ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

BÁO CÁO BÀI TẬP LỚN

Môn học: Dữ liệu lớn

Đề tài:

Tìm những đối tượng giống nhau trong tập dữ liệu lớn

GVHD: Thoại Nam

Nhóm lớp: L01
SVTH: Nguyễn Hoàng Phúc
1927030
Nguyễn Đức Tuấn
1927040

TP. HỒ CHÍ MINH, THÁNG 6/2021

Mục lục

1 Tìm hiểu về Apache Spark **2**
1.1 Giới thiệu về Spark Framework . . . 2
1.2 Thiết lập Apache Spark Cluster trong Docker . . . 4
1.2.1 Tổng quan Spark: . . . 4
1.2.2 Apache Spark Cluster với Docker images hierarchy . . . 4
1.2.3 Tạo các images trong Docker . . . 5
1.2.4 Xây dựng images . . . 9
1.2.5 Composing the cluster . . . 10
1.2.6 Tạo ứng dụng PySpark . . . 12

2 Ứng dụng giải thuật Locality Sensitive Hashing để tìm các đối tượng giống nhau trong tập dữ liệu lớn **13**
2.1 Giới thiệu các khái niệm cơ bản về giải thuật Locality Sensitive Hashing . . . 13
2.1.1 Khái niệm Locality Sensitive Hashing: . . . 13
2.1.2 Bit Sampling LSH . . . 14
2.1.3 Signed Random Projections . . . 14
2.1.4 Euclidean and Manhattan LSH . . . 15
2.1.5 Clustering LSH . . . 15
2.2 Ứng dụng giải thuật Locality Sensitive Hashing tìm các văn bản giống nhau . . . 16
2.2.1 Mô tả tập văn bản đầu vào . . . 16
2.2.2 Chuyển từng văn bản thành tập các unique shingles . . . 16
2.2.3 Tạo Signature matrix . . . 17
2.2.4 Locality sensitive hashing . . . 18
2.2.5 Sử dụng LSH trong thư viện MLlib - pyspark với Bucketed Random Projection for Euclidean Distance . . . 19
2.2.6 Sử dụng LSH trong thư viện MLlib - pyspark với MinHash for Jaccard Distance . . . 20
2.3 Ứng dụng giải thuật Locality Sensitive Hashing tìm các ảnh giống nhau . . . 21
2.3.1 Mô tả tập ảnh đầu vào . . . 21
2.3.2 Tạo Signature matrix . . . 21
2.3.3 Sử dụng LSH trong thư viện MLlib - pyspark với MinHash for Euclidean Distance . . . 23
2.3.4 Brute force calculation với Cosin Similarity Distance . . . 24
2.3.5 Locality Sensitive Hashing dựa trên Cosin Similarity Distance 25

3 Tham khảo **27**

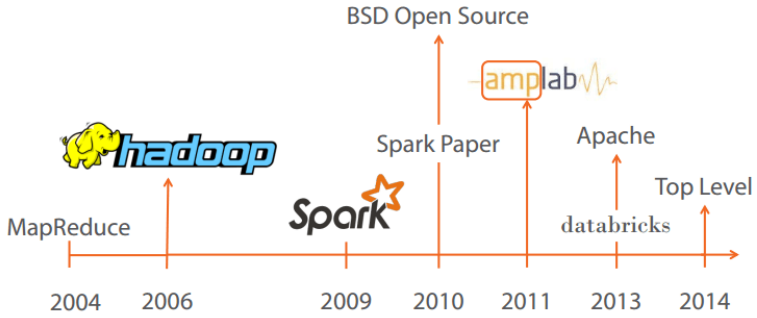
1 Tìm hiểu về Apache Spark

1.1 Giới thiệu về Spark Framework

- Ngày nay có rất nhiều hệ thống đang sử dụng Hadoop để phân tích và xử lý dữ liệu lớn. Ưu điểm lớn nhất của Hadoop là được dựa trên một mô hình lập trình song song với xử lý dữ liệu lớn là MapReduce, mô hình này cho phép khả năng tính toán có thể mở rộng, linh hoạt, khả năng chịu lỗi, chi phí rẻ. Điều này cho phép tăng tốc thời gian xử lý các dữ liệu lớn nhằm duy trì tốc độ, giảm thời gian chờ đợi khi dữ liệu ngày càng lớn.

- Dù có rất nhiều điểm mạnh về khả năng tính toán song song và khả năng chịu lỗi cao nhưng Apache Haddop có một nhược điểm là tất cả các thao tác đều phải thực hiện trên ổ đĩa cứng điều này đã làm giảm tốc độ tính toán đi gấp nhiều lần.

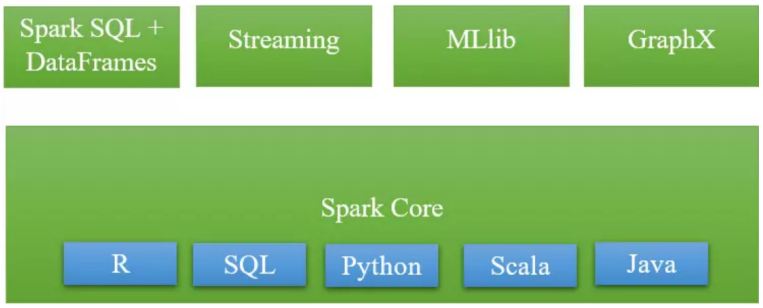
- Để khắc phục được nhược điểm này thì Apache Spark được ra đời. Apache Spark có thể chạy nhanh hơn 10 lần so với Haddop ở trên đĩa cứng và 100 lần khi chạy trên bộ nhớ RAM.



Hình 1: Thời gian ra đời Spark

- Apache Spark là một framework mã nguồn mở tính toán cụm, được phát triển sơ khởi vào năm 2009 bởi AMPLab. Sau này, Spark đã được trao cho Apache Software Foundation vào năm 2013 và được phát triển cho đến nay.

- Tốc độ xử lý của Spark có được do việc tính toán được thực hiện cùng lúc trên nhiều máy khác nhau. Đồng thời việc tính toán được thực hiện ở bộ nhớ trong (in-memories) hay thực hiện hoàn toàn trên RAM.



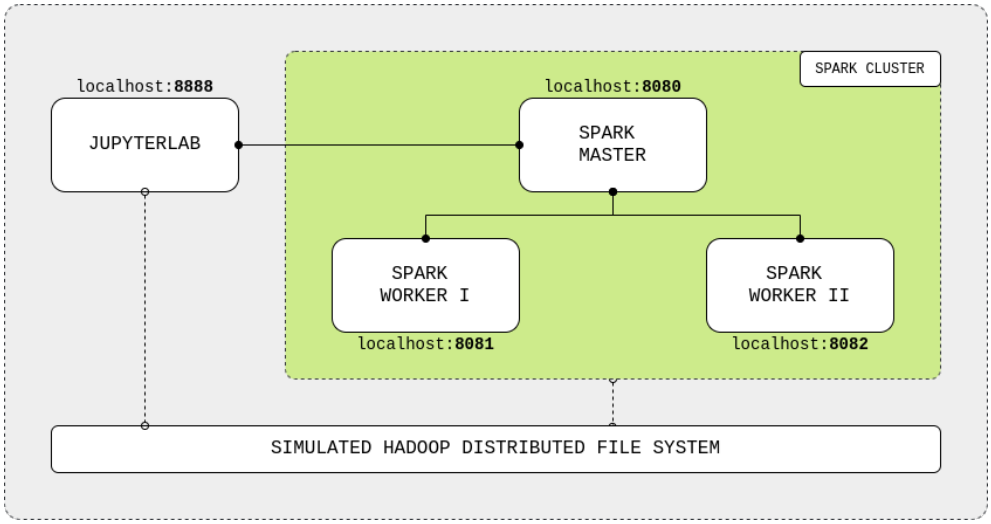
Hình 2: Thành phần của Spark

- Apache Spark gồm có 5 thành phần chính : Spark Core, Spark Streaming, Spark SQL, MLlib và GraphX. Trong Bài tập lớn này, có sử dụng về MLlip.
- **MLlib** (Machine Learning Library): MLlib là một nền tảng học máy phân tán bên trên Spark do kiến trúc phân tán dựa trên bộ nhớ. Theo các so sánh benchmark Spark MLlib nhanh hơn 9 lần so với phiên bản chạy trên Hadoop (Apache Mahout).

