A Recommender System for Recovering Relevant JavaScript Packages from Web Repositories

Hernan C. Vazquez
Faculty of Sciences
UNICEN University
Tandil, Buenos Aires, Argentina
hvazquez@exa.unicen.edu.ar

J. Andres Diaz-Pace
ISISTAN Research Institute
CONICET & UNICEN University
Tandil, Buenos Aires, Argentina
andres.diazpace@isistan.unicen.edu.ar

Santiago A. Vidal
ISISTAN Research Institute
CONICET & UNICEN University
Tandil, Buenos Aires, Argentina
santiago.vidal@isistan.unicen.edu.ar

Claudia Marcos
ISISTAN Research Institute
CIC & UNICEN University
Tandil, Buenos Aires, Argentina
claudia.marcos@isistan.unicen.edu.ar

***Abstract*—When developing JavaScript (JS) applications, the assessment of JS packages has become a difficult and time-consuming task for developers, due to the growing number of technology options available. Given a technology need, a common developers' strategy is to browse software repositories via search engines (e.g., NPM, Google) and identify candidate JS packages. However, these engines might return a long list of results, which often causes information overloading issues in the developer. Furthermore, the results should be ranked according to the developer's criteria, but weighting the available criteria to choose a JS package is not straightforward. To address these problems, we propose a two-phase recommender system for assisting developers in retrieving and ranking JS packages in a semi-automated fashion. The first phase uses a meta-search technique for collecting JS packages that meet the developer's needs. Based on criteria used by other projects on the Web, the second phase applies a machine learning technique to infer a ranking of relevant packages for the output of the first phase. We performed an initial evaluation of our approach with the NPM package repository and obtained satisfactory results in terms of both the accuracy of the retrieved packages and the quality of the ranking for the developers.**

***Index Terms*—technology selection, component-based design, Javascript packages, software repositories.**

I. Introduction

The use of software technologies, such as libraries and frameworks, can greatly improve developers' productivity in terms of accelerating development cycles and delivering value to customers. Nonetheless, an inappropriate technology selection can negatively affect the software products and the business goals of the organization [1]. In the context of JavaScript (JS) applications, the task of selecting (and reusing) a JS package that fulfills the needs of a development task can be a complex and time-consuming decision-making process for developers. One of the reasons for this complexity is the availability of a large number of options in Web repositories, such as NPM[1] (Node Package Manager) [2]. Thus, JS developers have to regularly search, evaluate and compare several JS packages for their applications, and keeping up-to-date with technology becomes challenging. This process can be perceived as a "technological fatigue"[2] by developers.

We argue that one of the reasons for JS technological fatigue is the lack of effectiveness in the search engines for JS repositories. As a consequence, developers often resort to general-purpose search engines (e.g., Google or Bing) with the hope of having better results. However, the downside of such engines is that they tend to return long lists of documents, and then developers have to navigate within each result to find candidate JS technologies. This leads to information overloading issues. Furthermore, once the developer identifies a set of candidate packages for her application, she must analyze each one to decide the best fit for her needs. Normally, this decision is driven by package features, such as: popularity in the community, or number of downloads, among others. Assigning weights to these features for comparison purposes is not straightforward. For instance, NPM uses an AHP (Analytic Hierarchy Process) [3] technique to compare JS technologies.

Information overloading issues have been studied in various disciplines [4], and Web search engines are one of the main tools to face the problem [5]. Although search engines for software repositories have received some attention in the literature, there are still challenges in finding packages relevant to a particular development (or technological) need [6] [7] [8] [9]. Previous works have addressed software technology selection from different perspectives [10] [11] [12] [13] [14], mostly focusing on the evaluation of candidate items based on specific characteristics, and departing from a predetermined group, while leaving aside the aspect of searching for relevant candidates. In addition, the characteristics are often manually assessed by developers before decision-making.

In this work, we propose a recommender system to assist developers in searching and ranking JS packages. The system works in two phases called *ST-Retrieval* and *ST-Rank* (ST

[1]https://www.npmjs.com/

[2]https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4

stands for Software Technologies). Given a developer's query expressing a JS technological need, *ST-Retrieval* applies a meta-search strategy [15], which combines the results from both a JS-specific engine and general-purpose engines. The latter allows us to expand the search scope for JS technologies.

Based on the packages recovered by *ST-Retrieval*, *ST-Rank* ranks them by relevancy by means of a Machine Learning (ML) model. This model is built using a pairwise learning-to-rank method [16]. In this way, the ML model can infer a ranking of JS packages based on an analysis of features (e.g., number of stars in the repository, number of releases, number of contributors, etc.) extracted from JS projects on the Web (e.g., GitHub). To assess package relevancy, we employ a metric that assesses the relationship between the projects in which a given package was used and the project's popularity.

