

qSpell: Spelling Correction of Web Search Queries using Ranking Models and Iterative Correction

Yasser Ganjisaffar, Andrea Zilio, Sara Javanmardi, Inci Cetindil,
Manik Sikka, Sandeep Katumalla, Narges Khatib, Chen Li, Cristina Lopes
School of Information & Computer Sciences
University of California, Irvine
Irvine, CA, USA
yganjisa@ics.uci.edu

ABSTRACT

In this work we address the challenging problem of Web search queries in order to build a speller that proposes the most plausible spelling alternatives for each query. First we generate a large set of candidates using diverse approaches including enumerating all possible candidates in edit distance of one, fuzzy search on known data sets, and word breaking. Then, we extract about 150 features for each query-candidate pair and train a ranking model to order the candidates such that the best candidates are ranked on the top of the list. We show that re-ranking top results, iterative correction, and post-processing of the results can significantly increase the precision of the spell checker. The final spell checker, named *qSpell*¹, achieves a precision of 0.9482 on a test data set randomly collected from real search queries. qSpell won the 3rd prize in the recent Microsoft’s Speller Challenge.

1. INTRODUCTION

The spelling correction problem is typically formulated using the noisy channel model [9]. Given an input query q , we want to find the best correction c^* among all the candidate corrections:

$$c^* = \underset{c}{\operatorname{argmax}} P(q|c)P(c),$$

where $P(q|c)$ is the error model probability which shows the transformation probability from c to q , and $P(c)$ is the language model probability which shows the likelihood that c is a correctly spelled candidate.

The noisy channel model only considers error model and language model probabilities. However, a Web-scale spelling correction system needs to consider signals from diverse sources to come up with the best suggestions. We treat the spelling correction problem as a ranking problem where we first generate a set of candidates from the query and then rank them

¹<http://flamingo.ics.uci.edu/spellchecker/>

by learning a ranking model on the features extracted from them. The ranking process is expected to bring the best candidates to the top of the ranked list.

We extract several features from query-candidate pairs and use a machine learning based ranking algorithm for training a ranking model on these features. The top- k results of this ranking model are then re-ranked using another ranker. This step further increases the quality of the ranked list.

Given that it is hard to generate the best candidates for queries that require several corrections, we use an iterative approach which applies one fix at each iteration. In the final step, we use a rule-based system to process the final ranked list and decide how many candidates from the top of the list should be suggested as possible good candidates for this query. In the next sections, we describe more details of our spell correction system.

2. ERROR MODEL

For computing error model probabilities we used the approach proposed by Brill in [9]. To train this model, we needed a training set of $\langle s_i, w_i \rangle$ string pairs, where s_i represents a spelling error and w_i is the corresponding corrected word. We extracted these pairs from the query reformulation sessions that we extracted from the AOL query log [1]. We processed this query log and extracted pairs of queries (q_1, q_2) which belonged to the same user and were issued within a time window of 5 minutes. The queries are further limited to cases where the edit distance between q_1 and q_2 is less than 4 and the user has not clicked on any result for query q_1 and has clicked on at least one result for query q_2 . We processed about 20M queries in this query log and extracted about 634K training pairs. Following the approach proposed in [9] for each training pair (q_1, q_2) , we counted the frequencies of edit operations $\alpha \rightarrow \beta$. These frequencies are then used for computing $P(\alpha \rightarrow \beta)$, which shows the probability that when users intended to type the string α they typed β instead.

To extract edit operations from the training set, we first aligned the characters of q_1 and q_2 based on their Damerau-Levenshtein distance [13]. Then for each mismatch in the alignment, we found all possible edit operations within a sliding window of a specific size. As an example, we extract the following edit operations from the training pair $\langle \text{satellite}, \text{satillite} \rangle$:

- Window size 1: $e \rightarrow i$;

Table 1: Language Model Datasets

Dataset	No. of tokens	No. of ngrams
Google	1T	615M (up to trigrams)
Yahoo	3B	166M (up to trigrams)
Wikipedia Body	1.4B	134M (up to trigrams)

- Window size 2: $te \rightarrow ti, \quad el \rightarrow il$;
- Window size 3: $tel \rightarrow til, \quad ate \rightarrow ati, \quad ell \rightarrow ill$.

