

# 基础数据类型



東北大學  
Northeastern University



HORIZON  
昊宸科技



1

数据类型

2

基本统计

3

算法演示

4

实例应用KMeans



# 基本数据类型



- Local Vector ( 向量 )
- Labeled point ( 带类别的向量 )
- Local matrix ( 本地矩阵 )
- Distributed matrix ( 分布式矩阵 )

- Vector ( 向量 )

  - Dense vector ( 稠密 )

  - Sparse vector ( 稀疏 )

  - Labeled point ( 带标签 )

- Matrix ( 矩阵 , 看做一堆向量构成的 )

  - Local Matrix

  - Distributed Matrix

# Vector Dense & Sparse



- 表格数据表示用户购买商品的矩阵，ABC 表示商品类别，ID表示用户ID

ID	A	B	C
1	1	1	0
2	1	0	1
3	1	0	0
4	0	0	0
5	0	0	0

Dense : [1 , 1 , 0]

Sparse : size 7

indices[0][1]

values[1][1]

# Vector Dense & Sparse



## ➤ Dense Vector ( 向量 )

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
```

```
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
```

## ➤ Sparse Vector ( 向量 )

➤ 参数1表示长度，参数2表示索引值，参数3索引对应的数值

```
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
```

表示 长度为3的vector，在下标为0 和 2的位置数值为 1 和 3

➤ P1 : 长度 p2 index& value

```
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

## ➤ 训练集

1200 w 条记录

500 个特征

稀疏度 10%

	Dense	Sparse
Storage	47GB	7GB
Time	240s	58s

结论：尽量使用稀疏向量来进行模型训练



➤ Label + Vector ( 调用MLLIB 分类算法 , 所有数据必须是 LabelPoint )

```
import org.apache.spark.mllib.regression.LabeledPoint  
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))  
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0,  
3.0)))
```

## ➤ Local Matrix

DenseMatrix

SparseMatrix

$$\begin{Bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{Bmatrix}$$

```
val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
```

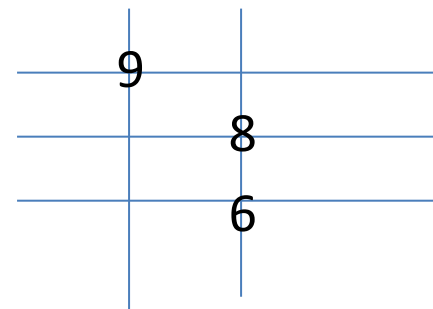
```
val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1), Array(9, 6, 8))
```

```
def sparse(numRows: Int, numCols: Int, colPtrs: Array[Int], rowIndices: Array[Int], values:  
Array[Double]): Matrix
```

Creates a column-major sparse matrix in Compressed Sparse Column (CSC) format.

<b>numRows</b>	number of rows
<b>numCols</b>	number of columns
<b>colPtrs</b>	the index corresponding to the start of a new column
<b>rowIndices</b>	the row index of the entry
<b>values</b>	non-zero matrix entries in column major

Annotations      @Since( "1.2.0" )



9	
	8
	6

## ➤ SparseMatrix

new **SparseMatrix**(numRows: Int, numCols: Int, colPtrs: Array[Int], rowIndices: Array[Int], values: Array[Double])

Column-major sparse matrix. The entry values are stored in Compressed Sparse Column (CSC) format. For example, the following matrix

```
1.0 0.0 4.0
0.0 3.0 5.0
2.0 0.0 6.0
```

is stored as values: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0], rowIndices=[0, 2, 1, 0, 1, 2], colPointers=[0, 2, 3, 6].

---

**numRows** number of rows

**numCols** number of columns

**colPtrs** the index corresponding to the start of a new column

**rowIndices** the row index of the entry. They must be in strictly increasing order for each column

**values** non-zero matrix entries in column major

---

*Annotations* @Since( "1.2.0" )

## ➤ SparseMatrix

new **SparseMatrix**(numRows: Int, numCols: Int, colPtrs: Array[Int], rowIndices: Array[Int], values: Array[Double])

Column-major sparse matrix. The entry values are stored in Compressed Sparse Column (CSC) format. For example, the following matrix

```
1.0 0.0 4.0
0.0 3.0 5.0
2.0 0.0 6.0
```

is stored as values: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0], rowIndices=[0, 2, 1, 0, 1, 2], colPointers=[0, 2, 3, 6].

