# 分布式计算代码整理

lec2-text-file

**今天的主题**

并行计算基本概念

并行计算与分布式计算

Apache Spark 简介与安装

**1) 并行计算**

a. **概念**

a) 一种计算模型

b) 将总任务分成若干子任务

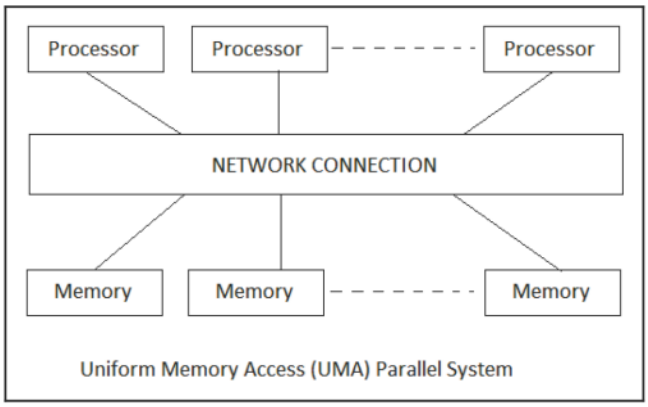
c) 每个子任务同时执行

d) 主要目的是提升计算效率

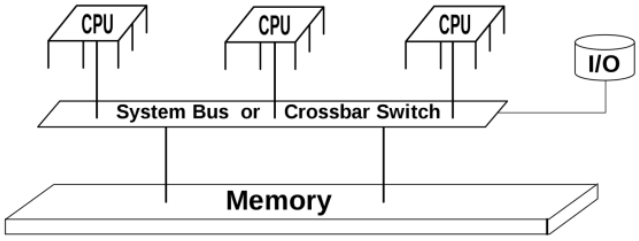
b. **架构**

a) 一类重要的用来实现并行计算的架构叫做统一内存访问架构

b) (Uniform Memory Access, UMA)




c) 例如我们日常使用的多核个人电脑或多 CPU 服务器




d) 狭义的并行计算通常指代这类由共享内存机制实现的计算

c. **Amdahl 定律**

a) 用于预测并行系统的加速比

$$S_{latency}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

$S$: 整体提升倍数

$s$: 可并行部分的加速比

$p$: 可被并行部分所占时间的比例

b) 一个简单例子

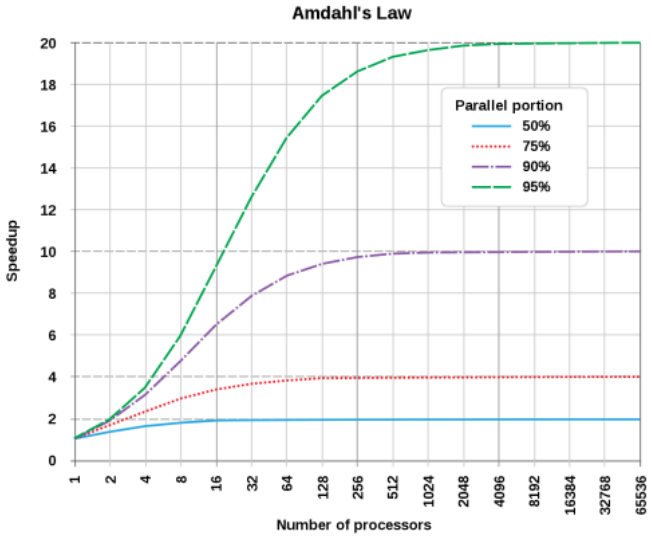
$p_1 = 0.11, p_2 = 0.18, p_3 = 0.23, p_4 = 0.48$

$s_1 = 1, s_2 = 5, s_3 = 20, s_4 = 1.6$

$$S_{latency} = \frac{1}{\frac{p1}{s1}+\frac{p2}{s2}+\frac{p3}{s3}+\frac{p4}{s4}}$$

$$= \frac{1}{\frac{0.11}{1}+\frac{0.18}{5}+\frac{0.23}{20}+\frac{0.48}{1.6}} = 2.19$$

c) 不能被并行的部分决定了并行效果的上限




**2) 分布式计算**

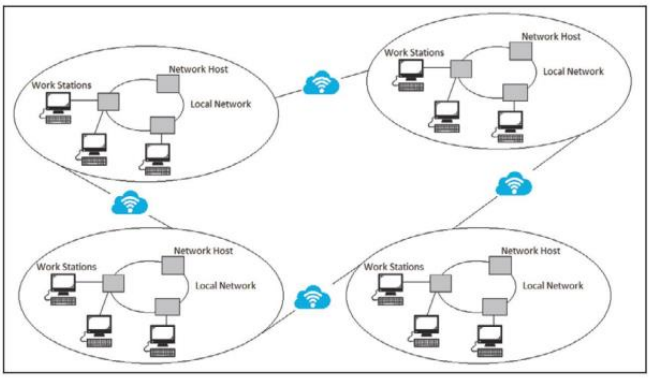
a. **概念**

a) 利用多台通过网络连接在一起的计算机共同完成某项计算任务

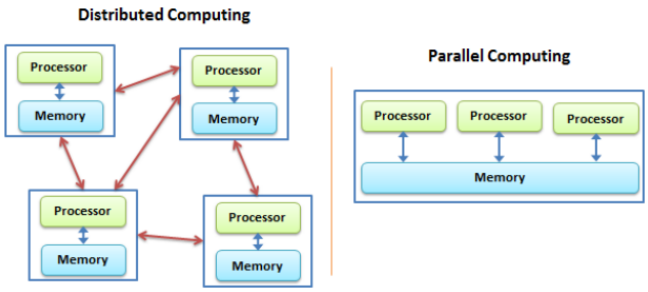
b) 与并行计算有许多相通之处

c) 但分布式系统中每台机器有独立的内存

d) 除了提升计算效率之外，另一重要目的是扩展计算的规模




b. **对比**




a) 从扩展性的角度来看

并行计算：受单机性能制约，如内存大小

分布式计算：理论上可以无限进行扩展

c. 实现

分布式计算的实现要考虑以下两个问题

a) 数据如何存储?

b) 计算任务如何执行?

**3) Hadoop**

a. 简介

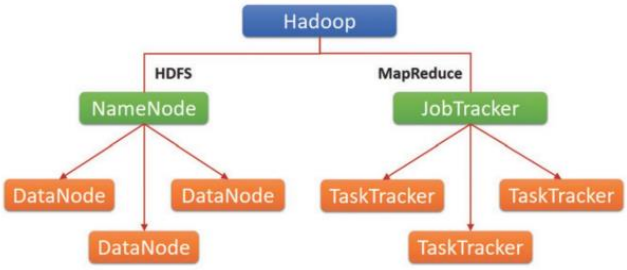
a) Hadoop 的官方名称是 Apachae Hadoop

b) 是由 Apache 软件基金会开发和维护的一套开源分布式计算框架

c) 可以扩展到上千台机器组成的集群

b. Hadoop

a) Hadoop ≈ HDFS + MadReduce

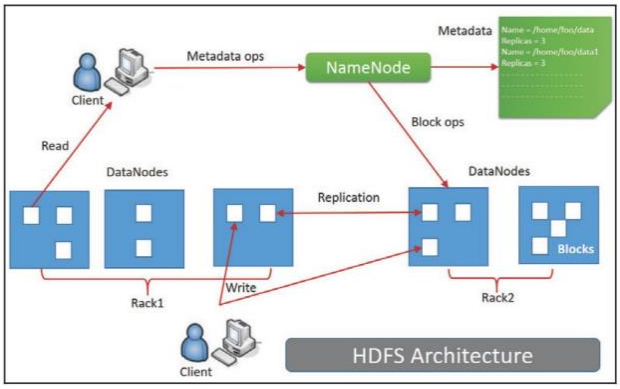



c. HDFS

a) HDFS 是 Hadoop 提供的分布式文件系统

b) 解决了大规模数据的存储问题

c) 实现高容错、低成本的数据存储方案



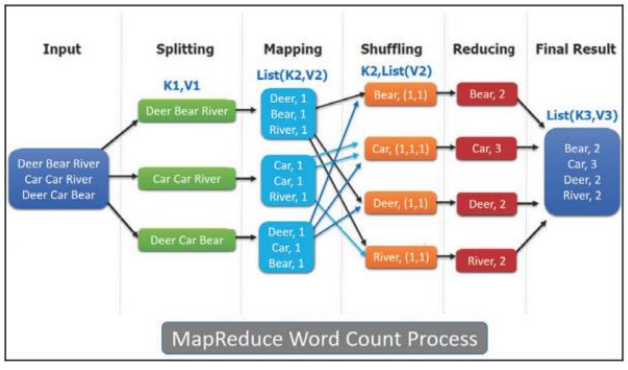

d. MapReduce

a) MapReduce 是 Hadoop 采用的计算框架

b) 把数据处理过程分成两个阶段：Map 和 Reduce

c) Map 用来对输入数据进行变换

d) Recude 用来汇总结果




e. 不足

a) Hadoop 对于分布式计算有着重大意义

b) 但也存在一些不足

c) 表达能力较弱（一些操作难以仅用 map 和 reduce 完成）

d) 不容易实现迭代算法

e) 基于磁盘进行数据传递，效率较低

f) 交互性不强

**4) Spark**

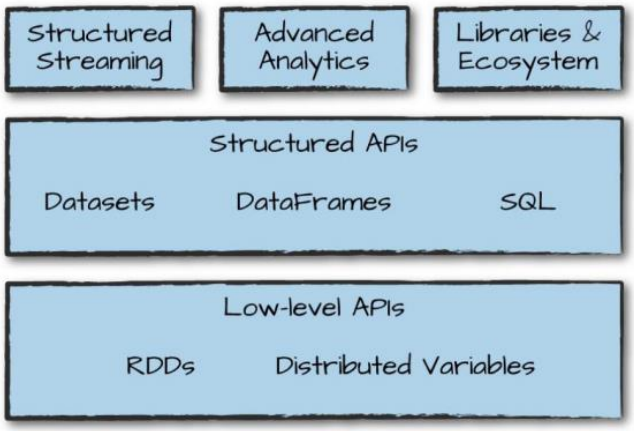
a. 简介

a) 与 Hadoop 类似，Spark 的“全名”叫 Apache Spark

b) 是由 Apache 软件基金会开发和维护的一套开源大数据分析平台

c) 针对 Hadoop 的缺陷进行了多项改进

b. 工具集




c. 编程语言

a) Spark 本身利用 Scala 编程语言编写

b) 提供了多种语言的接口（Python、R、Java 等）

c) Scala 版本功能最全

d) Python 和 R 在数据科学中更常见

e) 后面的课程中将主要使用 Python 接口，即 PySpark

**1、配置和启动 PySpark:**

```
import findspark
findspark.init()
```

spark 是用 scala 编写的，scala 底层用的 java 8 及以上，使用 python 编写 spark 程序，需要用到 pyspark 第三方包去转为 jvm 中调用核心，而 findspark 可以提供简便的初始化 spark 环境，后续直接使用 pyspark 即可

```
from pyspark.sql import SparkSession
# 本地模式
spark = SparkSession.builder.master("local[*]).
        appName("Reading Text").getOrCreate()
```

Apache Spark 2.0 引入了 SparkSession，其为用户提供了一个统一的切入点来使用 Spark 的各项功能，并且允许用户通过它调用 DataFrame 和 Dataset 相关 API 来编写 Spark 程序。

- SparkSession 主要用在 sparkSQL 中，当然也可以用在其他场合，他可以代替 SparkContext;
- SparkSession 实际上封装了 SparkContext，另外也

封装了 SparkConf、sqlContext，随着版本增加，可能更多

- 我们尽量使用 SparkSession ，如果发现有些 API 不在 SparkSession 中，也可以通过 SparkSession 拿到 SparkContext 和其他 Context 等

| Name | Type | Description |
| --- | --- | --- |
| sparkContext | SparkContext | spark功能的主要入口点。可以通过 sparkConext在集群上创建RDD, accumulators 和 broadcast variables |
| existingSharedState | Option[SharedState] | 一个内部类负责保存不同session的状态状态 |
| parentSessionState | Option[SessionState] | 复制父session的状态 |



Builder 包含一些方法:

- **appName(String name)**

用来设置应用程序名字，会显示在 Spark web UI 中，自己起名就行

- **config 函数**

重载函数，是针对不同的情况，使用不同的函数，但是他们的功能都是用来设置配置项的。

config(SparkConf conf)
根据给定的 SparkConf 设置配置选项列表。
config(String key, boolean value)
设置配置项，针对值为 boolean 的
config(String key, double value)
设置配置项，针对值为 double 的
config(String key, long value)
设置配置项，针对值为 long 的
config(String key, String value)
设置配置项，针对值为 String 的

- **enableHiveSupport 函数**

表示支持 Hive，包括 链接持久化 Hive metastore，支持 Hive serdes，和 Hive 用户自定义函数

- **getOrCreate()**

获取已经得到的 SparkSession，或则如果不存在则创建一个新的基于 builder 选项的 SparkSession

- **master(String master)**

设置 Spark master URL 连接，比如"local" 设置本地运行 ， "local[3]" 本地运行 3 cores ， 或则 "spark://master:7007"运行在 spark standalone 集群。

- withExtensions(scala.Function1<SparkSessionExtensions,scala.runtime.BoxedUnit> f)

这允许用户添加 Analyzer rules, Optimizer rules, Planning Strategies 或则 customized parser.这一函不常见的。

```
sc = spark.sparkContext
```

SparkContext 相关组件

- **SparkConf**

Spark 配置类，配置已键值对形式存储，封装了一个 ConcurrentHashMap 类实例 settings 用于存储 Spark 的配置信息。

- **SparkEnv**

SparkContext 中非常重要的类，它维护着 Spark 的执行环境，所有的线程都可以通过 SparkContext 访问到同一个 SparkEnv 对象。

- **LiveListenerBus**

SparkContext 中的事件总线，可以接收各种使用方的事件，并且异步传递 Spark 事件监听与 SparkListeners 监听器的注册。

- **SparkUI**

为 Spark 监控 Web 平台提供了 Spark 环境、任务的整个生命周期的监控。

- TaskScheduler

为 Spark 的任务调度器，Spark 通过他提交任务并且请求集群调度任务。因其调度的 Task 由 DAGScheduler 创建，所以 DAGScheduler 是 TaskScheduler 的前置调度。

- DAGScheduler

为高级的、基于 Stage 的调度器， 负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage，并将 Stage 作为 Tasksets 提交给底层调度器 TaskScheduler 执行。

- HeartbeatReceiver

心跳接收器，所有 Executor 都会向 HeartbeatReceiver 发送心跳，当其接收到 Executor 的心跳信息后，首先更新 Executor 的最后可见时间，然后将此信息交给 TaskScheduler 进一步处理。

- ExecutorAllocationManager

Executor 动态分配管理器，根据负载动态的分配与删除 Executor，可通过其设置动态分配最小 Executor、最大 Executor、初始 Executor 数量等配置。

- ContextClearner

上下文清理器，为 RDD、shuffle、broadcast 状态的异步清理器，清理超出应用范围的 RDD、ShuffleDependency、Broadcast 对象。

- SparkStatusTracker

低级别的状态报告 API，只能提供非常脆弱的一致性机制，对 Job（作业）、Stage（阶段）的状态进行监控。

- HadoopConfiguration

Spark 默认使用 HDFS 来作为分布式文件系统，用于获取 Hadoop 配置信息。

```
print(spark)
print(sc)
```

*<pyspark.sql.session.SparkSession object at 0x00000266E2558F40>*

*<SparkContext master=local[*] appName=Reading Text>*

**2、查看环境变量**

```
import os
print(os.getenv("HADOOP_HOME"))
print(os.getenv("SPARK_HOME"))

!pyspark --version
```

环境变量是在操作系统中一个具有特定名字的对象，它包含了一个或者多个应用程序所将使用到的信息。通俗理解为：一些被指定的文件夹路径，目的是为了更快

速方便的找到想要的文件和文件夹。

**用法**：**os.getenv(key, default = None)**

**参数**:

key:表示环境变量名称的字符串

默认值(可选)：表示 key 不存在时默认值的字符串。如果省略，则默认设置为“无”。

**返回类型**：此方法返回一个字符串，该字符串表示环境变量键的值。如果 key 不存在，则返回默认参数的值。

**3、利用 PySpark 读取文本文件。注意此时只是创建了一个对文件的描述，文件内容并没有加载到内存中。**

```
file = sc.textFile("data/UNv1.0.en-zh.zh")
```

补充:在写 Spark 程序时,如果数据源分散在不同位置,则需要在 sc.TextFile 的时候指定多个数据源

**主要提出的方法有三种**:1、将多个文件变成一个 list 作为参数，准确的说是 list_str，后面会详细解释，2、使用 union 连接，3、使用 repartition，我们这里采用最简单直接的第一种方法，修改后的写法是:

**正确写法**：sc.TextFile( filename1 + "," + filename2 + "," + filename3)

```
print(file)
```

*data/UNv1.0.en-zh.zh MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0*

**4、查看文件的行数**

```
print(file.count())
```

*15886041*

**5、查看前 5 行**

```
text1 = file.take(5)
print(*enumerate(text1), sep="\n")
```

*(0, '第 918(1994)号决议')*

*(1, '1994 年 5 月 17 日安全理事会第 3377 次会议通过')*

*(2, '安全理事会，')*

*(3, '重申其以往关于卢旺达局势的所有决议，特别是成立联合国卢旺达援助团(联卢援助团)的 1993 年 10 月 5 日第 872(1993)号决议，延长联卢援助团任务期限至 1994 年 7 月 29 日的 1994 年 4 月 5 日第 909(1994)号决议，以及调整联卢援助团的任务规定的 1994 年 4 月 21 日第 912(1994)号决议，')*

*(4, '回顾安理会主席以安理会名义在 1994 年 4 月 7 日发表的声明(S/PRST/1994/16)和在 1994 年 4 月 30 日发表的声明(S/PRST/1994/21)，')*

**enumerate 函数详解**

enumerate(iteration, start)函数默认包含两个参数，其中 iteration 参数为需要遍历的参数，比如字典、列表、元组等，start 参数为开始的参数，默认为 0（不写 start 那就是从 0 开始）。enumerate 函数有两个返回值，第一个返回值为从 start 参数开始的数，第二个参数为 iteration 参数中的值。

**6、查看随机抽取的 10 行（可能会比较慢！访问 http://localhost:4040 查看进度）：**

```
text2 = file.takeSample(withReplacement=False,
                        num=10, seed=12)
print(*enumerate(text2), sep="\n")
```

*(0, '第八编. 共同支助事务')*

*(1, '14. 在提出这一建议的同时，我必须提请注意观察员部队经费短缺问题。')*

*(2, '报告期间内，在国家政策、立法和做法方面是否发生了不利于第十三条所载权利的变化？')*

*(3, '议程项目 39')*

*(4, '三. 审议缔约国根据《公约》第九条提交的报告、评论和资料 25-40 9')*

*(5, '第十七届会议')*

*(6, '4. 2004 年 9 月 21 日至 25 日，特别报告员在联合国人权事务高级专员办事处(人权高专办)简要介绍了情况。')*

*(7, 'IMF')*

*(8, '因此，有必要调整观察员部队的核定兵力，增加 10 名军事特遣队人员。')*

*(9, '《民法》中，组织、协会、企业、基金会及其他具有私营性质的实体被定义为私营法人。')*

**7、利用 Filter 操作筛选符合条件的行**

```
# 所有包含“乌克兰“的行
ukraine = file.filter(lambda x: x.__contains__(“乌克兰“)
print(ukraine)
```

*PythonRDD[3] at RDD at PythonRDD.scala:53*

注意，运行上述语句**并不会直接进行计算**，而只是把**要进行的操作保存了下来**(这一段运行的很快)。只有当**真正需要获取结果**时计算才会发生，比如下面获取前 5 行的操作:

```
text3 = ukraine.take(5)
print(*enumerate(text3), sep= “\n”)
```

*(0, '根据乌克兰和格鲁吉亚共和国政府的请求，美国航天局与空间雷达实验室 2 号协同工作的和科学家于 1994 年 10 月得到了有关航天飞机的数据，将这些数据转送了两国。')*

*(1, '这个机构使用卫星图象、作物模型以及遥感气候数据支持国务院对前苏联各国的粮食需要，特别是遭受干旱的乌克兰作出的评价。')*

*(2, '最近签订的协议包括 1994 年 6 月 30 日与俄罗斯联邦签订的协议和 1994 年 9 月 16 日与乌克兰签订的协议。')*

*(3, '白俄罗斯的种族比例如下:白俄罗斯人,78%;俄罗斯人,13%;波兰人,4%;乌克兰人,3%;犹太人 1%;其它民族,1%。')*

*(4, '乌克兰 第十三次报告 1994 年 1 月 5 日 -')*

Filter 操作的参数是一个函数，该函数输入一个字符串，输出 **True 或 False** 的取值。

filter 函数功能是对元素进行过滤，对每个元素应用 f 函数，返回值为 true 的元素在 RDD 中保留，**返回值为 false 的元素将被过滤掉**。内部实现相当于生成 FilteredRDD(this, sc.clean(f))

上面该函数通过 lambda 表达式实现，也可以**使用定义好的函数：**

```
def my_filter(x) :
    return x.__contains__(“乌克兰”) and x.__conatins__(“俄罗斯”)

ukraine2 = file.filter(my_filter)
text4 = ukraine2.take(5)
```

```
print(*enumerate(text4), sep="\n")
```

*(0, '最近签订的协议包括 1994 年 6 月 30 日与俄罗斯联邦签订的协议和 1994 年 9 月 16 日与乌克兰签订的协议。')*

*(1, '白俄罗斯的种族比例如下:白俄罗斯人,78%;俄罗斯人,13%;波兰人,4%;乌克兰人,3%;犹太人 1%;其它民族,1%。')*

*(2, '为了根据俄罗斯和乌克兰的本国及国际方案对地球表面进行观测，包括对自然环境进行环境监测，并且为了获得北极地区冰帽状况以及太平洋暗礁地带状况的最新情报，于 1995 年 8 月 31 日发射了一颗 Sich-1 号卫星（乌克兰）。')*

*(3, '该空间站装备的科学仪器是由以下国家的科学家和专家设计的：奥地利、比利时、加拿大、古巴、捷克共和国、芬兰、法国、德国、希腊、匈牙利、意大利、吉尔吉斯斯坦、波兰、罗马尼亚、俄罗斯联邦、斯洛伐克、瑞典、乌克兰、大不列颠及北爱尔兰联合王国、乌兹别克斯坦以及欧洲航天局的成员国。')*

*(4, '该航天器的初期轨道，其近地点为 500 公里，远地点为 300,000 公里，在预期三年的飞行期限间，达到运行轨道的稳定标准和从俄罗斯联邦与乌克兰地面站基本连续可见度标准，达到从发射后一年近地点高度迅速变换达到 40,000 公里的高度。')*

**8、Map 操作可以对数据的每一行进行变换，例如将字符串分割成若干字串的列表：**

```
split = file.map(lambda x: x.split(“, ”))
text5 = split.take(5)
print(*enumerate(text5), sep="\n")
```

*(0, ['第 918(1994)号决议'])*

*(1, ['1994 年 5 月 17 日安全理事会第 3377 次会议通过'])*

*(2, ['安全理事会', ''])*

*(3, ['重申其以往关于卢旺达局势的所有决议', '特别是成立联合国卢旺达援助团(联卢援助团)的 1993 年 10 月 5 日第 872(1993)号决议', '延长联卢援助团任务期限至 1994 年 7 月 29 日的 1994 年 4 月 5 日第 909(1994)号决议', '以及调整联卢援助团的任务规定的 1994 年 4 月 21 日第 912(1994)号决议', ''])*

*(4, ['回顾安理会主席以安理会名义在 1994 年 4 月 7 日发表的声明(S/PRST/ 1994/16)和在 1994 年 4 月 30 日发表的声明(S/PRST/1994/21)', ''])*

Flat map 可以把上述列表展开：

```
split2 = file.flatMap(lambda x: x.split(“, ”))
text6 = split2.take(10)
print(*enumerate(text6), sep="\n")
```

*(0, '第 918(1994)号决议')*

*(1, '1994 年 5 月 17 日安全理事会第 3377 次会议通过')*

*(2, '安全理事会')*

*(3, '')*

*(4, '重申其以往关于卢旺达局势的所有决议')*

*(5, '特别是成立联合国卢旺达援助团(联卢援助团)的 1993 年 10 月 5 日第 872(1993)号决议')*

*(6, '延长联卢援助团任务期限至 1994 年 7 月 29 日的 1994 年 4 月 5 日第 909(1994)号决议')*

*(7, '以及调整联卢援助团的任务规定的 1994 年 4 月 21 日第 912(1994)号决议')*

*(8, '')*

*(9, '回顾安理会主席以安理会名义在 1994 年 4 月 7 日发表的声明(S/PRST/ 1994/16)和在 1994 年 4 月 30 日发表的声明(S/PRST/1994/21)')*

关闭 Spark：

```
sc.stop()
```

搜索 **spark 算子**进行补充

## lec3-functional

Python 函数式编程简介

PySpark 基础

**1）函数式编程**

a. 函数式编程

a) 一种编程范式、风格、思维方式

b) 与大数据处理、Spark 框架高度相关

c) Python 提供了多种工具来实现

b. 动机

函数式编程的处理方式：

a) 将数据表达为某种集合

b) 遍历集合中的元素，同时进行处理

c) 内存占用可控，对算法提出要求（遍历只能向前不能后退）

c. 核心思想

a) 将数据想象成信息流

b) 每次只能看一小批数据

c) 看完之后决定要做什么

d) 操作完成后该批数据就被“丢弃”

e) 下一批数据进入

f) 往复循环

**2）迭代器**

a. 迭代器

a) 函数式编程中的核心概念

b) 访问数据的一种机制

c) 迭代器每次返回数据流中的一个元素

d) 使用者不需要知道数据是如何存储的

e) 只需要定义好对元素进行何种操作

f) 迭代器只能向前不能后退

g) YOLO：You Only Live/Look Once

**1、例 1：迭代器**

**“索引式”访问元素**：

```
lst = [1, 1, 2, “a”, 3.14]
for i in range(len(lst)):
    print(lst[i])
```

*1*

*1*

*2*

*a*

*3.14*

**“迭代器式”访问**：

```
it = iter(lst)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

```
1
1
2
a
3.14
```

**迭代器遍历**:

```
it = iter(lst)
for e in lst:
    print(e)
1
1
2
a
3.14
```

**有些集合无法用索引进行访问**:

```
st = set(lst) #  集合唯一性  没有顺序
print(st)
{3.14, 1, 2, 'a'}
```

**以下将出现错误**:

