슬라이드 01: 特征处理章节介绍

好，现在我们进入第二大部分特征提取、转换和选择及局部敏感哈希。

这一部分包含四个子章节：

- 9.6.1 特征提取（Feature Extraction）
- 9.6.2 特征转换（Feature Transformation）
- 9.6.3 特征选择（Feature Selection）
- 9.6.4 局部敏感哈希（Locality Sensitive Hashing）

特征处理是机器学习中非常重要的环节。

俗话说"数据和特征决定了机器学习的上限，而模型和算法只是逼近这个上限而已"。

bījìn zhège shàngxiàn éryǐ

所以掌握好特征处理技术对我们来说非常重要。我们先从特征提取开始。

슬라이드 02: 特征提取操作介绍

首先我们看看特征提取操作。

spark.ml包提供的提取操作包括以下几种：

1. **TF-IDF** - 这是文本处理中最常用的技术
2. **Word2Vec** - 词向量技术
3. **CountVectorizer** - 计数向量化
4. **FeatureHasher** - 特征哈希

今天我们主要详细讲解TF-IDF，因为它在文本挖掘和自然语言处理中应用非常广泛。

大家有听说过TF-IDF吗？

슬라이드 03: TF-IDF原理讲解

现在我们详细学习TF-IDF。

"词频－逆向文件频率"(TF-IDF) 是一种在文本挖掘中广泛使用的特征向量化方法，它可以体

现一个文档中词语在语料库中的重要程度。

我们来看看相关的定义:

- 词语由 **t** 表示
- 文档由 **d** 表示
- 语料库由 **D** 表示

词频TF(t,d) 是词语t在文档d中出现的次数。 **文件频率DF(t,D)** 是包含词语的文档的个数。

TF-IDF就是在数值化文档信息，衡量词语能提供多少信息以区分文档。

看公式:

- **IDF(t,D) = log(|D|+1)/(DF(t,D)+1)**
- **TFIDF(t,d,D) = TF(t,d) · IDF(t,D)**

这个公式的直觉是什么呢？如果一个词在某个文档中出现频率高，但在整个语料库中出现频率低，那它就很有区分性。

明白了吗?

슬라이드 04: Spark ML中的TF-IDF实现

在Spark ML库中，TF-IDF被分成两部分来实现:

TF部分（+哈希）： HashingTF是一个Transformer，在文本处理中，接收词条的集合然后把这些集合转化成固定长度的特征向量。这个算法在哈希的同时会统计各个词条的词频。

IDF部分： IDF是一个Estimator，在一个数据集上应用它的 fit() 方法，产生一个IDFModel。该IDFModel接收特征向量（由HashingTF产生），然后计算每一个词在文档中出现的频次。IDF会减少那些在语料库中出现频率较高的词的权重。

为什么要分成两部分呢？这样设计更灵活，我们可以单独使用TF部分，也可以两个一起用。

这个设计思路大家理解吗?

슬라이드 05: TF-IDF处理流程

现在我们看看具体的处理过程描述:

在下面的代码段中，我们以一组句子开始，然后:

1. **首先使用分解器Tokenizer** 把句子划分为单个词语
2. **对每一个句子（词袋）**，使用HashingTF将句子转换为特征向量
3. **最后使用IDF重新调整特征向量**（这种转换通常可以提高使用文本特征的性能）

这是一个典型的文本处理流水线：文本 → 分词 → 词频统计 → IDF权重调整 → 最终特征向量

每一步都有其特定的作用，组合起来就能得到高质量的文本特征表示。

接下来我们看具体的代码实现。

슬라이드 06: TF-IDF代码实现 - 第1步

现在我们开始看具体的代码实现。

第1步：导入TF-IDF所需要的包

```
>>> from pyspark.ml.feature import HashingTF, IDF, Tokenizer
>>> from pyspark.sql import SparkSession
```

这里我们导入了三个关键的类：

- **HashingTF** - 用于计算词频并进行哈希
- **IDF** - 用于计算逆文档频率
- **Tokenizer** - 用于分词

第2步：创建一个集合，每一个句子代表一个文件

```
# 创建一个 SparkSession（如果尚未创建）
>>> spark = SparkSession.builder.appName("example").getOrCreate()


# 创建数据集
>>> data = [
    (0, "I heard about Spark and I love Spark"),
    (0, "I wish Java could use case classes"),
```

(1, "Logistic regression models are neat")

]

这里我们创建了三个句子，每个句子代表一个文档。第一列是标签，第二列是文本内容。

大家注意看，这些句子包含了一些重复的词，比如"Spark"出现了两次，这正好可以展示TF-IDF的效果。

슬라이드 07: TF-IDF代码实现 - 创建DataFrame

接下来我们把数据转换为DataFrame：

```
# 将数据集转换为DataFrame
>>> sentence_data = spark.createDataFrame(data, ["label", "sentence"])


# 显示DataFrame
>>> sentence_data.show()
+-----+-------------------+
|label|           sentence|
+-----+-------------------+
|    0|I heard about Spa...|
|    0|I wish Java could...|
|    1|Logistic regressi...|
+-----+-------------------+
```

大家看到，DataFrame已经创建成功了。这里显示时文本被截断了，这是为了适应屏幕显示。实际的完整文本还在那里。

现在我们有了结构化的数据，可以开始应用我们的文本处理流水线了。

슬라이드 08: TF-IDF代码实现 - 第3步分词

第3步：用Tokenizer把每个句子分解成单词

```
# 创建Tokenizer对象并设置输入列和输出列
>>> tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

# 使用Tokenizer拆分句子列
>>> words_data = tokenizer.transform(sentence_data)

# 显示DataFrame，不截断显示
>>> words_data.show(truncate=False)
```

结果是这样的：

```
+-----+-----------------------------------+---------------------------------------
|label|sentence                                |words

+-----+-----------------------------------+---------------------------------------
|0    |I heard about Spark and I love Spark|[i, heard, about, spark, and, i, love, spark]
|
|0    |I wish Java could use case classes  |[i, wish, java, could, use, case, classes] |
|1    |Logistic regression models are neat |[logistic, regression, models, are, neat] |
+-----+-----------------------------------+---------------------------------------
```

大家看，每个句子现在都被分解成了单词数组。注意所有词都被转换成了小写，这是Tokenizer的默认行为。

这一步为后续的词频统计做好了准备。

슬라이드 09: TF-IDF代码实现 - 第4步哈希

第4步：用HashingTF的transform()方法把每个"词袋"哈希成特征向量

```
# 创建HashingTF对象并设置输入列、输出列以及特征数量
>>> hashing_tf = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=2000)


# 使用HashingTF将单词列转换为特征向量
>>> featurized_data = hashing_tf.transform(words_data)
```

这里我们设置哈希表的桶数为2000。为什么选择2000呢？这是一个经验值，对于我们这个小例子来说已经足够了。在实际项目中，你们可能需要根据词汇量的大小来调整这个参数。

HashingTF会把每个单词映射到0到1999之间的一个索引，并统计每个索引位置的词频。

📊 슬라이드 10: TF-IDF代码实现 - 显示哈希结果

让我们看看哈希的结果：

```
# 显示DataFrame，不截断显示
>>> featurized_data.select("words", "rawFeatures").show(truncate=False)
```

结果是这样的：

```
+---------------------------------------------+---------------------------------------------------------------
|words                                                  |rawFeatures                                                  |
+---------------------------------------------+---------------------------------------------------------------
|[i, heard, about, spark, and, i, love, spark]|(2000,[240,673,891,956,1286,1756],[1.0,1.0,1.0,1.0,2.0,2.0])        |
|[i, wish, java, could, use, case, classes]   |(2000,[80,342,495,1133,1307,1756,1967],[1.0,1.0,1.0,1.0,1.0,1.0,1.0])|
|[logistic, regression, models, are, neat]    |(2000,[286,763,1059,1604,1871],[1.0,1.0,1.0,1.0,1.0])                |
```

+---+---

这个输出格式需要解释一下，我们下一页详细说明。

슬라이드 11: 输出结果解释

我们来详细解释一下第一行的输出结果：

输出结果	含义
2000	代表哈希表的桶数
[240,673,891,956,1286,1756]	代表"i","heard","about","and","love","spark"6个单词的哈希值。注意：哈希操作没有顺序性，所以索引240并不对应单词"i"
[1.0,1.0,1.0,1.0,2.0,2.0]	分别表示各单词的出现次数

特别注意最后两个2.0，这表示"i"和"spark"在第一个句子中各出现了2次。这就是词频统计的结果。

哈希的好处是什么呢？它可以把任意大小的词汇表映射到固定大小的向量空间，这样就解决了维度爆炸的问题。

大家理解这个表示方法了吗？

슬라이드 12: TF-IDF代码实现 - 第5步IDF

第5步：调用IDF方法来重新构造特征向量的规模

```
# 创建IDF对象并设置输入列和输出列
>>> idf = IDF(inputCol="rawFeatures", outputCol="features")


# 拟合数据以创建IDF模型
>>> idf_model = idf.fit(featurized_data)
```

注意这里，idf是一个评估器（Estimator），在特征向量上应用它的 fit() 方法，会产生一个IDF

Model（名称为idf_model）。

第6步：调用IDFModel的transform()方法，得到每一个单词对应的TF-IDF度量值

使用IDF模型转换数据

>>> rescaled_data = idf_model.transform(featurized_data)

现在rescaled_data就包含了最终的TF-IDF特征向量。

这个过程体现了我们之前学过的Estimator和Transformer的关系，对吗？

슬라이드 13: TF-IDF最终结果展示

我们来看看最终的TF-IDF结果：

使用IDF模型转换数据

>>> rescaled_data = idf_model.transform(featurized_data)

这一步完成后，我们就得到了每一个单词对应的TF-IDF度量值。

TF-IDF的魅力在于：

- 如果一个词在当前文档中出现频繁（高TF），但在其他文档中很少出现（高IDF），那么它就有很高的TF-IDF值
- 反之，如果一个词在所有文档中都很常见（低IDF），即使在当前文档中出现频繁，TF-IDF值也不会太高

这样就能突出那些真正具有区分性的词汇。

슬라이드 14: TF-IDF完整结果

让我们看看完整的结果：

显示DataFrame中的特征列和标签列

>>> rescaled_data.select("features", "label").show(truncate=False)

输出结果很长，我就不全部读了。大家可以看到，原来的词频值（如1.0, 2.0）现在都变成了TF-IDF值（如0.693...）。

这些数值反映了每个词在整个语料库中的重要性。数值越大，说明这个词越有区分性。

到这里，我们的TF-IDF特征提取就完成了！这些特征向量现在可以用于后续的机器学习任务，比如文本分类、聚类等。

大家对TF-IDF的整个流程清楚了吗？

슬라이드 15: 特征转换操作介绍

现在我们进入9.6.2特征转换部分。

spark.ml包提供了大量的特征转换操作，我来给大家介绍一下：

从(1)到(24)，一共24种不同的转换操作！包括：

- **Tokenizer** - 分词器
- **StringIndexer** - 字符串索引
- **PCA** - 主成分分析
- **StandardScaler** - 标准化
- **MinMaxScaler** - 最小最大缩放
- 还有很多其他的...

这么多操作说明什么？说明特征工程在机器学习中真的非常重要！今天我们重点学习几个最常用的。

슬라이드 16: 特征转换应用场景

让我给大家解释一下为什么需要特征转换。

在机器学习处理过程中，为了方便相关算法的实现，经常需要把标签数据（一般是字符串）转化成**整数索引**，或是在计算结束后将整数索引还原为相应的标签。

比如说，你们有一个"颜色"特征，值是"红"、"绿"、"蓝"。但是大多数机器学习算法只能处理数字，不能直接处理中文字符。所以我们需要把"红"转换成0，"绿"转换成1，"蓝"转换成2。

Spark ML包中提供了几个相关的转换器：

- **StringIndexer** - 字符串转索引

- **IndexToString** - 索引转字符串
- **OneHotEncoder** - 独热编码
- **VectorIndexer** - 向量索引

它们提供了十分方便的特征转换功能。

📊 슬라이드 17: StringIndexer详解

现在我们详细学习**StringIndexer（字符串索引器）**。

StringIndexer转换器可以把一列类别型的特征（或标签）进行编码，使其数值化。索引的范围从0开始，该过程可以使得相应的特征索引化，使得某些无法接受类别型特征的算法可以使用，并提高诸如决策树等机器学习算法的效率。

这里有个重要的规则：**索引构建的顺序为标签的频率，优先编码频率较大的标签，所以出现频率最高的标签为0号**。

为什么这样设计呢？因为在很多机器学习算法中，编码为0的类别会被当作"默认"类别，把最频繁的类别设为0可以提高算法效率。

如果输入的是数值型的，StringIndexer会把它转化成字符型，然后再对其进行编码。

这个设计很贴心，对吧？

📊 슬라이드 18: StringIndexer代码实现 - 导入包

现在我们看具体的代码实现。

首先，引入所需要使用的类：

python

```
>>> from pyspark.ml.feature import StringIndexer, IndexToString
>>> from pyspark.sql import SparkSession
```

这里我们同时导入了StringIndexer和IndexToString，因为它们通常是成对使用的。StringIndexer把字符串转成数字，IndexToString把数字转回字符串。

这就像是一对"翻译器"，一个负责中译英，一个负责英译中。

📊 슬라이드 19: StringIndexer代码实现 - 创建数据

其次，构建1个DataFrame，设置StringIndexer的输入列和输出列的名字：

python

```
# 创建一个 SparkSession（如果尚未创建）
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 创建数据集
>>> data = [
    (0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")
]

# 将数据集转换为DataFrame
>>> df1 = spark.createDataFrame(data, ["id", "category"])

# 创建StringIndexer对象并设置输入列和输出列
>>> indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
```

大家注意看这个数据：

- "a"出现了3次（最频繁）
- "c"出现了2次
- "b"出现了1次（最少）

根据我们刚才说的规则，"a"应该被编码为0，对吗？我们等会儿验证一下。

📊 슬라이드 20: StringIndexer执行结果

最后，通过fit()方法进行模型训练，用训练出的模型对原数据集进行处理：

python

```
# 对DataFrame进行索引
>>> indexed_df = indexer.fit(df1).transform(df1)


# 显示DataFrame
>>> indexed_df.show()
+---+--------+-------------+
| id|category|categoryIndex|
+---+--------+-------------+
|  0|       a|          0.0|
|  1|       b|          2.0|
|  2|       c|          1.0|
|  3|       a|          0.0|
|  4|       a|          0.0|
|  5|       c|          1.0|
+---+--------+-------------+
```

验证一下我们的预测：

- "a"（出现3次）→ 0.0 ✓
- "c"（出现2次）→ 1.0 ✓
- "b"（出现1次）→ 2.0 ✓

完全符合频率排序的规则！这就是StringIndexer的工作原理。

슬라이드 21: IndexToString介绍

现在我们学习**IndexToString（索引转字符串)**。

与StringIndexer相反，IndexToString的作用是把标签索引的一列重新映射回原有的字符型标签。

IndexToString一般都和StringIndexer配合使用。典型的使用场景是：

1. 先用StringIndexer转化成标签索引
2. 进行模型训练
3. 在预测标签的时候再把标签索引转化成原有的字符标签

这就像是一个完整的"翻译-处理-翻译回来"的过程。

当然，Spark允许使用自己提供的标签，不一定要用原来的标签。这给了我们更多的灵活性。

슬라이드 22: IndexToString代码实现

让我们看看IndexToString的具体使用：

python

```
# 创建一个 IndexToString 转换器
>>> index_to_string = IndexToString(inputCol="categoryIndex", outputCol="originalCategory")

# 使用转换器将索引列转换回原始类别列
>>> string_data = index_to_string.transform(indexed_df)

# 显示DataFrame中的"id"和"originalCategory"列
>>> string_data.select("id", "originalCategory").show()
+---+----------------+
| id|originalCategory|
+---+----------------+
```

| 0| a|

| 1| b|

| 2| c|

| 3| a|

| 4| a|

| 5| c|

+---+----------+

大家看，我们成功地把数字索引又转换回了原来的字符串！

这个过程展示了特征转换的可逆性，这在实际应用中非常重要。

슬라이드 23: VectorIndexer介绍

现在我们学习**VectorIndexer（向量索引器）**。

StringIndexer对单个类别型特征进行转换。但是如果特征都已经被组织在一个向量中，又想对其中某些单个分量进行处理，则可以利用VectorIndexer类进行转换。

VectorIndexer类的**maxCategories参数**可以自动识别类别型特征，并将原始值转换为类别索引。它基于特征值的数量来识别需要被类别化的特征。

简单来说：那些取值数最多不超过maxCategories的特征，将会被类型化并转化为索引。

比如说，如果某个特征只有3种不同的值，而我们设置maxCategories=5，那么这个特征就会被当作类别特征处理。但如果某个特征有100种不同的值，就会被当作连续特征处理。

这个自动识别功能很智能，对吧？

슬라이드 24: VectorIndexer代码实现 - 数据准备

首先引入所需要的类，并构建数据集：

python

```
>>> from pyspark.ml.feature import VectorIndexer
>>> from pyspark.ml.linalg import Vectors
```

```
>>> from pyspark.sql import SparkSession

# 创建Spark会话
>>> spark = SparkSession.builder.appName("example").getOrCreate()

# 创建示例数据
>>> data = [Vectors.dense(-1.0, 1.0, 1.0),
            Vectors.dense(-1.0, 3.0, 1.0),
            Vectors.dense(0.0, 5.0, 1.0)]

# 创建DataFrame并指定列名
>>> df = spark.createDataFrame([(vector,) for vector in data], ["features"])
```