---

**numRows**    number of rows

**numCols**    number of columns

**colPtrs**    the index corresponding to the start of a new column

**rowIndices** the row index of the entry. They must be in strictly increasing order for each column

**values**    non-zero matrix entries in column major

---

*Annotations*            @Since( "1.2.0" )

- RowMatrix
- IndexedRowMatrix
- CoordinateMatrix
- BlockMatrix

# Distributed Matrix—RowMatrix



## ➤ RowMatrix

**import** org.apache.spark.mllib.linalg.distributed.RowMatrix

```
val rows: RDD[Vector] = ... // an RDD of local vectors  
// Create a RowMatrix from an RDD[Vector].  
val mat: RowMatrix = new RowMatrix(rows)  
  
// Get its size.  
val m = mat.numRows()  
val n = mat.numCols()  
  
// QR decomposition  
val qrResult = mat.tallskinnyQR(true)
```

# Distributed Matrix—IndexedRowMatrix



## ➤ IndexedRowMatrix

```
case class IndexedRow(index: Long, vector: Vector)
```

```
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix
```

```
val rows: RDD[IndexedRow] = ... // an RDD of indexed rows
// Create an IndexedRowMatrix from an RDD[IndexedRow].
val mat: IndexedRowMatrix = new IndexedRowMatrix(rows)

// Get its size.
val m = mat.numRows()
val n = mat.numCols()

// Drop its row indices.
val rowMat: RowMatrix = mat.toRowMatrix()
```

➤ 常用算法计算奇异值，矩阵乘法等要求输入IndexedRowMatrix

# Distributed Matrix—CoordinateMatrix



➤ **CoordinateMatrix**      `case class MatrixEntry(i: Long, j: Long, value: Double)`

➤ 每一个元素是一个三元组，位置信息 + value 值

**import org.apache.spark.mllib.linalg.distributed.{CoordinateMatrix,  
MatrixEntry}**

```
val entries: RDD[MatrixEntry] = ... // an RDD of matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val mat: CoordinateMatrix = new CoordinateMatrix(entries)

// Get its size.
val m = mat.numRows()
val n = mat.numCols()

// Convert it to an IndexRowMatrix whose rows are sparse vectors.
val indexedRowMatrix = mat.toIndexedRowMatrix()
```



# Distributed Matrix—BlockMatrix



## ➤ BlockMatrix

由于拆分方便，利于分布式的矩阵计算

```
import org.apache.spark.mllib.linalg.distributed.{BlockMatrix, CoordinateMatrix, MatrixEntry}
```

分块矩阵将一个矩阵分成若干块，例如：

$$P = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}$$

可以将其分成四块

$$P_{11} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, P_{12} = \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, P_{21} = \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, P_{22} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}.$$

从而矩阵P有如下形式

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix}.$$

```
val entries: RDD[MatrixEntry] = ... // an RDD of (i, j, v) matrix entries
// Create a CoordinateMatrix from an RDD[MatrixEntry].
val coordMat: CoordinateMatrix = new CoordinateMatrix(entries)
// Transform the CoordinateMatrix to a BlockMatrix
val matA: BlockMatrix = coordMat.toBlockMatrix().cache()

// validate whether the BlockMatrix is set up properly. Throws an Exception when it is not valid.
// Nothing happens if it is valid.
matA.validate()

// calculate A^T A.
val ata = matA.transpose.multiply(matA)
```

# 汇总统计



## ➤ 基本统计值

min

max

mean

.....

```
val conf = new SparkConf().setMaster("local").setAppName("SummaryStatistics")
val sc = new SparkContext(conf)
val root = SummaryStatisticsDemo.getClass.getResource("/")
val rdd = sc.textFile(root + "correlations.csv")
val observations: RDD[Vector] = rdd.filter(_.split(",")(0) != "NumPregnancies").map {
  line =>
    val array = line.split(",")
    val dArray = array.map(_.toDouble)
    Vectors.dense(dArray)
}
// Compute column summary statistics.
val summary: MultivariateStatisticalSummary = Statistics.colStats(observations)
println(summary.mean) // a dense vector containing the mean value for each column
println(summary.variance) // column-wise variance
println(summary.numNonzeros) // number of nonzeros in each column
```

## ➤ 相关性系数(Correlations)

Correlations, 相关度量, 目前Spark支持两种相关性系数: 皮尔逊相关系数(pearson)和斯皮尔曼等级相关系数(Spearman)

Pearson 相关性:连续数据, 正态分布, 线性关系, 用pearson相关系数是最恰当;上述任一条件不满足, 就用spearman相关系数

Spearman 相关性:两个定序测量数据之间也用spearman相关系数, 不能用pearson相关系数

```
val seriesX: RDD[Double] = sc.parallelize(Array(1, 2, 3, 3, 5)) // a series
// must have the same number of partitions and cardinality as seriesX
val seriesY: RDD[Double] = sc.parallelize(Array(11, 22, 33, 33, 555))

// compute the correlation using Pearson's method. Enter "spearman" for Spearman's method. If a
// method is not specified, Pearson's method will be used by default.
val correlation: Double = Statistics.corr(seriesX, seriesY, "pearson")
println(s"Correlation is: $correlation")

val data: RDD[Vector] = sc.parallelize(
  Seq(
    Vectors.dense(1.0, 10.0, 100.0),
    Vectors.dense(2.0, 20.0, 200.0),
    Vectors.dense(5.0, 33.0, 366.0))
) // note that each Vector is a row and not a column

// calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method
// If a method is not specified, Pearson's method will be used by default.
val correlMatrix: Matrix = Statistics.corr(data, "pearson")
```

