ABSTRACT

Title of proposal: SELECTIVE SEARCH ARCHITECTURES AND BRUTE FORCE SCAN TECHNIQUES FOR SUMMARIZING SOCIAL MEDIA POSTS

Yulu Wang, Doctor of Philosophy, 2018

Proposal directed by: Professor Jimmy Lin
Department of Computer Science and
College of Information Studies
University of Maryland

There is an increasing trend of social media usage in recent years and users desire a search system that can provide relevant content related to their information needs. However, the nature of social posts presents challenges for existing search functions. Firstly, as a stream of documents is constantly arriving at a high speed, it is required that the search system provides a real-time service that can guarantee new content be available for search immediately after it arrives. Secondly, given that the data volume contains substantive redundant information, the search engine should be able to detect redundant content and eliminate it before returning a summary of relevant documents to users. In reality, it is unlikely that users desire a list of documents containing redundant information.

Previous work that supports real-time search uses an inverted index as the core of the search architecture. However, building/updating indexes in real-time is complex. Additionally, an inverted index is a non-regular data structure, and

query evaluation results in data access patterns that are at odds with modern processor architectures. Where the summarization of social media posts is concerned, although plenty of work has been done in this field, there lacks a validated evaluation methodology to measure such systems' effectiveness.

To achieve real-time search for relevant documents, in this work, we first exploit *selective search* architecture. Selective search divides collections based on document similarity into multiple partitions (shards), builds inverted index on each of them, and selects the partitions that are estimated to contain relevant documents for retrieval. For real-time service, our selective search architecture divides document stream into temporal segments at different granularities and performs either batch or online clustering algorithms on them. We also take advantage of word embeddings to reduce the dimensionality of document vectors for clustering to reduce computational cost. Results show that by applying this selective search technique, we are able to achieve a level of precision statistically indistinguishable from an exhaustive search while providing substantially higher search efficiency.

For query evaluation on the unpartitioned documents, we abandon inverted index and present a simple and fast indexing process based on *brute force scans* technique that is to search over array representations of the documents. A final result list is then generated by merging the relevant documents with those from selected partitions. We show that our brute force scan techniques outperform the traditional search engines built on inverted indexes in terms of both query latency and throughput by exploiting parallelism when the collection size is under a few million.

To summarize relevant documents, we first develop an evaluation framework, then demonstrate that the evaluation metrics proposed in this framework are capable of stably capturing user preferences in a social posts summarization system. With the evaluation metrics as a guideline, we build a system that is based on convolutional neural networks to compute document similarity and exploits single-pass clustering to perform the summarization task.

Selective search architecture, brute force scan techniques, evaluation framework and summarization implementation comprise our real-time search system, that can retrieve relevant documents and summarize the results in response to a query. Although our system is designed to specially focus on Twitter data, we believe the techniques exploited could generalize to real-time summarization for other social media posts.

SELECTIVE SEARCH ARCHITECTURES AND BRUTE FORCE SCAN TECHNIQUES FOR SUMMARIZING SOCIAL MEDIA POSTS

by

Yulu Wang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Professor Jimmy Lin, Chair/Advisor
Professor Aravind Srinivasan, Dean's Representative
Associate Professor Hal Daumé III
Assistant Professor John Dickerson
Professor Douglas W. Oard

© Copyright by
Yulu Wang
2018

Dedication

To Pan, my parents and my sister.

Acknowledgments

It's finally the end, or in Chinese, it's time to draw a period to this doctoral program. As much as I struggled throughout the years, I'm grateful to those who have made this experience a cherished and unforgettable one.

Specials thanks to my best advisor ever, Jimmy. I have learnt a lot from him both personally and academically. As an advisor, Jimmy is certainly well knowledgeable. Aside from that, he has always been understanding, helpful and never giving up on me. He always thinks one step ahead for me, and is ready to help whenever I needed him, even after he moved to Waterloo, where he has his own department, lab and students to worry about. In my eyes, this kind of support is far more important than any research guidance that, without him, I wouldn't be able to survive my PhD all these years. Words cannot express my gratitude I owe him, and I just wish God brought all that he wants and deserves.

I would like to thank my boyfriend, Pan. All these years, he proved that accompany is the longest confession of love. He has enlightened my life during those dark days with his humor and passion, and a view to a whole different world. I learnt from him to be confident, independent and encounter the real me. So without him, I wouldn't be growing up stronger and meeting the better me. Like him, I rather believe in showing love in actions than in words, hence, I'll always be there when he needs me the most.

The people I owe the gratitude most are my family, who have sacrificed a lot for me, my parents, and my sister Yufei. If it were not for this PhD, I would have

been back in home and sharing happiness and sorrow together with them. Yet they never complained, and always show love and support. I know that even if I'm no one, or I have nothing left with me, they always have my back and there's always home. The more understanding they show, the more I feel I own them. There is hardly anything I can do over here, but may health and happiness be with my family for now and forever.

Thanks are also given to all the staff in the Computer Science department, including Jenny, Jodie, Sharron, etc. who have made the department like home.

I would also like to thank Aravind, Doug, Hal and John for sparing their time to review my thesis and serving on the dissertation committee.

Thank all my friends for their friendship and all the happiness they brought to me.

Finally, thank all the people not listed here who have entered my life and made it colorful.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

With the recent prevalence of personal laptops and smart phones, social media has been used frequently by society. Billions of people use social media for many reasons including sharing their interests and experiences, following their friends, and following recent news and events.

Consequently, there is an increasing volume of data available every second; such data arrives in a streaming fashion at a high speed. Meanwhile, as social media enables users to not only share their original posts but also to share posts from other users, it adds to the data volume by contributing redundant information. Using Twitter as an example, users are able to retweet a tweet from someone they follow. While they may add additional comments, the original tweet is still republished and recorded multiple times.

Just as web users use a web search to find the information they need, social media users have specific requirements that must be fulfilled in order to search for data that is relevant to their need. This is expressed as a query in the search box. This happens when, for instance, breaking news happens and a user wants to find out more information or when a user wants to search for an old post by a friend or acquaintance. It is therefore necessary to have a social posts search engine, which

has become an important application in Information Retrieval (IR), because the vast quantities of data collected make it difficult if not impossible for a human to track manually.

Given the nature of social posts as discussed above, a search engine must address two challenges:

- The search engine should be able to deal with a dynamic collection and provide real-time search service to guarantee that new content is available for search immediately after it arrives. This is necessary because new content is generated and added to the old content at a high speed, and as much of the content is time sensitive – particularly news and world events – users want to be provided with lists of relevant posts once they are created.

- The search engine should be able to detect redundant content and eliminate it before returning a summary of relevant documents to the user, as it is unlikely that users want to consume a list of documents that contains redundant information.

Although researchers have thoroughly explored traditional search techniques during the past few decades and these techniques have been found to work well in response to user needs, these searches have largely been performed on static collections. It has not been until recent years that researchers have shifted their attention to searching of social media posts and have developed search systems that address dynamic collections. However, these systems have performance bottlenecks, which we will discuss later in greater detail. Researchers also do not pay enough

attention to summarizing social posts, nor have they established standard guidelines to evaluate such a system.

To address these shortcomings, in this work we chose Twitter as the social media platform for our research and propose a new search architecture that can index tweets in real-time. More specifically, we exploited *selective search* and *brute force scans* techniques to achieve real-time search for relevant tweets with a high level efficiency without degrading effectiveness. Then, to evaluate the effectiveness of the system in summarizing tweets, we developed a evaluation framework. In this framework, we introduced several evaluation metrics, which we then demonstrated are capable of stably capturing user preferences in a tweet summarization system. The evaluation framework sets up a guideline that helps to improve tweet summarization systems. Under these guidelines, we exploited deep learning techniques to detect and eliminate redundant relevant tweets to implement a summarizing system.

1.1 Overview

1.1.1 Selective Search

The key to building a real-time search system for large Twitter collection involves both how to store the collection (whether in memory or disk, whether distributed or in a single machine) and what form of data structure the collection should be stored as so the retrieval time can be optimized. We will discuss the format in a later section. Whereas in terms of how to store the collection, traditionally, the search solution for large collections is to build search systems where the collection is

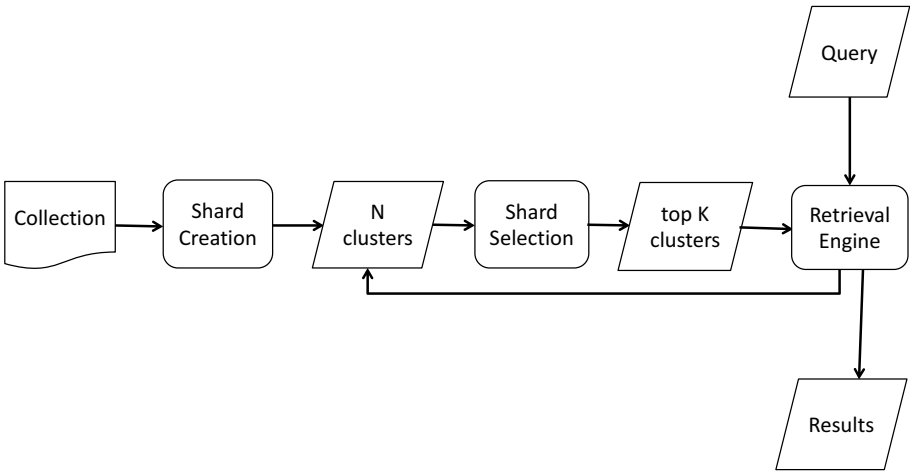

Figure 1.1: Selective search architecture.

divided into smaller subsets (shards) randomly by hashing, with each being put in a server in distributed search architecture or put in a core in a single machine. For query evaluation, a broker assigns queries to the partition servers and gather results retrieved from all the servers. This is the so-called exhaustive search as all the shards are evaluated in response to each query. As an alternative to exhaustive search, *selective search* is often used to reduce search costs when computational resources are limited. It divides collections based on document similarity into multiple shards, and selects only a few shards that are estimated to contain relevant documents for the query to search. To build a fast real-time search system, we apply selective search in our system to improve search efficiency, with the goal of not degrading effectiveness significantly.

Figure 1.1 shows a typical selective search architecture. It involves two steps:

- **Shard Creation**: This step partitions the collection into a set of shards following a document allocation policy. Document allocation policies include *random*, *source-based* and *topic-based* (e.g., K-means) document allocation [2] (more details will follow). Topic-based document allocation has been shown to be the most effective among the three methods of document allocation [3].

- **Shard Selection**: For every query, this step identifies a small number (K) of shards from the total N shards that are relevant to the query, and outputs top K shards to the "retrieval engine" for retrieval. The "retrieval engine" is discussed in detail in the next section, and it's basically responsible for retrieving relevant documents in response to a query. These retrieved documents are then merged together to generate a final list of results.

Researchers have studied selective search techniques in recent years [2–5] in order to reduce computation costs while maintaining a high level of effectiveness. However, previous work has the following limitations:

- Shard creation based on batch k-means clustering cannot support large datasets [4]. This is due to the document representation format, where each document is represented as a sparse vector in the vocabulary space. Such vectors represent a huge dimension, so computing distance between vectors and centroid of these vectors for k-means leads to extremely high computing cost and resources requirements. Although there has been a recent study where applied Sample-Based k-means to address this limitation [2], it runs the risk of not being able

to capture important information contained in non-selected documents.

- It is not able to handle a dynamic collection. Shard creation is a one-time, expensive, offline process that works well for a static collection. But in real-time search with a dynamic collection, new documents must be assigned to the closest shards soon after they become available. If we remain with the traditional shard creation technique – that is, to rebuild the whole new collection for every newly encountered document – the process would become impractically slow.

To address the first challenge above, in Chapter 4, we explore the feasibility of using word embeddings. More specifically, we explore Global Vectors for Word Representation (GloVe) [6] to generate document vectors. These document vectors are dense vectors in small dimensions, of usually not more than a hundred. Therefore, it dramatically reduces the computing cost for clustering.

To be able to handle dynamic collections as in our case, differing from the traditional selective search approach that takes the collection as a whole and performs shard creation, our selective search approach creates shards incrementally. To explain this in detail, in our approach, the document stream is divided into temporal segments, and shard creation is performed within each segment, using either batch or online clustering algorithms to cluster the shards at different granularities. At query time, only a subset of the shards in each temporal segment are examined for relevance based on some selection criteria which will be discussed in Chapter 4. Therefore, for each segment, we only consider a subset of the documents that

are estimated to contain relevant documents for query evaluation and thus reduce computational cost.

Experiments have shown that by applying this selective search technique we are able to achieve a level of precision statistically indistinguishable from an exhaustive search while providing substantially higher search efficiency. We also observe that there is no significant effectiveness difference between batch vs. online clustering algorithms, and between hourly vs. daily temporal segments. All the experiments and results have been published in WSDM 2017 [7].

1.1.2 Search Architecture

In terms of what form of the data structure should be used, when building search systems for static collection, the typical architecture used is shown in Figure 1.2. In such a system, the first step is to build an inverted index for the collection. An inverted index is a data structure that stores mapping from words to lists of documents called postings lists that contain the word. This inverted index is constructed and maintained by an indexer. When retrieving relevant documents for a query, the retrieval engine goes to the inverted index, performs a lookup operation to locate all the documents containing the query terms, computes the query-document score based on a scoring model, and returns the top k results. Aside from simple lookup, the retrieval engine usually executes a learning-to-rank pipeline to improve the result ranking.

The approach works well for a static collection search because building an

index is a one-time operation; once the index is built, it can be used to retrieve relevant documents. Similarly, inverted index also works well for the complete temporal segments because no new documents will be added to any already created complete segment therefore they can be considered as static. We then apply this search architecture in our complete temporal segments discussed above, and build an inverted index for each shard created within each temporal segment. In response to a query, top K shards out of total of N shards within each segment that are estimated to contain the most relevant documents are selected, and the retrieval engine goes to the corresponding inverted indexes, retrievals all the relevant documents, merges them and returns the top k results. This is shown in part 1 of Figure 4.1.

However, in a real-time selective search system, there is still this "leftover" piece of data in the last incomplete temporal segment that has not been partitioned into shards, where a stream of documents is constantly flowing into at a high rate of speed. For these documents, it is necessary to perform indexing when they are ingested, to make these documents immediately searchable. In a search system that achieves this, an indexer constantly indexes the newly encountered documents and updates the new index with the old index, performing an incremental indexing.

Incremental indexing algorithms have been studied extensively in the past. Early systems assume that the inverted index does not fit into the system memory so it resides on disk. Slow disk searches lead to high query latency. Recent systems that provide real-time updating within the system's memory like EarlyBird [8] have complex update processes that involve compression. While we want to explore a completely novel approach different from inverted index, in Chapter 3 we build a

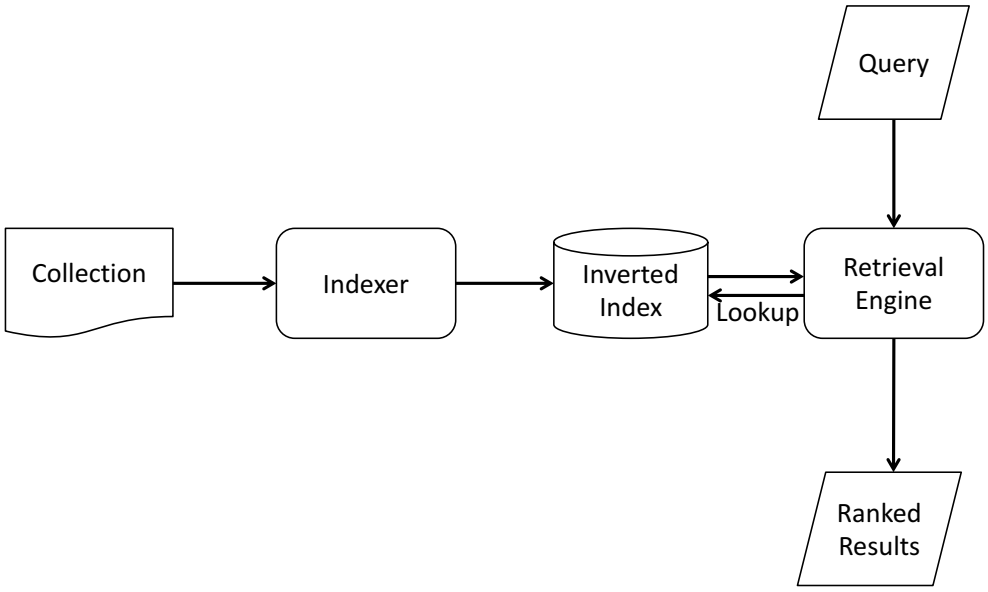

Figure 1.2: Search systems for static collection.

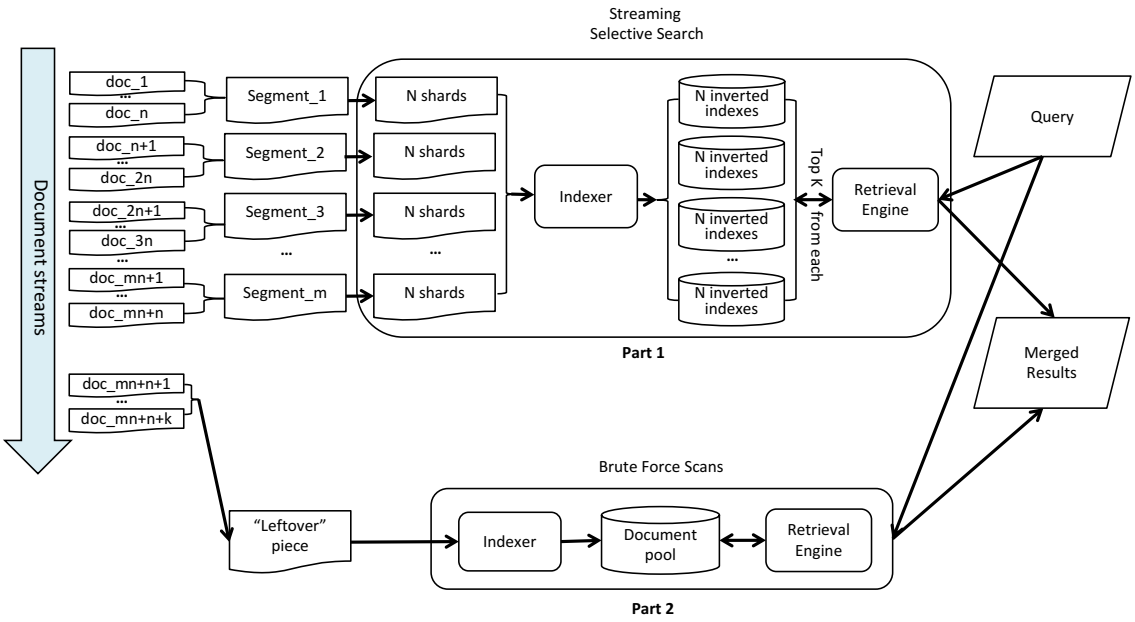

Figure 1.3: Proposed search system for streaming documents. Part 1 for temporal segments, and part 2 for "leftover" pieces.

retrieval architecture based on brute force scans of document representations. (See part 2 in Figure 4.1 for a search architecture based on brute force scans). This differs from the typical real-time search service in two ways:

- "Index" becomes a document pool that stores document representations. Each document is represented as an array of integers, and whenever a new document (tweet) comes, the "indexer" appends the document vector to the existing in-memory document pool. So the "indexing" is done in an incremental fashion. Note that this document vector representation is different from that used in selective search.

- For retrieval, we process each query in the same way as a tweet and perform a brute force scan over the document pool to compute the query-document scores. Thus, query evaluation can take advantage of modern processor architectures to efficiently process batches of contiguous memory as opposed to random data access as with an inverted index.

Experiments with tweets data have shown that "indexing" with brute force scan techniques is 13 times faster than that with inverted index.

Based on the idea, we also improved the retrieval performance by exploiting SIMD (single instruction, multiple data) instructions and modern multi-core processors.

Brute force scan techniques are quite efficient and promising in a real-time tweet search. Preliminary results from the published 2015 ICTIR paper [9] have shown that we are just a few steps away from being comparable to or even beating the

traditional search engines built on inverted indexes in terms of query latency. And after optimizations, we show that our brute force scan techniques outperform the traditional search engines built on inverted indexes in terms of both query latency and throughput by exploiting parallelism when the collection size is under a few million, and it is a better option than inverted index for the "leftover" piece in our selective search system where the data is no more than a million.

Up until this point, we have introduced the two major parts of our real-time tweet search system that aims to return a list of relevant tweets efficiently without degrading effectiveness.

1.1.3 Tweet Summarization and Evaluation

For any user query, the goal of a search system is to return as many relevant documents as possible to the user; relevance is what the user wants. However, in reality, as a system that serves user needs, we should consider more than simply relevance. This is especially true for tweets, since there are often duplicate and near-duplicate tweets saying almost the same thing. In such cases, users will be overwhelmed by these duplicate and near-duplicate tweets, even if they are relevant, and it would be desirable if the system could summarize the relevant tweets and return to the user a list of results, each of which covers a different aspect of the topic. For example, when the Boston Marathon bombings occurred, multiple media authorities were tweeting about the event at almost the same time. Additionally, users were retweeting the original tweets – occasionally with some additional com-

mentary. A user is more likely to want to see how an incident develops, including how many people were injured, how they were treated, any suspects, etc., instead of multiple resources discussing the fact of the Boston bombing. However, there has not been a great deal of attention drawn to this field, and there is a lack of empirical evaluation to measure the system's effectiveness.

To address these issues, in TREC 2014, as one of the TREC Microblog track organizers I helped introduce a task called the Tweet Timeline Generation (TTG) task, where the goal of the system is to return a list of non-redundant, chronologically sorted relevant tweets. Since TTG and tweet summarization reference the same concept, we will use TTG and tweet summarization interchangeably.

I also developed and validated an evaluation methodology for TTG to evaluate the effectiveness of the submitted system (details in Chapter 5). A typical evaluation process works as follows: first, gather human judgments regarding the "ground truth" of the collection for the information needed, then develop one or more evaluation metrics. Humans evaluating system output are called assessors and can be experts in certain fields or online Mechanical Turkers. The evaluation metric can be used as a function that takes the output of the system and the "ground truth" and returns a score reflecting how effective the system is in satisfying the needs of the user.

Furthermore, Garrick Sherman (a graduate student from UIUC) and I verified that the evaluation metrics developed are capable of stably capturing user preferences. More specifically, we answered two questions asked in information retrieval evaluation: 1) Is the evaluation stable with respect to different assessors? In other

words, if system A scores higher than system B based on one assessor, is this still the case based on another assessor? This is necessary because we want an evaluation that provides a solid guideline for us to develop the system to strive for a high metric score. 2) Is the difference meaningful to a user in the sense that a system with a higher metric score is preferable to the user than a system with a lower metric score. Chapter 6 shows that the answers to both questions are yes.

Both the development and the validation of the evaluation methodology was published in the 2015 SIGIR paper [10]. In this paper, I co-developed the evaluation framework and verified evaluation stability while Garrick did most of the work to validate the correlation between evaluation metrics and user preferences.

1.1.4 Tweet Summarization System

With respect to summarization system, different approaches have been proposed by TREC 2014 Microblog track participants to accomplish the task[1]. These approaches can be categorized into clustering algorithms and ranking techniques [11–20]. Out of all the approaches, the one proposed by Lv et al. [15] using hierarchical clustering is proven to achieve the highest effectiveness score.

In our work, we built a system that exploited convolutional neural networks to compute tweet similarity [21] and applied single-pass clustering on the chronologically sorted tweets to summarize the relevant tweets, which we call the CNNTTG system. The system works like this: for each new tweet, we compare it with all the tweets in the existing clusters, and put it in a new cluster if the largest similarity

[1] `http://trec.nist.gov/pubs/trec23/trec2014.html`

with each tweet is smaller than the threshold, otherwise remove it. This resembles Cui et al. [12] work where they also uses single-pass clustering. However, our work differs from theirs in that while they use the cosine similarity of two tweets projected onto the vocabulary space with the use of word2vec [22] as the similarity score, our approach obtains the tweet similarity score from a neural networks model that exceeds the state of art sentence similarity models. By applying our approach on the TREC runs to remove redundant tweet, we showed that our approach is able to achieve higher metric scores by removing redundant tweets while keeping novel tweets compared to TREC runs.

1.2 Contributions and Outline

Our contributions for this work can be summarized as follows:

- **Streaming Selective Search**: We developed a selective search approach on document streams that is novel in partitioning and utilizing temporal organization strategies. Moreover, we took advantage of word embeddings to build our document vectors with low dimensions, thus making the clustering process faster and requiring less memory. Experiments have shown that by applying this selective search technique, we are able to achieve precision that is indistinguishable from an exhaustive search while providing substantially higher efficiency.

- **Brute Force Scans**: We did away with the traditional architecture built on an inverted index, and explored a completely novel approach, brute force

scans techniques. This approach is suitable for real-time search as the indexing is easier to update incrementally. By taking advantage of modern processor architecture, we achieve a performance that is better than the traditional search engines built on inverted indexes in terms of both query latency and throughput by exploiting parallelism when the collection size is under a few million.

- **TTG Evaluation**: We developed the evaluation framework for TTG task. We also showed that differences with assessors do not affect the evaluation of the TTG task, and that user preferences correlate to the metrics we proposed. Together, these components create a guideline for developing TTG systems.

- **CNNTTG Summarization System**: We built a TTG system that is based on convolutional neural networks to compute tweet similarity and by applying a single-pass clustering algorithm on the TREC runs, we showed that our approach is able to achieve higher metric scores by removing redundant tweets while keeping novel tweets compared to TREC runs.

With all that being introduced, to develop a complete tweet search summarizing system, we propose a pipeline shown in Figure 1.4. Our system is comprised of four parts: Streaming Selective Search (part 1), Brute Force Scans (part 2), Tweet Summarization (part 3) and TTG evaluation (part 4). When a stream of tweets arrives, they are first divided into temporal segments, part 1 then clusters each into N shards of tweets, and builds inverted index for each shard. Meanwhile, part 2 "indexed" the "leftover" tweets as arrays of integers to the end of the document

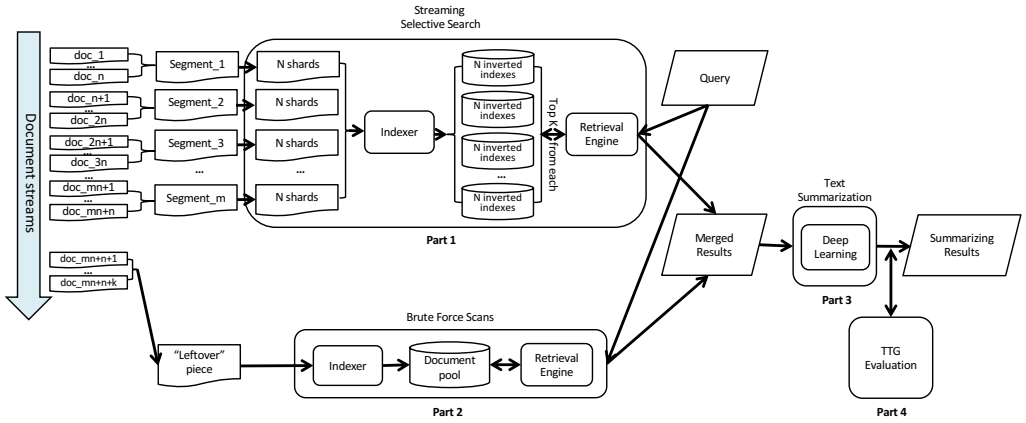

Figure 1.4: System pipeline: system composes of four parts, Streaming Selective Search (part 1), Brute Force Scans (part 2), Tweet Summarization (part 3) and TTG Evaluation (part 4).

pool. Whenever a user inputs a query, part 1 selects the top K shards from each segment that are closest to the query and runs a lookup in each corresponding inverted index to return top k results for each of them. In the mean time, part 2 runs a brute force scan over the document pool to get the top k results. All the results are then merged into a list of k results. Part 3 takes these as input and generates a summary of them. Finally, we keep improving the effectiveness of the TTG system with the help of part 4 until the metric score is satisfactory.

An alternate pipeline to what is proposed is to de-duplicate tweets at indexing time. However, we argue that this slows indexing down. This is because de-

duplication typically has quadratic time complexity, and it dramatically increases indexing time where the tweets are in the order of magnitude of millions, compared to if it were performed in the last stage where the returned relevant tweets are in the order of only a few hundreds. This additional time cost doesn't get compensated by the time saved on brute force scan. In addition, redundant content might be a good indicator of document relevance, thus keeping them till retrieval time helps to identify more relevant tweets. Both reasons motivate us to perform summarization as the very last stage in the pipeline.

In the following chapters, we discuss each of the contributions in detail. We first present brute force scans techniques in Chapter 3, and the reason is because we would like to study the cross-over point for the collection size as to whether brute force scan techniques or inverted index should be chosen as the search architecture for the "leftover" piece in our real-time selective search system. We then present streaming selective search in Chapter 4, the development of TTG evaluation methodology in Chapter 5, the validation of the TTG evaluation methodology in Chapter 6, our TTG system in Chapter 7. Finally we conclude in Chapter 8.

Chapter 2: Related Work

This chapter presents a literature review of all the components used in our search system. We first begin by reviewing the traditional approaches by which real-time search is handled. We then discuss the feasibility and advantages of brute force scans techniques for real-time search given modern processor architectures. Following is a review of the selective search techniques that have been used in IR research and techniques we have adapted and exploited in real-time scenarios to improve efficiency. Finally, we present a background of the Tweet Timeline Generation task, the evaluations for similar tasks, and methods applied by researchers to accomplish this task.

2.1 Real-Time Search Architecture

2.1.1 Inverted Index

As introduced in the previous chapter, search engines often have an inverted index as the core of the architecture. An inverted index is essentially a hashmap like data structure, where the key is each word occurring in the collection and its corresponding value is a list of documents that have at least one occurrence of the

word. Such list is called a postings list, and each document in this list is called a posting. A posting usually contains information like docID which is an integer, therefore the postings list can be sorted by docID for easy retrieval. An index might contain additional information like the *term frequency* (tf), which are the number of occurrences of the word in the document, and positions in which the word occurs in the document. Another frequently used term is *document frequency* (df), which is the total number of documents containing a certain word, or in other words, the length of the postings list for this word.

2.1.2 Query Evaluation

An information need in search can be expressed as a query containing a few words. For any given query, the process of deciding each document's relevance to the query and returning the relevant documents to the query is query evaluation.

Two types of queries are often used: *Boolean queries* and *ranked queries*. In Boolean queries, all the documents that contain the query terms will be returned as the results to the user. Hence, it's a word to word match. However, this soon can become troublesome for the users as the collection grows and the results can become too long to process manually. Therefore, for ranked queries, we would like to not only retrieval the relevant documents but also give each document a score based on a scoring model, rank them according to the relevance to the query and return top K results to users for examination. In our real-time search system, we are interested in ranked query processing.

Previously, several ranking/scoring models have been proposed by researchers. Vector space model is a model that converts each document into a vector in the vocabulary space. Salton et al. [23] formally explained vector space model. In this model, each component of the vector is a function value of tf and df scores. For retrieval, each query is treated as a bag of words and converted to a unit vector in the vocabulary space. Then, a similarity formulation is used to compute per document-query score. Salton [24] first considers cosine measure to compute similarity between documents and queries. Other early work like Ivie [25] also considered similarity functions.

An alternative to the vector space model has been probabilistic models [26–31], which has been shown by Hiemstra [32] to outperform vector space model on the Cranfield test collection. Probabilistic model is this model in which, given the document and query, the probability of a document being relevant to a certain query is computed. Maron and Kuhns [33] explored Probability Ranking Principle, which is the basis probabilistic model. Yu and Salton [34] first introduced Binary Indepedence Model, a model that has traditionally been used with Probability Ranking Principle. On a different thread, Robertson et al. [35] introduced the Okapi measure or BM25. Jones et al. [36] combines an amount of the probabilistic models with experiments on TREC data and the experiments demonstrate the model's effectiveness and robustness for a large test collection.