We counted the frequency of each of these edit operations in the training set and used these frequencies for computing probability of each edit operation and picking the most probable edits when computing the error model probabilities for different candidates.

3. LANGUAGE MODELS

We implemented three different smoothed language models in our system: Stupid Backoff [8], Absolute Discounting [10], and Kneser Ney [10] smoothing. Moreover each of these language models is computed on three different datasets: Google ngrams [7], Yahoo ngrams [3], and Wikipedia body ngrams (Table 1). Each of these data sets has different properties and we were expecting this diversity to improve the accuracy of our speller. The Google ngrams dataset is collected from public Web pages and therefore also includes many misspellings as well. On the other hand, the Yahoo ngrams dataset is collected from news Websites and therefore is cleaner. However, in terms of number of tokens it is much smaller than Google ngrams dataset. Both of these datasets are based on crawls of the Web in 2006 and therefore do not cover new topics. For this reason, we also created an ngram dataset from Wikipedia articles by processing the dump of Wikipedia articles content released in Jan 2011 [2]. Table 1 shows the properties of these datasets. We used a MapReduce cluster to normalize the ngrams in these datasets. For example “The”, “the” and “THE” are all normalized to “the” and their frequencies are aggregated.

Each of these language models has parameters that need to be tuned. We used parallel grid search on a MapReduce cluster to tune the parameters of these Language models for unigram, bigram and trigram levels. To train these language models, we used a training set of (q, c^*) pairs, where q is a query and c^* is the optimal candidate for this query. The goal of the training process was to find optimal values for language model parameters such that the conditional probability $P(c^*|q)$ is maximized:

$$P(c^*|q) = \frac{P(q|c^*)P(c^*)}{\sum_i P(q|c_i)P(c_i)}$$

As for the training set, we extracted 634K pairs of (q, c^*) from the AOL query log as explained in Section 2.

In addition to the above-mentioned word based language models, we also used a character based language model. We used a Hadoop job to extract the frequencies of 1–5 character grams from the Google data set and used it with Stupid Backoff smoothing as an additional language model. Note that this language model is not sparse and therefore we rarely need to backoff to lower-order ngrams. Therefore the choice of the smoothing method does not have any significant effect on the features that are extracted from this language model.

4. NGRAMS SERVICE

Given that we needed to query the ngram datasets very frequently it was important to have a service which can return the frequency of each given ngram in a short amount of time. For this purpose, we used the “compress, hash and displace” algorithm as described in [6] to create a minimal perfect hash (MPH) function for each dataset. This hash function is constructed on a set of N grams and assigns a unique number between 1 and N to each of them. This unique number can then be used to read the frequency of the ngram from an array.

While the hash function is guaranteed to return a unique number for each of the ngrams in the corpus, it still returns some unique number between 1 and N for any other string that has not been in the corpus. To reduce the occurrence of these false positives, we followed the approach suggested in [12]. In this approach, in addition to storing the frequencies of ngrams, we stored a fingerprint for each ngram as well. These fingerprints help in detecting a significant portion of the false positives.

Figure 1 demonstrates this process. The hash function returns a unique number for each ngram. This unique number is then used as an index to read the fingerprint and frequency of the ngram from an array. We first check the fingerprint of ngram and verify that it matches the expected fingerprint. If fingerprints do not match, we return zero frequency. Otherwise the frequency value which is stored in the array is returned.

We store a 20-bit fingerprint for each ngram. The largest frequency in Google data sets is 23,142,960,758 which can be stored in 35 bits. However, there are only 528,917 unique frequency numbers in this data set. Therefore, to save memory space, we sorted the unique frequency numbers and stored them in an array with 528,917 elements. Then for each ngram instead of storing the raw frequency of the ngram, we stored the index into this unique frequencies array (Figure 1).

