Spark MLlib 中文讲课脚本

슬라이드01：课程开始

大家好！今天我们要学习的是Spark MLlib，也就是Spark的机器学习库(Machine Learning Library)。

今天这一章的内容非常重要，因为机器学习(Machine Learning)现在在大数据领域

应用得非常广泛(guǎngfàn)，而Spark MLlib就是专门为大数据环境(huánjìng)下的机器学习而设计的。

专门为...而设计的 : ...를 위해 특별히 설계된

슬라이드02：课程大纲介绍

好，我们先来看看今天要学习的内容大纲。

今天的课程分为11个部分：

- 首先，我们会讲基于大数据的机器学习，让大家了解为什么要用Spark来做机器学习
- 然后介绍MLlib库的整体概况
- 接下来会详细讲解基本的数据类型，这是使用MLlib的基础
- 还有基本的统计分析工具
- 机器学习流水线(Pipeline)的概念
- 特征提取(Feature Extraction)、转换和选择
- 各种机器学习算法，包括分类、聚类、协同过滤等
- 最后是模型选择

大家可以看到，这门课程内容很丰富。

슬라이드3：什么是机器学习

好，我们正式开始。

首先来了解一下什么是机器学习。

机器学习可以看做是一门人工智能(Artificial Intelligence)的科学，它的主要研究对象是人工智能。

简单来说，机器学习就是利用数据或者以往(yǐwǎng 기존의)的经验，来优化计算机程序的性能标准(biāozhǔn)。

大家注意这个图，机器学习有三个关键词：

算法(Algorithm)、经验(Experience)、性能(Performance)。

- T代表算法，就是我们要用什么方法
- E代表经验，也就是我们的训练数据
- P代表性能，就是我们要达到什么目标

这三者形成一个循环，通过不断地训练和评估来提升模型的性能。

슬라이드4：传统机器学习的局限性(jú xiàn xìng limitations)

那么为什么我们要在大数据环境下做机器学习呢？

传统的机器学习算法有很多局限性：

第一，由于技术和单机存储的限制，传统机器学习只能在少量数据上使用，往往需要依赖数据抽样(Data Sampling)。但是大家想想，抽样可能会丢失很多重要信息。

第二，大数据技术的出现让我们可以在全量数据上进行机器学习，这样效果会更好。

第三，机器学习算法通常涉及大量的迭代计算(**dié dài** Iterative Computing)。

第四，基于磁盘(cí pán)的MapReduce不适合进行大量迭代计算，因为每次迭代都要读写磁盘，效率很低。

最后，基于内存的Spark比较适合(**shì hé)**进行大量迭代计算，因为数据可以缓存**(huǎn cún)**在内存中。

슬라이드 05：Spark在机器学习中的优势

相比而言，Spark立足于内存计算(In-Memory Computing)，特别适用于迭代计算，能很好地与机器学习算法相匹配。

这也是近年来Spark平台流行的重要原因，业界的很多业务都从Hadoop平台转向了Spark平台。

Spark提供了一个基于海量数据的机器学习库，它提供了常用机器学习算法的分布式实现(Distributed Implementation)。

对普通开发者而言，只需要有Spark编程基础，

并且了解机器学习算法的基本原理和方法中相关参数的含义，

就可以轻松地通过调用**(diào yòng,호출)**相应的API来实现基于海量数据的机器学习过程。

这就是Spark MLlib的强大之处！

立足于 : ~에 기반을 두고 있다

能很好地与...相匹配 : ~와 매우 잘 부합하다

슬라이드6：MLlib库概述

现在我们来详细了解一下MLlib。

MLlib是Spark的机器学习库，目标是简化机器学习的工程实践工作。

MLlib由一些通用的学习算法和工具组成，包括：

- **算法工具**：常用的学习算法，如分类(Classification)、回归(Regression)、聚类(Clustering)和协同(xié tóng)过滤(Collaborative Filtering)
- **特征化工具**：特征提取、转化、降维和选择工具
- **流水线(Pipeline)**：用于构建、评估和调整机器学习工作流的工具
- **持久性(Persistence)**：保存和加载算法、模型和管道
- **实用工具**：线性代数(Linear Algebra)、统计、数据处理等工具

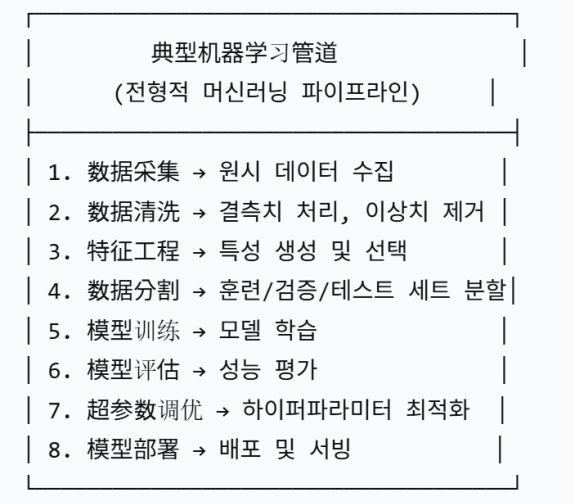
这些工具让我们可以很方便地构建完整的机器学习应用。

模型 (Model)

- 데이터에서 학습된 패턴의 **수학적 표현**
- 예측, 분류, 군집화 등의 작업 수행

管道 (Pipeline)

- 데이터 처리 단계를 **자동화된 흐름**으로 연결
- 전처리 → 특성 추출 → 모델 학습 → 평가의 일련 과정



슬라이드 7：MLlib 支持的算法

我们来看看MLlib目前支持哪些主要的机器学习算法。

这个表格很全面地列出了各种算法：

基本统计包括摘要统计(zhāi yào tǒng jì Summary Statistics)、相关性分析(Correlations)等。

分类和回归包括支持向量机(SVM)、逻辑回归(Logistic Regression)、

线性回归(Linear Regression)、朴素贝叶斯(pǔ sù bèi yè s Naive Bayes)、

决策树(Decision Trees)、

随机森林(Random Forest)等。

协同过滤主要是交替最小二乘法(ALS)。

聚类算法有K-Means、高斯混合模型(Gaussian Mixture Model)、

潜在狄利克雷分配(LDA)等。

降维算法包括奇异值分解(SVD)、主成分分析(PCA)。

特征提取和转换有TF-IDF、Word2Vec、标准化器(StandardScaler)等。

可以说覆盖(fù gài)了机器学习的各个方面。

슬라이드8：两个包的区别

这里要特别注意，Spark机器学习库从1.2版本以后被分为两个包：

spark.mllib包含基于RDD的原始算法API。

这个包历史比较长，在1.0以前的版本就已经包含了，提供的算法实现都是基于原始的RDD(Resilient Distributed Dataset)。

spark.ml则提供了基于DataFrames的高层次API，可以用来构建机器学习工作流(Pipeline)。ML Pipeline弥补(mí bǔ, 보완하다. 보충하다)了原始MLlib库的不足，向用户提供了一个基于DataFrame的机器学习工作流式API套件。

