CS838-1 Advanced NLP: Automatic Summarization

Andrew Goldberg (goldberg@cs.wisc.edu)

March 16, 2007

1 Introduction

Automatic summarization involves reducing a text document or a larger corpus of multiple documents into a short set of words or paragraph that conveys the main meaning of the text.

Extractive methods work by selecting a subset of existing words, phrases, or sentences in the original text to form the summary. In contrast, *abstractive* methods build an internal semantic representation and then use natural language generation techniques to create a summary that is closer to what a human might generate. Such a summary might contain words not explicitly present in the original. The state-of-the-art abstractive methods are still quite weak, so most research has focused on extractive methods, and this is what we will cover.

Two particular types of summarization often addressed in the literature are *keyphrase extraction*, where the goal is to select individual words or phrases to "tag" a document, and *document summarization*, where the goal is to select whole sentences to create a short paragraph summary.

2 Keyphrase Extraction

2.1 Task Description and Example

The task is the following. You are given a piece of text, such as a journal article, and you must produce a list of keywords or keyphrases that capture the primary topics discussed in the text. In the case of research articles, many authors provide manually assigned keywords, but most text lacks pre-existing keyphrases. For example, news articles rarely have keyphrases attached, but it would be useful to be able to automatically do so for a number of applications discussed below.

Consider the example text from a recent news article:

The Army Corps of Engineers, rushing to meet President Bush's promise to protect New Orleans by the start of the 2006 hurricane season, installed defective flood-control pumps last year despite warnings from its own expert that the equipment would fail during a storm, according to documents obtained by The Associated Press.

An extractive keyphrase extractor might select "Army Corps of Engineers," "President Bush," "New Orleans," and "defective flood-control pumps" as keyphrases. These are pulled directly from the text. In contrast, an abstractive keyphrase system would somehow internalize the content and generate keyphrases that might be more descriptive and more like what a human would produce, such as "political negligence" or "inadequate protection from floods." Note that these terms do not appear in the text and require an deep understanding, which makes it difficult for a computer to produce such keyphrases.

Keyphrases have many applications, such as to improve document browsing by providing a short summary. Also, keyphrases can improve information retrieval—if documents have keyphrases assigned, a user could search by keyphrase to produce more reliable hits than a full-text search. Also, automatic keyphrase extraction can be useful in generating index entries for a large text corpus.

2.2 Keyphrase Extraction as Supervised Learning

Beginning with the Turney paper, many researchers have approached keyphrase extraction as a supervised machine learning problem. Given a document, we construct an example for each unigram, bigram, and trigram found in the text (though other text units are also possible, as discussed below). We then compute various features describing each example (e.g., does the phrase begin with an upper-case letter?). We assume there are known keyphrases available for a set of training documents. Using the known keyphrases, we can assign positive or negative labels to the examples. Then we learn a classifier that can discriminate between positive and negative examples as a function of the features. Some classifiers make a binary classification for a test example, while others assign a probability of being a keyphrase. For instance, in the above text, we might learn a rule that says phrases with initial capital letters are likely to be keyphrases.

After training a learner, we can select keyphrases for test documents in the following manner. We apply the same example-generation strategy to the test documents, then run each example through the learner. We can determine the keyphrases by looking at binary classification decisions or probabilities returned from our learned model. If probabilities are given, a threshold is used to select the keyphrases.

Keyphrase extractors are generally evaluated using precision and recall. Precision measures how many of the proposed keyphrases are actually correct. Recall measures how many of the true keyphrases your system proposed. The two measures can be combined in an F-score, which is the harmonic mean of the two ($F = \frac{2PR}{P+R}$). Matches between the proposed keyphrases and the known keyphrases can be checked after stemming or applying some other text normalization.

