Build Your Own Search Engine

An Introduction to Machine Learning Methods in Text Mining

Evan Hernandez (ehernandez4@wisc.edu, 907-092-9055)
David Liang (dliang23@wisc.edu, 907-034-1251)
Alec Yu (amyu@wisc.edu, 907-075-3653)

December 2016

Executive Summary

Search pervades everyday life in the twenty-first century. Students, researchers, and ordinary people alike enjoy the privilege of record-speed information retrieval from the Internet. Despite its fundamental importance, search is complicated. It leans heavily on the subject of text mining—the extraction of pertinent information from natural language documents—a subject which faces many challenges, both theoretical and practical. How does one model the content of a textual document? The same words appear in many documents, but each document may use those words differently. What determines a document's semantics? What about semantic similarity? Most importantly, how can one measure the semantic features of a document in real time? It turns out that many of these questions can be addressed at least in part by machine learning. In particular, the issue of semantic categorization can be reduced to a multiple-classification problem, one that is accurately solved by support vector machines.

We address these questions in the present paper, namely the questions of textual document representation and categorization. We will discuss methods for representing textual data, isolating the semantics of a textual document, and predicting attributes of new documents with learned classifiers. Specifically, we will discuss: term-frequency/inverse-document-frequency representations of text and measures of similarity between different texts; semantic noise reduction of many documents via latent semantic analysis; and supervised categorization of textual documents using k-nearest-neighbors and support vector machines. At the end, the reader will be tasked with building a simple search engine for the Reuters-21578 corpus, whose implementation will depend on the methods of text mining discussed in the background.

Contents

1 Background

1.1 Problem Description

Consider a law firm that wants to build a strong defense for its client: thousands of legal documents are available to the firm, but only a small fraction of the information contained in these documents is useful for the given case. How would lawyers go about gathering pertinent information or evidence? On one hand, the lawyers could read every word in every document available and then hope to remember it all by the end. On the other hand, such brute force is inhumanly difficult and exorbitantly expensive. More to the point, not all available information is useful. In fact, chances are that the vast majority of legal documents are irrelevant to the case at hand. For example, if the lawyers are handling a case on tax evasion, they probably do not want to read about domestic laws. The firm ultimately needs an automated and highly efficient method for organizing textual information and for retrieving the pertinent information quickly.

This problem of organizing textual information is not new. Humans began writing somewhere around 5000 B.C. and since then have created a monolithic amount of written information. For hundreds of years, the only way to research pertinent information was to visit a library and page through book after book. The process was slow and oftentimes unfruitful. Then, with the advent of the internet, the world rejoiced at the new availability of information and yet faced an organizational crisis. What should be done with this monolith of textual information? In what ways might all this information be useful? Most importantly, how could this information be made accessible to the general public? These questions emphasized the need for search engines, programs that could troll through large masses of web content and pick out only the relevant bits of information in real time. During the 90s, companies like Yahoo!, Ask Jeeves, and eventually Google sprang up, each offering unique methods for indexing webpages and providing speedy information retrieval. Search engines have since become a staple and keystone for day-to-day life in the twenty-first century.

Underpinning much of search is the concept of **text mining**. Text mining refers to the organization and extraction of information from natural language documents. Note that search and text mining are not one and the same. Search engines typically undertake a number of steps to produce results, including indexing and query enrichment. By contrast, text mining is a more general collection of methods for extracting data from textual documents. Some examples of text mining include part-of-speech tagging, concept extraction, and sentiment analysis. In this paper, we will focus on the representation of textual documents as vectors and the categorization of textual documents into predefined categories. We will see that, with the right precautions, document categorization works out to be a fairly straightforward classification problem.

Document categorization has many immediate applications. Lawyers may wish to automatically sort large quantities of text into semantic categories (domestic law, tax law, etc.); companies may wish to categorize incoming email and eliminate spam; search engines may want to categorize new queries to restrict their search space; and so forth. Despite its usefulness, document categorization is not trivial. It is not immediately obvious what determines whether a document belongs to a category. Language is by nature ambiguous. Some studies from cognitive science suggest that this ambiguity is evolutionarily advantageous because it eases the language learning process for young children, but that ease does not surface for computers. In particular, polysemy—the presence of multiple meanings for the same word—makes it difficult for computers to disambiguate between pairs of sentences like "I climbed the steep bank," and "I deposited a check in the bank." On the other hand, language is often redundant and dilutes information with unnecessary synonyms and clarifiers like pronouns and tense. Worse yet, the categorizer may have limited access to information about the documents. In the case of web search engines, the webpages will not come with category labels; it is up to the search engine to organize the data into semantic categories.

