Infrastructure Support for Evaluation as a Service

Jimmy Lin[1] and Miles Efron[2]

[1] The iSchool, University of Maryland, College Park
[2] Graduate School of Library and Information Science, University of Illinois, Urbana-Champaign

jimmylin@umd.edu, mefron@illinois.edu

ABSTRACT

How do we conduct large-scale community-wide evaluations for information retrieval if we are unable to distribute the document collection? This was the challenge we faced in organizing a task on searching tweets at the Text Retrieval Conference (TREC), since Twitter's terms of service forbid redistribution of tweets. Our solution, which we call "evaluation as a service", was to provide an API through which the collection can be accessed for completing the evaluation task. This paper describes the infrastructure underlying the service and its deployment at TREC 2013. We discuss the merits of the approach and potential applicability to other evaluation scenarios.

Categories and Subject Descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

Keywords: TREC Microblog; tweet search

1. INTRODUCTION

Large-scale community-wide evaluations are integral to information retrieval research and play an important role in advancing the state of the art. Typically, they are organized around shared tasks, collections, and metrics, which support meaningful comparisons between systems. The Text Retrieval Conferences (TRECs) [9] in the US exemplify such evaluations, and the model has been successfully replicated in Europe (CLEF) and Asia (NTCIR, FIRE).

The Cranfield Paradigm [3], which underlies the evaluation methodology in TREC and other evaluation series, assumes that researchers can acquire the document collection under study—whether via physical CD-ROMs or DVDs (in the early days), hard drives (today), or directly downloadable "from the cloud". What if this is not possible? One example is a collection of tweets: Twitter's terms of service forbid redistribution of such data and thus it would not be permissible for an organization to host a collection of tweets for download by researchers. Although there are third-party resellers of Twitter data, the costs are too high for distributing research collections. Other examples of data that make wide distribution difficult include electronic medical records, a subject of substantial interest by researchers today—for obvious privacy concerns. Similar issues exist for email search and desktop search as well.

We present one workable solution to this challenge that has been operationalized in the TREC 2013 Microblog track: an approach we call "evaluation as a service", a play on current cloud computing technologies such as "infrastructure as a service" (IaaS), "platform as a service" (PaaS), and "software as a service" (SaaS). The basic idea is that, instead of distributing the collection, the evaluation organizers provide an API through which the collection can be accessed for completing the evaluation task. This paper describes the infrastructure underlying the service and its deployment at TREC 2013. We conclude with a discussion of its merits and potential applicability to other evaluation scenarios.

2. EVALUATION AS A SERVICE

The "evaluation as a service" approach implemented in the TREC 2013 Microblog track evolved out of an attempt to address deficiencies in the data distribution approach implemented in the previous two iterations of the track, which began in 2011. To provide better context we offer a quick recap here, but refer the reader to previous track overview papers for more details [5, 7]. In TREC 2011 and 2012, the Microblog track used the Tweets2011 collection, specifically created for those evaluations. Since Twitter's terms of service prohibit redistribution of tweets, it was necessary to develop an alternative mechanism for researchers to obtain the collection. The track organizers devised a process whereby NIST distributed the *ids* of the tweets (rather than the tweets themselves). Given these ids and a downloading program developed by the organizers, a participant could "recreate" the corpus. Since the downloading program accessed the `twitter.com` site directly, the tweets were delivered in accordance with Twitter's terms of service.

The "download it yourself" approach adequately addressed the no-redistribution issue but exhibited scalability limits. In particular, the speed of the downloading program, which had built-in rate limiting for "robotic politeness", set a practical upper bound on collection size. The Tweets2011 collection originally contained 16 million tweets, which is small by modern standards. For 2013, we hoped to increase the collection size by at least an order of magnitude, which required a completely new approach.

Our solution was to implement evaluation as a service. We gathered a collection of tweets centrally, but instead of distributing the tweets, we provided a service API through

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2577014.

which participants could access the tweets to complete the task. Below, we describe this approach in more detail.

2.1 Collection Construction

To build the official collection, we developed a custom crawler using the twitter4j Java library[1] that gathered tweets from Twitter's streaming API.[2] We crawled all tweets from the public sample stream between February 1 and March 31, 2013 (inclusive). This level of access is available to anyone with a Twitter account and does not require special authorization. The collection was gathered from two separate virtual machine instances on Amazon's EC2 service, one on the east coast of the US, and the other on the west coast of the US. The redundant setup guarded against network outages and other operational issues during the collection period. Fortunately, no downtime was experienced during the data collection period, so one of the copies was simply designated as the official collection.

