A recommendation engine for Wikipedia articles based on constrained training data

John Rothfels Brennan Saeta Emin Topalovic

Abstract

We consider the problem of generating recommendations for Wikipedia articles based on constrained data. Modern recommendation systems commonly use a combination of collaborative filtering techniques and content-based methods to predict new items of interest for a user. We focus our problem on generating recommendations via content-based analysis using only a small set of *liked* articles as training data and no other information about user preferences or other users in the system. We find that our methods are promising for users with many *likes*, but that our algorithms do not generalize well to more constrained data. Concretely, other methods are needed to produce good results when a user has a small set of *likes*.

1 Introduction

The popularity of Wikipedia has skyrocketed since its inception. People use the site to learn, to research, and to explore. Given its prevalent use, and the previously successful applications of machine learning to recommendation problems, we propose to build a recommendation engine for Wikipedia articles. We formulate our problem as follows: given a (non-empty) set L_i of Wikipedia articles that a user i has enjoyed or *liked*, we wish to generate a non-intersecting set R_i ($L_i \cap R_i = \emptyset$) of Wikipedia articles the user will also find interesting or enjoyable to read. Our challenge is in generating measures of similarity between articles that generalizes to allowing our recommendation engine to accurately suggest articles related to the user's preferences, as well as *out-of-the-box* articles which (possibly unrelated to the user's likes) he or she may still enjoy. An additional challenge is working with the massive dataset that comprises the 17+ million articles of Wikipedia.

Unique to our problem is the nature of the data over which we wish to generate recommendations. We imagine a situation in which a user has potentially *liked* only a small number of articles ($|L_i| \geq 4 \ \forall \ i$), based on what we believe to be a reasonable assumption about how much a user can be expected to manually curate their preferences. While some users might actively curate their interests, others may not. We wish to generate good recommendations in either case, without resorting to collaborative filtering techniques such as generating recommendations based on what other (similar) users have liked.

2 Preliminary Steps

2.1 Generating a Dataset and Subset of Wikipedia Articles

Our first task was to work towards generating a dataset on which to test our recommendation algorithms. We needed not only an unbiased dataset which reflects the diverse preferences of multiple Wikipedia users, but also a more tractable subset A of Wikipedia articles on which to test our approach. At the time we started our project, there were over 17 million articles on Wikipedia, comprising about 33 gigabytes of data. Given this large amount of data, retrieving a similarity measure of each of these articles to users' preferences would be computationally infeasible (with our hardware setup).

Since we found no preexisting data for Wikipedia user *likes*, to generate a dataset we asked real users (via a web survey which we wrote and distributed) to select Wikipedia articles that they would be interested in reading. To make this survey more feasible for users to complete, we again wanted a reasonable sized subset of articles for them to choose from which would allow for diversity of articles and topics but not incentivize users to rush through the survey, causing them to submit disingenuous preferences.

For the reasons given above, $|A|$ was set arbitrarily at 863. However, we felt that choosing this subset should *not* be done randomly, since many articles in Wikipedia are stubs or pages redirecting to other articles. We wanted to choose only articles with high content value so that the selections would be rich and more inclusive of users' actual interests. Towards this end, we implemented and ran the PageRank algorithm [5]. We first processed our articles into a matrix G such that $G_{i,j} = 1$ if article i links to article j. After a little less than 180 hours, our implementation[1] had converged such that the change in ℓ_1 norm was less than 1×10^{-5}. Given each article's PageRank, we selected the top $m = 863$ articles to be A.

We received 50 submissions to our web survey, each containing at least 15 *liked* articles, with some submissions including more than 100 selections. From each user's submission, we randomly chose 30% of the selections to constitute a *like set* (L_i) which we use for generating recommendations; the other 70% constitutes the articles we hope to recommend and which all reported data is based on.

Note: in the recommendation domain, it is often infeasible to give more than a small number of recommendations (e.g. due to space limitations on a web page) and in our situation, we are indeed concerned with returning a reasonable number of recommendations. Moving forward, our decision was to try and create algorithms which learn to return good recommendation sets without artificial pruning or post-processing steps, so recommendation sets are reasonably sized by default.

