Санкт-Петербургский Государственный
Политехнический Университет

Распределенная реализация алгоритма поиска похожих объектов

Выполнили:
Безруков Вадим
Панов Лев
Толмачев Александр
Шеина Екатерина
гр. 63601/2

Преподователь:
Клавдиев Д.В.

Санкт-Петербург, 2014

Оглавление

Постановка задачи 2
Описание метода решения 2
Вычисление рейтингов 2
Определение похожести сайтов 3
Предсказание рейтингов 3
Описание технологии Apache Spark 4
Resilient Distributed Datasets 4
Способы хранения данных 4
Интерфейсы взаимодействия 5
Архитектура распределенных вычислений 5
Рекомендации к оборудованию 6
Реализация алгоритма 7
Результаты испытаний 7
Приложение. Исходный код программы. 8

Постановка задачи

Имеется большой объем данных, содержащих информацию о посещении пользователями интернет-сайтов. Данные представляют собой текстовые файлы, каждая строчка которых содержит информацию о посещении определенным пользователем определенного сайта. Пользователи и сайты определяются своими уникальными идентификаторами.

Требуется реализовать алгоритм, который по имеющемуся массиву данных и номеру конкретного сайта определяет тех пользователей, которые не посещали данный сайт, но с высокой вероятностью им заинтересуются. Реализация алгоритма должна быть распределенной, то есть должна позволять обрабатывать данные на кластере из нескольких машин. Также требуется провести испытания полученного решения и исследовать зависимость времени, затрачиваемого на получение результата, в зависимости от количества используемых вычислительных ресурсов.

Описание метода решения

Идея подхода, выбранного нами для решения задачи, состоит в следующем. Каждый пользователь посещает какие-то сайты чаще, какие-то реже. Можно предположить, что чем чаще пользователь посещает сайт, тем более этот сайт ему интересен. Таким образом, можно составить картину интересов пользователя, основываясь на частоте посещений (посчитать для данного пользователя рейтинги посещаемых им сайтов). В свою очередь, каждый сайт можно охарактеризовать тем, какие пользователи и насколько часто его посещают. Разумно предположить, что похожие сайты посещают похожие пользователи (имеющие общие интересы). Соответственно, можно определить, насколько два сайта похожи, по тому, насколько похожи их аудитории. Далее, имея информацию о том, какие сайты в какой мере интересны для пользователя, и о том, насколько сайты похожи между собой, можно предсказать, насколько пользователю будет интересен некоторый сайт, на котором он не был: можно оценить рейтинг для данного сайта, используя известные рейтинги посещаемых им сайтов с учетом того, насколько каждый из посещаемых сайтов похож на данный.

Таким образом, для решения задачи требуется выполнить следующие этапы:

1. Вычислить рейтинги сайтов для каждого пользователя.
2. Определить, насколько сайт, поданный на вход алгоритму, похож со всеми остальными сайтами.
3. Предсказать рейтинг данного сайта для всех пользователей, которые его не посещали.

Затем остается отсортировать пользователей по убыванию предсказанного рейтинга и выдать заданное число пользователей из начала этого списка.

Вычисление рейтингов

Исходные данные содержат информацию о том, какие сайты каждый конкретный пользователь сколько раз посещал. Располагая такой информацией, можно вы-

числить пользовательский рейтинг для сайта следующим образом:

$$r_{u,s} = \frac{num\_visits_{u,s}}{\max\limits_{s \in Visited(u)} num\_visits_{u,s}}, \tag{1}$$

где $num\_visits_{u,s}$ – число посещений сайта s пользователем u, $Visited(u)$ – множество сайтов, посещенных пользователем u. Таким образом, рейтинг представляет собой число из интервала $[0, 1]$ и отражает то, какой популярностью данный сайт пользуется у данного пользователя.