More recent probabilistic approaches are based on language models [37–39]. A typical language model that is widely used is the query likelihood model [40]. Unlike previous probabilistic models that captures the probability of a document

being relevant to a query, the query likelihood model is a language model widely used in information retrieval, wherein the approach is to estimate a language model for each document, and then rank documents by the likelihood of the query according to the estimated language model. To adjust the maximum likelihood estimator to compensate for data sparseness, different smoothing techniques like bayesian smoothing using Dirichlet priors proposed by Zhai and Laffery [41] can be used.

Because the focus of our work is not on developing a better ranking algorithm to improve search results but rather on search efficiency in a real-time scenario, we will use query likelihood model in all our experiments throughout our work and refer readers to the above cited papers if interested.

2.1.3 Index Construction

The process to construct an inverted index is called *index construction* or *indexing*; the process or machine that performs it is the *indexer* [40]. Early research studied index construction assuming they reside on disk due to the limitation of memory. Harman and Candela [42] proposed an approach to accumulate postings on disk, and postings lists are stored as linked lists. This approach does not need to maintain the collection vocabulary in main memory because postings are accumulated on disk. However, retrieving postings lists for query evaluation can be expensive as there are many random disk accesses, hence is not practical in a real setting. Other approaches like sorted-based approach [43] and two-pass approach [44] operate postings in a limited amount of memory and accumulate them in

a compressed format to disk. Heinz and Zobel [45] proposed an efficient single-pass index construction that does not require the entire vocabulary of the collection to be stored in memory. Compared with [42], these approaches dramatically reduced the time to construct an index by hundreds of times [45].

Retrieval from an on-disk index can be slow due to the slow disk access. However, with the growth of the memory capacity in recent years, an index stored in memory becomes possible. To give a few applications of in-memory index, Luk and Lam [46] built small inverted indexes in memory where they use a linked list data structure to represent postings. Busch et al. [8] recently introduced their Twitter real-time search system Earlybird where an in-memory index is constructed and maintained. Another application was implemented by Asadi et al. [47] whose work deals with arbitrary length of documents rather than tweets with limited length. And the utilization of compression of the postings lists makes in-memory index possible in these apporaches. And a good study of various compression methods can be found in a book written by Moffat and Zobel [48].

When the volumes of data gets too large for one single machine to handle, another approach widely used to speed up retrieval is through distributed search systems, where terms or documents can be partitioned into different nodes in the cluster or different cores in a machine. In this case, parallelism with multiple threads can be exploited to achieve lower latency. There are two different ways of splitting the collection across multiple machines, either by terms or by documents. In a search system where the index is term-partitioned [49], vocabulary is partitioned into each machine with corresponding postings lists stored in the same machine.

And to handle a query, the system only needs to process the machines that have the corresponding query terms' postings lists. This requires fewer machine seeks but the coordinating machine can become a bottleneck in performance. In a search system where the index is document-partitioned [50, 51], collection is partitioned and each subcollection is assigned to a different machine. Thus, an index needs to be built for each partition. And to handle a query, all the indexes need to be examined for relevance.

There have been extensive studies during the past to study the pros and cons with term-partitioned and document-partitioned index [52–55]. Most studies show that document distribution is likely to be superior in practice.

2.1.4 Incremental Indexing

Real-time search engines that use an inverted index require that the index be updated with one document at a time so that the newly encountered documents are immediately available for search. To solve this problem, researchers have long been exploring different index update algorithms.

In the early days, memory was limited and indexes did not fit into memory, and resided on a disk. Therefore, an update process should address the tradeoff between update speed and query evaluation speed. For example, if we continue to keep the index on disk as arrays, query evaluation can be performed quickly because one disk seek is enough to load the postings list for a query term. However, an index update can become complex and slow because it involves space reallocation. On the

other hand, if we use a discontiguous data structure-linked list to represent each postings list, an index update is then simply appended to the new postings at the end of the current postings list. However, we need to run several disk seeks before we get the whole postings list, which leads to slow query evaluation.

Under this constraint, several index update algorithms have been proposed; they fall into three types: *Re-build, Re-merge, In-place*. *Re-build* [42–45, 56–58] involves simply rebuilding the entire index. In this process, old index is kept for retrieval until it has been replaced by the new index. Re-building is simple yet expensive, so it is only used when the update is needed rarely. *Re-Merge* [59–64] first stores new documents in memory in the format of postings (in-memory postings), until the memory usage reaches a limit. It then reads the postings list from the disk, merges it with in-memory postings, and finally writes the new postings list onto the disk. Reading postings lists one at a time requires multiple interactions with the disk, leading to high latency. *In-place* [65–68], like Re-merge, also keeps track of in-memory postings, but whenever an update is triggered, it appends the in-memory postings to the end of the postings list on the disk. Thus, it requires re-allocating space whenever the on-disk postings cannot hold new postings. Because of this, it requires random disk seeks, and can be even slower than Re-merge with sequential disk access.

In summary, the works listed above assume that the inverted index does not fit into memory and reside on disks. Disk access is slow and generally has slow query evaluation. Additionally, frequently interacting with the disk is at odds with an indexing streaming of documents.

With the growth of memory capacity in recent years, researchers have begun exploring indexing in main memory. Luk and Lam [46] explored the feasibility of in-memory indexes for small datasets. Recently, Busch et al. [8] introduced Earlybird, Twitter's real-time search service. In their work, an index is kept in memory and updated using the in-place approach, and compression is applied in a post-processing phase. Following this, Asadi et al. [69] take advantage of block compression and perform on-the-fly compression when building and updating the index. All these, unlike merging in-memory postings with on-disk index, have the advantage of faster query evaluation [70]. However, these update algorithms assume inverted index as well and have the following limitations:

- The update process is complex. This is true for [8, 69] because they require a compression process that complicates the update.
- The memory requirement is high. For performance considerations, indexes and other related data must be kept completely in memory. Such related data includes document features that are required by (machine learned) rankers to elevate effectiveness [69].
- An inverted index is a non-regular data structure in which data access patterns are at odds with modern processor architecture that excels in processing contiguous memory blocks.

In a distributed search system, index update is simple for a system with a document-partitioned architecture. One simple solution is that one of the hosts is designated as the dynamic collection and rebuild is performed on this single

machine using one of the approaches explained above depending on how the index is stored [39]. Whereas for a system with a term-partition architecture, for all the machines that contain document terms as part of the index vocabulary, their indexes need to be rebuilt.

2.1.5 Other approaches to indexing

Although inverted index has been widely used and studied extensively by researchers during the past decades, it is not the only data structure that has been proposed for text search.

For cases where queries are rare or collection is small, a strategy that is simply doing an overall scan to evaluate each document against the query is reasonable [39]. In this approach, the collection doesn't have to be stored in a particular form of data structure, and can just be used for retrieval with the way it's stored. However, this approach is not attractive as it is fairly slow than other approaches.

Even for large collections, inverted index is not the only technology that has been proposed. Other alternatives include *suffix arrays* and *signature files* and Zobel and Moffat [39] provides with a detailed explanation of the pros and cons of these different indexing strategies.

Suffix arrays was first introduced by Manber and Myers [71], and it is a data structure that maintains an array of position pointers for each document pointing to each word, and these pointers are then sorted according to the words they point at. In order to determine whether a query term occurs in a document, a binary search

can be performed to locate the position of the term. However, suffix arrays have drawbacks compared to an inverted index. First, the pointer is accessed via binary search so compression is not an option. Second, pointers take too much of space than inverted index that's almost twice the size of the collection. Finally, there is no equivalent of ranked querying [39]. For a list of references, please see [72–75].

The material on signature files is based on [76]. In this indexing model, each word is converted to a signature of w bits, and a document descriptor is obtained by doing a bit operation on all the descriptors of the words. All these document descriptors are then stored in a file called signature file. To perform query evaluation, the query descriptor is obtained the same way as the document descriptors, and all the documents whose descriptors are a superset of the query descriptor are retrieved. The *false matches* are separated from the *true matches* with only the latter returned to the user as answers [39]. Faloutsos et al. [77] presented a different way of storing the signature files.

As suffix arrays, signature files also have drawbacks compared to an inverted index. First, as the number of false matches is positively correlated with the collection size, eliminating false matches can become costly as collection grows larger. Besides, the longer a document is, there is a higher possibility that a document will be labeled as a false match. Second, like suffix arrays, signature file indexes are large compared to compressed inverted files and are therefore difficult to construct or maintain. Finally, they require more disk accesses for short queries [39].

Interestingly, signature files recently has drawn attention back. Bing search engine, previously used inverted index, has developed and deployed an index based

on bit-sliced signatures known as BitFunnel to save operational costs [78]. This provides proof that inverted index might not be the best option any more given today's computer systems and architectures. Compared with our brute force scans techniques, signature files represents each document with a few bytes, whereas document pool represents each document with N integers, where N being the number of terms in each document. Therefore, signature files are more compact. However, signature files have false positives, while brute force scans does not. Hence, there is a space/accuracy tradeoff. Whether signature files outperforms brute force scans is a question to be answered in future work.

2.2 Brute Force Scans

As discussed in the previous section, many indexing approaches have been proposed and applied in the past few decades. Among all these approaches, inverted index is the one mostly used and proved to be quiet efficient with large collections [79, 80]. Suffix arrays and signature files are also used in some rare cases due to the drawbacks discussed above. An overall scan over the collection is a naive approach that most people dismiss.

In our work, we draw our attention back to the simplest approach to indexing, and propose brute force scans techniques as part of the core of our search system. At first glance, it seems unlikely that an architecture based on brute-force scans will be very efficient. However, we show that by taking advantages of the modern processor architecture, this approach is worth considering.

Firstly, one of the biggest challenges in designing software for modern processor architectures is the so-called *memory wall* [81]. To be more specific, interactions between processor and main memory dominates a system's running time. Unfortunately, for several decades now, increases in processor speed have far outpaced improvements in memory latency [9]. This means that RAM is becoming slower relative to the CPU. Therefore, to achieve higher throughput, just by speeding up the process is not sufficient because it is the RAM that is the bottleneck hence memory access needs to be optimized.

Modern architectures address this problem by introducing hierarchical *cache memories*, called L1, L2 and L3 cache memories, with increasing size and decreasing speed between the processores [82]. The cache memories serve as a buffer between the processor and the main memory. It's designed based on the assumption that a region in memory accessed by the program is likely to be requested later. Then the data in this particular region in memory can be stored in the cache memories. If the program asks for the same data, then there is a *cache hit* and system does not have to go to main memory to fetch the data with a high latency. On the other hand, if the data requested does not reside in L1, L2, L3, the main memory will be fetched one after another if the previous one does not contain the data, with an increase in latency. We call accessing a data not present in cache a *cache miss*. So although memory latencies are hidden by hierarchical caches, cache misses are expensive.

With the usage of caching, developers can take advantages of two properties of cache memories when building an efficient search system architecture. First, data is transferred from main memory to cache memories in blocks, called cache lines.

Therefore, even when the data in one particular memory location is requested, the data around it within a cache line is fetched together, so subsequent accesses to memory within that cache line can be very fast. Second, the program can prefetch the data from main memory into cache memories if the program accesses memory in a predictable sequential way.

Thus, it is imperative that developers structure memory access patterns to take advantage of prefetching. Inverted indexing and query evaluation with inverted indexes, unfortunately, are rife with irregular memory access and pointer chasing, which leads to poor performance. Brute force scans result in simple, predictable memory stride patterns.

Secondly, modern processor architecture also introduces pipelining to improve throughput. In a pipelined model, all the instructions are dispatched into different stages in one clock cycle. At each clock cycle, all the instructions are executed and advance to the next stage. Hence, the pipeline execution enables multiple instructions being executed at the same time.

However, pipelining suffers from two dangers: *data hazards* and *control hazards*. Data hazards occur when an instruction requires the result of a successive one, such as when dereferencing pointers. Control hazards occur when instructions are halted due to conditional jumps like if-else clauses etc. Although branch prediction techniques are proposed to pre-execute the instructions from the most likely branch, the hazard still can not be eliminated when the prediction is wrong and the processor has to undo all the instructions. Previous work in database management system has shown that these two hazards cost as twice the execution time as it is

supposed to be [83].

Branches have far reduced the maximum performance of a processor, and inverted indexing suffers exactly this inefficiency. Brute force scan, on other hand, can be coded with minimal branching and yield higher throughput.

Thirdly, the vector SIMD (single instruction multiple data) instructions in today's processors are common, which were designed to accelerate the performance of applications that perform repetitive operations on large arrays of numbers [84]. SIMD is a class of parallel computing in which a single instruction is performed on multiple data points simultaneously, hence supports data level parallelism and increases performance. Streaming SIMD Extensions (SSE) is an SIMD instruction set extension to the x86 architecture, and then was subsequently expanded by Intel to SSE2, SSE3, SSSE3, and SSE4. Recently, Advanced Vector Extensions (AVX), an advanced version of SSE, was announced by Intel and it provides a number of instructions for operating special 128-bit registers (e.g., four 32-bit integers), and 256-bit registers (e.g., eight 32-bit integers). Moreover, a later version AVX2 expands most vector integer AVX instructions to operate on 256-bit registers.

SIMD instructions have been used in database design especially table scan to improve performance. Zhou and Ross [84] used SIMD instructions to implement database operations, including sequential scans, aggregation, index operations and joins, and showed in their work that by taking advantage of parallelism and eliminating conditional branch instructions with SIMD, the CPU time for the new algorithms is from 10% to more than four times less than for the traditional algorithms. Similarly, Willhalm et al. [85] proposed a SIMD approach to accelerate

table scan operations over compressed in-memory column-store database systems, and showed that their approach considerably accelerates table scans and scales well with the number of cores. This gives confidence that SIMD instructions can be used in brute force scan to improve performance, and in our work, we mostly exploited AVX2 instructions to improve the performance of our search system.

Another reason worth mentioning is the increase in the number of possible cores a machine can have. With more cores, we can exploit parallelism to reduce query latency even on one machine, and compare the parallel performance of bruce force scan techniques versus inverted index. And this opens another opportunity for us to take advantage of modern processor architectures. Similar work has done by Qiao et al. [86] on main-memory database scan with multi-core scans. In their work, they proposed a novel FullSharing scheme to allow all queries to share the cache belonging to a given core when performing base-table I/O. It was able to remove the memory I/O bottleneck and increase Business Intelligence query throughput against main-memory database with multi-cores by a factor of 2 to 2.5. Another thread of work done by Cringean et al. [87] focuses on developing a parallel text retrieval system using a microprocessor network. Through the use of the transputer, a high-performance processor developed especially for use in multi-processor systems, the system achieves fast searching on serial text files, that is comparable to those obtained from inverted file systems.

Finally, in real-time search, users most often only care about the latest results. An inverted index can be implemented by traversing the posting lists "backwards" and exiting early when enough results have been accumulated. However,

most search systems are not designed this way and traditional query evaluations using these systems are not trivial. With brute force scans, since documents are arranged chronologically, we can simply scan from the end of the document pool and exit early when we have reached a proper time.

All the five reasons explained above motivated us to explore the feasibility of a brute force scans techniques based search architecture and answer the question whether that can outperform an inverted index.

2.3 Selective Search and Word Embeddings

In a distributed search system where documents are distributed among multiple machines, when queries arrive, they are passed to all the machines. Then all the relevant documents are retrieved from the indexes built on each machine and merged to generate a final list of results to be returned to the user. It is often done in parallel [88–92], and this type of search is referred to as exhaustive search.

However, exhaustive search might not be feasible when the collection is enormously large and the computational resources are limited. Selective Search is then proposed as an alternative approach. Selective Search divides a collection into multiple shards and selects only a few shards that are estimated to contain relevant documents for the query [2]. Selective search is often used to reduce search costs when the computational resources are limited. And although only a subset of the collection is considered for relevance, experiments have shown that it does not degrade the effectiveness much [2].

To summarize, Selective Search involves two steps: shard creation and shard selection. We'll present a literature of these two steps below.

2.3.1 Selective Search

2.3.1.1 Shard Creation

Shard Creation is to partition the collection into a set of shards. What policy to follow or what strategies to partition the collection is a research problem that draws much attention. Here we call this policy a *document allocation policy*.

Kulkarni and Callan [3] introduced three document allocation policies including *random*, *source-based* and *topic-based* document allocation. Random document allocation *randomly* partitions each document into one of N shards with equal probability. While the other two policies, Source-based document allocation and topic-based document allocation are motivated by the *cluster hypothesis* that "Closely associated documents tend to be relevant to the same requests" as defined by Rijsbergen [93]. The hypothesis suggests that if similar documents are grouped together, the relevant documents for a query reside in only a small number of clusters. To be more specific, source-based document allocation partitions documents such that documents within the same shard come from the same *resource*, such as a web host. Topic-based document allocation considers documents that are semantically similar and are closely associated and partitions them into the topical shards.

Both random document allocation and resource-based document allocation are easy to perform and do not require sophisticated algorithms, while topic-based

document allocation needs some kind of language processing techniques so as to group semantically similar documents together. Previous experiments have shown that topic-based document allocation has the best performance among the three allocation policies [3]. Below we will give a brief literature review of different topic-based document allocations that have been studied and how they are refined over time.

Prior research has shown that cluster-based and category-based document allocation were effective to define topic-based shards [5]. Document clusterings have been studied extensively in IR for the past few decades to improve retrieval efficiency or effectiveness [94, 95]. Following these studies, Xu and Croft [4] proposed a collection partitioning technique with a two-pass k-means clustering algorithm. In the first pass, the clusters are initialized with the first k documents. Each new document is assigned to the closest cluster. The second pass corrects possible mistakes made in the first pass. For retrieval, they employed Kullback-Liebler divergence-based distance metrics to select the relevant clusters. Specifically, they select the clusters that have the smallest KL divergence between the query's unigram language model and the clusters' unigram language model. The results show that by selecting the top 10 out of a total of 100 clusters, the performance is as good as that of an exhaustive search. However, when applying Xu and Croft's method to larger datasets, the computational cost becomes huge.

On a different thread, Puppin et al. [96] proposed a novel document partition and collection selection approach based on co-clustering queries and documents, with the help of a query log data. During the training process, documents are ranked and

recorded for each query. A co-clustering is then performed for queries and retrieved documents. Documents not retrieved are put in a separate cluster. Results show that the strategy is more effective than that of random partitioning. However, this method used external resources and has a generalization problem of being applied to other datasets.

To solve this problem of applying k-means to larger datasets, Kulkarni and Callan [3, 5] proposed a sample-based k-means approach. Specifically, they used uniform sampling to sample a small subset of documents from the collection and apply k-means to them. All the other documents are assigned to the K clusters using the symmetric version of negative Kullback-Liebler divergence. The approach cuts the search cost to less than 1/5th of that of the exhaustive search with, on average, no loss of accuracy. More recently, Callan [2] proposed a Size-bounded Sample-based (SB^2) k-means shard selection approach guaranteed to divide the collection into equal sized partitions so as to support a better selective search performance.

These approaches have several disadvantages in common: Firstly, they assume that collection is static and the clustering is processed in an off-line fashion. Therefore, these methods are not suitable for handling a streaming of documents (in our case, tweets) that require immediate clustering of the newly-encountered documents. Secondly, the computing cost of both k-means is high, and requires a lot of memory. This is due to the document representation format. Each document is represented as a sparse vector where each dimension corresponds to a separate term. If a term occurs in the document, its value in the vector is non-zero. Such a vector is in a tremendously large dimension, so computing distance between vectors and centroid

of these vectors for k-means leads to extremely high computing cost and resources requirements. Additionally, although sample-based k-means promise to achieve the same performance as an exhaustive search, it only uses partial information of the entire collection and might lose important information contained in non-selected documents. It would be desirable if we could make use of the entire collection to capture more information while still maintaining high efficiency.

To address the challenges, we propose the selective search approach with word embeddings vector representations for clustering(literature review later).

2.3.1.2 Shard Selection

Key to successful search is Shard Selection, that identifies from the partitioned shards a small set of shards that are relevant to the query. This usually starts with a ranking algorithm that sorts the shards in the order of relevance to the query, with the assumption that a shard that is more relevant to the query contains more relevant documents. Over the past few decades, researchers have studied this problem extensively, and the proposed ranking algorithms can be classified into three families: vocabulary-based, sample-based and feature-based algorithms.

In a vocabulary-based approach, each partition is represented by a language model and compared against the query to estimate the relevance [89,97–100]. It then selects the top k partitions. In Gravano and Garcia-Molina's work [99], both queries and documents are represented as vectors in the vector-space model, and documents are ranked by the cosines similarity between itself and the query. Such an approach

is efficient in that once we have preprocessed the documents into vectors, shard selection is a linear complexity operation.

Sample-based approaches like ReDDE [101] and SUSHI [102] samples documents from each partition to create a central sample index (CSI), which is then queried to estimate the distribution of relevant documents across the resources. This is more complex than a vocabulary-based approach. Additionally, building CSI and then doing retrieval a second time make the whole process less efficient.

In a feature-based approach, each partition is associated with a number of features, and a classification model is then used to rank partitions based on these features(see work of Arguello et al. [103]). An evaluation of the approach proposed in their work has shown that the approach performed at the same level as ReDDE [101] or significantly better than a vocabulary-based approach [98]. However, the approach requires training on the test queries, which might not be available in some cases.

In our work, we implemented ReDDE as an alternative to a cosine similarity approach, and did not find significant differences. Therefore, we adopted cosine similarity for efficiency reasons.

Shard selection not only is responsible for ranking the shards according to their relevance to the queries, but also determines the number of shards that need to be consulted for a given query. The goal of selective search is to achieve maximum accuracy while minimizing the cost by searching as few shards as possible [104]. Although a number of shard ranking approaches have been proposed as we have discussed above, not much attention has been drawn to explore the possibility of

a functional shard cut-off selection algorithm. Many previous work just used fixed cut-off numbers on all queries [96,98,101,105]. This approach is easy to implement, however, is shown to be either over or under estimating the minimum number of shards that need to be processed [104]. To list a few other approaches, Thomas and Shokouhi [102] proposed a cut-off estimator that is query-specific. Puppin et al. [92] used a selecting criteria that is query independent, which is to use the load on the system to determine the number of shards to be examined. Recently, Kulkarni et al. [104] presented three shard ranking algorithms that transform a flat CSI document ranking into a tree structure and by employing this hierarchy, they propose an approach that can dynamically estimate a query-specific minimal shard rank cutoff. Additionally, they have demonstrated that the search cost is substantially reduced while still maintaining accuracy that's on par with a strong baseline and with exhaustive search.

2.3.1.3 Evaluation Metrics

In speaking of measuring a selective search system's ability in reducing the cost without degrading accuracy, accuracy and cost should be properly defined before researchers can run the experiments. Accuracy, or effectiveness, is a measurement that tells how a system satisfies user's need given a certain query. Several evaluation metrics that have been widely used to measure effectiveness of a search ranking system are nDCG [106], Precision at k(P@k, where k=30 is often used) and Mean Average Precision(MAP) [107]. To give an example, P@10, P@30, MAP and nDCG@10 are

used in [2, 104, 108] to serve as the effectiveness measurement for their proposed selective search systems.

Aside from the evaluation metrics, different cost metrics are also proposed to measure search efficiency and search cost. Kulkarni [104] defined the cost metric as the average number of documents from selected shards that contain at least one query term over all queries, as is defined as follows:

$$C(q) = \sum_{i=1}^{n} |S_i(q)| \tag{2.1}$$

where, n is the number of top shards searched for query q, $S_i(q)$ is the set of documents in shard i that contain at least one query term. Similar cost metrics have been used in other research work by Moffat et al. [91] and Strohman et al. [109].

Aly et al. based their measure on the measure by Kulkani et al. and added in the cost metric the cost for executing the selection algorithm, arguing that the selection algorithm itself can require substantial resources, and should be reflected in the efficiency measure [108]. In their work, cost is defined as follows:

$$C_R(q) = C_{SEL}(q) + C(q) \tag{2.2}$$

where $C(q)$ is the measure by Kulkani et al., and $C_{SEL}(q)$ denotes the selection cost. $C_{SEL}(q)$ depends on the type of selection algorithm. In a sample-based algorithm, it's defined as $C_{SEL}(q) = |S_{CSI}(q)|$, where $S_{CSI}(q)$ is the set of documents in the CSI that have at least one query time. For vocabulary-based algorithm, $C_{SEL}(q) = N$, where N is the number of shards in the collection.

2.3.2 Word Embeddings

We use word embeddings, specifically GloVe to reduce the computational cost. The Global Vectors for Word Representation (GloVe) model [6] is a family of semantic vector space language models. Semantic vector space language models represent each document with a vector of real values and help computers to understand the meaning of human language. Such vectors are widely used in a variety of applications like information retrieval [40], document classification [110], question answering [111], named entity recognition [112], and parsing [113]. Two main model families for learning word vectors are: 1) global matrix factorization methods, such as latent semantic analysis (LSA) [114] and 2) local context window methods, such as the skip-gram model of Mikolov et al. [115].

GloVe is a global log-bilinear regression model for the unsupervised learning of word representations. It combines the advantages of the two major model families: global matrix factorization, and local context window methods. GloVe efficiently leverages statistical information by training only on the nonzero elements in a word-word cooccurrence matrix, rather than on the entire sparse matrix or on individual context windows in a large corpus. It outperforms other models on word analogy, word similarity, and named entity recognition tasks [6]. As GloVe is not the focus of our work, we encourage the readers to refer to the paper for more details.

2.4 Tweet Timeline Generation

Tweet Timeline Generation was introduced in the TREC 2014 Microblog track, of which I am a co-organizer. In this task, given the sampled tweet collection and queries, each system is required to return a list of non-redundant chronologically sorted tweets for each query. This task is similar to a few tasks, including topic detection and tracking (TDT) [116–118], "other" nuggets in question answering [119–121], nugget-based evaluations in DARPA's BOLT program,[1] and temporal summarization at TREC [122, 123]. They all share the property that atomic units of information should be grouped together to semantic equivalence classes. In TDT, each document discusses an event, hence atomic units are documents, and we group the ones that discuss the same event into the same group. In question answering, answers are formed by short phrases providing relevant information to the questions. In temporal summarization, the atomic units are formed by sentences. We see that atomic units are defined differently with respect to the different information needs of the tasks. For TTG evaluations, we simply adopt the solution used by some evaluations. That is, we declare the atomic units *by fiat*: tweets.

At first glance, it might seem that TTG is no different than TDT as they all group documents into topical clusters. However, TTG is different from TDT in that: Firstly, TDT contains five tasks, topic tracking, link detection, topic detection, first story detection and story segmentation, out of which, the two major tasks are topic detection and topic tracking. Topic detection requires detecting novel, previously

[1] `http://www.nist.gov/itl/iad/mig/bolt_p1.cfm`

unknown topics, deciding for a new document, whether it should be grouped into an existing topic or form a new topic. Topic tracking detects stories that discuss a previously known topic, and only this task resembles TTG in that given a set of pre-defined topics, the system is required to assign a new document into one of the topics if it contains information that is related to any of the topics. Secondly, even topic tracking differs from TTG in that for topic tracking, the tracking systems are given sets of on-topic stories and a portion of the corpus to train models on [118], while TTG system has no information about what the stories are like for a given topic and has to decide on its own or generate its own training data. Thirdly, the evaluation on a TDT system is easier because it is based on a well-defined corpus that all the systems use to perform the task, while for TTG, as the participants have to use the results from their tweet ad hoc retrieval as the starting point, where the tweets are gathered from the public Twitter stream from a certain time range in 2013, different TGT systems have different sets of collection to perform the task, and it adds the difficulty of evaluation. Fourthly, while TDT corpora are collection of news, including both text and speech, TTG corpora is just a collection of tweets, which is relatively short compared to news and therefore it is possible to apply some simple clustering algorithms to perform the task. Finally and most importantly, TDT focuses on recall as it aims in classifying all the documents into one of the pre-defined topics, while TTG's evaluation places an emphasize on the balance between precision and recall which is discussed in Chapter 5.

Once the atomic units have been defined, in order to measure systems' effectiveness in performing the task, we need the "ground truth" for evaluation purposes.

Therefore, defining the semantic equivalence classes ("ground truth") becomes a challenge the evaluation should address. It is challenging because such judgments are subjective. Just as assessors disagree over relevance judgments (see [124] for a nice summary), they do not agree on whether two pieces of information share the same content. Typically, for absolute system performance, we can simply take the judgments from one *single* assessor. However, because our goal is to induce systems comparisons, we can follow Voorhees's work [125], which demonstrated the stability of system comparisons with respect to different assessors on relevant judgments, and make a reasonable assumption that this finding also holds for semantic clustering tasks.

To our knowledge, this assumption has not been validated at scale. They either adopted judgments from one single assessor [119,120] or studied the assessors' differences where the judgments were restricted to a single set of nuggets [126]. A recent work on nugget-based evaluation methodology [127] has not studied the differences either. To verify our assumption for the tweet timeline generation task, we obtained two independent sets of semantic clusters for all topics. The results have shown that system comparisons are stable with respect to assessor differences in the TTG task.

With regards to the correlation between effectiveness metrics and user preferences in TTG, there have been a good amount of research studying such correlation with other tasks [128–136]. Early studies found no such correlation [128, 129, 131]. Smith and Kantor [134] showed that users do well with poor search results because they reformulate queries. Similarly, Lin and Smucker [137] found browsing to be

another way to compensate for poor search results. However, recent work detects a correlation between system effectiveness measured by metrics and user preferences [133, 135], and between precision and user preferences [136].

Given the above results, we can make a reasonable assumption that such a correlation exists for tweet timeline generation tasks. Additionally, the improvements on the system effectiveness can lead to improvements in users' information seeking abilities. To validate this, we follow the setup of Sanderson et al. [135], which shows assessors pairs of systems with effectiveness differences, and ask for preferences. However, our approach differs from their work in that: Firstly, their experiments are based on ad hoc retrieval task while ours is based on tweet timeline generation. Secondly, while they only choose pairs of systems with "large" and "small" effectiveness differences, we group the system pairs into buckets of different magnitudes of effectiveness differences. Finally, they obtained judgments from Amazon's Mechanical Turk, whereas ours came from local assessors.

2.5 Text Summarization with Deep Learning

Because the Tweet Timeline Generation task groups information together into semantic equivalence classes, it can be considered as a clustering task where the number of clusters is unspecified and needs to be determine by the system.

Different approaches have been proposed by the TREC 2014 Microblog track participants to accomplish the task[2]. These approaches can be categorized into

[2] `http://trec.nist.gov/pubs/trec23/trec2014.html`

clustering algorithms and ranking techniques.

Klein et al. [11] and Cui et al. [12] applied Affinity Propagation clustering algorithm [13] to group tweets. Affinity Propagation clustering has the advantage that the number of clusters can be dynamically computed without having to be predetermined. Cui et al. [12] also used the Single-pass clustering algorithm in their work. In this algorithm, each new tweet is compared with all the tweets in the existing clusters, and is put in a new cluster if the largest similarity with each tweet is smaller than the threshold, or is put in the cluster that has the largest doc-to-doc similarity. Lv et al. [15] applied both star clustering [14] and hierarchical clustering to perform the clustering task. Magdy et al. [16] used 1NN clustering. Hasanain and Elsayed [17] applied online-clustering [18]. Differently, Lu et al. [19] took advantage of Modified Maximal Marginal Relevance model [20] to give score between doc and query and doc to doc and used a function of these two scores as the selection criteria to select the next tweet into the final results. In order to run all the clustering algorithms, a distance or similarity score between each pair of documents needs to be defined beforehand. Cui et al. [12] represented documents as term weighted vectors based on word2vec [22] and used cosine similarity as the the similarity score between two documents. Lv et al. [15] converted documents to term frequency vectors and also used cosine similarity as the basis of their distance function. Magdy et al. [16] used a variant of jaccard similarity as the similarity score between two documents. Of all the approaches, the one proposed by Lv et al. [15] using hierarchical clustering is proved to achieve the highest effectiveness score.

Though many approaches have been explored to tackle the problem as dis-

cussed above, deep learning, however, to our knowledge, is an area that hasn't received much attention to solve a clustering problem like this. Deep learning is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using multiple processing layers with complex structures or is otherwise composed of multiple non-linear transformations [138–141]. Deep learning has been exploited in many supervised approaches [142–145]. The success behind deep learning is that it can learn useful information for data visualization and classification. However, little attention has been paid to leveraging deep learning for unsupervised clustering problems. Only recently, Chen [146] proposed deep belief network with nonparametric clustering such that the approach can learn features for clustering and infer model complexity in a unified framework. This shows that clustering with deep learning is indeed a reasonable practice and worth exploring.

