor模型，设置学习率和随机种子

    start\_time = time.time()# 记录起始时间

    reg.fit(X\_train, y\_train)# 拟合模型

    y\_predict = reg.predict(X\_test)# 预测测试集

    y\_predict\_list.append(y\_predict)# 记录预测结果

    elapsed\_time = time.time() - start\_time# 计算训练时间

    # 计算MSE和R²

    test\_mse = mean\_squared\_error(y\_test, y\_predict)

    test\_r2 = r2\_score(y\_test, y\_predict)

    # 绘制真实值和预测值的折线图

    plt.figure()

    plt.plot(y\_test, label='真实值', color='purple')

    plt.plot(y\_predict, label='预测值', color='orange')

    plt.legend()

    plt.title(f"学习率={eta:.6f}, MSE={test\_mse:.6f}, R^2={test\_r2:.6f}, 训练耗时={elapsed\_time:.6f}秒")

# 绘制最后一次测试集和预测值的折线图

plt.figure()

plt.plot(y\_test, label='真实值', color='purple')

plt.plot(y\_predict, label='预测值', color='orange')

plt.legend()

plt.title(f"学习率={eta:.6f}, MSE={test\_mse:.6f}, R^2={test\_r2:.6f}, 训练耗时={elapsed\_time:.6f}秒")

plt.show()

# 查看训练完成后的SGDRegressor模型的系数和截距

print("R2:",reg.score(X\_train, y\_train))# R^2

print("coef:\n",reg.coef\_)# 系数

print("intercept:",reg.intercept\_)# 偏置

print("MSE:",mean\_squared\_error(y\_test, y\_predict))# 均方误差

print("params:",reg.get\_params())# 参数

# 可视化

plt.figure(figsize=(10,5))

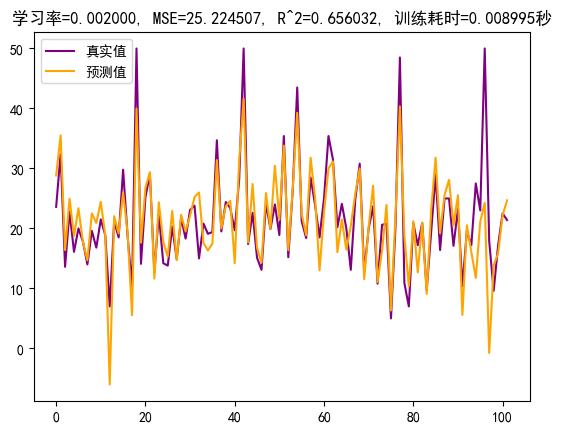
plt.plot(y\_test)

plt.plot(y\_predict)

plt.xticks(())

plt.show()

运行结果：



图表, 折线图

描述已自动生成

图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成

R2: 0.749384748659065 coef: [-0.90158104 0.49285972 -0.00631569 0.76490925 -1.75383491 3.26184982 -0.19531696 -2.82857441 1.37910373 -0.87348503 -1.96813398 1.13438027 -3.5867964 ] intercept: [22.81140031] MSE: 25.054530831361923 params: {'alpha': 0.0001, 'average': False, 'early\_stopping': False, 'epsilon': 0.1, 'eta0': 0.006500000000000001, 'fit\_intercept': True, 'l1\_ratio': 0.15, 'learning\_rate': 'invscaling', 'loss': 'squared\_error', 'max\_iter': 1000, 'n\_iter\_no\_change': 5, 'penalty': 'l2', 'power\_t': 0.25, 'random\_state': 42, 'shuffle': True, 'tol': 0.001, 'validation\_fraction': 0.1, 'verbose': 0, 'warm\_start': False}

正规化方程优化是通过最小二乘法直接计算出的结果，训练集不变，结果、计算量也不会改变。随机梯度下降是对每个批次的数据中随机抽取一个计算损失、修正系数，结果具有随机性，计算量也不确定。

