

Outline

Module 1 : 大數據簡介

Module 2 : Hadoop Ecosystem介紹

Module 3 : Hadoop 平台安裝

Module 4 : Hadoop 分散式檔案系統 (HDFS)

Module 5 : Hadoop MapReduce

Module 6 : Apache Hive

Module 7 : Sqoop與Flume

Module 8 : Apache Spark

Module 9 : Spark 平台安裝

Module 10 : RDD — Resilient distributed dataset

Module 11 : Scala 程式開發基礎

Module 12 : Spark SQL 及 DataFrame

Module 13 : Spark 機器學習函式庫(MLlib)



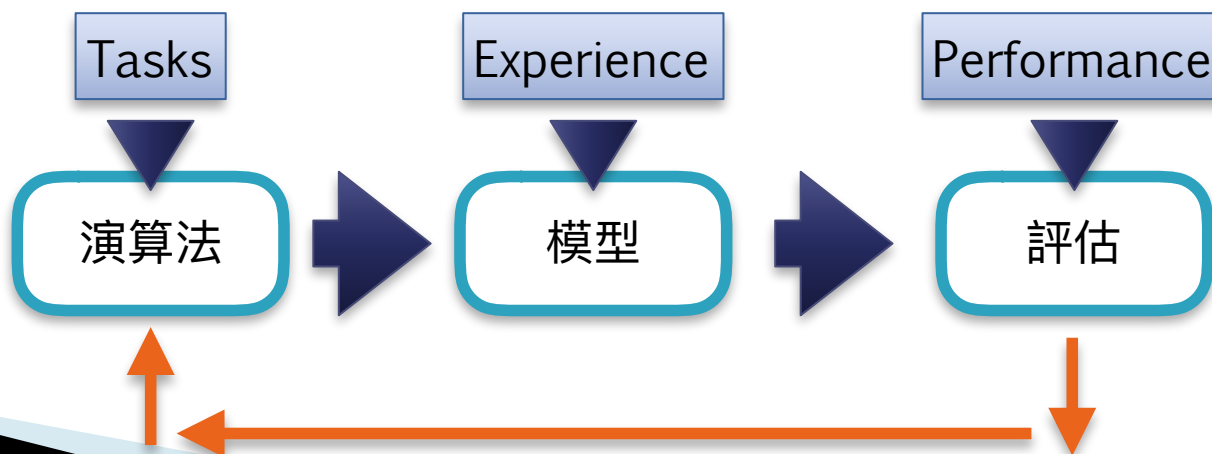
先談談機器學習

▶ 機器學習的定義

- 機器學習是一門**人工智慧的科學**，該領域的主要研究物件是人工智慧，特別是**如何在經驗學習中改善具體演算法的效能**。
- 機器學習是**用資料或以往的經驗**，以此**最佳化電腦程式的效能標準**。

▶ 常見分類

- 監督學習
- 非監督學習



機器學習的常見應用

- ▶ 資料探勘
- ▶ 電腦視覺
- ▶ 自然語言處理
- ▶ 生物特徵識別
- ▶ 搜尋引擎
- ▶ 醫學診斷
- ▶ 檢測信用卡欺詐
- ▶ 證券市場分析
- ▶ DNA序列測序
- ▶ 語音和手寫識別
- ▶ 戰略遊戲和機器人應用

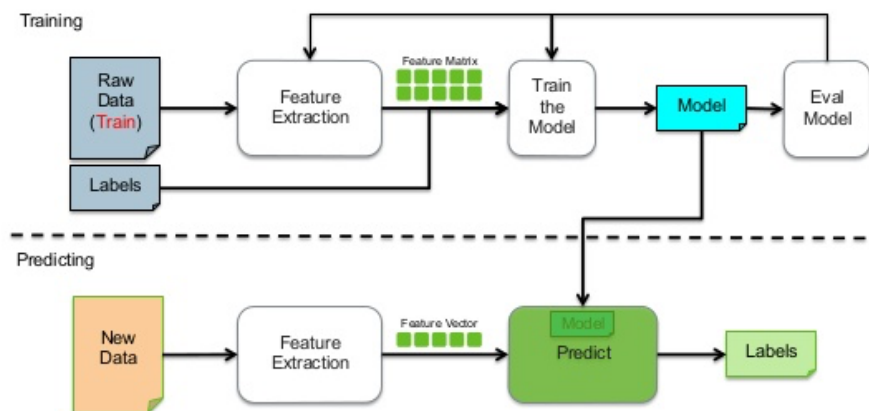


機器學習分類

▶ 監督學習(Supervised learning)

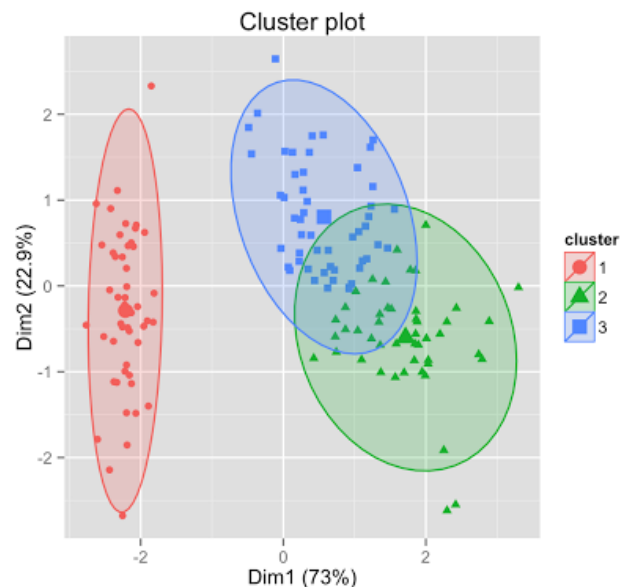
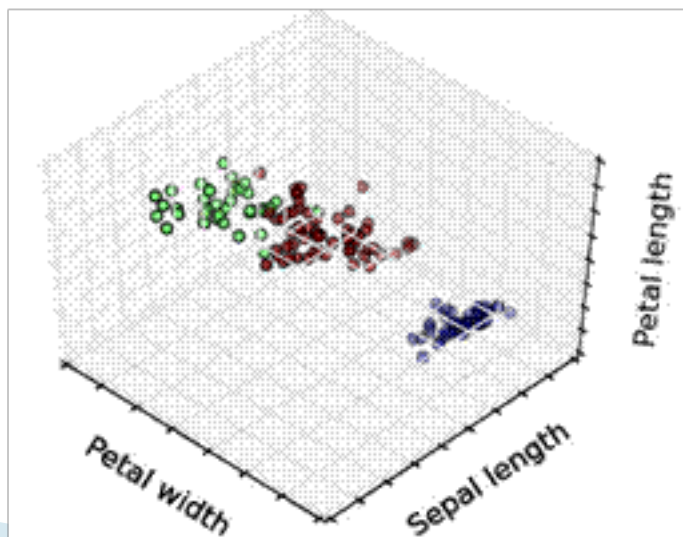
- 從給定的訓練資料集(Training Set)中學習出一個函式，當新的資料到來時，可以根據這個函式預測結果。
- 監督學習的訓練集要求是包括輸入和輸出，也可以說是特徵(Features)和目標(Label)。
- 常見的監督學習演算法包括Regression和Classification類演算法(如決策樹、最小平方法等)。

Supervised Learning Workflow



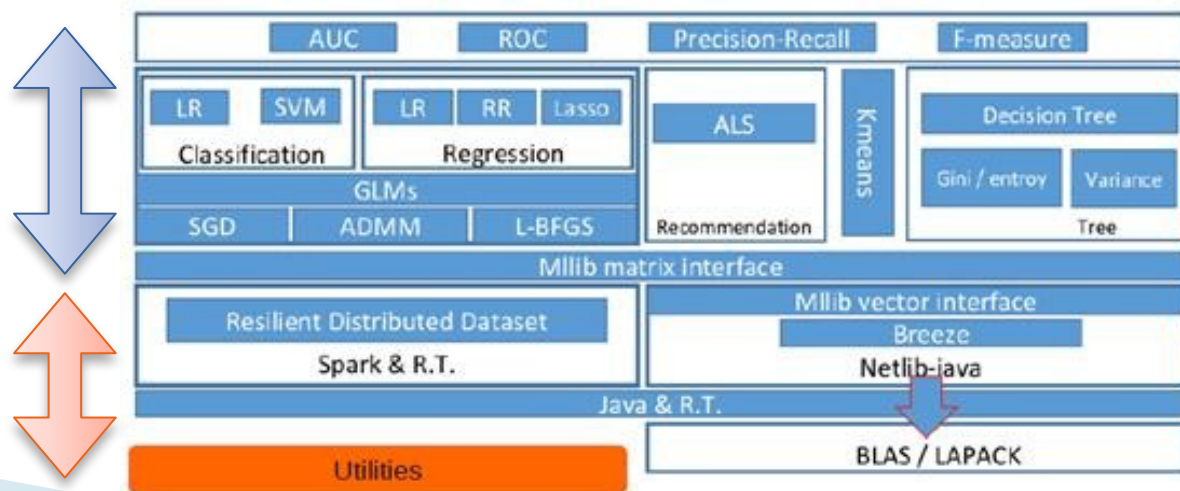
機器學習分類

- ▶ 非監督學習(Unsupervised learning)
 - 非監督學習與監督學習相比，訓練集沒有人為標註的結果(沒有Label)。
 - 模型用來推斷出資料的內在結構。
 - 常見的非監督學習演算法為Clustering類演算法(如KMeans)。



開始來談Spark MLlib吧…

- ▶ MLlib是Machine Learning library的縮寫，是Spark對機器學習常用演算法的實作
- ▶ 包含二個主要部份：
 - 底層基礎：RDD、矩陣及向量處理函式庫
 - 演算法函式庫：各類演算法及效能評估方法實作



Spark MLlib內建演算法概觀

Spark MLlib		
	Categorical Qualitative	Continuous Quantitative
Unsupervised Extracting structure	Clustering K-means	Dimension Reduction Singular Value Decomposition (SVD) Principal Component Analysis (PCA)
Supervised Making prediction	Classification Naive Bayes Decision Trees Ensembles of Trees (Random Forests and Gradient-Boosted Trees)	Regression linear models Support Vector Machines logistic regression linear regression
Recommender Associating user item	Collaborative Filtering Alternating Least Squares (ALS)	
Optimization Finding minima		Optimization Stochastic Gradient Descent Limited-memory BFGS (L-BFGS)
Feature Extraction Processing text	Feature Extraction Transformation TF-IDF - Word2Vec Standard Scaler - Normalizer	

Spark MLlib實戰－Let's biking

- ▶ 任務：由Bike Sharing Dataset中的各項外在因素(如天候，假日)與租借量的數據來建立預測模型，以預測新的外在因素下可能的租借量
- ▶ 資料集：<https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset>
 - 本檔案是由自行車共享系統收集而來，共兩個檔案：
 - hour.csv: 記錄2011.01.01~2012.12.30每小時的外在因素及租借量，共17,379筆資料
 - day.csv: 為hour.csv資料的匯總（以日為單位）



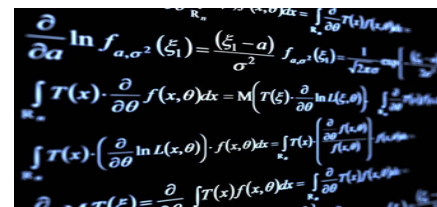
Bike Sharing Dataset欄位說明

	屬性	資料型態	描述
1	instant	數值	record index
2	dteday	數值	date
3	season	類別	season (springer , summer , fall , winter)
4	yr	類別	year (2011, 2012)
5	mnth	類別	month (1 to 12)
6	hr	類別	hour (0 to 23) (for hour.csv only)
7	holiday	數值	weather day is holiday or not
8	weekday	數值	day of the week (0 to 6)
9	workingday	數值	if day is neither weekend nor holiday is 1, otherwise is 0.
10	weathersit	類別	Clear , Mist , Light Snow , Heavy Rain (1 to 4)
11	temp	數值	Normalized temperature in Celsius. The values are divided to 41 (max)
12	atemp	數值	Normalized feeling temperature in Celsius. The values are divided to 50 (max)
13	hum	數值	Normalized humidity. The values are divided to 100 (max)
14	windspeed	數值	Normalized wind speed. The values are divided to 67 (max)
15	casual	數值	count of casual users
16	registered	數值	count of registered users
17	cnt	數值	count of total rental bikes including both casual and registered