3 Method and Experiments

3.1 Initial Strategy and Results

Given A, the next task was to convert each article into a feature vector. We chose to start by considering the verbs, nouns, and adjectives appearing across all articles in A as separate sets of features. As a preprocessing step, we formed a dictionary of verbs, nouns, and adjectives which appear in our corpus. To standardize the dictionaries, we used Stanford's Part-of-Speech tagger [7] to select nouns, adjectives and verbs; we used Porter's stemming algorithm [8] to reduce dictionary dimension.

For each article $a \in A$, we produced feature vectors of the form $V_a = \{v_1, ..., v_n\}$ where each v_i is either the count in a of the i^{th} dictionary word or $I\{v_i \in a\}$, a binary variable indicating existence of the i^{th} dictionary word in a. We produced separate feature vectors for nouns, verbs, and adjectives.

On each vector type, we evaluated the capability of k-means to produce meaningful clusters of articles. We found that while a few clusters were in fact informative (see below), most were either very sparse or too large to be useful no matter the number of clusters generated. This motivated our exploration of better feature choices.

Some examples of (qualitatively) interesting clusters:

- Articles of Confederation, American Civil War, Abraham Lincoln
- Ancient Egypt, Avicenna

3.2 Revised Strategy

We decided to build a new dictionary containing the stems of all "useful" words in A. More concretely, we ran Porter's stemmer over all $a \in A$, and further introduced a list of standard stop words to remove uninformative English words from a such as "the", giving us a set of words a'. The dictionary generated is the union of all a'. Finally, to reduce the dimension of the dictionary, we removed words whose frequencies across all articles fell out of a specified range (arbitrarily set at 5-20), and were able to reduce the number of words in the dictionary from $330,000$ to under $20,000$. This choice was motivated by the intuition that the most useful words to use for generating content similarity measures between articles are those that are neither extremely rare nor extremely common.

We further decided to switch from the naive approach of generating feature vectors from count/existence of nouns/adjectives/verbs and instead began considering methods that have proven to be successful in related works [6] [3]. We chose to do term frequency-inverse document frequency (TFIDF) as a measure of word importance within each article. This allowed us to introduce a more intelligent notion of word importance, rather than count. To introduce semantic information, we considered latent semantic analysis (LSA) which encodes the contextual usage of words [2]. We further considered latent Dirichlet allocation (LDA) as a way to consider a different feature paradigm, namely one which models a mixture of deduced topics as the representation for each article [1].

[1]Written in Python, leveraging the Numpy package.

Continuing on, we will differentiate between the different features choices as $V_a^{[tfidf|lsa|lda]}$.

We applied clustering to our feature vectors using k-means. We utilized k-means in two ways which we will refer to as *naive* and *hierarchical*. For *Naive k-means*, we cluster V_A into $k \in \{25, 50, 100, 500\}$ clusters, where V_A is the set of feature vectors for A. For a given user i, the recommendation set R_i returned is the union of clusters containing articles from L_i. This method gives us an idea of how well our features allow simple clustering to not only cluster potential articles of interest, but further prune out irrelevant articles to give a manageable recommendation set. *Hierarchical clustering* is a more direct way of producing a targeted and manageable recommendation set. We begin as we do with the naive approach, namely clustering V_A. For a given user i, we then iteratively re-cluster the union of clusters containing articles in L_i until we achieve convergence; the recommendation set R_i returned is the union of all re-clustered clusters containing articles in L_i. Convergence is reached when subsequent runs stop adding a significant number of new articles to the recommendation set which would be returned. Concretely, we define convergence as:

$$\Sigma_{n \in N} \frac{n}{4} \leq 15$$

where N is the set of number of articles returned in the four iterations previous to the current one. Namely, n_i is the number of articles added to the recommendation set going from iteration $i-1$ to i. The following is an outline of hierarchical k-means:

```
def hierarchical_k_means(articles):
  rec = set() # Articles to recommend
  # Add all articles that are clustered
  # with elements in the likeset
  for article in likeset:
    rec += articlesInCluster(article)
  if len(rec) < num_recs or converged:
    return rec
  return hierarchical_k_means(rec)
```

We also wanted to see how supervised learning algorithms could perform in generating recommendation sets. Namely, we considered the efficacy of logistic regression and SVMs with gaussian and linear kernels. This presented the unique challenge of deducing negative, or *disliked* examples, given that our input is only L_i, a set of *likes* for user i. To do this, we leverage clustering and choose the *dislike set* D_i to be those articles least similar to articles in L_i. Concretely, we define the weight of a positive cluster C_p to be the number of articles from L_i that are in that cluster. For each positive cluster, we sort all non-positive clusters by distance to the centroid. The farthest clusters are weighted more than the closer ones. Finally, once we've calculated this for all positive clusters, we sample articles from the negative clusters according to their total weights. These chosen articles are the *dislike set*.