```
print(st[0])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 print(st[0])

TypeError: 'set' object is not subscriptable
```

**但可以用迭代器**:

```
it = iter(st)
print(next(it))
print(next(it))
print()
# for  循环
for e in st:
   print (e)
3.14
1

3.14
1
2
a
```

**例子+练习**:

简单：计算文件行数

中等：计算序列样本方差

困难：随机抽样

**2、例 2：计算文件行数**

给定一个文本文件，计算文件的行数

此时迭代器每次取出文件的一行，形成一个字符串

**文件可以作为一个迭代器**:

```
file = open("data/UNv1.0.en-zh.zh", encoding="utf-8")
file
<_io.TextIOWrapper name='data/UNv1.0.en-zh.zh' mode='r' encoding='utf-8'>

len(next(file)) #  每次提取一条字符
26

file = open("data/UNv1.0.en-zh.zh", encoding="utf-8")
print(next(file))
print(next(file))
file.close()
第 918(1994)号决议

1994 年 5 月 17 日安全理事会第 3377 次会议通过
```

**遍历文件的每一行**:

```
file = open("data/UNv1.0.en-zh.zh", encoding="utf-8")
count = 0
for line in file: # 没用文件内容，移动的步骤是我们需要的信息
    count = count + 1
file.close()
print(count)
15886041
```

**一种更推荐的语法**:

```
count = 0
with  open("data/UNv1.0.en-zh.zh",  encoding="utf-8")
as file: #  结束后自动关闭
    for line in file:
        count = count + len(line)
print(count)
701876199
```

**3、例 3：计算序列样本方差**

给定一个数值型序列，计算其样本方差，要求只能遍历数据一遍

**生成数据**:

```
import math
vec = [math.sin(i + math.exp(i)) for i in range(100)]
```

**简单做法：先计算均值，再计算方差**:

```
#  记录样本量
count = 0
#  所有元素之和
vec_sum = 0.0
for e in vec:
    vec_sum = vec_sum + e
    count = count + 1
#  计算均值
mean = vec_sum / count
#  计算方差
```

```
vec_var = 0.0
for e in vec:
    vec_var = vec_var + (e - mean) ** 2
vec_var = vec_var / (count - 1)
print(vec_var)
```

*0.4639116338164097*

**一次遍历方法：$(n-1)S^2 = \sum_i (x_i - \bar{x})^2 = \sum_i x_i^2 - n\bar{x}^2$**

```
count = 0
ss = 0.0
s = 0.0
for e in vec:
    ss = ss + e * e
    s = s + e
    count = count + 1
mean = s / count
vec_var = (ss - count * mean * mean) / (count - 1)
print(vec_var)
```

*0.4639116338164096*

**4、例 4：随机抽样**

给定一个正数序列$(w_1, w_2, \dots, w_n)$和一个数值序列$(v_1, v_2, \dots, v_n)$，要求从$\{v_i\}$中随机抽取一个元素，其中每个$v_i$以$p_i = \frac{w_i}{\sum_j^n w_j}$的概率被抽取到。要求只能遍历数据一遍

---

***Algorithm* 1** *The Select Algorithm*

---

***Input***: $\{a_1, \dots, a_n\}, a_i \geq 0, read\ in\ one\ pass, i.e., one\ sequential\ read, over\ the\ data.$

***Output***: $i^*, a_{i^*}$

1. $D = 0.$
2. ***for*** $i = 1\ to\ n\ do$
3. $\quad D = D + a_i$
4. $\quad With\ probability\ \frac{a_i}{D}, let\ i^* = i\ and\ a_{i^*} = a_i$
5. ***end for***
6. $Return\ i^*, a_{i^*}$

---

The following lemma establishes that in one pass over the data one can sample an element according to certain probability distributions

***Lemma* 1** *Suppose that* $\{a_1, \dots, a_n\}, a_i \geq 0, are\ read\ in\ one\ pass, i.e., one\ sequential\ read\ over\ the\ data, by\ the\ SELECT\ algorithm. Then\ the\ SELECT\ algorithm\ requires\ O(1)\ additional\ storage\ space\ and\ return\ i^*\ such\ that\ Pr[i^* = i] = a_i / \sum_{i'=1}^n a_{i'}$

$\boldsymbol{w = (w_1, \dots, w_n)}$，$\boldsymbol{v = (v_1, \dots, v_n)}$，每个 $\boldsymbol{v_i}$ 以 $\frac{w_i}{\sum_j w_j}$ 的概率被抽到

a. **收集**

a) 如果想按传统的方式将迭代器的元素一次性取出来，可以将迭代器转换成列表

b) lst = list(it)

c) 但对于大型的数据，谨慎进行该操作

d) 因为可能会占用非常多的内存

```
# 生成数据
import string
wvec = list(range(1, 27))
vvec = list(string.ascii_uppercase)
```

**知识点 1：同时迭代两个集合**

```
it = zip(wvec, vvec)
print(next(it))
print(next(it))
print()

for w, v in zip(wvec, vvec):
    # print(w, v)
    print(f"w is {w}, v is {v}")
```

*(1, 'A')*

*(2, 'B')*

*w is 1, v is A*

*w is 2, v is B*

*w is 3, v is C*

*w is 4, v is D*

*w is 5, v is E*

*w is 6, v is F*

*w is 7, v is G*

*w is 8, v is H*

*w is 9, v is I*

*w is 10, v is J*

*w is 11, v is K*

*w is 12, v is L*

*w is 13, v is M*

*w is 14, v is N*

*w is 15, v is O*

*w is 16, v is P*

*w is 17, v is Q*

*w is 18, v is R*

*w is 19, v is S*

*w is 20, v is T*

*w is 21, v is U*

*w is 22, v is V*

*w is 23, v is W*

*w is 24, v is X*

*w is 25, v is Y*

*w is 26, v is Z*

**解法：**

```
# 设计随机数种子
import random
random.seed(123)

D = 0.0
i = 0           # 迭代器中每个元素的索引
loc = 0         # 最终取出的 v 元素的位置（索引）
item = None  # 最终取出的 v 元素
for w, v in zip(wvec, vvec):
    D = D + w
    prob = w / D
    # 以这一概率选择 v
    if random.random() <= prob:
        loc = i
        item = v
    i = i + 1 # 后续被去掉
print(loc)
print(item)
```

```
24
Y
```

**知识点 2：迭代集合的同时获取索引**

```
it = enumerate(wvec)
print(nect(it))
print(next(it))
print()
it = enumerate(zip(wvec, vvec))
print(next(it))
print(next(it))
```

```
(0, 1)
(1, 2)


(0, (1, 'A'))
(1, (2, 'B'))
```

**注意此处迭代元素的写法：**

```
for i, (w, v) in enumerate(zip(wvec, vvec)):
    print(f"i is {i}, w is {w}, v is {v}")
```

```
i is 0, w is 1, v is A
i is 1, w is 2, v is B
i is 2, w is 3, v is C
i is 3, w is 4, v is D
i is 4, w is 5, v is E
i is 5, w is 6, v is F
i is 6, w is 7, v is G
i is 7, w is 8, v is H
i is 8, w is 9, v is I
i is 9, w is 10, v is J
i is 10, w is 11, v is K
i is 11, w is 12, v is L
i is 12, w is 13, v is M
i is 13, w is 14, v is N
i is 14, w is 15, v is O
i is 15, w is 16, v is P
i is 16, w is 17, v is Q
i is 17, w is 18, v is R
i is 18, w is 19, v is S
i is 19, w is 20, v is T
i is 20, w is 21, v is U
i is 21, w is 22, v is V
i is 22, w is 23, v is W
i is 23, w is 24, v is X
i is 24, w is 25, v is Y
i is 25, w is 26, v is Z
```

**改写之前的方法，省去 i 的手动更新**:

```
def random_select(wvec, vvec):
    D = 0.0
    Idc = 0 # 最终取出的 v 元素的位置（索引）
    item = None  # 最终取出的 v 元素
    for i, (w, v) in enumerate(zip(wvec, vvec)):
        D = D + w
        prob = w / D
        # 以这一概率选择 v
        if random.random() <= prob:
            loc = i
            item = v
    return loc, item
```

**测试抽样概率**:

```
import collections
random.seed(123)
res = [random_select(wvec, vvec)[1] for i in range(10000)]
# 计算频率
elements_count = collections.Counter(res)
for key, value in elements_count.items():
    print(f"{key}: {value}")
```

```
Y: 744
V: 618
L: 351
Q: 506
X: 679
E: 123
Z: 731
R: 495
U: 569
W: 637
S: 551
O: 428
C: 70
```

```
M: 403
N: 433
K: 298
T: 533
I: 276
P: 445
J: 327
D: 120
F: 182
G: 182
H: 220
B: 56
A: 23
```

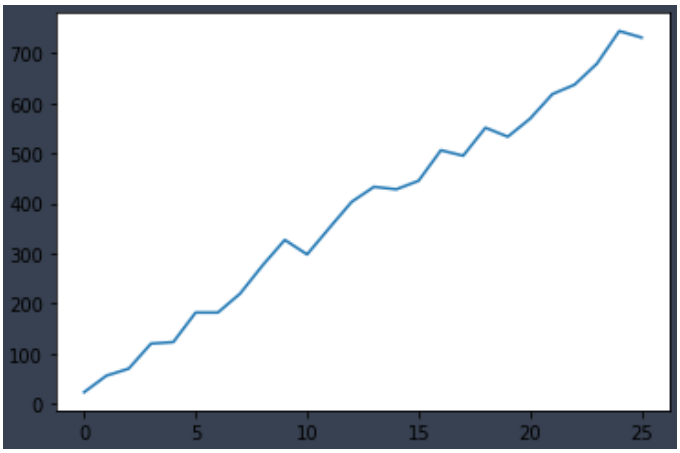
**按 key 排序**:

```
freq = list(elements_count.items())
freq.sort()
print(freq)
```

```
[('A', 23), ('B', 56), ('C', 70), ('D', 120), ('E', 123), ('F', 182), ('G', 182), ('H', 220), ('I', 276), ('J', 327), ('K', 298), ('L', 351), ('M', 403), ('N', 433), ('O', 428), ('P', 445), ('Q', 506), ('R', 495), ('S', 551), ('T', 533), ('U', 569), ('V', 618), ('W', 637), ('X', 679), ('Y', 744), ('Z', 731)]
```

**作图**:

```
import matplotlib.pyplot as plt
freqs = [v for k, v in freq]
plt.plot(freqs)
plt.show()
```




**5、例 5：Reduce**

b. **归约**

a) 即 Reduce 操作

b) 将迭代器中的元素逐个进行二元运算

```
def add(x, y):
    return x + y

import functools # 加载这个包
x = [1, 2, 3, 4, 5]
it = iter(x)
xsum = functools.reduce(add, it)
print(xsum)
```

```
15
```

reduce 的第三个参数可以设定规约的初值:

```
def mult(x, y):
    return x * y

it = iter(x)
xprod = functools.reduce(mult, it, 1)
print(xprod)
```

```
120
```

**6、例 6：Filter**

1) **迭代器变换**

a. **变换**

a) 在上节课中，我们使用了 PySpark 的 filter()和 map()来对数据进行处理

b) 事实上，这正是函数式编程的思想

**Filter** 将一个迭代器转变为一个新的迭代器，相当于对原始信息流进行了一次过滤，只有满足条件的元素会被新迭代器访问

**it 是原始的迭代器**

```
x = list(range(10))
it = iter(x)
print(next(it)) # 0
print(next(it)) # 1
print(next(it)) # 2
```

```
0
1
2
```

**it_filtered 是“过滤”后的迭代器**

```
def is_even(x):
    return x % 2 == 0

it = iter(x)
it_filtered = filter(is_even, it)
print(next(it_filtered)) # 0
print(next(it_filtered)) # 2
print(next(it_filtered)) # 4
```

```
0
2
4
```

Filter 生成新的迭代器时不会实际进行过滤的计算，上例中 it_filtered = filter(is_even, it)不会对原始数据进行计算，只有需要 it_filtered 返回元素时才会

**7、例 7：Map**

Map 同样是将一个迭代器变换成另一个新迭代器取出的元素都是原迭代器元素的变换。同样，生成迭代器时不会实际发生计算

**it 是原始的迭代器:**

```
x = list(range(10))
```

```
it = iter(x)
print(next(it)) # 0
print(next(it)) # 1
print(next(it)) # 2
```
*0*

*1*

*2*

**it_mapped 是变换后的迭代器：**

```
def square(x):
    return x * x # , x  同时返回元组

it = iter(x)
it_mapped = map(square, it) # map 输入输出可以不是同一类型
print(next(it_mapped))   # 0
print(next(it_mapped))   # 1
print(next(it_mapped))   # 4
```
*(0, 0)*

*(1, 1)*

*(2, 4)*

**8、例 8：islice**

**其他工具**

itertools 模块提供了很多其他有用的迭代工具。例如 itertools.islice()可以用来实现 PySpark 中的 take()操作。生成一个截断长度的迭代器

更多的可参考 https://docs.python.org/zh-cn/3/library/itertools.html#module-itertools

**it 是原始的迭代器, it_n 是长度截断的迭代器：**

```
import itertools
x = list(range(100))
it = iter(x)
it_n = itertools.islice(it, 5)
for e in it_n:
    print(e)
```
*0*

*1*

*2*

*3*

*4*

**9、例 9：组合使用，Lambda 表达式**

map, filter, islice 等可以互相叠加。本质上是把一个迭代器转换成另一个迭代器。

map 和 filter 都需要一个函数作为参数。通常可以先定义函数，再传入。而利用 Lambda 表达式（即匿名函数）可以在行内直接定义简单地函数 https://www.runoob.com/python3/python3-function.html

```
import itertools
it = iter(range(10000))
it_new = filter(lambda x: x % 2 == 0, it)
it_new = map(lambda x: x * x, it_new)
it_new = itertools.islice(it_new, 10)
print(list(it_new))
```
*[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]*

**10、练习：使用 Map/Reduce 等函数式编程工具计算向量的均值，规定只能使用迭代器一次，不能使用 for 循环，且不知道向量的长度。**

```
import math
vec = [math.sin(i + math.exp(i)) for i in range(100)]
it = iter(vec)

def add(x, y):
    return x[0] + y[0], x[1] + y[1], x[2] + y[2]

it_new = map(lambda x: (1, x, x*x), it)
count, s, ss = functools.reduce(add, it_new)
mean = s / count
vec_var = (ss - count * mean * mean) / (count - 1)
print(mean)
print(vec_var)
```
*-0.026577815770465873*

*0.46773473930329096*

## lec4-danmu

**函数式编程是一种不同以往的范式尽量“忘记”之前的思维方式**

综合使用函数式编程技巧提取 B 站视频的 N 条实时弹幕，主要知识点：

a. B 站 API 调用（附加知识）

b. 网络编程（附加知识）

c. 字符串操作

d. filter()和 map()等

**1、整体思路**

1）给定一个视频网址，获取视频的 BV 号，如 V1nL411x7jH

2）给定 BV 号，返回视频信息：https://api.bilibili.com/x/web-interface/view?bvid=BV1nL411x7jH

其中包含弹幕号 CID，如 454378887

3）获取弹幕：https://api.bilibili.com/x/v1/dm/list.so?oid=454378887

4）输入一个视频网址，获取视频的 BV 号，例如：https://www.bilibili.com/video/BV1nL411x7jH => BV1nL411x7jH

5）实际情况中网址后面可能带有问号，需要将其去除，例如：

https://www.bilibili.com/video/BV1nL411x7jH?from=search&seid=13994942

6）如果视频不包含 BV 号，则抛出异常，例如:

https://www.bilibili.com/video/av463010182

7) Baidu "python string documentation"

Doc:

https://docs.python.org/zh-cn/3/library/string.html

```
url1 = "https://www.bilibili.com/video/BV1nL411x7jH"
url2 =
"https://www.bilibili.com/video/BV1nL411x7jH?from=search&seid=13994942816196159115&spm_id_from=333.337.0.0"
url3 =
"https://www.bilibili.com/video/BV1nL411x7jH/?spm_id_from=333.788"
url4 = https://www.bilibili.com/video/av463010182
```

```
bv = list(filter(lambda s: s.startswith("BV"),bv))
bv
```

*['BV1nL411x7jH']*

```
def get_bv_from_url(url):
    bv = url.split("/") # 列表，其中包含字符串
    bv = list(filter(lambda s: s.startswith("BV"),bv))
    if len(bv) <1:
        raise RuntimeError("BV warning") # 这个特别注意一下，自己生成警告
    bv = bv[0]
    qmark = bv.find("?") # 确定是否含"?"
    if qmark >= 0:
        bv = bv[:qmark] # 取索引，此处表示取得 qmark 之前的内容
    return bv
```

```
bv = get_bv_from_url(url2)
bv
```

*'BV1nL411x7jH'*

**获取 CID**

https://api.bilibili.com/x/web-interface/view?bvid=BV1nL411x7jH => 454378887

```
api = "https://api.bilibili.com/x/web-interface/view?
      bvid="
url = api + bv
url
```

*'https://api.bilibili.com/x/web-interface/view?bvid=BV1nL411x7jH'*

**Baidu "python 获取 url 内容"**

https://www.cnblogs.com/blogzyq/p/11067648.html

```
# 固定 code 读取网页内容
import urllib.request
response = urllib.request.urlopen(url)
string = response.read()
html = string.decode("utf-8") # 返回的数据类型为 json
print(html)
```

*{"code":0,"message":"0","ttl":1,"data":{"bvid":"BV1nL411x7jH","aid":463010182,"videos":1,"tid":95,"tname":"数码","copyright":1,"pic":"http://i1.hdslb.com/bfs/archive/040964501595ca857e1e88ffdaa3951564f4ba9d.jpg","title":"【硬件科普】带你认识 CPU 第 00 期——什么是 MOSFET","pubdate":1632024016,"ctime":1632024054,"desc":"本期视频是我们新系列视频 ...*

*:false},"is_season_display":true,"user_garb":{"url_image_ani_cut":""},"honor_reply":{"honor":[{"aid":463010182,"type":3,"desc":"全站排行榜最高第 55 名","weekly_recommend_num":0},{"aid":463010182,"type":4,"desc":"热门","weekly_recommend_num":0}]},"like_icon":"","need_jump_bv":false}}*

**Baidu "python json"**

https://www.runoob.com/python/python-json.html

```
import json
doc = json.loads(html) # 返回的是 python 中的字典，对 json 格式内容进行解析
doc
cid = doc["data"]['cid']
print(cid) # 得到了当期网址的 cid
```

*454378887*

```
import urllib.request
import json
def get_cid_from_bv(bv):
    api = "https://api.bilibili.com/x/web-interface/
          view?bvid="
    url = api + bv
    response = urllib.request.urlopen(url)
    string = response.read()
    html = string.decode("utf-8") # 返回的数据类型为 json
    doc = json.loads(html)
    cid = doc["data"]['cid']
    return cid

cid = get_cid_from_bv(bv)
cid
```

**获取弹幕**

https://api.bilibili.com/x/v1/dm/list.so?oid=454378887

```
api = "https://api.bilibili.com/x/v1/dm/list.so?oid="
url = api + str(cid)
url
```

*'https://api.bilibili.com/x/v1/dm/list.so?oid=454378887'*

```
response = urllib.request.urlopen(url)
response.getheader("Content-Encoding")
string = response.read()
string
b'\xbc\xbd{o\\Y\x92\'\xf6U\xe4\x02\x0c\xd8k\xed\x9d\x13\x11\'\xcecQS\x0b\xbbw\x170\
x8c\xa9)\xa0\xb7\xed?l\xc3\xc8go\xc1]U=]=\xed\xf5\x7fI=/\x8a"\xa9\xf7\x8b\x12%\x95(\x
a9T%\x92\xaa\xd2\x83L\x92\xd2w\x99\xce{3\xf9\x97\xbe\x82\x11qn&o\xa6n\x8e\xc7\xc0
...
xed\xa7`\xd7.0\x93\x13\xf1\xd6\xd8\x93s\x9d\x96z\xae\xf8s\xb7.W\xd9\xf0+\x97\x8c\'`\x
ef/\xab\x96\xb0\xde8\xfe\xcbj\x9d\xae\x89S\x11Z\x8b~\xb6l\xae\xca\x0f=\xf4;\x9dX\x8b
\xe42\xdfb\x85\x0f\xcdH\xaa\xfc\xf3L\xcd\xd0\xfd\xe7\x7f\xfe\xfa\xff\x0b\x00\x00\xff\xff'
```

```
# 跟上个模块应该内容一样
response = urllib.request.urlopen(url)
html = response.read() # 返回的数据类型为 json
print(html) # 这里是加密的
b'\xbc\xbd{o\\Y\x92\'\xf6U\xe4\x02\x0c\xd8k\xed\x9d\x13\x11\'\xcecQS\x0b\xbbw\x170\
x8c\xa9)\xa0\xb7\xed?l\xc3\xc8go\xc1]U=]=\xed\xf5\x7fI=/\x8a"\xa9\xf7\x8b\x12%\x95(\x
a9T%\x92\xaa\xd2\x83L\x92\xd2w\x99\xce{3\xf9\x97\xbe\x82\x11qn&o\xa6n\x8e\xc7\xc0
...
\xdd~\nv\xed\x0239\x11o}B9\xd7i\xa9\xe7\x8a\x8ew\xfbR\x95\r\xbf|\xd1h\x07\xf6\xff\xb
2j\t\xeb\xcd\xcf\x7fY\xad\xb3?q*Bk\xd1\xcf\x96\xcdE\xfe\xa1\x87~\xa7\x13\xabe\x91\\\x
a6o\xac\xe0\xa6\x19\x98\x95\x7f\x9e\xa9\x19\xba\xff\xfc\xcf_\xff\x7f\x01\x00\x00\xff\xff'
```

**Baidu "python content encoding deflate 乱码"**

https://blog.csdn.net/weixin_43116971/article/details/121291281

https://renjie.me/2016/06/04/python-decompresshttp-response-gzip-and-deflate-content-encoding/

https://www.jianshu.com/p/2c2781462902?winzoom=1

https://blog.csdn.net/vieri_ch/article/details/45220119

https://stackoverflow.com/questions/61174385/how-can-i-inflate-this-zlib-byte-string-in-python

```
import zlib
html = zlib.decompress(string, wbits=-zlib.MAX_WBITS).
        decode ('utf-8')
html
'<?xmlversion="1.0"
encoding="UTF8"?><i><chatserver>chat.bilibili.com</chatserver><chatid>454378887</
chatid><mission>0</mission><maxlimit>1000</maxlimit><state>0</state><real_name
>0</real_name><source>k-v</source><d
p="367.16500,5,25,15138834,1632551536,0,3b1923ee,58463292992643072,10"> 红 石
中 继 器</d><d p="118.05900,1,25,16777215,1634644458,0,1b757676,584633063515
01312,10">师傅别念了，徒儿知错了</d><d
...
</d><d p="493.67000,5,25,15452926,1639196647,0,ad3bd97f
,587755406624455 68,5">这里的电压应该指的是「电势」</d><d p="559.
72100,1,25,16777215,1639196266,0,f21daa8f,587753409747512 32,5"> 厉 害 厉 害 懂 了
</d></i>'
```

**Baidu "python 格式化 xml"**

https://www.cnblogs.com/atrox/p/13579541.html

```
import xml.dom.minidom
xml2 = xml.dom.minidom.parseString(html)
xml_pretty_str = xml.toprettyxml()
print(xml_pretty_str)
<?xml version="1.0" ?>
<i>
        <chatserver>chat.bilibili.com</chatserver>
        <chatid>454378887</chatid>
        <mission>0</mission>
        <maxlimit>1000</maxlimit>
        <state>0</state>
        <real_name>0</real_name>
        <source>k-v</source>
        <dp="367.16500,5,25,15138834,1632551536,0,3b1923ee,58463292992643072,10"
        >红 石 中 继 器</d>
        <dp="118.05900,1,25,16777215,1634644458,0,1b757676,58463306351501312,10"
        >师傅别念了，徒儿知错了</d>
...
        <dp="152.34300,1,25,16777215,1639229457,0,d4393107,58792742773623808,5"
>硅和磷</d>
        <d p="74.72800,1,25,16777215,1639229379,0,d4393107,5879270160816435 2,5">
        水这玩意可太复杂了</d>
        <d  p="30.66700,1,25,16777215,1639226518,0,2377630,58791202104048128,5">
        半导体物理路过</d>
</i>
```

```
def filter_line(line):   #filter 提取到有弹幕的行（特点：
                          \t<d）
    pass

def clean_line(line): # 用 map 将一行含有弹幕正文和其
                      他信息的数据 mapping 到纯净的弹幕信息
    pass

it = filter(filter_line, lines)
it = map(clean_line, it)
list(it)
```

a. 如果想按传统的方式将迭代器的元素一次性取出来，可以将迭代器转换成列表

b. lst = list(it)

c. 但对于大型的数据，谨慎进行该操作

d. 因为可能会占用非常多的内存

## lec5-pyspark-rdd

a. **数据结构**

a) 在之前的课程中，我们曾演示用 PySpark 进行简单的数据处理

b) 事实上，其操作与基于迭代器的函数式编程非常相近
c) PySpark 中支持该类运算的结构叫做 **RDD(Resilient Distributed Dataset)**
d) 后续牵涉到统计模型时主要进行**矩阵和向量**的操作
e) 我们使用 Numpy 模块提供的数据结构
b. **RDD**
a) RDD 是 Spark/PySpark 中的核心底层数据结构
b) 与迭代器类似，RDD 可以进行 Map、Filter、Reduce 和收集等常见操作
c) 但 RDD 还有一些独特的性质
d) RDD 具有容错机制（Resilient)
e) RDD 是分布式的（Distributed)
f) 与迭代器不同，RDD 可以重复进行使用

**1、准备工作**

**配置和启动 PySpark:**

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
spark = SparkSession.builder.\
        master("local[*]").\
        appName("PySpark RDD").\
        getOrCreate()
sc = spark.sparkContext #存成变量，后面反复调用
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

*<pyspark.sql.session.SparkSession object at 0x0000022A7F740AF0>*

*<SparkContext master=local[*] appName=PySpark RDD>*

**2、RDD**

**创建一个包含了数据的列表：**

```
import math
vec = [math.sin(i + math.exp(i)) for i in range(100)]
```

**将其转换为分布式数据结构**：（textFile 是从外部存储中读取数据来创建 RDD 的方法）
通过调用 SparkContext 的 **parallelize** 方法，在一个已经存在的 Scala 集合上创建的（一个 Seq 对象）。集合的对象将会被拷贝，创建出一个可以被并行操作的分布式数据集