— Features
— Label

實作情境及對應的Spark MLlib功能

- ▶ 快速的了解資料的概括狀況
 - 取得各欄位概述統計量(**Summary Statistics**): 使用 **MultivariateStatisticalSummary** 類別及 **Statistics** 物件
 - 取得各Feature欄位(外在因素)與Label欄位(租借量)間的相關性 (**correlation**): 使用 **Statistics** 物件
- ▶ 觀察資料的群聚狀況
 - **Clustering**: 使用 **KMeans** 物件
- ▶ 預測每天 / 每小時租借量 **是否** 會超過特定門檻
 - **Classification**: 使用 **Decision Tree** 及 **LogisticRegressionWithSGD** 物件
- ▶ 預測每天 / 每小時 **租借量**
 - **Regression**: 使用 **Decision Tree** 及 **LinearRegressionWithSGD** 物件


$$\begin{aligned} \frac{\partial}{\partial a} \ln f_{a, \sigma^2}(\xi_1) &= \frac{(\xi_1 - a)}{\sigma^2} \\ f_{a, \sigma^2}(\xi_1) &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(\xi_1 - a)^2}{2\sigma^2}\right\} \\ \int_{-\infty}^{\infty} T(x) \cdot \frac{\partial}{\partial \theta} f(x, \theta) dx &= M\left(T(\xi) \cdot \frac{\partial}{\partial \theta} \ln f(\xi, \theta)\right) \\ \int_{-\infty}^{\infty} T(x) \cdot \left(\frac{\partial}{\partial \theta} \ln f(x, \theta)\right) \cdot f(x, \theta) dx &= \int_{-\infty}^{\infty} T(x) \cdot \left(\frac{\partial}{\partial \theta} f(x, \theta)\right) dx \\ \frac{\partial}{\partial \theta} \ln T(\xi) &= \frac{\partial}{\partial \theta} \int_{-\infty}^{\infty} T(x) f(x, \theta) dx = \int_{-\infty}^{\infty} T(x) \frac{\partial}{\partial \theta} f(x, \theta) dx \end{aligned}$$

Outline

- ▶ **Spark MLlib 實戰**
 - 概述統計量(summary statistics)
 - Clustering
 - Regression




快速了解資料的概括狀況

▶ 概述統計量(Summary Statistics)

- 給資料快速且簡單描述，包含**最大值**，**最小值**，**平均數**，**變異數**以及**總數量**等
- 只適用於**數值型態**欄位
- Spark的實作方法：
 - 方法1: 針對**各別**數值欄位取出該欄位所有值，存入**RDD[Double/Float/Int]**中，再呼叫**RDD的stats方法**取得該欄位之統計量
 - 方法2: 將**所有**數值欄位的欄位值存入**RDD[Vector]**中，再透過**Statistics.colStats方法**取得所有欄位統計值

Day	Height
12	6.5
13	6.2
14	6.6
15	7.1
16	7.2
17	6.8
18	6.2
19	6.4
20	7.3
21	7.1
22	6.3
23	6.8
24	6.4



Average Height = 6.68

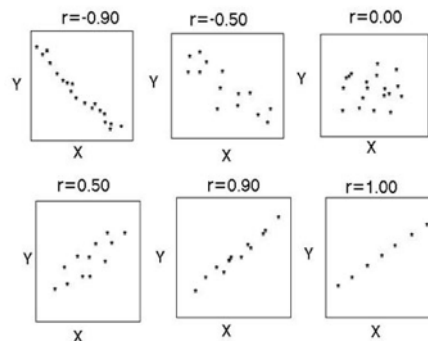
Minimum Height = 6.2

Maximum Height = 7.3

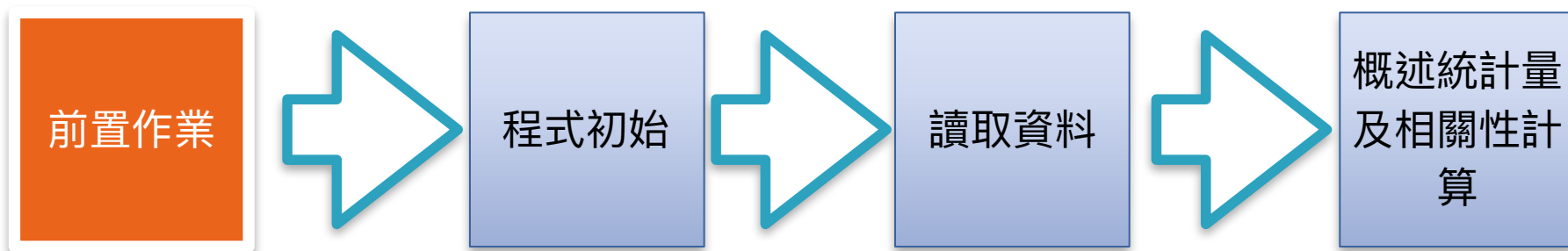
快速了解資料的概括狀況

- ▶ 取得各外在因素欄位與租借量欄位間的相關性(**correlation**)
 - 統計學中，相關(**Correlation**，或稱**相關係數**)，顯示**兩個隨機變量之間線性關係的強度和方向**
 - Spark中目前支援**Pearson** 和 Spearman 相關
 - r透過呼叫**Statistics.corr**方法取得
 - $0 < |r| < 0.3$ (低度相關)
 - $0.3 \leq |r| < 0.7$ (中度相關)
 - $0.7 \leq |r| < 1$ (高度相關)
 - $r = 1$ (完全正相關)

$$r = \frac{\sum XY - \frac{(\sum X)(\sum Y)}{n}}{\sqrt{\left(\sum X^2 - \frac{(\sum X)^2}{n}\right) \left(\sum Y^2 - \frac{(\sum Y)^2}{n}\right)}}$$



實作的主要步驟

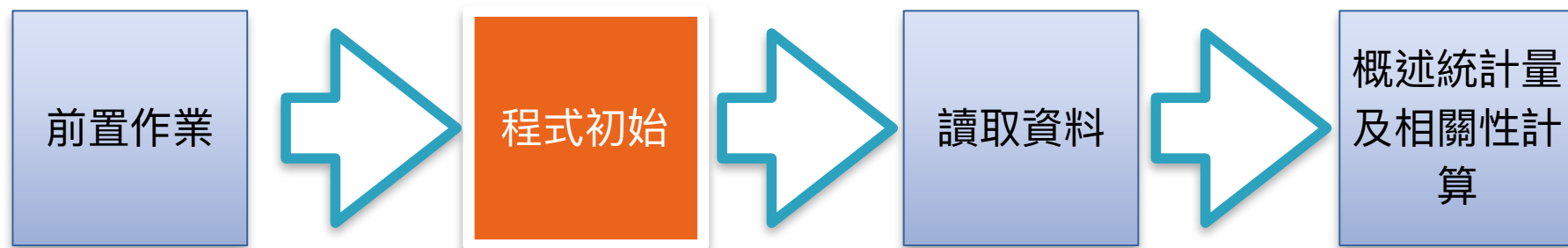


- A. 建立Scala專案
- B. 新增Package及Scala Object
- C. 新增data Folder、放置所需資料檔
- D. 設定專案所需Library

前置作業

- ▶ ScalaIDE中的Scala專案、folder、package及Object結構
 - SummaryStat (專案名稱)
 - src
 - bike (package名稱)
 - BikeSummary (scala object名稱)
 - data (folder名稱)
 - hour.csv
- ▶ 設定Build Path
 - 引入 Spark安裝目錄/assembly/target/scala-2.11/jars/
 - 確定專案內的Scala container版本為2.11.8

實作的主要步驟



- A. import所需套件
- B. 定義main方法作為Driver Program的程式入口
- C. 設定程式的Log層級
- D. 初始化SparkContext

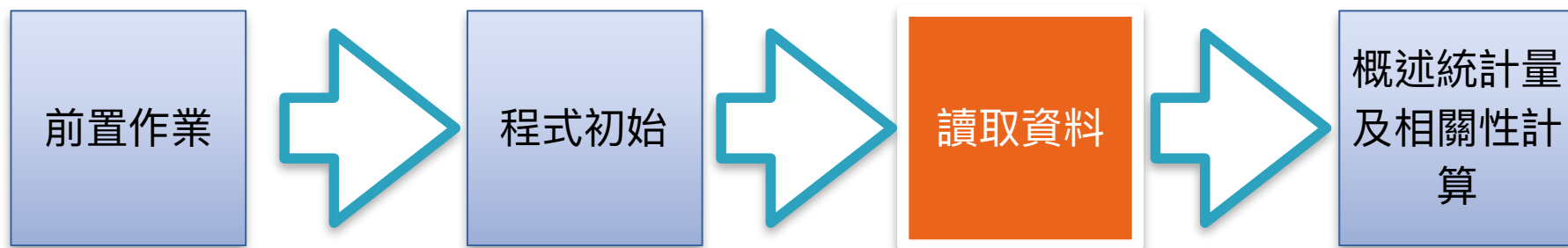
程式初始

```
//import spark rdd library
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
//import Statistics library
import org.apache.spark.mllib.stat.{ MultivariateStatisticalSummary,
Statistics }

object BikeSummary {
  def main(args: Array[String]): Unit = {
    Logger.getLogger(com).setLevel(Level.OFF) //set logger
    //initialize SparkContext
    val sc = new SparkContext(new SparkConf().setAppName("BikeSummary").setMaster("local[*]"))
  }
}
```

- ▶ 使用spark-shell時sparkContext已自動初始化在sc變數中
- ▶ 自己開發Driver Program要自行建立sc
 - appName - 在管理介面識別Driver Program用
 - master - 指定master之URL

實作的主要步驟



- ▶ 實作prepare方法
 - 實作讀取input file、取出Features及Label欄位、並將欄位內容轉為RDD

先探討一下prepare方法的實作

```
def prepare(sc: SparkContext): RDD[???] = {  
    val rawData=sc.textFile(data/hour.csv) //read hour.csv in data folder  
    val rawDataNoHead=rawData.mapPartitionsWithIndex { (idx, iter) =>  
        { if (idx == 0) iter.drop(1) else iter } } //ignore first row(column name)  
    val lines:RDD[Array[String]] = rawDataNoHead.map { x =>  
        x.split(, ).map { x => x.trim() } } //split columns with comma  
    val bikeData:RDD[???]=lines.map{ ... } //??? depends on your impl  
}
```

- ▶ 在lines.map中實作取出features(第3~14欄)及label(第17欄)的動作並回傳RDD
- ▶ 回傳的RDD型態對後面程式的實作影響甚鉅，請認真考慮以下幾種回傳型態的利弊：
 - 回傳RDD[Array]
 - 回傳RDD[Tuple]
 - 回傳RDD[BikeShareEntity]

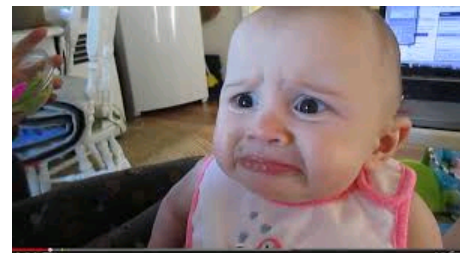
回傳型態設計探討

▶ 回傳RDD[Array]:

- val bikeData:RDD[Array[Double]] =lines.
map{x=>(x.slice(3,13).map(.toDouble) ++ Array(x(16).toDouble))}
- 利弊: prepare實作容易，後面用起來痛苦(要記欄位在Array中的index)，也容易出包

▶ 回傳RDD[Tuple]:

- val bikeData:RDD[(Double, Double, Double, ..., Double)]
=lines.map{case(season,yr,mnth,...,cnt)=>(season.toDouble, yr.toDouble,
mntth.toDouble,...cnt.toDouble)}
- 利弊: prepare實作較不易，後面用起來痛苦，比較不會出包(可用較佳的變數命名來接回傳值)
- 例: val features = bikeData.map{case(season,yr,mnth,...,cnt)=> (season, yr,
math, ..., windspeed)}}



回傳型態設計探討

▶ 回傳RDD[自訂Class]：

- `val bikeData:RDD[BikeShareEntity] = lines.map{ x=> BikeShareEntity(...) }`
- 利弊: prepare實作痛苦，後面用起來快樂(用entity物件操作，**不用管欄位位置、抽象化**)，不易出包
- 例: `val labelRdd = bikeData.map{ ent => { ent.label } }`