重复拟合并通过metric. mean\_squared\_error计算每一次预测结果与标签的误差平方和的平均值，得到MSE，再计算1000次的平均值mean\_MSE。可以发现最小二乘法的MSE是固定的；随机梯度下降法的结果不稳定，但整体的值小于最小二乘法，说明随机梯度下降法的拟合效果较好。

1. **数据归一化对算法有什么影响？**

将这两行代码注释：

文本

描述已自动生成

直接利用split之后的数据集进行训练，不进行归一化，最终预测结果相近。系数的分布有所不同。

分别运行修改后和修改前的代码，得到下面的结果

正规方程，未归一化: MSE = 24.2911, R2 = 0.6687, 时间 = 0.0019 秒

正规方程，归一化: MSE = 24.2911, R2 = 0.6687, 时间 = 0.0004 秒

梯度下降，未归一化: MSE = 196.8981, R2 = -1.2862, 时间 = 0.001999 秒

梯度下降，归一化: MSE = 25.0545, R2 = 0.6583, 时间 = 0.00205 秒

从结果上看，归一化后的系数分布在0~10的范围内，便于计算；

从过程上看，不归一化的训练集需要每次设定不同的学习率，不同特征需要的最佳学习率是数量级也不同，设置不合理会导致无法收敛；

从拟合方法上看，最小二乘法几乎不受影响，随机梯度下降收到了较大的影响。

将所有特征的数据归一化，缩放至同一尺度内，可以防止不同特征的属性值差异过大，导致系数更新速率不同，影响最终训练结果。

1. **梯度下降算法中的学习率如何影响其工作？**

为了探究学习率对梯度下降算法的影响，我们设定了一组从0.002到0.0065不同的学习率，分别用随机梯度下降法训练回归模型，依然通过上述三个指标来对判断模型的好坏并判断学习率对随机梯度下降法算法性能的影响。

运行代码得到的结果在上面可见，其中

学习率 = 0.002, MSE = 25.224, R2 = 0.656, 运行时间 = 0.00555 秒

学习率 = 0.0025, MSE =25.00405, R2 = 0.659, 运行时间 = 0.0045 秒

学习率 = 0.003, MSE =25.097, R2 = 0.657, 运行时间 = 0.0029 秒

学习率 = 0.0035, MSE =24.945, R2 = 0.659, 运行时间 = 0.0029 秒

学习率 = 0.004, MSE =24.828, R2 = 0.661, 运行时间 = 0.0030 秒

学习率 = 0.0045, MSE =24.846, R2 = 0.661, 运行时间 = 0.0029 秒

学习率 = 0.005, MSE =24.755, R2 = 0.662, 运行时间 = 0.0033 秒

学习率 = 0.0055, MSE =25.228, R2 = 0.655, 运行时间 = 0.0020 秒

学习率 = 0.006, MSE =25.135, R2 = 0.657, 运行时间 = 0.0020 秒

学习率 = 0.0065, MSE =25.054, R2 = 0.658, 运行时间 = 0.0026 秒

不同的学习率的模型MSE和并无太大差距别，运行时间变化趋势也并非单调的。学习率在梯度下降算法中控制着参数更新的步长。

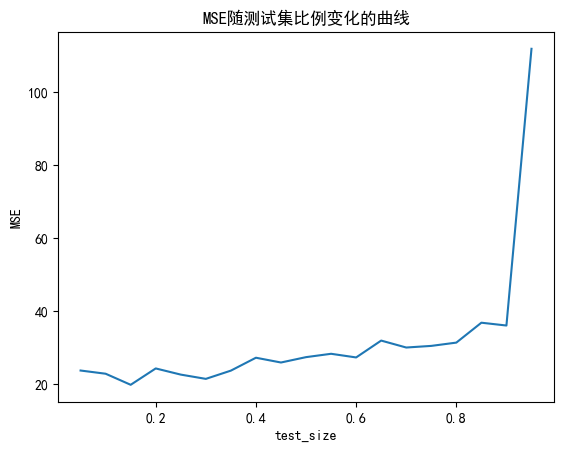
过高的学习率可能导致模型在最小值附近震荡甚至发散，而过低的学习率会导致收敛过慢。