```
dat = sc.parallelize(vec) # 基于网络传输方法，比直接调用数据慢
dat # 迭代器思路，大数据处理思路
```

*ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274*

dat 的类型是 RDD（Resilient Distributed Dataset）,是一种类似于迭代器的结构，可以看作是某种数据类型的容器。例如，dat 代表了一些数字的集合。

**类似于 python 中原生的函数式编程工具，可以在 RDD 中使用 Map/Filter/Reduce 等操作。例如计算求和**:

```
dat.reduce(lambda x, y: x + y) # pyspark 比在 python 简介
```

*-2.246114436451575*

**灵活使用 Map 函数计算均值：**

```
dat2 = dat.map(lambda x :(1, x))
dat2 # 将 RDD 当做容器，生成新的 RDD，可以调用 reduce，可以写在一行，能否接着套用，原则在于是否回 RDD
```

*PythonRDD[3] at RDD at PythonRDD.scala:53*

```
dat.map(lambda x: (1, x)).reduce(lambda x, y: (x[0] + y[0], x[1] + y[1])) # 同时获得样本量和求和
```

*(100, -2.246114436451575)*

```
# 自己写一个生成方差
dat3 = dat.map(lambda x: (1, x, x*x)).reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2]))
mean = dat3[1] / dat3[0]
vec_var = (dat3[2] - dat3[0] * mean * mean) / (dat3[0] - 1)
print(vec_var)
```

*0.4639116338164096*

**Filter 操作：**

```
dat.filter(lambda x: x > 0).reduce(lambda x, y: x + y)
```

*28.015734417272107*

```
dat.filter(lambda x: x <= 0).reduce(lambda x, y: x + y)
```

*-30.261848853723677*

**使用 collect()函数可以将 RDD 中的数据全部取出返回给 Python. 但对于大型数据请谨慎操作！**

```
dat.collect() #一般不会使用，除非已知数据大小，很危险，知道最终长度固定 生成列表
```

*[0.8414709848078965,*
*-0.5452515566923345,*
*... 全部显示出来*
*0.7870114601458702,*
*-0.7206198329616429]*

**RDD 还提供了许多便捷的函数，如 count()用来计算数据、容器的大小，take()返回前 n 个元素等等。完整的函数列表可以参考官方文档**

```
dat.count() # 多少个元素，未必是一个行，可能把若干行打包成一行
```

*100*

```
dat.take(5) # 数据前 5 行放在内存中，相当于小型 collect，也可以用 first
```

**3、RDD 文件操作**

**利用 Numpy 创建一个矩阵，并写入文件：**

```
import os
```

```
import numpy as np
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 100
p = 5
mat = np.random.normal(size=(n, p))

if not os.path.exists("data"):
    os.makedirs("data", exist_ok=True)
np.savetxt("data/mat_np.txt", mat, fmt="%f", delimiter="\t")
```

**PySpark 读取文件并进行一些简单操作：**

```
file = sc.textFile("data/mat_np.txt")

# 打印矩阵行数
print(file.count())

# 空行
print()

# 打印前 5 行
text = file.take(5) # 对数据格式有所了解
print(*text, sep="\n")
```

```
100


-1.085631    0.997345    0.282978    -1.506295    -0.578600
1.651437    -2.426679    -0.428913    1.265936    -0.866740
-0.678886    -0.094709    1.491390    -0.638902    -0.443982
-0.434351    2.205930    2.186786    1.004054    0.386186
0.737369    1.490732    -0.935834    1.175829    -1.253881
```

```
file.first() # 本质还是一个字符串。真正需要的是里面的数字
```

```
'-1.085631\t0.997345\t0.282978\t-1.506295\t-0.578600'
```

**file 的类型也是 RDD。file 代表了一些字符串的集合，每个元素是矩阵文件中的一行：**

```
print(type(file))
print(type(file.first()))
```

```
<class 'pyspark.rdd.RDD'>
<class 'str'>
```

**我们可以对 RDD 进行变换，使一种元素类型的 RDD 变成另一种元素类型的 RDD。例如，将 file 中的每一个字符串变成一个 Numpy 向量，那么变换的结果就是以 Numpy.array 为类型的 RDD。**

```
line = file.first()
line
```

```
'-1.085631\t0.997345\t0.282978\t-1.506295\t-0.578600'
```

```
line.split("\t")
```

```
['-1.085631', '0.997345', '0.282978', '-1.506295', '-0.578600']
```

**为此，我们需要先编写一个转换函数：**

```
# str => np.array
def str_to_vec(line):
    # 分隔字符串
    str_vec = line.split("\t")
    # 将每一个元素从字符串变换成数值型
    num_vec = map(lambda s: float(s), str_vec) #字符串映射
    # 创建 Numpy 向量
    return np.fromiter(num_vec, dtype=float)
print(file.first())
print(str_to_vec(file.first()))
```

```
-1.085631    0.997345    0.282978    -1.506295    -0.578600
[-1.085631   0.997345   0.282978 -1.506295 -0.5786   ]
```

**np.fromiter**(iterable, dtype, count = -1)
从一个循环对象中提取数字，产生新的数组
Iterable：为生成数组提供数据的对象
dtype：生成的数组内数据类型

```
n, xsum = sc.textFile("data/mat_np.txt").\
    map(str_to_vec).\
    map(lambda x:(1,x)).\
    reduce(lambda x, y:(x[0] +y[0], x[1] + y[1]))
xsum / n
```

```
array([-0.0970826 ,  0.00832708, -0.03945197, -0.01719718, -0.04781526])
```

**4、RDD 分区**

RDD 的一个重要功能在于可以分区（分块），从而支持分布式计算。查看 RDD 的分区数：

```
file.getNumPartitions() # 会根据数据大小自动分区
```

```
2
```

RDD.getNumPartitions() → int
对于分区数可以通过 getNumPartitions()方法查看 list 被分成了几部分：rdd.getNumPartitions()
还可以手动指定分区数，从而支持更高的并行度。注意调用 repartition()函数不改变原有 RDD，要是用分区后的 RDD，需要重新赋值：

```
file_p10 = file.repartition(10)
print(file.getNumPartitions())
print(file_10.getNumPartitions()) # 增加并行度，生成一个新的 RDD
```

```
2
10
```

我们可以按分区对数据进行转换，例如将每个分区的数据转成 Nunpy 矩阵。需要使用的函数是 **mapPartitions**(), 其接收一个函数作为参数，该函数将对每个分区的迭代器进行变换。某些分区可能会是空集，需要做特殊处理。

```
# Iter[str] => Iter[matrix]，以迭代器为输入，以迭代器为
```

输出

```
def part_to_mat(iterator):
    # Iter[str] => Iter[np.array]
    iter_arr = map(str_to_vec, iterator) # 用刚刚的函数，转化为迭代器

    # Iter[np.array] => list(np.array)
    dat = list(iter_arr)

    # list(np.array) => matrix
    if len(dat) < 1: # Test zero iterator，万一给的迭代器为 0
        mat = np.array([])
    else:
        mat = np.vstack(dat)

    # matrix => Iter[matrix]
    yield mat #返回迭代器用 yield
```

函数原型：

- **vstack**(tup) ，参数 tup 可以是元组，列表，或者 numpy 数组，返回结果为 numpy 的数组。

```
v1 = np.array([1,2,3])
v2 = np.array([4,5,6])
v3 = np.array([7,8,9])
v4 = np.array([10,11,12])
np.vstack([v1,v2,v3,v4])
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

- **hstack()函数**

  函数原型：hstack(tup) ，参数 tup 可以是元组，列表，或者 numpy 数组，返回结果为 numpy 的数组。它其实就是水平(按列顺序)把数组给堆叠起来，vstack()函数正好和它相反。

```
import numpy as np
a=[[1],[2],[3]]
b=[[1],[2],[3]]
c=[[1],[2],[3]]
d=[[1],[2],[3]]
print(np.hstack((a,b,c,d)))
```

```
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]]
```

变换后的结果依然是一个 RDD，但此时元素类型变成了矩阵

```
dat_p10 = file_p10.mapPartitions(part_to_mat) # 对每一个分区做一个映射的操作
```

```
# 100 个向量的 RDD 变成 10 个矩阵的 RDD
print(type(dat_p10))
print()
print(dat_p10.first())# 边缘情况，第一个分区为 0
print()
print(dat_p10.take(3))
print()
print(dat_p10.count())
```

```
<class 'pyspark.rdd.PipelinedRDD'>


[]


[array([], dtype=float64),
 array([[-1.085631,  0.997345,  0.282978, -1.506295, -0.5786  ],
        [ 1.651437, -2.426679, -0.428913,  1.265936, -0.86674 ],
        [-0.678886, -0.094709,  1.49139 , -0.638902, -0.443982],
        [-0.434351,  2.20593 ,  2.186786,  1.004054,  0.386186],
        [ 0.737369,  1.490732, -0.935834,  1.175829, -1.253881],
        [-0.637752,  0.907105, -1.428681, -0.140069, -0.861755],
        [-0.255619, -2.798589, -1.771533, -0.699877,  0.927462],
        [-0.173636,  0.002846,  0.688223, -0.879536,  0.283627],
        [-0.805367, -1.727669, -0.3909  ,  0.573806,  0.338589],
        [-0.01183 ,  2.392365,  0.412912,  0.978736,  2.238143]]),
 array([[-1.294085, -1.038788,  1.743712, -0.798063,  0.029683],
        [ 1.069316,  0.890706,  1.754886,  1.495644,  1.069393],
        [-0.772709,  0.794863,  0.314272, -1.326265,  1.417299],
        [ 0.807237,  0.04549 , -0.233092, -1.198301,  0.199524],
        [ 0.468439, -0.831155,  1.162204, -1.097203, -2.1231  ],
        [ 1.039727, -0.403366, -0.12603 , -0.837517, -1.605963],
        [ 1.255237, -0.688869,  1.660952,  0.807308, -0.314758],
        [-1.085902, -0.732462, -1.212523,  2.087113,  0.164441],
        [ 1.150206, -1.267352,  0.181035,  1.177862, -0.335011],
        [ 1.031114, -1.084568, -1.363472,  0.379401, -0.379176]])]


10
```

我们可以用 filter 过滤掉空的分区：

```
dat_p10_nonempty = dat_p10.filter(lambda x: x.shape[0] > 0) # 过滤掉为 0 的 RDD

print(type(dat_p10_nonempty))
print()
print(dat_p10_nonempty.first())
print()
print(dat_p10_nonempty.count())
```

```
<class 'pyspark.rdd.PipelinedRDD'>


[[-1.085631  0.997345  0.282978 -1.506295 -0.5786  ]
 [ 1.651437 -2.426679 -0.428913  1.265936 -0.86674 ]
 [-0.678886 -0.094709  1.49139  -0.638902 -0.443982]
```

```
[-0.434351  2.20593   2.186786  1.004054  0.386186]
[ 0.737369  1.490732 -0.935834  1.175829 -1.253881]
[-0.637752  0.907105 -1.428681 -0.140069 -0.861755]
[-0.255619 -2.798589 -1.771533 -0.699877  0.927462]
[-0.173636  0.002846  0.688223 -0.879536  0.283627]
[-0.805367 -1.727669 -0.3909    0.573806  0.338589]
[-0.01183   2.392365  0.412912  0.978736  2.238143]]

7
```

**5、RDD 操作案例-求列和**

np.array 版本的 RDD 求矩阵的列和：

```
dat.reduce(lambda x1, x2: x1 + x2)
-2.246114436451575
```

从输入矩阵文件开始，将操作串联：

```
file.map(str_to_vec).reduce(lambda x1, x2: x1 + x2)
array([-9.70826 ,  0.832708, -3.945197, -1.719718, -4.781526])
```

使用分区版本的 RDD，先在每个分区上求列和：

```
sum_part = dat_p10_nonempty.map(lambda x: np.sum(x, axis=0))
sum_part.collect()
[array([-1.694266,  0.948677,  0.106428,  1.133682,  0.169049]),
 array([ 3.66858 , -4.315501,  3.881944,  0.689979, -1.877668]),
 array([-5.599247, -2.846053,  2.978673,  5.934997,  0.565914]),
 array([-5.60762 ,  0.977661, -5.157818, -2.868979, -0.570433]),
 array([-1.430094,  1.021662, -1.322512, -7.564743, -5.2081  ]),
 array([ 0.883702,  0.571757, -3.832934, -1.569966,  1.929   ]),
 array([ 0.070685,  4.474505, -0.598978,  2.525312,  0.210712])]
```

再将分区结果汇总：

```
sum_part.reduce(lambda x1, x2: x1 + x2)
array([-9.70826 ,  0.832708, -3.945197, -1.719718, -4.781526])
```

从输入矩阵文件开始，将操作串联：

```
file.repartition(10).\
    mapPartitions(part_to_mat).\
    filter(lambda x: x.shape[0] > 0).\
    map(lambda x : np.sum(x, axis=0)).\
    reduce(lambda x1, x2: x1 + x2)
```

使用真实值检验：

```
np.sum(mat, axis=0)
array([-9.7082586 ,  0.83270703, -3.94519179, -1.71971787, -4.78152553])
```

**6、RDD 操作案例 – 矩阵乘法**

模拟数据和真实值：

```
np.random.seed(123)
v = np.random.uniform(size=p)
res = mat.dat(v)
res
array([-1.65326187,  0.43284335, -0.83326669,  1.65616556,  0.47393998, -1.20594195, -1.09926452,
       -0.24483357, -0.58399139,  2.91984625, -1.22159268,  2.99167578,  0.04907967,  0.00526486,
       -1.78033411, -1.03704672,  1.27253333,  0.0280204 ,  0.88785436,  0.03485989,  1.45756374,
       -1.26733834,  0.89596346, -0.65027554,  1.24724097,  0.01338995, -0.45613812,  1.06057634,
        0.33513133,  0.30420446, -1.8306843 ,  0.81135409,  0.8563569 , -0.59189289, -0.58993733,
        0.85925493,  0.20665867, -2.07373852,  0.23232788, -2.69748055,  1.19285523, -0.22831252,
       -0.75495708,  1.04599886, -0.59922216, -2.14049979, -0.68492854,  0.13322705,  0.11576237,
       -1.07628496,  0.98308603,  2.28403745,  0.31327103,  0.97450293, -2.19087869, -1.38414598,
       -2.06428815, -1.19693787, -2.20837322,  1.79393849,  0.37940968,  0.98364566,  2.12782768,
        0.17228872, -1.42418937, -0.66160026,  0.20736396, -0.42352417, -1.83096405,  0.75557361,
       -1.87660221, -1.93437067, -0.51802796,  0.70099077, -2.27776851, -0.17137795, -0.77013413,
       -0.33715716, -0.46570004, -0.22885299,  0.07744646,  0.65965659,  1.30432415, -3.05410919,
       -1.55812228, -0.35166363, -0.26695372,  1.71736731,  1.42907711,  0.74512303, -1.17590892,
        1.28153134,  0.34006662,  1.1969479 ,  1.68259996, -2.70844742, -0.21291717,  2.74992919,
       -2.1979523 ,  0.60576651])
```

np.array 版 RDD：

```
res1 = dat.map(lambda x : x.dot(v)).collect()
res1[ :10]
```

分区版 RDD

```
res_part = dat_p10_nonempty.map(lambda x: x.dot(v)).
           collect()
res_part
[array([-1.65326236,  0.43284381, -0.83326654,  1.65616548,  0.47393997, -1.20594265, -1.09926439,
        -0.24483374, -0.58399159,  2.91984624]),
 array([-1.22159275,  2.99167581,  0.04907979,  0.0052652 , -1.78033393, -1.03704719,  1.27253296,
         0.02802034,  0.88785453,  0.03485997]),
 array([ 1.45756404, -1.26733862,  0.89596327, -0.65027561,  1.24724115,  0.01338989, -0.45613776,
         1.06057673,  0.33513193,  0.30420455,  2.28403732,  0.31327091,  0.97450361, -2.19087935,
        -1.38414658, -2.06428804, -1.19693768, -2.20837397,  1.79393855,  0.37941031]),
 array([-1.8306849 ,  0.81135346,  0.85635656, -0.59189308, -0.58993783,  0.8592545 ,  0.20665878,
        -2.07373867,  0.23232755, -2.69748044,  0.9836457 ,  2.12782845,  0.17228866, -1.42418964,
        -0.66160031,  0.20736295, -0.4235236 , -1.83096434,  0.75557361, -1.87660252]),
 array([ 1.19285543, -0.22831212, -0.75495698,  1.04599886, -0.59922233, -2.14049959, -0.68492885,
         0.13322687,  0.11576229, -1.07628444, -1.93437101, -0.51802806,  0.70099126, -2.27776847,
        -0.17137845, -0.77013423, -0.33715737, -0.46569988, -0.22885317,  0.07744686]),
 array([ 0.98308645,  0.65965705,  1.30432399, -3.05410973, -1.55812232, -0.35166336, -0.26695394,
         1.7173679 ,  1.42907711,  0.74512261, -1.17590882]),
 array([ 1.28153159,  0.34006666,  1.19694819,  1.68260023, -2.70844726, -0.21291761,  2.74992875,
        -2.1979517 ,  0.60576649])]
```

拼接分区结果：

```
np.concatenate(res.part)
array([-1.65326236,  0.43284381, -0.83326654,  1.65616548,  0.47393997, -1.20594265, -1.09926439,
       -0.24483374, -0.58399159,  2.91984624, -1.22159275,  2.99167581,  0.04907979,  0.0052652 ,
       -1.78033393, -1.03704719,  1.27253296,  0.02802034,  0.88785453,  0.03485997,  1.45756404,
       -1.26733862,  0.89596327, -0.65027561,  1.24724115,  0.01338989, -0.45613776,  1.06057673,
        0.33513193,  0.30420455,  2.28403732,  0.31327091,  0.97450361, -2.19087935, -1.38414658,
       -2.06428804, -1.19693768, -2.20837397,  1.79393855,  0.37941031, -1.8306849 ,  0.81135346,
        0.85635656, -0.59189308, -0.58993783,  0.8592545 ,  0.20665878, -2.07373867,  0.23232755,
       -2.69748044,  0.9836457 ,  2.12782845,  0.17228866, -1.42418964, -0.66160031,  0.20736295,
       -0.4235236 , -1.83096434,  0.75557361, -1.87660252,  1.19285543, -0.22831212, -0.75495698,
        1.04599886, -0.59922233, -2.14049959, -0.68492885,  0.13322687,  0.11576229, -1.07628444,
```

```
-1.93437101, -0.51802806,  0.70099126, -2.27776847, -0.17137845, -0.77013423, -0.33715737,
-0.46569988, -0.22885317,  0.07744686,  0.98308645,  0.65965705,  1.30432399, -3.05410973,
-1.55812232, -0.35166336, -0.26695394,  1.7173679 ,  1.42907711,  0.74512261, -1.17590882,
 1.28153159,  0.34006666,  1.19694819,  1.68260023, -2.70844726, -0.21291761,  2.74992875,
-2.1979517 ,  0.60576649])
```

```
sc.stop()
```

## lec5-numpy

**Numpy 基础**

如遇到不清楚的函数或主题，可以查阅官方文档、阅读教程或利用搜索引擎寻求帮助。

**1、矩阵和向量**

首先导入 Numpy 包：

```
import numpy as np
```

利用 Numpy 可以方便地创建向量和矩阵：

```
vec = np.array([1.0, 2.0, 5.0])
print(vec)
```

```
[1. 2. 5.]
```

```
[1.0, 2.0, 5.0] + [1.0, 2.0, 5.0]
```

```
[1.0, 2.0, 5.0, 1.0, 2.0, 5.0]
```

```
mat = np.array([[1.0, 2.0, 2.0], [3.0, 5.0, 4.5]])
print(mat) # R 语言不指定按列排列，Python 按行排列
```

```
[[1.  2.  2. ]
 [3.  5.  4.5]]
```

```
vec = np.linspace(start=1.0, stop=5.0, num=12)
print(vec) # 生成等距点，用这个函数
```

```
[1.         1.36363636 1.72727273 2.09090909 2.45454545 2.81818182
 3.18181818 3.54545455 3.90909091 4.27272727 4.63636364 5.        ]
```

```
mat = np.reshape(vec, (3, 4))
print(mat) # 按行进行优先排列
```

```
[[1.         1.36363636 1.72727273 2.09090909]
 [2.45454545 2.81818182 3.18181818 3.54545455]
 [3.90909091 4.27272727 4.63636364 5.        ]]
```

```
vec.reshape(3, 4) #调用方法不同
```

```
array([[1.        , 1.36363636, 1.72727273, 2.09090909],
       [2.45454545, 2.81818182, 3.18181818, 3.54545455],
       [3.90909091, 4.27272727, 4.63636364, 5.        ]])
```

一些特殊的向量和矩阵有专用的创建方法。

**1-向量/矩阵**

```
np.ones(5)
```

```
array([1., 1., 1., 1., 1.])
```

```
np.ones((3, 2))
```

```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

**0-向量/矩阵**

```
np.zeros(3)
```

```
array([0., 0., 0.])
```

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

**单位阵：**

```
np.eye(5)
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

**对角矩阵：**

```
d = np.array([1, 3, 5])
np.diag(d)
```

```
array([[1, 0, 0],
       [0, 3, 0],
       [0, 0, 5]])
```

Python 中下标是从 0 开始的，请一定要记住这一点，否则会造成很多逻辑错误。

```
print(vec[0])
print(vec[2])
print(mat[1, 1])
```

```
1.0
1.7272727272727273
2.8181818181818183
```

负数的下标表示从尾部往前数：

```
print(vec[-1])
print(vec[-2])
print(mat[1, -1])
```

```
5.0
4.636363636363637
3.5454545454545454
```

可以用冒号选取向量中的一段范围，格式为 x[start:end]，选出的元素包含 x[start]，不包含 x[end]。

```
print(vec[1:3])
```

```
[1.36363636 1.72727273]
```

```
print(mat[:, :2])
```

```
[[1.         1.36363636]
```

```
[2.45454545 2.81818182]
[3.90909091 4.27272727]]
```

在编写函数时，经常需要各种测试数据，**此时可以用 Numpy 来生成各类随机数**。在需要用到随机数之前，一定要先设置随机数种子，以使结果可重复。

```
np.random.seed(123)
```

生成**均匀分布随机数**:

np.random.uniform(low,high,size)

```
unif = np.random.uniform(low=0.0, high=1.0, size=5)
print(unif)
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897]
```

**正态分布随机数**: scale 标准差

```
norm = np.random.normal(loc=0.0, scale=1.0, size=(2, 5))
print(norm)
[[ 0.32210607 -0.05151772 -0.20420096  1.97934843 -1.61930007]
 [-1.11396442 -0.44744072  1.66840161 -0.14337247 -0.6191909 ]]
```

Numpy 提供了许多数学函数对向量和矩阵进行操作:

```
print(unif - 0.5)
[ 0.19646919 -0.21386067 -0.27314855  0.05131477  0.21946897]
```

```
print(norm * 2)
[[ 0.64421214 -0.10303544 -0.40840193  3.95869687 -3.23860013]
 [-2.22792883 -0.89488144  3.33680322 -0.28674495 -1.2383818 ]]
```

```
print(np.exp(norm))
[[1.38003115 0.94978682 0.81529851 7.23802539 0.19803726]
 [0.32825504 0.63926211 5.30368367 0.86643129 0.53837986]]
```

```
print(np.log(unif))
[-0.36173173 -1.2512764  -1.48345987 -0.59544936 -0.32924188]
```

也可以对向量和矩阵进行**汇总**:

```
np.sum(unif)
2.4802437129808985
```

```
np.mean(norm)
-0.02291311494411293
```

汇总可以按行或者按列进行，这由 axis 参数决定。0 表示运算时第一个维度（行）在变化，1 表示运算时第二个维度（列）在变化。再次提醒，python 中以 0 表示第一个元素！

```
np.mean(norm, axis=0)  # 对第一个维度（行标在变化）求均值
array([-0.39592917, -0.24947922,  0.73210032,  0.91798798, -1.11924548])
```

```
np.var(norm, axis=1)  # 对第二个维度（列标在变化）求方差
array([1.33033757, 0.90853666])
```

Numpy 中还有一种常见的操作称为**广播**（教程），即矩阵和向量进行逐元素运算时，向量可以自动扩展以匹配矩阵的维度。考虑如下矩阵和向量:

```
mat = np.arange(12).reshape(3, 4)
mat
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
v1 = np.arange(3)
v1
array([0, 1, 2])
```

```
v2 = np.arange(4)
v2
array([0, 1, 2, 3])
```

将 v1 加到 mat 的每一列:

```
v1.reshape(3, 1)
array([[0],
       [1],
       [2]])
```

```
mat + v1.reshape(3, 1)
array([[ 0,  1,  2,  3],
       [ 5,  6,  7,  8],
       [10, 11, 12, 13]])
```

将 v2 加到 mat 的每一行:

```
mat + v2.reshape(1, 4)
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14]])
```

对行操作时，也可以不用 reshape()直接进行运算:

```
mat + v2
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14]])
```

**练习**

(a) 生成 10000 个服从(0, 1)间均匀分布的随机数，赋值给变量 x，并打印其前 10 个元素。

```
x = np.random.uniform(low=0.0, high=1.0, size=10000)
x[0:10]
array([0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759,
       0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338])
```

(b) 创建向量 y，令其在数学上等于 y=-log(x)，其中 log 为自然对数。打印 y 的最后 10 个元素。

```
y = np.log(x)
```

```
y = y * (-1)
y[-11:-1]
array([0.84335741, 3.99163798, 2.31756265, 1.87519688, 0.570267  ,
       0.93833598, 0.71646581, 0.32602648, 1.04784042, 0.30370373])
```

(c) 查找在 Python 中绘制图形的方法，绘制 y 的直方图。

(d) 猜测或证明 y 服从什么分布，并简要说明理由。

**2、线性代数操作**

**矩阵转置$A^T$**:

```
np.random.seed(123)
A = np.random.normal(size=(5, 3))
A
array([[-1.0856306 ,  0.99734545,  0.2829785 ],
       [-1.50629471, -0.57860025,  1.65143654],
       [-2.42667924, -0.42891263,  1.26593626],
       [-0.8667404 , -0.67888615, -0.09470897],
       [ 1.49138963, -0.638902  , -0.44398196]])
```

```
A.transpose()
array([[-1.0856306 , -1.50629471, -2.42667924, -0.8667404 ,  1.49138963],
       [ 0.99734545, -0.57860025, -0.42891263, -0.67888615, -0.638902  ],
       [ 0.2829785 ,  1.65143654,  1.26593626, -0.09470897, -0.44398196]])
```

