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

\(^1\)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 consid-
ered 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 [16]). Kvasir utilizes informa-
tion retrieval (IR) [6, 13] which belongs to the content-based
filtering and emphasizes the semantics contained in unstruc-
tured 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 inte-
grated into Hummingbird, Google’s new search algorithm.

3. KVASIR ARCHITECTURE

Kvasir implements an LSA-based index and search ser-
vice, 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 back-
end uses one simple REST API as below to connect the
frontend, which gives flexibility to all possible frontend im-
plementations 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.

```json
POST https://api.kvasir/query?info=url
"results": [  
"title": document title,  
"similarity": similarity metric,  
"page_url": link to the document,  
"timestamp": document create date} ]
```

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

Crawler collects raw documents from the Web then com-
piles 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 in-
stances 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 dictio-
nary refers to the vocabulary of a language model, its qual-
ity directly impacts the model performance. To build the
dictionary, we exclude both extremely rare and extremely
common terms, and keep $10^2$ 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 ML-
lib is unable to perform SVD on a data set of $10^7$ 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 $10^9$ (TF-IDF) to $10^6$ (LSA), $k$-NN search in a
$10^4$-dimension space is still a great challenge. Naïve 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 accu-
racacy and efficiency. Section 4.2 presents PANNS in details.

KServer runs within a web server, processes the users re-
quests 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 vec-
tor. 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 $n$
documents and a dictionary of $n$ terms will be converted to a
matrix $A = m \times n$ 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 $\sqrt{n} \times n$, 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 con-
ventional $k$-d tree is capable of searching efficiently in such
high dimensional spaces even though the dimensionality has
already been reduced from $10^9$ to $10^6$ 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,
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.

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.

#### 5.1 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, the index trees is frequently accessed at any given time.

#### 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 $f(x) = \frac{1}{x^\alpha}$, where $x$ is the content index, $n$ is the number of content, and $\alpha$ controls the skewness of the distribution. Smaller values of $\alpha$ lead to more uniform distributions while large $\alpha$ values assign more mass to elements with small $i$. It has been empirically demonstrated that in real-world data following a power-law, the $\alpha$ values typically range between 0.9 and 1.0.
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-(\(\alpha,n\)) is used to model the content popularity.

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

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.

8. REFERENCES