

(2) 准备带转换的代码文件：将 6.3 节代码文件“6-3 动态图另一种梯度方法.py”复制到本地，用于升级转换。

### 6.13.2 使用工具转换源码

安装好 TensorFlow 2.x 版本之后，在命令行中执行以下操作。

- (1) 激活该版本的虚环境（作者的 TensorFlow 2.x 版本所在的虚环境为 tf2）。
- (2) 用 `tf_upgrade_v2` 工具进行转换。具体命令如下：

```
activate tf2
tf_upgrade_v2 --infile 6-3_动态图另一种梯度方法.py --outfile ./ 6-22_tf2code.py
```

该命令执行后，会在本地目录下生成一个 `report.txt` 文件。

在 `report.txt` 文件里记录了 `tf_upgrade_v2` 工具的详细转化工作。具体内容如下：

```
Processing file '6-3 动态图另一种梯度方法.py'
outputting to './6-22_tf2code.py'

'6-3 动态图另一种梯度方法.py' Line 18
-----
.....  

Renamed function 'tf.train.Saver' to 'tf.compat.v1.train.Saver'
Old: saver = tf.train.Saver([W,b], max_to_keep=1)
~~~~~
New: saver = tf.compat.v1.train.Saver([W,b], max_to_keep=1)
~~~~~
```

同时，在本地路径下也生成了源代码文件“6-22\_tf2code.py”。

### 6.13.3 修改转换后的代码文件

因为 TensorFlow 2.x 版本不支持 contrib 模块，所以需要将用到 contrib 部分的代码全都删掉。具体代码如下：

代码 6-22 `tf2code` (片段)

```
01 import tensorflow as tf
02 import numpy as np
03 import matplotlib.pyplot as plt
04 import tensorflow.contrib.eager as tfe      # 不再支持 contrib 模块，所以需要删掉
05
06 tfe.enable_eager_execution()                 # 默认就是启动动态图，所以需要删掉
07 print("TensorFlow 版本: {}".format(tf.version))
08 print("Eager execution: {}".format(tf.executing_eagerly()))
09 .....
10 # 将 tfe 改为 tf
```

```

11 W = tfe.Variable(tf.random.normal([1]), dtype=tf.float32, name="weight")
12 b = tfe.Variable(tf.zeros([1]), dtype=tf.float32, name="bias")
13 .....
14 #定义 saver, 演示两种操作检查点文件的方法
15 savedir = "logeager/"
16 savedirx = "logeagerx/"
17 saver = tf.compat.v1.train.Saver([W,b], max_to_keep=1)
18 saverx = tfe.Saver([W,b])      #删除 contrib 的检查点文件操作
19
20 kpt = tf.train.latest_checkpoint(savedir)          #找到检查点文件
21 kpx = tf.train.latest_checkpoint(savedirx)         #找到检查点文件
22 if kpt!=None:
23     saver.restore(None, kpt)
24     saverx.restore(kpx)                            #删除 contrib 的恢复检查点文件操作
25 .....
26 #显示训练中的详细信息
27 if step % display_step == 0:
28     cost = getcost(x, y)
29     print ("Epoch:", step+1, "cost=", cost.numpy(), "W=", W.numpy(), "b=", b.numpy())
30     if not (cost == "NA" ):
31         plotdata["batchsize"].append(global_step.numpy())
32         plotdata["loss"].append(cost.numpy())
33     saver.save(None, savedir+"linermode1ckpt", global_step)
34     saverx.save(savedirx+"linermode1epkt", global_step) #删除生成检查点文件的操作
35 .....

```

在修改代码时，直接将调用 contrib 模块操作检查点文件的代码删掉即可。程序运行时，会用静态图的方式对检查点文件进行操作。

代码运行后，程序输出的结果与 6.3.5 小节一致，这里不再详述。

#### 6.13.4 将代码升级到 TensorFlow 2.x 版本的经验总结

下面将升级代码到 TensorFlow 2.x 版本的方法汇总起来，有如下几点。

##### 1. 最快速转化的方法

在代码中没有使用 contrib 模块的情况下，可以在代码最前端加上如下两句，直接可以实现的代码升级。

```

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

```

这种方法只是保证代码在 TensorFlow 2.x 版本上能够运行，并不能发挥 TensorFlow 的最大性能。

##### 2. 使用工具进行转化的方法

在代码中没有使用 contrib 模块的情况下，用 `tf_upgrade_v2` 工具可以快速实现代码升级。

当然 `tf_upgrade_v2` 工具并不是万能的，它只能实现基本的 API 升级。一般在转化完成之后还需要手动二次修改。

### 3. 将静态图改成动态图的方法

静态图可以看作程序的运行框架，可以将输入输出部分原样的套用在函数的调用框架中。具体步骤如下：

- (1) 将会话（session）转化成函数。
- (2) 将注入机制中的占位符（`tf.placeholder`）和字典（`feed_dict`）转化成函数的输入参数。
- (3) 将会话运行（`session.run`）后的结果转化成函数的返回值。

在实现过程中，可以通过自动图功能，用简单的函数逻辑替换静态图的运算结构。自动图的详细介绍请参考 6.1.16 小节。

### 4. 将共享变量的作用于转成 Python 对象的命名空间

在定义权重参数时，用 `tf.Variable` 函数替换 `tf.get_variable` 函数。每个变量的命名空间（`variable_scope`）用类对象空间进行替换，即将网络封装成类的形式来搭建模型。

在封装类的过程中，可以继承 `tf.keras` 接口（如：`tf.keras.layers.Layer`、`tf.keras.Model`）也可以继承更底层的接口（如 `tf.Module`、`tf.layers.Layer`）。

在对模型进行参数更新时，可以使用实例化类对象的 `variables` 和 `trainable_variables` 属性来控制参数。

### 5. 升级 TF-slim 接口开发的程序

TensorFlow 2.x 版本将彻底抛弃 TF-slim 接口，所以升级 TF-slim 接口程序会有较大的工作量。官方网站给出的指导建议是：如果手动将 TF-slim 接口程序转化为 `tf.layers` 接口实现（因为二者的使用方法相对比较类似，见 6.6 节），则可以满足基本使用；如果想与 TensorFlow 2.x 版本结合得更加紧密，则可以再将其转化为 `tf.keras` 接口。

到样本中的潜在规则，提高出更好的预测效果。

## 2. 特征工程的方法

特征工程可以理解为数据科学中的一门，包含了许多实用的技巧和方法。对于初学者来说，入门并不难，不过随着深度学习的发展，越来越多的解决方案开始出现，使得机器学习算法来降低人工干预的需求。因此，本书将更多地把精力放在特征工程的基础部分。

# 第3篇 进阶

在特征工程部分，本书将主要介绍以下3种。

本篇主要讲解机器学习算法的相关内容，主要分为两部分：特征工程、神经网络。

在特征工程部分，主要介绍特征列变换、机器学习的使用方法。这些方法都具有强解释性。

在神经网络部分，主要介绍卷积神经网络、循环神经网络的相关模型。这些模型都是目前相对主流的成熟模型。

通过本篇的学习，读者可以学会如何选择模型，以及使用模型完成特定的机器学习任务。

- 第7章 特征工程——会说话的数据

- 第8章 卷积神经网络（CNN）——图像处理中应用最广泛的模型

- 第9章 循环神经网络（RNN）——处理序列样本的神经网络

## 1. 简洁数据特征

简洁的数据特征是机器学习模型成功的关键。例如，收入、年龄、教育程度等特征，即数据之间彼此没有显著的关联，称为“新民主义”特征。

在对离散数据进行特征工程时，通常会遇到两个主要的问题，具体分为两类：

- 具有固定数量的样本（例如，航班延误情况）。在这种情况下，特征值不能直接进行交换。

- 没有固定类别的样本（例如，成年人、中学生）。这种情况下，特征值可以任意交换。

## 第8章 基础算工量表

然后可以通过对特征进行转换（如取对数或平方根）或对特征进行归一化处理，使特征满足线性假设。例如，身高、体重、年龄、性别等，数据通常的同不言其本身影响不大，但是当特征量级差距过大时，线性假设就无法满足。因此，在对特征进行转换时，需要根据特征的性质进行不同的处理。

在对连续数据特征进行转换时，常对其进行线性或归一化处理，使其满足线性假设。例如，年龄、性别、收入等特征属于离散型特征，通常会将其转化为二进制特征；而身高、体重、年龄等特征属于连续型特征，通常会将其转化为区间特征。如果特征之间存在线性相关性，那么在对特征进行转换时，可以有

# 第 7 章

## 特征工程——会说话的数据

特征工程本质上是一种工程方法，即从原始数据中提取最优特征，以供算法或模型使用。在机器学习任务中，应用领域不同，特征工程的重要程度也不同。

- 在数值分析任务中，特征工程的重要性尤为突出。能否提取出好的特征，对模型的训练结果有很大影响。一旦提取不到有用的样本特征，或是太多无用的样本特征进入模型，都会让模型的精度大打折扣。
- 在图像处理任务中，特征工程的作用不大，因为在图像处理任务中，图片样本都是像素值在 0~255 的数字，是固定值域。
- 在文本处理任务中，将样本进行分词、向量化之后，也会将值域统一起来。不再需要使用特征工程的方法对样本数值进行重组。

本章重点介绍在数值分析任务中，从样本里提取特征，并进行转换的各种方法。如果读者掌握了这些方法，便可以根据已有任务选择合适的处理方法，对样本数据进行有效特征的提取，完成数值的分析。

### 7.1 快速导读

在学习实例之前，有必要了解特征工程的基础知识。

#### 7.1.1 特征工程的基础知识

特征工程发生在训练模型之前的样本预处理环节。

在数值分析任务中，不同的样本具有不同的字段属性，如名字、年龄、地址、电话等，这些信息是以不同形式存在的。如果想要使用算法或模型进行分析，则需要将样本中的信息转化成模型能够处理的数据——浮点型数据。这便是特征工程主要做的事情。

##### 1. 特征工程的作用

在特征工程中，为了降低模型的拟合难度，除需要对字段属性做数值转化外，还需要根据任务本身做属性的增减。这相当于用人的理解力对数据做一次加工，帮助神经网络更好地理解数据。特征工程做得越好，数据的表征能力就会越强。

在训练模型环节，表征能力强的样本会给神经网络一个明显的指导信号，使模型更容易学

到样本中的潜在规则，表现出更好的预测效果。

## 2. 特征工程的方法

特征工程可以理解为数据科学中的一种，包含了许多数据分析的知识和技巧，让初学者很难入门。不过随着深度学习的发展，越来越多的解决方案倾向于通过拟合能力更强的机器学习算法来降低人工干预度，减小对特征工程的依赖程度。这使得特征工程的作用越来越接近于单纯的数值转化。所以，读者只需要掌握一些特征工程的基本方法即可，不再需要将更多的精力放在特征工程算法上。

在特征工程中，常用的特征提取方法有以下3种。

- 单纯对特征的选择操作。
- 通过特征之间的运算，构造出新的特征（比如有两个特征 $x_1$ 、 $x_2$ ，通过计算 $x_1+x_2$ 来生成一个新的特征）。
- 通过某些算法来生成新的特征（比如主成分分析算法，或先经过深度神经网络算出一部分特征值）。

这3种方法在使用时，只有相关的指导思想，没有固定的使用模式。除依靠个人经验外，还可以用机器学习算法进行筛选，但用机器学习算法进行筛选的过程会需要大量的算力作为支撑。

### 7.1.2 离散数据特征与连续数据特征

样本的数据特征主要可以分为两类：离散数据特征和连续数据特征。

#### 1. 离散数据特征

离散数据特征类似于分类任务中的标签数据（例如，男人、女人）所表现出来的特征，即数据之间彼此没有连续性。具有该特征的数据被叫作离散数据。

在对离散数据做特征变换时，常常将其转化为one-hot编码或词向量，具体分为两类。

- 具有固定类别的样本（例如，性别）：处理起来比较容易，可以直接按照总的类别数进行变换。
- 没有固定类别的样本（例如，名字）：可以通过hash算法或类似的散列算法将其分散，然后再通过词向量技术进行转化。

#### 2. 连续数据特征

连续数据特征类似于回归任务中的标签数据（例如，年纪）所表现出来的特征，即数据之间彼此具有连续性。具有该特征的数据被叫作连续数据。

在对连续数据做特征变换时，常对其做对数运算或归一化处理，使其具有统一的值域。

#### 3. 连续数据与离散数据的相互转化

在实际应用中，需要根据数据的特性选择合适的转化方式，有时还需要实现连续数据与离散数据间的互相转化。

例如，对一个值域跨度很大（例如， $0.1 \sim 10000$ ）的特征属性进行数据预处理时，可以有

以下 3 种方法。

- (1) 将其按照最大值、最小值进行归一化处理。
- (2) 对其使用对数运算。
- (3) 按照其分布情况将其分为几类，做离散化处理。

具体选择哪种方法还要看数据的分布情况。假设数据中有 90% 的样本在 0.1~1 之间，只有 10% 的样本在 1000~10000 之间。那么使用第（1）种和第（2）种方法显然不合理。因为这两种方法只会将 90% 的样本与 10% 的样本分开，并不能很好地体现出这 90% 的样本的内部分布情况。

而使用第（3）种方法，可以按照样本在不同区间的分布数量对样本进行分类，让样本内部的分布特征更好地表达出来。

### 7.1.3 了解特征列接口

特征列（tf.feature\_column）接口是 TensorFlow 中专门用于处理特征工程的高级 API。用 tf.feature\_column 接口可以很方便地对输入数据进行特征转化。

特征列就像是原始数据与估算器之间的中介，它可以将输入数据转化成需要的特征样式，以便传入模型进行训练。

### 7.1.4 了解序列特征列接口

序列特征列接口（tf.contrib.feature\_column.sequence\_feature\_column）是 TensorFlow 中专门用于处理序列特征工程的高级 API。它是在 tf.feature\_column 接口之上的又一次封装。该 API 目前还在 contrib 模块中，未来有可能被移植到主版本中。

在序列任务中，使用序列特征列接口（sequence\_feature\_column）会大大减少程序的开发量。

在序列特征列接口中一共包含以下几个函数。

- sequence\_input\_layer：构建序列数据的输入层。
- sequence\_categorical\_column\_with\_hash\_bucket：将序列数据转化成离散分类特征列。
- sequence\_categorical\_column\_with\_identity：将序列数据转化成 ID 特征列。
- sequence\_categorical\_column\_with\_vocabulary\_file：将序列数据根据词汇表文件转化成特征列。
- sequence\_categorical\_column\_with\_vocabulary\_list：将序列数据根据词汇表列表转化成特征列。
- sequence\_numeric\_column：将序列数据转化成连续值特征列。

在 7.5 节还会演示序列特征列 API 的使用实例。

### 7.1.5 了解弱学习器接口——梯度提升树（TFBT 接口）

TFBT 接口实现了梯度提升树（gradient boosted trees）算法。梯度提升树算法适用于多种机器学习任务。

TFBT 是一个弱学习器接口，其中包括两套 API，都可以处理“分类任务”和“回归任务”。以“分类任务”为例，这两套 API 如下所示。

- contrib 模块中的 API: tensorflow.contrib.boosted\_trees 接口。
- 估算器框架中的 API: tf.estimator.BoostedTreesClassifier 接口。

其中，contrib 模块中的 API 都是非官方支持的第三方实验型 API，其功能较新、较全，但不稳定。

在主框架中的“回归任务”的接口是 tf.estimator.BoostedTreesRegressor。



#### 提示：

在 TensorFlow 2.x 版本中没有 contrib 模块。建议读者优先使用估算器框架中的 API。

### 7.1.6 了解特征预处理模块 (tf.Transform)

特征预处理模块 (tf.Transform) 是一个对数据进行预处理的库。在训练 NLP、数值分析等模型时，利用它可以很方便地对数据进行预处理，例如：

- 将输入值做平均值计算或标准偏差归一化处理。
- 根据输入文本生成字典，并将文本按照字典转换为索引。
- 将输入的连续值数据特征按照指定界限进行划分（桶机制），并根据划分的界限将其转化为整数索引（离散数据特征）。

#### 1. 安装 tf.Transform 模块

tf.Transform 模块独立于 TensorFlow 安装包，需要另外单独安装，具体方法是，在命令行里输入以下命令：

```
pip install tensorflow-transform
```

#### 2. 了解 tf.Transform 的依赖库

如果 tf.Transform 模块在本地分布运行，则会依赖于 Apache Beam 库。如果在 TPU 云上运行，则依赖于 Google Cloud Dataflow。



#### 提示：

截至本书定稿时，tf.Transform 模块还在开发之中，只支持 Python 2.7 到 Python 3.0，对于本书使用的 Python 3.6 版本并不支持。这是由于当前的 tf.Transform 在 Python 3.0 之后的版本上还存在未能解决的重要 Bug，影响模块的发布。

读者可以先关注该模块的发布信息，具体链接如下：

<https://github.com/tensorflow/transform/>

如果在未来发布了支持 Python 3.6 版本的 tf.Transform 模块，则便可以使用。

### 7.1.7 了解因子分解模块

TensorFlow 中提供了一个因子分解模块 (factorization)，其中包含 GMM (高斯混合模型)、kmeans (聚类算法)、WALS (加权矩阵分解) 算法。它们是估算器框架的 3 种实例化实现。

因子分解模块的用法与估算器的用法基本一致，它可以使机器学习代码与深度学习无缝连接。它的具体使用方法见本书 7.4 节。

### 7.1.8 了解加权矩阵分解算法

加权矩阵分解 (WALS) 算法采用加权交替矩阵分解的最小二乘法实现，能够将非常稀疏的矩阵因子分解成两个稠密矩阵的乘积，如图 7-1 所示。

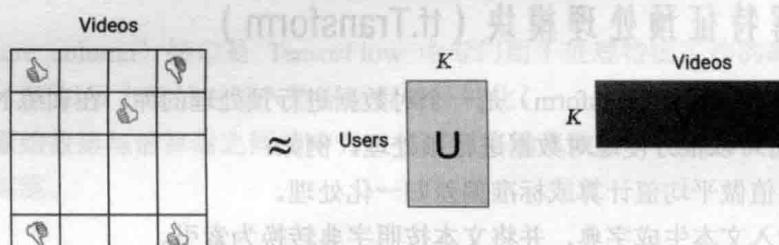


图 7-1 WALS 算法示例图

在图 7-1 的左侧：

- 横坐标为 Videos，代表视频的 ID。
- 纵坐标为 Users，代表用户的 ID。
- 二者交叉的方格，代表某个用户对某个视频给予的好评或差评。

如果该数据来自于一个视频网站，则 Users 和 Videos 的个数会非常多，且这两个字段都属于离散型字段。如果用 one-hot 编码来表征，则是非常庞大的维度，并且大部分的值都为 0，这显然不合适。

图 7-1 左侧的矩阵被分解成了右侧的两个稠密矩阵：Users 与 Videos。

- 在 Users 矩阵中，行数代表用户的个数，列数为  $K$ （可以在算法中指定）。
- 在 Videos 矩阵中，列数为  $K$ （与 Users 的行数相同），行数为视频的个数。

在图 7-1 中，左侧的矩阵可以理解为由右侧的 Users 矩阵与 Videos 矩阵相乘得来（中间的  $K$  个维度会在相乘过程中被约分掉）；图 7-1 右侧的 Users 矩阵与 Videos 矩阵可以理解为由左侧矩阵分解得来。

在训练 WALS 模型时，只关注稀疏矩阵中有值的部分。具体步骤如下：

- (1) 将 Users 矩阵与 Videos 矩阵中指定的元素相乘。
- (2) 将 (1) 的结果与对应的标签评论值进行比较，计算损失。
- (3) 根据损失来调整权重参数。
- (4) 多次迭代，使得 Users 矩阵与 Videos 矩阵中指定元素的相乘结果，越来越接近图 7-1 左侧中对应位置的评论值。

在使用 WALS 模型时，便可以在 Users 矩阵与 Videos 矩阵的相乘结果中，找到指定用户的所有视频评论值。如果对该评论值进行排名，便是一个关于该用户的推荐算法。

另外，通过 WALS 算法之后，每个用户或每个视频便都可以用  $K$  个维度的向量来表示，再也不需要用庞大的 one-hot 编码来表示了。

### 7.1.9 了解 Lattice 模块——点阵模型

TensorFlow 中的 Lattice 模块是一个点阵模型（插值查找表），该模型通过“单调校准插值查找表”算法实现。该算法通过插值方式配合单调函数来学习样本中的数据特征，具有很好的可解释性，善于解决低维数据的相关任务。

在样本比较少的情况下，Lattice 模块的准确性会高于深度学习模型的准确性，同时 Lattice 模块也为用户提供了更高的算法透明度。

更多理论可参考以下链接：

<http://jmlr.org/papers/v17/15-243.html>

<https://ai.google/research/pubs/pub46327>

#### 1. 安装 Lattice 模块

Lattice 不在 TensorFlow 的官方模块里，所以需要单独安装。现有的二进制包只支持 Linux 与 Mac 操作系统（如果要在 Windows 系统中使用，则需要对源码进行编译）。具体安装命令如下：

```
pip install tensorflow-lattice
```

#### 2. Lattice 的内部模块介绍

安装好的 Lattice 模块可用于回归和分类任务。它能够单独进行计算处理（类似计算器的使用方法），也可以作为神经网络中的一层进行联合的计算处理。

Lattice 模块内部包括以下 6 个子模块。

- 校准线性模型：将每个特征进行一维线性转换，然后把所有校准后的特征进行线性连接。它适用于训练非常小或没有复杂的非线性输入交互的数据集。
- 校准点阵模型：将校准后的特征用两层单个点阵模型进行非线性连接，可以展现数据集中的复杂非线性交互。它适用于特征数在 10 个以下的数据集。
- 随机微点阵模型（Random Tiny Lattices，RTL）：是一个优化后的点阵模型。它可以使参数的数量可控。正常来讲，一个具有  $D$  个特征的点阵模型，需要至少  $2^D$  个参数。RTL 的微点阵单元是由若干个点阵相加所组成，这种结构使得整体的点阵参数不至于随着特征的增加而呈指数级增长。在实现时，给每个微点阵单元设置一个特征维度  $DL$ ，每个微点阵单元从总的  $D$  维特征中随机取  $DL$ （微点阵单元的特征维度）个特征来进行运算。然后由任意多个微点阵单元组成整个 RTL。
- 集合的微点阵模型（Ensembled Tiny Lattices，ETL）：与 RTL 模型类似，只不过每个微点阵的维度是个随机值。同时还会对输入的  $D$  维特征做一次线性变换。相比 RTL 模型，ETL 模型的灵活度更强，但会缺乏一些可解释性，而且也需要更长的时间训练。

