Image Similarity Detection

Using LSH and Tensorflow

Andrey Gusev

June 6, 2018

Help you discover and do what you love.

100b+
Pins

2b+
Boards

People on Pinterest each month 200m+

10b+
Recommendations/Day

Agenda

1. Neardup, clustering and LSH
2. Candidate generation
3. Deep dive
4. Candidate selection
5. TF on Spark

Neardup


FIRE-ROASTED RICOTTA WITH CORN AND PEPPERS

Not Neardup

Unrelated

Duplicate

Neardup

Clustering

Not An Equivalence Class

Formulation

For each image find a canonical image which represents an equivalence class.

Problem

Neardup is not an equivalence relation because neardup relation is not a transitive relation.

It means we can not find a perfect partition such that all images within a cluster are closer to each other than to the other clusters.

Incremental approximate K-Cut

Incrementally:

1. **Generate candidates via batch LSH search**
2. **Select candidates via a TF model**

3. Take a transitive closure over selected candidates
4. Pass over clusters and greedily select sub-clusters (K-Cut).

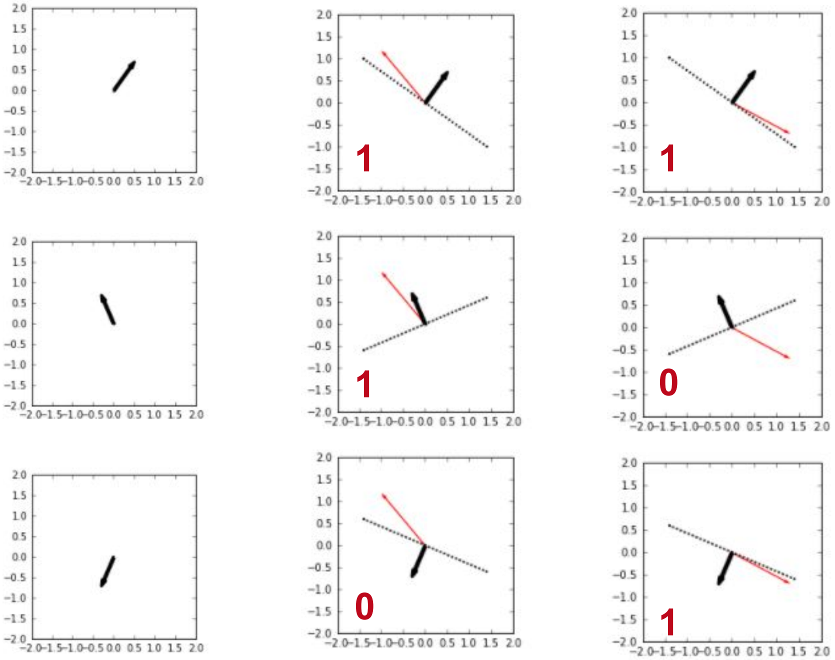
LSH

Embeddings and LSH

- **Visual Embeddings** are high-dimensional vector representations of entities (in our case images) which capture semantic similarity.
    - Produced via Neural Networks like VGG16, Inception, etc.

- Locality-sensitive hashing or **LSH** is a modern technique used to reduce dimensionality of high-dimensional data while preserving pairwise distances between individual points.

LSH: Locality Sensitive Hashing



- Pick random projection vectors **(black)**
- For each embeddings vector determine on which side of the hyperplane the embeddings vector lands

- On the same side: set bit to 1
- On different side: set bit to 0

Result 1: **<1 1 0>**
Result 2: **<1 0 1>**

LSH terms

Pick optimal number of terms and bits per term

- *1001110001011000 -> [00]1001 - [01]1100 - [10]0101 - [11]1000*
- *[x] → a term index*

Candidate Generation

Neardup Candidate Generation

- Input Data:

  ```
  RDD[(ImgId, List[LSHTerm])] // billions
  ```

- Goal:

  ```
  RDD[(ImgId, TopK[(ImgId, Overlap))]
  ```

Nearest Neighbor (KNN) problem formulation

Neardup Candidate Generation

Given a set of documents each described by LSH terms, example:

`A → (1,2,3)`

`B → (1,3,10)`

`C → (2,10)`

And more generally:

$D_i \rightarrow [t_j]$

Where each $\mathbf{D_i}$ is a document and $\mathbf{[t_j]}$ is a list of LSH terms (assume each is a 4 byte integer)

Results:

`A → (B,2), (C,1)`

`B → (A,2), (C,1)`

`C → (A,1), (B,1)`

Spark Candidate Generation

1. Input `RDD[(ImgId, List[LSHTerm])]` ← both **index** and **query** sets
2. flatMap, groupBy input into `RDD[(LSHTerm, PostingList)]` ← an **inverted index**
3. flatMap, groupBy into `RDD[(LSHTerm, PostingList)]` ← a **query list**
4. Join (2) and (3), flatMap over queries posting list, and groupBy query ImgId; `RDD[(ImgId, List[PostingList])]` ← **search results by query.**
5. Merge *List[List[ImgId]]* into `TopK(ImgId, Overlap)` counting number of times each ImgId is seen → ***`RDD[ImgId, TopK[(ImgId, Overlap)]`***.