1. **数据集的划分对模型性能的影响**

为了探究训练集和测试机比例对模型的影响，我使用np.linspace生成一个包含19个值的列表，表示测试集比例的变化范围（从0.05到0.95），分别训练模型并比较MSE的变化

1. import numpy as np
2. from sklearn.model\_selection import train\_test\_split
3. from sklearn import preprocessing
4. from sklearn.linear\_model import LinearRegression
5. from sklearn.metrics import mean\_squared\_error
6. import matplotlib.pyplot as plt
7. data\_file = r"boston\_house\_prices.csv"
8. with open(data\_file, encoding='utf-8') as f:
9. data = np.loadtxt(data\_file, delimiter=',', skiprows=2)
10. X = data[:, :-1]    # X：数据集的特征值
11. y = data[:, -1]     # y：数据集的标签值
12. test\_MSE\_list = []
13. rates = np.linspace(0.05,0.95,19).tolist()  # np.linspace生成一个包含19个值的列表，表示测试集比例的变化范围（从0.05到0.95）
14. for rate in rates:
15. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=rate, random\_state=54)
16. std\_scaler = preprocessing.StandardScaler().fit(X\_train) # 统计训练集的均值、方差
17. X\_train\_std = std\_scaler.transform(X\_train)
18. X\_test\_std  = std\_scaler.transform(X\_test)
20. # 用归一化的数据代替原始数据
21. X\_train = X\_train\_std
22. X\_test  = X\_test\_std
23. reg = LinearRegression()            # 创建对象,采用的还是正规方程法
24. reg.fit(X\_train, y\_train)           # 拟合
25. y\_predict = reg.predict(X\_test)     # 测试集预测值
26. test\_MSE\_list.append(mean\_squared\_error(y\_test, y\_predict))
27. plt.plot(rates,test\_MSE\_list)
28. plt.xlabel("test\_size")
29. plt.ylabel("MSE")
30. plt.title("MSE随测试集比例变化的曲线")
31. plt.show()
32. # 由结果图可以看出，当测试集比例在0.05到0.15之间时，MSE值较小，说明模型的预测能力较好
33. # 当测试集比例大于0.15时，MSE值逐渐增大，说明模型的预测能力变差
34. # 因为训练集的数据越多，模型的训练效果才越好，所以测试集比例越大，模型的预测能力越差

运行结果如图：



由结果图可以看出，当测试集比例在0.05到0.15之间时，MSE值较小，说明模型的预测能力较好

当测试集比例大于0.15时，MSE值逐渐增大，说明模型的预测能力变差，因为训练集的数据越多，模型的训练效果才越好，所以测试集比例越大，模型的预测能力越差，足够的训练数据可以帮助线性回归模型更准确地估计模型参数，提升模型的预测性能。如果训练数据量不足，模型可能无法捕捉数据的真实模式，从而影响模型的预测精度。如果训练集太小，模型可能会过拟合训练数据，即在训练数据上表现良好但在测试数据上表现不佳。反之，如果训练集太大，而测试集太小，可能会低估模型在未见数据上的泛化能力。

5. **使用其他的线性回归模型**

我分别使用实验教学文档中提到的Lasso，Ridge和多项式回归来进行模型的训练

代码如下：

Lasso

from sklearn.linear\_model import Lasso

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import make\_pipeline

from sklearn.metrics import mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

# 创建一个Pipeline对象，包含两个步骤：首先，使用StandardScaler对特征进行标准化；

# 然后，使用Lasso进行回归。这个Pipeline对象就相当于一个带有Lasso正则化的线性回归模型

pipeline = make\_pipeline(StandardScaler(), Lasso(alpha=0.1))

# 用训练数据来训练这个Pipeline

pipeline.fit(X\_train, y\_train)

# 在测试集上进行预测

y\_predict = pipeline.predict(X\_test)

# 计算相关系数（R²得分）和均方误差（MSE）

train\_score = pipeline.score(X\_train, y\_train)

test\_mse = mean\_squared\_error(y\_test, y\_predict)

test\_r2 = r2\_score(y\_test, y\_predict)