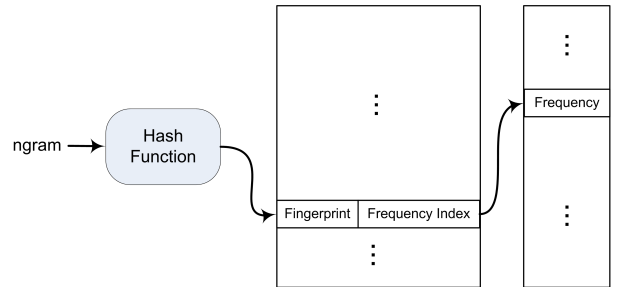


Figure 1: A memory-efficient data structure for fast look up of ngram frequencies

Using this approach we would be saving 15 bits per ngram. We were able to fit the Normalized Google ngrams dataset (unigrams to trigrams) in about 3.2GB of memory and the look up time was about 3 micro seconds. We measured the false positives rate by submitting the four-grams and five-grams in this data set and the rate was 9.6×10^{-7} which was small enough for our purpose.

5. CANDIDATE GENERATION

We implemented three different candidate generators in

Table 2: Word segmentation with different data sets and language model smoothing techniques

Dataset	Language Model Smoothing	Recall
Google	Stupid Backoff	96.79%
Google	Absolute Discounting	94.54%
Google	Kneser Ney	93.48%
Yahoo	Stupid Backoff	92.36%
Yahoo	Absolute Discounting	90.93%
Yahoo	Kneser Ney	92.01%
Wikipedia	Stupid Backoff	90.91%
Wikipedia	Absolute Discounting	89.17%
Wikipedia	Kneser Ney	90.18%

our system that generate about 3,000 candidates for each query. First, a character-based candidate generator generates all possible candidates within an edit distance of 1. It considers replacing each character with all possible characters in the alphabet, transposing each pair of adjacent characters, deleting each character and etc.

A quick analysis of the AOL query logs showed that 16% of the query corrections that we had extracted from this query log differ from the original query only in adding/removing spaces. The followings are some examples:

- `ebayauction` → `ebay auction`
- `broccoliandcheesebake` → `broccoli and cheese bake`
- `i cons` → `icons`

In order to handle this class of queries, we implemented the word segmentation algorithm described in [15] with some minor changes. For each possible segmentation, we query a language model to compute the probability of different segmentations of the query. The most probable segmentations are then added to the candidate list.

In order to find out which data set and language model smoothing is more appropriate for the word segmentation task, we filtered the (q, c^*) pairs that were extracted from AOL query logs and extracted 40,000 pairs where the difference of query q and the expected candidate c^* is only in space. Then we used this data set to see which choice of the ngrams data sets and language model smoothing techniques maximized the probability of generation of c^* after applying the word segmentation algorithm on the query. Table 2 shows the results. As this table shows, the Stupid Backoff language model on Google ngrams data set outperformed the others and therefore we used this combination in our word segmentation module.

None of the above two candidate generators can generate candidates where corrections are at edit distances of more than 1 and not only in adding or removing spaces. For example, for the query “`draigs list irvine`” we want to have “`craigslist irvine`” as a candidate. We used the Flamingo package² to perform fuzzy search and quickly find known unigrams with a small edit distance to unigrams and bigrams of the query. We extracted 790K most popular unigrams from the Google ngrams dataset and 947K most popular unigrams from the Wikipedia dataset. This step resulted in a set of 1.3M unigrams which were indexed by the Flamingo package.

²<http://flamingo.ics.uci.edu/>

6. LEARNING TO RANK CANDIDATES

In this section, we describe the details of the machine learning approach that we took for training a ranking model that orders candidates. We first needed to have a training data set. During the speller challenge competition, participants were provided with a data set of publicly available TREC queries from 2008 Million Query Track [4]. The data set also includes human judgments for correct spellings of each query as suggested by several human assessors. However, queries in this data set are sampled from queries which have at least one click in the .gov domain [5]. Therefore this data set was highly biased towards a special class of queries and was not a good representative for the general search queries.

