SIEMENS

Отчёт по заданию на стажировку. Градиентный спуск. Java+Spark.

Выполнил Ощепков Артём
semitro_8@mail.ru
github.com/semitro/GradientDescent

Санкт-Петербург 2018

Задание

Реализовать распределенную версию градиентного спуска в Apache Spark на Java.

Описание программы

Градиентный спуск реализован дважды: сначала в последовательной, а затем и в параллельной версиях. В обоих количество коэффициентов θ произвольно.
На первом шаге алгоритма всем θ присваивается значение 0. Далее θ_i изменяются сильнее или слабее в зависимости от величины частной производной.
Ошибка на каждой следующей итерации оценивается как средняя разность между θ_j и θ_{j+1}. Алгоритм останавливается тогда, когда ошибка становится меньше указанной точности или если она увеличилась в сравнении с предыдущей.
Отмечу, что функция ошибок растёт с увеличением размера набора данных и не может быть минимизирована меньше определённого порога, так как модель линейной регрессии не может точно аппроксимировать произвольный набор данных.

Запуск

1. Собрать jar

git clone https://github.com/semitro/GradientDescent
cd GradientDescent
mvn clean compile assembly:single

2. Положить поближе к нему датасеты

cp ./src/main/resources/*csv ./target

3. Запустить

cd ./target

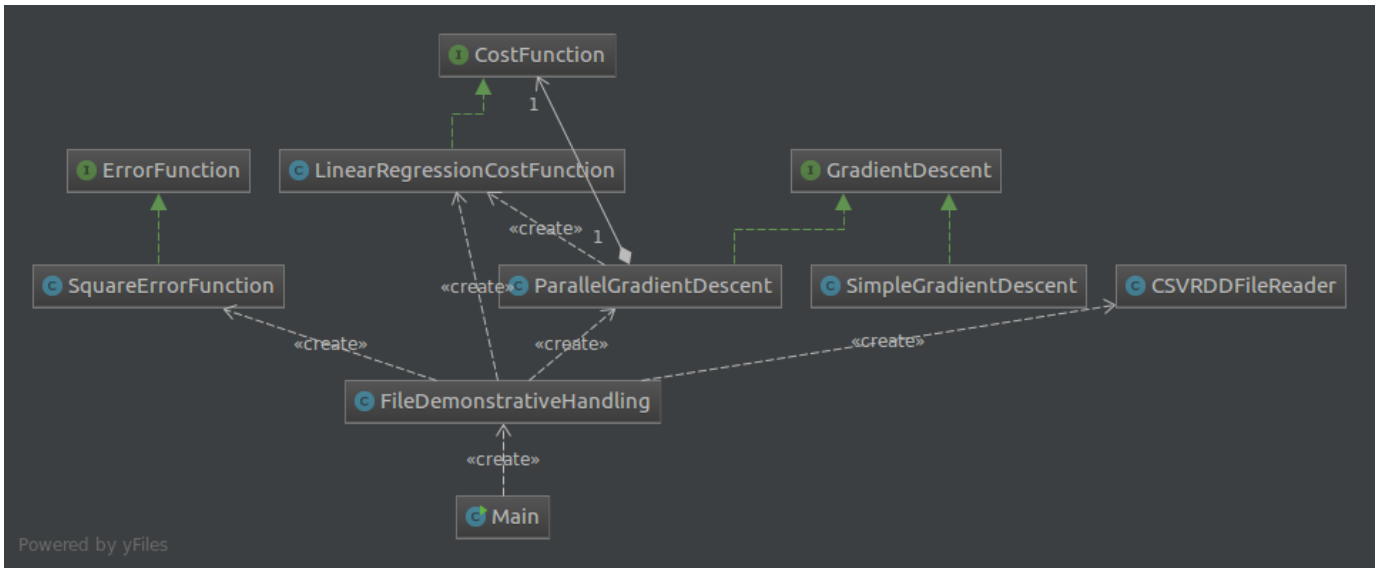
java -jar ./GradientDescent*.jar dataset.csv epsilon speed

java -jar ./GradientDescent.jar dataset1.csv 0.1 0.0005*

4. Для запуска юнит-тестов

mvn test

Диаграмма классов



Тестирование

Написано 3 категории Unit-тестов:

- Подбор коэффициентов, хранящихся в памяти для последовательной и параллельной реализации.

К примеру, для теста { {2.0, 2.0, 4.0}, {4.0, 4.0, 8.0} результат должен быть примерно { 1.0, 1.0, 0.0 }, т.к. 4 = 1*2 + 1*2 + 0

- Работа с csv-файлами на для параллельного спуска:
  dataset1.csv (284 K, 2 поля) при epsilon = 1. и step=0.0005:
  61.907, 256.400
  33.513, 118.679
  70.349, 300.218
  …
  Значение функции ошибки J с начальными коэффициентами: 6.18E8.
  Средняя разность между предсказанием и действительным значением снизилась с 7219.14 на первой итерации до 4.28 на последней.
  Для полученных коэффициентов [4.28, -0.045] J приняла значение 3561802, т.е. уменьшилась.
  По приведённой выборке данных видно, что полученные коэффициенты удовлетворяют набору данных:
  70.35*4.28 - 0.045 = 301.05 ≈ 300.22

Алгоритм отработал за 7679 ms на Celeron N2830 @ 2.16GHz × 2
На датасете размера 3мб и 10 полей при epsilon = 5.0, speed = 5.0E-14 средняя погрешность предсказания снизилась с 1.18E7 до 802 за 81838 ms.
J при этом снизилась с 1.84930974E8 до 43898538,31.

