CHAPTER 3

TOWARD IMPROVED HASHTAG SEGMENTATION

I have argued that although a non-segmented hashtag can be a feature, there are cases in which it may benefit a classifier to segment a hashtag into individual features. However, if there is an unrecognized term in a hashtag that would cause a segmenter with a static source corpus to improperly segment it, we would probably be better off not segmenting at all. How do we increase our confidence that a hashtag is properly segmented?

One method is updating the source corpus of the segmenter, and therefore our language model, so that it reflects the most current English language usage possible: the slang, the surprising new topics, the heretofore obscure but newly relevant names of people and places, and so on. Then if a candidate term can be shown to be sufficiently statistically significant, it can be accepted as a word. But how, exactly, can the model be reliably updated? The original Brants & Franz corpus released in 2006, known officially as the "Web 1T 5-gram Version 1" corpus but referred to informally as the "trillion-word" corpus, contains 1,024,908,267,229 tokens; its frequency distribution of unigrams holds 13,588,391 types, or entries. It totals 24 GB of compressed data. Although there does not appear to be much information regarding the trillion-word corpus' parameters of data collection publicly available, in 2009 Brants & Franz released the "Web 1T 5-gram, 10 European Languages Version 1" corpus. That corpus, which is approximately 28 GB compressed, is reported by them to have been gathered between October and December of 2008[1]. It hardly needs be said that this scale of data collection is not easily replicable with current consumer-grade technology, and certainly not within the timeframe required by the task before us. If a motivation for segmenting hashtags is to classify sentiment regarding topical events, a data collection timeframe of months will not suffice.

[1] http://www.ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2009T25 Retrieved December 19th, 2012.

In the pages that follow, I lay out a segmentation procedure that attempts to gracefully handle the time and topicality constraints necessitated by hashtag input while maintaining the robustness of Norvig's existing algorithm. I begin with a description of the existing procedure.

3.1 NORVIG'S ALGORITHM

In Chapter 1 I touched upon the segmentation method outlined by Norvig in Segaran and Hammerbacher 2009. Following him, I summarized his algorithm as first training a probabilistic language model, then enumerating a candidate set of segmentations for a test string, and finally selecting the most probable candidate based on probabilities derived from training. We will now explore these steps in further detail, as this research is a further application of his algorithm.

In this research, I work with the unigrams corpus also used by Norvig. Rather than directly using the approximately 13-million-type unigrams corpus included in the trillion-word corpus, Norvig created a cleaner version by removing all tokens that did not consist solely of alphabetic characters, combining entries in order to create case-insensitive types, and reducing the number of entries to the top 333,333 most common words. He found that these entries sufficed to retain 98% coverage while reducing the corpus size to 5 MB.

When training a language model for segmenting, we must consider how to map real-world factors to something quantifiable. A model, by definition, abstracts away unimportant aspects of the system that it simulates. In the case of our language model, we gloss over all the context inherent in a productive language system with a frequency distribution; i.e., the language's morphology, syntax, orthography, semantics and other linguistic factors are reduced to an accounting of its terms. Following Norvig's example, given the string *choosespain* (derived from a URL), we as humans apply our world knowledge to hopefully parse it as *choose spain* rather than *chooses pain*. This knowledge is deep, broad, and often unspoken, but we rely on it to choose the former as being a more probable interpretation. A language model based on our source corpus has no proximate access to that knowledge, but it does ultimately have access to the quantifiable consequences of that knowledge, namely that the term *choose spain* is more probable than *chooses pain*.

The model's terms need not include only bigrams; the trillion-word corpus holds unigrams through five-grams. Using this *N*-GRAM data, we can treat our language model as a MARKOV model, and estimate the probability of a word sequence *W*, or any word *w* within it, given a history *h*. Generically, assuming a sequence of random variables $X_1 \dots X_n$, the probability of such a sequence can be determined by the chain rule:

$$P(X_1 \dots X_n) = P(X_1)P(X_2|X_1)P(X_3|X_1^2) \dots P(X_n|X_1^{n-1})$$
$$= \prod_{k=1}^{n} P(X_k|X_1^{k-1})$$

Given this, for the word sequence *W* of length *N* being represented as w_1^n, we see:

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2) \dots P(w_n|w_1^{n-1})$$
$$= \prod_{k=1}^{n} P(w_k|w_1^{k-1})$$

However, if *N* is sufficiently large, the productivity of language dictates that for some *n*, $P(w_1^{n-1})$ is zero, which in turn causes our joint probability estimate to become zero as well. Hence we apply the Markov assumption, which states that the probability of a word *w* can be approximated by setting *N* to be relatively small:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$$

