Proceedings of the Australasian Language Technology Workshop 2007, pages 40-48

Practical Queries of a Massive n-gram Database

Tobias Hawker
School of
Information Technologies
University of Sydney
NSW 2006, Australia
toby@it.usyd.edu.au

Mary Gardiner
Centre for Language Technology
Macquarie University
gardiner@ics.mq.edu.au

Andrew Bennetts
Canonical Ltd.
andrew@puzzling.org

Abstract

Large quantities of data are an increasingly essential resource for many Natural Language Processing techniques. The Web 1T corpus, a massive resource containing n-gram frequencies produced from one trillion words drawn from the World Wide Web, is a relatively new corpus whose size will increase performance on many data-hungry applications. In addition, a fixed resource of this kind reduces reliance on using web results as experimental data, increasing replicability of researchers' results.

However, effectively utilising a resource of this size presents significant challenges. We discuss the challenges of using a data source of this magnitude, and describe strategies for overcoming these, including efficient extraction of queries including wildcards, and specialised data compression. We present a software suite, "Get 1T", implementing these techniques, released as free software for use by the natural language research community, and others.

1 Introduction

The size and quality of data used for statistical Natural Language Processing can have a major impact on the results a system achieves (Banko and Brill, 2001). The World Wide Web gives researchers access to an unprecedented quantity of machine-readable natural language text. However, even for techniques that can take advantage of unannotated text, there is much more web data available than can normally be used, because this same size overwhelms the processing resources available to most researchers. One way of overcoming this practical problem is to use the processing resources of commercial search engines, by using the number of hits they report for frequency estimation — see for example Grefenstette (1999), Turney (2001), Keller and Lapata (2003) and Nakov and Hearst (2005). There are however several drawbacks to this approach, recently highlighted by Kilgarriff (2007), particularly replicability of experiments.

The recent release of the Web 1T corpus (Brants and Franz, 2006)[1] presents an opportunity to access web-scale data for n-gram frequency estimation without the drawbacks of using a search engine. The corpus provides frequency counts for n-grams up to five tokens long, drawn from approximately 1 trillion words of web data. Promising results have already been achieved using this resource for Word Sense Disambiguation and Lexical Substitution (Hawker, 2007; Yuret, 2007) and for Noun-Phrase Bracketing (Vadas and Curran, 2007).

However, the scale of even this distilled collection of web data presents significant processing challenges. Naïve methods for extracting the desired information from the corpus, such as linear search or keyword indexing, are hopelessly impractical, and even algorithms with good asymptotic performance such as binary search are limited in their applicability. Approaches that attempt to overcome the scaling problems by using indices for the set of discrete to-

[1] http://www.ldc.upenn.edu/Catalog/docs/LDC2006T13/readme.txt

kens in the corpus are also rendered intractable by the size of the vocabulary. In this paper we present solutions to these difficulties of scale, strategies for practical extraction of the items of interest from this mountain of data.

1.1 Wild Cards

For many NLP applications, it is useful to be able to discover not only the frequency of explicitly specified patterns, but also the frequency of patterns where any token is permissible in certain locations. These 'wildcard' queries allow, for example, the determination of Point-wise Mutual Information in Hawker (2007).

As an example, consider the trigram (from the corpus): *feet seem light*. If we are interested in how likely the word *seem* is to occur in this context, we must not only find frequencies for *feet seem light* but also entries for *feet so light* and *feet the light*. In the notation we use for our queries, we can capture this idea of finding the aggregate frequency for all trigrams with *feet* in the first position and *light* in the third with the query `feet <*> light`. Note that to find the aggregate frequency for *all* matching trigrams, we must employ a search strategy which will indicate a match for *any* suitable string.

In the remainder of this paper, we briefly describe the details of the Web 1T corpus and consider and then rule out possible approaches to using it that are not practical at this scale. We then introduce two practical approaches that allow for the extraction of only the information of interest including one that permits the use of wildcards. Our software tool, "Get 1T", released to the community as free software, has implemented these approaches.

