Document Vector on Hacker News Dataset

Yiran Sheng

12/16/2015

Project Introduction and Highlights

This project aims to perform classification and regression tasks on Hacker News dataset, using language features extracted purely from submission titles and comments texts. Specifically, we utilizes neural language models `word2vec`(Mikolov et al. 2013) and paragraph vector (Le and Mikolov 2014) to extract word and document embeddings. In addition, we setup a web serving layer using Locality Sensitive Hashing for fast querying similar items (words of documents) from the dataset.

We are able to train both `word2vec` and `doc2vec` models on all submissions from the dataset (1.8 million rows) - with both submission title and top level comments texts as input. However, subcequent regression and classification models using document vectors as inputs do not perform well at all. This is somewhat expected, as submission quality depends a lot more on linked webpage content rather than its title wording.

Regression Task: prediction submission score

Model	Mean Square Error
Random Forrest, log score	1.1030
Random Forrest, raw score	1632.3640

Classification Task: prediction whether submission is flagged

Model	False Positive	False Negative
Flagged title classification	0.2100	0.7336
Flagged title classification(tfidf)	0.96437	0.022589

On the other hand, the language models themselves prove useful in capturing the HN audience's vocabulary and language semantics. We seeded the `Word2Vec` model with pre-trained `GloVe`[link] vectors, and an example query for similar words to "github" produces:

```
>>> mod.most_similar("github")
[(u'sourceforge', 0.8272511959075928), (u'codeplex', 0.7057037949562073),
(u'appstore', 0.5705950260162354), (u'iheartradio', 0.5395055413246155),
(u'stickam', 0.5194863080978394), (u'spotify', 0.5147385001182556),
(u'ustream', 0.5136187672615051), (u'webpage', 0.5071792006492615),
(u'bandcamp', 0.5047038793563843), (u'repository', 0.5015265941619873)]
```

After training on the Hacker News dataset, the same query yields:

```
>>> mod.most_similar("github")
```

```
[(u'sourceforge', 0.7420217394828796), (u'bitbucket', 0.7206469774246216),
(u'heroku', 0.7109839916229248), (u'stackoverflow', 0.6746459603309631),
(u'digitalocean', 0.6725961565971375), (u'cloudflare', 0.6723034977912903),
(u'pastebin', 0.6686413288116455), (u'trello', 0.6658083200454712),
(u'twofactor', 0.6638146638870239), (u'shortener', 0.6634611487388611)]
```

Our domain-specific model seem to encode the associations for technology and startup related terms better than general purpose models.

Overview of Word2Vec / Doc2Vec

The last two years, the method and tool developed by Mikolov et al. (2013) for learning word embeddings has gained a lot of traction. The model forms the basis for the study of Le & Mikolov (2014) which extends word vectors to document vectors. At its core, both models attempt to associate words with points in high dimentional space. The general idea is to train a softmax classifier using a given word in a sentence as input and its surrounding context words as output (continous bag of words), or the other way around (skip-gram). Document vector works similarly by taking document tag/id as inputs as well in the prediction task.

Data Source and Summary

We use Hacker News October data dump hosted on Google BigQuery [link] as the data source for this project. The dataset is 4GB in size, and contains 2 million submissions and 20 million comments. The full table `fh-bigquery:hackernews.full_201510` has the following schema.

Field	Type	NULLABLE
by	STRING	NULLABLE
score	INTEGER	NULLABLE
time	INTEGER	NULLABLE
title	STRING	NULLABLE
type	STRING	NULLABLE
url	STRING	NULLABLE
text	STRING	NULLABLE
parent	INTEGER	NULLABLE
deleted	BOOLEAN	NULLABLE
dead	BOOLEAN	NULLABLE
descendants	INTEGER	NULLABLE
id	INTEGER	NULLABLE
ranking	INTEGER	NULLABLE

Although relatively small in size, the dataset provides ample information for text mining. We run a few `SQL` queries directly in BigQuery to get a sense of various measures of corpus size relevant to word2vec and doc2vec model training.

