风险交易识别

一、实验名称：风险交易识别

二、实验目的

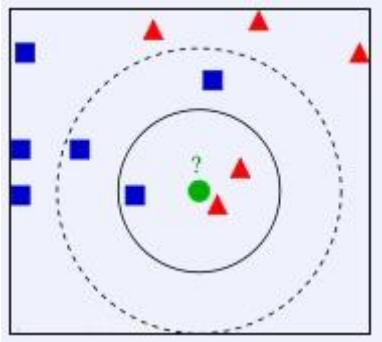
对 30 维过程的交易进行风险识别。

三、实验原理

本次实验中应用了多种模型，以下按顺序说明它们的原理。另外，本题目中要求以 f1 作为判别模型效果的标准，这里也对其原理进行说明。

1、K 近邻

在特征空间中，如果一个样本附近的 k 个最近样本的大多数都属于某一类别，则该样本也属于这一类别。



在寻找最近的 k 个样本时，使用欧式距离进行度量：

$$dis_{ij} = \sqrt{\sum_{n=1}^{30} (x_{i_n} - x_{j_n})^2}$$

其中 i 表示第 i 个测试点，j 表示第 j 个样本点，n 表示第 n 个维度

Dis 中最小的 k 个值对应的下标，它们对应的样本点的标签的众数（这里忽略多个众数的情况）即是所求的测试点的标签。

2、 感知机

感知机寻找一组(w,b)作为系数和常数项来实现分类超平面，也可以看作是对(x1 …… x30 1）的增广维度寻找系数组。

感知机模型的损失函数为：

$$-\sum_{i \in M} (w^T x_i + b) \cdot y_i$$

其中 M 是误分类点集合

在迭代中，利用梯度下降法最小化损失函数，即在数据点误分类时，利用损失函数对参数的偏导和迭代步长对参数进行变更：

$$w = w + cxy$$

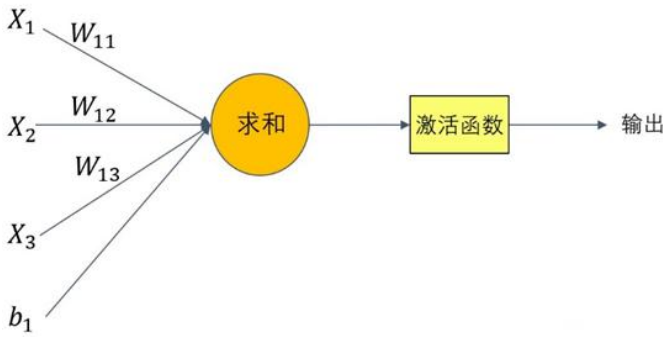
$$b = b + cy$$

其中 c 是迭代步长

最后得到损失函数最小的（w,b），便能指导绘制分类超平面，进而得到分类的结果。

3、 全连接神经网络

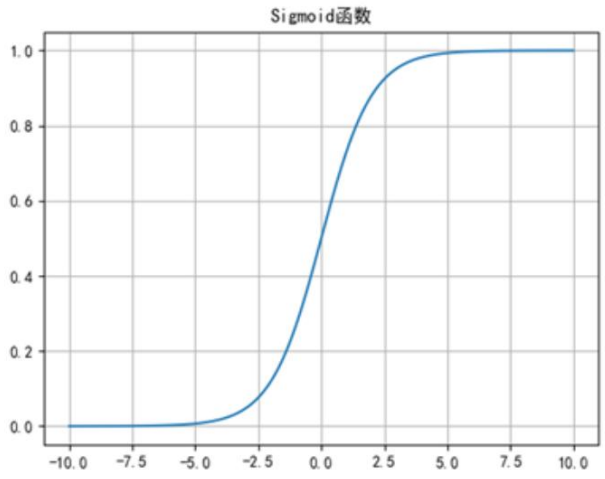
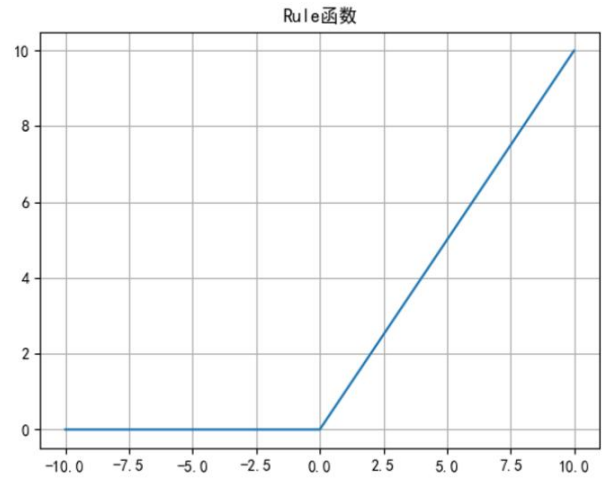
全连接神经网络模仿生物学中神经元的工作状态，将数学过程转换为一层一层的计算。