# 打印相关系数和均方误差

print('训练集相关系数: ', train\_score)

print("测试集MSE: ", test\_mse)

print("测试集R²: ", test\_r2)

# 绘制真实值和预测值的折线图

plt.figure(figsize=(12, 6))

plt.plot(y\_test, label='真实值', color='purple')

plt.plot(y\_predict, label='预测值', color='orange')

plt.legend()

plt.title("Lasso回归预测结果")

plt.xlabel("样本索引")

plt.ylabel("房价")

plt.show()

# 通过调节Lasso模型中的alpha参数，可以控制正则化的强度，避免过拟合或欠拟合

# 较高的alpha值会增加正则化力度，减少过拟合的风险，但可能导致欠拟合

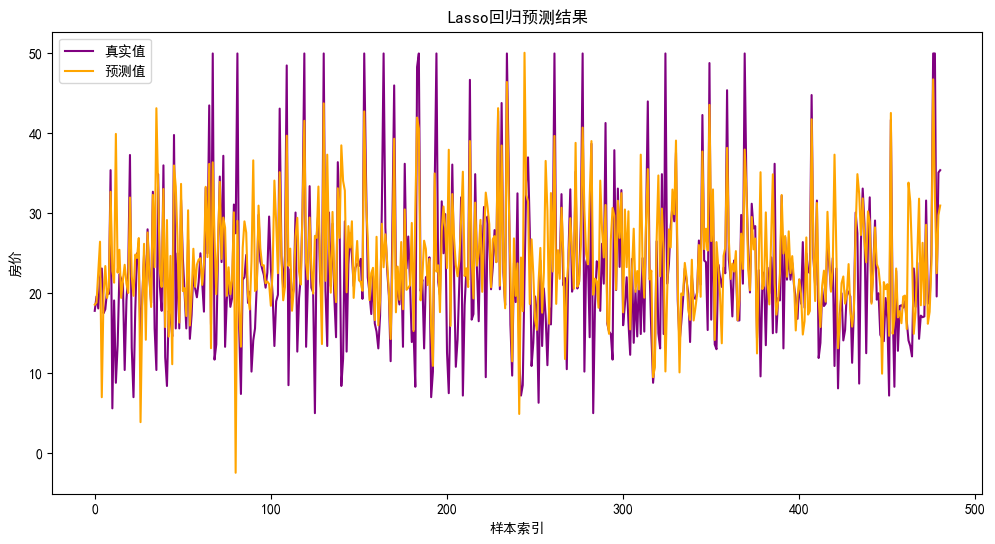
# 较低的alpha值会减少正则化力度，模型可能更好地拟合训练数据，但可能导致过拟合

运行结果：

训练集相关系数: 0.9288132515243274

测试集MSE: 69.7909090945629

测试集R²: 0.18962577021764915



Ridge

from sklearn.linear\_model import Ridge

# 这段代码使用了Ridge回归模型，这是一种线性回归模型，

# 在多元线性回归基础上加入了L2正则项，针对模型中存在的共线性关系的为变量增加一个小的平方偏差因子，防止不同特征之间相互关联影响模型训练，来约束模型的权重

# 这里的alpha参数就是正则化项的强度，alpha的值越大，正则化项的影响就越大。在这个例子中，alpha的值被设置为0.5。

from sklearn.model\_selection import GridSearchCV

# 定义要搜索的alpha值

parameters = {'alpha': [1e-10, 1e-5, 1e-4, 1e-3,1e-2, 1, 5, 10,11, 20]}

ridge = Ridge()

# 创建GridSearchCV对象，使用折交叉验证（cv=5），训练数据被分成了5份，

# 每次都使用其中的4份来训练模型，然后在剩下的1份上进行验证，这样一共可以进行5次训练和验证，更准确地评估模型的性能

ridge\_regressor = GridSearchCV(ridge, parameters, scoring='neg\_mean\_squared\_error', cv=5)

# 使用GridSearchCV对象拟合数据

ridge\_regressor.fit(X\_train, y\_train)

