LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	LEADS: Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D1.3
Title of Deliverable:	Enhanced content discovery and processing
Contractual Date of Delivery:	M24 – 9/30/2014
Actual Date of Delivery:	9/30/2014

Abstract

This deliverable presents the different techniques we use for pre-processing the user published data for (1) detecting content spam in the web pages users publish, (2) detecting comment/tag spam users submitted with the web pages, and (3) leveraging user publishing behaviour to improve the performance of the crawling-based data collection of the LEADS platform by collecting new versions of web pages only when their content is deemed to change.

List of Contributors

Name	Organization	E-mail
Xiao Bai	BM-Y!	xbai@yahoo-inc.com
Diego Marron	BM-Y!	diegom@yahoo-inc.com
Tim Potter	BM-Y!	tep@yahoo-inc.com
Ata Turk	BM-Y!	ata@yahoo-inc.com
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Nikolaos Giatrakos	TSI	ngiatrakos@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Nikolaos Pavlakis	TSI	npavlakis@softnet.tuc.gr imper-
Ioakim Perros	TSI	ros@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	christof.fetzer@tu-dresden.de
André Martin	TUD	andre.martin@tu-dresden.de
Do Le Quoc	TUD	do@se.inf.tu-dresden.de
Frezewd Lemma Tena	TUD	frezewd_lemma.tena@mailbox.tu-dresden.de
Frank Busse	TUD	frank.busse@tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Xiao Bai	xbai@yahoo-inc.com	11.09.2014
Approved by GA Member 1	Etienne Riviere	etienne.riviere@unine.ch	17.09.2014
Approved by GA Member 2	Christof Fetzer	christof.fetzer@gmail.com	17.09.2014

Contents

List of Figures

List of Tables

GLOSSARY

EU	European Union
FP7	Seventh Framework Programme
DaaS	Data-as-a-Service
GBDT	Gradient Boosting Decision Tree
VW	Vowpal Wabbit
AUC	Area Under the Curve
RMSE	Root Mean Square Error

Executive summary

LEADS is a decentralized Data-as-a-Service (DaaS) framework that runs on an elastic collection of micro-clouds to gather, store and process data. Data collection in LEADS is performed in a fully distributed way through two mechanisms: crawling-based data collection, which is a geographically distributed version of the traditional web crawling, and user-aided data collection, which relies on LEADS users to actively report their data to LEADS.

The user-aided data collection aims to collect user generated data and data that are difficult to be collected by crawling such as dynamic web pages. It utilizes a user publishing interface that enables real-time publishing of user generated content. The focus of this deliverable is the application oriented data pre-processing, which ensures that only high quality data are incorporated to LEADS. Up to M12, we had mainly focused on the design and implementation of the user-aided publishing interface. For the M13 - M24 period covered by this document, we report on our efforts for pre-processing of user published data as well as user behaviour based Web crawling and particularly page refreshing.

In a nutshell, the user-aided data collection relies on users to provide data to the LEADS platform. Users can send the web pages they create or they visit, along with their own tags or comments to the LEADS platform. The user-publishing interface is designed as a plug-in on web browsers, and is currently implemented as an extension of the Chrome web browser. User-supplied data and the associated metadata sent to LEADS are pre-processed to detect possible malicious inputs. In this deliverable we explain the machine-learning-based pre-processing modules developed for identifying spam Web pages and spam metadata.

Furthermore, we also evaluate, using large-scale datasets collected by both crawling and by means comparable to user-aided data publishing, the potential of using user information that can be obtained via the user-publishing interface for providing better page refresh policies for crawling-based data collection. Our analysis results are very promising. In essence, we show that user page visit counts that can be collected by the user-publishing plugin are good indicators for predicting whether a page should be refreshed or not.

1. Introduction

The user-publishing interface of LEADS is a very strong data collection mechanism utilizing a push-based model and crowdsourcing for data collection, rather than the classical pull-based crawling. It not only enables seamless, faster and more cost-effective data collection, but also opens the path of social and semantic association of content via user supplied comments/tags.

Recall that the user-publishing interface can collect user generated data and data that are difficult to be collected by the crawling-based data collection mechanism. It enables real-time publishing/collection of user-generated content by relying on users to provide data to LEADS. Users can send the web pages they create or visit, along with their own tags or comments to the LEADS platform. A key component for enabling this functionality is what we call the user-aided publishing interface, which allows users to either explicitly or implicitly send their data (as well as the associated metadata) to LEADS. The user-publishing interface is designed as a plug-in for web browsers: currently it is implemented as a Chrome extension and users are able to use it easily on their devices.

Since the user-aided data collection acts as an alternative to crawling in certain senses, mechanisms pipelined along with crawling needs to be employed to the user-aided data collection as well. In crawling, to ensure the quality of the collected datasets, various filtering mechanisms such as spam and adult content filtering are deployed. Especially, detection of spam pages is utmost importance to ensure that the collected dataset has sufficient quality to be used in the applications that make use of it. Similarly, it is important for LEADS to make sure that the data collected via the user-publishing interface is not spam, or at least be aware of the level of "spamminess "of the collected data.