- 校准层：对数据进行分段线性校准，可以对接到其他神经网络里。
- 点阵层：对数据进行内插值查表转化，可以对接到其他神经网络里。

## 7.1.10 联合训练与集成学习

联合训练 (joint training) 与集成学习 (ensemble learning) 都属于使用多模型处理单一任务的训练方法。

二者的相同之处是：将多种学习算法组合在一起，以便取得更好的结果。例如，采用多个分类器对数据集进行预测，从而提高整个分类器的泛化能力。

二者的不相同之处是：

- 集成学习方法中的每个模型都是独立进行训练的，模型的融合过程发生在最终的预测阶段。
- 联合训练方法中的所有模型都是同时训练的，彼此共享误差。模型的融合过程发生在训练阶段。每个模型的权重会随整体的训练误差进行调整。

本书 7.2 节实例中介绍的 wide\_deep 模型，使用的就是联合训练方法。

## 7.2 实例 31：用 wide\_deep 模型预测人口收入

本实例用 wide\_deep 模型预测人口收入。wide\_deep 模型来自于谷歌公司，在 Google Play 的 APP 推荐算法中就使用了该模型。

wide\_deep 模型的核心思想是：结合线性模型的记忆能力 (memorization) 和 DNN 模型的泛化能力 (generalization)，在训练过程中同时优化两个模型的参数，从而实现最优的预测能力。

### 实例描述

有一个人口收入的数据集，其中记录着很多人的详细信息及收入情况。

现需要训练一个机器学习模型，使得该模型能够找到个人的详细信息与收入之间的关系。最终实现：在给定一个人的具体详细信息之后，该模型能估算出他的收入水平。

本实例具有很好的学习价值，下面就来详细讲解一下。

### 7.2.1 了解人口收入数据集

该数据集的具体信息见表 7-1。

表 7-1 人口收入数据集

| 数据集项目  | 具体值   |
|--------|-------|
| 数据集的特征 | 多元    |
| 实例的数目  | 48842 |

续表

| 数据集项目 | 具体值    |
|-------|--------|
| 区域    | 社会     |
| 属性特征  | 分类, 整数 |
| 属性的数目 | 14个    |

数据集中收集了 20 多个地区的人口数据, 每个人的详细信息包括年龄、职业、教育等 14 个维度, 一共有 48842 条数据。本实例从其中取出 32561 条数据用作训练模型的数据集, 剩余的数据将作为测试模型的数据集。

## 1. 部署数据集

在本书的配套资源里提供了两个数据集文件——adult.data.csv 与 adult.test.csv, 将这两个文件复制到本地代码的 income\_data 文件夹下, 如图 7-2 所示。

用 wide and deep 模型预测人口收入 &gt; income\_data

名称

- adult.data.csv
- adult.test.csv

图 7-2 人口收入数据集

在图 7-2 中, adult.data.csv 是训练数据集, adult.test.csv 是测试数据集。

## 2. 数据集内容介绍

用 Excel 打开数据集文件, 便可以看到具体内容, 如图 7-3 所示。

| A  | B                   | C                   | D                     | E                | F             | G                  | H      | I           | J           | K     | L | M | N | O |
|----|---------------------|---------------------|-----------------------|------------------|---------------|--------------------|--------|-------------|-------------|-------|---|---|---|---|
| 1  | 25 Private          | 226802 11th         | 7 Never-married       | Machine-op-inspt | Own-child     | Black              | Male   | 0           | 0 40 area_A | <=50K |   |   |   |   |
| 2  | 38 Private          | 89814 HS-grad       | 9 Married-civ-spouse  | Farming-fishing  | Husband       | White              | Male   | 0           | 0 50 area_A | <=50K |   |   |   |   |
| 3  | 28 Local-gov        | 336951 Assoc-acdm   | 12 Married-civ-spouse | Protective-serv  | Husband       | White              | Male   | 0           | 0 40 area_A | >50K  |   |   |   |   |
| 4  | 44 Private          | 160323 Some-college | 10 Married-civ-spouse | Machine-op-inspt | Husband       | Black              | Male   | 7688        | 0 40 area_A | >50K  |   |   |   |   |
| 5  | 18 ?                | 103497 Some-college | 10 Never-married      | ? Own-child      | White         | Female             | 0      | 0 30 area_A | <=50K       |       |   |   |   |   |
| 6  | 34 Private          | 198693 10th         | 6 Never-married       | Other-service    | Not-in-family | White              | Male   | 0           | 0 30 area_A | <=50K |   |   |   |   |
| 7  | 29 ?                | 227026 HS-grad      | 9 Never-married       | ? Unmarried      | Black         | Male               | 0      | 0 40 area_A | <=50K       |       |   |   |   |   |
| 8  | 63 Self-emp-not-inc | 104626 Prof-school  | 15 Married-civ-spouse | Prof-specialty   | Husband       | White              | Male   | 3103        | 0 32 area_A | >50K  |   |   |   |   |
| 9  | 24 Private          | 369667 Some-college | 10 Never-married      | Other-service    | Unmarried     | White              | Female | 0           | 0 40 area_A | <=50K |   |   |   |   |
| 10 | 55 Private          | 104996 7th-8th      | 4 Married-civ-spouse  | Craft-repair     | Husband       | White              | Male   | 0           | 0 10 area_A | <=50K |   |   |   |   |
| 11 | 65 Private          | 184454 HS-grad      | 9 Married-civ-spouse  | Machine-op-inspt | Husband       | White              | Male   | 6418        | 0 40 area_A | >50K  |   |   |   |   |
| 12 | 36 Federal-gov      | 212465 Bachelors    | 13 Married-civ-spouse | Adm-clerical     | Husband       | White              | Male   | 0           | 0 40 area_A | <=50K |   |   |   |   |
| 13 | 26 Private          | 82091 HS-grad       | 9 Never-married       | Adm-clerical     | Not-in-family | White              | Female | 0           | 0 39 area_A | <=50K |   |   |   |   |
| 14 | 58 ?                | 299831 HS-grad      | 9 Married-civ-spouse  | ? Husband        | White         | Male               | 0      | 0 35 area_A | <=50K       |       |   |   |   |   |
| 15 | 48 Private          | 279724 HS-grad      | 9 Married-civ-spouse  | Machine-op-inspt | Husband       | White              | Male   | 3103        | 0 48 area_A | >50K  |   |   |   |   |
| 16 | 43 Private          | 346189 Masters      | 14 Married-civ-spouse | Exec-managerial  | Husband       | White              | Male   | 0           | 0 50 area_A | >50K  |   |   |   |   |
| 17 | 20 State-gov        | 444554 Some-college | 10 Never-married      | Other-service    | Own-child     | White              | Male   | 0           | 0 25 area_A | <=50K |   |   |   |   |
| 18 | 43 Private          | 128354 HS-grad      | 9 Married-civ-spouse  | Adm-clerical     | Wife          | White              | Female | 0           | 0 30 area_A | <=50K |   |   |   |   |
| 19 | 37 Private          | 60548 HS-grad       | 9 Widowed             | Machine-op-inspt | Unmarried     | White              | Female | 0           | 0 20 area_A | <=50K |   |   |   |   |
| 20 | 40 Private          | 85019 Doctorate     | 16 Married-civ-spouse | Prof-specialty   | Husband       | Asian-Pac-Islander | Male   | 0           | 0 45 ?      | >50K  |   |   |   |   |
| 21 | 34 Private          | 107914 Bachelors    | 13 Married-civ-spouse | Tech-support     | Husband       | White              | Male   | 0           | 0 47 area_A | >50K  |   |   |   |   |
| 22 | 34 Private          | 238588 Some-college | 10 Never-married      | Other-service    | Own-child     | Black              | Female | 0           | 0 35 area_A | <=50K |   |   |   |   |
| 23 | 72 ?                | 132015 7th-8th      | 4 Divorced            | ?                | Not-in-family | White              | Female | 0           | 0 6 area_A  | <=50K |   |   |   |   |

图 7-3 数据集的内容

图 7-3 中, 每一行都有 15 列, 代表一个人的 15 个数据属性。每个属性的意义及取值见表 7-2。

表 7-2 数据集字段的含义

| 列 | 字段                      | 取值   |
|---|-------------------------|--|
| A | 年龄 (age)                | 连续值  |
| B | 工作类别 (workclass)        | Private (私企)、Self-emp-not-inc (自由职业)、Self-emp-inc (雇主)、Federal-gov (联邦政府)、Local-gov (地方政府)、State-gov (州政府)、Without-pay (没有工资)、Never-worked (无业)  |
| C | 权重值 (fnlwgt)            | 连续值  |
| D | 教育 (education)          | Bachelors (学士)、Some-college、11th、HS-grad (高中)、Prof-school (教授)、Assoc-acdm、Assoc-voc、9th、7th-8th、12th、Masters (硕士)、1st-4th、10th、Doctorate (博士)、5th-6th、Preschool (学前班)  |
| E | 受教育年限 (education_num)   | 连续值  |
| F | 婚姻状况 (marital_status)   | Married-civ-spouse (已婚)、Divorced (离婚)、Never-married (未婚)、Separated (分居)、Widowed (丧偶)、Married-spouse-absent (已婚配偶缺席)、Married-AF-spouse (再婚)   |
| G | 职业 (occupation)         | Tech-support (技术支持)、Craft-repair (工艺修理)、Other-service (其他服务)、Sales (销售)、Exec-managerial (行政管理)、Prof-specialty (专业教授)、Handlers-cleaners (操作工人清洁工)、Machine-op-inspct (机器操作)、Adm-clerical (ADM 职员)、Farming-fishing (农业捕鱼)、Transport-moving (运输搬家)、Priv-house-serv (家庭服务)、Protective-serv (保安服务)、Armed-Forces (武装部队) |
| H | 关系 (relationship)       | Wife (妻子)、Own-child (自己的孩子)、Husband (丈夫)、Not-in-family (不是家庭成员)、Other-relative (其他亲戚)、Unmarried (未婚)   |
| I | 种族 (race)               | White (白种人)、Asian-Pac-Islander (亚洲太平洋岛民)、Amer-Indian-Eskimo (印度人)、Other (其他)、Black (黑种人)   |
| J | 性别 (gender)             | Female (女性)、Male (男性)  |
| K | 收益 (capital_gain)       | 连续值  |
| L | 损失 (capital_loss)       | 连续值  |
| M | 每周工作时间 (hours_per_week) | 连续值  |
| N | 地区 (native_area)        | area_A、area_B、area_C、area_D、area_E、area_F、area_G、area_H、area_I、Greece、area_K、area_L、area_M、area_N、area_O、area_P、Italy、area_R、Jamaica、area_T、Mexico、area_S、area_U、France、area_W、area_V、Ecuador、area_X、Columbia、area_Y、Guatemala、Nicaragua、area_Z、area_1A、area_1B、area_1C、area_1D、Peru、area_#、area_1G              |
| O | 收入档次 (income_bracket)   | >5 万美元、≤5 万美元  |

## 7.2.2 代码实现：探索性数据分析

探索性数据分析（Exploratory Data Analysis, EDA）是指，对原始样本进行特征分析，找到有价值的特征。常用的方法之一是：用散点图矩阵（scatterplot matrix 或 pairs plot）将样本特征可视化。可视化的结果可用于分析样本分布、寻找单独变量间的关系或发现数据异常情况，有助于指导后续的模型开发。

这里介绍一个工具——seaborn（<https://seaborn.pydata.org>），它能够在 Python 环境中快速创建散点图矩阵，并支持定制化。

下面举一个对数据进行可视化的例子，代码如下：

```
import seaborn as sns
import pandas as pd
import warnings
warnings.simplefilter(action = "ignore", category = RuntimeWarning) #忽略警告（遇到空值的情况，会有警告）

_CSV_COLUMNS = [                                         #CSV 文件的列名
    'age', 'workclass', 'fnlwgt', 'education', 'education_num',
    'marital_status', 'occupation', 'relationship', 'race', 'gender',
    'capital_gain', 'capital_loss', 'hours_per_week', 'native_area',
    'income_bracket'
]
evaldata = r"income_data\adult.data.csv"                  #加载 CSV 文件
df = pd.read_csv(evaldata,names=_CSV_COLUMNS,skiprows=0,encoding = "ISO-8859-1")
#,encoding = "gbk") #,skiprows=1,columns=list('ABCD')

df.loc[df['income_bracket']=='<=50K','income_bracket']=0 #字段转化
df.loc[df['income_bracket']==">>50K','income_bracket']=1 #字段转化
df1 = df.dropna(how='all',axis = 1)                      #数据清洗：将空值数据去掉
sns.pairplot(df1)                                         #生成交叉表
```

运行代码之前，需要先通过 pip install seaborn 命令安装 seaborn 工具。运行之后便会看到其生成的字段交叉图表，如图 7-4 所示。

从图 7-4 中可以看出，seaborn 工具将数值类型的字段以交叉表的方式统一罗列了出来。可以得到以下结果。

- 最终的 income\_bracket（收入档次）与前面的任何单一字段都没有明显的直接联系。
- 从 capital\_gain（收益）字段来看，高收入与低收入人群之间存在着很大的差距。
- 从 hours\_per\_week（每周工作时间）字段来看特别高与特别低的人群都没有很好的年收益。
- 学历低的人群获得高收益的概率非常低。

在实际操作中，可以将其他非数值的字段数值化。对于较大数值的字段也可以取对数，将其控制在统一的取值区间。还可以在图上将某个字段的类别用不同颜色显示，从而方便分析。

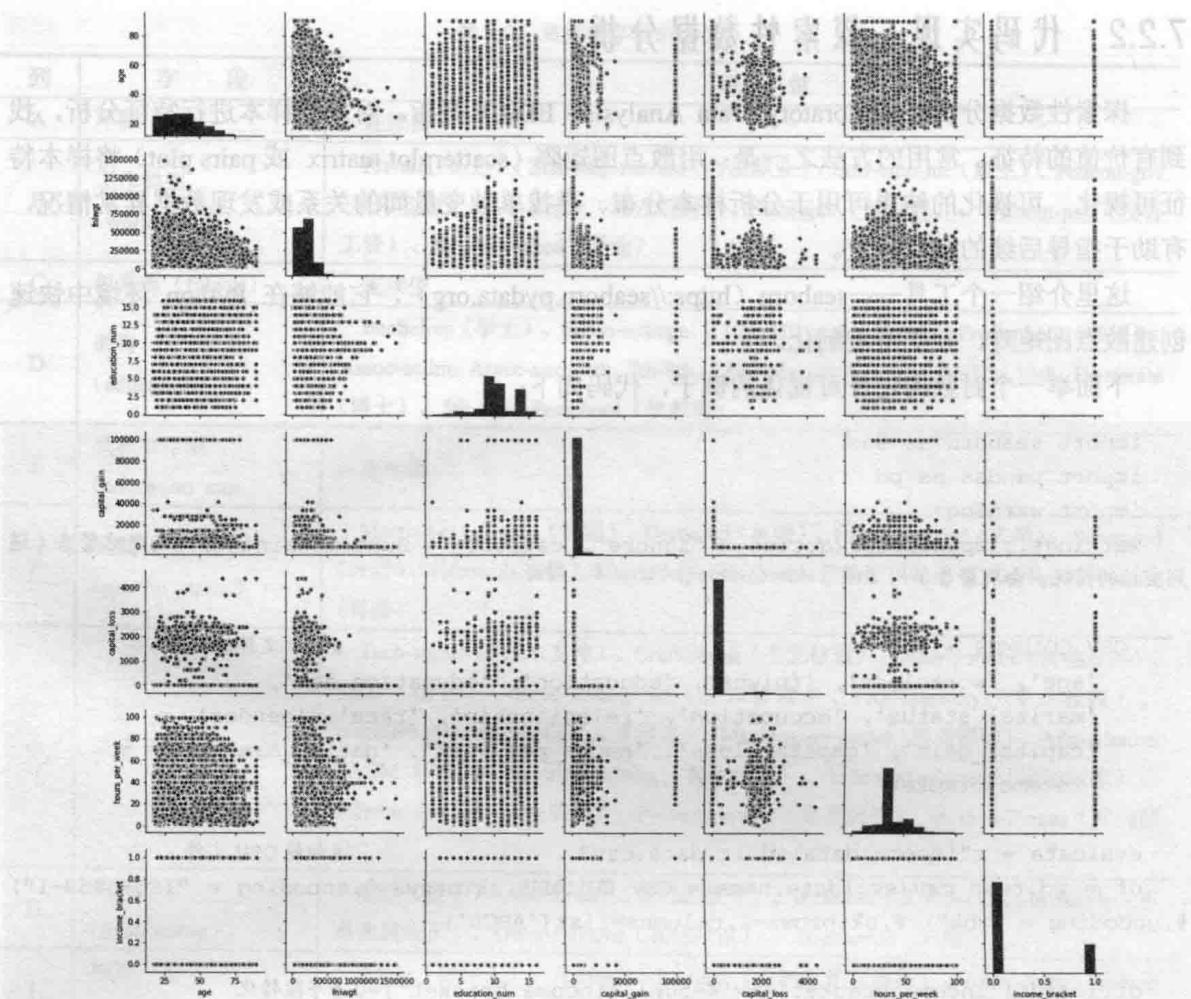


图 7-4 字段交叉图表

更多方法可参考官网或以下教程：

```
https://github.com/WillKoehrsen/Data-Analysis/blob/master/pairplots/Pair%20Plots.ipynb
```

### 7.2.3 认识 wide\_deep 模型

wide\_deep 模型可以理解成是由以下两个模型的输出结果叠加而成的。

- wide 模型是一个线性模型（浅层全连接网络模型）。
- deep 模型是 DNN 模型（深层全连接网络模型）。

#### 1. wide\_deep 模型的训练方式

wide\_deep 模型采用的是联合训练方法。模型的训练误差会同时反馈到线性模型和 DNN 模型中进行参数更新。

## 2. wide\_deep 模型的设计思想

在 wide\_deep 模型中，wide 模型和 deep 模型具有各自不同的分工。

- **wide 模型：**一种浅层模型。它通过大量的单层网络节点，实现对训练样本的高度拟合性。它的缺点是泛化能力很差。
- **deep 模型：**一种深层模型。它通过多层的非线性变化，使模型具有很好的泛化性。它的缺点是拟合度欠缺。

将二者结合起来——用联合训练方法共享反向传播的损失值来进行训练——可以使两个模型综合优点，得到最好的结果。

关于该模型的更多介绍可以参考论文：

<https://arxiv.org/pdf/1606.07792.pdf>

## 7.2.4 部署代码文件

将本书配套资源里的数据集与依赖文件复制到本地代码的同级目录下，如图 7-5 所示。

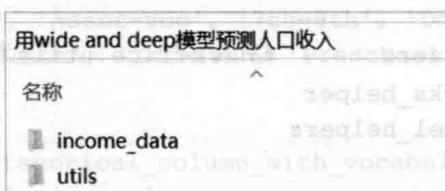


图 7-5 代码文件的结构

图 7-5 中有两个文件夹。其中，文件夹 income\_data 里是数据集（见 7.2.1 小节），文件夹 utils 是本实例代码要依赖的库文件。

文件夹 utils 中的代码文件如图 7-6 所示。

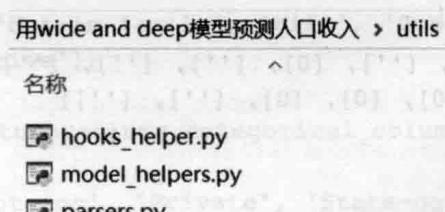


图 7-6 utils 中的代码文件

从图 7-6 中可以看到，utils 文件夹里有 3 个代码文件：

- **hooks\_helper.py：**模型的辅助训练工具。它以钩子函数的方式输出训练过程中的内容。
- **model\_helpers.py：**模型的辅助训练工具。实现早停功能。即在训练过程中，当损失值小于阈值时，自动停止训练。
- **parsers.py：**程序的辅助启动工具。利用它可以方便地设置和解析启动参数。

## 7.2.5 代码实现：初始化样本常量

编写代码引入库模块，并对如下常量进行初始化：

- 样本文件的列名常量。
- 每列样本的默认值。
- 样本集数量。
- 模型前缀。

具体代码如下：

代码 7-1 用 wide\_deep 模型预测人口收入

```

01 import argparse
02 import os
03 import shutil
04 import sys
05
06 import tensorflow as tf
07
08 from utils import parsers      # 引入 office.utils 模块
09 from utils import hooks_helper
10 from utils import model_helpers
11
12 _CSV_COLUMNS = [                # 定义 CSV 文件的列名
13     'age', 'workclass', 'fnlwgt', 'education', 'education_num',
14     'marital_status', 'occupation', 'relationship', 'race', 'gender',
15     'capital_gain', 'capital_loss', 'hours_per_week', 'native_area',
16     'income_bracket'
17 ]
18
19 _CSV_COLUMN_DEFAULTS = [        # 定义每一列的默认值
20     [0], [''], [0], [''], [0], [''], [''], [''],
21     [0], [0], [0], [''], ['']]
22
23 _NUM_EXAMPLES = {              # 定义样本集的数量
24     'train': 32561,
25     'validation': 16281,
26 }
27
28 LOSS_PREFIX = {'wide': 'linear/', 'deep': 'dnn/'} # 定义模型的前缀