这里我们创建了3个3维向量。大家看看这些数据：

- 第0维：有-1.0和0.0两种值（类别特征）
- 第1维：有1.0, 3.0, 5.0三种值（可能是类别特征）
- 第2维：都是1.0（肯定是类别特征）

我们设置maxCategories=2，看看VectorIndexer如何自动识别。

슬라이드 25: VectorIndexer执行和结果

然后，构建VectorIndexer转换器，设置输入和输出列，并进行模型训练：

python

```
# 创建VectorIndexer对象并设置输入列、输出列以及最大不同特征值的数量
>>> vector_indexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=2)
```

```
>>> indexerModel = vector_indexer.fit(df)
>>> indexedData = indexerModel.transform(df)

>>> indexedData.show()
+-------------+---------------+
|         features|indexedFeatures|
+-------------+---------------+
|[-1.0,1.0,1.0]|  [1.0,1.0,0.0]|
|[-1.0,3.0,1.0]|  [1.0,3.0,0.0]|
| [0.0,5.0,1.0]|  [0.0,5.0,0.0]|
+-------------+---------------+
```

大家看结果：

- 第0维：-1.0→1.0，0.0→0.0（被转换了）
- 第1维：保持不变（因为有3种值，超过了maxCategories=2）
- 第2维：1.0→0.0（被转换了）

这正好验证了我们的预测！

📊 슬라이드 26: VectorIndexer分类特征确认

接下来，通过VectorIndexerModel的categoryMaps成员来获得被转换的特征及其映射：

python

```
# 获取分类特征信息的索引
>>> categorical_features = set(indexerModel.categoryMaps.keys())
```

 # 打印分类特征信息

>>> print(f"Chose {len(categorical_features)} categorical features: {', '.join(map(str, categorical_features))}")

Chose 2 categorical features: 0, 2

这里可以看到，共有两个特征被转换，分别是0号和2号。

这证实了我们的分析：

- 特征0：2种不同值（≤2）→ 类别特征
- 特征1：3种不同值（>2）→ 连续特征
- 特征2：1种不同值（≤2）→ 类别特征

VectorIndexer的自动识别功能工作得很好！

📊 슬라이드 27: VectorIndexer最终演示

最后，把模型应用于原有的数据，并打印结果：

python

 # 使用VectorIndexerModel对DataFrame进行特征列索引

>>> indexed = indexerModel.transform(df)

 # 显示DataFrame

>>> indexed.show()

```
+--------------+-------------+
|      features|      indexed|
+--------------+-------------+
|[-1.0,1.0,1.0]|[1.0,1.0,0.0]|
|[-1.0,3.0,1.0]|[1.0,3.0,0.0]|
| [0.0,5.0,1.0]|[0.0,5.0,0.0]|
```

```
+-------------+-------------+
```

到这里，我们对VectorIndexer的学习就结束了。

📊 总结和回顾

好，同学们，今天的课程就要结束了。我们来总结一下今天学习的内容：

今天学习的核心概念：

1. **机器学习流水线（ML Pipeline）**的四个核心组件：
    - **DataFrame** - 数据的载体
    - **Transformer** - 数据转换器
    - **Estimator** - 模型估计器
    - **Pipeline** - 工作流管理器
2. **流水线的工作机制**：
    - 训练阶段：Pipeline → PipelineModel
    - 预测阶段：PipelineModel → 预测结果
3. **特征处理技术**：
    - **TF-IDF** - 文本特征提取
    - **StringIndexer/IndexToString** - 类别特征转换
    - **VectorIndexer** - 向量特征自动识别

重要的设计思想：

- **统一的API（API)**：所有组件都遵循相同的接口规范
- **可复用性**：训练好的模型可以反复使用
- **自动化**：很多处理过程可以自动完成

这些概念和技术在实际的机器学习项目中都非常重要。希望大家课后多练习，加深理解。

有什么问题可以随时问我。下节课我们将继续学习更多的特征处理技术。

谢谢大家！**슬라이드** 1: 课程标题介绍

同学们好！今天我们来学习第9.5章的内容——机器学习流水线（Machine Learning Pipeline）。

这是一个非常重要的概念，特别是当我们需要处理复杂的机器学习任务时。流水线可以让我们把多个数据处理步骤串联起来，形成一个完整的、可重复使用的机器学习工作流程。

今天的内容主要分为两个部分：

- 首先我们会学习流水线的基本概念
- 然后我们会看看流水线是如何工作的

大家准备好了吗？我们开始吧！

Spark 特征选择与局部敏感哈希 - 中文讲课脚本

第一张슬라이드：特征选择概述

大家好！今天我们来学习一个非常重要的主题，就是特征选择（Feature Selection）。

首先我们看一下这个标题"9.6.3 特征选择"。那么什么是特征选择呢？

大家可以看到这里的定义：特征选择（Feature Selection）指的是在特征向量中选择出那些"优秀"的特征，组成新的、更"精简"的特征向量的过程。

这个概念很重要，让我用一个简单的例子来解释一下。比如说我们有一个数据集，里面有很多列数据，有些列对我们的分析很有用，有些列可能没什么用，甚至会干扰我们的分析。特征选择就是要把那些有用的列挑出来，把没用的列去掉。

它在高维数据分析中十分常用，为什么呢？因为可以剔除掉"冗余"和"无关"的特征，从而提升学习器的性能。

想象一下，如果你有一个包含1000个特征的数据集，但其中只有50个特征真正有用，那么使用所有1000个特征不仅浪费计算资源，还可能降低模型的准确性。这就是为什么我们需要特征选择的原因。

第二张슬라이드：特征选择操作

现在我们来看看具体有哪些特征选择的方法。在Spark机器学习库（Spark ML）中，主要有五种特征选择的操作。今天我们先看前三种：

首先是**VectorSlicer**。这个名字听起来可能有点复杂，但其实很简单。它的作用类似于MATLAB或者numpy中的"列切片"。什么意思呢？就是说你可以指定一些索引，比如说你想要第1列、第3列和第5列，VectorSlicer就帮你把这些列提取出来，生成一个新的特征向量。

第二个是**RFormula**。这个更有意思，它提供了一种R语言风格的特征向量列选择功能。如果你学过R语言，你就知道R语言有一套很方便的公式语法。RFormula让你可以用类似的语法来选择特征。比如你可以写一个表达式，它会根据这个表达式自动选择相应的特征列。

第三个是**ChiSqSelector**，这是基于卡方检验（Chi-Square Test）的特征选择方法。这个方法比较高级，它通过卡方选择的方法来进行特征选择。重要的是，它需要一个已有标签的数据集。ChiSqSelector会针对每一个特征与标签的关系进行卡方检验，从而选择出那些统计意义上区分度最强的特征。

简单来说，就是看哪些特征和你想预测的目标最相关。

第三张슬라이드：更多特征选择方法

接下来我们看另外两种方法：

第四个是**UnivariateFeatureSelector**。这个可以看成是ChiSqSelector的扩展版本。为什么说是扩展呢？因为它不仅支持分类数据，还支持连续型数据的特征选择。

Spark会很聪明地根据特征类型和标签类型来自动选择评估函数：

- 当特征与标签都是分类型时，它用的也是chi-squared，这时候功能就相当于ChiSqSelector
- 当特征是连续型，标签是分类型时，它会用ANOVATest
- 当特征与标签类型都是连续型时，它用的是F-value

这样就能处理各种不同类型的数据了。

第五个是**VarianceThresholdSelector**。这个方法相对简单一些，它的思路是：如果一个特征的方差很小，说明这个特征的值变化不大，那它可能就不太有用。

所以VarianceThresholdSelector会把方差值小于设定阈值的特征删除。默认阈值是0，这意味着只有方差为0的特征，也就是所有值都相同的特征，才会被删除。

比如说有一列数据全部都是"男"，没有"女"，那这一列对于性别分析就没有意义了。

第四张슬라이드：卡方选择实例介绍

好，现在我们来看一个具体的例子。我们选择卡方选择来做演示，因为这是一个很经典的方法。

卡方选择是统计学上常用的一种有监督特征选择方法。什么叫有监督呢？就是说我们有标签，有正确答案。它通过对特征和真实标签之间进行卡方检验，来判断该特征和真实标签的关联程度，进而确定是否对其进行选择。

和spark.ml包中的大多数学习方法一样，spark.ml包中的卡方选择也是以"评估器+转换器"（Estimator + Transformer）的形式出现的。这是Spark ML的一个重要设计模式。主要由ChiSqSelector和ChiSqSelectorModel两个类来实现。

现在我们开始写代码。首先，进行环境的设置，引入卡方选择器所需要使用的类。

大家看到这里的代码：

```
from pyspark.sql import SparkSession
```

from pyspark.ml.linalg import Vectors

from pyspark.sql import Row

from pyspark.ml.feature import ChiSqSelector

这些都是我们需要导入的模块。

第五张슬라이드：创建实验数据

接下来，我们创建一些实验数据。这是一个比较简单的例子，有3个样本、4个特征维度的数据集，标签有1和0两种。我们将在此数据集上进行卡方选择。

首先创建Spark会话（SparkSession）：

spark = SparkSession.builder.appName("example").getOrCreate()

然后创建数据集。大家可以看到这个data数组：

```
data = [
    (1, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1),
    (2, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0),
    (3, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0)
]
```

每一行包含三个部分：ID、特征向量和标签。比如第一行，ID是1，特征向量是[0.0, 0.0, 18.0, 1.0]，标签是1。

注意这里我们用的是Vectors.dense()来创建密集向量（Dense Vector）。

第六张슬라이드：数据格式转换

现在我们需要把这个数据转换成DataFrame格式，因为Spark ML都是基于DataFrame来工作的。

df = spark.createDataFrame([Row(id=id, features=vec, label=label) for id, vec, label in data])

df.show()

运行后我们可以看到这样一个表格:

```
+---+-----------------+-----+
| id|         features|label|
+---+-----------------+-----+
|  1|[0.0,0.0,18.0,1.0]|    1|
|  2|[0.0,1.0,12.0,0.0]|    0|
|  3|[1.0,0.0,15.0,0.1]|    0|
+---+-----------------+-----+
```

这样我们的数据就准备好了。可以看到有三行数据，每行都有ID、特征和标签三列。

第七张슬라이드：配置卡方选择器

现在我们来配置卡方选择器。用卡方选择进行特征选择器的训练。为了便于观察效果，我们设置只选择和标签关联性最强的一个特征。

```
selector = ChiSqSelector(
    numTopFeatures=1,
    featuresCol="features",
    labelCol="label",
    outputCol="selected-feature"
)
```

让我解释一下这些参数:

- numTopFeatures=1：表示我们只选择最好的1个特征
- featuresCol="features"：指定输入的特征列名
- labelCol="label"：指定标签列名
- outputCol="selected-feature"：指定输出列的名字

你可以通过setNumTopFeatures()方法来调整要选择的特征数量。

第八张슬라이드：执行特征选择

最后，我们用训练出的模型对原数据集进行处理。

```
# 拟合ChiSqSelector模型
selector_model = selector.fit(df)

# 使用模型进行特征选择
result = selector_model.transform(df)

# 显示结果，不截断显示
result.show(truncate=False)
```

运行后我们可以看到结果：

```
+---+------------------+-----+----------------+
|id |features          |label|selected-feature|
+---+------------------+-----+----------------+
|1  |[0.0,0.0,18.0,1.0]|1    |[18.0]          |
|2  |[0.0,1.0,12.0,0.0]|0    |[12.0]          |
|3  |[1.0,0.0,15.0,0.1]|0    |[15.0]          |
+---+------------------+-----+----------------+
```

太好了！我们可以看到，第3列特征被选出作为最有用的特征列。原来的特征向量有4个维度，现在新的selected-feature列只包含一个值，就是原来第3列（索引为2）的值：18.0、12.0、15.0。

这说明通过卡方检验，系统认为第3列特征与标签的关联性最强。

第九张슬라이드：局部敏感哈希简介

好，现在我们来学习另一个重要的概念：局部敏感哈希（Locality Sensitive Hashing，LSH）。

局部敏感哈希是一种被广泛应用在聚类（Clustering）、近似最近邻（Approximate Nearest Neighbor，ANN）、近似相似度连接（Approximate Similarity Join）等操作的哈希方法。

它的基本思想是什么呢？就是将那些在特征空间中相邻的点尽可能地映射到同一个哈希桶中。

想象一下，你有很多数据点散布在一个空间里，相似的点应该在相近的位置。传统的哈希是为了让数据分布得尽可能均匀，但LSH恰恰相反，它要让相似的数据落到同一个桶里。

spark.ml包目前提供了两种LSH方法：

第一种是**BucketedRandomProjectionLSH**，它使用欧几里得距离（Euclidean Distance）作为距离度量方法。这适合处理连续数值型的数据。

第二种是**MinHash**，它使用Jaccard距离作为距离度量方法。显然，根据Jaccard相似度的性质，它只能够处理二元(0-1值)向量。也就是说，你的数据必须是只包含0和1的向量。

这两种方法各有各的应用场景，我们会根据数据的特点来选择合适的方法。

总结

好的，今天我们学习了特征选择和局部敏感哈希的基本概念。特征选择帮助我们从大量特征中挑选出最有用的，而局部敏感哈希则帮助我们快速找到相似的数据。

这些都是大数据处理和机器学习中非常实用的技术。希望大家课后多练习，特别是那个卡方选择的例子，大家可以尝试改变参数，看看结果有什么变化。

有什么问题吗？

9.7 分类算法课堂讲授脚本

슬라이드 1 - 课程开始

同学们好！今天我们来学习第 9.7 节的内容——分类算法。大家请看屏幕。

分类算法，这是机器学习(Machine Learning)和数据挖掘(Data Mining)领域中一个非常重要的技术。那什么是分类呢？简单来说，分类就是要根据我们已有的数据集的特点，来构造一个分类函数或者分类模型。我们也把这个模型叫做分类器(Classifier)。

这个分类器有什么作用呢？它能够把那些我们不知道类别的新样本，准确地分配到我们预先定义好的类别当中去。比如说，我们有一些鸢尾花的数据，通过训练好的分类器，就能判断出一朵新的鸢尾花到底属于哪个品种。

슬라이드 2 - 分类的数学表示

好，我们来看看分类的具体规则是怎么表示的。

首先，我们有一个训练数据集 T，这个 T 里面的每一条记录都包含了若干个属性的特征向量。我们用向量 X 等于(x1, x2, ..., xn)来表示这些特征。注意啊，这里的 xi，也就是第 i 个特征，它可以有不同的值域。

如果一个属性的值域是连续的，比如说身高、体重这种，我们就叫它连续属性(Numerical Attribute)。如果不是连续的，比如说性别、颜色这种，我们就叫它离散属性(Discrete Attribute)。

然后呢，我们用 C 等于(c1, c2, ..., ck)来表示类别属性，也就是说我们的数据集有 k 个不同的类别。

那么分类的目标是什么呢？就是要找到一个从特征向量 X 到类别属性 C 的映射函数 f(X)→C。通过分析训练集中出现的特性，为每一个类找到一种准确的分类函数或者分类模型，这样就能把隐含的映射函数表示出来。

슬라이드 3 - 训练和测试阶段

接下来我们说说构造分类模型的过程。这个过程一般分为两个阶段：训练阶段和测试阶段。

在构造模型之前，我们要先把数据集随机地分成两部分：训练数据集(Training Dataset)和测试数据集(Test Dataset)。

第一步，我们用训练数据集来构造分类模型，这个过程叫做训练(Training)。 第二步，我们用测试数据集来评估模型的分类准确率，这个过程叫做测试(Testing)。

如果我们认为模型的准确率可以接受，那就可以用这个模型对其他的数据元组进行分类了。

这里有个很重要的特点：一般来说，测试阶段的代价远远低于训练阶段的代价。为什么呢？因为训练需要大量的计算，而测试相对来说比较简单。

슬라이드 4 - Spark MLlib 支持的算法

好，现在我们来看看都有哪些分类算法。

分类算法有很多种不同的类型，比如说支持向量机(Support Vector Machines, SVM)、决策树算法(Decision Tree)、贝叶斯算法(Bayes Algorithm)等等。

在我们的课程中，我们主要使用 Spark 的 mllib 包。这个 spark.mllib 包支持各种分类算法，涉及的问题类型主要包含二分类(Binary Classification)、多分类(Multi-class Classification)和回归分析(Regression)等。

大家看这个表格，这是表 9-4，列出了 spark.mllib 包为不同类型问题提供的算法：

对于二分类问题，我们有线性支持向量机、逻辑斯谛回归(Logistic Regression)、决策树、随机森林(Random Forest)、梯度上升树(Gradient Boosting Tree)、朴素贝叶斯(Naive Bayes)。

对于多类分类，我们有逻辑斯谛回归、决策树、随机森林、朴素贝叶斯。

对于回归问题，我们有线性最小二乘法、Lasso 回归、岭回归(Ridge Regression)、决策树、随机森林、梯度上升树、保序回归(Isotonic Regression)。

슬라이드 5 - 章节结构

今天我们主要学习两个算法：

9.7.1 逻辑斯谛回归分类算法(Logistic Regression Classification Algorithm) 9.7.2 决策树分类算法(Decision Tree Classification Algorithm)

我们先从逻辑斯谛回归开始。

슬라이드 6 - 逻辑斯谛回归简介

逻辑斯谛回归(Logistic Regression)，这是统计学习中的经典分类方法，它属于对数线性模型(Log-linear Model)。

这个算法有个很重要的特点：logistic 回归的因变量可以是二分类的，也可以是多分类的。也就是说，它既能处理只有两个类别的问题，比如判断邮件是不是垃圾邮件；也能处理多个类别的问题，比如判断鸢尾花的品种。

슬라이드 7 - 逻辑斯谛回归的数学模型

现在我们来看逻辑斯谛回归的数学表示。

这里我们看到的是二项逻辑斯谛回归模型的公式。

第一个公式(9-7)表示的是：给定输入 x 的情况下，输出 Y 等于 1 的概率。

第二个公式(9-8)表示的是：给定输入 x 的情况下，输出 Y 等于 0 的概率。

这里，x 属于 n 维实数空间，是我们的输入特征向量；Y 属于{0,1}，是我们的输出；w 我们称为权值向量(Weight Vector)；b 我们称为偏置(Bias)；w·x 表示 w 和 x 的内积。