现在建议大家优先使用spark.ml，因为它更加现代化，而且与Spark SQL的集成更好。

슬라이드9：基本数据类型概览

接下来我们学习MLlib中的基本数据类型。

这部分内容包括四个方面：

1. 本地向量(Local Vectors)
2. 标注点(Labeled Points)
3. 本地矩阵(**(jǔ zhèn** Local Matrices)
4. 数据源(Data Sources)

这些是使用MLlib进行机器学习的基础，大家一定要掌握好。

슬라이드10：本地向量

首先来看本地向量。 **搜索 (sōusuǒ)와 혼동**

MLlib支持两种类型的本地向量：

稠密向量(DenseVector)：使用双精度浮点型数组来表示每一维的元素。比如向量(1.0, 0.0, 3.0)用稠密向量表示就是[1.0,0.0,3.0]。

稀疏向量(SparseVector)：基于一个整型索引 **(suǒ yǐn)** 数组和一个双精度浮点型的值数组。同样的向量(1.0, 0.0, 3.0)用稀疏向量表示就是(3, [0,2], [1.0, 3.0])。

稀疏向量的格式是：(向量长度, [非零元素的索引], [非零元素的值])。

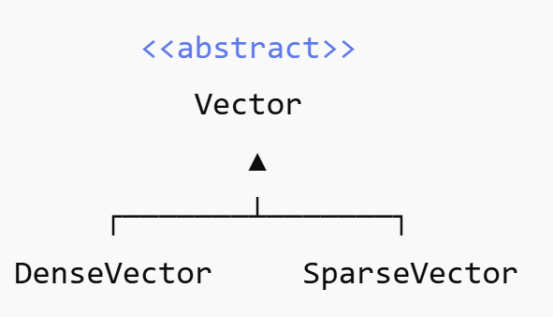
什么时候用稀疏向量呢？

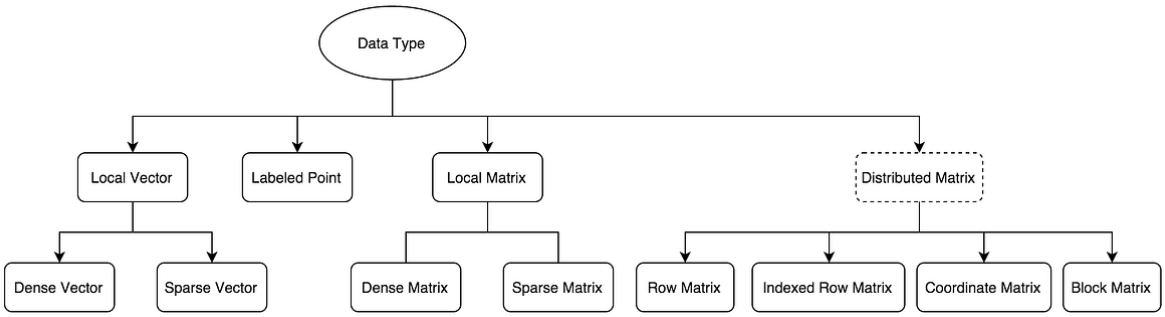
当向量中大部分元素都是0的时候，用稀疏向量可以节省很多存储空间。

슬라이드11：向量的基类

所有本地向量都以org.apache.spark.ml.linalg.Vector为基类，DenseVector和SparseVector分别是它的两个继承类。

所以我们推荐使用Vectors工具类下定义的工厂方法来创建本地向量，这样比较规范。





팩토리 메서드란? (정의)

✓ 정의

- **객체 생성을 전담하는 메서드**
- new 를 직접 쓰지 않고
  메서드를 통해 객체를 생성하도록 하는 설계 방식

👉 **"어떤 객체를 만들지 결정하는 책임을 생성자에서 분리"**

왜 팩토리 메서드를 쓰는가? (핵심 이유)

1) 구현 클래스에 의존하지 않게 하기

- Client 코드가 DenseVector, SparseVector 를 **직접 알 필요 없음**

2) 확장에 강함 (Open–Closed Principle)

- 새로운 Vector 타입 추가 시
  Client 코드 수정 없음

3) 생성 로직을 한 곳에 집중

- 검증, 예외 처리, 최적화 가능

팩토리 메서드는
"선택을 누가 하느냐"가 아니라
"객체 생성을 누가 책임지느냐"로 판단한다.