**矩阵乘法$B = A^T A$**

```
B = np.matmul(A.transpose(), A)
B
array([[12.31177166,  0.46519338, -6.44684349],
       [ 0.46519338,  2.3825244 , -0.86831276],
       [-6.44684349, -0.86831276,  4.61600385]])
```

**$Bx$:**

```
x = np.random.normal(size=3)
Bx = np.matmul(B, x)
Bx
array([-18.4193173 ,  3.15481065, 10.97897045])
```

也可以调用对象自己的 dot()函数

```
B = A.transpose().dot(A)
# B = A.T.dot(A)
Bx = B.dot(x)
print(B)
print()
print(Bx)
[[12.31177166  0.46519338 -6.44684349]
 [ 0.46519338  2.3825244  -0.86831276]
 [-6.44684349 -0.86831276  4.61600385]]

[-18.4193173   3.15481065 10.97897045]
```

**注意**：如果最终目的是计算$A^T Ax$，应先计算$Ax$，再计算$A^T(Ax)$。为什么？

**核心准则 1：**多个矩阵相乘（向量也视为矩阵），优先进行维度较小的操作

**原理**：矩阵$A_{m*n}$和$B_{n*p}$相乘，计算复杂度为$O(mnp)$

**核心准则 2**：不到万不得已，不要算矩阵的逆！

**原理**：很多牵涉到矩阵求逆的运算，其实可以归结为解线性方程组$x = A^{-1}b => Ax = b$

解$Ax = b$要比先算$A^{-1}$再乘以$b$高效的多

**矩阵求逆**:

```
Binv = np.linalg.inv(B)
Binv
array([[0.3272113 , 0.11021878, 0.47772586],
       [0.11021878, 0.48774192, 0.24568345],
       [0.47772586, 0.24568345, 0.93005856]])
```

```
Binv.dot(B)
array([[ 1.00000000e+00,  4.07425102e-19, -5.08361378e-16],
       [ 1.08233492e-16,  1.00000000e+00, -3.97109825e-17],
       [ 9.79113472e-16, -4.49596903e-18,  1.00000000e+00]])
```

注意：实际计算中通常不需要显式地求逆，而应尽可能转化成解线性方程组。

解线性方程组$Bx = d$，即计算$x = B^{-1}d$

```
d = np.random.normal(size=3)
x = np.linalg.solve(B, d)
x
array([0.72336281, 0.48018414, 1.26033807])
```

验证解的正确性:

```
print(d)
print(B.dot(x))
[1.0040539  0.3861864  0.73736858]
[1.0040539  0.3861864  0.73736858]
```

求解**特征值**和**特征向量**:

```
evals, evecs = np.linalg.eigh(B)
print(evals)
print()
print(evecs)
[ 0.77881759  2.50876381 16.02271851]

[[-0.45250424 -0.20887627 -0.86695479]
 [-0.31951962  0.94561067 -0.06105468]
 [-0.83255458 -0.24938156  0.49463291]]
```

```
def compute(X, y):
    return X.dot(np.linalg.inv(X.T.dot(X))).dot(X.T).dot(y)
```

## lec5-numerical 数值计算基础

**1、原则 1：矩阵相乘，小维度优先**

**核心准则 1：**多个矩阵相乘（向量也视为矩阵），优先进

行维度较小的操作

**原理**：矩阵$A_{m*n}$和$B_{n*p}$相乘，计算复杂度为$O(mnp)$

**矩阵$A_{n*p}$，向量$x_{p*1}$，计算$A^TAx$**

```
import numpy as np
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 2000
p = 1000
A = np.random.normal(size=(n, p))
x = np.random.normal(size=p)
```

**方法 1**：先计算$A^TA$，再与$x$相乘

```
%timeit A.transpose().dot(A).dot(X) # O(pnp + pp)
```
*136 ms ± 13.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)*

**方法 2**：先计算$Ax$，再左乘$A^T$

```
%timeit A.transpose().dot(A.dot(x)) # O(2np)
```
*4.12 ms ± 364 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)*

经验法则：对于更一半的矩阵乘法$A_{m*n}B_{n*p}C_{p*r}$，如果$n \approx p$且$m > r$，则优先计算$BC$，反之优先计算$AB$

```
np.random.seed(123)
m = 1000
n = 500
p = 200
r = 100
A = np.random.normal(size=(m, n))
B = np.random.normal(size=(n, p))
C = np.random.normal(size=(p, r))
```

```
%timeit A.dot(B).dot(C)
```
*18 ms ± 714 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)*

```
%timeit A.dot(B.dot(C))
```
*12.1 ms ± 808 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)*

**2、原则 2：尽量避免显式矩阵求逆**

**核心准则 2**：不到万不得已，不要算矩阵的逆！

**原理**：很多牵涉到矩阵求逆的运算，其实可以归结为解线性方程组$x = A^{-1}b => Ax = b$

解$Ax = b$要比先算$A^{-1}$再乘以$b$高效得多

**矩阵$A_{n*n}$，向量$b_{n*1}$，计算$A^{-1}b$**

```
np.random.seed(123)
n = 1000
A = np.random.normal(size=(n, n))
b = np.random.normal(size=n)
```

**方法 1**：先计算$A^{-1}$，再与$b$相乘

```
%timeit np.linalg.inv(A).dot(b)
```
*474 ms ± 74.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)*

**方法 2**：解线性方程组$Ax = b$

```
%timeit np.linalg.solve(A, b)
```
*257 ms ± 51.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)*

线性方程组右边也可以是矩阵，即$A_{n*n}$，$B_{n*p}$，计算$A^{-1}B$

```
np.random.seed(123)
n = 1000
p = 100
A = np.random.normal(size=(n, n))
B = np.random.normal(size=(n, p))
```

```
%timeit np.linalg.inv(A).dot(B)
```
*387 ms ± 39.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)*

```
%timeit np.linalg.solve(A, B)
```
*259 ms ± 48.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)*

**3、原则 3：利用矩阵的特殊结构**

**核心准则 3：**尽可能利用矩阵的特殊结构来减少计算量

例如：

计算$WA$，其中$A_{n*p}$，$W_{n*n}$是一个对角矩阵

$WA$相当于将$W$的对角线元素乘到$A$的每一列，避免了直接的矩阵乘法

类似地，如果需要计算$AW$，其中$A_{n*p}$，$W_{p*p}$是一个对角矩阵

只需将$W$的对角线元素乘到$A$的每一行

**矩阵$A_{n*n}$，对角矩阵$W_{n*n}$，计算$WA$和$AW$**

```
np.random.seed(123)
n = 1000
A = np.random.normal(size=(n, n))
w = np.random.normal(size=n)
W = np.diag(w)
```

```
%timeit W.dot(A)
```
*37 ms ± 646 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)*

```
%timeit A.dot(W)
```
*33 ms ± 713 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)*

令$w_{n*1}$表示$W$的对角元素，$WA$相当于将$A$的每一列乘以$w$，$AW$相当于将$A$的每一行乘以$w^T$。此时可利用 Numpy 的广播机制进行运算。

```
%timeit A * w.reshape(n, 1)
```
*5.15 ms ± 508 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)*

```
%timeit A *w # 默认行向量，并非矩阵和向量乘法，利用广播
```
*5.05 ms ± 261 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)*

**核心准则 4**：尽可能将多个向量运算的循环合并为矩阵运算

$(Ax_1, Ax_2, \dots, Ax_p) => AX$，其中$X = (x_1, x_2, \dots, x_p)$

$A \in R^{m*n}, x_i \in R^n, X \in R^{n*p}$

**原理**：虽然理论上的计算复杂度相同，但软件层面对矩阵间的运算往往优化更好

**4、练习：回归分析**

回归分析中给出了模型的预测公式，计算

$$\hat{y} = X(X^T X)^{-1} X^T y$$

假设$X_{n*p}$，$n > p$，应如何编写程序进行计算？

## lec6-distributed 基础分布式计算

1) **分布式算法**

a. 矩阵乘法

b. 解线性方程组

c. 线性模型

d. 岭回归

e. Logistic 回归

f. 梯度下降法

g. 牛顿法

2) **矩阵乘法**

$Xv$

$X'X$

$X'v$

**1、准备工作**

**配置和启动 PySpark:**

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
#  本地模式
spark = SparkSession.builder.\
        master("local[*]").\
        appName("PySpark RDD").\
        getOrCreate()
sc = spark.sparkContext
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

```
<pyspark.sql.session.SparkSession object at 0x000002076D986040>

<SparkContext master=local[*] appName=PySpark RDD>
```

**利用 Numpy 创建一个矩阵，并写入文件：**

```
import numpy as np
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 100
p = 5
mat = np.random.normal(size=(n, p))
np.savetxt("data/mat_np.txt", mat, fmt="%f", delimiter="\t")
```

**PySpark 读取文件并进行一些简单操作：**

```
file = sc.textFile("data/mat_np.txt")

#  打印矩阵行数
print(file.count())

#  空行
print()

#  打印前 5 行
text = file.take(5)
print(*text, sep="\n")
```

```
100


-1.085631	0.997345	0.282978	-1.506295	-0.578600
1.651437	-2.426679	-0.428913	1.265936	-0.866740
-0.678886	-0.094709	1.491390	-0.638902	-0.443982
-0.434351	2.205930	2.186786	1.004054	0.386186
0.737369	1.490732	-0.935834	1.175829	-1.253881
```

```
file.first()
```

```
'-1.085631\t0.997345\t0.282978\t-1.506295\t-0.578600'
```

**2、进行分区映射（MapPartitions）**

```
file_p10 = file.repartition(10)
print(file.getNumPartitions())
print(file_p10.getNumPartitions())
```

```
2
10
```

```
# str => np.array
def str_to_vec(line):
    #  分割字符串
    str_vec = line.split("\t")
    #  将每一个元素从字符串变成数值型
    num_vec = map(lambda s: float(s), str_vec)
    #  创建 Numpy 向量
    return np.fromiter(num_vec, dtype=float)

# Iter[str] => Iter[matrix]
def part_to_mat(iterator):
    # Iter[str] => Iter[np.array]
    iter_arr = map(str_to_vec, iterator)

    # Iter[np.array] => list(np.array)
    dat = list(iter_arr)

    # list(np.array) => matrix
    if len(dat) < 1:   # Test zero iterator
        mat = np.array([])
```

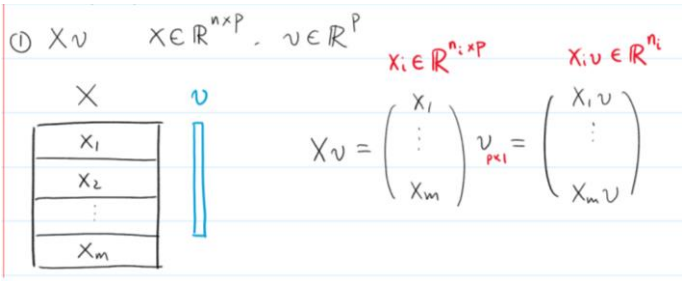
```
    else:
        mat = np.vstack(dat)

    # matrix => Iter[matrix]
    yield mat
```

```
dat = file_p10.mapPartitions(part_to_mat).filter(lambda x: x.shape[0] > 0)
print(dat.count())
```

*7*

### 3、矩阵乘法$Xv$




**模拟数据和真实值：**

```
np.random.seed(123)
v = np.random.uniform(size=p)
res = mat.dot(v)
res
```

*array([-1.65326187, 0.43284335, -0.83326669, 1.65616556, 0.47393998, -1.20594195, -1.09926452, -0.24483357, -0.58399139, 2.91984625, -1.22159268, 2.99167578, 0.04907967, 0.00526486, -1.78033411, -1.03704672, 1.27253333, 0.0280204 , 0.88785436, 0.03485989, 1.45756374, -1.26733834, 0.89596346, -0.65027554, 1.24724097, 0.01338995, -0.45613812, 1.06057634, 0.33513133, 0.30420446, -1.8306843 , 0.81135409, 0.8563569 , -0.59189289, -0.58993733, 0.85925493, 0.20665867, -2.07373852, 0.23232788, -2.69748055, 1.19285523, -0.22831252, -0.75495708, 1.04599886, -0.59922216, -2.14049979, -0.68492854, 0.13322705, 0.11576237, -1.07628496, 0.98308603, 2.28403745, 0.31327103, 0.97450293, -2.19087869, -1.38414598, -2.06428815, -1.19693787, -2.20837322, 1.79393849, 0.37940968, 0.98364566, 2.12782768, 0.17228872, -1.42418937, -0.66160026, 0.20736396, -0.42352417, -1.83096405, 0.75557361, -1.87660221, -1.93437067, -0.51802796, 0.70099077, -2.27776851, -0.17137795, -0.77013413, -0.33715716, -0.46570004, -0.22885299, 0.07744646, 0.65965659, 1.30432415, -3.05410919, -1.55812228, -0.35166363, -0.26695372, 1.71736731, 1.42907711, 0.74512303, -1.17590892, 1.28153134, 0.34006662, 1.1969479 , 1.68259996, -2.70844742, -0.21291717, 2.74992919, -2.1979523 , 0.60576651])*

**每个 RDD 分区上进行计算：**

```
res_part = dat.map(lambda x: x.dot(v)).collect()
res_part
```

*[array([-1.65326236, 0.43284381, -0.83326654, 1.65616548, 0.47393997, -1.20594265, -1.09926439, -0.24483374, -0.58399159, 2.91984624]),*

*array([-1.22159275, 2.99167581, 0.04907979, 0.0052652 , -1.78033393, -1.03704719, 1.27253296, 0.02802034, 0.88785453, 0.03485997]),*

*array([ 1.45756404, -1.26733862, 0.89596327, -0.65027561, 1.24724115, 0.01338989, -0.45613776, 1.06057673, 0.33513193, 0.30420455, 2.28403732, 0.31327091, 0.97450361, -2.19087935, -1.38414658, -2.06428804, -1.19693768, -2.20837397, 1.79393855, 0.37941031]),*

*array([-1.8306849 , 0.81135346, 0.85635656, -0.59189308, -0.58993783, 0.8592545 , 0.20665878, -2.07373867, 0.23232755, -2.69748044, 0.9836457 , 2.12782845, 0.17228866, -1.42418964, -0.66160031, 0.20736295, -0.4235236 , -1.83096434, 0.75557361, -1.87660252]),*

*array([ 1.19285543, -0.22831212, -0.75495698, 1.04599886, -0.59922233, -2.14049959, -0.68492885, 0.13322687, 0.11576229, -1.07628444, -1.93437101, -0.51802806, 0.70099126, -2.27776847, -0.17137845, -0.77013423, -0.33715737, -0.46569988, -0.22885317, 0.07744686]),*

*array([ 0.98308645, 0.65965705, 1.30432399, -3.05410973, -1.55812232, -0.35166336, -0.26695394, 1.7173679 , 1.42907711, 0.74512261, -1.17590882]),*

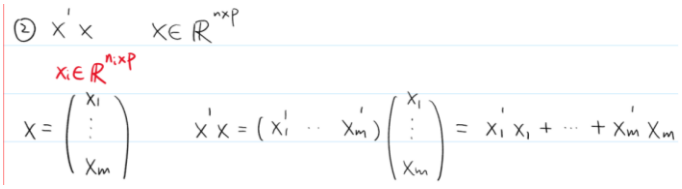
*array([ 1.28153159, 0.34006666, 1.19694819, 1.68260023, -2.70844726, -0.21291761, 2.74992875, -2.1979517 , 0.60576649])]*

**拼接分块结果**:

```
np.concatenate(res_part)
```

*array([-1.65326236, 0.43284381, -0.83326654, 1.65616548, 0.47393997, -1.20594265, -1.09926439, -0.24483374, -0.58399159, 2.91984624, -1.22159275, 2.99167581, 0.04907979, 0.0052652 , -1.78033393, -1.03704719, 1.27253296, 0.02802034, 0.88785453, 0.03485997, 1.45756404, -1.26733862, 0.89596327, -0.65027561, 1.24724115, 0.01338989, -0.45613776, 1.06057673, 0.33513193, 0.30420455, 2.28403732, 0.31327091, 0.97450361, -2.19087935, -1.38414658, -2.06428804, -1.19693768, -2.20837397, 1.79393855, 0.37941031, -1.8306849 , 0.81135346, 0.85635656, -0.59189308, -0.58993783, 0.8592545 , 0.20665878, -2.07373867, 0.23232755, -2.69748044, 0.9836457 , 2.12782845, 0.17228866, -1.42418964, -0.66160031, 0.20736295, -0.4235236 , -1.83096434, 0.75557361, -1.87660252, 1.19285543, -0.22831212, -0.75495698, 1.04599886, -0.59922233, -2.14049959, -0.68492885, 0.13322687, 0.11576229, -1.07628444, -1.93437101, -0.51802806, 0.70099126, -2.27776847, -0.17137845, -0.77013423, -0.33715737, -0.46569988, -0.22885317, 0.07744686, 0.98308645, 0.65965705, 1.30432399, -3.05410973, -1.55812232, -0.35166336, -0.26695394, 1.7173679 , 1.42907711, 0.74512261, -1.17590882, 1.28153159, 0.34006666, 1.19694819, 1.68260023, -2.70844726, -0.21291761, 2.74992875, -2.1979517 , 0.60576649])*

### 4、矩阵乘法$X^TX$




**真实值：**

```
res = mat.transpose().dot(mat)
res
```

*array([[ 9.35643395e+01, -2.39739879e-02, 4.75846887e+00, 2.27729644e+01, 5.35952824e+00], [-2.39739879e-02, 1.09769750e+02, 2.74564778e+00, -3.29329848e-01, 1.11698743e+01], [ 4.75846887e+00, 2.74564778e+00, 1.09077973e+02, 6.41825678e+00, -7.53446301e+00], [ 2.27729644e+01, -3.29329848e-01, 6.41825678e+00, 9.95562632e+01, 7.71274621e+00], [ 5.35952824e+00, 1.11698743e+01, -7.53446301e+00, 7.71274621e+00, 9.19863380e+01]])*

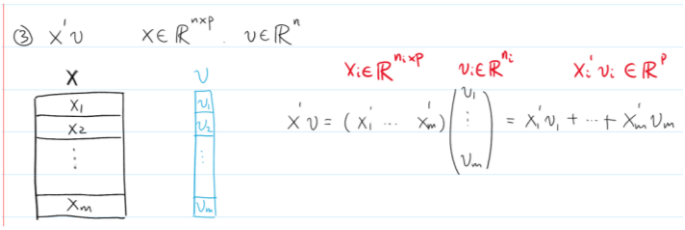
**每个 RDD 分区上进行计算：**

```
res = dat.map(lambda x: x.transpose.dot(x)).reduce
(lambda x, y: x + y)
res
```

*array([[ 9.35643453e+01, -2.39794314e-02, 4.75847395e+00, 2.27729716e+01, 5.35953520e+00], [-2.39794314e-02, 1.09769741e+02, 2.74564629e+00, -3.29332977e-01, 1.11698788e+01], [ 4.75847395e+00, 2.74564629e+00, 1.09077969e+02, 6.41825316e+00, -7.53445385e+00], [ 2.27729716e+01, -3.29332977e-01, 6.41825316e+00, 9.95562607e+01, 7.71275158e+00],*

```
[ 5.35953520e+00,  1.11698788e+01, -7.53445385e+00,  7.71275158e+00,  9.19863445e+01]]
```

**5、矩阵乘法$X^Tv$**



**以 mat 的前 4 列为 X，最后一列为 v：**

```
X = mat[ :, :-1]
v = mat[ :, -1]
res = X.transpose().dot(v)
res
```

```
array([ 5.35952824, 11.1698743 , -7.53446301,  7.71274621])
```

**每个 RDD 分区上进行计算：**

```
def Xitv(part) :
    Xi = part[:, :-1]
    vi = part[:, -1]
    return Xi.transpose().dot(vi)
res = dat.map(Xitv).reduce(lambda x, y: x + y)
res
```

```
array([ 5.3595352 , 11.16987882, -7.53445385,  7.71275158])
```

**关闭 Spark 连接：**

```
sc.stop()
```

## lec7-matprod 分布式矩阵乘法

同上一节

## lec7-regression 分布式回归模型

a. 考虑回归问题$y = \beta_0 + \beta' x + \varepsilon$

$n \gg p$

回归系数估计值的表达式$\hat{\beta} = (X'X)^{-1}X'Y$

注意还需考虑截距项

b. 当$p$不太大时

$X'X$和$X'Y$都可以装进内存

a) 从原始数据生成 RDD

b) 分别计算$X'X$和$X'Y$

c) 解线性方程组，得到$\hat{\beta} = (X'X)^{-1}X'Y$

c. 首先了解数据的**存储格式**

a) 有没有表头?

b) 数据按什么分隔?

c) $Y$和$X$的位置如何?

d. 其他细节

a) 要保证$X$和$Y$始终处在同一个 RDD 中

b) 将$X$和$Y$作为一个整体进行 RDD 分区

c) 添加截距项（如何操作？）

**1、准备工作**

配置和启动 PySpark :

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
#  本地模式
spark = SparkSession.builder.\
        master("local[*]").\
        appName("PySpark RDD").\
        getOrCreate()
sc = spark.sparkContext
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

```
<pyspark.sql.session.SparkSession object at 0x000001CC5E267040>

<SparkContext master=local[*] appName=PySpark RDD>
```

利用 Numpy 生成模拟数据，并写入文件。

首先生成$n \gg p$的数据

**np.set_printoptions** 是 numpy 中的一个函数，用来设置 numpy 数组的打印选项。例如，可以使用该函数设置打印数组时的精度、打印的行数和列数、是否使用科学计数法等。

```
import numpy as np
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 100000
p=100
x1 = np.random.normal(size=(n, p))
beta1 = np.random.normal(size=p)
y1 = x1.dot(beta1) + np.random.normal(scale=0.1, size=n)
dat = np.hstack((y1.reshape(n, 1), x1))
np.savetxt("data/reg_tall.txt", dat, fmt = "%f", delimiter = "\t")
```

以及$n < p$的数据:

```
import numpy as np
n = 500
p = 5000
x2 = np.random.normal(size=(n, p))
beta2 = np.random.normal(size=p)
beta2[10:]=0.0
y2 = x2.dot(beta2) +np.random.normal(scale=0.1, size=n)
dat = np.hstack((y2.reshape(n, 1), x2))
np.savetxt("data/reg_wide.txt", dat, fmt = "%f", delimiter = "\t")
```

PySpark 读取文件并进行一些简单操作:

```
file1 = sc.textFile("data/reg_tall.txt")
```

```
# 打印矩阵行数
print(file1.count())

# 空行
print()

# 打印前 5 行，并将每行字符串截尾
text = file1.map(lambda x: x[:70] + "…").take(5)
print(*text, sep="\n")
```

```
100000

-0.492572  -1.085631  0.997345   0.282978   -1.506295  -0.578600  1.651437   -2…
11.643889  0.642055   -1.977888  0.712265   2.598304   -0.024626  0.034142   0.17…
-11.441109 0.703310   -0.598105  2.200702   0.688297   -0.006307  -0.206662  -0…
0.618083   0.765055   -0.828989  -0.659151  0.611124   -0.144013  1.316606   -0.7…
-8.438569  1.534090   -0.529914  -0.490972  -1.309165  -0.008660  0.976813   -1…
```

```
file2 = sc.textFile("data/reg_wide.txt")

# 打印矩阵行数
print(file2.count())

# 空行
print()

# 打印前 5 行，并将每行字符串截尾
text = file2.map(lambda x: x[:70] + "…").take(5)
print(*text, sep="\n")
```

```
500

-1.596108  0.517731   0.702122   0.615923   0.732042   -0.782366  -0.246075  0.57…
2.762893   -0.094939  1.242466   -0.805527  0.050097   -1.616350  -1.434936  -0…
2.324372   -0.538700  -0.204445  -0.380780  -0.073454  1.104704   -0.752930  -0…
-0.692888  -1.293058  0.155704   -0.046255  0.023168   -0.860021  -1.061860  0…
-3.841189  0.544471   -0.028014  -0.250308  0.499110   -0.410225  0.103303   -0…
```

**2、$n \gg p$**

回归系数估计值的显式解为$\hat{\beta} = (X'X)^{-1}X'Y$。当$n \gg p$且$p$不太大时，$X'X$为$p * p$矩阵，$X'Y$为$p * 1$向量，均可放入内存。因此，此时问题的核心在于计算$X'X$与$X'Y$

首先进行分区映射：

```
file_p10 = file1.repartition(10)
print(file_p10.getNumPartitions())
```

```
10
```

```
# str => np.array
def str_to_vec(line):
    # 分隔字符串
    str_vec = line.split("t ")
    # 将每一个元素从字符串变成数值型
    num_vec = map(lambda s: float(s), str_vec)
    # 创建 Numpy 向量
    return np.fromiter(num_vec, dtype=float)

# Iter[str] => Iter[matrix]
def part_to_mat(iterator):
    # Iter[str]=> Iter[np.array]
    iter_arr = map(str_to_vec, iterator)
    # 可以选择在这里加一列 1，或者后续 map 时进行转换
    # Iter[np.array] => list(np.array)
    dat = list(iter_arr)

    # list(np.array) => matrix
    if len(dat) < 1: # Test zero iterator
        mat = np.array([])
    else:
        mat = np.vstack(dat)

    # matrix => Iter[matrix]
    yield mat
```

```
dat = file_p10.mapPartitions(part_to_mat).filter(lambda x: x.shape[0] > 0)
print(dat.count())
```

```
10
```

```
dat.first()
```

```
array([[ 1.3881828e+01,  1.1315380e+00, -3.2151000e-01, ...,  1.4565320e+00,
         1.0468540e+00, -4.0916600e-01],
       [ 1.2336276e+01,  1.9662150e+00,  9.3850400e-01, ..., -4.5653900e-01,
        -1.3955400e-01, -6.9250800e-01],
       [-1.7039820e+00, -5.0162800e-01,  4.7969000e-02, ..., -4.9629100e-01,
        -6.4533200e-01, -1.2678130e+00],
       ...,
       [-8.9065730e+00,  4.3269400e-01, -9.6707000e-01, ...,  4.0304000e-02,
         2.5447400e-01, -4.0170000e-03],
       [ 5.3757290e+00,  6.7905000e-01, -1.6643800e+00, ..., -1.4532600e-01,
        -5.7877000e-02,  1.0376050e+00],
       [-3.1090000e-01,  1.3800060e+00, -6.6354200e-01, ...,  7.0082800e-01,
         1.0642810e+00, -5.6755600e-01]])
```

