Fast Large-Scale Approximate Graph Construction for NLP

Amit Goyal and Hal Daumé III
Dept. of Computer Science
University of Maryland
College Park, MD
{amit,hal}@umiacs.umd.edu

Raul Guerra
Dept. of Computer Science
University of Maryland
College Park, MD
rguerra@cs.umd.edu

Abstract

Many natural language processing problems involve constructing large nearest-neighbor graphs. We propose a system called **FLAG** to construct such graphs approximately from large data sets. To handle the large amount of data, our algorithm maintains approximate counts based on sketching algorithms. To find the approximate nearest neighbors, our algorithm pairs a new distributed online-PMI algorithm with novel fast approximate nearest neighbor search algorithms (variants of PLEB). These algorithms return the approximate nearest neighbors quickly. We show our system's efficiency in both intrinsic and extrinsic experiments. We further evaluate our fast search algorithms both quantitatively and qualitatively on two NLP applications.

1 Introduction

Many natural language processing (NLP) problems involve graph construction. Examples include constructing polarity lexicons based on lexical graphs from WordNet (Rao and Ravichandran, 2009), constructing polarity lexicons from web data (Velikovich et al., 2010) and unsupervised part-of-speech tagging using label propagation (Das and Petrov, 2011). The later two approaches construct nearest-neighbor graphs between word pairs by computing nearest neighbors between word pairs from large corpora. These nearest neighbors form the edges of the graph, with weights given by the distributional similarity (Turney and Pantel, 2010) between terms. Unfortunately, computing the distributional similarity between all words in a large vocabulary is computationally and memory intensive when working with large amounts of data (Pantel et al., 2009). This bottleneck is typically addressed by means of commodity clusters. For example, Pantel et al. (2009) compute distributional similarity between 500 million terms over a 200 billion words in 50 hours using 100 quad-core nodes, explicitly storing a similarity matrix between 500 million terms.

In this work, we propose Fast Large-Scale Approximate Graph (**FLAG**) construction, a system that constructs a fast large-scale approximate nearest-neighbor graph from a large text corpus. To build this system, we exploit recent developments in the area of approximation, randomization and streaming for large-scale NLP problems (Ravichandran et al., 2005; Goyal et al., 2009; Levenberg et al., 2010). More specifically we exploit work on Locality Sensitive Hashing (LSH) (Charikar, 2002) for computing word-pair similarities from large text collections (Ravichandran et al., 2005; Van Durme and Lall, 2010). However, Ravichandran et al. (2005) approach stored an enormous matrix of all unique words and their contexts in main memory, which is infeasible for very large data sets. A more efficient online framework to locality sensitive hashing (Van Durme and Lall, 2010; Van Durme and Lall, 2011) computes distributional similarity in a streaming setting. Unfortunately, their approach can handle only additive features like raw-counts, and not non-linear association scores like pointwise mutual information (PMI), which generates better context vectors for distributional similarity (Ravichandran et al., 2005; Pantel et al., 2009; Turney and Pantel, 2010).

In **FLAG**, we first propose a *novel* distributed online-PMI algorithm (Section 3.1). It is a streaming method that processes large data sets in one pass while distributing the data over commodity clusters

Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pages 1069–1080, Jeju Island, Korea, 12–14 July 2012. ©2012 Association for Computational Linguistics

and returns context vectors weighted by pointwise mutual information (PMI) for all the words. Our distributed online-PMI algorithm makes use of the Count-Min (CM) sketch algorithm (Cormode and Muthukrishnan, 2004) (previously shown effective for computing distributional similarity in our earlier work (Goyal and Daumé III, 2011)) to store the counts of all words, contexts and word-context pairs using only $8GB$ of main memory. The main motivation for using the CM sketch comes from its linearity property (see last paragraph of Section 2) which makes CM sketch to be implemented in distributed setting for large data sets. In our implementation, **FLAG** scaled up to 110 GB of web data with 866 million sentences in less than 2 days using 100 quad-core nodes. Our intrinsic and extrinsic experiments demonstrate the effectiveness of distributed online-PMI.

After generating context vectors from distributed online-PMI algorithm, our goal is to use them to find fast approximate nearest neighbors for all words. To achieve this goal, we exploit recent developments in the area of existing randomized algorithms for random projections (Achlioptas, 2003; Li et al., 2006), Locality Sensitive Hashing (LSH) (Charikar, 2002) and improve on previous work done on PLEB (Point Location in Equal Balls) (Indyk and Motwani, 1998; Charikar, 2002). We propose novel variants of PLEB to address the issue of reducing the pre-processing time for PLEB. One of the variants of PLEB (FAST-PLEB) with considerably *less* pre-processing time has effectiveness comparable to PLEB. We evaluate these variants of PLEB both quantitatively and qualitatively on large data sets. Finally, we show the applicability of large-scale graphs built from **FLAG** on two applications: the Google-Sets problem (Ghahramani and Heller, 2005), and learning concrete and abstract words (Turney et al., 2011).

2 Count-Min sketch

The Count-Min (CM) sketch (Cormode and Muthukrishnan, 2004) belongs to a class of 'sketch' algorithms that represents a large data set with a compact summary, typically much smaller than the full size of the input by processing the data in one pass. The following surveys comprehensively review the streaming literature (Rusu and Dobra, 2007; Cormode and Hadjieleftheriou, 2008) and sketch techniques (Charikar et al., 2004; Li et al., 2008; Cormode and Muthukrishnan, 2004; Rusu and Dobra, 2007). In our another recent paper (Goyal et al., 2012), we conducted a systematic study and compare many sketch techniques which answer point queries with focus on large-scale NLP tasks. In that paper, we empirically demonstrated that CM sketch performs the best among all the sketches on three large-scale NLP tasks.

CM sketch uses hashing to store the approximate frequencies of all items from the large data set onto a small sketch vector that can be updated and queried in constant time. CM has two parameters ϵ and δ: ϵ controls the amount of tolerable error in the returned count and δ controls the probability with which the error exceeds the bound ϵ.