내가 new 키워드를 사용하지 않고 객체를 생성하는 것에 의미가 있음

슬라이드12：向量创建代码示例

我们来看具体的代码示例：

```
# PySpark 向量创建示例代码 (벡터 생성 예제 – Factory Method)

# 导入必要的库
from pyspark.ml.linalg import Vector, Vectors    # Vectors : Factory Method Supply class
from pyspark.sql import SparkSession


def main():
    # 创建 SparkSession (PySpark 代码中必需)
    spark = SparkSession.builder ₩
        .appName("VectorExample") ₩
        .getOrCreate()

    try:
        # 1. 创建稠密向量 (밀집 벡터 dense vector)
        # dense() 方法接收一个列表作为参数，创建所有元素都存储的向量
        dv = Vectors.dense([2.0, 0.0, 8.0])  # 工厂方法 (Factory Method)

        # 2. 创建稀疏向量 (희소 벡터 sparse vector) 方法1
        # sparse() 方法的三个参数：
        #   - 第1个参数: 向量大小 (维度)
        #   - 第2个参数: 非零元素的索引数组
        #   - 第3个参数: 非零元素的值数组
        sv1 = Vectors.sparse(3, [0, 2], [2.0, 8.0])

        # 3. 创建稀疏向量 (희소 벡터 sparse vector) 方法2
        # sparse() 方法的两个参数：
        #   - 第1个参数: 向量大小 (维度)
        #   - 第2个参数: (索引, 值) 元组的序列
        sv2 = Vectors.sparse(3, [(0, 2.0), (2, 8.0)])

        # 打印结果
        print("=== PySpark 向量创建示例 ===")
```

```
            print(f"稠密向量 dv: {dv}")
            print(f"稀疏向量 sv1: {sv1}")
            print(f"稀疏向量 sv2: {sv2}")

            # 检查向量信息
            print("₩n=== 向量详细信息 ===")
            print(f"dv 类型: {type(dv)}")
            print(f"dv 大小: {dv.size}")
            print(f"dv 值: {dv.toArray()}")

            print(f"₩nsv1 类型: {type(sv1)}")
            print(f"sv1 大小: {sv1.size}")
            print(f"sv1 值: {sv1.toArray()}")
            print(f"sv1 非零元素数量: {sv1.numNonzeros()}")

            # 比较两个稀疏向量
            print(f"₩nsv1 和 sv2 是否相等? {sv1 == sv2}")

        finally:
            # 清理资源 (重要: 避免内存泄漏)
            spark.stop()
            print("₩nSparkSession 已停止。")


if __name__ == "__main__":
    main()
```

大家注意这里的参数：

- sparse方法的第一个参数是向量的长度
- 第二个参数是非零元素的索引
- 第三个参数是对应的值

运行结果显示，两种稀疏向量的创建方法得到了相同的结果。

슬라이드13：标注点

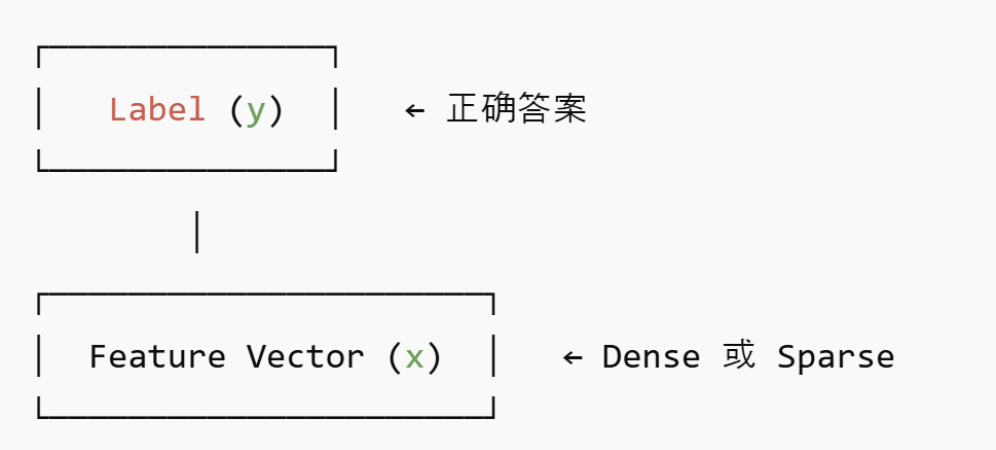
接下来我们学习标注点(Labeled Point)。

标注点是一种带有标签(Label/Response)的本地向量，通常用在监督学习算法中。它可以是稠密向量，也可以是稀疏向量。

由于标签是用双精度浮点型来存储的，因此标注点类型在回归和分类问题上都可以使用。

- 对于二分类问题，正样本的标签为1，负样本的标签为0
- 对于多类别的分类问题，标签应该是一个以0开始的索引序列：0，1，2...