参数估计的方法是：在给定训练样本点和已知的结果后，对于一个或多个未知参数故参数函数的所有可能取值，找到最符合样本点分布的参数（或参数组合）。

假设 P(Y=1|x)=π(x)，P(Y=0|x)=1-π(x)，这是公式(9-9)。

然后我们采用极大似然法(Maximum Likelihood Method)来估计 w 和 b，似然函数就是公式(9-10)这个连乘的形式。

这里 N 是训练样本的个数，(xi, yi)表示样本变量 xi 对应的值为 yi。为了求解，我们对其对数似然进行估计。

슬라이드 8 - 损失函数和正则化

我们继续看数学推导。

公式(9-11)就是对数似然函数 L(w)。从 L(w)求极大值，就能得到 w 的估计值。

但是呢，为了避免过拟合(Overfitting)的问题，我们一般会对成本 L(w)增加正则化项(Regularization Term)。这就是公式(9-12)。

这里，参数 γ 称为正则化系数(Regularization Coefficient)，用于定义正则化的权重；α 称为 Elastic Net 参数，取值介于 0 和 1 之间。当 α=0 时，采用 L2 正则化；当 α=1 时，采用 L1 正则化。

求极值的方法可以是梯度下降法(Gradient Descent)、梯度上升法(Gradient Ascent)等等。

슬라이드 9 - 实验任务描述

好，理论讲完了，现在我们来做实验。

任务描述：我们以 iris 数据集为例进行分析。这个 iris 数据集大家可以从这个网址下载：http://dblab.xmu.edu.cn/blog/wp-content/uploads/2017/03/iris.txt

iris 数据集是以鸢尾花的特征作为数据来源的。这个数据集包含 150 个数据，分为 3 类，每类 50 个数据，每个数据包含 4 个属性。它是在数据挖掘、数据分类中非常常用的测试集、训练集。

为了便于理解，这里我们主要用后两个属性（花瓣的长度和宽度）来进行分类。

슬라이드 10 - 第 1 步：导入必要的库

现在我们开始实际操作。

第 1 步：导入本地向量 Vector 和 Vectors，导入所需要的类。

大家看屏幕上的代码，我们需要导入这些模块：

```
from pyspark.ml.linalg import Vector, Vectors

from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer

from pyspark.ml.classification import LogisticRegression

from pyspark.ml import Pipeline, PipelineModel

from pyspark.sql import Row

from pyspark.ml.classification import LogisticRegressionModel

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

from pyspark.ml.feature import VectorAssembler
```

from pyspark.sql.types import DoubleType

这些都是我们在 Spark MLlib 中进行机器学习需要用到的核心模块。

슬라이드 11 - 第 2 步：读取数据

第 2 步：使用 spark.read.csv 方法读取 CSV 文件，再将 4 个特征转化为 Double 类型。

首先，我们指定 CSV 文件路径。注意这里用的是 file://协议，表示本地文件系统。

然后创建一个 SparkSession，如果还没有创建的话。SparkSession 是 Spark 2.0 以后的统一入口点。

接下来使用 Spark 读取 CSV 文件并推断列的数据类型，然后重命名列。这里我们把列名改为 c0、c1、c2、c3、label，这样比较简洁。

为什么要转化为 Double 类型呢？这是为了适应文件末尾空行等情况，保证数据的一致性。

슬라이드 12 - 数据类型转换和特征组装

继续第 2 步的操作：

我们需要将列的数据类型转换为 Double。大家看这个代码，我们使用 select 方法和 cast 方法来完成类型转换。

然后呢，我们创建 VectorAssembler 并设置输入列和输出列。VectorAssembler 是什么呢？它是一个特征转换器(Feature Transformer)，可以将多个列组合成一个向量列。这在机器学习中非常重要，因为算法通常需要一个包含所有特征的向量作为输入。

最后，我们使用 VectorAssembler 将特征列组装成特征向量，并且只保留 features 和 label 这两列。

슬라이드 13 - 查看数据

第 2 步的最后，我们来看看数据是什么样子的。

使用 data.show()方法，我们可以看到 DataFrame 的内容。大家看屏幕，每一行都有一个 features 向量和一个 label。

features 列包含了四个数值，分别对应鸢尾花的四个特征：萼片长度、萼片宽度、花瓣长度、花瓣宽度。

label 列显示的是鸢尾花的品种，这里我们看到的是"Iris-setosa"，这是山鸢尾的意思。

注意下面显示"only showing top 20 rows"，表示只显示了前 20 行数据。

슬라이드 14 - 第 3 步：索引转换

第 3 步：分别获取标签列和特征列，进行索引并进行重命名。

这里我们做两个索引转换：

首先，使用 StringIndexer 对标签列进行索引。为什么要这么做呢？因为机器学习算法通常需要数值型的标签，而我们的标签是字符串型的，所以需要转换。StringIndexer 会将字符串标签转换为数值索引。

然后，使用 VectorIndexer 对特征列进行索引。VectorIndexer 主要是用来处理分类特征的，它会自动识别哪些特征是分类的，哪些是连续的。

这两个操作都使用了 fit 方法，表示我们要先在数据上拟合这些转换器，学习到转换的规则。

슬라이드 15 - 第 4 步：设置算法参数

第 4 步：设置 LogisticRegression 算法的参数。

这里我们设置了几个重要的参数：

- labelCol="indexedLabel"：指定标签列
- featuresCol="indexedFeatures"：指定特征列
- maxIter=100：最大迭代次数为 100 次
- regParam=0.3：正则化参数为 0.3
- elasticNetParam=0.8：弹性网络参数为 0.8

大家注意，这些参数的具体含义我们可以通过 explainParams()方法来获取，这个方法会显示所有可设置的参数及其说明，还能看到程序已经设置的参数的结果。

这种参数设置的方式在 Spark MLlib 中非常常见，可以让我们灵活地调整算法的行为。

슬라이드 16 - LogisticRegression 参数详解（上）

现在我们详细看看 LogisticRegression()的参数及其含义。这是表 9-5。

elasticNetParam：这是 ElasticNet 参数 α，α 介于 0 和 1 之间，默认为 0。当 α=0 时，采用 L2 正则化；当 α=1 时，采用 L1 正则化。

family：用来设置描述模型的标签分类。有三个选项：auto、binomial 和 multinomial，默认为 auto。auto 会自动选择分类的数量，如果分类数等于 1 或 2，设置为 binomial（二分类），否则设置为 multinomial。

featuresCol：用来设置特征列名，默认为"features"。

fitIntercept：用来设置是否匹配一个截距项，默认为 true。

labelCol：用来设置标签列名，默认为"label"。

maxIter：用来设置最大的迭代次数，默认为 100。

predictionCol：用来设置预测列名，默认为"prediction"。

슬라이드 17 - LogisticRegression 参数详解（下）

我们继续看其他的参数：

probabilityCol：用来设置预测属于某一类的条件概率的列名，默认值为"probability"。这里要注意，因为不是所有的模型输出都是精确校准后的概率估计，所以这些概率应当视为置信度，而不是精确的概率估计。

rawPredictionCol：用来设置原始预测值（也称为置信度）列名。

regParam：用来设置正则化参数，默认为 0。

standardization：用来设置是否在模型拟合前对训练特征进行标准化处理，默认为 true。

threshold：用来设置二元分类预测的阈值，默认为 0.5。

thresholds：这个比较复杂，是多元分类中用来调整每一个分类预测概率的阈值参数。

tol：用来设置迭代算法的收敛阈值，默认为 1.0E-6。

weightCol：用来设置权重列名，如果没有设置，则把所有实例的权重设为 1。

슬라이드 18 - 第 5 步：构建 Pipeline

第 5 步：设置一个 IndexToString 的转换器，把预测的类别重新转化成字符型的。构建一个机器学习流水线(Pipeline)，设置各个阶段。

首先我们创建 IndexToString 转换器。这个转换器的作用是什么呢？前面我们把字符串标签转换成了数值，现在我们要把预测结果再转换回字符串，这样更容易理解。

然后我们创建 Pipeline。Pipeline 是 Spark MLlib 中一个非常重要的概念，它可以把多个数据处理和机器学习的步骤串联起来。

看这里的 stages 参数，我们设置了四个阶段：

1. label_indexer：标签索引化
2. feature_indexer：特征索引化
3. lr：逻辑回归算法
4. label_converter：将预测结果转换回字符串

上一个阶段的输出将是下一个阶段的输入，这样就形成了一个完整的机器学习工作流。

슬라이드 19 - 第 6 步：训练和预测

第 6 步：把数据集随机分成训练集和测试集，其中训练集占 70%。

我们使用 randomSplit 方法，传入[0.7, 0.3]，表示 70%用于训练，30%用于测试。

然后进行模型训练。Pipeline 本质上是一个评估器(Estimator)，当 Pipeline 调用 fit()的时候就产生了一个 PipelineModel，它是一个转换器(Transformer)。

最后进行预测。这个 PipelineModel 可以调用 transform()来进行预测，生成一个新的 DataFrame。也就是说，我们利用训练得到的模型对测试集进行验证。

这个过程体现了机器学习的标准流程：数据分割→模型训练→模型预测→模型评估。

슬라이드 20 - 第 7 步：查看预测结果

第 7 步：输出预测的结果。

我们用 select()选择要输出的列：predictedLabel（预测标签）、label（真实标签）、features（特征）、probability（概率）。

然后用 collect()获取所有行的数据，用 for 循环把每行都输出。

大家看屏幕上的输出结果。每一行显示了：

- 真实标签：比如 Iris-setosa
- 特征向量：比如[4.4,2.9,1.4,0.2]
- 概率分布：比如[0.611..., 0.215..., 0.172...]，这表示属于三个类别的概率
- 预测标签：Iris-setosa

从这个例子可以看出，模型预测这朵花属于 Iris-setosa 的概率最高（约 61%），所以预测结果就是 Iris-setosa，和真实标签一致。

슬라이드 21 - 第 8 步：模型评估

第 8 步：对训练的模型进行评估。

我们创建一个 MulticlassClassificationEvaluator 实例。这是 Spark MLlib 提供的多分类评估器，可以计算各种评估指标。

我们用 setter 方法把预测分类的列名和真实分类的列名进行设置：

- setLabelCol("indexedLabel")：设置真实标签列
- setPredictionCol("prediction")：设置预测标签列

然后计算预测准确率。从结果可以看到，Logistic Regression 的准确率是 0.839，也就是 83.9%。这个结果还是不错的！

准确率是评估分类模型最常用的指标之一，它表示预测正确的样本数占总样本数的比例。

슬라이드 22 - 第 9 步：获取模型参数

第 9 步：通过 Pipeline Model 来获取训练得到的逻辑斯谛模型。

因为 lrPipelineModel 是一个 PipelineModel，我们可以通过调用它的 stages()方法来获取其中的各个模型。这里我们要的是第 3 个阶段（索引为 2）的模型，也就是 LogisticRegression 模型。

然后我们可以查看模型的各种参数:

- Coefficients（系数矩阵）：这是一个 3×4 的稀疏矩阵，显示了各个特征的权重
- Intercept（截距）：三个类别对应的截距值
- numClasses（类别数）：3 个类别
- numFeatures（特征数）：4 个特征

这些参数就是我们训练得到的逻辑回归模型的核心。通过这些参数，模型就能对新的数据进行分类预测了。

课程总结

好，今天我们学习了逻辑斯谛回归分类算法，从理论到实践都进行了详细的讲解。

我们学习了:

1. 分类算法的基本概念和数学表示
2. 逻辑回归的数学原理和公式推导
3. 使用 Spark MLlib 进行逻辑回归的完整流程
4. 如何评估模型性能和查看模型参数

大家回去之后，可以尝试调整不同的参数，看看对模型性能有什么影响。下节课我们将学习决策树分类算法。

有什么问题吗?

9.7.2 决策树分类算法 - 中文讲课脚本

第 1 张슬라이드

大家好！今天我们来学习第 9.7.2 节的内容，也就是决策树分类算法(Decision Tree Classification Algorithm)。

首先，我们来看看什么是决策树。决策树(Decision Tree)是机器学习中一种非常重要的分类和回归方法。今天我们主要关注它在分类问题上的应用。

大家可以把决策树想象成一个倒立的树形结构。这个树的每个内部节点代表对某个特征的测试，比如说"这个人的年龄是不是大于 30 岁？"每个分支代表测试的结果，比如"是"或者"不是"。而每个叶子节点代表最终的分类结果。

决策树的工作原理是这样的：当我们要对新数据进行分类时，我们从根节点开始，根据数据的特征值一步步向下走，直到达到叶子节点，叶子节点的类别就是我们的预测结果。

决策树学习通常包括三个重要步骤，大家记住这三个步骤：

1. 特征选择 - 选择哪个特征来分割数据
2. 决策树的生成 - 实际构建这棵树
3. 决策树的剪枝 - 简化树结构，避免过拟合

第 2 张슬라이드

现在我们详细讲讲第一个步骤：特征选择(Feature Selection)。

特征选择是决策树算法中非常关键的一步。想象一下，如果我们有很多个特征，比如年龄、收入、教育水平等等，我们应该选择哪个特征来首先分割我们的数据呢？

这就需要用到一些数学准则来帮助我们做决定。最常用的准则有三种：

1. 信息增益(Information Gain)
2. 信息增益比(Information Gain Ratio)
3. 基尼指数(Gini Index)

基本思路是什么呢？我们会计算每个特征的这些指标值，然后比较它们的大小。选择信息增益最大的特征，或者信息增益比最大的特征，或者基尼指数最小的特征。

为什么要这样选择呢？因为我们希望每次分割都能让数据变得更"纯净"，也就是说，分割后的每个子集中同一类别的样本越多越好。

第 3 张슬라이드

现在我们来看看这些数学公式。不要被这些公式吓到，我们一个个来理解。

信息增益(Information Gain)： 公式是 $g(D,A) = H(D) - H(D|A)$

这个公式的意思是：信息增益等于原始数据集的熵减去按特征 A 分割后的条件熵。熵越小，数据越纯净。所以信息增益越大，说明这个特征的分割效果越好。

信息增益比(Information Gain Ratio)： 公式是 $g_R(D,A) = g(D,A) / H_A(D)$

信息增益比是为了解决信息增益偏向于选择取值较多的特征这个问题。通过除以特征 A 本身的熵来进行归一化。

基尼指数(Gini Index)： 公式是 $Gini(p) = 1 - \sum p_k^2$

基尼指数衡量的是数据集的不纯度。基尼指数越小，数据集越纯净。在决策树的 CART 算法中经常使用基尼指数。

这些公式看起来复杂，但概念很简单：我们就是要找到能够最好地分离不同类别的特征。

第 4 张슬라이드

接下来我们讲第二个步骤：决策树的生成(Decision Tree Generation)。

这个过程是一个递归的过程，我来给大家详细解释一下：

首先，我们从根节点开始。对于根节点，我们计算所有可能特征的信息增益（或者其他准则），然后选择信息增益最大的特征作为根节点的分割特征。

然后，根据这个特征的不同取值，我们创建子节点。比如说，如果我们选择的特征是"年龄"，可能的取值是"年轻"、"中年"、"老年"，那么我们就创建三个子节点。

接下来，对每个子节点，我们重复同样的过程：计算剩余特征的信息增益，选择最佳特征继续分割。

这个过程什么时候停止呢？我们需要设置停止条件：

1. 最基本的条件是：当前节点下的所有样本都属于同一个类别
2. 或者所有样本的特征值都相同了
3. 在实际应用中，我们通常还会设置一些提前停止的条件，比如限制树的最大深度，或者限制叶子节点的最小样本数量，这样可以防止过拟合。

第 5 张슬라이드

现在我们来谈第三个步骤：决策树的剪枝(Pruning)。

为什么需要剪枝呢？让我给大家解释一下这个问题。

如果我们让决策树无限制地生长下去，它最终可能会对训练数据拟合得非常好，甚至可能达到100%的准确率。听起来很好，对吧？

但是问题来了！这样的树在面对新的、没见过的测试数据时，表现往往很差。这就是我们常说的"过拟合"现象。

过拟合就像是死记硬背。学生可能把教科书上的每道题都背得滚瓜烂熟，但一遇到新题目就不会做了。

剪枝就是为了解决这个问题。我们的目标是找到一个既能较好地拟合训练数据，又不会过于复杂的决策树。

剪枝的基本思想是：如果一个子树的复杂度很高，但对提高整体准确率的贡献很小，那我们就把这个子树"剪掉"，用一个叶子节点来代替它。

第 6 张슬라이드

我们来看看剪枝的数学表达。

剪枝通常通过最小化整个决策树的损失函数来实现：

C_α(T) = C(T) + α|T|

让我解释一下这个公式的各个部分：

- C(T) 表示模型在训练数据上的预测误差
- |T| 表示树的叶子节点数量（衡量模型复杂度）
- α 是一个平衡参数

这个公式很有意思：第一项 C(T)希望模型准确，第二项 α|T|希望模型简单。参数 α 控制这两者之间的平衡。

- 当 α 比较大时，我们更注重模型的简单性，最优的树会比较小
- 当 α 比较小时，我们更注重模型的准确性，最优的树会比较大

通过调整 α 的值，我们可以得到不同复杂度的最优子树，然后在验证集上测试，选择表现最好的那一个。

第 7 张슬라이드

好了，理论部分我们就讲到这里。现在我们来看一个实际的例子。

我们要使用著名的 iris 数据集来演示决策树算法。Iris 数据集是机器学习中的"Hello World"，几乎每个学机器学习的人都会用到它。

这个数据集的背景是这样的：它包含了 150 个鸢尾花的样本，分为 3 个不同的品种，每个品种有 50 个样本。每个样本有 4 个特征：

1. 萼片长度(Sepal Length)
2. 萼片宽度(Sepal Width)
3. 花瓣长度(Petal Length)
4. 花瓣宽度(Petal Width)

