**哈尔滨工程大学**

**《计算思维与问题求解（二）》大作业**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 班级： | 20230641 |  |
|  | 学号： | 2023064108 |  |
|  | 姓名： | 高明亮 |  |

实验时间： 2023 年 第12~13周

**哈尔滨工程大学计算机科学与技术学院**

实验题目：基于机器学习算法的数据分类

**一、编程语言及运行环境**

编程语言：C++

开发工具：Microsoft Visual Studio

软件环境：Windows 11

硬件环境：搭载Inter CPU的笔记本电脑

**二、问题描述及功能介绍**

实现基于多种机器学习算法的数据分类功能。下载一个二分类数据集，分别使用支持向量机、逻辑回归、k近邻、随机森林、BP神经网络实现二分类功能。将数据集划分为训练集和测试集两部分，对每种机器学习算法计算测试集上分类结果的accuracy、precision、recall、F1、AUC和ROC曲线等评价指标，使用图表展示五种方法的评价指标对比情况。实现五种机器学习分类算法。

**三、数据结构**

列出主要的数据结构。

结构体（Struct）：用于存储数据集每个样本，以及绘制ROC曲线时的点

向量（Vector）：存储特征向量，特征标签，预测标签，训练集以及验证集

二维向量（Vector of Vectors）：BP神经网络中的权重矩阵

字符串 （string）：用于存储文件名和读取的行数据。

输入输出流 （ifstream, ofstream）：用于读取CSV文件和写入ROC曲线和AUC值到文件。

**四、设计思想**

该实验题目主要通过编写五种机器学习算法，对一组可二分类的数据集（本实验采用Raisin葡萄干数据集）进行对应的分类操作，并对各自算法的性能指标进行评估，以图表的形式横向对比算法之间的性能优劣。在每个算法中都存在共通或功能类似模块及函数功能，包括对于原始数据集从文件中读取；随机打乱数据集（可选功能）；对数据集按照一定比例分割出训练集和验证集并显示读取和划分的样本数量，供后续机器学习模型的训练；输出当前实现算法的分类结果以及每个样本的判断正误情况；甚至是在模型训练完成并对验证集进行分类验证后计算各类相关评估指标并保存到文件供后续以图表方式直观对比结果。除此之外，还应该编写一个程序，实现功能从这五个算法分类后保存的文件里读取到分类的结果，通过调用在C++语言上可以搭载调用的图形库，如”“EasyX”,将抽象难懂的数据转化为生动形象图表语言并显示在屏幕上，以供对分类结果进行参考。

一、在BP神经网络算法中，我们主要需要实现的功能模块有：1.数据读取和处理 2.数据归一化 3.神经网络的构建与初始化 4.训练和评估。

数据读取和处理部分，读取csv格式的原始数据集中的数据，并将特征标签存储在一个“data”向量中，两种标签使用one-hot 编码，Kecimen 对应 {1, 0}，Besni 对应 {0, 1}。

通过数据归一化操作，通过遍历所有数据点，找出每个特征的最小值和最大值并分别存储每个特征的最小值和最大值在minVals 和 maxVals，将每个特征的值按公式 (x - min) / (max - min) 进行归一化，得到一个范围在 [0, 1] 之间的新值。以确保不同特征的量级相同，提高模型训练的稳定性和效率。

其次是神经网络的构建与初始化。首先需要通过创建神经网络类来定义网络结构和参数：

(1)输入层、隐藏层、输出层的大小：

输入层大小 (inputSize)：输入特征的数量，在这个算法中是7（因为Raisin数据集每个样本有7个特征）。隐藏层大小 (hiddenSize)：隐藏层神经元的数量，在这里设定为10。输出层大小 (outputSize)：输出神经元的数量，在这里是2（因为有两个类别，Kecimen和Besni）。

(2)各层之间的权重和偏置：

隐藏层激活值(hiddenLayer)：大小为hiddenSize。输出层激活值 (outputLayer)：大小为outputSize。输入层到隐藏层的权重 (weightsInputHidden)：大小为inputSize x hiddenSize的二维向量。隐藏层到输出层的权重 (weightsHiddenOutput)：大小为hiddenSize x outputSize的二维向量。隐藏层的偏置 (biasHidden)：大小为hiddenSize。输出层的偏置 (biasOutput)：大小为outputSize。

(3)学习率 (learningRate)：用于更新权重时的步长，提高训练精度。

(4) 随机化权重和偏置的初值：使用rand()函数生成一个范围在 [-0.5, 0.5] 的随机数作为初始权重和偏置。确保它们是随机的，从而打破对称性，促进提高网络模型的训练结果。

前向传播和反向传播是训练中的两个核心步骤。前向传播用于计算神经网络的输出，每个隐藏层神经元的输入是所有输入特征的加权和加上偏置，每个输出层神经元的输入是所有隐藏层神经元的加权和加上偏置，然后通过激活函数（sigmoid 函数）得到输出。而反向传播用于根据损失函数调整权重和偏置，输出层误差是输出值和目标值之差乘以 sigmoid 函数的导数隐藏层误差是输出层误差的加权和乘以 sigmoid 函数的导数

随后分别使用学习率、输入和隐藏层误差来调整输入层到隐藏层的权重和偏置；使用学习率、隐藏层输出和输出层误差来调整隐藏层到输出层的权重和偏置。

最后就是开始训练神经网络模型，训练神经网络是一个迭代过程，通过反复地调整网络参数（权重和偏置）来最小化损失函数，模型能够逐渐学到输入数据和输出标签之间的映射关系。随后在分类完成后计算各项评估指标。

二、KNN，K近邻分类算法是一种基于实例的学习方法，通过计算待分类的样本和已知样本的距离，选取距离最近的k个样本，用投票的方式确定待分类样本的类别。程序主要包含的模块有：数据加载、数据预处理、距离计算、邻居选择、投票机制、分类得分计算，以及最终的性能评估