# 打印最优的alpha值

print(ridge\_regressor.best\_params\_)

# 打印最优模型的MSE

print(ridge\_regressor.best\_score\_)

# 使用最优模型在测试集上进行预测

y\_predict = ridge\_regressor.predict(X\_test)

# 计算并打印测试集上的MSE

reg = Ridge(alpha=.5)            # 创建对象

reg.fit(X\_train, y\_train)           # 拟合

y\_predict = reg.predict(X\_test)     # 测试集预测值

print("test\_MSE: ",mean\_squared\_error(y\_test, y\_predict))

plt.title("Ridge回归预测结果")

plt.plot(y\_test)

plt.plot(y\_predict)

plt.plot(y\_test,label='Test')

plt.plot(y\_predict,label='Predict')

plt.legend() # 添加图例

plt.show()

# 最终在测试集上的均方误差比直接多元线性回归小一些，说明正则项损失函数是比较有效的

运行结果：

{'alpha': 5}

-30.986120973531854

test\_MSE: 126.55507028531561

图表, 折线图

描述已自动生成

多项式回归

from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make\_pipeline

# Pipeline是一个包装器，用于封装多个处理步骤，以便它们可以被当作一个整体来使用

# 首先利用PolynomialFeatures构建数据集的各阶幂，再利用之前的多元线性回归LinearRegression()进行拟合

# 创建一个Pipeline对象，包含两个步骤：首先，使用PolynomialFeatures生成2阶的多项式特征；

# 然后，使用LinearRegression进行回归。这个Pipeline对象就相当于一个2阶的多项式回归模型

pipeline = make\_pipeline(PolynomialFeatures(5), LinearRegression())

# 当模型多项式阶数大于2阶，相关系数为 1.0，表示在训练集上模型的拟合效果非常好，然而测试集上的均方误差却相当高

# 这是过拟合（overfitting）的表现，这里选用1阶

# 用训练数据来训练这个Pipeline。

# 这个过程包括两个步骤：首先，使用PolynomialFeatures对训练数据进行转换，生成 2 阶的多项式特征；然后，使用LinearRegression拟合这些特征和目标值

pipeline.fit(X\_train,y\_train)                                      #训练模型

y\_predict = pipeline.predict(X\_test)#在测试集上预测结果并保存在pred变量中

score = pipeline.score(X\_train,y\_train)#计算相关系数

print('相关系数: ', score)

print("test\_MSE: ",mean\_squared\_error(y\_test, y\_predict))

plt.plot(y\_test,label='Test')

plt.plot(y\_predict,label='Predict')

plt.legend() # 添加图例

plt.title("多项式回归预测结果,采用训练阶数：{}".format(5))

plt.show()

# 经过测试发现，PolynomialFeatures(n)中阶数n=2时效果尚可，比直接线性回归稍差一些。当n>=3时会发散，即过拟合

运行结果：

相关系数: 1.0 test\_MSE: 121108342215.88416

图表, 条形图, 直方图

描述已自动生成

通过查阅更多资料以及对上面结果的分析，我们可以总结出：

1. 线性回归：适用于特征数量较少且多重共线性不严重的数据集。
2. 岭回归：适用于特征数量较少且多重共线性不严重的数据集。
3. Lasso 回归：适用于特征数量较少且多重共线性不严重的数据集。
4. 多项式回归：适用于特征数量较少且多重共线性不严重的数据集。

**题目二：**

本题目需要自己编写梯度下降法BGD函数来优化线性回归模型

import numpy as np

import matplotlib.pyplot as plt

import time

plt.rcParams['axes.unicode\_minus'] = False

def load\_data(test\_rate=0.2):

    # 1.从文件导入数据

    data\_file = r"boston\_house\_prices.csv"

    with open(data\_file, encoding='utf-8') as f:

        data = np.loadtxt(data\_file, delimiter=',', skiprows=2)

    X = data[:, :-1]

    y = data[:, -1]