Определение похожести сайтов

Каждый из рассматриваемых сайтов можно охарактеризовать тем, насколько он популярен у тех или иных пользователей, то есть вектором пользовательских рейтингов, вычисленных на предыдущем этапе. Тогда, введя меру похожести для этих векторов, можно определить то, насколько два сайта похожи между собой. В качестве такой меры похожести мы используем коэффициент корреляции Пирсона:

$$corr_{s_1,s_2} = \frac{\sum\limits_{u \in Visited(s_1,s_2)} (r_{u,s_1} - \overline{r}_{s_1})(r_{u,s_2} - \overline{r}_{s_2})}{\sqrt{\sum\limits_{u \in Visited(s_1,s_2)} (r_{u,s_1} - \overline{r}_{s_1})^2} \sqrt{\sum\limits_{u \in Visited(s_1,s_2)} (r_{u,s_2} - \overline{r}_{s_2})^2}}, \tag{2}$$

где $Visited(s_1, s_2)$ – множество пользователей, посетивших оба сайта s_1 и s_2, $r_{u,s}$ – рейтинг сайта s для пользователя u, $\overline{r}_s$ – средний рейтинг сайта s по всем пользователям.

Предсказание рейтингов

Пусть t – сайт, поданный на вход алгоритму. Тогда для каждого из пользователей, которые не посещали этот сайт, можно оценить рейтинг следующим образом:

$$\widetilde{r}_{u,t} = \overline{r}_u + \frac{\sum\limits_{s \in Visited(u)} (r_{u,s} - \overline{r}_u)\, corr_{s,t}}{\sum\limits_{s \in Visited(u)} |corr_{s,t}|}, \tag{3}$$

где $\overline{r}_u$ – средний рейтинг посещаемых сайтов для пользователя u, $Visited(u)$ – множество сайтов, посещаемых пользователем u. То есть предсказываемый рейтинг вычисляется как сумма среднего рейтинга по посещаемым сайтам и средневзвешенного отклонения от среднего, где веса – коэффициенты корреляции для посещаемого сайта и сайта, для которого предсказывается рейтинг.

Описание технологии Apache Spark

Для реализации алгоритма мы выбрали технологию распределенных вычислений Apache Spark. Это относительно новый фреймворк для анализа больших массивов данных, уже получивший довольно большую популярность. Разработчики Spark заявляют о его высокой производительности по сравнению с Apache Hadoop, вплоть до 100-кратного превосходства для некоторых задач.

Apache Spark является проектом с открытым исходным кодом. Работа над этим фреймворком начиналась в 2009 году в рамках научно-исследовательского проекта в Калифорнийском университете Беркли. В 2013 году проект перешел в Apache Software Foundation. А в 2014 году он стал одним наиболее приоритетных проектов Apache.

Resilient Distributed Datasets

В основе фреймворка лежит концепция RDD (Resilient Distributed Dataset) – абстракция для высокоэффективной и отказоустойчивой работы с распределенными массивами данных. RDD – это неизменяемая коллекция объектов, распределенная по машинам кластера. Они может постоянно находятся в оперативной памяти, что позволяет обеспечивать высокую скорость доступа к данным при вычислениях. Например, при RDD размером до 39 ГБ гарантируется скорость доступа менее 1 с.

С точки зрения интерфейса RDD представляет собой единую коллекцию данных. Spark сам заботится о ее разбиении на части и размещении в оперативной памяти и на жестких дисках машин, а также о связи частей между собой. В то время как пользователь работает с RDD как с локальной коллекцией данных.

Вычисления в Apache Spark организуются как последовательности преобразований над RDD. История преобразований коллекции данных сохраняется, поэтому в случае сбоев утраченная часть коллекции может быть заново восстановлена. Это обеспечивает устойчивость к сбоям (fault-tolerance). Также вычисления происходят “ленивым” образом: непосредственно преобразования данных происходят только тогда, когда требуются результаты вычислений. Это позволяет организовывать вычисления более оптимально – группировать определенные преобразования и выполнять их партиями, более эффективно распараллеливать независимые преобразования, а также вообще не выполнять тех вычислений которые не требуются для получения конечного результата.

