



下载APP



29 | Spark MLlib Pipeline : 高效开发机器学习应用

2021-11-17 吴磊

《零基础入门Spark》

课程介绍 >



讲述：吴磊

时长 17:36 大小 16.13M



你好，我是吴磊。

前面我们一起学习了如何在 Spark MLlib 框架下做特征工程与模型训练。不论是特征工程，还是模型训练，针对同一个机器学习问题，我们往往需要尝试不同的特征处理方法或是模型算法。

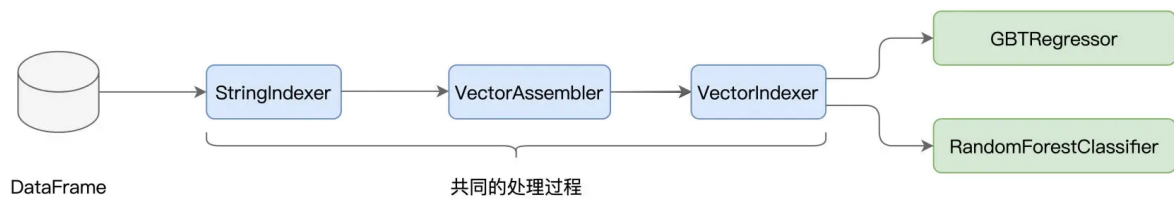


结合之前的大量实例，细心的你想必早已发现，针对同一问题，不同的算法选型在开发的过程中，存在着大量的重复性代码。



以 GBDT 和随机森林为例，它们处理数据的过程是相似的，原始数据都是经过 StringIndexer、VectorAssembler 和 VectorIndexer 这三个环节转化为训练样本，只不

过 GBDT 最后用 GBRegressor 来做回归，而随机森林用 RandomForestClassifier 来做分类。



极客时间

重复的处理逻辑

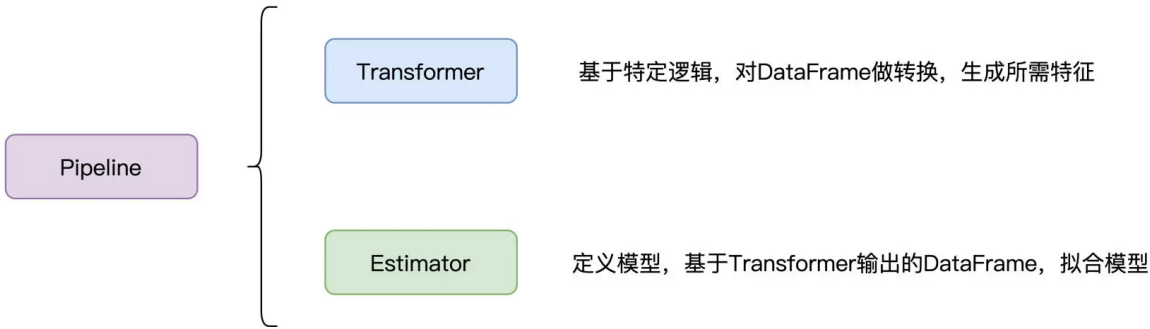
不仅如此，在之前验证模型效果的时候我们也没有闭环，仅仅检查了训练集上的拟合效果，并没有在测试集上进行推理并验证。如果我们尝试去加载新的测试数据集，那么所有的特征处理过程，都需要在测试集上重演一遍。无疑，这同样会引入大量冗余的重复代码。

那么，有没有什么办法，能够避免上述的重复开发，让 Spark MLlib 框架下的机器学习开发更加高效呢？答案是肯定的，今天这一讲，我们就来说说 Spark MLlib Pipeline，看看它如何帮助开发者大幅提升机器学习应用的开发效率。

Spark MLlib Pipeline

什么是 Spark MLlib Pipeline 呢？简单地说，Pipeline 是一套基于 DataFrame 的高阶开发 API，它让开发者以一种高效的方式，来打造端到端的机器学习流水线。这么说可能比较抽象，我们不妨先来看看，Pipeline 都有哪些核心组件，它们又提供了哪些功能。