Though the idea of applying deep learning directly to solve unsupervised clustering problems needs further exploration, another practical solution to clustering is to model sentence similarity with deep learning strategies and then apply Single-pass clustering algorithm. While most previous work on modeling sentence similarity are based on feature engineering [147–151], recent work has shifted the attention to modeling with distributed representations and neural network architectures. Socher et al. [152] proposed using a recursive neural network to model sentences, where it recursively computes the sentence representations. Hu et al. [153] used convolutional neural networks to model sentences and used entire sentence representations to compute similarities. Tai et al. [154] proposed the dependency tree Long Short

Term Memory (LSTM) neural networks to model sentences with the use of syntactic parsers and achieved state-of-art performance on the SemEval-2014 semantic textual similarity task [155]. Yin and Schütze [156] proposed a convolutional neural network architecture for the Microsoft Research paraphrase (MSRP) identification task [157], however, their best results require unsupervised pretraining. He et al. [21] proposed using a convolutional neural network to extract features at multiple levels of granularity to model each sentence, and comparing pairs of local regions of sentence representation to compute similarity. Without using parsers, their work achieved comparable performance to that of Tai et al. [154] on the SemEval-2014 task. And with no pretraining step, their work outperformed the model proposed by Yin and Schütze [156] on the MSRP task.

In our work, we will be using the convolutional neural networks model proposed by He et al. [21] to generate tweet representations and compute pairwise similarities. We then applied Single-pass clustering on the tweets, where each new tweet is compared with all the tweets in the existing clusters, and is put in a new cluster if the largest similarity with each tweet is smaller than the threshold, or is put in the cluster that has the largest doc-to-doc similarity.

Chapter 3: Brute Force Scans

As we discussed in Chapter 1, real-time tweet search requires the rapid ingestion of tweet streams and the guarantee that tweets are available for search immediately after they arrive. Previous work uses an inverted index as the core of the search architecture. In our work, however, we proposed a completely different search architecture that abandons the inverted index to address the challenge. Our search architecture is based on brute force scans of document representations. A traditional search architecture involves indexing documents and retrieving relevant documents for a user query. Our search architecture also has these two main parts and works as follows: tweets are represented with a simple data structure: array of integers, and stored in a built-in memory buffer which we call the document pool. Whenever we encounter a new tweet, in order to "index" it, we first convert it into arrays of integers, and then append them to the document pool. For retrieval, we first convert a query into a vector of integers the same way we converted the tweet. Then, we perform brute force scans over the document pool and compute query-document scores according to a scoring model.

We exploit the vector SIMD (single instruction multiple data) instructions that are common in today's processors to increase brute force scan performance.

Advanced Vector Extensions (AVX) are extensions to the x86 instruction set architecture that supports SIMD processing. AVX provides a number of instructions for operating special 128-bit registers (e.g., four 32-bit integers), and 256-bit registers (e.g., eight 32-bit integers); AVX2 expands most vector integer AVX instructions to operate on 256-bit registers. Our goal is to exploit these instructions to achieve high instruction throughput.

To present the work, in the following sections we first introduce the proposed search architecture, and discuss in detail how it indexes the tweet streams and performs retrieval with different query evaluation approaches. Then we discuss exploiting modern multi-core processors to increase performance. Following are the experiments we ran to study the efficiency of our techniques. We compare these experiments to Lucene, a traditional search system that uses an inverted index. Finally we conclude the work.

3.1 Approach

3.1.1 Document Representations

Before we proceed to describe the process of "indexing" the tweet streams and the resulting representations of tweets, we first introduce a dictionary that maps terms to 32-bit integer term ids. Starting from 1, every new term is assigned the next available term ID. Hence, whenever a new document is encountered, we first convert it to an array of unique term IDs, then append it to the end of the document pool. A second array of 8-bit integers stores corresponding term frequencies (tfs).

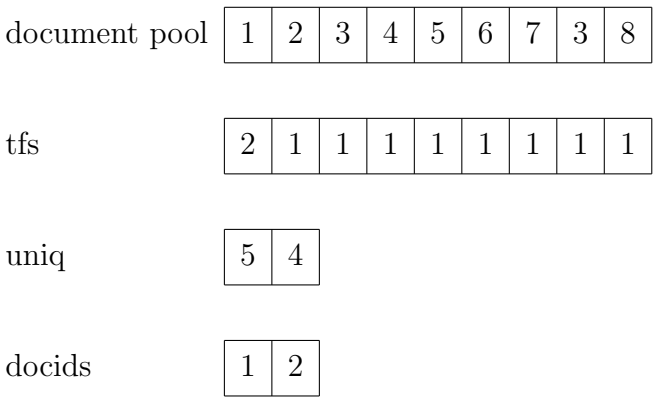

Figure 3.1: Unpadded document representations for two tweets "BBC News: The BBC cuts budget" and "Just watched The Rite" after tokenization.

We also keep track of an array of 8-bit integers that stores the number of unique terms in each document (uniq) for locating a certain document in the document pool. A final array stores the document IDs (docids). This is the unpadded version of data structures. To take advantage of AVX2 instructions, we also have padded data structures where we pad the array of term IDs and tfs for each document to the nearest multiple of eight with 0s to make the document boundaries align with SIMD instructions. We also modify the uniq array to reflect the new length. Figure 3.1 illustrates unpadded data structures for two tweets: "BBC News: The BBC cuts budget" and "Just watched The Rite" (tokenized to "bbc new the bbc cut budget" and "just watch the rite"). Figure 3.2 illustrates the padded data structures for the two tweets. To compute the document scores, we also store the document lengths in an array of 8-bit integers and the collection frequencies including: 1) total number of terms in the collection, 2) a map from a term to its collection frequency. But these data structures are not specific to our approach.

It is worth noting that we did *not* compress either the unpadded or padded document pool. There are two reasons for this: Firstly, because term IDs grow as

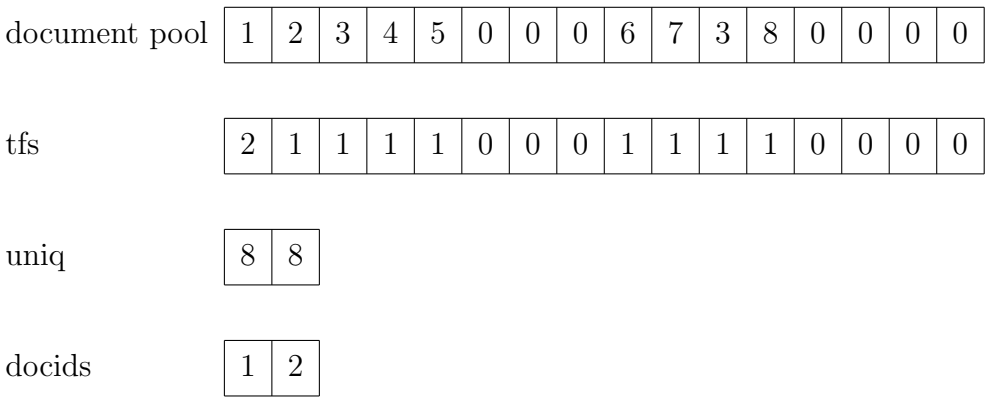

Figure 3.2: Padded document representations for two tweets "BBC News: The BBC cuts budget." and "Just watched The Rite." after tokenization.

new tweets are encountered, meaning that term IDs are relatively large, compressing does not save much space. We have tried compression using SIMD-BP128 [158], though it saved a small amount of memory, we decided that because of the slower performance, the tradeoff was not worthwhile. Although it is possible to assign term IDs like ordering terms by frequency to facilitate compressions, this would complicate dictionary construction in a real-time scenario. Secondly, it is easier to exploit SIMD instructions if we leave the document pool uncompressed.

3.1.2 Query Evaluation

To retrieve relevant documents for a user query from the data structures above, the basic idea is to scan the document pool and determine the relevance of each document. This is possible due to the simplicity of the data structures. To compute the relevance we use query likelihood model [40].

Although writing code to accomplish retrieval seems simple, the search efficiency can be highly affected by how the code is written to evaluate the query.

Therefore, we exploited different code optimizations and explored a number of query evaluation approaches based on a brute force scan of the document representations to compute query-document scores (query likelihood) to study the effects. To return k documents, the top k documents with highest scores are retained in a heap and returned after all documents are processed. Six different implementations are described below:

Scan1: The first approach (Algorithm 1) operates on the unpadded document pool and consists of three nested loops: over all documents, over all unique terms in each document, and over all query terms. Documents that are posted after the query time are skipped during evaluation. Here, b is used as the offset in the document pool and starts with 0, s stores the evaluation score for each document, and Q is an array representation of the query. The notation $S(\cdot)$ is shorthand for computing the score contribution of the current query term.

Scan2: Our second approach (Algorithm 2) also operates on the unpadded document pool and is the same as the first approach except that we unroll the innermost loop (i.e., lines 6–10). Here, we have query evaluation functions for every query length (a relatively small number), and every query is directed to its corresponding function for evaluation. By hard coding the number of query terms, we avoid branch erroneously predicting the innermost loop of Scan1.

AVXScan1: This approach, modified based on Algorithm 2, aims to avoid more branch mispredictions, and applies SIMD instructions on the padded document pool. To be more specific, the two inner loops of Algorithm 1 are replaced with SIMD

Algorithm 1 Scan1

```
1:  procedure SCAN1(Q, pool, tfs, uniq, docids)
2:      b ← 0
3:      for i ← 1 to N do
4:          s ← 0
5:          for j ← 1 to uniq[i] do
6:              for k ← 1 to |Q| do
7:                  if Q[k] = pool[b + j] then
8:                      s ← s + S(·)
9:                  end if
10:             end for
11:         end for
12:         if s > 0 then
13:             heap.add(docids[i], s)
14:         end if
15:         b ← b + uniq[i]
16:     end for
17: end procedure
```

instructions. It functions as follows: every query term ID is repeated eight times and represented by a vector of eight 32-bit integers (in 256-bit registers). It is then compared to eight integers from the document pool by the AVX2 vectorized comparison operation. The result is a mask and indicates whether there is a match between the integers. If there is a match, we compute the query term score contribution and add it to the current score. Compared to Algorithm 2, we scan eight unique terms within a document at a time instead of one, which reduces branch mispredicts. In the pseudocode, the notation pool_256 indicates a 256-bit register storing eight integers from the document pool. Similarly, Q_256 is a 256-bit register storing the replicated query term. Function "compare" compares the two vectors and stores the

Algorithm 2 Scan2

```
    procedure SCAN2(Q, pool, tfs, uniq, docids)
2:      b ← 0
        for i ← 1 to N do
4:          s ← 0
            for j ← 1 to uniq[i] do
6:              if Q[1] = pool[b + j] then
                    s ← s + S(·)
8:              end if
                Same for Q[2], Q[3]... from line 6 to 8 if there are any
10:         end for
            if s > 0 then
12:             heap.add(docids[i], s)
            end if
14:         b ← b + uniq[i]
        end for
16: end procedure
```

matching result in a variable called "mask", and mask is non-zero if there is a match and zero otherwise. s_256 is a vector (denoted as V) of eight 32-bit integers where each integer stores the score of the query term assuming there is a match between the query term and the document term in the corresponding location. Finally, the function "and" operates on mask and s_256 to get a 256-bit integer that keeps only the score from V where there is a match between the document term and the query term. These 8 integers are then added together to the final evaluation score as the partial score for the eight document terms being compared. The addition is done with several SIMD instructions, which is omitted in the pseudocode.

AVXScan2: This approach (Algorithm 4) modifies Algorithm 2 by unrolling the outermost loop to consider six documents at a time instead of one. This is mo-

Algorithm 3 AVXScan1

```
procedure AVXSCAN1(Q, pool, tfs, uniq, docids)
    b ← 0
    for i ← 1 to N do
        s ← 0
        for j ← 1 to uniq[i] do
            pool_256 ← pool(b + j : b + j + 7)
            Q_256 ← (Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1])
            mask ← compare(pool_256, Q_256)
            if mask ≠ 0 then
                s_256 ← {V|V_k = S(·), k ← 0 to 7}
                part_score ← and(mask, s_256)
                s ← s + Σ_{i=1}^{8} part_score(i)
            end if
            Same for Q[2], Q[3]... from line 7 to 13 if there are any
            j ← j + 8
        end for
        if s > 0 then
            heap.add(docids[i], s)
        end if
        b ← b + uniq[i]
    end for
end procedure
```

tivated by the fact that a branch misprediction usually costs between 10 and 20 clock cycles, rendering a CPU idle, where it could otherwise be executing multiple instructions. The outermost loop is the part that wastes the clocks the most because the document collection is usually in the millions or billions. If we can avoid some of the branch mispredictions, the system can become more efficient. To describe the process in detail, inside the outermost loop, instead of writing six for loops to iterate through the term ID of each document, which does not help reduce the num-

ber of mispredictions, we have one loop that iterates through the term IDs of every six tweets. In addition, a six-document array holds partial scores and after we have computed all the query-document scores, each non-zero element is then inserted into the heap. The only additional change is that we keep track of an array "pos" that stores the corresponding position of each term ID in the six-document array so that for each block of eight integers from the document pool, we know which element in the six-document array benefits from the score contribution.

The motivation behind this approach is to reduce loop overhead. The number of documents to process at once is heuristically determined and the reason for choosing six as a unit is that it is large enough to reduce the iterations of the outmost for loop therefore reducing the number of mispredictions, but it is not too large. Larger unit otherwise requires a larger array holding partial scores, hence leading to longer code (with more conditional statements to check each score for positiveness) that is difficult to compile.

AVXScan3: This approach(Algorithm 5) explores more optimization strategies and modifies Algorithm 4 in two ways: First, we no longer keep track of a 256-bit register score vector, nor perform a binary add operation with the mask and several SIMD addition instructions to obtain the final evaluation score. Instead, we compute the location of the matching document term from the mask and compute the score directly. Because a 256-bit mask is hard to manipulate, we first convert it into an 8-bit integer where each bit indicates whether there is a match between the document term and the query term in the corresponding location, 1 if there is a match and 0

Algorithm 4 AVXScan2

procedure AVXSCAN2(Q, pool, tfs, uniq, docids)

$b \leftarrow 0$

for $i \leftarrow 1$ to N **do**

$s[1:6] \leftarrow 0$

for $j \leftarrow 1$ to $\sum_{k=i}^{i+5} \text{uniq}[k]$ **do**

pool_256 $\leftarrow$ pool$(b+j : b+j+7)$

Q_256 $\leftarrow (Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1])$

mask $\leftarrow$ *compare*(pool_256, Q_256)

if mask $\neq 0$ **then**

s_256 $\leftarrow \{V | V_k = S(\cdot), k \leftarrow 0 \text{ to } 7\}$

part_score $\leftarrow$ *and*(mask, s_256)

ps $\leftarrow$ *pos*$[b+j]$

$s[\text{ps}] \leftarrow s[\text{ps}] + \sum_{i=1}^{8} \text{part\_score}(i)$

end if

Same for Q[2], Q[3]... from line 7 to 14 if there are any

$j \leftarrow j + 8$

end for

if $s[1] > 0$ **then**

heap.add(docids$[i], s[1]$)

end if

if $s[2] > 0$ **then**

heap.add(docids$[i], s[2]$)

end if

...

if $s[6] > 0$ **then**

heap.add(docids$[i], s[6]$)

end if

$b \leftarrow b + \sum_{k=i}^{i+5} \text{uniq}[k]$

$i \leftarrow i + 6$

end for

end procedure

otherwise. We then maintain a map (denoted as *loc_dict*) from the mask value to the corresponding document term location where the keys are 1, 2, 4, 8, 16, 32, 64 an 128 and values are 0, 1, 2, 3, 4, 5, 6, 7 respectively. There are only eight possible key value pairs because only one term can match the query term at a time, as the document pool only stores unique term IDs. Each value then implies a position in the eight integer document vector where the query evaluation needs to be performed. Therefore, we can directly perform query evaluation for that particular document term given a mask. This is motivated by the fact that even SIMD instructions are fast in processing multiple data simultaneously, multiple of them like additions are still slow and cause the CPU to take the clock cycles to run. Second, Algorithm 4 has one outer loop that iterates through the term IDs of every six tweets where the number of documents to process at once is heuristically determined. However, after running experiments with all possible unit values from two up to twelve, we found that ten is a unit that gains the best performance. Therefore, we change the outer loop to iterate through the term IDs of ten documents in this algorithm.

AVXScan4: This approach(Algorithm 6) modifies Algorithm 5 by pre-storing the evaluation scores in a 1D array for each term ID in the document pool. In this algorithm, $S(\cdot)$ is no longer computed on the fly while scanning through the documents, but rather fetched from the prestored score array (denoted as *score_arr*). This score array is constructed at the same time when the document pool is constructed, and is computed exactly the same way as in Algorithm 5, in other words, using query likelihood model. The motivation behind this approach is that the overall performance

Algorithm 5 AVXScan3

procedure AVXSCAN3(Q, pool, tfs, uniq, docids)

Initialize map *loc_dict*

$b \leftarrow 0$

for $i \leftarrow 1$ to N **do**

$s[1:10] \leftarrow 0$

for $j \leftarrow 1$ to $\sum_{k=i}^{i+5} \text{uniq}[k]$ **do**

pool_256 $\leftarrow$ pool$(b+j : b+j+7)$

Q_256 $\leftarrow (Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1])$

mask $\leftarrow$ *converto8bits*(*compare*(pool_256, Q_256))

if mask $\neq 0$ **then**

ps $\leftarrow$ *pos*$[b+j]$

s[ps] $\leftarrow s$[ps] + $S(\cdot)$ at location b + *loc_dict*$[mask]$

end if

Same for Q[2], Q[3]... from line 8 to 13 if there are any

$j \leftarrow j + 8$

end for

if $s[1] > 0$ **then**

heap.add(docids$[i], s[1]$)

end if

if $s[2] > 0$ **then**

heap.add(docids$[i], s[2]$)

end if

...

if $s[9] > 0$ **then**

heap.add(docids$[i], s[9]$)

end if

$b \leftarrow b + \sum_{k=i}^{i+9} \text{uniq}[k]$

$i \leftarrow i + 10$

end for

end procedure

of all the previous algorithms are slowed down by the query evaluation, which involves in logarithmic computation. Logarithmic computation is generally expensive in today's computers, and can even be slower than memory access. Therefore, if we were able to store the evaluation scores in memory, accessing it during evaluation should be fairly faster than computing the score on the fly.

3.1.3 Exploiting Parallelism

Modern processors contain multiple cores, thus support multithreading architecture that can manage multiple program executions at a time. To take advantages of this, we also exploit parallelism in our brute force scan approach. The two strategies are inter-query parallelism and intra-query parallelism.

With inter-query parallelism, we run query evaluations on multiple threads, and each thread concurrently operates on one query and scans the entire document pool to retrieve relevant tweets in response to that query. In this approach, as the number of threads grows, query throughput (the number of queries a program can handle at one time) increases up to a certain point, after which performance no longer increases due to resource contention. With this approach, we focus on the throughput (the higher, the better) in our experiments. In theory, there are both synchronized versions and non-synchronized versions of this approach, and they work as follows: synchronized means that every thread waits for all the other threads to finish their scanning job before we assign new queries to each of them. Because brute force scans read through the document pool in a sequential order, this

Algorithm 6 AVXScan4

procedure AVXSCAN4(Q, pool, tfs, uniq, docids)

$score\_arr \leftarrow []$

for $i \leftarrow 1$ to $|pool|$ **do**

$score\_arr[i] \leftarrow S(\cdot)$ at $pool[i]$

end for

Initialize map loc_dict

$b \leftarrow 0$

for $i \leftarrow 1$ to N **do**

$s[1:10] \leftarrow 0$

for $j \leftarrow 1$ to $\sum_{k=i}^{i+5} \text{uniq}[k]$ **do**

pool_256 $\leftarrow$ pool$(b+j : b+j+7)$

Q_256 $\leftarrow (Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1], Q[1])$

mask $\leftarrow converto8bits(compare(\text{pool\_256}, \text{Q\_256}))$

if mask $\neq 0$ **then**

ps $\leftarrow pos[b+j]$

$s[\text{ps}] \leftarrow s[\text{ps}] + score\_arr[b + loc\_dict[mask]]$

end if

Same for Q[2], Q[3]... from line 12 to 17 if there are any

$j \leftarrow j + 8$

end for

if $s[1] > 0$ **then**

heap.add(docids$[i], s[1]$)

end if

if $s[2] > 0$ **then**

heap.add(docids$[i], s[2]$)

end if

...

if $s[9] > 0$ **then**

heap.add(docids$[i], s[9]$)

end if

$b \leftarrow b + \sum_{k=i}^{i+9} \text{uniq}[k]$

$i \leftarrow i + 10$

end for

end procedure

guarantees that at any given time, every thread requests the same block from the memory so as to avoid content conflict. Non-synchronized means that each thread does not rely on the status of other threads and can proceed with the next query once it is finished with the current one. The downside to this approach is that there might be content conflict, but it is unlikely that a thread becomes idle. Hence, this approach utilizes the threads to the maximum. Experiments have shown that the synchronized version has a much smaller throughput than the unsynchronized one, indicating that less resource conflict does not compensate for the time wasted in waiting for other threads, so we only keep the unsynchronized version as we run experiments.

With intra-query parallelism, we split the collection into equal parts and build a document pool for each of them. Every portion is allocated a thread to run brute force scans; each thread maintains its own heap. Unlike inter-query parallelism, which processes multiple queries concurrently at one time, intra-query parallelism processes one query at any given time. There is a synchronization point in waiting for all threads to finish running the current query, then, before all threads proceed to the next query, we merge all heaps to produce a final top k ranking. With this approach, as the number of threads grows, query latency decreases to a certain point where the benefits of increased parallelism do not compensate for the synchronization costs. With this approach, we focus on query latency (the lower, the better) in our experiments.

3.2 Experimental Setup

We write our code in C for both inter-query and intra-query parallelism, and compare the performance with the open-source search engine Lucene (version 4.3.1). All code used in our experiments is released under an open-source.[1]

Although Lucene is implemented in Java and might seem unreasonable as a baseline, there are two reasons why we did so: First, Lucene is widely used in production by companies like Twitter, Bloomberg, and LinkedIn. Hence, it is more mature than other research systems implemented in C/C++. Second, Lucene has pretty low search latency compared to other search systems like Indri [159]. Therefore, it acts as a good baseline to compare with. Third, brute force scans and Lucene searches share two features that other research systems are lacking: real-time indexing, and support for multi-threaded retrieval. These similarities make a reasonable comparison between Lucene and our brute force scan approach. Some implementation details worth mentioning are that for the Lucene search, we load all indexes into the memory before running the experiments so as to make a fair comparison. Intra-query parallelism with Lucene is achieved by building an index for each of the split portions and running the Lucene search on them in parallel with multiple threads. Finally, for both brute force scans and Lucene, we retrieved the top 1000 hits, as per usual the practice in IR research.

We conducted our experiments on a machine with two Intel Xeon E5-2676 v3 processors at 2.4 GHz. Each processor has 40 cores, giving us a total of 80 cores

[1] `https://github.com/lintool/c-bfscan`

capable of supporting 80 virtual cores via hyperthreading. Our machine has 160 GB of RAM. The machine runs on an Amazon Elastic Compute Cloud (Amazon EC2) instance that runs Ubuntu (release 16.04 LTS) and we used gcc version 4.9.1. All reported results are the average of three trials.

We used test collections from the Microblog tracks at TREC 2011 and 2012 [160, 161], called the Tweets2011 collection, which consists of an approximately 1% sample (after some spam removal) of tweets from January 23, 2011 to February 7, 2011 (inclusive), totaling approximately 16 million tweets. We first verified with TREC 2011 topics (a total of 50) that our brute force scan approach generates the same scores for each document per query as Lucene. Then our experiments focused exclusively on efficiency. For these experiments, to get reliable measurements, we took the first 1000 queries from the TREC 2005 terabyte track efficiency queries. As these queries lack timestamps contained in TREC topics, we simply scan the entire collection without any early termination.

After processing, the size of the unpadded document pool was 162 million; all data structures (but not vocabulary) occupied 1026 MB of memory. The padded document pool expanded to 225 million, or an increase of 39%. All the padded data structures occupied 1341 MB of memory. The processing of generating document pool took about 6 mins. For reference, Lucene index occupies 4.8 GB in total including positions and metadata, and it took 78 mins in total, which is 13 times slower than that of brute force scans.

To learn the scalability of our brute force scan techniques, same experiments are run on a larger test collection from the Microblog tracks at TREC 2013 [162]

called the Tweets2013 collection, which consists of sample of tweets from February 1, 2013 to March 31, 2013 (inclusive), totaling approximately 259 million tweets. After processing, the size of the unpadded document pool was 2.5 billion; all data structures occupied 13.9G of memory. The padded document pool expanded to 3.5 billion, or an increase of 39%. All the padded data structures occupied 18.5G of memory. The processing of generating document pool took about 108 mins. Lucene index occupies 46 GB in total including positions and metadata, and it took 1421 mins in total, which is 13 times slower than that of brute force scans.

3.3 Results

3.3.1 Single-Threaded Experiments

Results for single-threaded experiments are shown in the first row of Table 3.1. Here we report the average query latency (over three trials) of Lucene and brute force scan approaches with 95% confidence intervals. We found that our AVXScan4 approach has the lowest query latency, that is 17% decrease over Lucene, whereas Scan1 approach is about 4.2× slower than Lucene. Every successive improvement to the basic brute force scan improves query latency. By unrolling the inner for loop and writing a query evaluation function for queries of different lengths (Scan2), we get a big gain (decreased by 39%) over Scan1. Although replacing the inner loop with SIMD instructions (AVXScan1) does not yield big improvements (decreased by 51% over Scan1), processing six documents at a time (AVXScan2) makes a noticeable contribution, that is a 59% decrease over Scan1. Another big improvement

(decreased by 78% over Scan1) is achieved by reducing the number of SIMD instructions to compute the document-query score (AVXScan3). However, pre-storing the score (AVXScan4) doesn't yield big improvements (decreased by 80% over Scan1). The bottom line is that the best brute force scan approach is 17% faster than the performance of Lucene.

3.3.2 Multi-Threaded Experiments

Results for multi-threaded performance exploiting intra-query parallelism are shown in the rest of Table 3.1. Here we report query latency of Lucene versus the brute force scan approaches with 95% confidence intervals. Note that instead of showing the performance for all possible number of threads, which is unnecessary and takes too much space, we chose only several points that represent the performance differences.

Looking vertically at the table, that is to compare the performance with the different number of threads of a specific approach, we see that Lucene achieves maximum performance with 32 threads, but not 40, that is equal to the number of physical cores in our machine. This is because the benefit of parallelism with 40 threads does not compensate the latency of waiting for that many threads. Increasing to 80 threads does not improve and even decreases the performance, meaning that hyperthreading does not help in the Lucene search. On the contrary, looking at other columns, Scan1 and Scan2 achieve the maximum performance with 40 threads while the other six brute force scan approaches achieve the maximum performance

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
1	224.1±0.1	934.4±0.6	566.6±3.5	456.1±1.0	381.5±1.1	203.0±1.5	185.0±0.7
2	125.7±1.3	526.8±9.3	334.9±1.0	260.0±1.9	219.5±1.2	115.3±4.7	112.6±0.7
4	100.9±1.0	298.2±0.5	193.8±0.3	153.8±0.2	131.1±1.4	75.9±1.2	70.9±0.9
8	78.6±2.1	172.1±1.0	114.7±0.7	90.3±0.6	79.7±1.5	50.2±0.3	47.7±0.3
12	73.9±1.6	141.8±1.2	96.1±0.5	74.4±0.7	67.3±0.3	48.1±1.8	46.5±0.1
16	67.3±0.4	126.0±1.0	85.2±0.9	65.9±0.6	64.4±0.8	47.3±0.3	46.3±0.3
24	57.3±0.4	103.4±0.3	68.4±1.1	60.8±2.4	62.0±0.9	46.8±0.4	46.6±0.9
32	56.7±1.8	88.9±3.8	57.5±1.8	57.2±0.7	56.6±0.9	47.2±0.8	46.4±1.3
40	62.1±1.7	79.7±2.4	56.5±0.4	58.6±0.6	60.3±1.1	50.6±1.1	51.3±0.4
80	78.7±1.0	90.5±1.9	72.0±0.8	70.6±1.1	73.0±2.0	65.1±1.0	67.2±1.3

Table 3.1: Latency (in milliseconds) with 95% confidence intervals of Lucene versus brute force scan, exploiting intra-query parallelism on Tweets2011.

with smaller number of threads, that is 32. This is because Scan1 and Scan2 process on the unpadded data structure that is smaller than padded data structure, hence the latency variance among different threads is lower, leading to less waiting time in synchronization. Similar to Lucene, we see that all of our brute force scans approaches also struggle with hyperthreading in performance.

If we look at the table horizontally – that is to compare the performance of different approaches under the same setting – we see a decrease in the query latency as we improve our brute force scan approaches with multiple threads. However, an interesting fact to note is that there is a different relative performance ordering of the brute force scan approaches compared to the single-threaded results: AVXScan1 performs better than AVXScan2 beyond 24 threads, whereas AVXScan2 beats AVXScan1 under a single-threaded setting. Overall, the maximum performance we achieve with brute force scans (AVXScan4 with 32 threads) is faster than Lucene under the same setting. The bottom line is that the best brute force scan approach is 18% faster than the best Lucene performance.

The results for multi-threaded performance exploiting inter-query parallelism are shown in Table 3.2. Here we follow the same arrangement as in Table 3.1 and report throughput (query processed per second) of Lucene versus the brute force scan approaches with 95% confidence intervals. Viewing the table vertically, we see that Lucene achieves the best performance with 32 threads. This suggests that the resource contention outweighs the benefits gained by additional parallelism. Similarly, all of our brute force scan approaches achieve maximum performance with 32 threads. If we look at the table horizontally, we see an increase in the throughput

of the six brute force scan approaches. With single thread, both AVXScan3 and AVXScan4 beat the throughput of Lucene. Interestingly, just as in the case of intra-query parallelism, there is a different relative performance ordering of the brute force scan approaches compared to the single-threaded results: AVXScan3 performs better than AVXScan4 beyond 24 threads, whereas AVXScan4 beats AVXScan3 under a single-threaded setting. This suggests more resource contention when there is more data to access. AVXScan3 with 32 threads achieves the maximum performance over all six approaches, while it has a slower performance on a single thread compared to AVXScan4. Overall, unfortunately, none of our brute force scan approaches achieves throughput that is competitive to that of Lucene. The bottom line is that inter-query parallelism is only 35% slower than the best Lucene performance.

3.3.3 Experiments with additional datasets

To study the scalability of our brute force scan techniques, we run the same experiments on Tweets2013 and show the multi-threaded performance exploiting intra-query parallelism in Table 3.3, and multi-threaded performance exploiting inter-query parallelism in Table 3.4 the same way as in Table 3.1 and Table 3.2.