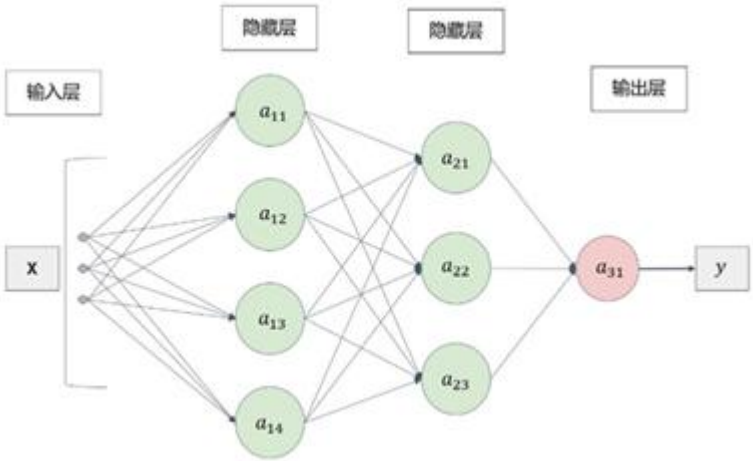
在每层当中，最基本的是一个神经元，其表达式同上面的感知机模型：

$$a = h(wx + b)$$

其中，b 称为偏置，用于控制神经元被激活的容易程度；而 w 表示各信号的权重，用于控制各信号的重要性。h()为激活函数，是一种非线性函数。本次使用了 Sigmoid 函数和 ReLU 函数：

激活函数	函数	导函数	函数图像
Sigmoid	$y = \frac{1}{1+e^{-z}}$	$y' = y(1-y)'$	
ReLU	$y = \begin{cases} z, z > 0 \\ 0, z \le 0 \end{cases}$	$y' = \begin{cases} 1, z > 0 \\ 0, z \le 0 \end{cases}$	

神经网络的整体结构，可以分为输入层、隐藏层（一层或者多层的神经网络层）、输出层。



神经网络的输出是通过前向传播得到的。其将数据特征作为输入，进入隐藏层，与对应的权重相乘，加上偏置，通过激活函数进行激活，并作为下一层神经网络层的输入。不断重复上述过程直到神经网络的输出层，得到输出值。

在反向传播时，通过梯度下降的方法，最小化损失函数，得到收敛的模型。本次在模型中使用的损失函数是交叉熵损失函数，其多用于分类问题：

$$J_x = -\frac{1}{m}\sum_{i=1}^{m}(y_i log\sigma(w^T x_i + b) + (1-y_i)log(1-\sigma(w^T x_i + b)))$$

在梯度下降时，同样使用原参数减去步长乘以J_x对其的偏导。

最终，用训练好的模型处理样本，可以得到对应的预测结果。

4、f1

作为一个常用的分类性能评估指标，f1 分数是精确率（precision）和召回率（recall）的调和平均数，用于衡量分类模型在识别正样本时的准确性。

这里给出 f1 的计算公式：

正确分类的正样本实例（真正例，True Positives，TP）

错误分类为正样本的负样本实例（假正例，False Positives，FP）

错误分类为负样本的正样本实例（假负例，False Negatives，FN）

$$精确率：precision = \frac{TP}{TP+FP}$$

$$召回率：recall = \frac{TP}{TP+EN}$$

$$f1: f1 = 2 \times \frac{precision \times \mathrm{Recall}}{precision + \mathrm{Recall}}$$

四、实验过程与成果

数据使用：

训练集：train.csv

测试集：pred.csv

1、观察数据：

train.csv 给出了一个交易过程的 30 个维度的信息，并给出了对应的风险标签。可以发现，每个维度的数值有正有负，而标签只有 1 和 0 两种。一共 100000 条数据，标签为 1 的只有 300 个，占 0.3%，这个比例是极小的，对分类的精度提出了很高的要求。

