DolphinDB Shark 白皮书

DATABASE
ANALYTICS
STREAMING

内容

引言

因子挖掘是通过分析大量数据，识别影响资产价格变动的关键因素的过程。传统的因子挖掘方法主要基于经济理论与投资经验，难以有效表达复杂的非线性关系，无法充分挖掘数据中的潜在信息。随着数据集的丰富以及计算机算力的提高，基于遗传算法的自动因子挖掘方法已被广泛应用。遗传算法通过随机生成公式，模拟自然进化过程，全面、系统地搜索因子特征空间，发现传统方法难以构建的隐藏因子。

Shark 是 DolphinDB 3.0 推出的 CPU-GPU 异构计算平台，其内置了 GPLearn 高性能因子挖掘功能，用户可以通过 Shark 直接读取存储在 DolphinDB 中的数据，然后调用 GPLearn 遗传算法自动从数据中挖掘出因子。Shark 利用 GPU 加速遗传算法的挖掘速度和因子计算效率，与基于CPU的Python GPLearn相比，在千万行的数据上可以提速80X倍。除此之外，Shark GPLearn 还内置实现了一些金融算子；并且提供分组语义在挖掘因子时自动按照分组进行计算，从而支持从三维数据中挖掘因子。

第 1 章. 量化金融因子挖掘方法

目前，金融领域挖掘因子的方法有基于统计分析和经济理论的传统因子挖掘方法和基于遗传算法的挖掘方法等。传统的因子挖掘方法一般依赖数学工具以及研究员的投资经验，挖掘出来的因子具有一定的可解释性，常见的因子如估值、波动率等都是通过这种方法挖掘得到的。但是传统的方法难以挖掘出数据的一些非线性特征和潜在信息。基于遗传算法的挖掘方法，通过利用启发式算法充分搜索数据的特征空间，可以发现一些隐藏的因子，但这些因子一般可解释性较差。本章将简要介绍一下这两种方法。

1.1 传统因子挖掘方法

传统的因子挖掘方法主要基于统计分析和经济理论，从而发现影响资产回报的关键因素。用户一般通过回归分析、因子载荷分析和投资组合构建等方法来识别和利用这些因子，然后构建因子模型。因子模型可以识别和量化不同因素对资产收益的影响，发现不同资产或投资组合之间的收益差异，从而帮助用户进行更有效的风险管理和资产配置。例如，用户可以通过控制对某些高风险因子的敞口，从而降低整个投资组合的风险。

传统的因子挖掘方法依赖于人工设计和选择特征工程方法，这导致传统方法难以有效处理复杂的非线性关系和高维数据，从而无法充分挖掘数据中的潜在信息。

1.2 遗传算法自动挖掘因子

遗传算法会随机生成一组公式，通过模拟自然界生物进化的过程，从数据中搜索出一些有效的因子。遗传算法作为一种启发式算法，可以全面、系统地搜索因子的特征空间，从而发现隐藏的、难以通过传统方法构建的因子。但是通过这种方法挖掘出的因子，一般可解释性较差，需要经过严格检验之后，才能在实际中使用。尽管如此，这种方法可以突破研究员的思维局限，为因子研究提供更多的可能性。

第 2 章. 遗传算法介绍

遗传算法最初由美国密歇根大学的 J.Holland 提出，是一种通过模拟自然界生物进化的过程来搜索近似最优解的算法。在因子挖掘这个场景中，初始时遗传算法会随机生成一组公式，然后模拟生物学中遗传、突变、杂交等现象，使得一定数量的初始解按照最优的方向进化。

2.1 遗传算法的总体流程

图 2-1 描述了遗传算法的总体流程。在初始时，遗传算法会随机生成一组公式，然后衡量这些公式与给定数据的相符程度，得到的分数被称为适应度。适应度衡量了公式与给定数据的相符程度，对于不同的应用，可以使用不同的适应度函数，例如对于回归问题，可以使用公式结果和目标值之间的均方误差为适应度，对于遗传算法生成的选股因子，可以使用因子收益率来作为适应度。

紧接着，遗传算法会从这些公式中，依据适应度选出合适的公式作为下一代进化的父代。这些被选择出来的公式会通过多种方法进化，形成不同的后代公式。后代公式会继续循环这个过程，不断进化，直至公式拟合数据的分布。

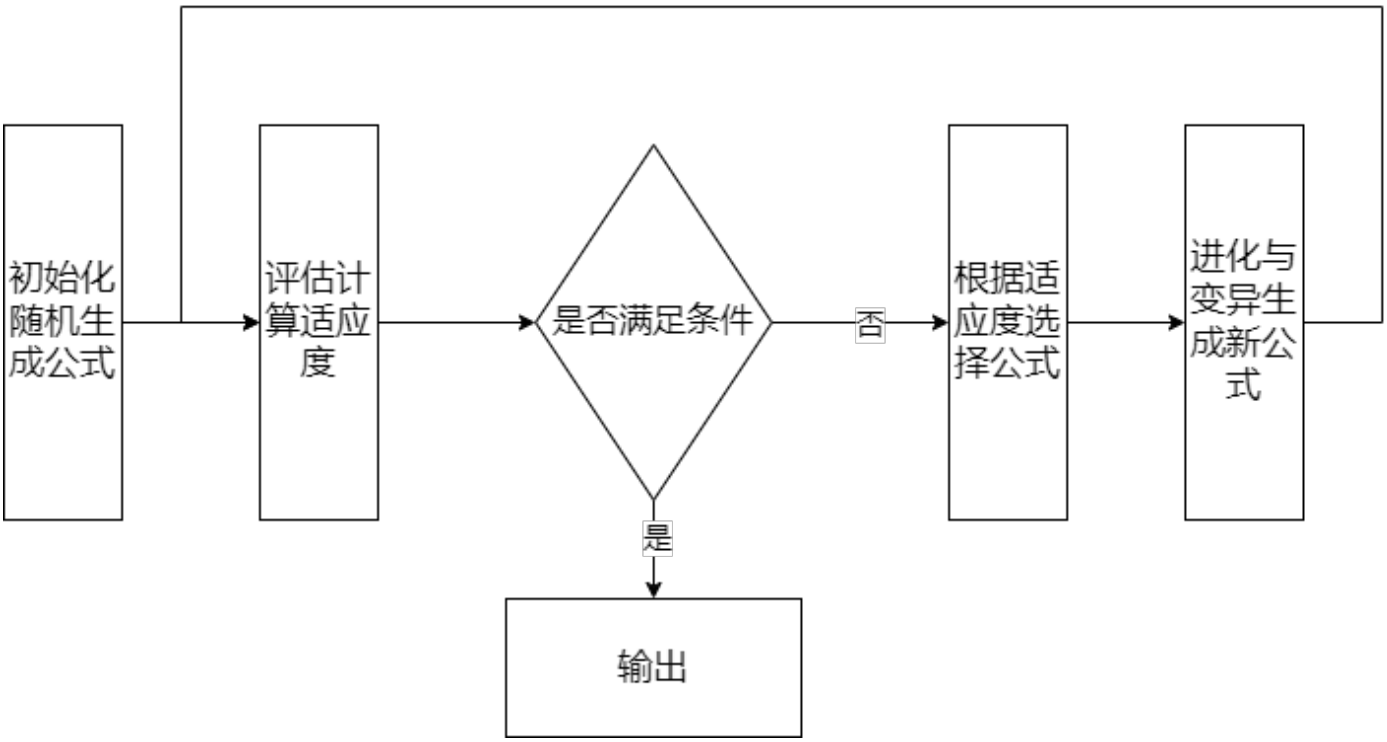

图 2-1 遗传算法总体流程

2.2 遗传算法的进化

遗传算法会将公式表示成二叉树的方式，以便于后续的进化。对于公式 $y = add(mul(a, b), cos(a))$，遗传算法会表示为图 2-2 所示的二叉树。