Note that, unlike crawled data, data collected through the user-publishing interface contains user comments in addition to the page contents in html format. An analysis of these comments along with the supplied content/html for spam is also important to ensure the quality the data processing services the LEADS platform provides.

To ensure high quality content and service in LEADS, user-supplied data and the associated metadata sent to LEADS are pre-processed to detect possible malicious inputs. In this deliverable we explain the machine-learning-based pre-processing modules developed for identifying spam Web pages and spam metadata.

The user-publishing interface is implemented as a browser extension and it can have further benefits. In this deliverable we focus on one such possible benefit, namely determining the refreshing order of the collected data. This deliverable also evaluates, using large-scale datasets collected by both crawling and by means comparable to user-publishing interface, the potential of using user information that can be obtained via the user-publishing interface for providing better page refresh policies for crawling-based data collection. Our analysis results are very promising. In essence, we show that user page visit counts that can be collected by the user-publishing plugin are good indicators on whether the content of a page has changed and whether the page should be refreshed or not.

The rest of the deliverable is organized as follows. In Section 2, we present the spam detection mechanisms employed by our user-aided data collection plugin. Section 3 presents an analysis of the potential benefits obtainable by analysing user-browsing behaviours through the user-aided data collection plugin and using it for better data collection refreshing. We conclude this deliverable in Section 4. Appendix A briefly reports the progress of the interaction between Ensemble, the LEADS geo-distributed storage platform implemented in WP2, and Nutch the web crawler developed in WP1 (see D1.2).

2. Spam detection in user-aided data collection

This section describes the techniques to pre-process the data collected through the publishing interface to detect malicious inputs. We perform two kinds of pre-processing on the data collected through the user-publishing interface: Content spam detection and tag/comment spam detection. We describe the processes involved in these pre-processing steps and experimental evaluation showing their effectiveness in Sections 3.1 and 3.2 respectively. We also describe the integration of these techniques to the publishing interface in Section 3.3.

2.1 Content spam detection in user-aided data collection

In this section we describe the approach we follow for detecting the “spamminess” of user-supplied content/html. Our content/html spam detection system utilizes a pre-trained machine-learning model. We build a regression mechanism utilizing a ground truth dataset obtained by combining the Waterloo Spam Rankings [CSC10] with the ClueWeb09 dataset [CLU09].

The ClueWeb09 dataset contains around 1 billion Web pages in html format. For training, we took a random 2M sample from the 1 billion pages and used the Fusion spam scores [WSR09] provided by the University of Waterloo for these pages as spam score labels. These spam scores range between 0 and 99 and anything above 70 is considered as spam. Using the ground-truth dataset we trained word n-gram models to predict the ``content spam score'' of documents. An n-gram is a contiguous sequence of n items from a given sequence of text. We use the main text of webpages for training. In order to extract the main text of html pages in the ClueWeb dataset, we used the Goose Library [GO14], which eliminates the html tags, images and other scripting codes and outputs a clean text-only representation of the html document.

The trained machine-learning model predicts the “content spam score” for any given html or content. Whenever a user submits a URL with scrapped html or parsed content to our user-aided data collection system, user submitted content is passed through our machine learning model, and a ``content spam score'' is assigned to the user submitted content.

For learning, we use the Vowpal Wabbit (VW) machine learning toolkit [ACD+11, HBB10], which is a fast, out-of-core learning tool. VW toolkit employs the hashing trick (feature hashing), hence it is possible to restrict the size of the model, which also helps in fast scoring of tags/comments. Specifically, hashing trick is a fast and space-efficient way of vectorizing features, i.e. turning arbitrary features into indices in a vector or matrix. It works by applying a hash function to the features and using their hash values as indices directly, rather than looking the indices up in an associative array. We use VW in daemon mode during the actual integration. This enables us to load the model into memory once and execute each prediction using the memory-resident model.

Note that we are performing a regression task by assigning spam scores in the range [0, 99] to pages. However, to evaluate and compare the effectiveness of the n-gram models, we performed offline tests, performing first regression to predict the scores and then converting them to binary predictions using the score 70 as threshold, i.e. any page with a spam score over 70 is assumed as a spam page and 70 and below is considered non-spam. In Table 1 we report the accuracy of the n-gram models for various values of “n”. The values are obtained using 10-fold cross validation on the sampled dataset. We should note that the ratio of examples labelled as “spam” to those that are labelled as “not-spam” in our sampled dataset is almost one; hence there is no majority group in the dataset. The n-gram models have high accuracy but rather low Area Under the Curve (AUC). AUC is a metric for binary classification. Different from accuracy that simply measures the fraction of correct labels,

AUC considers all possible thresholds to label the samples. Various thresholds result in different true positive/false positive rates. Higher AUC value indicates better classification. A score for a perfect classifier would be 1. The reason that the n-gram models have low AUC is the fact that we are missing to label some pages as spam (i.e., false negatives), especially pages that have labels close to the threshold score. Since the actual prediction we perform is not a binary prediction, we also present the Root Mean Square Error (RMSE) values, normalized to the 0-100 range. As shown in Table 1, 4-gram model performs the best RMSE but has a rather large model size and slightly lower accuracy, whereas bigram model performs close to 4-gram in terms of RMSE, but has better prediction accuracy and has a more reasonable model size. Hence, in our deployment, we used the bigram model as our final prediction model.