2、K 近邻：

首先使用一些非神经网络的方法。第一个尝试了 K 近邻模型，效果一般。因为风险点极少，K 值只能取 1，而为了代码运行时间，训练集数量被限制在了 10000 个，这其中的风险点又可能没有覆盖全。

训练集中标签为 1 的样本数量	训练集验证 f1	训练时间	测试集中标签为 1 的样本数量
264	0.7411	206.911304s	165

```
>> KNB
   264

    0.7411

历时 206.911304 秒。
   165
```

本模型最大的问题在于，并不能武断地认为靠近风险点的点都是风险点。

3、感知机

其后使用感知机模型。由于完整的训练集会耗时过长，只使用了 1000 个进行训练，并用其检验整个训练集，得到了 99.80%左右的正确率。然而，这个错误率与风险样本占比（0.3%）接近，并不能说明模型效果良好，需要进一步验证。

为了能够使用整个训练集的数据，需要对数据进行筛选降维，考虑从 30 个维度中挑选出一定数量的维度。为此，需要获得对最后风险评估最有影响的几个维度。这有多种实现方法，本代码中给出了两种，一种是小训练集均值与 w 乘积表示权重，一种是随机森林算法。

前者的权重公式如下：

$$weight_n = \frac{\sum_{i=1}^{100000} X_{n_i}}{100000} \times w_n$$

其中 n 表示第 n 个维度

在挑选出一定比例的重要维度后，便可以带入整个训练集训练模型。这时，代码的运行时间得到了极大的保障，并且对正确率的影响几乎可以不计。

由于模型给出预测时，并不是直接给出 1 或 0，而是一些介于-1 到 1 的值，我们需要获得合适的阈值，对高于阈值的测试点标 1，低于阈值的测试点标 0。为了在用训练集验证时获得最高的 f1 结果，我们以小步长遍历阈值的合理范围，得到对应 f1 结果最大的阈值。

最终，模型的 f1 值可以达到 0.7 的水平，但并不稳定。以下给出几组结果：

小训练集 30 维度		完整训练集 5 维度		测试集风险数目（即标 1 数目）
threshold	f1	threshold	f1	
0.0087	0.3046	0.0096	0.7111	223
0.0368	0.8225	0.0259	0.4566	322
0.0362	0.7772	0.0039	0.6335	194

在使用本模型进行辅助的初步判断时，可以多次重复运行，找到结果良好的一次运行结果。

```
>> perception
    0.0087

    0.3046

历时 0.673498 秒。
    0.0056

    0.7520

    267
```

4、全连接神经网络

之后尝试了神经网络的方法，这里选择的是全连接神经网络。

先将训练集导入，按列划分为维度向量和风险标签后，按 4：1 的比例随机划分为训练集和验证集。模型由三个密集连接层构成，前两层都有 64 个神经元，并以 Relu 作为激活函数，最后一层只有一个神经元，以 Sigmoid 作为激活函数。

在模型完成训练后，同样需要设定阈值，这里也采用迭代的方法找到最好的阈值（不同的训练会得到不同的阈值，也会得到不一样的结果）。

在 epoch=50 的训练后，得到如下运行过程和运行结果：

```
2024-06-13 21:22:56.054186: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-06-13 21:22:56.055435: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with default inter op sett
Epoch 1/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0113 - accuracy: 0.9985 - val_loss: 0.0062 - val_accuracy: 0.9987
Epoch 2/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0048 - accuracy: 0.9992 - val_loss: 0.0061 - val_accuracy: 0.9991
Epoch 3/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0043 - accuracy: 0.9992 - val_loss: 0.0050 - val_accuracy: 0.9991
Epoch 4/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0037 - accuracy: 0.9991 - val_loss: 0.0056 - val_accuracy: 0.9991
Epoch 5/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0033 - accuracy: 0.9992 - val_loss: 0.0053 - val_accuracy: 0.9991
Epoch 6/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0028 - accuracy: 0.9992 - val_loss: 0.0072 - val_accuracy: 0.9989
Epoch 7/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0027 - accuracy: 0.9992 - val_loss: 0.0073 - val_accuracy: 0.9988
Epoch 8/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0023 - accuracy: 0.9994 - val_loss: 0.0068 - val_accuracy: 0.9991
Epoch 9/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0021 - accuracy: 0.9994 - val_loss: 0.0069 - val_accuracy: 0.9988
Epoch 10/50
2500/2500 [==============================] - 3s 1ms/step - loss: 0.0021 - accuracy: 0.9995 - val_loss: 0.0067 - val_accuracy: 0.9991
```