To address these problems, we need an effective way to encode the semantics of a document without getting lost in the ambiguity and noisy redundancy of natural language. We present methods for doing so in the proceeding section.

1.2 Representation of Textual Data

In order to extract information from documents, we must first define what we mean by document. For our purposes, a **document** is a sequence of natural language tokens. In this sense, the sentence "I ate an apple" is as much a document as the entire play *Hamlet*. It is worth noting that although our definition of document is limited to text, other definitions are typically more general so as to include any collection of information in written or electronic form. In fact, many of the methods for text document representation and categorization that we discuss here will apply to other types of documents as well.

As one might expect, raw strings are not conducive to mathematical manipulation and therefore are not conducive to large-scale search. Performing string comparisons is both slow and inaccurate; the user is looking for a document semantically similar to his query, not for an exact textual match to his query. The alternative is to represent documents as vectors by defining a global list of features, and then encoding each document in terms of these features. Thus, for a corpus with m features, each document will be encoded as an m-dimensional vector. The list of features is typically determined by the characteristics of the entire corpus.

Here, we will employ the term-document model. In the term-document model, each feature corresponds to a term from the collection of all terms taken over all documents. Its value is given by the number of times that term appears in the document. For example, let $d = [d_1\ d_2\ \ldots\ d_m]^T$ be the feature vector given by some new document, and $\mathcal{V}$ the collection of all terms (features) in the corpus. If term $v \in \mathcal{V}$ appears in the document k times, we have $d_v = k$, noting that we let each term correspond to some index of the

vector. Unfortunately, this encoding is still meaningless: a document might contain the word "the" in 1256 different places, giving it a large feature value, but that does not mean the word "the" is semantically salient. As a first step to remedying this, we need to weight each term inversely to how often it is used throughout the corpus, under the assumption that rarer words better identify the semantics of a document. This encoding is called the **term frequency-inverse document frequency (tf-idf)** because each term frequency feature is multiplied by the inverse of its corpus frequency.

This encoding gives rise to a natural similarity measure for documents, namely, the normalized dot product or cosine similarity measure. For documents d_i and d_j in the same corpus, the similarity of d_i and d_j is given by $\frac{d_i^T d_j}{||d_i|| \cdot ||d_j||}$. Any features that are not shared by both documents will be zeroed out, so the result is entirely determined by tf-idf values for terms that appear in both documents. Moreover, since common terms will have low weights, they will contribute little to the dot product, while shared rare terms like "interpolate" or "Nietzsche" will contribute a great deal to the dot product. This is in line with our assumption that rare words probably have semantic salience. Note that we can also obtain a measure of term similarity in this way: if we arrange each document vector as a row in a matrix, we see that the columns form feature vectors for terms, where the i^{th} feature is the tf-idf of that term in document i. This is an intuitive representation because terms that are semantically similar will likely appear in the same document. As one might expect, the normalized dot product of two term vectors gives a rough similarity measure of those terms.

The term-document model is good at highlighting the semantic hotspots of a document—that is, the rare and potentially salient words—but it does not completely remove the redundancy of natural language. We must take other steps to reduce the noise from each document. One cheap solution is to remove short words from the feature space. This works because most short words are either articles or acronyms that are spelled out elsewhere in the text; however, there is still the risk of eliminating important information. Supposing we chose to remove all words of length three or less, say, in an effort to remove "the" from consideration, we might in the process eliminate the word "bee" from a text on endangered species, which is clearly undesirable. A more effective approach is to remove all **stop words** from the corpus, where a stop word is defined to be any word that contributes no information to the text, like "the." The collection of stop words varies from case to case.

Finally, before featurizing a corpus, we should make sure to remove any unnecessary prefixes or suffixes from each word, so that the words "walk" and "walked" are not considered separate features. This is called **stemming**. We omit the details.

1.3 Latent Semantic Analysis

In the previous section, we hinted that an entire corpus can be represented as a single matrix, where each row is a document vector. In fact, this representation exposes information about the corpus as a whole, and provides a means for extracting and simplifying that information.