4 Results

Given our choice to look for algorithms which return appropriately sized recommendation sets, we chose F_1 score as a metric of performance. This score rewards recommendation sets that contain articles we know a user will like (based on the 70% held out data) but penalizes for recommending other articles. Most importantly, the F_1 score will penalize us for returning large recommendation sets when the user has specified only a small number of *likes*. For each of our 50 data points (and for a given choice of algorithm), we compute an F_1 score from the recommendation set returned by the algorithm. Numbers reported below are average F_1 scores across all 50 data points.

To determine the significance of our results, we computed t-tests (unpaired, heteroscedastic, two-tailed) between the F_1 scores of our algorithms and found no statistically significant difference between any of our algorithms or feature choices ($p \geq 0.05$).

4.1 Clustering

	25	50	100	500
V^{TFIDF}	0.097	0.09	0.097	0.084
V^{LSA}	0.1	0.099	0.098	0.088
V^{LDA}	0.096	0.095	0.092	0.05

Table 1: F_1 score of naive clustering versus cluster size

	25	500
V^{TFIDF}	0.072	0.083
V^{LSA}	0.07	0.09
V^{LDA}	0.068	0.05

Table 2: F_1 score of hierarchical clustering versus cluster size

4.2 Supervised Methods

	Gaussian	Linear	Regression
V^{TFIDF}	0.99	0.069	0.07
V^{LSA}	0.059	0.072	0.073
V^{LDA}	0.014	0.023	0.023

Table 3: F_1 score of different supervised methods

5 Conclusion

5.1 Recommendation Set Sizes

Across most of our algorithms (hierarchical clustering the exception) we found that recommendation set sizes were generally quite large, often greater than 100. While this made the recall of our algorithms quite good, our precision was often very poor. Despite various changes to our algorithms (e.g. increasing the number of clusters, increasing the size of the *dislike* set), recommendation sizes continued to be large, leading us to suspect that post-processing on recommendation sets is necessary to reduce size. We consider options in the Future Work section.

5.2 Small Input Sizes

As we looked over numbers, we noticed that our algorithms consistently perform better for users with larger like sets. In order to understand this, we plotted the F_1 score as a function of the size of a user's like set, L_i. We find a practically linear relationship. As users tell us more information about what they like, our system works better and better.

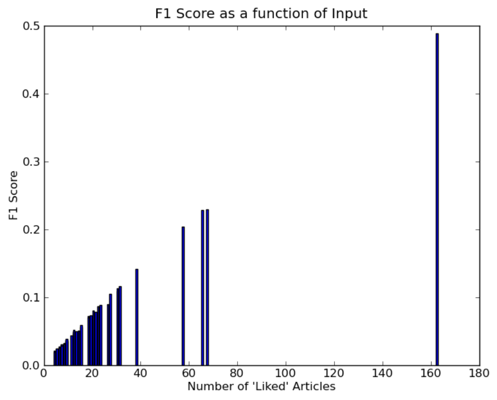

Figure 1: More input data gives us better results. Graph of SVM with linear kernel.

We see an almost identical graph for all of our algorithms, leading us to suspect that one of our fundamental problems is the constraint of our problem, namely to have algorithms which perform well given small like sets. We may simply need more training data in order to perform well.

5.3 Classification sensitivity to Negative Article Selection

We wanted to see how much of a difference a simple negative article selection performs against our hand-crafted one.[2] We ran this for TFIDF features, using a Gaussian kernel.

Hand-crafted	0.098880461728
Simple	0.0585947853134

Table 4: TFIDF features, Gaussian kernels. Two different negative generation systems.

We find that our negative article generator improves the F_1 score of our Gaussian SVM. This is statistically significant result.[3] Using a better negative-article selection algorithm can potentially increase our F_1 score.