Looking at Table 3.3, we see that unlike with Tweets2011, while AVXScan4 still beats Lucene with single thread, it is not the same with AVXScan3. Lucene achieves maximum performance with 32 threads as with Tweets2011. Increasing to 80 threads does not improve the performance, either. Looking at other columns, Scan1 and Scan2 achieve maximum performance with 40 threads while AVXScan1

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
1	5.8±0.0	1.2±0.0	2.0±0.0	2.4±0.0	4.4±0.1	6.8±0.0	7.4±0.0
2	11.0±0.0	2.3±0.0	4.0±0.0	4.8±0.0	8.4±0.1	12.3±0.0	13.2±0.1
4	29.8±0.1	4.5±0.0	7.8±0.0	9.3±0.0	16.2±0.1	23.1±0.1	24.6±0.1
8	46.9±0.5	8.9±0.0	15.4±0.0	18.1±0.1	29.7±0.0	41.0±0.2	42.2±0.1
12	65.5±0.9	13.2±0.0	22.8±0.1	25.9±0.2	40.0±0.5	52.2±1.0	53.6±2.4
16	78.1±0.7	17.2±0.1	29.4±0.6	32.4±0.1	47.1±0.3	59.5±2.5	60.1±4.3
24	100.0±2.6	22.4±0.0	38.2±0.5	41.1±0.4	53.6±0.4	63.2±3.0	62.9±0.3
32	106.1±2.2	24.4±0.1	41.9±0.3	46.1±0.5	56.6±2.5	68.6±0.2	67.7±4.3
40	99.9±1.0	23.3±0.2	38.9±1.1	44.3±0.3	52.0±5.4	53.9±6.1	48.8±3.3
80	100.2±1.9	19.5±0.2	33.0±0.4	38.7±0.2	50.7±4.8	56.0±1.4	53.9±2.7

Table 3.2: Throughput (query per second) with 95% confidence intervals of Lucene versus brute force scan, exploiting inter-query parallelism on Tweets2011.

and AVXScan2 achieve maximum performance with 32 threads, and AVXScan3 and AVXScan4 with 80 threads have the best performance. Looking at the table horizontally, we see a decrease in the query latency as we improve our brute force scan approaches with multiple threads. The different relative performance ordering of the brute force scan approaches compared to the single-threaded results still exists. Overall, the maximum performance we achieve with brute force scans (Scan2 with 40 threads) is no longer faster than Lucene under the same setting, and it is more than 2 *times* slower than Lucene. The bottom line is that intra-query parallelism does not appear to be the best way to exploit multi cores using brute force scan techniques with larger collections.

Viewing Table 3.4, we see that Lucene achieves the best performance with 40 (or 80) threads. On the contrary, Scan1 and Scan2 achieve maximum performance with 40 threads while all other four brute force scan approaches achieve maximum performance with 80 threads. If we look at the table horizontally, we also see an increase in the throughput of the six brute force scan approaches. With single thread, only AVXScan4 beats the throughput of Lucene. Interestingly, a different relative performance ordering of the brute force scan approaches compared to the single-threaded results occurs between Scan2 and AVXScan1, AVXScan2: Scan2 performs better than AVXScan1 and AVXScan2 with 32 and 40 threads, whereas AVXScan1 and AVXScan2 beats Scan2 under a single-threaded setting. AVXScan3 with 80 threads achieves the maximum performance over all six approaches, while it has a slower performance on a single thread compared to AVXScan4. Overall, unfortunately, none of our brute force scan approaches achieves throughput that is

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
1	2489.8±4.8	13681.4±12.9	8084.0±14.9	6297.0±10.3	5243.0±14.9	2498.7±24.7	2277.6±10.7
2	898.7±1.9	7082.1±5.2	4176.3±10.6	3268.1±1.1	2742.6±14.1	1338.8±11.1	1238.8±13.9
4	738.2±4.9	3707.5±9.5	2196.5±5.3	1700.6±4.5	1457.5±5.9	776.5±10.8	737.7±39.6
8	304.1±1.5	1884.1±2.1	1136.8±1.6	922.6±9.1	826.8±5.6	570.7±9.5	561.5±1.7
12	245.1±1.7	1356.8±3.6	863.8±1.6	728.6±8.2	685.1±6.0	548.5±7.0	544.3±14.1
16	224.9±1.6	1221.1±9.0	776.3±5.0	649.3±8.2	625.0±5.3	539.5±7.1	539.2±4.8
24	225.7±0.9	954.0±2.5	622.6±2.5	534.5±19.1	528.8±30.2	544.9±5.6	541.2±9.2
32	202.5±2.6	761.1±4.0	519.9±3.8	493.9±89.4	510.1±48.7	563.1±2.5	563.9±3.6
40	202.7±1.0	661.5±1.6	489.4±2.9	521.9±72.1	559.8±45.1	548.7±1.3	561.7±0.3
80	220.0±1.3	806.0±3.4	547.7±2.4	565.6±12.3	545.5±13.2	513.5±1.7	510.4±1.0

Table 3.3: Latency (in milliseconds) with 95% confidence intervals of Lucene versus brute force scan, exploiting intra-query parallelism on Tweets2013.

anywhere close to that of Lucene. The bottom line is that inter-query parallelism does not appear to be the best way to exploit multi cores using brute force scan techniques with larger collections either.

Experimental results from Tweets2011 and Tweets2013 with Lucene and different brute force scan approaches show that the relative performance between our approaches and Lucene is not consistent across collections of different sizes. Not surprisingly, the smaller the collection is, the better performance our brute force scan approaches achieve compared to Lucene. This is due to the fact that Lucene and brute force scan approaches examine a different number of documents for each query. Brute force scan approaches perform a linear scan over the entire collection

Threads	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
1	0.4±0.0	0.1±0.0	0.1±0.0	0.2±0.0	0.2±0.0	0.3±0.0	0.5±0.0
2	0.8±0.0	0.1±0.0	0.3±0.0	0.3±0.0	0.3±0.0	0.8±0.0	0.9±0.0
4	2.5±0.0	0.3±0.0	0.5±0.0	0.6±0.0	0.6±0.0	1.5±0.0	1.6±0.0
8	3.3±0.0	0.6±0.0	1.0±0.0	1.2±0.0	1.2±0.0	2.5±0.1	2.6±0.1
12	7.0±0.1	0.8±0.0	1.5±0.0	1.7±0.0	1.7±0.0	3.0±0.0	3.0±0.2
16	6.2±0.1	1.1±0.0	1.9±0.0	2.2±0.0	2.2±0.0	3.3±0.1	3.4±0.2
24	7.2±0.0	1.5±0.0	2.5±0.0	2.7±0.1	2.7±0.1	3.3±0.2	3.4±0.2
32	8.1±0.0	1.6±0.0	2.7±0.0	2.6±0.0	2.6±0.0	3.4±0.0	3.5±0.1
40	8.5±0.1	1.7±0.0	2.9±0.0	2.6±0.0	2.6±0.0	3.0±0.1	3.1±0.1
80	8.5±0.1	1.6±0.0	2.8±0.0	3.1±0.1	3.1±0.1	4.2±0.3	4.1±0.3

Table 3.4: Throughput (query per second) with 95% confidence intervals of Lucene versus brute force scan, exploiting inter-query parallelism on Tweets2013.

and examine every document in the document pool, while Lucene only examines a portion of the collection that only have query terms. As the collection grows, there are more documents to be checked by brute force scan approaches than Lucene and the benefits gained from brute force scan approaches cannot compensate the cost of disk access of more data.

3.3.4 Cross-over corpus size of Lucene vs. Brute Force Scans

Although brute force scan approaches even with multithread parallelism is not ideal for tweet search when the collection is large, we would still like to study the cross-over point for the corpus size when Lucene and brute force scan approaches achieve the same performance when exploiting parallelism at its fullest. This therefore sets a guideline for what architecture should be chosen for different sizes of corpus that achieves the best performance.

To create collections of different sizes, we first sort Tweets2013 chronologically, then start from the first tweet, and keep collecting tweets until 1%, 5%, 10%, 15%, 20%, 25%, 50% of the collection have been selected. This gives us different collections with size of 2.6, 13, 26, 39, 52, 65, 130 million tweets respectively. We then run the same experiments as that with Tweets2011 and record the results.

Results for multi-threaded performance exploiting intra-query parallelism on different corpus sizes are shown in Table 3.5. Here we report query latency of Lucene versus the brute force scan approaches with 95% confidence intervals. Each cell gives the the lowest latency for each technique out of all number of threads. So

for example, the cell in the first row and first column records the best performance of Lucene in terms of latency out of all possible threads when exploiting intra-query parallelism on a collection of 2.6 million tweets.

Looking at the table vertically, it is not surprising to see that each technique achieves a better performance when the collection is smaller. However, if we view the table horizontally, it is interesting to note that while none of our brute force scan approaches beat Lucene beyond collection size of 52 million(inclusive), when the collection size decreases, like in the case where we have 39 million tweets, AVXScan2, AVXScan3 and AVXScan4(in bold) achieve no significantly different performance compared to Lucene. For smaller collections with size of 26 and 13 million, Scan2 and AVXScan1 are also better options for achieving lower latency than Lucene. And lastly, Lucene achieves performance that is not anywhere close to any of our brute force scan approaches when the collection is as small as 2.6 million tweets.

To view the results clearly, we plot all the points from Table 3.5 in Figure 3.3. Again, brute force scan approaches gain better performance compared to Lucene when the collection gets smaller. Although more than one brute force scan approach outperforms Lucene in accomplishing a real-time search task, there're still some factors that should be taken into account as in which one should be chosen. Scan2 is favorable when there are restrictions on the memory as AVXScan approaches consume more memory in order for the document pool to be in align with the boundaries of AVX instructions. However, when latency is more concerned, AVXScan3 and AVXScan4 should be considered. Bottom line: brute force scan approaches are better options than Lucene when the collection size is under 39 million.

Collection Size (tweets in M)	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
2.6	25.5±1.0	16.9±0.9	11.4±0.2	11.1±0.4	11.1±0.3	7.9±0.1	7.0±0.1
13	48.1±1.7	54.1±0.3	35.4±0.6	28.8±0.4	27.8±0.8	26.4±0.4	26.1±0.3
26	67.1±1.8	95.5±1.0	64.1±0.4	54.0±0.3	53.3±0.5	50.5±0.4	47.9±0.4
39	**73.4 ± 0.9**	128.5±0.7	90.4±0.6	77.5±0.8	**75.0±1.1**	**73.2±0.5**	**73.3±0.9**
52	87.6±1.1	159.4±0.1	113.6±0.5	100.1±1.1	100.0±0.5	96.6±0.9	97.7±0.3
65	105.1±0.2	187.3±0.5	129.5±1.2	118.1±0.7	119.0±0.6	114.4±0.6	110.2±0.5
130	134.3±0.9	663.1±3.7	489.7±1.9	496.9±55.7	447.1±24.3	498.6±1.9	511.3±2.3
259	220.0±1.3	661.5±1.6	489.4±2.9	493.9±89.4	510.1±48.7	513.5±1.7	510.4±1.0

Table 3.5: Latency (in milliseconds) with 95% confidence intervals of Lucene versus brute force scan, exploiting inter-query parallelism on collections of different sizes selected from Tweets2013. Each cell gives the lowest latency for each technique out of all number of threads. Cross-over point is written in bold.

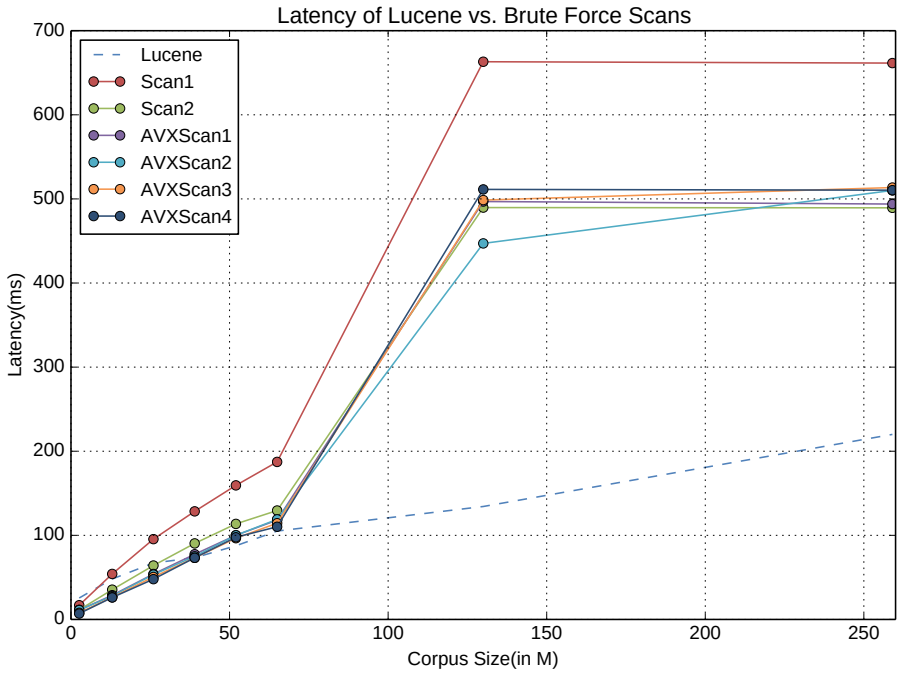

Figure 3.3: Latency(ms) of Lucene vs. Brute Force Scan approaches on different corpus sizes. We show the best performance out of all possible threads for each approach.

Similarly, results for multi-threaded performance exploiting inter-query parallelism on different corpus sizes are shown in Table 3.6 with the same layout as in Table 3.5. Looking at the table vertically, it is not surprising to see that each technique achieves a better performance when the collection is smaller. Looking at the table horizontally, we see that none of our brute force scan approaches beat Lucene beyond collection size of 26 million(inclusive), and this is proven by the results in Table 3.1 with Tweets2011, that is about 16 million tweets. When the collection size decreases, like in the case where we have 13 million tweets, AVXScan2, AVXScan3

Collection Size (tweets in M)	Lucene	Scan1	Scan2	AVXScan1	AVXScan2	AVXScan3	AVXScan4
2.6	193.1±7.84	160.8±0.72	284.8±9.31	330.0±10.00	333.9±22.66	564.1±81.99	539.7±43.58
13	**127.0±4.68**	35.3±0.27	62.3±1.06	101.3±2.54	**118.7±7.59**	135.8±15.48	146.3±7.78
26	98.9±2.32	17.5±0.05	30.9±0.31	47.6±2.18	57.3±1.63	59.2±4.37	68.2±2.24
39	72.8±0.49	11.7±0.04	20.7±0.21	30.6±0.49	33.4±1.31	37.4±2.63	44.1±4.26
52	59.3±1.74	8.8±0.01	15.5±0.07	23.5±2.04	25.4±1.98	27.7±2.27	33.4±6.58
65	48.0±0.12	7.0±0.02	12.4±0.07	17.5±0.89	18.3±0.69	20.9±1.00	24.0±2.76
130	25.1±0.11	3.5±0.01	6.2±0.06	8.4±0.27	8.7±0.28	9.7±0.26	10.6±0.89
259	8.5±0.05	1.7±0.00	2.9±0.03	3.1±0.14	3.5±0.17	4.2±0.28	4.1±0.32

Table 3.6: Throughput (query per second) with 95% confidence intervals of Lucene versus brute force scan, exploiting inter-query parallelism on collections of different sizes selected from Tweets2013. Each cell gives the best throughput for each technique out of all number of threads. Cross-over point is written in bold.

and AVXScan4(in bold) either achieve no significantly different performance or better performance compared to Lucene. For smaller collection with 2.6 million tweets, except for Scan1 that achieves throughput that is 17% lower than that of Lucene, all the other brute force scan approaches achieve performance that is significantly better than Lucene.

We also plot all the points from Table 3.5 in Figure 3.3. Same with intra-query parallelism, brute force scan approaches gain better performance compared to Lucene when the collection gets smaller. Although inter-query parallelism does not appear to be the best way to exploit multi cores using brute force scan techniques with Tweets2011 or Tweets2013, it is still to be proved to be an option to consider

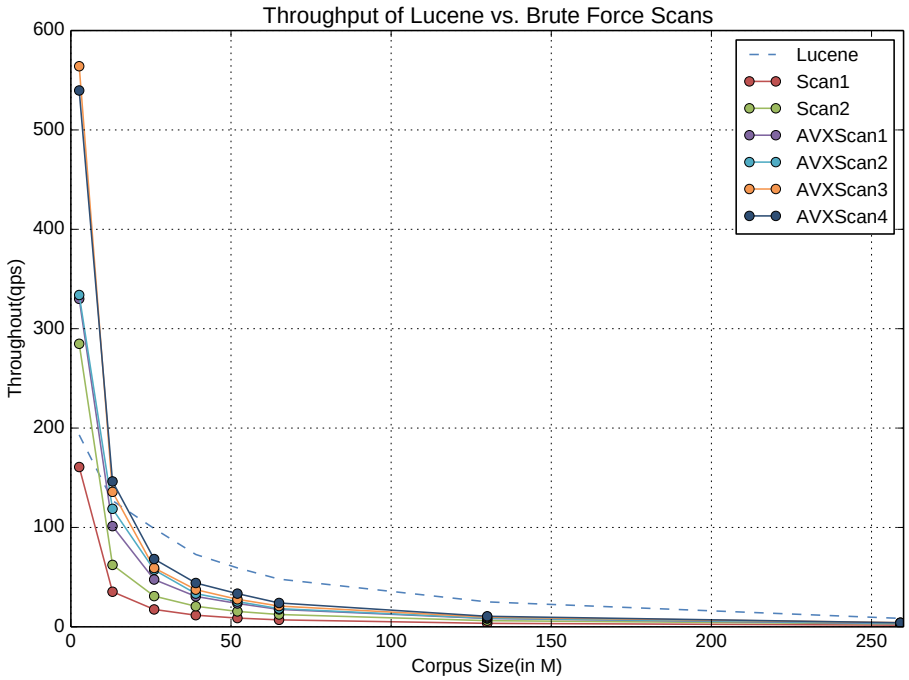

Figure 3.4: Throughput(qps) of Lucene vs. Brute Force Scan approaches on different corpus sizes. We show the best performance out of all possible threads for each approach.

when the collection is relative small. The bottom line is that brute force scan approaches are better options than Lucene when the collection size is under 13 million.

3.4 Discussion

From the above results, we have learned that our brute force scan approaches are better options than Lucene both in latency and throughput when the collection is relatively small. To recap, "indexing" with brute force scan approaches are 13

times faster than with inverted index in Lucene, and are better options than Lucene for retrieval when the collection size is under 39 million when intra-parallelism is exploited. When inter-parallelism is exploited, brute force scan approaches are better options than Lucene when the collection size is under 13 million.

One might notice and argue that optimizations like unrolling in our approaches can be automatically performed by a compiler. However, we argue that the answer is no. This is because all these optimizations require understanding the semantics of all the nested loops – loop over documents, loop over query, etc. In our case, the vectorization isn't a straightforward vectorization of an entire array, as one has to understand the semantics of document boundaries, etc. Thus, these optimizations can't be done automatically.

Compared to Lucene, our brute force scan approaches have three advantages in real-time search. First, real-time search requires fast incremental indexing, as documents are ingested in a streaming fashion at a high speed. It is necessary for new documents to be indexed and available for search immediately after they become available. This is not trivial for Lucene, but is relatively easy for brute force scans, as we need only to convert new documents to arrays of integers and append them to the end of the document pool.

Second, in a real-time search, users most often care only about the latest results. For an inverted index, this is implemented by traversing the posting lists "backwards" and exiting early when enough results have been accumulated. However, most search systems are not designed this way and traditional query evaluations are not trivial. For brute force scans, since documents are arranged chronologically,

we can simply scan from the end of the document pool, and exit when we have reached a proper time.

Third, simple data structure yields simple architecture, thus making it easier to be adopted to support other operations. For example, many web search engines adopt an architecture that begins with a simple ranker (e.g., BM25) followed by one or more complex (machine-learned) rankers. These complex rankers usually require access to document-level features (e.g., static priors), but such features are not stored in an inverted index, thus requiring a separate "document store". With brute force scan approaches, such "document stores" are not needed as we can suitably encode the document-level features and store them in the document pool.

To conclude, we show that our simple approach that abandons an inverted index and is based on brute force scans on documents has advantages over Lucene in a real-time scenario. Most importantly, it is most practical given today's hardware.

Chapter 4: Selective Search on Document Streams

In this chapter, we present the other main part of the real-time search components in our proposed system, selective search on document streams, see highlighted part in Figure 4.1. To recap, selective search is to divide a collection based on document similarity into multiple shards (shard creation), and select only a few shards that are estimated to be relevant to the query for search (shard selection). The motivation behind this approach is to reduce computation costs by considering only a subset of the documents while maintaining a high level of effectiveness. As discussed in Chapter 1, previous work on selective search assumes a static collection and performs partitioning on the collection in batches (e.g. using k-means clustering), and is therefore not suitable for handling dynamic collections like tweets where documents are arriving continuously so shards must be created incrementally. To solve this issue, we propose a novel selective search technique that differs from the traditional selective search in that shards are created incrementally. To achieve this, we articulate a design space, where we divide document streams into temporal segments at different granularities, and use either batch or online clustering algorithms to perform shard creation within each segment. We build inverted index for each shard within each temporal segment for query evaluation, and perform brute force scan

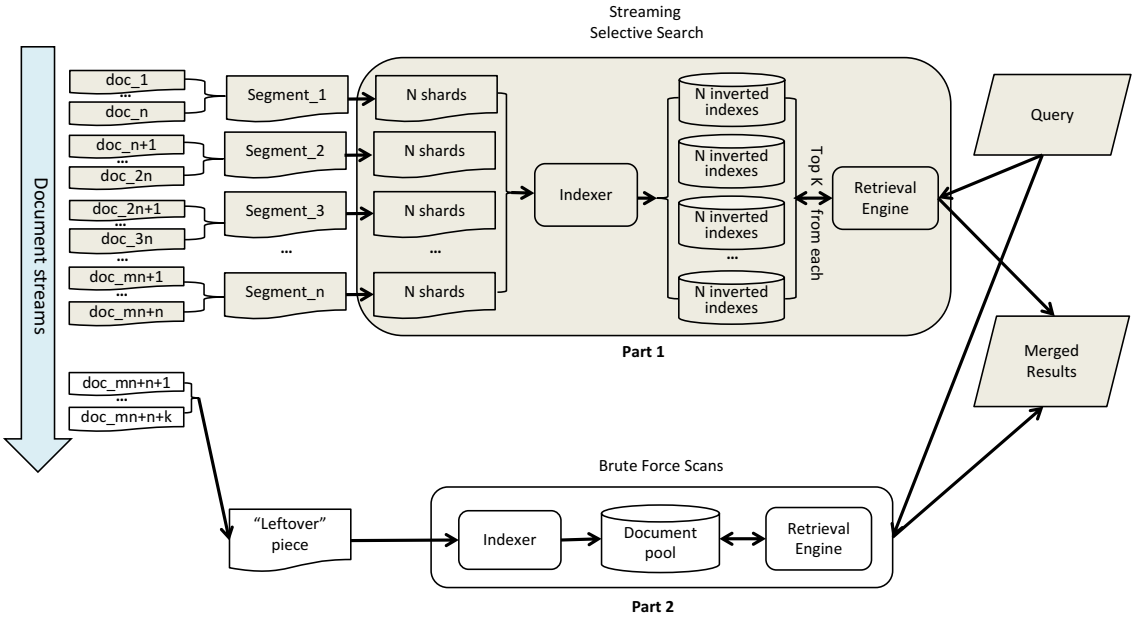

Figure 4.1: Selective search on document streams. Highlighted is the part for temporal segments.

over the "leftover" piece in the last incomplete segment to make the new incoming tweets immediately searchable, therefore developing a real-time search system.

Another problem with existing selective search approaches is that shard creation based on batch k-means clustering cannot support large datasets [4]. This is all due to the representation format of documents, where documents are converted into sparse vectors in huge vocabulary dimensions, computing distance between vectors and centroid of these vectors for k-means leads to extremely high computing cost and resources requirements. Even those methods that can support large datasets [2]

apply Sample-Based k-means to address this limitation, and run the risk of not being able to capture all the information a collection might contain. To solve this, we explore the feasibility of using word embeddings to reduce the dimensionality of the documents for clustering.

With the proposed selective search approach and the utilization of word embeddings, our goal is to minimize the number of documents for consideration to reduce computation efforts while achieving some level of results quality, hence the efficiency/effectiveness tradeoff. To study this, we use the data from TREC Microblog evaluations to do experiments, and observe that with cost that is 12% of an exhaustive search, we can achieve effectiveness roughly equal to that of an exhaustive search.

Since we have explained the strategy of brute force scans in detail in the previous Chapter, this chapter focuses mainly on describing how we achieve selective search on document streams. In the sections that follow, we first discuss in detail how to achieve incremental partitioning over collections by applying different segmenting strategies and clustering algorithms. Then we describe the preparation for the clustering, namely how to represent tweets as vectors with word embeddings. This is followed by the evaluation methodologies we adopted and experiments we ran to study the efficiency and effectiveness of selective search techniques. Finally, we conclude the work.

4.1 Real-Time Selective Search

Before we present our selective search approach, let us first define our problem: Given a stream of timestamped documents and a query Q at time t, the system should aim to return a ranked list of documents up until the query time. To study the efficiency/effectiveness tradeoff for selective search and to show a fair comparison to an exhaustive search, we apply the same ranking algorithm for both strategies. To make it simple, we rank documents using query likelihood with Dirichlet smoothing [41].

4.1.1 Design Space

Current selective search approaches assume static collections. They typically partition the collection into topical shards where documents within each shard share similar content. Within each shard, an inverted index is built for all the documents in that shard. For retrieval, they search against the inverted index for only the shards that are estimated to contain the most relevant documents, each selected shard returns relevant documents which are then merged into a final result list.

Talking about the technical details, as for partitioning, simple techniques like k-means clustering and sophisticated approaches based on topic modeling have been proposed, but it is not clear which one is more effective. This is supported by the research of Kulkarni and Callan [5], in which the researchers compared k-means and LDA but found no significant difference in effectiveness. Therefore, we use k-means clustering as the partitioning technique in our selective search for simplicity. As

shard and cluster refer to the same thing, we will use these two terms interchangeably.

Because existing k-means clustering techniques can handle data in batch but not in streaming setting, in order to adapt the k-means (e.g. Lloyd's algorithm) to our real-time streaming scenario, we divide the collection into temporal segments, and run batch k-means for each segment. More specifically, starting from the beginning, for every hour we accumulate documents until that hour completes, then apply a selective search on this hourly segment. Hence, every temporal segment would have N clusters after partitioning, and only a subset of these clusters will be considered for query evaluation. And because batch k-means is performed every hour, the search is somewhat delayed, and for now the newly encountered tweets are not searchable until after one hour when the clustering has been performed.

We can also modify the segment organizations for larger intervals to avoid searching multiple small segments, say perform selective search for every day. We still perform selective searches on hourly segments back to the previous complete day, and perform selective searches on daily segments back in time. Having even larger intervals brings us to a stage where the setting looks closer to selective search on a static collection.

In order to have contrasting experiments, we also consider an online variant of k-means clustering for shard creation. Online clustering has the advantage of updating clusters incrementally with every new instance, so they can evolve over time to be shifted to topics. However, one property of current online clustering algorithms [163–166] makes it difficult to apply these algorithms directly to our ap-

proach: they only keep track of cluster representatives, but not cluster assignments. That is to say, for every new instance, the algorithm will compute its nearest cluster and clusters will then be updated. Therefore, while it is possible to compute cluster assignments at a designated time when needed, computing cluster assignments every time a new instance arises is computationally unfeasible. This is in contrast to batch k-means, where the output is both cluster representatives and cluster assignments, and they are stable once the algorithm converges.

To solve the cluster assignment problem of online clustering, one obvious solution is to buffer the documents of which we would like to determine cluster assignments, then perform a second pass over the clusters to compute their nearest clusters. In our search scenario, this means performing online k-means over an hour of data, buffering these, and scanning clusters to perform cluster assignment.

See Figure 4.2 for an illustration for our selective search architecture.

4.1.2 Segment Organization

As previously discussed, we explore a variety of options for organizing segments and performing clustering. To summarize, there are a total of four different approaches. A discussion of each and the pros and cons of each attempt follows:

Hourly batch ($\mathbf{H}_B$). In this approach, we perform a selective search on hourly segments with batch clustering, from the most recent complete hour back to the beginning of the collection.

Hourly online ($\mathbf{H}_O$). This is exactly the same as the first approach except that

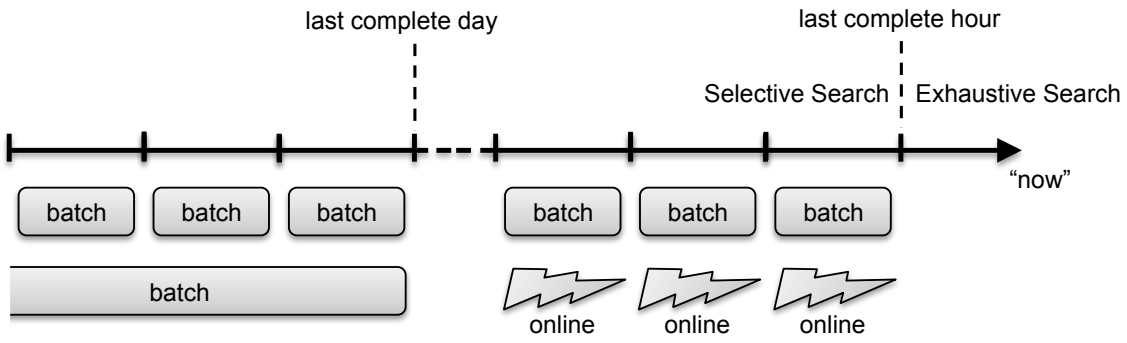

Figure 4.2: Schematic illustration of the design space for real-time selective search on document streams. We can apply batch or online k-means at hourly intervals and batch k-means at a coarser granularity (e.g., a day). At query time, results can be assembled from a combination of different segments.

hourly segments are partitioned using an online clustering algorithm.

Hourly batch + daily batch ($\mathbf{H}_B + D_B$). In this approach, we perform selective searches on daily segments using batch clustering until the most recent complete day. We then perform selective searches on hourly segments using batch clustering until the most recent complete hour.

Hourly online + daily batch ($\mathbf{H}_O + D_B$). This approach is the same as the above approach except that the hourly segments are partitioned using an online clustering algorithm. Note that the daily segments are constructed using batch clustering as there is no advantage to applying online clustering on such long intervals.

To show the considerations we have taken in choosing different segment orga-

nizations and clustering algorithms, below is a discussion of their advantages and disadvantages:

Batch vs. online clustering. As far as algorithm efficiency is concerned, online clustering must be performed a second pass for assignment computation, and it involves linear time complexity that is proportional to the collection size and dimension size. Whereas batch clustering typically has linear time complexity as well, it is proportional to the collection size, dimension size, cluster number and number of iterations until convergence, therefore is slower than online clustering. However, the complexity for both algorithms is not a problem as the segment to which we apply the clustering is in small interval (one hour or one day) and generally has a small number of documents compared to the entire collection. Therefore, it can fit into the memory.

For cluster quality, it is not clear which algorithm generates better clusters, in the sense that it groups more contextual similar tweets together and that by selecting top K clusters out of N clusters, it contains relevant tweets with higher effectiveness. This is hard to predict using only theory because both algorithms have their advantages. Batch clustering can take advantage of a broader range of information, whereas clusters of online clustering are sensitive to every instance, and every instance can have an effect on the update of the clusters. On the other hand, while good batch clustering depends largely on a good seeding process, online clustering can be adapted gradually to shift to actual topics represented by the streams. Therefore, it is up to us to determine which method is better.

Fine-grained vs. coarse-grained. Another variation of our segment organizations involves the method by which we divide the streaming collection. In terms of efficiency, coarse-grained (performing selective searches for each day) segmentation is more efficient as it has fewer segments. Thus, because we need to perform shard selection for each segment for the search, there are a fewer of them in total. Besides, as we need to perform clustering for each segment, fewer segments means less overhead in the clustering process.

However, for search quality, fine-grained segmentation enables us to search on a more fine-grained level and can yield higher quality. If it is not clear for hourly segmentation, let us take an extreme case where we segment the collection by every tweet. In this case, we are taking every tweet into consideration for relevance, unlike in other cases where we have only considered a subset of the collection. Therefore, a search on such an extreme fine-grained segmentation leads to higher effectiveness. Another advantage of fine-grained segmentation is that we can terminate the process early once we have accumulated enough results.

4.1.3 Indexing

As with indexing, for all four segmentation strategies, within each temporal segment, we build an inverted index for each shard, with global statistics stored in it. Selective search can then be performed on the already partitioned segments. This is shown in part 1 in Figure 4.1.

Yet there is still a "leftover" piece that must be addressed; the tweets from

	Tweets2011	Tweets2013
Documents per hour (in K)	40	170

Table 4.1: Number of documents per hour for Tweets2011 and Tweets2013

the last incomplete hour. For example, if a query is issued 15 minutes past the last hour, we have the partitions for every hour or day back until the beginning time according to which strategy we take, but the tweets posted in that 15-minute window remain unclustered. For this piece of data, we will generate document pool and perform exhaustive search with brute force scans techniques. The results are merged with the results returned from selective search on the partitioned segments to generate a final list of results. This is shown in part 2 in Figure 4.1.