2.2.1 Design Choices

Designing a supervised keyphrase extraction system involves deciding on several choices (some of these apply to unsupervised, too):

- *What are the examples?* The first choice is exactly how to generate examples. Turney and others have used all possible unigrams, bigrams, and trigrams without intervening punctuation

and after removing stopwords. Hulth showed that you can get some improvement by selecting examples to be sequences of tokens that match certain patterns of part-of-speech tags. Ideally, the mechanism for generating examples produces all the known labeled keyphrases as candidates, though this is often not the case. For example, if we use only unigrams, bigrams, and trigrams, then we will never be able to extract a known keyphrase containing four words. Thus, recall may suffer. However, generating too many examples can also lead to low precision.

- *What are the features?* We also need to create features that describe the examples and are informative enough to allow a learning algorithm to discriminate keyphrases from non-keyphrases. Typically features involve various term frequencies (how many times a phrase appears in the current text or in a larger corpus), the length of the example, relative position of the first occurrence, various boolean syntactic features (e.g., contains all caps), etc. The Turney paper used about 12 such features. Hulth uses a reduced set of features, which were found most successful in the KEA (Keyphrase Extraction Algorithm) work derived from Turney's seminal paper.

- *How many keyphrases to return?* In the end, the system will need to return a list of keyphrases for a test document, so we need to have a way to limit the number. Ensemble methods (i.e., using votes frpm several classifiers) have been used to produce numeric scores that can be thresholded to provide a user-provided number of keyphrases. This is the technique used by Turney with C4.5 decision trees. Hulth used a single binary classifier so the learning algorithm implicitly determines the appropriate number.

- *What learning algorithm?* Once examples and features are created, we need a way to learn to predict keyphrases. Virtually any supervised learning algorithm could be used, such as decision trees, Naive Bayes, and rule induction. In the case of Turney's GenEx algorithm, a genetic algorithm is used to learn parameters for a domain-specific keyphrase extraction algorithm. The extractor follows a series of heuristics to identify keyphrases. The genetic algorithm optimizes parameters for these heuristics with respect to performance on training documents with known keyphrases.

2.3 Unsupervised Keyphrase Extraction: TextRank

While supervised methods have some nice properties, like being able to produce interpretable rules for what features characterize a keyphrase, they also require a large amount of training data. Many documents with known keyphrases are needed. Furthermore, training on a specific domain tends to customize the extraction process to that domain, so the resulting classifier is not necessarily portable, as some of Turney's results demonstrate.

Unsupervised keyphrase extraction removes the need for training data. It approaches the problem from a different angle. Instead of trying to learn explicit features that characterize keyphrases, the TextRank algorithm exploits the structure of the text itself to determine keyphrases that appear "central" to the text in the same way that PageRank selects important Web pages. Recall this is based on the notion of "prestige" or "recommendation" from social networks. In this way, TextRank does not rely on any previous training data at all, but rather can be run on any arbitrary piece of

text, and it can produce output simply based on the text's intrinsic properties. Thus the algorithm is easily portable to new domains and languages.

TextRank is a general purpose graph-based ranking algorithm for NLP. Essentially, it runs PageRank on a graph specially designed for a particular NLP task. For keyphrase extraction, it builds a graph using some set of text units as vertices. Edges are based on some measure of semantic or lexical similarity between the text unit vertices. Unlike PageRank, the edges are typically undirected and can be weighted to reflect a degree of similarity. Once the graph is constructed, it is used to form a stochastic matrix, combined with a damping factor (as in the "random surfer model"), and the ranking over vertices is obtained by finding the eigenvector corresponding to eigenvalue 1 (i.e., the stationary distribution of the random walk on the graph).