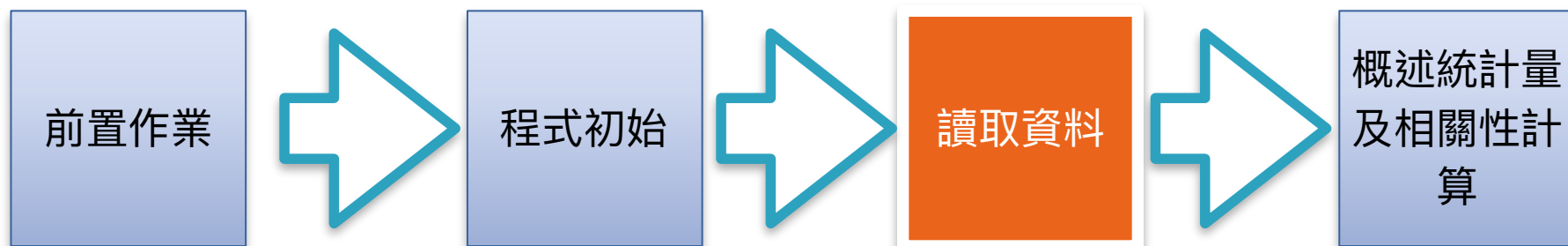
以Case Class來宣告自訂Class：

```
case class BikeShareEntity(instant: String, dteday:String, season:Double,
    yr:Double, mnth:Double, hr:Double, holiday:Double, weekday:Double,
    workingday:Double, weathersit:Double, temp:Double,
    atemp:Double, hum:Double, windspeed:Double, casual:Double,
    registered:Double, cnt:Double)
```

讀取資料時，透過map來建立RDD[BikeShareEntity]：

```
val bikeData = rawData.map { x =>
    BikeShareEntity(x(0), x(1), x(2).toDouble, x(3).toDouble, x(4).toDouble,
    x(5).toDouble, x(6).toDouble, x(7).toDouble, x(8).toDouble,
    x(9).toDouble, x(10).toDouble, x(11).toDouble, x(12).toDouble,
    x(13).toDouble, x(14).toDouble, x(15).toDouble, x(16).toDouble) }
```

實作的主要步驟

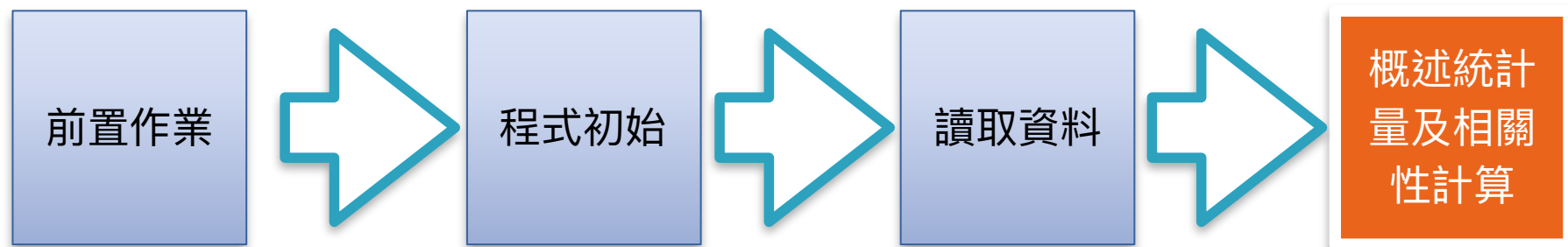


- ▶ 定義裝載資料的類別(Class)
- ▶ 實作prepare方法
 - 實作讀取input file、取出Features及Label欄位、並將欄位內容轉為RDD

加入Entity的Class定義

```
//import spark rdd library
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
//import Statistics library
import org.apache.spark.mllib.stat.
{ MultivariateStatisticalSummary, Statistics }
object BikeSummary {
  case class BikeShareEntity(·····)
  def main(args: Array[String]): Unit = {
    Logger.getLogger(com).setLevel(Level.OFF) //set logger
    //initialize SparkContext
    val sc = new SparkContext(new
      SparkConf().setAppName("BikeSummary").setMaster("local[*]"))
  }
}
```

實作的主要步驟



- ▶ 實作getFeatures方法
 - 決定要計算統計量的欄位
- ▶ 實作printSummary方法
 - 取得各欄位的概述統計量並列印在console
- ▶ 實作printCorrelation方法
 - 計算各欄位與租借量的相關性並列印在console

實作概述統計量的取得

getFeatures方法

```
def getFeatures(bikeData: BikeShareEntity): Array[Double] = {  
    //取出非會員、會員的租借量、以及租借量總和來求統計量  
    val featureArr = Array(bikeData.casual, bikeData.registered, bikeData.cnt)  
    featureArr  
}
```

printSummary方法

```
def printSummary(entRdd: RDD[BikeShareEntity]) = {  
    val dvRdd = entRdd.map { x => Vectors.dense(getFeatures(x)) } //組成RDD[Vector]  
    //呼叫Statistics.colStats取得Summary Statistics  
    val summaryAll = Statistics.colStats(dvRdd)  
    println("mean:" + summaryAll.mean.toArray.mkString(",")) //列印各欄位的平均值  
    println("variance:" + summaryAll.variance.toArray.mkString(",")) //列印各欄位的變異數  
}
```

```
===== summary all =====  
mean      :    0.497,    0.627,    0.19,    35.676,    153.787,    189.463  
variance   :    0.037,    0.037,    0.015,  2430.986, 22909.028, 32901.461  
nonZero Cnt : 17379.0, 17357.0, 15199.0, 15798.0, 17355.0, 17379.0
```

計算各欄位與租借量之相關性

printCorrelation方法

```
def printCorrelation(entRdd: RDD[BikeShareEntity]) = {  
  //取出租借量存入RDD[Double]  
  val cntRdd = entRdd.map { x => x.cnt }  
  val yrRdd = entRdd.map { x => x.yr } //取出年度值  
  val yrCorr = Statistics.corr(yrRdd, cntRdd) //計算兩個欄位的相關性  
  println("correlation:%s vs %s: %f".format(yr, cnt, yrCorr))  
  val seaRdd = entRdd.map { x => x.season } //取出season值  
  val seaCorr = Statistics.corr(seaRdd, cntRdd)  
  println("correlation:%s vs %s: %f".format(season, cnt, seaCorr))  
}
```

```
===== print Correlation =====  
correlation: yr vs cnt: 0.250495  
correlation: season vs cnt: 0.178056
```

練習：概述統計量的取得

A. 基礎實作練習

- 下載[BikeSummary.scala](#)放入SummaryStat專案中
- 下載[hour.csv](#)放入data目錄中
- 完成BikeSummary的實作(實作TODO部份)

B. 進階實作練習 — 以年月為篩選條件，取得各年月各欄位的概述統計量

- 在getFeatures及printSummary方法中實作以下邏輯
 - 在console中增加輸出溫度(temp)、溼度(hum)、風速(windspeed)的統計量
 - 以yr及mnth欄位為篩選條件，篩選出各年月的資料，取得溫度(temp)、溼度(hum)、風速(windspeed)、租借量(cnt)的概述統計量並在console印出

提示：

```
for (yr <- 0 to 1)
  for (mnth <- 1 to 12) {
    val yrMnRdd = entRdd.filter { ??? }.map { x => Vectors.dense(getFeatures(x)) }
    val summaryYrMn = Statistics.colStats( ??? )
    println("==== summary yr=%d, mnth=%d =====".format(yr,mnth))
    println("mean:" + ???)
    println("variance:" + ???)
  }
```

```
==== summary yr=0, mnth=1 =====
mean      :0.197, 0.574, 0.197, 4.467, 51.041, 55.507
variance:0.006, 0.029, 0.015, 38.238, 2144.761, 2363.968
==== summary yr=0, mnth=2 =====
mean      :0.284, 0.56, 0.229, 9.618, 64.673, 74.291
variance:0.013, 0.05, 0.026, 255.23, 3058.834, 4048.268
```

練習：計算各欄位與租借量之相關性

A. 基礎實作練習

- 完成[BikeSummary](#)中的printCorrelation方法
- 計算[hour.csv](#)各數值欄位[yr~windspeed]與cnt之相關性，並輸出在console

B. 進階實作練習 — 產生新的feature，並計算相關性

- 在printCorrelation方法中實作以下邏輯
 - 將yr和mnth組成新feature(名稱：yrmo， $yrmo = yr * 12 + mnth$)，計算yrmo及cnt的相關性

```
===== print Correlation =====  
correlation: yr vs cnt: 0.250495  
correlation: season vs cnt: 0.178056  
correlation: mnth vs cnt: 0.120638  
correlation: hr vs cnt: 0.394071  
correlation: holiday vs cnt: -0.030927  
correlation: weekday vs cnt: 0.026900  
correlation: workingday vs cnt: 0.030284  
correlation: weathersit vs cnt: -0.142426  
correlation: temp vs cnt: 0.404772  
correlation: atemp vs cnt: 0.400929  
correlation: hum vs cnt: -0.322911  
correlation: windspeed vs cnt: 0.093234
```

Outline

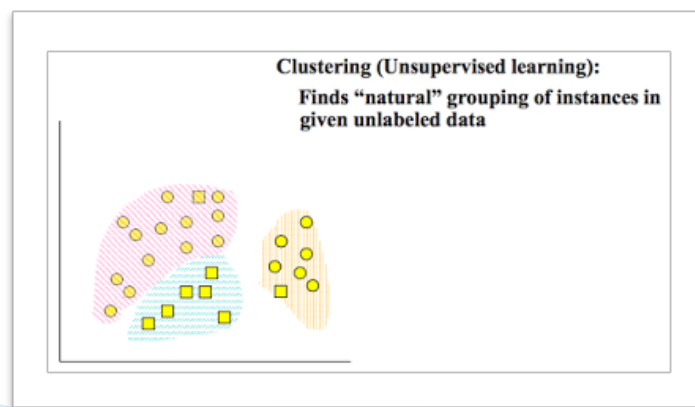
- ▶ **Spark MLlib 實戰**

- 概述統計量(summary statistics)
- **Clustering**
- Regression



Clustering (分群) 演算法介紹

- ▶ 是**非監督式學習**的演算法，預先準備好的Traing Set並不被特別標識(**沒有Label**)，常用以推斷數據的一些內在結構
- ▶ 將**比較相似的樣本**聚集在一起，形成**叢集(cluster)**。
- ▶ 以『**距離**』作為分類的依據，『**相對距離**』愈近的，『**相似程度**』愈高，歸類成同一叢集
- ▶ 常見應用
 - 客戶分群分析、產品內容或特性分類、相似的基因組或圖像特徵分析



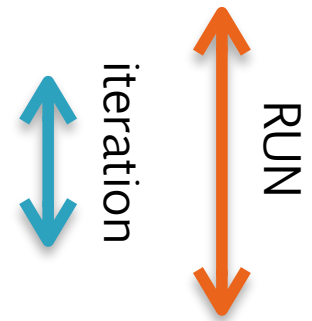
K-Means演算法介紹

- ▶ 最常之分群演算法，原理簡單易理解，效果亦佳
- ▶ 對於已知資料集(x_1, x_2, \dots, x_n)，K-Means要把這 n 個點劃分到 K 個集合中($k \leq n$)，使得組內平方和(WCSS within-cluster sum of squares)最小，亦即滿足