5.4 Overall

Recommending interesting Wikipedia articles given only a handful of good articles is not an easy task. Limiting ourselves to content-based recommendation systems, and only a few training examples per person is an especially challenging problem. Holding all other variables constant, we found that the F_1 score increased almost linearly as a function of the size of a user's *like set.* Further, we found that the classification algorithms were sensitive to the negative articles selected.

A general concern we have is about the dataset itself. Although we found a way to associate numbers with our algorithms, we feel that the dataset is far from perfect. Specifically, we have assumed that the survey results from each user contains every article in A that the user finds interesting. Since there is no way for us to enforce this, one can imagine that the true (or unbiased) set of results should contain many more articles than a user actually selects during the

[2]The simple negative article selection simply returns one article no matter what positive articles are passed in. We chose this over randomly selecting articles because this allows for a consistent measure across runs of the algorithm.

[3]Running a unpaired, heteroscedastic, two-sided t-test with the null hypothesis that the two means are the same, we find that we reject the null hypothesis ($p = 0.0217$).

survey, either because he/she forgot to select some articles, or tried to complete the survey in haste, or any other of a multitude of reasons. A more interesting example of this is somewhat philosophical, that often times users don't actually know what they want or might like. They may think they know, and will act accordingly, but there is no reason why some of our recommendations should be "wrong" in the way we say they are (i.e. if the recommendation is not in a user's 70% held out set).

6 Future Work

We found one of the major shortcomings of our approach was that we could successfully generate good recommendations when a user has a large set of preferences, but that when there is limited data, our problem is extremely hard to solve. For users with small *like sets*, it is possible (and potentially desirable) to use collaborative-filtering techniques to augment their *like set* before running our algorithms. More concretely, in a preprocessing step, we can augment a user's *like set* by finding the k users with most similar preferences and considering the union of their *like sets* as we run our algorithms.[4] Alternatively, we can base the recommendations themselves on the *like sets* of similar users, vis-a-vis a "Others like you enjoyed..." system.

Our research suggests that another shortcoming of our system is using feature vectors whose dimensions are too large. Previous work has obtained compelling results deducing article topic with only 10 to 20 features. This same work found that SVMs can be highly sensitive to feature vector size, and futhermore, that the optimal size can depend on kernel choice [3]. Even after removing all words in our dictionary which occur less than 5 and more than 20 times across all articles, we still have more than $20,000$ features.

Finally, we feel that our strategy for returning reasonably sized recommendation sets can use significant improvement. Recall our approach to this problem was to generate algorithms which (by default) return not only good recommendations, but also a small number of them (given the requirement that most recommendation engines are limited in the number of recommendations they can give). With the exception of our hierarchical clusterer, our algorithms (generally) return recommendation set sizes over 20 despite our efforts to prevent this from happening. While this boosts our recall, it negatively affects our precision. This suggests the need to modify our approach to find algorithms which generate good recommendation sets, but require additional post-processing steps which prune the recommendation sets to reasonably sized subsets. There are many feasible methods and heuristics to accomplish this; one suggestion is to have our SVMs not return the set of articles classified positive, but rather the k articles given the highest probability (margin) of being positive examples.

References

[1] David M. Blei, Andrew Ng, and Michael Jordan. Latent dirichlet allocation. *JMLR*, 3:993–1022, 2003.

[2] T. K. Landauer, P. W. Foltz, and D. Laham. An introduction to latent semantic analysis. *Discourse Processes*, 25:259–284, 1998.

[3] Larry M. Manevitz and Malik Yousef. One-class svms for document classification. *J. Mach. Learn. Res.*, 2:139–154, March 2002.

[4] Raymond J. Mooney and Loriene Roy. Content-based book recommending using learning for text categorization. In *Fifth ACM conference on Digital libraries*, DL '00, pages 195–204, New York, NY, USA, 2000. ACM.

[5] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[6] Wongkot Sriurai, Phayung Meesad, and Choochart Haruechaiyasak. Recommending related articles in wikipedia via a topic-based model. In *IICS*, pages 194–203, 2009.

[7] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of EMNLP*, EMNLP '00, pages 63–70, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[8] C.J. van Rijsbergen, S.E. Robertson, and M.F. Porter. New models in probabilistic information retrieval. *Unknown*, 1980.

[4]There are various measures of textual similarity which can be considered, some of which use the feature vectors we have already generated for each article.