Table 1: Accuracy of different n-gram models in content spam detection.

Model	Accuracy	Area Under the Curve (AUC)	Root-Mean-Square Error (RMSE)	Model Size
Unigram (n=1)	84.0%	0.56	100	27MB
Bigram (n=2)	84.2%	0.59	95	310MB
Trigram (n=3)	84.2%	0.61	93	912MB
4gram (n=4)	83.9%	0.62	92	1400MB

2.2 Spam detection for user published tags

In this section we describe the approach we follow for detecting the "spamminess" of comments/tags entered into our user-aided data collection system. Our user-supplied metadata spam detection system is comprised of a pre-trained machine learning system. Using a regression mechanism and a ground truth dataset obtained from the Yahoo Answers website, comprising a sample of the texts of answers provided to questions in the site between 01/04/2014 and 30/04/2014 and their assigned spam scores (by a Yahoo proprietary system), we assign a "comment spam score" to each word sequence observed in our dataset. Whenever a user submits a page with comments/tags to our browser-based data collection system, the user submitted comment sentences or groups of tags are evaluated as a sequence of words. We then use the machine learning model trained in the pre-processing step to assign a "spam score" to the user submitted comment sentences or groups of tags. For learning, again we use the VW machine learning toolkit [ACD+11, HBB10].

The dataset we use has around 5M comments, with assigned spam scores ranging between 0 and 256. These spam scores are assigned via a constantly updated system which brings the answers that it deems as spam to the attention of Yahoo editors, who in turn decide to remove or keep the answers. Unfortunately we cannot directly use the spam detection system of Yahoo due to its dependence on an extensive feature set (user features, historical features, etc.) not available in our application.

We conducted some offline experiments using our ground truth dataset to verify that the proposed models have good spam detection accuracy. Since the majority of the comments have low spam scores, majority voting is a very good baseline to compare against. In Table 2, we can see that the prediction accuracy and AUC scores along with the model sizes for the various n-gram models we tested. Note that due to the skewed nature of the dataset, it is possible to have high accuracy with low recall, hence the need for also reporting the AUC scores. In our deployment, we used the 4-gram

model as our final prediction model for detecting spams in user generated tags as it ensures the best accuracy and AUC with reasonable model size.

Table 2: Accuracy of different n-gram models and majority voting in tag spam prediction.

Model	Accuracy	AUC	Model Size
Majority voting	86.0%	~0.500	-
Unigram (n=1)	90.6%	0.808	3MB
Bigram (n=2)	91.9%	0.804	35MB
Trigram (n=3)	92.5%	0.803	115MB
4gram (n=4)	92.9%	0.804	210MB

2.3 Integration of spam detection modules with user publishing back-end

Both content and comment spam detection modules use VW, a very fast machine-learning tool, for spam score assignment and hence they perform very fast predictions. Still, since the user-publishing plugin is a user-facing application, it is very important to ensure that user satisfaction is not reduced by unnecessary waiting during the submission. In Figure 1, we present a simple architecture diagram for the user-aided data collection backend.

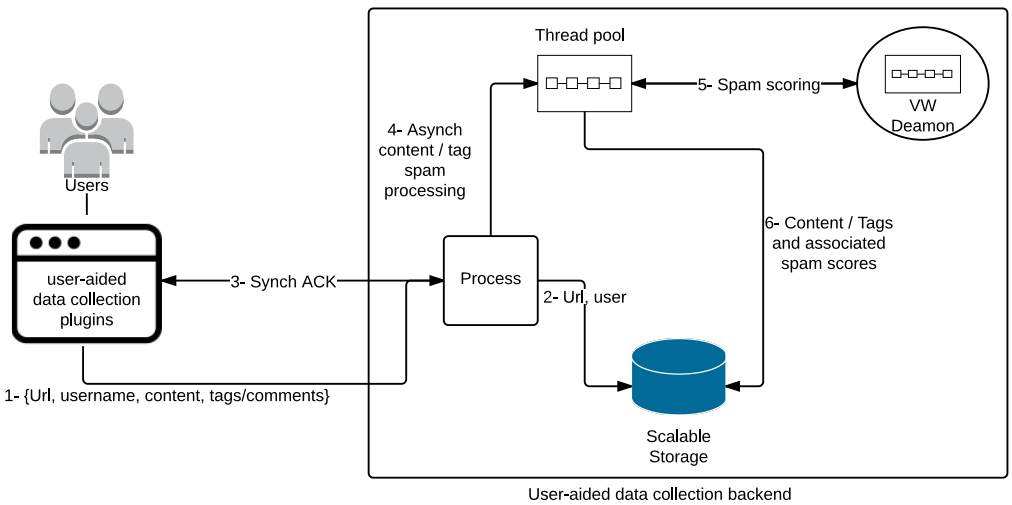

Figure 1: User-aided data collection backend architecture diagram.