Способы хранения данных

Если объем данных таков, что они не помещаются в оперативную память машин кластера, Spark будет выгружать их на жесткий диск. При это есть возможность указать, какие данные где могут располагаться – в оперативной памяти (для данных, к которым требуется частое обращение), в оперативной памяти и на жестком диске, только на жестком диске, какие данные сохранять в сериализованном виде, какие нет и т.д. Spark предоставляет пользователям гибкую настройку в вопросе использования памяти, которая позволяет оптимально использовать имеющиеся ресурсы.

Возможность хранения данных в оперативной памяти предоставляет большие преимущества для ряда вычислительных задач и является одной из сильных сторон фреймворка Spark. Для сравнения, Apache Hadoop не имеет такой возможности

и записывает промежуточные результаты map-reduce операций на жесткие диски машин, что очень затратно и существенно снижает производительность. Поэтому для задач, которые требуют значительного использования промежуточных данных, Spark работает до ста раз быстрее, чем Apache Hadoop.

Загружать исходные данные Spark позволяет как с локальной файловой системы машин, так и из распределенных файловых систем и хранилищ: HDFS, Amazon S3, Cassandra, HBase и т.д.

Интерфейсы взаимодействия

Spark предоставляет API для написания программ на Java, Python и Scala. Наиболее поддерживаемым языком является Scala, т.к. Spark сам написан на Scala и все основные тесты написаны на Scala. Благодаря функциональной выразительности предоставляемого API программы для Apache Spark получаются короче и проще, чем для Apache Hadoop.

Работа с RDD происходит при помощи вызова методов, предоставляемых в API. Методы делятся на две группы: Transformations и Actions. Методы, относящиеся к группе Transformations, производят преобразование коллекции данных и возвращают новую RDD (например, методы `map` и `reduceByKey`). Методы, относящиеся к Actions, выполняют операции с коллекцией данных и возвращают конечный результат (например, метод `count`, вычисляющий количество элементов в коллекции, или метод `count` который позволяет сохранить коллекцию данных на локальный диск). Все методы из Transformations ленивые, то есть их выполнение откладывается до вызова первого метода из Actions.

Архитектура распределенных вычислений

Приложения для Spark выполняются на кластере как самостоятельные наборы процессов, управляемые объектом SparkContext в основной программе (driver program). SparkContext подключается к менеджеру кластера (cluster manager), который отвечает за управление вычислениями. После подключения Spark запускает исполнителей (executor) на узлах кластера – процессы, которые выполняют вычисления и отвечающими за хранение данных на конкретном узле. После этого Spark загружает программный код (упакованный в JAR архив или Python файлы) и рассылает задачи (task) по исполнителям.

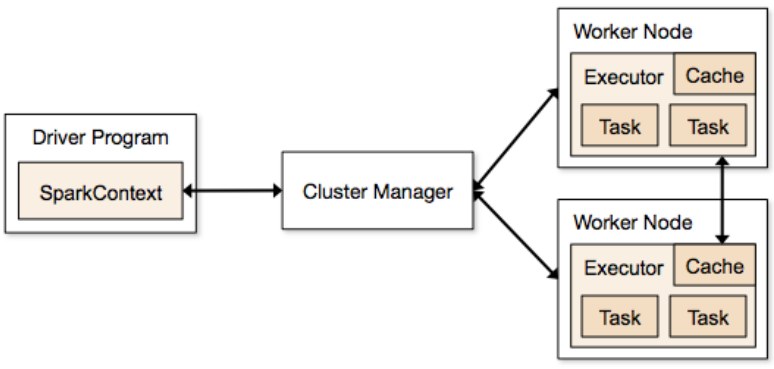

Рис. 1: Схема архитектуры вычислений на кластере

Некоторые особенности такой архитектуры:

1. Каждое приложение получает свои процессы исполнителей, которые работаю пока запущено приложение и выполняют задачи в разных потоках. Таким образом приложения изолированы друг от друга, как со стороны планирования (каждый драйвер распределяет только свои задачи), так и со стороны исполнения (задачи из разных приложений выполняются на разных Java-машинах). Однако, это означает, что данные не могут разделяться между различными Spark приложениями без использования внешнего хранилища.
2. Архитектура вычислений не зависит от используемого менеджера кластера.
3. Поскольку драйвер распределяет задачи на кластере, он должен работать близко к рабочим узлам, предпочтительно в одной и той же локальной сети.