Total word count from story submissions (estimate)

```
SELECT
  SUM(word_count)
FROM (
  SELECT
    COUNT(SPLIT(title, " ")) AS word_count
  FROM
    [fh-bigquery:hackernews.full_201510]
  WHERE
    type="story"
    AND title IS NOT NULL)
```

Result:

13680833 (13.7 million)

Total word count from comments (estimate)

```
SELECT
  SUM(word_count)
FROM (
  SELECT
    COUNT(SPLIT(REGEXP_REPLACE(text, r'\s+', ' '), ' ')) AS word_count
  FROM
    [fh-bigquery:hackernews.full_201510]
  WHERE
    type="comment"
    AND text IS NOT NULL)
```

Result:

495472673 (495 million)

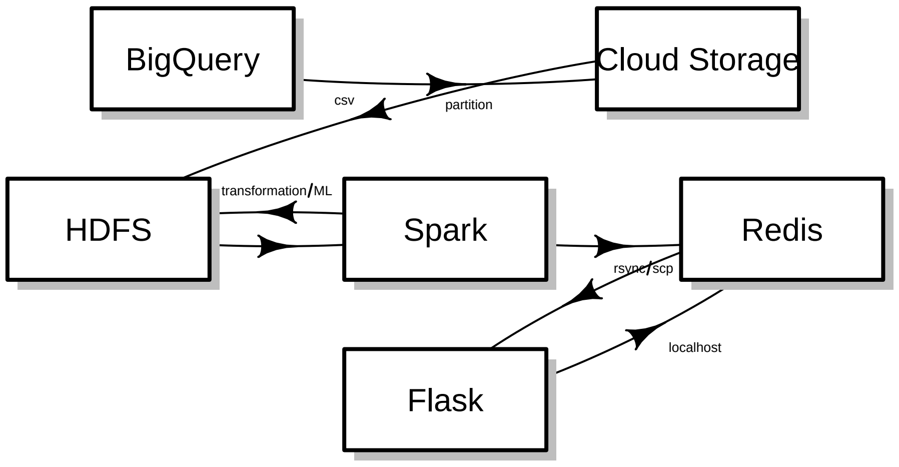
Comment word count quantiles (estimate)

```
SELECT
  QUANTILES(word_count, 5)
FROM (
  SELECT
    COUNT(SPLIT(REGEXP_REPLACE(text, r'\s+', ' '), ' ')) AS word_count
  FROM
    [fh-bigquery:hackernews.full_201510]
  WHERE
    type="comment"
    AND text IS NOT NULL)
```

Result:

Min.	25%	Median	75%	Max.
0	16	34	66	5439

Cluster Setup



Raw dataset is exported to Google Cloud Storage[link] in csv format using Google's python client. In this step, BigQuery automatically break table into partitions, we ended up with 57 csv files (with header) in GCS. A Hadoop/Spark cluster is set up on Softlayer, containing 5 nodes. Each node has 2 cores and 16 GB of RAM. A simple shell script run as a map-side only job downloads csv files from GCS, removes the header line and uploads them to HDFS.

Next, we utilizes Spark (`spark-sql`) and `spark-csv`[link] package to parse raw csv files as Spark `DataFrame` - all fields as treated as nullable, string fields and empty strings are kept as-is. Then we enforces schema through custom `spark-sql udf`s, each returning `scala`'s `Option` typed values. For example, the following UDF converts a `bool` column:

```
sqlContext.udf.register("optBool", (s: String) => {
    Option(s).filter(_.trim.nonEmpty).map(_.toLowerCase).flatMap({
        case "true" => Some(true)
        case "false" => Some(false)
        case _ => None})
})
```