我们的任务是根据这 4 个特征来判断鸢尾花属于哪个品种。

大家可以从这个网址下载数据：http://dblab.xmu.edu.cn/blog/wp-content/uploads/2017/03/iris.txt

这是一个非常经典的分类问题，也是测试分类算法性能的标准数据集。

第 8 张슬라이드

现在我们开始实际的编程实现。我们使用 Apache Spark(阿帕奇火花)的机器学习库 MLlib 来实现决策树算法。

首先，第 1 步是导入我们需要的所有包。我来逐一解释一下：

from pyspark.ml.classification import DecisionTreeClassifier

这个是决策树分类器(Decision Tree Classifier)的主要类。

from pyspark.ml import Pipeline

Pipeline(管道)用于将多个处理步骤串联起来，这样我们就可以一次性完成数据预处理、模型训练和预测。

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

这个用于评估多分类模型的性能。

from pyspark.ml.feature import VectorAssembler

VectorAssembler(向量汇编器)用于将多个特征列合并成一个特征向量列。

from pyspark.sql.types import DoubleType

from pyspark.sql.session import SparkSession

这两个是 Spark SQL(火花 SQL)的基础类型和会话管理。

大家在实际项目中，记得要先导入这些必要的包。

第 9 张슬라이드

第 2 步是读取和预处理我们的数据。

首先，我们创建一个 SparkSession(火花会话)：

spark = SparkSession.builder.appName("example").getOrCreate()

然后指定我们的 CSV 文件路径：

path = "file:///usr/local/spark/iris.data"

接下来使用 Spark(火花)读取 CSV 文件：

df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0", "c1", "c2", "c3", "label")

这里有几个重要的点：

- inferSchema="true" 让 Spark 自动推断数据类型
- toDF() 方法给列重新命名，我们把 4 个特征列命名为 c0 到 c3，把目标列命名为 label

为什么要转换为 Double 类型呢？这是为了处理可能存在的数据质量问题，比如文件末尾的空行等情况。在实际项目中，数据清洗是非常重要的一步。

第 10 张슬라이드

现在我们继续数据预处理的代码。

首先，将所有特征列转换为 Double 类型：

```
df_double = df_raw.select(
    df_raw["c0"].cast(DoubleType()),
    df_raw["c1"].cast(DoubleType()),
    df_raw["c2"].cast(DoubleType()),
    df_raw["c3"].cast(DoubleType()),
    df_raw["label"]
)
```

然后创建 VectorAssembler(向量汇编器)：

assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"], outputCol="features")

最后使用 VectorAssembler 将 4 个特征列组合成一个特征向量：

df = assembler.transform(df_double).select("features", "label")

为什么要这样做呢？因为 Spark MLlib(机器学习库)要求所有的特征必须组合成一个向量列。这是 Spark 的标准做法。

这样我们就得到了一个包含"features"和"label"两列的 DataFrame(数据框)，其中"features"列包含了所有 4 个特征的向量表示。

第 11 张슬라이드

第 3 步是进一步处理特征和标签，并将数据集分割为训练集和测试集。

首先，我们需要处理标签。因为我们的标签是字符串类型（比如"Iris-setosa"），但机器学习算法需要数值类型，所以我们使用 StringIndexer(字符串索引器)：

labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(df)

然后处理特征。虽然我们的特征已经是数值类型，但我们还是使用 VectorIndexer(向量索引器)来处理可能的类别特征：

featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(df)

接下来创建 IndexToString(索引转字符串)，用于将预测结果转换回原始的字符串标签：

labelConverter = IndexToString(inputCol="prediction", outputCol="predictedLabel", labels=labelIndexer.labels)

最后，我们将数据集随机分割为训练集和测试集，比例是 7:3：

trainingData, testData = df.randomSplit([0.7, 0.3])

这是机器学习的标准做法：用训练集训练模型，用测试集评估模型性能。

第 12 张슬라이드

第 4 步是创建决策树模型。

创建 DecisionTreeClassifier(决策树分类器)非常简单：

dtClassifier = DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")

这里我们设置了两个关键参数：

- labelCol 指定标签列
- featuresCol 指定特征列

DecisionTreeClassifier 有很多参数可以调节，比如最大深度、最小信息增益等。我们可以通过explainParams()方法查看所有可用的参数。

也可以使用 ParamMap(参数映射)来设置参数，这在需要同时设置多个参数时特别有用。

在实际项目中，参数调优是提高模型性能的重要手段。不过对于演示目的，我们使用默认参数就足够了。

第 13-15 张슬라이드

让我给大家详细介绍一下 DecisionTreeClassifier 的主要参数。理解这些参数对于调优模型非常重要。

核心参数：

- maxDepth: 树的最大深度。深度越大，模型越复杂，但也容易过拟合
- minInstancesPerNode: 每个节点的最小样本数。这个值越大，模型越保守
- minInfoGain: 分割节点所需的最小信息增益。设置这个值可以防止无意义的分割

特征相关参数：

- maxBins: 用于离散化连续特征的最大箱数。这个值影响算法的精度和速度
- impurity: 分割准则，可以选择"gini"或"entropy"

输出相关参数：

- predictionCol: 预测结果列名
- probabilityCol: 预测概率列名
- rawPredictionCol: 原始预测值列名

高级参数：

- checkpointInterval: 检查点间隔，用于容错
- seed: 随机种子，用于结果的可重复性

在实际应用中，你们需要根据具体问题和数据特征来调整这些参数。记住，没有万能的参数设置，需要通过实验来找到最佳组合。

第 16 张슬라이드

第 5 步是构建机器学习管道(Pipeline)并进行训练和预测。

Pipeline 是 Spark MLlib 的一个重要概念。它允许我们将多个处理步骤串联起来：

dtPipeline = Pipeline(stages=[labelIndexer, featureIndexer, dtClassifier, labelConverter])

我们的管道包含 4 个阶段：

1. 标签索引化
2. 特征索引化
3. 决策树分类
4. 标签转换

然后在训练数据上训练管道：

dtPipelineModel = dtPipeline.fit(trainingData)

最后在测试数据上进行预测：

dtPredictions = dtPipelineModel.transform(testData)

Pipeline 的好处是什么呢？它确保了训练和预测使用相同的数据处理步骤，避免了训练测试不一致的问题。而且 Pipeline 模型可以直接保存和加载，非常方便部署。

第 17 张슬라이드

现在让我们看看预测结果。

我们选择要显示的列：

selected = dtPredictions.select("predictedLabel", "label", "features")

然后显示前 100 行结果：

selected.show(100)

从输出结果我们可以看到：

- predictedLabel: 模型的预测结果
- label: 真实标签
- features: 输入特征

大部分预测都是正确的！比如第一行，真实标签是"Iris-setosa"，预测标签也是"Iris-setosa"。

我们可以看到三种鸢尾花：

- Iris-setosa（山鸢尾）
- Iris-versicolor（变色鸢尾）
- Iris-virginica（维吉尼亚鸢尾）

从这个结果看，我们的决策树模型表现得相当不错！

第 18 张슬라이드

第 6 步是评估模型性能。

我们使用 MulticlassClassificationEvaluator(多分类评估器)来计算准确率：

evaluator = MulticlassClassificationEvaluator(labelCol="indexedLabel", predictionCol="prediction")

dtAccuracy = evaluator.evaluate(dtPredictions)

结果显示：

Model Accuracy: 0.9575250312891115

这意味着我们的模型达到了 95.75%的准确率！这是一个非常好的结果。

在机器学习中，准确率并不是唯一的评估指标。对于不同的问题，我们可能还需要关注精确率(Precision)、召回率(Recall)、F1-score 等指标。但对于这个相对简单的分类问题，95.75%的准确率已经说明我们的决策树模型训练得很成功。

记住，在实际项目中，我们还应该进行交叉验证来更可靠地评估模型性能。

第 19 张슬라이드

最后，让我们来看看我们训练出的决策树到底长什么样。

首先获取训练好的决策树模型：

treeModelClassifier = dtPipelineModel.stages[2]

然后打印决策树的结构：

print("Learned classification tree model:₩n" + treeModelClassifier.toDebugString)

从输出结果我们可以看到：

- 这棵树的深度是 4
- 总共有 9 个节点
- 3 个类别，4 个特征

最有意思的是决策规则：

- 如果特征 2（花瓣长度）≤ 2.7，预测为类别 0（Iris-setosa）
- 否则继续根据其他特征进行判断

这个结果很有道理！在鸢尾花数据集中，Iris-setosa 的花瓣确实比其他两种要短很多，所以花瓣长度成为了最重要的分割特征。

通过查看决策树的结构，我们不仅可以理解模型的预测逻辑，还可以获得对数据的洞察。这就是决策树算法的一个重要优势：可解释性强。

总结

好的，今天我们学习了决策树分类算法的完整流程：

1. **理论基础**：了解了决策树的基本概念和三个关键步骤
2. **数学原理**：掌握了信息增益、信息增益比和基尼指数的计算方法
3. **实际应用**：使用 Spark MLlib 在 iris 数据集上实现了完整的决策树分类

决策树算法的优点：

- 易于理解和解释
- 不需要数据预处理（如标准化）
- 能够处理数值型和类别型特征
- 能够处理缺失值

决策树算法的缺点：

- 容易过拟合
- 对噪声敏感
- 可能产生偏向某些特征的树

下节课我们将学习如何使用集成学习方法（如随机森林）来改善决策树的性能。

大家有什么问题吗？

聚类算法教学讲义 - 中文课堂讲解脚本

슬라이드 1：聚类概述

大家好，今天我们来学习第 9.8 章的内容——聚类算法（Clustering Algorithms）。

首先，我们来了解一下什么是聚类。聚类，也叫群分析（Group Analysis），是机器学习（Machine Learning）和数据挖掘（Data Mining）中非常重要的一种技术。

聚类的目的很简单，就是把数据集中的数据对象分成若干个组，我们叫做簇（Cluster）。我们希望达到什么效果呢？就是同一个簇里面的样本要尽量相似，不同簇之间的样本要尽量不同。

大家可以看到，通过聚类生成的簇必须满足两个基本条件：第一，每个簇至少要包含一个数据对象；第二，每个数据对象只能属于一个簇。这就保证了我们的聚类结果是完整且不重叠的。

슬라이드 2：聚类形式化描述

现在我们来看看聚类算法的数学表示。

给定一组数据集合 D，D 中的每一条记录都是包含若干属性的特征向量，用向量 $x = (x_1, x_2, ..., x_n)$来表示。这里的 x_i 可以有不同的值域，当一个属性的值域为连续域时，我们称为连续属性（Continuous Attribute），否则为离散属性（Discrete Attribute）。

聚类算法的目标就是将数据集 D 划分为 k 个不相交的簇$\{C = c_1, c_2, ..., c_k\}$，其中 $c_i \cap c_j = \emptyset$ 当 $i \neq j$，且 $D = \bigcup_{i=1}^{k} c_i$。

这个公式告诉我们，所有的簇之间没有交集，但它们的并集就是整个数据集。

슬라이드 3：聚类分类及特征

聚类一般属于无监督分类（Unsupervised Classification）的范畴。什么意思呢？就是我们在没有关于分类的先验知识的情况下，按照一定的要求和规律对数据进行区分和分类。

聚类既可以作为一个单独过程，用于找寻数据内部的分布结构，也可以作为分类等其他学习任务的前驱过程。

聚类算法可以分为几大类：

- 划分法（Partitioning Method）
- 层次法（Hierarchical Method）
- 基于密度的方法（Density-Based Method）
- 基于网格的方法（Grid-Based Method）
- 基于模型的方法（Model-Based Method）

这些方法没有统一的评价指标，为什么呢？因为不同聚类算法的目标函数相差很大。有些聚类是基于距离的，比如 K-Means；有些是假设先验分布的，比如 GMM、LDA；有些是带有图聚类和谱分析性质的，比如谱聚类（Spectral Clustering）；还有些是基于密度的，比如 DBSCAN。所以聚类算法应该嵌入问题中进行评价。

슬라이드 4：目录

今天我们主要学习两个重要的聚类算法：

- 9.8.1 K-Means 聚类算法
- 9.8.2 GMM 聚类算法

슬라이드 5：K-Means 算法概述

现在我们开始学习第一个算法——K-Means 聚类算法。

K-Means 是一个迭代求解的聚类算法，属于划分（Partitioning）型的聚类方法。它的基本思想是什么呢？首先创建 K 个划分，然后迭代地将样本从一个划分转移到另一个划分来改善最终聚类的质量。

具体过程是这样的： 第一步，根据给定的 k 值，选取 k 个样本点作为初始划分中心； 第二步，计算所有样本点到每一个划分中心的距离，并将所有样本点划分到距离最近的划分中心；第三步，计算每个划分中样本点的平均值，将其作为新的中心； 第四步，循环进行第二到第三步直至最大迭代次数，或划分中心的变化小于某一预定义阈值。

这个过程听起来是不是很简单？但是效果很好。

슬라이드 6：K-Means 数据集介绍

现在我们用一个具体的例子来演示 K-Means 算法。

我们使用的是 UCI 数据集中的鸢尾花数据（Iris Dataset）。这个数据集很经典，在机器学习教学中经常使用。

Iris 数据的样本容量为 150，有四个实数值的特征，分别代表花朵四个部位的尺寸：花萼长度、花萼宽度、花瓣长度、花瓣宽度。数据还包含该样本对应鸢尾花的亚种类型，共有 3 种亚种类型：setosa、versicolor、virginica。

大家可以看到数据格式，每行包含四个数值特征和一个类别标签。

슬라이드 7：K-Means 第 1 步 - 引入必要的类

现在我们开始实际编程。首先是第一步：引入必要的类。

```
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.clustering import KMeans, KMeansModel
>>> from pyspark.ml.evaluation import ClusteringEvaluator
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.sql.types import DoubleType
>>> from pyspark.sql.session import SparkSession
```

这里我们导入了几个重要的类：

- Vectors 用于处理向量数据
- KMeans 和 KMeansModel 是我们的主要算法类
- ClusteringEvaluator 用于评估聚类效果
- VectorAssembler 用于将多个特征组合成一个向量
- DoubleType 用于数据类型转换

- SparkSession 是我们的 Spark 会话（Spark Session）

슬라이드 8：K-Means 第 2 步 - 创建数据集

第二步：创建数据集。

```
# 创建 SparkSession
>>> spark = SparkSession.builder.appName("example").getOrCreate()
# 加载数据
>>> path = "file:///usr/local/spark/iris.data"
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0", "c1", "c2", "c3", "label")
```

这里我们使用 spark.read.csv()方法读取 CSV 文件。注意，我们设置了"inferSchema"为"true"，这样 Spark 会自动推断数据类型。然后我们用 toDF()方法给列命名：c0, c1, c2, c3 代表四个特征，label 代表类别标签。

슬라이드 9：K-Means 数据转换

接下来我们需要将 4 个属性转化为 Double 类型，最后利用 VectorAssembler()将 4 个属性组合成向量。

```
# 将列的数据类型转换为 Double
>>> df_double = df_raw.select(
    df_raw["c0"].cast(DoubleType()),
    df_raw["c1"].cast(DoubleType()),
    df_raw["c2"].cast(DoubleType()),
    df_raw["c3"].cast(DoubleType()),
    df_raw["label"]
)
```

```
# 创建 VectorAssembler
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"],
outputCol="features")
# 使用 VectorAssembler 转换数据
>>> df = assembler.transform(df_double).select("features")
```

为什么要做这个转换呢？因为 Spark ML 的算法需要所有特征都在一个向量列中。
VectorAssembler 就是帮我们把多个列合并成一个特征向量（Feature Vector）。

슬라이드 10：K-Means 第 3 步 - 模型创建和训练

第三步：数据构建好后，我们就可以创建 Kmeans 实例，并进行参数设置。

```
# 创建 KMeans 模型
>>> kmeans = KMeans().setK(3) # 设置簇的数量
>>> kmeans.setFeaturesCol("features") # 设置特征列
>>> kmeans.setPredictionCol("prediction") # 设置预测列
# 训练 KMeans 模型
>>> kmeansmodel = kmeans.fit(df)
```

这里我们设置 K=3，因为我们知道鸢尾花有 3 个品种。
setFeaturesCol 指定特征列的名称，setPredictionCol 指定预测结果列的名称。最后用 fit()方法训练模型。

슬라이드 11：K-Means 参数说明（表 9-7）

现在我们来详细了解一下 Kmeans()的参数。大家可以看到这个表格，我来逐一解释：

- **featuresCol**：用于指明 DataFrame 中用于存储训练 KMeans 模型的特征列的名称，默认为"features"

- **predictionCol**：用于指明 DataFrame 中用于存储 KMeans 模型的预测结果列的名称，默认为"prediction"
- **k**：用于指明 KMeans 模型形成的簇的个数，默认为 2
- **maxIter**：用于指明 KMeans 模型训练时最大的迭代次数，超过该迭代次数，即使残差尚未收敛，训练过程也不再继续，默认为 20
- **Seed**：用于指明 KMeans（初始化）过程中产生随机数的种子，默认值是使用类名的 Long 类型哈希值

슬라이드 12：K-Means 参数说明（表续）

继续看剩下的参数：

- **tol**：用于指明 KMeans 模型训练时的残差收敛阈值，默认为 10^-4
- **initMode**：用于指明 KMeans 模型训练时寻找初始划分中心的方法，默认为 K-Means||()（即 KMeans++()的并行化版本）
- **InitSteps**：用于指明使用 K-Means||()方法进行初始化时的步数，默认为 2
- **weightCol**：用于设置权重列名，未定义默认值，如果没有设置或将其设置为空，则把所有实例的权重设为 1
- **distanceMeasure**：用于指明计算样本距离的方式，可选"euclidean"和"cosine"