与其余算法类似，KNN算法也包含对于数据的读取和预处理功能，实现的功能也基本一致，不过多赘述。

在实现数据集的预处理和划分过后，KNN算法的核心在于计算两个特征之间的欧几里得距离distance=，其中a和b是两个向量，n 是向量的维度。该距离度量用于衡量两个样本在特征空间中的相似度。对于验证集样本，计算它与训练集中所有样本的距离，并将算出的所有距离进行排序，按照升序顺序从小到大排列，并选取距离最小的前K个样本，这也是该算法被称为KNN的原因，统计该K个样本中各个类别出现的次数。根据这K个邻居样本中各个类别的比例占比，进行投票决定将当前验证样本划分认定为某一种类，从而实现分类。

在分类过后我们需要对该模型分类的结果进行评估，对于测试集中每个样本，计算其分类得分（属于“Kecimen”类的比例），根据得分确定预测标签。定义并初始化用于存储分类结果的变量，包括TP（真正类）、TN（真负类）、FP（假正类）、FN（假负类），再通过这几个计数器计算出整个程序算法的评估指标accuracy：分类正确的样本占总样本的比例，precision：被预测为正类的样本中实际为正类的比例，recall：实际为正类的样本中被正确预测为正类的比例，F1 Score：精确率和召回率的调和平均数。与其他算法不同，KNN中不存在阈值这一概念，所以在计算TPR和FPR时，我们采用分类的得分从高到低作为分隔进行ROC曲线上点的描绘。

三、Logistic Regression逻辑回归算法同样也是一种高效且适用于二分类问题的统计方法，逻辑回归的目标是根据输入特征预测一个二分类目标变量的概率。它假设输入特征与目标变量之间存在线性关系，然后通过一个非线性函数（Sigmoid函数）将线性组合的结果映射到0到1之间。

同其余算法程序，实现对数据的读取和分割数据集操作没有差异，我们着重关注实现的算法原理。在最开始时使用随机数生成器对每个特征生成初始化权重，并存储在向量“weights”中，选用合适的激活函数（Sigmoid函数）将线性组合的结果映射到0和1之间，作为预测的概率。在训练过程中，使用梯度下降法迭代更新权重，在外层迭代中进行一定量次数的迭代以保证模型逐步收敛，而在每次迭代中的内层迭代里，通过遍历每一个样本，计算线性组合（权重和特征的加权和），其次借助Sigmoid函数计算得到预测概率，将预测概率与真是的类别标签做差值，得到预测误差，根据对应误差的大小对当前1特征的权重进行修正更新，随后进入下一次迭代，直至每个特征的权重最后都会被修正到相应合适的值。经理上述步骤后，权重向量 “weights” 被不断更新，使得模型的预测逐步逼近真实值。整个过程利用梯度下降法，通过最小化对数损失函数来优化模型参数。

而在迭代更新权重的过程中，有两个参数会较为强烈地影响到最后的结果，一是学习率（Learning Rate）：这是控制每次更新权重的步长，也是决定后续评估指标精度的重要因素，学习率过大可能导致结果不收敛，拟合效果不理想；过小则会导致程序运行时间过长，收敛速度慢。其二是迭代次数，这保证了算法整体的收敛程度和模型精度。结合以上内容在训练结束后，我们可以对预测结果进行相应的指标评估。

四、Random forest随机森林是一种集成学习方法，通过构建多个决策树并将他们的预测结果进行集成来提高模型的准确性和稳定程度，属于监督学习算法之一，比较适合处理当前我们实验的二分类问题。同上文，我们依然需要对数据集进行读取和划分，这属于预处理部分，是算法训练的基础。其次是对模型的训练，因为在这个程序中，我调用了OpenCV的库内算法实现的随机森林，所以我们在这里重点详解随机森林的实现原理。

随机森林算法的原理主要包括两个方面：随机性和集成。随机性体现在样本的随机性和特征的随机性，通过引入随机性来构建多个决策树，从原始训练数据集中随机选择若干个样本（有放回抽样），每个决策树都是基于随机抽样的训练数据和随机选择的特征进行构建的。这种随机性能够有效地减少过拟合的风险，提高模型的泛化能力。而集成是指通过结合多个决策树的预测结果来完成最终的分类或回归任务，通常采用投票的方式进行集成。在随机森林中，每棵决策树的构建过程都是相互独立的，这意味着每棵决策树都是在不同的训练数据和特征子集上进行构建的，这种随机性能够有效地降低模型的方差，提高模型的稳定性。OpenCV库提供了cv::ml::RTrees类来实现随机森林算法，我们需要设置好算法中的一些参数，随后便可以进行模型训练，如setMaxDepth() - 设置每颗决策树的最大深度；setMinSampleCount() - 设置每个叶节点所需要的最小样本数；setMaxCategories() - 设置最大类别数，也就是分成的种类个数，在二分类问题中设置为2即可；setActiveVarCount() - 设置每个节点分割时随机选择的特征数量，这个参数控制了随机森林的随机性，减少过拟合现象；setTermCriteria(TermCriteria(TermCriteria::MAX\_ITER, 1000, 0.001)) – 以及设置迭代次数和学习率，与在逻辑回归算法中意义相同。

参数设置完毕后，我们可以使用OpenCV提供的train来对模型进行训练和预测结果输出了，随后依旧是对于各项指标进行评估检验。

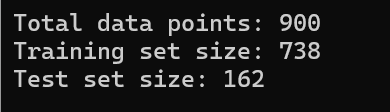
五、SVM支持向量机，同样作为一种用于分类和回归分析的监督学习模型，SVM的核心思想在于寻找到一个超平面来最大化分类的间隔，从而将数据进行分类。在我们这个实验里，因为处理的是一个线性可分问题，而不是非线性问题，所以我们并不需要使用核函数来把数据映射到更高维度的空间，SVM会直接寻找到一个超平面以分割数据。

与一样需要特征权重向量的Logistic逻辑回归算法不同，在设置初始权重时，我们需要将权重向量“weights”设为零向量，其长度即为特征的维度数；除同样需要的迭代次数和学习率以不断进行权重的修正和更新以外，在不断地循环和遍历数据集样本的过程中，还需要计算条件“condition”，即为标签与权重和特征的点点积的乘积。如果“condition”小于1，说明该数据点在当前超平面下分类错误或在边界内，需要进行更新权重以减少错误，借助设置惩罚参数lambda，以惩罚大的权重，防止过拟合现象出现；反之，如果“condition”大于等于1，表式该样本数据点在当前超平面下分类正确且在边界外，只需要进行正则化更新。在进行完毕所有轮次的迭代后，找到最大化类别间距并最小化分类错误的最佳权重，会被返回到特征向量“weights”内，用与后续的样本预测和指标评估。