CM sketch with parameters (ϵ,δ) is represented as a two-dimensional array with width w and depth d; where w and d depends on ϵ and δ respectively. We set w=$\frac{2}{\epsilon}$ and d=$\log(\frac{1}{\delta})$. The depth d denotes the number of pairwise-independent hash functions employed by the CM sketch; and the width w denotes the range of the hash functions. Given an input stream of items of length N $(\mathrm{x}_1, \mathrm{x}_2 \ldots \mathrm{x}_N)$, each of the hash functions h_k:$\{\mathrm{x}_1, \mathrm{x}_2 \ldots \mathrm{x}_N\} \rightarrow \{1 \ldots w\}, \forall 1 \leq k \leq d$, takes an item from the input stream and maps it into a position indexed by the corresponding hash function.

UPDATE: For each new item "x" with count c, the sketch is updated as:

$$\text{sketch}[k, h_k(\mathrm{x})] \leftarrow \text{sketch}[k, h_k(\mathrm{x})] + c, \ \forall 1 \leq k \leq d.$$

QUERY: Since multiple items can be hashed to the same index for each row of the array, hence the stored frequency in each row is guaranteed to *overestimate* the true count, which makes it a biased estimator. Therefore, to answer the point query (QUERY (x)), CM returns the minimum over all the d positions indexed by the hash functions.

$$\hat{c}(\mathrm{x}) = \min_k \ \text{sketch}[k, h_k(\mathrm{x})], \ \forall 1 \leq k \leq d.$$

All reported frequencies by CM exceed the true frequencies by at most ϵN with probability of at least $1 - \delta$. The space used by the algorithm is $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$. Constant time of $\mathrm{O}(\log(\frac{1}{\delta}))$ per each update and query operation.

CM sketch has a linearity property which states that: Given two sketches s_1 and s_2 computed (us-

ing the same parameters w and d, and the same set of d hash functions) over different input streams; the sketch of the combined data stream can be easily obtained by adding the individual sketches in $O(d \times w)$ time which is independent of the stream size. This property enables sketches to be implemented in distributed setting, where each machine computes the sketch over a small portion of the corpus and makes it *scalable* to large datasets.

The idea of conservative update (Estan and Varghese, 2002) is to only increase counts in the sketch by the minimum amount needed to ensure that the estimate remains accurate. We (Goyal and Daumé III, 2011) used CM sketch with conservative update (CM-CU sketch) to show that the update reduces the amount of over-estimation error by a factor of at least 1.5 on NLP data and showed the effectiveness of CM-CU on three important NLP tasks. The QUERY procedure for CM-CU is identical to Count-Min. However, to UPDATE an item "x" with frequency c, first we compute the frequency $\hat{c}(\mathrm{x})$ of this item from the existing data structure:

$(\forall 1 \leq k \leq d, \hat{c}(\mathrm{x}) = \min_k \ \mathrm{sketch}[k, h_k(\mathrm{x})])$

and the counts are updated according to:

$\mathrm{sketch}[k, h_k(\mathrm{x})] \leftarrow \max\{\mathrm{sketch}[k, h_k(\mathrm{x})], \hat{c}(\mathrm{x}) + c\}$.

The intuition is that, since the point query returns the minimum of all the d values, we will update a counter only if it is necessary as indicated by the above equation. This heuristic avoids the unnecessary updating of counter values to reduce the over-estimation error.

3 FLAG: Fast Large-Scale Approximate Graph Construction

We describe a system, **FLAG**, for generating a nearest neighbor graph from a large corpus. For every node (word), our system returns top l approximate nearest neighbors, which implicitly defines the graph. Our system operates in four steps. First, for every word "z", our system generates a sparse context vector $(\langle(\mathrm{c}_1, v_1); (\mathrm{c}_2, v_2) \ldots ; (\mathrm{c}_d, v_d)\rangle)$ of size d where c_d denotes the context and v_d denotes the PMI (strength of association) between the context c_d and the word "z". The context can be lexical, semantic, syntactic, and/or dependency units that co-occur with the word "z". We compute this efficiently using a new distributed online Pointwise Mutual Information algorithm (Section 3.1). Second, we project all the words with context vector size d onto k random vectors and then binarize these random projection vectors (Section 3.2). Third, we propose novel variants of PLEB (Section 3.3) with less pre-processing time to represent data for fast query retrieval. Fourth, using the output of variants of PLEB, we generate a small set of potential nearest neighbors for every word "z" (Section 3.4). From this small set, we can compute the Hamming distance between every word "z" and its potential nearest neighbors to return the l nearest-neighbors for all unique words.

3.1 Distributed online-PMI

We propose a *new* distributed online Pointwise Mutual Information (PMI) algorithm motivated by the online-PMI algorithm (Van Durme and Lall, 2009b) (page 5). This is a streaming algorithm which processes the input corpus in one pass. After one pass over the data set, it returns the context vectors for all query words. The original online-PMI algorithm was used to find the top-d verbs for a query verb using the highest approximate online-PMI values using a Talbot-Osborne-Morris-Bloom[1] (TOMB) Counter (Van Durme and Lall, 2009a). Unfortunately, this algorithm is prohibitively slow when computing contexts for *all* words, rather than just a small query set. This motivates us to propose a distributed variant that enables us to scale to large data and large vocabularies.[2]

We make three modifications to the original online-PMI algorithm and refer to it as the "modified online-PMI algorithm" shown in Algorithm 1. First, we use Count-Min with conservative update (CM-CU) sketch (Goyal and Daumé III, 2011) instead of TOMB. We prefer CM because it enables distribution due to its linearity property (Section 2) and footnote #1. Distribution using TOMB is not known in literature and we will like to explore that direction in future. Second, we store the counts of words ("z"), contexts ("y") and word-context pairs all together in

[1]TOMB is a variant of CM sketch which focuses on reducing the bit size of each counter (in addition to the number of counters) at the cost of incurring more error in the counts.