The astute reader will have noticed that the term-document model results in sparse vectors with many dimensions. Some of these dimensions are virtually useless, while

others match the semantic category of a document but are zeroed out because the term represented by that dimension does not appear in the document. Despite our efforts at normalization with the term-document model, we still have not accounted for the problem of synonomy. We want to uncover the latent semantic structures that underly a corpus. In other words, we would like to discover the semantic relationships between terms and to combine synonyms such that our term vectors (the columns of the corpus matrix) represent something closer to a semantic category than a specific word. For example, "chihuahua" and "dog" are different words but both belong to the same semantic category. From a search perspective, we would like to consider a document that mentions chihuahuas as similar to a query for dogs.

We can achieve this dimensionality reduction by performing **latent semantic analysis (LSA)** on the corpus matrix. Latent semantic analysis involves finding a low-rank approximation of the corpus matrix. To do this, we apply the Eckart-Young theorem, which states that the best k-rank approximation for an $n \times m$ matrix, $k < m$, over some unitarily invariant norm is given by the outer product of the the first k singular values and vectors of the original matrix. Formally, if the singular value decomposition of a matrix A is given by $\sum_{i=1}^{r} \sigma_i u_i v_i^\intercal$, then the best k-rank approximation to A is:

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^\intercal$$

We can think of this approximation in a number of ways. On one hand, some dimensions (terms) are being zeroed out, namely those dimensions with the least semantic significance across all documents, so we are reducing the sway of redundant or unhelpful term features. On the other hand, terms that were previously zeroed out in certain documents may now be nonzero. We can think of this as stretching terms into k of the most salient semantic categories across the entire corpus, where each term has varying membership to each category. Notice that this analysis depends heavily on the assumption that similar words in semantic space will generally co-occur in documents.

Typically, we choose $k \approx 100$, though studies have shown that there are good results with k as small as 50 or as large as 1000.

1.4 Categorization of Textual Documents

We now have the tools to discuss methods of document categorization. As we have suggested, document categorization boils down to a supervised classification problem with a few modifications. We will discuss and compare three different classifiers as they relate to document categorization: k-nearest neighbors (kNN), least squares (LS), and support vector machines (SVMs).

1.4.1 Formalization

We begin by formalizing document categorization as a classification problem. Let $\mathcal{D}$ denote the set of all documents in the corpus and $\mathcal{C}$ the set of all semantic categories, which we take to be predefined. Formally, our input is a set of tuples (d_i, C_i), where $d_i \in \mathcal{D}$ denotes the tf-idf vector for document i and $C_i \subset \mathcal{C}$ denotes the set of categories to which document i belongs. Notice that the labels are not individual objects but sets. It is easy to imagine cases where a document belongs to multiple semantic categories. Thus, for a new, unlabeled document d_* we would like to predict the set of semantic categories C_* to which it belongs.

Already it should be clear that this problem is more complicated than other classification problems. The classifiers we will discuss are prototypically binary classifiers—that is, new examples belong to exactly one of two classes—so we will have to find an adequate modification that (a) can classify examples into more than just two categories and (b) allows examples to belong to more than one category. This is called the problem of **multiple classification**, and we address it for each classification method in turn.

1.4.2 k-Nearest Neighbors

The first classification scheme we will discuss is **k-nearest neighbors**. We can imagine our document vectors as points in m-dimensional semantic space, where spatial proximity correlates with semantic similarity. The kNN classification scheme grounds itself in this idea and assumes that new, unlabeled examples will be similar to the labeled examples that are nearby in feature space. Specifically, for $k > 0$, the kNN classifier chooses the k points from the training data that are closest in Euclidean distance to a new data point and takes the majority class of the nearest neighbors to be the class of the new example. To visualize this, consider figure X: if $k = 3$, then the new point would be labeled as red since two of its three closest neighbors are red; if $k = 5$, the new point would be labeled blue.

kNN has the advantage of naturally addressing the mutliple-classification problem. For each of the k nearest neighbors, we increment the vote count for each class to which that neighbor belongs. We then ignore classes whose vote counts are lower than some constant threshold and choose the r most-voted of the remaining classes to be our classification.

This kNN classification scheme has been shown to work well for document categorization. In fact, it is the most common method of document categorization because it is easy to implement and because it does not require us to train a classifier. The latter advantage is also a significant disadvantage, however, because new training documents must be compared to every training document to obtain a classification, which results in $O(nm)$ classification time complexity for corpora with n documents and m term features. This complexity is especially bad when we consider that both n and m are typically quite large. Another disadvantage of kNN is that the classification results may vary as you change k. We could compute the optimal k by trying many values of k and choosing the one that minimizes classification error—this results in a pareto curve where k is graphed on the x-axis and classification error is graphed on the y-axis. Unfortunately, this computation is slow and expensive for large datasets.