```

代码第 28 行是模型前缀，该前缀输出结果会在格式化字符串时用到，在程序功能方面没有任何意义。

## 7.2.6 代码实现：生成特征列

定义函数 build\_model\_columns，该函数以列表的形式返回两个特征列，分别对应于 wide

模型与 deep 模型的特征列输入。

具体代码如下：

### 代码 7-1 用 wide\_deep 模型预测人口收入（续）

```

29 def build_model_columns():
30     """生成 wide 和 deep 模型的特征列集合."""
31     # 定义连续值列
32     age = tf.feature_column.numeric_column('age')
33     education_num = tf.feature_column.numeric_column('education_num')
34     capital_gain = tf.feature_column.numeric_column('capital_gain')
35     capital_loss = tf.feature_column.numeric_column('capital_loss')
36     hours_per_week = tf.feature_column.numeric_column('hours_per_week')
37
38     # 定义离散值列，返回的是稀疏矩阵
39     education = tf.feature_column.categorical_column_with_vocabulary_list(
40         'education', [
41             'Bachelors', 'HS-grad', '11th', 'Masters', '9th', 'Some-college',
42             'Assoc-acdm', 'Assoc-voc', '7th-8th', 'Doctorate', 'Prof-school',
43             '5th-6th', '10th', '1st-4th', 'Preschool', '12th'])
44
45     marital_status =
46         tf.feature_column.categorical_column_with_vocabulary_list(
47             'marital_status', [
48                 'Married-civ-spouse', 'Divorced', 'Married-spouse-absent',
49                 'Never-married', 'Separated', 'Married-AF-spouse', 'Widowed'])
50
51     relationship = tf.feature_column.categorical_column_with_vocabulary_
52         _list(
53             'relationship', [
54                 'Husband', 'Not-in-family', 'Wife', 'Own-child', 'Unmarried',
55                 'Other-relative'])
56
57     workclass = tf.feature_column.categorical_column_with_vocabulary_list(
58         'workclass', [
59             'Self-emp-not-inc', 'Private', 'State-gov', 'Federal-gov',
60             'Local-gov', '?', 'Self-emp-inc', 'Without-pay', 'Never-worked'])
61
62     # 将所有职业名称用 hash 算法散列成 1000 个类别
63     occupation = tf.feature_column.categorical_column_with_hash_bucket(
64         'occupation', hash_bucket_size=1000)
65
66     # 将连续值特征列转化为离散值特征列
67     age_buckets = tf.feature_column.bucketized_column(
68         age, boundaries=[18, 25, 30, 35, 40, 45, 50, 55, 60, 65])
69     # 定义基础特征列
70     base_columns = [

```

```

70     education, marital_status, relationship, workclass, occupation,
71     age_buckets,
72 ]
73 # 定义交叉特征列
74 crossed_columns = [
75     tf.feature_column.crossed_column(
76         ['education', 'occupation'], hash_bucket_size=1000),
77     tf.feature_column.crossed_column(
78         [age_buckets, 'education', 'occupation'], hash_bucket_size=1000),
79 ]
80
81 # 定义 wide 模型的特征列
82 wide_columns = base_columns + crossed_columns
83
84 # 定义 deep 模型的特征列
85 deep_columns = [
86     age,
87     education_num,
88     capital_gain,
89     capital_loss,
90     hours_per_week,
91     tf.feature_column.indicator_column(workclass), # 将 workclass 列的稀疏矩阵转成 One-hot
92     tf.feature_column.indicator_column(education),
93     tf.feature_column.indicator_column(marital_status),
94     tf.feature_column.indicator_column(relationship),
95     tf.feature_column.embedding_column(occupation, dimension=8), # 用嵌入词 embedding 将散列后的每个类别进行转换
96 ]
97
98 return wide_columns, deep_columns

```

在生成特征列的过程中，多处使用了 `tf.feature_column` 接口。读者可以先将其简单理解成对原始数据的数值变换。`tf.feature_column` 接口的详细使用方法见 7.4 节。

## 7.2.7 代码实现：生成估算器模型

将 wide 模型与 deep 模型一起传入 `DNNLinearCombinedClassifier` 模型进行混合训练。



**提示：**

`DNNLinearCombinedClassifier` 模型是一个混合模型框架，它可以将任意两个模型放在一起混合训练。

具体代码如下：

## 代码 7-1 用 wide\_deep 模型预测人口收入（续）

```

99 def build_estimator(model_dir, model_type):
100     """按照指定的模型生成估算器对象."""
101     wide_columns, deep_columns = build_model_columns()
102     hidden_units = [100, 75, 50, 25]
103     #将 GPU 个数设为 0，关闭 GPU 运算。因为该模型在 CPU 上的运行速度更快
104     run_config = tf.estimator.RunConfig().replace(
105         session_config=tf.ConfigProto(device_count={'GPU': 0}),
106         save_checkpoints_steps=1000)
107
108     if model_type == 'wide':                                #生成带有 wide 模型的估算器对象
109         return tf.estimator.LinearClassifier(
110             model_dir=model_dir,
111             feature_columns=wide_columns,
112             config=run_config)
113     elif model_type == 'deep':                             #生成带有 deep 模型的估算器对象
114         return tf.estimator.DNNClassifier(
115             model_dir=model_dir,
116             feature_columns=deep_columns,
117             hidden_units=hidden_units,
118             config=run_config)
119     else:
120         return tf.estimator.DNNLinearCombinedClassifier(  #生成带有 wide 和
121             model_dir=model_dir,
122             linear_feature_columns=wide_columns,
123             dnn_feature_columns=deep_columns,
124             dnn_hidden_units=hidden_units,
125             config=run_config)

```

## 7.2.8 代码实现：定义输入函数

定义估算器输入函数 `input_fn`，具体步骤如下：

(1) 用 `tf.data.TextLineDataset` 对 CSV 文件进行处理，并将其转成数据集。

(2) 对数据集进行特征抽取、乱序等操作。

(3) 返回一个由样本及标签组成的元组 (`features, labels`)。

第 (3) 步返回的元组的具体内容如下：

- `features` 是字典类型，内部的每个键值对代表一个特征列的数据。
- `labels` 是数组类型。

具体代码如下：

## 代码 7-1 用 wide\_deep 模型预测人口收入（续）

```

126 def input_fn(data_file, num_epochs, shuffle, batch_size): #定义输入函数
127     """估算器的输入函数."""

```

```

128 assert tf.gfile.Exists(data_file), ('%s not found. Please make sure you have run data_download.py and %s set the --data_dir argument to the correct path.' % data_file)
129
130
131
132 def parse_csv(value): #对文本数据进行特征抽取
133     print('Parsing', data_file)
134     columns = tf.decode_csv(value, record_defaults=_CSV_COLUMN_DEFAULTS)
135     features = dict(zip(_CSV_COLUMNS, columns))
136     labels = features.pop('income_bracket')
137     return features, tf.equal(labels, '>50K')
138
139 dataset = tf.data.TextLineDataset(data_file) #创建dataset数据集
140
141 if shuffle: #对数据进行乱序操作
142     dataset = dataset.shuffle(buffer_size=_NUM_EXAMPLES['train'])
143
144 dataset = dataset.map(parse_csv, num_parallel_calls=5)
145 dataset = dataset.repeat(num_epochs) #将数据集重复num_epochs次
146 dataset = dataset.batch(batch_size) #将数据集按照batch_size划分
147 dataset = dataset.prefetch(1)
148 return dataset

```

代码第 128 行，用断言（assert）语句判断样本文件是否存在。



#### 提示：

有关断言语句的更多信息，可以参考《Python 带我起飞——入门、进阶、商业实战》一书的 7.6 节。

代码第 132 行定义了内嵌函数 parse\_csv，用于将每一行的数据转化成特征列。

### 7.2.9 代码实现：定义用于导出冻结图文件的函数

定义函数 export\_model，用于导出估算器模型的冻结图文件。具体步骤如下：

- (1) 定义一个 feature\_spec 对象，对输入格式进行转化。
- (2) 用函数 tf.estimator.export.build\_parsing\_serving\_input\_receiver\_fn 生成函数 example\_input\_fn 用于输入数据。
- (3) 将样本输入函数 example\_input\_fn 与模型路径一起传入估算器的 export\_savedmodel 方法中，生成冻结图（见代码第 167 行）。



#### 提示：

用 export\_savedmodel 方法生成的冻结图可以与 tf.saving 模块配合使用。

export\_savedmodel 方法是通过调用 saved\_model 方法实现具体功能的。

saved\_model 方法是 TensorFlow 中非常有用的生成模型方法，该方法导出的冻结图可以

非常方便地部署到生产环境中。

更详细的介绍请参考本书的12.3节与13.2节。

具体代码如下：

#### 代码7-1 用wide\_deep模型预测人口收入（续）

```

149 def export_model(model, model_type, export_dir):    # 定义函数export_model，  
    用于导出模型  
150     """导出模型  
151  
152     参数：  
153         model：估算器对象  
154         model_type：要导出的模型类型，可选值有"wide"、"deep"或"wide_deep"  
155         export_dir：导出模型的路径  
156     """  
157     wide_columns, deep_columns = build_model_columns()    # 获得列张量  
158     if model_type == 'wide':  
159         columns = wide_columns  
160     elif model_type == 'deep':  
161         columns = deep_columns  
162     else:  
163         columns = wide_columns + deep_columns  
164     feature_spec = tf.feature_column.make_parse_example_spec(columns)  
165     example_input_fn = (  
166         tf.estimator.export.build_parsing_serving_input_receiver_fn(feature_spec  
    ))  
167     model.export_savedmodel(export_dir, example_input_fn)

```

#### 7.2.10 代码实现：定义类，解析启动参数

定义解析启动参数的类WideDeepArgParser，具体过程如下：

- (1) 将类WideDeepArgParser继承于类argparse.ArgumentParser。
- (2) 在类WideDeepArgParser中，添加启动参数“--model\_type”，用于指定程序运行时所支持的模型。
- (3) 在类WideDeepArgParser中，初始化环境参数。其中包括样本文件路径、模型存放路径、迭代次数等。

具体代码如下：

#### 代码7-1 用wide\_deep模型预测人口收入（续）

```

168 class WideDeepArgParser(argparse.ArgumentParser):    # 定义WideDeepArgParser  
    类，用于解析参数  
169     """该类用于在程序启动时的参数解析"""  
170

```

```

171 def __init__(self): # 初始化函数
172     super(WideDeepArgParser, self).__init__(parents=[parsers.BaseParser()])
    # 调用父类的初始化函数
173     self.add_argument(
174         '--model_type', '-mt', type=str, default='wide_deep', # 添加一个启动
        参数——model_type, 默认值为 wide_deep
175         choices=['wide', 'deep', 'wide_deep'], # 定义该参数的可选值
176         help='[default %(default)s] Valid model types: wide, deep, wide_deep.', # 定义启动参数的帮助命令
177         metavar='<MT>')
178     self.set_defaults( # 为其他参数设置默认值
179         data_dir='income_data', # 设置数据样本路径
180         model_dir='income_model', # 设置模型存放路径
181         export_dir='income_model_exp', # 设置导出模型存放路径
182         train_epochs=5, # 设置迭代次数
183         batch_size=40) # 设置批次大小

```

## 7.2.11 代码实现：训练和测试模型

这部分代码实现了一个 trainmain 函数，并在函数体内实现模型的训练及评估操作。在 trainmain 函数体内，具体的代码逻辑如下：

- (1) 对 WideDeepArgParser 类进行实例化，得到对象 parser。
- (2) 用 parser 对象解析程序的启动参数，得到程序中的配置参数。
- (3) 定义样本输入函数，用于训练和评估模型。
- (4) 定义钩子回调函数，并将其注册到估算器框架中，用于输出训练过程中的详细信息。
- (5) 建立 for 循环，并在循环内部进行模型的训练与评估操作，同时输出相关信息。
- (6) 在训练结束后，导出模型的冻结图文件。

具体代码如下：

**代码 7-1 用 wide\_deep 模型预测人口收入（续）**

```

184 def trainmain(argv):
185     parser = WideDeepArgParser() # 实例化 WideDeepArgParser 类,
        用于解析启动参数
186     flags = parser.parse_args(args=argv[1:]) # 获得解析后的参数 flags
187     print("解析的参数为: ", flags)
188
189     shutil.rmtree(flags.model_dir, ignore_errors=True) # 如果模型存在，则删除目录
190     model = build_estimator(flags.model_dir, flags.model_type) # 生成估算器对象
191     # 获得训练集样本文件的路径
192     train_file = os.path.join(flags.data_dir, 'adult.data.csv')
193     test_file = os.path.join(flags.data_dir, 'adult.test.csv') # 获得测试集
        样本文件的路径
194
195     def train_input_fn(): # 定义训练集样本输入函数

```

```

196     return input_fn( #返回输入函数，迭代输入 epochs_between_evals 次，并使用乱序
    后的数据集
197         train_file, flags.epochs_between_evals, True, flags.batch_size)
198
199 def eval_input_fn():                                #定义测试集样本输入函数
200     return input_fn(test_file, 1, False, flags.batch_size) #返回函数指针，用
    于在测试场景下输入样本
201
202 loss_prefix = LOSS_PREFIX.get(flags.model_type, '')#生成带有 loss 前缀的字
    符串
203 train_hook = hooks_helper.get_logging_tensor_hook( #定义钩子回调函数，获
    得训练过程中的状态
204     batch_size=flags.batch_size,
205     tensors_to_log={'average_loss': loss_prefix + 'head/truediv',
206                      'loss': loss_prefix + 'head/weighted_loss/Sum'})
207
208 #按照数据集迭代训练的总次数进行训练
209 for n in range(flags.train_epochs):
210     model.train(input_fn=train_input_fn, hooks=[train_hook])      #调用估算器
        的 train 方法进行训练
211     results = model.evaluate(input_fn=eval_input_fn) #调用 evaluate 进行评估
212     #定义分隔符
213     print('{0:-^60}'.format('evaluate at epoch {0}'.format(n + 1)))
214
215     for key in sorted(results):                                #显示评估结果
216         print('{0}: {1}'.format(key, results[key]))
217     #根据 accuracy 的阈值判断是否需要结束训练
218     if model_helpers.past_stop_threshold(
219         flags.stop_threshold, results['accuracy']):
220         break
221
222     if flags.export_dir is not None:          #根据设置导出冻结图文件
223         export_model(model, flags.model_type, flags.export_dir)

```

代码第 203 行定义了钩子函数，用于显示训练过程中的信息，其中“head/weighted\_loss/Sum”是模型中张量的名称。

因为该模型已经被 TensorFlow 完全封装好，官方并没有提供如何获取内部张量名称的方法。所以，如果想要再输出额外的节点，则需要查看源码，或通过读取检查点文件符号的方式自行查找（通过检查点文件查找张量名称的方法可以参考本书 12.2 节）。

## 7.2.12 代码实现：使用模型

定义 premain 函数，并在该函数内部实现以下步骤。

- (1) 调用模型的 predict 方法，对指定的 CSV 文件数据进行预测。
- (2) 将前 5 条数据的结果显示出来。

具体代码如下：

### 代码 7-1 用 wide\_deep 模型预测人口收入（续）

```

224 def premain(argv):
225     parser = WideDeepArgParser()          #实例化WideDeepArgParser类，用于解析启动参数
226     flags = parser.parse_args(args=argv[1:])    #获得解析后的参数flags
227     print("解析的参数为：", flags)
228     #获得测试集样本文件的路径
229     test_file = os.path.join(flags.data_dir, 'adult.test.csv')
230
231     def eval_input_fn():                  #定义测试集的样本输入函数
232         return input_fn(test_file, 1, False, flags.batch_size)      #该输入函数
233         按照batch_size批次，迭代输入1次，不使用乱序处理
234     model2 = build_estimator(flags.model_dir, flags.model_type)
235
236     predictions = model2.predict(input_fn=eval_input_fn)
237     for i, per in enumerate(predictions):
238         print("csv中第", i, "条结果为：", per['class_ids'])
239         if i==5:
240             break

```

代码第 234 行重新定义了模型 model2，并将模型 model2 的输出路径设为 flags.model\_dir 的值。



#### 提示：

代码第 234 行重新定义了模型部分，也可以改成使用热启动的方式。例如，可以用下列代码替换第 234 行代码：

```
model2 = build_estimator('./temp', flags.model_type, flags.model_dir)
```

## 7.2.13 运行程序

添加代码，实现以下步骤。

(1) 调用 trainmain 函数训练模型。

(2) 调用 premain 函数，用模型来预测数据。

具体代码如下：

### 代码 7-1 用 wide\_deep 模型预测人口收入（续）

```

241 if __name__ == '__main__':      #如果运行当前文件，则模块的名字__name__就会变为__main__
242     tf.logging.set_verbosity(tf.logging.INFO) #设置log级别为INFO。如果想要显示的信息少一些，则可以设置成ERROR
243     trainmain(argv=sys.argv)                #调用trainmain函数，训练模型
244     premain(argv=sys.argv)                 #调用premain函数，使用模型

```

代码运行后，输出以下结果：

```
解析的参数为： Namespace(batch_size=40, data_dir='income_data', epochs_between_evals=1,
.....
-----evaluate at epoch 1-----
accuracy: 0.8220011
accuracy_baseline: 0.76377374
auc: 0.87216777
auc_precision_recall: 0.6999677
average_loss: 0.3862863
global_step: 815
label/mean: 0.23622628
loss: 15.414527
precision: 0.8126649
prediction/mean: 0.23457089
recall: 0.32033283
Parsing income_data\adult.data.csv
Parsing income_data\adult.test.csv
-----evaluate at epoch 2-----
.....
csv 中第 0 条结果为: [0]
csv 中第 1 条结果为: [0]
csv 中第 2 条结果为: [0]
csv 中第 3 条结果为: [1]
.....
```

输出结果中有3部分内容，分别用省略号隔开。

- 第1部分为程序起始的输出信息。
- 第2部分为训练中的输出结果。
- 第3部分为最终的预测结果。

## 7.3 实例32：用弱学习器中的梯度提升树算法预测人口收入

本实例继续预测人口收入。不同的是，用弱学习器中的梯度提升树算法来实现。

### 实例描述

有一个人口收入的数据集，其中记录着每个人的详细信息及收入情况。

现需要训练一个机器学习模型，使得该模型能够找到一个人的详细信息与收入之间的关系。

最终实现：在给定一个人的具体详细信息之后，估算出该人的收入水平。

要求使用梯度提升树算法实现。

本实例将使用 `tf.estimator.BoostedTreesClassifier` (TFBT) 接口来实现梯度提升树的分类算法。该接口属于估算器框架中的一个具体算法的封装，具体用法与 7.2 节非常相似，可直接在 7.2 节代码上进行修改。

### 7.3.1 代码实现：为梯度提升树模型准备特征列

`tf.estimator.BoostedTreesClassifier` 接口目前只支持两种特征列类型：`bucketized_column` 与 `indicator_column`。这两种类型的特征列，在 7.4 节会详细介绍。

在数据预处理阶段，需要对 `tf.estimator.BoostedTreesClassifier` 接口不支持的特征列进行转化。

复制代码文件“7-1 用 wide\_deep 模型预测人口收入.py”到本地，并直接修改 `build_model_columns` 函数。

具体代码如下：

#### 代码 7-2 用梯度提升树模型预测人口收入

```

01 def build_model_columns():
02     """生成 wide 和 deep 模型的特征列集合"""
03     # 定义连续值列
04     age = tf.feature_column.numeric_column('age')
05     education_num = tf.feature_column.numeric_column('education_num')
06     .....
07     tf.feature_column.embedding_column(occupation, dimension=8),
08 ]
09 # 定义 boostedtrees 的特征列
10 boostedtrees_columns = [age_buckets,
11     tf.feature_column.bucketized_column(education_num, boundaries=[4, 5, 7,
12         9, 10, 11, 12, 13, 14, 15]),
13     tf.feature_column.bucketized_column(capital_gain, boundaries=[1000,
14         5000, 10000, 20000, 40000, 50000]),
15     tf.feature_column.bucketized_column(capital_loss, boundaries=[100,
16         1000, 2000, 3000, 4000]),
17     tf.feature_column.bucketized_column(hours_per_week, boundaries=[7, 14,
18         21, 28, 35, 42, 47, 56, 63, 70, 77, 90]),
19     tf.feature_column.indicator_column(workclass), # 将 workclass 列的稀疏矩阵
20     # 转成 one-hot 编码
21     tf.feature_column.indicator_column(education),
22     tf.feature_column.indicator_column(marital_status),
23     tf.feature_column.indicator_column(relationship),
24     tf.feature_column.indicator_column(occupation)
25 ]
26
27 return wide_columns, deep_columns, boostedtrees_columns

```

在转化特征列的过程中，需要将 `education_num`、`capital_gain`、`capital_loss`、`hours_per_week` 这 4 个连续数值的特征列转化成 `bucketized_column` 类型，见代码第 11、12、13、14 行。

### 7.3.2 代码实现：构建梯度提升树模型

下面在 `build_estimator` 函数里，用 `tf.estimator.BoostedTreesClassifier` 接口构建梯度提升树模型。具体代码如下：

### 代码 7-2 用梯度提升树模型预测人口收入（续）

```

22 def build_estimator(model_dir, model_type, warm_start_from=None):
23     """按照指定的模型生成估算器对象."""
24     wide_columns, deep_columns, boostedtrees_columns = build_model_columns()
25     hidden_units = [100, 75, 50, 25]
26     .....
27     elif model_type == 'deep':           #生成带有deep模型的估算器对象
28         return tf.estimator.DNNClassifier(
29             model_dir=model_dir,
30             feature_columns=deep_columns,
31             hidden_units=hidden_units,
32             config=run_config)
33     elif model_type=='BoostedTrees':      #构建梯度提升树模型
34         return tf.estimator.BoostedTreesClassifier(
35             model_dir=model_dir,
36             feature_columns=boostedtrees_columns,
37             n_batches_per_layer = 100,
38             config=run_config)
39     else:
40         .....