[2]The serialized online-PMI algorithm took a week to generate context vectors for all the words from **GW** (Section 4.1).

Algorithm 1 Modified online-PMI

Require: Data set D, buffer size B

Ensure: context vectors V, mapping word z to d-best contexts in priority queue $\langle$ y,PMI(z, y)$\rangle$

1: initialize CM-CU sketch to store approximate counts of words, context and word-context pairs
2: **for** each buffer B in the data set D **do**
3: initialize S to store $\langle$z,y$\rangle$ observed in B
4: **for** $\langle$z,y$\rangle$ in B **do**
5: set S ($\langle$z,y$\rangle$) =1
6: insert z, y and pair $\langle$z,y$\rangle$ in sketch
7: **end for**
8: **for** x in set S **do**
9: recompute vectors V(x) using current contexts in priority queue and {y|S($\langle$z,y$\rangle$)=1}
10: **end for**
11: **end for**
12: **return** context vectors V

the CM-CU sketch (in the original online-PMI algorithm, exact counts of words and contexts were stored in a hash table; only the pairs were stored in the TOMB data structure). Third, in the original algorithm, for each "z" a vector of top-d contexts are modified at the end of each buffer (refer Algorithm 1). However, in our algorithm, we only modify the list of those "z"'s which appeared in the recent buffer rather than modifying for all the "z"'s (Note, if "z" does not appear in the recent buffer, then its top-d contexts cannot be changed. Hence, we only modify those "z"s which appear in the recent buffer).

In our distributed online-PMI algorithm, first we split the data into chunks of 10 million sentences. Second, we run the modified online-PMI algorithm on each chunk in distributed setting. This stores counts of all words ("z"), contexts ("y") and word-context pairs in the CM-CU sketch, and store top-d contexts for each word in priority queues. In third step, we merge all the sketches using linearity property to sum the counts of the words, contexts and word-context pairs. Additionally we merge the lists of top-d contexts for each word. In the last step, we use the single merged sketch and merged top-d contexts list to generate the final distributed online-PMI top-d contexts list.

It takes around one day to compute context vectors for all the words from a chunk of 10 million sentences using first step of distributed online-PMI. We generated context vectors for all the 87 chunks (110 GB data with 866 million sentences: see Table 1) in one day by running one process per chunk over a cluster. The first step of the algorithm involves traversing the data set and is the most time intensive step. For the second step, the merging of sketches is fast, since sketches are two dimensional array data structures (we used the sketch of size 2 billion counters with 3 hash functions). Merging the lists of top-d contexts for each word is embarrassingly parallel and fast. The last step to generate the final top-d contexts list is again embarrassingly parallel and fast and takes couple of hours to generate the top-d contexts for all the words from all the chunks. If implemented serially the "modified online-PMI algorithm" on 110 GB data with 866 million sentences would take approximately 3 months.

The downside of the distributed online-PMI is that it splits the data into small chunks and loses information about the global best contexts for a word over all the chunks. The algorithm locally computes the best contexts for each chunk, that can be bad if the algorithm misses out globally *good* contexts and that can affect the accuracy of downstream application. We will demonstrate in our experiments (Section 4.2) by using distributed online-PMI, we do not lose any significant information about global contexts and perform comparable to offline-PMI over an intrinsic and extrinsic evaluation.

3.2 Dimensionality Reduction from $\mathbb{R}^D$ to $\mathbb{R}^k$

We are given context vectors for Z words, our goal is to use k random projections to project the context vectors from $\mathbb{R}^D$ to $\mathbb{R}^k$. There are total D unique contexts ($D >> k$) for all Z words. Let $(\langle(\mathrm{c}_1, v_1); (\mathrm{c}_2, v_2) \ldots; (\mathrm{c}_d, v_d)\rangle)$ be sparse context vectors of size d for Z words. For each word, we use hashing to project the context vectors onto k directions. We use k pairwise independent hash functions that maps each of the d context (c_d) dimensions onto $\beta_{d,k} \in \{-1, +1\}$; and compute inner product between $\beta_{d,k}$ and v_d. Next, $\forall k$, $\sum_d \beta_{d,k}.v_d$ returns the k random projections for each word "z". We store the k random projections for all words (mapped to integers) as a matrix A of size of $k \times Z$.

The mechanism described above generates random projections by implicitly creating a random projection matrix from a set of $\{-1, +1\}$. This idea of creating implicit random projection matrix

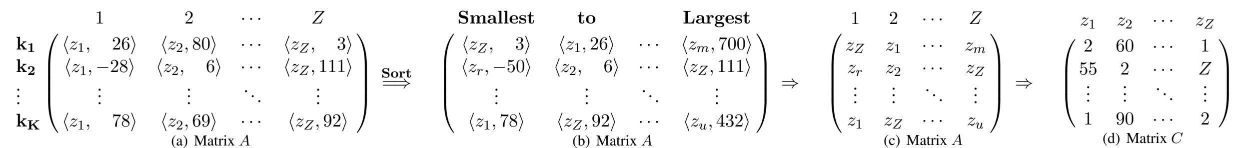

Figure 1: First matrix pairs the words $1 \cdots Z$ and their random projection values. Second matrix sorts each row by the random projection values from smallest to largest. Third matrix throws away the projection values leaving only the words. Fourth matrix maps the words $1 \cdots Z$ to their sorted position in the third matrix for each k. This allows constant query time for all the words.

is motivated by the work on stable random projections (Li et al., 2006; Li et al., 2008), Count sketch (Charikar et al., 2004), feature hashing (Weinberger et al., 2009) and online Locality Sensitive Hashing (LSH) (Van Durme and Lall, 2010). The idea of generating random projections from the set $\{-1, +1\}$ was originally proposed by Achlioptas (2003).