To ensure fast response to users, whenever a user submits some data to the LEADS system, the submitted URL (along with the username if supplied) is stored on the scalable storage system and upon completion of this operation, a HTTP response (e.g. 200, 400, 408 etc.) to the user is returned immediately. Note that in the later step, even if the content and/or user-supplied metadata is found to be spam, we currently still store the user and URL information for logging purposes. The spam data can

be excluded from the data processing tasks of LEADS at query time according to their labels that indicate whether they are spam or not.

To asynchronously pre-process the content and user supplied metadata, a separate thread that runs in the background is opened. This thread submits the content and user supplied metadata to the Vowpal Wabbit daemons running in the background. We should note that we have two separate VW daemons, one for content spam detection (listening for requests on port 26542) and one for comment/tag spam detection (listening for requests on port 26543), since the models trained for these two separate tasks are different. Each VW daemon also has their own input thread pool (with 10 threads) so they can respond to multiple queries at the same time ensuring further scalability. By decoupling the spam detection process, a seamless process flow experience is ensured for the user by immediately responding to user requests.

Currently, after predicting the spam scores for the content and user comments of a URL, we update the scalable storage and update the respective entry associated with that URL with these scores. In the final application, it is possible to not store the contents and/or comments scored above the thresholds determined via the two applications (70 for content spam detection and 64 for tag/comment spam detection).

3. User behaviour based web crawling

In LEADS, public data that is available in the Internet is collected in a fully distributed way, through two different mechanisms: crawling-based data collection and user-aided data collection, as presented in former deliverables [D1.2]. The crawling-based data collection mechanism relies on multiple crawlers located in geographically distributed micro-clouds to perform crawling in a collaborative way. Like the traditional web crawlers, these crawlers also have two responsibilities: down- loading new pages, and keeping previously downloaded pages fresh. We present in this section how we improve the freshness of the previously downloaded pages by leveraging the information obtained by the user-aided data collection mechanism.

3.1 Crawling-based data collection and page refreshing

The crawling-based data collection mechanism relies on web crawlers located in geographically distributed micro-clouds to fetch content from the Web. Each crawler is in charge of crawling a fraction of the web pages from the Web by following the link structure of the pages that have been fetched. The crawling-based data collection mechanism of LEADS differs from the traditional Web crawling by performing the crawling tasks in geographically distributed micro-clouds rather than in a single cloud, which aims at increasing the efficiency of data collection and the scalability of the system. Once a page is crawled, it is assigned to the cloud that ensures the best data locality for storage. The crawlers also need to communicate with each other to determine the set of web pages they should crawl to ensure the overall performance. Once a page is assigned to a crawler in a micro-cloud, that micro-cloud is in charge of refreshing its content if necessary.

As the Web evolves, the content of certain pages keep changing. Therefore, it is important to maintain the freshness of the data stored in the LEADS platform. Keeping the available data fresh requires re-fetching already fetched pages. A trivial solution consists of re-fetching all the pages assigned to each micro-cloud. However, this is in practice not feasible as it will incur huge cost and reduce the available resources for discovering and crawling new pages as the resources in each crawler are shared between the two complimentary tasks of crawling and refreshing.

In fact, there is no need to refresh all the pages that are previously crawled. For example, some pages like the homepage of a company may rarely change. Periodically re-fetching the content of such page does not bring any new data to the system but will incur additional cost. Therefore it is important to selectively refresh the web pages to achieve good balance between data freshness and system overhead.

A number of studies investigated the evolutionary properties of the Web [ATT+09, FMN+04, NCO04] and proposed page refreshing policies for crawlers. Some policies aimed to refresh the content that is more important, more often [CG00] or more likely to change [CG03, EMT01]. Others tried to maximize the impact on search result quality [FCV09, PO05]. There are also some policies that rely on the longevity of web pages to schedule re-crawling [CS08].

In LEADS, the system relies not only on the crawling-based mechanism to collect data, but also on the users (i.e., user-aided data collection). Therefore, the crawler in each micro-cloud can also have the opportunity to leverage the information provided by users to improve the scheduling of web page refreshing.

3.2 Motivating a different page refreshing policy

Current refreshing policies of web crawlers usually rely on the estimated importance of the pages but neglect the fact that a page may be very important to the search engine as it frequently appears in the search results of some queries but its content may never change. For instance, the Wikipedia page of a mathematic term may be of this kind. Frequently refreshing such pages imposes redundant workload on the crawlers and thus additional cost without actual benefits to the end users.