We have evaluated the effectiveness of the recommender system on the NPM repository for JS packages, using a predefined set of queries and 1000 popular projects from GitHub. For *ST-Retrieval*, our experiments showed an average precision improvement of 20%, and we were able to recover a larger number of relevant packages than using the default search engine provided by NPM. Regarding *ST-Rank*, we observed improvements of at least 20% on average when compared to the default ranking strategy followed by NPM. Our approach makes two contributions: (i) it can boost the performance of the NPM search system by leveraging on results provided by multiple search engines, and (ii) it supports developers by learning (from data from the JS community) how to rank packages that fulfill the developers' needs.

The rest of the paper is structured as follows. In Section II, we provide a brief description of the software technology selection problem, along with a motivational scenario of JS development. In Section III, we describe the two phases of the approach in detail. Section IV presents a series of experiments to evaluate *ST-Retrieval* and *ST-Rank*, and discusses their pros and cons. Section V covers related work. Finally, Section VI gives the conclusions and outlines future work.

II. Software Technology Selection

From a development perspective, the selection of software technologies influences both the development process and the quality of the final product [17]. The successful application of a given technology, such as a JS package, means that its usage for a task produces a desired objective [10]. This success also depends on contextual features, such as alignment between the developer's requirement and the chosen package, package maintenance support, or license type, among others.

As a motivating example of JS technology selection, let us consider a developer that needs to extract a barcode from an image to automate a process for processing codes from image files. The application context for this functionality is a Web browser. Initially, the developer goes to the NPM package repository and submits the query "extract barcode from image" to a search engine, which returns only the *bytescout*[3] package as output. Bytescout is a JS client for a cloud service. When reading about *bytescout*, she finds out that it is a paid service and that the JS client is not open-source. Also, when she looks at the description, NPM reports that *bytescout* has been downloaded 40 times in the last month, which might indicate that it is not very popular in the JS community. Let us assume that our developer is not convinced by these features, or that they are not aligned with her project needs. However, *bytescout* is the only technology returned by NPM. In this context, several options arise: (i) adopt the package despite disagreeing with its features, (ii) implement a solution for reading barcodes from scratch, (iii) submit a modified query to NPM to get more results, or (iv) rely on other information sources (e.g. Google, or NPMSearch, among others) to find alternative technologies. Let us suppose that our developer picks the third option and re-phrases the query as "barcode reader", which makes NPM return 16 results this time. After inspecting each result, she is still unconvinced about using any of those technologies, since they do not seem to be very popular nor have enough maintenance. The scenario so far shows the current limitations of JS-specific search engines, like NPM.

Let us assume that our developer goes instead for the fourth option and submits the query "extract barcode from image javascript package" to Google[4]. This query returns a list of Web pages that are inspected by the developer to check whether some JS packages are mentioned. In doing so, she realizes that a package called *QuaggaJS*[5] is referenced in three results from the top-10 pages of the list. As she is not aware of this technology, she goes back to the NPM repository and finds that *QuaggaJS* is more popular than *bytescout*, it is open-source and well-maintained by the community. At this point, our developer can either pick *QuaggaJS* to fulfill her development need, or keep looking for alternative packages. This scenario illustrates the challenge of using general-purpose search engines for JS packages and the issues related to their comparison. Overall, there are two implications of the example. First, the search and comparison of JS packages can take advantage of different information sources. Second, the data from existing JS projects can provide useful criteria (or feedback) to assess the relevance or fitness of a given package. These ideas inspired the development of the *ST-Retrieval* and *ST-Rank* components in our recommender system.

III. Approach

For retrieving and ranking relevant JS packages[6], we propose a two-phase recommender system as shown in Figure 1. Each phase involves different steps. Gray boxes in the figure correspond to steps or artifacts provided at configuration time, while white boxes refer to steps performed every time a new query is entered by the JS developer. The first phase, called *ST-Retrieval*, takes a developer's query and returns a list of candidate packages matching the query. To do so,

[3]https://bytescout.com/

[4]The last phrase intends to prevent Google from returning results for packages in other languages.

[5]https://serratus.github.io/quaggaJS/

[6]For simplicity, the words "package" and "technology" are used interchangeably as synonyms in the paper.

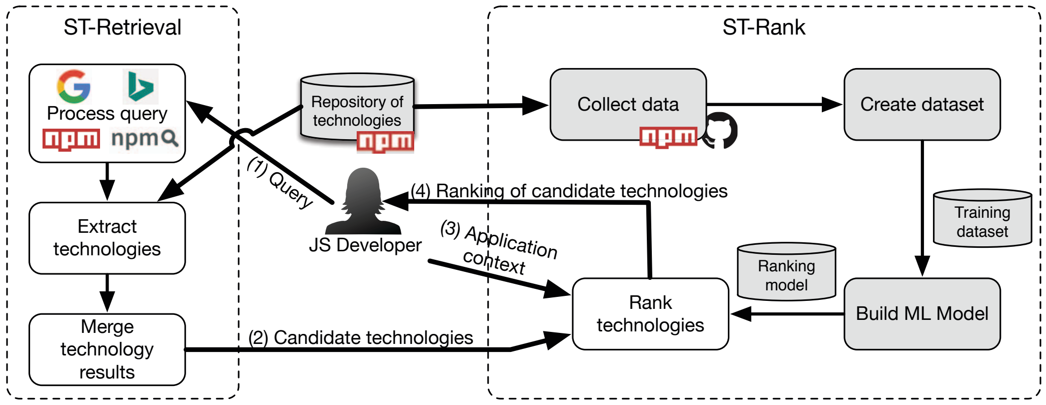


Fig. 1. Overview of the two-phase recommendation approach.