1.2 Thiết lập Apache Spark Cluster trong Docker

1.2.1 Tổng quan Spark:

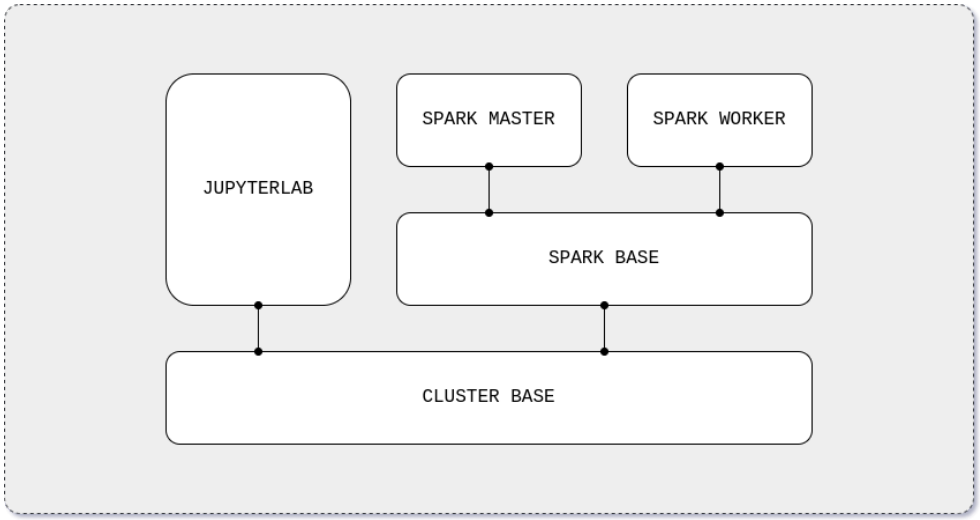
Cluster là tổng hợp của bốn thành phần chính: JupyterLab IDE, Spark master node và hai Spark workers nodes. Người dùng kết nối đến master node và gửi lệnh Spark qua GUI(Giao diện đồ họa) được cung cấp bởi Jupyter notebooks. Master node xử lý input và phân bổ tính toán workload đến worker nodes, sau đó gửi ngược kết quả lại đến IDE. Các components được kết nối bằng cách sử dụng localhost network và share dữ liệu với nhau thông qua một ổ đĩa được chia sẻ được gắn kết mô phỏng HDFS.



Hình 3: Apache Spark Cluster overview

1.2.2 Apache Spark Cluster với Docker images hierarchy

Như trên, chúng ta cần create, build và compose Docker images cho Jupyter và các Spark nodes để chạy cluster. Chúng ta sẽ sử dụng Docker image hierarchy sau đây.



Hình 4: Docker images hierarchy

- Cluster base image sẽ tải về và cài đặt các tool phần mềm phổ biến (Java, Python,...) và sẽ tạo thư mục được chia sẻ cho HDFS.
- Spark base image: Ứng dụng Apache Spark sẽ được tải về và cấu hình cả trên master node và worker nodes.
- Spark master image: sẽ được cấu hình framework để chạy master node.
- Spark worker image: sẽ được cấu hình framework để chạy worker node.
- JupyterLap image: sẽ sử dụng cluster base image để cài đặt và cấu hình IDE và PySpark, Apache Spark's Python API

1.2.3 Tạo các images trong Docker

1. Cluster base image:

   Đối với cluster base image, chúng ta sử dụng bản phân phối của Linux để cài đặt Java 8 (hoặc 11), theo yêu cầu của Spark [1]. Chúng ta cũng cần cài đặt Python 3 để hỗ trợ PySpark và tạo khối lượng chia sẻ để mô phỏng HDFS.

```
ARG debian_buster_image_tag=8-jre-slim
FROM openjdk:${debian_buster_image_tag}

# -- Layer: OS + Python 3.7

ARG shared_workspace=/opt/workspace

```

```
RUN mkdir -p ${shared_workspace} && \
    apt-get update -y && \
    apt-get install -y python3 && \
    ln -s /usr/bin/python3 /usr/bin/python && \
    rm -rf /var/lib/apt/lists/*

ENV SHARED_WORKSPACE=${shared_workspace}

# -- Runtime

VOLUME ${shared_workspace}
CMD ["bash"]
```

Listing 1: Dockerfile cho cluster base image

2. Spark base image:

   Đối với Spark base image, chúng ta sẽ lấy và cài đặt Apache Spark trong Standalone mode, cấu hình triển khai đơn giãn nhất của nó.

   Trong mode này, chúng ta sẽ sử dụng resource managers để thiết lập các containers để chạy master node hay các worker nodes. Ngược lại, các resource managers chẳng hạn như Apache Yarn phân bổ động các containers dưới dạng master node hoặc worker nodes theo khối lượng công việc của người dùng

   Hơn nữa, chúng ta sẽ nhận được phiên bản Apache Spark với hỗ trợ của Apache Hadoop để cho phép cụm mô phỏng HDFS bằng cách sử dụng shared volume được tạo trong base cluster image.

```
FROM cluster-base

# -- Layer: Apache Spark

ARG spark_version=3.0.0
ARG hadoop_version=2.7

RUN apt-get update -y && \
    apt-get install -y curl && \
    curl https://archive.apache.org/dist/spark/spark-${spark_version}/spark-${spark_version}-bin-hadoop${hadoop_version}.tgz -o spark.tgz && \
    tar -xf spark.tgz && \
    mv spark-${spark_version}-bin-hadoop${hadoop_version} /usr/bin/ && \
    mkdir /usr/bin/spark-${spark_version}-bin-hadoop${hadoop_version}/logs && \
    rm spark.tgz

```

```
16 ENV SPARK_HOME /usr/bin/spark-${spark_version}-bin-hadoop${
       hadoop_version}
17 ENV SPARK_MASTER_HOST spark-master
18 ENV SPARK_MASTER_PORT 7077
19 ENV PYSPARK_PYTHON python3
20
21 # -- Runtime
22
23 WORKDIR ${SPARK_HOME}
```

Listing 2: Dockerfile cho the Spark base image

Đầu tiên, tải về Apache Spark phiên bản 3.0.0 với hỗ trợ Apache Hadoop từ Apache repository chính thức [2], sau đó tải về các package. Cuối cùng, chúng ta định cấu hình bốn biến Spark chung cho cả master node và các worker nodes.

- **SPARK_HOME**: là vị trí Apache Spark đã cài đặt được sử dụng bởi framework để thiết lập các tác vụ;
- **SPARK_MASTER_HOST**: là hostname của master node được worker nodes sử dụng để kết nối.
- **SPARK_MASTER_PORT**: là cổng master node được worker nodes sử dụng để kết nối.
- **PYSPARK_PYTHON**: là vị trí Python đã cài đặt được Apache Spark sử dụng để hỗ trợ API Python của nó.