1.4.3 Least Squares

Next we will discuss the **least squares** classifier. Least squares classification grounds itself in the assumption that classes are determined as a linear combination of term features. We want to learn the optimal coefficients for said linear combination. To do this, we minimize the squared loss between predicted classes and actual classes in a set of training data. Formally, given our document matrix A, a vector w of weights, and a vector b of numerical labels, the predicted labels are given by $\text{sign}(Aw)$, where the sign is taken element-wise, and we wish to find $\text{argmin}_w ||Aw - b||_2^2 = \hat{w}$. We can then classify a new document a by evaluating $\text{sign}(a^\intercal \hat{w})$, where again the sign is element-wise. Practically, we can solve this problem by computing the pseudoinverse of A and multiplying it by b.

As we've already seen, least squares on its own is not sufficient for document categorization because it cannot accurately distinguish handle more than two categories. In practice, we have two choices for multiple-classification schemes with binary classifiers. The first scheme is called **one-vs-all classification**, where one classifier is trained per category that learns only to distinguish its category from all others, and each classifier returns a continuous measure of class membership as opposed to a strict yes/no answer. Thus, if there are k classes, then the one-vs-all schema trains k different classifiers. The second schema is called **one-vs-one classification**, where $k(k-1)/2$ binary classifiers are trained, one for each unique pair of categories. Each classifier learns to distinguish between its two given categories. To classify a new example, we apply each classifier and choose the class with the majority vote.

Both schemas have advantages and disadvantages. However, one-vs-one classification tends to be the method of choice. Although it requires us to train more classifiers, we train each classifier on fewer examples. Moreover, the voting system helps to eliminate noise in the data and gives us a means of determining multiple-class membership for new examples—we need only iterate through the class votes.

If we again think of document vectors as points in semantic space, then least squares finds the optimal hyperplanes for separating each pair of categories. Of course, the data might not be linearly separable, but we can make it separable lifting each point into a higher-dimensional space with kernel methods, and then we can apply least squares in the new feature space. We omit the details.

Least squares tends to perform well on document categorization tasks, particularly when the semantic structure of a corpus has been compacted with latent semantic analysis. Although the kernelized SVM is preferred to least squares in most applications, the two classification schemes have comparable performance. In both cases, the loss function is generally regularized with respect to the L_2 norm so as to ensure that no one term carries too much weight. This makes sense because the semantics of a document are almost never determined by a single term—rather, the semantics depend on the culmination of all terms and concepts presented in the document.

1.4.4 Support Vector Machines

The last classifier we will mention is the **support vector machine**. Formally, the support vector machine learns the optimal weight vector w such that the hinge loss given by $\sum_{i=0}^{n}(1 - y_i a_i^\intercal w)_+$ is minimized, where y_i is the label of the i^{th} document, a_i the feature vector for the i^{th} document, and the last subscript represents a soft threshold.

Like least squares, the SVM finds a separating hyperplane between two classes. However, least squares struggles with outliers in the data. If a point is categorized too correctly, then its squared distance from the hyperplane will be large and thus the least squares hyperplane will be pulled toward it. By contrast, the SVM does not penalize data that are categorized very well. This is because the SVM finds the hyperplane with the greatest margin between the two classes. As before, the data may not be linearly separable, but we can remedy the problem using kernel methods.

As we have suggested, the SVM is the preferred classifier for document categorization. When dealing with text classifiers, each word in the text is an entry in the vector of text features. With large text files, we are dealing with feature vectors with thousands of features.

2 Lab

Welcome to the lab where you?ll be building your own search engine! In this lab, you?ll be using the Reuters corpus which is the most widely used text collection for text categorization research. It has 21,578 documents, each labeled with multiple categories making this text classification different than regular single class classification. The lab will require you to program basic functions for a search engine like low-rank approximation, classification using least squares, k-nearest neighbors, and support vector machines (for comparison), in addition to a final search function that brings all of this together.

<how the search engine will work>

<what the skeleton looks like>

2.1 Setup

First, you will need to download our skeleton code and make sure you have the required software and libraries to run it. We used Python 2.7.11 and pip 9.0.1 on OSX 10.0.1, but these versions for Python and pip are not strictly required. If you do run into problems, we recommend that you just upgrade to the latest versions of pip and Python 2.7. The installation steps are as follows:

1. Install Python and pip. If you need help, look [https://wiki.python.org/moin/BeginnersGuide/Downloa and [https://pip.pypa.io/en/stable/installing/]here.