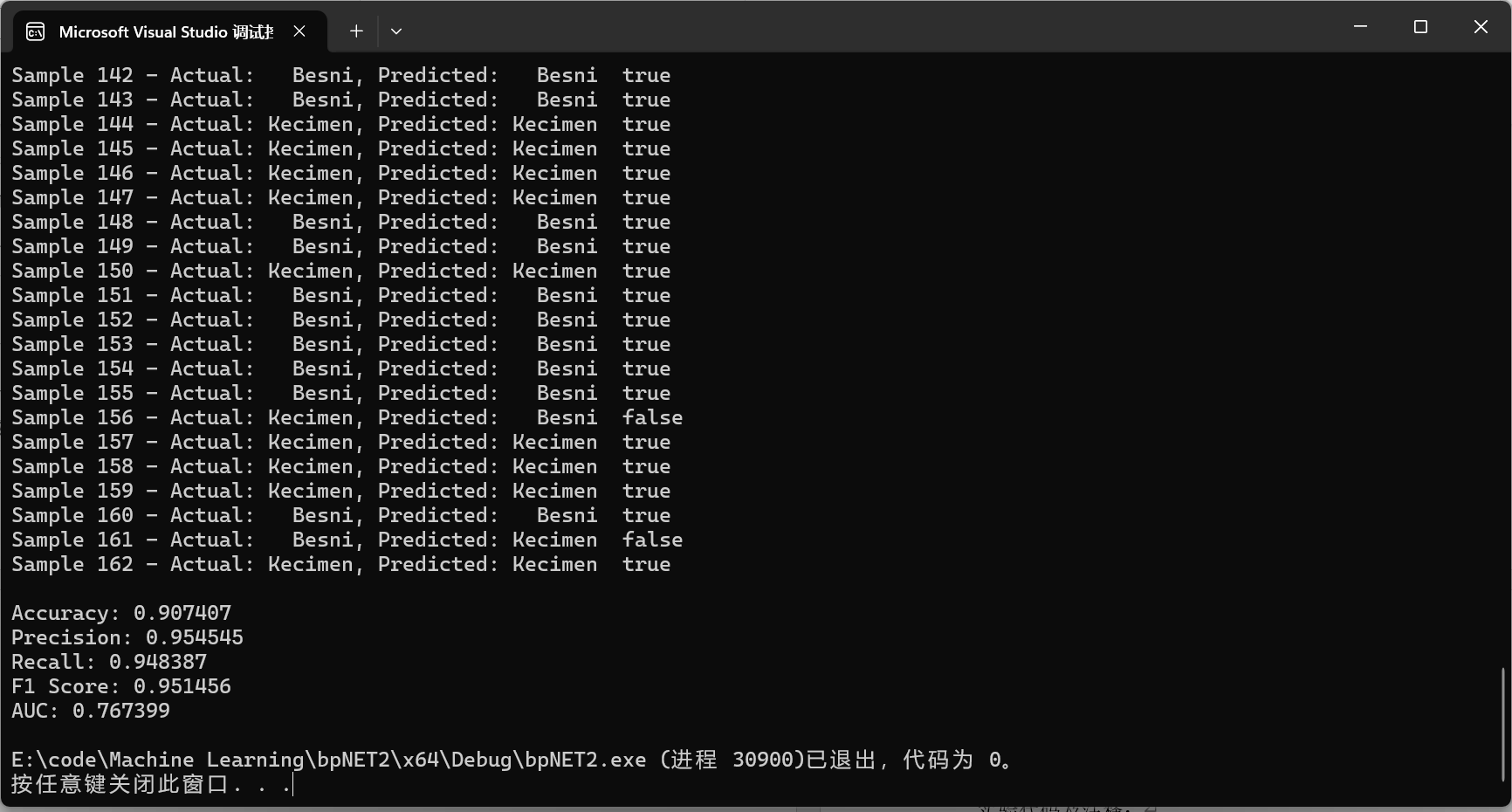
六、绘图程序，在绘图模块中实现的功能相比以上部分较为纯粹且简单，其主要实现的功能就是将上述五种算法的评估结果进行读取并调用EasyX图形库将数据实现可视化，如绘制ROC曲线的对比，既可以绘制在同一坐标轴下直观对比，也可以分开绘制单独观察某一种算法的情况。以及横向对比Accuracy，Precision，Recall，F1 Score数据绘制柱状图表，孰优孰劣一眼便知。具体实现可以在下面的实验结果与分析中具体查看。