## ➤ 分层抽样(Stratified Sampling)

分层取样（Stratified sampling）顾名思义，就是将数据根据不同的特征分成不同的组，然后按特定条件从不同的组中获取样本，并重新组成新的数组。Spark RDD api 中提供两种方式。

`sampleByKey` 和 `sampleByKeyExact` 的区别在于 `sampleByKey` 每次都通过给定的概率以一种类似于掷硬币的方式来决定这个观察值是否被放入样本，因此一遍就可以过滤完所有数据，最后得到一个近似大小的样本，但往往不够准确。

`sampleByKeyExtra` 会对全量数据做采样计算。对于每个类别，其都会产生  $(fk \cdot nk)$  个样本，其中  $fk$  是键为  $k$  的样本类别采样的比例； $nk$  是键  $k$  所拥有的样本数。`sampleByKeyExtra` 采样的结果会更准确，有99.99%的置信度，但耗费的计算资源也更多。

分层抽样使用的场景，预测疾病，真正得疾病的样本数往往比较少

- 是数理统计学中根据一定假设条件由样本推断总体的一种方法
- 卡方检验(Stratified Sampling)

卡方检验就是统计样本的实际观测值与理论推断值之间的偏离程度，实际观测值与理论推断值之间的偏离程度就决定卡方值的大小。

Spark目前支持皮尔森卡方检测（Pearson's chi-squared tests），包括“适配度检定”（Goodness of fit）以及“独立性检定”（independence）。



## ➤ 适配度检定(Goodness of fit)

method: 方法,这里采用pearson方法。

Statistic: 检验统计量。简单来说就是用来决定是否可以拒绝原假设的证据。检验统计量的值是利用样本数据计算得到的，它代表了样本中的信息。检验统计量的绝对值越大，拒绝原假设的理由越充分，反之，不拒绝原假设的理由越充分。

degrees of freedom: 自由度。表示可自由变动的样本观测值的数目，

pValue: 统计学根据显著性检验方法所得到的P 值。一般以 $P < 0.05$  为显著，  $P < 0.01$  为非常显著，其含义是样本间的差异由抽样误差所致的概率小于0.05 或0.01。

一般来说，假设检验主要看P值就够了。

## ➤ 适配度检验(Goodness of fit)

```
val land1 = Vectors.dense(1000.0, 1856.0)
```

```
val land2 = Vectors.dense(400, 560)
```

```
val c1 = Statistics.chiSqTest(land1, land2)
```

单从结果来看，两组数据满足相同的分布

假设有两块土地，通过下列数据来检验其开红花的比率是否相同：

土地一， 开红花:1000，开兰花:1856

土地二， 开红花:400.，开兰花:560

```
c1: org.apache.spark.mllib.stat.test.StatTest$ChiSquaredTestSummary =  
Chi squared test summary:  
method: pearson  
degrees of freedom = 1  
statistic = 52.0048019207683  
pValue = 5.536682223805656E-13  
Very strong presumption against null hypothesis
```



## ➤ 独立性检验 (Indenpendence)

卡方独立性检验是用来检验两个属性间是否独立。其中一个属性做为行，另外一个做为列，通过貌似相关的关系考察其是否真实存在相关性。比如天气温变化和肺炎发病率。

**Statistic:** 检验统计量。简单来说就是用来决定是否可以拒绝原假设的证据。检验统计量的值是利用样本数据计算得到的，它代表了样本中的信息。检验统计量的绝对值越大，拒绝原假设的理由越充分，反之，不拒绝原假设的理由越充分。

**degrees of freedom:** 自由度。表示可自由变动的样本观测值的数目，

**pValue:** 统计学根据显著性检验方法所得到的P 值。一般以 $P < 0.05$  为显著，  $P < 0.01$  为非常显著，其含义是样本间的差异由抽样误差所致的概率小于0.05 或0.01。

一般来说，假设检验主要看P值就够了。

# 核密度估计 Kernel density estimation



- Spark MLlib 提供了一个工具类 `KernelDensity` 用于核密度估算，核密度估算的意思是根据已知的样本估计未知的密度，属于非参数检验方法之一。
- 核密度估计的原理是。观察某一事物的已知分布，如果某一个数在观察中出现了，可认为这个数的概率密度很大，和这个数比较近的数的概率密度也会比较大，而那些离这个数远的数的概率密度会比较小。Spark 1.6.2 版本支持高斯核 (Gaussian kernel)

```
val data: RDD[Double] = sc.makeRDD(List(1.0,2.0,3.0,4.0,5.0))  
// Construct the density estimator with the sample data and a standard deviation for  
// kernels  
val kd = new KernelDensity()  
    .setSample(data)  
// 其中setBandwidth表示高斯核的宽度，为一个平滑参数，可以看做是高斯核的标准差。  
// 构造了核密度估计kd，就可以对给定数据数据进行核估计：  
    .setBandwidth(3.0)  
  
// Find density estimates for the given values  
val densities = kd.estimate(Array(-1.0, 2.0, 5.0))
```