这些参数大部分时候用默认值就可以了，但是了解它们很重要。

슬라이드 13：K-Means 第 4 步 - 聚类预测

第四步：通过 transform()方法对存储在 df 中的数据集进行整体处理，生成带有预测簇标签的数据集。

```
# 进行聚类预测
>>> results = kmeansmodel.transform(df)
# 打印结果
```

```
>>> for result in results.collect():
    print(str(result.features) + " => cluster " + str(result.prediction))
```

大家可以看到结果，每个样本都被分配到了一个簇中。比如[4.6,3.2,1.4,0.2]被分到了 cluster 1，[7.0,3.2,4.7,1.4]被分到了 cluster 0。

这里我们可以观察一下，特征值较小的样本（比如花瓣长度和宽度都比较小的）通常被分到同一个簇，这是符合直觉的。

슬라이드 14 : K-Means 第 5 步 - 聚类中心

第五步：通过 KMeansModel 类自带的 clusterCenters 属性获取模型的所有划分中心情况。

```
# 打印 KMeans 模型的聚类中心
>>> for center in kmeansmodel.clusterCenters():
    print("Clustering Center: {}".format(center))
```

聚类中心（Cluster Center）就是每个簇的"代表点"。从结果可以看到，三个聚类中心的坐标分别是：

- [5.88, 2.74, 4.39, 1.43]
- [6.85, 3.08, 5.72, 2.05]
- [5.01, 3.42, 1.46, 0.24]

注意最后一个中心的第三、四个维度（花瓣长度和宽度）明显比前两个小，这可能对应 setosa 品种。

슬라이드 15 : K-Means 第 6 步 - 模型评估

第六步：使用 spark.ml.evaluation.ClusteringEvaluator()计算 Silhouette 分数来度量聚类的有效性。

```
# 创建聚类评估器
>>> evaluator = ClusteringEvaluator()
```

```
# 计算轮廓系数
>>> silhouette = evaluator.evaluate(results)
>>> print("Silhouette Score:", silhouette)
Silhouette Score:0.7354567373091194
```

Silhouette Score（轮廓系数）是一个重要的聚类评估指标，值域为[-1,1]。越接近 1 表示簇内样本距离越小，不属于同一簇的样本距离越大。我们的结果 0.735 是一个不错的分数。

在 K 值未知的情况下，可利用该值选取合适的 K 值——通常选择使 Silhouette Score 最大的 K 值。

슬라이드 16：GMM 聚类算法标题

现在我们来学习第二个重要的聚类算法——GMM 聚类算法。

슬라이드 17：GMM 理论背景

GMM 是一种概率式的聚类算法，属于生成式模型（Generative Model）类型。它假设所有的数据样本都是由某一个给定参数的多元高斯分布所生成的。

具体地，给定聚类数 K，对于给定样本空间中的样本 x，一个 GMM 的概率密度函数可以由 K 个多元高斯分布组成的混合分布表示，即有：

$p(x) = \sum_{i=1}^{K} w_i \cdot p(x \mid \mu_i, \Sigma_i)$

这里，$p(x \mid \mu, \Sigma)$是以 μ 为均值向量、Σ 为协方差矩阵的多元高斯分布的概率密度函数。可以看出，GMM 由 K 个不同的多元高斯分布共同组成，每一个分布被称为 GMM 中的一个成分（Component），而 w_i 为第 i 个多元高斯分布在混合模型中的权重，且有$\sum_{i=1}^{K} w_i = 1$。

슬라이드 18：GMM 生成过程及 EM 算法

假设存在一个 GMM，那么样本空间中样本的生成过程是：以 $w_1,w_2,...,w_k$作为概率选择出一个混合成分，根据该混合成分的概率密度函数采样出相应的样本。实际上，权重可以被直观理解成相应成分产生的样本占总样本的比例。

利用 GMM 进行聚类的过程便是利用 GMM 生成数据样本的逆过程：给定聚类簇数 K，通过给定的数据集，以某一种参数估计方法，推导出每一个混合成分的参数（即均值向量 μ、协方差矩阵 Σ 和权重 w），每一个多元高斯分布成分即对应聚类后的一个簇。

GMM 在训练时使用了最大似然估计法（Maximum Likelihood Estimation），最大化以下对数似然函数：

$L = \ln \prod_{i=1}^{m} p(x^i)$ $L = \sum_{i=1}^{m} \ln [\sum_{j=1}^{K} w_j \cdot p(x^i \mid \mu_j, \Sigma_j)]$

슬라이드 19：EM 算法具体过程

L 无法直接通过解析方式求得解，故可采用"期望-最大化"（Expectation-Maximization, EM）方法求解。

具体过程如下：

第一步，根据给定的 K 值，初始化 K 个多元高斯分布及其权重；

第二步，根据贝叶斯定理，估计每个样本由每个成分生成的后验概率（EM 方法中的 E 步）；

第三步，根据均值、协方差的定义及第二步中求出的后验概率，更新均值向量、协方差矩阵和权重（EM 方法中的 M 步）；

第四步，重复第二到第三步，直到对数似然函数增加值已小于收敛阈值或达到最大迭代次数。

EM 算法是一个非常经典的算法，在很多机器学习问题中都有应用。

슬라이드 20：GMM 软聚类

参数估计过程完成后，对于每一个样本点，根据贝叶斯定理计算出其属于每一个簇的后验概率，并将样本划分到后验概率最大的簇上去。

相对于 K-
Means 等直接给出样本点的簇划分的聚类算法，GMM 这种给出样本点属于每个簇的概率的聚类算法，被称为软聚类（Soft Clustering / Soft Assignment）。

这是 GMM 相比 K-
Means 的一个重要优势：它不仅告诉我们样本属于哪个簇，还告诉我们属于每个簇的概率，这提供了更丰富的信息。

슬라이드 21：GMM 第 1 步 - 引入需要的包

现在我们开始 GMM 的实际编程。第一步：引入需要的包。

```
>>> from pyspark.ml.clustering import GaussianMixture, GaussianMixtureModel
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.sql.session import SparkSession
```

GMM 在 spark.ml.clustering 包下，具体实现分为两个类：用于抽象 GMM 的超参数并进行训练的 GaussianMixture 类和训练后的模型 GaussianMixtureModel 类。

슬라이드 22：GMM 第 2 步 - 创建数据集

第二步：创建数据集。这个步骤和 K-Means 完全一样。

```
# 创建 SparkSession
>>> spark = SparkSession.builder.appName("example").getOrCreate()
# 加载数据
>>> path = "file:///usr/local/spark/iris.data"
>>> df_raw = spark.read.option("inferSchema", "true").csv(path).toDF("c0", "c1", "c2", "c3", "label")
```

我们还是使用同样的 Iris 数据集，这样便于比较 K-Means 和 GMM 的效果。

슬라이드 23：GMM 数据转换

数据转换步骤也和 K-Means 一样：

```
# 将列的数据类型转换为 Double
>>> df_double = df_raw.select(
    df_raw["c0"].cast(DoubleType()),
    df_raw["c1"].cast(DoubleType()),
    df_raw["c2"].cast(DoubleType()),
    df_raw["c3"].cast(DoubleType()),
    df_raw["label"]
)
# 创建 VectorAssembler
>>> assembler = VectorAssembler(inputCols=["c0", "c1", "c2", "c3"],
outputCol="features")
# 使用 VectorAssembler 转换数据
>>> df = assembler.transform(df_double).select("features")
```

这里的步骤和之前完全一样，所以如果是连续的实验，可以重用之前的数据。

슬라이드 24：GMM 第 3 步 - 模型创建

第三步：数据构建好后，即可创建一个 GaussianMixture 对象，设置相应的超参数，并调用 fit()方法来训练一个 GMM 模型。

```
# 创建 Gaussian Mixture 模型
>>> gm =
GaussianMixture().setK(3).setPredictionCol("Prediction").setProbabilityCol("Probability")
# 训练模型
>>> gmm = gm.fit(df)
```

注意这里我们多设置了一个参数：setProbabilityCol("Probability")。这是因为 GMM 会输出每个样本属于各个簇的概率，我们需要指定存储这些概率的列名。

슬라이드 25：GMM 参数说明（表 9-8）

现在我们来看看 GaussianMixture()的参数。这里建立了一个简单的 GaussianMixture 对象并设定模型参数，设定其聚类数目为 3，其他参数取默认值。

主要参数包括：

- **featuresCol**：用于指明 DataFrame 中用于存储训练 GMM 的特征列的名称，默认为"features"
- **predictionCol**：用于指明 DataFrame 中用于存储 GMM 的预测结果列的名称，默认为"prediction"
- **probabilityCol**：用于指明 DataFrame 中用于存储 GMM 中每个样本的类条件概率（属于每一个簇的概率）向量的列名称，默认为"probability"
- **k**：用于指明 GMM 中独立高斯分布的个数，即其他聚类方法中的簇的个数，默认为 2

这个 probabilityCol 是 GMM 特有的，K-Means 没有这个参数。

슬라이드 26：GMM 参数说明（表续）

继续看其他参数：

- **maxIter**：用于指明 GMM 训练时最大的迭代次数，超过该迭代次数，即使残差尚未收敛，训练过程也不再继续，默认为 100
- **seed**：用于指明 GMM 训练过程中产生随机数的种子，默认值是使用类名的 Long 类型哈希值
- **tol**：用于指明 GMM 训练时的残差收敛阈值，默认为 10^-2
- **weightCol**：用于设置权重列名，未定义默认值，如果没有设置或将其设置为空，则把所有实例的权重设为 1

注意 GMM 的默认 maxIter 是 100，比 K-
Means 的 20 要大，这是因为 EM 算法通常需要更多迭代才能收敛。

슬라이드 27 : GMM 第 4 步 - 聚类预测

第四步：调用 transform()方法处理数据集并进行输出。除了可以得到样本的聚簇归属预测，GMM 还可以得到样本属于各个聚簇的概率（Probability 列）。

```
# 使用 Gaussian Mixture 模型进行聚类预测
>>> result = gmm.transform(df)
# 显示结果
>>> result.show(150, truncate=False)
```

大家可以看到结果表格，有三列：features、Prediction 和 Probability。

Probability 列很有意思，它显示了每个样本属于三个簇的概率。比如第一个样本
[5.1,3.5,1.4,0.2]属于簇 0 的概率接近 1，属于其他两个簇的概率几乎为 0。而有些样本的概率分布就比较均匀，说明分类的不确定性比较大。

슬라이드 28 : GMM 第 5 步 - 模型参数查看

第五步：得到模型后即可查看模型的相关参数。与 K-Means 不同，GMM 不直接给出划分中心，而是给出各个混合成分（多元高斯分布）的参数。

```
# 获取聚类簇数
>>> k = gmm.getK()
# 打印每个组件的权重、均值向量和协方差矩阵
>>> for i in range(k):
    print("Component {}: ".format(i))
    print("Weight: {}".format(gmm.weights[i]))
    print("Mu Vector: ₩n{}".format(gmm.gaussians[i].mean))
```

print("Sigma Matrix: ₩n{}".format(gmm.gaussians[i].cov))

GaussianMixtureModel 类的 weights 成员获取各个混合成分的权重，gaussians 成员获取各个混合成分。其中，GMM 的每一个混合成分都使用一个 MultivariateGaussian 类来存储，可以通过 gaussians 成员来获取各个混合成分的参数（均值向量和协方差矩阵）。

슬라이드 29：GMM Component 0 和 1 结果

现在我们来看具体的结果。

Component 0 的权重是 0.125，比较小，说明这个成分产生的样本相对较少。它的均值向量是[6.72, 2.80, 5.23, 1.57]，协方差矩阵显示了各个特征之间的相关性。

Component
1 的权重是 0.541，最大，说明这个成分产生了一半以上的样本。它的均值向量是[6.16, 2.89, 4.83, 1.70]。

从均值向量可以看出，这两个成分的花瓣长度和宽度都比较大，可能对应 versicolor 和 virginica 两个品种。

슬라이드 30：GMM Component 1 详细结果

让我们详细看看 Component 1 的结果：

权重：0.541 均值向量：[6.16, 2.89, 4.83, 1.70] 协方差矩阵是一个 4x4 的矩阵，对角线元素是各个特征的方差，非对角线元素是特征之间的协方差。

协方差矩阵告诉我们特征之间的相关性。比如第三个和第四个特征（花瓣长度和宽度）之间有比较强的正相关性，这符合生物学直觉。

슬라이드 31：GMM Component 0 和 1 对比

我们来对比一下 Component 0 和 Component 1：

Component 0:

- 权重较小(0.125)

- 均值向量显示花瓣较大[6.72, 2.80, 5.23, 1.57]

Component 1:

- 权重最大(0.541)
- 均值向量[6.16, 2.89, 4.83, 1.70]

两个成分的特征都比较相似，可能对应了两个比较相近的花朵品种。

슬라이드 32：GMM Component 2 结果

最后来看 Component 2：

权重：0.333 均值向量：[5.01, 3.42, 1.46, 0.24]

这个成分很有意思！大家注意看它的均值向量，特别是第三、四个维度（花瓣长度和宽度）：1.46 和 0.24，明显比前两个成分小得多。这很可能对应 setosa 品种，因为 setosa 的花瓣确实比较小。

协方差矩阵的对角线元素也都比较小，说明这个成分的特征比较集中，变异性较小。

这个结果很好地验证了我们的 GMM 模型！三个成分很可能分别对应了三个不同的鸢尾花品种。

总结

今天我们学习了两个重要的聚类算法：

K-Means 聚类算法

K-Means 是一个经典的硬聚类算法（Hard Clustering）：

- **优点**：简单易懂，计算效率高，适合大数据集
- **缺点**：需要预先指定 K 值，对初始中心敏感，假设簇是球形的
- **适用场景**：当我们知道大概的簇数量，且数据分布比较规律时

GMM 聚类算法

GMM 是一个概率式的软聚类算法（Soft Clustering）：

- **优点**：给出概率信息，更灵活，可以处理椭圆形的簇
- **缺点**：计算复杂度高，需要更多参数，容易过拟合
- **适用场景**：当我们需要知道不确定性信息，或者数据分布比较复杂时

实践建议

在实际应用中：

1. **如果追求简单快速**，先试 K-Means
2. **如果需要概率信息**，使用 GMM
3. **如果不知道 K 值**，可以用轮廓系数（Silhouette Score）来选择
4. **数据预处理很重要**：标准化、去异常值等

Spark ML 的优势

使用 Spark ML 进行聚类有几个好处：

- **分布式计算**：可以处理大规模数据
- **统一接口**：所有算法都使用相似的 API
- **管道支持**：可以很容易地与其他 ML 组件组合
- **自动并行化**：不需要手动管理并行计算

希望通过今天的学习，大家对聚类算法有了更深入的理解。下节课我们将学习更多高级的机器学习算法。

有什么问题吗？

课后练习建议

1. **尝试不同的 K 值**：在 Iris 数据集上尝试 K=2,3,4,5，观察轮廓系数的变化
2. **比较两种算法**：在同一数据集上比较 K-Means 和 GMM 的结果
3. **参数调优**：尝试调整 maxIter、tol 等参数，观察对结果的影响

4. **可视化**：如果可能的话，将聚类结果可视化，更直观地理解算法效果
5. **其他数据集**：在不同的数据集上测试这两种算法，比较它们的适用性

记住，聚类是一个探索性的过程，没有标准答案。重要的是理解算法原理，并能根据具体问题选择合适的方法。

好，今天的课就到这里。大家辛苦了！

频繁模式挖掘算法 - 中文课堂讲授脚本

第 1 张슬라이드：频繁模式挖掘算法概述

大家好，今天我们来学习一个很重要的数据挖掘技术 - 频繁模式挖掘算法。

首先，什么是频繁模式挖掘呢？简单来说，就是从大量的交易数据中找出经常一起出现的项目组合。比如说，我们去超市购物，经常会看到买面包的人也会买牛奶，买啤酒的人也会买薯片。这种关联性就是我们要挖掘的频繁模式。

在学术上，我们也叫它关联规则挖掘。这是数据挖掘分析过程中的一个重要环节。我们的目标是从各种数据库中找到频繁模式、关联性或者因果结构。

大家看这个公式：X ⇒ Y。这就是一个典型的关联规则，意思是如果交易中包含模式 X，那么这个交易很有可能也包含模式 Y。比如说，"如果买了面包，那么很可能也会买牛奶"。

这里有几个重要的条件：X 和 Y 都是项目集合的子集，而且它们之间没有交集，都不能是空集。

第 2 张슬라이드：频繁模式挖掘的形式化描述

现在我们用更正式的方式来描述频繁模式挖掘。

假设我们有一个包含交易的数据库 DB，里面有很多交易 T1, T2, 一直到 Tn。每个交易 Ti 都是项目集合 I 的子集。

这里有两个关键概念：

1. **支持度(Support)**：这是指数据库中包含模式 A 的交易数量，我们用 sup(A)来表示。
2. **最小支持度阈值**：如果 sup(A)大于等于这个阈值 ξ，我们就说 A 是频繁模式。

频繁模式挖掘的目标很明确，就是找出数据库中所有的频繁模式。

找到频繁模式后，我们还要生成关联规则。这就需要用到置信度计算函数 conf 和置信度阈值 ξc。我们把项目数大于 1 的频繁模式分为两个不相交的子集 X 和 Y，然后计算规则置信度 conf(X ⇒ Y)。如果这个置信度大于等于 ξc，我们就保留这个规则。