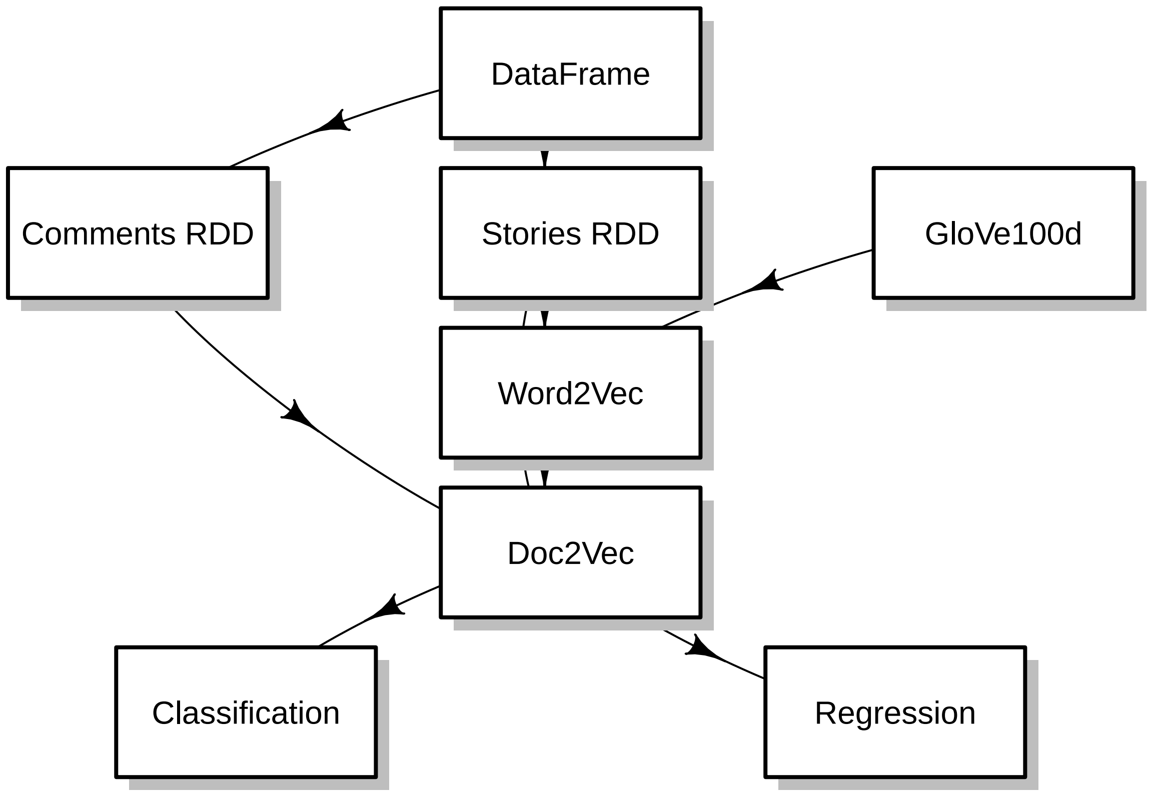
Subsequent transformations and M/L tasks are performed on Spark, with intermediate or reusable results (eg. trained models) sometimes stored back on HDFS.

Serving Layer

Trained models/results are exported to local file system and `rsync`ed to a separated server node (we have not automated this step due do time constraint). A few upload scripts reads model information, and upload useful bits to redis. In case of document vectors, we also processes them and pre-computes their `LSH` keys before dumping to redis. The hashing function(s) used to get bucket keys for each vector is also stored in `redis`.

Finally, we have a simple `Flask` api server running and communicating with `redis` on localhost. The server maintains a `LSH` query engine (python library `NearPy`), a light-weight "Fake" `Doc2Vec` model with just enough information loaded in memory to process real-time document vector inference for incoming sentences/documents. This api server performs nearest neighbor search for incoming http requests.

Machine Learning Pipeline



At the beginning of all M/L tasks, pre-processed dataset (stored as `parquet` on HDFS) is loaded into Spark as a `DataFrame` object. Some SQL queries on it produces two `RDD`s of interest:

1. `RDD[(id, title)]` for all stories/submissions
2. `RDD[(parentId, text)]` for all top level comments (whose `parent` is a story)

We produce a `Word2Vec` model through a single iteration training using story titles as inputs. This step serves two purposes, to build the vocabulary of stories and achieve a good starting point for initial weights of the model. Model is set to train on skip-gram, negative sampling model (compared to hierarchical sampling, this is computationally cheaper and compatible with `Doc2Vec`'s `PV-DBOW`).

Next, we merge this model with pre-trained `GloVe` vectors of the same dimension to re-use high quality word embeddings produced from much large wikipedia dataset. However, `GloVe` vectors are only half of model parameters, the hidden layer weights is not packaged. Therefore, we cannot "freeze" the `Word2Vec` model just yet when moving on to training document vectors, as we need to re-learn the hidden weights still.

Finally, both stories and comments (top level only) are used to train the `Doc2Vec` model. Each "document" consists of the story title and all top level comments. This process was carried out in mini-batches, document vector, word vector and hidden weights are learned jointly. Multiple parallel strategies are tried as detailed in the next section.

The output of `Doc2Vec` (document vectors) are stored in HDFS as pickleFiles, which we use for classification and regression tasks using standard models (RandomForrest and Logistic Regression) from Spark MLLib.

Distributed Doc2Vec Model Training

Document vector (paragraph vector, thought vector) (Le and Mikolov 2014) is not widely implemented in open source libraries. Python package `gensim` has a complete and optimized implementation which is most widely used. However, no distributed version is available due to recency of the model as well as general obstacles in parallelizing Stochastic Gradient Decent algorithms.

Our approach is to rely on `gensim`, but attempting to integrate it with Spark's `RDD` centric work flow.

Existing Solutions

Spark MLLib Word2Vec

Spark MLLib implements skip-gram, hierarchical sampling `word2vec` models, which closely mirrors the original `c` code in (Mikolov et al. 2013). The key difference between Spark's parallel version compared to single node version such as `gensim` is Spark wraps the training loops in a `mapPartitionsWithIndex` call on training data `RDD`[source]. In addition, helper data structures such as vocabulary, vocabulary hash and exponent table are broadcasted and cached on to all worker nodes.