注意此时每个分区上的数据同时包含了因变量和自变量，在使用自变量时，要将第一列排除。计算$X'X$

```
xtx = dat.map (lambda part : part[ :,  1:]. transpose(). dot(part[ :, 1 :])). reduce(lambda x, y : x + y)
xtx
```

```
array([[ 9.92814274e+04,  5.64799792e+02,  4.68363424e+01, ...,
         1.31821863e+02, -1.81293885e+01,
```

```
        2.77125486e+02],
       [ 5.64799792e+02,   1.00090223e+05,   4.34513678e+02, ...,   3.22355415e+02,   1.98867239e+02,
        9.69607877e+01],
       [ 4.68363424e+01,   4.34513678e+02,   9.92600734e+04, ...,  -2.01876920e+02,  -6.97509673e+02,
        2.97558656e+02],
       ...,
       [ 1.31821863e+02,   3.22355415e+02,  -2.01876920e+02, ...,   9.98741108e+04,   6.09301190e+01,
        1.44257513e+02],
       [-1.81293885e+01,   1.98867239e+02,  -6.97509673e+02, ...,   6.09301190e+01,   9.99329830e+04,
       -2.53690380e+02],
       [ 2.77125486e+02,   9.69607877e+01,   2.97558656e+02, ...,   1.44257513e+02,  -2.53690380e+02,
        9.95605285e+04]])
```

计算$X'Y$:

```
xty = dat.map(lambda part : part [ :, 1 :]. transpose(). dot(part[ :, 0])).reduce(lambda x, y : x + y)
xty
```

```
array([-5.72523147e+04, -1.13445442e+05,  1.14296626e+05,  6.70903227e+04,  6.40148536e+04,
       -1.73288654e+05,  9.65899313e+04, -7.04664986e+04,  1.17713274e+05, -1.24568031e+04,
       -5.50355360e+04,  4.50494339e+04,  1.53412736e+05,  9.03011121e+04,  3.68938036e+04,
       -6.74204369e+04, -1.38999203e+04,  1.03492109e+05,  1.00816430e+05, -2.49288673e+04,
        2.76828665e+04, -1.96876397e+05,  9.23669947e+04, -4.36091041e+04, -1.29989914e+05,
       -1.15080876e+05, -9.68461454e+04, -4.38833665e+04, -3.92874942e+04,  7.50460497e+03,
        6.75428356e+04,  4.49467215e+04,  7.90760279e+04,  1.28488448e+04, -1.64995344e+04,
        1.29426796e+05, -8.84459583e+04, -1.54233257e+05,  1.23672995e+03,  5.52665865e+03,
        1.74929996e+02, -3.51956381e+04, -1.75937385e+05, -1.33574238e+05, -1.57921700e+05,
       -1.29883192e+05,  9.35018217e+04,  8.71584103e+04, -9.00909788e+04, -1.11506523e+05,
       -6.16019121e+04, -1.57272495e+05, -1.46453819e+05, -1.40527176e+05,  3.13423142e+03,
       -2.07800766e+05,  2.40037936e+04, -1.48363420e+05, -5.08521863e+04, -1.10487813e+05,
        1.22702672e+05,  7.44173493e+04,  4.46138502e+04, -2.19904994e+04,  1.15480726e+05,
       -2.01607743e+04,  4.16933522e+04,  4.46330339e+04, -2.74800656e+04, -9.79389613e+04,
       -2.56165130e+05,  1.21395300e+05, -3.64900799e+04,  3.66243597e+04,  7.79829668e+04,
        1.61882828e+05, -3.76537010e+04,  4.32999240e+04, -1.41029327e+05, -5.01770196e+04,
       -2.66142200e+04, -1.66461170e+05,  1.08742185e+05,  4.59950832e+04,  2.59650598e+04,
       -6.16208528e+04,  1.23485967e+05, -2.24770490e+05, -5.02194921e+04, -1.13329872e+05,
        1.64669710e+05, -1.33654314e+05, -6.66513826e+03,  2.19806224e+05,  1.43449021e+05,
        5.11936828e+04, -7.39006443e+04,  4.10914977e+04,  2.12544303e+05,  1.74770852e+05])
```

此时剩下的操作即为求解线性方程组。由于$p$较小，故可以在内存中完成:

```
bhat = np.linalg.solve(xtx, xty)
bhat
```

```
array([-0.58422145, -1.11605591,  1.1559347 ,  0.68617539,  0.64534766, -1.70930592,  0.87296172,
       -0.69022365,  1.21031103, -0.18000063, -0.59629703,  0.45240243,  1.55780574,  0.93400416,
        0.33550102, -0.62756266, -0.16682495,  1.03999291,  0.99266921, -0.2220117 ,  0.26884866,
       -1.9555697 ,  0.93141768, -0.46989397, -1.3011378 , -1.08472   , -0.92674644, -0.46976713,
       -0.41262747,  0.09672668,  0.73804542,  0.43592289,  0.78154368,  0.09788819, -0.20673303,
        1.35677125, -0.84339913, -1.57384018, -0.02816233,  0.04550477, -0.00426462, -0.32000893,
       -1.73697639, -1.35751444, -1.61142709, -1.29011551,  0.92229365,  0.92287512, -0.87182444,
       -1.11120148, -0.64178182, -1.54097709, -1.47574519, -1.40014092,  0.05651123, -2.06681374,
        0.2406474 , -1.45930317, -0.4989418 , -1.08579381,  1.22881498,  0.71939479,  0.4744752 ,
       -0.21579098,  1.19156816, -0.18989885,  0.42140091,  0.48960889, -0.28646643, -0.92767184,
       -2.54505269,  1.21356683, -0.40318342,  0.40934314,  0.74640857,  1.5946434 , -0.36000768,
        0.41415481, -1.43610921, -0.47440119, -0.27697483, -1.66229639,  1.15769734,  0.4276756 ,
        0.22067074, -0.59538449,  1.16621113, -2.23499308, -0.4978505 , -1.1065741 ,  1.6025852 ,
       -1.32116701, -0.05669346,  2.21519396,  1.46738066,  0.54739973, -0.73702624,  0.44785602,
        2.14201589,  1.67501715])
```

与真值进行对比:

beta1

```
array([-0.58441387, -1.11534971,  1.15570434,  0.68635474,  0.64559564, -1.70989894,  0.87296263,
       -0.69061605,  1.21015702, -0.17972039, -0.59593691,  0.45252029,  1.55835773,  0.93393225,
        0.33544231, -0.62751865, -0.16601382,  1.03999001,  0.99253053, -0.22189606,  0.26883567,
       -1.95555529,  0.93138627, -0.47006585, -1.30103516, -1.0852571 , -0.92710125, -0.46948194,
       -0.41250335,  0.09711287,  0.73781056,  0.43563306,  0.78146188,  0.09794209, -0.20673932,
        1.3563857 , -0.84334031, -1.57440936, -0.02785942,  0.04491426, -0.00378645, -0.32005209,
       -1.73699327, -1.35755085, -1.61153494, -1.29018054,  0.92198227,  0.92329806, -0.87182087,
       -1.11109253, -0.64117813, -1.54063266, -1.47551246, -1.40012474,  0.05626203, -2.06673395,
        0.24069911, -1.45962908, -0.49910593, -1.08548   ,  1.22895895,  0.71943325,  0.47494861,
       -0.21579343,  1.19116701, -0.19017216,  0.4216233 ,  0.48972018, -0.2864368 , -0.92757763,
       -2.54487881,  1.2137159 , -0.40262445,  0.40946857,  0.74671794,  1.59476786, -0.36007466,
        0.41427829, -1.43588821, -0.47424149, -0.27699872, -1.66246218,  1.15732258,  0.42792354,
        0.22067973, -0.59542409,  1.16612341, -2.23575698, -0.49771051, -1.10650178,  1.60199211,
       -1.32128632, -0.05640532,  2.21514038,  1.46662192,  0.54730866, -0.7372505 ,  0.44819396,
        2.14210279,  1.67452645])
```

思考题：实际计算回归时，我们一般会加入截距项，此时应该如何修改程序，使其可以输出包含截距项的回归系数?

**3、$n < p$**

当$n < p$时，$X'X$不再可逆，最小二乘没有唯一解

首先获取维度信息:

```
n = file2.count()
n
```

```
500
```

```
p = str_to_vec(file2.first()).shape[0] – 1
p
```

```
5000
```

然后创建分区 RDD:

```
dat = file2. repartition(10). mapPartitions(part_to_mat). filter (lambda x: x.shape[0] > 0)
print(dat.count())
```

```
10
```

关闭 Spark 连接:

```
sc.stop()
```

## lec8-cg 共轭梯度法

1) **岭回归**

i. 依然是回归问题$y = \beta_0 + \beta' x + \varepsilon$

ii. $n \ll p$时，$X'X$不可逆

iii. 此时在损失函数上加入惩罚项$\lambda||\beta||^2$

iv. 回归系数估计值的表达式为$\hat{\beta}_\lambda = (X'X + \lambda I)^{-1}X'Y$

v. $\lambda$为一个给定的正数

vi. 注意当$n \ll p$时，$X'X + \lambda I$是一个高维的矩阵，无法直接解线性方程组

vii. 引入共轭梯度法

a. **线性方程组**

a) 考虑线性方程组$Ax = b$

b) 假设$A$是正定矩阵

c) 正定：$A$的特征值都大于 0。

d) 求解$x = A^{-1}b$总共分几步？

$p$步，某种意义上，$p$是$A$的维度（$A$是方阵）

b. **共轭梯度法**

a) 共轭梯度法(Conjugate gradient, CG)是一种解正定线性方程组的方法

b) 它有趣的地方在于，可以通过乘法运算$v \to Av$来得到逆运算结果$A^{-1}b$

c) 更有意思的是，数学上可以证明它在$p$步迭代之后就可能得到精确解

d)

**Target** : solve linear equation $Ax = b$. $A_{m*m}$ is positive definite

**Input**: $A, b, x_0$ (initial guess)

$r_0 \coloneqq b - Ax_0$

$p_0 \coloneqq r_0$

$k \coloneqq 0$

Loop until $k = m$

$\alpha_k \coloneqq \frac{r_k' r_k}{p_k' A p_k}$

$x_{k+1} \coloneqq x_k + \alpha_k p_k$

$r_{k+1} \coloneqq r_k - \alpha_k A p_k$

if $r_{k+1}$ is sufficiently small then exit loop

$\beta_k \coloneqq \frac{r_{k+1}' r_{k+1}}{r_k' r_k}$

$p_{k+1} \coloneqq r_{k+1} + \beta_k p_k$

$k \coloneqq k + 1$

End loop

**Output** : $x_{k+1}$

http://en.wikipedia.org/wiki/Conjugate_gradient_method

c. 适用范围

a) CG 尤其适合矩阵乘法能高效计算的场合

b) 疏矩阵

c) 分布式矩阵

但一定要注意验证正定性

d. 哪些矩阵是正定性？

a) 特征值均大于 0

b) 非退化分布的协方差矩阵

c) $X'X + \lambda I$, $\lambda > 0$

e. 利用 CG 来求解回归问题

https://cosx.org/2016/11/conjugate-gradient-for-regression/

**1、CG 简单实现**

当$n < p$时，$X'X$不再可逆，因此最小二乘法没有唯一解。此时我们可以采用岭回归的方法，其在最小二乘损失函数的基础上加入一个惩罚项$\lambda||\beta||^2$。岭回归估计的显式解为$\hat{\beta}_\lambda = (X'X + \lambda I)^{-1}X'Y$，其中$\lambda > 0$是一个给定的正数。

但注意到$X'X + \lambda I$是一个高维的矩阵，难以直接进行求解。因此我们采用共轭梯度法（参见 lec7-cg.ipynb）

解线性方程组$AX = b$， 其中$A$为正定矩阵

**np.copy** 返回给定对象的数组副本。

```
import numpy as np
np.set_printoptions(linewidth=100)

#https://en.wikipedia.org/wiki/Conjugate_gradient_method
# A [m x m], b [m], x0 [m]  这个 x0 是自己设计投入的
def cg (Afn, b, x0, eps=1e-3, print_progress=False, **Afn_arg):
    m = b.shape[0]
    # 初始解（注意此处应该复制 x0，否则程序退出时会修改 x0）
    x = np.copy(x0)
    # 初始残差向量
    r = b – Afn(x, **Afn_args)
    # 初始共轭梯度
    p = r

    for k in range(m):
        # 矩阵乘法
        Ap = Afn(p, **Afn_args) # 最重点的一步
        rr = r.dot(r)
        alpha = rr / p.dot(Ap)
        # 更新解
        x += alpha * p
        # 计算新残差向量
        rnew = r – alpha * Ap # 这里注意保持 rr 和 rnew 两者，后续需要继续计算，未必需要保存在列表里，可以设置打印装置，节省内存
        # 测试是否收敛
        norm = np.linalg.norm(rnew)
        if print_progress:
            print(f"Iter {k}, residual norm = {norm}")
        if norm < eps:
            break
```

```
        beta = rnew.dot(rnew) / rr
        # 更新共轭梯度
        p = rnew + beta * p # 注意 p 的更新
        # 更新残差向量
        r = rnew  # 这里需要注意，r 的更新
    return x
```

**np.linalg.norm()**用于求范数，linalg 本意为 linear(线性) + algebra(代数)，norm 则表示范数。

**代码简化有很多可学习之处**

https://en.wikipedia.org/wiki/Conjugate_gradient_method

进行简单的测试:

```
np.random.seed(123)
m = 10
x = np.random.normal(size=(m,m))
# A 为正定矩阵
A = x.transpose().dot(x)
b = np.random.normal(size=m)
# 直接求解
sol = np.linalg.solve(A, b)
sol
```

```
array([-136.71894832,  -33.85176812,   41.84149034,   73.42723076,
        12.50311233,  -41.91028953,   -6.01395804,  -30.7294723 ,
       -24.21718752,  -68.80957079])
```

CG 求解:

```
cg(A, b, x0=np.zeros(shape=m), print_progress=True)
```

```
Iter 0, residual norm = 3.411450076020753
Iter 1, residual norm = 5.762891232211009
Iter 2, residual norm = 7.818450458630149
Iter 3, residual norm = 7.264409878364003
Iter 4, residual norm = 10.31938154121863
Iter 5, residual norm = 8.114715301977117
Iter 6, residual norm = 4.326932882751063
Iter 7, residual norm = 6.726392470543052
Iter 8, residual norm = 9.4296825713747
Iter 9, residual norm = 4.140994365053639e-07


array([-136.71894831,  -33.85176812,   41.84149034,   73.42723076,
        12.50311233,  -41.91028953,   -6.01395805,  -30.72947231,
       -24.21718752,  -68.8095708 ])
```

数学上可以证明，CG 可以保证在$m$步后收敛，其中$m$为$A$的维度，但要注意前提$A$正定。实际使用中如果$A$的性质较好（最大特征值与最小特征值的比值较小），CG 往往在远小于$m$步时就可以收敛

```
np.random.seed(123)
m = 1000
x = np.random.normal(size=(m,m))
# A 为正定矩阵
A = x.transpose().dot(x) / m
# 计算条件数，即最大特征值与最小特征值的比值
print(np.linalg.cond(A))
b = np.random.normal(size=m)
# 难以收敛
# cg(A, b, x0=np.zeros(shape=m), print_progress=True)
print()
A = A + np.eye(m)
print(np.linalg.cond(A))
sol = cg(A, b, x0=np.zeros(shape=m), print_progress=True)
```

```
12220940.930245189
Iter 0, residual norm = 31.596734403985483
Iter 1, residual norm = 32.635197235204686
Iter 2, residual norm = 34.41959611312991
Iter 3, residual norm = 35.473369515912076
Iter 4, residual norm = 36.086207315115146
Iter 5, residual norm = 34.80087186726599
Iter 6, residual norm = 33.8142079449918
Iter 7, residual norm = 33.767295509312426
Iter 8, residual norm = 34.418322504025035
Iter 9, residual norm = 32.88661389363654
Iter 10, residual norm = 30.339976643334143
...
Iter 989, residual norm = 77.47877457019644
Iter 990, residual norm = 72.71818508663523
Iter 991, residual norm = 68.1604644754725
Iter 992, residual norm = 75.8287018173949
Iter 993, residual norm = 64.95214287527648
Iter 994, residual norm = 67.88541157155726
Iter 995, residual norm = 69.66294774833307
Iter 996, residual norm = 70.45449085538544
Iter 997, residual norm = 64.70303416139902
Iter 998, residual norm = 73.3523087241172
Iter 999, residual norm = 72.04388159044062


5.02499268571343
Iter 0, residual norm = 15.524400175022858
Iter 1, residual norm = 6.275357557223782
Iter 2, residual norm = 2.4447891045397867
Iter 3, residual norm = 0.9770907847185353
Iter 4, residual norm = 0.3761522298898337
Iter 5, residual norm = 0.14006062769910327
Iter 6, residual norm = 0.05404412976837954
Iter 7, residual norm = 0.0201751884223988
Iter 8, residual norm = 0.007640617406259879
Iter 9, residual norm = 0.0027718632793437316
Iter 10, residual norm = 0.0010250641894911012
Iter 11, residual norm = 0.00039961370358766886
```

**2、更通用的 CG 实现**

在上述实现中我们可以发现，用到$A$的地方仅仅是计算

矩阵乘法$Ax$，而并未直接使用其他信息，例如$A$中每个元素的取值。因此，我们可以传入一个计算矩阵乘法的函数给 CG，使其可以适用于不同类型的矩阵

```
def cg(Afn, b, x0, eps=1e-3, print_progress=False, **Afn_args)
    m = b.shape[0]
    # 初始解（注意此处应该复制 x0，否则程序退出时会修改 x0）
    x = np.copy(x0)
    # 初始残差向量
    r = b – Afn(x, **Afn_args)
    # 初始共轭梯度
    p = r

    for k in range(m);
        # 矩阵乘法
        Ap = Afn(p, **Afn_args)
        rr = r.dot(r)
        alpha = rr / p.dot(Ap)
        # 更新解
        x += alpha * p
        # 计算新残差向量
        rnew = r – alpha * Ap
        # 测试是否收敛
        norm = np.linalg.norm(rnew)
        if print_progress:
            print(f"Iter {k}, residual norm = {norm}")
        if norm < eps:
            break
        beta = rnew + beta * p
        # 更新残差向量
        r = rnew
    return x
```

此时要是用 CG 时我们需要提供一个函数，而不是 A 本身：

```
def mat_prod(x, mat):
    return mat.dot(x)

sol = cg(mat_prod, b, x0=np.zeros(shape=m), print_process=True)
```

*Iter 0, residual norm = 15.524400175022858*

*Iter 1, residual norm = 6.275357557223782*

*Iter 2, residual norm = 2.4447891045397867*

*Iter 3, residual norm = 0.9770907847185353*

*Iter 4, residual norm = 0.3761522298898337*

*Iter 5, residual norm = 0.14006062769910327*

*Iter 6, residual norm = 0.0540441297683795 4*

*Iter 7, residual norm = 0.0201751884223988*

*Iter 8, residual norm = 0.007640617406259879*

*Iter 9, residual norm = 0.0027718632793437316*

*Iter 10, residual norm = 0.0010250641894911012*

*Iter 11, residual norm = 0.0003996137035876688 6*

这种通用写法的好处是我们可以针对一些特殊的矩阵定义高效的矩阵运算。以对角矩阵为例：

```
def diag_mat_prod(x, diag_elements):
    return diag_elements * x # 逐元素相乘，非矩阵乘法

sol= cg(diag_mat_prod, b, x0=np.zeros(shape=m), print_progress=True, diag_elements=np.diagonal(A))
```

*Iter 0, residual norm = 0.7308521647081689*

*Iter 1, residual norm = 0.023921170204540255*

*Iter 2, residual norm = 0.0008450647663286893*

**以下为岭估计剩余部分**

a. 解决思路

a) 从原始数据生成 RDD（与线性回归步骤相同）

b) 计算$X'y$

c) 定义运算$h \to Ah$,，其中$A = X'X + \lambda I$

d) 利用 CG 解线性方程组

先计算$b = X'y$

```
b = dat.map (lambda part : part [ :, 1 :]. transpose(). dot(part[ :, 0])).reduce(lambda x, y : x + y)
b
```

*array([ 587.9138736 , 994.93758422, -290.75454383, ..., 83.58650008, 38.1770588 , -169.13607987])*

b. CG

a) 利用 CG 求解$Ax = b$时，我们只需要定义计算$Ah$的“运算符”即可，其中$h$是任意的向量

b) 并不需要真正计算出$A$

c) 例如对于$A = X'X + \lambda I$，计算$Ah = X'Xh + \lambda h$要比计算$A$本身高效得多!

我们需要定义一个函数计算$(X'X + \lambda I)v = X'Xv + \lambda v$，其中第一项可以分布式进行（参见笔记）

$A = X'X + \lambda I$

$Ah = X'Xh + \lambda h$

$X'Xh$可分布式进行

计算$X_i'X_iv = X_i'(X_iv)$时应计算$X_iv$!

$$X = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{pmatrix} \quad X \in \mathbb{R}^{n \times p}, \quad X_i \in \mathbb{R}^{n_i \times p}$$

$$X'Xv = (X_1' \; X_2' \; \cdots \; X_m') \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_m \end{pmatrix} v$$

$$= (X_1' \; X_2' \; \cdots \; X_m') \begin{pmatrix} X_1v \\ X_2v \\ \vdots \\ X_mv \end{pmatrix} = X_1'X_1v + \cdots + X_m'X_mv$$

```
def xtxv(part, v) :
    x = part[ :, 1 :]
```

```
    return x.transpose().dot(x.dot(v))

def ridge_prod(v, lam, rdd):
    first_term = rdd.map (lambda part: xtxv( part, v)).
    reduce(lambda x, y: x + y)
    second_term = lam * v
    return first_term + second_term
```

接下来调用 CG 函数，取$\lambda = 0.01n$

```
lam = 0.01 * n
sol = cg(ridge_prod, b, x0=np.zeros(shape=p), eps=1e-
    3, print_progress=True, lam=lam, rdd=dat)
```

*Iter 0, residual norm = 1481.917167549695*

*Iter 1, residual norm = 466.0822472187485*

*Iter 2, residual norm = 147.85796038621356*

*Iter 3, residual norm = 45.90723754521532*

*Iter 4, residual norm = 14.762430871488222*

*Iter 5, residual norm = 4.536826324346681*

*Iter 6, residual norm = 1.363067641086451*

*Iter 7, residual norm = 0.42869364511935115*

*Iter 8, residual norm = 0.1281486732860055*

*Iter 9, residual norm = 0.0406395031406421*

*Iter 10, residual norm = 0.012878574833312855*

*Iter 11, residual norm = 0.004065362064116831*

*Iter 12, residual norm = 0.0012345032488642316*

*Iter 13, residual norm = 0.00038790480079836035*

```
sol[ :30]
```

*array([ 0.11450369,  0.18828283, -0.06202929, -0.09377662,  0.04571471, -0.05918935,  0.11296308,*
*-0.21948877,  0.04293136, -0.07589163, -0.01182078, -0.00730034, -0.0139177 , -0.00679802,*
*0.01078317, -0.00942863,  0.01584149,  0.00366287,  0.02912715,  0.03035421,  0.00098531,*
*-0.02246434,  0.0137688 ,  0.00694125,  0.01863678, -0.01505763,  0.00384798,  0.0209291 ,*
*-0.02197662,  0.01174556])*

## lec9-logistic-regression 分布式 Logistic 回归模型

a. **Logistic 回归**

a) 假定$Y|x \sim Bernoulli(\rho(\beta' x))$

b) $\rho(x) = \frac{1}{1+e^{-x}}$，即 Sigmoid 函数

c) $\rho(\beta' x)$代表$Y$取 1 的概率

d) 给定数据$(y_i, x_i), i = 1, \dots, n$

e) 估计$\beta$

b. **目标函数**

a) 利用极大似然准则

$$L(\beta) = -\sum_{i=1}^{n}\{y_i log\rho_i + (1-y_i)log(1-\rho_i)\}$$

其中$\rho_i = \rho(x_i'\beta)$

b) 找到一个$\beta$的取值，使得$L(\beta)$最小

c) 符号定义

$y$：因变量矩阵，$n * 1$

$X$：自变量矩阵，$n * (p+1)$

$\rho$：$\rho_i = \rho(x_i'\beta)$，$n * 1$

$W$：以$\rho_i(1-\rho_i)$为对角线元素的对角矩阵

c. **迭代算法**

a) 与线性回归不同的是，Logistic 回归的系数估计没有显式解

b) 需要使用迭代式优化算法来求解

d. **优化问题**

a) 找到参数$x \in R^d$的取值，使得函数$f(x)$的取值达到最小

b) 记为$min_x f(x)$

c) $f(x)$通常具有一些特定的性质，如可导、凸性等

d) 一阶算法（**梯度下降法**）

二阶算法（**牛顿法**）

e. 导数

a) 一阶导数（梯度）向量化表示

$$\frac{\partial L(\beta)}{\partial \beta} = X'(\rho - y)$$

$$n * (p+1) \quad n * 1$$

b) 二阶导数（Hessian 矩阵）向量化表示

$$\frac{\partial^2 L(\beta)}{\partial \beta \partial \beta'} = X'WX$$

$$W = diag(\rho_1(1-\rho_1), \dots, \rho_n(1-\rho_n))$$

**1、准备工作**

配置和启动 PySpark:

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
# 本地模式
spark = SparkSession.builder.\
    master("local[*]").\
    appName("Logistic Regression").\
    getOrCreate()
sc = spark.sparkContext
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

*<pyspark.sql.session.SparkSession object at 0x000001CE6FD76370>*

*<SparkContext master=local[*] appName=Logistic Regression>*

假设$n \gg p$，利用 Numpy 生成模拟数据，并写入文件

**Scipy** 是一个用于数学、科学、工程领域的常用软件包，可以处理插值、积分、优化、图像处理、常微分方程数值解的求解、信号处理等问题。它用于有效计算 Numpy 矩阵，使 Numpy 和 Scipy 协同工作，高效解决问题。

**special** 特殊方程

```
import os
import numpy as np
from scipy.special import expit, logit
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 100000
p = 100
x = np.random.normal(size=(n, p))
beta = np.random.normal(size=p)
prob = expit(x.dot(beta)) # p = 1 / (1 + exp(-x * beta))
y = np.random.binomial(1, prob, size=n) #数值为 0 或 1
dat = np.hstack((y.reshape(n, 1), x)) # y 和 x 组合在一起
if not os.path.exists("data"):
    os.makedirs("data", exist_ok=True)
np.savetxt("data/logistic.txt", dat, fmt="%f", delimiter="\t")
```

其中 expit()即 Sigmoid 函数，表达式为$\rho(x) = \frac{1}{1+e^{-x}}$

PySpark 读取文件并进行一些简单操作:

```
file = sc.textFile("data/logistic.txt") #注意查看数据，然后看一下如何分隔，可以通过打印第一行

# 打印矩阵行数
print(file.count())

# 空行
print()

# 打印前 5 行，并将每行字符串截尾
text = file.map(lambda x : x[ :70] + " ... ").take(5)
print(*text, sep="\n")
```

```
100000


0.000000  -1.085631  0.997345  0.282978  -1.506295  -0.578600  1.651437  -2.4...
1.000000  0.642055  -1.977888  0.712265  2.598304  -0.024626  0.034142  0.179...
0.000000  0.703310  -0.598105  2.200702  0.688297  -0.006307  -0.206662  -0.0...
1.000000  0.765055  -0.828989  -0.659151  0.611124  -0.144013  1.316606  -0.7...
0.000000  1.534090  -0.529914  -0.490972  -1.309165  -0.008660  0.976813  -1....
```

**2、牛顿迭代法**

a. **利用牛顿法迭代**

参考

https://online.stat.psu.edu/stat508/lesson/9/9.1/9.1.2

b. **迭代公式**

$$x^{new} = x^{old} - \alpha\left(\frac{\partial^2 f(x)}{\partial x \partial x'}\right)^{-1} * \frac{\partial f(x)}{\partial x}|_{x=x^{old}}$$

牛顿法可以"自适应"步长

可以固定$\alpha = 1$，也可手动调节

c.