第 3 张슬라이드：算法目录

好，今天我们主要学习两个算法：

1. **FP-Growth 算法** - 这是一个经典的频繁模式挖掘算法
2. **PrefixSpan 算法** - 这个专门用于挖掘序列数据中的频繁模式

让我们先从 FP-Growth 算法开始。

第 4 张슬라이드：FP-Growth 算法介绍

FP-Growth 算法有两个核心组成部分：

第一个是**频繁模式树(Frequent Pattern Tree)**，简称 FP-Tree。这是一种扩展的前缀树结构，是对数据库的压缩表示。这个树状结构不仅保留了数据库中的项目集，还记录了项目集之间的关联。这种压缩形式的好处是什么呢？它能够避免在寻找频繁模式时多次扫描数据库，大大提高了效率。

第二个是**模式片段增长挖掘(Pattern Fragment Growth Mining)**算法。这是一种运用分治思想扫描 FP-Tree 获取频繁模式的算法。

这两个部分配合起来，就形成了完整的 FP-Growth 算法。

第 5 张슬라이드：FP-Tree 构建过程

现在我们详细看看 FP-Tree 是怎么构建的。

FP-Tree 的构建是通过将每个项目集逐一映射到树中的一个路径来实现的，频繁项目就是路径上的节点。在树上，频繁出现的节点比不频繁出现的节点有更大的机会被共享。

构建过程分为两个步骤：

第一步：扫描一次交易数据库 DB，收集频繁项目集 F 和项目对应的支持度。将 F 中的频繁项目按支持度降序排列得到 L，即排序的频繁项目列表。

第二步：创建 FP-Tree 的根 T，并标记为"null"。对 DB 中的每个交易 t 执行以下步骤： ① 根据 L 的顺序，选择 t 中的频繁项目重排序 ② 记 t 中已排序的频繁项目列表为[p, P]，其中 p 是第一个元素，P 是列表剩余的元素 ③ 然后调用函数 insert_tree([p, P], T)

这个 insert_tree 函数的功能是：如果 T 有一个子节点 N 与 p 同属一个项目，则该子节点 N 计数加 1；否则创建一个（与 p 同属一个项目的）子节点 N 并将其计数置 1。
如果 P 非空，则递归调用 insert_tree(P, N)。

第 6 张슬라이드：FP-Growth 算法执行过程

FP-Tree 构建完成后，FP-Growth 通过模式片段增长挖掘算法找到频繁模式。

这个算法从一个频繁项目开始，将其作为初始后缀模式(Suffix Pattern)，得到其条件模式库(Conditional Pattern Base)。这是一个由与后缀模式共同出现的频繁项目集组成的"子数据库"。

再据此构建其条件 FP-Tree(Conditional FP-Tree)，并使用这样的树递归地进行挖掘。这里的条件 FP-Tree 可以从树根开始到后缀模式（不含）的所有路径组合得到。

模式的增长是通过连接后缀模式与由条件 FP-Tree 产生的新模式来实现的。

定义 FP-Tree Tree，节点 α 与 FP-Growth 算法函数 FP-Growth(Tree,α)，该函数执行步骤如下。

第 7 张슬라이드：FP-Growth 算法详细步骤

具体的执行步骤是这样的：

第一种情况：如果 Tree 只包含一条路径 P，那么对路径 P 的所有节点组合（记为 β）执行如下操作：生成模式 β∩α，且置 sup(β∩α)为 β 中节点的最小支持度。

第二种情况：否则，对 Tree 中每一个项目第一次加入 Tree 的节点 ai 执行如下操作： ① 生成模式 β = ai∩α，且置 sup(β) = sup(ai) ② 构建 β 的条件模式库与条件 FP-Tree Treeβ，如果 Treeβ ≠ ∅，则调用 FP-Growth(Treeβ, β)

通过调用 FP-Growth(Tree, null)即可搜索数据库中的所有频繁模式。

这个算法的巧妙之处在于使用了分治的思想，将大问题分解成小问题来解决。

第 8 张슬라이드：Spark 中的 FP-Growth 实现

现在我们来看看在实际编程中怎么使用 FP-Growth 算法。

在 Spark (Apache Spark)中，FP-Growth()函数位于 pyspark.ml.fpm 包下。下面的例子采用 Spark 自带的 sample_fpgrowth 数据集，在 Spark 的安装目录下可以找到该文件。

文件路径是：/usr/local/spark/data/mllib/sample_fpgrowth.txt

这个文件的格式很简单：每行代表一次交易，交易中的项目以空格分开。在我们的例子中，我们设置最小支持度为 0.5、置信度阈值为 0.6。

第 9 张슬라이드：FP-Growth 实现 - 第 1 步

好，现在我们开始实际编程。

第 1 步：引入需要的包

```
>>> from pyspark.ml.fpm import FPGrowth
>>> from pyspark.sql import SparkSession
```

这里我们导入两个重要的模块：

- FPGrowth：这是我们要使用的频繁模式挖掘算法
- SparkSession：这是 Spark (Apache Spark)的入口点

第 10 张슬라이드：FP-Growth 实现 - 第 2 步

第 2 步：读取 sample_fpgrowth 数据集中的每一行，并用空格分割其中的项目，将分割出的项目转化成列表

```
# 创建 SparkSession
>>> spark = SparkSession.builder ₩
    .appName("example") ₩
    .getOrCreate()
```

```
# 加载文本数据并创建 DataFrame
>>> data = spark.sparkContext.textFile(
    "file:///usr/local/spark/data/mllib/sample_fpgrowth.txt") ₩
    .map(lambda t: (t.split(" "),)) ₩
    .toDF(["items"])
```

这段代码的作用是：

1. 首先创建一个 SparkSession (Apache Spark 会话)
2. 然后读取文本文件
3. 使用 map 函数将每行按空格分割
4. 最后转换成 DataFrame (数据框架)，列名为"items"

第 11 张슬라이드：DataFrame 输出结果

让我们看看读取的数据是什么样子的：

```
# 显示 DataFrame
>>> data.show(truncate=False)
+------------------------+
|items                   |
+------------------------+
|[r, z, h, k, p]         |
|[z, y, x, w, v, u, t, s]|
|[s, x, o, n, r]         |
|[x, z, y, m, t, s, q, e]|
|[z]                     |
```

|[x, z, y, r, q, t, p] |

+------------------------+

可以看到，每一行都是一个项目列表，代表一次购物交易。比如第一行[r, z, h, k, p]表示这次交易买了 r、z、h、k、p 这 5 个商品。

第 12 张슬라이드：FP-Growth 实现 - 第 3 步

第 3 步：数据构建好后，即可创建 FP-Growth 模型，并进行参数设置

```
# 创建 FPGrowth 模型
>>> fpgrowth = FPGrowth() ₩
    .setItemsCol("items") ₩
    .setMinSupport(0.5) ₩
    .setMinConfidence(0.6)

# 拟合模型到数据集
>>> model = fpgrowth.fit(data)
```

这里我们设置了三个重要参数：

- setItemsCol("items")：指定包含交易项目的列名
- setMinSupport(0.5)：设置最小支持度为 50%
- setMinConfidence(0.6)：设置最小置信度为 60%

然后用 fit 方法训练模型。

第 13 张슬라이드：FPGrowth()参数说明

让我给大家详细解释一下 FPGrowth()函数的各个参数：

参数	含义
itemsCol	用于指明 DataFrame (数据框架)中用于存储训练 FP-Growth 模型的特征列的名称，默认为"items"
minConfidence	用于指明生成关联规则的最小置信度，默认为 0.8
minSupport	用于指明频繁模式的最小支持度，取值属于[0.0,1.0]。任何出现次数超过 minSupport * size-of-the-dataset 的模式将被输出到频繁项集，默认为 0.3
predictionCol	用于指明 DataFrame (数据框架)中用于存储 FP-Growth 模型的预测结果列的名称，默认为"prediction"

这些参数的选择会直接影响挖掘结果的质量和数量。

第 14 张슬라이드：FP-Growth 实现 - 第 4 步

第 4 步：输出频繁模式集

```
>>> model.freqItemsets.show()
+------------+----+
|       items|freq|
+------------+----+
|         [s]|   3|
|      [s, x]|   3|
|         [r]|   3|
|         [y]|   3|
|      [y, x]|   3|
|   [y, x, z]|   3|
|      [y, t]|   3|
```

```
|    [y, t, x]|    3|
|[y, t, x, z]|    3|
|    [y, t, z]|    3|
|        [y, z]|    3|
|           [x]|    4|
|        [x, z]|    3|
|           [t]|    3|
|        [t, x]|    3|
|    [t, x, z]|    3|
|        [t, z]|    3|
|           [z]|    5|
+------------+----+
```

이 결과는 매우 흥미롭습니다! 우리는 다음을 볼 수 있습니다:

- 단일 항목 [z]는 5번 나타나며 가장 빈번합니다
- 항목 [x]는 4번 나타납니다
- 많은 항목 조합이 있으며, 예를 들어 [y, x, z]는 3번 나타납니다

이것들이 우리가 찾은 빈번한 패턴입니다.

15번째 슬라이드: FP-Growth 구현 - 5단계

5단계: 생성된 연관 규칙 출력

```
>>> model.associationRules.show()
+----------+----------+----------+------------------+-------+
|antecedent|consequent|confidence|              lift|support|
+----------+----------+----------+------------------+-------+
```

| [t]| [y]| 1.0| 2.0| 0.5|

| [t]| [x]| 1.0| 1.5| 0.5|

| [t]| [z]| 1.0| 1.2| 0.5|

| [y, t, x]| [z]| 1.0| 1.2| 0.5|

| [x]| [s]| 0.75| 1.5| 0.5|

...

这些关联规则告诉我们：

- antecedent（前件）：如果有这个项目组合
- consequent（后件）：那么很可能也有这个项目组合
- confidence（置信度）：这个规则的可信度
- lift（提升度）：这个规则的强度
- support（支持度）：这个规则的普遍性

比如第一条规则[t] → [y]，置信度是 1.0，意思是如果买了 t，100%会买 y。

第 16 张슬라이드：FP-Growth 实现 - 第 6 步

第 6 步：对输入交易应用生成的关联规则，并将结果作为预测值输出

```
>>> model.transform(data).show()
+--------------------+----------+
|                items|prediction|
+--------------------+----------+
|     [r, z, h, k, p]|    [y, x, t]|
|[z, y, x, w, v, u...|        []|
|     [s, x, o, n, r]|    [y, z, t]|
|[x, z, y, m, t, s...|        []|
```

| [z]| [y, x, t]|

|[x, z, y, r, q, t...| [s]|

+--------------------+----------+

这个功能很实用！根据已有的购买项目，系统可以预测客户可能还会买什么。

比如第一行，客户买了[r, z, h, k, p]，系统预测他可能还会买[y, x, t]。这就是推荐系统的基础原理。

第 17 张슬라이드：PrefixSpan 算法介绍

好，现在我们来学习第二个算法 - PrefixSpan 算法。

PrefixSpan 算法用于挖掘序列数据中的频繁序列模式。什么是序列数据呢？序列数据是由若干个数据项目集组成的序列，项目之间有时间上的先后关系。

比如<a(bc)(bd)>，它由 a、bc、bd 三个项目集数据组成。这里的关键是有时间顺序：先有 a，然后是 bc，最后是 bd。

频繁序列就是满足最小支持度要求的子序列。子序列的数学定义是：对子序列 A = {a_1, a_2, ⋯, a_m}和序列 B = {b_1, b_2, ⋯, b_n}，如果存在下标序列 $1 \le i_1 \le i_2 \le \cdots \le i_m \le n$，满足 $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}$, ⋯, $a_m \subseteq b_{im}$，则称 A 是 B 的子序列。

通俗地讲，A 序列的每个元素必须按顺序地是 B 序列的对应元素的子集。

第 18 张슬라이드：PrefixSpan 的核心概念

下面解释 PrefixSpan 算法中的重要概念：前缀、投影、后缀。

前缀的定义：对子序列 A = {a_1, a_2, ⋯, a_m}和序列 B = {b_1, b_2, ⋯, b_n}，如果满足： ① $a_1 = b_1$, $a_2 = b_2$, ⋯, $a_{m-1} = b_{m-1}$ ② $a_m \subseteq b_m$, $m \le n$ ③ (b_m - a_m)中的元素按顺序排列在 a_m后面

则称 A 是 B 的前缀。通俗地讲，前缀要求 A 的前 m-1 个元素都必须等于 B 的前 m-1 个元素，A 的第 m 个元素是 B 的第 m 个元素的前面的一部分。

比如<a>、<ab>、<abc>都是序列<a(bc)(bd)>的前缀，但<ac>不是<a(bc)(bd)>的前缀。

第 19 张슬라이드：投影和后缀概念

投影的定义：给定序列 A、B，且 A 是 B 的子序列，序列 C 是序列 B 的投影，当且仅当： ① C 是 B 的子序列 ② C 包含前缀 A ③ 不存在比 C 更长的序列 D，使 D 是 B 的子序列，并且 D 包含前缀 A

特别注意：前缀这个概念是针对投影而言的。序列 A 是投影 C 的前缀，只需序列 A 是序列 B 的子序列。

比如，<ac>不是序列 B=<a(bc)(bd)>的前缀，却是序列 B 关于<ac>的投影 C=<ac(bd)>的前缀。

每个前缀都有一个对应的后缀，即 B 的投影 C 减去前缀 A 所剩的序列，记作 B/A。如果前缀的最后一项是项目集的一部分，则用一个"_"来占位表示。

我们看表 9-10，这里展示了指定前缀后的序列<a(bc)(bd)>的投影和后缀。

第 20 张슬라이드：PrefixSpan 算法执行过程

PrefixSpan 算法的目标是挖掘出满足给定的最小支持度要求的频繁序列。它采用分而治之的思想，先从长度为 1 的前缀开始，搜索对应的投影数据库得到长度为 1 的前缀对应的频繁序列，再对长度为 2 的前缀进行搜索，不断递归地进行搜索。

以表 9-11 中的序列数据为例，支持度设为 50%。

我们有 4 条序列：

1. <bac>
2. <c(ab)>
3. <a(bc)(bd)>
4. <cf>

算法的第一步是找出长度为 1 的频繁序列。因此，我们需要对表 9-11 中的序列数据进行统计，结果为：<a>:3、<b>:3、<c>:4、<d>:1、<f>:1。

<a>:3 的意思是在 4 条序列中，有 3 条包含了序列<a>。因为支持度是 50%，所以统计值小于 2 的子序列不属于频繁序列。在结果中，<d>和<f>的统计值只有 1，所以得到的长度为 1 的频繁序列为<a>、<b>、<c>。

第 21 张슬라이드：PrefixSpan 分而治之过程

PrefixSpan 的思想是分而治之，由第一步的结果可以将频繁序列的全体集合划分为 3 部分：① 包含前缀<a>的频繁序列 ② 包含前缀<b>的频繁序列 ③ 包含前缀<c>的频繁序列

下面以<a>为例解释递归挖掘的过程，其他节点同理。递归过程需要构造前缀<a>的投影数据库，如表 9-12 所示。

与第一步的做法相同，对<a>的投影数据库中的元素进行统计，结果为：<b>:1、<_b>:1、<c>:2、<d>:1。

这里要特别注意的是，<b>和<_b>是不同的，<b>与前缀<a>不是同一项目集，而<_b>与前缀<a>属于同一项目集。除了<c>外，其他元素都不满足最小支持度的要求，所以第 2 轮得到的前缀为<a>的长度为 2 的频繁序列为<ac>。

根据递归搜索，第 3 轮需要构造出前缀<ac>的投影数据库。<ac>的投影只有一项，就是<bd>，元素<b>和<d>的计数都只有 1，都不满足支持度的要求，递归挖掘结束。前缀为<a>的频繁序列的全集为<a>、<ac>。仿照上述过程，可以依次求出前缀为<b>和前缀为<c>的频繁序列。

第 22 张슬라이드：PrefixSpan 算法流程总结

PrefixSpan 的算法流程总结如下：

(1) 对长度为 1 的前缀进行计数，得到所有的频繁 1 项序列，长度 L=1。

(2) 对于每个长度为 L 的、满足最小支持度要求的前缀进行递归挖掘： ① 如果前缀对应投影数据库为空，则递归返回。 ② 统计对应投影数据库各项的支持度。如果各项支持度都小于阈值，则递归返回。 ③ 将满足支持度阈值要求的各个单项和目前的前缀合并，得到若干个新的前缀。 ④ L=L+1，前缀为第③步得到的新前缀，分别递归执行第②步。

这个算法的优势在于它不需要生成候选序列，直接在投影数据库上操作，效率很高。

第 23 张슬라이드：Spark 中的 PrefixSpan 实现

现在我们来看看在 Spark (Apache Spark)中怎么使用 PrefixSpan 算法。

PrefixSpan 算法位于 pyspark.ml.fpm 包下。下面给出 PrefixSpan 的具体例子，使用上文例子作为输入数据，1 对应 a，2 对应 b，以此类推。序列总共有 4 条，最小支持度设为 0.5，这意味着只有出现频率至少为 2 的子序列才会被选中。

第 24 张슬라이드：PrefixSpan 实现 - 第 1 步

第 1 步：引入需要的包

```
>>> from pyspark.ml.fpm import PrefixSpan
>>> from pyspark.sql import SparkSession
```

和 FP-Growth 类似，我们需要导入 PrefixSpan 算法和 SparkSession (Apache Spark 会话)。

第 25 张슬라이드：PrefixSpan 实现 - 第 2 步

第 2 步：构造输入数据

```
# 创建 SparkSession
>>> spark = SparkSession.builder ₩
    .appName("example") ₩
    .getOrCreate()

# 定义小型测试数据
>>> small_test_data = [
    [[2], [1], [3]],
    [[3], [1, 2]],
```

```
    [[1], [2, 3], [2, 4]],
    [[3], [5]]
]

# 转换为 DataFrame
>>> data = [(seq,) for seq in small_test_data]
>>> df = spark.createDataFrame(data, ["sequence"])
```