3. Spark master image

   Đối với Spark master image, chúng ta sẽ thiết lập ứng dụng Apache Spark để chạy như master node. Chúng ta sẽ cấu hình các cổng mạng để cho phép kết nối mạng với các worker nodes và hiển thị giao diện web master, một trang web để giám sát các hoạt động của master node. Cuối cùng, chúng tôi sẽ thiết lập lệnh khởi động container để khởi động nút như một thể hiện của master.

```
1 FROM spark-base
2
3 # -- Runtime
4
5 ARG spark_master_web_ui=8080
6
7 EXPOSE ${spark_master_web_ui} ${SPARK_MASTER_PORT}
8 CMD bin/spark-class org.apache.spark.deploy.master.Master >>
      logs/spark-master.out
```

Listing 3: Dockerfile cho Spark master image

Đặt biến môi trường **SPARK_MASTER_PORT** để cho phép các worker kết nối với master node. Sau đó, expose **spark_master_web_ui** để cho phép chúng ta truy cập trang giao diện người dùng web master. Cuối cùng, chúng ta đặt lệnh khởi động container để chạy lệnh triển khai, tích hợp sẵn Spark với master class [3] làm đối số của nó.

4. Spark worker image

   Đối với Spark worker image, chúng ta sẽ thiết lập ứng dụng Apache Spark để chạy dưới dạng một worker node. Tương tự như master node, chúng ta sẽ cấu hình cổng mạng để hiển thị giao diện người dùng web worker, một trang web để theo dõi các hoạt động của worker node và thiết lập lệnh khởi động container để khởi động nút dưới dạng thể hiện của worker.

```
FROM spark-base

# -- Runtime

ARG spark_worker_web_ui=8081

EXPOSE ${spark_worker_web_ui}
CMD bin/spark-class org.apache.spark.deploy.worker.Worker
    spark://${SPARK_MASTER_HOST}:${SPARK_MASTER_PORT} >> logs/
    spark-worker.out
```

Listing 4: Dockerfile cho Spark worker image

Chúng ta expose **spark_worker_web_ui** để cho phép chúng ta truy cập trang giao diện người dùng web worker. Cuối cùng, chúng ta đặt lệnh khởi động container để chạy lệnh triển khai, tích hợp sẵn Spark với workerr class [4] làm đối số của nó.

5. JupyterLab image

   Đối với JupyterLab image, chúng ta quay lại và bắt đầu lại từ cluster base image. Chúng ta sẽ cài đặt và cấu hình IDE cùng với bản phân phối Apache Spark hơi khác so với bản được cài đặt trên các nút Spark.

```
FROM cluster-base

# -- Layer: JupyterLab

ARG spark_version=3.0.0
ARG jupyterlab_version=2.1.5
```

```
7
8  RUN apt-get update -y && \
9      apt-get install -y python3-pip && \
10     pip3 install wget pyspark==${spark_version} jupyterlab==$
       {jupyterlab_version}
11
12 # -- Runtime
13
14 EXPOSE 8888
15 WORKDIR ${SHARED_WORKSPACE}
16 CMD jupyter lab --ip=0.0.0.0 --port=8888 --no-browser --allow
       -root --NotebookApp.token=
```

Listing 5: Dockerfile cho JupyterLab image

Đầu tiên, cài đặt **pip**, trình quản lý gói của Python và các công cụ phát triển Python để cho phép cài đặt các gói Python trong quá trình xây dựng images và chạy thực container. Sau đó, hãy lấy JupyterLab và PySpark từ Python Package Index(PyPI). Cuối cùng, expose cổng mặc định để cho phép truy cập vào giao diện web JupyterLab và đặt lệnh khởi động container để chạy ứng dụng IDE.

1.2.4 Xây dựng images

Sau khi có Dockerfile, chúng ta thực hiện build để tạo ra các images. Các từ khóa **build-arg** trên Dockerfiles để chỉ định phiên bản phần mềm, chúng ta có thể dễ dàng thay đổi phiên bản Apache Spark và JupyterLab mặc định cho cụm.

```
1  # -- Software Stack Version
2
3  SPARK_VERSION="3.0.0"
4  HADOOP_VERSION="2.7"
5  JUPYTERLAB_VERSION="2.1.5"
6
7  # -- Building the Images
8
9  docker build \
10   -f cluster-base.Dockerfile \
11   -t cluster-base .
12
13 docker build \
14   --build-arg spark_version="${SPARK_VERSION}" \
15   --build-arg hadoop_version="${HADOOP_VERSION}" \
16   -f spark-base.Dockerfile \
17   -t spark-base .
18
19 docker build \
```

```
  -f spark-master.Dockerfile \
  -t spark-master .

docker build \
  -f spark-worker.Dockerfile \
  -t spark-worker .

docker build \
  --build-arg spark_version="${SPARK_VERSION}" \
  --build-arg jupyterlab_version="${JUPYTERLAB_VERSION}" \
  -f jupyterlab.Dockerfile \
  -t jupyterlab .
```

Listing 6: Dockerfile cho Building the cluster images

1.2.5 Composing the cluster

File docker-compose.yml chứa công thức cho cụm của chúng ta. Ở đây, chúng ta sẽ tạo các container nút JuyterLab và Spark, hiển thị các cổng của chúng cho mạng localhost và kết nối chúng với HDFS mô phỏng.

```
version: "3.6"
volumes:
  shared-workspace:
    name: "hadoop-distributed-file-system"
    driver: local
services:
  jupyterlab:
    image: jupyterlab
    container_name: jupyterlab
    ports:
      - 8888:8888
    volumes:
      - shared-workspace:/opt/workspace
  spark-master:
    image: spark-master
    container_name: spark-master
    ports:
      - 8080:8080
      - 7077:7077
    volumes:
      - shared-workspace:/opt/workspace
  spark-worker-1:
    image: spark-worker
    container_name: spark-worker-1
    environment:
      - SPARK_WORKER_CORES=1
      - SPARK_WORKER_MEMORY=512m
```

```
    ports:
      - 8081:8081
    volumes:
      - shared-workspace:/opt/workspace
    depends_on:
      - spark-master
  spark-worker-2:
    image: spark-worker
    container_name: spark-worker-2
    environment:
      - SPARK_WORKER_CORES=1
      - SPARK_WORKER_MEMORY=512m
    ports:
      - 8082:8081
    volumes:
      - shared-workspace:/opt/workspace
    depends_on:
      - spark-master
```

Listing 7: Cluster's Docker compose file – docker-compose.yml

Chúng ta bắt đầu bằng cách tạo dung lượng Docker cho HDFS mô phỏng. Tiếp theo, chúng ta tạo một container cho mỗi thành phần cụm. Container jupyterlab expose cổng IDE và liên kết shared workspace directory của nó với ổ đĩa HDFS. Tương tự như vậy, spark-master container hiển thị cổng giao diện người dùng web và cổng kết nối master-worker của nó và cũng liên kết với ổ đĩa HDFS.

Cuối cùng tạo hai container worker Spark có tên là spark-worker-1 và spark-worker-2 . Mỗi container hiển thị cổng giao diện người dùng web của nó (được map lần lượt ở cổng 8081 và 8082) và liên kết với ổ đĩa HDFS. Các vùng chứa này có một bước môi trường chỉ định phân bổ phần cứng của chúng:

- SPARK_WORKER_CORE: là số core;
- SPARK_WORKER_MEMORY: là dung lượng RAM.