The reason of choosing brute force scans on the "leftover" piece is that: as it is shown in Chapter 3 that brute force scan approaches achieve better performance both in latency and throughput than Lucene when exploiting parallelism when the collection size is under 2.6 million. Table 4.1 shows that the average number of documents per hour is 40K for Tweets2011 and 170K for Tweets2013, they're both far smaller than 2.6 million. Therefore, brute force scan approaches are better options than Lucene in performing an exhaustive search for the tweets in the last incomplete hour.

4.1.4 Clustering Implementations

For k-means clustering and its variations used in current selective search approaches, documents are converted into vectors of vocabulary space. Vectors can be in tremendously high dimensions either due to a large dataset or big vocabulary size. This is the case for tweet streaming. Although it was our initial goal to apply batch and online clustering to such vectors, the high computation cost prevented us from doing so. To provide a fair comparison, in Ackermann et al. [163] StreamKM++ paper, the largest dataset used was 11 million in 57 dimensions, with the largest number of dimensions being 68. In contrast, our data (16 million tweets) already has a vocabulary size that exceeds one million. Although for batch clustering, it is possible to reduce the vocabulary space by discarding all the terms whose term frequencies fall below a certain threshold. However, this is not true for online clustering, where both vocabulary and term frequencies are updated in real-time.

To further explain the need for dimensionality reduction, for k means clustering, the issue is that the centroid vectors have features that are the union of all documents in the cluster. After averaging, every word that is found in at least one document in the cluster will have non-zero weight. Therefore, the centroid vectors become very large, i.e., non-sparse. In batch clustering, this isn't an issue because we can use disk. In contrast, in streaming clustering, all the vectors can't fit in memory. With word embeddings being in use, this isn't an issue any more because the centroid vectors are capped at a maximum size (i.e., the number of dimensions of the embedding).

To reduce the vector dimensionality of the documents, we take advantage of word embeddings from recent work [6], and project each document into a vector of dimension d (d is in the magnitude of hundreds or less). It works as follows: We first train GloVe word embeddings of Pennington et al. [163] on the Edinburgh tweet collection [167]. This corpus comprises 97 million tweets from November 11th 2009 to February 1st 2010. We deliberately chose a corpus that is disjointed from our test collection, so as to make sure that any result generated by GloVe and used by our test collection does not have any future information our test collection might contain. GloVe produces a global log-bilinear regression model that combines the advantages of two model families for continuous word representations: global matrix factorization and local context methods. The output of it is real-value vectors for each word occurred in the corpus in dimension d:

The document vector then simply takes the average of all of the individual word vectors. Here, we do take the term frequency into account, so if a term occurs twice, it is counted twice in the computation. All vocabulary terms are treated as vectors of zeroes.

For batch clustering, we used k-means implemented in Apache Spark's Machine Learning Library (MLlib) [168] that implements a parallelized variant of the k-means++ method [169] called kmeans|| [170]. It is an efficient parallel version of the inherently sequential k-means++ that reduces the number of passes needed to obtain a good initialization while obtaining a nearly optimal solution.

For online clustering, we used StreamKM++ [163] implementation written in Java from MOA toolkit [171]. To recap, StreamKM++ maintains a coreset, each

of which represents a weighted point from the data stream. Such corset is designed with a tree structure for fast operation. Whenever a new instance arises, the coreset updates itself to include the new point. Therefore, at any time, we have cluster representatives but not assignments. After every hour, we must perform a second pass over these representatives to compute cluster assignments.

Before running k-means clustering, it is up to us to define the number of clusters we want to have for each segment. To trade between efficiency and cluster quality as discussed in the previous section for fine-grained and coarse-grained segmentation, we choose a middle ground number 100 for simplicity as the number of clusters to be generated throughout all segments for all four segmenting strategies. For example, for $H_O + D_B$, every daily segment is partitioned into 100 clusters with batch clustering, and every hourly segment is partitioned into clusters with online clustering.

Until now, we have introduced the manner in which the shard creation is handled with different segment organization strategies and clustering algorithms. In the following section, we will demonstrate how a subset of the collection is selected for a search and how the query evaluation is performed in such a scenario.

4.1.5 Cluster Selection and Document Ranking

Given the partitions for every segment, our goal of cluster selection is to select the clusters from each segment that are most likely to contain relevant documents with respect to a query. Those selected will be put through a ranking algorithm

and results will then be merged to generate a final ranked list of results.

For cluster selection, we adopted a simple technique based on cosine similarity. In this approach, the query is converted into a vector as the document does, and compared to all 100-cluster representatives within each segment. Clusters are then ranked based on cosine similarity. As a comparison, we also implemented a more complex algorithm ReDDE [101], but did not find significant improvement. While ReDDE requires a sample index, for a cosine similarity approach, we need only to maintain 100 dense vectors. Hence, we remain with cosine similarity for simplicity and efficiency.

As for how many clusters are selected within each segment, we define a global variable to represent this number. Instead of using sophisticated techniques to automatically compute the smallest number of clusters containing the most relevant documents based on some criteria, we select the same number of clusters within every segment throughout the collection for all four-segment organizations. For example, for $H_B + D_B$, if we select the top 10 clusters for every day, we also select top 10 clusters for every hour. Of course there are more effective approaches, but we leave this as a future plan and continue to use the simple approach for now.

Once the clusters are selected, we rank the documents for each selected cluster using query likelihood with Dirichlet smoothing. We then merge the results to generate a final ranked list. For every query, we are careful to use the collection statistics available at query time when computing document rankings. That is, we have a dynamic dictionary that maps terms to their collection frequencies in real-time, and we use the collection frequency of a term up until the query time when

computing query likelihood. Therefore, we avoid using information from the future, although previous work has shown that this does not affect the effectiveness [172].

4.2 Experimental Setup

For our experiments, we used test collection known as Tweets2011 from TREC 2011, TREC 2012, consisting of 16 million tweets, a sample of approximately 1% of tweets from January 23, 2011 to February 7, 2011 (inclusive). There are 50 topics for TREC 2011 and 60 topics for TREC 2012. Each topic consists of a query with a timestamp, indicating when the query is issued. Our system's task is to return a ranked list of results in response to each query, where all results are before the query time. To learn the scalability of our approach, same experiments are run on a larger test collection from the Microblog tracks at TREC 2013 [162] called the Tweets2013 collection, which consists of sample of tweets from February 1, 2013 to March 31, 2013 (inclusive), totaling approximately 259 million tweets. There are 60 topics for TREC 2013.

We implement hourly batch (H_B), hourly online (H_O), hourly batch + daily batch ($H_B + D_B$), and hourly online + daily batch ($H_O + D_B$) on both Tweets2011 and Tweets2013 as discussed in previous sections. For each strategy, we built an inverted index for each shard created for each temporal segment. Query evaluation is then performed against these inverted indexes for all the temporal segments up till the last complete hour/day. For the "leftover" piece in the last incomplete hour, we run brute force scan by exploiting parallelism to achieve lower latency.

4.3 Evaluation Methodology

To evaluate system effectiveness, we follow the evaluation methodology introduced in Chapter 1. To gather "ground truth", we had NIST assessors use a standard pooling strategy to use the depth 100 tweets from all submitted runs in TREC, and judge the tweets on a three-way scale of "not relevant", "relevant", and "highly relevant" with the score of 0, 1 an 2 respectively. The relevant and highly relevant tweets are all considered relevant. To measure effectiveness, we use official TREC evaluation metrics, namely precision at rank 30 (P30) and average precision (AP) at rank 1000. Here, higher metric scores indicate better system effectiveness.

To study the efficiency/effectiveness of our proposed four selective search approaches compared with an exhaustive search, we must first answer a question: effectiveness is easy is record, but how do we measure efficiency? In our experiments, recording time to measure efficiency is a bad solution because it is unstable. A more desirable solution would be to record the number of tweets that are examined from selected clusters. However, it is not a fair comparison to use this number when comparing different queries. Because these queries are issued at different times, even if we select the same number of clusters, some queries might have more tweets examined than other queries. Also, different cluster sizes make it unreasonable to use this number. For aggregation purposes and to solve this issue, we normalize this number into a fraction of the collection size that would have been examined before the query time.

Specifically, there are two types of statistics we are interested in: first is the

individual level. That is, if we select n clusters within each segment (this can then be operationalized as a fraction of the collection), what level of effectiveness can we achieve? The other is in an aggregation level. If we examine a particular fraction of the collection, what level of effectiveness can we achieve? For this, we are still analyzing the cluster level because each number of clusters selected would result in a pair of efficiency/effectiveness values. Therefore, we would bucket the efficiency values and average effectiveness across all points that fall into that bucket. Note that the efficiency/effectiveness values are already the average over all topics, so we make sure we are taking all topics into account.

To avoid the effects a random selection of k-means might bring, we repeat the above experiments five times and take the average over all trials.

4.4 Results

4.4.1 Segment Organizations

To study the first question from the previous section, we vary the number of clusters examined per segment for each of the four segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch ($H_B + D_B$), and hourly online + daily batch ($H_O + D_B$) and record efficiency/effectiveness compared with that of an exhaustive search. For these experiments, we used word embeddings of 25 dimensions.

We show the results in Table 4.2. Due to limited space, we show that only 2, 3, 4, 5, 10 and 20 clusters are represented. The first row of each block shows

the effectiveness of an exhaustive search that ranks all the tweets prior to the query time. For each strategy, we show the average effectiveness over five trials as well as the 95% confidence interval. We see that the confidence interval is relatively small, indicating that our selective search is robust for a random seeding of the clustering process. We also did statistical significance testing using a paired t-test between each of our approaches with an exhaustive search. The symbols $\blacktriangledown$ and $\blacktriangle$ show significant differences at $p < 0.05$.

Looking at P30 for each strategy vertically, we see that by selecting three or four clusters out of 100, we can achieve effectiveness indistinguishable from that of an exhaustive search. By selecting more clusters, it is even possible to achieve significantly better effectiveness than an exhaustive search. The explanation behind it is that it's the cluster hypothesis at work – documents close to other relevant docs are more likely to be relevant, and false positives are being excluded because they landed in partitions that don't get selected. This finding is consistent with previous work [173]. This indicates that our clustering algorithm does a good job in grouping close tweets together into one cluster while leaving any other noise outside. On the other hand, looking at AP, in order to be comparable to an exhaustive search, we must examine approximately 10 clusters per segment. And this is because AP considers recall as well.

Table 4.3 shows a translation of the number of clusters into a fraction of the collection. Results are also computed on an average of five trials. Since the 95% confidence interval is less than 0.005, we can simply omit these from the table.

To look at the statistics in an aggregated level, we follow the bucket procedure

P30	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.3182			
2	0.2954±0.0082 ▼	0.3012±0.0046 ▼	0.2972±0.0037 ▼	0.2982±0.0033 ▼
3	0.3134±0.0078	0.3172±0.0029	0.3132±0.0132	0.3144±0.0104
4	0.3241±0.0068	0.3270±0.0065	0.3209±0.0046	0.3218±0.0038
5	0.3255±0.0048 ▲	0.3282±0.0060 ▲	0.3235±0.0080	0.3234±0.0051
10	0.3252±0.0016 ▲	0.3265±0.0043 ▲	0.3235±0.0025 ▲	0.3229±0.0030 ▲
20	0.3244±0.0018 ▲	0.3245±0.0011 ▲	0.3238±0.0010 ▲	0.3234±0.0009 ▲
AP	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.2368			
2	0.1608±0.0025 ▼	0.1671±0.0085 ▼	0.1592±0.0074 ▼	0.1614±0.0078 ▼
3	0.1861±0.0051 ▼	0.1928±0.0026 ▼	0.1865±0.0091 ▼	0.1874±0.0062 ▼
4	0.2039±0.0061 ▼	0.2086±0.0044 ▼	0.2039±0.0063 ▼	0.2054±0.0045 ▼
5	0.2150±0.0055 ▼	0.2181±0.0026 ▼	0.2144±0.0034 ▼	0.2156±0.0028 ▼
10	0.2379±0.0047	0.2415±0.0017 ▲	0.2390±0.0041	0.2396±0.0043
20	0.2467±0.0009 ▲	0.2474±0.0015 ▲	0.2470±0.0010 ▲	0.2468±0.0003 ▲

Table 4.2: P30 and AP scores for different numbers of clusters examined under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), hourly online + daily batch (H_O+D_B) on Tweets2011. ▼/▲ indicate significant differences compared to exhaustive search ($p < 0.05$).

Clusters	H_B	H_O	H_B+D_B	H_O+D_B
2	0.021	0.024	0.021	0.022
3	0.032	0.035	0.033	0.033
4	0.044	0.047	0.045	0.045
5	0.056	0.594	0.057	0.057
10	0.115	0.121	0.117	0.117
20	0.231	0.242	0.235	0.236

Table 4.3: Number of clusters examined for each of our segment organizations, translated into a fraction of the entire collection on Tweets2011.

and plot the effectiveness against the fraction of the collection for each of the segment organizations in Figure 4.3 with a bucket size of 0.2. The horizontal line shows the effectiveness with an exhaustive search. For both AP and P30, there is a significant difference between our approach and an exhaustive search at the beginning, which is in line with the previous findings in Table 4.2. As we proceed all the way to the right, the effectiveness of selecting a fraction of 1 of the collection finally converges to an exhaustive search.

We select a few representative points from Figure 4.3 and show them in Table 4.4. This table is laid out in the same format as Table 4.2, except that the first column is the fraction bucket. As before, we average the effectiveness scores over five trials and show the 95% confidence interval. Also, we annotate the results of significant testing in the same manner. Again, the small confidence interval suggests

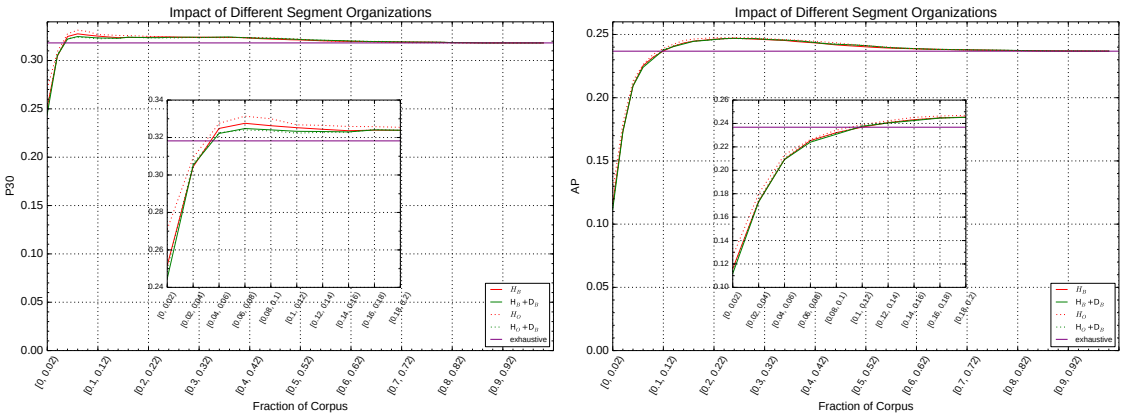

Figure 4.3: Effectiveness (P30 and AP) vs. efficiency (fraction of the collection examined) under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), and hourly online + daily batch (H_O+D_B) on Tweets2011.

that our selective search approaches are robust to the randomization of clustering.

Looking at all tables horizontally, that is, to compare the four segment organizations under same setting, we do not observe significant differences. All four strategies are able to achieve the same level of effectiveness. Choosing any one of them is fine, but some are preferred over others when there are specific requirements. For example, if the resource is limited, we might want to choose online clustering with StreamKM++ as that only requires one single server whereas batch clustering with MLlib requires several servers. When the queries are not issued frequently, it is preferred to apply selective search over daily segments as that is computationally efficient because it reduces the number of cluster selections required over the

P30	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.3182			
$[0.02, 0.04)$	0.3044±0.0273	0.3092±0.0228	0.3052±0.0260	0.3063±0.0250
$[0.04, 0.06)$	0.3248±0.0086	0.3275±0.0089	0.3222±0.0099	0.3226±0.0067
$[0.10, 0.12)$	0.3253±0.0033 ▲	0.3269±0.0028 ▲	0.3233±0.0031 ▲	0.3223±0.0041
$[0.20, 0.22)$	0.3242±0.0027 ▲	0.3248±0.0031 ▲	0.3236±0.0014 ▲	0.3235±0.0011 ▲
AP	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.2368			
$[0.02, 0.04)$	0.1735±0.0355 ▼	0.1799±0.0367 ▼	0.1729±0.0395 ▼	0.1744±0.0373 ▼
$[0.04, 0.06)$	0.2095±0.0173 ▼	0.2126±0.0128 ▼	0.2091±0.0163 ▼	0.2105±0.0115 ▼
$[0.10, 0.12)$	0.2369±0.0071	0.2395±0.0032	0.2376±0.0074	0.2381±0.0085
$[0.20, 0.22)$	0.2462±0.0012 ▲	0.2476±0.0029 ▲	0.2461±0.0017 ▲	0.2459±0.0017 ▲

Table 4.4: Selected efficiency operating points from Figure 4.3. ▼/▲ indicate significant differences compared to exhaustive search ($p < 0.05$).

segments.

The results shown in Figure 4.3 assume that the cost of each segment organization is the total number of documents from selected shards. Although this is true for a brute force scan approach where all the documents from the selected shards need to be examined, it is not applicable for query evaluation that is with an architecture with inverted index as its core. Query evaluation with inverted index only examines documents that contain at least query terms and therefore needs a different efficiency vs. effectiveness analysis. We take the cost metric defined by Kulkarni et al. [104] that is the average number of documents from selected shards that contain at least one query term over all queries, as is defined as follows:

$$C(q) = \sum_{i=1}^{n} |S_i(q)| \tag{4.1}$$

where, n is the number of top shards searched for query q, $S_i(q)$ is the set of documents in shard i that contain at least one query term. Other cost metrics like the one proposed by Aly et al. [108] reflect the selection algorithm in the efficiency measure arguing that the selection algorithm itself can require substantial resources, however, as our selection algorithm is nothing complex but just a ranking process for different cosine similarity scores between each selected shard and the query, the cost can be ignored.

We show the average effectiveness over five trials vs. cost associated with an inverted index under different segment organizations in Figure 4.4 with the same layout as that of Figure 4.3. Instead of showing the absolute effectiveness score(P30 and AP) on the y-axis, we show the relative score, that is the fraction of the effective-

	Tweets2011	Tweets2013
Cost(in M)	0.2	5.5

Table 4.5: Exhaustive search cost on different collections

ness score compared to an exhaustive search. Instead of showing the absolute cost on the x-axis, we show the fraction of the cost compared to an exhaustive search. Here, the cost of an exhaustive search, or the number of documents containing at least query term for all the topics in the entire collection is shown in Table 4.5, that is 0.2 million. Looking at the figure, we see that there is no significantly different performance with different segment organizations. Our approaches can achieve the same P30 score compared to exhaustive search but with only about 12% of the cost. Again, in order to achieve AP score that is comparable with exhaustive search, the cost increases to about 34% because recall has to be taken into consideration. Finally, there is this zigzag shape for our approaches, and this is because there are not as many points falling into certain buckets as in other buckets, and there is noise for buckets with fewer points.

We select a few representative points from Figure 4.4 and show them in Table 4.6. Different from the layout of Table 4.2, we didn't show the fraction of effectiveness for exhaustive search as it's simply 1. Again, the small confidence interval suggests that our selective search approaches are robust to the randomization of clustering. And looking at all tables horizontally, we do not observe significant

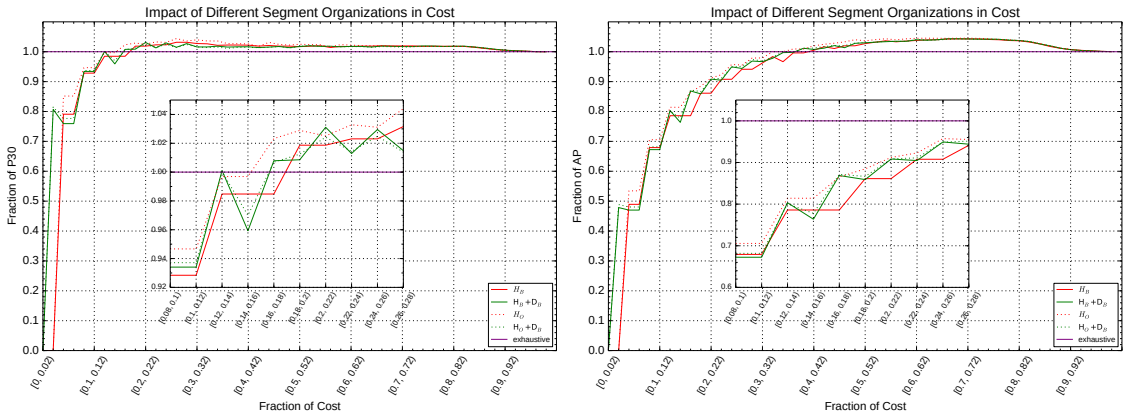

Figure 4.4: Effectiveness (fraction of P30 and AP) vs. cost (fraction of the collection examined) under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), and hourly online + daily batch (H_O+D_B) on Tweets2011.

differences. All four strategies are able to achieve the same level of effectiveness.

We run the same experiments and perform the same analysis on Tweets2013. Table 4.7 shows P30 and AP scores under different segment organizations when different numbers of clusters are examined. We see that the confidence interval is relatively small, indicating that our selective search is robust for a random seeding of the clustering process even for larger collections.

Looking at P30 for each strategy vertically, we see that by selecting ten clusters out of 100, we can achieve effectiveness indistinguishable from that of an exhaustive search. This is worse than that with Tweet2011, which achieves same P30 as that of an exhaustive search by examining only about three or four clusters. And this is an

Fraction of P30	H_B	H_O	H_B+D_B	H_O+D_B
$[0.06, 0.08)$	0.7909±0.0287	0.8517±0.0227	0.7594±0.0409	0.7767±0.0482
$[0.08, 0.10)$	0.9284±0.0259	0.9467±0.0144	0.9341±0.0115	0.9371±0.0105
$[0.10, 0.12)$	0.9284±0.0259	0.9467±0.0144	0.9341±0.0115	0.9371±0.0105
$[0.12, 0.14)$	0.9848±0.0245	0.9969±0.0091	1.0010±0.0093	1.0003±0.0126
$[0.38, 0.40)$	1.0224±0.0091	1.0261±0.0134	1.0165±0.0040	1.0146±0.0040
Fraction of AP	H_B	H_O	H_B+D_B	H_O+D_B
$[0.06, 0.08)$	0.4894±0.0437	0.5345±0.0299	0.4701±0.0689	0.4805±0.0683
$[0.08, 0.10)$	0.6792±0.0104	0.5345±0.0299	0.6724±0.0311	0.6818±0.0327
$[0.10, 0.12)$	0.6792±0.0104	0.7056±0.0360	0.6724±0.0311	0.6818±0.0327
$[0.12, 0.14)$	0.7859±0.0217	0.8141±0.0110	0.8034±0.0048	0.8005±0.0081
$[0.38, 0.40)$	1.0048±0.0198	1.0197±0.0072	1.0116±0.0112	1.0114±0.0240

Table 4.6: Selected cost operating points from Figure 4.4.

interesting observation that needs further investigation. However, ten clusters out of 100 is still a reasonably good baseline. Looking at AP, in order to be comparable to an exhaustive search, we must examine approximately 13 clusters per segment. This is slightly worse than that with Tweets2011.

Table 4.8 shows a translation of the number of clusters into a fraction of the collection. Results are also computed on an average of five trials. We omit the 95% confidence interval from the table as it's less than 0.005.

We follow the bucket procedure and plot the effectiveness against the fraction of the collection for each of the segment organizations in Figure 4.5 on Tweets2013.

P30	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.3722			
4	0.3349±0.0031 ▼	0.3366±0.0105 ▼	0.3362±0.0125 ▼	0.3368±0.0126 ▼
6	0.3587±0.0080 ▼	0.3547±0.0038 ▼	0.3597±0.0023 ▼	0.3598±0.0020 ▼
10	0.3733±0.0060	0.3684±0.0052	0.3730±0.0086	0.3733±0.0090
13	0.3785±0.0033 ▲	0.3780±0.0058 ▲	0.3794±0.0080 ▲	0.3796±0.0084 ▼
16	0.3805±0.0029 ▲	0.3797±0.0035 ▲	0.3797±0.0041 ▲	0.3797±0.0041 ▲
20	0.3823±0.0031 ▲	0.3808±0.0009 ▲	0.3814±0.0016 ▲	0.3814±0.0016 ▲
AP	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.1948			
4	0.1472±0.0034 ▼	0.1482±0.0043 ▼	0.1484±0.0057 ▼	0.1488±0.0058 ▼
6	0.1654±0.0064 ▼	0.1655±0.0022 ▼	0.1668±0.0049 ▼	0.1669±0.0048 ▼
10	0.1858±0.0035 ▼	0.1832±0.0035 ▼	0.1848±0.0052 ▼	0.1849±0.0054 ▼
13	0.1926±0.0013	0.1917±0.0033	0.1916±0.0039	0.1916±0.0040
16	0.1966±0.0013 ▲	0.1959±0.0019	0.1950±0.0010	0.1950±0.0010
20	0.1996±0.0018 ▲	0.1992±0.0009 ▲	0.1978±0.0025 ▲	0.1979±0.0025 ▲

Table 4.7: P30 and AP scores for different numbers of clusters examined under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), hourly online + daily batch (H_O+D_B). ▼/▲ indicate significant differences compared to exhaustive search ($p < 0.05$) on Tweets2013.

Clusters	H_B	H_O	H_B+D_B	H_O+D_B
4	0.047	0.046	0.047	0.047
6	0.070	0.069	0.071	0.071
10	0.118	0.115	0.120	0.120
13	0.154	0.151	0.156	0.156
16	0.188	0.185	0.191	0.191
20	0.234	0.231	0.237	0.237

Table 4.8: Number of clusters examined for each of our segment organizations, translated into a fraction of the entire collection on Tweets2013.

We then select a few representative points from Figure 4.5 and show them in Table 4.9. The small confidence interval suggests that our selective search approaches are robust to the randomization of clustering. No significant differences are observed for the four segment organizations under same setting. All four strategies are able to achieve the same level of effectiveness.

We show the average effectiveness over five trials vs. cost associated with an inverted index under different segment organizations in Figure 4.6. Here, the cost of exhaustive search, or the number of documents containing at least one query term for all the topics in the entire Tweets2013 collection is shown in Table 4.5, that is 5.5 million.

Looking at the figure, we see that there is no significantly different performance with different segment organizations. Our approaches can achieve the same P30

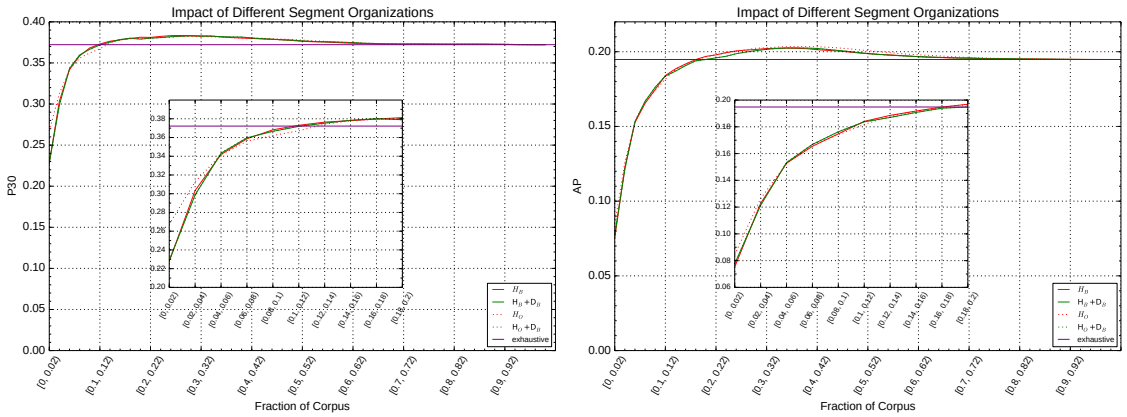

Figure 4.5: Effectiveness (P30 and AP) vs. efficiency (fraction of the collection examined) under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), and hourly online + daily batch (H_O+D_B) on Tweets2013.

score compared to exhaustive search but with about 34% of the cost, that is more cost compared with Tweets2011, and this observation is in line with the effectiveness vs. efficiency tradeoff between Tweets2013 and Tweets2011. To view this figure clearly, we select a few representative points from Figure 4.6 and show them in Table 4.10.

4.4.2 Distribution of Cluster Sizes

Due to the nature of clustering algorithm, it is not guaranteed that the cluster sizes are equal. And one might wonder that whether that would affect efficiency. For example, if we have one cluster that is significantly larger than other clusters,

P30	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.3722			
[0.04, 0.06)	0.3420±0.0144 ▼	0.3416±0.0130 ▼	0.3432±0.0176 ▼	0.3431±0.0174 ▼
[0.10, 0.12)	0.3730±0.0057	0.3672±0.0055	0.3719±0.0074	0.3721±0.0081
[0.14, 0.16)	0.3780±0.0036 ▲	0.3780±0.0058 ▲	0.3784±0.0082	0.3786±0.0087
[0.24, 0.26)	0.3832±0.0021 ▲	0.3811±0.0017 ▲	0.3819±0.0019 ▲	0.3819±0.0019 ▲
AP	H_B	H_O	H_B+D_B	H_O+D_B
exhaustive	0.1948			
[0.04, 0.06)	0.1528±0.0120 ▼	0.1527±0.0094 ▼	0.1533±0.0110 ▼	0.1535±0.0109 ▼
[0.10, 0.12)	0.1841±0.0047 ▼	0.1817±0.0047 ▼	0.1837±0.0057 ▼	0.1838±0.0059 ▼
[0.14, 0.16)	0.1918±0.0041	0.1917±0.0033	0.1907±0.0043	0.1908±0.0045
[0.24, 0.26)	0.2006±0.0021 ▲	0.2003±0.0013 ▲	0.1987±0.0021 ▲	0.1987±0.0021 ▲

Table 4.9: Selected efficiency operating points from Figure 4.5. ▼/▲ indicate significant differences compared to exhaustive search ($p < 0.05$).

selecting it in the cluster selection process means examining a fair fraction of the collection. To analyze this in more detail, we show the distributions of cluster sizes for hourly batch, hourly online and daily batch in Figure 4.7. In this figure, we do not show the largest cluster that comprises 12.6%, 10.4% and 16.6% of all tweets respectively because most of them are non-English tweets. The clusters reveal a distribution of cluster size that is not uniform. However, such imbalance does not affect the efficiency. In Table 4.3, the size of the clusters examined over the collection

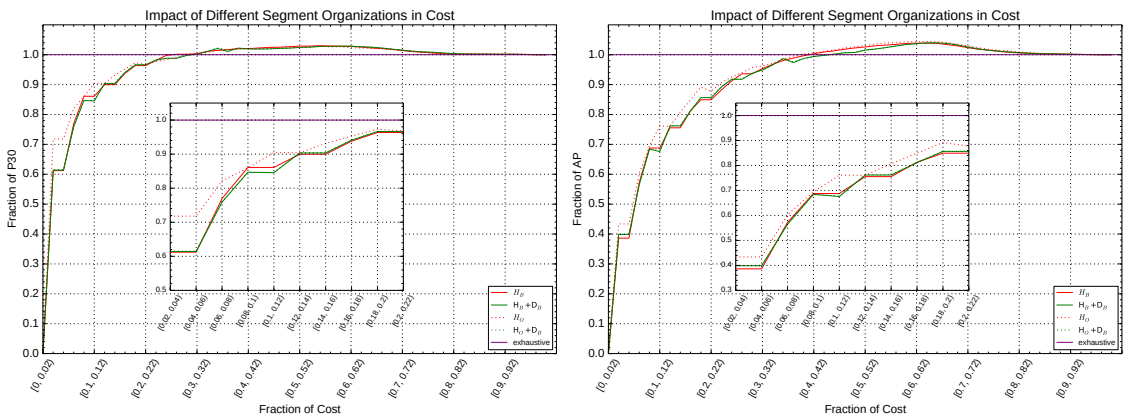

Figure 4.6: Effectiveness (fraction of P30 and AP) vs. cost (fraction of the collection examined) under different segment organizations: hourly batch (H_B), hourly online (H_O), hourly batch + daily batch (H_B+D_B), and hourly online + daily batch (H_O+D_B) on Tweets2013.

is roughly proportional to the number of clusters, indicating a non-affected behavior by the clustering process.