这里我们定义了 4 个序列：

- 第一个序列：先有 2，然后 1，最后 3
- 第二个序列：先有 3，然后同时有 1 和 2
- 第三个序列：先有 1，然后同时有 2 和 3，最后同时有 2 和 4
- 第四个序列：先有 3，然后 5

注意这里每个序列都有时间顺序，这是与 FP-Growth 的重要区别。

第 26 张슬라이드：PrefixSpan 实现 - 第 3 步

第 3 步：创建 PrefixSpan 模型，并进行参数设置

```
# 创建 PrefixSpan 模型
>>> model = PrefixSpan() ₩
    .setMinSupport(0.5) ₩
    .setMaxPatternLength(5) ₩
    .setMaxLocalProjDBSize(32000000)
```

这里设置了三个重要参数：

- setMinSupport(0.5)：最小支持度 50%
- setMaxPatternLength(5)：最大模式长度为 5

- setMaxLocalProjDBSize(32000000)：本地投影数据库的最大大小

第 27 张슬라이드：PrefixSpan()参数说明

让我详细解释一下 PrefixSpan()的参数：

参数	含义
MinSupport	用于指定某子序列被认定为频繁序列模式的最小支持度，取值属于[0.0, 1.0]，默认为 0.1
MaxPatternLength	用于指定频繁序列模式的最大长度，默认为 10
MaxLocalProjDBSize	用于指定在开始对投影数据库进行迭代处理之前，数据库中允许的最大项目数，默认为 32000000
SequenceCol	用于指定数据集中 sequence 列的名称（默认为"sequence"），该列中的空行将被忽略

这些参数的设置会影响算法的性能和结果质量。

第 28 张슬라이드：PrefixSpan 实现 - 第 4 步

第 4 步：查找频繁序列模式

```
>>> result = model.findFrequentSequentialPatterns(df)
```

这一步执行实际的序列模式挖掘。PrefixSpan 算法会分析我们的序列数据，找出满足最小支持度要求的所有频繁序列模式。

第 29 张슬라이드：PrefixSpan 实现 - 第 5 步

第 5 步：输出结果

```
>>> result.show()
```

```
+----------+----+
|  sequence|freq|
+----------+----+
|      [[3]]|   4|
|      [[1]]|   3|
|      [[2]]|   3|
|[[1], [3]]|   2|
|[[3], [2]]|   2|
+----------+----+
```

非常好！让我们分析一下这个结果：

- [[3]]出现了 4 次：这意味着在我们的 4 个序列中，都包含元素 3
- [[1]]出现了 3 次：有 3 个序列包含元素 1
- [[2]]出现了 3 次：有 3 个序列包含元素 2
- [[1], [3]]出现了 2 次：有 2 个序列先有 1 后有 3 的模式
- [[3], [2]]出现了 2 次：有 2 个序列先有 3 后有 2 的模式

这些结果告诉我们序列数据中的时间顺序规律。比如[[1], [3]]表示 1 之后经常会出现 3，这在商业应用中可以用来做时序推荐。

课程总结

好，今天我们学习了两个重要的频繁模式挖掘算法：

FP-Growth 算法：

- 主要用于挖掘频繁项目集
- 核心是 FP-Tree (频繁模式树)
- 适用于传统的购物篮分析

- 不考虑项目之间的时间顺序

PrefixSpan 算法：

- 专门用于序列数据的频繁模式挖掘
- 考虑时间顺序关系
- 使用分治思想和投影技术
- 适用于时序数据分析

这两个算法在 Spark (Apache Spark)的 MLlib 中都有实现，可以处理大规模数据。在实际应用中，我们可以根据数据特点选择合适的算法：

- 如果数据没有时间顺序，用 FP-Growth
- 如果数据有明确的时间顺序，用 PrefixSpan

这些算法在电商推荐、市场篮分析、用户行为分析等领域都有广泛应用。

大家有什么问题吗？我们可以讨论一下。

9.10 协同过滤算法（Collaborative Filtering Algorithm）讲课脚本

슬라이드 1：课程概述

大家好！今天我们来学习 Spark 机器学习(Spark Machine Learning)中一个非常重要的主题——**协同过滤算法**(Collaborative Filtering Algorithm)。

今天我们要讲的内容主要分为两个部分。第一部分是理解推荐算法的原理(Recommendation Algorithm Principles)，第二部分是学习 **ALS 算法**(Alternating Least Squares Algorithm)的具体实现方法。

协同过滤是目前 Netflix、Amazon，还有我们熟悉的淘宝等电商平台广泛使用的推荐系统核心技术。通过这项技术，我们可以精准地向用户推荐他们可能喜欢的商品或内容。

슬라이드 2：推荐算法的原理（第一部分）

好，那我们先来看看协同过滤算法的基本原理。

协同过滤算法主要分为两种类型。

第一种是**基于用户的协同过滤**(User-Based Collaborative Filtering)。这种方法的核心思想是什么呢？就是"喜好相似的人会喜欢相似的东西"这个假设。比如说，如果用户 A 和用户 B 对电影的评分模式非常相似，那么 A 喜欢的电影就可以推荐给 B。

第二种是**基于物品的协同过滤**(Item-Based Collaborative Filtering)。这种方法反过来关注物品之间的相似性。逻辑是"喜欢这个商品的人也会喜欢那个商品"。

但是在 Spark 的 **MLlib**(Machine Learning Library)中，使用的是稍微不同的方法。就是**基于模型的协同过滤**(Model-based Collaborative Filtering)，它利用**隐语义因子**(Latent Semantic Factors)来表示用户和物品，并通过这个来预测缺失的评分。

슬라이드 3：推荐算法的原理（第二部分）

现在我们再了解一个重要概念，就是用户反馈(User Feedback)的类型。

在实际构建推荐系统时，我们可以把用户的反馈分为两种类型。

第一种是**显性反馈**(Explicit Feedback)。这是用户直接表达自己偏好的行为。比如说，给电影打 1 到 5 分的评分，或者点击"喜欢"、"不喜欢"按钮等等。

第二种是**隐性反馈**(Implicit Feedback)。这是用户虽然没有直接表达偏好，但通过行为间接体现出来的。比如说，网页浏览(Browsing)、点击(Click)、购买(Purchase)、点赞(Like)、分享(Share)等等。

实际上，获取隐性反馈比显性反馈要容易得多。因为用户不会对所有商品都打分，但是点击、购买这些行为数据可以自动收集。

传统的**矩阵分解**(Matrix Factorization)基础协同过滤主要使用显性反馈数据。

슬라이드 4：ALS 算法概述

现在我们来正式学习今天的核心——**ALS 算法**。

ALS 是 **Alternating Least Squares** 的缩写，中文叫做**交替最小二乘法**。

这个方法的基本思想是什么呢？就是**矩阵分解**(Matrix Factorization)。

大家想象一下，你们在运营一个网上商城。有一个用户和商品之间的评分矩阵(Rating Matrix)。但是这个矩阵大部分地方都是空的。为什么呢？因为一个用户不可能购买所有商品并且打分。

ALS 的目标就是把这个大的评分矩阵分解成两个小矩阵。一个是**用户对隐含特征的偏好矩阵**，另一个是**物品包含的隐含特征矩阵**。

通过这个过程，我们可以预测原来空缺的评分，并且基于此向用户提供个性化推荐。

슬라이드 5：ALS 算法的数学原理

现在我们稍微深入一些技术层面。有数学出现了，大家别害怕！理解概念就行。

假设我们有一个叫 **R** 的用户-物品评分矩阵，大小是 m×n。这里 m 是用户数，n 是物品数。

ALS 把这个 **R** 矩阵分解成两个低维矩阵 **P**(m×k)和 **Q**(n×k)。这里 k 是**隐含因子**(Latent Factors)的个数，它比 m 和 n 小得多。

用公式表示就是：**R ≈ P × Q^T**

这就是**降维**(Dimensionality Reduction)的概念。**P** 和 **Q** 也叫做**低秩矩阵**(Low-rank Matrix)。

问题是怎么找到这两个矩阵呢？这里就出现了**损失函数**(Loss Function)的概念。

我们的目标是最小化 **L(P,Q)这个损失函数。这个函数测量实际评分和预测评分之间的差异，同时还包含正则化项**(Regularization Term)来防止**过拟合**(Overfitting)。

슬라이드 6：ALS 算法的求解过程

好，现在我们来具体看看 ALS 是怎么解决问题的。

ALS 的核心思想是"**交替优化**"。什么意思呢？

原来的问题是要同时优化 **P** 和 **Q** 两个矩阵，这是个复杂的问题。但是 ALS 把它分成两个简单的问题。

第一步：固定 **Q**，只优化 **P**。 **第二步**：固定 **P**，只优化 **Q**。

重复这个过程，逐渐收敛到最优解。

数学上用**偏导数**(Partial Derivative)。把偏导数设为 0，可以得到**解析解**(Analytical Solution)。

슬라이드上的公式就是每一步更新 **P** 和 **Q** 的公式。这里 **I** 是**单位矩阵**(Identity Matrix)，**λ** 是**正则化参数**(Regularization Parameter)。

通过这种方式，可以高效地解决复杂的优化问题。

슬라이드 7：实验准备

现在理论学得差不多了，我们用 Spark 来实际实现 ALS！

Spark 的 **spark.ml** 包里已经实现了 ALS，我们不需要自己写数学公式。是不是很方便？

今天的实验我们用 **MovieLens 数据集**。这是电影推荐系统中最著名的基准数据集之一。Spark 里已经包含了样本数据，不用单独下载。

数据文件路径如下： /usr/local/spark/data/mllib/als/sample_movielens_ratings.txt

这个文件的每一行包含用户 ID、电影 ID、评分和时间戳。

我们会用基本的**ALS.train()**方法来构建推荐模型，并且评估模型性能。

슬라이드 8：开始编程 - 导入包

好，现在开始正式编程。首先要导入需要的包。

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import SparkSession
from pyspark.sql import Row
```

我来解释一下各个包的作用：

- **RegressionEvaluator**：评估回归模型性能的工具。我们用它来计算 **RMSE**(Root Mean Square Error)。
- **ALS**：今天的主角，交替最小二乘法算法。
- **SparkSession**：Spark 应用程序的入口点。
- **Row**：把数据组织成行(Row)形式的时候用。

这样导入完成后，基本准备工作就完成了。

슬라이드 9：定义数据结构

现在我们来定义处理数据的结构。

首先创建 **SparkSession**：

```
spark = SparkSession.builder.appName("example").getOrCreate()
```

然后定义 **Rating 类**。这个类表示电影评分数据的结构：

```
class Rating:
    def __init__(self, userId, movieId, rating, timestamp):
        self.userId = userId
```

```
        self.movieId = movieId
        self.rating = rating
        self.timestamp = timestamp
```

这个类有 4 个字段：

- **userId**：用户 ID
- **movieId**：电影 ID
- **rating**：评分（通常是 1-5 之间的值）
- **timestamp**：评分时间

使用这种结构化数据，后面转换成 **DataFrame** 就容易多了。

슬라이드 10：数据解析和加载

现在我们来读取和处理实际的数据文件。

首先定义 **parseRating** 函数：

```
def parseRating(line):
    fields = line.split("::")
    assert len(fields) == 4
    return Rating(int(fields[0]), int(fields[1]), float(fields[2]), int(fields[3]))
```

这个函数接收文本文件的一行，转换成 Rating 对象。MovieLens 数据用"::"分隔，所以用 split("::")。

然后读取文件，生成 **DataFrame**：

```
ratings = spark.sparkContext ₩
    .textFile("file:///usr/local/spark/data/mllib/als/sample_movielens_ratings.txt") ₩
    .map(parseRating) ₩
```

.map(lambda x: Row(userId=x.userId, movieId=x.movieId, rating=x.rating, timestamp=x.timestamp)).toDF()

这段代码按以下步骤进行：

1. **textFile()**：读取文本文件，转换成 **RDD**(Resilient Distributed Dataset)
2. **map(parseRating)**：把每一行转换成 Rating 对象
3. **map(lambda ...)**：把 Rating 对象转换成 Row 对象
4. **toDF()**：最终转换成 DataFrame

슬라이드 11 : 数据确认

我们来确认数据是否正确加载了。

ratings.show()

执行这个命令会看到如下结果：

```
+------+-------+------+----------+
|userId|movieId|rating| timestamp|
+------+-------+------+----------+
|     0|      2|   3.0|1424380312|
|     0|      3|   1.0|1424380312|
|     0|      5|   2.0|1424380312|
...
```

这里可以观察到几个特点：

1. **userId** 从 0 开始。
2. **rating** 是 1.0 到 5.0 之间的值。
3. **timestamp** 是 Unix 时间戳格式。
4. 可以看到同一个用户(userId=0)对多部电影评分。

DataFrame 的结构按照我们想要的方式构建好了。默认只显示前 20 行，所以会出现"only showing top 20 rows"的信息。

슬라이드 12 : 数据分割

现在进行机器学习中最重要的步骤之一——**数据分割**(Data Splitting)。

training, test = ratings.randomSplit([0.8, 0.2])

这段代码把全部数据随机分割为**训练集**(Training Set) 80%和**测试集**(Test Set) 20%。

为什么要这样分割数据呢？

- **训练集**：用来训练模型。ALS 算法看这些数据来学习用户和物品的隐含特征。
- **测试集**：用来评估模型性能。用模型从未见过的数据来测量预测准确度。

这样分割的原因是为了防止**过拟合**(Overfitting)，准确评估模型的泛化性能。

如果用训练数据来评估，模型可能"记住"了数据，结果会比实际性能好。

슬라이드 13 : 构建 ALS 模型

现在我们来创建两种不同类型的 ALS 模型。

```
# 创建显式评分的 ALS 模型
alsExplicit = ALS(maxIter=5, regParam=0.01, userCol="userId",
                  itemCol="movieId", ratingCol="rating")

# 创建隐式评分的 ALS 模型
alsImplicit = ALS(maxIter=5, regParam=0.01, implicitPrefs=True,
                  userCol="userId", itemCol="movieId", ratingCol="rating")
```

我来解释两个模型的区别：

显式模型 (Explicit Model)：

- 直接使用用户评分。
- 评分越高表示偏好越高。

隐式模型 (Implicit Model)：

- implicitPrefs=True 是关键参数。
- 把评分解释为**置信度**(Confidence)而不是偏好。
- 有评分本身就是"喜欢"的信号，评分大小表示置信度。

共同参数：

- **maxIter=5**：最多迭代 5 次
- **regParam=0.01**：正则化参数（防止过拟合）
- **userCol, itemCol, ratingCol**：分别是用户、物品、评分列的名称

슬라이드 14 : ALS 参数详细说明

ALS 有很多参数。在实际项目中适当调整这些参数很重要。我们来看看主要参数：

核心参数：

- **rank (默认 10)**：隐含因子个数。越高可以学习复杂模式，但过拟合风险增加。
- **maxIter (默认 10)**：最大迭代次数。
- **regParam (默认 0.1)**：正则化强度。越高越能防止过拟合。

数据相关参数：

- **userCol/itemCol (默认"user"/"item")**：用户和物品 ID 列名称。
- **ratingCol (默认"rating")**：评分列名称。
- **predictionCol (默认"prediction")**：预测结果列名称。

性能相关参数：

- **numUserBlocks/numItemBlocks (默认 10)**：并行处理的块数。

- **checkpointInterval (默认 10)**：检查点保存间隔。

特殊参数：

- **implicitPrefs (默认 false)**：是否启用隐式反馈模式。
- **alpha (默认 1.0)**：隐式反馈模式中的置信度基准线。
- **nonnegative (默认 false)**：是否使用非负约束。

슬라이드 15：模型训练

现在我们来实际训练两个模型。

```
# 使用训练数据拟合显式评分的 ALS 模型
modelExplicit = alsExplicit.fit(training)


# 使用训练数据拟合隐式评分的 ALS 模型
modelImplicit = alsImplicit.fit(training)
```

fit()方法执行期间发生了什么呢：

1. **初始化**：随机初始化用户和物品的隐含特征向量。
2. **迭代优化**：
    - 固定物品特征，优化用户特征
    - 固定用户特征，优化物品特征
    - 重复这个过程 maxIter 次
3. **收敛**：损失函数不再下降时停止学习。

Spark 的优点是整个过程都是**分布式处理**(Distributed Processing)。即使大规模数据也能高效学习。

训练完成后，modelExplicit 和 modelImplicit 对象就创建好了。它们包含学习到的用户和物品隐含特征。

슬라이드 16：执行预测

用训练好的模型对测试集进行预测。

```
# 使用显式评分的 ALS 模型进行预测并移除 NaN 值
predictionsExplicit = modelExplicit.transform(test).na.drop()

# 使用隐式评分的 ALS 模型进行预测并移除 NaN 值
predictionsImplicit = modelImplicit.transform(test).na.drop()
```

这里有个重要要点：

transform()方法：

- 用训练好的模型对新数据进行预测。
- 对每个(用户，物品)对计算预测评分。

na.drop()方法的必要性： 测试集可能包含训练集中没有的用户或物品。这种情况下，模型没有学习到该用户或物品的隐含特征，无法预测。

比如：

- 新用户评价了电影
- 对新电影有评分

这种情况会返回 **NaN**(Not a Number)值，用 na.drop()删除这些行。

实际运营环境中，需要单独的策略来解决这种**冷启动问题**(Cold Start Problem)。

슬라이드 17：显式模型预测结果

我们来看看显式模型的预测结果。