Let $\boldsymbol{y}$ be the colomn vector of $y_i$

Let $\boldsymbol{X}$ be the $N * (p+1)$ input matrix

Let $\boldsymbol{p}$ be the N-vector of fitted probabilities with ith element $p(x_i; \beta^{old})$

Let $\boldsymbol{W}$ be an $N * N$ diagonal matrix of weights with ith element $p(x_i; \beta^{old})(1 - p(x_i; \beta^{old}))$

c. The Newton-Raphson step is:

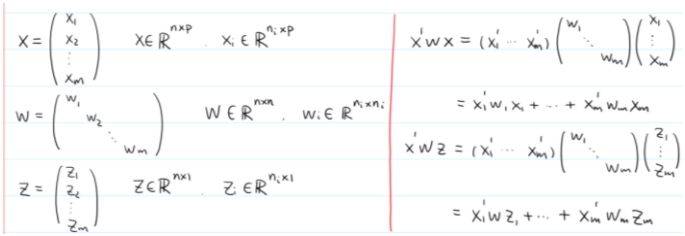
$$\beta^{new} = \beta^{old} + (X^T W X)^{-1} X^T (y - p)$$

$$= (X^T W X)^{-1} X^T W \left(X\beta^{old} + W^{-1}(y - p)\right)$$

$$= (X^T W X)^{-1} X^T W z$$

where $z \triangleq X\beta^{old} + W^{-1}(y - p)$

分块计算



d. **解决思路**

a) 从原始数据生成 RDD（与线性回归步骤相同）

b) 将$\beta^{old}$发送至每个分区

c) 在每个分区上计算$p$和$W$的对角线

d) 在每个分区上计算$X^T W X$和$X^T W z$

e) 汇总分区结果，计算完整的$X^T W X$和$X^T W z$

f) 更新$\beta$

g) 反复迭代直至收敛

当$n \gg p$且$p$不太大时，$X'WX$为$p * p$矩阵，$X'Wz$为$p * 1$向量，均可放入内存。因此，此时问题的核心在于计算$X'WX$和$X'Wz$

首先进行分区映射:

```
# file = file.repartition(10)
print(file.getNumPartitions())
```

```
3
```

```
# str => np.array
def str_to_vec(line):
    # 分割字符串
    str_vec = line.split("\t")
    # 将每一个元素从字符串变成数值型
    num_vec = map(lambda s: float(s), str_vec)
    # 创建 Numpy 向量
    return np.fromiter(num_vec, dtype=float)

# Iter[str] => Iter[matrix]
def part_to_mat(iterator):
    # Iter[str] => Iter[np.array]
    iter_arr = map(str_to_vec, iterator)
```

```
    # Iter[np.array] => list(np.array)
    dat = list(iter_arr)

    # list(np.array) => matrix
    if len(dat) < 1: # test zero iterator
        mat = np.array([])
    else:
        mat = np.vstack(dat)

    # matrix => Iter[matrix]
    yield mat
```

```
dat = file.mapPartitions(part_to_mat).filter(lambda x : x.shape[0] > 0)
print(dat.count())
dat.cache()
```

*3*

*PythonRDD[5] at RDD at PythonRDD.scala:53*

```
dat.first()
```

```
array([[ 0.        , -1.085631,  0.997345, ..., -1.363472,  0.379401, -0.379176],
       [ 1.        ,  0.642055, -1.977888, ..., -0.110851, -0.341262, -0.217946],
       [ 0.        ,  0.70331 , -0.598105, ...,  0.415695,  0.160544,  0.819761],
       ...,
       [ 0.        ,  0.348678, -2.078281, ..., -1.108426, -1.385233, -0.173929],
       [ 0.        , -1.716787,  0.370369, ..., -0.125681, -0.398374, -0.987013],
       [ 0.        ,  0.86831 , -0.9381  , ...,  0.023153, -0.897204,  0.535759]])
```

注意此时每个分区上的数据同时包含了因变量$y$和自变量$X$。给定当前估计$\beta^{old}$，计算每个分区上的统计量$X'WX$和$X'Wz$

```
def compute_stats(part_mat, beta_old):
    # 提取 X 和 y
    y = part_mat[:, 0]
    x = part_mat[:, 1:]
    # X * beta
    xb = x.dot(beta_old)
    # rho(X * beta)
    prob = expit(xb)
    # W 的对角线元素
    w = prob *  (1.0 – prob) + 1e-6
    # X'W，数组广播操作，避免生成完整的 W
    xtw = x.transpose() * w
    # X'WX
    xtwx = xtw.dot(x)
    # X'Wz
    z = xb + (y – prob) / w
    xtwz = xtw.dot(z)
    # 目标函数：sum(y * log(porb) + (1 – y) * log(1-porb))
    objfn = -np.sum(y * np.log(prob + 1e-8) + (1.0 – y) * np.log(1.0 – prob + 1e-8))
    return xtwx, xtwz, objfn
```

主循环：

```
import time

# 根据数据动态获取维度，不要使用之前模拟时的变量
p = dat.first().shape[1] – 1
# beta 初始化为 0 变量
beta_hat = np.zeros(p)
# 记录目标函数值
objvals = []

# 最大迭代次数
maxit = 30
# 收敛次数
eps = 1e-6

t1 = time.time()
for i in range(maxit):
    # 完整数据的 X'WX 和 X'Wz 是各分区的加和
    xtwx, xtwz, objfn = dat.map(lambda part: conpute_stats(part, beta_hat)) .\
            reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2]))
    # 计算新 beta
    beta_new = np.linalg.solve(xtwx, xtwz)
    # 计算 beta 的变化
    resid = np.linalg.norm(beta_new – beta_hat)
    print(f"Iteration {i}, objfn = {objfn}, resid = {resid}")
    objvals.append(objfn) # 存下来后续查看函数变化，帮助 debug
    # 如果 beta 几乎不再变化，退出循环
    if resid < eps:
        break
    # 更新 beta
    beta_hat = beta_new
t2 = time.time
print(f"\nfinished in {t2 – t1} seconds")
```

*Iteration 0, objfn = 69314.71605599453, resid = 1.570403743898301*

*Iteration 1, objfn = 32637.520394548366, resid = 1.3912432651305593*

*Iteration 2, objfn = 21628.54319311961, resid = 1.7393341248434806*

*Iteration 3, objfn = 16007.123573787421, resid = 2.0827596366952865*

*Iteration 4, objfn = 13331.196934357777, resid = 2.063670753391924*

*Iteration 5, objfn = 12380.364067131337, resid = 1.320062748258974*

*Iteration 6, objfn = 12209.479535070568, resid = 0.351657288363309*

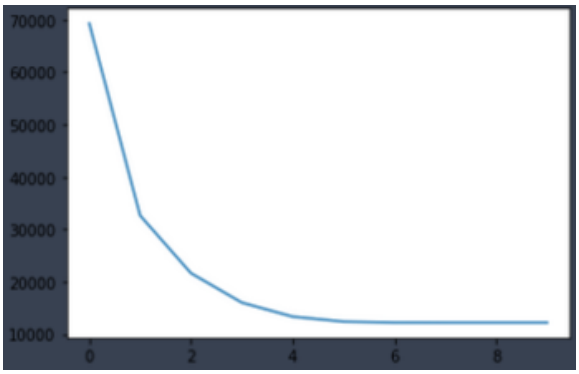
```
Iteration 7, objfn = 12201.993763468883, resid = 0.01883251942906698
Iteration 8, objfn = 12201.974858771522, resid = 6.669254092120501e-05
Iteration 9, objfn = 12201.97485851805, resid = 6.386264232949389e-08

finished in 80.42024254798889 seconds
```

绘制目标函数变化曲线：

```
import matplotlib.pyplot as plt
plt.plot(objvals)
```

```
[<matplotlib.lines.Line2D at 0x1ce094f4100>]
```




beta_hat

```
array([-0.60081137, -1.11699943,  1.18280741,  0.67081184,  0.66972214, -1.7050926 ,  0.87667438,
       -0.69162028,  1.22498276, -0.19295167, -0.57348698,  0.4438535 ,  1.58654734,  0.95466432,
        0.35391181, -0.64090597, -0.17474585,  1.02542448,  0.97981925, -0.21859861,  0.24757379,
       -1.95900962,  0.95583415, -0.46660488, -1.31970033, -1.07853375, -0.96233439, -0.49092673,
       -0.3885543 ,  0.11810773,  0.73827597,  0.41778154,  0.77903403,  0.14037218, -0.20613105,
        1.33428288, -0.83007447, -1.58758058, -0.04096082,  0.0541461 , -0.00642397, -0.33036421,
       -1.77087006, -1.35171751, -1.5966188 , -1.29920738,  0.97837832,  0.94548003, -0.84784072,
       -1.07477454, -0.65768838, -1.52644047, -1.44272699, -1.41389125,  0.07668639, -2.07684364,
        0.23669264, -1.45794974, -0.4810764 , -1.11145764,  1.22263435,  0.71045908,  0.45432555,
       -0.20531513,  1.18659728, -0.16870397,  0.41973803,  0.47919672, -0.27444064, -0.92214026,
       -2.52772108,  1.22723675, -0.42598365,  0.40072802,  0.75623853,  1.60377197, -0.3722838 ,
        0.40608403, -1.41559872, -0.48177148, -0.29070555, -1.65141506,  1.17015611,  0.44435796,
        0.21493992, -0.59912131,  1.16463415, -2.22811656, -0.50241883, -1.11147313,  1.60793224,
       -1.32742954, -0.05349037,  2.20936195,  1.46316333,  0.54528273, -0.72404971,  0.47847074,
        2.16549095,  1.6497254 ])
```

beta

```
array([-0.58441387, -1.11534971,  1.15570434,  0.68635474,  0.64559564, -1.70989894,  0.87296263,
       -0.69061605,  1.21015702, -0.17972039, -0.59593691,  0.45252029,  1.55835773,  0.93393225,
        0.33544231, -0.62751865, -0.16601382,  1.03999001,  0.99253053, -0.22189606,  0.26883567,
       -1.95555529,  0.93138627, -0.47006585, -1.30103516, -1.0852571 , -0.92710125, -0.46948194,
       -0.41250335,  0.09711287,  0.73781056,  0.43563306,  0.78146188,  0.09794209, -0.20673932,
        1.3563857 , -0.84334031, -1.57440936, -0.02785942,  0.04491426, -0.00378645, -0.32005209,
       -1.73699327, -1.35755085, -1.61153494, -1.29018054,  0.92198227,  0.92329806, -0.87182087,
       -1.11109253, -0.64117813, -1.54063266, -1.47551246, -1.40012474,  0.05626203, -2.06673395,
        0.24069911, -1.45962908, -0.49910593, -1.08548   ,  1.22895895,  0.71943325,  0.47494861,
       -0.21579343,  1.19116701, -0.19017216,  0.4216233 ,  0.48972018, -0.2864368 , -0.92757763,
       -2.54487881,  1.2137159 , -0.40262445,  0.40946857,  0.74671794,  1.59476786, -0.36007466,
        0.41427829, -1.43588821, -0.47424149, -0.27699872, -1.66246218,  1.15732258,  0.42792354,
        0.22067973, -0.59542409,  1.16612341, -2.23575698, -0.49771051, -1.10650178,  1.60199211,
       -1.32128632, -0.05640532,  2.21514038,  1.46662192,  0.54730866, -0.7372505 ,  0.44819396,
        2.14210279,  1.67452645])
```

**3、梯度下降法迭代**

a. **迭代公式**

$$\beta^{new} = \beta^{old} - \alpha * \frac{\partial L(\beta)}{\partial \beta}|_{\beta=\beta^{old}}$$

注意，$\frac{\partial L(\beta)}{\partial \beta}$通常会随着$n$而增长，建议使用$n^{-1}\frac{\partial L(\beta)}{\partial \beta}$

$\alpha$的选取需进行一些尝试

$$\beta^{new} = \beta^{old} - \alpha * n^{-1}X'(prob - y)$$

其中$prob$是$\rho(X\beta^{old})$组成的向量

b. **实现方法**

a) 从原始数据生成 RDD（与线性回归步骤相同）

b) 在每一个分区上计算$\rho = \rho(X\beta)$

c) 分布式地计算$X'(\rho - y)$

d) 计算梯度并更新$\beta$

e) 反复迭代直至收敛

```
def compute_obj_grad(part_mat, beta_old):
    #  提取 X 和 y
    y = part_mat[:, 0]
    x = part_mat[:, 1:]
    # X * beta
    xb = x.dot(beta_old)
    # rho(X * beta)
    prob = expit(xb)
    #  目标函数：sum(y * log(prob) + (1 - y) * log(1 – prob)) 取 log 所以防止接近 0
    obj = -np.sum(y * np.log(prob + 1e-8) + (1.0 – y) * np.log(1.0 – prob + 1e-8))
    #  梯度：X'(prob-y)
    grad = x.transpose().dot(prob – y)
    #  该分块的样本量
    ni = x.shape[0]
    return ni, obj, grad
```

主循环：

```
import time

#  根据数据动态获取维度，不要使用之前模拟时的变量
p = dat.first().shape[1] – 1
# beta 初始化为 0 向量
beta_hat2 = np.zeros(p)
#  记录目标函数值
objvals = []

#  最大迭代次数
maxit = 100
#  步长
```

```
step_size = 10.0
# 收敛条件
eps = 1e-5

t1 = time.time()
for i in range(maxit):
    # 完整数据的样本量和梯度是各分区的加和
    n, objfn, grad = dat.map(lambda part: compute_obj_grad(part, beta_hat2)).\
        reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2])
    objfn /= n
    grad /= n
    # 计算新 beta
    beta_new = beta_hat2 – step_size * grad
    # 计算梯度的变化
    grad_norm = np.linalg.norm(grad)
    beta_norm = np.linalg.norm(beta_hat2)
    print(f"Iteration {i}, objfn = {objfn}, grad_norm = {grad_norm}, beta_norm = {beta_norm}")
    objvals.append(objfn)
    # 如果梯度值较小，退出循环
    if grad_norm < eps or grad_norm < eps * beta_norm:
        break
    # 更新 beta
    beta_hat2 = beta_new
t2 = time.time()
print(f"\nfinished in {t2 - t1} seconds")
```

```
Iteration 0, objfn = 0.6931471605599453, grad_norm = 0.3936737955731282, beta_norm = 0.0
Iteration 1, objfn = 0.17950616436038194, grad_norm = 0.0316199315477914 55, beta_norm = 3.936737955731282
Iteration 2, objfn = 0.1707471024683261, grad_norm = 0.025383527519615347, beta_norm = 4.229491074256976
Iteration 3, objfn = 0.16469381927034402, grad_norm = 0.022392841191143965, beta_norm = 4.483061356681379
Iteration 4, objfn = 0.15994191805009847, grad_norm = 0.020095697896829853, beta_norm = 4.7069573090021795
Iteration 5, objfn = 0.1560914605394923, grad_norm = 0.018258205753545906, beta_norm = 4.907896894608809
Iteration 6, objfn = 0.152897837488854, grad_norm = 0.016748376887099004, beta_norm = 5.090465232475998
Iteration 7, objfn = 0.1502003244937587, grad_norm = 0.015481630776221285, beta_norm = 5.257937245323649
Iteration 8, objfn = 0.14788823548439792, grad_norm = 0.014400867947102373, beta_norm = 5.412743195401856
Iteration 9, objfn = 0.14588247717072558, grad_norm = 0.013465993183607724, beta_norm = 5.556742613138677
Iteration 10, objfn = 0.14412480971842243, grad_norm = 0.012647938122615896, beta_norm = 5.691394177422893
Iteration 11, objfn = 0.14257126682057134, grad_norm = 0.011925062170741834, beta_norm = 5.817865937443336
Iteration 12, objfn = 0.1411879521826045, grad_norm = 0.011280889333472271, beta_norm = 5.937109570948278
Iteration 13, objfn = 0.13994825854141418, grad_norm = 0.010702632504312599, beta_norm = 6.049912019790597
Iteration 14, objfn = 0.13883097223518998, grad_norm = 0.010180201043249668, beta_norm = 6.156932372321495
Iteration 15, objfn = 0.13781894778048367, grad_norm = 0.009705515359357654, beta_norm = 6.258728823693962
Iteration 16, objfn = 0.13689816010194886, grad_norm = 0.009272022409833214, beta_norm = 6.355778783321509
Iteration 17, objfn = 0.13605701336545797, grad_norm = 0.008874346154474 52, beta_norm = 6.448494138086995
Iteration 18, objfn = 0.135285828086233, grad_norm = 0.008508030764485124, beta_norm = 6.537233020784583 5
Iteration 19, objfn = 0.1345764545648474, grad_norm = 0.008169348894986737, beta_norm = 6.622309010968415
Iteration 20, objfn = 0.13392197743700873, grad_norm = 0.007855156438380 58, beta_norm = 6.703998418850022
Iteration 21, objfn = 0.1333164869928753, grad_norm = 0.007562781033686115, beta_norm = 6.782546116935563
Iteration 22, objfn = 0.13275490013571217, grad_norm = 0.007289935458333011, beta_norm = 6.858170256913878
Iteration 23, objfn = 0.13223281873103665, grad_norm = 0.007034649611633403, beta_norm = 6.931066120581165
Iteration 24, objfn = 0.13174641645887447, grad_norm = 0.006795216562626027, beta_norm = 7.0014092906910035
Iteration 25, objfn = 0.13129234763395733, grad_norm = 0.006570149358979504, beta_norm = 7.069358282352835
Iteration 26, objfn = 0.1308676731290005, grad_norm = 0.006358146156071961, beta_norm = 7.1350567425741245
Iteration 27, objfn = 0.13046979973838996, grad_norm = 0.006158061841497278, beta_norm = 7.198635301136423
Iteration 28, objfn = 0.1300964301957204, grad_norm = 0.005968884776077151, beta_norm = 7.260213137750762
Iteration 29, objfn = 0.12974552170449705, grad_norm = 0.005789717598891612, beta_norm = 7.31989931665069
Iteration 30, objfn = 0.1294152513225752, grad_norm = 0.005619761285490 6875, beta_norm = 7.377793929257837
Iteration 31, objfn = 0.12910398690310299, grad_norm = 0.005458301829174562, beta_norm = 7.433989077447 72
Iteration 32, objfn = 0.12881026256986186, grad_norm = 0.005304699051684 98, beta_norm = 7.488569723643183
Iteration 33, objfn = 0.12853275791571542, grad_norm = 0.005158377153604571, beta_norm = 7.541614429027028
Iteration 34, objfn = 0.12827028027573967, grad_norm = 0.005018816694620 2855, beta_norm = 7.593195997268795
Iteration 35, objfn = 0.1280217495533896, grad_norm = 0.004885547755641 3035, beta_norm = 7.643382038062671
Iteration 36, objfn = 0.1277861851774676, grad_norm = 0.004758144082995 1825, beta_norm = 7.692235462293708
Iteration 37, objfn = 0.12756269484613764, grad_norm = 0.004636218052813881, beta_norm = 7.739814918652 01
Iteration 38, objfn = 0.12735046477658493, grad_norm = 0.004519416323679728, beta_norm = 7.786175179895874
Iteration 39, objfn = 0.12714875122876557, grad_norm = 0.004407416069438546, beta_norm = 7.831367485645722
Iteration 40, objfn = 0.1269568731117669, grad_norm = 0.004299921703166 0176, beta_norm = 7.87543984750987
Iteration 41, objfn = 0.12677420551369728, grad_norm = 0.004196662018630233, beta_norm = 7.918437321453148
Iteration 42, objfn = 0.12660017402235005, grad_norm = 0.004097387688020212, beta_norm = 7.960402251582864
Iteration 43, objfn = 0.12643424972538803, grad_norm = 0.004001869064817464, beta_norm = 8.001374488914445
Iteration 44, objfn = 0.12627594479643414, grad_norm = 0.003909894248948146, beta_norm = 8.041391588167828
Iteration 45, objfn = 0.12612480858799444, grad_norm = 0.003821267378135 8635, beta_norm = 8.080488985217173
Iteration 46, objfn = 0.12598042416417365, grad_norm = 0.003735807114968 5822, beta_norm = 8.118700157455 65
Iteration 47, objfn = 0.12584240521614753, grad_norm = 0.003653345303825 6006, beta_norm = 8.156056769032 29
Iteration 48, objfn = 0.12571039331170 09, grad_norm = 0.003573727577566 25275, beta_norm = 8.192588802659262
Iteration 49, objfn = 0.1255840554371324, grad_norm = 0.003496803281867679, beta_norm = 8.228324679468106
Iteration 50, objfn = 0.12546308179570206, grad_norm = 0.003422442541097454, beta_norm = 8.263291368205424
Iteration 51, objfn = 0.12534718383175386, grad_norm = 0.003350517385263492, beta_norm = 8.297514484897777
Iteration 52, objfn = 0.12523609245384146, grad_norm = 0.003280909099275 5737, beta_norm = 8.331018383977161
Iteration 53, objfn = 0.1251295564337437, grad_norm = 0.003213510198603 3466, beta_norm = 8.363826241739408
Iteration 54, objfn = 0.12502734096129228, grad_norm = 0.003148214872649254, beta_norm = 8.395960132904719
Iteration 55, objfn = 0.12492922633752078, grad_norm = 0.003084927357984 3482, beta_norm = 8.427441100960513
Iteration 56, objfn = 0.12483500679086647, grad_norm = 0.003023556962886 7995, beta_norm = 8.458289222889059
Iteration 57, objfn = 0.12474448940305792, grad_norm = 0.002964018500368 4673, beta_norm = 8.488523668814915
Iteration 58, objfn = 0.12465749313296645, grad_norm = 0.002906231870165 84, beta_norm = 8.518162757048227
Iteration 59, objfn = 0.12457384792811323, grad_norm = 0.002850121678646 5068, beta_norm = 8.54722400494824
Iteration 60, objfn = 0.12449339391475166, grad_norm = 0.002795616892649201, beta_norm = 8.57572417598622
Iteration 61, objfn = 0.12441598065850838, grad_norm = 0.002742650523748 9168, beta_norm = 8.603679323347011
Iteration 62, objfn = 0.12434146648849106, grad_norm = 0.002691159339849 386, beta_norm = 8.631104830373525
Iteration 63, objfn = 0.1242697178785775, grad_norm = 0.002641083601362 5104, beta_norm = 8.658015448127356
Iteration 64, objfn = 0.12420060888030499, grad_norm = 0.002592366819545 6315, beta_norm = 8.684425330311 42
Iteration 65, objfn = 0.12413402060239499, grad_norm = 0.002544955534839 5253, beta_norm = 8.710348065776119
Iteration 66, objfn = 0.12406984073248886, grad_norm = 0.002498799113287991, beta_norm = 8.735796708809094
Iteration 67, objfn = 0.12400796309714504, grad_norm = 0.002453849559328 6864, beta_norm = 8.760783807389293
Iteration 68, objfn = 0.12394828725656666, grad_norm = 0.002410061343428 3173, beta_norm = 8.785321429569125
```

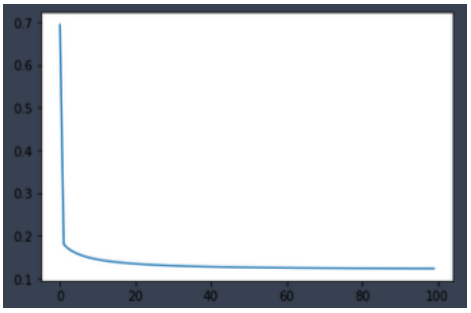
```
Iteration 69, objfn = 0.12389071813089733, grad_norm = 0.0023673912431967 49, beta_norm = 8.809421188133046
Iteration 70, objfn = 0.12383516565524808, grad_norm = 0.0023257981967571106, beta_norm = 8.83309426366746
Iteration 71, objfn = 0.12378154446090804, grad_norm = 0.0022852431672748304, beta_norm = 8.856351426164386
Iteration 72, objfn = 0.1237297735804463, grad_norm = 0.0022456890176599637, beta_norm = 8.87920305527054
Iteration 73, objfn = 0.12367977617463996, grad_norm = 0.0022071003945559773, beta_norm = 8.90165915928352
Iteration 74, objfn = 0.1236314792793652, grad_norm = 0.0021694436208159 13, beta_norm = 8.923729392987962
Iteration 75, objfn = 0.12358481357076832, grad_norm = 0.0021326865957448995, beta_norm = 8.945423074416546
Iteration 76, objfn = 0.12353971314719503, grad_norm = 0.0020967987024575 42, beta_norm = 8.966749200613487

Iteration 77, objfn = 0.1234961153264 9797, grad_norm = 0.0020617507217607 7, beta_norm = 8.987716462471678
Iteration 78, objfn = 0.12345396045747288, grad_norm = 0.0020275147520281 83, beta_norm = 9.008333258708701
Iteration 79, objfn = 0.12341319174428828, grad_norm = 0.0019940641345815714, beta_norm = 9.028607709041665
Iteration 80, objfn = 0.12337375508287568, grad_norm = 0.0019613733841397645, beta_norm = 9.048547666615898
Iteration 81, objfn = 0.12333559890834206, grad_norm = 0.0019294181239348587, beta_norm = 9.06816072973816
Iteration 82, objfn = 0.12329867405254823, grad_norm = 0.0018981750251317927, beta_norm = 9.087454252961106
Iteration 83, objfn = 0.12326293361107299, grad_norm = 0.0018676217502194 52, beta_norm = 9.1064355357561931
Iteration 84, objfn = 0.12322833281885032, grad_norm = 0.0018377369000706061, beta_norm = 9.12511094145502
Iteration 85, objfn = 0.12319482893382788, grad_norm = 0.0018084999643941987, beta_norm = 9.1434876885751 83
Iteration 86, objfn = 0.12316238112805124, grad_norm = 0.0017798912753271944, beta_norm = 9.161572077765491
Iteration 87, objfn = 0.12313095038562766, grad_norm = 0.0017518919639346153, beta_norm = 9.179370391201 03
Iteration 88, objfn = 0.12310049940706919, grad_norm = 0.0017244839194057754, beta_norm = 9.196888722377691
Iteration 89, objfn = 0.12307099251955554, grad_norm = 0.0016976497507523248, beta_norm = 9.214132983692 97
Iteration 90, objfn = 0.12304239559269532, grad_norm = 0.0016713727508296237, beta_norm = 9.231108913643741
Iteration 91, objfn = 0.12301467595939744, grad_norm = 0.0016456368625174804, beta_norm = 9.247822083664296
Iteration 92, objfn = 0.12298780234149607, grad_norm = 0.0016204266469094205, beta_norm = 9.264277904626196
Iteration 93, objfn = 0.12296174477980065, grad_norm = 0.0015957272533716 95, beta_norm = 9.280481633020043
Iteration 94, objfn = 0.12293647456826749, grad_norm = 0.0015715243913440842, beta_norm = 9.296438376837855
Iteration 95, objfn = 0.12291196419201436, grad_norm = 0.0015478043037645468, beta_norm = 9.312153101173484
Iteration 96, objfn = 0.12288818726891935, grad_norm = 0.0015245553742008914, beta_norm = 9.327630633557273
Iteration 97, objfn = 0.12286511849456569, grad_norm = 0.0015017599422449803, beta_norm = 9.342875669040147
Iteration 98, objfn = 0.1228427335903126, grad_norm = 0.0014794106031081238, beta_norm = 9.357892775041249
Iteration 99, objfn = 0.12282100925428764, grad_norm = 0.0014574938646124001, beta_norm = 9.372686395972353

finished in 760.5360913276672 seconds
```

