Kvasir: Seamless Integration of Latent Semantic Analysis-Based Content Provision into Web Browsing

Liang Wang Sotiris Tasoulis Teemu Roos Jussi Kangasharju
Department of Computer Science
University of Helsinki, Finland
{firstname.lastname}@cs.helsinki.fi

ABSTRACT

The Internet is overloading its users with excessive information flows, so that effective content-based filtering becomes crucial in improving user experience and work efficiency. We build Kvasir, a semantic recommendation system, atop latent semantic analysis and other state-of-art technologies to seamlessly integrate an automated and proactive content provision service into web browsing. We utilize the power of Apache Spark to scale up Kvasir to a practical Internet service. Herein we present the architecture of Kvasir, along with our solutions to the technical challenges in the actual system implementation.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering, information filtering*

Keywords

Information Retrieval; Content-based Filter; Web Browsing; Latent Semantic Analysis; Random Projection; Big Data

1. INTRODUCTION

The Internet is overloading its users with excessive information flows. Therefore, smart content provision and recommendation become more and more crucial in improving user experience and work efficiency. E.g., many users are most likely to read several articles on the same topic while surfing on the Web. Hence many news websites (e.g., The New York Times, BBC News and Yahoo News) usually group similar articles together and provide them on the same page so that the users can avoid launching another search for the topic. However, most of such services are constrained within a single domain, and cross-domain content provision is usually achieved by manually linking to the relevant articles on different sites. Meanwhile, companies like Google and Microsoft take advantage of their search engines and provide customizable keywords filters to aggregate related articles across various domains for user to subscribe. However, to subscribe a topic, a user needs to manually extract keywords from an article, and then to switch between different search services while browsing the web pages.

Seamless integration of intelligent content provision into web browsing at user interface level remains an open research question. No universally accepted optimal design exists. Herein we propose Kvasir[1], a system built atop latent semantic analysis (LSA). We show how Kvasir can be integrated with existing state-of-art technologies (e.g., Apache Spark, machine learning, etc.). Kvasir automatically looks for the similar articles when a user is browsing a web page and injects the search results in an easily accessible panel within the browser view for seamless integration. By presenting the architecture, we show how we tackle the scalability challenges confronting Kvasir in building and indexing high dimensional language database.

2. BACKGROUND AND RELATED WORK

There are several parallel efforts in integrating intelligent content provision and recommendation in web browsing. They differentiate between each other by the main technique used to achieve the goal. The initial effort relies on the semantic web stack proposed in [2], which requires adding explicit ontology information to all web pages so that ontology-based applications (e.g., Piggy bank [9]) can utilize ontology reasoning to interconnect content semantically. Though semantic web has a well-defined architecture, it suffers from the fact that most web pages are unstructured or semi-structured HTML files, and content providers lack of motivation to adopt this technology. Collaborative Filtering (CF) [4, 11], which was first coined in Tapestry [7], is a thriving research area and also the second alternative solution. Recommenders built atop CF exploit the similarities in users' rankings to predict one user's preference on a specific content. CF attracts more research interest these years due to the popularity of online shopping (e.g., Amazon, eBay, Taobao, etc.) and video services (e.g., YouTube, Vimeo, Dailymotion, etc.). However, recommender systems need user behavior rather than content itself as explicit input to bootstrap the service, and is usually constrained within a single domain. Cross-domain recommenders [5, 12] have made progress lately, but the complexity and scalability

WWW'15 Companion, May 18–22, 2015, Florence, Italy.
ACM ACM 978-1-4503-3473-0/15/05.
http://dx.doi.org/10.1145/2740908.2742825.

[1] Kvasir is the acronym for *Knowledge ViA Semantic Information Retrieval*, it is also the name of a Scandinavian god in Norse mythology who travels around the world to teach and spread knowledge and is considered extremely wise.

need further investigation. Search engines can be considered as the third alternative though users need explicitly extract keywords from a page then launch another search. The ranking of the search results is based on link analysis on the underlying graph structure of interconnected pages (e.g., PageRank [14] and HITS [10]). Kvasir utilizes information retrieval (IR) [6,13] which belongs to the content-based filtering and emphasizes the semantics contained in unstructured web text. A text corpus is transformed to a suitable representation depending on the specific math models (e.g., set-theoretic, algebraic, or probabilistic), based on which a numeric score is calculated for ranking. Context awareness is the most significant advantage in IR, which has been integrated into Hummingbird, Google's new search algorithm.