```
predictionsExplicit.show()
```

结果如下：

```
+------+-------+------+----------+----------+
|userId|movieId|rating| timestamp|prediction|
+------+-------+------+----------+----------+
|    12|     22|   2.0|1424380312| 3.2479236|
|    12|     35|   5.0|1424380312| -0.530218|
...
```

这里可以观察到几个有趣的点：

预测准确度分析：

- 用户 12 对电影 22 实际评了 2.0 分，模型预测 3.25 分。差距约 1.25 分。
- 用户 12 对电影 35 实际评了 5.0 分，模型预测-0.53 分。这是相当大的误差。

预测值范围：

- 显式模型的预测值可能超出原始评分范围(1-5)。可能出现负数或大于 5 的值。
- 这是矩阵分解特性上的自然现象。

模型解释：

- 可以看到有些预测准确，有些不准确。
- 这就是为什么模型评估很重要的原因。

슬라이드 18：隐式模型预测结果

现在我们来看隐式模型的结果。

predictionsImplicit.show()

结果：

```
+------+-------+------+----------+------------+
|userId|movieId|rating| timestamp| prediction|
+------+-------+------+----------+------------+
```

| 12| 22| 2.0|1424380312| 0.22080728|

| 12| 35| 5.0|1424380312| 0.16498214|

...

与**显式模型的区别**：

1. **预测值范围**：隐式模型的预测值大部分在 0 和 1 之间的小值。
2. **解释方式**：
    - 显式模型：预测值直接代表评分
    - 隐式模型：预测值代表用户偏好概率或置信度
3. 稳**定性**：隐式模型的预测值在更稳定的范围内。

实际应用角度：

- 显式模型："这个用户会给这部电影打几分？"
- 隐式模型："这个用户喜欢这部电影的概率是多少？"

两种方法在实际推荐系统中都很有用。

슬라이드 19：模型评估

最后我们来定量评估两个模型的性能。

```
# 创建回归评估器
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
                                predictionCol="prediction")

# 使用评估器计算显式评分模型的 RMSE
rmseExplicit = evaluator.evaluate(predictionsExplicit)

# 使用评估器计算隐式评分模型的 RMSE
```

```
rmseImplicit = evaluator.evaluate(predictionsImplicit)


print("RMSE for Explicit Model: ", rmseExplicit)
print("RMSE for Implicit Model: ", rmseImplicit)
```

结果解释：

RMSE for Explicit Model: 1.7802848438813463

RMSE for Implicit Model: 1.8752274536739975

什么是 RMSE (Root Mean Square Error)？

- 预测值和实际值之间的均方根误差。
- 值越小模型越准确。
- 和原始数据有相同的单位（这里是评分单位）。

结果分析：

1. **显式模型**：RMSE = 1.78
    - 平均约有 1.78 分的误差。
2. **隐式模型**：RMSE = 1.88
    - 比显式模型误差稍高。

性能考虑事项：

- 这个例子中数据相对较少，所以误差较大。
- 实际大规模数据集中可以期待更好的性能。
- 评分范围是 1-5，1.8 分左右的误差还有改进空间。

模型改进方向：

1. **更多数据**：更多用户和物品数据
2. **超参数调优**：调整 rank, regParam, maxIter 等
3. **特征工程**：利用额外的用户或物品特征

4. **集成方法**：使用多个模型的组合

课程总结

今天我们深入学习了**协同过滤**(Collaborative Filtering)和 **ALS 算法**(Alternating Least Squares)。

核心内容总结：

1. **协同过滤的基本原理**：
    - 基于用户 vs 基于物品的方法
    - 显式 vs 隐式反馈
2. **ALS 算法的数学原理**：
    - 矩阵分解(Matrix Factorization)
    - 交替优化(Alternating Optimization)
    - 正则化(Regularization)
3. **Spark 中的实际实现**：
    - 数据预处理
    - 模型训练
    - 预测和评估

实务应用建议：

- 实际项目中需要更多数据和超参数调优。
- 需要针对冷启动问题(Cold Start Problem)制定单独策略。
- 为了实时推荐，还要考虑模型服务(Model Serving)架构。

下一步：

- 学习更复杂的推荐算法（基于深度学习的推荐系统）
- 通过 A/B 测试评估推荐系统

- 利用实时流数据进行在线学习

基于今天学的内容，大家也可以构建自己的推荐系统！

如果有问题，随时可以询问。大家辛苦了！

Spark MLlib 模型选择 - 中文讲课脚本

第 1 张슬라이드：标题页

大家好！今天我们要学习 Spark MLlib 中一个非常重要的主题——模型选择(Model Selection)。我们今天的内容分为两个部分：首先是模型选择工具，然后是如何用交叉验证来选择模型。

模型选择在机器学习中是一个关键环节，它直接影响到我们最终模型的性能。让我们开始吧！

第 2 张슬라이드：模型选择概述

现在让我们来了解一下什么是模型选择。

大家看，在机器学习中，模型选择是一个非常重要的任务。简单来说，就是我们要用数据来找到解决具体问题的最佳模型和参数。这个过程我们通常叫做调优(Tuning)。

调优可以在两个层面进行：第一，可以在独立的估计器(Estimator)中完成，比如逻辑回归；第二，也可以在包含多种算法、特征工程和其他步骤的流水线(Pipeline)中完成。

这里有一个重要的实践原则：我们应该一次性调优整个流水线，而不是独立地调优流水线中的每个组成部分。为什么呢？因为各个组件之间可能存在相互影响，整体优化往往比局部优化效果更好。

第 3 张슬라이드：模型选择工具介绍

好，现在我们来看看 MLlib 为我们提供了哪些模型选择工具。

MLlib 支持两个主要的模型选择工具：第一个是交叉验证器(CrossValidator)，第二个是训练-验证切分(TrainValidationSplit)。

要使用这些工具，我们需要准备三个东西：

第一，待调优的算法或流水线。这就是我们要优化的目标。

第二，一系列参数表(ParamMap)。这些是可选参数，我们也叫它参数网格搜索空间(Parameter Grid Search Space)。简单来说，就是我们要尝试的不同参数组合。

第三，评估模型拟合程度的准则或方法。我们需要一个标准来判断哪个模型更好。

这三个要素缺一不可，它们构成了模型选择的基础框架。

第 4 张슬라이드：模型选择工具工作原理

现在让我详细解释一下模型选择工具是如何工作的。

整个过程分为三个步骤：

第一步，将输入数据划分为训练数据和测试数据。这是机器学习的基本操作。

第二步，对于每个训练数据和测试数据的组合，我们会遍历一组 ParamMap。用每一个 ParamMap 参数来拟合估计器，得到训练后的模型，然后使用评估器来评估模型表现。

第三步，选择性能表现最优的模型所对应的 ParamMap。

这个过程看起来简单，但是计算量可能会很大，特别是当参数组合很多的时候。

第 5 张슬라이드：CrossValidator 详细说明

让我们详细了解一下 CrossValidator 是如何工作的。

CrossValidator 采用的是 k 折交叉验证的方法。它将数据集切分成 k 个折叠，每个折叠都会被分别用于训练和测试。

举个例子，当 k=3 时，CrossValidator 会生成 3 个训练数据和测试数据的组合。每一个组合中，训练数据占 2/3，测试数据占 1/3。

为了评估一个 ParamMap，CrossValidator 会计算这 3 个不同组合在评估器拟合出的模型上的平均评估指标。

在找出最好的 ParamMap 后，CrossValidator 会使用这个 ParamMap 和整个的数据集，来重新拟合评估器。

也就是说，通过交叉验证找到最佳的 ParamMap，利用此 ParamMap 在整个训练集上可以训练出一个泛化能力强、误差小的最佳模型。

这种方法的优点是结果比较可靠，但缺点是计算成本较高。

第 6 张슬라이드：TrainValidationSplit 说明

由于交叉验证的代价比较高昂，Spark 也为超参数调优提供了另一个选择——TrainValidationSplit。

TrainValidationSplit 创建单一的训练数据和测试数据对。它使用 trainRatio 参数将数据集切分成两部分。

例如，当设置 trainRatio=0.75 时，TrainValidationSplit 会将数据切分出 75%作为训练集，25%作为测试集，来生成训练数据和测试数据的组合。

最终它会使用最好的 ParamMap 和完整的数据集来拟合估计器。

相对于 CrossValidator 对每一个参数进行 k 次评估，TrainValidationSplit 只对每个参数组合评估一次，因此它的评估代价没有这么高。

但是，当训练数据集不够大的时候，其结果相对不够可信。

所以在选择这两种方法时，我们需要在计算成本和结果可靠性之间做权衡。

第 7 张슬라이드：实际应用示例介绍

现在我们通过一个实际的例子来演示如何使用 CrossValidator。

虽然使用 CrossValidator 的代价可能会异常高，但是对比启发式的手动调优，这是一种选择参数的行之有效的方法。

接下来我们会通过一个完整的实例来演示如何使用 CrossValidator 从整个网格的参数中选择合适的参数。我们会使用经典的 Iris 数据集来进行多分类任务。

第 8 张슬라이드：步骤 1 - 导入必要的包

好，让我们开始实际操作。第一步是导入必要的包。

大家可以看到，我们需要导入很多包：

首先是特征处理相关的包，比如 HashingTF 和 Tokenizer 用于文本处理，VectorAssembler 用于特征向量组装。

然后是模型选择相关的包，CrossValidator 和 ParamGridBuilder 是我们今天的重点。

接着是分类算法相关的包，我们今天会用到 LogisticRegression。

还有评估相关的包，MulticlassClassificationEvaluator 用于多分类评估。

最后是一些基础的包，比如 SparkSession、数据类型等。

这些包的导入顺序不是固定的，但是建议大家按照功能分类来组织导入语句，这样代码会更清晰。

第 9 张슬라이드：步骤 2 - 数据读取和预处理介绍

第二步是读取 Iris 数据集，并进行预处理。

这里有几个重要点需要注意：

首先，我们要分别获取标签列和特征列，进行索引、重命名，并设置机器学习工作流。

其次，我们通过交叉验证把原始数据集分割为训练集与测试集。这里要强调的是，只有训练集才可以用在模型的训练过程中，测试集则作为模型完成之后用来评估模型优劣的依据。

另外，训练集中的样本数量必须足够大，一般至少要大于总样本数的 50%，且两个子集必须从完整集合中均匀取样。

现在让我们看具体的代码实现。

第 10 张슬라이드：数据类型转换和特征向量创建

大家看这段代码，我们首先要做数据类型转换。

Iris 数据集的特征都是数值型的，但是读取进来可能是字符串类型，所以我们需要将所有特征列转换为 Double 类型。这里我们使用 cast 方法来进行类型转换。

然后，我们创建特征向量。在 Spark MLlib 中，所有的机器学习算法都期望输入是向量形式的特征。我们使用 VectorAssembler 来将多个特征列组合成一个特征向量列。

VectorAssembler 的 inputCols 参数指定输入的列名，outputCol 参数指定输出的列名。

最后，我们通过 transform 方法应用这个转换，并选择我们需要的 features 和 label 列。

这样，我们就得到了标准的机器学习数据格式。

第 11 张슬라이드：数据分割和索引操作

接下来我们进行数据分割和索引操作。

首先是数据分割。我们使用 randomSplit 方法将数据按 7:3 的比例分为训练集和测试集。这个比例是比较常用的，但根据具体情况也可以调整。

然后是标签索引。在 Spark MLlib 中，分类算法通常要求标签是数值型的索引，而不是字符串。StringIndexer 可以将字符串标签转换为数值索引。

接着是特征向量索引。VectorIndexer 会自动检测哪些特征是分类特征，哪些是连续特征，并进行相应的处理。

注意这里我们都使用了 fit 方法，这是因为索引器需要先扫描整个数据集来确定索引映射关系。

这些预处理步骤虽然看起来繁琐，但是对于后续的模型训练非常重要。

第 12 张슬라이드：创建 Logistic Regression 模型和 Pipeline

现在我们来创建 Logistic Regression 模型和 Pipeline。

首先创建 LogisticRegression 模型。这里我们指定了 labelCol 和 featuresCol 参数，分别对应索引后的标签列和特征列。maxIter 参数设置最大迭代次数为 50。

然后创建标签转换器。模型的预测结果是数值索引，但是我们希望看到原始的字符串标签，所以用 IndexToString 将预测结果转换回原始标签。

最后创建 Pipeline。Pipeline 是 Spark MLlib 中的一个重要概念，它可以将多个处理步骤串联起来，形成一个完整的机器学习工作流。

我们的 Pipeline 包含四个阶段：标签索引器、特征索引器、逻辑回归模型和标签转换器。

Pipeline 的好处是我们可以一次性处理整个工作流，而不需要手动管理每个步骤的输入输出。

第 13 张슬라이드：步骤 3 - 构造参数网格

第三步是使用 ParamGridBuilder 构造参数网格。

这里我们要调优两个重要参数：

第一个是 regParam，这是公式(9-12)中的 γ，用于定义正则化项的权重。正则化可以防止过拟合，这个参数很重要。

第二个是 elasticNetParam，这是 α，称为 Elastic Net 参数，取值介于 0 和 1 之间。当 α=0 时，使用 L2 正则化；当 α=1 时，使用 L1 正则化；介于两者之间时，使用 Elastic Net 正则化。

在我们的例子中，elasticNetParam 设置 2 个值（0.2 和 0.8），regParam 设置 3 个值（0.01、0.1 和 0.5），所以最终有 3×2=6 个不同的模型将被训练和评估。

ParamGridBuilder 使用链式调用的方式，非常简洁明了。

第 14 张슬라이드：步骤 4 - 构建交叉验证器

第四步是构建针对整个机器学习工作流的交叉验证类。

我们需要定义三个重要组件：

第一个是 estimator，也就是我们要调优的估计器，这里是我们刚才创建的 Pipeline。

第二个是 estimatorParamMaps，也就是参数网格，这里是我们刚才构建的 paramGrid。

第三个是 evaluator，也就是评估器。对于回归问题，我们可以选择 RegressionEvaluator；对于二元分类，可以选择 BinaryClassificationEvaluator；对于多分类问题，我们选择 MulticlassClassificationEvaluator。

numFolds 参数设置交叉验证的折数，这里设置为 3。

然后我们调用 fit 方法在训练数据上进行交叉验证。这个过程可能会比较耗时，因为需要训练多个模型。

评估器里默认的评估准则可以通过 setMetricName 方法重写，比如可以使用 accuracy、precision、recall 等。

第 15 张슬라이드：步骤 5 - 进行预测

第五步是调用 transform 方法对测试数据进行预测，并输出结果及精度。

交叉验证完成后，我们得到了最佳的模型 cvModel。现在我们可以用这个模型对测试数据进行预测。

transform 方法会对测试数据进行预测，返回包含预测结果的 DataFrame。

我们可以使用 show 方法显示前 20 个预测结果，这样可以直观地看到模型的预测效果。

预测结果包含多个列：predictedLabel 是预测的标签，label 是真实标签，features 是特征向量，probability 是预测概率。

通过比较 predictedLabel 和 label，我们可以看出模型的预测准确性。

第 16 张슬라이드：输出预测结果和计算准确度

现在我们来看具体的预测结果输出和准确度计算。

首先，我们可以遍历预测结果，输出每个样本的详细信息。从输出可以看到，对于一个 Iris-virginica 样本，模型给出了三个类别的概率，最高概率对应 Iris-virginica，所以预测正确。

然后，我们创建 MulticlassClassificationEvaluator 来计算模型的准确度。这个评估器会比较预测标签和真实标签，计算出整体的准确率。

通过 evaluate 方法，我们可以得到模型在测试集上的准确度。这个数值可以帮助我们判断模型的性能是否满足要求。

如果准确度不够高，我们可能需要调整参数网格，增加更多的参数组合，或者尝试不同的算法。

第 17 张슬라이드：步骤 6 - 获取最优模型

第六步是获取最优的逻辑回归模型，并查看其具体的参数。

交叉验证完成后，我们可以通过 bestModel 属性获取最佳模型。这个模型是在整个训练集上使用最佳参数训练得到的。

由于我们使用的是 Pipeline，最佳模型实际上是一个 PipelineModel。我们需要通过 stages 属性获取其中的 LogisticRegression 模型，它位于第 2 个位置（索引为 2）。

然后我们可以查看模型的各种参数：

coefficientMatrix 是模型的系数矩阵，每一行对应一个类别的系数。

interceptVector 是截距向量。

numClasses 显示类别数量，这里是 3。

numFeatures 显示特征数量，这里是 4。

这些信息可以帮助我们更好地理解模型的内部结构。

第 18 张슬라이드：模型参数详细信息

最后，我们来看看模型参数的详细信息。

explainParam 方法可以显示参数的详细说明。

对于 regParam，显示它是正则化参数，必须大于等于 0，默认值是 0.0，当前值是 0.01。这说明交叉验证选择了较小的正则化强度。

对于 elasticNetParam，显示它是 Elastic Net 混合参数，取值范围是[0,1]。当 alpha=0 时，使用 L2 惩罚；当 alpha=1 时，使用 L1 惩罚。默认值是 0.0，当前值是 0.2，这意味着模型主要使用 L2 正则化，但也包含少量 L1 正则化。

这些参数的选择反映了数据的特性和模型的复杂度需求。

第 19 张슬라이드：本章小结

好，让我们来总结一下今天学习的内容。

Spark 在机器学习方面的发展非常快，目前已经支持了主流的统计和机器学习算法。MLlib 以计算效率高而著称，是一个非常优秀的基于分布式架构的开源机器学习库。

MLlib 能有效简化机器学习的工程实践工作，并可被方便地扩展到大规模的数据集上进行模型训练和预测。

今天我们重点学习了模型选择的方法，包括 CrossValidator 和 TrainValidationSplit 两种工具。我们通过一个完整的例子，演示了如何使用交叉验证来选择逻辑回归模型的最佳参数。

关键点包括：参数网格的构建、交叉验证的配置、模型的训练和评估，以及最终模型的分析。

希望大家通过今天的学习，能够掌握 Spark MLlib 中模型选择的基本方法，为将来的机器学习项目打下坚实的基础。

有什么问题，我们可以一起讨论！