Pipeline 的核心组件有两类，一类是 Transformer，我们不妨把它称作“转换器”，另一类是 Estimator，我把它叫作“模型生成器”。我们之前接触的各类特征处理函数，实际上都属于转换器，比如 StringIndexer、MinMaxScaler、Bucketizer、VectorAssembler，等等。而前面 3 讲提到的模型算法，全部都是 Estimator。

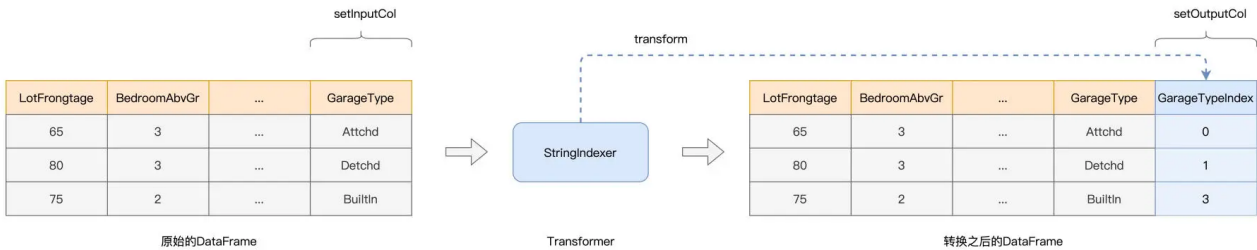


Pipeline核心组件

Transformer

我们先来说说 Transformer，数据转换器。在形式上，Transformer 的输入是 DataFrame，输出也是 DataFrame。结合特定的数据处理逻辑，Transformer 基于原有的 DataFrame 数据列，去创建新的数据列，而新的数据列中，往往包含着不同形式的特征。

以 StringIndexer 为例，它的转换逻辑很简单，就是把字符串转换为数值。在创建 StringIndexer 实例的时候，我们需要使用 setInputCol(s) 和 setOutputCol(s) 方法，来指定原始数据列和期待输出的数据列，而输出数据列中的内容就是我们需要的特征，如下图所示。



以StringIndexer为例，演示Transformer的作用

结合图示可以看到，Transformer 消费原有 DataFrame 的数据列，然后把生成的数据列再追加到该 DataFrame，就会生成新的 DataFrame。换句话说，Transformer 并不是“就地”（Inline）修改原有的 DataFrame，而是基于它去创建新的 DataFrame。

实际上，每个 Transformer 都实现了 `setInputCol(s)` 和 `setOutputCol(s)` 这两个（接口）方法。除此之外，Transformer 还提供了 `transform` 接口，用于封装具体的转换逻辑。正是基于这些核心接口，Pipeline 才能把各式各样的 Transformer 拼接在一起，打造出了特征工程流水线。

一般来说，在一个机器学习应用中，我们往往需要多个 Transformer 来对数据做各式各样的转换，才能生成所需的训练样本。在逻辑上，多个基于同一份原始数据生成的、不同“版本”数据的 `DataFrame`，它们会同时存在于系统中。

不过，受益于 Spark 的惰性求值（Lazy Evaluation）设计，应用在运行时并不会出现多份冗余数据重复占用内存的情况。

不过，为了开发上的遍历，我们还是会使用 `var` 而不是用 `val` 来命名原始的 `DataFrame`。原因很简单，如果用 `val` 的话，我们需要反复使用新的变量名，来命名新生成的 `DataFrame`。关于这部分开发小细节，你可以通过回顾 [🔗 上一讲](#) 的代码来体会。

Estimator

接下来，我们来说说 Estimator。相比 Transformer，Estimator 要简单得多，它实际上就是各类模型算法，如 GBDT、随机森林、线性回归，等等。Estimator 的核心接口，只有一个，那就是 `fit`，中文可以翻译成“拟合”。

Estimator 的作用，就是定义模型算法，然后通过拟合 `DataFrame` 所囊括的训练样本，来生产模型（Models）。这也是为什么我把 Estimator 称作是“模型生成器”。

不过，有意思的是，虽然模型算法是 Estimator，但是 Estimator 生产的模型，却是不折不扣的 Transformer。

要搞清楚为什么模型是 Transformer，我们得先弄明白模型到底是什么。所谓机器学习模型，它本质上就是一个参数（Parameters，又称权重，Weights）矩阵，外加一个模型结构。模型结构与模型算法有关，比如决策树结构、GBDT 结构、神经网络结构，等等。