**五、实验结果与分析**

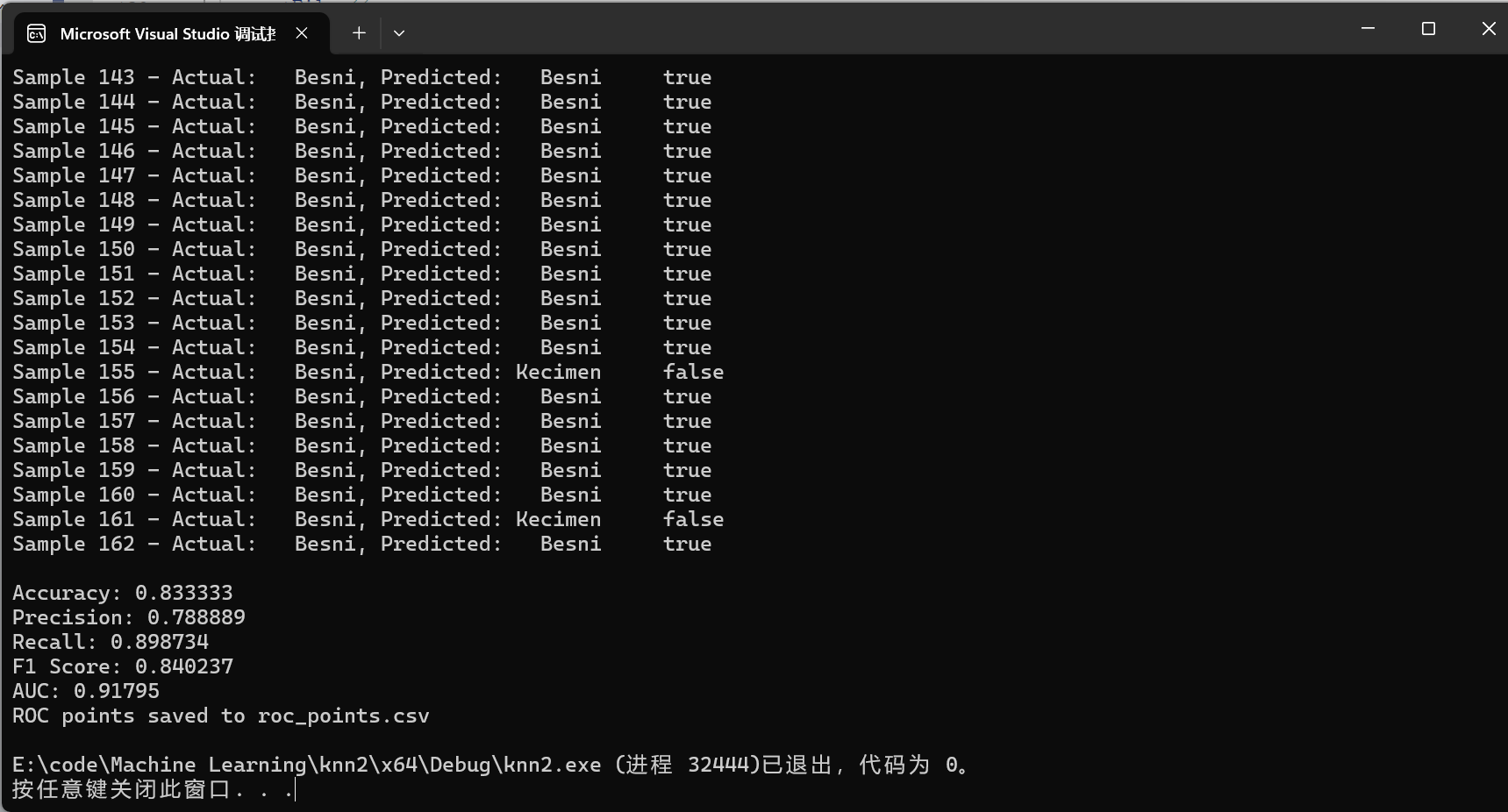
从csv格式的原始数据集中读取数据，并按照指定的比例对数据集样本进行划分并进行显示。在预测完成后输出每个样本的预测情况和指标计算结果。