For example, given the sequence *Top Congressional leaders met with President Obama*, we can calculate the probability of *Obama* as $P(Obama|President)$ rather than $P(Obama|Top\ Congressional\ leaders\ met\ with\ President)$ by using a bigram model, which estimates the probability of a word given all previous words $P(w_n|w_1^{n-1})$ by computing the conditional probability of the single most recent word $P(w_n|w_{n-1})$. It follows that the probability of the whole sequence according to a bigram model is:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

or, allowing for special sentence-delineating characters that ensure that bigram contexts across all words create a probability distribution (see Chen & Goodman 1998):

$$P(< s > Top\ Congressional\ leaders\ met\ with\ President\ Obama\ </s >)$$
$$= P(Top| < s >)P(Congressional|Top)P(leaders|Congressional)P(met|leaders)$$
$$P(with|met)P(President|with)P(Obama|President)P(</s > |Obama)$$

Let us consider the application of a bigram model to the task of hashtag segmentation. On the afternoon of December 29th, 2012, I examined my Twitter-curated personalized trending topics list and selected *#MentionPerfection* as a potential exemplar of how a hashtag might be handled by a bigram model. Passing over the issue of how to insert <*s*> into an existing frequency distribution, we shall attempt to compute the bigram probability of *perfection* given an immediately preceding *mention*: $P(perfection|mention)$. This is calculated by normalizing the count of the bigram *mention perfection* with the unigram count of *mention*:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

(This normalization is permissible due to the fact that the unigram count of *mention* is equal to the sum of all bigrams beginning with *mention*.) Manually calculating this MAXIMUM LIKELIHOOD ESTIMATION then becomes a simple matter of grepping the bigram corpus for the count of *mention perfection* and the unigram corpus for the count of *mention* and dividing the former by the latter:

```
brandon@Mímisbrunnr:~/code/segmenter/corpora$ grep -nw "mention
perfection" bigrams.txt
```

However, we immediately run into a problem: *mention perfection* does not occur in the bigram corpus. According to the bigram model, *#MentionPerfection* cannot exist!

To deal with this, we have three options. We could implement some sort of SMOOTHING algorithm with our bigram counts by "stealing" some probability mass from existing bigrams to assign to bigrams that don't occur in training. Alternatively and/or additionally, we could tweak our procedure to BACKOFF to constituent unigram counts when bigram counts are not available and/or have a sufficiently low probability. However, as Norvig notes, in addition to the irritation of implementing these changes and adjusting the bigram corpus to allow <*s*> and </*s*>, there is also the consideration that *N* > *1*-gram data takes up more room in RAM than unigrams do. Additionally, as mentioned previously, a large number of hashtags are only a few words in length, if not only one word long. If we were to fully implement a backoff algorithm for hashtag segmentation that began with 5-gram probabilities, as is allowed by the trillion-word corpus, it is unclear whether the additional computing costs would be worthwhile.

In his work, Norvig eventually settles on a bigram model, but he does not consider the effects of a minimal sequence length. For that reason, I utilize his earlier method of simply relying on a unigram model, which neatly avoids some of the issues discussed above; the question of how to handle unseen words is given more attention later on in this work. In the unigram model, the probability of a word sequence *W* of length *N* is simply the product of the probability of each individual word:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k)$$

According to this model, the probability of *#MentionPerfection* would simply be the joint probabilities of *mention* and *perfection*, both of which do occur in the unigram corpus:

```
brandon@Mímisbrunnr:~/code/segmenter/corpora$ grep -nw "mention"
unigrams.txt
3935:mention      18291476
```

```
brandon@Mímisbrunnr:~/code/segmenter/corpora$ grep -nw "perfection"
unigrams.txt
11450:perfection  4079350
```

What still must be considered, even in this simplified model, is a method of handling words not seen in the training corpus. (After all, there are no 0-grams to back off to.) I again use Norvig's method, which takes into account the number *N* of tokens in the unigrams corpus and the relative probabilities of unseen words of various lengths in order to arbitrarily set the likelihood of an unknown word <*UNK*>:

$$P(< UNK >) = \frac{10}{N * 10^{length(<UNK>)}}$$

However, the use of this method in my research is considered in the context of having earlier adjusted the training corpus in a manner that attempts to reduce the likelihood of finding unknown words; again, this will be revisited at a later point in this work.