模型的核心用途就是做推断（Inference）或者说预测。给定数据样本，模型可以推断房价、推断房屋类型，等等。在 Spark MLlib 框架下，数据样本往往是由 `DataFrame` 封装

的，而模型推断的结果，还是保存在（新的）DataFrame 中，结果的默认列名是 “predictions”。

其实基于训练好的推理逻辑，通过增加 “predictions” 列，把一个 DataFrame 转化成一个新的 DataFrame，这不就是 Transformer 在做的事情吗？而这，也是为什么在模型算法上，我们调用的是 fit 方法，而在做模型推断时，我们在模型上调用的是 transform 方法。

构建 Pipeline

好啦，了解了 Transformer 和 Estimator 之后，我们就可以基于它们去构建 Pipeline，来打造端到端的机器学习流水线。实际上，一旦 Transformer、Estimator 准备就绪，定义 Pipeline 只需一行代码就可以轻松拿下，如下所示。

 复制代码


```
1 import org.apache.spark.ml.Pipeline
2
3 // 像之前一样，定义各种特征处理对象与模型算法
4 val stringIndexer = _
5 val vectorAssembler = _
6 val vectorIndexer = _
7 val gbtRegressor = _
8
9 // 将所有的Transformer、Estimator依序放入数组
10 val stages = Array(stringIndexer, vectorAssembler, vectorIndexer, gbtRegressor)
11
12 // 定义Spark MLlib Pipeline
13 val newPipeline = new Pipeline()
14 .setStages(stages)
```

可以看到，要定义 Pipeline，只需创建 Pipeline 实例，然后把之前定义好的 Transformer、Estimator 纷纷作为参数，传入 setStages 方法即可。需要注意的是，**一个 Pipeline 可以包含多个 Transformer 和 Estimator，不过，Pipeline 的最后一个环节，必须是 Estimator，切记。**

到此为止，Pipeline 的作用、定义以及核心组件，我们就讲完了。不过，你可能会说：“概念是讲完了，不过我还是不知道 Pipeline 具体怎么用，以及它到底有什么优势？”别着急，光说不练假把式，接下来，我们就结合 GBDT 与随机森林的例子，来说说 Pipeline 的具体用法，以及怎么用它帮你大幅度提升开发效率。


首先，我们来看看，在一个机器学习应用中，Pipeline 如何帮助我们提高效率。在上一讲，我们用 GBDT 来拟合房价，并给出了代码示例。

现在，咱们把代码稍微调整一下，用 Spark MLlib Pipeline 来实现模型训练。第一步，我们还是先从文件创建 DataFrame，然后把数值型字段与非数值型字段区分开，如下所示。

 复制代码

```
1 import org.apache.spark.sql.DataFrame
2
3 // rootPath为房价预测数据集根目录
4 val rootPath: String = _
5 val filePath: String = s"${rootPath}/train.csv"
6
7 // 读取文件，创建DataFrame
8 var engineeringDF: DataFrame = spark.read.format("csv").option("header", true)
9
10 // 所有数值型字段
11 val numericFields: Array[String] = Array("LotFrontage", "LotArea", "MasVnrArea")
12
13 // Label字段
14 val labelFields: Array[String] = Array("SalePrice")
15
16 import org.apache.spark.sql.types.IntegerType
17
18 for (field <- (numericFields ++ labelFields)) {
19   engineeringDF = engineeringDF
20     .withColumn(s"${field}Int", col(field).cast(IntegerType))
21     .drop(field)
22 }
```

数据准备好之后，接下来，我们就可以开始着手，为 Pipeline 的构建打造零件：依次定义转换器 Transformer 和模型生成器 Estimator。在上一讲，我们用 StringIndexer 把非数值字段转换为数值字段，这一讲，咱们也依法炮制。


 复制代码

```
1 import org.apache.spark.ml.feature.StringIndexer
2
3 // 所有非数值型字段
4 val categoricalFields: Array[String] = Array("MSSubClass", "MSZoning", "Street")
5
6 // StringIndexer期望的输出列名
7 val indexFields: Array[String] = categoricalFields.map(_ + "Index").toArray
8
9 // 定义StringIndexer实例
```



```
10 val stringIndexer = new StringIndexer()
11 // 批量指定输入列名
12 .setInputCols(categoricalFields)
13 // 批量指定输出列名，输出列名与输入列名，必须要一一对应
14 .setOutputCols(indexFields)
15 .setHandleInvalid("keep")
```