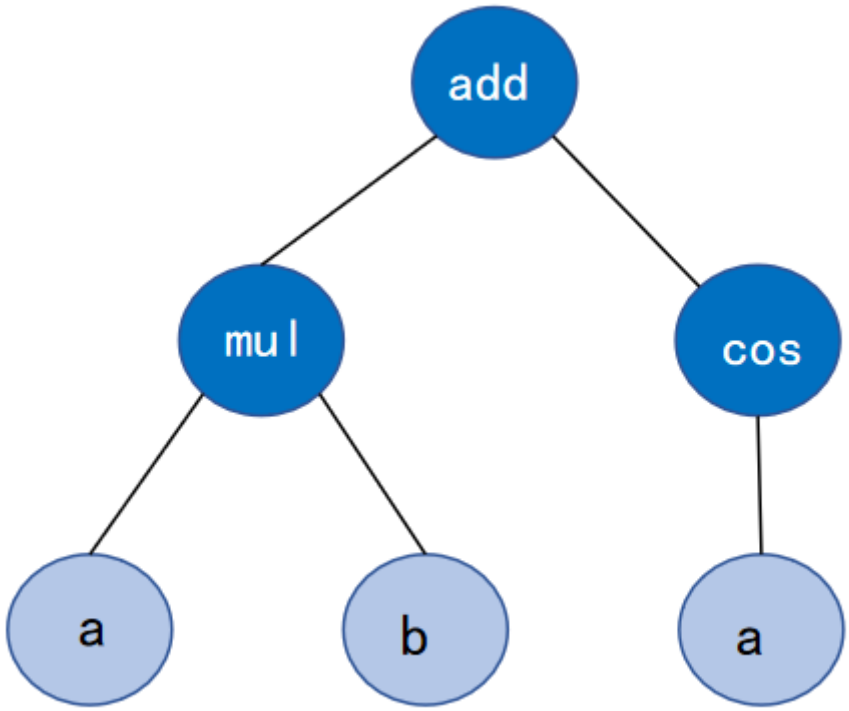

图 2-2 公式树

对于这些公式，遗传算法会依据适应度选出合适的公式作为下一代进化的父代。父代的选择，一般采用轮盘赌策略。选择出父代后，遗传算法会采用多种策略进行进化，常见的进化策略有交叉变异、子树变异、点变异、Hoist 变异等，以下将详细介绍这几种变异策略。

2.2.1 交叉变异

图 2-3 描述了交叉变异的过程。左边的树作为父代，然后从所有公式中随机选择一颗子树作为捐赠者。交叉变异会从捐赠者中选择一颗子树，与父代的子树进行替换。最终后代由父代的子树和捐赠者的子树共同构成。

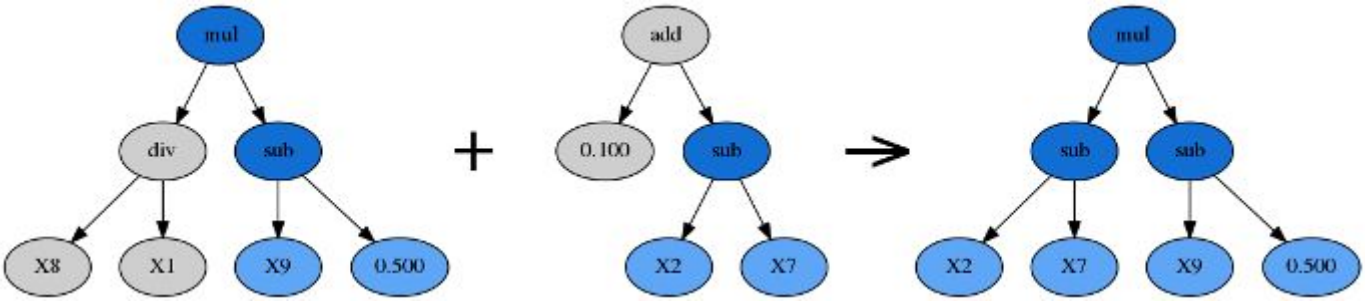

图 2-3 交叉变异

2.2.2 子树变异

图 2-4 描述了子树变异的过程。子树变异是一种激进的变异操作，左边的树作为父代，父代公式树的子树可以被完全随机生成的子树所取代。 这可以将已被淘汰的公式重新引入公式种群，以维持公式多样性。

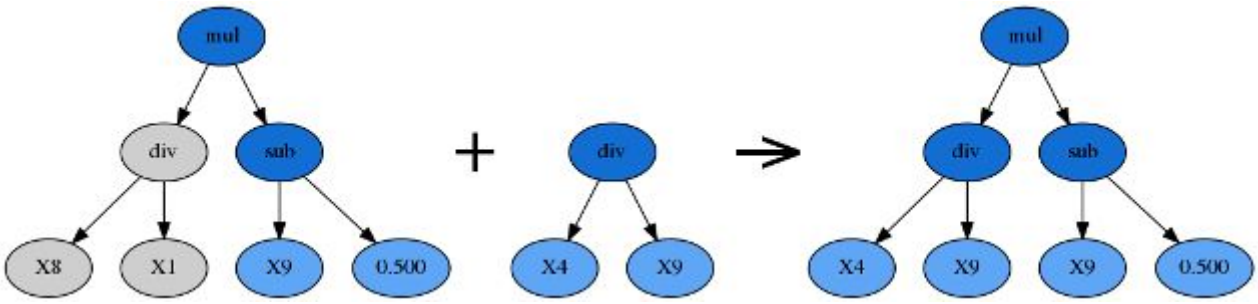

图 2-4 子树变异

2.2.3 Hoist 变异

图 2-6 描述了 Hoist 变异的过程。Hoist 变异是一种对抗公式树过于复杂的方法，左边的树作为父代，然后会从父代公式树中，随机选择子树进行删除。

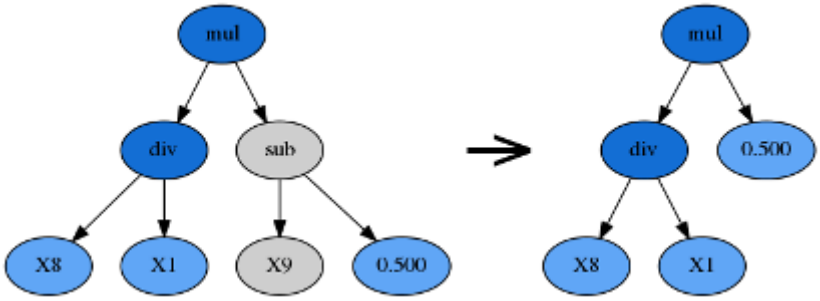

图 2-6 Hoist 变异

2.2.4 点变异

图 2-5 描述了点变异的过程。点变异选择左边的树作为父代，然后对树中的节点进行随机替换。与子树变异一样，它也可以将已淘汰的公式重新引入种群中以维持公式多样性。

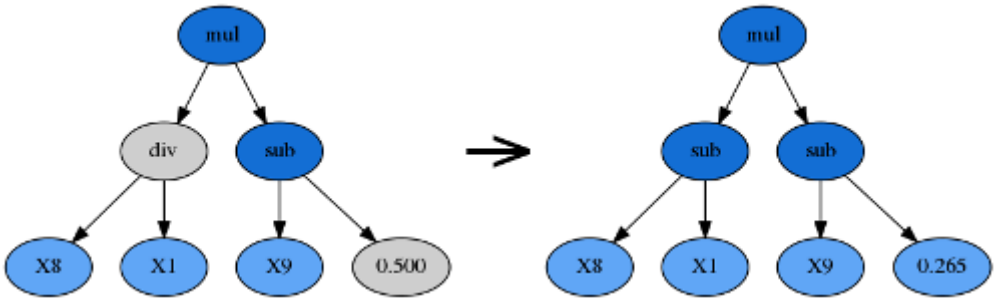

图 2-5 点变异

2.3 Python 遗传算法库 gplearn

gplearn 是一款基于 Python 的遗传算法库，其提供了类似 scikit-learn 的调用方式。下面就展示了一个利用 gplearn 进行遗传算法挖掘的例子。

```
import numpy as np
import genetic
# 设置训练参数
gp = genetic.SymbolicRegressor(population_size=1000,
generations=10, init_depth = (2,6), const_range=[-1, 1],
metric=mse, function_set=['add', 'sub', 'mul', 'div'],
p_crossover=0.9 ,p_subtree_mutation=0.01, p_hoist_mutation=0.01,
p_point_mutation=0.01)

# 开始训练，data为输入数据，target是拟合的数据
gp.gpFit(data, target)
```

上例演示了 gplearn 的使用流程，首先设置遗传算法的训练参数，然后调用`gpFit`函数开始训练，其中 *data* 为输入数据，*target* 是拟合的数据。表1展示了 gplearn 的主要参数。

参数名称	**含义**
population_size	每轮进化过程中，公式的数量
generations	进化轮次
init_depth	公式树的初始化深度
const_range	公式中常数的取值范围
metric	适应度函数
function_set	公式使用的函数集
p_crossover	交叉变异概率
p_subtree_mutation	子树变异概率
p_hoist_mutation	Hoist 变异概率
p_point_mutation	点变异概率

尽管 gplearn 作为一款成熟的遗传算法库被广泛引用，但是它在因子挖掘中面临以下问题：

1. 挖掘效率低。在挖掘因子的过程中，遗传算法需要对大量复杂公式计算适应度，计算效率较低。
2. 无法直接处理三维数据：Python gplearn 只支持从二维数据中挖掘特征。然而，在金融领域，三维数据（由时间、标的和特征构成）被广泛应用，常用于挖掘时序因子或横截面因子。如果需要支持三维数据处理，则必须对 gplearn 进行定制化改造。