3. KVASIR ARCHITECTURE

Kvasir implements an LSA-based index and search service, and its architecture can be divided into *frontend* and *backend*. Figure 1 illustrates the workflow and innards of the system. The frontend is implemented as an extension in Chrome browser. The browser extension only sends the page URL back to the KServer whenever a new tab/window is created. The KServer running at the backend retrieves the content of the given URL then responds with the most relevant documents in a database. The results are returned in JSON strings. The browser extension presents the results on the page being browsed. From the user's perspective, a user only interacts with the frontend by checking the list of recommended articles that may interest him. The backend uses one simple *REST API* as below to connect the frontend, which gives flexibility to all possible frontend implementations and makes it easy to mash-up new services atop Kvasir. Line 1 gives an example request while line 2-6 give an example response containing the metainfo of a file.

```
1  POST https://api.kvasir/query?info=url
2  {"results": [
3    {"title": document title,
4     "similarity": similarity metric,
5     "page_url": link to the document,
6     "timestamp": document create date} ]}
```

The backend system implements indexing and searching functionality which consist of five components: Crawler, Cleaner, DLSA, PANNS and KServer. Three components (i.e., Cleaner, DLSA and PANNS) are wrapped into one Spark library.

Crawler collects raw documents from the Web then compiles into two data sets. One is the English Wikipedia dump containing about 4 million articles, and another is compiled from over 300 news feeds of the high-quality content providers containing about 330 000 articles. Multiple instances of the Crawler run in parallel on different machines.

Cleaner cleans the unstructured text and converts the corpus into term frequency-inverse document frequency (TF-IDF) model. We clean the text by removing HTML tags and stopwords, deaccenting, tokenization, etc. The dictionary refers to the vocabulary of a language model, its quality directly impacts the model performance. To build the dictionary, we exclude both extremely rare and extremely common terms, and keep 10^5 most popular ones as *features*.

DLSA builds up an LSA-based model from the previously constructed TF-IDF model. The operations involve large matrix multiplications and time-consuming SVD. Since MLlib is unable to perform SVD on a data set of 10^5 features with limited RAM, we implemented our own stochastic SVD on Spark using rank-revealing technique in Section 4.1.

PANNS indexes an LSA model to enable fast k-NN search in high dimensional spaces. Though dimensionality has been reduced from 10^5 (TF-IDF) to 10^3 (LSA), k-NN search in a 10^3-dimension space is still a great challenge. Naive linear search using one CPU takes over 6 seconds to finish in a database of 4 million entries, which is unacceptably long for any realistic services. PANNS implements a parallel RP-tree algorithm which makes a reasonable tradeoff between accuracy and efficiency. Section 4.2 presents PANNS in details.

KServer runs within a web server, processes the users requests and replies with a list of similar documents. KServer uses the index built by PANNS to perform fast search in the database. The ranking of the search results is based on the cosine similarity metric. We deployed multiple KServer instances on different machines and implemented a simple round-robin mechanism to balance the request loads.

4. PROPOSED ALGORITHMS

The source code and the demo videos can be found in [1].

4.1 Distributed Stochastic SVD

The vector space model belongs to algebraic language models, where each document is represented with a row vector. Each element in the vector represents the weight of a term in the dictionary calculated in a specific way. E.g., it can be simply calculated as the frequency of a term in a document, or slightly more complicated TF-IDF. The length of the vector is determined by the size of the dictionary (i.e., number of features). A text corpus containing m documents and a dictionary of n terms will be converted to a $A = m \times n$ row-based matrix. LSA utilizes SVD to reduce n by only keeping a small number of linear combinations of the original features. To perform SVD, we need calculate the covariance matrix $C = A^T \times A$, which is a $n \times n$ matrix and is usually much smaller than A.

Though we can parallelize the calculation of C by dividing A into k smaller chunks of size $[\frac{m}{k}] \times n$, then aggregate the partial results as $C = A^T \times A = \sum_{i=1}^{k} A_i^T \times A_i$. However, C might still be too big to fit into memory. Our solution is using rank-revealing QR [8] to approximate A with a thinner matrix B, which essentially leads to a stochastic SVD and is able to process much larger matrix than native MLlib.

4.2 Parallel Randomized Partition Tree

Finding the most relevant documents in an LSA model is equivalent to finding the nearest neighbors for a given point. The distance is usually measured with the cosine similarity of two vectors. However, neither naive linear search nor conventional k-d tree is capable of searching efficiently in such high dimensional spaces even though the dimensionality has already been reduced from 10^5 to 10^3 by LSA. The gain in speed is usually achieved by sacrificing some accuracy.