To validate our assumption that important pages may not change at the time of refreshing scheduled by the traditional web crawlers, we conduct a large-scale analysis using real datasets from Yahoo web search engine. Specifically, we use Yahoo Toolbar logs to mimic the data collected through user-aided data collection of LEADS. We randomly sampled 305K URLs that are frequently refreshed by the Yahoo web search engine and are visited by Yahoo Toolbar users. Among these URLs, the content of 151K URLs remain the same between their two consecutive crawls (i.e., refreshing them do not bring any benefit to the search engine), while the content of 154K URLs indeed change between two consecutive crawls.

Figure 2 shows the distribution of pages according to the time difference between their two consecutive crawls. We can see from this figure that the URLs whose content has changed are refreshed less frequently than the pages whose content has not changed. On average, the pages whose content has changed are refreshed every 12 days while the pages whose content has not changed are refreshed every 9 days.

Clearly, refreshing the pages whose content have not changed would incur additional cost for the search engine without bringing any benefit in terms of user experience. A well-designed page refreshing policy should only refresh a page if its content has changed since the last crawl. Therefore, our objective is to design a new page refreshing policy that can leverage the data collected through the user-aided data collection to improve the performance of the crawling-based data collection by avoiding redundant refreshing.

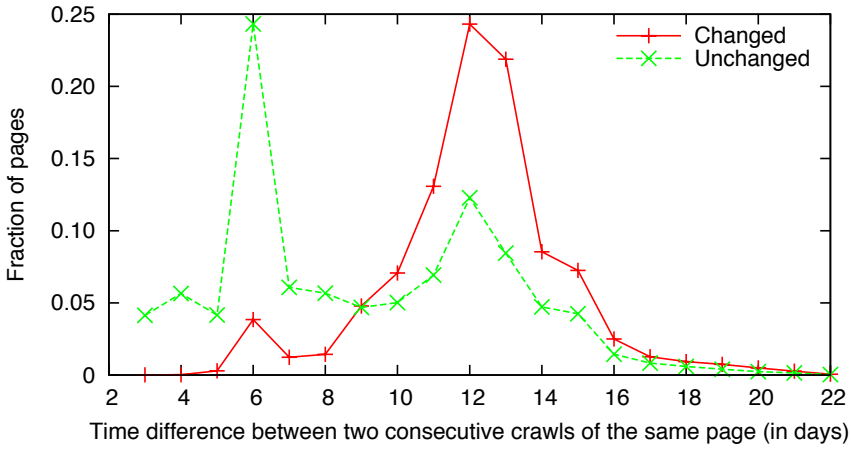

Figure 2: Distribution of page refreshing frequency.

3.3 User-behaviour enhanced web page refreshing

There are several ways for a web page to change. For instance, the change may correspond to addition or deletion of part of the text in a web page. In this case, the page needs to be refreshed to keep the data analysis users may perform in LEADS up-to-date. A change may also correspond to addition or deletion of links in a web page. In this case, refreshing the page cannot only improve the quality of data analysis, but also help the crawling-based data collection mechanism to discover and collect more pages. In this work, we do not distinguish the different changes. Instead, we aim at refreshing a page only when we predict that some change happens on it.

Traditional approaches usually rely on the frequency of changes (e.g., how often the content of the page is updated) [CG03B] or the relevance of changes (e.g., how the change would influence user experience in applications like web search) of each page [CG00, CG03] to predict whether a page should be refreshed. Later approaches demonstrate that content features are better indicators of content change than the frequency of changes. However, all the existing approaches focus on the features that are related to the page itself to make the prediction.

In fact, user behaviour in browsing the web pages may help making more accurate prediction about web page change. For instance, given an international conference, it is likely that the page for announcing the accepted papers receives comparable number of visits before the notification date of paper acceptance. When the notification day arrives, more and more users may come to the page to check the accepted papers. The page receives more visits as users expect to see changes on this page. In the contrary, not many users would visit the submission page on the notification day. Intuitively, a well-designed crawler should be able to refresh the page for announcing the accepted papers after observing an increasing number of user visits, and keep the submission page as it is when observing a stable number of user visits.

To validate this assumption, we compute the number of visits each page receives between their two consecutive crawls using the same dataset as described in Section 3.2. The number is averaged for all pages on each day. We can see from **Figure 3** that the pages whose content have changed and thus deserve a refresh are consistently more visited than the pages whose content remains the same between two consecutive crawls. This suggests that the number of visits a page receives can be a good indicator for predicting whether the page should be refreshed or not.