    # 随机打乱数据

    np.random.seed(42)  # 使用相同的随机种子，保证每次运行得到相同的结果

    shuffle\_indices = np.random.permutation(len(X)) # 生成随机序列

    X, y = X[shuffle\_indices], y[shuffle\_indices]   # 用随机序列对数据进行重排

    # 2.将原数据集拆分成训练集和测试集，按照test\_rate划分

    num\_test = int(len(X) \* test\_rate)

    X\_test, y\_test = X[:num\_test], y[:num\_test]

    X\_train, y\_train = X[num\_test:], y[num\_test:]

    # 3、返回训练和测试数据集的特征和标签

    return X\_train, y\_train, X\_test, y\_test

def data\_processing(train\_data, test\_data, kind=1):

    # 1、传入train\_data,test\_data以及选择那一种归一化的kind 参数

    train\_std = np.zeros\_like(train\_data, dtype=float)  # 生成一个和train\_data维度相同的全0数组

    test\_std = np.zeros\_like(test\_data, dtype=float)

    # 当kind=1时，进行z-score标准化处理；当kind=2时，进行最大最小归一化处理

    if kind==1:

        avg = np.mean(train\_data, axis=0)   # 均值

        std = np.std(train\_data, axis=0)    # 标准差

        train\_std = (train\_data - avg) / std

        test\_std = (test\_data - avg) / std

    # 3.最大最小归一化处理

    else:

        max = np.max(train\_data, axis=0)

        min = np.min(train\_data, axis=0)

        train\_std = (train\_data - min)/(max - min)

        test\_std = (test\_data - min)/(max - min)

    # 返回处理后的数据集

    return train\_std, test\_std

def get\_batches(dataset, labels, batch\_size=1, drop\_last=False):    # 将数据集划分为多个批次，每个批次的样本数量为 batch\_size

    # drop\_last：是否丢弃最后一个不足batch\_size的批次

    batches = []

    index = 0

    data\_len = dataset.shape[0]

    if drop\_last:

        data\_len -= data\_len % batch\_size

    while index < data\_len:

        # 如果数据集的长度大于等于index+batch\_size，则取出index到index+batch\_size的数据

        if index + batch\_size < data\_len:

            x = dataset[index:index + batch\_size]

            y = labels[index:index + batch\_size]

        elif not drop\_last:

            x = dataset[index:]

            y = labels[index:]

        index += batch\_size

        batches.append([x, y])

    # 返回划分好的批次,其中包含了批次数据集的的特征和标签

    return batches

class LinearRegression\_numpy(object):

    def \_\_init\_\_(self, num\_of\_weights):

        # 初始化系数w的值，利用随机数生成函数生成服从标准正态分布的随机数

        # 这里是将截距 b并入了系数w中，所以系数w的长度比特征数多1

        self.w = np.random.randn(num\_of\_weights + 1)

    # 将预测输出的过程以“类和对象”描述

    def forward(self, X):

        # y\_pred = np.dot(X, self.w)  # 矩阵乘法

        # 这个函数的作用是将X和self.w进行矩阵乘法运算，得到y\_pred

        return np.dot(X, self.w[1:]) + self.w[0]

    def loss(self, X, y):

        # 这个函数的作用是计算模型在数据集X上的均方误差

        y\_pred = self.forward(X)            # 调用forward函数，得到预测值

        return np.mean((y - y\_pred)\*\*2)

    def gradient(self, X, y):

        # 这个函数的作用是计算模型在数据集X上的梯度

        y\_pred = self.forward(X)            # 调用forward函数，得到预测值

        grad\_w = -2/len(X) \* np.dot(X.T, (y - y\_pred))  # 根据公式计算梯度，这里是对w求导

        grad\_b = -2/len(X) \* np.sum(y - y\_pred)     # 根据公式计算梯度，这里是对b求导

        return grad\_w, grad\_b

    def update(self, grad\_w, grad\_b, eta):

        # 这个函数的作用是根据梯度和学习率更新系数w（包含截距）

        self.w[1:] -= eta \* grad\_w  # 根据梯度和学习率更新系数w

        self.w[0] -= eta \* grad\_b   # 根据梯度和学习率更新截距b

    def train\_BGD(self, X, y, num\_epoches, eta):