Technically, we use RP-tree algorithm to cluster the points by partitioning the space into smaller subspaces recursively. Since RP-tree requires generating and storing huge amount of random vectors for index building and searching, it poses a significant challenge on both reducing the index size and parallelizing RP-tree algorithm. To address this challenge,

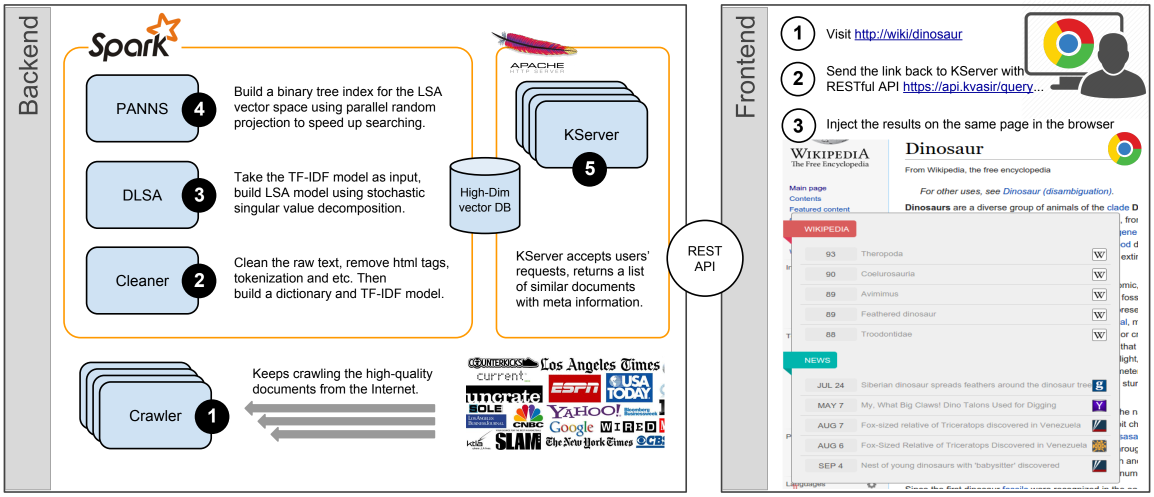
Backend
Spark
PANNS
4
Build a binary tree index for the LSA vector space using parallel random projection to speed up searching.
DLSA
3
Take the TF-IDF model as input, build LSA model using stochastic singular value decomposition.
Cleaner
2
Clean the raw text, remove html tags, tokenization and etc. Then build a dictionary and TF-IDF model.
High-Dim vector DB
APACHE HTTP SERVER
KServer
5
KServer accepts users' requests, returns a list of similar documents with meta information.
REST API
Crawler
1
Keeps crawling the high-quality documents from the Internet.
Frontend
1 Visit http://wiki/dinosaur
2 Send the link back to KServer with RESTful API https://api.kvasir/query...
3 Inject the results on the same page in the browser
WIKIPEDIA The Free Encyclopedia
Dinosaur
From Wikipedia, the free encyclopedia
For other uses, see Dinosaur (disambiguation).
Dinosaurs are a diverse group of animals of the clade D
Main page
Contents
Featured content
WIKIPEDIA
93 Theropoda
90 Coelurosauria
89 Avimimus
89 Feathered dinosaur
88 Troodontidae
NEWS
JUL 24 Siberian dinosaur spreads feathers around the dinosaur tree
MAY 7 My, What Big Claws! Dino Talons Used for Digging
AUG 7 Fox-sized relative of Triceratops discovered in Venezuela
AUG 6 Fox-Sized Relative of Triceratops Discovered in Venezuela
SEP 4 Nest of young dinosaurs with 'babysitter' discovered
Languages

Figure 1: Kvasir architecture – there are five major components in the backend system, and they are numbered based on their order in the workflow. The frontend is implemented in a Chrome browser, and connects the backend with a RESTful API.

we propose to use a pseudo random seed in building and storing search index. Instead of maintaining a pool of pre-generated random vectors, we just need a random seed for each RP-tree. The computation node can build all the random vectors on the fly from the given random seed.

4.3 Caching to Scale Up

The index will eventually become too big to fit into a memory. One engineering solution is using MMAP provided in operating systems which maps a file from hard-disk to memory space without actually loading it into a physical memory. The loading only happens whenever there is a cache miss. Search performance may degrade if the access pattern is truly random on a huge index. In practice, this is highly unlikely since the pattern of user requests follows a clear Zipf-like distribution, which indicates only a small part of the index trees is frequently accessed at any given time.