# Machine Learning

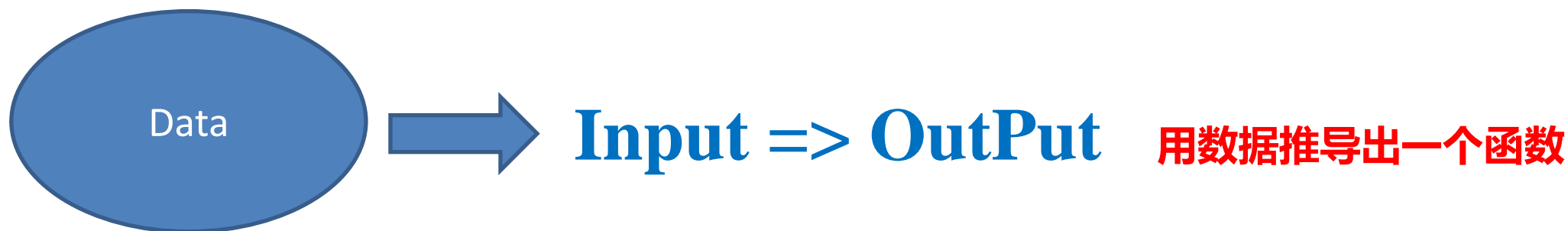


# 什么是Machine Learning



➤ Machine learning is a study of computer algorithms that improve automatically through experience.

通过不断的累计经验，让计算机能够更准确的做一些事情



## ➤ 有监督 (supervisor)

对具有概念标记（分类）的训练样本进行学习，以尽可能对训练样本集外的数据进行标记（分类）预测。常见应用场景是分类问题和回归问题。

## ➤ 无监督(unsupervisor)

没有人工参与预先分类的学习过程；对不含分类标签的样本数据进行学习，或是将具有相似特征的数据聚集在一起。应用场景包括关联规则的学习以及聚类等进行提取。

# 机器学习算法分类——有监督学习



- 训练集：用来训练model 的数据
- 测试集：验证model 的数据
- 样本：每一行数据
- 特征：每一列数据

NumPregn	PG2	DBP	TSFT	SI2	BMI	DPF	Age	Class
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	26	1
10	115	0	0	0	35.3	0.134	29	0
2	197	70	45	543	30.5	0.158	53	1
8	125	96	0	0	0	0.232	54	1
4	110	92	0	0	37.6	0.191	30	0
10	168	74	0	0	38	0.537	34	1
10	139	80	0	0	27.1	1.441	57	0
1	189	60	23	846	30.1	0.398	59	1
5	166	72	19	175	25.8	0.587	51	1
7	100	0	0	0	30	0.484	32	1

以分类为例我们最终要得到这样的一个分类器

$f(\text{NumPregnancies}, \text{PG2}, \text{BDP} \dots) \Rightarrow \text{Class}(0, \text{或者} 1)$

## ➤ 有监督 (supervisor)

分类算法：离散变量预测，如预测明天是阴、晴还是雨。

回归算法：连续变量预测，如预测明天温度是多少

## ➤ 分类算法

决策树、KNN、SVM、贝叶斯分类、感知器

## ➤ 回归算法

线性回归

## ➤ 无监督(unsupervisor)

聚类

- 两个算法包
- Spark.mllib: 包含原始的API,构建在RDD 之上
- Spark.ml:基于DataFrame 构建高级API

Spark.ml 具备更优秀的性能和更好的扩展性

Spark.mllib 仍然在更新，并且包含更多的算法



➤ MLlib 是 Apache Spark 中的组件之一，专注于机器学习

MLlib 是 Spark 中的核心机器学习库

由 AMPLab 实验室的 MLbase 团队开发

由来自各个机构的超过80个代码贡献者

支持 Scala, Python, Java 和 R 语言

# 聚类算法Clustering



➤ Clustering is an **unsupervised** learning problem.