Để compose các cluster, run the Docker compose file:

```
docker-compose up
```

Listing 8: Composing the cluster – run.sh

Sau khi hoàn tất, kiểm tra giao diện người dùng web thành phần:

- JupyterLab at localhost:8888;
- Spark master at localhost:8080;
- Spark worker I at localhost:8081;
- Spark worker II at localhost:8082;

1.2.6 Tạo ứng dụng PySpark

Với cụm đã được thiết lập và chạy, tạo ứng dụng PySpark.

```
1
2 from pyspark.sql import SparkSession
3
4 spark = SparkSession.\
5         builder.\
6         appName("pyspark-notebook").\
7         master("spark://spark-master:7077").\
8         config("spark.executor.memory", "512m").\
9         getOrCreate()
10
11 import wget
12
13 url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
      iris/iris.data"
14 wget.download(url)
15
16 data = spark.read.csv("iris.data")
17
18 data.show(n=5)
19
20 >>> +---+---+---+---+-----------+
21     |_c0|_c1|_c2|_c3|        _c4|
22     +---+---+---+---+-----------+
23     |5.1|3.5|1.4|0.2|Iris-setosa|
24     |4.9|3.0|1.4|0.2|Iris-setosa|
25     |4.7|3.2|1.3|0.2|Iris-setosa|
26     |4.6|3.1|1.5|0.2|Iris-setosa|
27     |5.0|3.6|1.4|0.2|Iris-setosa|
28     +---+---+---+---+-----------+
29 only showing top 5 rows
```

Listing 9: Composing the cluster – run.sh

Mở JupyterLab IDE và tạo sổ ghi chép Jupyter Python. Tạo ứng dụng PySpark bằng cách kết nối với nút chính Spark bằng đối tượng phiên Spark với các tham số sau:

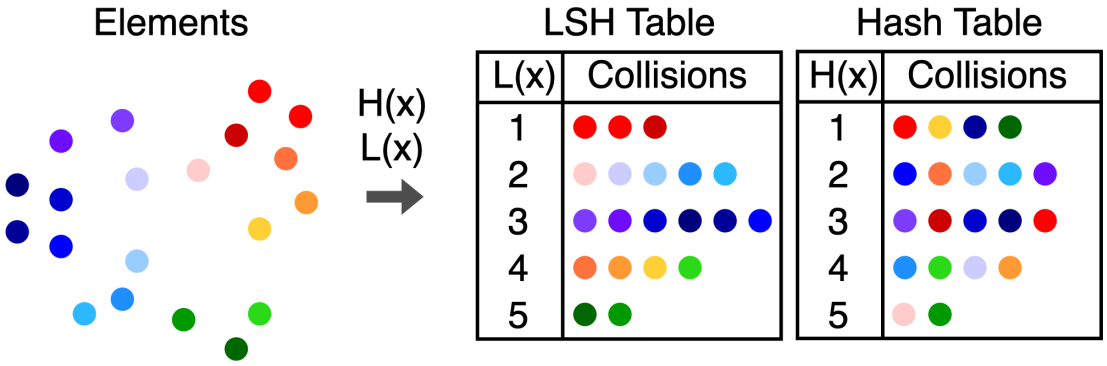
- *appName* là tên của ứng dụng
- *master* là URL kết nối master của Spark, được sử dụng bởi các nút của Spark worker để kết nối với master node của Spark;
- *config* là cấu hình Spark chung cho standalone mode. Ở đây, đang khớp bộ nhớ trình thực thi, tức là một quy trình JVM của worker Spark, với bộ nhớ worker node được cung cấp.

2 Ứng dụng giải thuật Locality Sensitive Hashing để tìm các đối tượng giống nhau trong tập dữ liệu lớn

2.1 Giới thiệu các khái niệm cơ bản về giải thuật Locality Sensitive Hashing

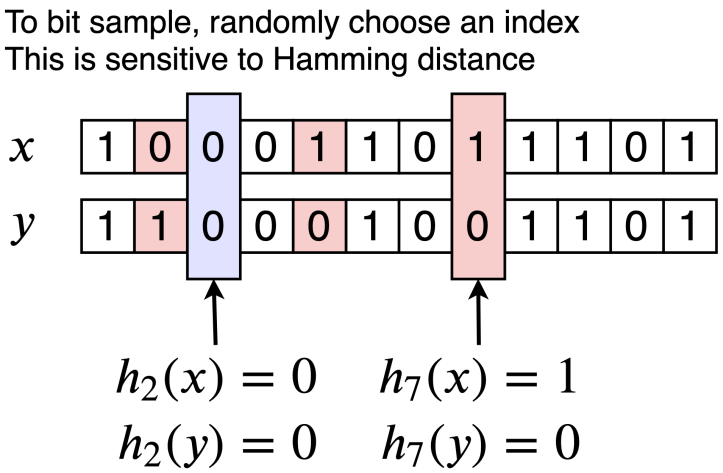
2.1.1 Khái niệm Locality Sensitive Hashing:

Về cơ bản, Locality Sensitive Hashing ánh xạ các đối tượng giống nhau vào cùng một rổ, đồng thời các đối tượng khác nhau sẽ nằm trong các rổ khác nhau. Hình dưới là ví dụ sử dụng hai hàm băm L(x) (Locality Sensitive Hashing) và H(x) (Hàm băm thường). L(x) giữ hầu hết những đối tượng trong cùng nhóm ban đầu vào cùng một rổ.



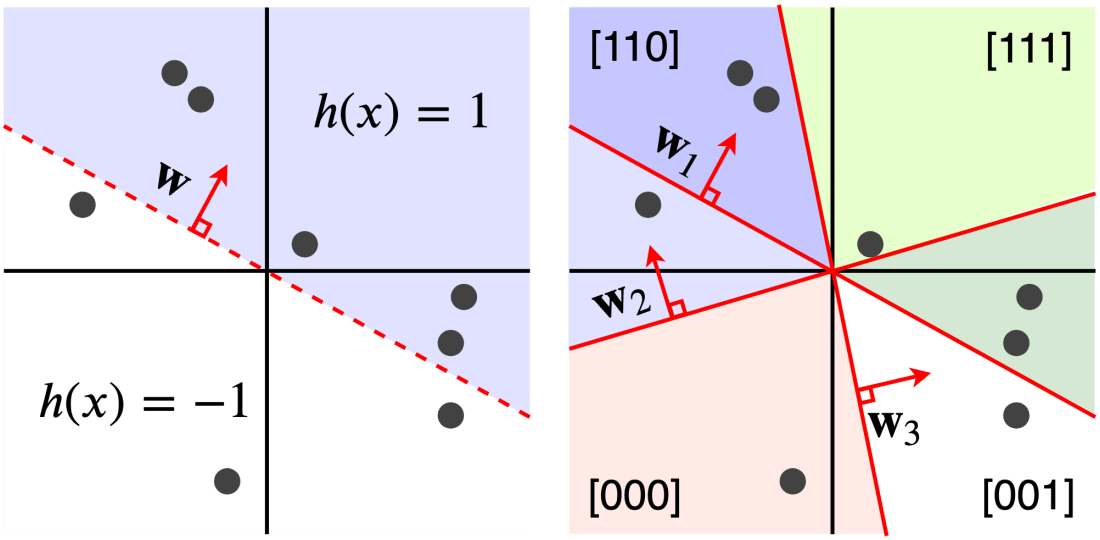
Tiếp theo là một vài phép đo được sử dụng phổ biến trong LSH.