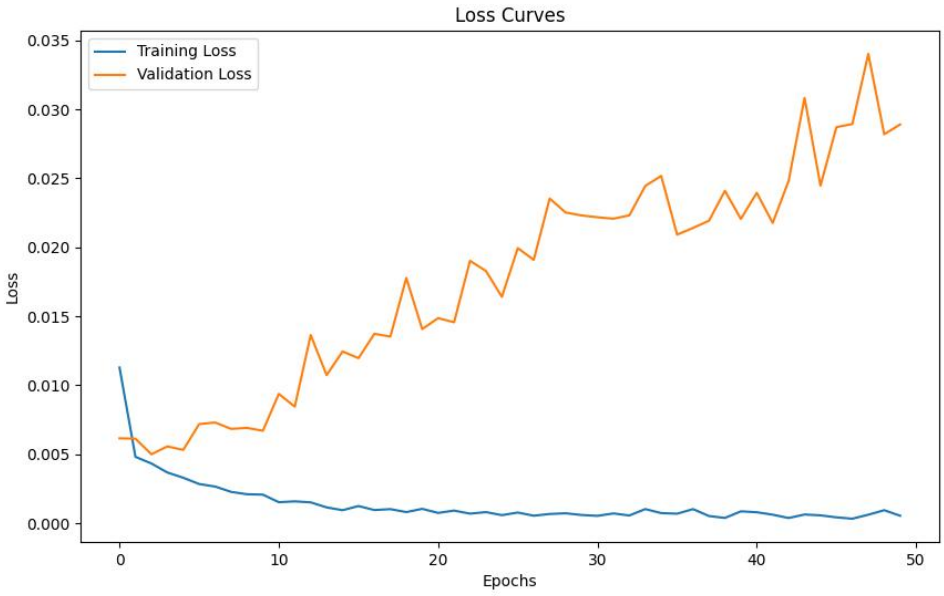
……

```
Epoch 41/50
2500/2500 [==============================] - 3s 1ms/step - loss: 7.9789e-04 - accuracy: 0.9998 - val_loss: 0.0239 - val_accuracy: 0.9991
Epoch 42/50
2500/2500 [==============================] - 3s 1ms/step - loss: 6.2075e-04 - accuracy: 0.9998 - val_loss: 0.0218 - val_accuracy: 0.9991
Epoch 43/50
2500/2500 [==============================] - 3s 1ms/step - loss: 3.7862e-04 - accuracy: 0.9999 - val_loss: 0.0248 - val_accuracy: 0.9991
Epoch 44/50
2500/2500 [==============================] - 3s 1ms/step - loss: 6.3428e-04 - accuracy: 0.9998 - val_loss: 0.0308 - val_accuracy: 0.9991
Epoch 45/50
2500/2500 [==============================] - 3s 1ms/step - loss: 5.7235e-04 - accuracy: 0.9998 - val_loss: 0.0245 - val_accuracy: 0.9991
Epoch 46/50
2500/2500 [==============================] - 3s 1ms/step - loss: 4.2829e-04 - accuracy: 0.9998 - val_loss: 0.0287 - val_accuracy: 0.9991
Epoch 47/50
2500/2500 [==============================] - 3s 1ms/step - loss: 3.2836e-04 - accuracy: 0.9999 - val_loss: 0.0289 - val_accuracy: 0.9989
Epoch 48/50
2500/2500 [==============================] - 3s 1ms/step - loss: 6.1883e-04 - accuracy: 0.9998 - val_loss: 0.0340 - val_accuracy: 0.9990
Epoch 49/50
2500/2500 [==============================] - 3s 1ms/step - loss: 9.4478e-04 - accuracy: 0.9998 - val_loss: 0.0282 - val_accuracy: 0.9991
Epoch 50/50
2500/2500 [==============================] - 3s 1ms/step - loss: 5.4340e-04 - accuracy: 0.9998 - val_loss: 0.0289 - val_accuracy: 0.9987
3125/3125 [==============================] - 2s 706us/step
f1_best=0.9750559329846158
3125/3125 [==============================] - 2s 652us/step
```

loss（训练损失）	Accuracy（训练准确度）	val_loss（验证损失）	val_accuracy（验证准确度）	f1	阈值	训练集风险数目（标签为 1 数目）	训练集风险数目（标签为 1 数目）
5.4340e-04	0.9998	0.0289	0.9987	0.9750559329846158	0.2659	283	177