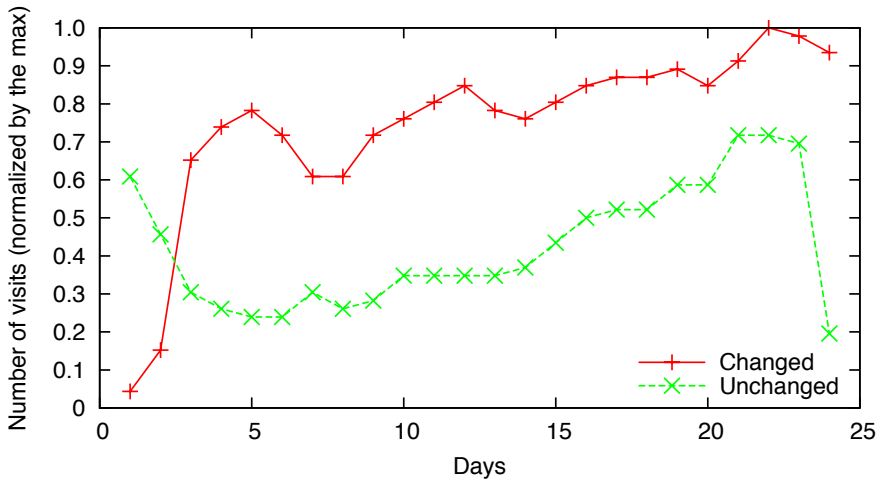

Figure 3: Relationship between web page change and user click behaviour.

Similarly, we show in Figure 4 the distribution of pages according to the average time difference between two visits from the same user. The y-axis value depicts the fraction of pages having the average time difference not larger than the x-axis value. We can see that on average, users wait more time before visiting again a page whose content has changed, while they wait less time before visiting again a page whose content has not changed. This seems to be contradictory to our intuition that users may visit a news portal or their social network pages frequently to discover new content. In fact, this is because in practice, a well-designed web crawler does not refresh frequently changed pages very often to avoid spending too many crawling resources on a few pages and to ensure the overall utility of the refreshing. Therefore, in this work, we focus on the pages whose content do not change very frequently. This may explain why the changed pages are not repeatedly visited by the same user within short time. Again, we can see the clear distinction between the two kinds of pages. This suggests that the time difference between the two visits of the same user on the same page can be a good indicator for predicting whether the page should be refreshed or not.

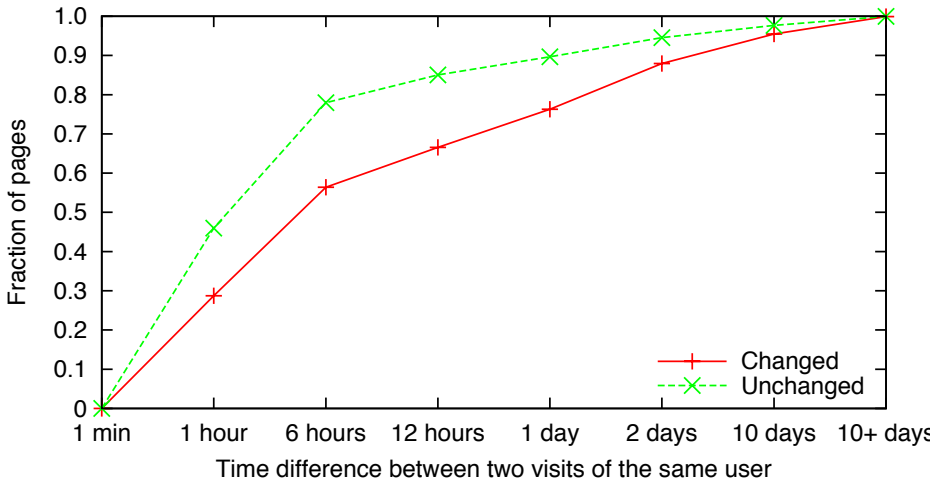

Figure 4: Relationship between web page change and user revisit behaviour.

Therefore, we propose to use a supervised machine learning approach to predict whether a web page should be refreshed or not based on past user behaviour on the page. The training instances can be collected over historical crawling.

We first identify a set of features based on user behaviour as follows.

- Visit count: Number of visits a web page received since the last refresh/crawl of the page. As we have seen from Figure 3, pages whose content has changed and the pages whose content have not changed receive clearly different number of visits.
- Visit rate: Average number of visits a web page received during unit time (e.g., an hour, a day, etc.). This is the visit count divided by the time difference between the current time and the last refresh/crawl of the page.
- User count: Number of users who have visited a web page since the last refresh/crawl of the page.
- User rate: Average number of users who have visited a web page during one time unit (e.g., an hour, a day, etc.). This is the user count divided by the time difference between the current time and the last refresh/crawl of the page.
- Revisit count: Number of times the same user clicks on a web page since the last refresh/crawl of the page. As we have seen from Figure 4, the pages whose content has changed and the pages whose content have not changed receive clearly different number of revisits.
- Revisit user count: Number of users who have clicked the web page more than once since the last refresh/crawl of the page.
- Revisit rate: Average number of times the same user clicks on a web page during one time unit (e.g., an hour, a day, etc.). This is the revisit count divided by the time difference between the current time and the last refresh/crawl of the page.
- Revisit delay: Time difference between consecutive visits of the same user on the same page. Here we consider a set of different variants, such as maximum time difference, minimum time difference, average time difference, etc., as features.