6.1 Training Data Set

To have an unbiased data set for training and testing the ranking models, we randomly sampled 11,134 queries from the publicly available AOL and 2009 Million Query Track query sets and asked 8 human assessors to provide spelling corrections for these queries. Each query was judged by at least one human assessor. Queries for which the human assessors had provided at least one suggestion which was different from the original query were reviewed by at least another human assessor. In order to assist the human assessors in providing the most plausible suggestions for each query, we had designed an interface that was showing Google and Bing search results for each query.

A total of 12,042 spelling suggestions were proposed for the queries. From these suggestions 2,001 of them are different from the original query; while in 905 of the cases the difference is only in adding or removing spaces. Out of the remaining 1,096 cases, 73% of the suggestions are at edit distance of 1 from the original query and 23% of the suggestions are at edit distance of 2 from the original query.

We used 6,000 of the queries in our data set for 5-fold cross validation and determining which features are helpful to be included in the feature set. The remaining 5,134 queries are kept untouched for evaluation purposes. We refer to this dataset as JDB2011 in the rest of this paper and it is available publicly³. A spell checker which always returns the original query without any change will get a baseline precision of **0.8971** on the test queries.

In order to train a ranking model on this data set, we need to assign a label to each query-candidate pair to show the preference for different candidates. Then the ranking model would be trained on these samples to learn to rank better candidates on top of the others. As mentioned in section 5, we generated about 3,000 candidates for each query. We want the candidates that match suggestions provided by human assessors to be ranked on top of the list. Then we prefer candidates which are within an edit distance of 1 from any of these suggestions and etc. Therefore, we labeled the candidates according to this preference logic and used these labels for training the ranking model.

6.2 Ranking Features

We extracted 89 *raw features* for each query-candidate pair. These features include: error model features, candidate language model features, query language model features, surface features capturing differences of the query and

³<http://flamingo.ics.uci.edu/spellchecker/>

Table 3: Entity Datasets

Source	No. of Entities
Wikipedia titles	3.1 M
dmoz domains	2.1 M
dmoz titles	6.5 M
IMDB	3.3 M
Porno domains	448 K

the candidate, frequency of the query and the candidate and their substrings in different lists such as Wikipedia titles, dmoz and imdb. Table 3 shows the number of entities in different lists that we used for this purpose.

In addition to these raw features, we also extracted 49 *meta features* for each query-candidate pair. Meta features are computed by applying some basic operations on the original ranking features. For example, we expected a good candidate to be highly probable based on the error model probability, $P(q|c)$. Also we expected the ratio of the $P(c)/P(q)$ to be high. Therefore we used the following meta-feature which combines these three ranking features:

$$\log P(q|c) + \log P(c) - \log P(q).$$

6.3 Ranking Model

We used an Ensemble of gradient boosted trees [16] for learning a ranking model from the extracted features. Given that only about 15% of the queries in our data set needed spell correction, the training data set was imbalance and therefore the ranker was preferring the candidate which is equal to query (as this option is correct in 85% of the cases). For handling this problem we randomly selected several balanced datasets from the original training set and trained different ensembles on each of them. The final ranker is a bagged ensemble that uses the average of these ensembles to determine the ordering of candidates. As mentioned in [11], the bagged ensemble also results in a more accurate model that also has lower variance. The precision at position 1 for this ranker on JDB2011 test set is **0.9055**.

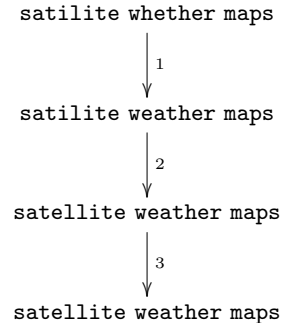
7. RE-RANKING TOP CANDIDATES

Re-ranking of top search results has shown to improve the quality of rankings in information retrieval problems [14]. Given that in spelling correction the main focus is on the top of the ranked list of candidates, we added a re-ranker module on top of the ranker module. This significantly improved the results of the original ranker. There are two main reasons for the success of the re-ranker: 1) It focuses only on top- k results and therefore it is solving a much easier problem compared to the original ranker which needs to consider about 3,000 candidates for each query. 2) In the re-ranking phase we added some more features which are extracted from top k results. These features include the score computed in the ranking phase, average, max, min, and standard deviation of the scores of the top- k candidates, etc. In contrast to the original ranking features, which are only extracted from query-candidate pairs, these new features also consider the top candidates of the query and therefore include valuable information that may help in the ranking process.