代码只需要运行不到两分钟，较为迅速。

以下给出了在传播过程中训练集和验证集的损失函数值的变化：



具体结果可以参见 train_with_labels.csv 和 pred_with_labels.csv，前者给出了验证时的标签实际值(Label_p 列)和阈值处理后的值(Label_b 列)，后者给出了预测值(Label 列)。

这个 f1 值颇高，用训练集验证的结果也与实际接近，说明模型结果很好。

五、代码与解析

代码有三个，具体见附件，以下对其进行逐个解析

1、K 近邻：KNB.m

导入数据集，并用列号作为索引，将其划分为特征向量和数据标签。

```
% 训练验证集导入
Train_Validation = readtable('train.csv');
num_t_v = height(Train_Validation);
X = Train_Validation{:,3:32};
Y = Train_Validation{:,33};
```

用训练集验证模型。由于使用全部训练集的代码运行时间过长，这里仅拣选了前 10000 个样本作为训练集，所有样本做测试集。通过调用 K 近邻的函数，得到了预测结果，并输出其中标签为 1 的数量作为对比验证。另外计算 f1 作为评判。

```
% 验证
```

```
tic()
n = 10000;
XTrain = X(1:n,:);
YTrain = Y(1:n);
result = KNB_fun(XTrain,YTrain,X);
disp(length(result(result == 1)))
% 计算真正例（TP）、假正例（FP）、真反例（TN）和假反例（FN，没用，不算）
TP = sum(Y .* result);
FP = sum((1 - Y) .* result);
FN = sum(Y .* (1 - result));
% 计算精确率和召回率，进而计算 F1
precision = TP / (TP + FP + eps);
recall = TP / (TP + FN + eps);
f1 = 2 * (precision * recall) / (precision + recall + eps);
disp(f1)
toc()
```

用测试集测试模型。由于使用全部训练集的代码运行时间过长，这里仍是仅拣选了前 10000 个训练集样本作为训练集，测试整个测试集。通过调用 K 近邻的函数，得到了预测结果，并输出其中标签为 1 的数量。

```
% 测试
Predict = readtable('pred.csv');
XTest = Predict{:,3:32};
YTest = KNB_fun(XTrain,YTrain,XTest);
disp(length(YTest(YTest == 1)))
```

定义了 K 近邻模型的函数。这里使用了实验课的成果，并将其封装为函数，上面也叙述了其原理，不再赘述。重点在于 k 值的选定。由于不知道风险点是不是集中分布，而风险点密度又过小，只能选 1。

```
% K 近邻函数
function result=KNB_fun(XTrain,YTrain,X)
n = height(XTrain);
m = height(X);
YTry = zeros(m,1);                  % 预测集
dis = zeros(n,1);                   % 距离集
k = 1;                              % 近邻数目
k_dis = zeros(k,1);                 % 近邻类别集
for i=1:m
    for j=1:n                       % 计算欧氏距离
        dis(j) = norm(X(i,:)-XTrain(j,:));
    end
    [~, index] = sort(dis);         % 升序排序，index 记录下标
    for j=1:k                       % k 个最近邻数据点的类别标签
        k_dis(j) = YTrain(index(j));
    end
    YTry(i) = mode(k_dis);          % 标签众数即为所求
```

```
end
result = YTry;
end
```

2、感知机：perception.m

导入数据集，与上一个代码完全一致，不再赘述。

```
% 训练验证集导入
Train_Validation = readtable('train.csv');
num_t_v = height(Train_Validation);
X = Train_Validation{:,3:32};
Y = Train_Validation{:,33};
```