2.1.2 Bit Sampling LSH



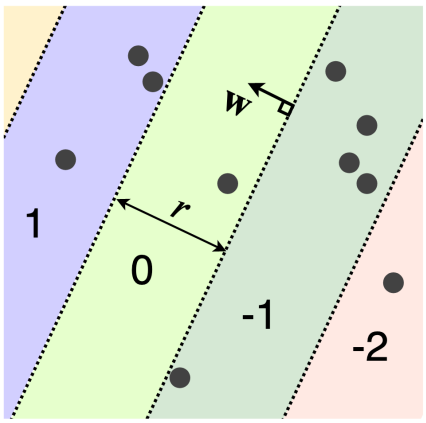
Bit Sampling LSH là một trong những hàm LSH đơn giản nhất và ít tốn chi phí nhất. Nó được liên kết với khoảng cách Hamming. Cho hai bit vector có độ dài n là x và y, khoảng cách Hamming d(x, y) là số bit khác nhau giữa hai vectơ. Chọn ngẫu nhiên m bit nhỏ hơn n-1 và tìm khoảng cách Hamming giữa 2 vector ứng với m bit đã chọn đó.

2.1.3 Signed Random Projections



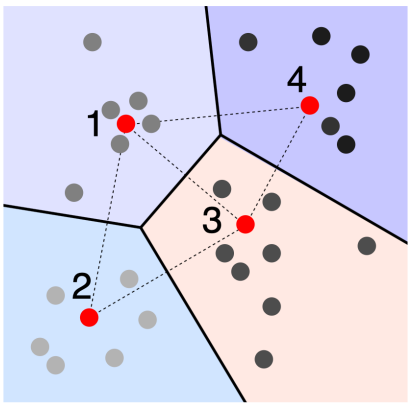
Về cơ bản, hàm này tìm m siêu phẳng dựa trên vector trực giao với biên w và lần lượt cắt đôi không gian và gán các điểm ở một nửa là 1 và nửa còn lại là -1. Sau m phép cắt ta sẽ được các rổ chứa đối tượng giống nhau.

2.1.4 Euclidean and Manhattan LSH



Các hàm LSH dùng khoảng cách Euclid (L2) và Manhattan (L1) cũng dựa trên các phép chiếu ngẫu nhiên, cắt không gian thành nhiều mảnh. Mỗi phần sẽ trở thành một rổ. Hàm băm giống như Signed Random Projections, nhưng làm tròn thay vì lấy dấu của phép chiếu w. Điều này tạo ra các rổ song song và vuông góc với w. Trong đó tham số r chỉ độ rộng của mỗi rổ.

2.1.5 Clustering LSH



Ý tưởng giải thuật là cố gắng tìm trung tâm các cụm sao cho ước lượng gần đúng việc phân các cụm dữ liệu gần nhau về cùng 1 rổ. Các trung tâm này có thể được tìm thấy bằng cách sử dụng phân cụm k-means, phân cụm lồi hoặc bất kỳ phương pháp phân cụm nào khác. Các rổ băm cho L(x) cũng giống như các ô Voronoi cho tập các centroid. Với một đối tượng cần băm, tìm trung tâm gần nhất và trả về chỉ số rổ đó.

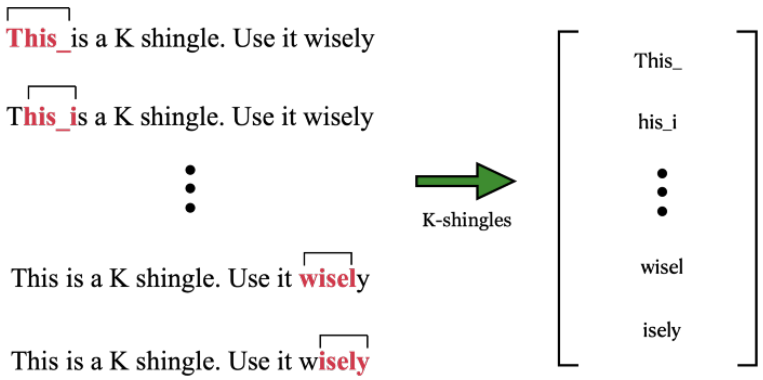
2.2 Ứng dụng giải thuật Locality Sensitive Hashing tìm các văn bản giống nhau

2.2.1 Mô tả tập văn bản đầu vào

Tạo một tập văn bản gồm 6 bài báo, trong đó 4 bài báo khác nhau hoàn toàn và 2 bài báo được trộn từ 4 bài báo trước, nhằm kiểm tra hiệu quả tìm các văn bản tương tự bằng giải thuật LSH.

```
#Out:
id: 0, doccument: article5.txt
id: 1, doccument: article2.txt
id: 2, doccument: article1.txt
id: 3, doccument: article6.txt
id: 4, doccument: article3.txt
id: 5, doccument: article4.txt
Average char-length: 3627.1666666666665
Min char-length: 2412
Max char-length: 5873
```

2.2.2 Chuyển từng văn bản thành tập các unique shingles



Trong phạm vi bài báo cáo ta chọn k (độ dài mỗi shingle) là 5. Kết quả:

```
#Out:
Found 3953 unique shingles, out of 5873 possible.
Found 2091 unique shingles, out of 3291 possible.
Found 2060 unique shingles, out of 3009 possible.
Found 2245 unique shingles, out of 3505 possible.
Found 2782 unique shingles, out of 3673 possible.
Found 1918 unique shingles, out of 2412 possible.
```

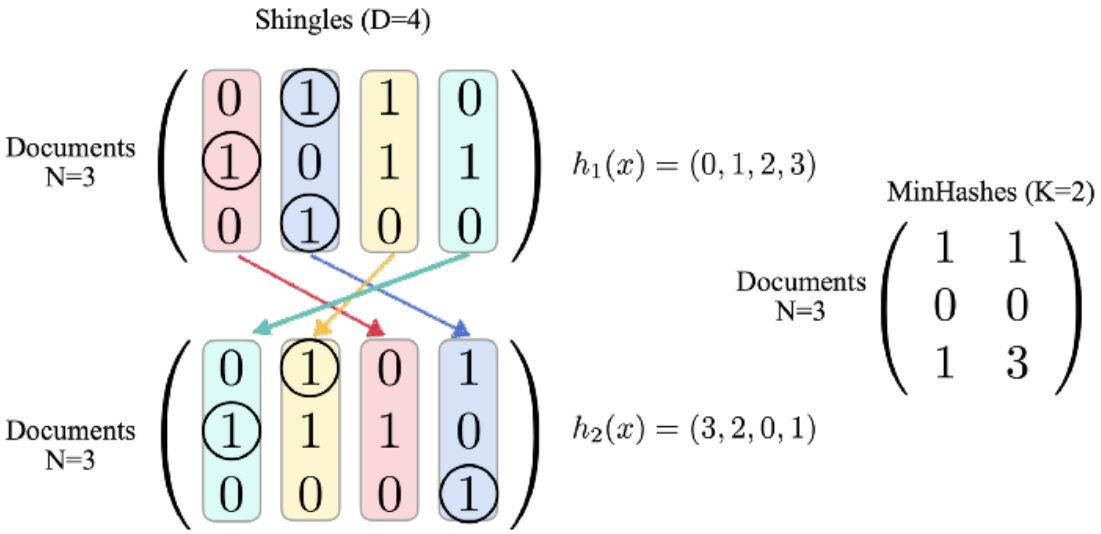
Brute force similarity scores bằng khoảng cách Jaccard:

```
#Out:
Pair  Score            Pair  Score
--------------           --------------
(0, 1)  0.294            (1, 5)  0.051
(0, 2)  0.336            (2, 3)  0.090
(0, 3)  0.384            (2, 4)  0.069
(0, 4)  0.083            (2, 5)  0.050
(0, 5)  0.400            (3, 4)  0.079
(1, 2)  0.093            (3, 5)  0.178
(1, 3)  0.704            (4, 5)  0.052
(1, 4)  0.081
```

Sau khi đã xác định được tập các shingle của từng văn bản, chuyển các văn bản thành 1 bit vector D chiều (số unique shingles có trong tất cả văn bản), 1 ý rằng có shingle đó trong văn bản ngược lại là 0.