2. Install Python libraries dill, nltk, sklearn, and numpy through pip.

3. Download the data by running `python -m nltk.downloader punkt stopwords reuters` or by using `nltk.downloader()` from within Python and installing the punkt, stopwords, and reuters datasets.

4. Finally, you?ll need the code we?ve provided, which you can find in the appendix or at `https://github.com/EvanFredHernandez/byose`.

2.2 Warmup

1. Get the document vector with the document id ?training/309? using the `document_vector` function of a Corpus instance. The ?training/309? document

is about gold mines in South Africa. The document vector is long, but take a look and explain what the vector represents (what do the columns correspond to?) and why there are so many zeros. If you?re curious about the content of the document, you can call the static method `document_text(doc_id)` to have a look.

2. Now get the document vectors for ?training/309? and ?training/448? and compute the dot product between them. The ?training/448? document is about Brazilian gold mines. Now compute the dot product between ?training/309? and ?training/3358? and compare this result to the previous. The ?training/3358? document is about agriculture and cereal ingredients. What do the dot products mean? Do the results make sense, relative to each other?

3. Now suppose that you have a matrix X that?s composed of stacked document vectors, like the ones you retrieved in the preceding parts. What would be the meaning of a matrix $XX^\intercal$ and each of its elements? What about $X^\intercal X$?

2.3 Build Your Own Search Engine

You will now implement the functions used by the search engine.

1. Implement k-rank approximation. In the test method, use your function to find the rank 2 approximation of [[1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6]]. We?ll use this method later.

2. Let?s try categorizing documents the old-fashioned way and use k-nearest neighbors.

   a) Implement the `knn` function. Since the document vector could be in the document matrix, be sure to not count the document vector when found in the document matrix as a nearest neighbor.

   b) Find document with ID 'training/1684' and read the text. Then use the `knn` function to find its 3 nearest neighbors (not including itself) and read the text of those documents. Are they similar? Does these results make sense?

   c) Now test the accuracy of your `knn` function by implementing `test_knn`. For each category, take all documents in that category, classify each of them, and check to see if the category is in the classification.

3. We will now implement the classification component of the lab.

   a) Implement `train_ls_classifier` (with ridge regularization) and `compute_categorization_erro` Use your favorite iterative method for the first. Use 10000 iterations, with $\lambda = 0.001$ and $\gamma = 0.001$. Train on two very similar categories (coconut vs. coconut_oil) and two very different categories (coconut vs. copper). Which performed better? Why?

   b) Implement `train_svm_classifier` using gradient descent. Use L2 regulariza-

tion for the sake of comparison. Train on same categories as above. How do your results compare to the least squares classifier?

c) Use these functions to implement the functions `train_one_vs_one_classifier` and `one_vs_one_classify`. We will represent one-vs-one classifiers as the tuple (<weights>, <+1 category>, <-1 category>). Note: Make sure you are computing $k(k-1)/2$ classifiers and NOT k^2 classifiers.

4. Finally, write the search function and then test it out! Run main.py with a few of your own queries. You can also try ?bahia cocoa zone.? How does it work?

3 Final Thoughts

In the last 20 years, the number of internet users has increased hundred-fold. Now over half of the world population uses it on a daily basis and a large part of that growth is attributed to the availability of basically all documented knowledge in the past several decades. The size of the internet is estimated to be in the thousands of terabytes; with all this data floating around, the most effective way to find what you want is through the use of search engines. The most popular, Google, receives almost 6 billion search queries each day, and text classification plays a big role in processing them. Text categorization techniques are used to classify news stories, to find interesting information, and to guide a user?s search. Historically, search engine technology and text classification techniques have grown hand in hand. Text classification isn?t just limited to search engines, it also has applications in many other areas such as spam filtering and language identification. By exploring text classification, we can develop a fundamental understanding in the inner workings of the world?s most useful tool. We hope to have taught you interesting and insightful information in this section. We?d now like to guide you through the construction of a simple search engine where you?ll apply what you?ve learned!

A Appendix

We have included the solutions to each exercise. The skeleton code is omitted for concision, but can be found at `https://github.com/EvanFredHernandez/byose`.

A.1 Warmup Solutions

<warmup solutions here>

A.2 Lab Solutions

<lab solutions here>