第 3 章. Shark GPLearn 高性能因子挖掘

本章通过 Shark GPLearn 使用示例介绍其主要功能，并对比了 Shark 与 Python gplearn 库的性能。

3.1 Shark GPLearn 使用示例

首先通过简单的 DolphinDB 脚本介绍 Shark 自动挖掘因子的过程。

```
// 生成测试数据
def prepareData(num){
    total = num
    data=table(total:total, `a`b`c`d,[FLOAT,FLOAT,FLOAT,FLOAT])
    data[`a]=rand(1.0, total)
    data[`b]=rand(1.0, total)
    data[`c]=rand(1.0, total)
    data[`d]=rand(1.0, total)
    return data
}
def f(x){
    return x * x *x *x / (x * x *x *x + 1)
}
num = 1000000
source = prepareData(num)
predVec = (select f(a) + f(b) + f(c) + f(d) as predVec from source)[`predVec]

// 创建GPLearnEngine
engine = createGPLearnEngine(source, predVec, populationSize=1000,
generations=10, initDepth = [2,6], constRange=0, fitness="mae",
functionSet=['add', 'sub', 'mul', 'div'])

// 进行挖掘，得到因子
trainRes=engine.gpFit(programNum=5,printCorr=true)
```

上例演示了 Shark 进行因子挖掘的基本流程：

- 首先通过`createGPLearnEngine`函数创建 *GPLearnEngine*，其中 *source* 是输入数据，*predVec* 是 *target* 数据，生成的因子将会拟合 *target* 数据的分布，*populationSize* 代表训练过程中的公式数量，*generations* 代表进化的轮次，*initDepth* 代表公式树的初始化深度，*constRange* 代表公式中常数的取值范围，当 *constRange* 为 0 时，表示公式中不允许生成任何常数，*fitness* 代表使用的适应度函数，*functionSet* 代表此次训练过程中使用的函数。更多参数请参阅 Dolphin GPLearn 用户手册 | 附录。
- 然后调用`engine.gpFit(5,true)`开始挖掘因子，*programNum* 代表输出最优的 5 个公式，*printCorr* 表示同时输出 *outProgram* 个公式之间的相关性。

3.2 自定义函数

在因子挖掘时，有时候希望将上次训练得到的公式，加入到第二轮训练的 *functionSet* ，或者想对公式得到的结果进行一些预处理之后，再进行适应度函数的计算。此时可以通过使用`addGpFunction`和`setGpFitnessFunc`函数，自定义训练函数和适应度函数。

```
// 自定义训练函数
def myFunc(x , y) {
    return add(x,y)
}
addGpFunction(engine, myFunc)

// 自定义适应度函数
def myFitness(x, y) {
    return spearmanr(zscore(clip(x, med(x) - 5 * mad(x,true), med(x) + 5 *
 mad(x,true))), y)
}
setGpFitnessFunc(engine, myFitness)
```

上例展示了自定义训练函数和适应度函数的基本流程：

- 自定义训练函数。首先在 DolphinDB 脚本中，定义函数`myFunc`，`myFunc`中使用到的函数必须是 Shark 已经支持的函数的自由组合，然后调用`addGpFunction`函数进行添加。之后在因子挖掘的过程中，`myFunc`就会加入函数集 *functionSet* 中。
- 自定义适应度函数。首先在 DolphinDB 脚本中，定义函数`myFitness`，`myFitness`中使用到的函数必须是已经支持的函数的自由组合，然后调用`setGpFitnessFunc`设置适应度函数。之后在因子挖掘的过程中，就会使用`myFitness`作为适应度函数。

3.3 Shark 新特性

相比 gplearn，Shark 具有更丰富的算子库，并且提供了高效的 GPU 版本实现；并且Shark 引入分组语义，可以在训练中分组计算，从而支持在三维数据中挖掘因子；为了充分发挥 GPU 的性能， Shark 还支持单机多卡进行遗传因子挖掘。

丰富的算子库

传统的 gplearn 只支持简单有限的基础数学算子，但是这些基础算子很难挖掘出数据的深层次特征。DolphinDB 作为一款基于高性能时序数据库，支持复杂分析与流式处理的实时计算平台，内置各种滑动窗口、时序处理等丰富的函数库。Shark 将部分函数移植至 GPU，可以更高效地挖掘高质量因子。目前 Shark 支持的算子如下表所示：

函数名	描述
add(x,y)	加法
sub(x,y)	减法
mul(x,y)	乘法
div(x,y)	除法， 如果除数的绝对值小于 0.001，返回 1

mul(x,y)	乘法
div(x,y)	除法， 如果除数的绝对值小于 0.001，返回 1
max(x,y)	最大值
min(x,y)	最小值
sqrt(x)	按照绝对值开方
log(x)	取对数，如果 x 小于 0.001，将返回 0
neg(x)	相反数
reciprocal(x)	倒数，如果x 的绝对值小于0.001，将返回0
abs(x)	绝对值
sin(x)	正弦函数
cos(x)	余弦函数
tan(x)	正切函数
sig(x)	sigmoid 函数
signum(x)	返回 x 的符号标志
mcovar(x, y, n)	滑动窗口为 n 时，x 和 y 的协方差
mcorr(x, y, n)	滑动窗口为 n 时，x 和 y 的相关性
mstd(x, n)	滑动窗口为 n 时，x 的样本标准差
mmax(x, n)	滑动窗口为 n 时，x 的最大值
mmin(x, n)	滑动窗口为 n 时，x 的最小值
msum(x, n)	滑动窗口为 n 时，x 的和
mavg(x, n)	滑动窗口为 n 时，x 的平均数
mprod(x, n)	滑动窗口为 n 时，x 的积
mvar(x, n)	滑动窗口为 n 时，x 的样本方差
mvarp(x, n)	滑动窗口为 n 时，x 的总体方差
mstdp(x, n)	滑动窗口为 n 时，x 的总体标准差
mimin(x, n)	滑动窗口为 n 时，x 的最小值下标
mimax(x, n)	滑动窗口为 n 时，x 的最大值下标
mbeta(x, y, n)	滑动窗口为 n 时，x 在 y 上的回归系数的最小二乘估计
mwsum(x, y, n)	滑动窗口为 n 时，x 和 y 的内积
mwavg(x, y, n)	滑动窗口为 n 时，x 以 y 为权重的加权平均值
mfirst(x,n)	滑动窗口为 n 时，计算窗口的第一个元素
mlast(x,n)	滑动窗口为 n 时，计算窗口的最后一个元素
mrank(x,asc, n)	滑动窗口为 n 时，计算 x 在窗口内的排名
ratios(x)	返回 x(n)\x(n-1) 的值
deltas(x)	返回 x(n)-x(n-1) 的值

deltas(x)	返回 x(n)-x(n-1) 的值

在算子库中，滑动窗口函数的大小，Shark 会从用户给定的范围内随机选择。

```
engine = createGPLearnEngine(source, predVec, windowRange=[7,14,21])
```

上例展示了如何指定滑动窗口的范围的例子，Shark 会在训练时，对公式中的滑动窗口算子，从 *[7,14,21]* 中随机选取一个值作为窗口大小。

三维数据的支持

传统的 gplearn 只支持从二维数据挖掘特征，但在金融领域，常常需要利用时间、标的物和特征构成的三维数据，来挖掘时序因子或横截面因子。为了应对这种情况，Shark 提供了分组语义，允许在计算过程中进行分组计算，从而支持对三维数据进行挖掘。通过这种方式，用户可以更加灵活地处理金融数据，挖掘出更加准确和有效的因子，提升金融数据分析的质量和效率。

```
engine = createGPLearnEngine(source, predVec, groupCol="SecurityID")
```

上例展示了如何利用分组进行三维数据挖掘的例子。对于 *source* 表，它存储的是二维数据，如果在创建*GPLearnEngine* 时指定 *groupCol*，那么 *source* 表将会在训练过程中，按照 *groupCol* 进行分组间计算，从而支持三维数据的挖掘。

初始化公式

在因子挖掘时，有时需要在上一轮挖掘出的较优的因子的基础上继续挖掘。为了满足这一需求，Shark 提出了初始化公式的功能。

```
functionSet = [ 'add','sub','mul','div','sqrt','log','abs','neg','reciprocal',
                "mcorr","mcovar","mstd","mmax","mmin","msum","mavg","mbeta","mprod",
                "mvarp","mstdp","mwsum","mwavg"]
engine = createGPLearnEngine(source, predVec,
          initProgram=[<mavg(a, 10)>, <msum(b, 10)>], functionSet=functionSet)
```