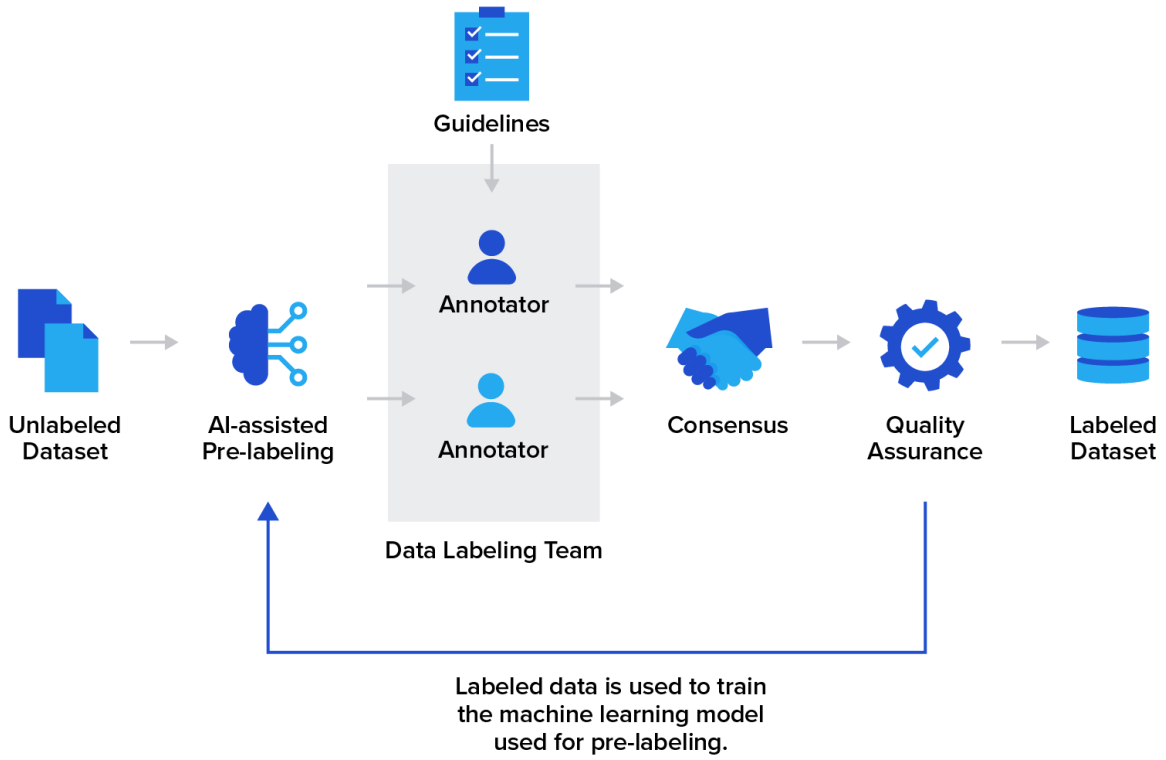
这样设计是为了保持一致性，方便算法处理。



Labeled Point = Label（答案） + Vector（特征）

annotator(标注者)：labeling하는 사람

Data Labeling Pipeline



슬라이드14：标注点代码示例

我们来看标注点的创建方法：

```
from pyspark.sql import SparkSession
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.linalg import Vectors


def main():
    # 1. SparkSession 생성 (PyCharm 필수)
    spark = SparkSession.builder ₩
        .appName("LabeledPointExample") ₩
        .master("local[*]") ₩
        .getOrCreate()

    # 2. 稠密向量(Dense Vector) 标注点
    # 标签为 1.0（可视为正样本）
    pos = LabeledPoint(
        1.0,
        Vectors.dense([2.0, 0.0, 8.0])
    )

    # 3. 稀疏向量(Sparse Vector) 标注点
    # 标签为 0.0（可视为负样本）
    neg = LabeledPoint(
        0.0,
        Vectors.sparse(3, [0, 2], [2.0, 8.0])
    )

    # 4. 输出结果
    print("Positive Example:", pos)
    print("Negative Example:", neg)

    # 5. Spark 종료
    spark.stop()
```

```
if __name__ == "__main__":
    main()
```

注意这里我们使用的是pyspark.mllib.regression包，

这是因为LabeledPoint属于原始的mllib API。

输出结果显示，标注点包含了标签和特征向量两部分信息。

슬라이드15：LIBSVM格式

在实际的机器学习问题中，稀疏向量数据是非常常见的。

MLlib提供了读取LIBSVM格式数据的支持，这个格式被广泛用于LIBSVM、LIBLINEAR等机器学习库。

LIBSVM: Library for Support Vector Machines

LIBLINEAR: Library for Large Linear Classification

LIBSVM格式的特点是：

label index1:value1 index2:value2 index3:value3 ...

其中：

- label是该样本yàng běn点的标签值
- index:value代表非零元素的索引和元素值
- 特别注意：index是以1开始并递增的

这种格式对于稀疏数据非常高效。

基本形式

<label> <index1>:<value1> <index2>:<value2> ...

各部分含义

- label
  - 分类任务：类别标签（如 1、-1、0）
  - 回归任务：连续数值
- index:value
  - index：特征编号（从 1 开始）
  - value：该特征的取值
  - 只记录非 0 的特征

👉 一行 = 一个样本（样本 = Labeled Point）

슬라이드16：读取LIBSVM文件

我们来看如何读取LIBSVM格式的文件：

这里使用了spark.read.format("libsvm")来指定文件格式，Spark会自动解析LIBSVM格式的数据。

```
from pyspark.sql import SparkSession


def main():
    # 1 LIBSVM 格式数据文件路径
    # file:/// 表示本地文件系统
    path = "file:///usr/local/spark/data/mllib/sample_libsvm_data.txt"

    # 2 创建 SparkSession（PyCharm 中必须显式创建）
    spark = SparkSession.builder ₩
        .appName("LIBSVMExample") ₩
        .master("local[*]") ₩
        .getOrCreate()

    try:
        # 3 读取 LIBSVM 格式的数据
        # Spark 会自动解析为 (label, features) 的 DataFrame
        examples = spark.read.format("libsvm").load(path)

        # 4 查看 DataFrame 的结构
        print("=== 数据结构 (Schema) ===")
        examples.printSchema()

        # 5 以表格形式查看前几条数据（DataFrame 视角）
        print("₩n=== 示例数据（DataFrame 视角，前 5 行）===")
        examples.show(5, truncate=False)

        # 6 使用 head() 取出第一个样本（Row 对象）
        print("₩n=== 使用 head() 取得的第一个样本 ===")
        first_row = examples.head()
        print(first_row)
```

```
        # 7 分别访问 label 和 features（Labeled Point 的两个组成部分）
        print("₩n=== 解析 head() 返回的 Row ===")
        print("Label:", first_row.label)
        print("Features:", first_row.features)
        print("Features 类型:", type(first_row.features))

        # 8 统计样本数量
        print("₩n=== 样本数量 ===")
        print(examples.count())

    finally:
        # 9 关闭 SparkSession（非常重要）
        spark.stop()
        print("₩nSparkSession 已停止。")


if __name__ == "__main__":
    main()
```

슬라이드17：查看加载的数据

我们来查看加载进来的数据：

>>> examples.head()

输出结果显示，数据被解析成了包含label和features两列的DataFrame。

features列是一个SparseVector，包含了692维的稀疏向量，只存储非零元素。

大家可以看到，这个数据集可能是手写数字识别的数据，因为有很多像素xiàng sù值。

픽셀(pixel)

슬라이드18：本地矩阵

现在我们学习本地矩阵。

本地矩阵具有整型的行、列索引值和双精度浮点型(double类型)的元素值，它存储在单机上。

MLlib支持两种本地矩阵：

- **稠密矩阵(DenseMatrix)**：将所有元素的值存储在一个列优先(Column-major)的双精度型数组中
- **稀疏矩阵(SparseMatrix)**：将非零元素以列优先的CSC(Compressed Sparse Column)模式进行存储