为了减少代码的时间复杂度，需要寻找特征向量中权重较高的几个维度。这里有两个方法，分别是小训练集获得 w、b，w 与对应 x 的均值之积表示权重和随机森林。

```
% 得到 x 个最重要的维度
tic()
% 小训练集获得 w、b，w 与对应 x 的均值之积表示权重
n = 1000;     % 小训练集数据集数目
XTrain = X(1:n,:);
YTrain = Y(1:n);
[w1, b1] = perception_fun(XTrain,YTrain,30);
[threshold1, f11] = calculate_metrics(X,Y,w1,b1);
disp(threshold1)
disp(f11)
X_average = sum(X(:,1:30))./num_t_v;
X_weight = X_average .* w1';
[~, indices] = sort(X_weight, 'descend');
%{
% 随机森林
rfModel = TreeBagger(100, X, Y, 'Method', 'classification',
'OOBPredictorImportance', 'on');
[~, indices] = sort(rfModel.OOBPermutedVarDeltaError, 'descend');
%}
x = 5;
indices = indices(1:x);
toc()
```

调用感知机函数使用全集对选定的维度进行训练，并调用 f1 函数得到对应的 f1 值和阈值。

```
% 全集训练
X = X(:, indices);
[w2, b2] = perception_fun(X,Y,x);
[threshold2, f12] = calculate_metrics(X,Y,w2,b2);
disp(threshold2)
disp(f12)
```

调用感知机函数进行预测，并利用得到的阈值对预测结果进行赋值。

```
% 测试
```

```
Predict = readtable('pred.csv');
num_p = height(Predict);
XTest = Predict{:,3:32};
XTest = XTest(:, indices);
YTest = XTest * w2 + b2;
YTest(YTest>threshold2) = 1;
YTest(YTest<1) = 0;
disp(length(find(YTest==1)))
```

定义了感知机模型的函数。这里使用了实验课的成果，并将其封装为函数，上面也叙述了其原理，不再赘述。

```
% 感知机函数
function [w_result,b_result]=perception_fun(X,Y,w_num)
w = zeros(w_num,1);
b = 0;
x = 100000;        % 迭代次数
c = 0.0001;         % 步长
for i = 1:x
    a = randi([1,height(Y)]);
    if (X(a,:)*w+b)*Y(a) <= 0
        w = w+(X(a,:)').*c.*Y(a);
        b = b+c*Y(a);
    end
end
w_result=w;
b_result=b;
end
```

定义了获取最佳 f1 值及对应阈值的函数。这里以 0.0001 为步长，遍历了阈值的可能范围，找到其中最好的 f1 值。

```
% 最佳 F1 值
function [best_threshold, best_f1] = calculate_metrics(X,Y,w,b)
best_threshold = 0;
best_f1 = 0;
for threshold = 0:0.0001:0.1
    result = X * w + b;
    result(result>threshold) = 1;
    result(result<1) = 0;
    % 计算真正例（TP）、假正例（FP）、真反例（TN）和假反例（FN，没用，不算）
    TP = sum(Y .* result);
    FP = sum((1 - Y) .* result);
    FN = sum(Y .* (1 - result));
    % 计算精确率和召回率，进而计算 F1
    precision = TP / (TP + FP + eps);
    recall = TP / (TP + FN + eps);
    f1 = 2 * (precision * recall) / (precision + recall + eps);
```

```
    if f1 > best_f1
        best_f1 = f1;
        best_threshold = threshold;
    end
end
end
```

3、全连接神经网络：FCNN.py

导入要用的包。

Numpy 是 Python 的一个核心库，用于处理、运算大型多维数组和矩阵。

Pandas 用于数据预处理和特征工程。

Keras 是一个高级神经网络 API，可以运行在 TensorFlow 上，其中 Sequential 是一个定义简单堆叠层（即顺序模型）的模型类；layers 模块包含神经网络中的各种层，Dense 是全连接层；optimizers 模块包含用于优化神经网络参数的优化算法，Adam 是其中广泛使用的一种。

sklearn 是一个机器学习库。train_test_split 是其 model_selection 模块中的一个函数，用于将数据集拆分为训练集和验证集。f1_score 是其 metrics 模块中的一个函数，用于计算 f1 分数。

Matplotlib 是一个数据可视化绘图库，提供了与 MATLAB 类似的绘图框架。pyplot 是其中一个子模块，用于绘制各种类型的图形和图表。

```
import numpy as np
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
import matplotlib.pyplot as plt
```

导入数据集，并用列名作为索引，将其划分为特征向量和数据标签。并使用相应的函数，将其按 4：1 的比例随机划分为训练集和测试集，并且这种随机将在每次运行时保持一致。

```
# 导入数据集并划分训练集和验证集
t_v = pd.read_csv('train.csv')
x_col = ['V' + str(i) for i in range(1, 31)]
X = t_v[x_col]
X = X.astype('float32')
y = t_v['Label']
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)
```