1. BP神经网络



1. KNN

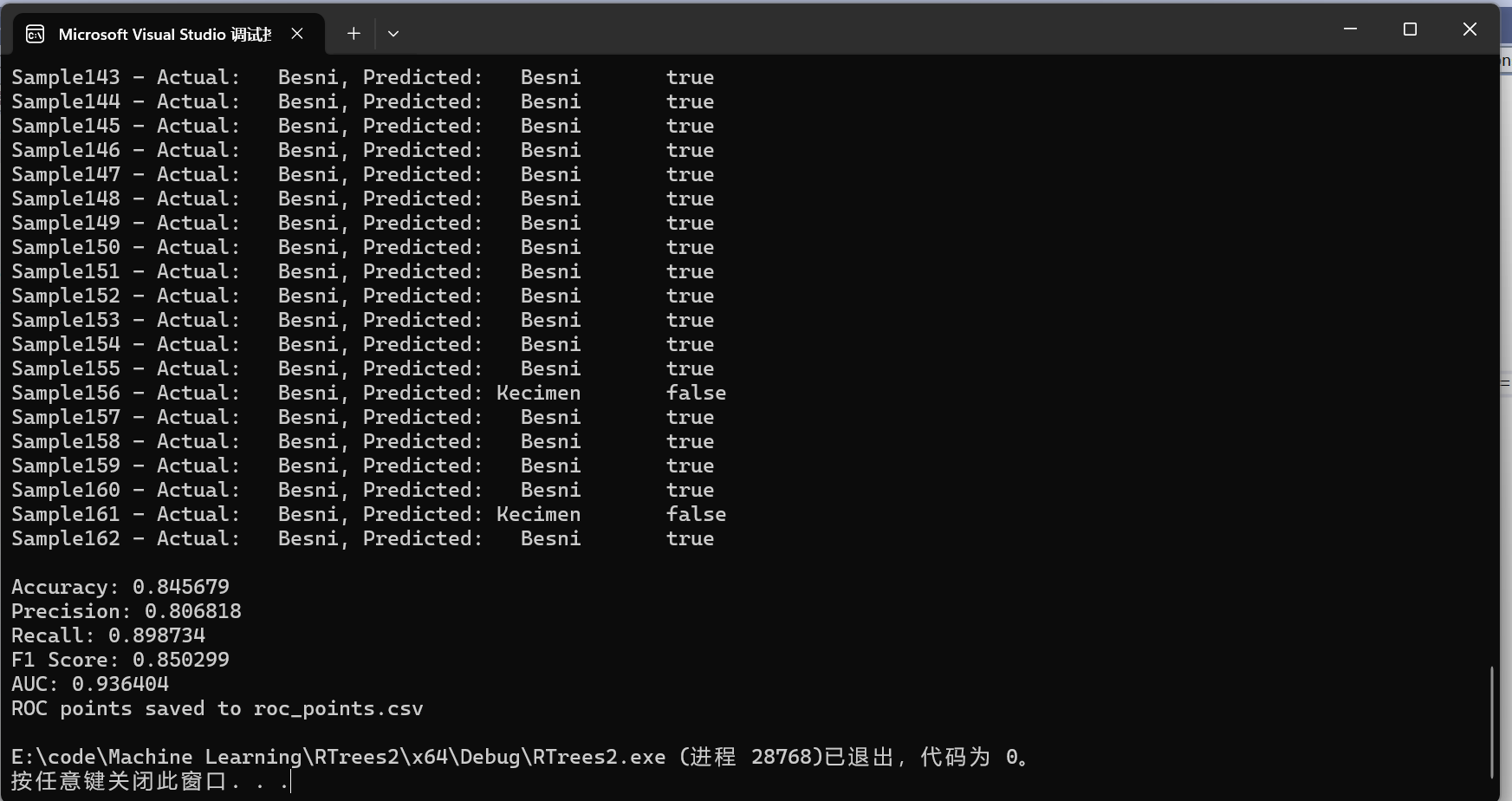


1. 逻辑回归Logic

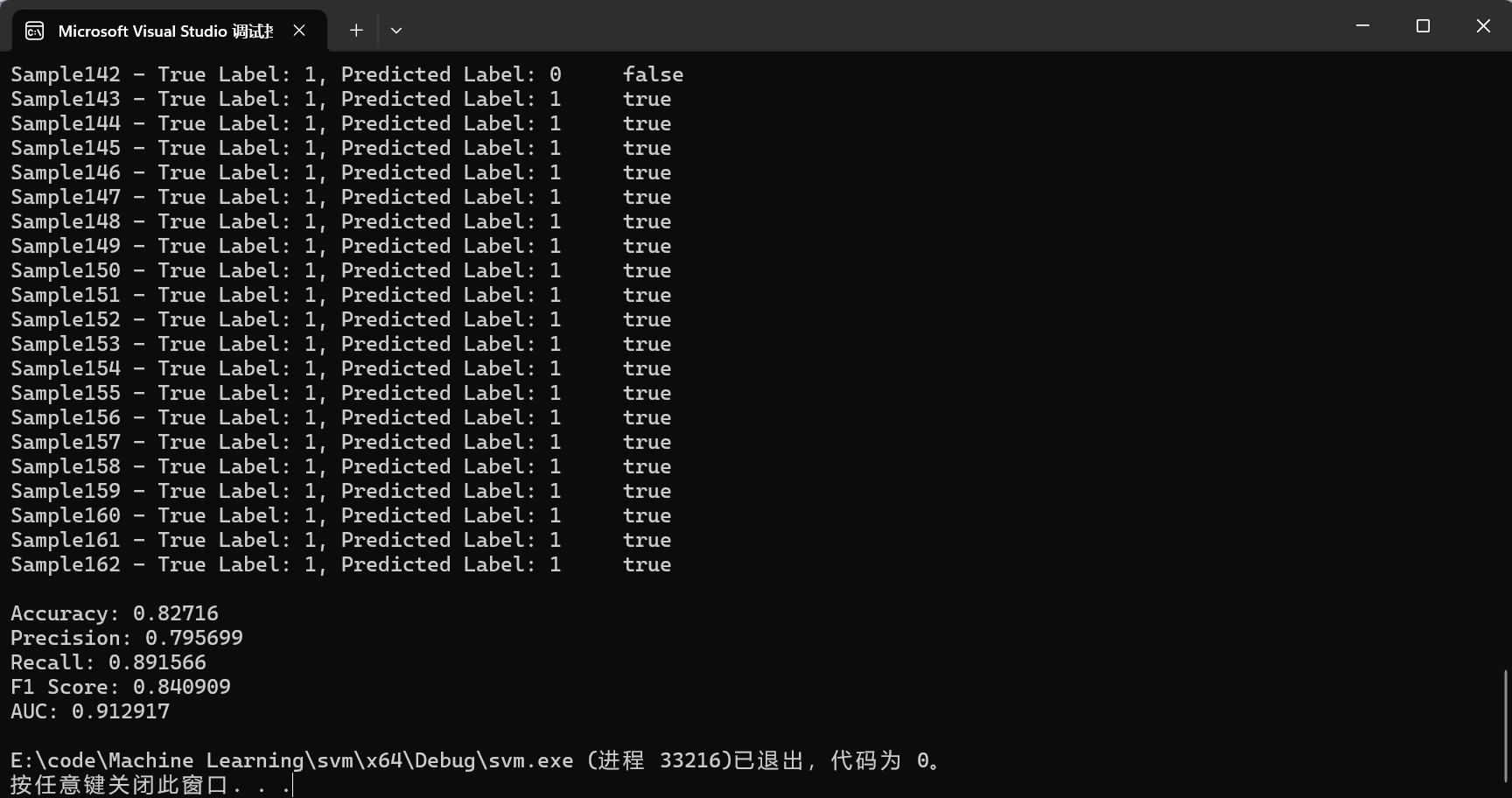
文本

描述已自动生成

1. 随机森林

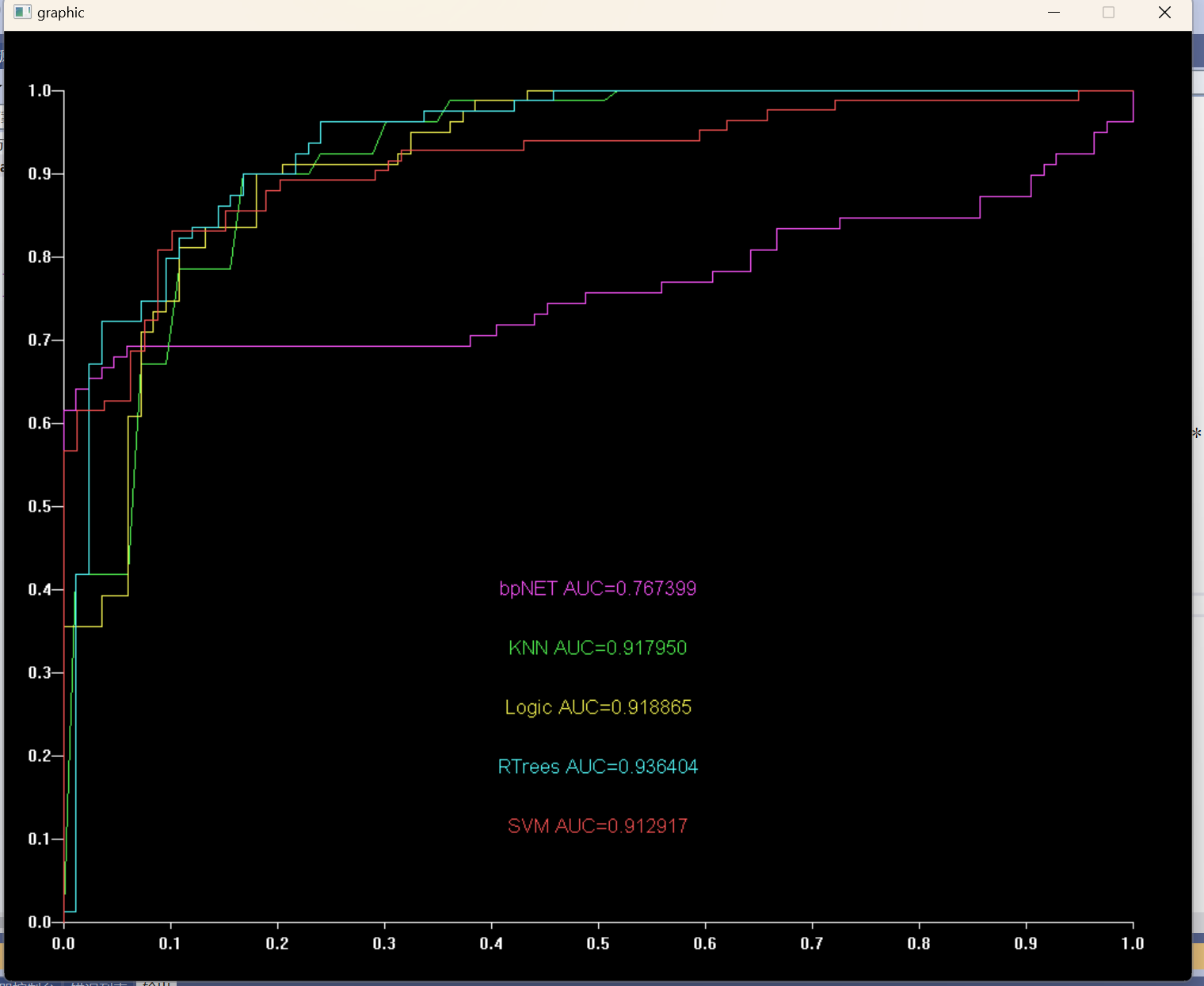


1. 支持向量机SVM、



6.各算法ROC曲线对比图

图片包含 图形用户界面

描述已自动生成

7.各算法评估指标柱状对比图

图表, 条形图

描述已自动生成

从上述的各项截图中，我们可以清晰地看到每个算法最后的预测分类结果情况，除此之外我们也很容易发现大部分算法最后拟合出来的效果都比较符合预期，各指标均能保持在80%及以上，最好的甚至可以高达90%或更高，这说明我们对于模型的参数设置，数据处理以及训练都是有效且拟合程度较高的。再结合其余多项指标，从柱状图能显著看出BP神经网络无论在哪方面都明显优于其他算法，且非常逼近于1，但反观其ROC曲线又会发现其拟合效果又略低于其他算法，这可能是因为产生了过拟合现象或是选择的阈值不合适，我在多次调整参数后也没有得到较为明显的改善效果，等待后续继续寻找可能存在的问题并进行调优。

KNN,随机森林以及SVM算法的各项评估数据都非常接近，不管是从准确率、精确率、F1分数、召回率亦或是ROC曲线的趋势和AUC值，很难找出他们之间明显的性能差异，总体都能保证在80%+的准确分类，说明模型的拟合效果大致能分类出不同类别的数据，但仍存在一定的改进和优化空间，这可能也与数据集或是算法的编程实现的关系导致的。例如，我们使用的数据集是Raisin葡萄干数据集属于现实里的情况，无法避免会出现偶然特例从而影响模型的决策能力，当然数据集的样本数量也可能限制了模型的能力，我们今后可以尝试使用样本数量更多，特征更丰富，分类更明确的其他数据集来对模型进行训练和验证，这应该会对其提高分类结果的准确率有很大的帮助。

最后是逻辑回归算法，从柱状图中可以明显看到其精确率很高但在召回率上只有55%左右，这两个指标地显著差异说明了该算法在将一个样本预测为正的时候，其真实标签也为正的概率很大；而在在所有真实标签为正的样本中，将样本浴厕为正的占比不高。总结来说就是模型的决策非常谨慎保守，在找全所有真实标签为正的能力较为欠缺，但一旦被他预测是正的样本，其真实标签为正的概率极高。