* `PostingList = List[ImgId]`

Orders of magnitude too slow.

Deep Dive

Dictionary encoding

```
def mapDocToInt(termIndexRaw: RDD[(String, List[TermId])]): RDD[(String, DocId)] = {
 // ensure that mapping between string and id is stable by sorting
 // this allows attempts to re-use partial stage completions
 termIndexRaw.keys.distinct().sortBy(x => x).zipWithIndex()
}
```

```
val stringArray = (for (ind <- 0 to 1000) yield randomString(32)).toArray
val intArray = (for (ind <- 0 to 1000) yield ind).toArray
```

108128 Bytes*
4024 Bytes*
25x

* https://www.javamex.com/classmexer/

Variable Byte Encoding

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

- One bit of each byte is a continuation bit; overhead
- int → byte (best case)
- 32 char string up to 25x4 = 100x memory reduction

https://nlp.stanford.edu/IR-book/html/htmledition/variable-byte-codes-1.html

Inverted Index Partitioning

Inverted index is skewed

```
/**
 * Build partitioned inverted index by taking module of docId into partition.
 */
def buildPartitionedInvertedIndex(flatTermIndexAndFreq: RDD[(TermId, (DocId, TermFreq))]):
RDD[((TermId, TermPartition), Iterable[DocId])] = {

 flatTermIndexAndFreq.map { case (termId, (docId, _)) =>
   // partition documents within the same term to improve balance
   // and reduce the posting list length
   ((termId, (Math.abs(docId) % TERM_PARTITIONING).toByte), docId)
 }.groupByKey()
}
```

Packing

(Int, Byte) => Long

Before:

*Unsorted: **128.77 MB** in 549ms*

*Sort+Limit: 4.41 KB in **7511ms***

After:

*Unsorted: **38.83 MB** in 219ms*

*Sort+Limit: 4.41 KB in **467ms***

```
def packDocIdAndByteIntoLong(docId: DocId, docFreq: DocFreq): Long = {
  (docFreq.toLong << 32) | (docId & 0xffffffffL)
}

def unpackDocIdAndByteFromLong(packed: Long): (DocId, DocFreq) = {
  (packed.toInt, (packed >> 32).toByte)
}
```

Slicing

Split query set into slices to reduce spill and size for "widest" portion of the computation. Union at the end.

Additional Optimizations

- **Cost based optimizer** - significant improvements to runtime can be realized by analyzing input data sets and setting performance parameters automatically.
- **Counting** - jaccard overlap counting is done via low level, high performance collections.
- **Off heaping** serialization when possible *(spark.kryo.unsafe)*.

Generic Batch LSH Search

- Can be applied generically to KNN, embedding agnostic.

- Can work on arbitrary large query set via slicing.

Candidate Selection

TF DNN Classifier

- Transfer learning over VGG16
- Visual embeddings
- XOR hamming bits
- Learning still happens at >1B pairs
- Batch size of 1024, Adam optimizer

Vectorization: mapPartitions + grouped

- During training and inference vectorization reduces overhead.
- Spark mapPartitions + grouped allows for large batches and controlling the size. Works well for inference.
- 2ms/prediction on c3.8xl CPUs with network of 10MM parameters .

```
input.mapPartitions { partition: Iterator[(ImgInfo, ImgInfo)] =>

   // break down large partitions into groups and score per group
   partition.grouped(BATCH_SIZE).flatMap { group: Seq[(ImgInfo, ImgInfo)] =>
     // create tensors and score as features: Array[Array[Float]] --> Tensor.create(features)
    }
}
```

One TF Session per JVM

- Reduce model loading overhead, load once per JVM; thread-safe.

```
object TensorflowModel {
 lazy val model: Session = {
   SavedModelBundle.load(...).session()
 }
}
```

Summary

- Candidate Generation uses Batch LSH Search over terms from visual embeddings.

- Batch LSH scales to billions of objects in the index and is embedding agnostic.

- Candidate Selection uses a TF classifier over raw visual embeddings.

- Two-pass transitive closure to cluster results.

Thanks!