2.3.1 Design Choices

Using TextRank involves a few decisions:

- *What should vertices be?* The vertices should correspond to what we want to rank. Potentially, we could do something similar to the supervised methods and create a vertex for each unigram, bigram, trigram, etc. However, to keep the graph small, the authors decide to rank individual unigrams in a first step, and then include a second step that merges highly ranked adjacent unigrams to form multi-word phrases. This has a nice side effect of allowing us to produce keyphrases of arbitrary length. For example, if we rank unigrams and find that "advanced," "natural," "language," and "processing" all get high ranks, then we would look at the original text and see that these words appear consecutively and create a final keyphrase using all four together. Note that the unigrams placed in the graph can be filtered by part of speech. The authors found that adjectives and nouns were the best to include. Thus, some linguistic knowledge comes into play in this step.

- *How should we create edges?* Edges are created based on word co-occurrence in this application of TextRank. Two vertices are connected by an edge if the unigrams appear within a window of size N in the original text. N is typically around 2–10. Thus, "natural" and "language" might be linked in a text about NLP. "Natural" and "processing" would also be linked because they would both appear in the same string of N words. These edges build on the notion of "text cohesion" and the idea that words that appear near each other are likely related in a meaningful way and "recommend" each other to the reader.

- *How are the final keyphrases formed?* Since this method simply ranks the individual vertices, we need a way to threshold or produce a limited number of keyphrases. The technique chosen is to set a count T to be a user-specified fraction of the total number of vertices in the graph. Then the top T vertices/unigrams are selected based on their stationary probabilities. A post-processing step is then applied to merge adjacent instances of these T unigrams. As a result, potentially more or less than T final keyphrases will be produced, but the number should be roughly proportional to the length of the original text.

2.3.2 Why It Works

It is not initially clear why applying PageRank to a co-occurrence graph would produce useful keyphrases. One way to think about it is the following. A word that appears multiple times throughout a text may have many different co-occurring neighbors. For example, in a text about machine learning, the unigram "learning" might co-occur with "machine," "supervised," "unsupervised," and "semi-supervised" in four different sentences. Thus, the "learning" vertex would be a central "hub" that connects to these other modifying words. Running PageRank/TextRank on the graph is likely to rank "learning" highly. Similarly, if the text contains the phrase "supervised classification," then there would be an edge between "supervised" and "classification." If "classification" appears several other places and thus has many neighbors, it's importance would contribute to the importance of "supervised." It it ends up with a high rank, it will be selected as one of the top T unigrams, along with "learning" and probably "classification." In the final post-processing step, we would then end up with keyphrases "supervised learning" and "supervised classification."

In short, the co-occurrence graph will contain densely connected regions for terms that appear often and in different contexts. A random walk on this graph will have a stationary distribution that assigns large probabilities to the terms in the centers of the clusters. This is similar to densely connected Web pages getting ranked highly by PageRank.

3 Document Summarization

Like keyphrase extraction, document summarization hopes to identify the essence of a text. The only real difference is that now we're dealing with larger text units—whole sentences instead of words and phrases. While some work has been done in abstractive summarization (creating an abstract synopsis like that of a human), the majority of summarization systems are extractive (selecting a subset of sentences to place in a summary).

Before getting into the details of some summarization methods, we'll mention how summarization systems are typically evaluated. The most common way is using the so-called *ROUGE (Recall-Oriented Understudy for Gisting Evaluation)* measure (http://haydn.isi.edu/ROUGE/). This is a recall-based measure that determines how well a system-generated summary covers the content present in one or more human-generated model summaries known as references. It is recall-based to encourage systems to include all the important topics in the text. Recall can be computed with respect to unigram, bigram, trigram, or 4-gram matching, though ROUGE-1 (unigram matching) has been shown to correllate best with human assessments of system-generated summaries (i.e., the summaries with highest ROUGE-1 values correlate with the summaries humans deemed the best). ROUGE-1 is computed as

$$\text{ROUGE-1}(\text{system}, \text{reference}) = \frac{\#\text{ unigrams in reference that appear in system}}{\#\text{ unigrams in reference summary}}. \quad (1)$$

If there are multiple references, the ROUGE-1 scores are averaged. Because ROUGE is based only on content overlap, it can determine if the same general concepts are discussed between an automatic summary and a reference summary, but it cannot determine if the result is coherent or the sentences flow together in a sensible manner. High-order n-gram ROUGE measures try to judge fluency to some degree.