- Генерация датасета на 1M. Точек с пятью переменными (закомментирован по-умолчанию, т.к. для его выполнения требуется несколько минут)

Масштабируемость

Для тестирования масштабируемости программы был организован вычислительный кластер из 4-х машин под управлением Apache Spark и сгенерирован датасет весом 89 Мб с 5-ю переменными и 1-м миллионом точек.

Программа запускалась с погрешностью 25.0 и скоростью обучения 0.00005 на последовательно наращиваемом кластере, изначально состоящем из одной 4-х ядерной Windows-машины, а после — трёх машин под управлением Linux и одной под Windows с суммарным количеством ядер 14.

В результате удалось добиться повышения скорости выполнения спуска с 770 до 267 с., то есть в 2.8 раз!

На представленном ниже графике можно заметить возрастающие участки. Провал в скорости выполнения происходит тогда, когда к кластеру подключается самый маломощный из имеющихся компьютеров: 2-х ядерный Celeron N2830. Тестирование показало, что его вычислительная мощность не окупила возникающие накладные расходы на взаимодействие с ним.

Таким образом, масштабируемость данной реализации параллельного градиентного спуска с использованием фреймворка Spark показана.

См. график на следующей странице.

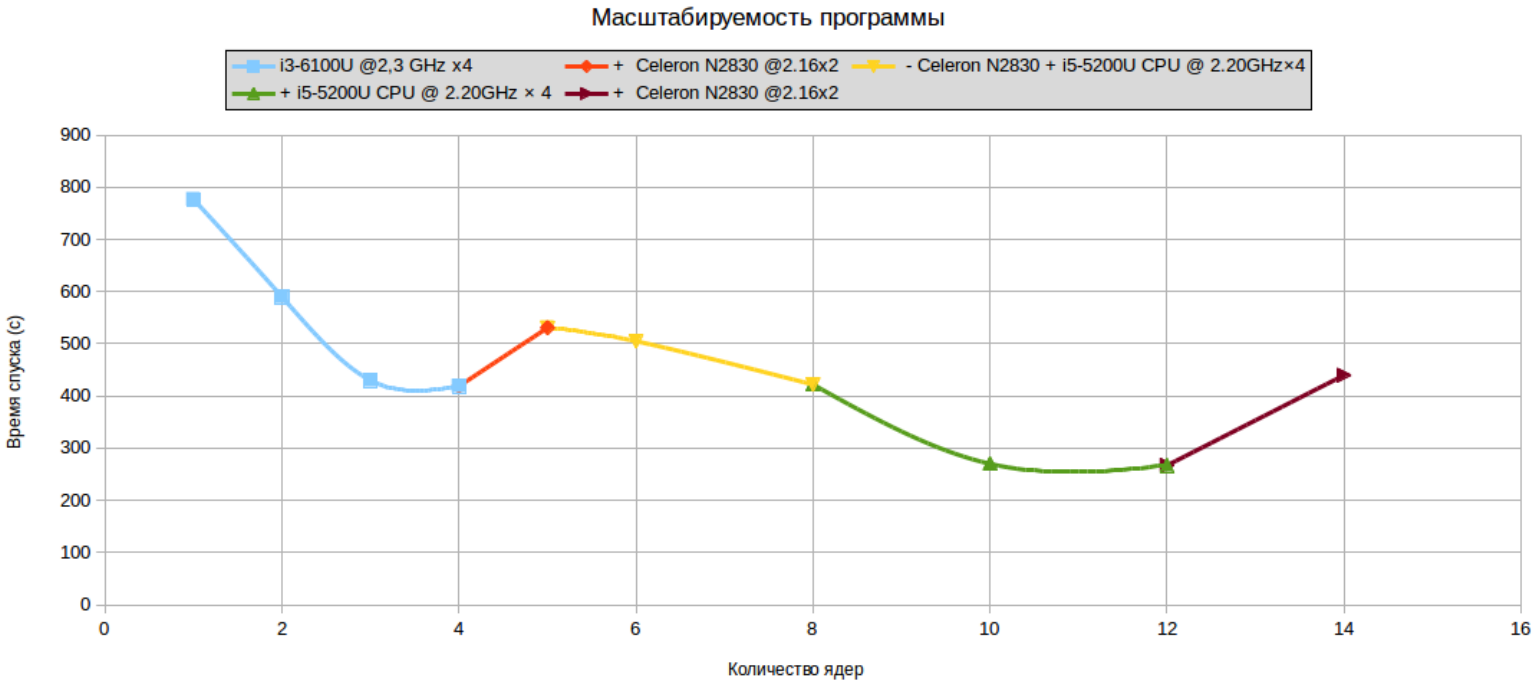


График 1- зависимость времени выполнения алгоритма от количества ядер кластера.

Примечание: указан временной интервал от начала вычисления непосредственно градиентного спуска до его окончания.

Замеры времени выполнения

Windows 10 on Intel core i3-6100U @2,3 GHz x4

776.988 ms (13.0 min) — для 1-го ядра

589.561 ms (9.8 min) — для 2-х ядер

430.141 ms (7.2 min) — для 3-х ядер

419.836 ms (6.9 min) — для 4-х ядер

+ Ubuntu 16.04 on Celeron N2830 @2.16 GHz x2

531.578 ms (8.7 min) — для 5-ти ядер

505.445 ms (8.52 min) — для 6-ти ядер

- Celeron + Ubuntu 16.04 on intel® Core™ i5-5200U CPU @ 2.20GHz x4

422.459 ms (7.0 min) — для 8-и ядер

+ Ubuntu 16.04 on intel® Core™ i5-5200U CPU @ 2.20GHz x4

269.300 ms (4.5 min) — для 10-ти ядер

266.669 ms (4.5 min) — для 12-ти ядер

+ Ubuntu 16.04 on Celeron N2830 @2.16 GHz x2

440.396 ms (7.4 min) — для 14-ти ядер