本地矩阵的基类是pyspark.ml.linalg.Matrix，和向量类似，我们也有Matrices工具类来创建矩阵实例。

슬라이드19：创建稠密矩阵

我们来看稠密矩阵的创建：

```
# PyCharm에서 실행할 PySpark 코드
from pyspark.sql import SparkSession
from pyspark.ml.linalg import Matrices


def main():
    """메인 실행 함수"""
    # SparkSession 생성 (PyCharm에서 실행할 때 필수)
    spark = SparkSession.builder ₩
        .appName("MatrixExample") ₩
        .master("local[*]") ₩
        .getOrCreate()

    try:
        # 3행 2열의 밀집 행렬 생성
        # 열 우선 방식: [1.0, 3.0, 5.0, 2.0, 4.0, 6.0]
```

```
        # 실제 행렬:
        # [1.0, 2.0]
        # [3.0, 4.0]
        # [5.0, 6.0]

        # 创建一个3行2列的稠密矩阵[ [1.0, 2.0], [3.0, 4.0], [5.0, 6.0] ]
        # 注意，这里的数组参数是列优先的，即按照列的方式从数组中提取元素

        dm = Matrices.dense(3, 2, [1.0, 3.0, 5.0, 2.0, 4.0, 6.0])

        # 행렬 출력
        print("稠密矩阵(Dense Matrix):")
        print(dm)

        # 추가 정보 출력 (선택사항)
        print("₩n행렬 정보:")
        print(f"행 수: {dm.numRows}")
        print(f"열 수: {dm.numCols}")

        # 행렬을 배열로 변환하여 확인
        print("₩n행렬 데이터 (열 우선 방식):")
        print(dm.toArray().tolist())

    finally:
        # SparkSession 종료
        spark.stop()
        print("₩nSparkSession이 종료되었습니다.")


if __name__ == '__main__':
    main()
```

创建一个3行2列的稠密矩阵

>>> dm = Matrices.dense(3, 2, [1.0, 3.0, 5.0, 2.0, 4.0, 6.0])

注意：这里的数组参数是列优先的，也就是说先填第一列，再填第二列。

所以[1.0, 3.0, 5.0, 2.0, 4.0, 6.0]会变成：

[[1., 2.],

[3., 4.],

[5., 6.]]

这个要特别注意，和我们平常的行优先思维不太一样。

슬라이드20：创建稀疏矩阵

稀疏矩阵的创建稍微复杂一些：

>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9.0, 6.0, 8.0])

这里有三个数组参数：

- 第一个[0, 1, 3]是列指针，表示每一列元素的开始索引
- 第二个[0, 2, 1]是行索引，表示对应元素属于哪一行
- 第三个[9.0, 6.0, 8.0]是按列优先排列的所有非零元素

通过这种CSC格式，我们可以高效地存储稀疏矩阵。

```
# PyCharm에서 실행할 PySpark 코드
from pyspark.sql import SparkSession
from pyspark.ml.linalg import Matrices


def create_dense_matrix():
    """밀집 행렬 생성 및 출력"""
    print("₩n" + "=" * 50)
    print("밀집 행렬(Dense Matrix) 예제")
    print("=" * 50)

    # 3행 2열의 밀집 행렬 생성
    # 列优先 방식: [1.0, 3.0, 5.0, 2.0, 4.0, 6.0]
    # 실제 행렬:
    # [1.0, 2.0]
    # [3.0, 4.0]
    # [5.0, 6.0]

    dm = Matrices.dense(3, 2, [1.0, 3.0, 5.0, 2.0, 4.0, 6.0])

    # 행렬 출력
    print("생성된 밀집 행렬:")
    print(dm)

    # 추가 정보 출력
```

```
    print("₩n행렬 정보:")
    print(f"행 수: {dm.numRows}")
    print(f"열 수: {dm.numCols}")

    # 행렬을 배열로 변환하여 확인
    print("₩n행렬 데이터 (열 우선 방식):")
    print(dm.toArray().tolist())

    # 행렬 내용을 더 보기 좋게 출력
    print("₩n행렬 시각화:")
    for i in range(dm.numRows):
        row = []
        for j in range(dm.numCols):
            row.append(dm[i, j])
        print(f"행 {i}: {row}")

    return dm


def create_sparse_matrix():
    """희소 행렬 생성 및 출력"""
    print("₩n" + "=" * 50)
    print("创建一个3行2列的稀疏矩阵[ [9.0,0.0], [0.0,8.0], [0.0,6.0]]")
    print("=" * 50)

    # 희소 행렬 설명:
    # 생성할 행렬:
    # [9.0, 0.0]
    # [0.0, 8.0]
    # [0.0, 6.0]

    # 매개변수 설명:
    # 第1个数组参数表示列指针，即每一列元素的开始索引
    # 第2个数组参数表示行索引，即对应的元素属于哪一行
    # 第3个数组参数即按列优先排列的所有非零元素，通过列指针和行索引即可判断每个元素所在的位置
    # Matrices.sparse(행수, 열수, 열포인터, 행인덱스, 값)
    # - 열포인터: 각 열의 시작 인덱스 [0, 1, 3]
    # - 행인덱스: 각 값의 행 위치 [0, 2, 1] * sorting
```

```
    # - 값: 비제로 값 [9.0, 6.0, 8.0]

    # 열포인터 [0, 1, 3] 해석:
    # - 0번 열: 인덱스 0부터 0까지 (1개 요소)
    # - 1번 열: 인덱스 1부터 2까지 (2개 요소)

    # 행인덱스와 값 매칭:
    # - 인덱스 0: 행 0, 값 9.0 → (0,0) = 9.0
    # - 인덱스 1: 행 2, 값 6.0 → (2,1) = 6.0
    # - 인덱스 2: 행 1, 값 8.0 → (1,1) = 8.0

    sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9.0, 6.0, 8.0])

    # 행렬 출력
    print("创建的稀疏矩阵:")
    print(sm)


    return sm



def main():
    """메인 실행 함수"""
    # SparkSession 생성
    spark = SparkSession.builder ₩
        .appName("MatrixExample") ₩
        .master("local[*]") ₩
        .getOrCreate()

    try:
        print("PySpark 행렬 생성 예제")
        print("=" * 50)

        # 1. 밀집 행렬 생성
        dense_matrix = create_dense_matrix()

        # 2. 희소 행렬 생성
```

```
            sparse_matrix = create_sparse_matrix()

        finally:
            # SparkSession 종료
            spark.stop()
            print("₩n" + "=" * 50)
            print("SparkSession이 종료되었습니다.")
            print("=" * 50)


if __name__ == '__main__':
    main()
```

CSC에서 row_indices는
"각 열(column) 내부에서 행(row) 번호가 오름차순으로 정렬되는 것이 원칙(관례)"입니다.