2.2.3 Tạo Signature matrix

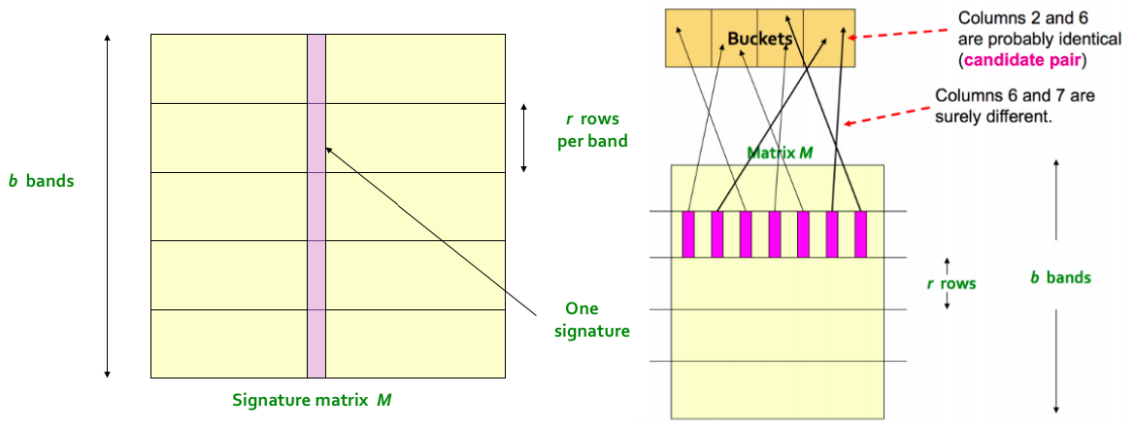
Mục đích nhằm làm giảm số chiều nhưng vẫn giữ được sự tương quan giữa từng vector với nhau.



2.2.4 Locality sensitive hashing

Mục đích nhằm chia các vector d chiều thành b band với độ rộng r. Nếu có bất kì 1 band nào cho kết quả trùng khớp với nhau giữa 2 vector thì ta xem như 2 vector đó là tương đương nhau. Trong đó b * r = d. Bằng việc thay đổi b và r, ta được các hệ quả sau:

- Tăng b, giảm r => Tăng false positive.
- Giảm b, tăng r => Giảm false positive.




```
#Out:
Pair     Approx     True     %Error
(0, 1)     0.500    0.294    69.78
(0, 2)     0.400    0.336    18.97
(0, 3)     0.500    0.384    30.17
(0, 4)     0.200    0.083    141.05
(0, 5)     0.600    0.400    49.93
(1, 2)     0.300    0.093    221.78
(1, 3)     0.800    0.704    13.68
(1, 4)     0.200    0.081    147.01
(1, 5)     0.200    0.051    289.08
(2, 3)     0.300    0.090    233.80
(2, 4)     0.000    0.069    100.00
(2, 5)     0.100    0.050    100.48
(3, 4)     0.200    0.079    152.47
(3, 5)     0.400    0.178    125.16
(4, 5)     0.200    0.052    286.93
True pairs: {(0, 1), (1, 3), (0, 5), (0, 3), (0, 2)}
Candidate pairs: {(0, 1), (1, 2), (1, 3), (0, 5), (2, 3), (0, 3),
    (0, 2), (3, 5)}
```

```
False negatives: 0
Potential false positives: 4
```

2.2.5 Sử dụng LSH trong thư viện MLlib - pyspark với Bucketed Random Projection for Euclidean Distance

Input của model BucketedRandomProjectionLSH là DataFrame với data là các Vector dense hoặc Vector sparse.

```
#Code:
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import col
#Chuyen cac vector shingle 0, 1 thanh Vectors.sparse
data = []
for i in range(len(documents)):
    potition = [j for j in range(num_col_shingle) if col_shingle[j] in doc_shingles[i]]
    sub_len = len(potition)
    data.append((i, Vectors.sparse(num_col_shingle, potition, [1.0] * sub_len)))
dfA = spark.createDataFrame(data, ["id", "features"])
#Tao model
from pyspark.ml.feature import BucketedRandomProjectionLSH

brp = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", bucketLength=2.0,
                                  numHashTables=3)
model = brp.fit(dfA)

print("Approximately joining dfA and dfB on Euclidean distance smaller than 58:")
model.approxSimilarityJoin(dfA, dfA, 58, distCol="EuclideanDistance").select(col("datasetA.id").alias("idA"),
col("datasetB.id").alias("idB"), col("EuclideanDistance")).filter("datasetA.id < datasetB.id").show()
#Out:
Approximately joining dfA and dfB on Euclidean distance smaller than 58:
+---+---+------------------+
|idA|idB| EuclideanDistance|
+---+---+------------------+
|  0|  1| 57.39337940912697|
|  0|  3| 52.51666402200353|
|  0|  5|50.149775672479336|
|  3|  5| 53.91660226683428|
|  1|  3|27.459060435491963|
|  0|  2|54.653453687758834|
+---+---+------------------+
```

Chọn khoảng cách là 58 khi chạy mô hình ta thu được các cặp văn bản tương tự nhau như trên, kết quả giống với thiết kế dữ liệu đầu vào.

2.2.6 Sử dụng LSH trong thư viện MLlib - pyspark với MinHash for Jaccard Distance

Tương tự BucketedRandomProjectionLSH, input của model MinHash for Jaccard Distance vẫn là DataFrame với data là các Vector dense hoặc Vector sparse.

```
#Code:
from pyspark.ml.feature import MinHashLSH

mh = MinHashLSH(inputCol="features", outputCol="hashes",
    numHashTables=5)
model = mh.fit(dfA)

print("Approximately joining dfA and dfB on distance smaller than
    0.83:")

model.approxSimilarityJoin(dfA, dfA, 0.83, distCol="
    JaccardDistance").select(col("datasetA.id").alias("idA"),
col("datasetB.id").alias("idB"),col("JaccardDistance")).filter("
    datasetA.id < datasetB.id").show()
#Out:
Approximately joining dfA and dfB on distance smaller than 0.83:
+---+---+------------------+
|idA|idB|   JaccardDistance|
+---+---+------------------+
|  1|  3|0.2962671905697446|
|  0|  5|0.5998092058192226|
|  0|  1|0.7055043906618119|
|  0|  3|0.6158999553372041|
+---+---+------------------+
```

Chọn khoảng cách là 0.83 khi chạy mô hình ta thu được các cặp văn bản tương tự nhau như trên, kết quả giống với thiết kế dữ liệu đầu vào.

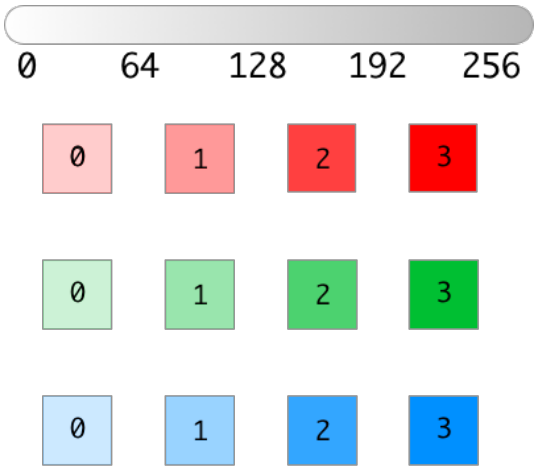
2.3 Ứng dụng giải thuật Locality Sensitive Hashing tìm các ảnh giống nhau

2.3.1 Mô tả tập ảnh đầu vào

Amsterdam Library of Object Images gồm ảnh màu 1000 vật thể khác nhau, mỗi vật được chụp theo 24 hướng khác nhau. Tập ảnh test là 10 cặp vật thể, mỗi cặp là 2 hướng nhìn khác nhau được lấy ra từ 1000 vật thể trong tập ALOI.