上例就展示了如何初始化公式，继续挖掘的例子。对于新的 *engine*，主要指定 *initProgram* 参数，新的 *engine* 将会利用 *initProgram* 中的元代码初始化公式。除此之外，还可以通过显式设置 *functionSet* 参数，指定生成的因子所使用的公式的范围，从而挖掘出更有效的因子。

多卡支持

为了充分发挥 GPU 的性能，Shark 支持单机多卡训练。在多卡训练中，采用数据并行的模式，即将训练数据均匀分配给各个计算卡，每块卡分别计算对应的适应度。最终的适应度值为各个卡上适应度值的平均值。多卡训练能够有效利用多个 GPU 的并行计算能力，加快整个训练过程的速度，提高训练效率。

```
engine = createGPLearnEngine(source, predVec, deviceId=[0,1])
```

上例就展示了如何在 Shark 中开启多卡的训练，只需要在创建 engine 时，指定 *deviceId* 参数，其中 0 和 1 代表 GPU 的 ID。

3.4 性能对比

本节对比了 Shark 和传统的 Python gplearn 的性能。

测试在一台 CentOS Linux 的服务器上进行的，CPU 型号为 AMD EPYC 7513 32-Core ，分配内存为 32 GB，GPU 型号为 NVIDIA A800 80GB。详细配置如表 3-1 所示：

软硬件项	信息
OS（操作系统）	CentOS Linux 7 (Core)
内核	3.10.0-1160.el7.x86_64
CPU 类型	AMD EPYC 7513 32-Core Processor
CPU 逻辑核数	32 core（config 配置）
内存	512GB（config 配置）
GPU	NVIDIA A800 80GB PCIe

表3-1 测试环境配置

测试时，gplearn 与 Shark 的训练参数如表 3-2 所示：

训练参数	参数值
公式的数量	1000
进化轮次	10
公式树的初始化深度	[2, 6]
适应度函数	MSE
公式使用的函数集	`['add', 'sub', 'mul', 'div', 'sqrt','log', 'abs','neg', 'max','min', 'sin', 'cos', 'tan']`
交叉变异概率	0.9
子树变异概率	0.01
Hoist 变异概率	0.01
点变异概率	0.01
n_jobs	32， gplearn 独有参数，开启32个进程并行计算

测试数据是随机生成的 N 行 5 列的 table，且数据范围为[-1.0,1.0]，通过以下公式计算目标值：

$$res = 1/(a^4+1) + 1/(b^4+1) + 1/(c^4+1) + 1/(d^4+1) + 1/(e^4+1)$$

表 3-3 展示了不同数据规模下，gplearn 与 Shark 的运行时间对比，时间单位为**秒（s）**。

Rows	100K	1M	10M	100M
Shark	1.7	2.7	9.5	84
gplearn	15.8	64.1	822.0	5458.5

可以看到，随着数据规模的增加，Shark GPLearn 相对 gplearn 的加速比也显著提升，在 1000 万行测试数据中，达到了 86 倍的加速比，当测试数据达到1亿行，python gplearn 需要耗费 1.5 小时，而 Shark GPLearn只需要1.4 分钟。

第 4 章. Shark GPLearn 架构介绍

Shark GPLearn 是基于 Shark 异构计算平台的高性能因子挖掘功能，可以直接从 DolphinDB 中读取数据，并调用 GPU 进行自动因子挖掘、因子计算，加快投研效率，适用于对性能具有较高要求的各类应用场景。图 4-1 描述了 Shark GPLearn的基本架构。

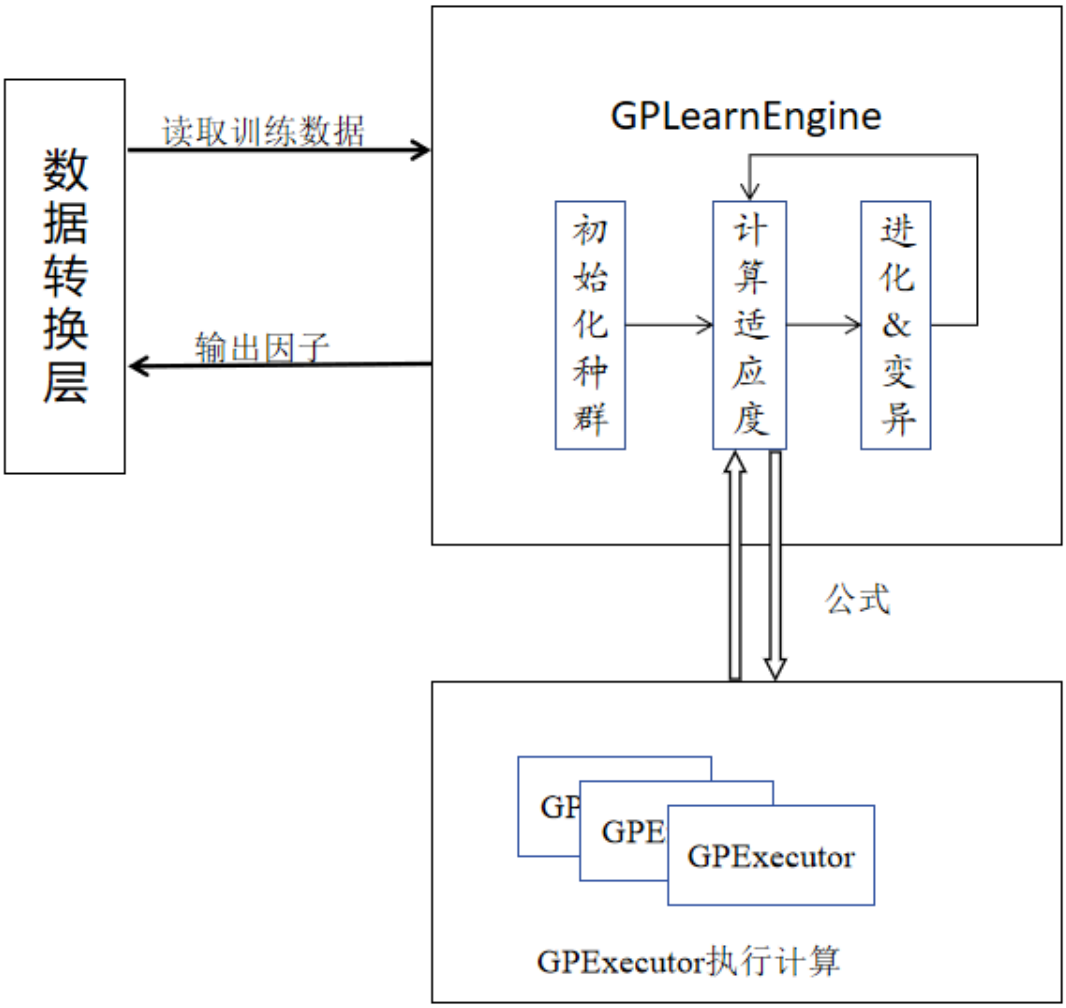

图 4-1 Shark GPLearn 自动因子挖掘基本架构

数据转换层就是沟通 GPU 与数据库、GPU 与 CPU 的桥梁。在计算开始时，数据转换层从 DolphinDB 中读取存储的数据至 CPU，然后将 CPU 上的数据复制到 GPU，以便后续的计算过程中使用。当 GPU 完成计算后，数据转换层将 GPU 中的数据复制到 CPU，然后再将其存储到 DolphinDB 中。

GPLearn 主要分为两大模块，GPLearnEngine 和 GPExecutor。其中 GPLearnEngine 主要负责训练时的调度工作，主要包括种群生成、进化和变异操作等；GPExecutor 主要负责挖掘出的因子的适应度计算。GPLearnEngine 的调度基本与 gplearn 保持一致，具体过程请参阅第2章。下面将详细介绍 GPExecutor 的执行流程。

GPExecutor 首先将因子从二叉树转换为前缀表达式的逆序形式。举例来说，对于形如$div(x, add(y, 0.15))$ 的公式，会被转换为`[0.15, y, add, x, div]`的形式。这个表达式的逆序就是因子的执行队列，其中变量和常数被称为数据算子，而函数则被称为函数算子。接下来，GPExecutor 会按顺序遍历这个执行队列。对于数据算子，它会加载相应的数据入栈；而对于函数算子，则会依据函数的参数，将相应的数据出栈，然后利用GPU

执行相应的函数，并将最终的结果入栈。最终，栈中的元素就是因子的值。得到因子的值之后，将会调用适应度函数与预测数据计算适应度。

第 5 章. 场景应用

本章节用 Shark 实现了一个基于股票日频 K 线数据的因子挖掘示例。

本章节中实现的**因子挖掘流程：**

（1）数据清洗：

- 选取 2020 年 8 月 12 日至 2023 年 6 月 19 日期间的股票日频 K 数据。
- 计算训练相关指标。
- 进行数据类型转化、空值处理等数据预处理操作。