На данный момент Spark поддерживает три кластерных менеджера:

- Standalone – простой менеджер кластера, поставляемый вместе со Spark. Позволяет быстро настроить кластер.
- Apache Mesos – стандартный менеджер кластера, также позволяющий использовать Hadoop MapReduce и сервисные приложения.
- Hadoop YARN – менеджер ресурсов в Hadoop 2.

Рекомендации к оборудованию

Системы хранения данных

Поскольку большинство встречаемых задач основываются на чтении входных данных с внешних системах хранения информации(в том числе Hadoop File System, или HBase) особенно важно разместить данные как можно ближе к системе. Рекомендуется следующее:

- По возможности, запускать Spark на идентичном железе с файловой системой HDFS. Проще всего настроить Spark в режиме standalone кластера на схожих узлах. Также можно запускать Spark и Hadoop в общей кластерной группе с помощью систем для планирования заданий и управления кластерами типа MESOS, Hadoop YARN.
- Если нет возможности использования схожих узлов, то рекомендуется разместить узлы в локальной сети с HDFS(распределенной файловой системой).
- На узлах, занимающихся вычислениями лучше не хранить “ленивые” данные типа HBase. (Правильнее разделять узлы для ленивых данных и для вычислительных узлов)

Локальные диски

Во время работы Spark используемые данные для вычислений хранятся в оперативной памяти, в то же время используются и локальные хранилища для данных не поместившихся в RAM. Рекомендовано использовать 4-8 дисковых хранилищ на узел настроенных без RAID.

Память

Spark может использовать 8Gb и до сотен гигабайт оперативной памяти на одной машине. Рекомендуется использовать не более 75% от общего количества оперативной памяти на машине. Сколько именно оперативной памяти может потребоваться, напрямую зависит от поставленной задачи. Эффективность использования памяти зависит и от эффективности жестких дисков и их файловой системы.

Также виртуальная машина Java не всегда ведет себя корректно с более чем 200 Gb оперативной памяти. Если имеются машины с бóльшим колчеством оперативной памяти, можно использовать несколько виртуальных машин на одном узле.

Сеть

В процессе работы Spark, множество распределенных данных передаются по сети. Использование 10-гигабитной или более быстрой сети – лучший способ ускорить эту передачу. Это особенно важно для "распределенных вычислений" таких как group-bys, reduce-bys и SQL joins. Вы можете следить за передаваемыми данными Spark с помощью сетевого мониторинга.

Многоядерные процессоры

Spark поддерживает многоядерные процессоры, т.к. создает минимальное разделение задач между потоками. Для сложных вычислений можно использовать 8-16 ядерные процессоры для ускорения обработки данных.

Реализация алгоритма

Мы реализовали алгоритм на языке Python, используя Python API фреймворка Apache Spark. Код программы приведен в приложении.