We end up adopting both ideas in our own implementation.

DeepDist and Downpour SGD

DeepDist (Neumann 2015) is small python library to facilitate training deep networks in a distributed fashion. It implements a Downpour SGD (Dean et al. 2012) like algorithm. In our use case:

- Master Node hold gensim `Doc2Vec` model
- Worker Node fetch mode from master over http, serialization via `pickle`
  - Master node runs `Flask` server in threaded mode
- Worker train on each partition through `mapPartitions`
- Worker sends back deltas in model params
- Thread coordination and synchronization with a simple read-write lock
  - many concurrent readers
  - one writer, priority over readers

However, this approach still leaves many things to be desired. Here's some key challenges we encountered:

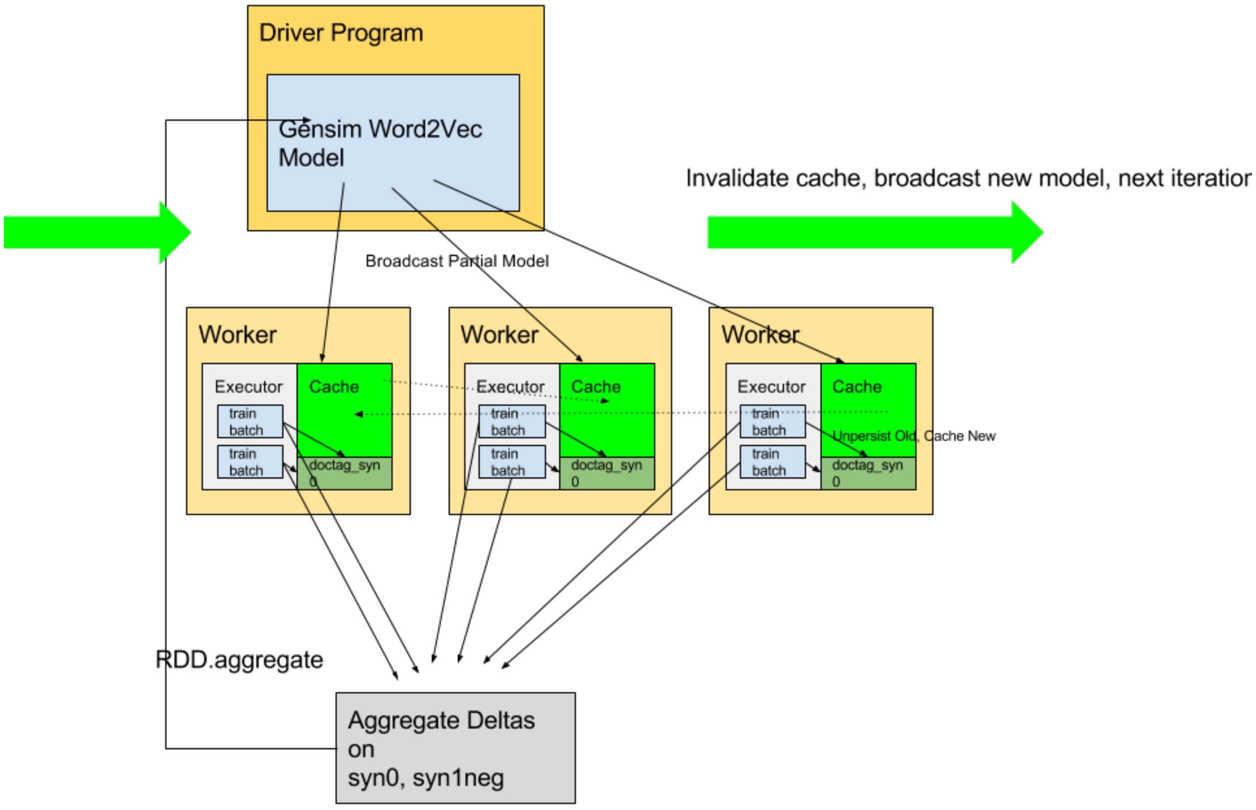
- Limited to `yarn-client` deploy mode
- Limited by driver and executor memory (Doc2Vec models are very large)
  - `gensim` uses `numpy` memory map, and stores a huge chunk of model paramters on disk only
  - We ended up modify `DeepDist` heavily to run external `rsync` commands for model synchronization
- Limited by `python`'s `pickle`, which has troubles to serialize large `numpy` arrays
- GC Pressure

Custom Implementation

We created a prototype solution for running gensim Doc2Vec on Spark. Only PV-DBOW with negative sampling is implemented. In general, Stochastic Gradient Decent algorithms are hard to parallelize, however `Doc2Vec` model belongs to a special class. The docuement vectors or input layer weights for document tags are local to training data. Each document only updates its corresponding row in the document vector matrix. This makes it possible to paralle partially the model and take advantage of data locality.