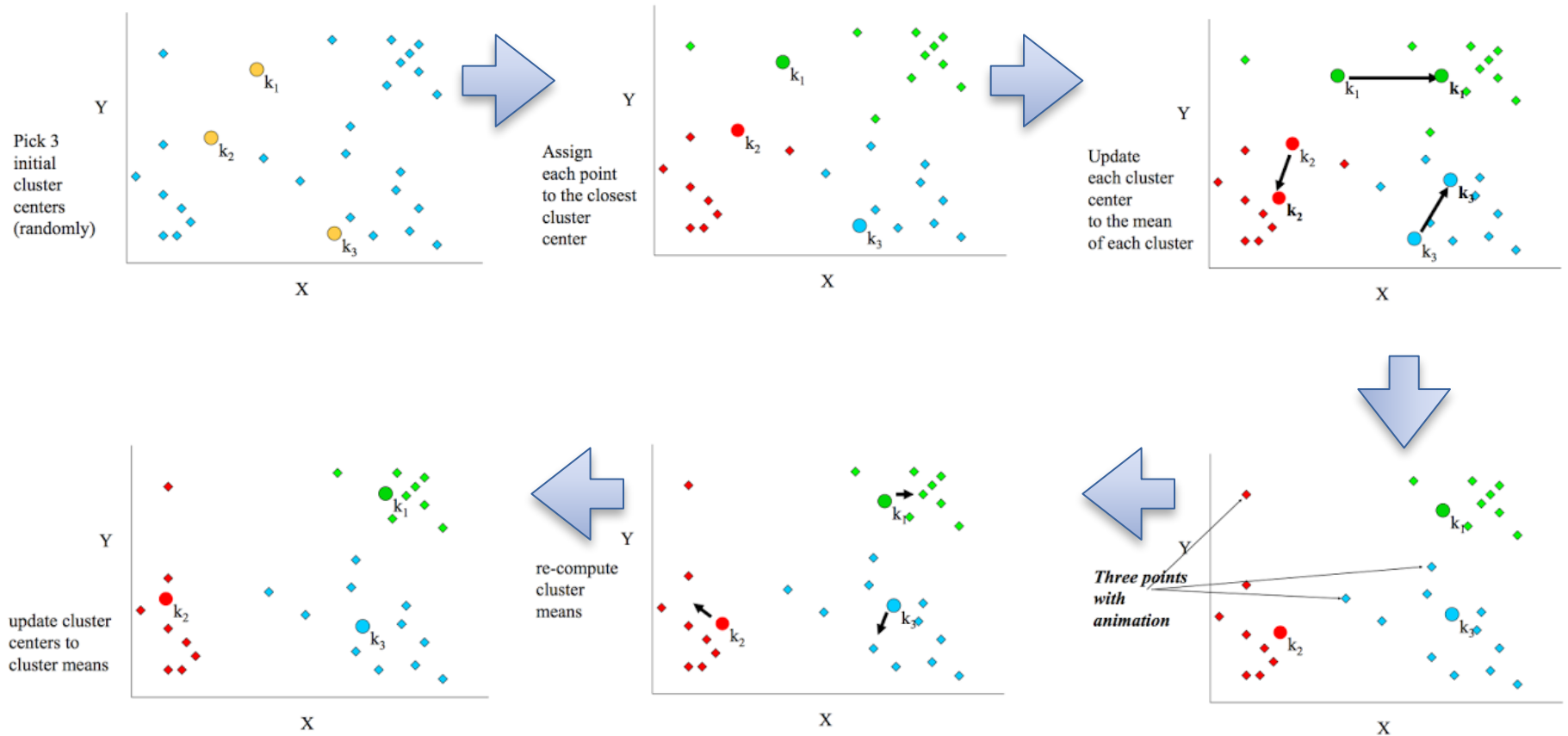
$$\arg \min_S \sum_{i=1}^k \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mu_i\|^2, \text{ 其中 } \mu_i \text{ 是 } S_i \text{ 中所有點的均值。}$$

- ▶ 演算法步驟

- A. 選擇 K 個點作為初始中心
- B. 將每個資料點指派到最近的中心，形成 K 個叢集
- C. 重新計算每個叢集的中心(該叢集資料點的平均)
- D. 重覆B及C步驟，直到中心不再變化



K-Means演算法圖例



ref: <http://mropengate.blogspot.tw/2015/06/ai-ch16-5-k-introduction-to-clustering.html>

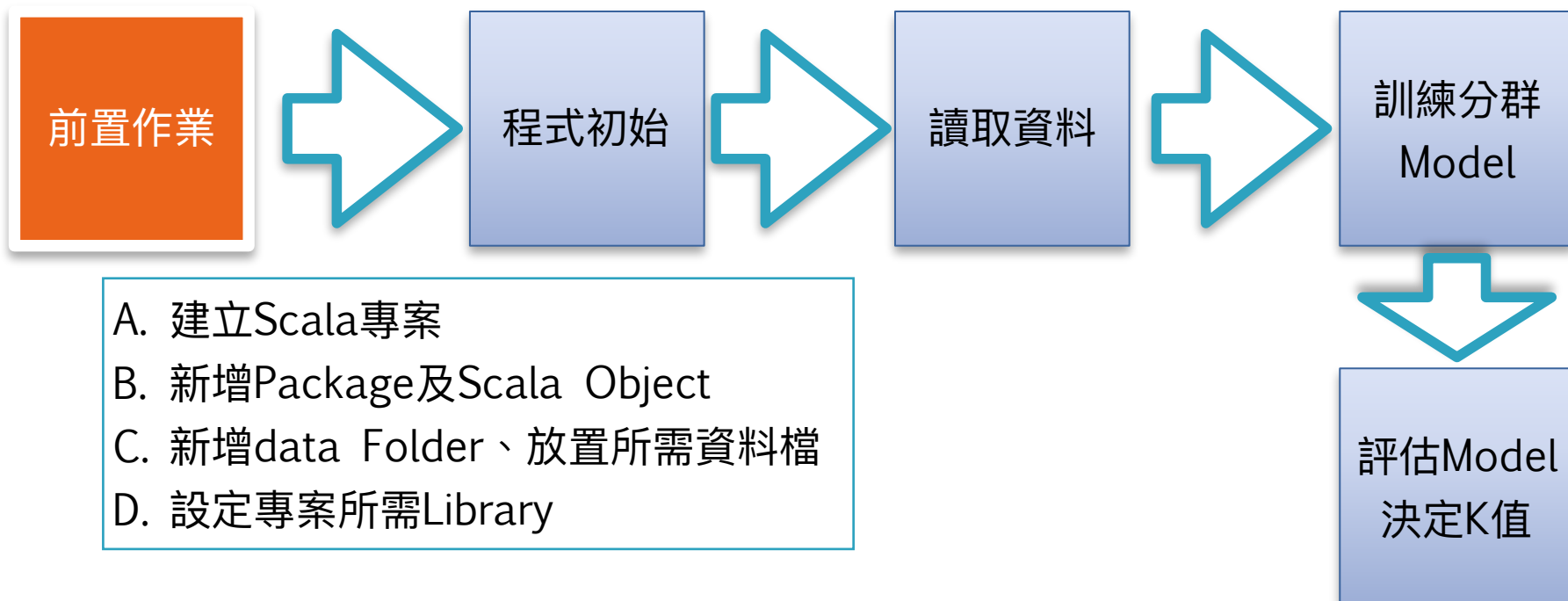
K-Means in Spark MLlib

- ▶ 透過呼叫**KMeans.train**方法來訓練Model(KMeansModel)
 - `val model=KMeans.train(data, numClusters, maxIterations, runs)`
 - `data`：要進行分群的資料(RDD[Vector])
 - `numClusters`：分群的**群數(K)**
 - `maxIterations`：每個run運行的最大Iteration數，當**中心點不再改變或iteration達到maxIterations**即完成model運算
 - `runs`：KMeans**不保證能得到最佳解**，故可指定跑多個run，回傳最佳的model
- ▶ 透過呼叫**model.clusterCenters**取得每個Feature的叢集中心
- ▶ 透過呼叫**model.computeCost**方法取得**WCSS**，以評估model績效

以K-Means分析BikeSharing資料

- ▶ 將hour.csv資料送進KMeans演算法中，分為五群，並將各欄位值之叢集中心列印在console中，觀察各群中心值是否有值得注意之處
 - Features：yr, season, mnth, hr, holiday, weekday, workingday, weathersit, temp, atemp, hum, windspeed, cnt(在此，cnt不是Label，而是Feature之一)
 - numClusters：5 (分5群)
 - maxIterations：20 (每個run至多跑20個iteration)
 - runs：3 (跑3個Run選出最佳之model)

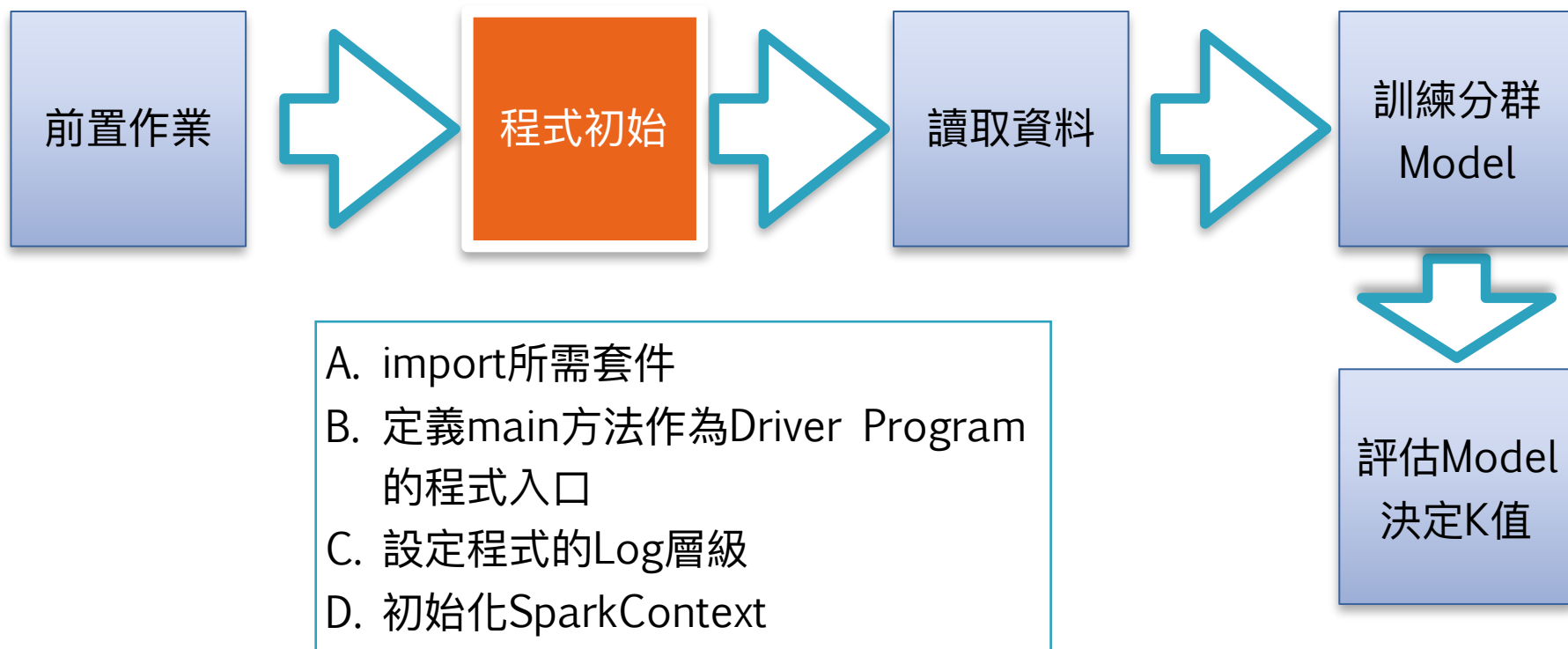
分群實作的主要步驟



前置作業

- ▶ ScalaIDE中的Scala專案、folder、package及Object結構
 - Clustering (專案名稱)
 - src
 - bike (package名稱)
 - BikeShareClustering (scala object名稱)
 - data (folder名稱)
 - hour.csv
- ▶ 設定Build Path
 - 引入 Spark安裝目錄/assembly/target/scala-2.11/jars/
 - 確定專案內的Scala container版本為2.11.8