（2）模型训练：

- `createGPLearnEngine` 创建 *GPLearn* 引擎。
- 设置模型参数，包括各种变异的概率、适应度函数等。
- 训练遗传算法模型，批量生成指定数目的因子公式。

（3）因子评价：

- 使用 Alphalens 模块计算因子不同持仓周期下的 IC 值。
- 对自动挖掘的因子进行 IC 值分析，计算 IC 序列的均值、标准差等。

5.1 导入数据

先将数据导入 DolphinDB 的数据库，完成数据准备工作。

（1）创建分布式库表

```
dbName = "dfs://stockDayKDetail"
tbName = "stockDayK"
// 创建数据库
db = database(dbName, RANGE, date(datetimeAdd(1980.01M,0..80*12,'M')))
// 创建分布式表
colNames = ["SecurityID","Exchange","TradeDate","PreClosePrice","ActPreClosePrice",
            "OpenPrice","HighestPrice","LowestPrice","ClosePrice","TurnoverVol",
            "TurnoverValue","DealAmount","PE","PE1","PB",
            "NegMarketValue","MarketValue","ChgPct","TurnoverRate","UpdateTime"]
colTypes = ["SYMBOL","SYMBOL","DATE","DOUBLE","DOUBLE",
            "DOUBLE","DOUBLE","DOUBLE","DOUBLE","LONG",
            "DOUBLE","LONG","DOUBLE","DOUBLE","DOUBLE",
            "DOUBLE","DOUBLE","DOUBLE","DOUBLE","TIMESTAMP"]
kLineSchema = table(1:0, colNames, colTypes)
```

```
createPartitionedTable(dbHandle=database(dbName), table=kLineSchema,
      tableName=tbName, partitionColumns='TradeDate')
```

分区存储

- DolphinDB 支持数据分区存储，对于不同分区数据支持并发写入，对于同一个分区数据支持并发查询。
- 分区原则：建议落在1个最小分区的数据占内存的大小约 150MB~500MB。
- 经测试日频 K 线数据采用“**时间维度按年范围分区**”的分区方案和 OLAP 存储引擎存储的综合性能最佳。

（2）导入数据

```
fileDir = "/home/test/testData/dayK/2020_2023_day_k.csv"
dbName = "dfs://stockDayKDetail"
tbName = 'stockDayK'

// 导入数据
def loadCsvData(fileDir, dbName, tbName){
    colNames =
 ["SecurityID","Exchange","TradeDate","PreClosePrice","ActPreClosePrice",
            "OpenPrice","HighestPrice","LowestPrice","ClosePrice","TurnoverVol",
            "TurnoverValue","DealAmount","PE","PE1","PB",
            "NegMarketValue","MarketValue","ChgPct","TurnoverRate","UpdateTime"]
    colTypes = ["SYMBOL","SYMBOL","DATE","DOUBLE","DOUBLE",
            "DOUBLE","DOUBLE","DOUBLE","DOUBLE","LONG",
            "DOUBLE","LONG","DOUBLE","DOUBLE","DOUBLE",
            "DOUBLE","DOUBLE","DOUBLE","DOUBLE","TIMESTAMP"]
    loadTextEx(database(dbName), tbName, `TradeDate, fileDir,
                schema=table(colNames, colTypes))
}
loadCsvData(fileDir, dbName, tbName)
```

分块导入

- DolphinDB 提供的 `loadTextEx` 函数可以将一个大的文本文件分割成很多个小块，依次将每一个文件块内的数据读取到内存并加载到指定的分布式表中。

5.2 数据预处理

```
def processData(dbName, tbName){
    // 选取 ["0", "3", "6"] 开头的 A 股数据
    idList = (select count(*) from loadTable(dbName, tbName) group by
 SecurityID).SecurityID
    idList = idList[left(idList, 1) in ["0", "3", "6"]]
    // 获取基础指标
```

```
    testData =  select SecurityID, TradeDate, OpenPrice as open, ClosePrice as
 close, LowestPrice as low,
                        HighestPrice as high, PreClosePrice as PreClose,
 ClosePrice-PreClosePrice as change,
                        ChgPct as PctChange, TurnoverVol as volume, TurnoverValue as
 amount,
                        TurnoverValue\TurnoverVol as AvgPrice,
 NegMarketValue\OpenPrice as FloatShrAmount,
                        NegMarketValue as valMv, TurnoverRate as turn,
                        move(ClosePrice, -20) as next20, move(ClosePrice, 1) as
 prev1, move(ClosePrice, 5) as prev5,
                        move(ClosePrice, 20) as prev20, move(ClosePrice, 30) as
 prev30,
                        (TradeDate-weekBegin(2017.01.03)) / 7 as weekNo
                from loadTable(dbName, tbName)
                where SecurityID in idList and TurnoverVol!=0
                context by SecurityID
    // 计算收益率和 52 周最高价/最低价
    testData = select SecurityID, TradeDate, next20\close-1 as yRet20,
                    open, close, low, high, PreClose, change, PctChange,
                    volume, amount, AvgPrice, FloatShrAmount, valMv,
                    close\prev1-1 as return_1d, close\prev5-1 as return_5d,
                    close\prev20-1 as return_20d, close\prev30-1 as return_30d,
                    turn, tmmax(weekNo, high, 52) as QuaterHigh, tmmin(weekNo, low,
 52) as QuaterLow
                from testData
                context by SecurityID
    // 删除空值
    testData = select * from testData where yRet20 != NULL and return_30d!=NULL
 order by SecurityID, TradeDate
    // 选取每天都有数据的股票
    dayNum = nunique(testData["TradeDate"])
    testData = select * from testData context by SecurityID having count(*)=dayNum
 order by SecurityID, TradeDate
    return double(testData)
}

dbName = "dfs://stockDayKDetail"
tbName = "stockDayK"
testData = processData(dbName, tbName)
```

代码说明：

- DolphinDB 内置 1600+ 的函数，可以覆盖金融领域绝大部分数据处理和分析的需求。
- 本例中，使用 `context by` 语句实现了分组计算；使用 `move` 函数实现数据偏移，获取指定日期的收盘价；使用 `tmmax` 和 `tmmin` 函数计算滑动窗口内的最高价/最低价。

- GPLearnEngine的输入数据不允许出现空值，所以在预处理的时候需要对空值进行删除或者填充的处理。本例中，将包含空值的行进行了过滤删除的处理。
- 除了分组列以外，GPLearnEngine的输入数据必须全部为 FLOAT 或 DOUBLE 类型。因为原始数据中成交量属于 LONG 类型的列，所以需要进行类型转化。本例中，用 double(testData) 的方式将表里所有数值类型的列转化为 DOUBLE 类型。

5.3 训练模型

```
xCols = ["return_1d", "return_5d", "return_20d", "return_30d", "turn", "QuaterHigh",
 "QuaterLow",
       "open", "close", "low", "high", "PreClose", "change", "PctChange",
       "volume", "AvgPrice", "FloatShrAmount", "valMv"]
yCol="yRet20"
idCol="SecurityID"
dateCol="TradeDate"
testSize = 0.8

// 拆分训练集和测试集
dateList = exec distinct TradeDate from testData order by TradeDate
splitDay = dateList[int(size(dateList)*testSize)]
trainDate = dateList[dateList<splitDay]
testDate = dateList[dateList>=splitDay]

// 获取训练集
dateColData = sql(select=sqlCol(dateCol), from=testData, where=expr(sqlCol(dateCol),
 in , trainDate), exec=true).eval()
trainX = sql(select=sqlCol(idCol<-xCols), from=testData, where=expr(sqlCol(dateCol),
 in , trainDate)).eval()
targetY = sql(select=sqlCol(yCol), from=testData, where=expr(sqlCol(dateCol), in ,
 trainDate), exec=true).eval()

// 配置算子库
functionSet = [ "add", "sub", "mul", "div", "abs", "sqrt", "log", "reciprocal",
 "max", "min", "tan",
               "mprod", "mavg", "msum", "mstdp", "mmin", "mmax", "mimin", "mimax",
 "mcorr", "mrank"]

// 创建 GPLearnEngine 引擎
engine = createGPLearnEngine(trainX,targetY,groupCol=idCol,
                populationSize=1000, generations=3, tournamentSize=20, functionSet =
 functionSet,
                initDepth=[1, 4], windowRange=[5,10,20,40,60,120,180], constRange=0,
 seed=123,
                crossoverMutationProb=0.6, subtreeMutationProb=0.01,
                hoistMutationProb=0.01,pointMutationProb=0.01,
                parsimonyCoefficient=0.0, minimize=false, deviceId=3)
```

```
// 自定义适应度函数
def myFitness(x, y, groupCol) {
  return mean(groupby(spearmanr, [x, y], groupCol))
}
setGpFitnessFunc(engine, myFitness, dateColData)

// 挖掘因子
timer res = engine.gpFit(50)
```