We show the distributions of cluster sizes for hourly batch, hourly online and daily batch on Tweets2013 in Figure 4.8. In these three figures, we do not show the largest cluster that comprises 9.7%, 7.6% and 13.4% of all tweets respectively because most of them are non-English tweets. Same as Tweets2011, the clusters reveal a distribution of cluster size that is not uniform. However, such imbalance does not affect the efficiency. In Table 4.8, the size of the clusters examined over the collection is roughly proportional to the number of clusters, indicating a non-affected behavior by the clustering process.

Fraction of P30	H_B	H_O	H_B+D_B	H_O+D_B
$[0.04, 0.06)$	0.6123±0.0247	0.7180±0.0226	0.6140±0.0161	0.6156±0.0189
$[0.20, 0.22)$	0.9636±0.0214	0.9680±0.0106	0.9663±0.0061	0.9667±0.0055
$[0.22, 0.24)$	0.9810±0.0205	0.9771±0.0114	0.9818±0.0119	0.9825±0.0113
$[0.34, 0.36)$	1.0143±0.0099	1.0156±0.0155	1.0210±0.0205	1.0224±0.0234
$[0.40, 0.42)$	1.0206±0.0111	1.0200±0.0093	1.0194±0.0155	1.0194±0.0155
Fraction of AP	H_B	H_O	H_B+D_B	H_O+D_B
$[0.04, 0.06)$	0.3862±0.0310	0.4338±0.0165	0.3988±0.0276	0.3990±0.0372
$[0.20, 0.22)$	0.8493±0.0327	0.8776±0.0104	0.8564±0.0250	0.8568±0.0246
$[0.22, 0.24)$	0.8822±0.0200	0.9068±0.0137	0.8920±0.0226	0.8925±0.0228
$[0.34, 0.36)$	0.9804±0.0219	0.9843±0.0167	0.9872±0.0020	0.9879±0.0035
$[0.40, 0.42)$	1.0032±0.0124	1.0059±0.0097	0.9939±0.0128	0.9939±0.0126

Table 4.10: Selected cost operating points from Figure 4.6.

4.4.3 Impact of Word Embeddings

In this section, we explore the impact of word embeddings on our selective search approaches. To be more specific, we are interested in two things: First, does word embeddings in different dimensions have an impact on the effectiveness? Second, does a different training corpus have an impact on the effectiveness?

To answer the first question, we repeat the same experiments as with the previous sections, but vary the number of dimensions to 5, 10, 25, 50, 100 when using GloVe to obtain word embeddings. We show the results in Figure 4.9 for hourly

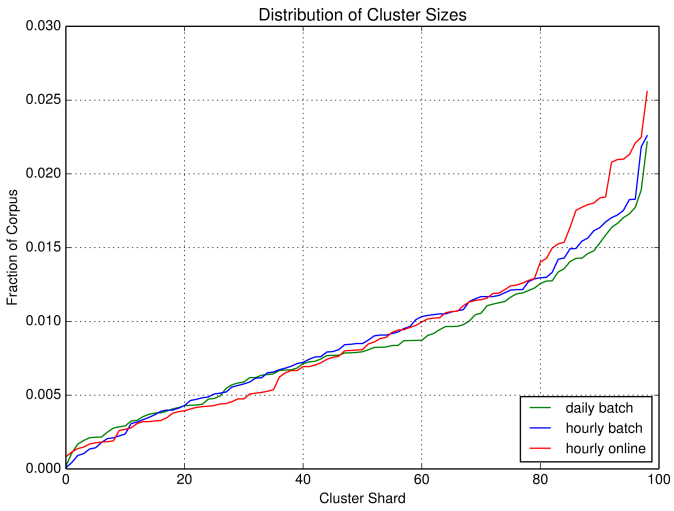

Figure 4.7: Typical distribution of cluster sizes under different conditions: hourly batch, hourly online, and daily batch on Tweets2011.

online (H_O) and hourly batch + daily online ($H_B + D_B$) segment organizations, with P30 on the left column and AP on the right column. Results for other segment organizations are similar, so we left them due to space limitation.

We see that the different dimensions do have an impact over the effectiveness. We achieve the best efficiency/effectiveness tradeoff with 10 and 25 dimensions. Good tradeoff means that higher effectiveness with the same efficiency. Such tradeoff starts to go worse as we reduce the dimensions as well as increase the dimensions. This is surprising because the best dimension is a bit lower than we've expected. Nevertheless, this is an interesting observation that we need more explorations.

To answer the second question, we take the test collection and train GloVe on it to generate word embeddings in 25 dimensions, then repeat the above experiments.

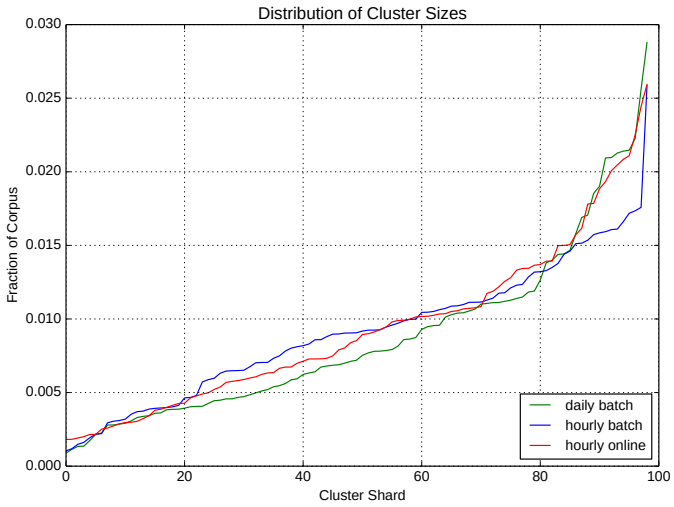

Figure 4.8: Typical distribution of cluster sizes under different conditions: hourly batch, hourly online, and daily batch on Tweets2013.

The motivation behind this is that although training on Edinburgh corpus is good as it does not bring in any future information hence avoid over-fitting, there is a problem behind it. As new documents are encountered in the test collection, new terms are also introduced. Therefore, there will be more and more terms that become out-of-vocabulary (OOV) terms in test collection, resulting in not being able to contribute to the document vector because they are considered as vectors of zeroes. And we would like to see in what level that could impact on the effectiveness.

We show the results in Figure 4.10 for hourly online + daily batch (H_O + D_B), with P30 on the left column and AP on the right column. Results for other conditions look similar. We see that there is not significant difference between the two, suggesting that our techniques are robust to differences of training corpus in

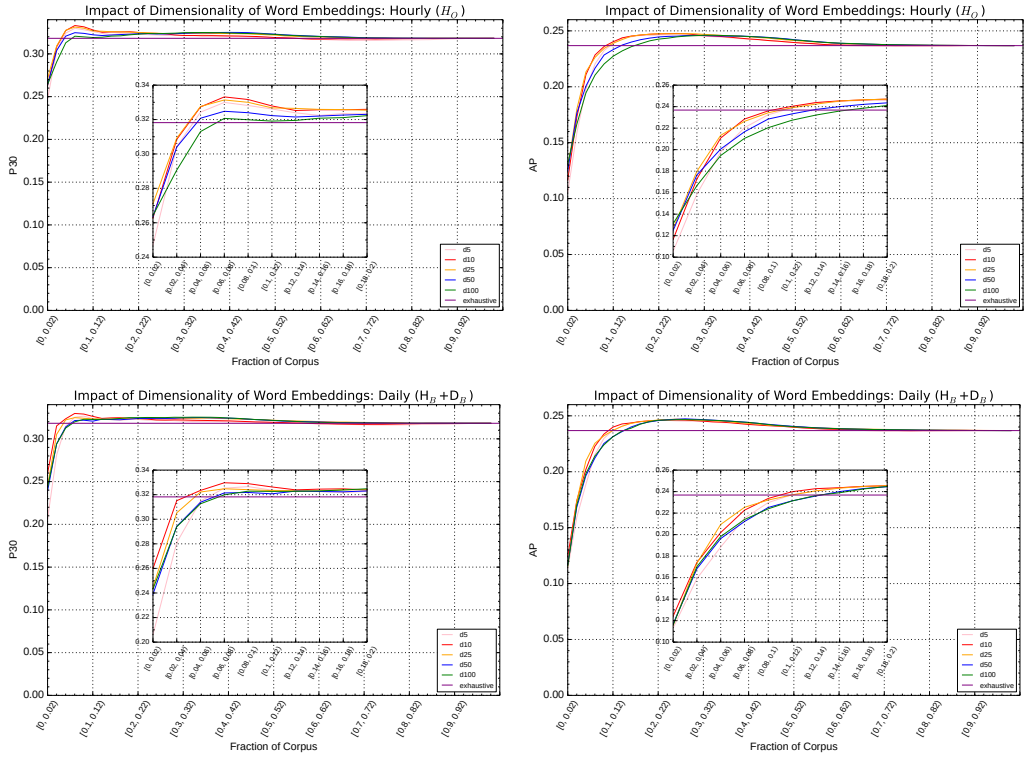

Figure 4.9: The effect of word embeddings of different dimensions on selective search effectiveness: the hourly online (H_O) and hourly batch + daily batch (H_B+D_B) segment organizations (averaged over five trials); P30 on the left, AP on the right, on Tweets2011.

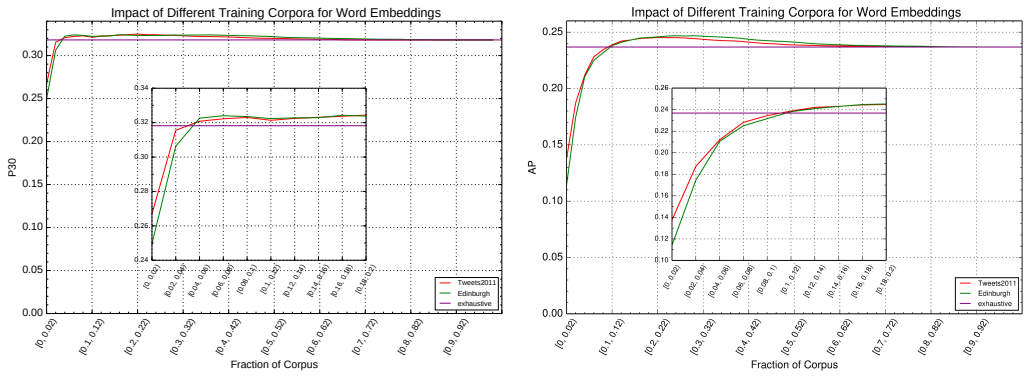

Figure 4.10: The impact of word embeddings trained on the Edinburgh tweet corpus vs. the Tweets2011 corpus (25 dimensions): the hourly online + daily batch (H_O+D_B) segment organization (averaged over five trials); P30 on the left, AP on the right, on Tweets2011.

the word embeddings.

To study the impact word embeddings in different dimensions have on the effectiveness with Tweets2013, we repeat the same experiments, vary the number of dimensions to 5, 10, 25, 50, 100 when using GloVe to obtain word embeddings and show the results in Figure 4.11 for hourly online (H_O) and hourly batch + daily online ($H_B + D_B$) segment organizations on Tweets2013, with P30 on the left column and AP on the right column. Results for other segment organizations are similar, so we left them due to space limitation. We see that, similar to Tweets2011, the different dimensions do have an impact over the effectiveness. We achieve the best efficiency/effectiveness tradeoff with 10 and 25 dimensions. Effectiveness starts to go worse with the same efficiency as we reduce the dimensions as well as increase

the dimensions.

To study the impact a different training corpus have on the effectiveness with Tweets2013, we take Tweets2013 and train GloVe on it to generate word embeddings in 25 dimensions, repeat the above experiments on Tweets2013 and show the results in Figure 4.12 for hourly online + daily batch ($H_O + D_B$), with P30 on the left column and AP on the right column. Results for other conditions look similar. We see that there is not significant difference between the two, suggesting that our techniques are robust to differences of training corpus in the word embeddings.

4.5 Conclusion

In this chapter, we have proposed a system that can provide efficient real-time search service. To achieve this, we introduce selective search on temporal segments using both batch and online clustering to reduce the number of the tweets we need to examine, and also exploit word embeddings to reduce dimensionality of document vector to yield faster clustering process. Experiments have shown that, we can achieve as good effectiveness as exhaustive search when we are able to dramatically increase efficiency. Our approach is novel, generalizes well on bigger collection and provides a starting point for the real-time search systems aiming for high efficiency.

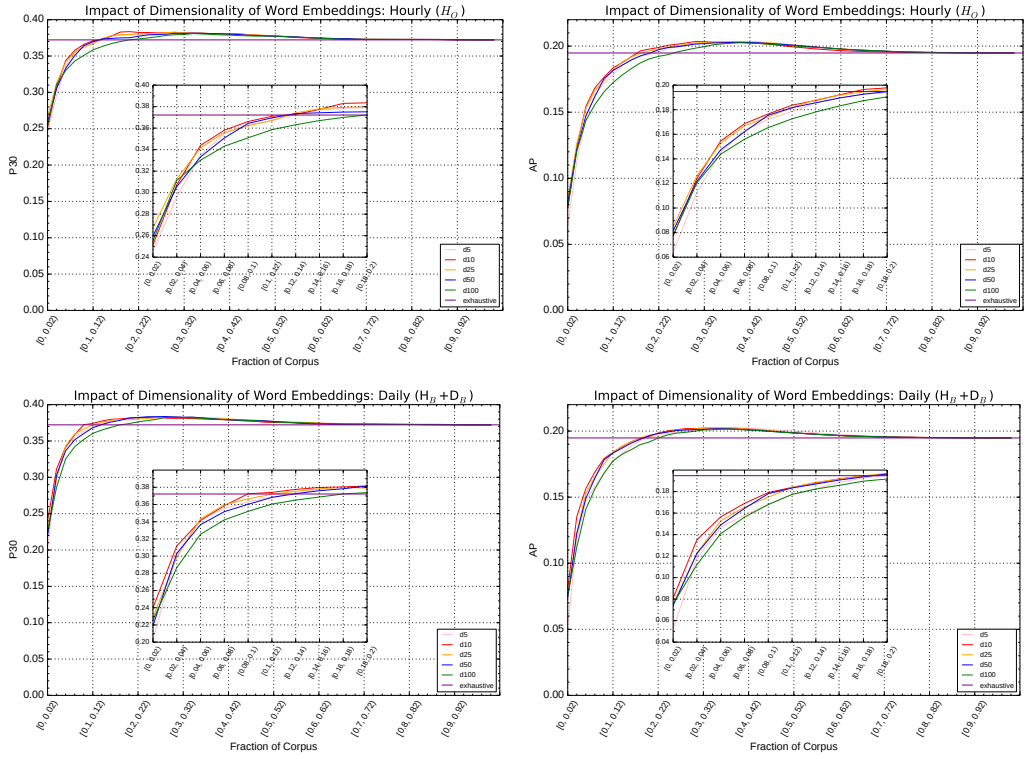

Figure 4.11: The impact of word embeddings of different dimensions on selective search effectiveness: the hourly online (H_O) and hourly batch + daily batch (H_B+D_B) segment organizations (averaged over five trials); P30 on the left, AP on the right, on Tweets2013.

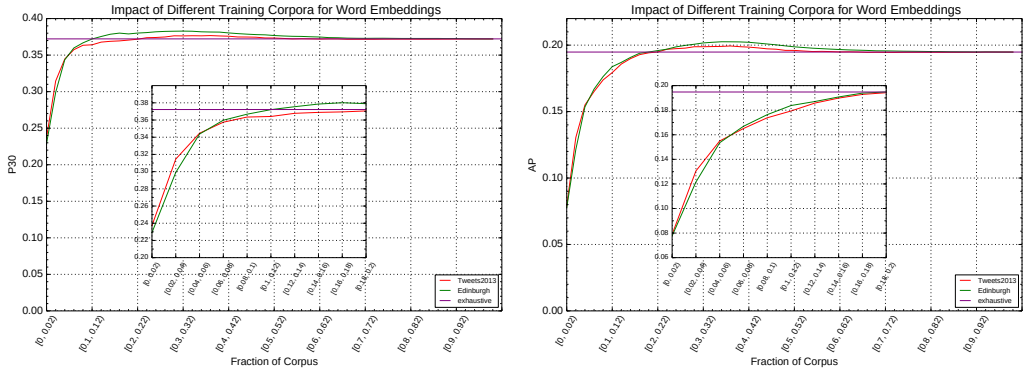

Figure 4.12: The impact of word embeddings trained on the Edinburgh tweet corpus vs. the Tweets2013 corpus (25 dimensions): the hourly online + daily batch (H_O+D_B) segment organization (averaged over five trials); P30 on the left, AP on the right, on Tweets2013.

Chapter 5: Tweet Timeline Generation

Once we have built the real-time tweet search system as described in the previous two chapters, one critical thing as discussed in Chapter 1 is to address the user's need in acquiring a summary of these relevant tweets so each result describes the topic in a different aspect. This is motivated by the fact that there are often duplicate and near-duplicate tweets saying almost the same thing. In such cases, users will be overwhelmed by these duplicate and near-duplicate tweets, even if they are relevant, and it would be desirable if the system could summarize the relevant tweets and return to the user a list of results, each of which covers a different aspect of the topic.

As a fact, the summarization task has already been defined as the Tweet Timeline Generation (TTG) task at the TREC 2014 Microblog track. As the co-organizer of the track, I helped develop evaluation methodology to evaluate a TTG system. In this chapter, we first introduce TTG task and then describe the proposed evaluation framework.

5.1 Task

5.1.1 Definition

Tweet Timeline Generation task at TREC 2014 Microblog track, is defined as follows: "At time T, I have an information need expressed by query Q, and I would like a summary that captures relevant information". The system's goal is to produce a "summary" timeline, operationalized as a list of non-redundant, chronologically sorted relevant tweets. This is a supplement of another task, temporally-anchored ad hoc retrieval whose user model is as follows: "At time T, give me the most relevant tweets about an information need expressed as query Q" where the system's goal is to retrieve most relevant tweets for an information need. It's required that a participant must participate the ad hoc retrieval task if he wants to participate in TTG task. Beyond ad hoc retrieval, for TTG task, given an information need, there are two challenges a system needs to address:

- Detect novel tweets. This is equivalent to removing redundant tweets.
- Determine how many tweets to return. Because the number of novel tweets varies from information need to information need, it's up to the system to make a decision how many tweets to return.

Redundancy is defined as follows: For every pair of tweets, if the later one contains substantively similar information with respect to the first one, the later one is considered redundant; otherwise, it's considered as novel. Whether a later tweet contains substantively similar information can be operationalized as answering

a question: If I have seen tweet A, does tweet B contain any important information enough that I would be interested in seeing it? If the answer is no, then tweet B contains substantively similar information and so is redundant with respect to tweet A, otherwise, it's considered novel.

Redundancy has several properties that need to be pointed out here: it's *not* symmetric: if the later tweet B is redundant with respect to an earlier tweet A, then tweet A is not necessarily redundant with respect to tweet B. It is transitive: if tweet A precedes tweet B, tweet B precedes tweet C, B is redundant with respect to tweet A, tweet C is redundant with respect to tweet B, then tweet C is redundant with respect to tweet A.

5.1.2 Data

The collection used in the tasks is known as Tweets2013 and consists of 243 million tweets crawled from the public Twitter sample stream between February 1 and March 31, 2013 (inclusive). Participants can have basic search access to the collection and obtain tweet content as well as other metadata fields through a service API. For topics, NIST assessors developed a total of 55 topics (given to the participants as queries) as the information need and participants need to perform their tasks against these topics.

5.2 Evaluation Methodology

As discussed in the Chapter 1, the evaluation methodology works as follows: Given a task, to evaluate a system's performance in achieving the task, we gather judgments from human assessors for the task with the same dataset, and this along with the system's results, are fed into an evaluation metric that gives a score indicating how well the system does in satisfying human judgments. We will discuss in this section how the human judgments are collected, and in the next section demonstrate how we develop the metrics and show systems' effectiveness under these metrics.

Because the TTG task can be viewed as a clustering task, the process of assessing can be operationalized as first grouping relevant tweets into clusters in which tweets share substantively similar information. Within each cluster, the earliest tweet is novel and all the other tweets are redundant with respect to the earlier ones. And then choose one tweet from each cluster to generate a list of non-redundant tweets that's a summary of a topic. The strategies to choose which tweet depends on users' preferences, for example: the earliest one is preferred if the user is time-sensitive and would like to catch the trend of an event at an earlier moment; a tweet from an authority or his friends' circle is preferred if he would like to see something trustable. And we'll discuss in the next section how we'll address this when we discuss the evaluation metrics and for now just leave the clusters as the current judgments.

To obtain the relevant tweets, also the judgments for the temporally-anchored ad hoc retrieval task, NIST assessors used a standard pooling strategy to use the

depth 100 tweets from all submitted ad hoc runs and a random selection of 100 tweets per topic from each TTG run, and judge the tweets on a three-way scale of "not relevant", "relevant", "highly relevant" with the score of 0, 1 an 2 respectively. The relevant and highly relevant tweets are then used by the assessors to generate semantic clusters.

We developed a JavaScript-based web annotation interface to help the assessors to accomplish the task, see screenshot in Figure 5.1. And the annotation process can be explained as follows: For each topic, the tweets are sorted chronologically from earliest to latest and are shown one at a time at the bottom. For each tweet, the assessor can add it to an existing cluster by clicking the "Add" button next to the cluster if he thinks the tweet contains substantively similar information as any of the tweets, otherwise, hitting the space bar to make it a new cluster. At any time, the assessor can expand a cluster to show all the tweets contained in it or collapse a cluster where only the first tweet is shown. Finally, an undo feature allows assessors to reverse actions multiple times if he thinks he makes a mistake for a previous tweet.

There are two reasons why we explicitly design the interface this way: Firstly, it's in accordance with the definition of redundancy: whether a later tweet contains substantively similar information with respect to an earlier one, this is under the assumption that users see the tweets in a sequential order and make decisions for each tweet. So the design depicts how users perceive tweets in a real world. Secondly, as some topics have more than 200 relevant tweets, having the assessors see all the tweets and group them by dragging tweets up and down makes it hard to keep track

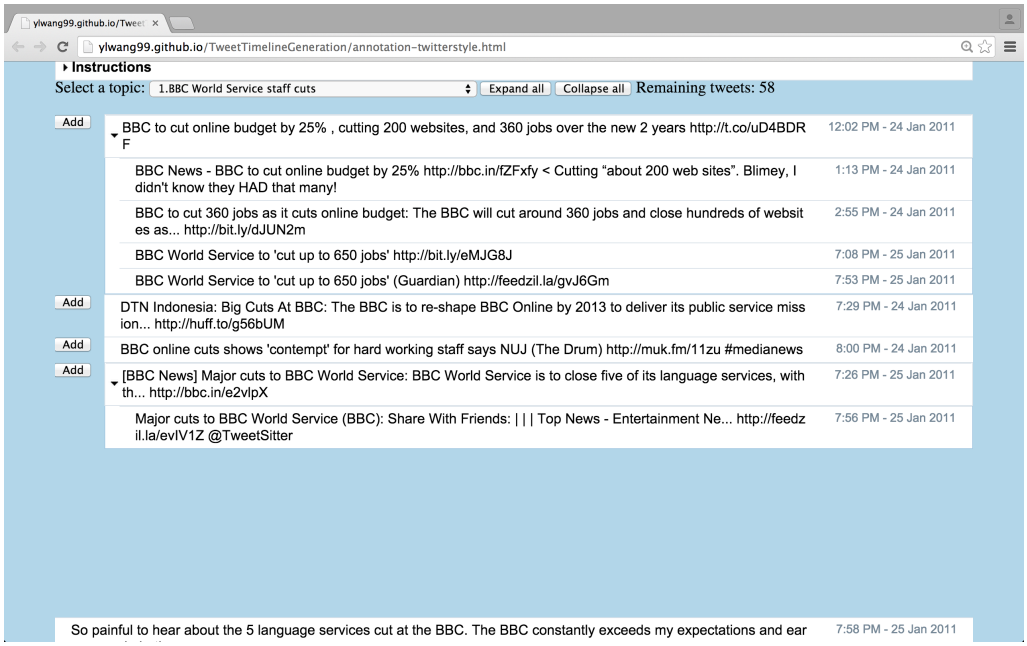

Figure 5.1: Screenshot of the annotation interface. Tweets are presented one at a time in chronological order (bottom). For each tweet, the assessor can add it to an existing cluster or create a new cluster.

of all the tweets at one time, hence is an impractical annotation task.

Due to resources constraints in NIST, we proceeded the annotation at University of Maryland and University of Illinois. The two assessors at the University of Maryland were graduate students in computer science (both male), and the two assessors at the University of Illinois were graduate students in library and information science (one male, on female). All of them have experiences using Twitter so it should be easy to get familiar with the interface. And we explicitly made the number of assessors small to keep annotation consistency across topics.

Assessors were first trained in a laboratory, and then they can perform the task on their own laptops at any place they're comfortable with. Topics are distributed to them in batch of 11 topics, and each batch contains roughly the same number of relevant tweets in total. Each assessor started with one batch and when he finished, he can ask for another batch to work on.

So the output of an annotation is a sorted list of tweet clusters. These clusters are sorted chronologically by the first tweet and within each cluster, all the tweet are sorted by their temporal order.

Each site started annotation in an opposite order, and when a covering set of all topics are obtained, we designated these as the "Official" judgments. Each site continued until both of them had a complete set of judgments of all topics, and those gave us the "Alternate" judgments.

5.3 Metrics and Results

To answer the question in the previous section as which tweet to choose in each cluster to form a summary, for simplicity, we leave the problem behind, but rather, when evaluating, we give one credit to a system as long as it returns *any* tweet from a cluster. All the other tweets returned within that cluster receive no credit. Hence, this can be considered as cluster-based evaluation, and we use cluster-based precision, recall and F_1 as the metrics, where the denominator for recall is simply the total number of clusters, and we call these the unweighted versions because retrieving one cluster receives one credit across all clusters and all topics. We also have weighted variant of recall and F_1. In this, each cluster is assigned a weight that's the sum of the relevance grades from all the tweets in that cluster and a system receives credit equal to the cluster's weight by retrieving any tweet from that cluster. The denominator of the recall is the sum of the clusters' weights, and weighted F_1 is computed the same way as unweighted F_1 but with weighted recall.

We received a total of 50 runs from 13 groups of participants to the TTG task. Figure 5.2 shows the precision vs. unweighted recall and precision vs. weighted recall for each run based on the Official judgments. Iso-F_1 contours are plotted in blue; points on the same contour line have the same F_1 score, but with different precision/recall tradeoffs. And this shows that runs make different decisions when balancing between precision and recall. For two runs that have same F_1 score but different precision/recall tradeoffs, the one with high precision returns higher percentage of tweets that are non-redundant while the one with high recall returns

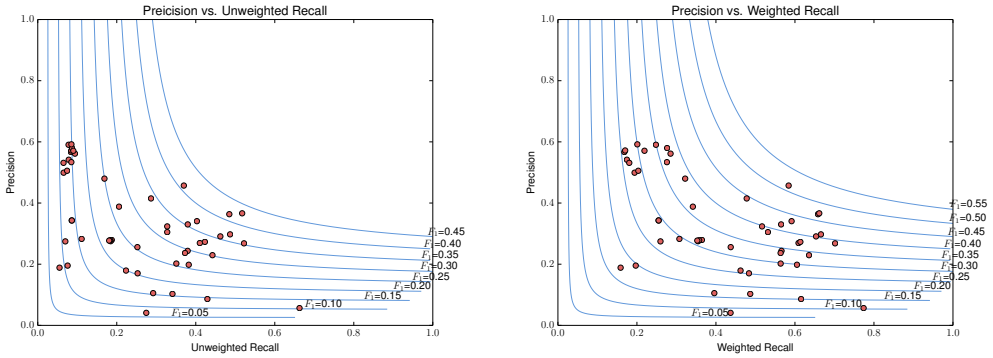

Figure 5.2: Scatter plots showing precision vs. unweighted recall (left) and precision vs. weighted recall (right) for all submitted runs in the TREC 2014 TTG task, overlaid with iso-F_1 contours.

more number of tweets that are non-redundant.

Chapter 6: Tweet Timeline Generation Evaluation Validation

In the previous chapter, we have demonstrated how to develop the TTG evaluation framework. In this chapter, we aim for the validation of the TTG evaluation framework. Specifically, we answer two questions that should be asked in information retrieval evaluation:

- As we develop the system to strive for high metric score, is the comparison between system with higher score and system with lower score stable with respect to assessor differences?
- Is the difference meaningful to a user?

As the co-organizer of the Microblog track, we have access to the "ground truth" and the system runs' results from TREC 2014 Microblog track, thus can answer these two questions by conducting an empirical analysis. In the sections that follow, we conduct experiments and show that evaluation is stable with respect to assessor differences and that user preferences generally correlate with effectiveness metrics. The entire study was a collaboration between me, Garrick (a graduate student from UIUC), Miles (an associate professor from UIUC), and Jimmy. All of us jointly designed the experiment, discussed analyses, etc. Our collaboration led to a SIGIR paper [10].

6.1 Assessor Differences

This section aims to answer the first question asked at the beginning of this chapter: is the evaluation stable with different assessors? Here the stable doesn't mean a system obtains the same effectiveness score based on two sets of judgments, but rather its effectiveness ranking among all other systems remains stable. For example, if system A scores higher than system B based on one set of judgments, the evaluation can be considered as stable if system A still scores higher than system B based on another set of judgments. We don't compare the absolute scores because there are uncontrollable variations among different assessors' judgments depending on their knowledge and their own understandings of the task, and it's not practical to obtain roughly the same judgments.

Following Voorhees' methodology [125] to examine the assessor differences' impact on system rankings for ad hoc retrieval, we show in Figure 6.1 the scores for all runs based on the official judgments and alternate judgments for each of the five metrics. Results are sorted by the scores based on the official judgments in descending order. We can see that system rankings are highly correlated for both sets of judgments except for weighted F_1 where we see a jagged blue portion.