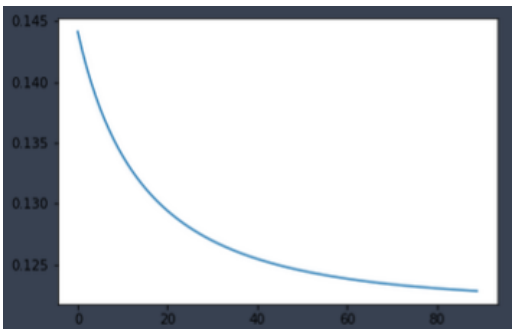
```
import matplotlib.pyplot as plt
plt.plot(objvals)
```

```
[<matplotlib.lines.Line2D at 0x227a43fdf70>]
```




```
plt.plot(objvals[10:])
```

```
[<matplotlib.lines.Line2D at 0x227a4472460>]
```




```
beta_hat[:10]
```

```
array([-0.60081137, -1.11699943,  1.18280741,  0.67081184,  0.66972214, -1.7050926 ,  0.87667438,
       -0.69162028,  1.22498276, -0.19295167])
```

```
beta_hat2[:10]
```

```
array([-0.53574652, -0.99449301,  1.05384849,  0.59716786,  0.59701593, -1.51937926,  0.78181547,
       -0.61447473,  1.09003962, -0.17255694])
```

关闭 Spark 连接：

```
sc.stop()
```

**方法对比**：

1. **梯度下降法**

计算简单，只需求一阶导数

适合高维问题，存储$p*1$向量

收敛较慢

需人工设置步长

2. **牛顿法**

需要求二阶导数（Hessian 矩阵）

存储$p*p$矩阵，计算$O(p^3)$

通常收敛较快

自适应步长

**小结**

1. 通过分布式计算方法得到精确解
2. 对原问题没有进行近似或抽样
3. 已实现方法

1）通过显式解计算（线性回归）

2）共轭梯度法（岭回归）

3）梯度下降法（Logistic 回归）

4）牛顿法（Logistic 回归）

4. 核心都是利用分块矩阵的计算规则

## lec10-logistic-lbfgs 分布式 Logistic 回归模型

是否可以结合梯度下降法与牛顿法的优点？

a. **L-BFGS 优化算法**

a) L-BFGS 算法的全称是 Limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm，是一种改进后的一阶算法

b) L-BFGS 是对 BFGS 算法的改进

c) BFGS 由四位优化算法数学家在 1970 年独立提出

d) L-BFGS 解决了 BFGS 的内存占用问题

e) L 代表 Limited-memory

核心思想：梯度逼近二阶导信息

f) 具体可以参考其论文

Liu, D.C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming, 45*(1), 503-528

g) L-BFGS 是一个比较复杂的算法

h) 但只需要“用户”提供函数值和梯度

i) L-BFGS 利用一阶导数（梯度）的信息来近似二阶导数（Hessian 矩阵）

j) 因此是一种拟牛顿法

k) 在 Python 中由 **scipy.optimize.minimize()**实现

**1、准备工作**

配置和启动 PySpark:

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
#  本地模式
spark = SparkSession.builder.\
        master("local[*]").\
        appName("Logistic Regression").\
        getOrCreate()
sc = spark.sparkContext
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

*<pyspark.sql.session.SparkSession object at 0x000001CD242651C0>*

*<SparkContext master=local[*] appName=Logistic Regression>*

假设$n \gg p$，利用 Numpy 生成模拟数据，并写入文件

```
import os
import numpy as np
from scipy.special import expit, logit
np.set_printoptions(linewidth=100)

np.random.seed(123)
n = 100000
p = 100
x = np.random.normal(size=(n,p))
beta = np.random.normal(size=p)
prob = expit(x.dot(beta)) # p = 1 / (1 + exp(-x * beta))
y = np.random.binomial(1, prob, size=n)
dat = np.hstack((y.reshape(n, 1), x))
if not os.path.exists(“data”)
        os.makedirs(“data”, exist_ok=True)
np.savetxt(“data/logistic.txt”, dat, fmt=”%f”, delimiter
=”\t”)
```

其中 expit()即 Sigmoid 函数，表达式为$\rho(x) = \frac{1}{1+e^{-x}}$

PySpark 读取文件并进行一些简单操作:

```
file = sc.textFile("data/logistic.txt")

#  打印矩阵行数
print(file.count())

#  空行
print()

#  打印前 5 行，并将每行字符串截尾
text = file.map(lambda x: x[:70] + "...").take(5)
print(*text, sep="\n")
```

*100000*

*0.000000  -1.085631  0.997345  0.282978  -1.506295  -0.578600  1.651437  -2.4...*

*1.000000  0.642055  -1.977888  0.712265  2.598304  -0.024626  0.034142  0.179...*

*0.000000  0.703310  -0.598105  2.200702  0.688297  -0.006307  -0.206662  -0.0...*

*1.000000  0.765055  -0.828989  -0.659151  0.611124  -0.144013  1.316606  -0.7...*

*0.000000  1.534090  -0.529914  -0.490972  -1.309165  -0.008660  0.976813  -1....*

```
# file = file.repartition(10)
print(file.getNumPartitions())
```

*3*

```
# str => np.array
def str_to_vec(line):
        #  分割字符串
        str_vec = line.split("\t")
        #  将每一个元素从字符串变成数值型
        num_vec = map(lambda s: float(s), str_vec)
        #  创建  Numpy  向量
        return np.fromiter(num_vec, dtype=float)

# Iter[str] => Iter[matrix]
def part_to_mat(iterator):
        # Iter[str] => Iter[np.array]
        iter_arr = map(str_to_vec, iterator)

        # Iter[np.array] => list(np.array)
        dat = list(iter_arr)

        # list(np.array) => matrix
        if len(dat) < 1:    # Test zero iterator
                mat = np.array([])
        else:
                mat = np.vstack(dat)

        # matrix => Iter[matrix]
```

```
    yield mat
```

```
dat = file.mapPartitions(part_to_mat).filter(lambda x: x.shape[0] > 0)
print(dat.count())
dat.cache() # 注意这一条，注释后，重新跑牛顿法，spark 机制
```

*3*

*PythonRDD[5] at RDD at PythonRDD.scala:53*

```
def compute_obj_grad(part_mat, beta_old):
    # 提取 X 和 y
    y = part_mat[:, 0]
    x = part_mat[:, 1:]
    # X * beta
    xb = x.dot(beta_old)
    # rho(X * beta)
    prob = expit(xb)
    # 目标函数：sum(y * log(prob) + (1 - y) * log(1 - prob))
    obj = -np.sum(y * np.log(prob + 1e-8) + (1.0 - y) * np.log(1.0 - prob + 1e-8)) # 作业中替换这一条
    # 梯度：X'(prob - y)
    grad = x.transpose().dot(prob - y)
    # 该分块的样本量
    ni = x.shape[0]
    return ni, obj, grad # 返回样本量，
```

**2、L-BFGS 算法**

```
import time
from scipy.optimize import minimize
# 利用梯度下降法中的函数，重新封装函数，重点是这个函数的书写
def logistic_obj_grad(beta, *args): # 两个参数，被优化参数，额外参数，这里是 rdd
    dat = args[0]
    n, objfn, grad = dat.map(lambda part : compute_obj_grad(part, beta)).\
        reduce(lambda x, y: (x[0] + y[0], x[1] + y[1], x[2] + y[2]))
    objfn /= n
    grad /=n
    return objfn, grad # 返回目标函数值和梯度

# 根据数据动态获取维度，不要使用之前模拟时的变量
p = dat.first().shape[1] – 1
# beta 初始化为 0 向量
beta_init = np.zeros(p)

# 以下为主函数，最终输出不在 jupyter，在 powershell 里
t1 = time.time()
res = minimize(logistic_obj_grad, beta_init, args=(dat,), method="L-BFGS-B", jac=True, options={"iprint": 1})
t2 = time.time() # jar 代表雅各比，查阅文档
print(f"\nfinished in {t2 - t1} seconds")

beta_hat3 = res["x"]
```

```
ree metrics is stopped
[I 16:42:13.830 NotebookApp] Saving file at /课程文档/Ch10/lec10-logistic-lbfgs.ipynb
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =          100     M =           10
 This problem is unconstrained.

At X0         0 variables are exactly at the bounds

At iterate    0    f=  6.93147D-01    |proj g|=  9.51033D-02

At iterate    1    f=  4.12775D-01    |proj g|=  4.48511D-02

At iterate    2    f=  2.90367D-01    |proj g|=  2.33642D-02

At iterate    3    f=  2.20953D-01    |proj g|=  1.25251D-02

At iterate    4    f=  1.76501D-01    |proj g|=  7.17077D-03

At iterate    5    f=  1.56494D-01    |proj g|=  1.29134D-02

At iterate    6    f=  1.34431D-01    |proj g|=  4.12419D-03

At iterate    7    f=  1.26567D-01    |proj g|=  1.06705D-03
[I 16:44:13.871 NotebookApp] Saving file at /课程文档/Ch10/lec10-logistic-lbfgs.ipynb

At iterate    8    f=  1.23291D-01    |proj g|=  1.07746D-03

At iterate    9    f=  1.22594D-01    |proj g|=  1.25155D-03

At iterate   10    f=  1.22107D-01    |proj g|=  5.56753D-04

At iterate   11    f=  1.22034D-01    |proj g|=  3.23365D-04

At iterate   12    f=  1.22027D-01    |proj g|=  1.89594D-04
```

```
At iterate   13    f=  1.22020D-01    |proj g|=  4.67660D-05

At iterate   14    f=  1.22020D-01    |proj g|=  1.70252D-05

At iterate   15    f=  1.22020D-01    |proj g|=  1.39425D-06

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
  100     15     17      1     0     0   1.394D-06   1.220D-01
  F =  0.12201974889611288

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
[I 16:46:13.806 NotebookApp] Saving file at /课程文档/Ch10/lec10-logistic-lbfgs.ipynb
```

*finished in 148.70640420913696 seconds*

```
beta_hat3[:10]
```

*array([-0.6008189 , -1.11699695,  1.18281089,  0.67080993,  0.66972429, -1.70506681,  0.87665613,*
*-0.69163408,  1.22499528, -0.19294343])*

```
res # 查看最终信息 nfev 代表调用时间
```

*fun: 0.12201974889611288*

*hess_inv: <100x100 LbfgsInvHessProduct with dtype=float64>*

*jac: array([-3.78505979e-07,  4.66461087e-08,  2.59699895e-07, -5.17654454e-08,  1.01806582e-07,*
*9.34622636e-07, -6.53375353e-07, -6.18120095e-07,  5.83893652e-07,  3.60248468e-07,*
*-6.95585250e-07, -1.18318271e-06, -4.52302415e-07,  5.05141784e-07, -6.19783773e-07,*
*4.66933814e-07,  1.97041971e-08,  1.81272498e-07,  5.08973953e-07, -7.25124976e-07,*

```
        -1.05011403e-07,  6.20911294e-08,  7.39379333e-07, -1.25224887e-07, -1.10609594e-06,
        -7.00620355e-07,  9.49437245e-07,  4.38357886e-08,  2.36030201e-07,  1.27913632e-07,
         1.39425027e-06, -8.31224295e-07,  7.90295191e-08, -6.89148134e-07,  1.54018399e-07,
         2.31098613e-07,  4.47804382e-07, -1.95025022e-07,  5.80822504e-07, -7.81645230e-07,
        -4.75782744e-07,  7.02556725e-07,  7.72141336e-08,  8.31959043e-07, -7.76663503e-07,
        -1.19448712e-07,  1.12184338e-07,  1.10033932e-06,  6.15561284e-07,  3.27044448e-07,
        -4.08486659e-07, -5.19921959e-07,  3.43236716e-07,  4.34448719e-07,  1.45794039e-07,
        -1.81699120e-07,  4.88860043e-07, -3.44659776e-07,  1.98532934e-08,  7.81402070e-10,
         1.14358326e-07, -2.85799209e-08,  5.15301506e-07, -3.05790351e-07,  1.02892979e-07,
         1.95239044e-07, -3.12977127e-07,  1.66692604e-07,  5.70324395e-07, -1.33755197e-08,
         3.98887262e-07, -1.56824981e-07,  3.35813096e-08,  4.00087799e-07,  4.85125883e-07,
         5.56836172e-07,  2.67976196e-07,  2.09297963e-07, -5.29822250e-07,  2.33842234e-07,
         4.67414300e-09, -8.69502767e-08,  5.31461604e-07, -6.59595690e-07,  4.20407751e-07,
         1.10929740e-08, -1.22047335e-06,  6.12146760e-07,  4.37329224e-07,  1.93950898e-07,
         1.88798555e-07,  7.38907132e-07,  2.14539188e-07, -1.07929064e-07, -1.08087297e-07,
         5.01373882e-08,  1.02683602e-07,  7.51128883e-07, -3.06733304e-07, -2.65165579e-07])
 message: 'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
    nfev: 17
     nit: 15
    njev: 17
  status: 0
 success: True
       x: array([-0.6008189 , -1.11699695,  1.18281089,  0.67080993,  0.66972429, -1.70506681,  0.87665613,
        -0.69163408,  1.22499528, -0.19294343, -0.57350255,  0.44382462,  1.5865338 ,  0.95467559,
         0.35389578, -0.64089343, -0.17474449,  1.02542704,  0.97982972, -0.21861658,  0.24757154,
        -1.95900535,  0.95584975, -0.46660809, -1.31972484, -1.07854777, -0.96230806, -0.49092421,
        -0.388548  ,  0.11811023,  0.73830868,  0.41775976,  0.77903498,  0.14035499, -0.20612771,
         1.33428587, -0.83006108, -1.58758327, -0.04094692,  0.05412639, -0.00643626, -0.33034605,
        -1.77086499, -1.35169643, -1.59663559, -1.29920797,  0.97837954,  0.94550552, -0.84782394,
        -1.07476544, -0.65769671, -1.52645048, -1.4427167 , -1.41387801,  0.07668988, -2.07684454,
         0.23670439, -1.45795688, -0.48107538, -1.11145633,  1.22263553,  0.71045683,  0.45433836,
        -0.20532225,  1.18659911, -0.16869929,  0.41972928,  0.47920058, -0.27442653, -0.9221396 ,
        -2.52770762,  1.22723114, -0.42598221,  0.40073775,  0.75624951,  1.60378355, -0.37227662,
         0.40608765, -1.41561035, -0.48176481, -0.29070585, -1.6514144 ,  1.17016784,  0.44434037,
         0.21494946, -0.59911931,  1.16460164, -2.22809757, -0.50240765, -1.1114668 ,  1.60793345,
        -1.32740984, -0.05348457,  2.20935598,  1.4631597 ,  0.54528456, -0.72404638,  0.47848859,
         2.16548084,  1.64971502])
```

关闭 Spark 连接：

```
sc.stop()
```

a. 算法优点：

a) 结合了梯度下降和牛顿法各自的优点

b) 只需推导和计算梯度，无需二阶导数

c) 内存消耗不大

d) 收敛速度较快

e) 是当前解光滑优化问题的标准算法之一

## lec10-shuffling 混洗（Shuffling）机制

1) **改进算法细节**

a. 实现细节

a) 数值稳定算法

b) 缓存机制

c) 混洗(Shuffling)机制

b. **数值稳定算法**

a) 在 Logistic 回归中，我们需要计算

$$\rho(x) = \frac{1}{1+e^{-x}}$$

b) 为了计算目标函数还需要$\log\rho(x)$和$\log(1-\rho(x))$

c) $\rho(x) = \frac{1}{1+e^{-x}}$

问题 1：当$x$很小的负值时，$e^{-x} \to +\infty$，造成$\rho(x)$计算不稳定

问题 2：当$\rho(x)$接近于 0 或 1 时，$log\rho(x)$或$log(1-\rho(x))$会出现 NaN

i. **问题 1**

i) 对于问题 1，一种解决方法是对$x$的取值分类讨论

$$x \geq 0, sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$x < 0, sigmoid(x) = \frac{e^{x}}{1+e^{x}}$$

ii) 此时对于任意的$x$，分子与分母都是稳定的取值

iii) scipy.special.expit()函数进行了这样的处理，也很方便地手动实现

```
In [4]: import numpy as np
        def sigmoid(x):
            x = np.array(x)
            e = np.exp(-np.abs(x))
            numer = np.where(x >= 0.0, 1.0, e)
            denom = 1.0 + e
            return numer / denom

        sigmoid([-1000, -100, -10, 0, 10, 100, 1000])

Out[4]: array([0.00000000e+00, 3.72007598e-44, 4.53978687e-05, 5.00000000e-01,
               9.99954602e-01, 1.00000000e+00, 1.00000000e+00])
```

ii, **问题 2**

i) 对于问题 2，一种简单粗暴的方法是在 log()函数中加上一个很小的正数，但还有一种更好的方式

ii) 可以先推到出$\log\rho(x)$的形式

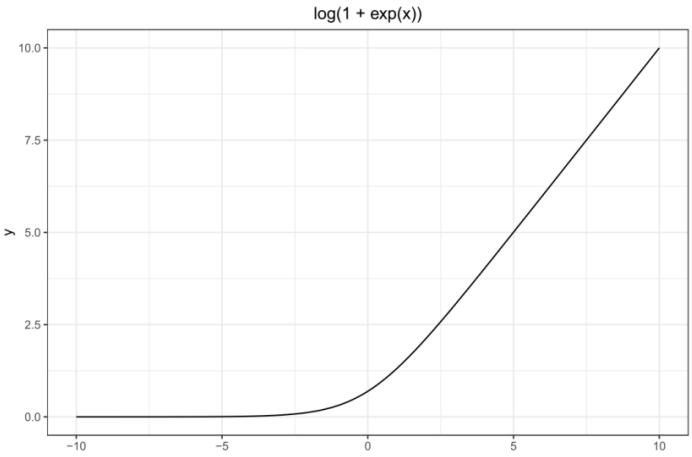
$$\log\ \rho(x) = x - \log(1+e^{x})$$

$$\log\ (1-\rho(x)) = -\log(1+e^{x})$$

iii) 核心在于计算函数是$s(x) = \log(1+e^{x})$

iv) 事实上，$s(x) = \log(1+e^{x})$是一个数值稳定的函数

v) 但需要用特殊的计算方法




vi) 如果直接计算，那么$x$很大时 exp(x)将会溢出

vii) 但是可以发现

$$s(x) = \log(1 + e^x) = x + \log(1 + e^{-x})$$

在$x$很大时与$x$是同一量级

viii) 因此可以分类讨论

$$s(x) = \begin{cases} \log(1 + e^x) & x < 0 \\ x + \log(1 + e^{-x}) & x \geq 0 \end{cases}$$

思考：如何用 Numpy 进行向量化实现

```
import numpy as np
np.exp(1000.0) # 报错
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_5096\1925336385.py:2: RuntimeWarning:
overflow encountered in exp
  np.exp(1000.0) # 报错
inf
```

```
import numpy as np
x = np.array([-2, -1, 3, 4, 5])
print(x > 0)
a = np.array([1, 2, 3, 4, 5])
b = np.array([-1, -2, -3, -4, -5])
np.where(x > 0, a, b) #R 语言中 ifelse
```

```
[False False  True  True  True]
array([-1, -2,  3,  4,  5])
```

c. **缓存机制**

a) 在利用 Spark 对 RDD 进行操作时，通常会叠加很多次变换(map, filter 等等)

b) 理论上每次取数据时都要重复整个流程

c) 为了避免重复操作，可以将中间的某些结果进行缓存

d) 即将变换后的计算结果存进内存

e) 下次需要数据时直接从内存调取

f) 缓存是 Spark 框架非常重要的一个机制

g) 在内存允许的情况下可以显著改善计算效率

h) 使用方法非常简单，rdd.cache()

i) 但也需要注意内存的使用情况

j) 必要时需要配置 PySpark，增加内存使用上限

```
import findspark
findspark.init()

from pyspark.sql import SparkSession
# 本地模式
spark = SparkSession.builder.\
        master("local[*]").\
        appName("Shuffling").\
        getOrCreate()
sc = spark.sparkContext
# sc.setLogLevel("ERROR")
print(spark)
print(sc)
```

```
<pyspark.sql.session.SparkSession object at 0x00000201EA0571C0>
<SparkContext master=local[*] appName=Shuffling>
```

d. **混洗**

a) 在回归的例子中我们发现

b) **MapPartitions** 之后有时数据没有按原始的顺序排列

c) 这是因为划分分区，即调用 **repartition()**时触发了数据混洗(Shuffling)机制

d) 一方面，混洗会使数据重新进行划分，增加通信成本，从而降低运算效率

e) 另一方面，要增加分区数目就必须进行混洗

f) 实际使用中需要一些权衡

g) 如果只需减少分区数目，可以使用 **coalesce()**函数，避免混洗操作

h) 如果一定需要增加分区数，则混洗不可避免

i) 此时需要注意计算结果的顺序

**创建一个简单的 RDD：**

```
import string

rdd = sc.parallelize(string.ascii_uppercase, numSlices=5)
# 这里定义分成 5 份
print(rdd.collect())
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
```

```
rdd.getNumPartitions()
```

```
5
```

**将每个分区的字母合并成一个字符串，从而可以知道分区是如何划分的：**

```
def concat_letters(iter):
        yield "".join(iter)

rdd.mapPartitions(concat_letters).collect()
# 基本上按照等量原则
```

```
['ABCDE', 'FGHIJ', 'KLMNO', 'PQRST', 'UVWXYZ']
```

将 RDD 重新分区，可以看出来字母的顺序产生了变化：

```
rdd2 = rdd.repartition(6)
```

```
print(rdd2.collect())
['A', 'B', 'C', 'D', 'E', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'K', 'L', 'M', 'N', 'O', 'F', 'G', 'H', 'I', 'J']
```

```
rdd2.mapPartitions(concat_letters).collect()
# 数据比较小的时候，随机进行划分
# 注意算法是否依赖数据顺序
['', 'ABCDE', 'PQRSTUVWXYZ', '', 'KLMNO', 'FGHIJ']
```

使用 **toDebugString()**查看 RDD 包含的操作:

```
print(rdd.toDebugString().decode("UTF-8")
print("\n\n")
print(rdd2.toDebugString().decode("UTF-8")
(5) ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 []



(6) MapPartitionsRDD[6] at coalesce at NativeMethodAccessorImpl.java:0 []
 |  CoalescedRDD[5] at coalesce at NativeMethodAccessorImpl.java:0 []
 |  ShuffledRDD[4] at coalesce at NativeMethodAccessorImpl.java:0 []
 +-(5) MapPartitionsRDD[3] at coalesce at NativeMethodAccessorImpl.java:0 []
    |  PythonRDD[2] at RDD at PythonRDD.scala:53 []
    |  ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 []
```

可以看出来 rdd2 包含了混洗(Shuffling)操作，这也是数据顺序打乱的原因。

如果需要减小分区数目，可以使用 coalesce()函数，避免混洗操作

```
rdd3 = rdd.coalesce(numPartitions=2, shuffle=False)
print(rdd3.collect())
rdd3.mapPartitions(concat_letters).collect()
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']


['ABCDEFGHIJ', 'KLMNOPQRSTUVWXYZ']
```

```
print(rdd3.toDebugString().decode("UTF-8"))
(2) CoalescedRDD[8] at coalesce at NativeMethodAccessorImpl.java:0 []
 |  ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 []
```

如果需要增加分区数目，同时需要保持数据顺序，可以在原始 RDD 中增加索引信息:

```
print(rdd.zipWithIndex().collect()) # 如果想略过多少行，可以使用 filter 进行操作
[('A', 0), ('B', 1), ('C', 2), ('D', 3), ('E', 4), ('F', 5), ('G', 6), ('H', 7), ('I', 8), ('J', 9), ('K', 10), ('L', 11), ('M', 12), ('N', 13),
('O', 14), ('P', 15), ('Q', 16), ('R', 17), ('S', 18), ('T', 19), ('U', 20), ('V', 21), ('W', 22), ('X', 23), ('Y', 24), ('Z', 25)]
```

```
rdd4 = rdd.zipWithIndex().repartition(5)
print(rdd4.collect())
[('A', 0), ('B', 1), ('C', 2), ('D', 3), ('E', 4), ('K', 10), ('L', 11), ('M', 12), ('N', 13), ('O', 14), ('P', 15), ('Q', 16), ('R', 17),
('S', 18), ('T', 19), ('F', 5), ('G', 6), ('H', 7), ('I', 8), ('J', 9), ('U', 20), ('V', 21), ('W', 22), ('X', 23), ('Y', 24), ('Z', 25)]
```