在上一讲，定义完 StringIndexer 实例之后，我们立即拿它去对 engineeringDF 做转换。不过在构建 Pipeline 的时候，我们不需要这么做，只需要把这个“零件”定义好即可。接下来，我们来打造下一个零件：VectorAssembler。

 复制代码

```
1 import org.apache.spark.ml.feature.VectorAssembler
2
3 // 转换为整型的数值型字段
4 val numericFeatures: Array[String] = numericFields.map(_ + "Int").toArray
5
6 val vectorAssembler = new VectorAssembler()
7 /** 输入列为：数值型字段 + 非数值型字段
8 注意，非数值型字段的列名，要用indexFields，
9 而不能用原始的categoricalFields，不妨想一想为什么？
10 */
11 .setInputCols(numericFeatures ++ indexFields)
12 .setOutputCol("features")
13 .setHandleInvalid("keep")
```

与上一讲相比，VectorAssembler 的定义并没有什么两样。


下面，我们继续来打造第三个零件：VectorIndexer，它用于帮助模型算法区分连续特征与离散特征。

 复制代码

```
1 import org.apache.spark.ml.feature.VectorIndexer
2
3 val vectorIndexer = new VectorIndexer()
4 // 指定输入列
5 .setInputCol("features")
6 // 指定输出列
7 .setOutputCol("indexedFeatures")
8 // 指定连续、离散判定阈值
9 .setMaxCategories(30)
10 .setHandleInvalid("keep")
```


到此为止，Transformer 就全部定义完了，原始数据经过 StringIndexer、VectorAssembler 和 VectorIndexer 的转换之后，会生成新的 DataFrame。在这个最新的 DataFrame 中，会有多个由不同 Transformer 生成的数据列，其中“indexedFeatures”列包含的数据内容即是特征向量。

结合 DataFrame 一路携带过来的“SalePriceInt”列，特征向量与预测标的终于结合在一起了，就是我们常说的训练样本。有了训练样本，接下来，我们就可以着手定义 Estimator。

 复制代码


```
1 import org.apache.spark.ml.regression.GBTRegressor
2
3 val gbtRegressor = new GBTRegressor()
4 // 指定预测标的
5 .setLabelCol("SalePriceInt")
6 // 指定特征向量
7 .setFeaturesCol("indexedFeatures")
8 // 指定决策树的数量
9 .setMaxIter(30)
10 // 指定决策树的最大深度
11 .setMaxDepth(5)
```

好啦，到这里，Pipeline 所需的零件全部打造完毕，零件就位，只欠组装。我们需要通过 Spark MLlib 提供的“流水线工艺”，把所有零件组装成 Pipeline。

 复制代码

```
1 import org.apache.spark.ml.Pipeline
2
3 val components = Array(stringIndexer, vectorAssembler, vectorIndexer, gbtRegre
4
5 val pipeline = new Pipeline()
6 .setStages(components)
```

怎么样，是不是很简单？接下来的问题是，有了 Pipeline，我们都能用它做些什么呢？

 复制代码

```
1 // Pipeline保存地址的根目录
2 val savePath: String = _
3
4 // 将Pipeline物化到磁盘，以备后用（复用）
```



```
5 pipeline.write
6 .overwrite()
7 .save(s"${savePath}/unfit-gbdt-pipeline")
8
9 // 划分出训练集和验证集
10 val Array(trainingData, validationData) = engineeringDF.randomSplit(Array(0.7,
11
12 // 调用fit方法, 触发Pipeline计算, 并最终拟合出模型
13 val pipelineModel = pipeline.fit(trainingData)
```

首先, 我们可以把 Pipeline 保存下来, 以备后用, 至于怎么复用, 我们待会再说。再者, 把之前准备好的训练样本, 传递给 Pipeline 的 fit 方法, 即可触发整条 Pipeline 从头至尾的计算逻辑, 从各式各样的数据转换, 到最终的模型训练, 一步到位。