Replying on these user behaviour based features, we use the Gradient Boosting Decision Tree (GBDT) model [FF99, FM99] to predict whether a web page should be refreshed or not based on past user behaviour on the page. Given a list of URLs to refresh, the model will predict a confidence score, lying between 0.5 and 1, to indicate the likelihood of the content of this page to change. Whether and when to refresh a page can then be determined based on this confidence score. A high confidence score indicates a high likelihood of content change and thus a higher necessity of crawling a new version of a page.

To evaluate the performance of our approach, especially the quality of the user behaviour based features for predicting content change of web pages, we use the same dataset as described in Section 3.2. Specifically, we use the 305K sampled URLs that are frequently refreshed by the Yahoo Search engine. For each of these URLs, each time when it is refreshed by the Yahoo search engine, we use our model to predict whether it is content is likely to change and whether we really want to refresh it. If our model mispredicts the content of a page has not changed while it is actually changed, we have a false negative prediction. This will hurt the quality of our data collection as we may miss some content change that will later reflected in data processing tasks of LEADS. If our model predicts the content of a page has changed but it has not, we have a false positive prediction. This will waste the system resources to refresh the page and thus negatively influence the quality of our data collection as the fewer resources can be used to refresh the pages whose content actually has changed. Both false negative and false positive predictions should be avoided. We compare the shingles of each page after two crawls made by Yahoo search engine to determine whether its content has changed be-

tween the two crawls. Yet, we do not know the exact time when the changes happen on the page. We use this information as the ground truth to evaluate our approach.

We use the public machine learning software Weka[1] to evaluate our approach. As baselines, we compare our GBDT model against other state-of-the-art machine learning models: Naïve Bayes, Random Forest and J48 Decision Tree.

We can see from Table 3 that our GBDT model ensures the highest accuracy of prediction among all the models. Specifically, 72.5% pages are corrected refreshed or not refreshed according to the user behaviour features. More importantly, our GBDT model ensures low false negative rate, which is important for the quality of the data collection of LEADS. Although the Random Forest model ensures lower false negative rate than the GBDT model, the corresponding false positive rate is much higher. Therefore, we use the GBDT model on top of the default crawl ordering mechanism of the crawlers to predict whether a web page in the data collection of LEADS should be refreshed or not. That is, when a web page is to be refreshed according to the default crawl ordering mechanism, we further leverage the user behaviour features to make more accurate prediction to make the final decision on whether to refresh the page to reduce redundant refreshing.

Table 3: Accuracy of different machine learning models w.r.t. user behaviour based features.

Model	False Negative Rate	False Positive Rate	Accuracy
GBDT	0.227	0.048	0.725
Naïve Bayes	0.408	0.025	0.567
Random Forest	0.186	0.130	0.684
J48 Decision Tree	0.263	0.123	0.614

To further improve the accuracy of the prediction, we add a set of features related to each URL to refresh as follows:

- URL age: Time since the last refresh/crawl of the page.
- URL length: Number of characters in the URL string of the page.
- Host Length: Number of characters in the hostname of the page.
- IsDynamic: Whether the page is a dynamic page or not. The content of a dynamic page is dynamically generated by the server when receiving the URL based on the variable parameters that are provided to the server in the URL.
- URL depth: The depth of a URL is the number of visits needed to reach that page from the homepage. It usually corresponds to the number of slashes in the URL,
- Parameter count: If the page is dynamic, the value is the number of parameters in the URL. If the page is static, the value is zero.
- URL importance: This is the importance of a web page given by the crawl-based data collection. It reflects the importance of the page to search results and its importance to discovery new pages.

Combining these features with the user behaviour based features we propose, we train the different machine learning models again. We can see from Table 4 that using the URL features help to improve

[1] http://www.cs.waikato.ac.nz/ml/weka/

the prediction accuracy from 0.725 to 0.798 and the GBDT model is always better than the baseline models. Therefore, we use the GDBT model and the user behaviour based features enhanced with the URL features for the crawling based data collection of LEADS.

Table 4: Accuracy of different machine learning models w.r.t. user behaviour based features and URL features.

Model	False Negative Rate	False Positive Rate	Accuracy
GBDT	0.126	0.076	0.798
Naïve Bayes	0.330	0.029	0.641
Random Forest	0.097	0.127	0.777
J48 Decision Tree	0.126	0.077	0.797

4. Conclusion

This deliverable focused on the pre-processing of the data obtained via the user-publishing interface, defining the machine learning models employed for predicting spam scores of web content and user comments, and explaining the integration of the pre-processing modules with the user-publishing back-end. Performed offline experiments indicate that in spam detection our models can achieve high accuracy. Furthermore, this document also presents one potential side benefit of the user-publishing plugin, namely efficient refresh ordering. Via offline experiments, the gains possible with this kind of approach to refresh ordering is demonstrated.