Next we create a binary matrix B using matrix A by taking sign of each of the entries of the matrix A. If $A(i,j) \geq 0$, then $B(i,j) = 1$; else $B(i,j) = 0$. This binarization creates Locality Sensitive Hash (LSH) function that preserves the cosine similarity between every pair of word vectors. This idea was first proposed by Charikar (2002) and used in NLP for large-scale noun clustering (Ravichandran et al., 2005). However, in large-scale noun clustering work, their approach had to store the random projection matrix of size $D \times k$; where D denotes the number of all unique contexts (which is generally large and $D >> Z$) and in this paper, we do not explicitly require storing a random projection matrix.

3.3 Representation for Fast-Search

We describe three approaches to represent the data (matrix A and B from Section 3.2) in such a manner that finding nearest neighbors is fast. These three approaches differ in amount of pre-processing time. First, we propose a naive baseline approach using random projections independently with the best pre-processing time. Second, we describe PLEB (Point Location in Equal Balls) (Indyk and Motwani, 1998; Charikar, 2002) with the worst pre-processing time. Third, we propose a variant of PLEB to reduce its pre-processing time.

3.3.1 Independent Random Projections (IRP)

Here, we describe a naive baseline approach to arrange nearest neighbors next to each other by using Independent Random Projections (IRP). In this approach, we pre-process the matrix A. First for matrix A, we pair the words $z_1 \cdots z_Z$ and their random projection values as shown in Fig. 1(a). Second, we sort the elements of each row of matrix A by their random projection values from smallest to largest (shown in Fig. 1(b)). The sorting step takes $O(ZlogZ)$ time (We can assume k to be a constant). The sorting operation puts all the nearest neighbor words (for each k independent projections) next to each other. After sorting the matrix A, we throw away the projection values leaving only the words (see Fig. 1(c)). To search for a word in matrix A in constant time, we create another matrix C of size $(k \times Z)$ (see Fig. 1(d)). Matrix C maps the words $z_1 \cdots z_Z$ to their sorted position in the matrix A (see Fig. 1(c)) for each k.

3.3.2 PLEB

PLEB (Point Location in Equal Balls) was first proposed by Indyk and Motwani (1998) and further improved by Charikar (2002). The improved PLEB algorithm puts in operation *all* k random projections together. It randomly permutes the ordering of k binary LSH bits (stored in matrix B) for all the words p times. For each permutation it sorts all the words lexicographically based on their permuted LSH representation of size k. The sorting operation puts all the nearest neighbor words (using k projections together) next to each other for all the permutations. In practice p is generally large, Ravichandran et al. (2005) used $p = 1000$ in their work.

In our implementation of PLEB, we have a matrix A of size $(p \times Z)$ similar to the first matrix in Fig. 1(a). The main difference to the first matrix in Fig. 1(a) is that bit vectors of size k are used for sorting rather than using scalar projection values. Similar to Fig. 1(c) after sorting, bit vectors are discarded and

a matrix C of size $(p \times Z)$ is used to map the words $1 \cdots Z$ to their sorted position in the matrix A. Note, in IRP approach, the size of A and C matrix is $(k \times Z)$. In PLEB generating random permutations and sorting the bit vectors of size k involves worse pre-processing time than using IRP. However, spending more time in pre-processing leads to finding better approximate nearest neighbors.

3.3.3 FAST-PLEB

To reduce the pre-processing time for PLEB, we propose a variant of PLEB (FAST-PLEB). In PLEB, while generating random permutations, it uses all the k bits. In this variant, for each random permutation we randomly sample without replacement q ($q << k$) bits out of k. We use q bits to represent each permutation and sort based on these q bits. This makes pre-processing *faster* for PLEB. Section 4.3 shows that FAST-PLEB only needs $q = 10$ to perform comparable to PLEB with $q = 3000$ (that makes FAST-PLEB 300 times faster than PLEB). Here, again we store matrices A and C of size $(p \times Z)$.

3.4 Finding Approximate Nearest Neighbors

The goal here is to exploit three representations discussed in Section 3.3 to find approximate nearest neighbors quickly. For all the three methods (IRP, PLEB, FAST-PLEB), we can use the same fast approximate search which is simple and fast. To search a word "z", first, we can look up matrix C to locate the k positions where "z" is stored in matrix A. This can be done in constant time (Again assuming k (for IRP) and p (for PLEB and FAST-PLEB) to be a constant.). Once, we find "z" in each row, we can select b (beam parameter) neighbors ($b/2$ neighbors from left and $b/2$ neighbors from right of the query word.) for all the k or p rows. This can be done in constant time (Assuming k, p and b to be constants.). This search procedure produces a set of bk (IRP) or bp (PLEB and FAST-PLEB) potential nearest neighbors for a query word "z". Next, we compute Hamming distance between query word "z" and the set of potential nearest neighbors from matrix B to return l closest nearest neighbors. For computing hamming distance, all the approaches discussed in Section 3.3 require all k random projection bits.

4 Experiments

We evaluate our system **FLAG** for fast large-scale approximate graph construction. First, we show that using distributed online-PMI algorithm is as effective as offline-PMI. Second, we compare the approximate nearest neighbors lists generated by **FLAG** against the exact nearest neighbor lists. Finally, we show the quality of our approximate similarity lists generated by **FLAG** from the web corpus.

4.1 Experimental Setup

Data sets: We use two data sets: Gigaword (Graff, 2003) and a copy of news web (Ravichandran et al., 2005). For both the corpora, we split the text into sentences, tokenize and convert into lower-case. To evaluate our approximate graph construction, we evaluate on three data sets: Gigaword (**GW**), Gigaword + 50% of web data (**GWB50**) and Gigaword + 100% ((**GWB100**)) of web data. Corpus statistics are shown in Table 1. We define the context for a given word "z" as the surrounding words appearing in a window of 2 words to the left and 2 words to the right. The context words are concatenated along with their positions -2, -1, +1, and +2.