因为目标是未知的，所以无监督学习技术不会学习如何预测目标值。但是它可以学习数据的结构并找出相似输入的群组，或者学习哪些输入类型可能出现，哪些输入类型不可能出现。

➤ K-Means聚类算法主要分为三个步骤:

(1) 第一步是为待聚类的点寻找聚类中心

(2) 第二步是计算每个点到聚类中心的距离，将每个点聚类到离该点最近的聚类中去

(3) 第三步是计算每个聚类中所有点的坐标平均值，并将这个平均值作为新的聚类中心

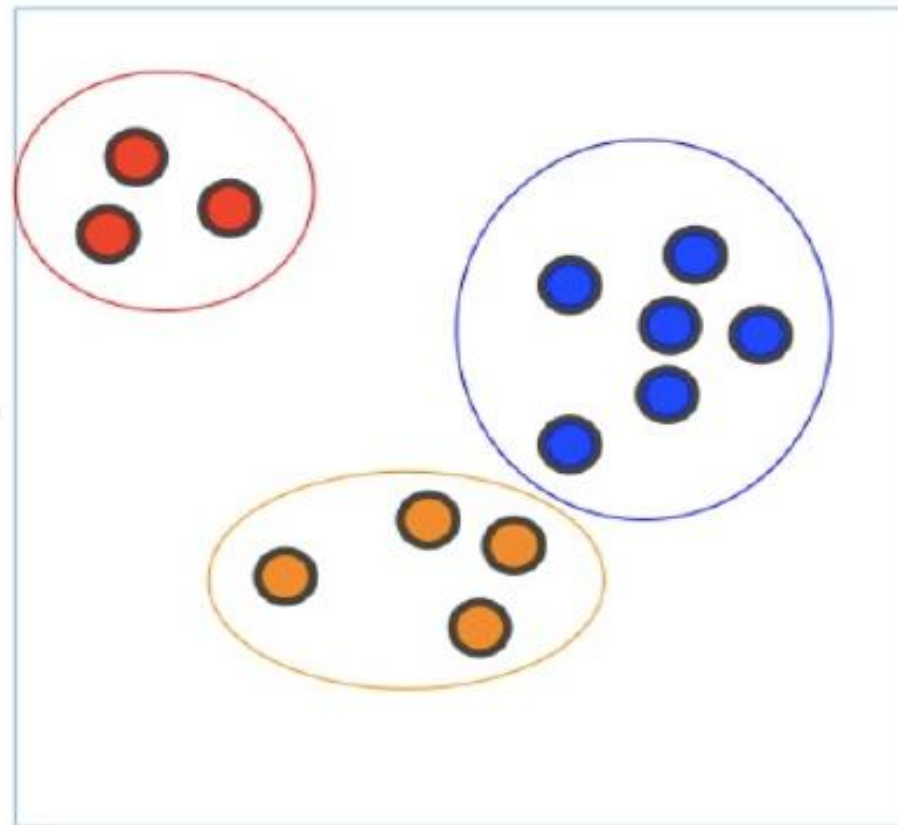
反复执行(2)、(3)，直到聚类中心不再进行大范围移动或者聚类次数达到要求为止

# K-Means

Given data points

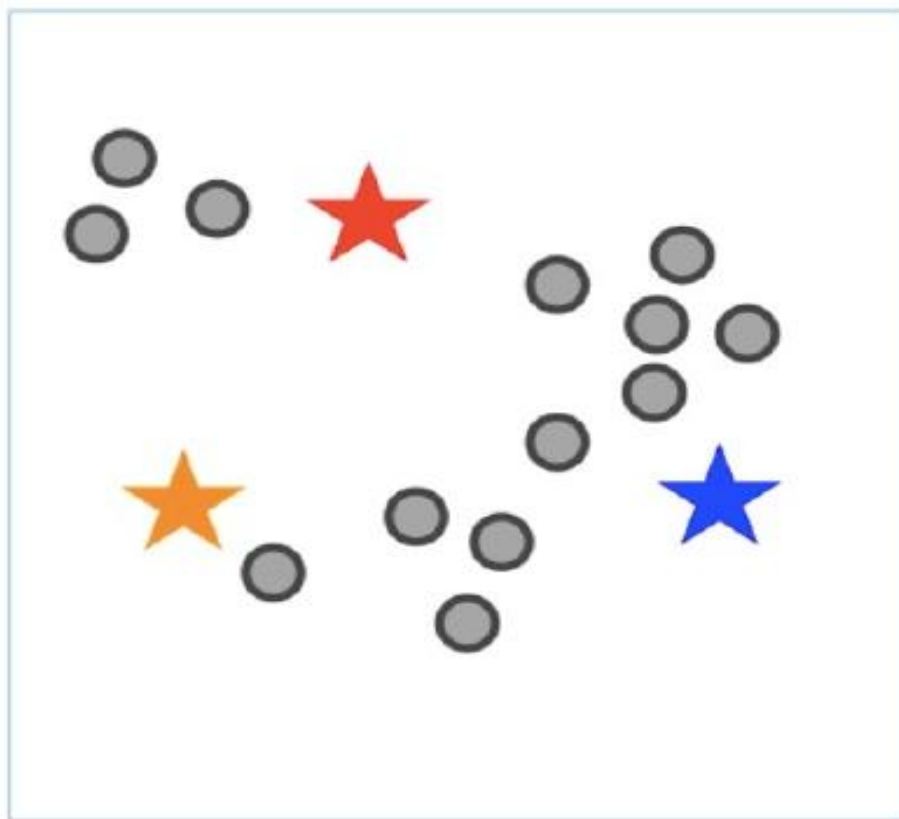


Find meaningful clusters