```

在 build\_estimator 函数中，构建模型的过程是通过参数 model\_type 来实现的。如果 model\_type 的值是 BoostedTrees，则创建梯度提升树模型。



#### 提示：

如想了解 tf.estimator.BoostedTreesClassifier 接口的参数，可以通过输入命令 help (tf.estimator.BoostedTreesClassifier)，或参考以下官网文档进行查看：

[https://www.tensorflow.org/api\\_docs/python/tf/estimator/BoostedTreesClassifier](https://www.tensorflow.org/api_docs/python/tf/estimator/BoostedTreesClassifier)

### 7.3.3 代码实现：训练并导出梯度提升树模型

本小节代码实现以下两个操作。

- 在 trainmain 函数里修改代码，实现梯度提升树模型的训练过程。
- 在 export\_model 函数中指定需要导出的列，将梯度提升树模型导出。

下面具体介绍。

#### 1. 训练模型

在 trainmain 函数里修改代码，如果 model\_type 的值是 BoostedTrees，则直接训练，不再使用钩子函数。具体代码如下：

### 代码 7-2 用梯度提升树模型预测人口收入（续）

```

41 def trainmain(argv):
42     parser = WideDeepArgParser()    #实例化WideDeepArgParser类，用于解析启动参数

```

```

43 .....代码实现：为梯度提升树（续）人口收入梯度提升树模型实现 S-T 第九步
44 loss_prefix = LOSS_PREFIX.get(flags.model_type, '')#格式化输出 loss 值的前缀
45 train_hook = hooks_helper.get_logging_tensor_hook() #定义钩子函数，用于获得
46 训练过程中的状态
47 batch_size=flags.batch_size,
48 tensors_to_log={'average_loss': loss_prefix + 'head/truediv',
49 'loss': loss_prefix + 'head/weighted_loss/Sum'}
50 #将总迭代数按照 epochs_between_evals 分段，并循环对每段进行训练
51 for n in range(flags.train_epochs):
52     if flags.model_type == 'BoostedTrees':
53         model.train(input_fn=train_input_fn) #不使用钩子函数，直接训练
54     else:
55         model.train(input_fn=train_input_fn, hooks=[train_hook])#用 train
56 方法进行训练
57 results = model.evaluate(input_fn=eval_input_fn) #用 evaluate 方法进行
评估
58 .....

```

## 2. 导出模型

在 export\_model 函数中指定梯度提升树模型的导出列。具体代码如下：

### 代码 7-2 用梯度提升树模型预测人口收入（续）

```

57 def export_model(model, model_type, export_dir): #定义函数export_model，用
于导出模型
58 .....
59 elif model_type == 'deep' :
60     columns = deep_columns
61 elif 'BoostedTrees'==model_type:
62     columns = boostedtrees_columns
63 .....

```

### 7.3.4 代码实现：设置启动参数，运行程序

在 WideDeepArgParser 类中，直接设置默认启动参数为 BoostedTrees，并运行程序。具体代码如下：

### 代码 7-2 用梯度提升树模型预测人口收入（续）

```

64 class WideDeepArgParser(argparse.ArgumentParser): #定义 WideDeepArgParser
类，用于解析参数
65 .....
66     self.add_argument(
67         '--model_type', '-mt', type=str, default='BoostedTrees', #添加一个
启动参数—model_type，默认值为 wide_deep
68         choices=['wide', 'deep', 'wide_deep', "BoostedTrees"], #定义该参
数的可选值

```

```

69      help='[default %(default)s] Valid model types: wide, deep, wide_deep.',
# 定义启动参数的帮助命令
70      metavar='<MT>')
71      .....

```

代码运行后输出结果。以下是迭代 5 次后的训练结果。

```

.....
-----evaluate at epoch 5-----
accuracy: 0.8509305
accuracy_baseline: 0.76377374
auc: 0.90430105
auc_precision_recall: 0.7602789
average_loss: 0.3266305
global_step: 4075
label/mean: 0.23622628
loss: 0.3265387
precision: 0.762292
prediction/mean: 0.24224414
recall: 0.53614146

```

以下是模型的预测结果。

```

解析的参数为: Namespace(batch_size=40, data_dir='income_data', epochs_between_evals=
1, export_dir='income_model_exp', model_dir='income_model', model_type='BoostedTrees',
multi_gpu=False, stop_threshold=None, train_epochs =5)
Parsing income_data\adult.test.csv
csv 中第 0 条结果为: [0]
csv 中第 1 条结果为: [0]
csv 中第 2 条结果为: [0]
csv 中第 3 条结果为: [1]
csv 中第 4 条结果为: [0]
csv 中第 5 条结果为: [0]

```

### 7.3.5 扩展：更灵活的 TFBT 接口

在 TensorFlow 中, contrib 模块中的 TFBT 梯度提升树接口具有更多的功能及更灵活的用法。具体链接如下:

```
https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/boosted\_trees
```

在该链接里, 还有关于 tensorflow.contrib.boosted\_trees 接口的其他实例, 读者可以自行研究。

## 7.4 实例 33：用 feature\_column 模块转换特征列

通过 7.2、7.3 节的实例, 读者可以学会如何使用 feature\_column 模块。然而 feature\_column 模块只能处理张量类型的对象, 这对于使用者来说仍然是个“黑盒”(对其内部的变化过程不清楚)。下面将 feature\_column 模块内部的数值运算过程呈现出来。

## 实例描述

用模拟数据作为输入，调用 `feature_column` 模块的特征列转化功能，实现以下操作，观察特征列的转化效果。

- (1) 用 `feature_column` 模块处理连续值特征列；
- (2) 用 `feature_column` 模块处理离散值特征列；
- (3) 用 `feature_column` 模块将连续值特征列转化成离散值特征列；
- (4) 用 `feature_column` 模块将多个离散值特征列合并生成交叉特征列。

该实例的代码比较零散，每一个代码文件演示一个特征列的变化操作。

### 7.4.1 代码实现：用 `feature_column` 模块处理连续值特征列

连续值类型是 TensorFlow 中最简单、最常见的特征列数据类型。本实例通过 4 个小例子演示连续值特征列常见的使用方法。

#### 1. 显示一个连续值特征列

编写代码定义函数 `test_one_column`。在 `test_one_column` 函数中具体完成了以下步骤：

- (1) 定义一个特征列。
- (2) 将带输入的样本数据封装成字典类型的对象。
- (3) 将特征列与样本数据一起传入 `tf.feature_column.input_layer` 函数，生成张量。
- (4) 建立会话，输出张量结果。

在第（3）步中用 `feature_column` 接口的 `input_layer` 函数生成张量。`input_layer` 函数生成的张量相当于一个输入层，用于往模型中传入具体数据。`input_layer` 函数的作用与占位符定义函数 `tf.placeholder` 的作用类似，都用来建立数据与模型之间的连接。

通过这几个步骤便可以将特征列的内容完全显示出来。该部分内容有助于读者理解 7.2 节实例中估算器框架的内部流程。具体代码如下：

#### 代码 7-3 用 `feature_column` 模块处理连续值特征列

```

01 # 导入 TensorFlow 模块
02 import tensorflow as tf
03
04 # 演示只有一个连续值特征列的操作
05 def test_one_column():
06     price = tf.feature_column.numeric_column('price')          # 定义一个特征列
07
08     features = {'price': [[1.], [5.]]}                      # 将样本数据定义为字典的类型
09     net = tf.feature_column.input_layer(features, [price])   # 传入
10    input_layer 函数，生成张量
11
12    with tf.Session() as sess:                                # 建立会话输出特征
13        tt = sess.run(net)

```

```

13     print(tt)
14
15 test_one_column()

```

因为在创建特征列 price 时只提供了名称“price”（见代码第 6 行），所以在创建字典 features 时，其内部的 key 必须也是“price”（见代码第 8 行）。

定义好函数 test\_one\_column 之后，便可以直接调用它（见代码第 15 行）。整个代码运行之后，显示以下结果：

```

[[1.]
[5.]]

```

结果中的数组来自于代码第 8 行字典对象 features 的 value 值。在第 8 行代码中，将值为 [[1.], [5.]] 的数据传入了字典 features 中。

在字典对象 features 中，关键字 key 的值是“price”，它所对应的值 value 可以是任意的一个数值。在模型训练时，这些值就是“price”属性所对应的具体数据。

## 2. 通过占位符输入特征列

将占位符传入字典对象的值 value 中，实现特征列的输入过程。具体代码如下：

**代码 7-3 用 feature\_column 模块处理连续值特征列（续）**

---

```

16 def test_placeholder_column():
17     price = tf.feature_column.numeric_column('price')      # 定义一个特征列
18     # 生成一个 value 为占位符的字典
19     features = {'price':tf.placeholder(dtype=tf.float64)}  # 生成带有占位符的字典
20     net = tf.feature_column.input_layer(features, [price]) # 传入 input_layer 函数，生成张量
21
22     with tf.Session() as sess:                                # 建立会话输出特征
23         tt = sess.run(net, feed_dict={                         # 以注入机制传入数值
24             features['price']: [[1.], [5.]]                  # 生成转换后的具体列值
25         })
26         print(tt)
27
28 test_placeholder_column()

```

在代码第 19 行，生成了带有占位符的字典对象 features。

代码第 23~25 行，在会话中以注入机制传入数值 [[1.], [5.]]，生成转换后的具体列值。

整个代码运行之后，输出以下结果：

```

[[1.]
[5.]]

```

## 3. 支持多维数据的特征列

在创建特征列时，还可以让一个特征列对应的数据有多维，即在定义特征列时为其指定形状。

**提示：**

特征列中的形状是指单条数据的形状，并非整个数据的形状。

具体代码如下：

**代码 7-3 用 feature\_column 模块处理连续值特征列（续）**

```

29 def test_reshaping():
30     tf.reset_default_graph()
31     price = tf.feature_column.numeric_column('price', shape=[1, 2])#定义特征列，并指定形状
32     features = {'price': [[[1., 2.]], [[5., 6.]]]}      #传入一个三维的数组
33     features1 = {'price': [[3., 4.], [7., 8.]]}          #传入一个二维的数组
34     net = tf.feature_column.input_layer(features, price)    #生成特征列张量
35     net1 = tf.feature_column.input_layer(features1, price)   #生成特征列张量
36     with tf.Session() as sess:                                #建立会话输出特征
37         print(net.eval())
38         print(net1.eval())
39 test_reshaping()

```

在代码第 31 行，在创建 price 特征列时，指定了形状为[1,2]，即 1 行 2 列。

接着用两种方法向 price 特征列注入数据（见代码第 32、33 行）

- 在代码第 32 行，创建字典 features，传入了一个形状为[2,1,2]的三维数组。这个三维数组中的第一维是数据的条数（2 条）；第二维与第三维要与 price 指定的形状[1,2]一致。
- 在代码第 33 行，创建字典 features1，传入了一个形状为[2,2]的二维数组。该二维数组中的第一维是数据的条数（2 条）；第二维代表每条数据的列数（每条数据有 2 列）。

在代码第 34、35 行中，都用 tf.feature\_column 模块的 input\_layer 方法将字典 features 与 features1 注入特征列 price 中，并得到了张量 net 与 net1。

代码运行后，张量 net 与 net1 的输出结果如下：

```

[[1. 2.] [5. 6.]]
[[3. 4.] [7. 8.]]

```

结果输出了两行数据，每一行都是一个形状为[2,2]的数组。这两个数组分别是字典 features、features1 经过特征列输出的结果。

**提示：**

代码第 30 行的作用是将图重置。该操作可以将当前图中的所有变量删除。这种做法可以避免在 Spyder 编译器下多次运行图时产生数据残留问题。

**4. 带有默认顺序的多个特征列**

如果要创建的特征列有多个，则系统默认会按照每个列的名称由小到大进行排序，然后将数据按照约束的顺序输入模型。具体代码如下：

## 代码 7-3 用 feature\_column 模块处理连续值特征列（续）

```

40 def test_column_order():
41     tf.reset_default_graph()
42     price_a = tf.feature_column.numeric_column('price_a') # 定义了3个特征列
43     price_b = tf.feature_column.numeric_column('price_b')
44     price_c = tf.feature_column.numeric_column('price_c')
45
46     features = {                                     # 创建字典传入数据
47         'price_a': [[1.]],
48         'price_c': [[4.]],
49         'price_b': [[3.]],
50     }
51
52     # 生成输入层
53     net = tf.feature_column.input_layer(features, [price_c, price_a,
54                                                 price_b])
55
56     with tf.Session() as sess:                      # 建立会话输出特征
57         print(net.eval())
58 test_column_order()

```

在上面代码中，实现了以下操作。

- (1) 定义了3个特征列（见代码第42、43、44行）。
- (2) 定义了一个字典features，用于具体输入（见代码第46行）。
- (3) 用input\_layer方法创建输入层张量（见代码第53行）。
- (4) 建立会话(session)，输出输入层结果（见代码第55行）。

将程序运行后，输出以下结果：

```
[[1. 3. 4.]]
```

输出的结果为[[1. 3. 4.]]所对应的列，顺序为price\_a、price\_b、price\_c。而input\_layer中的列顺序为price\_c、price\_a、price\_b（见代码第53行），二者并不一样。这表示，输入层的顺序是按照列的名称排序的，与input\_layer中传入的顺序无关。



### 提示：

将input\_layer中传入的顺序当作输入层的列顺序，这是一个非常容易犯的错误。

输入层的列顺序只与列的名称和类型有关（7.4.3小节“5. 多特征列的顺序”中还会讲到列顺序与列类型的关系），与传入input\_layer中的顺序无关。

## 7.4.2 代码实现：将连续值特征列转化成离散值特征列

下面将连续值特征列转化成离散值特征列。

### 1. 将连续值特征按照数值大小分类

用 `tf.feature_column.bucketized_column` 函数将连续值按照指定的阈值进行分段，从而将连续值映射到离散值上。具体代码如下：

#### 代码 7-4 将连续值特征列转化成离散值特征列

```

01 import tensorflow as tf
02
03 def test_numeric_cols_to_bucketized():
04     price = tf.feature_column.numeric_column('price')      # 定义连续值特征列
05
06     # 将连续值特征列转化成离散值特征列，离散值共分为 3 段：小于 3、3~5 之间、大于 5
07     price_bucketized = tf.feature_column.bucketized_column(price,
08     boundaries=[3.])
09
10     features = {                                         # 定义字典类型对象
11         'price': [[2.], [6.]],
12     }
13
14     # 生成输入张量
15     net = tf.feature_column.input_layer(features, [price, price_bucketized])
16
17     with tf.Session() as sess:                         # 建立会话输出特征
18         sess.run(tf.global_variables_initializer())
19         print(net.eval())
20
21 test_numeric_cols_to_bucketized()

```

代码运行后，输出以下结果：

```
[[2. 1. 0. 0.]
 [6. 0. 0. 1.]]
```

输出的结果中有两条数据，每条数据有 4 个元素：

- 第 1 个元素为 `price` 列的具体数值。
- 后面 3 个元素为 `price_bucketized` 列的具体数值。

从结果中可以看到，`tf.feature_column.bucketized_column` 函数将连续值 `price` 按照 3 段来划分（小于 3、3~5 之间、大于 5），并将它们生成 one-hot 编码。

### 2. 将整数值直接映射到 one-hot 编码

如果连续值特征列的数据是整数，则还可以直接用 `tf.feature_column.categorical_column_with_identity` 函数将其映射成 one-hot 编码。

函数 `tf.feature_column.categorical_column_with_identity` 的参数和返回值解读如下。

- 需要传入两个必填的参数：列名称(`key`)、类的总数(`num_buckets`)。其中，`num_buckets` 的值一定要大于 `key` 列中所有数据的最大值。
- 返回值：为 `_IdentityCategoricalColumn` 对象。该对象是使用稀疏矩阵的方式存放转化后

的数据。如果要将该返回值作为输入层传入后续的网络，则需要用 `indicator_column` 函数将其转化为稠密矩阵。

具体代码如下：

#### 代码 7-4 将连续值特征列转化成离散值特征列（续）

```

19 def test_numeric_cols_to_identity():
20     tf.reset_default_graph()
21     price = tf.feature_column.numeric_column('price') # 定义连续值特征列
22
23     categorical_column =
24         tf.feature_column.categorical_column_with_identity('price', 6)
25     one_hot_style = tf.feature_column.indicator_column(categorical_column)
26     features = { # 将值传入定义字典
27         'price': [[2], [4]],
28     }
29     # 生成输入层张量
30     net = tf.feature_column.input_layer(features, [price, one_hot_style])
31     with tf.Session() as sess:
32         sess.run(tf.global_variables_initializer())
33         print(net.eval())
34
35 test_numeric_cols_to_identity()
      price = tf.feature_column.numeric_column('price')

```

代码运行后，输出以下结果：

```

[[2. 0. 0. 1. 0. 0.]
 [4. 0. 0. 0. 0. 1. 0.]]

```

结果输出了两行信息。每行的第 1 列为连续值 `price` 列内容，后面 6 列为 one-hot 编码。

因为在代码第 23 行，将 `price` 列转化为 one-hot 时传入的参数是 6，代表分成 6 类。所以在输出结果中，one-hot 编码为 6 列。

### 7.4.3 代码实现：将离散文本特征列转化为 one-hot 与词向量

离散型文本数据存在多种组合形式，所以无法直接将其转化成离散向量（例如，名字属性可以是任意字符串，但无法统计总类别个数）。

处理离散型文本数据需要额外的一套方法。下面具体介绍。

#### 1. 将离散文本按照指定范围散列的方法

将离散文本特征列转化为离散特征列，与将连续值特征列转化为离散特征列的方法相似，可以将离散文本分段。只不过分段的方式不是比较数值的大小，而是用 `hash` 算法进行散列。

用 `tf.feature_column.categorical_column_with_hash_bucket` 方法可以将离散文本特征按照 `hash` 算法进行散列，并将其散列结果转化为离散值。

该方法会返回一个 `_HashedCategoricalColumn` 类型的张量。该张量属于稀疏矩阵类型，不能

直接输入 `tf.feature_column.input_layer` 函数中进行结果输出，只能用稀疏矩阵的输入方法来运行结果。

具体代码如下：

### 代码 7-5 将离散文本特征列转化为 one-hot 编码与词向量

```

01 import tensorflow as tf
02 from tensorflow.python.feature_column import _LazyBuilder
03
04 #将离散文本按照指定范围散列
05 def test_categorical_cols_to_hash_bucket():
06     tf.reset_default_graph()
07     some_sparse_column =
08         tf.feature_column.categorical_column_with_hash_bucket(
09             'sparse_feature', hash_bucket_size=5) #得到格式为稀疏矩阵的散列特征
10
11     builder = _LazyBuilder({                      #封装为 builder
12         'sparse_feature': [['a'], ['x']],        #定义字典类型对象
13     })
14     id_weight_pair = some_sparse_column._get_sparse_tensors(builder) #获得
15     矩阵的张量
16
17     with tf.Session() as sess:
18         #该张量的结果是一个稀疏矩阵
19         id_tensor_eval = id_weight_pair.id_tensor.eval()
20         print("稀疏矩阵: \n", id_tensor_eval)
21
22         dense_decoded = tf.sparse_tensor_to_dense(id_tensor_eval,
23             default_value=-1).eval(session=sess)      #将稀疏矩阵转化为稠密矩阵
24         print("稠密矩阵: \n", dense_decoded)
25
26 test_categorical_cols_to_hash_bucket()

```

本段代码运行后，会按以下步骤执行：

- (1) 将输入的['a']、['x']使用 hash 算法进行散列。
- (2) 设置散列参数 `hash_bucket_size` 的值为 5。
- (3) 将第 (1) 步生成的结果按照参数 `hash_bucket_size` 进行散列。
- (4) 输出最终得到的离散值 (0~4 之间的整数)。

上面的代码运行后，输出以下结果：

稀疏矩阵：

```
SparseTensorValue(indices=array([[0, 0],
[1, 0]], dtype=int64), values=array([4, 0], dtype=int64), dense_shape=array([2,
1], dtype=int64))
```

稠密矩阵：

```
[[4]
[0]]
```

从最终的输出结果可以看出，程序将字符 a 转化为数值 4；将字符 b 转化为数值 0。将离散文本转化成特征值后，就可以传入模型，并参与训练了。



### 提示：

有关稀疏矩阵的更多介绍可以参考《深度学习之 TensorFlow——入门、原理与进阶实战》一书中的 9.4.17 小节。

## 2. 将离散文本按照指定词表与指定范围混合散列

除用 hash 算法对离散文本数据进行散列外，还可以用词表的方法将离散文本数据进行散列。

用 `tf.feature_column.categorical_column_with_vocabulary_list` 方法可以将离散文本数据按照指定的词表进行散列。该方法不仅可以将离散文本数据用词表来散列，还可以与 hash 算法混合散列。其返回的值也是稀疏矩阵类型。同样不能将返回的值直接传入 `tf.feature_column.input_layer` 函数中，只能用“1. 将离散文本按照指定范围散列”中的方法将其显示结果。

具体代码如下：

代码 7-5 将离散文本特征列转化为 one-hot 编码与词向量（续）

```

24 from tensorflow.python.ops import lookup_ops
25 #将离散文本按照指定词表与指定范围混合散列
26 def test_with_1d_sparse_tensor():
27     tf.reset_default_graph()
28     #混合散列
29     body_style =
30         tf.feature_column.categorical_column_with_vocabulary_list(
31             'name', vocabulary_list=['anna', 'gary', 'bob'], num_oov_buckets=2)
32     #稀疏矩阵
33     #稠密矩阵
34     builder = _LazyBuilder({
35         'name': ['anna', 'gary', 'alsa'],      #定义字典类型对象, value 为稠密矩阵
36     })
37     #稀疏矩阵
38     builder2 = _LazyBuilder({
39         'name': tf.SparseTensor(      #定义字典类型对象, value 为稀疏矩阵
40             indices=((0,), (1,), (2,)),
41             values=('anna', 'gary', 'alsa'),
42             dense_shape=(3,)),
43     })
44
45     id_weight_pair = body_style._get_sparse_tensors(builder) #获得矩阵的张量
46     id_weight_pair2 = body_style._get_sparse_tensors(builder2) #获得矩阵的张量
47
48     with tf.Session() as sess:                                #通过会话输出数据
49         sess.run(lookup_ops.tables_initializer())

```

```

50
51     id_tensor_eval = id_weight_pair.id_tensor.eval()
52     print("稀疏矩阵: \n", id_tensor_eval)
53     id_tensor_eval2 = id_weight_pair2.id_tensor.eval()
54     print("稀疏矩阵 2: \n", id_tensor_eval2)
55
56     dense_decoded = tf.sparse_tensor_to_dense(id_tensor_eval,
57         default_value=-1).eval(session=sess)
58     print("稠密矩阵: \n", dense_decoded)
59 test_with_1d_sparse_tensor()

```

代码第 29、30 行向 `tf.feature_column.categorical_column_with_vocabulary_list` 方法传入了 3 个参数，具体意义如下所示。