To look at the impact different judgments have on the system rankings in more detail, we show in table 6.1 the number of rank swaps for each metric. A rank swap is a pairwise comparison where according to one set of judgments, system A scores higher than system B, but according to another set of judgments, system B scores higher than system A. Out of a total $(50 * 49)/2 = 1225$ possible pairwise

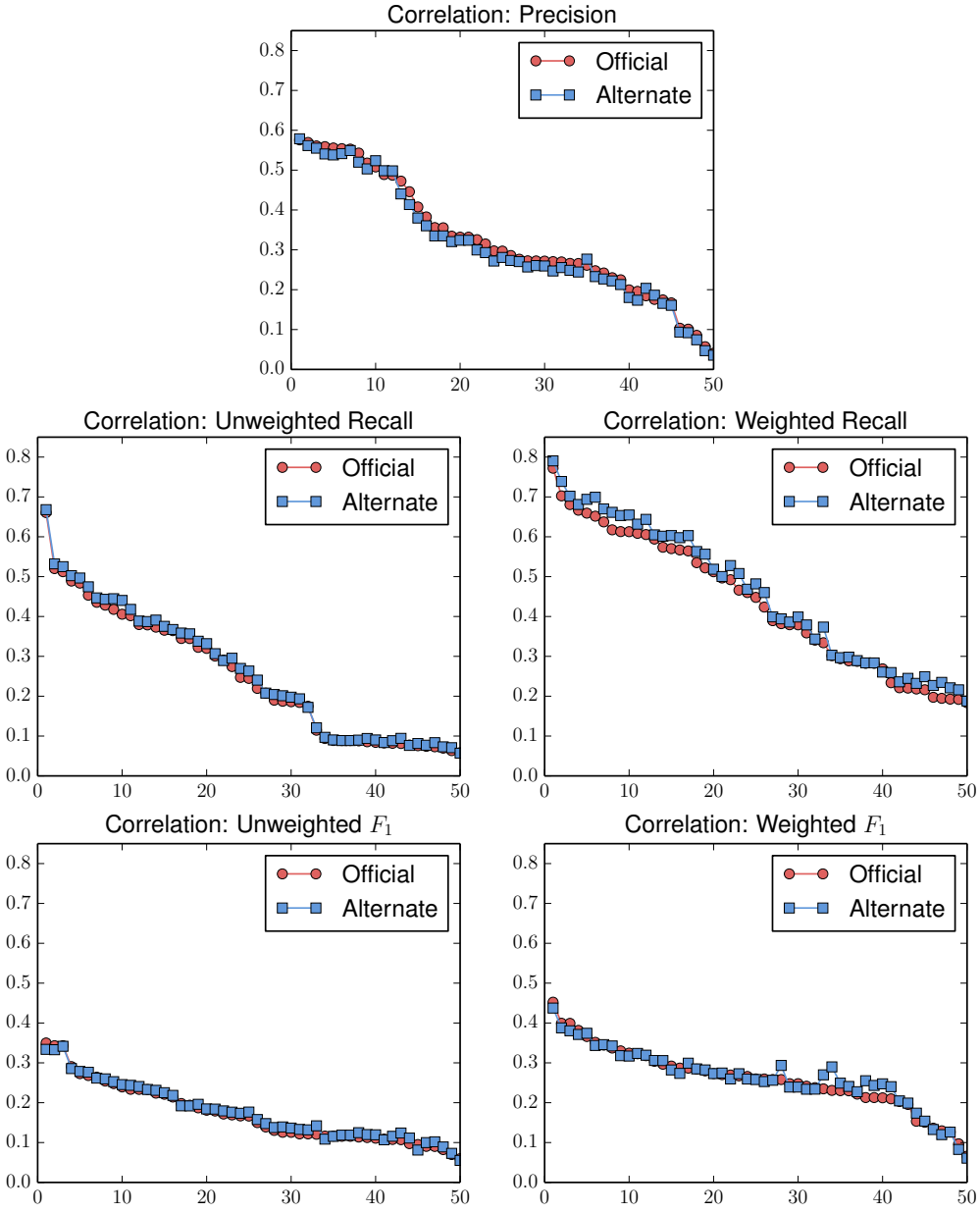

Figure 6.1: Comparison between scores based on the official judgments and the alternate judgments for various metrics. Runs are sorted by score based on the official judgments in descending order.

Metric	Rank Swaps	Kendall's τ
precision	30	0.951
unweighted recall	27	0.956
weighted recall	22	0.964
unweighted F_1	46	0.925
weighted F_1	90	0.853

Table 6.1: Count of rank swaps and Kendall's τ correlation based the official and alternate judgments for each metric.

comparisons, the number of rank swaps shown here is quite small. We also show the Kendall's τ correlation, a measure of rank correlation used to measure the association between two measured quantities. These values are in the same range as those reported by Voorhees [125] for ad hoc retrieval, and are generally regarded by the IR community to be stable with respect to assessor differences.

Although there's high correlation between system rankings based on two sets of judgments, we're still concerned whether the rank swaps occur when the score differences are small. To study this, we plotted a histogram in Figure 6.2 showing the number of rank swaps for unweighted F_1 and weighted F_1 that have the lowest Kendall's τ correlations, binned by score differences. We see that the score differences are small. So evaluation is considered stable with respect to assessor differences.

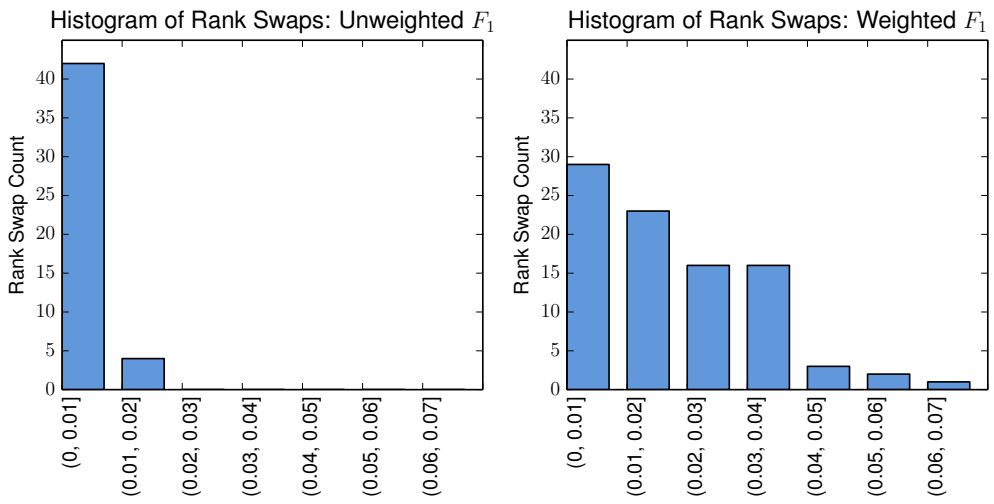

Figure 6.2: Histogram of rank swaps for unweighted F_1 and weighted F_1, binned by score differences.

6.2 User Preferences

In this section, we tackle the second question brought up at the beginning of this chapter: are the different effectiveness scores for TTG systems meaningful to a user? Here, the meaningful can be explained like this: If metric gives system A higher score than system B, and a user also prefers system A when he's shown both systems' outputs, then we say there's a strong correlation between the metric and user preferences and the metric can capture the system differences that satisfy the user preferences. In the following, we'll first present the methodology we use to obtain user preferences on TTG systems with respect to different metrics, and then demonstrate the strategies we use to analyze the correlation between metrics and user preferences and discuss the results, finally draw a conclusion.

6.2.1 Analysis Methodology

When studying user preferences over different TTG systems with different effectiveness, unlike simple tasks like ad hoc retrieval, which can be achieved by showing pairs of system outputs to assessors and asking for their preferences given any metric because the metric is independent with each other, for TTG task, the process is more complicated for two reasons: first, TTG system results are variable in length; second, and most importantly, to study each metric's impact on users preferences of the systems, we need to eliminate the effects other metrics might have on them because these metrics themselves are dependent. To be more specific, there are three types of effectiveness differences we are interested in:

- When two systems have roughly the same recall, how are they sensitive to the precision differences?
- When two systems have roughly the same precision, how are they sensitive to the recall differences?
- When two systems have similar precision/recall tradeoffs, how are they sensitive to F_1 differences?

The strategy we follow is to sample system output from the submitted TREC 2014 TTG runs and generate system comparisons on a topic-basis for each of the three types, which we call *precision sampling, recall sampling*, and F_1 *sampling* respectively. Before sampling, we perform some filterings following the principle of reducing the impact outliers may have on our assessment and easing the assessment

task for an assessor:

First step of filtering involves removing topics. Out of the total 55 topics, select 30 topics that neither have too many or too few relevant tweets. A topic having too many relevant tweets adds difficulty for assessing while too few relevant tweets are not sufficiently enough to study user preferences. The filtering process works as follows: Let $x_i = r_i - \bar{r}$ be the difference between r_i, the number of relevant tweets for topic i, and $\bar{r}$, the median number of relevant tweets over all topics: the probability of "drawing" topic i is proportional to the density at x_i of a normal distribution with $\mu = 0$ and $\sigma = 20$.

Secondly, we perform filtering of the submitted runs: for each topic, discard runs that return more than 41 tweets, which is the median length of submitted runs. The reason is that study has shown that assessors have difficulties making judgments for long results and we want to explicitly reduce the burden on assessing so as to increase accuracy.

After filtering, we conduct a sampling process to sample the three types of comparisons. And this process also involves two steps:

Firstly, for each topic, randomly select 20 "base" runs, and for each of the "base" run, sample up to 20 different "comparison" runs based on the following criteria:

- For *precision sampling*, select runs that differ with base runs in recall by less than 0.1, but more than 0.1 in precision. Setting the difference to be greater than 0.1 makes sure that it's relatively easy for assessors to make judgments.

- For *recall sampling*, select runs that differ with base runs in precision by less than 0.1, but more than 0.1 in recall.

- For F_1 *sampling*, select runs that exhibit similar precision/recall tradeoffs with base runs, to be more specific, the runs where the value of $T = |P - R|$ is within 0.1 of the T for the base run, but the difference in precision or recall between the two runs is greater than 0.1.

For all the chosen comparison runs, discard the pairs where the two runs differ in length by more than 20 tweets. This, again, is designed explicitly to make assessment process easy.

Secondly, out of the large number of remaining comparison runs, we randomly sample 180 pairs consisting of roughly the same number of pairs for precision sampling, recall sampling and F_1 sampling. And these 180 pairs represent a single batch we give the assessors to judge. We can generate another batch by following the same filtering and sampling process. Note that all sampling was performed on the cluster-based TTG metrics and we used the unweighted variants of the metrics for simplicity.

To help the assessors to accomplish the task, we developed a web-based annotation interface, the screenshot of which is shown in Figure 6.3. Here, it displays a comparison for a given topic (shown at the top) between two system outputs from the 180 sampled pairs: System #1 on the left and System #2 on the right, and they're randomly put in the two positions to avoid any bias. Each system's output is ordered chronologically, and each tweet is displayed together with its timestamp.

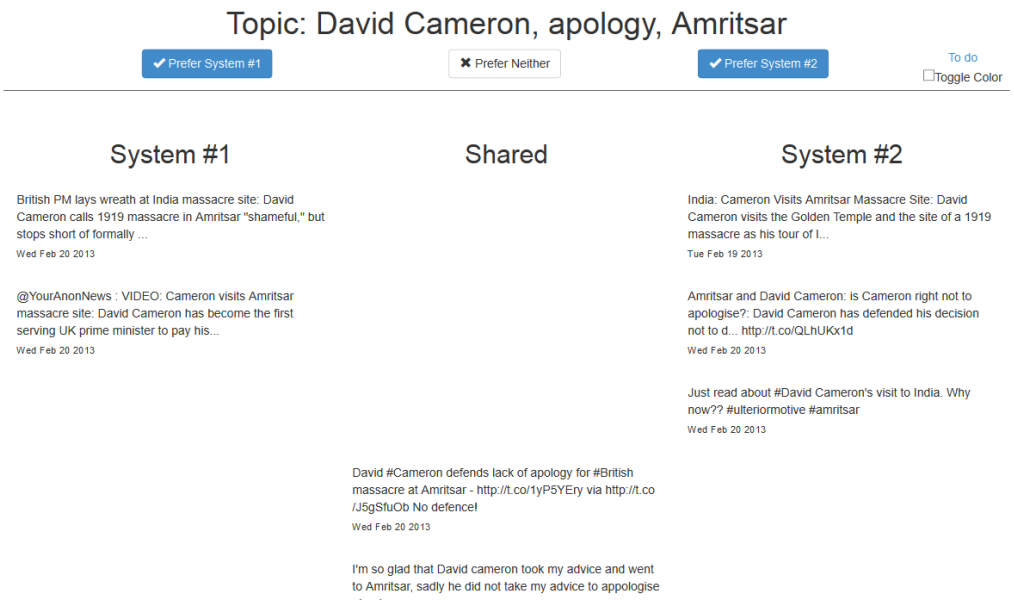

Figure 6.3: A screenshot of the web-based assessment interface for eliciting preference judgments. System outputs are presented in the left and right columns, with shared content in the middle column.

Tweets returned by both systems are displayed in the middle column, and are sorted together with other tweets from the two systems, meaning that there might be gaps in an system output, indicating one system returns more tweets than another prior to the shared tweets, and this is indeed the case in Figure 6.3.

For each comparison, we ask the assessors to make a judgment on which system they prefer using their own judgments and understandings. Regardless of the absolute quality of both systems, we want them to only care about the relative quality between the two. One can click on "Prefer System #1" on the top left if he prefers an output from system #1 and click on "Prefer System #2" on the top right if he prefers an output from system #2 and click on "Prefer Neither" if there isn't quality difference between the two. Note that, we *did not* illustrate what a

good system should be nor do we mention precision, recall or F_1 to the assessors, but rather trust their own judgments on what a good system should be, so to eliminate any bias and also learn which metric best captures user preferences for a TTG system.

We conducted the assessment at University of Maryland and University of Illinois. The two assessors at the University of Maryland were graduate students in computer science (both male), and the three assessors at the University of Illinois include two graduate students in library and information science (one male, on female) and one former graduate student in library and information science (male) who was working nearby. These assessors are different from the assessors for the TTG clustering annotation task, and we explicitly decided on this to avoid any prior an assessor might have.

Assessors were first trained in a laboratory to learn the task and the web interface. Then they can perform the task on their own laptops at any place they're comfortable with. Each assessor started with one batch of 180 comparisons and when he finished, he can ask for another batch to work on.

The server hosted the web assessment interface recorded interactions including preference judgments and their timestamps.

6.2.2 Results

To study the agreement between user preferences and preferences measured by the metrics, we use Cohen's κ as the agreement metric. Cohen's κ is a statistic

that measures the agreement between two raters who each classifies N items into C mutually exclusive categories. In our case, we have two raters, one rater is our assessor, while the other rater is the metric, and N is the total number of comparisons the assessor needs to make judgments on, and the number of categories of C is 2 where first category means system A is better than system B, and the other category means system B is better than system A. Cohen's κ ranges from -1 to $+1$, where 0 indicates agreement by chance.

To compute Cohen's κ, we first discard any "prefer neither" judgments. In precision sampling case, we analyze the correlation between user preferences and preferences implied by different precision scores. In recall sampling case, we analyze the correlation between user preferences and preferences implied by different unweighted recall scores and weighted recall scores. In F_1 sampling case, we analyze the correlation between user preferences and preferences implied by different unweighted F_1 scores and weighted F_1 scores.

All the correlations are computed both in an aggregate level and individual level. Figure 6.4 shows these correlations binned by score differences, aggregated across all assessors. We aggregate the samples together which have metric score differences of more than 0.4 because there're only so few of them. The error bars show the 95% confidence interval, computed following Fleiss et al. [174]. The numbers at the bottom of each bar are the number of "prefer neither" judgments (on the left) and the number of preference judgments (on the right). By the definition of sampling, there are no samples falling into the [0, 0.1) bucket for precision and unweighted recall, but some samples did fall into that bucket for other metrics.

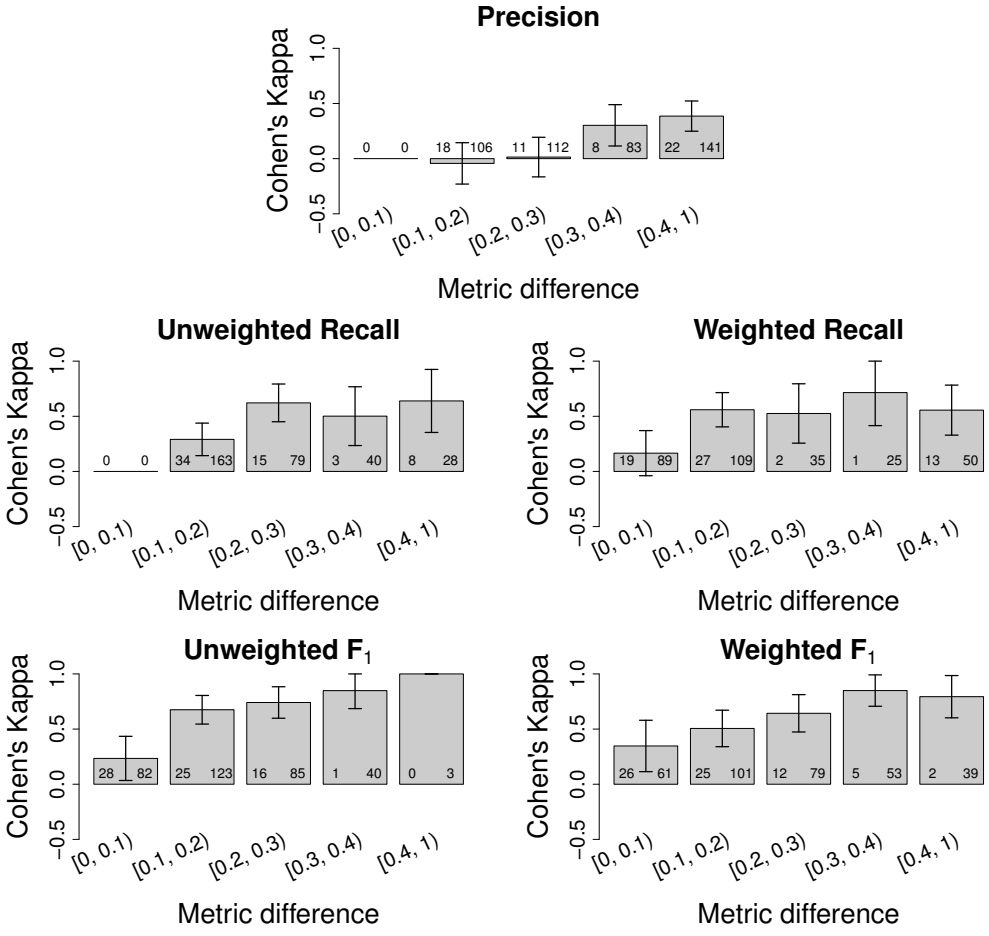

Figure 6.4: Cohen's κ for each metric binned by differences in the sampled metric. Error bars show 95% confidence intervals. The numbers in the bottom left of each bar show the number of "prefer neither" while those in the bottom right show the number of preference judgments in that condition.

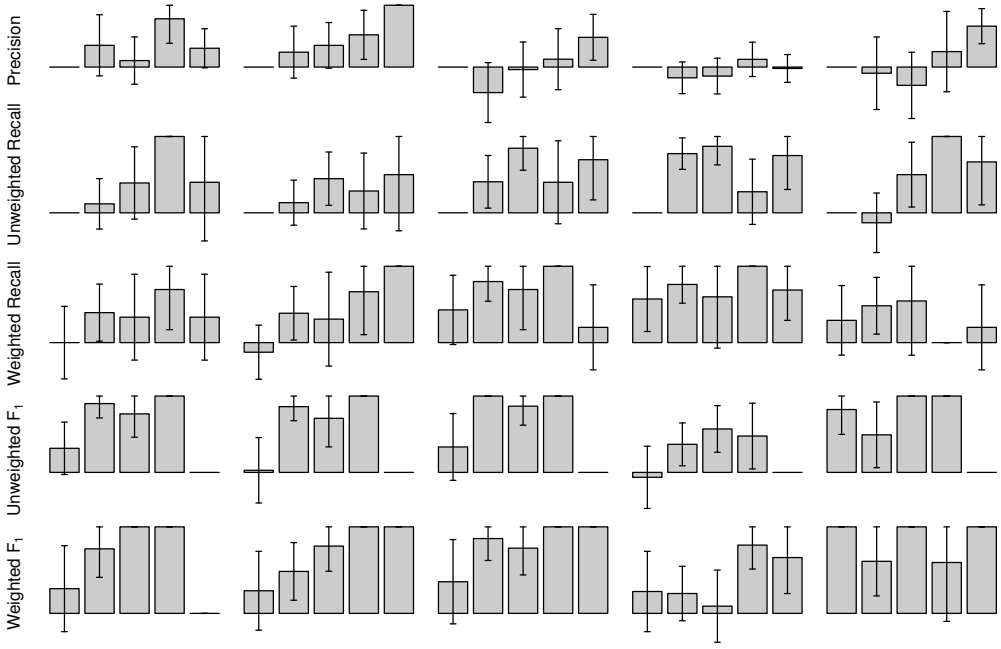

Figure 6.5: Cohen's κ for each metric (row), for each individual assessor (column) arranged in "small multiples". Each bar chart is organized in the same manner as those in Figure 6.4.

Figure 6.5 shows the correlation between individual assessor's preference and implied preference by each of the metrics. Each row represents a metric, and each column represents an assessor. Although the charts are shown without labels, they're organized in the same manner as in Figure 6.4. Note that confidence intervals are much larger because fewer samples fall into each bucket.

Given the correlations, there are two things we're interested in and we'd like to find out in a statistical way:

- Are the correlations high enough overall to indicate strong correlation between user preferences and preferences implied by each of the metrics?

- Is there a strong relationship between agreement and metric differences? Or

put it another way, are the correlations higher when metric differences are higher? Intuitively, when metric differences are high between two systems, indicating a large quality gap between the two systems, a reasonable assumption is that it should be easier for assessors to make judgments that agree with the metric preferences. Note that we arranged the bar charts the way that there is an increasing easiness of preference judgments from left to right. The leftmost bucket represents the samples with metric differences less than 0.1, in which case we would expect assessors have a hard time making judgments, whereas in cases where samples have metric differences more than 0.4 as represented by the rightmost bucket, assessors should have an easier time achieving higher agreement with metric preferences.

We replicate the guide proposed by Landis and Koch [1], and show them in Table 6.2.

To answer the first question, under the interpretation of κ, the overall agreement of the metrics are relatively high, from "moderate" to "substantial" on average, indicating strong correlations, except for precision, where we observe relatively low agreement between assessors and the precision metric. The problem is: our sampling process selected only those runs that have relatively short timelines, and such short timeline although probably has low precision, doesn't impose much "pain" to the assessor. In other words, the assessor can't easily tell which system is worse. However, had we selected runs with timeline that's an order of magnitude longer, thus imposing much burden, the assessment would become much easier.

κ	Strength of Agreement
<0.00	Poor
0.00-0.20	Slight
0.21-0.40	Fair
0.41-0.60	Moderate
0.61-0.80	Substantial
0.81-1.00	Almost perfect

Table 6.2: Interpretations of κ from [1].

For the second question, we do observe an increasing agreement trend from left to right, both at the aggregate level and individual level. And to analyze this statistically, we again take the interpretation of κ, and see that the minimum κ ranges from "poor" (for precision) to "fair" (for weighted F_1), and the maximum κ ranges from "fair" (for precision) to "almost perfect" (for unweighted and weighted F_1). We also observe that minimum κ occurs at the leftmost bucket for metrics and maximum κ occurs at the right buckets for metrics. Given these, we can claim that there's a strong relationship between the agreement and metric differences, the larger metric difference two systems have, the stronger agreement assessors have with the metrics.

To strengthen the statistical analyses of such relationship, we performed analysis of variance (ANOVA) comparing the κ values across bins for each metric.

Metric	F-value	p-value
precision	$F(3, 16) = 3.397$	0.0437
unweighted recall	$F(3, 16) = 1.749$	0.1970
weighted recall	$F(4, 20) = 1.475$	0.2470
unweighted F_1	$F(3, 16) = 5.093$	0.0115
weighted F_1	$F(4, 20) = 1.322$	0.2960

Table 6.3: ANOVA results for each metric. Metrics displayed in bold show a statistically significant difference ($p < 0.05$) in mean κ across bins.

ANOVA is a collection of statistical models used to analyze the differences among group means and their associated procedures (such as "variation" among and between groups). In our case, we would like to know the variance of κ values across bins for each metric. Note that the κ values are aggregated among assessors and we consider them as the "group" means when computing ANOVA. The results are shown in table 6.3.

Statistically significant differences in mean κ across bins are found for precision and unweighted F_1 at $p < 0.05$. This found supports the strong relationship between the agreement and metric differences.

One surprising observation is that there isn't statistically significant difference for κ values across bins for unweighted and weighted recall. Although the κ values are overall high, meaning the differences between systems are detectable by assessors,

they are not sensitive to the magnitude of the differences. We believe this is due to the nature of recall evaluation, which requires the knowledge of the complete relevance judgments. While it's easy to detect recall differences when some tweets returned by one system are absent from another system, it's hard to tell how much better one is to another without access to all the relevant tweets. And this is not true for precision because precision can be directly assessed from the displayed tweets, and Figure 6.4 verifies this.

Finally, we analyze the relationship between assessors' time spent on a comparison with the agreement. Figure 6.6 shows the κ values for each of the metrics, aggregated by all assessors, binned in intervals of 30 seconds. The numbers at the top of each bar shows the number of "prefer neither" and preference judgments. Missing bars indicate an undefined κ while dashes indicate a κ of zero. While we expected a trend of decreasing κ values from left to right as the time spent increases, because intuitively "harder comparisons" (thus low agreement) lead to long time decision, we didn't observe such trend, which is surprising.

Once again, we didn't explicitly ask the assessors to follow any specific rules like considering redundancy and it's up to them to take whatever into account. From the results, we see that assessors *are* sensitive to redundancy, and user preferences do agree with preferences measured by the metrics. Therefore, we can reasonably claim that systems metrics are meaningful in capturing what users care about in tweet summarization systems.

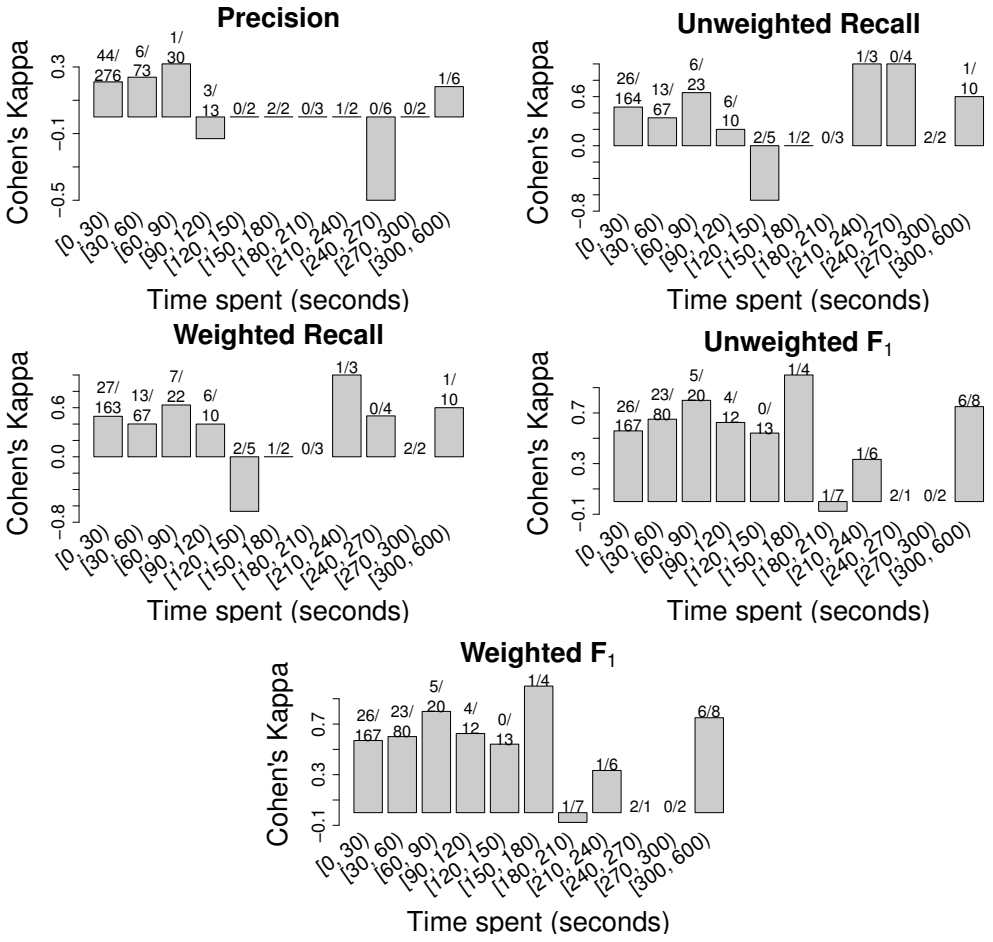

Figure 6.6: Cohen's κ binned by time spent on each comparison (at 30 second intervals). Numbers at the top of each bar show the number of "prefer neither" and preference judgments. Missing bars indicate an undefined κ while dashes indicate a κ of zero.

Chapter 7: Tweet Timeline Generation System

As discussed in previous sections, the list of results a Twitter search system returns in response to each query contains duplicate, near-duplicate tweets, and it is unlikely that users desire a list of documents containing redundant information. In this chapter, we design a TTG System that can return a list of non-redundant, chronologically sorted relevant tweets based on the relevant documents that have been retrieved from the search system, perform evaluation introduced in Chapter 5 and compare the effectiveness of our system against all the runs submitted to TREC 2014 and 2015 Microblog Track.

7.1 Approach

The TTG task can be viewed as a clustering task, therefore can be addressed by applying various clustering algorithms to group semantically similar tweets into the same cluster and pick one document from each cluster to form the final list of results. Another reasonable way to solve this task is to compute pairwise semantic similarity and throw one tweet in each pair that has a similarity score above some threshold, and return the rest of the tweets as the final list of results.

We follow the second approach and exploit the Sentence Similarity Model

with Convolutional Neural Networks proposed by He et al. [21]. In their work, the neural networks model takes pairs of sentences annotated with a score indicating how closely the two sentences are related as the training data, where the higher score, the more closely-related the two sentences are, and compute sentence similarity score that is between 0 and 5 (inclusive) for any given pair of sentences, where 0 indicates not similar at all, and 5 indicates extremely similar. This is suitable for our TTG task because each tweet can be viewed as a sentence due to length limitation therefore we can generate pairs of tweets out from the search results and compute tweet similarity score by exploiting their model without having to worry about applying complex strategy to compute document similarity as other collections do. And we call our system CNNTTG.

7.2 Experimental Setup

We used the sentence similarity model provided by He et al. [21] to compute sentence similarity scores for our tweet data. The training data of their work is from previous SemEval Semantic Textual Similarity competitions from 2012 to 2015. It is collected from diverse domains, such as news, Wikipedia, social media, image/video descriptions and others. In total it consists of 14,342 human-annotated sentence pairs, and each pair has a similarity score$\in [0, 5]$ which increases with similarity. They trained on the data to compute the sentence similarity model, and we used it directly without training on other dataset.

In this work, we don't build our search system and then run CNNTTG on top

of it, as this requires an architecture involving sophisticated ranking algorithms in order to be able to compare with TTG runs submitted by participants in TREC Microblog Track, which is not the focus of our work. We rather take all TTG runs submitted to TREC Microblog Track and run CNNTTG with different threshold values on each of them. To be more specific, we choose 4.0 and 4.5 as the threshold values to be experimented, and these numbers are heuristically determined. For each run, we generate all pairs of tweets for each topic, and for any pair that has a similarity score provided by the testing data that is higher than a given threshold, we remove one tweet from the results. We then run evaluation and compute recall, weighted recall, precision, F_1 and weighted F_1 scores on all refined runs and compare the performance our CNNTTG system with each of the original runs. As we're only removing tweets from the original runs, the goal is to remove only redundant tweets so as to improve precision but to keep recall and weighted recall.

7.3 Results

We show the performance of each original 2014 TREC TTG run with the highest F_1 score from each submitted group and our CNNTTG with similarity threshold 4.0 and 4.5 measured by recall, weighted recall, precision, F_1 and weighted F_1 in Table 7.1. All the original runs are sorted by weighted F_1 score. Metric scores are written in bold for our CNNTTG system if it is higher than that of the original run, and with underline if it is lower than that of the original run.

Looking at the table, we see that our system achieves precision that's higher

than or the same to all the original runs for both threshold values except for the worst run JufeLdkeSum2 with threshold of 4.0. This is expected as CNNTTG aims to remove tweets that are semantically similar with other tweets. However, both recall and weighted recall are decreased for all the runs except for run SCIAI3cm4aTTG with threshold of 4.0. With threshold of 4.5, three runs decreased in recall and two runs decreased in weighted recall. And none of the runs increases recall nor weighted recall for both thresholds. This indicates that while our system is capable in removing redundant tweets according to our annotators, it also removes some clusters for certain topics. While our system only outperforms four runs in F_1 score with threshold of 4.0 and outperforms six runs with threshold of 4.5, it achieves higher weighted F_1 score than 10 out of 13 runs with both thresholds. This indicates that the increase in precision outweighs the loss in recall, or in other words, the number of redundant tweets removed outweighs the number of clusters removed. And this is especially true for the weighted recall, meaning that our system is capable of removing clusters with lower weight.

Compare our CNNTTG system with different threshold values, we see that a system with the threshold of 4.5 achieves better performance than the one with the threshold of 4.0 for every metric. This suggests that the neural networks model tends to assign a pretty high similarity score for tweets that contain substantive similar information and what threshold to choose still remains further investigation as to reflect user preferences.