定义全连接神经网络模型。模型由三个密集连接层构成。第一层的输入维度是 30，有 64 个神经元，并以 Relu 作为激活函数；第二层接受第一层的输出结果，同样有 64 个神经元，并以 Relu 作为激活函数；最后一层接受第二层的输出结果，只有一个神经元，以 Sigmoid 作为激活函数。

```
# 定义模型
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(30,)))
```

```
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

配置模型的学习过程。这里指定了 Adam 优化器，其是一种自适应学习率的优化算法，学习率设置为 0.001；指定了损失函数为二元交叉熵，其常用于二分类问题，衡量了模型预测的概率分布与真实概率分布之间的差异；另指定了评估指标为准确率。

```
# 编译模型
model.compile(optimizer=Adam(learning_rate=0.001),
loss='binary_crossentropy', metrics=['accuracy'])
```

训练模型。这里确定了训练集和验证集，并将训练轮数 epochs 设定为 50，每个 epoch 中模型会遍历整个训练数据集一次。而在一个 epoch 中，每个批次（batch）将使用 batch_size=32 的样本。指定 verbose=1，这会在训练时显示进度条。最后返回一个 History 对象，它包含了训练过程中的损失值和评估指标值。

```
# 训练模型
history = model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_val, y_val), verbose=1)
```

用模型预测整个数据集，用其来找到最适宜的阈值。在遍历中，会在获得更大的 f1 值时更新阈值和 f1，而阈值每次移动的步长是 0.0001。找到最优的阈值后，会将阈值处理前后的预测值输出到一个新的表格中。

```
# 找到预测阈值（高于阈值的为 1，低于的为 0）
predictions_p = model.predict(X)
t_v['Label_p'] = predictions_p
threshold = 0
threshold_best = 0
f1_best = 0
while threshold < 1:
    predictions_p_binary = np.where(predictions_p > threshold, 1, 0)
    f1 = f1_score(y, predictions_p_binary, average='macro')
    if f1 > f1_best:
        threshold_best = threshold
        f1_best = f1
    threshold += 0.0001
predictions_p_binary = np.where(predictions_p > threshold_best, 1, 0)
t_v['Label_b'] = predictions_p_binary
t_v.to_csv('train_with_labels.csv', index=False)
print(f"f1_best={f1_best}")
```

之后用模型预测测试集，并使用之前获得的阈值进行赋值判断。最后得到的预测值会输出到另一个新的表格中。

```
# 测试
t = pd.read_csv('pred.csv')
X_test = t[x_col]
X_test = X_test.astype('float32')
predictions = model.predict(X_test)
predictions_binary = np.where(predictions > threshold_best, 1, 0)
```

```
t['Label'] = predictions_binary
t.to_csv('pred_with_labels.csv', index=False)
```

最后，利用 history 中的信息，绘制训练过程中，训练集和验证集损失函数的变化。

```
# 绘制损失和验证损失曲线
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Curves')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

六、分析

本问题给出了大量的数据，数据的维度也很高，需要将其分为两类，这对一些分类边缘的数据，即一些异常数据的分类问题提出了很高的要求。最开始使用的 K 近邻模型，有一定的指导意义，但运行时间过长。其后使用感知机模型，但在代码运行时间上遇到了很大的问题，这表明这种传统的机器学习算法，也并不适用于超大量数据集的实际问题。但是，在寻找方法减少时间损耗后，感知机模型也能给出一个有参考意义的结果，对处理问题有一定帮助。在此基础上，运用全连接神经网络这种最基本的神经网络，便取得了很好的成果。在评价模型效果时，采用了 f1 作为标准。这个标准平衡了准确度和召回率，可以在迭代中找到其最好的值。

最后采纳的全连接神经网络模型，在结果上比较优秀，然而仍是无法达到极高的判断精确度。这其实并不是模型准确度的问题。面对超大量的数据，少数风险点被掩藏了起来，这在现实情况中也是合理而正常的。当前的正确率已经可以很好地作为辅助工具，帮助评估该问题情况下的风险。为了进一步提高模型的鲁棒性，可以尝试更多的不同的神经网络。

七、模型结果

在解决本题目时，尝试了多种模型，各有不同的结果。这里选择了全连接神经网络的结果作为题目的解。请详见 pred_with_labels.csv，其最后的 Label 列即是预测结果。