2 Web 1T

The Web 1T corpus (Brants and Franz, 2006) is a collection of n-grams and their frequency of occurrence as found in 1,024,908,267,229 tokens (approximately 1 trillion) comprising 95,119,665,584 sentences of text from publicly accessible web pages. The corpus aims to cover unique pages in English. It is filtered to exclude tokens such as those with problematic encoding, token length, large quantities of accented characters, and unprintable/control characters.

n	distinct n-grams
1	13,588,391
2	314,843,401
3	977,069,902
4	1,313,818,354
5	1,176,470,663

Table 1: Number of distinct patterns

The n-grams counted cover patterns of tokens from unigrams to 5-grams. Tokens with case differences are treated as distinct. The collection is filtered by a cut-off frequency of 200 occurrences for unigrams, and 40 occurrences for bigrams to 5-grams. There are n-grams in the corpus that contain a token with fewer than 200 unigram instances — those tokens within these n-grams are represented with a special token (<UNK>). Sentence boundaries are also included as tokens, denoted by <S> and </S>, thus for example a 5-gram might contain 4 words and a sentence boundary.

The number of unique patterns arising from this collection is itself very large, as shown in Table 1.

With the patterns and counts stored in text format, one per line, and then compressed, the frequency collection occupies around 25GB of disk space, and is distributed by the LDC on 6 DVDs.

2.1 Comparison with Search Engine Hits

Recently, Kilgarriff (2007) has raised issues surrounding the use of search engine counts for determining relative frequencies of various instances of natural language. Owing to the commercially sensitive and competitive nature of search engine strategies, the process behind the counts reported by search engines is not generally known, and the data on the web changes over time; thus the counts reported are potentially unreliable. Changes may occur in the algorithm, query syntax, or even using the same search engine for identical queries on consecutive days — Kilgarriff (2007) found the results from 6 in 30 queries differed by at least a factor of two on consecutive days, and speculates that this may be due to queries being serviced by different computers at different parts of the update cycle. This results in experiments that may be impossible to replicate, even by one researcher over time, let alone by others.

Aside from replicability concerns, This may account for some findings that search results are less reliable than corpus counts in, for example, context-sensitive spelling correction (Liu and Curran, 2006).

By contrast, the approach for constructing the Web 1T corpus has been disclosed (Brants and Franz, 2006), and is intentionally aimed at yielding accurate n-gram models, rather than estimating the popularity of query keywords for a web search designed to locate pages. Since Web 1T is distributed to researchers, there are no limits on the number of queries that can be performed in a given time frame aside from any imposed by resource limitations of the researcher. The values for different queries are directly comparable, as they cover the same corpus. Importantly, experiments are entirely repeatable, as the results of queries using a specific release of the corpus do not change.

The n-gram database approach is not without a few drawbacks however. Snippets giving broader context than the maximum 5-token n-gram size are not available. The original source of a document, useful for such things as topic determination, is lost. Parsing such short fragments is, in general, not likely to be reliable. POS tagging is possible on the longer n-gram sequences, though is likely to be somewhat less reliable than typical POS tagging performance on full sentences.

Another serious drawback is that the resources required for performing queries are not provided on massively redundant hardware at essentially zero cost, as is the case for commercial search engines. This requires researchers to query the data on their own hardware using software that is able to handle this quantity of data efficiently.

This paper aims to detail approaches that permit practical use of the Web 1T corpus. We have released free software tools implementing these approaches to the community[2] under the GNU GPL.

3 Infeasible Approaches to Processing

In the following discussion, the question of accessing the data will be framed in terms of *queries* — particular n-gram patterns or abstractions thereof, such as `feet so light` or `feet <*> light`, whose count is desired from the corpus, much as would be submitted to a search engine API.

In working with a corpus at this scale, it is undesirable access the disk to retrieve data that is not useful for the application at hand, and very problematic to consider the same data multiple times. Any strategy for querying the corpus must thus attempt to avoid reading unneeded data, and where it must, make as few passes over the data as possible.