Corpus	GW	GWB50	GWB100
Unzipped Size (GB)	12	60	110
# of sentences (Million)	57	463	866
# of tokens (Billion)	2.1	10.9	20.0

Table 1: Corpus Description

4.2 Evaluating Distributed online-PMI

Experimental Setup: First we do an intrinsic evaluation to quantitatively evaluate the distributed online-PMI vectors against the offline-PMI vectors computed from Gigaword (**GW**). Offline-PMI computed from the sketches have been shown as effective as exact PMI by Goyal and Daumé III (2011). To compute offline-PMI vectors, we do two passes over the corpus. In the first pass, we store the counts of words, contexts and word-context pairs computed from **GW** in the Count-Min with conservative update (CM-CU) sketch. We use the CM-CU sketch of size 2 billion counters (bounded 8 GB memory)

with 3 hash functions. In second pass, using the aggregated counts from the sketch, we generate the offline-PMI vectors of size $d = 1000$ for every word. For rest of this paper for distributed online-PMI, we set $d = 1000$ and the size of the buffer=$10,000$ and we split the data sets into small chunks of 10 million sentences.

Intrinsic Evaluation: We use four kinds of measures: precision (P), recall (R), f-measure (F1) and Pearson's correlation (ρ) to measure the overlap in the context vectors obtained using online and offline PMI. ρ is computed between contexts that are found in offline and online context vectors. We do this evaluation on 447 words selected from the concatenation of four test-sets mentioned in the next paragraph. On these 447 words, we achieve an average P of .97, average R of .96 and average F1 of .97 and a perfect average ρ of 1. This evaluation show that the vectors obtained using online-PMI are as effective as offline-PMI.

Extrinsic Evaluation: We also compare online-PMI effectiveness on four test sets which consist of word pairs, and their corresponding human rankings. We generate the word pair rankings using online-PMI and offline-PMI strategies. We report the Pearson's correlation (ρ) between the human and system generated similarity rankings. The four test sets are: **WS-353** (Finkelstein et al., 2002) is a set of 353 word pairs. **WS-203**: A subset of WS-353 with 203 word pairs (Agirre et al., 2009). **RG-65**: (Rubenstein and Goodenough, 1965) has 65 word pairs. **MC-30**: A subset of RG-65 dataset with 30 word pairs (Miller and Charles, 1991).

The results in Table 2 shows that by using distributed online-PMI (by making a single pass over the corpus) is comparable to offline-PMI (which is computed by making two passes over the corpus).

For generating context vectors from **GW**, for both offline-PMI and online-PMI, we use a frequency cutoff of 5 for word-context pairs to throw away the rare terms as they are sensitive to PMI (Church and Hanks, 1989). Next, **FLAG** generates online-PMI vectors from **GWB50** and **GWB100** and uses frequency cutoffs of 15 and 25. The higher frequency cutoffs are selected based on the intuition that, with more data, we get more noise, and hence not considering word-context pairs with frequency less than 25 will be better for the system. As **FLAG** is going to use the context vectors to find nearest neighbors, we also throw away all those words which have ≤ 50 contexts associated with them. This generates context vectors for $57,930$ words from **GW**; $95,626$ from **GWB50** and $106,733$ from **GWB100**.

Test Set	WS-353	WS-203	RG-65	MC-30
Offline-PMI	.41	.55	.40	.52
Online-PMI	.41	.56	.39	.51

Table 2: Evaluating word pairs ranking with online and offline PMI. Scores are evaluated using ρ metric.

	10		25		50		100	
	R	ρ	*R*	ρ	*R*	ρ	*R*	ρ
IRP	.40	.53	.38	.51	.35	.54	.34	.51
q=1	.24	.62	.20	.63	.18	.59	.17	.54
q=5	.47	.60	.43	.57	.40	.57	.37	.53
q=10	.53	.58	.49	.56	.45	.55	.42	.53
q=100	.53	.60	.50	.59	.46	.56	.43	.53
q=3000	.54	.58	.50	.59	.46	.56	.43	.54

Table 4: Varying parameter q for FAST-PLEB with fixed $p = 1000$, $k = 3000$ and $b = 40$. Results reported on recall and ρ.

4.3 Evaluating Approximate Nearest Neighbor

Experimental Setup: To evaluate approximate nearest neighbor similarity lists generated by **FLAG**, we conduct three experiments. We evaluate all the three experiments on 447 words (test set) as used in Section 4.2. For each word, both exact and approximate methods return $l = 100$ nearest neighbors. The exact similarity lists for 447 test words is computed by calculating cosine similarity between 447 test words with respect to all other words. We also compare the LSH (computed using Hamming distance between all words and test set.) approximate nearest neighbor similarity lists against the exact similarity lists. LSH provides an upper bound on the performance of our approximate search representations (IRP, PLEB, and FAST-PLEB) for fast-search from Section 3.3) . We set the number of projections $k = 3000$ for all three methods and for PLEB and FAST-PLEB, we set number of permutations $p = 1000$ as used in large-scale noun clustering work (Ravichandran et al., 2005).

Evaluation Metric: We use two kinds of measures, recall and Pearson's correlation to measure the overlap in the approximate and exact similarity lists. Intuitively, recall (R) captures the number of

	IRP								**PLEB**								**FAST-PLEB**							
	10		25		50		100		10		25		50		100		10		25		50		100	
	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ
LSH	.55	.57	.52	.56	.49	.54	.46	.52	.55	.57	.52	.56	.49	.54	.46	.52	.55	.57	.52	.56	.49	.54	.46	.52
20	.29	.50	.26	.55	.25	.54	.24	.50	.50	.59	.45	.60	.41	.57	.37	.55	.48	.58	.42	.58	.38	.58	.35	.55
30	.36	.55	.33	.56	.31	.55	.30	.52	.53	.59	.48	.59	.44	.56	.41	.54	.51	.57	.47	.57	.42	.56	.40	.54
40	.40	.53	.38	.51	.35	.54	.34	.51	.54	.58	.50	.59	.46	.56	.43	.54	.53	.58	.49	.56	.45	.55	.42	.53
50	.44	.56	.42	.54	.39	.54	.37	.52	.54	.58	.51	.57	.47	.56	.44	.53	.54	.58	.50	.56	.46	.55	.44	.53
100	.53	.59	.49	.54	.46	.55	.43	.53	.55	.56	.52	.56	.48	.54	.46	.53	.55	.57	.52	.56	.48	.54	.46	.53