分群實作的主要步驟



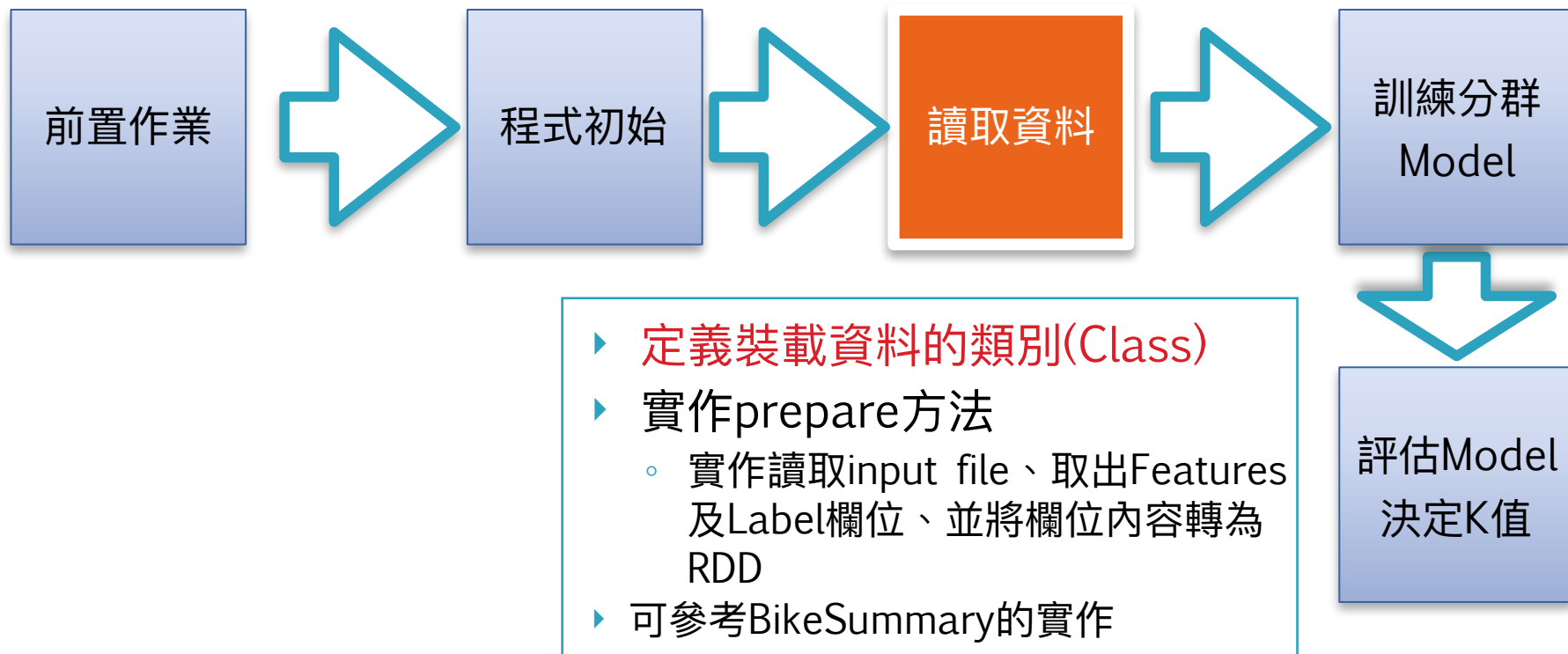
程式初始

```
//import spark rdd library
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
//import KMeans library
import org.apache.spark.mllib.clustering.{ KMeans, KMeansModel }

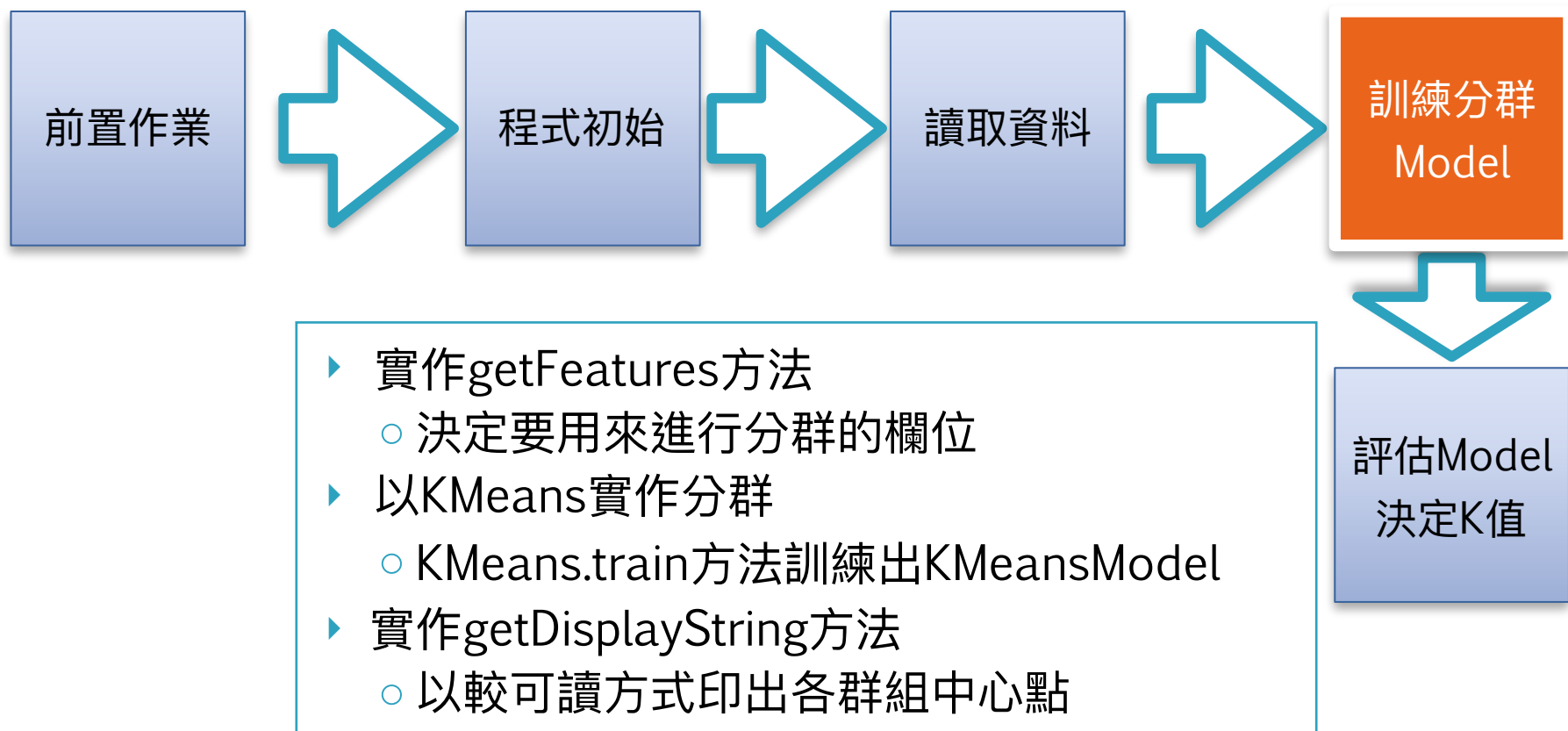
object BikeShareClustering {
    def main(args: Array[String]): Unit = {
        Logger.getLogger(com).setLevel(Level.OFF) //set logger
        //initialize SparkContext
        val sc = new SparkContext(new SparkConf().setAppName("BikeClustering").setMaster("local[*]"))
    }
}
```

- ▶ 引入KMeans相關Library
- ▶ 自己開發Driver Program要自行建立sc
 - appName - 在管理介面識別Driver Program用
 - master - 指定master之URL

分群實作的主要步驟



分群實作的主要步驟



getFeatures及getDisplayString方法實作

getFeatures方法

```
def getFeatures(bikeData: BikeShareEntity): Array[Double] = {  
    val featureArr = Array(bikeData.cnt, bikeData.yr, bikeData.season,  
        bikeData.mnth, bikeData.hr, bikeData.holiday, bikeData.weekday,  
        bikeData.workingday, bikeData.weathersit, bikeData.temp,  
        bikeData.atemp, bikeData.hum, bikeData.windspeed, bikeData.casual,  
        bikeData.registered)  
    featureArr  
}
```

getDisplayString方法

```
def getDisplayString(centers:Array[Double]): String = {  
    val dispStr = ""cnt: %.5f, yr: %.5f, season: %.5f, mnth: %.5f, hr: %.5f,  
        holiday: %.5f, weekday: %.5f, workingday: %.5f, weathersit: %.5f, temp:  
        %.5f, atemp: %.5f, hum: %.5f, windspeed: %.5f, casual: %.5f, registered:  
        %.5f""  
    .format(centers(0), centers(1), centers(2), centers(3), centers(4),  
        centers(5), centers(6), centers(7), centers(8), centers(9), centers(10),  
        centers(11), centers(12), centers(13), centers(14))  
    dispStr  
}
```

以KMeans實作分群

以初始參數執行

```
//取出Features，作成RDD[Vector]
val featureRdd = bikeData.map { x =>
  Vectors.dense(getFeatures(x)) }
val model = KMeans.train(featureRdd, 5, 20, 3) //先預設跑K=5，
20個Iteration，3個Run
```

印出各群組中心點

```
var clusterIdx = 0
model.clusterCenters.sortBy { x => x(0) }.foreach { x => {
  println("center of cluster %d - %s".format(clusterIdx,
    getDisplayString(x.toArray) ))
  clusterIdx += 1
} } //注意在這裡我們用Cnt的叢集中心由小至大作排序輸出，以便觀察結果
```

分群實作主程式架構

```
//K-Means
import org.apache.spark.mllib.clustering.{ KMeans, KMeansModel }

object BikeShareClustering {
  def main(args: Array[String]): Unit = {
    Logger.getLogger(com).setLevel(Level.OFF) //set logger
    //初始化SparkContext
    val sc = new SparkContext(new SparkConf().setAppName(BikeClustering).setMaster(local[*]))

    println("===== preparing data =====")
    val bikeData = prepare(sc) //讀取hour.csv，回傳RDD[BikeShareEntity]
    bikeData.persist()

    println("===== clustering by KMeans =====")
    //取出Features，作成RDD[Vector]
    val featureRdd = bikeData.map { x => Vectors.dense(getFeatures(x)) }
    val model = KMeans.train(featureRdd, 5, 20, 3) //先預設跑K=5，20個Iteration，3個Run

    var clusterIdx = 0
    model.clusterCenters.sortBy { x => x(0) }.foreach { x => {
      println("center of cluster %d - %s".format(clusterIdx, getDisplayString(x.toArray) ))
      clusterIdx += 1
    } } //注意在這裡我們用Cnt的叢集中心由小至大作排序輸出，以便觀察結果

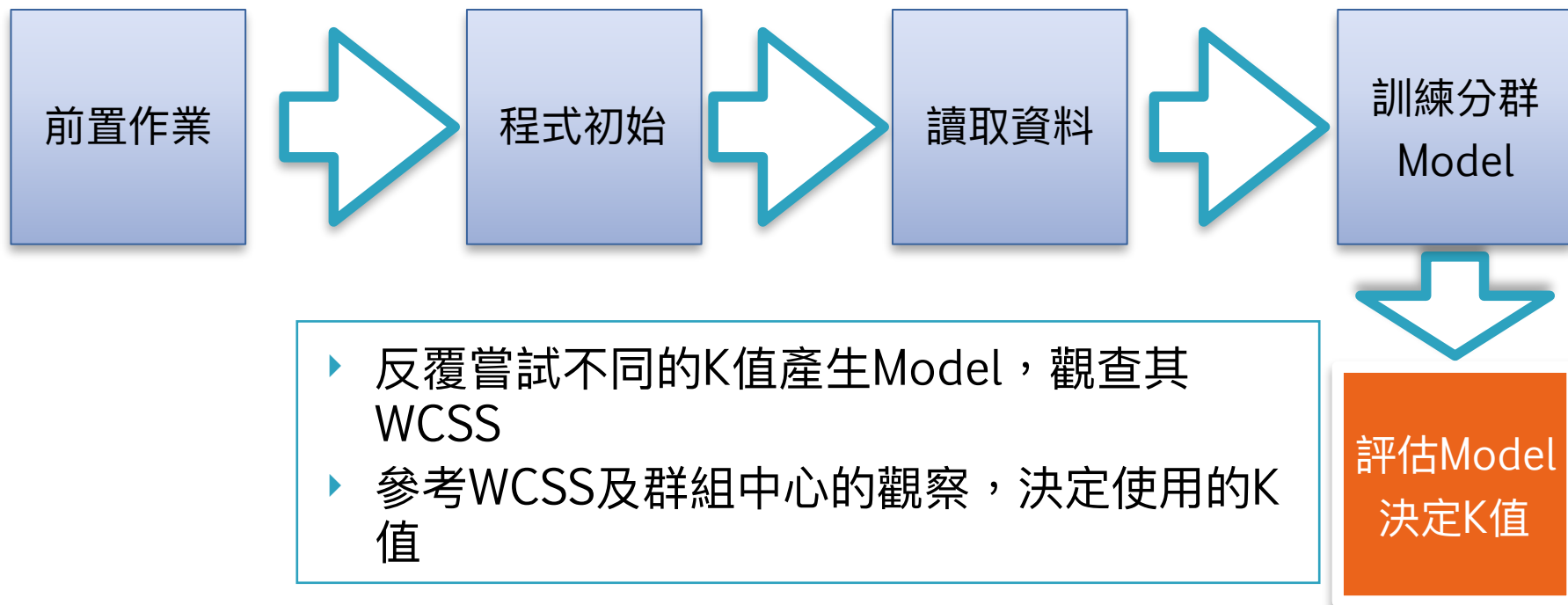
    bikeData.unpersist()
  }
}
```