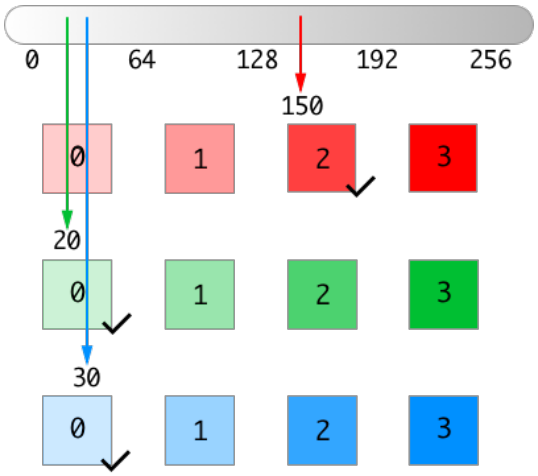
2.3.2 Tạo Signature matrix

Một ảnh màu đầu vào là một ma trận H*W (H là số pixel theo chiều dọc, W là chiều ngang), tại mỗi điểm ảnh là bộ 3 giá trị (số thực từ 0 đến 1 hoặc số nguyên từ 0 đến 255) cho mảng màu Red, Green và Blue theo đúng thứ tự.



Như hình ảnh trên ta chia mỗi thang màu thành 4 mức có độ rộng bằng nhau, vậy ta cần 4**3 rổ để lưu 1 ảnh - 1 vector pixelCounts có 64 chiều, khởi tạo pixelCounts = [0]*64. Duyệt H*W điểm ảnh, nếu giá trị tại điểm ảnh được hash vào index i của pixelCounts thì ta cộng giá trị tại index i lên 1.Sau khi duyệt hết ta sẽ được 1 vector Signature cho ảnh.
Cho ví dụ, tại 1 điểm ảnh bất kì có giá trị (150, 20, 30) như trong hình dưới, 150 ứng với giá trị 2, 20 ứng với giá trị 0, 30 ứng với giá trị 0. Lúc này tính index i bằng cách i = 2*(4**2) + 0*(4**1) + 0 = 32. Cộng pixelCounts[32] thêm 1. Dễ thấy rằng (155, 22, 35) cũng cho kết quả i tương tự, vì vậy càng chia nhỏ thang màu ta sẽ tạo được các vector Signature phân biệt nhau hơn, bù lại vector Signature sẽ tăng số chiều theo hàm mũ yêu cầu thời gian tính toán lâu hơn.

```python
#Code:
#x la gia tri cua mau, n la so thang mau duoc chia
def pixelNormalize(x, n):
    band = 1 / n
    for i in range(0, n):
        if x >= band * i and x <= band * (i + 1):
            return i
```



Sau khi hoàn thành cho H*W điểm ảnh ta sẽ thu được pixelCounts có giá trị sau:

R	G	B	# Pixels	R	G	B	# Pixels	R	G	B	# Pixels	R	G	B	# Pixels
0	0	0	0	1	0	0	17993	2	0	0	3506	3	0	0	9628
0	0	1	4476	1	0	1	224	2	0	1	18	3	0	1	0
0	0	2	1289	1	0	2	55	2	0	2	0	3	0	2	0
0	0	3	1975	1	0	3	3	2	0	3	2	3	0	3	0
0	1	0	4094	1	1	0	24483	2	1	0	3160	3	1	0	2796
0	1	1	928	1	1	1	45140	2	1	1	1312	3	1	1	80
0	1	2	1045	1	1	2	153	2	1	2	0	3	1	2	1
0	1	3	91	1	1	3	11	2	1	3	6	3	1	3	1
0	2	0	459	1	2	0	960	2	2	0	2243	3	2	0	3168
0	2	1	14	1	2	1	380	2	2	1	4355	3	2	1	929
0	2	2	15	1	2	2	20	2	2	2	2034	3	2	2	1105
0	2	3	20	1	2	3	19	2	2	3	37	3	2	3	7
0	3	0	2482	1	3	0	57	2	3	0	67	3	3	0	376
0	3	1	17	1	3	1	30	2	3	1	123	3	3	1	268
0	3	2	0	1	3	2	4	2	3	2	164	3	3	2	1283
0	3	3	53	1	3	3	30	2	3	3	58	3	3	3	2303

2.3.3 Sử dụng LSH trong thư viện MLlib - pyspark với MinHash for Euclidean Distance

Input của model MinHash for Euclidean Distance vẫn là DataFrame với data là các Vector dense hoặc Vector sparse.

```
#Code:
for file, value in imageAll.items():
    print(value[0])
    imagesNormalized = value[1]
    data = []
    for i in range(len(imagesNormalized)):
        data.append((int(i), Vectors.dense(imagesNormalized[i]),))

    dfA = spark.createDataFrame(data, ["id", "features"])
    mh = MinHashLSH(inputCol="features", outputCol="hashes",
   numHashTables=5)
    model = mh.fit(dfA)

    print("Approximately joining dfA and dfB on Euclidean distance
    smaller than 0.55:")
    model.approxSimilarityJoin(dfA, dfA, 0.55, distCol="
   EuclideanDistance")\
        .filter("datasetA.id < datasetB.id").show()
#Out:
Approximately joining dfA and dfB on Euclidean distance smaller
   than 0.55:
+--------------------+--------------------+-------------------+
|            datasetA|            datasetB|  EuclideanDistance|
+--------------------+--------------------+-------------------+
```

```
|[9, [15160.0,0.0,...|[15, [15412.0,0.0...|0.18803418803418803|
|[3, [17860.0,0.0,...|[5, [18936.0,0.0,...|0.23076923076923073|
|[13, [16858.0,1.0...|[18, [18099.0,0.0...| 0.2289719626168224|
|[2, [16987.0,0.0,...|[6, [17055.0,0.0,...|0.39862542955326463|
|[4, [10274.0,0.0,...|[16, [10596.0,0.0...| 0.3048128342245989|
|[10, [17160.0,0.0...|[12, [16811.0,0.0...|0.40740740740740744|
|[8, [18255.0,0.0,...|[17, [19648.0,0.0...| 0.3035714285714286|
|[1, [14084.0,0.0,...|[7, [14061.0,0.0,...| 0.4665354330708661|
+--------------------+--------------------+-------------------+
```

2.3.4 Brute force calculation với Cosin Similarity Distance

Ta tính khoảng cách để phân biệt 2 hình ảnh dựa theo góc giữa 2 vector Signature.

```
#Code:
#Cosin Similarity function
def cossim(u,v):
    norm = np.linalg.norm(u)*np.linalg.norm(v)
    cosine = u@v/norm
    ang = np.arccos(cosine)
    return 1-ang/np.pi

thresh = 0.979

for file, value in imageAll.items():
    if file != 'mix':
        continue
    print(value[0])
    imagesNormalized = np.array(value[1])
    images = value[2]
    true_pairs_dict = {}
    start = time.time()
    for (i,u),(j,v) in itertools.combinations([(i,x) for i,x in
    enumerate(imagesNormalized)],2):
        val = cossim(u,v)
        if val > thresh:
            true_pairs_dict[(i,j)] = val

    # save just the keys without the values. Easier to compare
    later to LSH
    true_pairs = set(true_pairs_dict.keys())
    print(f"Discovered pairs: " + str(len(true_pairs)) + ' pairs')
    for k, v in true_pairs_dict.items():
        print(f"Pair: {k},\tSimilarity: {v:.4f}.")
        display_multiple_img(images[k[0]], images[k[1]])
    bruteForceTime = time.time() - start
    print(f"Brute force calculation time: {bruteForceTime}")
```