F1分数便是反应这两个指标的综合能力，他是他们的调和平均数，反应算法查的又准又全的能力，而召回率的过低数值最终使得其数值最高的精确率也没能得到较好的F1分数，反而成为了五个算法中得分最低的一个。究其原因可能是因为训练样本不够全面，模型还没有彻底掌握分辨特征的能力，导致决策保守，我们后续可以通过修改损失函数，增加对正类样本的惩罚，鼓励模型提高召回率，积极做出正类预测，或者是处理更复杂且分明的样本来提高决策能力。

综上，我们大致分析了各个模型的预测结果以及其指标所展现出的问题，通过对可能的问题进行分析和提出相应的解决方案，在以后我们可以持续的优化模型算法，直至达到符合我们理论预期的结果。

**六、总结与心得体会**

在当初刚选择这个实验的时候，我其实是抱着选择一个相对容易完成的任务的理由，但在真正去理解题目所要让我实现的功能时，我又感觉到有些无从下手，因为毕竟机器学习对我来说是一个完全陌生的领域，之前只是听说过AI之类的相关内容，对原理不甚了解。于是当我在网上寻找相关资料时，发现大部分的文章都是去在大量叙述算法的实现原理以及大量复杂数学公式的推导，起初看的头晕眼花，后来找了一些用简单易懂的语言只着重算法思想的文章才对这五种算法有了大致的理解和构想。随后在程序的编写中，也遇到了许多困难走了不少弯路，比如在最初想实现SVM算法时需要调用一个现有库，库的下载编译甚至配置让我感到几近崩溃，最终也没能实现那个库部署。最后在和老师以及助教们的交流并结合一些资料，决定不用库函数去尝试编写程序，在性能方面可能不及现有算法但也应该能大致实现效果，于是开启了一个好头。俗话说万事开头难，后面的进展比较顺利，每个算法在各个方面多少都存在许多的共通之处，比如在对数据集的读取处理划分几乎不需要做多少改动。而在有的算法中还存在着相似的让模型收敛的办法，如梯度下降不断修正权重等等实现，都让我的进度有了不小的进展。当然中间也有许多小插曲，数据集的更换，参数的调整，也都曾困扰过我，好在最后算是找到了相对适合的数据集和能让结果运行出漂亮的数据的参数设置。到这里实验的大致实现就完成一大部分了，但是仍需要将我们的结果数据分析对比然后可视化处理，数据的分析也不尽相同，只需要了解这个指标代表的含义，理解需要记录的一些数据就可以很快的去计算出结果。可视化也相对顺利，得益于上学期计算思维课的大作业程序设计是做一个有图形界面的游戏，我自然而然的继续沿用了画图的思路，借助EasyX图形库将我们的结果具象的显示到图表中。