執行結果觀察

```
===== clustering by KMeans =====  
center of cluster 0 - cnt: 35.39294, yr: 0.41623, season: 2.26411, mnth: 6.01171, hr: 7.45651, holiday: 0.03757,  
  weekday: 2.93829, workingday: 0.64048, weathersit: 1.52250, temp: 0.40921, atemp: 0.39772, hum: 0.70163,  
  windspeed: 0.17507, casual: 5.36452, registered: 30.02842  
center of cluster 1 - cnt: 165.47143, yr: 0.44407, season: 2.54814, mnth: 6.66001, hr: 14.11686, holiday: 0.02117,  
  weekday: 3.03915, workingday: 0.73112, weathersit: 1.41390, temp: 0.51019, atemp: 0.48792, hum: 0.60487,  
  windspeed: 0.19962, casual: 30.07231, registered: 135.39912  
center of cluster 2 - cnt: 312.45857, yr: 0.59479, season: 2.73164, mnth: 7.04583, hr: 14.45355, holiday: 0.02793,  
  weekday: 3.04520, workingday: 0.67420, weathersit: 1.33459, temp: 0.58650, atemp: 0.55527, hum: 0.55932,  
  windspeed: 0.19618, casual: 67.88387, registered: 244.57470  
center of cluster 3 - cnt: 503.23012, yr: 0.71205, season: 2.81145, mnth: 7.18434, hr: 14.58012, holiday: 0.02349,  
  weekday: 3.10361, workingday: 0.63735, weathersit: 1.27169, temp: 0.61653, atemp: 0.58179, hum: 0.53614,  
  windspeed: 0.20986, casual: 107.17349, registered: 396.05663  
center of cluster 4 - cnt: 733.24715, yr: 0.98859, season: 2.84981, mnth: 7.25856, hr: 14.34791, holiday: 0.00570,  
  weekday: 2.96958, workingday: 0.97909, weathersit: 1.27376, temp: 0.62019, atemp: 0.58302, hum: 0.54856,  
  windspeed: 0.20031, casual: 71.73384, registered: 661.51331
```

- ▶ yr、season、mnth、hr大致隨著cnt增加，但不明顯
- ▶ weathersit隨著cnt增加而下降(晴天較多人租借)
- ▶ temp及atemp隨著cnt增加而增加(氣溫高較多人租借)
- ▶ hum隨著cnt增加而下降(溼度高可能是下雨天)
- ▶ 大致和correlation分析結果相近

分群實作的主要步驟



K-Means參數調教

- ▶ 透過呼叫`model.computeCost`方法取得WCSS，以評估model績效 (WCSS越小代表群聚效果越佳)
- ▶ 透過嘗試計算不同的numClusters之WCSS值來決定較佳之群組數(K)
- ▶ 不見得需要使用WCSS最小者，主要考慮程式效能及分群結果是否足以讓我們了解資料內部結構

```
println("===== tuning parameters =====")
for (k <- Array(5,10,15,20, 25)) {
  //嘗試計算不同的numClusters之WCSS
  val iterations = 20
  val tm = KMeans.train(featureRdd, k, iterations,3)
  println("k=%d, WCSS=%f".format(k, tm.computeCost(featureRdd)))
}
```

```
===== tuning parameters =====
k=5, WCSS=89540755.504054
k=10, WCSS=36566061.126232
k=15, WCSS=23705349.962375
k=20, WCSS=18134353.720998
k=25, WCSS=14282108.404025
```

練習：K-Means實作

A. 基礎實作練習

- 下載[BikeShareClustering.scala](#)，放入Scala專案中
- 下載[hour.csv](#)放入data目錄中
- 完成BikeShareClustering的實作(實作TODO部份)

B. 進階實作練習 — 加入新的feature，觀察分群效果

- 在BikeClustering專案中實作以下邏輯
 - 將yrmo加入getFeatures的回傳值中，執行KMeans，將叢集中心輸出在console中，觀察yrmo的趨勢
 - 調整numClusters的值(ex:10,20,50,75,100)，選出理想值並觀察各分群的特性

注意：

- ▶ K-Means演算法**不保證可得到最佳分群結果**
- ▶ KMeans演算時因初始點為**隨機選擇**，故每次**執行結果不會完全相同**

練習II：輸出結果，製作圖表

- 由文字輸出不易看出趨勢，可嘗試將叢集中心點輸出存成csv檔後，透過Excel製作圖表觀察

```
val title=
Array("cnt,yr,season,mnth,hr,holiday,weekday,workingday,weathersit,temp,atemp,hum,windspeed,casual,registered,yrmo")//欄位名稱
val centerList = model.clusterCenters.sortBy( x => x(0) ).map { x => x.toArray.mkString(",") }
//用逗號分隔的csv檔
val centerRdd = sc.parallelize(title ++ centerList)//轉成RDD
centerRdd.saveAsTextFile("data/output")//輸出至textFile
```

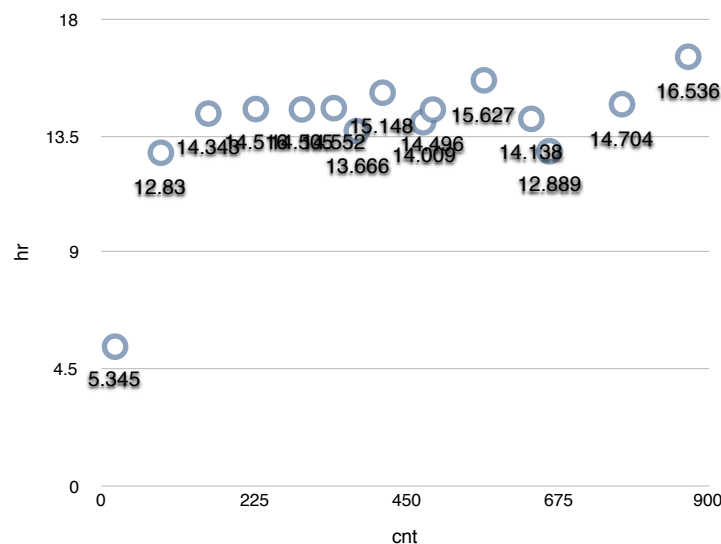
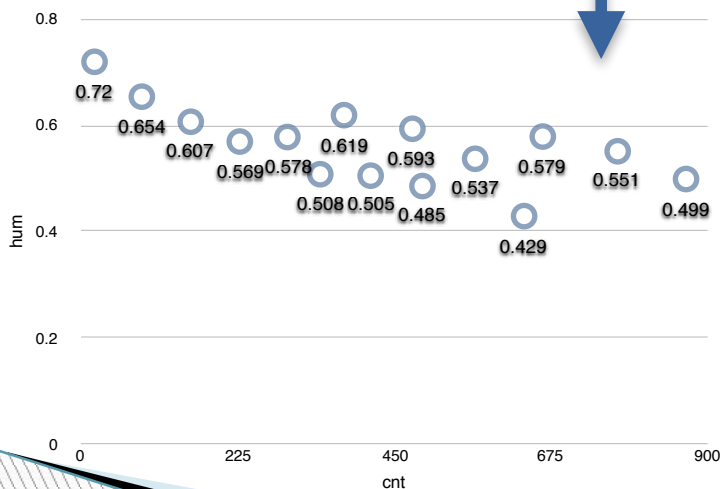
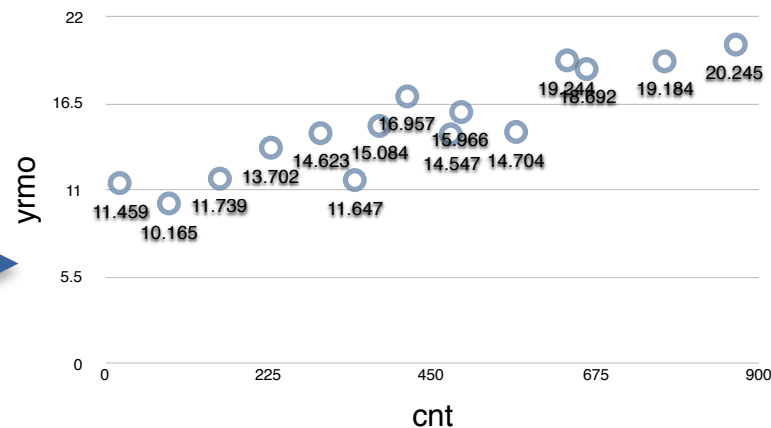
- merge output的結果到一個檔案：

- linux: cat data/output/part-* > res.csv
- windows: type data\output\part-* > res.csv

cnt	yr	season	mnth	hr
19.241806554756195	0.4370503597122302	2.326538768984812	6.214428457234212	5.344724220623501
87.20074855392991	0.36849268458659407	2.1932630146308267	5.7427696495406595	12.830214358625382
157.47584632940283	0.4214530239634842	2.556865728413846	6.681247622670217	14.343096234309623
227.88662239089186	0.5659392789373814	2.646584440227704	6.910815939278938	14.516129032258064
296.2088709677419	0.6314516129032258	2.7225806451612904	7.045967741935484	14.504838709677419
343.45288753799394	0.39361702127659576	2.7264437689969605	6.924012158054711	14.551671732522797
377.5379426644182	0.6290050590219224	2.9005059021922426	7.536256323777403	13.66610455311973
415.8289855072464	0.8	2.889855072463768	7.356521739130435	15.147826086956522
475.84401709401715	0.6132478632478633	2.807692307692308	7.188034188034189	14.00854700854701
490.2633053221289	0.7759103641456583	2.661064425770308	6.65546218487395	14.495798319327731

練習II：輸出結果，製作圖表

cnt	yr	season	mnth
19.241806554756195	0.4370503597122302	2.326538768984812	6.214428457234212
87.20074855392991	0.36849268458659407	2.1932630146308267	5.7427696495406595
157.47584632940283	0.4214530239634842	2.556865728413846	6.681247622670217
227.88662239089186	0.5659392789373814	2.646584440227704	6.910815939278938
296.2088709677419	0.6314516129032258	2.7225806451612904	7.045967741935484
343.45288753799394	0.39361702127659576	2.7264437689969605	6.924012158054711
377.5379426644182	0.6290050590219224	2.9005059021922426	7.536256323777403
415.8289855072464	0.8	2.889855072463768	7.356521739130435
475.84401709401715	0.6132478632478633	2.807692307692308	7.188034188034189
490.2633053221289	0.7759103641456583	2.661064425770308	6.65546218487395



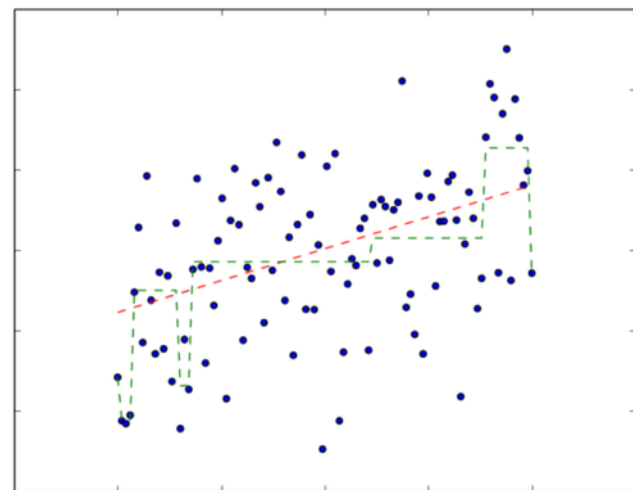
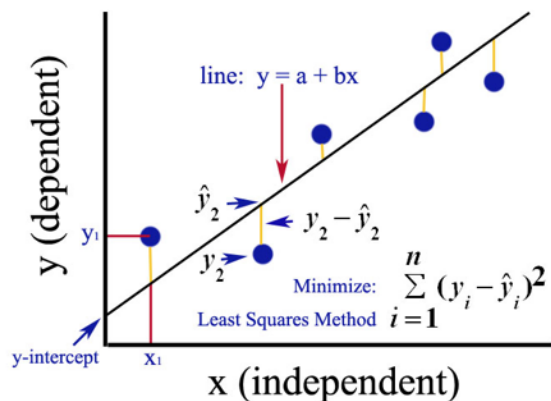
Outline

- ▶ **Spark MLlib 實戰**
 - 概述統計量(summary statistics)
 - Clustering
 - **Regression**



迴歸演算法介紹

- ▶ 是監督式演算法的一種，不同於分類演算預測類別，迴歸演算法用來預測數值
- ▶ 常用於股價、房價、租借量等預測分析
- ▶ 常見演算法
 - 線性最小平方法(Least Squares)、Lasso、脊迴歸(ridge regression)、決策樹