Messages are delivered in JSON from Twitter's streaming API: these messages contain posted tweets as well as notifications of tweets that have been deleted. The crawler packages all messages in one-hour compressed chunks. Thus, the collection is comprised of 1416 gzipped files. In total, we gathered 259 million tweets, although at the time of the evaluation the collection was reduced to 243 million tweets after the removal of deleted tweets.

We made a decision early in the track planning phase that all software infrastructure associated with the evaluation would be open source and hosted on GitHub.[3] The code for the crawler was developed during January 2013, with input and discussion on a mailing list for developers. By mid-January, we had an operational crawler ready for testing by a few volunteers, and on January 23, 2013, the crawler was released to all participants.

The official collection period was publicized on the track mailing list well in advance of the actual start date, which gave participants the opportunity to run the crawler themselves to gather contemporaneous tweets. Although these crawls may not have the same content as the official collection, they are nevertheless useful for computing term statistics, background models, etc. Based on an informal survey conducted over the track mailing list in November 2013, at least half a dozen groups from around the world gathered their own local private collections.

2.2 API Specification

The main idea behind "evaluation as a service" is to provide a shared API using which participants can complete the evaluation without needing access to the raw collection. To this end, we provided a search API built using Thrift.[4]

Thrift is a software framework for developing scalable services. It was originally developed at Facebook, but is now an open-source Apache project. The framework has gained popularity over the last several years and is currently an integral part of production software stacks at many internet companies, including Facebook and Twitter. Thrift provides an Interface Definition Language (IDL) for describing services and data types. From these definitions, the Thrift compiler automatically generates RPC clients and servers as well as code for serializing, deserializing, and manipulating the defined datatypes. Thrift handles generation of boilerplate code for communications protocols, object transport, method invocation, and other functionalities necessary to build distributed services. The framework provides support for Java, C++, Python, Ruby, as well as many other languages, which allows the development of language-neutral services. For example, a Python Thrift client can communicate easily with a Thrift server written in Java because the communication protocols and data types are defined in a language-independent manner.

[1]`http://twitter4j.org/en/index.html`
[2]`https://dev.twitter.com/docs/streaming-apis`
[3]`http://twittertools.cc/`
[4]`http://thrift.apache.org/`

```
struct TQuery {
  1: string group,
  2: string token,
  3: string text,
  4: i64 max_id,
  5: i32 num_results
}

struct TResult {
  1: i64 id,
  2: double rsv,
  3: string screen_name,
  4: i64 epoch,
  5: string text,
  6: i32 followers_count,
  7: i32 statuses_count,
  8: string lang,
  9: i64 in_reply_to_status_id,
 10: i64 in_reply_to_user_id,
 11: i64 retweeted_status_id,
 12: i64 retweeted_user_id,
 13: i32 retweeted_count
}
```

Figure 1: Thrift definition of a query and a result.

```
service TrecSearch {
  list<TResult> search(1: TQuery query)
    throws (1: TrecSearchException error)
}

exception TrecSearchException {
  1: string message
}
```

Figure 2: Thrift definition of the search API. The service accepts a query and returns a list of results (as defined in Figure 1).

The Thrift definitions of the two main data types in the TREC Microblog search API are shown in Figure 1. The Interface Definition Language is similar to a C struct, and contains an enumeration of numbered fields, each with a type and a name. Most of the types are self-evident; `i32` represents a 32-bit integer (`int` in Java), while `i64` represents a 64-bit integer (`long` in Java). The `TQuery` object represents a query, which contains the query text, a `max_id` (i.e., requests the service to return only results smaller than the id), and the number of results requested. For simplicity, the service is stateless; access control is granted through a `group` and `token`, which must be passed in the query each time. The `TResult` object defines the search result (more details later). The service definition is shown in Figure 2. The single method `search`, receives a `TQuery` object and returns a list of `TResult` objects.

The service for the evaluation was written in Java using the open-source Lucene search engine (version 4.3.1 at the

Table 1: Detailed Description of a Search Result.