然后定义一个分区映射函数，在合并数据行时获取分区中第一条数据的索引:

```
def concat_letters_with_order(iter): # 操作很复杂，对 Index 进行排序，保留第一个数据的索引
    letters_and_indices = list(iter)
    letters = map(lambda x: x[0], letters_and_indices)
    indices = map(lambda x: x[1], letters_and_indices)
    if len(letters_and_indices) < 1:
        yield ()
    else:
        first_ind = next(indices)
        combined_letters = "".join(letters)
        yield combined_letters, first_ind

rdd5 = rdd4.mapPartitions(concat_letters_with_order)
print(rdd5.collect())
[(), ('ABCDE', 0), ('KLMNOPQRST', 10), ('FGHIJ', 5), ('UVWXYZ', 20)]
```

然后过滤空分区，并按生成的索引对 RDD 进行排序;

```
rdd5.filter(lambda x: len(x) > 1).collect()
[('ABCDE', 0), ('KLMNOPQRST', 10), ('FGHIJ', 5), ('UVWXYZ', 20)]
```

```
rdd5.filter(lambda x: len(x) > 1).sortBy(lambda x: x[1], ascending=True).collect()
[('ABCDE', 0), ('FGHIJ', 5), ('KLMNOPQRST', 10), ('UVWXYZ', 20)]
```

e. **排列不变性**

a) 许多模型和问题具有排列不变性

b) 即数据的行打乱后，最终的计算结果不变

c) 例：回归系数，似然函数

d) 反例：观测的预测值

f. **保留顺序**

a) 如果既需要增加分区数，又需要保持数据的顺序，可以在原始数据中加入索引信息

b) 然后在计算结果时将分区的顺序一并返回

c) 最后按索引顺序

## lec11-admm1

内容概述

典型机器学习模型的分布式算法

ADMM 算法（一）

**1、引言**

1) **扩展**

a. 事实上，很多统计和机器学习模型都可以写出形如

$$\min_{\beta} L(\beta) = \sum_{i=1}^{n} l_i(\beta)$$

的表达式

b. 其中$l_i(\beta)$代表第$i$个观测上的损失函数

c. $l_i(\cdot)$依赖于数据$(y_i, x_i)$

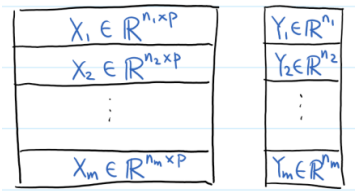
d. $l_i(\cdot)$是光滑函数（二阶可导）

2) **数据切分**

a. 数据按行切分

b. 每个分块包含一部分观测

c. 每个分块包含所有的变量

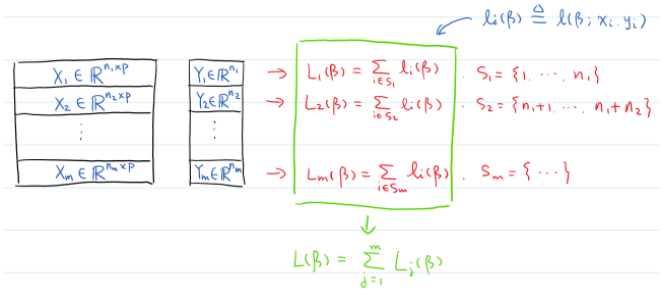


3) **计算思路**

a. 分布式计算每个分块上的损失函数（及梯度）求和

b. 再汇总所有分块的结果，计算总损失函数（及梯度）

c. 利用梯度下降或 L-BFGS 等方法更新参数



d. 例子 Poisson 回归

a) $Y_i|x_i \sim Poisson(\lambda_i)$

b) $\lambda_i = e^{\beta' x_i}$

c) 给定数据$(y_i, x_i)$，$i = 1, \dots, n$，估计$\beta$

d) $L(\beta) =?$，$\frac{\partial L(\beta)}{\partial \beta} =?$

4) **挑战**

a. 然而，实际情况中有一些问题不符合如上的框架，例如：

a) 目标函数不光滑

b) 参数存在约束

b. 例子

a) Least absolute deviations:

$$\min_x \left|\left|Y - X\beta\right|\right|_1$$

b) Lasso:

$$\min_x \frac{1}{2}\left|\left|Y - X\beta\right|\right|^2 + \lambda\left|\left|\beta\right|\right|_1$$

c) SVM:

$$\min_{w,b} \left|\left|w\right|\right|^2$$

$$s.t. \quad y_i(w' x_i + b) \geq 1$$

c. 目标

我们希望能有一个足够通用的框架

a) 支持不光滑的目标函数

b) 处理参数的约束

c) 实现分布式计算

## 2、ADMM

a. 概览

a) 一种解复杂优化问题的方法

b) 对一类统计和机器学习模型提供通用的并行框架

b. ADMM

a) **Minimize** $\boldsymbol{f(x) + g(z)}$

**Subject to** $\boldsymbol{Ax + Bz = c}$

$\boldsymbol{x}$: $\boldsymbol{n}$**维向量**

$\boldsymbol{z}$: $\boldsymbol{m}$**维向量**

$\boldsymbol{A[p * n]}$, $\boldsymbol{B[p * m]}$, $\boldsymbol{c[p * 1]}$: **约束条件**

$\boldsymbol{f(\cdot)}$, $\boldsymbol{g(\cdot)}$: **凸函数**

b) 凸函数

$f(x), x \in R^n$是凸函数

对任意$0 \leq t \leq 1$及$x_1, x_2 \in R^n$,

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

如果$f(x)$有二阶导数，那么该二阶导非负

多变量时，二阶导（Hessian 矩阵）非负定

https://zhuanlan.zhihu.com/p/56876303

lec11-convex.pdf

c. 例子

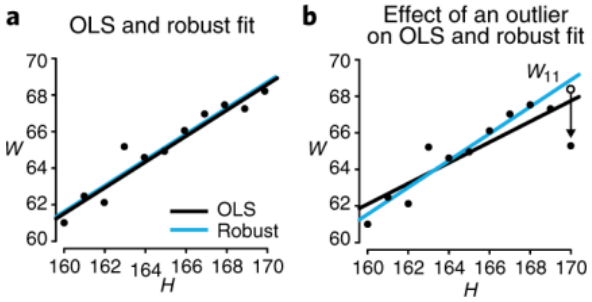
a) **LDA**

i. 回归: $\boldsymbol{\min_x \left|\left|Ax - b\right|\right|^2}$

ii. Least absolute deviations: $\min_x \left|\left|Ax - b\right|\right|_1$

iii. 其中$\left|\left|v\right|\right|_1 = |v_1| + \cdots + |v_p|$，若$v = (v_1, \dots, v_p)'$

iv. 起到稳健回归的作用（中位数回归）




v. **转换成 ADMM 形式**

vi. 令$f = 0, g = \left|\left|\cdot\right|\right|_1$

vii. **Minimize** $\boldsymbol{\left|\left|z\right|\right|_1}$

**Subject to** $\boldsymbol{Ax - z = b}$

b) **Lasso**

i. 回归: $\min_x \left|\left|Ax - b\right|\right|^2$

ii. Lasso: $\boldsymbol{\min_x \frac{1}{2}\left|\left|Ax - b\right|\right|^2 + \lambda\left|\left|x\right|\right|_1}$

iii. 起到变量选择的作用

iv. 选取适当的$\lambda$，$x$的一些元素会变成 0

v. 转换成 **ADMM 的形式**

vi. 令$f(x)=\frac{1}{2}||Ax-b||^2, g(z)=\lambda||z||_1$

vii. **Minimize** $\frac{1}{2}||\boldsymbol{Ax}-\boldsymbol{b}||^2+\boldsymbol{\lambda}||\boldsymbol{z}||_1$

**Subject to** $\boldsymbol{x}-\boldsymbol{z}=\boldsymbol{0}$

**3、ADMM 算法（一）**

a. **增广矩阵**

a) $L_\rho(x,z,y)=f(x)+g(z)+y'(Ax+Bz-c)$

$$+\left(\frac{\rho}{2}\right)||Ax+Bz-c||^2$$

b) $y[p*1]$为辅助变量

c) $\rho$为任意给定的正数

b. **ADMM 算法**

a) $x^{k+1}=argmin_x\ L_\rho(x,z^k,y^k)$

$z^{k+1}=argmin_z\ L_\rho(x^{k+1},z,y^k)$

$y^{k+1}=y^k+\rho(Ax^{k+1}+Bz^{k+1}-c)$

b) 等价形式

$$\boldsymbol{x^{k+1}=argmin_x\ f(x)+\frac{\rho}{2}||Ax+Bz^k-c+u^k||^2}$$

$$\boldsymbol{z^{k+1}=argmin_z\ g(z)+\frac{\rho}{2}||Ax^{k+1}+Bz-c+u^k||^2}$$

$$\boldsymbol{u^{k+1}=u^k+Ax^{k+1}+Bz^{k+1}-c}$$

c. 停止条件

a) 定义两类残差

b) 对偶问题残差 $s^{k+1}=\rho A'B(z^{k+1}-z^k)$

c) 原问题残差 $r^{k+1}=Ax^{k+1}+Bz^{k+1}-c$

d) 当$||r^k||$和$||s^k||$小于某些阈值时停止算法

参见 lec11-admm1.pdf

d. **例：LAD**

a)

$x^{k+1}=(A'A)^{-1}A'(b+z^k-u^k)$

$z^{k+1}=S_{\frac{1}{\rho}}(Ax^{k+1}-b+u^k)$

$u^{k+1}=u^k+Ax^{k+1}-z^{k+1}-b$

b) $\boldsymbol{S_K(a)}$**称为 Soft-thresholding 运算符**

$$\boldsymbol{S_\kappa(a)=\begin{cases}a-\kappa & a>\kappa\\ 0 & |a|\le\kappa\\ a+\kappa & a<-\kappa\end{cases}}$$

e. **例：Lasso**

a)

$$\boldsymbol{x^{k+1}=(A'A+\rho I)^{-1}(A'b+\rho(z^k-u^k))}$$

$$\boldsymbol{z^{k+1}=S_{\frac{\lambda}{\rho}}(x^{k+1}+u^k)}$$

$$\boldsymbol{u^{k+1}=u^k+x^{k+1}-z^{k+1}}$$

f. 适用范围

a) ADMM 使用的问题通常有以下一些特征

i. 没有显式解

ii. 目标函数不是处处可导，如带有$||\cdot||_1$

iii. 带有线性约束

iv. 每个分步更新都有显式解，或可以较容易地计算

b) 难点在于如何将某个优化问题转换成 ADMM 形式，同时让分步更新有显式解

c) lec11-admm1.pdf 的第 6 章给出了若干例子

d) “通用”的分布式计算框架

i. 一致性优化（Consensus）

ii. 共享优化（Sharing）

## lec12

**1、ADMM 算法（二）一致性优化问题**

1) **优化问题**

考虑一个可分的优化问题

$minimize\ f(x)=\sum_{i=1}^N f_i(x)$

$x\in R^n, f_i(x)$是凸函数

注意$x$指的是抽象的参数，不是数据

数据通常包括在$f_i$中

2) **一致性问题**

a. 转换成 ADMM 形式

Minimize $\sum_{i=1}^N f_i(x_i)$

Subject to $x_i-z=0, i=1,\dots,N$

b. 注意，此时需要被优化的参数包括$z, x_1,\dots,x_N$，共$(N+1)n$个

c. 全局一致性问题：所有局部变量相等

d. 假设我们有$N$台机器

e. 那么每台机器可以独立地计算$\min_{x_i} f_i(x_i)$

f. 但还有额外的约束$x_1=x_2=\cdots=x_N$

g. 因此机器之间需要交换信息

3) **迭代算法**

a.

$$x_i^{k+1}:=argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-z^k)+\left(\frac{\rho}{2}\right)||x_i-z^k||_2^2\right)$$

$$z^{k+1}:=\frac{1}{N}\sum_{i=1}^N\left(x_i^{k+1}+\left(\frac{1}{\rho}\right)y_i^k\right)$$

$$y_i^{k+1}:=y_i^k+\rho(x_i^{k+1}-z^{k+1})$$

b. 可以证明，$z^k=\bar{x}^k$

$\bar{x}^k$是$x_1^k,\dots,x_N^k$的平均

算法可以进一步化简

$$x_i^{k+1}:=argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)||x_i-\bar{x}^k||_2^2\right)$$

$$y_i^{k+1}:=y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$

c. 意义

$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

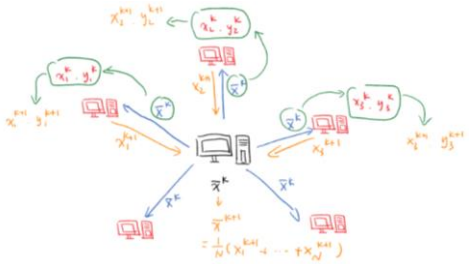
$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$

许多统计和机器学习模型都可以写成这种形式（似然函数平均）

每个$x_i^k$的更新是完全并行的（Map）

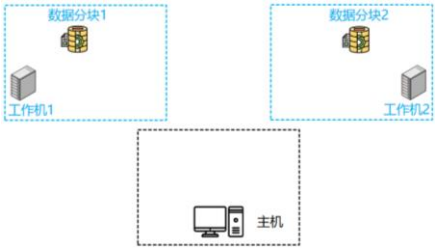
$\bar{x}^k$负责收集每个分块的信息（Reduce）

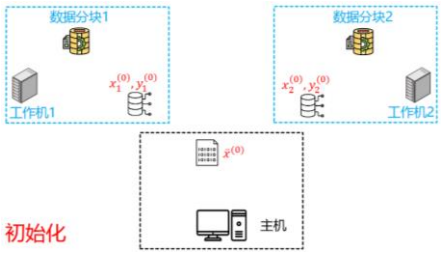


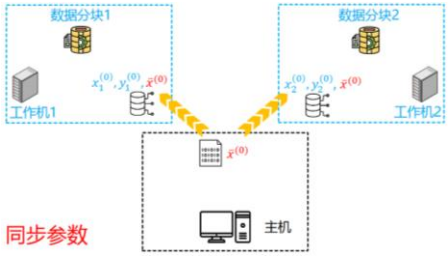
$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$



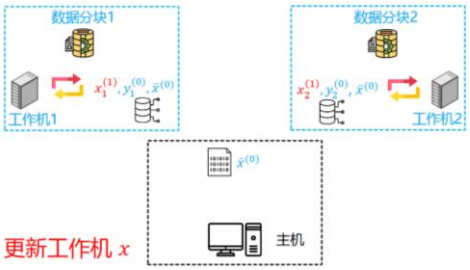




$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

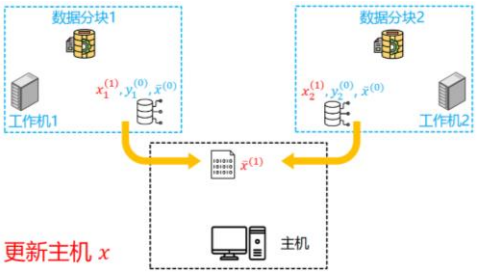
$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$

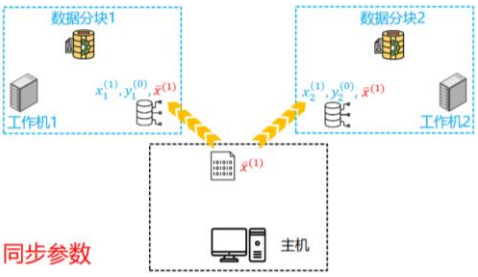



$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$

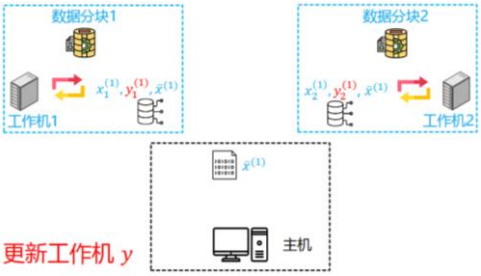



$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

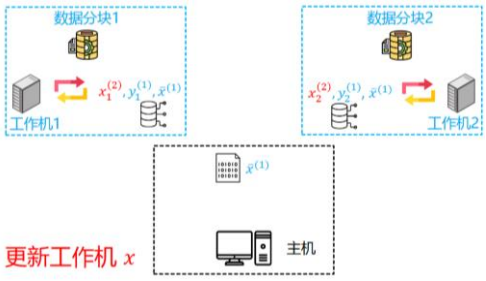
$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$




$minimize\ \ f(x)=\sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i)+y_i^{KT}(x_i-\bar{x}^k)+\left(\frac{\rho}{2}\right)\left||x_i-\bar{x}^k\right||_2^2\right)$$

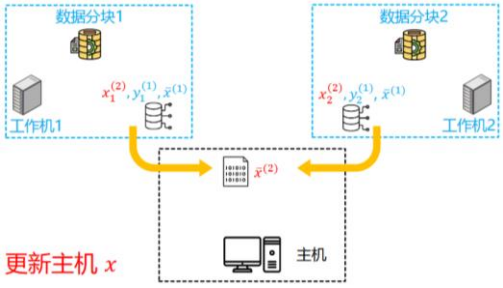
$$y_i^{k+1} := y_i^k+\rho(x_i^{k+1}-\bar{x}^{k+1})$$




$minimize\ \ f(x) = \sum_{i=1}^{N} f_i(x)$

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + y_i^{KT}(x_i - \bar{x}^k) + \left(\frac{\rho}{2}\right)\left||x_i - \bar{x}^k\right|\right|_2^2\Bigg)$$

$$y_i^{k+1} := y_i^k + \rho(x_i^{k+1} - \bar{x}^{k+1})$$




4) 例：线性回归

如果原问题是最小二乘回归

将数据按观测切为$N$块

那么每个$f_i$就是每个分块上的损失函数

每个分块上各自求解一个线性方程组

5) **正则项**

有时我们需要对参数加入全局的正则项

优化问题

Minimize $f(x) = g(x) + \sum_{i=1}^{N} f_i(x)$

$f_i(x), g(x)$是凸函数

例：Lasso

$$\min_{x} \frac{1}{2}\left||Ax - b|\right|^2 + \lambda\left||x|\right|_1$$

转换成 ADMM 形式

Minimize $\sum_{i=1}^{N} f_i(x_i) + g(z)$

Subject to $x_i - z = 0, i = 1, \ldots, N$

6) **迭代算法**

$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + y_i^{KT}(x_i - z^k) + \left(\frac{\rho}{2}\right)\left||x_i - z^k\right|\right|_2^2\Bigg)$$

$$z^{k+1} := argmin_z\left(g(z) + \sum_{i=1}^{N}\left(-y_i^{kT}z + \left(\frac{\rho}{2}\right)\left||x_i^{k+1} - z\right|\right|_2^2\right)\Bigg)$$

$$y_i^{k+1} := y_i^k + \rho(x_i^{k+1} - z^{k+1})$$

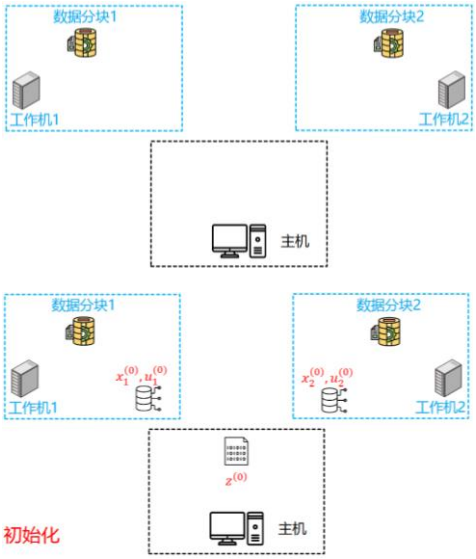
简化形式

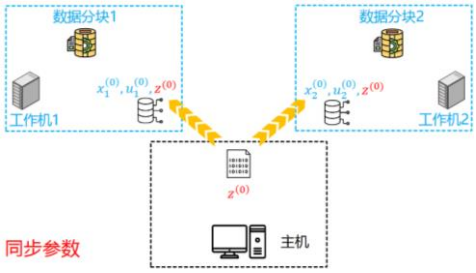
$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\Bigg)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\Bigg)$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$

示意图

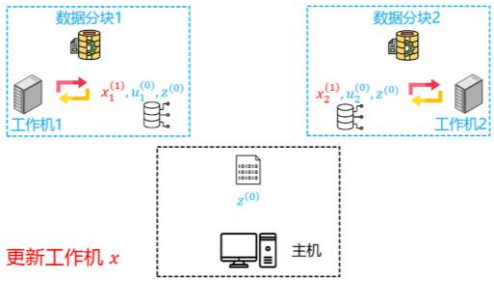



$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\Bigg)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\Bigg)$$

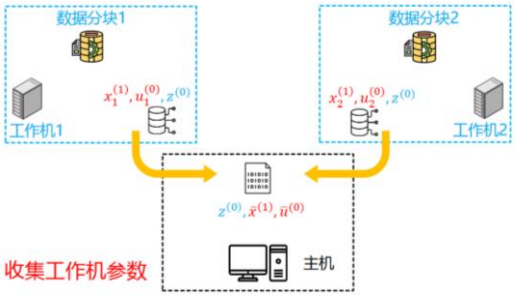
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\Bigg)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\Bigg)$$

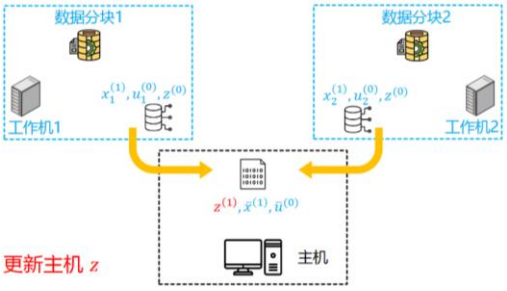
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

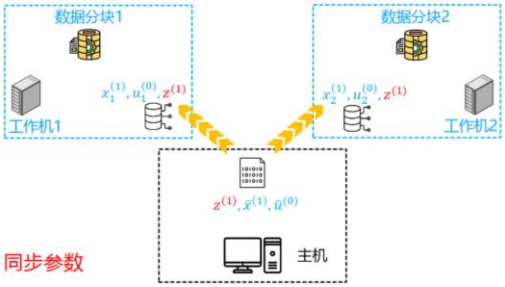
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

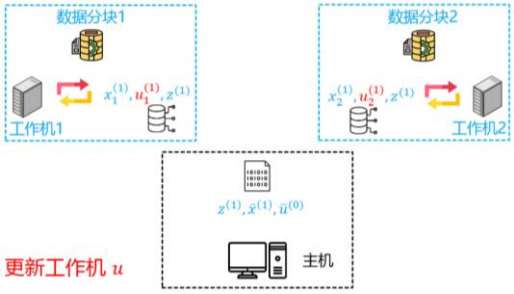
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

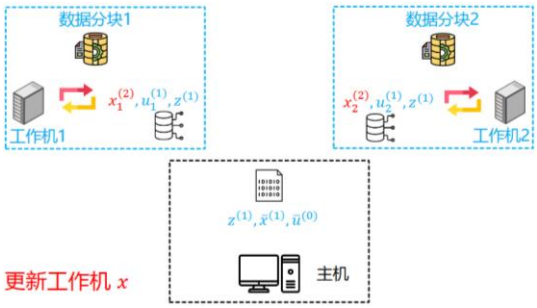
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

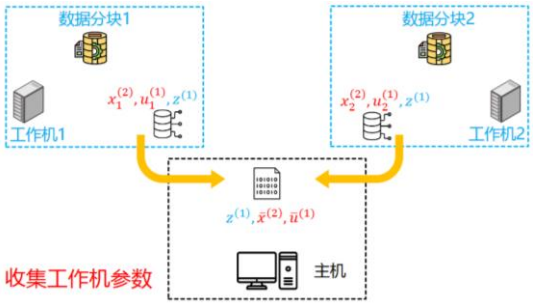
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

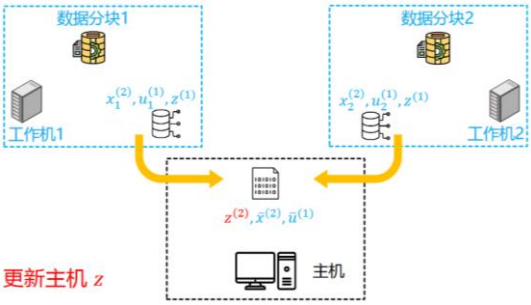
$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




$$x_i^{k+1} := argmin_{x_i}\left(f_i(x_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\right)$$

$$z^{k+1} := argmin_z\left(g(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\right)$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$




7) 例：Lasso

$$\min_x \frac{1}{2}\left||Ax - b|\right|^2 + \lambda\left||x|\right|_1$$

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_N \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$$

$$x_i^{k+1} := \left(A_i^T A_i + \rho I\right)^{-1}\left(A_i^T b_i + \rho(z^k - u_i^k)\right)$$

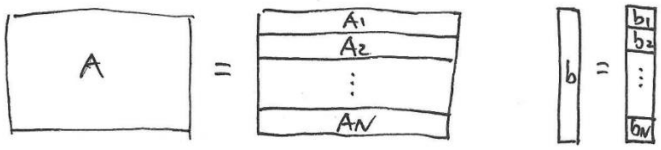
$$z^{k+1} := S_{\frac{\lambda}{\rho N}}(\bar{x}^{k+1} + \bar{u}^k)$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$

8) 典型问题

$$\min_x l(Ax - b) + r(x)$$

$x$: 参数向量
$A, b$: 数据矩阵/向量
9) 数据切分
按行切分
每个分块包含一部分观测
每个分块包含所有的变量




$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_N \end{bmatrix}, b = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$$

$l(Ax - b) = \sum_{i=1}^{N} l_i(Ax_i - b_i)$
Minimize $\sum_{i=1}^{N} l_i(Ax_i - b_i) + r(z)$
Subject to $x_i - z = 0,\ i = 1, \ldots, N$
10) 迭代算法

$$x_i^{k+1} := argmin_{x_i}\left(l_i(A_i x_i - b_i) + \left(\frac{\rho}{2}\right)\left||x_i - z^k + u_i^k\right|\right|_2^2\Big)$$

$$z^{k+1} := argmin_z\left(r(z) + \left(\frac{N\rho}{2}\right)\left||z - \bar{x}^{k+1} - \bar{u}^k\right|\right|_2^2\Big)$$

$$u_i^{k+1} := u_i^k + x_i^{k+1} - z^{k+1}$$

11) 扩展阅读
https://joegaotao.github.io/2014/02/11/admm-stat-compute/

## lec13

### 1、ADMM 算法（三）一致性优化补充

1) 通用框架
“通用”的分布式计算框架
一致性优化（Consensus）
共享优化（Sharing）