Pipeline fit 方法的输出结果, 即是训练好的机器学习模型。我们最开始说过, 模型也是 Transformer, 它可以用来推断预测结果。

看到这里, 你可能会说: “和之前的代码实现相比, Pipeline 也没有什么特别之处, 无非是用 Pipeline API 把之前的环节拼接起来而已”。其实不然, 基于构建好的 Pipeline, 我们可以在不同范围对其进行复用。对于机器学习应用来说, 我们**既可以在作业内部实现复用, 也可以在作业之间实现复用, 从而大幅度提升开发效率。**

作业内的代码复用


在之前的模型训练过程中, 我们仅仅在训练集与验证集上评估了模型效果。实际上, 在工业级应用中, 我们最关心的, 是模型在测试集上的泛化能力。就拿 Kaggle 竞赛来说, 对于每一个机器学习项目, Kaggle 都会同时提供 train.csv 和 test.csv 两个文件。

其中 train.csv 是带标签的, 用于训练模型, 而 test.csv 是不带标签的。我们需要对 test.csv 中的数据做推断, 然后把预测结果提交到 Kaggle 线上平台, 平台会结合房屋的实际价格来评判我们的模型, 到那时我们才能知道, 模型对于房价的预测到底有多准 (或是有多不准)。

要完成对 test.csv 的推断, 我们需要把原始数据转换为特征向量, 也就是把“粗粮”转化为“细粮”, 然后才能把它“喂给”模型。

在之前的代码实现中，要做到这一点，我们必须把之前加持到 train.csv 的所有转换逻辑都重写一遍，比如 StringIndexer、VectorAssembler 和 VectorIndexer。毫无疑问，这样的开发方式是极其低效的，更糟的是，手工重写很容易会造成测试样本与训练样本不一致，而这样的不一致是机器学习应用中的大忌。

不过，有了 Pipeline，我们就可以省去这些麻烦。首先，我们把 test.csv 加载进来并创建 DataFrame，然后把数值字段从 String 转为 Int。

 复制代码

```
1 import org.apache.spark.sql.DataFrame
2
3 val rootPath: String = _
4 val filePath: String = s"${rootPath}/test.csv"
5
6 // 加载test.csv，并创建DataFrame
7 var testData: DataFrame = spark.read.format("csv").option("header", true).load
8
9 // 所有数值型字段
10 val numericFields: Array[String] = Array("LotFrontage", "LotArea", "MasVnrArea")
11
12 // 所有非数值型字段
13 val categoricalFields: Array[String] = Array("MSSubClass", "MSZoning", "Street")
14
15 import org.apache.spark.sql.types.IntegerType
16
17 // 注意，test.csv没有SalePrice字段，也即没有Label
18 for (field <- (numericFields)) {
19   testData = testData
20     .withColumn(s"${field}Int", col(field).cast(IntegerType))
21     .drop(field)
22 }
```

接下来，我们只需要调用 Pipeline Model 的 transform 方法，就可以对测试集做推理。还记得吗？模型是 Transformer，而 transform 是 Transformer 用于数据转换的统一接口。

 复制代码


```
1 val predictions = pipelineModel.transform(testData)
```

有了 Pipeline，我们就可以省去 StringIndexer、VectorAssembler 这些特征处理函数的重复定义，在提升开发效率的同时，消除样本不一致的隐患。除了在同一个作业内部复用 Pipeline 之外，我们还可以在不同的作业之间对其进行复用，从而进一步提升开发效率。

作业间的代码复用

对于同一个机器学习问题，我们往往会尝试不同的模型算法，以期获得更好的模型效果。例如，对于房价预测，我们既可以用 GBDT，也可以用随机森林。不过，尽管模型算法不同，但是它们的训练样本往往是类似的，甚至是完全一样的。如果每尝试一种模型算法，就需要从头处理一遍数据，这未免过于低效，也容易出错。

有了 Pipeline，我们就可以把算法选型这件事变得异常简单。还是拿房价预测来举例，之前我们尝试使用 GBTRegressor 来训练模型，这一次，咱们来试试 RandomForestRegressor，也即使用随机森林来解决回归问题。按照惯例，我们还是结合代码来进行讲解。