Note that ROUGE is similar to the BLEU measure for machine translation, but BLEU is precision-based, because translation systems favor accuracy.

3.1 Overview of Supervised Learning Approaches

Supervised text summarization is very much like supervised keyphrase extraction, and we won't spend much time on it. Basically, if you have a collection of documents and human-generated summaries for them, you can learn features of sentences that make them good candidates for inclusion in the summary. Features might include the position in the document (i.e., the first few sentences are probably important), the number of words in the sentence, etc. The main difficulty in supervised extractive summarization is that the known summaries must be manually created by extracting sentences so the sentences in an original training document can be labeled as "in summary" or "not in summary." This is not typically how people create summaries, so simply using journal abstracts or existing summaries is usually not sufficient. The sentences in these summaries do not necessarily match up with sentences in the original text, so it would difficult to assign labels to examples for training. Note, however, that these natural summaries can still be used for evaluation purposes, since ROUGE-1 only cares about unigrams.

3.2 Unsupervised Approaches: TextRank and LexRank

The unsupervised approach to summarization is also quite similar in spirit to unsupervised keyphrase extraction and gets around the issue of costly training data. Some unsupervised summarization approaches are based on finding a "centroid" sentence, which is the mean word vector of all the sentences in the document. Then the sentences can be ranked with regard to their similarity to this centroid sentence.

A more principled way to estimate sentence importance is using random walks and eigenvector centrality. LexRank is an algorithm essentially identical to TextRank, and both use this approach for document summarization. The two methods were developed by different groups at the same time, and LexRank simply focused on summarization, but could just as easily be used for keyphrase extraction or any other NLP ranking task.

3.2.1 Design Choices

- *What are the vertices?* In both LexRank and TextRank, a graph is constructed by creating a vertex for each sentence in the document.
- *What are the edges?* The edges between sentences are based on some form of semantic similarity or content overlap. While LexRank uses cosine similarity of TF-IDF vectors, TextRank uses a very similar measure based on the number of words two sentences have in common (normalized by the sentences' lengths). The LexRank paper explored using unweighted edges after applying a threshold to the cosine values, but also experimented with using edges with weights equal to the similarity score. TextRank uses continuous similarity scores as weights.

- *How are summaries formed?* In both algorithms, the sentences are ranked by applying PageRank to the resulting graph. A summary is formed by combining the top ranking sentences, using a threshold or length cutoff to limit the size of the summary.

3.2.2 TextRank and LexRank Differences

It is worth noting that TextRank was applied to summarization exactly as described here, while LexRank was used as part of a larger summarization system (MEAD) that combines the LexRank score (stationary probability) with other features like sentence position and length using a linear combination with either user-specified or automatically tuned weights. In this case, some training documents might be needed, though the TextRank results show the additional features are not absolutely necessary.

Another important distinction is that TextRank was used for *single document* summarization, while LexRank has been applied to *multi-document* summarization. The task remains the same in both cases—only the number of sentences to choose from has grown. However, when summarizing multiple documents, there is a greater risk of selecting duplicate or highly redundant sentences to place in the same summary. Imagine you have a cluster of news articles on a particular event, and you want to produce one summary. Each article is likely to have many similar sentences, and you would only want to include distinct ideas in the summary. To address this issue, LexRank applies a heuristic post-processing step that builds up a summary by adding sentences in rank order, but discards any sentences that are too similar to ones already placed in the summary. The method used is called Cross-Sentence Information Subsumption (CSIS).