纵观整个过程，跌跌撞撞但也算是在最后尽力完成了任务，这当中不仅提高了C++语言的编程能力，同时也对我搜寻有用的数据和资料并将其理解和转化为自己可用的材料的能力提出了重大考验。而我在这个过程中也算是对机器学习的几种常见模型算法有了初步的认识，无论是支持向量机（SVM）、逻辑回归、K近邻（KNN）、随机森林，还是BP神经网络，每种算法都有其独特的实现方法，在实现不同功能的能力各有千秋。我所实现的只是一个最简单的功能，对一组二分类数据集实现二分类，真正的机器学习所处理的问题远比这复杂的多，所处理的数据量也是难以想象的庞大和复杂。借此契机，提前体验一下自己今后在学习生活中可能遇到的各种困难和问题吧，锻炼自己解决问题和抗压能力，即使一路过来并不那么一帆风顺，但卒有所获，不愧对于自己。

**七、附录**

代码行数：

实验代码及注释：

|  |
| --- |
| // 加载数据集函数  vector<Raisin> loadDataSet(const string& fileName) {  ifstream file(fileName);  string line;  vector<Raisin> dataSet;  if (!file.is\_open()) {  cerr << "Error: Unable to open file " << fileName << endl;  return dataSet;  }  while (getline(file, line)) {  stringstream ss(line);  string item;  Raisin raisin;  while (getline(ss, item, ',')) {  try {  raisin.features.push\_back(stof(item)); // 将特征添加到向量中  }  catch (const invalid\_argument&) {  raisin.label = item; // 将标签保留为字符串  }  }  dataSet.push\_back(raisin);  }  return dataSet;  }  // 划分数据集函数  void splitDataSet(const vector<Raisin>& dataSet, vector<Raisin>& trainSet, vector<Raisin>& testSet, float trainRatio = 0.82) {  vector<Raisin> shuffled = dataSet;  //random\_device rd;  //mt19937 g(rd());  //shuffle(shuffled.begin(), shuffled.end(), g);  size\_t trainSize = static\_cast<size\_t>(trainRatio \* shuffled.size());  trainSet.assign(shuffled.begin(), shuffled.begin() + trainSize); // 分配训练集  testSet.assign(shuffled.begin() + trainSize, shuffled.end()); // 分配测试集  }  // 评估模型函数  vector<double> evaluateModel(LogisticRegression& model, vector<vector<double>>& X\_test, vector<int>& y\_test) {  int true\_positive = 0, true\_negative = 0, false\_positive = 0, false\_negative = 0;  vector<double> y\_probs;  for (size\_t i = 0; i < X\_test.size(); i++) {  int prediction = model.predict(X\_test[i]);  double proba = model.predict\_proba(X\_test[i]);  y\_probs.push\_back(proba);  // 显示每个样本的分类结果  cout << "Sample " << i + 1 << " - Actual: " << y\_test[i] << ", Predicted: " << prediction << ", Probability: " << proba << endl;  if (prediction == 1 && y\_test[i] == 1) true\_positive++;  if (prediction == 0 && y\_test[i] == 0) true\_negative++;  if (prediction == 1 && y\_test[i] == 0) false\_positive++;  if (prediction == 0 && y\_test[i] == 1) false\_negative++;  }  double accuracy = (true\_positive + true\_negative) / static\_cast<double>(X\_test.size());  double precision = true\_positive / static\_cast<double>(true\_positive + false\_positive);  double recall = true\_positive / static\_cast<double>(true\_positive + false\_negative);  double f1 = 2 \* (precision \* recall) / (precision + recall);  cout << endl;  cout << "Accuracy: " << accuracy << endl;  cout << "Precision: " << precision << endl;  cout << "Recall: " << recall << endl;  cout << "F1 Score: " << f1 << endl;  vector<pair<double, double>> roc\_points = calculateROC(y\_probs, y\_test);  // 计算 AUC  double auc = calculateAUC(roc\_points);  cout << "AUC: " << auc << endl;  cout << endl;  //保存AUC的值  ofstream outFile;  outFile.open("AUC.txt");  outFile << auc;  outFile.close();  //保存其余指标  ofstream outFile2;  outFile2.open("RESULT.txt");  outFile2 << accuracy << " " << precision << " " << recall << " " << f1;  outFile2.close();  return y\_probs;  }  // 保存 ROC 曲线上的点到 CSV 文件  void saveROCCurvePoints(const string& filename, const vector<pair<double, double>>& roc\_points) {  ofstream file(filename);  if (!file.is\_open()) {  cerr << "Failed to create file: " << filename << endl;  return;  }  file << "FPR,TPR" << endl;  for (const auto& point : roc\_points) {  file << point.first << "," << point.second << endl;  }  file.close();  cout << "ROC curve points saved to file: " << filename << endl;  }  //BP神经网络核心部分  // 前向传播  vector<double> forward(const vector<double>& input) {  // 隐藏层  for (int j = 0; j < hiddenSize; ++j) {  hiddenLayer[j] = 0.0;  for (int i = 0; i < inputSize; ++i)  hiddenLayer[j] += input[i] \* weightsInputHidden[i][j];  hiddenLayer[j] += biasHidden[j];  hiddenLayer[j] = sigmoid(hiddenLayer[j]);  }  // 输出层  for (int k = 0; k < outputSize; ++k) {  outputLayer[k] = 0.0;  for (int j = 0; j < hiddenSize; ++j)  outputLayer[k] += hiddenLayer[j] \* weightsHiddenOutput[j][k];  outputLayer[k] += biasOutput[k];  outputLayer[k] = sigmoid(outputLayer[k]);  }  return outputLayer;  }  // 反向传播  void backward(const vector<double>& input, const vector<int>& target) {  // 计算输出层误差  vector<double> outputError(outputSize);  for (int k = 0; k < outputSize; ++k) {  outputError[k] = (outputLayer[k] - target[k]) \* sigmoidDerivative(outputLayer[k]);  }  // 计算隐藏层误差  vector<double> hiddenError(hiddenSize);  for (int j = 0; j < hiddenSize; ++j) {  hiddenError[j] = 0.0;  for (int k = 0; k < outputSize; ++k)  hiddenError[j] += outputError[k] \* weightsHiddenOutput[j][k];  hiddenError[j] \*= sigmoidDerivative(hiddenLayer[j]);  }  // 更新隐藏层到输出层的权重和偏置  for (int j = 0; j < hiddenSize; ++j) {  for (int k = 0; k < outputSize; ++k) {  weightsHiddenOutput[j][k] -= learningRate \* outputError[k] \* hiddenLayer[j];  }  }  for (int k = 0; k < outputSize; ++k) {  biasOutput[k] -= learningRate \* outputError[k];  }  // 更新输入层到隐藏层的权重和偏置  for (int i = 0; i < inputSize; ++i) {  for (int j = 0; j < hiddenSize; ++j) {  weightsInputHidden[i][j] -= learningRate \* hiddenError[j] \* input[i];  }  }  for (int j = 0; j < hiddenSize; ++j) {  biasHidden[j] -= learningRate \* hiddenError[j];  }  }  // Sigmoid激活函数  double sigmoid(double x) {  return 1.0 / (1.0 + exp(-x));  }  // Sigmoid函数的导数  double sigmoidDerivative(double x) {  return x \* (1.0 - x);  }  };  //KNN核心部分  // 计算欧几里得距离  float euclideanDistance(const vector<float>& a, const vector<float>& b) {  float sum = 0.0;  for (size\_t i = 0; i < a.size(); ++i) {  sum += pow(a[i] - b[i], 2);  }  return sqrt(sum);  }  float knnClassifyScore(const vector<Raisin>& trainSet, const vector<float>& testInstance, int k) {  vector<pair<float, string>> distances;  for (const auto& instance : trainSet) {  float distance = euclideanDistance(instance.features, testInstance);  distances.push\_back(make\_pair(distance, instance.label));  }  sort(distances.begin(), distances.end());  map<string, int> labelCounts;  for (int i = 0; i < k; ++i) {  labelCounts[distances[i].second]++;  }  return static\_cast<float>(labelCounts["Kecimen"]) / k; // 返回属于"Kecimen"类的比例  }  //逻辑回归核心部分  // 训练模型  void train(vector<vector<double>>& X, vector<int>& y, double learning\_rate, int num\_iterations) {  int m = X.size(); // 样本数量  int n = weights.size(); // 特征数量  for (int iter = 0; iter < num\_iterations; iter++) {  for (int i = 0; i < m; i++) {  double z = 0.0;  for (int j = 0; j < n; j++) {  z += weights[j] \* X[i][j]; // 计算线性组合  }  double h = sigmoid(z); // 应用 Sigmoid 函数  double error = h - y[i]; // 计算误差  for (int j = 0; j < n; j++) {  weights[j] -= (learning\_rate \* error \* X[i][j]) / m; // 更新权重  }  }  }  }  // 预测概率  double predict\_proba(vector<double>& X) {  double z = 0.0;  for (size\_t i = 0; i < X.size(); i++) {  z += weights[i] \* X[i]; // 计算线性组合  }  return sigmoid(z); // 应用 Sigmoid 函数  }  // 预测  int predict(vector<double>& X) {  return (predict\_proba(X) >= 0.5) ? 1 : 0; // 阈值为0.5  }  };  // 训练随机森林模型函数  void trainRandomForest(const vector<Raisin>& trainSet, Ptr<RTrees>& model) {  Mat trainData(trainSet.size(), trainSet[0].features.size(), CV\_32F);  Mat trainLabels(trainSet.size(), 1, CV\_32S);  for (size\_t i = 0; i < trainSet.size(); ++i) {  for (size\_t j = 0; j < trainSet[i].features.size(); ++j) {  trainData.at<float>(i, j) = trainSet[i].features[j]; // 填充训练数据  }  trainLabels.at<int>(i, 0) = (trainSet[i].label == "Kecimen") ? 1 : 0; // 将标签转换为整数  }  model = RTrees::create();  model->setMaxDepth(10); // 设置最大深度  model->setMinSampleCount(2); // 设置最小样本数  model->setRegressionAccuracy(0);  model->setUseSurrogates(false);  model->setMaxCategories(2);  model->setPriors(Mat());  model->setCalculateVarImportance(true);  model->setActiveVarCount(4);  model->setTermCriteria(TermCriteria(TermCriteria::MAX\_ITER, 1000, 0.001));  model->train(trainData, ROW\_SAMPLE, trainLabels); // 训练模型  }  //SVM核心部分  // SVM训练  vector<double> trainSVM(const vector<DataPoint>& trainSet, double learningRate, double lambda, size\_t epochs) {  if (trainSet.empty()) {  cerr << "Training set is empty!" << endl;  return vector<double>();  }  size\_t featureSize = trainSet[0].features.size();  vector<double> weights(featureSize, 0.0);  for (size\_t epoch = 0; epoch < epochs; ++epoch) {  for (const auto& point : trainSet) {  double y = point.label == 1 ? 1 : -1;  double condition = y \* dotProduct(weights, point.features);  if (condition < 1) {  for (size\_t i = 0; i < featureSize; ++i) {  weights[i] += learningRate \* ((y \* point.features[i]) - (2 \* lambda \* weights[i]));  }  }  else {  for (size\_t i = 0; i < featureSize; ++i) {  weights[i] += learningRate \* (-2 \* lambda \* weights[i]);  }  }  }  }  return weights;  }  // SVM预测（返回决策值）  double predictDecision(const vector<double>& weights, const vector<double>& features) {  return dotProduct(weights, features);  }  // SVM预测（返回标签）  int predict(const vector<double>& weights, const vector<double>& features) {  double result = predictDecision(weights, features);  return result >= 0 ? 1 : 0;  } |