- `name`: 代表列的名称，这里的列名就是 `name`。
- `vocabulary_list`: 代表词表，其中词表里的个数就是总的类别数。这里分为 3 类 ('anna', 'gary', 'bob')，对应的类别为 (0,1,2)。
- `num_oov_buckets`: 代表额外的值的散列。如果 `name` 列中的数值不在词表的分类中，则会用 hash 算法对其进行散列分类。这里的值为 2，表示在词表现有的 3 类基础上再增加两个散列类。不在词表中的 `name` 有可能被散列成 3 或 4。

### 提示:

`tf.feature_column.categorical_column_with_vocabulary_list` 方法还有第 4 个参数：`default_value`，该参数默认值为 -1。

如果在调用 `tf.feature_column.categorical_column_with_vocabulary_list` 方法时没有传入 `num_oov_buckets` 参数，则程序将只按照词表进行分类。

在按照词表进行分类的过程中，如果 `name` 中的值在词表中找不到匹配项，则会用参数 `default_value` 来代替。

第 33、38 行代码，用 `_LazyBuilder` 函数构建程序的输入部分。该函数可以同时支持值为稠密矩阵和稀疏矩阵的字典对象。

运行代码，输出以下结果：

```

稀疏矩阵:
SparseTensorValue(indices=array([[0, 0],
[1, 0],
[2, 0]], dtype=int64), values=array([0, 1, 4], dtype=int64),
dense_shape=array([3, 1], dtype=int64))
稀疏矩阵 2:
SparseTensorValue(indices=array([[0, 0],
[1, 0],
[2, 0]], dtype=int64), values=array([0, 1, 4], dtype=int64),
dense_shape=array([3, 1], dtype=int64))

```

稠密矩阵:

```
[ [0]
[1]
[4] ]
```

结果显示了3个矩阵：前两个是稀疏矩阵，最后一个为稠密矩阵。这3个矩阵的值是一样的。具体解读如下。

- 从前两个稀疏矩阵可以看出：在传入原始数据的环节中，字典中的value值可以是稠密矩阵或稀疏矩阵。
- 从第3个稠密矩阵中可以看出：输入数据name列中的3个名字('anna', 'gary', 'alsa')被转化成了(0,1,4)3个值。其中，0与1是来自于词表的分类，4是来自于hash算法的散列结果。



### 提示：

在使用词表时要引入lookup\_ops模块，并且，在会话中要用lookup\_ops.tables\_initializer()对其进行初始化，否则程序会报错。

### 3. 将离散文本特征列转化为one-hot编码

在实际应用中，将离散文本进行散列之后，有时还需要对散列后的结果进行二次转化。下面就来看一个将散列值转化成one-hot编码的例子。

#### 代码7-5 将离散文本特征列转化为one-hot编码与词向量（续）

```
60 #将离散文本转化为one-hot编码特征列
61 def test_categorical_cols_to_onehot():
62     tf.reset_default_graph()
63     some_sparse_column = tf.feature_column.categorical_column_with_hash_bucket(
64         'sparse_feature', hash_bucket_size=5) #定义散列的特征列
65     #转化成one-hot编码
66     one_hot_style = tf.feature_column.indicator_column(some_sparse_column)
67
68     features = {
69         'sparse_feature': [[['a']], [['x']]],
70     }
71     #生成输入层张量
72     net = tf.feature_column.input_layer(features, one_hot_style)
73     with tf.Session() as sess: #通过会话输出数据
74         print(net.eval())
75
76 test_categorical_cols_to_onehot()
```

代码运行后，输出以下结果：

```
[ [0. 0. 0. 0. 1.]
[1. 0. 0. 0. 0.] ]
```

结果中输出了两条数据，分别代表字符“a”“x”在散列后的 one-hot 编码。

#### 4. 将离散文本特征列转化为词嵌入向量

词嵌入可以理解为 one-hot 编码的升级版。它使用多维向量更好地描述词与词之间的关系。下面就来使用代码实现词嵌入的转化。

**代码 7-5 将离散文本特征列转化为 one-hot 编码与词向量（续）**

```

77 #将离散文本转化为 one-hot 编码词嵌入特征列
78 def test_categorical_cols_to_embedding():
79     tf.reset_default_graph()
80     some_sparse_column =
81         tf.feature_column.categorical_column_with_hash_bucket(
82             'sparse_feature', hash_bucket_size=5)           #定义散列的特征列
83     #词嵌入列
84     embedding_col = tf.feature_column.embedding_column(some_sparse_column,
85                                                       dimension=3)
86
87     features = {                                     #生成字典对象
88         'sparse_feature': [['a'], ['x']],
89     }
90
91     #生成输入层张量
92     cols_to_vars = {}
93     net = tf.feature_column.input_layer(features, embedding_col,
94                                         cols_to_vars)
95
96     with tf.Session() as sess:                      #通过会话输出数据
97         sess.run(tf.global_variables_initializer())
98         print(net.eval())
99
100 test_categorical_cols_to_embedding()
```

在词嵌入转化过程中，具体步骤如下：

- (1) 将传入的字符“a”与“x”转化为 0~4 之间的整数。
- (2) 将该整数转化为词嵌入列。

代码第 91 行，将数据字典 features、词嵌入列 embedding\_col、列变量对象 cols\_to\_vars 一起传入输入层 input\_layer 函数中，得到最终的转化结果 net。

代码运行后，输出以下结果：

```
[[ 0.08975066  0.34540504  0.85922384]
 [-0.22819372 -0.34707746 -0.76360196]]
```

从结果中可以看到，每个整数都被转化为 3 个词嵌入向量。这是因为，在调用 tf.feature\_column.embedding\_column 函数时传入的维度 dimension 是 3（见代码第 83 行）。



### 提示：

在使用词嵌入时，系统内部会自动定义指定个数的张量作为学习参数，所以运行之前一定要对全局张量进行初始化（见代码第94行）。本实例显示的值，就是系统内部定义的张量被初始化后的结果。

另外，还可以参照本书7.5节的方式为词向量设置一个初始值。通过具体的数值可以更直观地查看词嵌入的输出内容。

## 5. 多特征列的顺序

在大多数情况下，会将转化好的特征列统一放到input\_layer函数中制作成一个输入样本。input\_layer函数支持的输入类型有以下4种：

- numeric\_column 特征列。
- bucketized\_column 特征列。
- indicator\_column 特征列。
- embedding\_column 特征列。

如果要将7.4.3小节中的hash值或词表散列的值传入input\_layer函数中，则需要先将其转化成indicator\_column类型或embedding\_column类型。

当多个类型的特征列放在一起时，系统会按照特征列的名字进行排序。

具体代码如下：

代码7-5 将离散文本特征列转化为one-hot编码与词向量（续）

```

98 def test_order():
99     tf.reset_default_graph()
100    numeric_col = tf.feature_column.numeric_column('numeric_col')
101    some_sparse_column =
102        tf.feature_column.categorical_column_with_hash_bucket(
103            'asparse_feature', hash_bucket_size=5) #稀疏矩阵，单独放进去会出错
104
105    embedding_col = tf.feature_column.embedding_column(some_sparse_column,
106        dimension=3)
107    #转化为one-hot特征列
108    one_hot_col = tf.feature_column.indicator_column(some_sparse_column)
109    print(one_hot_col.name)          #输出one_hot_col列的名称
110    print(embedding_col.name)       #输出embedding_col列的名称
111    print(numeric_col.name)         #输出numeric_col列的名称
112    features = {                  #定义字典数据
113        'numeric_col': [[3], [6]],
114        'asparse_feature': [['a'], ['x']],
115    }
116
117    #生成输入层张量
118    cols_to_vars = {}

```

```

117     net = tf.feature_column.input_layer(features,
118     [numeric_col, embedding_col, one_hot_col], cols_to_vars)
119
120     with tf.Session() as sess:          #通过会话输出数据
121         sess.run(tf.global_variables_initializer())
122         print(net.eval())
123 test_order()

```

上面代码中构建了 3 个输入的特征列：

- numeric\_column 列。
- embedding\_column 列。
- indicator\_column 列。

其中，embedding\_column 列与 indicator\_column 列由 categorical\_column\_with\_hash\_bucket 方法列转化而来（见代码第 104、106 行）。

代码运行后输出以下结果：

```

asparse_feature_indicator
asparse_feature_embedding
numeric_col
[[-1.0505784 -0.4121129 -0.85744965 0. 0. 0. 0. 1. 3.]
 [-0.2486877 0.5705532 0.32346958 1. 0. 0. 0. 0. 6.]]

```

输出结果的前 3 行分别是 one\_hot\_col 列、embedding\_col 列与 numeric\_col 列的名称。

输出结果的最后两行是输入层 input\_layer 所输出的多列数据。从结果中可以看出，一共有两条数据，每条数据有 9 列。这 9 列数据可以分为以下 3 个部分。

- 第 1 部分是 embedding\_col 列的数据内容（见输出结果的前 3 列）。
- 第 2 部分是 one\_hot\_col 列的数据内容（见输出结果的第 4~8 列）。
- 第 3 部分是 numeric\_col 列的数据内容（见输出结果的最后一列）。
- 这三个部分的排列顺序与其名字的字符串排列顺序是完全一致的（名字的字符串排列顺序为 asparse\_feature\_embedding、asparse\_feature\_indicator、numeric\_col）。

#### 7.4.4 代码实现：根据特征列生成交叉列

在本书 7.2 节中用 tf.feature\_column.crossed\_column 函数将多个单列特征混合起来生成交叉列，并将交叉列作为新的样本特征，与原始的样本数据一起输入模型进行计算。

本小节将详细介绍交叉列的计算方式，以及函数 tf.feature\_column.crossed\_column 的使用方法。具体代码如下：

##### 代码 7-6 根据特征列生成交叉列

```

01 from tensorflow.python.feature_column import _LazyBuilder
02 def test_crossed():
03     a = tf.feature_column.numeric_column('a', dtype=tf.int32, shape=(2,))
04     b = tf.feature_column.bucketized_column(a, boundaries=(0, 1))      #离散值转化

```

```

05     crossed = tf.feature_column.crossed_column([b, 'c'], hash_bucket_size=5)
06                                         #生成交叉列
07
08     builder = _LazyBuilder({
09         'a': tf.constant((( -1., -1.5), (.5, 1.))),
10         'c': tf.SparseTensor(
11             indices=((0, 0), (1, 0), (1, 1)),
12             values=['cA', 'cB', 'cC'],
13             dense_shape=(2, 2)),
14     })
15
16     id_weight_pair = crossed._get_sparse_tensors(builder) #生成输入层张量
17     with tf.Session() as sess2:                            #建立会话 session, 取值
18         id_tensor_eval = id_weight_pair.id_tensor.eval()
19         print(id_tensor_eval)                             #输出稀疏矩阵
20
21         dense_decoded = tf.sparse_tensor_to_dense(id_tensor_eval, default_value
22             =-1).eval(session=sess2)                         #输出稠密矩阵
23
24     test_crossed()

```

代码第5行用 `tf.feature_column.crossed_column` 函数将特征列 b 和 c 混合在一起，生成交叉列。该函数有以下两个必填参数。

- `key`: 要进行交叉计算的列。以列表形式传入（代码中是`[b,'c']`）。
- `hash_bucket_size`: 要散列的数值范围（代码中是 5）。表示将特征列交叉合并后，经过 hash 算法计算并散列成 0~4 之间的整数。



### 提示：

`tf.feature_column.crossed_column` 函数的输入参数 `key` 是一个列表类型。该列表的元素可以是指定的列名称（字符串形式），也可以是具体的特征列对象（张量形式）。

如果传入的是特征列对象，则还要考虑特征列类型的问题。因为 `tf.feature_column.crossed_column` 函数不支持对 `numeric_column` 类型的特征列做交叉运算，所以，如果要对 `numeric_column` 类型的列做交叉运算，则需要用 `bucketized_column` 函数或 `categorical_column_with_identity` 函数将 `numeric_column` 类型转化后才能使用（转化方法见 7.4.2 小节）。

代码运行后，输出以下结果：

```

SparseTensorValue(indices=array([[0, 0],
[0, 1],
[1, 0],
[1, 1],
[1, 2],
[2, 0],
[2, 1],
[2, 2],
[3, 0],
[3, 1],
[3, 2],
[4, 0],
[4, 1],
[4, 2]]),
values=b'cA\ncB\ncC\ncA\ncB\ncC\ncA\ncB\ncC\ncA\ncB\ncC\ncA\ncB\ncC')

```

```
[1, 3]], dtype=int64), values=array([3, 1, 3, 1, 0, 4], dtype=int64),
dense_shape=array([2, 4], dtype=int64)
[[ 3 1 -1 -1] [ 3 1 0 4]]
```

程序运行后，交叉矩阵会将以下两矩阵进行交叉合并。具体计算方法见式（7.1）：

$$\text{cross} \left( \begin{bmatrix} -1 & -1.5 \\ 0.5 & 1 \end{bmatrix}, \begin{bmatrix} cA' & cB' \\ cC' & cD' \end{bmatrix} \right) = \begin{bmatrix} \text{hash('cA', hash(-1)) \% size} & \text{hash('cA', hash(-1.5)) \% size} \\ \text{hash('cB', hash(0.5)) \% size} & \text{hash('cB', hash(1.)) \% size} \end{bmatrix} \quad (7.1)$$

式（7.1）中，size 就是传入 crossed\_column 函数的参数 hash\_bucket\_size，其值为 5，表示输出的结果都在 0~4 之间。

在生成的稀疏矩阵中，[0,2]与[0,3]这两个位置没有值，所以在将其转成稠密矩阵时需要为其加两个默认值“-1”。于是在输出结果的最后一行，显示了稠密矩阵的内容[[ 3 1 -1 -1] [ 3 1 0 4]]。该内容中用两个“-1”进行补位。

## 7.5 实例 34：用 sequence\_feature\_column 接口完成自然语言处理任务的数据预处理工作

本节用 sequence\_feature\_column 接口处理序列特征数据。

### 实例描述

将模拟数据作为输入，用 sequence\_feature\_column 接口的特征列转化功能，生成具有序列关系的特征数据。

该实例属于自然语言处理（NLP）任务中的样本预处理工作（见 8.1.9 小节）。

### 7.5.1 代码实现：构建模拟数据

假设有一个字典，里面只有 3 个词，其向量分别为 0、1、2。

用稀疏矩阵模拟两个具有序列特征的数据 a 和 b。每个数据有两个样本：模拟数据 a 的内容是[2][0,1]。模拟数据 b 的内容是[1][2,0]。

具体代码如下：

#### 代码 7-7 序列特征工程

```
01 import tensorflow as tf
02
03 tf.reset_default_graph()
04 vocabulary_size = 3
05 sparse_input_a = tf.SparseTensor(
06     indices=((0, 0), (1, 0), (1, 1)),           #假设有 3 个词，向量为 0、1、2
07     values=(2, 0, 1),                           #定义一个稀疏矩阵，值为：
08     dense_shape=(2, 2))                         #[2] 只有 1 个序列
09
10 sparse_input_b = tf.SparseTensor(               #[0, 1] 有两个序列
11     indices=((0, 0), (1, 0), (1, 1)),           #定义一个稀疏矩阵，值为：
12     values=(1, 2, 0),                           #[1]
```

```

12     values=(1, 2, 0),           #[2, 0]列) 空工正静风者 T-T 风分
13     dense_shape=(2, 2))

```

代码第5、10行分别用tf.SparseTensor函数创建两个稀疏矩阵类型的模拟数据。

## 7.5.2 代码实现：构建词嵌入初始值

词嵌入过程将字典中的词向量应用到多维数组中。在代码中，定义两套用于映射词向量的多维数组（embedding\_values\_a与embedding\_values\_b），并对其进行初始化。



### 提示：

在实际使用中，对多维数组初始化的值，会被定义成-1~1之间的浮点数。这里都将其初始化成较大的值，是为了在测试时让显示效果更加明显。

具体代码如下：

### 代码7-7 序列特征工程（续）

```

14 embedding_dimension_a = 2
15 embedding_values_a = (          #为稀疏矩阵的3个值(0, 1, 2)匹配词嵌入初始值
16     (1., 2.),                  #id 0
17     (3., 4.),                  #id 1
18     (5., 6.)                   #id 2
19 )
20 embedding_dimension_b = 3
21 embedding_values_b = (          #为稀疏矩阵的3个值(0, 1, 2)匹配词嵌入初始值
22     (11., 12., 13.),          #id 0
23     (14., 15., 16.),          #id 1
24     (17., 18., 19.)           #id 2
25 )
26 #自定义初始化词嵌入
27 def _get_initializer(embedding_dimension, embedding_values):
28     def _initializer(shape, dtype, partition_info):
29         return embedding_values
30     return _initializer

```

## 7.5.3 代码实现：构建词嵌入特征列与共享特征列

使用函数sequence\_categorical\_column\_with\_identity可以创建带有序列特征的离散列。该离散列会将词向量进行词嵌入转化，并将转化后的结果进行离散处理。

使用函数shared\_embedding\_columns可以创建共享列。共享列可以使多个词向量共享一个多维数组进行词嵌入转化。具体代码如下：

```

52 with tf.train.MonitoredSession() as sess:
53     print(global_vars[0].eval(session=sess)) #输出嵌入向量
54     print(global_vars[1].eval(session=sess)) #输出嵌入向量

```

**代码 7-7 序列特征工程（续）**

```

31 categorical_column_a =
32     tf.contrib.feature_column.sequence_categorical_column_with_identity( #带序
33         列的离散列
34         key='a', num_buckets=vocabulary_size)
35 embedding_column_a = tf.feature_column.embedding_column(#将离散列转为词向量
36         categorical_column_a, dimension=embedding_dimension_a,
37         initializer=_get_initializer(embedding_dimension_a,
38         embedding_values_a))
39
40 categorical_column_b =
41     tf.contrib.feature_column.sequence_categorical_column_with_identity(
42         key='b', num_buckets=vocabulary_size)
43 embedding_column_b = tf.feature_column.embedding_column(
44         categorical_column_b, dimension=embedding_dimension_b,
45         initializer=_get_initializer(embedding_dimension_b,
46         embedding_values_b))
47 #共享列
48 shared_embedding_columns = tf.feature_column.shared_embedding_columns(
49         [categorical_column_b, categorical_column_a],
50         dimension=embedding_dimension_a,
51         initializer=_get_initializer(embedding_dimension_a,
52         embedding_values_a))

```

**7.5.4 代码实现：构建序列特征列的输入层**

用函数 `tf.contrib.feature_column.sequence_input_layer` 构建序列特征列的输入层。该函数返回两个张量：

- 输入的具体数据。
- 序列的长度。

具体代码如下：

**代码 7-7 序列特征工程（续）**

```

47 features={                                     #将 a、b 合起来
48     'a': sparse_input_a,
49     'b': sparse_input_b,
50 }
51
52 input_layer, sequence_length =
53     tf.contrib.feature_column.sequence_input_layer(      #定义序列特征列的输入层
54         features,
55         feature_columns=[embedding_column_b, embedding_column_a])
56
57 input_layer2, sequence_length2 =
58     tf.contrib.feature_column.sequence_input_layer(      #定义序列输入层
59         features,
60         feature_columns=[embedding_column_a])

```

```

57     features,
58     feature_columns=shared_embedding_columns)
59 #返回图中的张量(两个嵌入词权重)
60 global_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)
61 print([v.name for v in global_vars])

```

代码第52行，用`sequence_input_layer`函数生成了输入层`input_layer`张量。该张量中的内容是按以下步骤产生的。

(1) 定义原始词向量。

- 模拟数据a的内容是[2][0,1]。
- 模拟数据b的内容是[1][2,0]。

(2) 定义词嵌入的初始值。

- `embedding_values_a`的内容是：[(1., 2.), (3., 4.), (5., 6.)]。
- `embedding_values_b`的内容是：[(11., 12., 13.), (14., 15., 16.), (17., 18., 19.)]。

(3) 将词向量中的值作为索引，去第(2)步的数组中取值，完成词嵌入的转化。

- 特征列`embedding_column_a`：将模拟数据a经过`embedding_values_a`转化后得到[[5., 6.], [0., 0.]] [[1., 2.], [3., 4.]]。
- 特征列`embedding_column_b`：将模拟数据b经过`embedding_values_b`转化后得到[[14., 15., 16.], [0., 0., 0.]] [[17., 18., 19.], [11., 12., 13.]]。



### 提示：

`sequence_feature_column`接口在转化词嵌入时，可以对数据进行自动对齐和补0操作。

在使用时，可以直接将其输出结果输入RNN模型里进行计算。

由于模拟数据a、b中第一个元素的长度都是1，而最大的长度为2。系统会自动以2对齐，将不足的数据补0。

(4) 将`embedding_column_b`和`embedding_column_a`两个特征列传入函数`sequence_input_layer`中，得到`input_layer`。根据7.4.3小节介绍的规则，该输入层中数据的真实顺序为：特征列`embedding_column_a`在前，特征列`embedding_column_b`在后。最终`input_layer`的值为：[[5., 6., 14., 15., 16.], [0., 0., 0., 0., 0.]] [[1., 2., 17., 18., 19.], [3., 4., 11., 12., 13.]]。

代码第61行，将运行图中的所有张量打印出来。可以通过观察TensorFlow内部创建词嵌入张量的情况，来验证共享特征列的功能。

## 7.5.5 代码实现：建立会话输出结果

建立会话输出结果。具体代码如下：

### 代码7-7 序列特征工程（续）

```

62 with tf.train.MonitoredSession() as sess:
63     print(global_vars[0].eval(session=sess))      #输出词向量的初始值
64     print(global_vars[1].eval(session=sess))

```

```

65     print(global_vars[2].eval(session=sess))           #输出序列输入层的内容
66     print(sequence_length.eval(session=sess))
67     print(input_layer.eval(session=sess))      #输出序列输入层的内容
68     print(sequence_length2.eval(session=sess))
69     print(input_layer2.eval(session=sess)) #输出序列输入层的内容
70 }

```

代码运行后，输出以下内容：

(1) 输出 3 个词嵌入张量。第 3 个为共享列张量。

```

['sequence_input_layer/a_embedding/embedding_weights:0',
'sequence_input_layer/b_embedding/embedding_weights:0',
'sequence_input_layer_1/a_b_shared_embedding/embedding_weights:0']

```

(2) 输出词嵌入的初始化值。

```

[[1. 2.]
 [3. 4.]
 [5. 6.]
 [[11. 12. 13.]
 [14. 15. 16.]
 [17. 18. 19.]]
 [[1. 2.]
 [3. 4.]
 [5. 6.]]