3.3 Why Unsupervised Summarization Works

These methods work based on the idea that sentences "recommend" other similar sentences to the reader. Thus, if one sentence is very similar to many others, it will likely be a sentence of great importance. The importance of this sentence also stems from the importance of the sentences "recommending" it. Thus, to get ranked highly and placed in a summary, a sentence must be similar to many sentences that are in turn also similar to many other sentences. This makes intuitive sense and allows the algorithms to be applied to any arbitrary new text. The methods are domain-independent and easily portable. One could imagine the features indicating important sentences in the news domain might vary considerably from the biomedical domain. However, the unsupervised "recommendation"-based approach applies to any domain.

4 Incorporating Diversity: GRASSHOPPER algorithm

As mentioned above, multi-document extractive summarization faces a problem of potential redundancy. Ideally, we would like to extract sentences that are both "central" (i.e., contain the main ideas) and "diverse" (i.e., they differ from one another). LexRank deals with diversity as a heuristic final stage using CSIS, and other systems have used similar methods, such as Maximal Marginal Relevance (MMR), in trying to eliminate redundancy in information retrieval results.

We (Jerry Zhu, Andrew Goldberg, Jurgen Van Gael, and David Andrzejewski) have developed a general purpose graph-based ranking algorithm like Page/Lex/TextRank that handles both "centrality" *and* "diversity" in a unified mathematical framework based on *absorbing Markov chain random walks*. (An absorbing random walk is like a standard random walk, except some states are now absorbing states that act as "black holes" that cause the walk to end abruptly at that state.) The algorithm is called GRASSHOPPER for reasons that should soon become clear. In addition to explicitly promoting diversity during the ranking process, GRASSHOPPER incorporates a prior ranking (based on sentence position in the case of summarization).

4.1 Intuitive Explanation

For summarization, all of the NLP-related details are the same as in LexRank (i.e., states represent sentences, and edges are based on cosine similarity). The difference comes into how the ranking is performed.

Imagine a random walker on the graph. He takes steps randomly according to the transition matrix, though in some steps, the walker teleports to another completely different part of the graph. This is like the damping factor in the "random surfer model," where sometimes the surfer visits adjacent Web pages by clicking a link, but sometimes he jumps to a completely different Web page. In our case, the teleportation is not performed uniformly as in PageRank. Instead, we incorporate a prior distribution over states, based on a user-provided initial ranking, which influences where the walker teleports.

The diversity is achieved by iteratively ranking items or sentences as follows. The first item ranked is the one with the highest stationary probability (same as in PageRank). The stationary distribution does not do anything to promote diversity, though. So once this item is ranked, we need to do something different to get diversity. Thus, we turn the ranked state into an absorbing state. As noted above, an absorbing state stops the walker dead in his tracks. With absorbing states in the graph, we can no longer compute a stationary distribution, since eventually all walks will get absorbed and end at one of the absorbing states. However, we can look at the expected number of visits to each state before getting absorbed, and use this as the ranking criteria. This makes sense because, if walks end when they reach absorbing states, then the states furthest from the absorbing states will get the most visits. Among those dissimilar states, the most central, important states will get the most visits. We next rank the node with the most expected visits, which should be sufficiently different than the previously ranked sentences because of the absorbing property.

If the graph contains a few tightly connected components (clusters), then the ranking will begin with the center of one cluster. (If we did not use the absorbing state, all the high stationary probability items would come from this cluster.) This state will become an absorbing state, which has the effect of dragging down the importance of nearby neighbors. Thus, the next state to be ranked will come from one of the other dissimilar clusters. The process repeats until all the states are ranked. As should now be clear, the algorithm hops between distant areas of the graph, hence the name GRASSHOPPER.