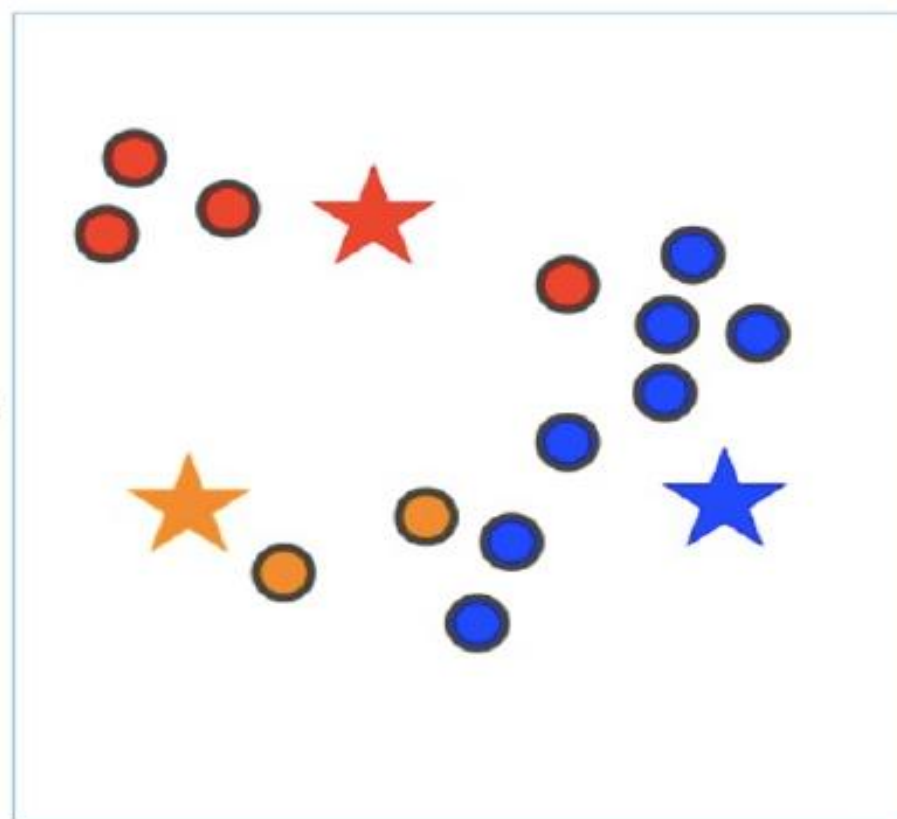


# K-Means

Choose cluster centers

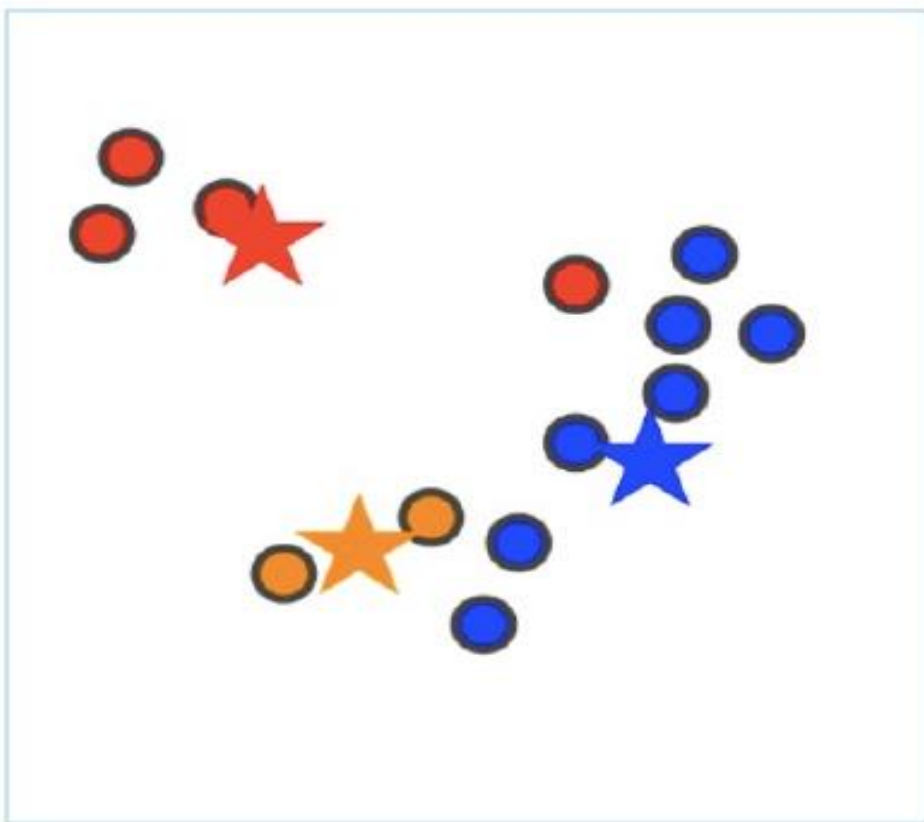


Assign points to clusters

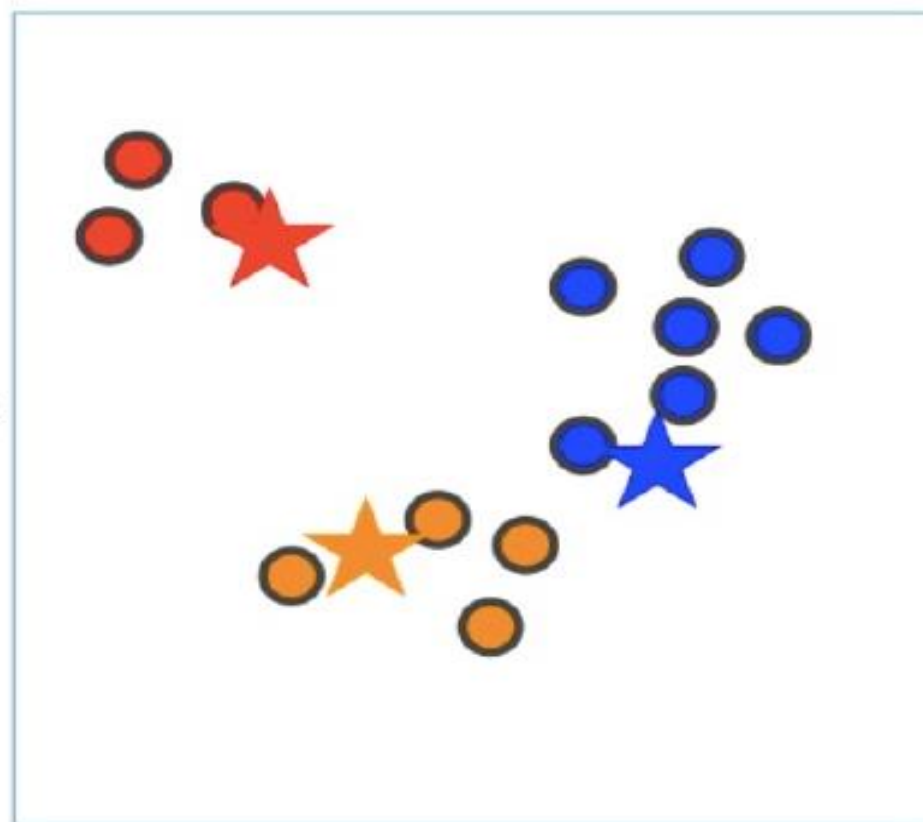


# K-Means

Choose cluster centers

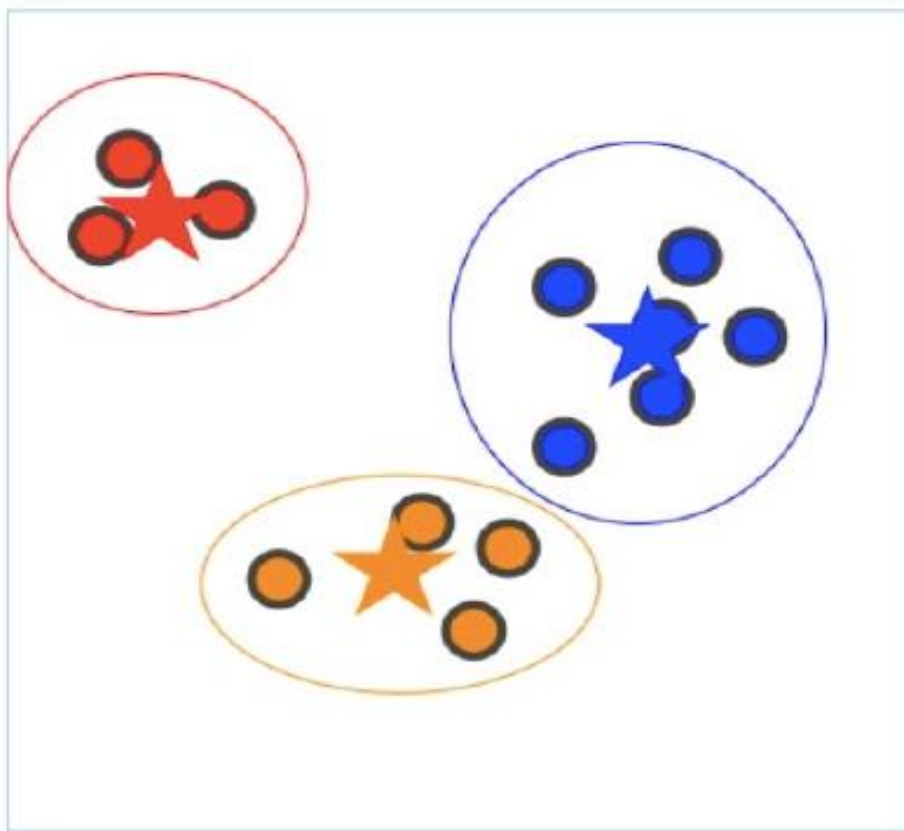


Assign points to clusters

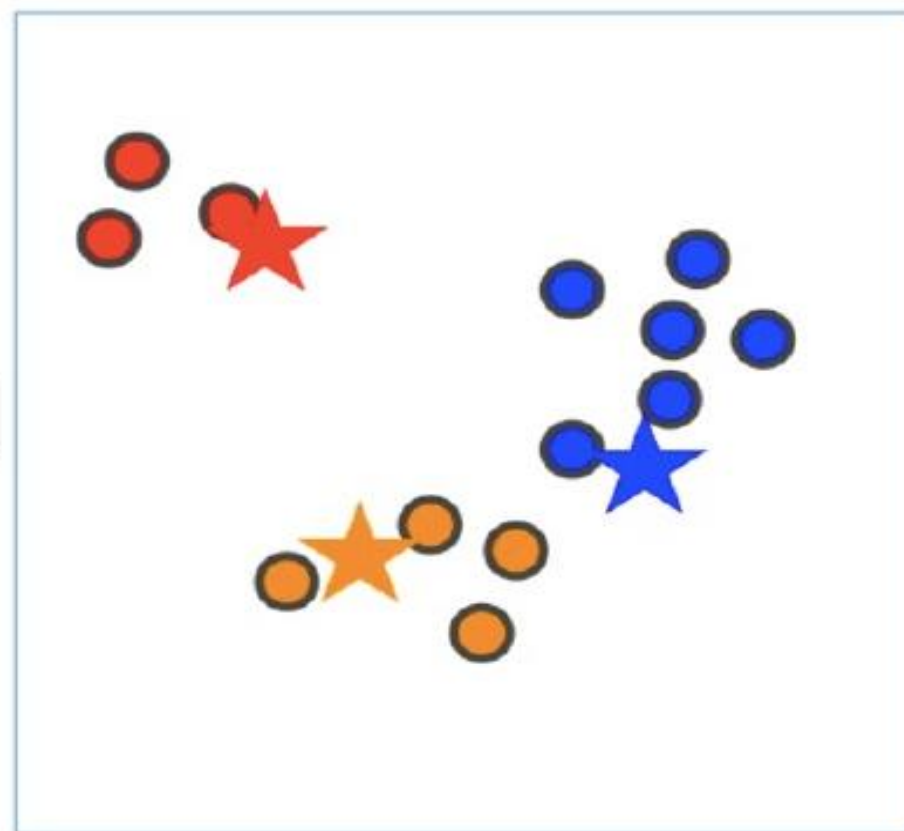


# K-Means

Choose cluster centers



Assign points to clusters