5. References

[ACD+11] Alekh Agarwal, Olivier Chapelle, Miroslav Dudik, John Langford, A Reliable Effective Terascale Linear Learning System, 2011.

[ATT+09] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas. The web changes everything: understanding the dynamics of web content. In Proc. 2nd ACM Int'l Conf. on Web Search and Data Mining, pages 282–291, 2009.

[CG00] J. Cho and H. Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In Proc. 26th Int'l Conf. on Very Large Data Bases, pages 200–209, 2000.

[CG02] J. Cho and H. Garcia-Molina. Parallel crawlers. In Proc. 11th Int'l Conf. on World Wide Web, pages 124–135, 2002.

[CG03] J. Cho and H. Garcia-Molina. Effective page refresh policies for Web crawlers. ACM Trans. Database Syst., 28(4):390–426, 2003.

[CG03B] J. Cho and H. Garcia-Molina. Estimating frequency of change. TOIT, 3(3):256–290, 2003.

[CLU09] http://www.lemurproject.org/clueweb09/index.php

[CS08] C. Olston and S. Pandey. Recrawl Scheduling Based on Information Longevity. In Proc. 17th Int'l Conf. on World Wide Web, pages 437-446, 2008.

[CSC10] Gordon V. Cormack, Mark D. Smucker, Charles L. A. Clarke, Efficient and Effective Spam Filtering and Re-ranking for Large Web Datasets, arXiv:1004.5168

[D1.2] D1.2: Real-time content discovery through on-the-fly publishing. LEADS. 2013.

[EMT04] N. Eiron, K. S. McCurley, and J. A. Tomlin. Ranking the web frontier. In Proc. 13th Int'l

Conf. on World Wide Web, pages 309–318, 2004.

[FCV09] D. Fetterly, N. Craswell, and V. Vinay. The impact of crawl policy on web search effectiveness. In Proc. 32nd Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval, pages 580–587, 2009.

[FMN+04] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. Softw. Pract. Exper., 34(2):213–237, 2004.

[FF99] J. H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. February 1999.

[FM99] J. H. Friedman. Stochastic Gradient Boosting. March 1999.

[GO14] https://github.com/GravityLabs/goose/wiki

[HBB10] M. Hoffman, D. Blei, F. Bach, Online Learning for Latent Dirichlet Allocation, in Neural Information Processing Systems (NIPS) 2010.

[NCO04] A. Ntoulas, J. Cho, and C. Olston. What's new on the web?: the evolution of the web from a search engine perspective. In Proc. 13th Int'l Conf. on World Wide Web, pages 1–12, 2004.

[PO05] S. Pandey and C. Olston. User-centric web crawling. In Proc. 14th Int'l Conf. on World-Wide Web, pages 401–411, 2005.

[WSR09] https://plg.uwaterloo.ca/~gvcormac/clueweb09spam/

Appendix A: Progress on geo-distributed storage and web crawling

This appendix briefly covers the state of the integration between Nutch, the web crawler in use in WP1, and Ensemble, the geo-distributed storage developed in WP2.

Ensemble: Ensemble is a geo-distributed storage platform that exports the same interface as Infinispan, the underlying storage infrastructure of LEADS. Ensemble is implemented on top of multiple out-of-the-box Infinispan deployments. Each such deployment materializes the storage of a single micro-cloud. Ensemble accesses a deployment via the HotRod Infinispan client-server protocol. An EnsembleCache (i.e., a data collection) implements the API of Infinispan atop multiple Infinispan deployments. To coordinate geo-distributed accesses to the storage, e.g., the concurrent creation of EnsembleCaches Ensemble stores both the list of micro-clouds, the definition of EnsembleCaches and the allocation of the latter to the former in a consistent and dependable way in ZooFence. ZooFence is a coordination service developed as part of WP2. It offers the same API as Apache ZooKeeper, but with improved performance when deployed at the geo-distributed scale.

Geo-distributing Nutch: We are working toward the integration of the Apache Nutch web crawler with the Ensemble infrastructure. Nutch is a state-of-the-art crawler that features a large set of techniques for efficient and sound data extraction from public Web pages. The combination of Nutch with Ensemble realizes a geographically-distributed web-crawler running transparently on top of the LEADS platform. Multiple Nutch instances can concurrently access the Web to retrieve public data and store it into Ensemble. The access to the Ensemble cache needs to be coordinated by the mechanisms previously mentioned. Nutch uses a series of queries, which are also natively supported by Infinispan on each site (we developed for this a connector between Nutch and other Apache projects and Infinispan, which is of interest also outside of the project).

We are in the process of integrating Nutch with Ensemble for the M25-M30 period. In details, our work will consist in (1) supporting versioned data, which is not a feature natively supported by Nutch, and (2) adding methods for energy and locality efficient crawling, by partitioning the Web graph across concurrent web crawlers in a smart way.