```

输出的结果共有 9 行，每 3 行为一个数组：

- 前 3 行是 embedding\_column\_a。
- 中间 3 行是 embedding\_column\_b。
- 最后 3 行是 shared\_embedding\_columns。

(3) 输出张量 input\_layer 的内容。

```

[1 2]
[[[5. 6. 14. 15. 16.] [0. 0. 0. 0. 0.]]
 [[1. 2. 17. 18. 19.] [3. 4. 11. 12. 13.]]]

```

输出的结果第 1 行是原始词向量的大小。后面两行是 input\_layer 的具体内容。

(4) 输出张量 input\_layer2 的内容。

```

[1 2]
[[[5. 6. 3. 4.] [0. 0. 0. 0.]]
 [[1. 2. 5. 6.] [3. 4. 1. 2.]]]

```

模拟数据 sparse\_input\_a 与 sparse\_input\_b 同时使用了共享词嵌入 embedding\_values\_a。每个序列的数据被转化成两个维度的词嵌入数据。

## 7.6 实例 35：用 factorization 模块的 kmeans 接口聚类 COCO 数据集中的标注框

本实例以 kmeans 接口为例，来演示 TensorFlow 中 factorization 模块的用法。

### 实例描述

有一个 JSON 格式文件，其中放置了一个图片数据集的标注信息。该标注的内容是每张图片里物体的位置。

通过编写代码，使用聚类算法，找出这些标注框中最常见的尺寸。

在 8.5 节中有一个 YOLO V3 模型的实例。在那个实例中，需要对原始样本进行预处理，即对所有的标注框做聚类计算，找出样本中最常见的标注框。该聚类算法可以用 factorization 模块的 kmeans 接口来实现。

### 7.6.1 代码实现：设置要使用的数据集

本实例支持两种数据集，可以通过参数 usecoco 进行设置：

- 如果 usecoco 是 1，则使用 COCO 数据集。
- 如果 usecoco 是 0，则使用模拟数据集。



#### 提示：

如果设置了使用 COCO 数据集，则需要下载 COCO 数据集样本，并安装其自带的 API 工具，具体做法见 8.6 节。

具体代码如下：

#### 代码 7-8 聚类 COCO 数据集中的标注框

```

01 import numpy as np
02 import tensorflow as tf
03 import matplotlib.pyplot as plt
04
05 usecoco = 1 #实例的演示方式。如设为 1，则表示使用 coco 数据集

```

### 7.6.2 代码实现：准备带聚类的数据样本

编写代码，进行模拟数据的制作与 COCO 数据的载入。具体代码如下：

#### 代码 7-8 聚类 COCO 数据集中的标注框（续）

```

06 def convert_coco_bbox(size, box):      #计算 box 的长宽和原始图像的长宽比
07     """
08     输入：
09     size: 原始图像大小

```

```

10     box: 标注 box 的信息
11     返回:
12         x、y、w、h 标注 box 和原始图像的比值
13     """
14     dw = 1. / size[0]
15     dh = 1. / size[1]
16     x = (box[0] + box[2]) / 2.0 - 1
17     y = (box[1] + box[3]) / 2.0 - 1
18     w = box[2]
19     h = box[3]
20     x = x * dw
21     w = w * dw
22     y = y * dh
23     h = h * dh
24     return x, y, w, h
25
26 def load_cocoDataset(annfile):          #读取 coco 数据集的标注信息
27     from pycocotools.coco import COCO
28     data = []
29     coco = COCO(annfile)
30     cats = coco.loadCats(coco.getCatIds())
31     coco.loadImgs()
32     base_classes = {cat['id'] : cat['name'] for cat in cats}
33     imgId_catIds = [coco.getImgIds(catIds = cat_ids) for cat_ids in
34     base_classes.keys()]
35     image_ids = [img_id for img_cat_id in imgId_catIds for img_id in
36     img_cat_id]
37     for image_id in image_ids:
38         annIds = coco.getAnnIds(imgIds = image_id)
39         anns = coco.loadAnns(annIds)
40         img = coco.loadImgs(image_id)[0]
41         image_width = img['width']
42         image_height = img['height']
43
44         for ann in anns:
45             box = ann['bbox']
46             bb = convert_coco_bbox((image_width, image_height), box)
47             data.append(bb[2:])
48     return np.array(data)
49
50 if usecoco == 1:                      #根据设置选择数据源
51     dataFile =
52         r"E:\Mask_RCNN-master\cocos2014\annotations\instances_train2014.json"
53     points = load_cocoDataset(dataFile)
54 else:                                #如果不使用 COCO 数据集，则直接随机生成一些数字来进行聚类
55     num_points = 100
56     dimensions = 2
57     points = np.random.uniform(0, 1000, [num_points, dimensions])

```

执行上面代码后，会得到一个形状为[n,2]的数据对象 points（numpy 类型）。在 7.6.4 小节中，points 对象将作为聚类的数据源输入聚类模型中。

### 7.6.3 代码实现：定义聚类模型

通过调用 `tf.contrib.factorization.KMeansClustering` 接口，可以创建一个实现聚类的估算器模型。其初始化函数的具体参数如下：

```
__init__(
    num_clusters,                      #待聚类的个数
    model_dir=None,                     #模型的路径
    initial_clusters=RANDOM_INIT,      #初始化中心点的方法
    distance_metric=SQUARED_EUCLIDEAN_DISTANCE, #评估举例的方法
    random_seed=0,                      #随机值种子
    use_mini_batch=True,                #是否使用小批次处理
    mini_batch_steps_per_iteration=1,   #当使用小批次处理时，运行几次更新一次中心点
    kmeans_plus_plus_num_retries=2,     #当 initial_clusters 使用 kmeans++ 算法时的采样次数
    relative_tolerance=None,           #停止阈值。如果聚类的 loss 变化值小于该值，则停止
训练。如果 use_mini_batch 为 True，则该参数无效
    config=None,                       #与估算器的 config 相同
    feature_columns=None              #可以对指定的某些特征列进行聚类。如果该参数值为 None，则代表按照全部特征列进行聚类
)
```

其中，参数 `initial_clusters` 的取值可以是：

- `RANDOM_INIT`（随机数）。
- `KMEANS_PLUS_PLUS_INIT`（kmeans++算法）。
- 指定的中心点数组。
- 自定义函数。

参数 `distance_metric` 的取值可以是 `SQUARED_EUCLIDEAN_DISTANCE`（欧几里得距离）或 `COSINE_DISTANCE`（夹角余弦距离）。



#### 提示：

如果变量 `mini_batch_steps_per_iteration` 等于 `num_inputs / batch_size`，则理论上程序的聚类结果应该会与 `use_mini_batch` 为 `False` 时的聚类结果相同。但实际上，该聚类方法的精度会略有偏差，这是因为在多线程的处理中，该聚类方法的内部没有加锁。

在以下代码中，首先指定了模型路径，接着定义了 `kmeans`（聚类）模型。在迭代过程中，如果发现模型收敛度小于 0.01，则停止训练。

#### 代码 7-8 聚类 COCO 数据集中的标注框（续）

```
55 num_clusters = 5                      #待聚类的个数
56 #定义聚类的配置文件
57 config=tf.estimator.RunConfig(model_dir='./kmeansmodel',
    save_checkpoints_steps=100)
```

```

58 # 定义聚类模型
59 kmeans = tf.contrib.factorization.KMeansClustering(config= config,
60           num_clusters=num_clusters,
61           use_mini_batch=False, relative_tolerance=0.01)

```

## 7.6.4 代码实现：训练模型

定义输入函数，并用 `kmeans.train` 方法训练模型。用 `kmeans.score` 方法可以得到模型运行的最终分数。具体代码如下：

代码 7-8 聚类 COCO 数据集中的标注框（续）

```

61 def input_fn():                      # 定义输入函数
62     return tf.train.limit_epochs(
63         tf.convert_to_tensor(points, dtype=tf.float32), num_epochs=300)
64 kmeans.train(input_fn)               # 训练模型
65 print("训练结束, score(cost) = {}".format(kmeans.score(input_fn)))

```

代码第 62、63 行，调用 `tf.train` 模块的 `limit_epochs` 方法，并传入数据集的迭代次数（300 次）。将 `limit_epochs` 方法的返回值作为输入函数 `input_fn` 的结果传入 `kmeans.train` 方法里进行训练。这样系统将会按照数据集的遍历次数来训练模型。

还可以在 `kmeans.train` 方法中直接指定迭代次数来控制模型的训练。

`kmeans.train` 方法的定义如下：

```

train(
    input_fn,                         # 输入函数
    hooks=None,                       # 钩子函数，用于显示训练过程中的信息
    steps=None,                        # 训练次数
    max_steps=None,                   # 最大训练次数
    saving_listeners=None            # 在保存模型之前和之后所需要调用的函数
)

```

其中，变量 `steps` 代表单次需要训练的次数，变量 `max_steps` 代表对总训练次数的限制。

## 7.6.5 代码实现：输出图示化结果

编写代码，输出图示化结果：

(1) 用 `kmeans.cluster_centers` 方法可以得到中心点结果。

(2) 用 `kmeans.predict_cluster_index` 方法可以得到输入数据对应的分类结果。

(3) 将中心点与分类结果一起显示出来。

具体代码如下：

代码 7-8 聚类 COCO 数据集中的标注框（续）

```

66 anchors = kmeans.cluster_centers()          # 获取中心点
67
68 box_w = points[:1000, 0]

```

```

69 box_h = points[:1000, 1]
70
71 def show_input_fn(): # 定义输入函数
72     return tf.train.limit_epochs(
73         tf.convert_to_tensor(points[:1000], dtype=tf.float32), num_epochs=1)
74 # 生成聚类结果
75 cluster_indices = list(kmeans.predict_cluster_index(show_input_fn))
76
77 plt.scatter(box_h, box_w, c=cluster_indices) # 图示化显示
78 plt.colorbar()
79 plt.scatter(anchors[:, 0], anchors[:, 1], s=800, c='r', marker='x')
80 plt.show()
81
82 if usecoco == 1: # 打印 COCO 最终结果
83     trueanchors = []
84     for cluster in anchors:
85         trueanchors.append([round(cluster[0] * 416), round(cluster[1] *
86         416)])
86     print("在 416*416 上面, 所聚类的锚点候选框为: ", trueanchors)

```

代码第 73 行, 从 points 对象中取出 1000 个点, 输入 kmeans 模型中进行预测。

代码运行后, 输出结果如图 7-7 所示。

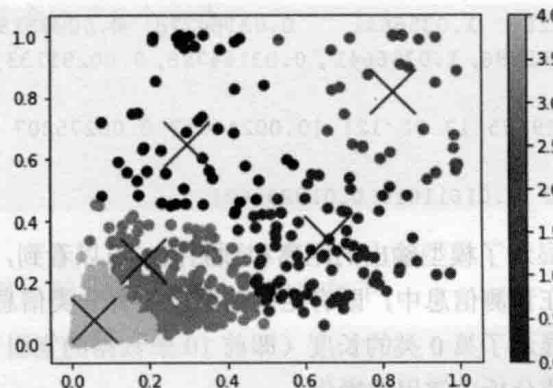


图 7-7 聚类结果

同时, 也输出以下信息:

```
在 416*416 上面, 所聚类的锚点候选框为: [[275.0, 142.0], [124.0, 270.0], [344.0, 341.0], [78.0, 112.0], [24.0, 32.0]]]
```

从输出的结果中, 可以看到聚类中心点的信息。每个中心点的  $x$ 、 $y$  坐标都代表 COCO 数据集中标注框的边长。

在 YOLO V3 模型 (YOLO V3 是一个目标识别模型) 中, 这些聚类中心点结果将被当作锚点候选框参与图像识别过程。有关 YOLO V3 模型的相关知识, 读者先有一个概念即可, 读到 8.6 节自然能够理解。

## 7.6.6 代码实现：提取并排序聚类结果

在数值分析领域，常常要对聚类结果进行分析。最基本的分析操作就是对每个分类的样本进行排序并提取。常用的接口有以下两个。

- 用 kmeans.transform 方法可以得到每个样本离中心点的距离。
- 用 kmeans.predict 方法可以得到分类和距离的全部信息。

具体代码如下：

代码 7-8 聚类 COCO 数据集中的标注框（续）

```

87 distance = list(kmeans.transform(show_input_fn)) #获得每个坐标离中心点的距离
88 predict = list(kmeans.predict(show_input_fn)) #对每个点进行预测
89 print(distance[0],predict[0]) #显示内容
90
91 firstclassdistance= np.array([ p['all_distances'][0] for p in predict if
92 p['cluster_index']==0 ]) #获取第 0 类数据
92 dataindexsort= np.argsort(firstclassdistance) #按照距离排序，并返回索引
93 #显示第 0 类，即前 10 条数据的索引和距离
94 print(len(dataindexsort),dataindexsort[:10],
firstclassdistance[dataindexsort[:10]])

```

代码运行后，输出以下结果：

```

[0.38012558 0.31952286 1.0356641 0.03164728 0.00291133] {'all_distances':
array([0.38012558, 0.31952286, 1.0356641, 0.03164728, 0.00291133], dtype=float32),
'cluster_index': 4}
51 [38 11 7 42 19 29 35 17 20 12] [0.00244057 0.00275207 0.00351655 0.00392741
0.00549358 0.00684953
0.01048928 0.0128122 0.01611018 0.01929462]

```

输出结果的第 1 行，显示了模型输出的距离和预测值。可以看到，在距离信息中，包括该点离所有中心点的距离；在预测信息中，既有距离信息，又有分类信息。

输出结果的第 3 行，显示了第 0 类的长度（即前 10 条数据的索引），以及这 10 条数据的距离。这部分操作是在数值分析中常用的操作。

## 7.6.7 扩展：聚类与神经网络混合训练

TensorFlow 中提供了一个基于 MNIST 训练的聚类实例。其中，先将 MNIST 图片聚类成指定的个数，然后再通过全连接网络进行分类训练。这个实例很有研究价值。

同时，本书也提供了该实例的源码供读者参考，见随书配套资源中的代码文件“7-9 mnistkmeans.py”。更多内容还可以查看以下链接：

<https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/factorization/examples/mnist.py>

## 7.7 实例 36：用加权矩阵分解模型实现基于电影评分的推荐系统

通过调用 TensorFlow 中的 tensorflow.contrib.factorization.WALSModel 接口实现一个加权矩阵分解（WALS）模型，并用该模型实现基于电影评分的推荐系统。

### 实例描述

有一个电影评分数据集，里面包含用户、电影、评分、时间字段。

要求设计模型，并用模型学习该数据的规律，为用户推荐喜欢看的其他电影。

有关加权矩阵分解算法见 7.1.8 小节，具体实现如下。

### 7.7.1 下载并加载数据集

通过以下链接，将电影评论数据集下载到本地：

<http://files.grouplens.org/datasets/movielens/ml-latest-small.zip>

下载之后，将其解压缩到本地代码的同级目录下，并按照以下步骤具体操作。

#### 1. 使用数据集

在电影评论数据集中有以下几个文件：

- links.csv。
- movies.csv。
- ratings.csv。
- README.txt。
- tags.csv。

这里只关心评分文件，即 ratings.csv。其内容如下：

```
userId,movieId,rating,timestamp
1,31,2.5,1260759144
1,1029,3.0,1260759179
```

#### 2. 代码实现：读取数据集，并按照时间排序

将数据加载到内存中，并按照时间对其排序。具体代码如下：

#### 代码 7-10 电影推荐系统

```
01 import os
02
03 DATASET_PATH = 'ml-latest-small'
04 RATINGS_CSV = os.path.join(DATASET_PATH, 'ratings.csv') #指定路径
05
06 import collections
07 import csv
```

```

08
09 Rating = collections.namedtuple('Rating', ['user_id', 'item_id', 'rating',
10   'timestamp'])
11 ratings = list()
12 with open(RATINGS_CSV, newline='') as f: #加载数据
13   reader = csv.reader(f)
14   next(reader) #跳过第一行的字段描述部分
15   for user_id, item_id, rating, timestamp in reader:
16     ratings.append(Rating(user_id, item_id, float(rating), int(timestamp)))
17 ratings = sorted(ratings, key=lambda r: r.timestamp) #排序
18 print('Ratings: {}'.format(len(ratings)))

```

代码运行后，显示以下结果：

```
Ratings: 100,004
```

输出结果中的“100,004”表示数据集的总条数为 100 004 条。

## 7.7.2 代码实现：根据用户和电影特征列生成稀疏矩阵

本小节的具体步骤如下：

- (1) 将用户数据与电影数据单独抽取出来。
- (2) 根据抽取出的数据索引生成字典。
- (3) 按照用户与电影两个维度生成网格矩阵。
- (4) 将该网格矩阵保存为稀疏矩阵。

具体代码如下：

### 代码 7-10 电影推荐系统（续）

```

19 import tensorflow as tf
20 import numpy as np
21
22 users_from_idx = sorted(set(r.user_id for r in ratings), key=int) #获得用户ID
23 users_from_idx = dict(enumerate(users_from_idx)) #生成索引与用户 ID 的正反向字典
24 users_to_idx = dict((user_id, idx) for idx, user_id in
25   users_from_idx.items())
26 print('User Index:', [users_from_idx[i] for i in range(2)])
27 #获得电影的 ID
28 items_from_idx = sorted(set(r.item_id for r in ratings), key=int)
29 items_from_idx = dict(enumerate(items_from_idx)) #生成索引与电影 ID 的正反向字典
30 items_to_idx = dict((item_id, idx) for idx, item_id in
31   items_from_idx.items())
32 print('Item Index:', [items_from_idx[i] for i in range(2)])
33 sess = tf.InteractiveSession() #将用户与电影交叉。填入评分

```

```

33 indices = [(users_to_idx[r.user_id], items_to_idx[r.item_id]) for r in ratings]
34 values = [r.rating for r in ratings]
35 n_rows = len(users_from_idx)
36 n_cols = len(items_from_idx)
37 shape = (n_rows, n_cols)
38 print("User factors shape: ", user_factors.shape) # 用户特征矩阵
39 P = tf.SparseTensor(indices, values, shape) # 生成稀疏矩阵
40 print(P)
41 print('Total values: {}'.format(n_rows * n_cols))

```

代码运行后，输出以下结果：

```

User Index: ['1', '2']
Item Index: ['1', '2']
SparseTensor(indices=Tensor("SparseTensor_11/indices:0", shape=(100004, 2),
                           dtype=int64), values=Tensor("SparseTensor_11/values:0", shape=(100004,),
                           dtype=float32), dense_shape=Tensor("SparseTensor_11/dense_shape:0", shape=(2,), dtype=int64))
Total values: 6,083,286

```

在输出的结果中可以看到：

- 前两行分别显示了用户的 ID 与电影的 ID。
- 第 3 行显示了所生成的稀疏矩阵。
- 最后一行显示了将用户与电影交叉后的矩阵大小为 6,083,286。



### 提示：

程序最终生成的矩阵尺寸非常巨大。对于超大矩阵最好的处理方法是，将其存储为稀疏矩阵。如果以稠密矩阵的形式存放到内存中，则会非常耗资源。

## 7.7.3 代码实现：建立 WALS 模型，并对其进行训练

调用 tensorflow.contrib.factorization.WALSModel 接口，建立 WALS 模型。WALSModel 接口支持分布式训练和正则化处理。具体参数可以使用 help 命令查看。

具体代码如下：

### 代码 7-10 电影推荐系统（续）

```

43 from tensorflow.contrib.factorization import WALSModel
44 k = 10 # 分解后的维度
45 n = 10 # 训练的迭代次数
46 reg = 1e-1 # 正则化的权重
47 # 创建 WALSModel
48 model = WALSModel(
49     n_rows, # 行数

```

```

50     n_cols,          #列数
51     k,              #分解后生成矩阵的维度
52     regularization=reg, #在训练过程中使用的正则化权重
53     unobserved_weight=0)
54
55 row_factors = tf.nn.embedding_lookup(          #从模型中取出行矩阵
56     model.row_factors,
57     tf.range(model._input_rows),
58     partition_strategy="div")
59 col_factors = tf.nn.embedding_lookup(          #从模型中取出列矩阵
60     model.col_factors,
61     tf.range(model._input_cols),
62     partition_strategy="div")
63 #获取稀疏矩阵中原始的行和列的索引
64 row_indices, col_indices = tf.split(P.indices,
65                                     axis=1,
66                                     num_or_size_splits=2)
67 gathered_row_factors = tf.gather(row_factors, row_indices) #根据索引从分解矩阵
68                                         中果敢拍出解
69                                         阵中取出对应的值
70 gathered_col_factors = tf.gather(col_factors, col_indices)
71 #将行和列相乘，得到预测的评分值
72 approx_vals = tf.squeeze(tf.matmul(gathered_row_factors,
73                                     gathered_col_factors,
74                                     adjoint_b=True))
75 P_approx = tf.SparseTensor(indices=P.indices,          #将预测结果组合成稀疏矩阵
76                             values=approx_vals,
77                             dense_shape=P.dense_shape)
78
79 E = tf.sparse_add(P, P_approx * (-1))           #让两个稀疏矩阵相减
80 E2 = tf.square(E)
81 n_P = P.values.shape[0].value
82 rmse_op = tf.sqrt(tf.sparse_reduce_sum(E2) / n_P) #计算 loss 值
83 #定义更新分解矩阵权重的 op
84 row_update_op = model.update_row_factors(sp_input=P)[1]
85 col_update_op = model.update_col_factors(sp_input=P)[1]
86
87 model.initialize_op.run()                         #按指定次数迭代训练
88
89 model.row_update_prep_gramian_op.run()           #训练并更新行（用户）矩阵
90 model.initialize_row_update_op.run()
91 row_update_op.run()
92
93 model.col_update_prep_gramian_op.run()           #训练并更新列（电影）矩阵
94 model.initialize_col_update_op.run()
95 col_update_op.run()