Having trained on the unigrams corpus in order to generate our probabilistic language model, the next task is to segment a test string and choose a segmentation derived from the probabilities found in the model. Consider the possible segmentations of the word *cat*: they are *c*|*at*, *ca*|*t*, *cat*, and *c*|*a*|*t*. In this three-character string, there are two locations that may either or both hold a word boundary and it is also possible for the string to not be segmented; we thus see four segmentation candidates. This indicates a combinatorial guideline: for an *n*-character string there are 2^{n-1} possible segmentations, since there is a binary word boundary modality for any of the $n-1$ positions between characters. Note also that in human parsing of these segmentations, the assumption of contextual independence afforded by a unigram model means that once *cat* is derived, we are free to move on to what we believe to be the next word. Yet consider the task of parsing *category*; having begun with an initial *cat*|*egory* parse, we move through the segmentations of *egory* before concluding that in fact we need to backtrack and readjust our initial *cat* segmentation.

Rather than use this bigram model in the algorithm, Norvig uses a `segment` function that recursively splits input text into a first word and a remainder, with the remainder then being treated as the input text. The computational inefficiencies of the recursion are handled

by using the dynamic programming technique of MEMOIZING, or caching, the substrings in order to avoid having to recalculate their probabilities. In this way, the algorithm multiplies the probabilities for all possible segmentations together and finds the candidate with the highest probability to be the most likely overall segmentation. Returning to the *cat*|*egory* example, `segment` would eventually compare the joint likelihood of $P(cat)$ and $P(egory)$ and find it to be smaller than $P(category)$; it would thereby ultimately choose *category* as the most likely "segmentation".

3.2 An Extension of the Existing Methodology

Thus far, I have outlined the problem space of a need to segment Twitter hashtags; in doing so, I have provided an overview of the workings and features of Twitter, sketched an existing word segmentation method, and argued for a hashtag-specific method of segmentation. It is here that I can plainly state the parameters of how this algorithm differs from the extant version. As previously mentioned, a possible method to handle false negatives – those terms in testing that should cross the threshold of lexicalization, but do not due to a lack of training data – is to orient our language model towards the domains that a given test document resides in. In addition to the work mentioned above, the task of researching methods of being able to quickly adjust the model to accommodate data not seen in training, the subsequent implementation of one such method, and an examination of its efficacy together constitute the contribution this work offers. In the sections that follow, I discuss the parameters of this work, along with some technical and operational challenges. I explore two possible methods of online corpus updating and describe the integration of one such method into the existing non-hashtag-centric methodology.

3.3 Operational Issues

Given the need to keep the training data for a given tweet relevant to that tweet, an obvious solution space involves some form of language model updating. However, as already noted, a wholesale training set update for every test document is infeasible. In tackling this problem, I begin by borrowing from topic-based language models the idea that since documents that share topics share similar term frequency distributions, different models should be trained for different topics (Gildea & Hofmann 1999). (Here I define TOPIC as a categorical distribution over some fixed vocabulary, after Blei et. al. 2003). Topic-mixture

models examine some history of a test document in order to determine how to weight a particular topic's training set in order to best generate a custom language model; however, I suggest that the brevity of text in any given test tweet implies that the number of topics in that tweet is low and that therefore mixing is not appropriate. This monographic characteristic of tweets leads to the intuition that terms found within a hashtag will also be seen in the text of documents related to that hashtag, given enough documents to sample from. If so, those documents could be a viable source of training data for segmenting the hashtag. The problem is twofold: how do we define this relationship, and how do we obtain such documents?

3.3.1 Web-based Results: A Dead End

One way is to answer both questions is to view them as a generalized QUERY EXPANSION problem: if a hashtag itself is treated as a query, then terms associated with that hashtag may refine and extend the query to return more relevant results than the hashtag alone. More specifically, the set of non-closed-class words contained in the text of a tweet are treated as the PSEUDO-RELEVANT results immediately retrieved by a "query" containing the hashtag; some combination of the members of these "results" are then sent as one or more queries to a WWW search engine to return training data.

The lure of this technique is the large amount of training data that is potentially available. Even better, through the use of a search engine API against a general query, it is often easy to return query-biased document summaries (Tombros & Sanderson 1998) that yields hyper-relevant information "for free", as it were. If this method is used, a frequency distribution over all terms found in the summaries (up to a given cutoff) generated from one or more queries containing one or more of the tweet "results" can be calculated. One then hopes that `segment` will correctly utilize this distribution.

However, several pragmatic factors conspire against the implementation of this technique. It assumes that an API is available to the extent needed. As of this writing, both Google[2] and Bing[3] have transitioned to paying services, which may not suit researchers. It

[2] https://developers.google.com/custom-search/v1/overview Updated August 12th, 2012 and retrieved January 2nd, 2013.

also assumes that the search engine used updates its index quickly enough to include results relevant to the hashtag. Perhaps the most critical, however, is the reliability of the results.