代码说明：

- 按照 *testSize* 的比例，将数据集拆分成了训练集和测试集。即取前 80% 的日期的数据作为训练集，剩下 20% 日期的数据作为测试集。
- 使用 `sql` 函数动态生成 SQL 语句，查询指定列（*xCols*）作为模型输入数据。
- 可以通过 *seed* 参数指定训练时的随机数种子。*seed* 相同时，每次训练出的结果一致。
- 可以通过 *functionSet* 参数指定生成公式的算子库。Shark GPLearn 目前支持的函数可以参考：Dolphin GPLearn 用户手册 | 附录
- 除了基础的数学公式，Shark 还实现了 m 系列的滑动窗口函数。配合 *groupCol* 参数，可以实现分组计算时序因子。比如本例中，按照股票代码分组，计算每只股票的滑动聚合值。
- Shark 支持自定义适应度函数。适应度函数有两种设置方式：
  - 通过 `createGPLearnEngine` 函数中的 *fitnessFunc* 参数选择内置的适应度函数。比如 *fitnessFunc="rmse"*，表示使用均方根误差。目前能支持的内置适应度函数可以参考：Dolphin GPLearn 用户手册 | 3.1 createGPLearnEngine
  - 通过 `setGpFitnessFunc` 函数，将用户的自定义函数作为模型的适应度函数。比如本例中，通过`spearmanr`、`groupby`、`mean` 三个函数实现了因子 rankIC 的计算逻辑，生成了一个自定义适应度函数。其中变量 *x* 表示根据生成的公式计算获得的因子值；变量 *y* 表示目标数据的真实值；*groupCol* 表示计算适应度函数时的分组列；`groupby(spearmanr, [x, y], groupCol)` 表示按照 *groupCol* 对变量 *x* 和 *y* 计算 *Spearmanr* 相关系数，即计算每天的 rankIC 值；最后通过 `mean` 对 rankIC 序列求平均值。目前，自定义的适应度函数由 GPU 支持的算子组成。
- 使用 `minimize` 参数控制适应度进化方向，当`minimize`为false时，优化目标是**最大化**适应度评分。
  - 对于 *mse* 均方误差等计算误差的适应度函数，优化目标应该是最小化。
  - 对于 *pearson* 皮尔逊矩阵相关系数等计算相关系数的适应度函数，优化目标是最大化。
- `createGPLearnEngine` 函数只是初始化并设置模型，只有执行 `gpFit` 时，模型才开始真正的训练。上例中 `gpFit(50)` 表示挑选最优的 50 个因子。通过修改对应参数，可以通过 Shark 快速挖掘出大量的因子。
- 使用 `timer` 函数统计程序运行耗时。本例中，测试数据集约 166W 行，包含 3002 只股票 2020.08.12~2022.11.23 共 554 天的数据。训练 50 个因子的耗时约 9 秒。

• 图 5-1 是挖掘出的因子公式示例：

program
mul(mul(turn, PreClose), msum(QuaterHigh, 5))
mul(turn, sub(high, AvgPrice))
mul(mul(mul(turn, QuaterHigh), abs(AvgPrice)), PreClose)
mul(mul(mul(turn, PreClose), abs(AvgPrice)), QuaterHigh)
mul(mul(mul(turn, QuaterHigh), close), PreClose)
mul(turn, mul(turn, QuaterHigh))
mul(mul(mul(turn, PreClose), abs(AvgPrice)), PreClose)
mul(mul(mul(turn, PreClose), abs(AvgPrice)), abs(AvgPrice))
mul(mul(mul(turn, PreClose), close), PreClose)
mmax(mul(turn, PreClose), 10)

图 5-1 挖掘出的部分因子

5.4 因子评价

通过 Shark 可以快速挖掘出大量的因子公式，但并不是所有的因子都是有效的。在因子正式被使用前，往往需要经过单因子评价、多因子回测等等步骤。

Alphalens 是 Quantopian 用 Python 开发的一个因子分析（评价）工具包，DolphinDB 已基于同样逻辑开发了同名的模块。通过 Alphalens 模块，可以计算出不同持仓周期下的因子 IC 值序列。

• 计算 IC 序列的均值 —— 因子显著性

• 计算 IC 序列的标准差——因子稳定性

• 计算 IC 序列均值与标准差的比值（IR）——因子有效性

本章节主要结合 DolphinDB 开发的 Alphalens 模块使用 IC 值分析法进行简单的因子评价。

```
use alphalens

// 计算单因子
def calFactor(testData, factorStr, idCol="SecurityID", dateCol="TradeDate"){
    factorTb = sql(
        select=[sqlColAlias(sqlCol(dateCol), "date"), sqlColAlias(sqlCol(idCol),
 "asset"), sqlColAlias(parseExpr(factorStr), "factor")],
        from=testData,
        groupBy=sqlCol(idCol),
        groupFlag=0).eval()
    // 处理空值
    result = select date, asset, factor.nullFill(avg(factor)) as factor from
 factorTb context by date having count(factor) != 0
    return result
```

```
}

// 单因子分析
def single_factor_analysis(testData, factorStr, dayClose, trainDate, testDate) {
    result = calFactor(testData, factorStr)
    if (result.rows() == 0) {
        name =
 ["program","Returns","Mean_IC","Std_IC","Mean_Std_IC_Ratio","In_Mean_IC","In_Std_IC
","In_Mean_Std_IC_Ratio","Out_Mean_IC","Out_Std_IC","Out_Mean_Std_IC_Ratio"]
        type =
 ["STRING","STRING","DOUBLE","DOUBLE","DOUBLE","DOUBLE","DOUBLE","DOUBLE","DOUBLE","
DOUBLE","DOUBLE"]
        return table(1:0, name, type)
    }
    // 获取中间分析结果
    cleanFactorAndForwardReturns = get_clean_factor_and_forward_returns(
            factor=result,
            prices=dayClose,
            groupby=NULL,
            binning_by_group=false,
            quantiles=[0, 0.2, 0.4, 0.6, 0.8, 1.0],  // 分5组
            bins=NULL,
            periods=[1, 5, 20],  // 持仓周期
            filter_zscore=NULL,  // 异常值过滤
            groupby_labels=NULL,
            max_loss=1.00,
            zero_aware=false,
            cumulative_returns=true  // 累计收益
        )
    // 计算IC平均值和标准差
    getIC = create_information_tear_sheet(cleanFactorAndForwardReturns,
 group_neutral=false, by_group=false)
    numericCol = getIC["ic"].columnNames()[1:]
    meanIC = each(mean, getIC["ic"][numericCol])
    stdIC = each(stdp, getIC["ic"][numericCol])
    // 样本内IC平均值和标准差
    inData = select * from getIC["ic"] where date in trainDate
    inMeanIC = each(mean, inData[numericCol])
    inStdIC = each(stdp, inData[numericCol])
    // 样本外IC平均值和标准差
    outData = select * from getIC["ic"] where date in testDate
    outMeanIC = each(mean, outData[numericCol])
    outStdIC = each(stdp, outData[numericCol])
    // 整理成表
    result_table = table(take(factorStr, size(numericCol)) as program, numericCol as
 Returns,
                round(abs(meanIC), 3) as Mean_IC, round(stdIC, 3) as Std_IC,
 round(abs(meanIC)/stdIC, 3) as Mean_Std_IC_Ratio,
```

```
                round(abs(inMeanIC), 3) as In_Mean_IC, round(inStdIC, 3) as
 In_Std_IC, round(abs(inMeanIC)/inStdIC, 3) as In_Mean_Std_IC_Ratio,
                round(abs(outMeanIC), 3) as Out_Mean_IC, round(outStdIC, 3) as
 Out_Std_IC, round(abs(outMeanIC)/outStdIC, 3) as Out_Mean_Std_IC_Ratio)
    return result_table
}

// 获取每天收盘价
dayClose = select ClosePrice as close
            from loadTable("dfs://stockDayKDetail", "stockDayK")
            pivot by TradeDate as date, SecurityID

// 调用自定义函数对批量获得的因子进行 IC 值分析
resIC = peach(single_factor_analysis{testData, , dayClose, trainDate, testDate},
 res["program"]).unionAll(false)
```