3.1 Direct Queries

One approach to extracting queries of interest might be to take advantage of the fact that the n-gram patterns are not listed arbitrarily, but are sorted into a lexicographically ascending order. This enables the use of a binary search to find a query without having to consider most entries in the database. As binary search complexity increases only with the logarithm of the number of database entries, it seems reasonable that this method might scale better to large quantities of data than any linear approach.

The Web 1T corpus data is split into a single directory for each n-gram length (1–5 tokens). The patterns are sorted in lexicographically ascending order, and split into files of at most 10,000,000 entries, individually compressed. An index file specifies the first n-gram in each file, which can also be used as a guide to the endpoint of the previous file.

Binary search for a small number of queries is tractable without the prohibitive requirement of storing all n-grams in RAM. The index files fit easily in RAM however, and finding which n-gram file to search depends on the number of unique patterns for that length (see Table 1). For 4-grams (the worst case), this leads to an average of about $\log_2 132$, or 7 iterations per query.

There are 10^7 n-grams in each compressed file, which leads to an average of $\log_2 10^7$, or about 23, tests of location per exact query. Many parts of the file must be retrieved to find each element. Each of these accesses requires computational effort (which in many cases will require a disk seek) and as the files do not have fixed record size, there is an additional burden of finding line-breaks before the actual n-gram record can be determined for comparison against the desired query. The string comparisons required are also computationally expensive.

For even a moderate number of queries (more than a few hundred), reads from all over the disk to

[2] Available to the public at `http://get1t.sf.net/`

retrieve each query are very inefficient. It is impractical to store the entire corpus in RAM, at least for commodity resources typically available. The corpus is of the order of 80GB uncompressed, which is well beyond RAM resources typically available to researchers at the time of writing.

Furthermore, case-insensitive queries are very problematic, as case differences may occur at any point in the string. In ASCII, the two cases of a letter, upper and lower, are lexicographically distant (values differ by 32), and thus otherwise identical queries may be separated by many entries or be in entirely different files. One potential solution is to re-generate the corpus for case-insensitive use. This is possible, and need only be performed once, but increases the storage burden, and while sorting data of this size is possible using existing techniques (for example, mergesort) it would not be a small task for typical resources by any means.

However, the most difficult drawback to overcome with binary searching is the desirability of performing queries with wildcards — queries whose strings are not specified exactly, but may match any tokens at some locations. Unless the wild component is near the end of the pattern (which permits the use of binary searches to select a subspace of the original, which can then be exhaustively searched) binary searches are completely impractical.

3.2 Indexing

Another possible approach to extracting queries from the data might be the use of a mechanism that takes advantage of the fact that the n-gram patterns of length 2–5 are comprised of tokens from the finite set of unigrams. Unfortunately, this approach too is very difficult for data on the scale of the Web 1T data. There are more than 13 million unigrams in the corpus, and assigning a unique binary value to each one would thus require 24 bits per token. Attempting to index the corpus using indices of 24 bits and only a single byte at each indexed location would require in excess of 100 TB for bigrams, with almost all locations empty.

We measured the entropy per token in the unigram frequencies, and determined there were 10.7 bits per token for case-sensitive measurement, and 10.3 bits per token for the case-insensitive case. However, while entropy coding allows the *average* number of bits per token to be brought closer to these ideal representations, some n-grams with combinations of infrequent tokens would have representations that far exceed this average size.

4 Practical Approaches to Processing

This section describes methods that render querying the Web 1T corpus practical by relying on the insight that the corpus should be read from disk as few times as possible — preferably once. Thus, our strategies require that pre-processing occur before the frequencies for the desired queries are extracted from the corpus. One of two pre-processing steps is required. A researcher might use our tool to pre-process the corpus, permitting quick, on-the-fly queries, at the price of only having approximate counts, and of false positive counts for some queries not present in the corpus. Or he or she might be able to specify queries in advance, in case we can compute exact counts in a single pass through the corpus.