Результаты испытаний

TBD

Приложение. Исходный код программы.

```
from pyspark import SparkContext, SparkConf
from argparse import ArgumentParser
from math import sqrt

APPLICATION_NAME = "Look-Alike Task"


# Parses user id and site id from log line. Each log line indicates that
# user with user_id visited site with site_id.
def parse_log_line(log_line):
    parts = log_line.split("\t")
    if len(parts) != 2:
        # invalid log line
        return []
    user_id = parts[0]
    site_id = int(parts[1])
    return [(user_id, site_id)]


# Performs distributed calculation of site ratings for each user.
# Rating calculation for particular user is based on visits count:
# rating(site) = visits_count(site) / max_visits_count
# Thus, rating is a number from range [0.0, 1.0]
def calculate_ratings(data):
    # Creates initial combiner for aggregating user visits count per site and max visits count.
    def create_combiner(site_id):
        site_id_to_visits_count = dict()
        site_id_to_visits_count[site_id] = 1
        return [site_id_to_visits_count, 1]

    # Updates combiner containing partially aggregated data by visited site id.
    def sum_combiner_with_value(combiner, site_id):
        site_id_to_visits_count = combiner[0]
        max_visits_count = combiner[1]
        if site_id not in site_id_to_visits_count:
            site_id_to_visits_count[site_id] = 0
        site_id_to_visits_count[site_id] += 1
        # max visits count can only change due to updated dict item
        combiner[1] = max(max_visits_count, site_id_to_visits_count[site_id])
        return combiner

    # Sums two combiners containing partially aggregated data.
    def sum_combiners(combiner1, combiner2):
        site_id_to_visits_count1 = combiner1[0]
        site_id_to_visits_count2 = combiner2[0]
        max_visits_count = combiner1[1]
        for site_id, visits_count in site_id_to_visits_count2.iteritems():
            if site_id not in site_id_to_visits_count1:
                site_id_to_visits_count1[site_id] = 0
            site_id_to_visits_count1[site_id] += visits_count
            # max visits count can only change due to updated dict items
            max_visits_count = max(max_visits_count, site_id_to_visits_count1[site_id])
        combiner1[1] = max_visits_count
        return combiner1

    # Calculates site ratings for particular user using aggregated user visits count per site
    # and max visits count.
    def calculate_site_ratings_for_user(key_value_pair):
        user_id = key_value_pair[0]
```

```
        combiner_result = key_value_pair[1]
        site_ratings_map = combiner_result[0]
        max_visits_count = float(combiner_result[1])
        for site_id in site_ratings_map:
            site_ratings_map[site_id] /= max_visits_count
        return user_id, site_ratings_map

    return data.combineByKey(create_combiner, sum_combiner_with_value, sum_combiners).\
        map(calculate_site_ratings_for_user)


# Performs distributed calculation of correlations between target site and all other sites.
def calculate_correlations(ratings_data, target_site_id):
    # Takes site ratings for particular user and produces pairs, where one element is rating
    # of target site and other one is rating of other site visited by this user.
    # If user has not visited target site, produces nothing.
    # The output is like the following:
    # site1_id: site1_rating, target_site_rating
    # site2_id: site2_rating, target_site_rating
    # ...
    # siteN_id: siteN_rating, target_site_rating
    def create_rating_pairs(user_ratings_pair):
        ratings = user_ratings_pair[1]
        if target_site_id not in ratings:
            return []
        target_site_rating = ratings[target_site_id]
        return [(site_id, (site_rating, target_site_rating)) for (site_id, site_rating)
                in ratings.iteritems() if site_id != target_site_id]

    # Creates initial combiner for aggregating data that needed to calculate Pearson correlation
    # coefficient. We need to collect sum of all user ratings for both sites, sum of squared
    # ratings for both sites, sum of rating products and count of occurred rating pairs.
    # So each item of created list is used to collect this data respectively.
    def create_combiner(site_ratings_pair):
        rating1 = site_ratings_pair[0]
        rating2 = site_ratings_pair[1]
        return [rating1, rating2, rating1 * rating1, rating2 * rating2, rating1 * rating2, 1]