代码说明：

- 调用 `gpFit` 函数进行模型训练，返回一个表，其中第一列为 STRING 类型表示挖掘出来的因子公式。想要使用这些公式进行因子计算，可以考虑以下两种方式：
  - 调用 `gpPredict` 函数，用 GPU 计算因子值。使用 `gpPredict` 函数时，需要传入训练的模型 *engine*、输入数据 *input* 和一个整形标量 *programNum*，表示对于 *engine* 训练出来的前 *programNum* 个公式，代入 *input* 数据的 GPU 计算结果。最终会返回一个 *programNum* 列的表，每一列表示一个因子值。需要注意的是，使用这种方式时，只能按顺序计算前 *programNum* 个因子，无法跳过前面的因子直接计算指定序号的因子。
  - 将字符串转化为可执行代码，用 CPU 计算因子值。比如本例中的 `calFactor` 函数，先通过 `parseExpr` 函数将表示公式的字符串转化为元代码；再使用 `sql` 函数，动态生成 sql 语句；最后通过 `eval` 函数执行元代码，以此获得对应公式的因子值。需要注意的是，GPU 实现的都是保护性算子，对于 `div` 、`log` 等算子产生的空值均做了填 0 处理；但 CPU 算子计算时，不会做类似的处理，允许空值的存在。
- 为了能更好地进行因子评价，上述代码对因子结果进行了一个空值处理。若当天所有股票的因子值为空值，那么忽略这一天的数据。
- 通过调用 Alphalens 模块中的 `get_clean_factor_and_forward_returns` 函数进行数据处理获得单因子分析中间结果；再调用 `create_information_tear_sheet` 函数计算因子的 IC 值。具体可以参考教程：Alphalens 在 DolphinDB 中的应用：因子分析建模实践
- 在 5.3 章节，将数据按照一定比例划分为训练集和测试集。上述代码中，inData 为训练集对应的结果，即样本内数据；outData 为测试集对应的结果，即为样本外数据。分别对全量数据、样本内数据、样本外数据，进行了 IC 值分析。
- `single_factor_analysis` 函数只能对单个因子进行分析。这里选择用 `peach` 函数将 `single_factor_analysis` 应用到所有因子公式上。
- 一般来说，对于 IC 值大于 0.03 且 IR 值大于 0.5 的单因子，可以初步认为其有效并考虑让其进行下一步的多因子回测。

• 图 5-2 是 5.3 章节中挖出的第一个因子的 IC 值：

program	Returns	Mean_IC	Std_IC	Mean_Std_IC_Ratio	In_Mean_IC	In_Std_IC	In_Mean_Std_IC_Ratio	Out_Mean_IC	Out_Std_IC	Out_Mean_Std_IC_Ratio
mmax(mul(turn, PreClose), 10)	forward_returns_1D	0.053	0.167	0.315	0.054	0.173	0.31	0.048	0.143	0.34
mmax(mul(turn, PreClose), 10)	forward_returns_5D	0.08	0.166	0.484	0.084	0.172	0.489	0.065	0.138	0.47
mmax(mul(turn, PreClose), 10)	forward_returns_20D	0.126	0.176	0.718	0.137	0.18	0.764	0.082	0.151	0.545

图 5-2 因子的 IC 值

5.5 模型优化

了能更快地挖掘出更有效的因子，往往需要用户对模型进行反复的调参尝试。

下面将在 5.3 章节的模型基础上，针对进化轮次、节俭系数、初始化公式等参数进行详细地对比实验。

5.5.1 进化轮次 generations

进化轮次 *generations* 是模型训练中最常见的参数，在 GPLearn 中代表种群进化的代数。以下是不同世代数时，因子的适应度和长度情况：

世代（generations）	种群		最佳个体	
	因子平均长度（Average length）	平均适应度（fitness）	因子长度（length）	适应度（fitness）
3	3.88	0.178599	5	0.186051
4	6.06	0.179865	9	0.187095
5	5.75	0.182020	11	0.187934
6	8.90	0.183746	21	0.188009

从上表可以看出：

- 迭代次数越多，适应度越高，但增长幅度有限。
- 迭代次数越多，因子长度越长，公式复杂度增高，失去可解释性。

5.5.2 节俭系数 parsimonyCoefficient

随着进化次数的增加，公式会膨胀，变得难以解释。

因子的节俭系数、长度和适应度的相关公式为：适应度 = *fitness* + 节俭系数*长度

节俭系数会惩罚过长的公式，所以可以考虑调节参数节俭系数控制公式的膨胀。

节俭系数设置为正数时，公式越长，适应度评分越高。当优化目标是最大化时，表示希望公式长度增加；当优化目标是最小化时，表示希望公式长度减少。

节俭系数设置为负数时，公式越长，适应度评分越低。当优化目标是最大化时，表示希望公式长度减少；当优化目标是最小化时，表示希望公式长度增加。

本例属于最大化优化目标，且不希望公式过长的情况，因此只测试 *generations=6* 时 *parsimonyCoefficient* ≤ *0* 的情况。

节俭系数（parsimonyCoefficient）	种群		最佳个体	
	因子平均长度（Average length）	平均适应度（fitness）	因子长度（length）	适应度（fitness）
0	8.90	0.183746	21	0.188009
-0.0001	7.21	0.178672	15	0.189786
-0.0005	4.90	0.181288	9	0.187095
-0.001	3.05	0.179668	3	0.181851
-0.01	1.01	0.168936	1	0.169338

从上表可以看出：

- 设置节俭系数可以有效地抑制公式膨胀。
- 节俭系数的绝对值越大，公式长度对适应度的影响越大。当设置过大时，长度影响甚至超过`fitness`函数的影响，不利于挖掘有效的因子。

5.5.3 初始化公式 initProgram

Shark GPLearn 支持设置初始化公式。用户可以把已经证实有效的因子设置为初始公式。基于已知的有效因子进行进化和变异，减少初始公式的随机性，能更有目标地进行因子挖掘。

从国泰君安 191 因子库中，选择了部分因子作为初始化公式。具体可以参考：国泰君安 191 Alpha 因子库。

```
// 配置算子库
functionSet = [ "add", "sub", "mul", "div", "abs", "sqrt", "log", "reciprocal",
 "max", "min", "tan",
         "mprod", "mavg", "msum", "mstdp", "mmin", "mmax", "mimin", "mimax",
 "mcorr", "mrank"]

// 配置初始化公式
alphaFactor = {
    "gtjaAlpha70": <mstd(volume * AvgPrice, 6)>
}
initProgram = take(alphaFactor.values(), 1000)

// 创建 GPLearnEngine 引擎
engine = createGPLearnEngine(trainX,targetY,groupCol=idCol,initProgram=initProgram,
                populationSize=1000, generations=4, tournamentSize=20, functionSet =
 functionSet,
                initDepth=[1, 4], windowRange=[5,10,20,40,60,120,180], constRange=0,
 seed=123,
                crossoverMutationProb=0.6, subtreeMutationProb=0.01,
```

```
                hoistMutationProb=0.01,pointMutationProb=0.01,
                parsimonyCoefficient=-0.0005, minimize=false, deviceId=3)

// 自定义适应度函数
def myFitness(x, y, groupCol) {
return mean(groupby(spearmanr, [x, y], groupCol))
}
setGpFitnessFunc(engine, myFitness, dateColData)

// 挖掘因子
timer res = engine.gpFit(50)
```

代码说明：

- 相比于 5.3 章节中的代码，本例新增了初始化公式的部分。
  - 这里选择了国泰君安 191 系列中的 70 号因子作为初始公式。
  - 当初始化的*initProgram*中公式的数量小于参数 *populationSize*时，*initProgram*中的公式和随机生成的 populationSize - size(initProgram) 个公式，共同组成初始公式。
  - `initProgram = take(alphaFactor.values(), 1000)` 这行代码里，通过 `take` 函数将有效因子反复构造 *populationSize* 个，即初始化的公式里只有指定的有效因子。以此达到必定基于指定的公式进行进化的目的。
- 初始因子的 IC 值分析如图 5-3 所示：

	program	Returns	Mean_IC	Std_IC	Mean_Std_IC_Ratio	In_Mean_IC	In_Std_IC	In_Mean_Std_IC_Ratio	Out_Mean_IC	Out_Std_IC	Out_Mean_Std_IC_Ratio
0	mstd(volume * AvgPrice, 6)	forward_returns_1D	0.056	0.135	0.414	0.056	0.137	0.407	0.057	0.126	0.449
1	mstd(volume * AvgPrice, 6)	forward_returns_5D	0.084	0.139	0.605	0.084	0.138	0.607	0.083	0.139	0.6
2	mstd(volume * AvgPrice, 6)	forward_returns_10D	0.1	0.143	0.7	0.101	0.139	0.729	0.097	0.16	0.607