The precision at position 1 on JDB2011 test set after re-ranking increases to **0.9412** which is significantly better than the precision of the ranking module.

8. ITERATIVE CORRECTION

The spelling correction approach that was described in the previous sections does not handle more complex cases where multiple terms in the query need to be corrected, or a single term needs multiple corrections and the correct term is not among the originally generated candidates. For handling these cases, we use iterative correction to apply one correction at each iteration. For example, for the query “satilite whether maps” we have the following iterations:



As this example shows, we stop when there is no change the query in the last iteration. The precision at position 1 on JDB2011 test set after applying iterative correction increases to **0.9453**.

9. FINAL POST-PROCESSING

In the final phase and after iterative correction is stopped, we need to pick one or more candidates from the top of the ranked list of candidates. For this purpose, we used post-processing rules that process the final ranked list of candidates and decide which ones should be picked as suggested corrections. For example, most of the times we want to suggest the top ranked candidates as well as other candidates that are on top of the list and their score is within a threshold from the score of the top-ranked candidate and differ from this candidate only in adding or removing space or in singular/plural cases.

In addition, we used some other rules that target different classes of queries. For example, given that a large portion of search queries are navigational queries, we use different lists compiled from dmoz and porno blacklists (Table 3) to detect domain names and treat them specially. In case of domain names, we show the original query and the most probable segmentation of the query. For example for query, “annualcreditreport” we suggest both “annualcreditreport” (which is a domain name) and “annual credit report” which is its most probable segmentation.

After applying the post-processing rules, the precision at position 1 on JDB2011 test set increases to **0.9482**.

Acknowledgments

Authors would like to thank Amazon.com for a research grant that allowed us to use their MapReduce cluster and Alexander Behm for his advise on using the Flamingo package. This work has been also partially supported by NIH grant 1R21LM010143-01A1 and NSF grants OCI-074806 and IIS-1030002.

10. REFERENCES

- [1] AOL query log.
<http://www.gregsadetsky.com/aol-data/>.
- [2] English wikipedia dumps.
<http://dumps.wikimedia.org/enwiki/latest/>.
- [3] Yahoo! Webscope dataset, N-Grams, version 2.0.
http://research.yahoo.com/Academic_Relations.
- [4] Microsoft speller challenge trec data, 2011.
<http://tinyurl.com/6xry3or>.
- [5] J. Allan, B. Carterette, J. A. Aslam, V. Pavlu, and E. Kanoulas. Million query track 2008 overview. In E. M. Voorhees and L. P. Buckland, editors, *The Sixteenth Text REtrieval Conference Proceedings (TREC 2008)*. National Institute of Standards and Technology, December 2009.
- [6] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. In A. Fiat and P. Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer Berlin / Heidelberg, 2009.
- [7] T. Brants and A. Franz. Web 1T 5-gram Version 1, 2006.
- [8] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *In EMNLP*, pages 858–867, 2007.
- [9] E. Brill and R. C. Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL '00, pages 286–293, 2000.
- [10] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, 1996.
- [11] Y. Ganjisaffar, R. Caruana, and C. Lopes. Bagging gradient-boosted trees for high precision, low variance ranking models. In *SIGIR 2011: Proceedings of the 34th Annual International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 2011.
- [12] D. Guthrie and M. Hepple. Storing the web in memory: space efficient language models with constant time retrieval. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 262–272, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.
- [13] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.
- [14] I. Matveeva, A. Laucius, C. Burges, L. Wong, and T. Burkard. High accuracy retrieval with multiple nested ranker. In *SIGIR*, pages 437–444, 2006.
- [15] P. Norvig. *Beautiful Data*, chapter Natural language corpus data, pages 219–242. Calif.: O'Reilly, 2009.
- [16] Q. Wu, C. Burges, K. Svore, and J. Gao. Ranking, boosting and model adaptation. Technical report, Microsoft Technical Report MSR-TR-2008-109, 2008.