Table 3: Evaluation results on comparing LSH, IRP, PLEB, and FAST-PLEB with $k = 3000$ and $b = \{20, 30, 40, 50, 100\}$ with exact nearest neighbors over **GW** data set. For PLEB and FAST-PLEB, we set $p = 1000$ and for FAST-PLEB, we set $q = 10$. We report results on recall (R) and ρ metric. For IRP, we sample first p rows and only use p rows rather than k.

	GW								**GWB50**								**GWB100**							
	10		25		50		100		10		25		50		100		10		25		50		100	
	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ	R	ρ
LSH	.55	.57	.52	.56	.49	.54	.46	.52	.51	.55	.46	.54	.44	.52	.42	.48	.48	.58	.45	.52	.42	.49	.40	.47
IRP	.40	.53	.37	.53	.35	.54	.34	.51	.29	.50	.27	.51	.25	.51	.24	.47	.26	.57	.24	.49	.23	.48	.22	.45
PLEB	.54	.58	.50	.59	.46	.56	.43	.54	.46	.58	.42	.56	.38	.53	.36	.51	.44	.57	.40	.56	.36	.52	.33	.49
FAST-PLEB	.53	.58	.49	.56	.45	.55	.42	.53	.46	.56	.41	.56	.37	.54	.35	.51	.43	.57	.38	.55	.35	.52	.32	.50

Table 5: Evaluation results on comparing LSH, IRP, PLEB, and FAST-PLEB with $k = 3000$, $b = 40$, $p = 1000$ and $q = 10$ with exact nearest neighbors across three different data sets: **GW**, **GWB50**, and **GWB100**. We report results on recall (R) and ρ metric. The gray color row is the system that we use for further evaluations.

nearest neighbors that are found in both the lists and then Pearson's (ρ) correlation captures if the relative order of these lists is preserved in both the similarity lists. We also compute R and ρ at various $l = \{10, 25, 50, 100\}$.

Results: For the first experiment, we evaluate IRP, PLEB, and FAST-PLEB against the exact nearest neighbor similarity lists. For IRP, we sample first p rows and only use p rather than k, this ensures that all the three methods (IRP, PLEB, and FAST-PLEB) take the same query time. We vary the approximate nearest neighbor beam parameter $b = \{20, 30, 40, 50, 100\}$ that controls the number of closest neighbors for a word with respect to each independent random projection. Note, with increasing b, our algorithm approaches towards LSH (computing Hamming distance with respect to all the words). For FAST-PLEB, we set $q = 10$ ($q << k$) that is the number of random bits selected out of k to generate p permuted bit vectors of size q. The results are reported in Table 3, where the first row compares the LSH approach against the exact similarity list for test set words. Across three columns we compare IRP, PLEB, and FAST-PLEB. For all methods, increasing b means better recall. If we move down the table, with $b = 100$, IRP, PLEB, and FAST-PLEB get results comparable to LSH (reaches an upper bound). However, using large b implies generating a long potential nearest neighbor list close to the size of the unique context vectors. If we focus on the gray color row with $b = 40$ (This will have comparatively *small* potential list and return nearest neighbors in *less* time), IRP has worse recall with best pre-processing time. FAST-PLEB ($q = 10$) is comparable to PLEB (using all bits $q = 3000$) with pre-processing time 300 times faster than PLEB. For rest of this work, **FLAG** will use FAST-PLEB as it has best recall and pre-processing time with fixed $b = 40$.

For the second experiment, we vary parameter $q = \{1, 5, 10, 100, 3000\}$ for FAST-PLEB in Table 4. Table 4 demonstrates using $q = \{1, 5\}$ result in worse recall, however using $q = 5$ for FAST-PLEB is better than IRP. $q = 10$ has comparable recall to $q = \{100, 3000\}$. For rest of this work, we fix $q = 10$ as it has best recall and pre-processing time.

For the third experiment, we increase the size of the data set across the Table 5. With the increase in size of the data set, LSH, IRP, PLEB, and FAST-PLEB ($q = 10$) have worse recall. The reason for such a behavior is that the number of unique context vectors is greater for big data sets. Across all the

jazz	yale	soccer	physics	wednesday
reggae	harvard	basketball	chemistry	tuesday
rockabilly	cornell	hockey	mathematics	thursday
rock	fordham	lacrosse	biology	monday
bluegrass	rutgers	handball	biochemistry	friday
indie	dartmouth	badminton	science	saturday
baroque	nyu	softball	microbiology	sunday
ska	ucla	football	geophysics	yesterday
funk	princeton	tennis	economics	tues
banjo	stanford	wrestling	psychology	october
blues	loyola	rugby	neuroscience	week

Table 6: Sample Top 10 similarity lists returned by FAST-PLEB with $k = 3000$, $p = 1000$, $b = 40$ and $q = 10$ from **GWB100**.

three data sets, FAST-PLEB has recall comparable to PLEB with best pre-processing time. Hence, for the next evaluation to show the quality of final lists we use FAST-PLEB with $q = 10$ for **GWB100** data set.

In Table 6, we list the top 10 most similar words for some words found by our system **FLAG** using **GWB100** data set. Even though **FLAG**'s approximate nearest neighbor algorithm has less recall with respect to exact but still the quality of these nearest neighbor lists is excellent.