4.1 Pre-processing the Corpus

This strategy makes queries of the corpus practical by compressing the massive quantity of data to a manageable level. This is achieved in two ways: by reducing the resolution of the frequency information and then by replacing the n-gram strings themselves with implicit information based on the location in the compressed data. Depending on the task, many statistical approaches can handle some noise in the data; although fidelity may be somewhat reduced, the impact is not all negative — it may also smooth.

We thus construct a single monolithic binary file for each length of n-gram. Instead of storing the exact string of the n-gram alongside its count, as in the original data, we stored the quantised set of frequencies at *locations* determined by a hash value of the n-grams concerned.

For many applications, the absolute resolution of the frequency information used is not as important as the magnitude. In particular, in many statistical approaches the absolute difference between two quantities will not be as informative as the ratio: the difference between a pattern occurring 500,000 and 500,050 times is much less significant that the difference between patterns occurring 50 and 100 times. It is thus desirable to have the resolution of

the frequency approximately constant with respect to magnitude, thus preserving the dynamic range of all frequencies. A suitable transformation is to store a value corresponding to the logarithm of the frequency. If this value is then quantised, a great deal of compression is possible. For implementation on practical hardware, an integer number of bytes is preferable. Using a single byte for each frequency allows 255 discrete magnitude levels to be encoded.

The frequencies of patterns in the data have a very large range, from 40 to 95,119,665,584. The quantisation is performed with a logarithm base determined by the maximum count. A zero byte represents an unseen n-gram (frequency of zero) and the remaining 255 quantisation levels are selected such that their span is uniform in the logarithmic space. The use of this logarithmic scale means that error introduced by the quantisation is roughly proportional to the magnitude of the frequency being quantised. Using zero bytes for unseen n-grams simply requires the initialisation of all locations to zero before beginning the compression process.

To permit optimal use of the relatively small quantised representation, the quantisation ranges were determined separately for each value of n, ensuring the maximum count for that number of tokens was transformed to the maximum integer value. As the quantised values are converted back to approximate counts in linear space when being reported, this scaling ensures acceptable precision for each quantisation scaling while still yielding comparable frequencies, even between different n-gram types.

When translating quantised values back to approximate frequencies, such as when forming features from the frequencies, all instances of a given quantised value are interpreted as the geometric mean of the boundaries of the range of integers that fall within it. This is also the arithmetic mean of the logarithms concerned.

There is also a choice to be made when a collision is found during compression — when an n-gram hashes to a value that is already non-zero in the compressed file. At this point there are several options: the existing value can be combined with the new value, such as being added together; a choice can be made between the two values, such as the smaller or larger, or the existing or new value; or a compromise value, such as the geometric mean of the two values, can be stored. Which mechanism is suitable depends on the downstream task being performed. The implementation allows this choice to be made by the user at the time the data is compressed.

For a given file size, there is a trade-off between the resolution of the frequency information and the number of unique keys, which in turn influences the probability of collisions. For example, counts can be made finer resolution by using two bytes per file location. This is at the expense of the hash function, which will thus have one fewer bits of key space for data files of the same size.

4.1.1 Retrieval

To perform a query, the desired search string is hashed, the quantised value retrieved at the location indicated by the hash, and transformed back into a linear count. Practical hash sizes are around 30–32 bit hashes, which yield indices of 1–4GB. The size of the hash is constrained by the fact that it must fit in RAM for building the hash to be tractable. Uniform hashing necessarily involves a pseudo-random sequence of locations. This implies that if RAM were insufficient for the chosen hash size, disk seeks would be frequent, and impact very strongly on performance. It would be possible to build parts of larger hash files in RAM, using several passes over the data, but this requires 2^n passes for each additional n bits of hash, which would also quickly becomes prohibitive.

4.1.2 Properties

This strategy cannot yield a false negative result for a query — a zero count when the queried pattern does in fact occur. However, false positives are possible due to hash collisions and can become problematic. The collisions are due to there being an infinite number of possible patterns that would map to each location, yet the hash value being finite.