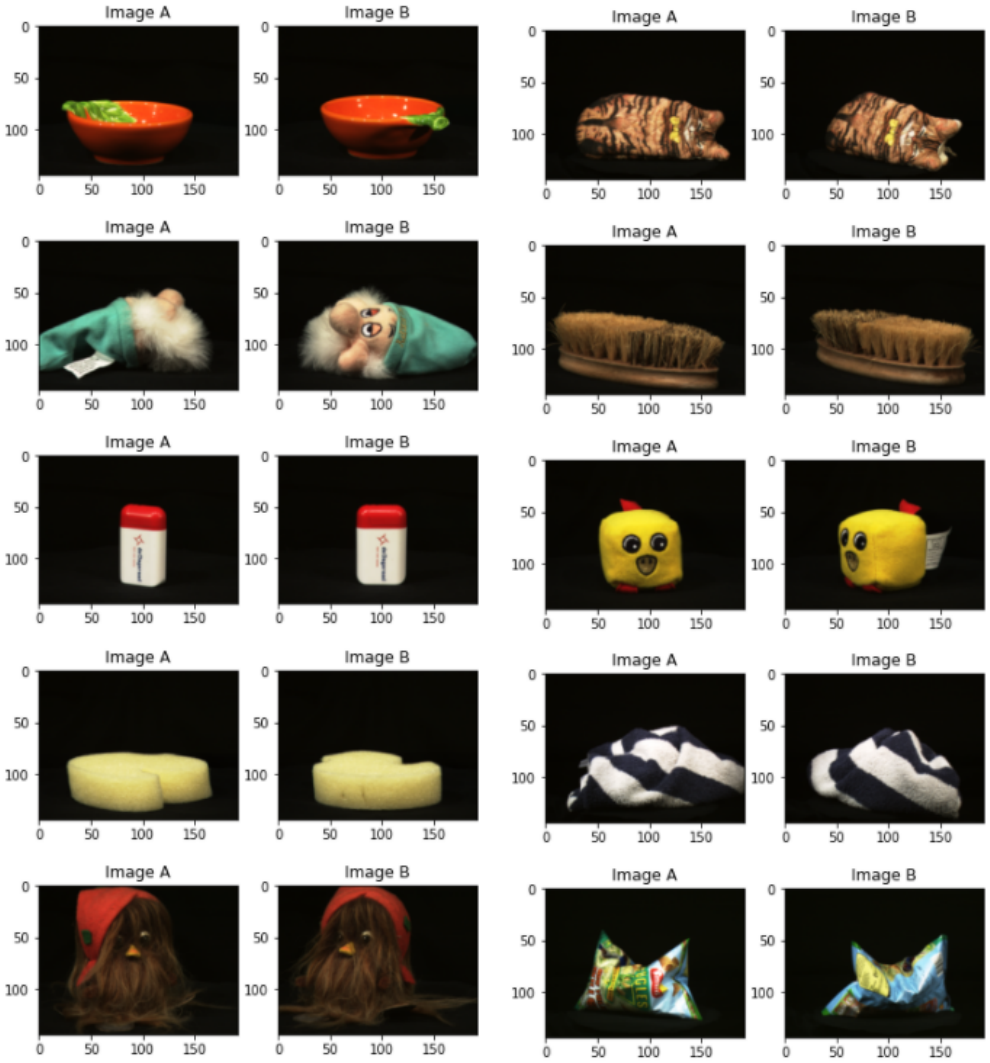
```
#Out:
Discovered pairs: 10 pairs
Pair: (0, 11),  Similarity: 0.9853.
Pair: (1, 7), Similarity: 0.9809.
Pair: (2, 6), Similarity: 0.9824.
Pair: (3, 5), Similarity: 0.9872.
Pair: (4, 16),  Similarity: 0.9794.
Pair: (8, 17),  Similarity: 0.9918.
Pair: (9, 15),  Similarity: 0.9897.
Pair: (10, 12), Similarity: 0.9875.
Pair: (13, 18), Similarity: 0.9869.
Pair: (14, 19), Similarity: 0.9916.
Brute force calculation time: 0.5156736373901367
```

2.3.5 Locality Sensitive Hashing dựa trên Cosin Similarity Distance

```
#Code:
#Locality Sensitive Hashing theo Euclidean distance
def lsh_euclidean(A, b, r, thresh):
  N, D = A.shape
  n = b*r

  R = A@np.random.randn(D, n)
  S = np.where(R>0, 1, 0)

  S = np.split(S, b, axis=1)

  binary_column = 2**np.arange(r).reshape(-1,  1)

  S = np.hstack([M@binary_column for M in S])

  d = collections.defaultdict(set)
  with np.nditer(S,flags=['multi_index']) as it:
      for x in it:
          d[int(x)].add(it.multi_index[0])

  candidate_pairs = set()
  for k,v in d.items():
      if len(v) > 1:
          for pair in itertools.combinations(v, 2):
              candidate_pairs.add(tuple(sorted(pair)))

  lsh_pairs = set()
  for (i, j) in candidate_pairs:
      if cossim(A[i],A[j]) > thresh:
          lsh_pairs.add((i, j))

  return lsh_pairs
```

```
33
34 #Out:
35 Discovered pairs: 10 pairs
36 Brute force calculation time: 0.3671119213104248
37 LSH fasten more Brute force: 1.40
38 {(2, 6), (9, 15), (14, 19), (10, 12), (4, 16), (0, 11), (13, 18),
    (1, 7), (8, 17), (3, 5)}
```

Brute force calculation với Cosin Similarity Distance và Locality Sensitive Hashing với Cosin Similarity Distance thu được các cặp ảnh tương đồng giống nhau, nhưng thời gian dùng LSH nhanh hơn 1.4 lần. Vì chỉ dùng 20 ảnh để test nên chưa thấy được sự hiệu quả của LSH với O(n) và Brute force O(n**2). Nếu làm việc với tập dữ liệu lớn hiệu năng LSH sẽ rõ ràng hơn.



3 Tham khảo

Tài liệu

[1] *Spark Overview*, https://spark.apache.org/docs/latest/#downloading

[2] *Apache repository*, https://archive.apache.org/dist/spark/

[3] *Master class*, https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/deploy/master/Master.scala

[4] *Worker class*, https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/deploy/worker/Worker.scala

[5] *Locality Sensitive Hashing: How to Find Similar Items in a Large Set, with Precision*, https://towardsdatascience.com

[6] *Locality Sensitive Hashing*, https://spark.apache.org/docs/2.2.3/ml-features.html#locality-sensitive-hashing

[7] *Find Similar Images Based On Locality Sensitive Hashing*, https://lkaihua.github.io/posts/find-similar-images-based-on-locality-sensitive-hashing

[8] *4 Pictures that Explain LSH - Locality Sensitive Hashing Tutorial*, https://randorithms.com/2019/09/19/Visual-LSH.html

[9] *Amsterdam Library of Object Images (ALOI)*, https://aloi.science.uva.nl