Thrift field	JSON element	Optional?	Type	Description
`id`	`status.id`	no	long	unique tweet id assigned by Twitter
`rsv`		no	double	retrieval status value, i.e., document score
`screen_name`	`status.user.id`	no	String	user who posted the tweet
`epoch`		no	long	UNIX epoch second when the tweet was posted
`text`	`status.text`	no	String	text of the tweet
`followers_count`	`status.user.followers_count`	no	int	the number of followers the user has
`statuses_count`	`status.user.friends_count`	no	int	the number of tweets the user has posted
`lang`	`status.lang`	yes	String	the language of the tweet
`in_reply_to_status_id`	`status.in_reply_to_status_id`	yes	long	the id of the tweet that this tweet replies to
`in_reply_to_user_id`	`status.in_reply_to_user_id`	yes	long	the id of the user who posted the tweet that this tweet replies to
`retweeted_status_id`	`status.retweeted_status_id`	yes	long	the id of the tweet that this tweet is a retweet of
`retweeted_user_id`	`status.retweeted_user_id`	yes	long	the id of the user who posted the tweet that this tweet is a retweet of
`retweeted_count`	`status_retweet_count`	yes	int	number of times this tweet has been retweeted

time of the evaluation).[5] We provided a sample client in Java to illustrate the features of the API. In addition, we received the contribution of a Python client from the community, which was later integrated into the code base.

Search ranking was provided using Lucene's implementation of query-likelihood (`LMDirichletSimilarity`). Results were filtered such that tweets with ids greater than `max_id` (as specified in the `TQuery` object) were discarded. Each search result was populated with the fields described in Table 1 (corresponding to the Thrift definition in Figure 1). For each field, the table also provides its corresponding element in the original JSON messages from the Twitter streaming API, whether the element is optional (for example, only retweets have certain fields), the corresponding Java data type, and a short description.

3. TREC DEPLOYMENT

The service described in the previous section was deployed for the first time in the TREC 2013 Microblog evaluation. Implementation of the search infrastructure progressed during Spring 2013 and by June the service endpoint, which ran on Amazon's EC2 service, was released to all registered TREC participants. The service was available until the TREC evaluation deadline in August. We maintained two distinct services: one on the Tweets2011 collection created for the Microblog tracks in TREC 2011 and TREC 2012, and another on the new collection gathered in 2013. Both services behaved exactly the same, except on different document collections. Since evaluation data were available for the Tweets2011 collection, that service allowed participants to train their systems on old topics.

In the official evaluation, TREC received 71 runs from twenty groups around the world, making the Microblog track the largest at TREC 2013 in terms of number of participating teams. From June 7 to August 16, 2013, the API on the Tweets2011 collection served over 555k queries—this usage corresponded to teams developing systems with data from previous years. During roughly the same interval, the API on the 2013 collection served 93k queries.

Based on the level of participation and the usage statistics, the evaluation-as-a-service approach seems to have been successful. We were able to meet our original goal of expanding the collection by an order of magnitude while respecting Twitter's terms of service. The API was easy enough to use and did not pose a high barrier to entry for participation. At the same time, the API was rich enough to allow teams to explore the research questions they were interested in.

[5]`http://lucene.apache.org/`

4. DISCUSSION

The evaluation-as-a-service approach enabled community-wide evaluations on documents that cannot be distributed. However, we see other advantages to this model as well:

More meaningful system comparisons. Modern information retrieval systems have become complex collections of components for document ingestion, inverted indexing, query evaluation, document ranking, and machine learning. As a result, it can be difficult to isolate and attribute differences in effectiveness to specific components, algorithms, or techniques. Consider a baseline retrieval model such as BM25 or query-likelihood within the language modeling framework—alternative implementations may produce substantially different retrieval results due to small but consequential decisions such as the tokenization strategy, stemming algorithm, method for pruning the term space (e.g., discarding long or rare terms), and other engineering issues. Although the prevalence of open-source retrieval engines makes it possible for a researcher to see exactly what a system is doing, in practice few cross-system comparisons are performed with "calibration" on baseline models, making it sometimes difficult to compare advanced techniques based on different system implementations. In some cases, the effects that we are hoping to study are masked by differences we are not interested in.

The evaluation-as-a-service model addresses many of these issues by deploying a common API that is used by all participants. This means that everything "below" the API is *exactly* the same for everyone. Thus, we can be confident that differences in effectiveness can be attributed to retrieval techniques on top of the API, rather than "uninteresting" issues such as tokenization and stemming.

Support of open-source community development. A decision we made early on in the development of the evaluation infrastructure was that all associated software would be open source. It is desirable that all participants know exactly how the API works and have access to all the minor but potentially important decisions that were made in its implementation (see above).

We hope that this decision has the additional effect of more effectively fostering an open-source community of pluggable system components. There is growing recognition within the IR community that open source software helps advance the state of the art; a common API increases the likelihood that code components inter-operate, thus increasing the likelihood of adoption. Although there is already widespread availability of open-source retrieval engines, nearly all systems are monolithic in the sense that they were not designed for service decomposition along functional boundaries. This means that a particular algorithm developed for one system cannot be easily used by researchers who have written their code on another system due to interface incompatibilities. A common API begins to address this issue.