This approach is thus not suited to wildcards at all. Uniform hashing functions with finite hash values are one-way, and deliberately yield different hash values for inputs that differ only slightly. Thus the set of possible matches must be generated as inputs to the hashing function, and so each hypothetical match must be generated and treated as a query. The number of possible matches for any under-specified sequence is enormous, and even if

only a very small fraction of these queries results in false positives, the sheer number of queries required quickly renders the combined effect insurmountable.

4.1.3 Implementation

This software compresses each n-gram pattern length into a single file whose size in bytes is a power of two. The files may be read into RAM and queried in a random-access fashion, or counts can be retrieved directly from the disk.

The uniform distribution property of the hash function, desirable in minimising the number of hash collisions becomes problematic if accessing the compressed data directly from disk. If the n-gram frequencies are retrieved in the order they are used by the system, the hashing transformation renders the requests into a sequence of pseudo-random disk accesses, leading the disk heads to skip back and forth repeatedly. One possible solution is to map the file into memory for access. However, for a finite set of queries this too is sub-optimal, as many blocks of data must be read from disk that will not be used.

If however, the queries are collated, hashed, and sorted based upon their hash value, the retrieval software never needs to re-read or seek backwards. Even more advantageously, blocks not containing patterns of interest do not have to be read at all. For even a large number of queries this process is very rapid.

Python bindings have also been created for accessing files created using this approach. Disk access times are generally greater than the CPU work for retrieval, and the hashing itself is performed in a native C extension, so the performance penalty involved using an interpreted language is minor.

4.2 Pre-processing Queries

The approach described in this section does not include any approximations or false-positive counts, and allows queries that will match any token in specified positions. The implementation of this approach has been used to perform successful experiments in Word Sense Disambiguation and Lexical Substitution (Hawker, 2007). The price for these desirable properties is that all queries must be collated before the corpus information is extracted; on-the-fly queries are not possible.

The key idea in our approach is the reversal of the target of the search — from the database to the queries themselves. To adapt a well-known metaphor, we can view each query as a needle of a particular size and shape, and the Web 1T corpus as an enormous haystack, possibly containing a piece of hay that resembles the size and shape of the needle (i.e. an entry matching the query). In this metaphor, taking each desired needle individually and trying to find matches in the enormous haystack is clearly an insurmountable task for any large-scale haystack. Our method involves collecting all needles in which we are interested before any search is performed, and then considering each strand in the haystack in turn, finding whether it matches any of the needles. This is still an intensive task, but it is certainly tractable, as each piece of hay need be considered only once.

4.2.1 Search Strategy

Queries are provided to the program as input when the search is launched. Queries are stored in a nested hashtable, where the key to the hash table at any given level i is the i-th word in the query. For example, the query `frozen hell buckets` will be indexed under *frozen* at the first level, *hell* at the second with other queries beginning with `frozen` and under *buckets* at the third level with other queries beginning with `frozen hell`. The final mapping is to the counter representing the number of matches for this query. Each level contains a mapping to the next token. The algorithm for constructing these structures is detailed in Figure 1.

During query construction, wildcard placeholders are treated identically to other tokens. They are represented in our implementation by the token <*>, which is not present in the unigram counts for the Web 1T corpus. The counter is used in hashtables at the final token to store the frequencies found when patterns do match.

After all queries have been processed into these structures, each pattern of the appropriate length in the corpus is checked to determine if any matches occur. The recursive procedure used to check for matches is given in Figure 2. Initial arguments are $h = h_0$ and a depth = 1. The set of tokens and the frequency of those tokens is consistent across all invocations of the Search procedure for a given Web 1T entry, and is thus omitted from the arguments shown for brevity. The two recursive invocations are

```
Let h_1 be initial hashtable;
foreach query do
    Let h = h_1;
    foreach i ∈ 1 ... n do
        token_i = query[i];
        if h[token_i] does not exist then
            if i ≠ n then
                set h[token_i] = new hashtable;
            else
                set h[token_i] = counter
                initialised to 0;
            end
        end
        h = h[token_i]
    end
end
```