Decision Tree Regression in Spark

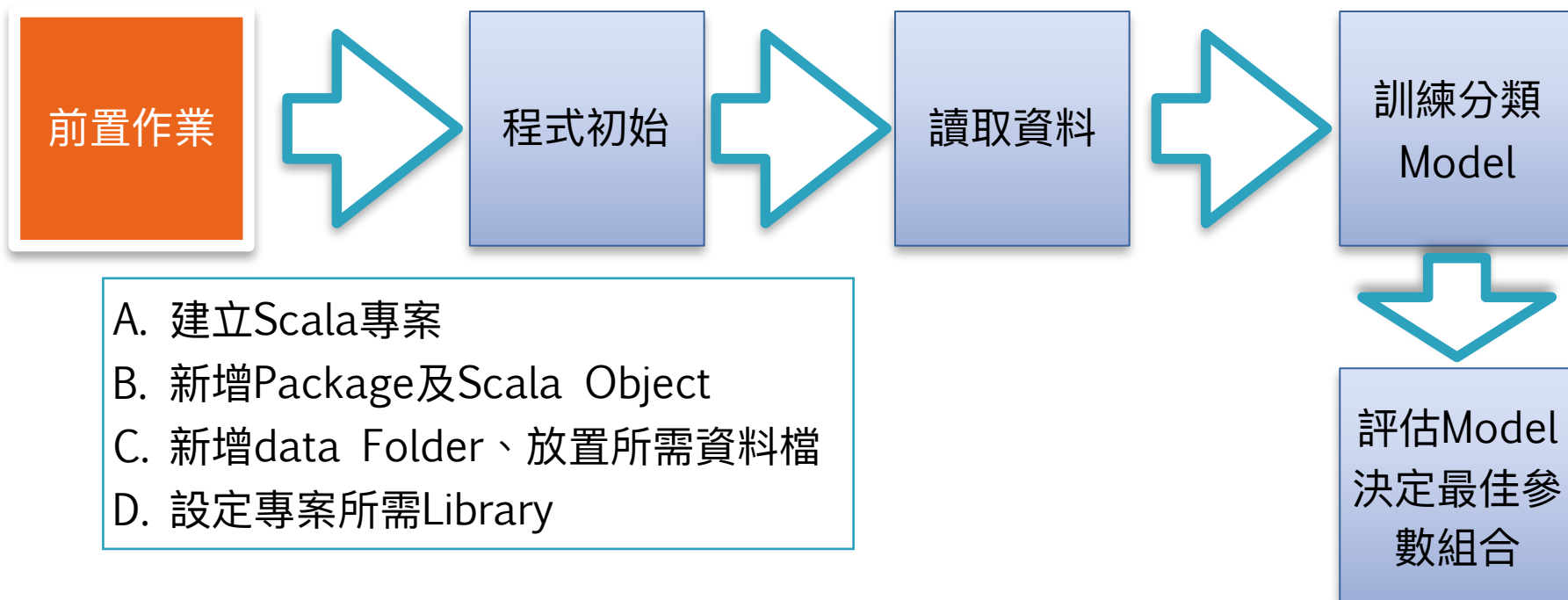
- ▶ `import org.apache.spark.mllib.tree.DecisionTree`
- ▶ `import org.apache.spark.mllib.tree.model.DecisionTreeModel`
- ▶ 透過呼叫`DecisionTree.trainRegressor`方法來訓練Model(`DecisionTreeModel`)
 - `val model=DecisionTree.trainRegressor(trainData, categoricalFeaturesInfo, impurity, maxDepth, maxBins)`
 - `trainData`：要用來訓練模型的資料，型別為`RDD[LabeledPoint]`
 - `categoricalFeaturesInfo`：指出`trainData`裡那些欄位是`categorical`，以`Map[欄位Index,類別數]`呈現，不指定則欄位會被視為`continuous`
 - 例：`Map(0->2,4->10)`代表欄位1,5為`categorical`，類別數分別為2,10
 - `impurity`：決策樹分裂節點時採用的判斷方式(固定為`variance`)
 - `maxDepth`：決策樹最大深度，深度越大某些情況能提高模型精準度，但會影響演算法效能及越可能造成`overfit`的狀況
 - `maxBins`：決策樹每個節點最大分支數，分支數越大能提高模型精準度，但會影響演算法效能
 - 若有指定`categoricalFeaturesInfo`，則`maxBins`至少要等於`categoricalFeaturesInfo`中最大的類別數

實作任務－來完成終極目標吧！

- ▶ 試著建立Model預測在某些外在因素下每小時的租借量會是多少
 - Features : yr, season, mnth, hr, holiday, weekday, workingday, weathersit, temp, atemp, hum, windspeed
 - Label : cnt欄位
 - impurity : gini
 - maxDepth : 5
 - maxBins : 30



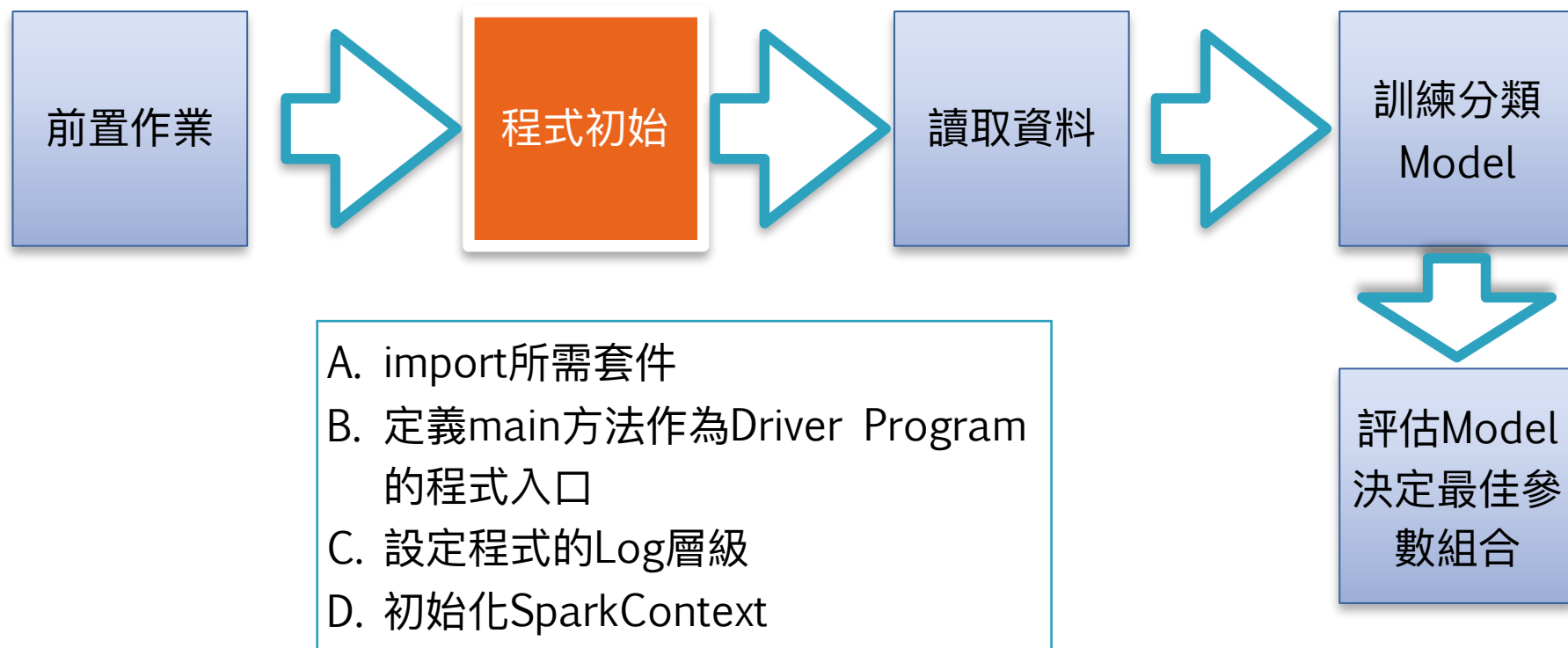
迴歸實作的主要步驟



前置作業

- ▶ ScalaIDE中的Scala專案、folder、package及Object結構
 - Regression (專案名稱)
 - src
 - bike (package名稱)
 - BikeShareRegressionDT (scala object名稱)
 - data (folder名稱)
 - hour.csv
- ▶ 設定Build Path
 - 引入 Spark安裝目錄/assembly/target/scala-2.11/jars/
 - 確定專案內的Scala container版本為2.11.8

迴歸實作的主要步驟



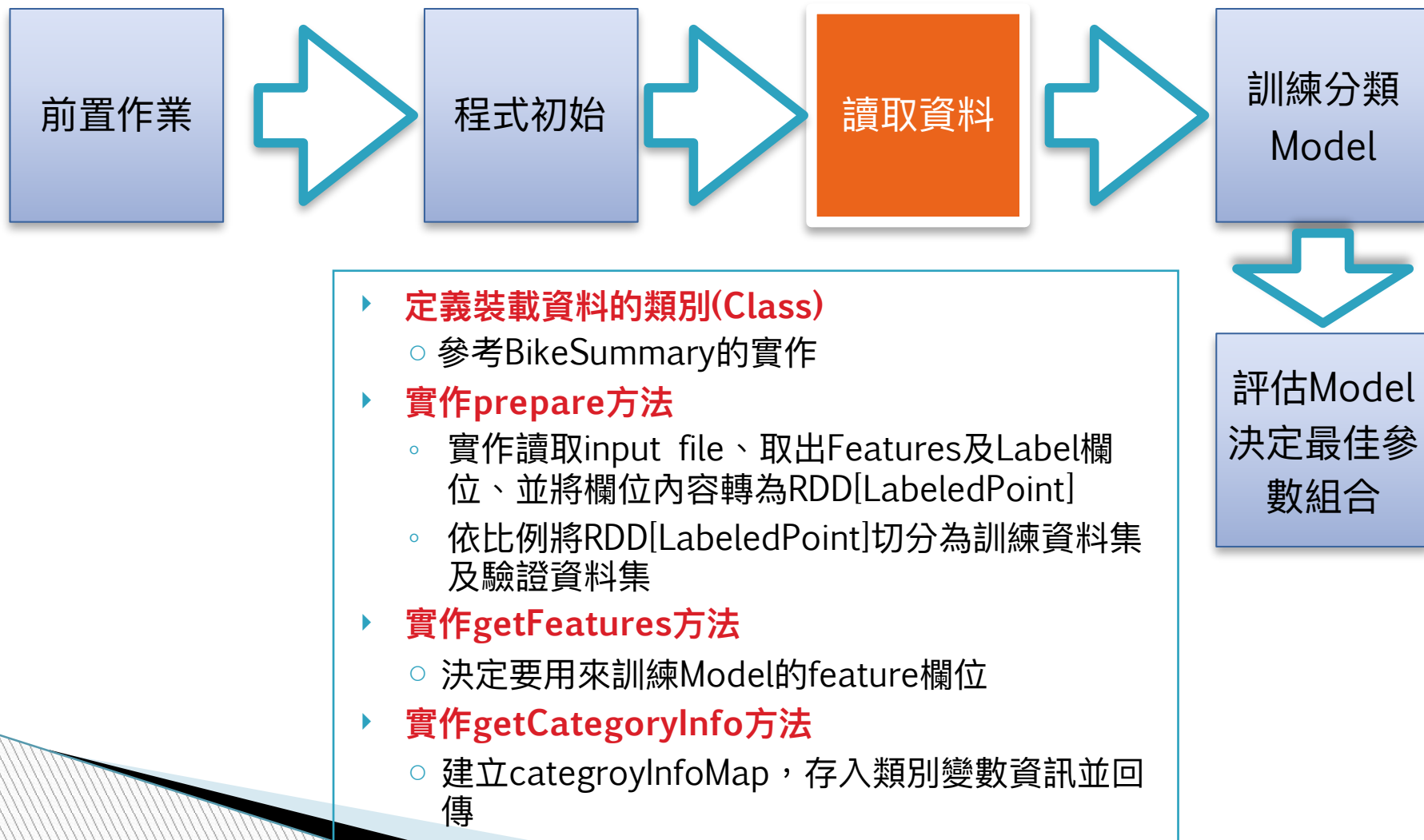
程式初始

```
//import spark rdd library
import org.apache.spark._
import org.apache.spark.SparkContext._
import org.apache.spark.rdd._
//import decision tree library
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel

object BikeShareRegressionDT {
  def main(args: Array[String]): Unit = {
    Logger.getLogger(com).setLevel(Level.OFF) //set logger
    //initialize SparkContext
    val sc = new SparkContext(new SparkConf().setAppName("BikeRegressionDT").setMaster("local[*]"))
  }
}
```

- ▶ 引入Decision Tree相關Library
- ▶ 自己開發Driver Program要自行建立sc
 - appName - 在管理介面識別Driver Program用
 - master - 指定master之URL