1️⃣ CSC에서 "순서"가 필요한 이유부터

CSC(Compressed Sparse Column)는 **열(column) 기준 저장 방식**입니다.

- 열 0의 값들
- 열 1의 값들
- 열 2의 값들
  … 을 **차례대로 values 배열에 저장**

👉 따라서 **순서는 두 단계로 나뉩니다**

1. **열 순서**: column 0 → 1 → 2 → …
2. **열 내부 순서**: row index 오름차순

주어진 원래 행렬

A =

[0 0 5]

[2 0 0]

[0 0 6]

- 행(row): 0, 1, 2
- 열(column): 0, 1, 2

1 열(column)별 non-zero 개수부터 확인

🔹 Column 0

[0

2

0]

- non-zero: 1개 → (row=1, value=2)

🔹 Column 1

[0

0

0]

- non-zero: **0개**

🔹 Column 2

[5

0

6]

- non-zero: 2개 → (row=0, value=5), (row=2, value=6)

2 Column pointers 계산

CSC에서:

- col_pointers[0] = 0
- 각 열이 끝날 때마다 **지금까지의 non-zero 누적 개수**를 기록
- 열이 3개 → col_pointers 길이는 **3 + 1 = 4**

누적 계산

- Column 0 끝: 1개 → 1
- Column 1 끝: 0개 추가 → 1
- Column 2 끝: 2개 추가 → 3

✅ 최종 column pointers

col_pointers = [0, 1, 1, 3]

3️⃣ 의미 해석 (아주 중요)

Column values 범위 설명

0 values[0:1] 값 1개

1 values[1:1] 값 없음 (빈 열)

2 values[1:3] 값 2개

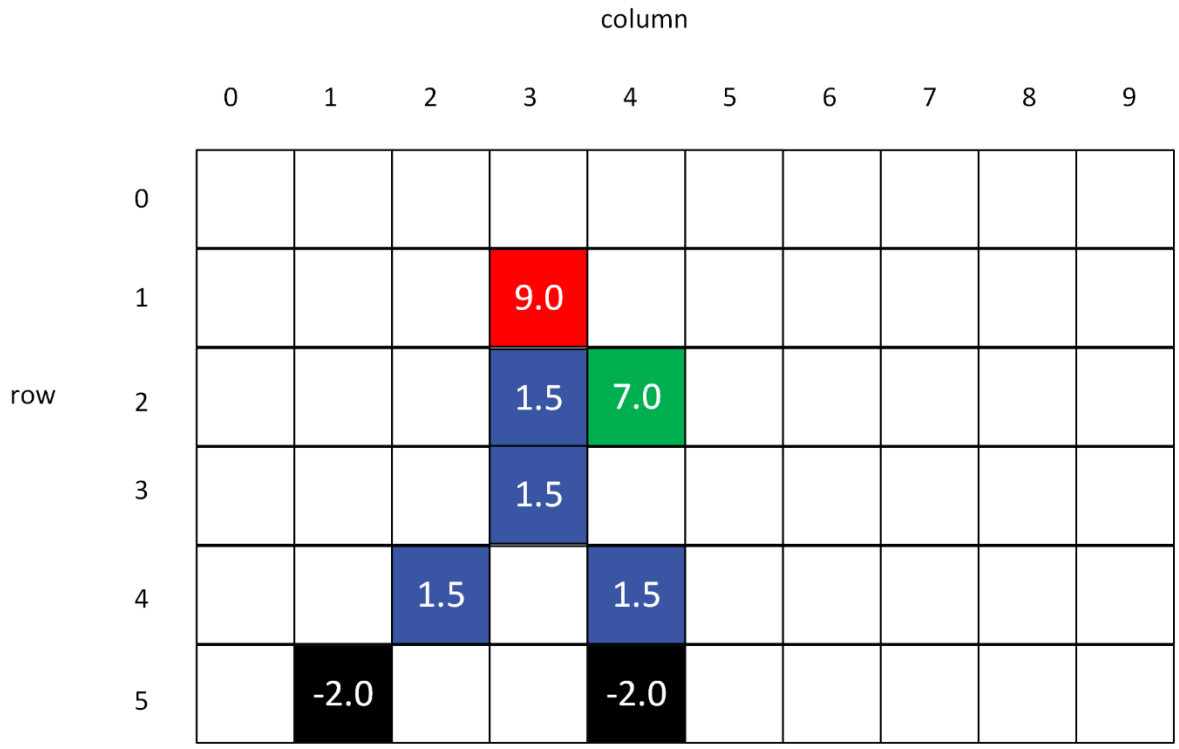
👉 중간에 같은 숫자(1, 1)가 반복되는 것은
그 열에 non-zero 값이 없다는 뜻입니다.

🔑 한 줄 결정판

주어진 행렬의 CSC **column pointers는 [0, 1, 1, 3]** 이다.

C SC matrix 후벼파기

다음 그림에 해당하는 CSC Matrix 를 만드시오.(visio 로 그린 그림)



풀이 결과는 다음과 같음

목표: 컬러(색상별 서로 다른 값)까지 포함한 CSC(Compressed Sparse Column) Matrix 를 만들고 Spark MLlib 에서 테스트하기

이번에 달라진 점

- **이전: 파란 칸만 존재(단색) → 모든 non-zero 가 동일 값(예: 1.0)**
- **이번: Red/Green/Blue/Black 가 섞임 → 색상별로 값을 임의로 다르게 부여하고 CSC 로 압축**

---

1) 이미지 패턴을 "좌표(row, col)"로 확정

1-1. 그리드(행렬) 크기

- **6 rows × 10 cols**

- **index 규칙(0-based)**
    - **row: 위→아래 0..5**
    - **col: 왼→오른쪽 0..9**

1-2. 색상별 non-zero 위치(그림 그대로)

- **Red**
    - **(1, 3)**
- **Green**
    - **(2, 4)**
- **Blue**
    - **(2, 3)**
    - **(3, 3)**
    - **(4, 2)**
    - **(4, 4)**
- **Black**
    - **(5, 1)**
    - **(5, 4)**

---

2) 색상 → 값(value) 매핑을 "임의로" 정의

2-1. 예시 매핑(서로 다르게, 보기 쉽게)