4.2 Gory Details

The raw transition matrix $\tilde{P}$ is created by normalizing the rows of the graph weight matrix W:

$$\tilde{P}_{ij} = \frac{w_{ij}}{\sum_{k=1}^{n} w_{ik}}. \tag{2}$$

$\tilde{P}_{ij}$ is the probability that the walker moves to j once at i. We then make it a teleporting random walk P by interpolating each row with the user-supplied initial distribution $\mathbf{r}$, using a parameter λ:

$$P = \lambda \tilde{P} + (1 - \lambda)\mathbf{1}\mathbf{r}^{\top}, \tag{3}$$

where $\mathbf{1}$ is an all-1 vector, and $\mathbf{1}\mathbf{r}^{\top}$ is the outer product. If $\lambda < 1$ and $\mathbf{r}$ does not have zero elements, it can be shown that P has a unique stationary distribution π,

$$\pi = P^{\top}\pi. \tag{4}$$

We take the state with the largest stationary probability to be the first item g_1 in the GRASSHOPPER ranking

$$g_1 = \text{argmax}_{i=1}^{n}\, \pi_i \tag{5}$$

As mentioned above, the stationary distribution alone does not account for diversity. It is therefore important to compute the expected number of visits in an absorbing Markov chain. Let G be the set of items ranked so far. We turn the states $g \in G$ into absorbing states by setting $P_{gg} = 1$ and $P_{gi} = 0, \forall i \neq g$. If we arrange items so that ranked ones are listed before unranked ones, we can write P as

$$P = \begin{bmatrix} \mathbf{I}_G & \mathbf{0} \\ R & Q \end{bmatrix}. \tag{6}$$

Here $\mathbf{I}_G$ is the identity matrix on G. Submatrices R and Q correspond to rows of unranked items from the original P in (3). The *fundamental matrix*

$$N = (\mathbf{I} - Q)^{-1} \tag{7}$$

gives the expected number of visits in the absorbing random walk. In particular N_{ij} is the expected number of visits to state j before absorption, if the random walk started at state i. We then average over all starting states to obtain v_j, the expected number of visits to state j. In matrix notation,

$$\mathbf{v} = \frac{N^{\top}\mathbf{1}}{n - |G|}, \tag{8}$$

where $|G|$ is the size of G. We select the state with the largest expected number of visits as the next item $g_{|G|+1}$ in GRASSHOPPER ranking:

$$g_{|G|+1} = \text{argmax}_{i=|G|+1}^{n}\, v_i. \tag{9}$$

The GRASSHOPPER algorithm is summarized in Figure 1.

To see how λ controls the tradeoff, note when $\lambda = 1$ we ignore user-supplied prior ranking $\mathbf{r}$, while when $\lambda = 0$ one can show that GRASSHOPPER returns the ranking specified by $\mathbf{r}$.

Input: graph weight matrix W, prior distribution $\mathbf{r}$, graph vs. prior trade-off parameter λ

1. Create the initial Markov chain P from $W, \mathbf{r}, \lambda$ (3).
2. Compute P's stationary distribution π. Pick the first item $g_1 = \text{argmax}_i\ \pi_i$.
3. Repeat until all items are ranked:
   (a) Turn ranked items into absorbing states (6).
   (b) Compute the expected number of visits $\mathbf{v}$ for all remaining items (8). Pick the next item $g_{|G|+1} = \text{argmax}_i\ v_i$

Figure 1: The GRASSHOPPER algorithm

In each iteration we need to compute the fundamental matrix (7). This can be an expensive operation, but because the Q matrix is only changed by removing one row and one column in every iteration, we can apply the Sherman-Morrison-Woodbury formula. Then we need to invert the matrix only once in the first iteration, but not in subsequent iterations. This presents a significant speed up.

4.3 GRASSHOPPER Wrap-up

We have applied GRASSHOPPER to multi-document extractive summarization and found its performance to be comparable to the best systems in a community evaluation using the same datasets. GRASSHOPPER has the benefit of requiring only a unified procedure to rank sentences according to centrality and diversity, while most other systems first rank by centrality and then apply heuristics to achieve diversity. Furthermore, GRASSHOPPER nicely incorporates a prior ranking based on sentences' original positions with the documents (a reasonably good indication of importance).