5. PRELIMINARY EVALUATION

Because scalability is the main challenge in Kvasir, the preliminary evaluation revolves around: (i) how fast we can build a database from scratch using the library we developed for Apache Spark; (ii) how fast the search function in Kvasir can serve users' requests. The evaluation is performed on a testbed of 10 Dell PowerEdge M610 nodes. Each node has 2 quad-core CPUs, 32GB memory, and is connected to a 10-Gbit network. All the nodes run Ubuntu SMP with a 3.2.0 kernel with ATLAS (Automatically Tuned Linear Algebra System) installed to support fast linear algebra operations. We only report the results on using Wikipedia data set since News data set leads to consistently better performance.

5.1 Database Building Time

To evaluate the efficiency of our Spark library in the backend, we first perform a sequential execution with one CPU to obtain a benchmark. Using one CPU takes over 35 hours to process the Wikipedia data set. Using 5 CPUs to parallelize the computation, it takes about 9 hours which is almost three times faster. Table 1 shows that the total building speed is improved sublinearly. Because the overhead from I/O and network operations eventually replace CPU overhead and become the main bottleneck in the system.

# of CPUs	Cleaner	DLSA	PANNS	Total
1	1.32	20.23	13.99	35.54
5	0.29	6.14	2.86	9.29
10	0.19	4.22	1.44	5.85
15	0.17	3.14	0.98	4.29
20	0.16	2.61	0.77	3.54

Table 1: The time needed (in hours) for building an LSA-based database. The time is decomposed to component-wise level. Search index uses 128 RP-trees with cluster size of 20.

By zooming in the component-wise overhead, DSLA contributes most of the computation time while Cleaner contributes the least. Cleaner's tasks are easy to parallelize due to its straightforward structure, but there are only marginal improvements after 10 CPUs since most of the time is spent in I/O operations and job scheduling. For DLSA, the parallelism is achieved by dividing a matrix into chunks then distributing the computations on multiple nodes. The partial results need to be exchanged among the nodes for aggregation, and therefore the penalty of the increased network traffic will eventually overrun the benefit of parallelism. Further investigation reveals that the percent of time in transmitting data increases from 10.5% to 37.2% (from 5 CPUs to 20 CPUs). On the other hand, indexing scales very well by using more computation nodes because PANNS does not require exchanging too much data among the nodes.

5.2 Accuracy and Scalability of Searching

Service time represents the amount of time to process a request, which is arguably the most important metric to measure the service scalability. We test the service time of KServer by using one web server in the aforementioned testbed. We model the content popularity with a Zipf distribution, whose probability mass function is $f(x) = \frac{1}{x^{\alpha} \sum_{i=1}^{n} i^{-\alpha}}$, where x is the content index, n is the number of content, and α controls the skewness of the distribution. Smaller values of α lead to more uniform distributions while large α values assign more mass to elements with small i. It has been empirically demonstrated that in real-world data following a power-law, the α values typically range between 0.9 and

(c, t)		(20,16)	(20,32)	(20,64)	(20,128)	(20,256)	(80,16)	(80,32)	(80,64)	(80,128)	(80,256)
Index (MB)		361	721	1445	2891	5782	258	519	1039	2078	4155
Precision (%)		68.5	75.2	84.7	89.4	94.9	71.3	83.6	91.2	95.6	99.2
$\alpha_1 = 1.0$	ms	2.2	3.7	5.1	5.9	6.8	4.6	7.9	11.2	13.7	16.1
$\alpha_2 = 0.9$	ms	3.4	4.3	6.0	6.8	7.6	7.2	9.5	14.9	15.3	17.1
$\alpha_3 = 0.8$	ms	4.3	4.9	6.7	7.9	8.4	9.1	11.7	15.2	17.4	17.9
$\alpha_4 = 0.7$	ms	5.5	6.3	7.4	8.5	9.3	11.6	13.4	16.1	17.7	18.5
$\alpha_5 = 0.6$	ms	6.1	6.7	7.9	8.8	9.8	13.9	16.0	18.5	19.8	21.1
$\alpha_6 = 0.5$	ms	6.7	7.3	8.2	9.0	10.3	16.6	17.8	19.9	20.4	23.1

Table 2: Scalability test on KServer with different index configurations and request patterns. (c, t) in the first row, c represents the maximum cluster size, and t represents the number of RP-trees. Zipf-(α, n) is used to model the content popularity.