- **Red = 9.0**
- **Green = 7.0**
- **Blue = 1.5**
- **Black = -2.0**
- **White(빈 칸) = 0.0**

---

3) Dense matrix(전체 행렬)를 "전부" 작성 (6×10)

3-1. Dense matrix 전체(각 행 row0~row5, 각 열 col0~col9)

- **row0: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]**
- **row1: [0, 0, 0, 9.0, 0, 0, 0, 0, 0, 0]**
- **row2: [0, 0, 0, 1.5, 7.0, 0, 0, 0, 0, 0]**
- **row3: [0, 0, 0, 1.5, 0, 0, 0, 0, 0, 0]**
- **row4: [0, 0, 1.5, 0, 1.5, 0, 0, 0, 0, 0]**
- **row5: [0, -2.0, 0, 0, -2.0, 0, 0, 0, 0, 0]**

---

4) Dense → CSC 로 변환: 열(column)별로 "전부" 전개

4-1. CSC 3 요소 복습

CSC 가 들고 있는 배열 3 개

- **colPtrs : 길이 = numCols + 1 (= 11)**
- **rowIndices : 길이 = nnz (= non-zero 개수)**
- **values : 길이 = nnz**

4-2. 열 0~9 를 "하나도 빠짐없이" 정리

Col 0

- **non-zero 없음**
- **rows: []**
- **values: []**

Col 1

- **non-zero: (5,1) = -2.0**

- **rows: [5]**
- **values: [-2.0]**

Col 2

- **non-zero: (4,2) = 1.5**
- **rows: [4]**
- **values: [1.5]**

Col 3

- **non-zero:**
    - **(1,3) = 9.0 (Red)**
    - **(2,3) = 1.5 (Blue)**
    - **(3,3) = 1.5 (Blue)**
- **rows: [1, 2, 3]**
- **values: [9.0, 1.5, 1.5]**

Col 4

- **non-zero:**
    - **(2,4) = 7.0 (Green)**
    - **(4,4) = 1.5 (Blue)**
    - **(5,4) = -2.0 (Black)**
- **rows: [2, 4, 5]**
- **values: [7.0, 1.5, -2.0]**

Col 5

- **non-zero 없음**
- **rows: []**
- **values: []**

Col 6

- **non-zero 없음**
- **rows: []**
- **values: []**

Col 7

- **non-zero 없음**
- **rows: []**
- **values: []**

Col 8

- **non-zero 없음**
- **rows: []**
- **values: []**

Col 9

- **non-zero 없음**
- **rows: []**
- **values: []**

---

5) 최종 CSC 배열 3 개(정답 형태)

5-1. rowIndices (열 순서대로 이어붙인 결과)

- **Col0: []**
- **Col1: [5]**
- **Col2: [4]**
- **Col3: [1, 2, 3]**
- **Col4: [2, 4, 5]**

- **Col5~9: []**
- **최종 rowIndices**
    - **[5, 4, 1, 2, 3, 2, 4, 5]**

5-2. values (rowIndices 와 1:1 대응)

- **Col1: [-2.0]**
- **Col2: [1.5]**
- **Col3: [9.0, 1.5, 1.5]**
- **Col4: [7.0, 1.5, -2.0]**
- **최종 values**
    - **[-2.0, 1.5, 9.0, 1.5, 1.5, 7.0, 1.5, -2.0]**

5-3. colPtrs (각 열 시작 offset 의 누적)

- **열별 nnz 개수: [0, 1, 1, 3, 3, 0, 0, 0, 0, 0]**
- **누적(시작 0 포함, 마지막 nnz 총합 포함)**
- **최종 colPtrs (길이 11)**
    - **[0, 0, 1, 2, 5, 8, 8, 8, 8, 8, 8]**

---

6) Spark MLlib 에서 CSC(SparseMatrix) 생성 + 테스트(검증) 절차

6-1. 테스트 전략(학생들에게 단계적으로 시킴)

- **(A) 우리가 만든 SparseMatrix 를 생성한다**
- **(B) .toArray()로 dense 로 복원해서 눈으로 확인한다**
- **(C) "기대 dense 행렬(expected)"과 완전일치 비교를 한다**
- **(D) 특정 열(j)을 골라 colPtrs[j]~colPtrs[j+1]로 slicing 해서 열 단위 복원도 확인한다**

---

7) PySpark 코드: 생성 + 검증(완전 재현 가능)

```
from pyspark.sql import SparkSession
from pyspark.ml.linalg import SparseMatrix

if __name__ == "__main__":
    spark = SparkSession.builder.appName("CSC_Color_Test").getOrCreate()

    # 1) 행렬 크기
    numRows = 6
    numCols = 10

    # 2) CSC 구성요소 (이번 컬러 예시의 정답)
    colPtrs = [0, 0, 1, 2, 5, 8, 8, 8, 8, 8, 8]
    rowIndices = [5, 4, 1, 2, 3, 2, 4, 5]
    values = [-2.0, 1.5, 9.0, 1.5, 1.5, 7.0, 1.5, -2.0]

    # 3) SparseMatrix(CSC) 생성
    sm = SparseMatrix(numRows, numCols, colPtrs, rowIndices, values)

    # 4) (검증 B) Dense로 복원해 출력
    dense = sm.toArray()  # numpy array (로컬)
    print("=== Dense (reconstructed) ===")
    print(dense)

    # 5) (검증 C) 기대 Dense 행렬(정답)과 완전일치 비교
    expected = [
        [0,   0,   0,   0,   0,   0, 0, 0, 0, 0],
        [0,   0,   0,   9.0, 0,   0, 0, 0, 0, 0],
        [0,   0,   0,   1.5, 7.0, 0, 0, 0, 0, 0],
        [0,   0,   0,   1.5, 0,   0, 0, 0, 0, 0],
        [0,   0,   1.5, 0,   1.5, 0, 0, 0, 0, 0],
        [0,  -2.0, 0,   0,  -2.0, 0, 0, 0, 0, 0],
    ]

    ok = (dense == expected).all()
    print("Matches expected dense matrix:", ok)

    # 6) (검증 D) 열 단위로 slicing하여 CSC 해석 확인
    # 예: col3, col4를 직접 복원해 보기
    def show_col(j):
        s = colPtrs[j]
        e = colPtrs[j + 1]
        print(f"\n--- Column {j} ---")
        print("start,end =", s, e)
        print("rowIndices =", rowIndices[s:e])
        print("values     =", values[s:e])

    show_col(3)
    show_col(4)
```

```
spark.stop()
```