 复制代码

```
1 import org.apache.spark.ml.Pipeline
2
3 val savePath: String = _
4
5 // 加载之前保存到磁盘的Pipeline
6 val unfitPipeline = Pipeline.load(s"${savePath}/unfit-gbdt-pipeline")
7
8 // 获取Pipeline中的每一个Stage (Transformer或Estimator)
9 val formerStages = unfitPipeline.getStages
10
11 // 去掉Pipeline中最后一个组件，也即Estimator: GBTRegressor
12 val formerStagesWithoutModel = formerStages.dropRight(1)
13
14 import org.apache.spark.ml.regression.RandomForestRegressor
15
16 // 定义新的Estimator: RandomForestRegressor
17 val rf = new RandomForestRegressor()
18 .setLabelCol("SalePriceInt")
19 .setFeaturesCol("indexedFeatures")
20 .setNumTrees(30)
21 .setMaxDepth(5)
22
23 // 将老的Stages与新的Estimator拼接在一起
24 val stages = formerStagesWithoutModel ++ Array(rf)
25
26 // 重新定义新的Pipeline
27 val newPipeline = new Pipeline()
```

```
28 .setStages(stages)
```

首先，我们把之前保存下来的 Pipeline，重新加载进来。然后，用新的 RandomForestRegressor 替换原来的 GBRegressor。最后，再把原有的 Stages 和新的 Estimator 拼接在一起，去创建新的 Pipeline 即可。接下来，只要调用 fit 方法，就可以触发新 Pipeline 的运转，并最终拟合出新的随机森林模型。

[复制代码](#)

```
1 // 像之前一样，从train.csv创建DataFrame，准备数据
2 var engineeringDF = _
3
4 val Array(trainingData, testData) = engineeringDF.randomSplit(Array(0.7, 0.3))
5
6 // 调用fit方法，触发Pipeline运转，拟合新模型
7 val pipelineModel = newPipeline.fit(trainingData)
```

可以看到，短短的几行代码，就可以让我们轻松地完成模型选型。到此，Pipeline 在开发效率与容错上的优势，可谓一览无余。

重点回顾

今天的内容就讲完啦，今天这一讲，我们一起学习了 Spark MLlib Pipeline。你需要理解 Pipeline 的优势所在，并掌握它的核心组件与具体用法。Pipeline 的核心组件是 Transformer 与 Estimator。

其中，Transformer 完成从 DataFrame 到 DataFrame 的转换，基于固有的转换逻辑，生成新的数据列。Estimator 主要是模型算法，它基于 DataFrame 中封装的训练样本，去生成机器学习模型。将若干 Transformer 与 Estimator 拼接在一起，通过调用 Pipeline 的 setStages 方法，即可完成 Pipeline 的创建。

Pipeline 的核心优势在于提升机器学习应用的开发效率，并同时消除测试样本与训练样本之间

不一致这一致命隐患。Pipeline 可用于作业内的代码复用，或是作业间的代码复用。

在同一作业内，Pipeline 能够轻松地在测试集之上，完成数据推断。而在作业之间，开发者可以加载之前保存好的 Pipeline，然后用“新零件”替换“旧零件”的方式，在复用大部分处理逻辑的同时，去打造新的 Pipeline，从而实现高效的模型选型过程。

在今后的机器学习开发中，我们要充分利用 Pipeline 提供的优势，来降低开发成本，从而把主要精力放在特征工程与模型调优上。

到此为止，Spark MLlib 模块的全部内容，我们就讲完了。

在这个模块中，我们主要围绕着特征工程、模型训练和机器学习流水线等几个方面，梳理了 Spark MLlib 子框架为开发者提供的种种能力。换句话说，我们知道了 Spark MLlib 能做什么事情、擅长做什么事情。如果我们能够做到对这些能力了如指掌，在日常的机器学习开发中，就可以灵活地对其进行取舍，从而去应对不断变化的业务需求，加油！

每日一练

我们今天一直在讲 Pipeline 的优势，你能说一说，Pipeline 有哪些可能的劣势吗？

欢迎你在留言区和我交流互动，也推荐你把这一讲分享给更多同事、朋友，说不定就能让他进一步理解 Pipeline。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 28 | 模型训练（下）：协同过滤与频繁项集算法详解

下一篇 30 | Structured Streaming：从“流动的 Word Count”开始

更多学习推荐

2021 最新大厂 Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取 



精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。