```

```

96
97     print('RMSE: {:.3f}'.format(rmse_op.eval()))    #输出 loss 值
98
99 user_factors = model.row_factors[0].eval()
100 item_factors = model.col_factors[0].eval()
101
102 print('User factors shape:', user_factors.shape)   #输出分解后的矩阵形状
103 print('Item factors shape:', item_factors.shape)

```

代码运行后，输出以下结果：

```

RMSE: 1.999
RMSE: 0.791
.....
RMSE: 0.538
User factors shape: (671, 10)
Item factors shape: (9066, 10)

```

输出结果的最后两行代表分解后的矩阵大小。可以看到，用户矩阵变成了(671, 10)，电影矩阵变成了(9066, 10)。

#### 7.7.4 代码实现：评估 WALS 模型

评估模型的具体步骤如下：

- (1) 找到数据集中评论最多的用户。
- (2) 从该用户评论中取出最后一次的评论记录。
- (3) 根据用户和评论记录中的电影，在分解矩阵中取值。
- (4) 将分解矩阵中的评分与第(2)步评论记录中的评分进行比较，计算出 WALS 模型的准确度。

具体代码如下：

#### 代码 7-10 电影推荐系统（续）

```

104 c = collections.Counter(r.user_id for r in ratings)
105 user_id, n_ratings = c.most_common(1)[0]
106 #找出评论最多的用户
107 print('评论最多的用户 {}: {:.d}'.format(user_id, n_ratings))
108
109 r = next(r for r in reversed(ratings) if r.user_id == user_id and r.rating
110 == 5.0) #找一条评论为 5 的数据
111 print('该用户最后一条 5 分记录: ', r)
112 #在预测模型中取值
113 i = users_to_idx[r.user_id]
114 j = items_to_idx[r.item_id]
115
116 u = user_factors[i] #取出 user 矩阵的值

```

```

117 print('Factors for user {}:\n'.format(r.user_id))
118 print(u)
119
120 v = item_factors[j]
121 print('Factors for item {}:\n'.format(r.item_id))
122 print(v)
123
124 p = np.dot(u, v)
125 print('Approx. rating: {:.3f}, diff={:.3f}, {:.3%}'.format(p, r.rating - p, p/r.rating))

```

代码运行后，输出以下结果：

```

评论最多的用户 547: 2,391
该用户最后一条 5 分记录： Rating(user_id='547', item_id='163949', rating=5.0,
timestamp=1476419239)
Factors for user 547:
[-0.11183977 -0.09171382 -0.10098672 -0.7796077  0.33030528 -0.03237698
 0.48777038  0.4614259 -0.6705016 -0.4126554 ]
Factors for item 163949:
[-0.29128832 -0.23886949 -0.263021  -2.0304952  0.8602844 -0.0843261
 1.270403   1.2017884 -1.7463298 -1.0747647 ]
Approx. rating: 4.740, diff=0.260, 94.791%

```

从输出结果可以看到。WALS 模型的准确率为 94.791%。

## 7.7.5 代码实现：用 WALS 模型为用户推荐电影

用 WALS 模型进行推荐电影的步骤如下：

- (1) 用 WALS 模型计算出该用户对所有电影的评分。
- (2) 从所有的评分中找出该用户在真实数据集中没有评论的电影。
- (3) 按照预测分值排序，
- (4) 将分值最大的前 10 个电影提取出来，推荐给用户。

具体代码如下：

### 代码 7-10 电影推荐系统（续）

```

126 #推荐排名
127 V = item_factors
128 user_P = np.dot(V, u)
129 print('预测出用户所有的评分, 形状为:', user_P.shape)
130 #该用户评论的电影
131 user_items = set(ur.item_id for ur in ratings if ur.user_id == user_id)
132
133 user_ranking_idx = sorted(enumerate(user_P), key=lambda p: p[1],
    reverse=True)
134 user_ranking_raw = [(items_from_idx[j], p) for j, p in user_ranking_idx]

```

```

135 user_ranking = [(item_id, p) for item_id, p in user_ranking_raw if item_id
136     not in user_items] #找到该用户没有评论过的所有电影评分
136
137 top10 = user_ranking[:10] #取出前 10 个
138
139 print('Top 10 items:\n')
140 for k, (item_id, p) in enumerate(top10): #得到该用户喜欢电影的排名
141     print('[{}]\t{} \t{:.2f}'.format(k+1, item_id, p))

```

代码运行后，输出以下结果：

```

预测出用户所有的评分，形状为：(9066,)

Top 10 items:
[1] 1211 6.85
[2] 1273 6.49
.....
[9] 2594 5.63
[10] 501 5.53

```

输出结果的第 1 行，显示该用户所有的评分数值（对应的 9066 个电影评分）。

接着，从未评分的电影中找出了 10 个评分最高的电影。

这些数据将代表用户有可能喜欢的电影，为用户推送过去。

## 7.7.6 扩展：使用 WALS 的估算器接口

TensorFlow 中还提供了一个高级接口——`tensorflow.contrib.factorization.WALSMatrixFactorization`。该接口继承于估算器，其用法与估算器完全一样。更多接口介绍还可以参考以下链接：

[https://www.tensorflow.org/api\\_docs/python/tf/contrib/factorization/WALSMatrixFactorization](https://www.tensorflow.org/api_docs/python/tf/contrib/factorization/WALSMatrixFactorization)

## 7.8 实例 37：用 Lattice 模块预测人口收入

本实例用继续 Lattice 模块预测人口收入。

### 实例描述

有一个人口收入的数据集，其中记录着每个人的详细信息及收入情况。

现需要训练一个机器学习模型，使得该模型能够找到个人的详细信息与收入之间的关系，从而实现在给定一个人的具体详细信息之后估算出该人的收入水平。

要求使用点阵模型（插值查找表算法）实现。

本实例将依次实现校准线性模型、校准点阵模型、随机微点阵模型、集合的微点阵模型的构建与使用，为读者演示具体的实现方法。

由于 Lattice 模块目前不支持 Windows 系统，本实例需要在 Linux 环境下运行。同时，必须保证本机已经安装了 Lattice 模块，要装方式见 7.1.9 小节。

### 7.8.1 代码实现：读取样本，并创建输入函数

本实例使用的人口收入数据集与 7.2 节的内容一样。7.2 节使用的数据集文件是 Windows 编码格式（GBK），而本实例使用的数据集文件是 utf-8 编码格式。该文件可以随书的配套资源中找到。具体步骤如下：

- (1) 将数据集文件“adult.data.csv.txt”“adult.test.csv.txt”放到本地代码的同级目录下。
- (2) 编写代码，用 pandas 模块将 CSV 文件载入。
- (3) 调用 `tf.estimator.inputs.pandas_input_fn` 接口，返回一个估算器输入函数。

具体代码如下：

**代码 7-11 用 Lattice 模块预测收入**

```

01 import os
02 import pandas as pd
03 import six
04 import tensorflow as tf
05 import tensorflow_lattice as tfl
06
07 # 定义数据集目录
08 testdir = "./income_data/adult.test.csv.txt"
09 traindir = "./income_data/adult.data.csv.txt"
10
11 batch_size = 1000 # 定义批次
12
13 # 定义列名，对应于 CSV 文件中的数据列
14 CSV_COLUMNS = [
15     "age", "workclass", "fnlwgt",
16     "education", "education_num",
17     "marital_status", "occupation", "relationship", "race", "gender",
18     "capital_gain", "capital_loss", "hours_per_week", "native_area",
19     "income_bracket"]
20 ]
21
22 _df_data = {}          # 以字典形式存放 CSV 文件的名称和对应的样本内容
23 _df_data_labels = {}   # 以字典形式存放 CSV 文件的名称和对应的标签内容
24
25 # 读入原始 CSV 文件，并转成估算器的输入函数
26 def get_input_fn(file_path, batch_size, num_epochs, shuffle):
27
28     if file_path not in _df_data: # 保证只读取一次 CSV 文件
29         # 读取 CSV 文件，并将样本内容放入 df_data 中
30         _df_data[file_path] = pd.read_csv(tf.gfile.Open(file_path),
31                                         names=CSV_COLUMNS, skipinitialspace=True,
32                                         engine="python", skiprows=1)
33
34     _df_data[file_path] = _df_data[file_path].dropna(how="any", axis=0)

```

```

35     #读取 CSV 文件，并将标签内容放入 _df_data_labels 中
36     _df_data_labels[file_path] = _df_data[file_path][“income_bracket”].apply(
37         lambda x: “>50K” in x).astype(int)
38
39     return tf.estimator.inputs.pandas_input_fn( #返回 pandas 结构的输入函数
40         x=_df_data[file_path],y=_df_data_labels[file_path],
41         batch_size=batch_size,shuffle=shuffle,
42         num_epochs=num_epochs,num_threads=1)

```

## 7.8.2 代码实现：创建特征列，并保存校准关键点

因为 Lattice 模块是通过插值查表法进行计算的，所以需要为其准备好用于插值的数据信息。这个数据信息被称为校准点。

Lattice 模块在对数据预处理时，会根据具体数据，在每个特征列的值域范围内取指定个数的关键点。模型会在每两个关键点之间，做分段的校准计算。

下面编写代码实现以下步骤：

- (1) 预处理特征列。
- (2) 将处理好的特征列按照指定关键点个数计算出校准点。
- (3) 将计算出的校准点保存起来，以便在下一步的运算时使用。



### 提示：

点阵模型可以支持稀疏矩阵张量的处理。经过离散转化后的特征列，不必再转为稠密矩阵，可以直接使用。

具体代码如下：

### 代码 7-11 用 Lattice 模块预测收入（续）

```

43 def create_feature_columns():#创建特征列
44     #离散列
45     gender = tf.feature_column.categorical_column_with_vocabulary_list(
46         “gender”, [“Female”, “Male”])
47     education = tf.feature_column.categorical_column_with_vocabulary_list(
48         “education”, [
49             “Bachelors”, “HS-grad”, “11th”, “Masters”, “9th”, “Some-college”,
50             “Assoc-acdm”, “Assoc-voc”, “7th-8th”, “Doctorate”, “Prof-school”,
51             “5th-6th”, “10th”, “1st-4th”, “Preschool”, “12th”
52         ])
53     marital_status =
54         tf.feature_column.categorical_column_with_vocabulary_list(
55             “marital_status”, [
56                 “Married-civ-spouse”, “Divorced”, “Married-spouse-absent”,
57                 “Never-married”, “Separated”, “Married-AF-spouse”, “Widowed”
58             ])

```

```

58 relationship = tf.feature_column.categorical_column_with_vocabulary_list(
59     "relationship", [
60         "Husband", "Not-in-family", "Wife", "Own-child", "Unmarried",
61         "Other-relative"
62     ])
63 workclass = tf.feature_column.categorical_column_with_vocabulary_list(
64     "workclass", [
65         "Self-emp-not-inc", "Private", "State-gov", "Federal-gov",
66         "Local-gov", "?", "Self-emp-inc", "Without-pay", "Never-worked"
67     ])
68 occupation = tf.feature_column.categorical_column_with_vocabulary_list(
69     "occupation", [
70         "Prof-specialty", "Craft-repair", "Exec-managerial",
71         "Adm-clerical",
72         "Sales", "Other-service", "Machine-op-inspct", "?",
73         "Transport-moving", "Handlers-cleaners", "Farming-fishing",
74         "Tech-support", "Protective-serv", "Priv-house-serv",
75         "Armed-Forces"
76     ])
77 race = tf.feature_column.categorical_column_with_vocabulary_list(
78     "race", [ "White", "Black", "Asian-Pac-Islander",
79         "Amer-Indian-Eskimo",
80         "Other",] )
81 native_area = tf.feature_column.categorical_column_with_vocabulary_list(
82     "native_area", [ "area_A", "area_B", "?", "area_C",
83         "area_D", "area_E", "area_F", "area_G", "area_H", "area_I",
84         "Greece", "area_K", "area_L", "area_M", "area_N", "area_O",
85         "area_P", "Italy", "area_R", "Jamaica", "area_T", "Mexico", "area_S",
86         "area_U", "France", "area_W", "area_V", "Ecuador", "area_X", "Columbia",
87         "area_Y", "Guatemala", "Nicaragua", "area_Z", "area_1A",
88         "area_1B", "area_1C", "area_1D", "Peru",
89         "area_#", "area_1G",] )
90 #连续值列
91 age = tf.feature_column.numeric_column("age")
92 education_num = tf.feature_column.numeric_column("education_num")
93 capital_gain = tf.feature_column.numeric_column("capital_gain")
94 capital_loss = tf.feature_column.numeric_column("capital_loss")
95 hours_per_week = tf.feature_column.numeric_column("hours_per_week")
96 #将处理好的特征列返回
97 return [ age, workclass, education, education_num, marital_status,
98         occupation, relationship, race, gender, capital_gain,
99         capital_loss, hours_per_week, native_area,]
100 #创建校准关键点
101 def create_quantiles(quantiles_dir):

```

```

102 batch_size = 10000 #设置批次
103 lattice_size=2
104 #创建输入函数
105 input_fn = get_input_fn(traindir, batch_size, num_epochs=1, shuffle=False)
106
107 tf1.save_quantiles_for_keypoints( #默认保存 1000 个校准关键点
108     input_fn=input_fn,          #输入函数
109     save_dir=quantiles_dir,      #默认会建立一个文件目录
110     feature_columns=create_feature_columns(),
111     num_steps=None)
112
113 quantiles_dir = "./"          #定义校准点保存路径
114 create_quantiles(quantiles_dir) #创建校准关键点信息
115 a =
116     tf1.load_keypoints_from_quantiles(["age"], quantiles_dir, num_keypoints=10,
117                                         output_min=17.0, output_max=90.0, )
118 with tf.Session() as sess:
119     print("加载 age 的关键点信息: ", sess.run(a))

```

代码第 114 行，调用 `create_quantiles` 函数，按照指定的列进行校准关键点的生成和保存。每个列都默认保存 1000 个点。

代码第 115 行，调用 `tf1.load_keypoints_from_quantiles` 函数，从 `age` 列中取出 10 个关键点并显示出来，用于测试。

代码运行后，输出 10 个校准点的内容。如下：

```
{'age': (array([17., 25., 33., 41., 49., 57., 65., 73., 81., 90.], dtype=float32),
array([17.        , 25.111111 , 33.222222 , 41.333332, 49.444443, 57.555553, 65.666664,
73.777777 , 81.888885, 90.        ], dtype=float32))}
```

同时可以看到，程序在 `quantiles` 文件夹下生成了校准点文件，如图 7-8 所示。

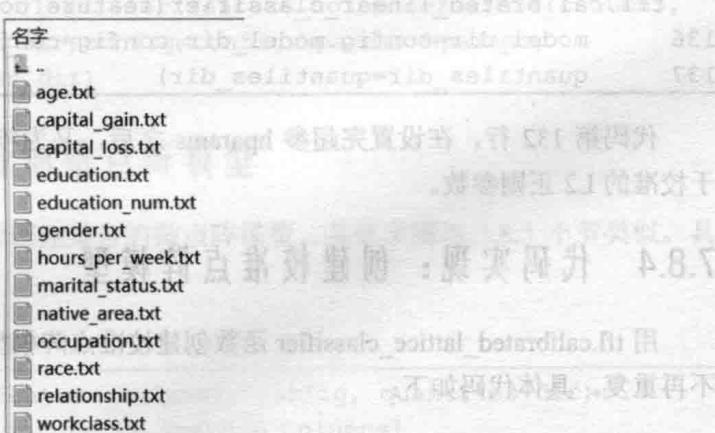


图 7-8 校准点文件

文件名称与代码第 43 行中 `create_feature_columns` 函数所返回的列是一一对应的。

### 7.8.3 代码实现：创建校准线性模型

用 `tfl.calibrated_linear_classifier` 函数返回一个校准线性模型。

该函数需要的两个关键参数的具体实现方法如下。

- `quantiles_dir`（校准关键点目录）：在 7.8.2 小节已经生成（见代码 114 行）。
- `hparams`（超参）：调用 `tfl.CalibratedLinearHParams` 函数，并指定特征列的名称、关键点的取值个数、学习率来生成超参（见代码 129 行）。

具体代码如下：

代码 7-11 用 Lattice 模块预测收入（续）

```

119 #输出超参，用于显示
120 def _pprint_hparams(hparams):
121     print("* hparams=[")
122     for (key, value) in sorted(six.iteritems(hparams.values())):
123         print("\t{}={}".format(key, value))
124     print("]")
125
126 #创建calibrated_linear模型
127 def create_calibrated_linear(feature_columns, config, quantiles_dir):
128     feature_names = [fc.name for fc in feature_columns]
129     hparams = tfl.CalibratedLinearHParams(feature_names=feature_names,
130                                         num_keypoints=200, learning_rate=1e-4)
131     #对部分列中的超参单独赋值
132     hparams.set_feature_param("capital_gain",
133                               "calibration_l2_laplacian_reg", 4.0e-3)
134     _pprint_hparams(hparams)           #输出超参
135
136     return tfl.calibrated_linear_classifier(feature_columns=feature_columns,
137                                              model_dir=config.model_dir, config=config, hparams=hparams,
138                                              quantiles_dir=quantiles_dir)

```

代码第 132 行，在设置完超参 `hparams` 之后，又为年收入列 “`capital_gain`” 单独指定了用于校准的 L2 正则参数。

### 7.8.4 代码实现：创建校准点阵模型

用 `tfl.calibrated_lattice_classifier` 函数创建校准点阵模型。其他步骤与 7.8.3 小节类似，这里不再重复。具体代码如下：

代码 7-11 用 Lattice 模块预测收入（续）

```

138 def create_calibrated_lattice(feature_columns, config, quantiles_dir):
139     feature_names = [fc.name for fc in feature_columns]
140     hparams = tfl.CalibratedLatticeHParams(feature_names=feature_names,
141                                         num_keypoints=200, lattice_l2_laplacian_reg=5.0e-3,

```

```

142     lattice_12_torsion_reg=1.0e-4, learning_rate=0.1,
143     lattice_size=2)
144
145 _pprint_hparams(hparams)
146
147 return tfl.calibrated_lattice_classifier(feature_columns=feature_columns,
148     model_dir=config.model_dir, config=config,
149     hparams=hparams, quantiles_dir=quantiles_dir)

```

## 7.8.5 代码实现：创建随机微点阵模型

调用 `tfl.calibrated_rtl_classifier` 函数，并指定组成微点阵单元的尺寸 `lattice_size` 来创建随机微点阵模型。

具体代码如下：

### 代码 7-11 用 Lattice 模块预测收入（续）

```

150 def create_calibrated_rtl(feature_columns, config, quantiles_dir):
151     feature_names = [fc.name for fc in feature_columns]
152     hparams = tfl.CalibratedRtlHParams(feature_names=feature_names,
153         num_keypoints=200, learning_rate=0.02,
154         lattice_12_laplacian_reg=5.0e-4, lattice_12_torsion_reg=1.0e-4,
155         lattice_size=3, lattice_rank=4, num_lattices=100)
156     #对部分列中的超参单独赋值
157     hparams.set_feature_param("capital_gain", "lattice_size", 8)
158     hparams.set_feature_param("native_area", "lattice_size", 8)
159     hparams.set_feature_param("marital_status", "lattice_size", 4)
160     hparams.set_feature_param("age", "lattice_size", 8)
161     _pprint_hparams(hparams)
162     return tfl.calibrated_rtl_classifier(feature_columns=feature_columns,
163         model_dir=config.model_dir, config=config, hparams=hparams,
164         quantiles_dir=quantiles_dir)

```

## 7.8.6 代码实现：创建集合的微点阵模型

用 `tfl.calibrated_etl_classifier` 函数创建集合的微点阵模型。其他步骤与 7.8.5 小节类似。具体代码如下：

### 代码 7-11 用 Lattice 模块预测收入（续）

```

165 def create_calibrated_etl(feature_columns, config, quantiles_dir):
166     feature_names = [fc.name for fc in feature_columns]
167     hparams = tfl.CalibratedEtlHParams(feature_names=feature_names,
168         num_keypoints=200, learning_rate=0.02,
169         non_monotonic_num_lattices=200, non_monotonic_lattice_rank=2,
170         non_monotonic_lattice_size=2, calibration_12_laplacian_reg=4.0e-3,
171         lattice_12_laplacian_reg=1.0e-5, lattice_12_torsion_reg=4.0e-4)

```

```

172
173     _pprint_hparams(hparams)
174
175     return tf.estimator.calibrated_etc_classifier(feature_columns=feature_columns,
176             model_dir=config.model_dir, config=config, hparams=hparams,
177             quantiles_dir=quantiles_dir)

```

## 7.8.7 代码实现：定义评估与训练函数

因为 Lattice 模块是依赖估算器框架实现的，所以其评估与训练函数也与估算器的用法一致。函数 evaluate\_on\_data 用于评估、函数 train 用于训练。

具体代码如下：

**代码 7-11 用 Lattice 模块预测收入（续）**

```

178 def evaluate_on_data(estimator, data): #用指定数据测试模型
179     name = os.path.basename(data) #获取输入数据的文件夹名称
180
181     #评估模型
182     evaluation = estimator.evaluate(input_fn=get_input_fn( #定义输入函数
183         file_path=data, batch_size=batch_size, num_epochs=1, shuffle=False),
184         feature_columns=feature_columns, name=name)
185     print(" Evaluation on '{}':\t准确率={:.4f}\t平均 loss={:.4f}".format(
186         name, evaluation["accuracy"], evaluation["average_loss"]))
187
188 def evaluate(estimator): #用测试数据集测试模型
189     evaluate_on_data(estimator, traindir)
190     evaluate_on_data(estimator, testdir)
191
192 def train(estimator, train_epochs, showtest = None):
193     if showtest==None: #不显示中间测试信息
194         input_fn =get_input_fn(traindir, batch_size, num_epochs=train_epochs,
195             shuffle=True)
196         estimator.train(input_fn=input_fn) #在训练过程中显示测试信息
197         epochs_trained = 0
198         loops = 0
199         while epochs_trained < train_epochs: #通过调用 estimator.train() 来训练模型
200             loops += 1
201             next_epochs_trained = int(loops * train_epochs / 10.0)
202             epochs = max(1, next_epochs_trained - epochs_trained)
203             epochs_trained += epochs
204             input_fn =get_input_fn(traindir, batch_size, num_epochs=epochs,
205             shuffle=True)
205             estimator.train(input_fn=input_fn)
206             print("Trained for {} epochs, total so far {}:".format(
207                 epochs, epochs_trained))
208             evaluate(estimator)

```

## 7.8.8 代码实现：训练并评估模型

用 for 循环依次生成校准线性模型、校准点阵模型、随机微点阵模型、集合的微点阵模型这 4 个模型，并对其进行训练和评估。

具体代码如下：

代码 7-11 用 Lattice 模块预测收入（续）

```

209 allfeature_columns = create_feature_columns() # 创建特征列
210 modelsfun = [
211     create_calibrated_linear,           # 创建 calibrated_linear 模型函数
212     create_calibrated_lattice,          # 创建 calibrated_lattice 模型函数
213     create_calibrated_rtl,             # 创建 calibrated_rtl 模型函数
214     create_calibrated_etl,             # 创建 calibrated_etl 模型函数
215 ]
216 for modelfun in modelsfun:           # 依次创建函数，对其进行评估
217     print('{0:-^50}'.format(modelfun.__name__)) # 分隔符
218
219     output_dir = "./model_" + modelfun.__name__
220     os.makedirs(output_dir, exist_ok=True)      # 创建模型路径
221     # 创建估算器配置文件
222     config = tf.estimator.RunConfig().replace(model_dir=output_dir)
223     # 创建估算器
224     estimator = modelfun(allfeature_columns, config, quantiles_dir)
225     train(estimator, train_epochs=10)            # 训练模型，迭代 10 次
226     evaluate(estimator)                         # 评估模型

```

代码运行后，输出以下结果：

```

-----create_calibrated_linear-----
* hparams=[
    calibration_bound=False
    .....
    num_keypoints=200
]
Evaluation on 'adult.data.csv.txt':    准确率=0.7593 平均 loss=1.3477
Evaluation on 'adult.test.csv.txt':    准确率=0.7639 平均 loss=1.3297
-----create_calibrated_lattice-----
* hparams=[
    calibration_bound=True
    .....
    num_keypoints=200
]
Evaluation on 'adult.data.csv.txt':    准确率=0.8657 平均 loss=0.2957
Evaluation on 'adult.test.csv.txt':    准确率=0.8659 平均 loss=0.2971
-----create_calibrated_rtl-----
* hparams=[
    calibration_bound=True
    .....