8) 학생들이 자주 틀리는 포인트(오류 체크리스트)

8-1. colPtrs 길이

- **반드시 numCols + 1**
- **마지막 값은 반드시 nnz (이번 예시에서는 8)**

8-2. rowIndices 의 값 범위

- **반드시 0 <= rowIndex < numRows**
- **이번 예시 rowIndices: [5,4,1,2,3,2,4,5] → 모두 0..5 범위 만족**

8-3. 열 내부 정렬

- **한 열 안에서는 rowIndices 를 보통 오름차순으로 둠(권장)**
- **이번 예시에서**
    - **col3: rows [1,2,3] 정렬 OK**
    - **col4: rows [2,4,5] 정렬 OK**

슬라이드21：数据源介绍

最后我们来学习数据源(Data Sources)。

数据源是指正在使用的数据的来源地，比如Parquet、CSV、JSON和JDBC等。

在MLlib中，我们主要介绍两种特定的数据源：

1. 图像数据源(Image Data Source)
2. LIBSVM数据源

这些数据源让我们可以方便地处理不同类型的数据。

슬라이드22：图像数据源

图像数据源用于从一个目录中加载图像文件，它可以通过Java库中的ImageIO将压缩图像（jpeg、png等）加载为原始图像表示。

加载的DataFrame有一个StructType列"image"，包含以下属性：

- **origin**: 图像的文件路径
- **height**: 图像的高度
- **width**: 图像的宽度
- **nChannels**: 图像的通道数
- **mode**: OpenCV兼容的图像模式
- **data**: 按OpenCV兼容顺序排列的图像字节

这种格式让我们可以在Spark中方便地处理图像数据，比如做图像分类或者计算机视觉任务。

슬라이드23：加载图像数据

我们来看如何加载图像数据：

```
from pyspark.sql import SparkSession

def main():
    # 1. # 创建一个 SparkSession（如果尚未创建） SparkSession 생성 (PyCharm에서 안정적으로 실행)
    spark = (
        SparkSession.builder
        .appName("ImageExample")
        .master("local[*]")
        .getOrCreate()
    )

    # 2. 第1步：读取Spark自带的图像数据源中数据 이미지 데이터 로드
    # 从文件加载图像数据
    # 这里假设您的图像数据位于 "file:///usr/local/spark/data/mllib/images/origin/kittens" 路径下
    image_path = "file:///usr/local/spark/data/mllib/images/origin/kittens"

    df = (
        spark.read.format("image")
        .option("dropInvalid", True)
        .load(image_path)
    )

    # 3. image 컬럼의 origin, width, height 선택
    selected_df = df.select(
        "image.origin",
        "image.width",
        "image.height"
    )

    # 4. 결과 출력 (truncate=False)
    # 第2步：输出image列的origin 、width和height属性值。
    # 选择DataFrame中的图像属性
    selected_df.show(truncate=False)
```

```
    spark.stop()



if __name__ == "__main__":
    main()
```

这里使用format("image")来指定图像格式，option("dropInvalid", True)表示丢弃无效的图像文件。

슬라이드24：显示图像属性

加载完图像后，我们可以查看图像的属性：

```
>>> selected_df = df.select("image.origin", "image.width", "image.height")
>>> selected_df.show(truncate=False)
```

输出结果显示了每个图像文件的路径、宽度和高度信息。这些信息对于后续的图像处理非常有用。

슬라이드25：LIBSVM数据源详解

MLlib库提供LibSVMDatasource类来将libsvm格式的数据加载为dataframe。

Libsvm是由国立台湾大学以C++语言开发的一个开源机器学习库，主要提供支持向量机的算法。

LIBSVM数据格式的特点：

- 每一行代表一个稀疏的特征向量
- 格式为：<label> <index1>:<value1> <index2>:<value2>, ...
- 特征向量是稀疏的，其他索引值对应的属性值默认为0

转换后的DataFrame有两列：

- **标签**：以Double类型存储
- **特征**：以Vector类型存储

슬라이드26：LIBSVM加载参数

我们来看LIBSVM数据加载的代码：

```
from pyspark.sql import SparkSession

def main():
    # 1. SparkSession 생성
    spark = (
        SparkSession.builder
        .appName("LIBSVMExample")
        .master("local[*]")
        .getOrCreate()
    )

    # 2. LIBSVM 데이터 로드
    data_path = "file:///usr/local/spark/data/mllib/sample_libsvm_data.txt"

    df = (
```

```
            spark.read.format("libsvm")
            .option("numFeatures", "780")
            .load(data_path)
        )

        # 3. 결과 출력 (상위 10개)
        df.show(10)

        spark.stop()


if __name__ == "__main__":
        main()
```

LibSVMDataSource有两个重要的option参数：

- **numFeatures**: 特征的数目。如果不指定，方法会自动确定特征数量，但会带来额外的开销
- **vectorType**: 特征向量的类型，可以是"sparse"或"dense"，默认值为"sparse"

建议大家在知道特征数量的情况下明确指定numFeatures，这样效率会更高。

슬라이드27：查看加载结果

最后我们来看加载的结果：

>>> df.show(10)

输出显示了10行数据，每行包含label和features两列。features列显示的是稀疏向量格式，比如(780,[127,128,129...]表示这是一个780维的稀疏向量。

这种格式在文本分析、图像识别等领域非常常用，因为很多特征都是稀疏的。

课程总结

好，今天的课程就到这里。我们学习了Spark MLlib的基础知识，包括：

1. **为什么要用Spark做机器学习** - 内存计算的优势
2. **MLlib库的整体架构** - 丰富的算法支持
3. **基本数据类型** - 向量、矩阵、标注点
4. **数据源处理** - LIBSVM、图像数据的加载

这些是使用MLlib进行机器学习的基础，希望大家课后多练习。下节课我们会继续学习统计分析工具和机器学习流水线。

有什么问题可以随时联系我，谢谢大家！