        # 这个函数的作用是使用梯度下降法训练模型

        loss\_history = [] # 用于保存每次迭代后的损失函数值

        for epoch\_id in range(num\_epoches): # 开始迭代，迭代周期是num\_epoches，每次迭代都要计算当前的梯度，并更新参数

            grad\_w, grad\_b = self.gradient(X, y)    # 计算梯度，计算当前参数下损失函数的梯度

            self.update(grad\_w, grad\_b, eta)        # 更新参数

            loss = self.loss(X, y)

            loss\_history.append(loss)               # 计算损失函数的值，保存到loss\_history中

        return loss\_history

    def train\_MBGD(self, X, y, num\_epoches, batch\_size, eta, drop\_last=False):

        loss\_history = []           # 用于保存每次迭代后的损失函数值

        np.random.seed(54)          # 设置随机种子，保证每次运行结果一致

        for epoch\_id in range(num\_epoches):

            # for X, y in get\_batches(X, y, batch\_size, drop\_last=drop\_last):

                # 经典的四步训练流程：前向计算->计算损失->计算梯度->更新参数（分别调用类的方法）

                indices = np.random.choice(len(X), batch\_size)  # 随机采样batch\_size个样本，返回采样样本的下标

                X\_batch, y\_batch = X[indices], y[indices]       # 根据下标取出对应的样本

                loss = self.loss(X\_batch, y\_batch)              # 计算损失函数值，保存到loss\_history中

                loss\_history.append(loss)

                grad\_w, grad\_b = self.gradient(X\_batch, y\_batch)# 计算当前参数下损失函数的梯度

                self.update(grad\_w, grad\_b, eta)                # 更新参数

        return loss\_history

plt.rcParams['axes.unicode\_minus'] = False

plt.rcParams['font.family'] = 'SimHei'

plt.rcParams['font.size'] = 11

num\_epoches = 2000 # 训练轮数

lr = 0.001234        # 学习率，BGD 和 MBGD 的学习率都采用这个值

# 1、获取数据并划分数据集

X\_train, y\_train, X\_test, y\_test = load\_data(test\_rate=0.2)

# 2、数据标准化/归一化处理

# data\_processing(X\_train, X\_test)

X\_train, X\_test = data\_processing(X\_train, X\_test, 1)

# 3、LinearRegression\_numpy 类实例化

reg\_bgd = LinearRegression\_numpy(X\_train.shape[1])# 调用类中自定义的 train\_BGD()，训练模型

start = time.time()

losses\_bgd = reg\_bgd.train\_BGD(X\_train, y\_train, num\_epoches=num\_epoches, eta=lr)

print("BGD 训练耗时",time.time()-start,"s")

# 4、使用测试集进行测试，模型评价

# 将模型训练得到的系数w与测试数据test\_data计算得到预测值。

# 计算测试集的MSE

MSE\_bgd = reg\_bgd.loss(X\_test, y\_test)

# 5、画出损失函数的变化趋势，画出预测值与真实值曲线

plt.figure()

plt.plot(np.arange(int(num\_epoches / 10),num\_epoches), losses\_bgd[int(num\_epoches / 10):])  # 只绘制后 90% 的训练轮数，都绘制的话趋势变化可视化效果不好

plt.xlabel("训练轮数")

plt.ylabel("损失函数值")

plt.title("BGD 损失函数变化趋势")

# plt.show()

# 测试集

plt.figure(figsize=(7,9))

plt.subplot(212)

plt.plot(y\_test,label='test\_true')

plt.plot(reg\_bgd.forward(X\_test),label='test\_predict')

plt.title("BGD Test Data MSE={:.2f}".format(MSE\_bgd))

# plt.show()

plt.legend() # 添加图例

# 训练集

MSE = reg\_bgd.loss(X\_train, y\_train)

plt.subplot(211)

plt.plot(y\_train,label='train\_true')

plt.plot(reg\_bgd.forward(X\_train),label='train\_predict')

plt.title("BGD Train Data MSE={:.2f}".format(MSE))

plt.legend() # 添加图例

plt.show()