图 5-3 初始因子的 IC 值

- 上述模型输出的前 10 个因子如图 5-4 所示：

	program
0	mul(mul(mul(volume, AvgPrice), mul(mul(volume, AvgPrice), AvgPrice)), AvgPrice)
1	mul(mul(volume, mul(mul(mul(volume, AvgPrice), AvgPrice), AvgPrice)), AvgPrice)
2	mul(mul(volume, AvgPrice), AvgPrice)
3	mul(mul(mul(volume, AvgPrice), AvgPrice), AvgPrice)
4	mul(mul(volume, AvgPrice), mmin(low, 5))
5	mstd(mul(mul(volume, mul(mul(mul(volume, AvgPrice), AvgPrice), AvgPrice)), AvgPrice), 6)
6	mul(mul(mul(volume, AvgPrice), AvgPrice), mul(volume, AvgPrice))
7	mul(mul(volume, AvgPrice), mul(mul(volume, AvgPrice), AvgPrice))
8	mul(mul(volume, mul(mul(volume, AvgPrice), AvgPrice)), AvgPrice)
9	mul(mul(mul(mul(volume, AvgPrice), AvgPrice), AvgPrice), AvgPrice)

图 5-4 挖掘出的因子

- 输出的第一个因子的 IC 值分析如图 5-5 所示：

	program	Returns	Mean_IC	Std_IC	Mean_Std_IC_Ratio	In_Mean_IC	In_Std_IC	In_Mean_Std_IC_Ratio	Out_Mean_IC	Out_Std_IC	Out_Mean_Std_IC_Ratio
0	mul(mul(mul(volume, AvgPrice), mul(mul(volume, AvgPrice), AvgPrice)), AvgPrice)	forward_returns_1D	0.056	0.167	0.334	0.057	0.173	0.329	0.052	0.141	0.366
1	mul(mul(mul(volume, AvgPrice), mul(mul(volume, AvgPrice), AvgPrice)), AvgPrice)	forward_returns_5D	0.085	0.169	0.501	0.088	0.174	0.503	0.073	0.148	0.498
2	mul(mul(mul(volume, AvgPrice), mul(mul(volume, AvgPrice), AvgPrice)), AvgPrice)	forward_returns_10D	0.106	0.176	0.602	0.111	0.178	0.622	0.087	0.166	0.524

图 5-5 因子的 IC 值

- 经比较可以发现：
  - 相较于 *initProgram*中使用的因子，样本内 IC 值有提升。
  - 因为初始公式只有一个，导致生成的因子都很相似，组成单一。

针对上述因子组成单一的问题，可以考虑通过以下两种方式丰富初始化种群：

- 增加初始化公式
- 调节 size(initProgram) \ populationSize 的比例，让初始种群里加入随机生成的公式

```
alphaFactor = {
    "gtjaAlpha177": <(20 - (19 - mimax(high, 20))) / 20 * 100>,
    "gtjaAlpha118": <msum(high - open, 20) / msum(open - low, 20) * 100>,
    "gtjaAlpha168": <-1 * volume / mavg(volume, 20)>,
    "gtjaAlpha31": <(close - mavg(close, 12)) / mavg(close, 12) * 100>,
    "gtjaAlpha34": < mavg(close, 12) / close>,
    "gtjaAlpha66": <(close - mavg(close, 6)) / mavg(close, 6) * 100>,
    "gtjaAlpha140": <-1*mcorr(open, volume, 10)>,
    "wqalpha101": <((close - open) / (high - low + 0.001))>
}
initProgram = take(alphaFactor.values(), 500)
```

代码说明：

- 初始化公式由 1 个增加到 8 个。
- 初代种群的 1000 个个体中，500 个是指定的初始化公式，500 个随机生成。

- 输出的前 10 个因子如图 5-6 所示：

	program
0	mul(turn, QuaterHigh)
1	sqrt(mul(turn, PreClose))
2	mul(turn, PreClose)
3	reciprocal(mul(turn, PreClose))
4	mul(mul(turn, QuaterHigh), QuaterHigh)
5	mul(mul(turn, PreClose), QuaterHigh)
6	mul(mul(turn, QuaterHigh), PreClose)
7	mul(mul(turn, PreClose), PreClose)
8	mul(turn, msum(QuaterHigh, 5))
9	mul(mul(turn, PreClose), msum(QuaterHigh, 5))

图 5-6 挖掘出的因子

- 经过筛选，表现比较好的因子如图 5-7 所示：

	program	Returns	Mean_IC	Std_IC	Mean_Std_IC_Ratio	In_Mean_IC	In_Std_IC	In_Mean_Std_IC_Ratio	Out_Mean_IC	Out_Std_IC	Out_Mean_Std_IC_Ratio
0	mul(turn, msum(QuaterHigh, 5))	forward_returns_1D	0.056	0.163	0.342	0.057	0.168	0.34	0.051	0.142	0.356
1	mul(turn, msum(QuaterHigh, 5))	forward_returns_5D	0.079	0.168	0.472	0.083	0.174	0.477	0.064	0.14	0.459
2	mul(turn, msum(QuaterHigh, 5))	forward_returns_20D	0.123	0.179	0.686	0.132	0.183	0.722	0.083	0.153	0.543
3	mul(turn, mstdp(change, 20))	forward_returns_1D	0.059	0.169	0.347	0.061	0.175	0.346	0.052	0.145	0.361
4	mul(turn, mstdp(change, 20))	forward_returns_5D	0.084	0.167	0.499	0.088	0.174	0.506	0.067	0.139	0.484
5	mul(turn, mstdp(change, 20))	forward_returns_20D	0.128	0.18	0.714	0.14	0.183	0.761	0.085	0.158	0.539
6	msum(mul(turn, PreClose), 20)	forward_returns_1D	0.045	0.177	0.253	0.046	0.183	0.253	0.039	0.154	0.256
7	msum(mul(turn, PreClose), 20)	forward_returns_5D	0.07	0.174	0.403	0.075	0.18	0.414	0.053	0.148	0.36
8	msum(mul(turn, PreClose), 20)	forward_returns_20D	0.119	0.183	0.652	0.132	0.187	0.704	0.072	0.159	0.452
9	mul(mvar(PreClose, 10), turn)	forward_returns_1D	0.051	0.169	0.3	0.052	0.176	0.296	0.046	0.142	0.324
10	mul(mvar(PreClose, 10), turn)	forward_returns_5D	0.076	0.17	0.448	0.081	0.177	0.459	0.056	0.136	0.41
11	mul(mvar(PreClose, 10), turn)	forward_returns_20D	0.117	0.178	0.66	0.128	0.182	0.706	0.073	0.152	0.479

图 5-7 表现较好的因子

5.5.4 其他参数

根据模型实现，用户进行参数调整时，可以参考以下说明：

- **populationSize** 每代种群种的公式数量。公式数量越多，消耗的算力越多，公式之间自由组合的空间越大。
- **generations** 进化的代数。进化代数越多，消耗的算力越大。进化代数过多，容易出现过拟合。
- **tournamentSize** 生成下一代公式时，参与竞争的公式数量。tournamentSize 越小，随机选择的范围越小。
- **constRange** 公式中包含的常量的范围。可以设置范围，避免出现一些无意义的常数。
- **windowRange** 滑动窗口函数的窗口大小取值范围。可以设置范围，避免出现一些无意义的窗口范围。
- **initDepth** 初始化公式树的深度范围。深度越深，公式越复杂，越难以解释。
- **initProgram** 初始化公式。指定某些公式进入初代种群，可以直接基于有效因子进行进化个变异。

- **functionSet** 初始化公式树和进化时选择的算子。算子库越丰富，自由组合的空间越大；但无意义的算子可能会降低挖掘效率。
- **fitnessFunc** 适应度函数，控制公式的进化方向。
- **parsimonyCoefficient** 节俭系数，会惩罚过长的公式，防止公式越来越膨胀，失去可解释性。
- **eliteCount** 精英数量，适应度最优的 eliteCount 个公式，会作为精英直接传递给下一代。
- **crossoverProb** 进行交叉变异的概率。交叉变异属于比较有效的进化方式。
- **subtreeMutationProb** 进行子树变异的概率。随机性较大，变异结果一般不稳定。
- **hoistMutationProb** 进行 hoist 变异的概率 。防止公式越来越复杂的一种变异方式，当公式深度较深时，可以提高 hoist 变异的概率。
- **pointMutationProb** 进行点变异的概率。随机性较大，变异结果一般不稳定。

第 6 章. 未来规划

6.1 适配国产计算卡

目前 Shark 高性能因子挖掘平台仅支持 NVIDIA 的 GPU。尽管 NVIDIA 的 GPU 在性能和稳定性方面表现出色，但其面临禁售等问题。Shark 高性能因子挖掘平台未来会适配更多国产计算卡，从而更好地满足信创需求，降低用户成本。

6.2 更多算子和更灵活的语义

目前 Shark 仅支持 DolphinDB 内置的部分算子。未来 Shark 将会扩展对更多数据分析算子的支持，并且允许用户通过脚本语言定义更加灵活的用户自定义函数。这允许用户能够利用更多丰富的算子进行数据分析和处理，从而更好地满足不同应用场景下的需求。