Figure 1: Query Structure Construction

```
Procedure Search(h, depth) begin
    token_i = query[depth];
    if h[tokens_i] exists then
        if depth ≠ n then
            Search(h[tokens_i], depth + 1)
        else
            counter = h[tokens_i];
            counter ← counter + frequency;
        end
    end
    if h[WILDCARD] exists then
        if depth ≠ n then
            Search(h[WILDCARD], depth + 1)
        else
            counter = h[WILDCARD];
            counter ← counter + frequency;
        end
    end
end
```

Figure 2: **Procedure** Search(`h, depth`)

to match the current token with both any query specifying that particular token at that location, and any query specifying a wildcard in that position.

4.2.2 Algorithmic Complexity

The time taken to add queries is maximised when each token in every query is novel, and thus is not present in the existing structure. In this case the time for each query is proportional to the number of tokens. This means that adding q queries of length n is $O(nq)$.

Considering our search strategy for a single corpus entry, the largest number of hashtable lookups are required when queries exist for all combinations of both wildcards and exactly matching tokens. This worst-case instance requires two hashtable lookups for the first token, four lookups for the second and so on: a total of $2^{n+1} - 2$ lookups for patterns of length n. In the asymptotic limit, the algorithmic complexity for each corpus entry is thus $O(2^n)$. If we let m represent the number of entries of length n in the corpus, the independent processing of each entry yields an overall complexity of $O(2^n m)$.

In practice however, there are several factors that combine to keep search performance fast. Firstly, n has a maximum value of 5, leading to only 62 hashtable lookups in the worst case. The actual performance of the system depends on the characteristics of the data and the queries. For the exponential behaviour to apply to large numbers of entries in the database, each of these distinct entries would have to be matched by a corresponding exact query, accompanied by a large number of wildcard permutations. As there are far fewer queries than database entries, the worst-case performance can not apply to more than a small fraction of the database entries. In the best case, very few entries match queries at all — comparison for most patterns can then be terminated at the first token, with a complexity of $O(m)$.

Even in the worst case for hashtable lookups, the algorithmic complexity does not contain a term for the number of queries. While searches over more queries and an abundance of wildcards will impact the running time to some extent, even with many millions of queries the overall performance has a practical upper bound. As Keller and Lapata (2003) observe, a single linguistic query may expand to many corpus queries to account for inflectional and other variation, and thus continued efficiency as the number of queries is increased is thus most desirable. This is not the case for binary search or any of the other methods discussed previously.

Memory requirements are related only to the number of, and similarities among the queries specified. The memory required for increasing the num-

ber of queries will be no worse than linearly proportional to the size of the query set. This is apparent when the case where each term in each query is unique is considered: each term will be stored exactly once in the data structure. Any overlaps in the leftmost tokens of queries will reduce the memory burden with respect to this worst-case requirement.

4.2.3 Implementation

This software runs once through the n-grams for each pattern length, and extracts counts for all specified queries. The input format for the queries matches that of the corpus exactly — one query per line — aside from the lack of the final tab character and numeric count. The queries are processed into the nested hashtable structures in RAM before any access is made to the corpus data. After construction of these structures, all Web 1T corpus files containing entries for patterns of the appropriate pattern length (each containing 10^7 counts) are processed in turn. Each is decompressed into RAM and once there each entry is checked against the query structures as described previously.

Following processing of all relevant corpus files, the results are written to text files. Case-sensitivity is configurable at run-time via command-line options. Counts are stored in RAM rather than written immediately, as for case-insensitivity and queries involving wildcard counts, the frequency reported may be the sum of many individual counts in the corpus.

It is also possible to find not just the total number of n-grams for a given wildcard, but also enumerate each match; this may also be configured via the command line at run time, but for 2 and 3-gram queries it is not recommended, as it generally results in a prohibitively large number of hits.