To strengthen the statistical analyses of system performance differences, we performed Repeated Measures ANOVA comparing the values across different sys-

TREC runs						CNNTTG (threshold=4.0)					CNNTTG (threshold=4.5)				
Run	recall	recall^w	precision	F_1	F_1^w	recall	recall^w	precision	F_1	F_1^w	recall	recall^w	precision	F_1	F_1^w
TTGPKUICST2	0.3698	0.5840	0.4571	0.3540	0.4575	0.3663	0.5797	**0.4621**	0.3524	**0.4589**	0.3698	0.584	0.4571	0.354	0.4575
EM50	0.2867	0.4779	0.4150	0.2546	0.3815	0.2846	0.4744	**0.4189**	0.253	**0.3818**	0.2867	0.4779	**0.4155**	0.2546	**0.3816**
hltcoeTTG1	0.4029	0.5915	0.3407	0.2760	0.3702	0.4013	0.5899	**0.3441**	0.2752	**0.3716**	0.4029	0.5915	0.3407	0.276	0.3702
QUTmpDecayTTgCL	0.3277	0.5167	0.3236	0.2440	0.3300	0.3236	0.5091	**0.3359**	0.2431	**0.3346**	0.3277	0.5167	**0.3249**	**0.2443**	**0.331**
PrisTTG2014b	0.4231	0.6137	0.2730	0.2461	0.3093	0.4177	0.6089	**0.2822**	**0.2472**	**0.3153**	0.4231	0.6137	**0.2754**	**0.2471**	**0.3112**
SRTD	0.0868	0.2764	0.5798	0.1324	0.2907	0.086	0.2749	**0.6207**	**0.1339**	**0.3039**	0.0868	0.2764	**0.5838**	**0.1326**	**0.2934**
3unique0	0.2522	0.4374	0.2558	0.1957	0.2744	0.2499	0.4356	**0.2614**	0.1949	**0.2777**	0.2522	0.4374	**0.2583**	**0.1959**	**0.2759**
udelRunTTG1	0.1873	0.3645	0.2793	0.1774	0.2669	0.1836	0.3597	**0.2823**	0.1741	0.2659	0.1872	0.3644	**0.2798**	0.1773	**0.2672**
UDInfoMMRWC5	0.0900	0.2191	0.5709	0.1338	0.2577	0.0899	0.2189	**0.583**	**0.1358**	**0.2631**	0.09	0.2191	**0.5782**	**0.1348**	**0.2608**
wistudt2bd	0.1111	0.3075	0.2827	0.1251	0.2305	0.1106	0.3072	**0.2833**	0.1245	0.2304	0.1111	0.3075	**0.2838**	0.1251	0.2305
SCIAI3cm4aTTG	0.0655	0.1941	0.4992	0.0921	0.2171	0.0655	0.1941	0.4992	0.0921	**0.2172**	0.0655	0.1941	0.4992	0.0921	**0.2172**
ICTNETAP4	0.2528	0.4836	0.1702	0.1559	0.2072	0.2441	0.473	**0.2142**	**0.1667**	**0.243**	0.2525	0.4832	**0.1934**	**0.1651**	**0.2285**
JufeLdkeSum2	0.4294	0.6156	0.0861	0.1180	0.1307	0.4242	0.611	0.0857	0.1167	0.1298	0.4291	0.6156	**0.0862**	0.118	**0.1308**

Table 7.1: Effectiveness of TREC 2014 runs vs. CNNTTG. Metric scores are written in bold for our CNNTTG system if it is higher than that of the original run, in underline if it is lower.

tems for each metric. In our case, we would like to know the variance of metric values across different CNNTTG thresholds compared to the original TREC runs for each metric. Note that the metric values are aggregated among different runs and we consider them as the "group" means when computing ANOVA. The results are shown in table 7.2.

Statistically significant differences in mean metric score across different systems are found for all metrics except for unweighted F_1 at $p < 0.05$. This finding supports our systems' better performances over TREC original runs for weighted recall, weighted F_1, yet worse performances for precision and unweighted recall.

We run the same experiments on 2015 TREC Real-Time Filtering Task: Period email digest. In this task, a system is required to identify interesting content on the

Metric	F-value	p-value
precision	$F(2, 24) = 17.60$	0.0001
unweighted recall	$F(2, 24) = 16.73$	0.0001
weighted recall	$F(2, 24) = 6.44$	0.0089
unweighted F_1	$F(2, 24) = 0.82$	0.4547
weighted F_1	$F(2, 24) = 3.64$	0.0432

Table 7.2: ANOVA results for each metric. Metrics displayed in bold show a statistically significant difference ($p < 0.05$) in mean metric scores across different systems.

user's interest profile and aggregate into an email digest that is periodically sent to a user. This task resembles TTG in that user's interest profile is an information need that can be considered the same as the topics in TTG task, and identifying interest content and aggregate them together is similar to clustering tweets into semantically equivalent classes. Different from TTG task, this task used nDCG@10 as the evaluation metric to measure the ability of a system to detect and remove a redundant information to a user.

We show highest nDCG@10 score from each submitted group and our CNNTTG with similarity threshold 4.0 and 4.5 measured by nDCG@10 in Table 7.3. All the original runs are sorted by nDCG@10 score. Metric scores are written in bold for our CNNTTG system if it is higher than that of the original run, and with underline if it is lower than that of the original run.

Looking at the table, we see that our system achieves higher nDCG@10 score than 12 out of 16 runs with threshold of 4.0 and higher score than 9 out of 16 runs with threshold of 4.5. While our system decreased only for run MPII_COM_MAXREP with threshold of 4.5, none of the runs decreases with threshold of 4.0. This is different from the experiments with TTG runs as it was proved that threshold of 4.5 provides better performance. And this is an interesting finding that needs further investigation. Overall speaking, by properly choosing a reasonable threshold, our CNNTTG system is able to detect and remove redundant tweets.

To strengthen the statistical analyses of system performance differences, we again performed Repeated Measures ANOVA comparing the values across different systems for nDCG@10. The results are shown in table 7.4.

Statistically significant differences in mean metric score across different systems are found for This found supports our systems' better performances over TREC original runs for nDCG@10.

7.4 Conclusion

In this chapter, we introduced our CNNTTG system that aims at removing redundant tweets given a list of relevant tweets for a topic by exploiting convolutional neural networks model to compute tweet similarity. Experiments haven shown that our system with a high threshold value is able to achieving statistically significant better performance by removing redundant tweets while keeping novel tweets compared to TREC runs. With the TTG evaluation framework as a guideline, we can

TREC runs		CNNTTG (threshold=4.0)	CNNTTG (threshold=4.5)
Run	nDCG@10	nDCG@10	nDCG@10
SNACS_LB	0.367	**0.3718**	**0.3677**
CLIP-B-0.6	0.2491	0.2491	0.2491
umd_hcil_run03	0.2471	0.2471	0.2471
PKUICSTRunB3	0.2343	**0.236**	**0.2352**
DALTREC_B_PREP	0.221	**0.2254**	**0.2228**
UWaterlooBT	0.22	0.22	0.22
MPII_COM_MAXREP	0.2093	**0.2095**	0.2092
hpclabpibm25mod	0.2046	**0.2055**	**0.205**
UNCSILS_WRM	0.2045	**0.2069**	**0.2057**
udelRun2B	0.2026	0.2026	0.2026
IRIT100KLTFIDF	0.1784	**0.1808**	**0.1786**
ECNURUNB1	0.161	**0.1741**	**0.1662**
prnaTaskB2	0.1463	**0.1492**	**0.1472**
BJUTllyQE	0.1334	**0.1359**	**0.1335**
QUBaselineB	0.1288	**0.129**	0.1288
UWCMBE1	0.1232	**0.1247**	0.1232

Table 7.3: Effectiveness of TREC 2015 runs vs. CNNTTG. Metric scores are written in bold for our CNNTTG system if it is higher than that of the original run, in underline if it is lower.

Metric	F-value	p-value
nDCG@10	$F(2, 32) = 9.14$	0.0007

Table 7.4: ANOVA results for nDCG@10. Metrics displayed in bold show a statistically significant difference ($p < 0.05$) in mean metric scores across different systems.

improve our system by aiming to achieve higher metric scores.

Chapter 8: Conclusion

This work introduced an efficient real-time tweet search system that can return a summarized list of results per query.

To achieve real-time search, we employed *selective search* and *brute force scans* techniques. In selective search: novel partitioning and temporal organization strategies support real-time cluster creation with high concentration of relevant tweets; The use of word embeddings provides document vectors in low dimension, hence reduces computing cost for clustering, and it also exhibits low variance in cluster size. Experiments have shown that by applying this selective search technique, we are able to achieve precision that's indistinguishable from exhaustive search by reducing the cost to about 12% of that of an exhaustive search. We also observe that there is no significant effectiveness difference between batch vs. online clustering algorithms, and between hourly vs. daily temporal segments.

Brute force scans technique abandons the traditional search architecture based on inverted index and searches over array representations of documents. It's easier to update incrementally than inverted index. And by making use of modern computer processors and exploiting parallelism, brute force scan approaches are better options than Lucene when the collection size is under 39 million when intra-parallelism

is exploited. When inter-parallelism is exploited, brute force scan approaches are better options than Lucene when the collection size is under 13 million.

We also introduced a tweet summarization task: Tweet Timeline Generation (TTG), and developed evaluation framework for it. By conducting user studies, we showed that the evaluation is stable with respect to different assessors and that the metrics are capable of capturing user preferences in the search results. In addition, we proposed employing deep learning techniques to accomplish this task and built our TTG system CNNTTG that is able to remove redundant tweets while keeping novel tweets compared to TREC runs.

Efficient real-time social media search has become a prominent IR application. And the desire to consume few information covering different aspects of a topic as opposed to a repeated of them has made our work meaningful. It's our hope that brute force scans technique can inspire people to shift the attention from inverted index to an entirely different approach and explore more with modern computer architectures to build efficient real-time search engines. Besides, the findings in the TTG evaluation make us believe that the evaluation reflects user preferences and this work can be applied to other cluster-based evaluations like question answering and temporal summarizations. Finally, although our system is designed to specially focus on Twitter data, we believe the techniques exploited could generalize to real-time summarization for other social media posts.

8.1 Limitations and Future Work

Aside from the contributions made by our work for the real-time search on tweets, there are limitations that we want to leave as future work:

- **Streaming Selective Search**: For shard selection in selective search, we simply chose cosine similarity and ran experiments with different k values to determine the ultimate cutoff value for all temporal segments. While in reality, the best cutoff value might vary for different collections and for different temporal segments. It is therefore necessary to develop a cutoff selection strategy that returns the smallest subset of documents without degrading effectiveness while not sophisticated so as to increase real-time search latency.

- **Brute Force Scans**: With AVX-512 extensions come into existence and is supported lately in Intel's certain models, exploiting it to further increase the performance of brute force scans is feasible, and it would be interesting to see how much improvement this can bring.

- **CNNTTG Summarization System**: The CNN model we adopted in our summarization system is complex in that it added a substantial amount of functional architecture engineering. While a simpler, deeper neural network architecture that trains on larger amount of data might outperform it, and can be used to improve our evaluation metric scores.

- **Generalization**: We only considered tweets in this work, and thus it is unclear to what extent our contributions generalize to other types of documents

like web pages, newswires and etc. Our framework for index segment organization is general, therefore can provide a starting point for future explorations of different information seeking scenarios. Yet the good performance of brute force scans depend highly on the short length of documents so it's still unclear whether longer documents like web pages can benefit from it. Further investigations and experiments are then needed to answer this question.

Bibliography

[1] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[2] Anagha Kulkarni and Jamie Callan. Selective search: Efficient and effective search of large textual collections. *ACM Transactions on Information Systems (TOIS)*, 33(4):17, 2015.

[3] Anagha Kulkarni and Jamie Callan. Topic-based index partitions for efficient and effective selective search. In *SIGIR 2010 Workshop on Large-Scale Distributed Information Retrieval*, volume 1, 2010.

[4] Jinxi Xu and W Bruce Croft. Cluster-based language models for distributed retrieval. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261. ACM, 1999.

[5] Anagha Kulkarni and Jamie Callan. Document allocation policies for selective searching of distributed indexes. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 449–458. ACM, 2010.

[6] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 12:1532–1543, 2014.

[7] Yulu Wang and Jimmy Lin. Partitioning and segment organization strategies for real-time selective search on document streams. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 2017.

[8] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and James Lin. Earlybird: Real-time search at twitter. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1360–1369. IEEE, 2012.

[9] Yulu Wang and Jimmy Lin. The feasibility of brute force scans for real-time tweet search. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, pages 321–324. ACM, 2015.

[10] Yulu Wang, Garrick Sherman, Jimmy Lin, and Miles Efron. Assessor differences and user preferences in tweet timeline generation. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 615–624. ACM, 2015.

[11] Jennifer Klein, Yishai Oltchik, Nerya Or, and Sara Cohen. Hu_db at trec 2014 microblog track. Technical report, HEBREW UNIV JERUSALEM (ISRAEL), 2014.

[12] Guoxin Cui, Fabin Shi, Xiaolei Liu, Xiaobo Hao, Xueke Xu, Yue Liu, and Xueqi Cheng. Ictnet at microblog track in trec 2014. Technical report, CHINESE ACADEMY OF SCIENCES BEIJING INST OF COMPUTING TECHNOLOGY, 2014.

[13] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[14] Xuanhui Wang and ChengXiang Zhai. Learn from web search logs to organize search results. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 87–94. ACM, 2007.

[15] Chao Lv, Feifan Fan, Runwei Qiang, Yue Fei, and Jianwu Yang. Pkuicst at trec 2014 microblog track: feature extraction for effective microblog search and adaptive clustering algorithms for ttg. Technical report, PEKING UNIV BEIJING (CHINA), 2014.

[16] Walid Magdy, Wei Gao, Tarek Elganainy, and Zhongyu Wei. Qcri at trec 2014: applying the kiss principle for the ttg task in the microblog track. Technical report, QATAR COMPUTING RESEARCH INST DOHA, 2014.

[17] Maram Hasanain and Tamer Elsayed. Qu at trec-2014: online clustering with temporal and topical expansion for tweet timeline generation. Technical report, QATAR UNIV DOHA, 2014.

[18] Hila Becker, Mor Naaman, and Luis Gravano. Beyond trending topics: Real-world event identification on twitter. *ICWSM*, 11(2011):438–441, 2011.

[19] Kuang Lu, Diego Roa, and Hui Fang. Concept based tie-breaking and maximal marginal relevance retrieval in microblog retrieval. Technical report, DELAWARE UNIV NEWARK DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 2014.

[20] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336. ACM, 1998.

[21] Hua He, Kevin Gimpel, and Jimmy Lin. Multi-perspective sentence similarity modeling with convolutional neural networks.

[22] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[23] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[24] Gerard Salton. The use of citations as an aid to automatic content analysis. *Information Storage and Retrieval, Report ISR-2, Harvard Computation Laboratory, in preparation*, 1962.

[25] Evan L Ivie. Search procedures based on measures of relatedness between documents. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1966.

[26] Stephen E Robertson. The probability ranking principle in ir. *Journal of documentation*, 33(4):294–304, 1977.

[27] W Bruce Croft and David J Harper. Using probabilistic models of document retrieval without relevance information. *Journal of documentation*, 35(4):285–295, 1979.

[28] Norbert Fuhr and Chris Buckley. A probabilistic learning approach for document indexing. *ACM Transactions on Information Systems (TOIS)*, 9(3):223–248, 1991.

[29] Norbert Fuhr. Probabilistic models in information retrieval. *The Computer Journal*, 35(3):243–255, 1992.

[30] Norbert Fuhr. Probabilistic datalog—a logic for powerful retrieval methods. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 282–290. ACM, 1995.

[31] Warren R Greiff, W Bruce Croft, and Howard Turtle. Computationally tractable probabilistic modeling of boolean operators. In *ACM SIGIR Forum*, volume 31, pages 119–128. ACM, 1997.

[32] Djoerd Hiemstra. A linguistically motivated probabilistic model of information retrieval. *Research and advanced technology for digital libraries*, pages 515–515, 1998.

[33] Melvin Earl Maron and John L Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM (JACM)*, 7(3):216–244, 1960.

[34] Clement T Yu and Gerard Salton. Precision weighting—an effective automatic indexing method. *Journal of the ACM (JACM)*, 23(1):76–88, 1976.

[35] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *Nist Special Publication Sp*, 109:109, 1995.

[36] K Sparck Jones, Steve Walker, and Stephen E. Robertson. A probabilistic model of information retrieval: development and comparative experiments: Part 2. *Information processing & management*, 36(6):809–840, 2000.

[37] Jay M Ponte and W Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 275–281. ACM, 1998.

[38] Fei Song and W Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321. ACM, 1999.

[39] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.

[40] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[41] Chengxiang Zhai and John Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 334–342. ACM, 2001.

[42] Donna Harman and Gerald Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581, 1990.

[43] Alistair Moffat and Timothy AH Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537, 1995.

[44] Edward A Fox and Whay C Lee. Fast-inv: A fast algorithm for building large inverted files. 1991.

[45] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.

[46] Robert WP Luk and Wai Lam. Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754, 2007.

[47] Nima Asadi, Jimmy Lin, and Michael Busch. Dynamic memory allocation policies for postings in real-time twitter search. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1186–1194. ACM, 2013.

[48] Alistair Moffat and Justin Zobel. Parameterised compression for sparse bitmaps. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 274–285. ACM, 1992.

[49] Berthier Ribeiro-Neto, Edleno S Moura, Marden S Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 105–112. ACM, 1999.

[50] David Hawking. Scalable text retrieval for large digital libraries. *Research and Advanced Technology for Digital Libraries*, pages 127–145, 1997.

[51] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[52] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[53] Andrew MacFarlane, Julie A McCann, and Stephen E Robertson. Parallel search using partitioned inverted files. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 209–220. IEEE, 2000.

[54] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on*, pages 8–17. IEEE, 1993.

[55] Fidel Cacheda, Vassilis Plachouras, and Iadh Ounis. Performance analysis of distributed architectures to index one terabyte of text. In *ECIR*, pages 394–408. Springer, 2004.

[56] Nicholas Lester, Justin Zobel, and Hugh E Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 15–23. Australian Computer Society, Inc., 2004.

[57] Ricardo Baeza-Yates and William Bruce Frakes. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[58] Ian H Witten, Alistair Moffat, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[59] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Information processing & management*, 42(4):916–933, 2006.

[60] Stefan Büttcher and Charles LA Clarke. Indexing time vs. query time: tradeoffs in dynamic information retrieval systems. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 317–318. ACM, 2005.

[61] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783. ACM, 2005.

[62] Nicholas Lester, Alistair Moffat, and Justin Zobel. Efficient online index construction for text databases. *ACM Transactions on Database Systems (TODS)*, 33(3):19, 2008.

[63] Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu, and Kunmei Wen. Efficient online index maintenance for ssd-based information retrieval systems. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 262–269. IEEE, 2012.

[64] Giorgos Margaritis and Stergios V Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 455–464. ACM, 2009.

[65] Anthony Tomasic, Hector Garcia-Molina, and Kurt Shoens. *Incremental updates of inverted lists for text document retrieval*, volume 23. ACM, 1994.

[66] Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *Information processing & management*, 41(2):275–288, 2005.

[67] Eric W Brown, James P Callan, and W Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB*, volume 94, page 9. Citeseer, 1994.

[68] Stefan Büttcher, Charles LA Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363. ACM, 2006.

[69] Nima Asadi and Jimmy Lin. Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 997–1000. ACM, 2013.

[70] Trevor Strohman and W Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 175–182. ACM, 2007.

[71] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[72] Gaston H Gonnet. *Examples of PAT applied to the Oxford English Dictionary*, volume 87. UW Centre for the New Oxford English Dictionary, 1987.

[73] Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. *Information Retrieval: Data Structures & Algorithms*, 66:82, 1992.

[74] Ricardo Baeza-Yates, Eduardo F Barbosa, and Nivio Ziviani. Hierarchies of indices for text searching. *Information Systems*, 21(6):497–514, 1996.

[75] Eduardo Barbosa, Gonzalo Navarro, Ricardo Baeza-Yates, Chris Perleberg, and Nivio Ziviani. Optimized binary search and text retrieval. pages 311–326, 1995.

[76] Christos Faloutsos and Stavros Christodoulakis. Description and performance analysis of signature file methods for office filing. *ACM Transactions on Information Systems (TOIS)*, 5(3):237–257, 1987.

[77] Christos Faloutsos and Raphael Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *VLDB*, volume 88, pages 280–293, 1988.

[78] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. Bitfunnel: Revisiting signatures for search. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 605–614. ACM, 2017.

[79] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[80] Djoerd Hiemstra. *Using language models for information retrieval*. Taaluitgeverij Neslia Paniculata, 2001.

[81] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[82] Bruce Jacob. The memory system: you can't avoid it, you can't ignore it, you can't fake it. *Synthesis Lectures on Computer Architecture*, 4(1):1–77, 2009.

[83] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB" 99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number DIAS-CONF-1999-001, pages 266–277, 1999.

[84] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156. ACM, 2002.

[85] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[86] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J Haas, and Guy M Lohman. Main-memory scan sharing for multi-core cpus. *Proceedings of the VLDB Endowment*, 1(1):610–621, 2008.

[87] Janey K Cringean, Roger England, Gordon A Manson, and Peter Willett. Parallel text searching in serial files using a processor farm. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 429–453. ACM, 1989.

[88] Brendon Cahoon, Kathryn S McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems (TOIS)*, 18(1):1–43, 2000.

[89] Jamie Callan. Distributed information retrieval. In *Advances in information retrieval*, pages 127–150. Springer, 2000.

[90] Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Design of a parallel and distributed web search engine. *arXiv preprint cs/0407053*, 2004.

[91] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10(3):205–231, 2007.

[92] Diego Puppin, Fabrizio Silvestri, Raffaele Perego, and Ricardo Baeza-Yates. Tuning the capacity of search engines: Load-driven routing and incremental caching to reduce and balance the load. *ACM Transactions on Information Systems (TOIS)*, 28(2):5, 2010.

[93] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[94] Gerard Salton. The smart retrieval system — experiments in automatic document processing. 1971.

[95] Nick Jardine and Cornelis Joost van Rijsbergen. The use of hierarchic clustering in information retrieval. *Information storage and retrieval*, 7(5):217–240, 1971.

[96] Diego Puppin, Fabrizio Silvestri, and Domenico Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st international conference on Scalable information systems*, page 34. ACM, 2006.

[97] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. The effectiveness of gioss for the text database discovery problem. In *ACM SIGMOD Record*, volume 23, pages 126–137. ACM, 1994.

[98] James P Callan, Zhihong Lu, and W Bruce Croft. Searching distributed collections with inference networks. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–28. ACM, 1995.

[99] Luis Gravano and Hector Garcia-Molina. Generalizing gloss to vector-space databases and broker hierarchies. 1999.

[100] Luis Gravano, Héctor García-Molina, and Anthony Tomasic. Gloss: text-source discovery over the internet. *ACM Transactions on Database Systems (TODS)*, 24(2):229–264, 1999.

[101] Luo Si and Jamie Callan. Relevant document distribution estimation method for resource selection. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 298–305. ACM, 2003.

[102] Paul Thomas and Milad Shokouhi. Sushi: scoring scaled samples for server selection. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 419–426. ACM, 2009.

[103] Jaime Arguello, Jamie Callan, and Fernando Diaz. Classification-based resource selection. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1277–1286. ACM, 2009.

[104] Anagha Kulkarni, Almer S Tigelaar, Djoerd Hiemstra, and Jamie Callan. Shard ranking and cutoff estimation for topically partitioned collections. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 555–564. ACM, 2012.

[105] Milad Shokouhi. Central-rank-based collection selection in uncooperative distributed information retrieval. In *ECiR*, volume 7, pages 160–172. Springer, 2007.

[106] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[107] Chris Buckley and Ellen M Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 33–40. ACM, 2000.

[108] Robin Aly, Djoerd Hiemstra, and Thomas Demeester. Taily: shard selection using the tail of score distributions. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 673–682. ACM, 2013.

[109] Trevor Strohman, Howard Turtle, and W Bruce Croft. Optimization strategies for complex queries. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 219–225. ACM, 2005.

[110] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[111] Stefanie Tellex, Boris Katz, Jimmy Lin, Aaron Fernandes, and Gregory Marton. Quantitative evaluation of passage retrieval algorithms for question answering. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 41–47. ACM, 2003.

[112] Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[113] Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. Parsing with compositional vector grammars. In *In Proceedings of the ACL conference*. Citeseer, 2013.

[114] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JAsIs*, 41(6):391–407, 1990.

[115] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.

[116] Charles L Wayne. Multilingual topic detection and tracking: Successful research enabled by corpora and evaluation. 2000.

[117] James Allan. *Topic detection and tracking: event-based information organization*, volume 12. Springer Science & Business Media, 2012.

[118] Jonathan G Fiscus and George R Doddington. Topic detection and tracking evaluation overview. In *Topic detection and tracking*, pages 17–31. Springer, 2002.

[119] Ellen Voorhees. Overview of the trec 2004 question answering track. In *TREC*, 2004.

[120] Hoa Dang, Jimmy Lin, and Diane Kelly. Overview of the trec 2006 question answering track. In *TREC*, 2006.

[121] Hoa Dang and Jimmy Lin. Different structures for evaluating answers to complex questions: Pyramids won't topple, and neither will human assessors. In *ACL*, 2007.

[122] Qi Guo, Fernando Diaz, and Elad Yom-Tov. Updating users about time critical events. In *Advances in Information Retrieval*, pages 483–494. Springer, 2013.

[123] Javed Aslam, Fernando Diaz, Matthew Ekstrand-Abueg, Richard McCreadie, Virgil Pavlu, and Tetsuya Sakai. Trec 2014 temporal summarization track overview. Technical report, DTIC Document, 2015.

[124] Peter Bailey, Nick Craswell, Ian Soboroff, Paul Thomas, Arjen P de Vries, and Emine Yilmaz. Relevance assessment: are judges exchangeable and does it matter. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 667–674. ACM, 2008.

[125] Ellen M Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. *Information processing & management*, 36(5):697–716, 2000.

[126] Jimmy Lin and Pengyi Zhang. Deconstructing nuggets: the stability and reliability of complex question answering evaluation. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 327–334. ACM, 2007.

[127] Virgil Pavlu, Shahzad Rajput, Peter B Golbus, and Javed A Aslam. Ir system evaluation using nugget-based test collections. In *Proceedings of the fifth ACM international conference on Web search and data mining*, pages 393–402. ACM, 2012.

[128] William Hersh, Andrew Turpin, Susan Price, Benjamin Chan, Dale Kramer, Lynetta Sacherek, and Daniel Olson. Do batch and user evaluations give the same results? In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 17–24. ACM, 2000.

[129] Andrew H Turpin and William Hersh. Why batch and user evaluations do not give the same results. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 225–231. ACM, 2001.

[130] James Allan, Ben Carterette, and Joshua Lewis. When will information retrieval be good enough? In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 433–440. ACM, 2005.

[131] Andrew Turpin and Falk Scholer. User performance versus precision measures for simple search tasks. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 11–18. ACM, 2006.

[132] Scott B Huffman and Michael Hochster. How well does result relevance predict session satisfaction? In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 567–574. ACM, 2007.

[133] Azzah Al-Maskari, Mark Sanderson, Paul Clough, and Eija Airio. The good and the bad system: does the test collection predict users' effectiveness? In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 59–66. ACM, 2008.

[134] Catherine L Smith and Paul B Kantor. User adaptation: good results from poor systems. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 147–154. ACM, 2008.

[135] Mark Sanderson, Monica Lestari Paramita, Paul Clough, and Evangelos Kanoulas. Do user preferences and evaluation measures line up? In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 555–562. ACM, 2010.

[136] Mark D Smucker and Chandra Prakash Jethani. Human performance and retrieval precision revisited. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 595–602. ACM, 2010.

[137] Jimmy Lin and Mark D Smucker. How do users find things with pubmed?: towards automatic utility evaluation with user simulations. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 19–26. ACM, 2008.

[138] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

[139] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[140] Yoshua Bengio, Aaron Courville, and Pierre Vincent. Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828, 2013.

[141] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[142] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[143] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[144] Hugo Larochelle, Michael Mandel, Razvan Pascanu, and Yoshua Bengio. Learning algorithms for the classification restricted boltzmann machine. *The Journal of Machine Learning Research*, 13(1):643–669, 2012.

[145] Yichuan Tang. Deep learning using support vector machines. *CoRR, abs/1306.0239*, 2013.

[146] Gang Chen. Deep learning with nonparametric clustering. *arXiv preprint arXiv:1501.03084*, 2015.

[147] Stephen Wan, Mark Dras, Robert Dale, and Cécile Paris. Using dependency-based features to take the 'para-farce' out of paraphrase. In *Proceedings of the Australasian Language Technology Workshop*, volume 2006, 2006.

[148] Nitin Madnani, Joel Tetreault, and Martin Chodorow. Re-examining machine translation metrics for paraphrase identification. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 182–190. Association for Computational Linguistics, 2012.

[149] Samuel Fernando and Mark Stevenson. A semantic similarity approach to paraphrase detection. In *Proceedings of the 11th Annual Research Colloquium of the UK Special Interest Group for Computational Linguistics*, pages 45–52, 2008.

[150] Dipanjan Das and Noah A Smith. Paraphrase identification as probabilistic quasi-synchronous recognition. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*, pages 468–476. Association for Computational Linguistics, 2009.

[151] Samer Hassan. *Measuring semantic relatedness using salient encyclopedic concepts*. University of North Texas, 2011.

[152] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809, 2011.

[153] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050, 2014.

[154] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[155] Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *SemEval@ COLING*, pages 1–8, 2014.

[156] Wenpeng Yin and Hinrich Schütze. Convolutional neural network for paraphrase identification. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 901–911, 2015.

[157] Bill Dolan, Chris Quirk, and Chris Brockett. Unsupervised construction of large paraphrase corpora: Exploiting massively parallel news sources. In *Proceedings of the 20th international conference on Computational Linguistics*, page 350. Association for Computational Linguistics, 2004.

[158] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[159] Jimmy Lin, Matt Crane, Andrew Trotman, Jamie Callan, Ishan Chattopadhyaya, John Foley, Grant Ingersoll, Craig Macdonald, and Sebastiano Vigna. Toward reproducible baselines: The open-source ir reproducibility challenge. In *European Conference on Information Retrieval*, pages 408–420. Springer, 2016.

[160] Iadh Ounis, Craig Macdonald, Jimmy Lin, and Ian Soboroff. Overview of the trec-2011 microblog track. In *Proceeddings of the 20th Text REtrieval Conference (TREC 2011)*, 2011.

[161] Ian Soboroff, Iadh Ounis, J Lin, and I Soboroff. Overview of the trec-2012 microblog track. In *Proceedings of TREC*, volume 2012, 2012.

[162] Jimmy Lin and Miles Efron. Overview of the trec-2013 microblog track. In *Proceeddings of the 22nd Text REtrieval Conference (TREC 2013)*, 2013.

[163] Marcel R Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. Streamkm++: A clustering algorithm for data streams. *Journal of Experimental Algorithmics (JEA)*, 17:2–4, 2012.

[164] Nir Ailon, Ragesh Jaiswal, and Claire Monteleoni. Streaming k-means approximation. In *Advances in Neural Information Processing Systems*, pages 10–18, 2009.

[165] Vladimir Braverman, Adam Meyerson, Rafail Ostrovsky, Alan Roytman, Michael Shindler, and Brian Tagiku. Streaming k-means on well-clusterable data. In *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms*, pages 26–40. SIAM, 2011.

[166] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Computing Surveys (CSUR)*, 46(1):13, 2013.

[167] Sasa Petrovic, Miles Osborne, and Victor Lavrenko. The edinburgh twitter corpus. 2010.

[168] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine learning in Apache Spark. In *arXiv:1505.06807v1*, 2015.

[169] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[170] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[171] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.

[172] Yulu Wang and Jimmy Lin. The impact of future term statistics in real-time tweet search. In *Advances in Information Retrieval*, pages 567–572. Springer, 2014.

[173] Zhuyun Dai, Yubin Kim, and Jamie Callan. How random decisions affect selective distributed search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 771–774. ACM, 2015.

[174] Joseph L Fleiss, Jacob Cohen, and BS Everitt. Large sample standard errors of kappa and weighted kappa. *Psychological Bulletin*, 72(5):323, 1969.