1.0. We plug in different α to generate the request stream. The next request is sent out as soon as the results of the previous one is successfully received. Round trip time (RTT) depends on network conditions and is irrelevant to the efficiency of the backend, hence is excluded from total service time in Table 2. We also experiment with various index configurations to understand how index impacts the server performance. The index is configured with two parameters: the maximum cluster size c and the number of search trees t. Note c determines how many random vectors contained in a RP-tree, which further impacts the search precision. The first row in Table 2 lists all the configurations. For a realistic $\alpha = 0.9$ and index $(20, 256)$, the throughput can reach over 1052 requests per second on a node of 8 CPUs.

Table 2 shows that including more RP-trees improves the search accuracy but increases the index size. Since we only store a random seed for all random vectors, the growth of index is due to storing tree structures. The search overhead also grows sublinearly with more trees. However, since searching in different trees are independent thus can be easily parallelized, the performance can be further improved by using more CPUs. Given a specific index configuration, the service time increases as α decreases, which attests our arguments in Section 4.3. Namely, we can exploit the highly skewed popularity to scale up Kvasir. As mentioned, increasing cluster size is equivalent to reducing the number of random projections and vice versa. We increase the maximum cluster size from 20 to 80 and present the result in the right half of Table 2. Contrary to the intuition that the precision might deteriorate with less random projections, we notice that the precision is improved. The reason is two-fold: first, large cluster size reduces the probability of misclassification for those projections close to the split point. Second, since we perform linear search within a cluster, larger cluster enables us explore more neighbors which leads to higher probability to find actual nearest ones. Nonetheless, also due to the linear search in larger clusters, the gain in the accuracy is at the price of inflated searching time.

6. DISCUSSIONS & FUTURE WORK

We can improve Kvasir in many ways. Firstly, we need not rebuild everything from scratch whenever new content arrives. LSA space can be incrementally updated [3], then new points can be added to the corresponding clusters in a RP-tree. Secondly, finer-grained re-ranking can be implemented by taking both a user's long-term and short-term interest into account. Such function can be achieved by extending one-class SVM. Thirdly, the frontend is not constrained within a browser but can be implemented in various ways on different platforms. Content providers can also integrate Kvasir service on their website to enhance user experience by automatically providing similar articles on the same page. Fourthly, Kvasir does not yet support full-fledged security and privacy. For security, DDoS attacks are difficult to defend against in general. For privacy, a user may not want Kvasir track their behavior for personalized results, thus a fine-grained privacy policy is needed in the future.

7. CONCLUSION

In this paper, we presented Kvasir which provides seamless integration of LSA-based content provision into web browsing. We proposed a parallel RP-tree algorithm and implemented stochastic SVD on Spark to tackle the scalability challenges. The proposed solutions were evaluated on the testbed and scaled well on multiple CPUs. The key components of Kvasir are implemented as an Apache Spark library, and the source code is publicly accessible on Github.

8. REFERENCES

[1] Kvasir project, http://cs.helsinki.fi/u/lxwang/kvasir.

[2] T. Berners-Lee, et al. The semantic web. In *Scientific american*, 284(5):28–37, 2001.

[3] M. Brand. Fast low-rank modifications of the thin singular value decomposition. In *Linear Algebra and its Applications*, 415(1):20 – 30, 2006.

[4] J. S. Breese, et al. Empirical analysis of predictive algorithms for collaborative filtering. In *UAI'98*, 1998.

[5] P. Cremonesi, et al. Cross-domain recommender systems. In *IEEE ICDMW*, 496–503, 2011.

[6] W. B. Frakes, et al. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[7] D. Goldberg, et al. Using collaborative filtering to weave an information tapestry. *ACM Commun.*, 1992.

[8] N. Halko, et al. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. In *SIAM Rev.*, 2011.

[9] D. Huynh, et al. Piggy bank: Experience the semantic web inside your web browser. In *ISWC 2005*, 2005.

[10] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 604–632, 1999.

[11] Y. Koren and R. Bell. Advances in collaborative filtering. In *Recommender Systems Handbook*, 2011.

[12] B. Li, et al. Can movies and books collaborate?: Cross-domain collaborative filtering for sparsity reduction. In *IJCAI'09*, 2009

[13] C. D. Manning, et al. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[14] L. Page, et al. The pagerank citation ranking: Bringing order to the web. Tech-report, 1999.