```

```

rtl_seed=12345
]
Evaluation on 'adult.data.csv.txt':    准确率=0.8647 平均 loss=0.3071
Evaluation on 'adult.test.csv.txt':    准确率=0.8663 平均 loss=0.3081
-----create_calibrated_etl-----
* hparams=[
    calibration_bound=True
    .....
    num_keypoints=200
]
Evaluation on 'adult.data.csv.txt':    准确率=0.8765 平均 loss=0.2730
Evaluation on 'adult.test.csv.txt':    准确率=0.8728 平均 loss=0.2815

```

输出结果被横线分隔符分成 4 段。每一段显示了对应模型的超参与训练结果。

每个模型中超参的意义不再展开介绍。如需要深入了解的读者，可以参考 GitHub 网站上的文档介绍。链接如下：

<https://github.com/tensorflow/lattice/blob/master/g3doc/tutorial/index.md>

## 7.8.9 扩展：将点阵模型嵌入神经网络中

在 7.1.9 小节中，我们介绍了点阵模型还可以与神经网络结合使用。本实例就来演示一下具体使用方法。

### 1. 特征列处理

如果把点阵模型当作一个层来处理，则需要将其输出结果转化为稠密矩阵，才可以与下一层神经网络连接。

修改代码文件“7-11 用 Lattice 模块预测收入.py”中的 create\_feature\_columns 函数，生成稠密矩阵。具体代码片段如下：

代码 7-12 Lattice 模块 DNN 结合（片段）

```

01 def create_feature_columns():#创建特征列
.....
03 #连续值列
04 age = tf.feature_column.numeric_column("age")
05 education_num = tf.feature_column.numeric_column("education_num")
06 capital_gain = tf.feature_column.numeric_column("capital_gain")
07 capital_loss = tf.feature_column.numeric_column("capital_loss")
08 hours_per_week = tf.feature_column.numeric_column("hours_per_week")
09
10 #转化为稠密矩阵
11 dnnfeature =
[age,education_num,capital_gain,capital_loss,hours_per_week,
12     tf.feature_column.indicator_column(gender),
13     tf.feature_column.indicator_column(education),
14     tf.feature_column.indicator_column(marital_status),

```

```

15     for i in range(len(fc)):
16         tf.feature_column.indicator_column(relationship),
17         tf.feature_column.indicator_column(workclass),
18         tf.feature_column.indicator_column(occupation),
19         tf.feature_column.indicator_column(race),
20         tf.feature_column.indicator_column(native_area),
21     ]
22     #将处理好的特征列返回
23     return [age, workclass, education, education_num, marital_status,
24             occupation, relationship, race, gender, capital_gain,
25             capital_loss, hours_per_week, native_area,],dnnfeature

```

代码第 11 行，统一将稀疏矩阵类型的离散列转化为稠密矩阵，并将所有需要处理的列放到 dnnfeature 列表中返回。

## 2. 保存校准关键点

因为点阵模型在进行运算时需要特征列所对应的校准关键点信息，所以，需要为稠密矩阵类型的特征列生成对应的校准关键点信息。

修改代码文件“7-11 用 Lattice 模块预测收入.py”中的 create\_quantiles 函数，将 create\_feature\_columns 函数返回的新特征列传入。

具体代码片段如下：

代码 7-12 Lattice 模块结合 DNN（片段）

```

25 def create_quantiles(quantiles_dir):
26     batch_size = 10000          #设置批次
27     _,fc = create_feature_columns()
28
29     #创建输入函数
30     input_fn = get_input_fn(traindir, batch_size, num_epochs=1, shuffle=False)
31
32     tfl.save_quantiles_for_keypoints(      #默认保存 1000 个校准关键点
33         input_fn=input_fn,
34         save_dir=quantiles_dir,           #默认会建立一个文件目录
35         feature_columns=fc,
36         num_steps=None)
37
38 quantiles_dir = "./dnnquant"
39 create_quantiles(quantiles_dir)          #创建校准关键点信息

```

代码运行后，会在本地路径 dnnquant/quantiles 下生成校准关键点的信息文件，如图 7-9 所示。

图 7-9 与图 7-8 相比，特征列的文件名称发生了变化。所有散列类型的特征列都在名字后面加了一个“\_indicator”（例如，图 7-8 中的 race 对应到图 7-9 中的名字为 race\_indicator），这表示该列是 one\_hot 编码类型。

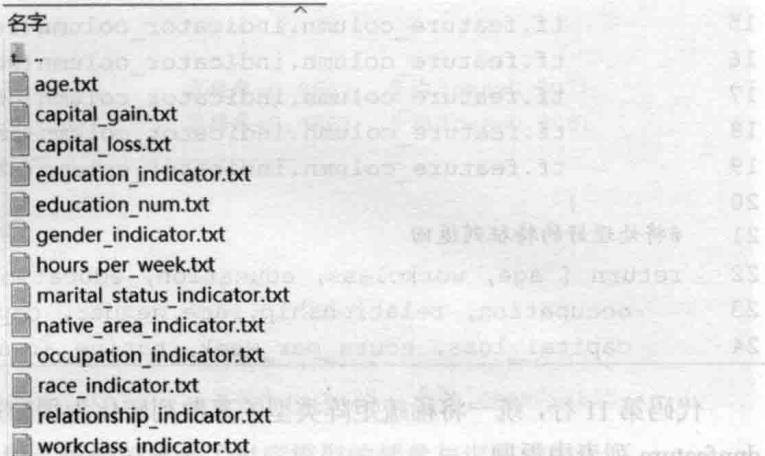


图 7-9 校准点文件  
（感兴趣的读者，可以参考 GitHub 上的点阵关键代码示例）

### 3. 创建点阵与神经网络的混合模型

搭建混合模型的方法，其实就是实现一个估算器的自定义模型。在模型里，除需要实现 DNN 模型外，还需要实现一个点阵的校准层。具体步骤如下：

(1) 对特征列进行循环，依次将列名与该列的输入张量放入字典，来构造输入数据 a（见代码第 54 行）。

(2) 将输入数据 a、超参 hparams、校准关键点路径 quantiles\_dir 一起传入 tfl.input\_calibration\_layer\_from\_hparams 函数，完成点阵层的计算。

在创建了点阵层之后便是实现 DNN 网络。具体包括：创建 DNN 网络、计算 loss 值、定义优化器等操作。

具体代码片段如下：

#### 代码 7-12 Lattice 模块结合 DNN（片段）

```

40 def create_calibrated_dnn(feature_columns, config, quantiles_dir):
41     feature_names = [fc.name for fc in feature_columns[1]]
42     print(feature_names)
43     print([fc.name for fc in feature_columns[0]])
44     hparams = tfL.CalibratedHParams(feature_names=feature_names,
45         num_keypoints=200, learning_rate=1.0e-3, calibration_output_min=-1.0,
46         calibration_output_max=1.0,
47         nodes_per_layer=10,                                #每层 10 个节点
48         layers=2,                                         #包括输出层，一共两层
49     )
50     _pprint_hparams(hparams)
51     def _model_fn(features, labels, mode, params): #构建含有点阵的神经网络模型
52         hparams = params
53         a = {}                                         #构建点阵层输入
54         for fc in feature_columns[1]:
55             a[fc.name] = tf.feature_column.input_layer(features,[fc])
56
57     #返回点阵层结果

```

```

58     (output, _, _, regularization) =
59         tfl.input_calibration_layer_from_hparams(
60             a, hparams, quantiles_dir)
61
62     #全连接隐藏层
63     for _ in range(hparams.layers - 1):
64         output = tf.layers.dense(
65             inputs=output, units=hparams.nodes_per_layer,
66             activation=tf.sigmoid)
67
68     #最终分类的输出层
69     logits = tf.layers.dense(inputs=output, units=1)
70     predictions = tf.reshape(tf.sigmoid(logits), [-1])
71
72     #计算损失值 loss
73     loss_no_regularization = tf.losses.log_loss(labels, predictions)
74     loss = loss_no_regularization
75     if regularization is not None:
76         loss += regularization
77     optimizer =
78         tf.train.AdamOptimizer(learning_rate=hparams.learning_rate)
79     train_op = optimizer.minimize(
80         loss,
81         global_step=tf.train.get_global_step(),
82         name="calibrated_dnn_minimize")
83
84     eval_metric_ops = {                         #用于输出中间结果
85         "accuracy": tf.metrics.accuracy(labels, predictions),
86         "average_loss": tf.metrics.mean(loss_no_regularization),
87     }
88     return tf.estimator.EstimatorSpec(mode, predictions, loss, train_op,
89                                         eval_metric_ops)
90     return tf.estimator.Estimator(           #调用构建好的模型生成估算器
91         model_fn=_model_fn, model_dir=config.model_dir,
92         config=config, params=hparams)

```

代码第 58 行，在用 `tfl.input_calibration_layer_from_hparams` 函数创建校准层的过程中，只使用了 4 个返回值中的两个。被忽略的两个返回值是：

- 以列表形式存在的列名称。
- 对数值映射的 OP（也是列表类型）。在单独训练校准模型时，它用于强制对数据进行单调处理。

因为该校准层是模型的中间层，所以不需要用到这两个返回值。

#### 4. 运行程序

完整代码见随书资源代码“7-12 Lattice 模块结合 DNN.py”。将该代码运行后，输出以下结果：

```

* hparams=[
    calibration_bound=False
    .....
    num_keypoints=200
]
Evaluation on 'adult.data.csv.txt':      准确率=0.7592 平均 loss=0.3821
Evaluation on 'adult.test.csv.txt':      准确率=0.7638 平均 loss=0.3545

```

显示的结果是模型超参详情和迭代训练 10 次之后的准确率。

本实例的重点是演示 Lattice 点阵模型的实现方法。如要得到更好的训练效果，还需要继续对超参进行调优。

## 7.9 实例 38：结合知识图谱实现基于电影的推荐系统

知识图谱（Knowledge Graph, KG）可以理解成一个知识库，用来存储实体与实体之间的关系。知识图谱可以为机器学习算法提供更多的信息，帮助模型更好地完成任务。

在推荐算法中融入电影的知识图谱，能够将没有任何历史数据的新电影精准地推荐给目标用户。

### 实例描述

现有一个电影评分数据集和一个电影相关的知识图谱。电影评分数据集里包含用户、电影及评分；电影相关的知识图谱中包含电影的类型、导演等属性。

要求：从知识图谱中找出电影间的潜在特征，并借助该特征及电影评分数据集，实现基于电影的推荐系统。

本实例使用了一个多任务学习的端到端框架 MKR。该框架能够将两个不同任务的低层特征抽取出来，并融合在一起实现联合训练，从而达到最优的结果。有关 MKR 的更多介绍可以参考以下链接：

<https://arxiv.org/pdf/1901.08907.pdf>

### 7.9.1 准备数据集

在 <https://arxiv.org/pdf/1901.08907.pdf> 的相关代码链接中有 3 个数据集：图书数据集、电影数据集和音乐数据集。本例使用电影数据集，具体链接如下：

<https://github.com/hwwang55/MKR/tree/master/data/movie>

该数据集中一共有 3 个文件。

- item\_index2entity\_id.txt：电影的 ID 与序号。具体内容如图 7-10 所示，第 1 列是电影 ID，第 2 列是序号。
- kg.txt：电影的知识图谱。图 7-11 中显示了知识图谱的 SPO 三元组（Subject-Predicate-Object），第 1 列是电影 ID，第 2 列是关系，第 3 列是目标实体。

- ratings.dat: 用户的评分数据集。具体内容如图 7-12 所示, 列与列之间用“::”符号进行分割, 第 1 列是用户 ID, 第 2 列是电影 ID, 第 3 列是电影评分, 第 4 列是评分时间(可以忽略)。

| item_index2entity_id.txt |
|--------------------------|
| 1 0                      |
| 2 1                      |
| 3 2                      |
| 4 3                      |
| 5 4                      |
| 8 5                      |
| 10 6                     |
| 11 7                     |

图 7-10 item\_index2entity\_id.txt

| kg.txt                       |
|------------------------------|
| 749 film.film.writer 2347    |
| 1410 film.film.language 2348 |
| 1037 film.film.language 2348 |
| 1888 film.film.writer 2349   |
| 1391 film.film.language 2348 |
| 6 1437 film.film.genre 2350  |
| 437 film.film.language 2351  |
| 766 film.film.genre 2350     |

图 7-11 kg.txt

| ratings.dat           |
|-----------------------|
| 1::1193::5::978300760 |
| 1::661::3::978302109  |
| 1::914::3::978301968  |
| 1::3488::4::978300275 |
| 1::2355::5::978824291 |
| 1::1197::3::978302268 |
| 1::1287::5::978302039 |
| 1::2804::5::978300719 |

图 7-12 kg.txt ratings.dat

## 7.9.2 预处理数据

数据预处理主要是对原始数据集中的有用数据进行提取、转化。该过程会生成两个文件。

- kg\_final.txt: 转化后的知识图谱文件。将文件 kg.txt 中的字符串类型数据转成序列索引类型数据, 如图 7-13 所示。
- ratings\_final.txt: 转化后的用户评分数据集。第 1 列将 ratings.dat 中的用户 ID 变成序列索引。第 2 列没有变化。第 3 列将 ratings.dat 中的评分按照阈值 5 进行转化, 如果评分大于等于 5, 则标注为 1, 表明用户对该电影感兴趣。否则标注为 0, 表明用户对该电影不感兴趣。具体内容如图 7-14 所示。

| kg_final.txt  |
|---------------|
| 749 0 2347    |
| 1410 1 2348   |
| 1037 1 2348   |
| 1888 0 2349   |
| 1391 1 2348   |
| 6 1437 2 2350 |
| 437 1 2351    |
| 766 2 2350    |

图 7-13 kg\_final.txt

| ratings_final.txt |
|-------------------|
| 0 0 1             |
| 0 1028 1          |
| 0 1159 1          |
| 0 1802 1          |
| 0 1551 1          |
| 0 1888 1          |
| 0 1684 1          |
| 0 1314 1          |

图 7-14 ratings\_final.txt

该部分代码在文件“7-13 preprocess.py”中实现。这里不再详述。

## 7.9.3 搭建 MKR 模型

MKR 模型由 3 个子模型组成, 完整结构如图 7-15 所示。具体描述如下。

- 推荐算法模型: 如图 7-15 的左侧部分所示, 将用户和电影作为输入, 模型的预测结果为用户对该电影的喜好分数, 数值为 0~1。
- 交叉压缩单元模型: 如图 7-15 的中间部分, 在低层将左右两个模型桥接起来。将电影评分数据集中的电影向量与知识图谱中的电影向量特征融合起来, 再分别放回各自的模型中, 进行监督训练。
- 知识图谱词嵌入 (Knowledge Graph Embedding, KGE) 模型: 如图 7-15 的右侧部分, 将知识图谱三元组中的前 2 个 (电影 ID 和关系实体) 作为输入, 预测出第 3 个 (目标实体)。

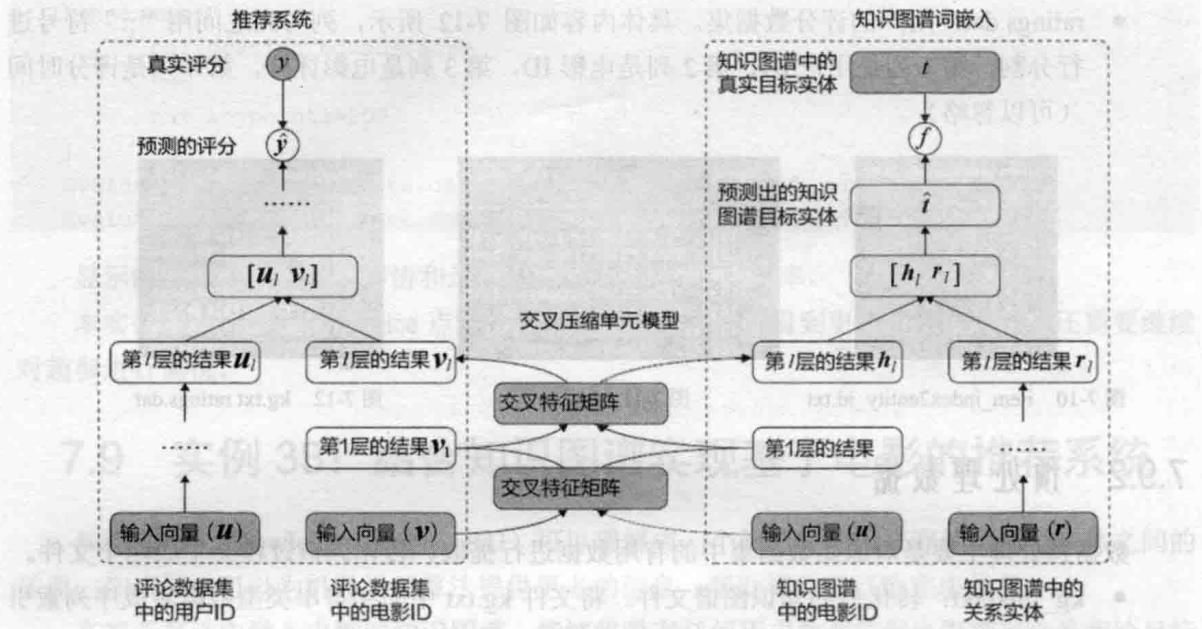


图 7-15 MKR 框架

在 3 个子模型中，最关键的是交叉压缩单元模型。下面就先从该模型开始一步一步地实现 MKR 框架。

### 1. 交叉压缩单元模型

交叉压缩单元模型可以被当作一个网络层叠加使用。如图 7-16 所示的是交叉压缩单元在第  $l$  层到第  $l+1$  层的结构。图 7-16 中，最下面一行是该单元的输入，左侧的  $v_l$  是用户评论电影数据集中的电影向量，右侧的  $e_l$  是知识图谱中的电影向量。

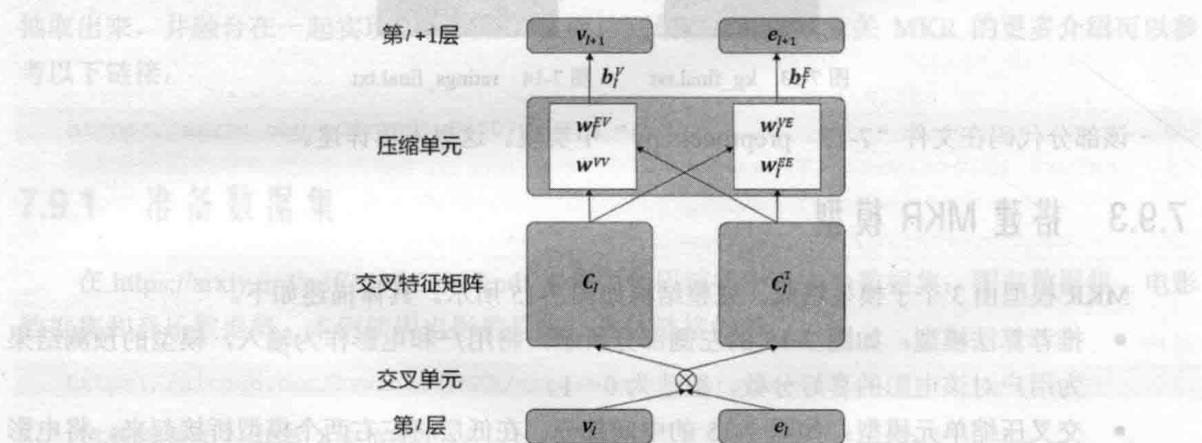


图 7-16 交叉压缩单元模型的结构

交叉压缩单元模型的具体处理过程如下：

- (1) 将  $v_l$  与  $e_l$  进行矩阵相乘得到  $c_l$ 。
- (2) 将  $c_l$  复制一份，并进行转置得到  $c_l^T$ 。实现特征交叉融合。