    # Merges pair of ratings (value) with list, containing partially aggregated data (combiner)
    def sum_combiner_with_value(combiner, site_ratings_pair):
        rating1 = site_ratings_pair[0]
        rating2 = site_ratings_pair[1]
        combiner[0] += rating1
        combiner[1] += rating2
        combiner[2] += rating1 * rating1
        combiner[3] += rating2 * rating2
        combiner[4] += rating1 * rating2
        combiner[5] += 1
        return combiner

    # Sums two lists with aggregated data.
    def sum_combiners(combiner1, combiner2):
        for i in xrange(len(combiner1)):
            combiner1[i] += combiner2[i]
        return combiner1

    # Calculates Pearson correlation coefficient using aggregated data.
    def calculate_correlation(key_value_pair):
        site_id = key_value_pair[0]
        combiner_result = key_value_pair[1]
        # obtain combined data
```

```
        ratings_sum1 = combiner_result[0]
        ratings_sum2 = combiner_result[1]
        squared_ratings_sum1 = combiner_result[2]
        squared_ratings_sum2 = combiner_result[3]
        ratings_product_sum = combiner_result[4]
        items_number = combiner_result[5]
        # calculate correlation coefficient itself
        mean1 = ratings_sum1 / items_number
        mean2 = ratings_sum2 / items_number
        std_dev1 = sqrt(squared_ratings_sum1 / items_number - mean1 * mean1)
        std_dev2 = sqrt(squared_ratings_sum2 / items_number - mean2 * mean2)
        if std_dev1 == 0 or std_dev2 == 0:
            # correlation will be 0, no need to return value
            return []
        correlation = (ratings_product_sum / items_number - mean1 * mean2) / (std_dev1 * std_dev2)
        return [(site_id, correlation)]

    return ratings_data.flatMap(create_rating_pairs).\
        combineByKey(create_combiner, sum_combiner_with_value, sum_combiners).\
        flatMap(calculate_correlation)


# Calculates predicted rating of target site for users, which hasn't visited it, basing on user
# ratings of sites and correlations between target site and other sites
def predict_user_ratings(site_ratings_data, correlation_map, target_site_id):
    # Calculates predicted rating of target site for particular user.
    # Produces pair (user_id, predicted_rating) if user hasn't visited target site and
    # nothing otherwise. Predicted rating is calculated as weighted deviation from average user
    # ratings of sites, where weights are correlation coefficients.
    def predict_rating_for_user(user_ratings_pair):
        user_id = user_ratings_pair[0]
        ratings = user_ratings_pair[1]
        # predict ratings only for users who hasn't visited target site
        if target_site_id in ratings:
            return []
        avg_rating = sum(ratings.itervalues(), 0.0) / len(ratings)
        predicted_rating = 0.0
        normalizing_sum = 0.0
        for site_id, site_rating in ratings.iteritems():
            if site_id in correlation_map:
                weight = correlation_map[site_id]
                predicted_rating += (site_rating - avg_rating) * weight
                normalizing_sum += abs(weight)
        if normalizing_sum != 0:
            predicted_rating /= normalizing_sum
        predicted_rating += avg_rating
        return [(user_id, predicted_rating)]

    return site_ratings_data.flatMap(predict_rating_for_user)


def main():
    args_parser = ArgumentParser()
    args_parser.add_argument("--input", help="input files location", required=True)
    args_parser.add_argument("--target-site", help="target site id", required=True)
    args_parser.add_argument("--users-count", help="number of users in result list", required=True)

    args = args_parser.parse_args()
    input_files_location = args.input
    target_site_id = int(args.target_site)
    users_count = int(args.users_count)
```

```
    configuration = SparkConf().setAppName(APPLICATION_NAME)
    spark_context = SparkContext(conf=configuration)
    input_data = spark_context.textFile(input_files_location).flatMap(parse_log_line)
    site_ratings_data = calculate_ratings(input_data).cache()  # cache as this data is used twice
    correlation_map = calculate_correlations(site_ratings_data, target_site_id).collectAsMap()
    predicted_ratings = predict_user_ratings(site_ratings_data, correlation_map, target_site_id)
    result = predicted_ratings.takeOrdered(users_count, lambda x: -x[1])
    spark_context.stop()

    for item in result:
        print item[0], ":", item[1]

if __name__ == "__main__":
    main()
```