For the final experiment, we demonstrate the preprocessing and query time results comparing LSH, IRP, PLEB, and FAST-PLEB with $k = 3000$, $p = 1000$, $b = 40$ and $q = 10$ parameter settings. For pre-processing timing results, we perform all the experiments (averaged over 5 runs) on **GWB100** data set with $106,733$ words. The second pre-processing step of the system **FLAG** (Section 3.2) that is dimensionality reduction from $\mathbb{R}^D$ to $\mathbb{R}^k$ took 8.8 hours. The pre-processing time differences among IRP, PLEB, and FAST-PLEB from third step (Section 3.3) are shown in second column of Table 7. Experimental results show that the naive baseline IRP is the fastest and FAST-PLEB has 120 times *faster pre-processing time* compared to PLEB.

For comparing query time among several methods, we evaluate over 447 words (Section 4.2). We report average timing results (averaged over 10 runs and 447 words) to find top 100 nearest neighbors for single query word. The results are shown in third column of Table 7. Comparing first and second rows show that LSH is 87 times faster than computing exact top-100 (cosine similarity) nearest neighbors. Comparing second, third, fourth and fifth rows of the table demonstrate that IRP, PLEB and FAST-PLEB methods are twice as fast as LSH.

Methods	Preprocessing	Query (seconds)
Exact	n/a	87
LSH	8.8 hours	0.59
IRP	7.5 minutes	0.28
PLEB	1.8 days	0.28
FAST-PLEB	22 minutes	0.26

Table 7: Preprocessing and query time results comparing exact, LSH, IRP, PLEB, and FAST-PLEB methods on **GWB100** data set.

Language	english	chinese	japanese	spanish	russian
Place	africa	america	washington	london	pacific
Nationality	american	european	french	british	western
Date	january	may	december	october	june
Organization	ford	microsoft	sony	disneyland	google

Table 8: Query terms for Google Sets Problem evaluation

5 Applications

We use the graph constructed by **FLAG** from **GWB100** data set (110 GB) by applying FAST-PLEB with parameters $k = 3000$, $p = 1000$, $q = 10$ and $b = 40$. The graph has $106,733$ nodes (words), with each node having 100 edges that denote the top $l = 100$ approximate nearest neighbors associated with each node. However, **FLAG** applied FAST-PLEB (approximate search) to find these neighbors. Therefore many of these edges can be noisy for our applications. Hence for each node, we only consider top 10 edges. In general for graph-based NLP problems; for example, constructing web-derived polarity lexicons (Velikovich et al., 2010), top 25 edges were used, and for unsupervised part-of-speech tagging using label propagation (Das and Petrov, 2011), top 5 edges were used.

5.1 Google Sets Problem

Google Sets problem (Ghahramani and Heller, 2005) can be defined as: given a set of query words, return top t similar words with respect to query words. To evaluate the quality of our approximate large-scale graph, we return top 25 words which have best aggregated similarity scores with respect to query words. We take 5 classes and their query terms (McIntosh and Curran, 2008) shown in Table 8 and our goal is to learn 25 *new* words which are similar with these 5 query words.

Language: german, french, estonian, hungarian, bulgarian
Place: scandinavia, mongolia, mozambique, zambia, namibia
Nationality: german, hungarian, estonian, latvian, lithuanian
Date: september, february, august, july, november
Organization: looksmart, hotbot, lycos, webcrawler, alltheweb

Table 9: Learned terms for Google Sets Problem

Concrete seeds	car, house, tree, horse, animal man, table, bottle, woman, computer
Abstract seeds	idea, bravery, deceit, trust, dedication anger, humour, luck, inflation, honesty

Table 10: Example seeds for bootstrapping.

We conduct a manual evaluation to directly measure the quality of returned words. We recruited 1 annotator and developed annotation guidelines that instructed each recruiter to judge whether learned values are similar to query words or not. Overall the annotator found almost all the learned words to be similar to the query words. However, the algorithm can not differentiate between different senses of the word. For example, "French" can be a language and a nationality. Table 9 shows the top ranked words with respect to query words.

5.2 Learning Concrete and Abstract Words

Our goal is to automatically learn concrete and abstract words (Turney et al., 2011). We apply bootstrapping (Kozareva et al., 2008) on the word graphs by manually selecting 10 seeds for concrete and abstract words (see Table 10). We use in-degree (sum of weights of incoming edges) to compute the score for each node which has connections with known (seeds) or automatically labeled nodes, previously exploited to learn hyponymy relations from the web (Kozareva et al., 2008). We learn concrete and abstract words together (known as mutual exclusion principle in bootstrapping (Thelen and Riloff, 2002; McIntosh and Curran, 2008)), and each word is assigned to only one class. Moreover, after each iteration, we harmonically decrease the weight of the in-degree associated with instances learned in later iterations. We add 25 new instances at each iteration and ran 100 iterations of bootstrapping, yielding 2506 concrete nouns and 2498 abstract nouns. To evaluate our learned words, we searched in WordNet whether they had 'abstraction' or 'physical' as their hypernym. Out of 2506 learned concrete nouns, 1655 were found in WordNet. According to WordNet, **74**% of those are *concrete* and 26% are abstract. Out of 2498 learned abstract nouns, 942 were found in WordNet. According to WordNet, 5% of those are concrete and **95**% are *abstract*. Table 11 shows the top ranked concrete and abstract words.

Concrete: girl, person, bottles, wife, gentleman, microcomputer, neighbor, boy, foreigner, housewives, texan, granny, bartender, tables, policeman, chubby, mature, trees, mainframe, backbone, truck
Abstract: perseverance, tenacity, sincerity, professionalism, generosity, heroism, compassion, commitment, openness, resentment, treachery, deception, notion, jealousy, loathing, hurry, valour

Table 11: Learned concrete/abstract words.