迴歸實作的主要步驟



prepare方法實作

```
def prepare(sc: SparkContext): (RDD[LabeledPoint], RDD[LabeledPoint])= {  
    val rawData=sc.textFile(data/hour.csv) //read hour.csv in data folder  
    val rawDataNoHead=rawData.mapPartitionsWithIndex { (idx, iter) =>  
        { if (idx == 0) iter.drop(1) else iter } } //ignore first row(column name)  
    val lines:RDD[Array[String]] = rawDataNoHead.map { x =>  
        x.split(, ).map { x => x.trim() } } //split columns with comma  
    val bikeData = lines.map{ x => BikeShareEntity(...) }//RDD[BikeShareEntity]  
    val lpData=bikeData.map { x => {  
        val label = x.cnt //預測目標為租借量欄位  
        val features = Vectors.dense(getFeatures(x))  
        new LabeledPoint(label, features ) //LabeledPoint由label及Vector組成  
    }  
    //以6:4的比例隨機分割，將資料切分為訓練及驗證用資料  
    val Array(trainData, validateData) = lpData.randomSplit(Array(0.6, 0.4))  
    (trainData, validateData)  
}
```

getFeatures及getCategoryInfo方法

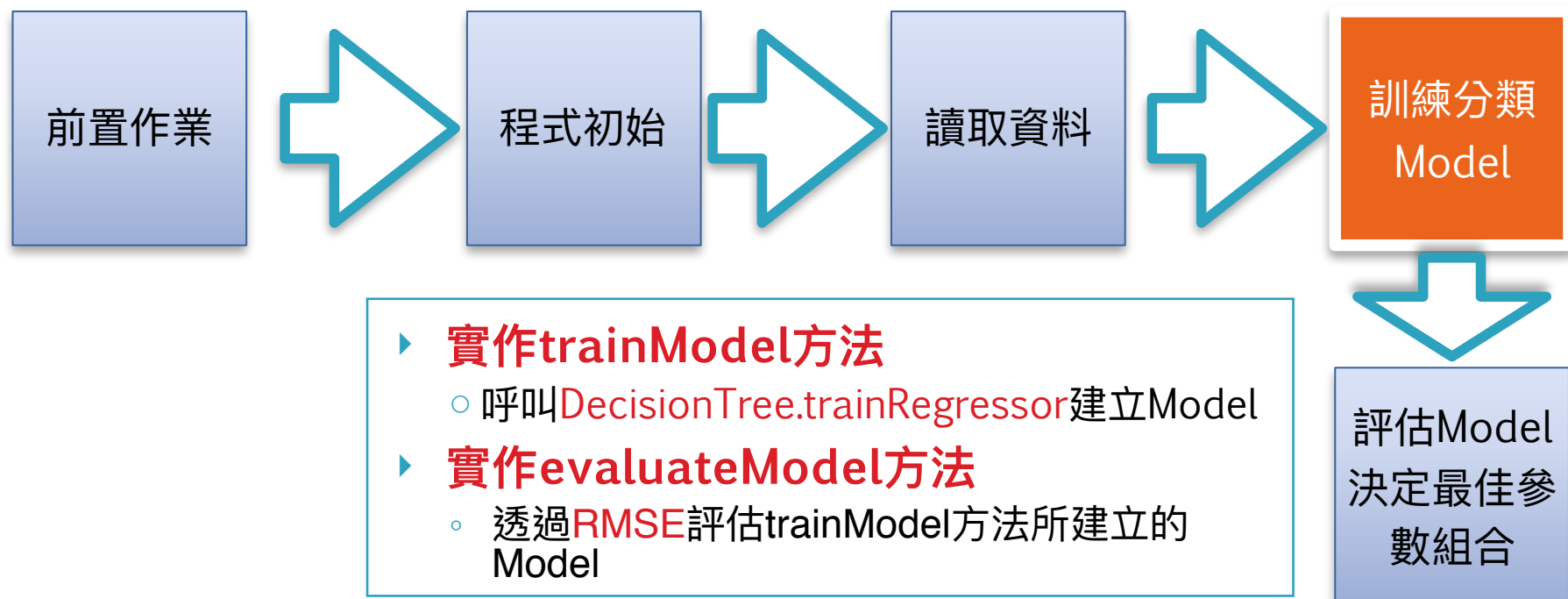
getFeatures方法

```
def getFeatures(bikeData: BikeShareEntity): Array[Double] = {  
    val featureArr = Array(bikeData.yr, bikeData.season - 1, bikeData.mnth - 1,  
        bikeData.hr, bikeData.holiday, bikeData.weekday, bikeData.workingday,  
        bikeData.weathersit - 1, bikeData.temp, bikeData.atemp, bikeData.hum,  
        bikeData.windspeed)  
    featureArr  
} //思考一下，為何season等Feature值要減1 ？ ？ ？
```

getCategoryInfo方法

```
def getCategoryInfo(): Map[Int, Int] = {  
    val categoryInfoMap = Map[Int, Int](  
        (/*"yr"*/ 0, 2), (/*"season"*/ 1, 4), (/*"mnth"*/ 2, 12),  
        (/*"hr"*/ 3, 24), (/*"holiday"*/ 4, 2), (/*"weekday"*/ 5, 7),  
        (/*"workingday"*/ 6, 2), (/*"weathersit"*/ 7, 4))  
    categoryInfoMap  
} //(欄位在featureArr中的index, 欄位值的distinct數)
```

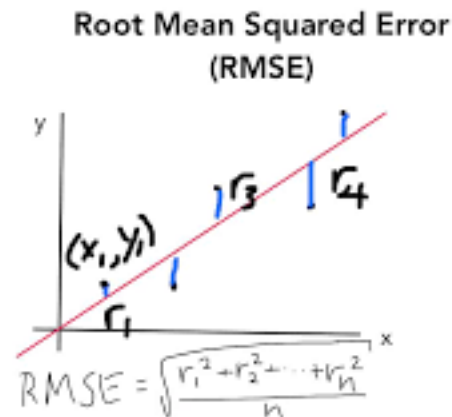
迴歸實作的主要步驟



RMSE(root-mean-square error)

- ▶ 方均根偏移(root-mean-square deviation)或方均根差(root-mean-square error)是一種常用的測量數值之間差異的**量度**
- ▶ 方均根偏移代表**預測值**和**實際值**之差的**樣本標準差**(sample standard deviation)
- ▶ 方均根偏移因其與**數值範圍**有關，因此只能用來比較不同模型間**某個特定變數**的預測誤差

$$\text{RMSE} = \sqrt{\frac{\sum_{t=1}^n (\hat{y}_t - y_t)^2}{n}}$$

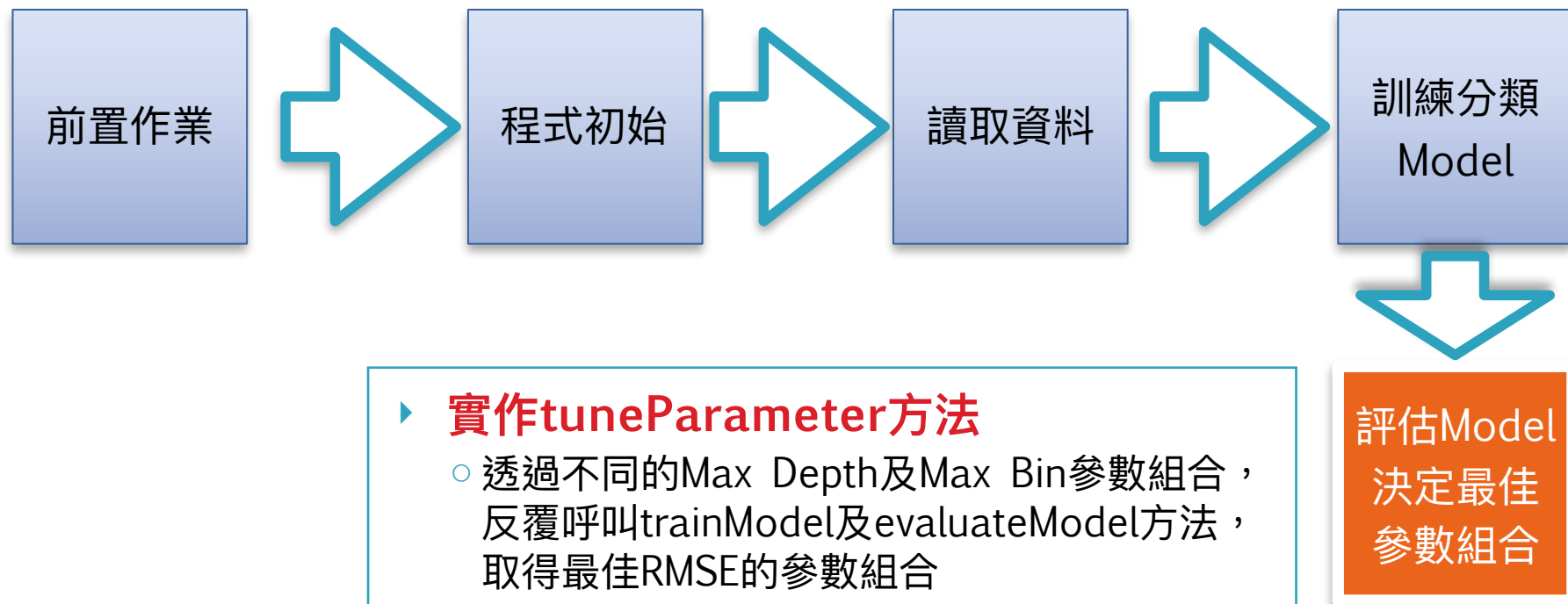


trainModel及evaluateModel方法

```
def trainModel(trainData: RDD[LabeledPoint],
    impurity: String, maxDepth: Int, maxBins: Int, cateInfo: Map[Int,Int]):
(DecisionTreeModel, Double) = {
    val startTime = new DateTime() //記錄方法起始時間
    val model = DecisionTree.trainRegressor(trainData, cateInfo, impurity, maxDepth,
maxBins) //建立Model
    val endTime = new DateTime() //記錄方法結束時間
    val duration = new Duration(startTime, endTime) //計算模型建立花費時間
    //MyLogger.debug(model.toDebugString) //列印Decision Tree節點及判斷式
    (model, duration.getMillis)
}

def evaluateModel(validateData: RDD[LabeledPoint], model: DecisionTreeModel): Double =
{
    val scoreAndLabels = validateData.map { data =>
        var predict = model.predict(data.features)
        (predict, data.label) //建立RDD[(模型預測值, 實際值)]以計算RMSE
    }
    val metrics = new RegressionMetrics(scoreAndLabels)
    val rmse = metrics.rootMeanSquaredError //呼叫 rootMeanSquaredError以取得rmse值
    rmse
}
```

迴歸實作的主要步驟



tuneParameter方法實作

```
def tuneParameter(trainData: RDD[LabeledPoint], validateData: RDD[LabeledPoint])
= {
  val impurityArr = Array(variance)
  val depthArr = Array(3, 5, 10, 15, 20, 25)
  val binsArr = Array(50, 100, 200)
  val evalArr =
    for (impurity <- impurityArr; maxDepth <- depthArr; maxBins <- binsArr)
      yield { //反覆以不同參數訓練model，取得最佳的RMSE參數組合
        val (model, duration) = trainModel(trainData, impurity, maxDepth,
          maxBins, cateInfo)
        val rmse = evaluateModel(validateData, model)
        println("parameter: impurity=%s, maxDepth=%d, maxBins=%d, auc=%f"
          .format(impurity, maxDepth, maxBins, rmse))
        (impurity, maxDepth, maxBins, rmse)
      }
  val bestEvalAsc = (evalArr.sortBy(_._4))
  val bestEval = bestEvalAsc(0) //RMSE由小到大排序，取得最佳參數組合
  println("best parameter: impurity=%s, maxDepth=%d, maxBins=%d, rmse=%f"
    .format(bestEval._1, bestEval._2, bestEval._3, bestEval._4))
}
```

練習：Decision Tree實作

A. 基礎實作練習

- 下載[BikeShareRegressionDT.scala](#)，放入Scala專案中
- 下載[hour.csv](#)放入data目錄中
- 完成BikeShareRegressionDT.scala的實作(實作TODO部份)

B. 進階實作練習 — 加入新的feature，觀察迴歸效果

- 在BikeShareRegressionDT中實作以下邏輯
 - 建立新feature – dayType(Double，類別型變數)，dayType值邏輯如下
 - 若holiday=0且workingday=0，則dataType=0
 - 若holiday=1，則dataType=1
 - 若holiday=0且workingday=1，則dataType=2
 - 將dayType納入feature中，訓練Model(提示：修改getFeatures及getCategoryInfo)
- 嘗試不要指定Categorical Info，看模型預測能力是否受影響

```
===== tuning parameters(CateInfo) =====  
parameter: impurity=variance, maxDepth=3, maxBins=50, rmse=118.424606  
parameter: impurity=variance, maxDepth=3, maxBins=100, rmse=118.424606  
parameter: impurity=variance, maxDepth=3, maxBins=200, rmse=118.424606  
parameter: impurity=variance, maxDepth=5, maxBins=50, rmse=93.138794  
parameter: impurity=variance, maxDepth=5, maxBins=100, rmse=93.138794  
parameter: impurity=variance, maxDepth=5, maxBins=200, rmse=93.138794
```