Solution for systems engineering issues. To reflect today's retrieval environment, modern document collections for IR evaluations have grown quite large—sizes in the tens of terabytes are common. Manipulating these large collections represents non-trivial engineering challenges. Despite the field's growing familiarity with large-scale distributed frameworks such as Hadoop, the available open-source solutions are not quite yet "turn key". Although the size of the tweet collection used in the TREC 2013 Microblog track remains manageable ($\sim$100 GB compressed), the large sizes of other document collections (e.g., ClueWeb12 or the TREC KBA corpus) pose a barrier to entry for many research groups. Even for well-resourced research teams with access to large compute clusters, the effort devoted to systems engineering challenges might be better spent on the development of retrieval techniques.

The evaluation-as-a-service model provides a solution to these systems engineering issues. Scalability challenges only need to be solved once, by the developers of the API. Participants need not be concerned about systems issues that are hidden behind the abstraction.

To be fair, however, the evaluation-as-a-service model suffers from several challenges:

Limited diversity in retrieval techniques. The biggest drawback of a common API is that it prescribes a particular approach to the evaluation task. Any abstraction necessarily restricts the flexibility of researchers to tackle the problem in creative ways. One possible way to mitigate this concern is to institute community-driven processes for refining the API so that researchers' needs are met. Since the code is open source, we are able to accept contributions that augment the capabilities of the service.

Unknown evaluation characteristics. The information retrieval literature has a tradition of studies that enhance our understanding of the limits of test collections, e.g., their reusability, stability, topic effects, and other related issues (e.g., [8, 1, 6], just to name a few). These studies collectively give us confidence that our evaluation tools can be "trusted". These meta-evaluations need to be conducted on the API.

Long-term availability of service APIs. One essential property of well-built test collections is reusability by researchers who did not participate in their original creation, often long after the initial evaluation. A traditional test collection, once created, requires relatively modest resources to maintain. Service APIs, on the other hand, are far more expensive to operate: first, hardware resources must be procured (whether physical servers or infrastructure "in the cloud"); second, humans must be in the loop for both administrative functions (e.g., granting access to new users) and to ensure availability (e.g., restarting the service when it crashes). Longevity is especially a concern—for example, TREC collections from the 1990s are still used today. In the evaluation-as-a-service model, we have not developed a process for sustaining the service API in the long term.

5. DEMONSTRATION

In this demonstration we will take the user on a tour of the infrastructure described in this short paper to provide a sense of how all the software components fit together. We have prepared two different open-source clients that use the API: the first performs minimal post-processing on the raw output to create a baseline ranking, while the second implements relevance models for query expansion [4], which has been shown to be very effective for the tweet search task. Users will be able to try different queries and examine the execution traces of both algorithms.

6. CONCLUSIONS

"Evaluation as a service" began as an attempt to overcome a specific usage restriction, but we see broader applicability for other evaluation scenarios where collections cannot be distributed, such as medical records search. We can push the idea even further, by shipping code to the evaluation infrastructure: Participants would submit their systems, and evaluation would be conducted without the participants ever touching the sensitive evaluation data. Although this approach is not entirely new – previous examples include the TREC 2005 Spam track [2] and the Music Information Retrieval Evaluation eXchange (MIREX) – the maturation of technologies such as virtualization and standardized RPC mechanisms make this approach easier to implement.

7. ACKNOWLEDGMENTS

This work has been supported by NSF under awards IIS-0916043, IIS-1217279, and IIS-1218043. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsor.

8. REFERENCES

[1] C. Buckley and E. M. Voorhees. Retrieval evaluation with incomplete information. *SIGIR*, 2004.
[2] G. Cormack and T. Lynam. TREC 2005 spam track overview. *TREC*, 2005.
[3] D. Harman. *Information Retrieval Evaluation*. Morgan & Claypool Publishers, 2011.
[4] V. Lavrenko and W. B. Croft. Relevance-based language models. *SIGIR*, 2001.
[5] I. Ounis, C. Macdonald, J. Lin, and I. Soboroff. Overview of the TREC-2011 Microblog Track. *TREC*, 2011.
[6] M. Sanderson and J. Zobel. Information retrieval system evaluation: Effort, sensitivity, and reliability. *SIGIR*, 2005.
[7] I. Soboroff, I. Ounis, C. Macdonald, and J. Lin. Overview of the TREC-2012 Microblog Track. *TREC*, 2012.
[8] E. M. Voorhees. Variations in relevance judgments and the measurement of retrieval effectiveness. *SIGIR*, 1998.
[9] E. M. Voorhees. The philosophy of information retrieval evaluation. *CLEF*, 2002.