6 Conclusion

We proposed a system, **FLAG** which constructs fast large-scale approximate graphs from large data sets. To build this system we proposed a distributed online-PMI algorithm that scaled up to 110 GB of web data with 866 million sentences in less than 2 days using 100 quad-core nodes. Our both intrinsic and extrinsic experiments demonstrated that online-PMI algorithm not at all loses globally good contexts and perform comparable to offline-PMI. Next, we proposed FAST-PLEB (a variant of PLEB) and empirically demonstrated that it has recall comparable to PLEB with 120 *times faster* pre-processing time. Finally, we show the applicability of **FLAG** on two applications: Google-Sets problem and learning concrete and abstract words.

In future, we will apply **FLAG** to construct graphs using several kinds of contexts like lexical, semantic, syntactic and dependency relations or a combination of them. Moreover, we will apply graph theoretic models on graphs constructed using **FLAG** for solving a large variety of NLP applications.

Acknowledgments

This work was partially supported by NSF Award IIS-1139909. Thanks to Graham Cormode and Suresh Venkatasubramanian for useful discussions and the anonymous reviewers for many helpful comments.

References

Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687.

Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. 2009. A study on similarity and relatedness using distributional and wordnet-based approaches. In *NAACL '09: Proceedings of HLT-NAACL*.

Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312:3–15, January.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *In Proc. of 34th STOC*, pages 380–388. ACM.

K. Church and P. Hanks. 1989. Word Association Norms, Mutual Information and Lexicography. In *Proceedings of ACL*, pages 76–83, Vancouver, Canada, June.

Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. In *VLDB*.

Graham Cormode and S. Muthukrishnan. 2004. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*.

Dipanjan Das and Slav Petrov. 2011. Unsupervised part-of-speech tagging with bilingual graph-based projections. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 600–609, Portland, Oregon, USA, June. Association for Computational Linguistics.

Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. *SIGCOMM Comput. Commun. Rev.*, 32(4).

L. Finkelstein, E. Gabrilovich, Y. Matias, E. Rivlin, Z. Solan, G. Wolfman, and E. Ruppin. 2002. Placing search in context: The concept revisited. In *ACM Transactions on Information Systems*.

Zoubin Ghahramani and Katherine A. Heller. 2005. Bayesian Sets. In *in Advances in Neural Information Processing Systems*, volume 18.

Amit Goyal and Hal Daumé III. 2011. Approximate scalable bounded space sketch for large data NLP. In *Empirical Methods in Natural Language Processing (EMNLP)*.

Amit Goyal, Hal Daumé III, and Suresh Venkatasubramanian. 2009. Streaming for large scale NLP: Language modeling. In *NAACL*.

Amit Goyal, Graham Cormode, and Hal Daumé III. 2012. Sketch algorithms for estimating point queries in NLP. In *Empirical Methods in Natural Language Processing (EMNLP)*.

D. Graff. 2003. English Gigaword. Linguistic Data Consortium, Philadelphia, PA, January.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613. ACM.

Zornitsa Kozareva, Ellen Riloff, and Eduard Hovy. 2008. Semantic class learning from the web with hyponym pattern linkage graphs. In *Proceedings of ACL-08: HLT*, pages 1048–1056, Columbus, Ohio, June. Association for Computational Linguistics.

Abby Levenberg, Chris Callison-Burch, and Miles Osborne. 2010. Stream-based translation models for statistical machine translation. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 394–402. Association for Computational Linguistics.

Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 287–296. ACM.

Ping Li, Kenneth Ward Church, and Trevor Hastie. 2008. One sketch for all: Theory and application of conditional random sampling. In *Neural Information Processing Systems*, pages 953–960.

Tara McIntosh and James R Curran. 2008. Weighted mutual exclusion bootstrapping for domain independent lexicon and template acquisition. In *Proceedings of the Australasian Language Technology Association Workshop 2008*, pages 97–105, December.

G.A. Miller and W.G. Charles. 1991. Contextual correlates of semantic similarity. *Language and Cognitive Processes*, 6(1):1–28.

Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. 2009. Web-scale distributional similarity and entity set expansion. In *Proceedings of EMNLP*.

Delip Rao and Deepak Ravichandran. 2009. Semi-supervised polarity lexicon induction. In *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*, pages 675–682, Athens, Greece, March. Association for Computational Linguistics.

Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and nlp: using locality sensitive hash function for high speed noun clustering. In *Proceedings of ACL*.

H. Rubenstein and J.B. Goodenough. 1965. Contextual correlates of synonymy. *Computational Linguistics*, 8:627–633.

Florin Rusu and Alin Dobra. 2007. Statistical analysis of sketch estimators. In *SIGMOD '07*. ACM.

M. Thelen and E. Riloff. 2002. A Bootstrapping Method for Learning Semantic Lexicons Using Extraction Pattern Contexts. In *Proceedings of the Empirical Methods in Natural Language Processing*, pages 214–221.

Peter D. Turney and Patrick Pantel. 2010. From frequency to meaning: Vector space models of semantics. *JOURNAL OF ARTIFICIAL INTELLIGENCE RESEARCH*, 37:141.

Peter Turney, Yair Neuman, Dan Assaf, and Yohai Cohen. 2011. Literal and metaphorical sense identification through concrete and abstract context. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 680–690. Association for Computational Linguistics.

Benjamin Van Durme and Ashwin Lall. 2009a. Probabilistic counting with randomized storage. In *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*.

Benjamin Van Durme and Ashwin Lall. 2009b. Streaming pointwise mutual information. In *Advances in Neural Information Processing Systems 22*.

Benjamin Van Durme and Ashwin Lall. 2010. Online generation of locality sensitive hash signatures. In *Proceedings of the ACL 2010 Conference Short Papers*, pages 231–235, July.

Benjamin Van Durme and Ashwin Lall. 2011. Efficient online locality sensitive hashing via reservoir counting. In *Proceedings of the ACL 2011 Conference Short Papers*, June.

Leonid Velikovich, Sasha Blair-Goldensohn, Kerry Hannan, and Ryan McDonald. 2010. The viability of web-derived polarity lexicons. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 777–785, Los Angeles, California, June. Association for Computational Linguistics.

Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120. ACM.