Suppose some scheme had been instituted to query a search engine with a bigram. As Nakov and Hearst (2005) note:

> "Consider the bigrams w_1w_4, w_2w_4, and w_3w_4 and a page that contains each bigram exactly once. A search engine will contribute a page count of 1 for w_4 instead of a frequency of 3; thus the number of page hits for w_4 can be smaller than that for the sum of the bigrams that contain it."

In this case, such concerns immediately throw out any possibility of modifying the search method to utilize simple hit counts. Even more important is the realization that search engine results are a black box with a number of variables affecting the results. Funahashi and Yamana (2010) attempted to gauge the reliability of hit counts in the face of the known phenomenon of the counts' "dancing", or changing between queries of the same term(s). This change occurs as the result of connecting to different servers between queries, any of which may hold a portion of an index that is not synchronized with other portions. Furthermore, the method and speed of index updating lead to variable periods of stability between the "dancing" phases. Funahashi and Yamana, examining the behavior of the Bing and Yahoo! search engines, concluded that the hit count for a given query could be deemed stable when it keeps almost the same number over the course of a week. The demands of gathering training data quickly for a currently viable topic indicate that this is a losing strategy. Generally speaking, due to the importance of having accurate representation of query results to generate a precise frequency distribution of terms, relying on web search to generate training data for this task does not seem currently viable.

3.3.2 Twitter-based Results

Another source of training data that seems more accessible is the collection of tweets containing a given hashtag. This relationship is clearly defined and it is considerably easier to obtain the text of these tweets via the Twitter API than it is to obtain relevant text from the general Web with a search engine API. Although the Twitter API suffers in comparison to

---

[3] http://datamarket.azure.com/dataset/bing/search Updated April 11th, 2012 and retrieved January 2nd, 2013.

the sheer quantity of the potential data available through a web API, it more than acquits itself with the richness and relevance of its resources. The Twitter API exposes tweets that can be filtered by hashtag, location, language, and other criteria; moreover, using it to retrieve training data focuses this research. Specifically, we can formulate null and alternative research hypotheses:

H_0: For a given non-segmented hashtag, there is no statistical difference between the performances of a segmenter using a static training corpus and a segmenter using a training corpus updated with terms found in tweets containing said hashtag.

H_a: For a given non-segmented hashtag, a segmenter using a training corpus updated with terms found in tweets containing said hashtag outperforms a segmenter using a static training corpus to a statistically significant degree.

3.4 TECHNICAL ISSUES

The Twitter API is actually composed of two discrete API groups: REST and streaming. The former follows established REST procedures with a series of HTTP requests, while the latter issues long-lived HTTP connections over which data is continually sent. Both are subject to rate limiting, but differ in the types of resources offered. The streaming APIs issue tweets and user data; the REST API offers a variety of resources and actions, such as tweet retrieval, profile updating, and spam reporting[4]. As of version 1.1 of the REST API, rate limits over a 15 minute interval are capped at 15 or 180 GET requests, depending on the requested resource.[5] The Public streaming API offers three endpoints by which to acquire tweets; only one of which, the "Firehose", returns all public statuses. Access to the Firehose is generally withheld from all but a few large third-party partners.[6] The other two endpoints, "Sample" and "Filter", are together sometimes referred to informally as the "Spritzer", due to the relatively small amount of data they return.[7] Much of the academic research that has been

[4] https://dev.twitter.com/docs/rate-limiting/1.1 Updated October 26th, 2012 and retrieved December 24th, 2012.

[5] https://dev.twitter.com/docs/rate-limiting/1.1/limits Retrieved December 24th, 2012.

[6] https://dev.twitter.com/docs/api/1.1/get/statuses/firehose Updated August 27th, 2012 and retrieved December 24th, 2012.

[7] Twitter reserves the right to adjust the flow of data through the Spritzer; as of this writing it is currently

performed on Twitter has utilized the "Spritzer" feed; see Bernstein et. al. 2010, Hong et. al. 2011, and Yang & Counts 2012, among others.

I accessed the Twitter API by several means. Rather than working directly with the API, I employed various Python libraries and other APIs that offered some amount of built-in error handling and other niceties. When acquiring REST resources, such as currently trending hashtags, I used the Python package *twitter*[8], which despite its name is not maintained nor officially sanctioned by Twitter. When retrieving streaming resources, I used *Requests*[9] to interact with Topsy[10], an analytics service that exposes a powerful API capable of quickly retrieving a large amount of Twitter data.

---

downsampled to about 1% of the 100% access represented by the Firehose. See https://dev.twitter.com/discussions/2907, retrieved December 24th, 2012.

[8] http://pypi.python.org/pypi/twitter/1.9.0 Retrieved December 24th, 2012.

[9] http://docs.python-requests.org/ Retrieved December 24th, 2012.

[10] http://topsy.com/ Retrieved December 24th, 2012.