运行代码得到结果如下图表

描述已自动生成 图表

描述已自动生成



**题目三：**

题目三中要求编程实现小批量梯度下降算法MBGD。

# 1、获取数据并划分数据集

X\_train, y\_train, X\_test, y\_test = load\_data(test\_rate=0.2)

# 2、数据标准化/归一化处理

# data\_processing(X\_train, X\_test)

X\_train, X\_test = data\_processing(X\_train, X\_test, 1)

reg\_mbgd = LinearRegression\_numpy(X\_train.shape[1])

MSE\_list = []

lr = 0.001        # 学习率，BGD 和 MBGD 的学习率都采用这个值

num\_epoches = 2000 # 训练轮数

fig, ax1 = plt.subplots()

# fig, ax2 = plt.subplots()

ax1.grid()

ax1.set\_xlabel('迭代次数')

ax1.set\_ylabel('损失函数值')

for batch\_size in [40,80,100,200]:

    # 当 batch\_size=len(X\_train)时，等价于BGD，当batch\_size=1时，等价于SGD

    num\_epoches\_mbgd = int(4\*batch\_size) # 确保对于不同的 batch\_size，训练的轮数相同，因为不同的 batch\_size,

                                            # 每轮的迭代次数等于样本总数/batch\_size，所以通过调整 num\_epoches 来保证每轮迭代次数相同

    MSE = 0

    start = time.time()

    losses\_mbgd = reg\_mbgd.train\_MBGD(X\_train, y\_train, num\_epoches=num\_epoches, batch\_size=batch\_size, eta=lr)

    y\_predict = reg\_mbgd.forward(X\_test)

    # plt.show()

    if batch\_size == 20:

        ax1.plot(losses\_mbgd,  linestyle='-', color='b', label='batch\_size=20 的损失曲线')

    elif batch\_size == 40:

        ax1.plot(losses\_mbgd,  linestyle='-', color='r', label='batch\_size={}的损失曲线'.format(40))

    elif batch\_size == 80:

        ax1.plot(losses\_mbgd,  linestyle='-', color='g', label='batch\_size={}的损失曲线'.format(80))

    elif batch\_size == 100:

        ax1.plot(losses\_mbgd,  linestyle='-', color='y', label='batch\_size={}的损失曲线'.format(100))

    elif batch\_size == 200:

        ax1.plot(losses\_mbgd,  linestyle='-', color='c', label='batch\_size={}的损失曲线'.format(200))

    ax1.legend(loc='upper left')

    losses\_mbgd=[]

    plt.figure()

    plt.figure(figsize=(7,9))

    plt.subplot(212)

    plt.plot(y\_test)

    plt.plot(reg\_bgd.forward(X\_test))

    MSE\_mbgd=reg\_mbgd.loss(X\_test, y\_test)

    plt.title("MBGD Test Data MSE={:.2f},when batch\_size is {:.4f}".format(MSE\_mbgd,batch\_size))

    # plt.show()

    # 训练集

    MSE = reg\_bgd.loss(X\_train, y\_train)

    plt.subplot(211)

    plt.plot(y\_train)

    plt.plot(reg\_bgd.forward(X\_train))

    plt.title("MBGD Train Data MSE={:.2f}".format(MSE))

    print("MBGD 训练耗时",(time.time()-start)/10.0,"s")

    print("num\_epoches={}, batch\_size={},lr={:.8f}, MSE={:.2f}".format(num\_epoches,batch\_size,lr,reg\_mbgd.loss(X\_test, y\_test)))

# 由图，较大的批量可以使模型对每一步的参数更新有更准确的估计，可能会使得模型在训练集上的表现更好

# 训练稳定性：较小的批量会导致训练过程中损失函数的震荡更加剧烈，而较大的批量可以使训练过程更加稳定

这样我们得到了如下的结果：文本

描述已自动生成

图表, 直方图

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成