5 Performance

Tools which employ both strategies described earlier have been implemented, and have been made available as free software. For performance reasons, C was used as the implementation language.

The speed of the approach using the pre-processed (compressed) corpus depends straightforwardly on the number of queries, and is generally constrained by the time to read data from disk.

The pre-processed query approach was implemented with adequate performance on commodity hardware as a design goal. As a result, the runtime speed is quick, and scales well with large numbers of queries. Experiments have been performed that search for over 1,000,000 5-grams with acceptable performance. For typical queries, the sparseness of hits among 5-grams, even in a resource spanning as much variety as the Web 1T corpus ensures that the worst-case performance is infrequent.

Processing all 5-grams for 10^6 queries on a computer with a 2.66GHz Xeon CPU took around 1 hour and 1.5GB of RAM. Simple parallelisation is easily achieved by processing patterns of different lengths on separate computers or CPU cores (assuming of course that sufficient RAM and CPUs are available). This approach can reduce the time taken for the entire run to the time for that of the longest-running pattern length. The memory footprint of the program is at worst linearly related to the number of queries, and as these tests show, is manageable for a large number of queries.

6 Conclusion

It is possible to query a resource the size of the Web 1T corpus using commodity hardware and effective hashing-based strategies. Software has been created that makes the use of this resource practical, and has been successfully used to harness the information available for NLP tasks including Word Sense Disambiguation, Lexical Substitution and Noun-Phrase Bracketing. The methods employed need only to make a single pass of the corpus data. In one approach, the corpus is transformed to a more amenable structure; in the other, the queries are indexed and searched, rather than the corpus.

The tools we have implemented, using the the techniques described in this paper, facilitate the use of the massive scale of data now available for more disparate and data-hungry NLP tasks.

7 Acknowledgements

Many thanks to James Curran and Jon Patrick for their assistance and advice. We also thank Charles B. Falconer, author of the free hashlib library for C, which we use for the pre-processed queries. Part of this work has been supported by the Australian Research Council under Discovery Project DP0558852.

References

Michelle Banko and Eric Brill. 2001. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Meeting of the Association for Computational Linguistics (ACL-01)*, pages 26–33. Toulouse, France.

Thorsten Brants and Alex Franz. 2006. Web 1T 5-gram corpus version 1. Technical report, Google Research.

Gregory Grefenstette. 1999. The WWW as a resource for example-based MT tasks. In *ASLIB Translating and the Computer Conference*. London.

Tobias Hawker. 2007. USYD: WSD and lexical substitution using the Web 1T corpus. In *Proceedings of the Fourth International Workshop on the Evaluation of Systems for the Semantic Analysis of Text (SemEval-07)*, pages 446–453. Prague, Czech Republic.

Frank Keller and Mirella Lapata. 2003. Using the web to obtain frequencies for unseen bigrams. *Computational Linguistics*, 29(3):459–484.

Adam Kilgarriff. 2007. Googleology is bad science. *Computational Linguistics*, 33(1):147–151.

Vinci Liu and James R. Curran. 2006. Web text corpus for natural language processing. In *Proceedings of the 11th Meeting of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 233–240.

Preslav Nakov and Marti Hearst. 2005. Search engine statistics beyond the n-gram: Application to noun compound bracketing. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 17–24. Ann Arbor, Michigan.

Peter D. Turney. 2001. Mining the web for synonyms: PMI-IR versus LSA on TOEFL. In *Proceedings of the Twelfth European Conference on Machine Learning (ECML-2001), Freiburg, Germany*, pages 491–502.

David Vadas and James R. Curran. 2007. Adding noun phrase structure to the penn treebank. In *Proceedings of the 45th annual meeting of the Association for Computational Linguistics (ACL)*, pages 240–247. Prague, Czech Republic.

Deniz Yuret. 2007. KU: Word sense disambiguation by substitution. In *Proceedings of the Fourth International Workshop on the Evaluation of Systems for the Semantic Analysis of Text (SemEval-07)*, pages 207–214. Prague, Czech Republic.