- K-Means 参数
- K 聚类个数，默认为2
- maxIterations 迭代次数，默认20
- Runs 并行度，默认为1
- initializationMode : 初始中心算法，默认 "k-means||" ;
- initializationSteps : 初始步长，默认5 ;
- epsilon : 中心距离阈值，默认 $1e-4$  ;
- seed : 随机种子。

## ➤ 数据集 KDD Cup 1999 数据集

kddcup.data.gz : 全量数据的数据集17.2M , 解压开708M包含490万个连接。数据集中包含每个连接的信息 , 发送的字节数、登录次数、TCP错误数等 , 数据格式为CSV。

kddcup.data\_10\_percent.gz : 10%的数据 , 可以用于探索。

kddcup.newtestdata\_10\_percent\_unlabeled.gz : 无标签的10%数据

## ➤ 初步尝试聚类

sc.textFile读取数据集 , 先尝试探索数据集看看数据有哪些类别标号 , 以及每类样本有多少

注意数据中有些特征不是数值类型 , 比如第二列可能取值为tcp , udp或 icmp , 但是Kmeans 算法都要求其数值型。

## ➤ K值的选择

## ➤ 1.实验报告

根据实验指导书编写实验报告，并在下节Spark课，上课之前提交给学委



联系我们

如何联系我们...



数据 引领变革

缔造新型 价值

联系  
我们

地址：沈阳市和平区三好街84-8号易购大厦319A

电话：024-88507865

邮箱：horizon@syhc.com.cn

公司网站：<http://www.syhc.com.cn>





# THANKS