The goal of Doc2Vec is to learn vector representation of each document in training set. For example, a dataset of 10 million documents and vector size 300, requires 300,000,000 floating number parameters (or a 300x1000,000 array). Fortunately, each data point (a.ka. sentence or document) only updates its corresponding row in the weights matrix during training process, therefore, it's possible to parallelize the model by zipping its parameters with training dataset: each partition only holds the parameters relevant to its own share of data.

`gensim` is used as a basis for this setup, training for sentence vectors are adapted to work on RDDs.



When training `Doc2Vec` in `PV-DBOW` model and using negative sampling, three `numpy` arrays are of interests in `gensim`, the fully captures the model state:

1. `model.syn0`
2. `model.syn1neg`
3. `model.docvecs.doctag_syn0`

In our implementation[source], we keep `syn0` and `syn1neg` centralized, as they are of limited size (size of total vocabulary). doctag_syn0 is held as RDD (each partition holds a single numpy array for it). Word2Vec model is broadcasted to all partitions.

In each training iteration, the following happens:

1. On each partition, Cython and BLAS powered procedure train_document_dbow from gensim.models.doc2vec_inner is called, and trains word vectors, document vector and hidden layer weights jointly
2. We record triplet (syn0 deltas, syn1neg deltas, doctag_syn0) and produce a new RDD with each partition holding a single triplet; and this new RDD is cached (as it will be used twice)
3. We aggregate all deltas through Spark's RDD.aggregate api, to sum all deltas, then apply deltas to model object in driver program
4. Previous generation of model broadcased is unpersisted, new model is broadcasted to all executors (runs actual training as aggregate is a Spark action)
5. Create new inputs from new RDD in step 4, training will not be re-run as we have cached results, then we invalid stale triplet RDD from previous iteration

By tweaking num_partitions and num_iterations, we can balance the trade-off between accuracy, speed and network overhead.

Test Results

Cornell Movie Reviews Dataset is used to test the approach out. Model is trained on 5 partitions and 20 iterations, and we were able to classify movie reviews labels with about **11%** error rate only from docvectors, with balanced false negative and postie rate:

```
*** Error Rate: 0.107995 ***
*** False Positive Rate: 0.107799 ***
*** False Negative Rate: 0.108191 ***
```

Query Optimization: Locality Sensitive Hashing

Once we obtain the vector form of submissions, we export it from HDFS to local file system and them upload to a seperated server node running redis on localhost. A few "tricks" are employed to allow for fast, real-time query of similar items (both words and submissions).

1. Locality Sensitive Hashing (through python package NearPy) with random binary projection hashing functions

2. "Fake" gensim model object:

   - a fake model only loads model parameters partially (syn0, syn1neg1, vocab etc.)
   - call train_document_dbow from gensim.models.doc2vec_inner when infering vectors for new documents

An example web client is also implemented[link] using angularjs, which connects to api server and allows end users to query for similar items.

Further Improvements

This project can be improved upon in many ways. Here's a to-do list for near term or lower hanging goals:

- Use better language parsers and tokenizers to clean up input texts
- Implement `LSH` for cosine similarity in Spark directly, so document vectors and their bucket hash are exported together to downstream consumers
- Stream `Doc2Vec` training results to `redis` host directly
- Join document vectors with other features extracted from dataset for M/L tasks

Further more, our protype implementation for distributed `Doc2Vec` training has much room to further evolve:

- Add option to training on mini batches
- Adopt the asynchronous design from `DeepDist`
- Implement AdaGrad for training loops to determine a per-parameter learning rate to combat the accuracy lost when parallelizing gradient decent

References

Dean, Jeffrey, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, et al. 2012. "Large Scale Distributed Deep Networks." In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a Meeting Held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 1232–40. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.

Le, Quoc V., and Tomas Mikolov. 2014. "Distributed Representations of Sentences and Documents." *CoRR* abs/1405.4053. http://arxiv.org/abs/1405.4053.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." *CoRR* abs/1301.3781. http://arxiv.org/abs/1301.3781.

Neumann, Dirk. 2015. "DeepDist: Lightning-Fast Deep Learning on Spark via Parallel Stochastic Gradient Updates." https://github.com/dirkneumann/deepdist.