we leverage multiple search engines. The query is written in natural language and specifies a technological requirement (e.g. “extract barcode from image”). Next, the second phase, called *ST-Rank*, is fed with the *ST-Retrieval* outputs along with an *application context* to generate a ranking of JS packages.

With the application context, we expect the developer to specify where a JS package will be applied (e.g., a Web browser), since certain packages only work in a particular execution environment. For this work, JS packages are classified according to three possible environments: (i) technologies running in a Web browser, (ii) technologies to be used in Node.js, and (iii) technologies that do not require a specific environment. This list is amenable to extensions in the future.

In *ST-Rank*, we seek to (semi-automatically) construct a package ranking by looking at technology features and decisions made in other JS projects. One interesting approach to achieve this goal is to take a data-driven perspective and build an ML model that “learns” from information available in Web repositories (e.g., Github) how to sort packages according to preference, importance, or popularity, among other factors. Thus, we apply here a *pairwise learning-to-rank* method [16].

A. *ST-Retrieval*

The retrieval of JS packages is treated as a meta-search problem [15]. In meta-search, the original query is sent in parallel to a set of search engines, each one returning an ordered list of items that satisfy the query. All the lists are combined into a new one that should keep the “best” items of the individual lists. The technical aspects for processing a query and merging its results are explained below.

1) Process query: Currently, we rely on four search engines to broaden the scope for a query: NPM, NPMSearch[7], Google, and Bing[8]. In general, our approach supports two types of engines: *domain-specific* and *general-purpose* engines. The former are designed for the search of JS technologies (NPM and NPMSearch), while the latter are used for general queries (Google and Bing).

[7]https://npmsearch.com
[8]https://www.bing.com

Both engine types allow developers to enter queries in natural language, which express functional or quality-attribute requirements to be met by a JS package. Before submitting a query, stop-words (e.g., articles, pronouns, prepositions) that do not provide information to the search engine are removed. For general-purpose engines, the query is augmented with a JS suffix (“javascript package”) to avoid results from other domains (e.g., packages being incompatible with the current project). Once the processed query is executed, it returns a set of Web pages (or documents) in HTML, XML or JSON format. The next step is about extracting the JS packages named in those documents.

2) Extract technologies: We represent and store JS packages (or technologies) as tuples of the form $< name, repository\_url, home\_url, description >$ where:

- *name*: corresponds to the package name in the repository (e.g., “quagga”).
- *repository_url*: is the package URL on the repository site (e.g., “https://www.npmjs.com/package/quagga”).
- *home_url*: is the URL of the package site (e.g., “https://serratus.github.io/quaggaJS/”).
- *description*: is the package description (e.g., “An advanced barcode-scanner written in JavaScript”).

Our approach requires an initial repository of JS technologies beforehand. This repository is generated through a Web crawling process on the NPM site[9]. Thus, when querying domain-specific engines, each result is expected to have a mapping with an existing package in the repository via its *name* and *repository_url*. On the other hand, when querying general-purpose engines, the results will be Web documents that might have references to zero or more JS packages. Thus, the references should be parsed and extracted from the documents. This extraction can be seen as a Named Entity Recognition (NER) task. NER [18] aims at classifying entities found in a given text into predefined categories (e.g., people, organizations, places, among others). In our work, the named entity category is “software technology”, and we

[9]In order to complement the information found in NPM, we only selected packages whose repositories were publicly available in GitHub

perform string-matching using a rule-based strategy [19]. In this way, those packages whose name or address (*home_url*, *repository_url*) match the category are extracted. Figure 2 shows an example of search results and JS packages obtained from the NPM and Google engines for the query "extract barcode from image". NPM returned one result (*bytescout*) matching a package name in the repository. Unlike NPM, Google returned an HTML document that was parsed for matches of package names or addresses. In particular, the package name in the resulting page (QuaggaJS) did not match any package in the repository (quagga) but the site address got a match (*home_url*, https://serratus.github.io/quaggaJS/).

3) Merge technology results: Based on the named packages just extracted, an ordered list of packages per search engine is created. For domain-specific engines, each package is simply assigned to the position of the result where it was named[10]. For general-purpose engines, a given result might have several named packages. In this case, the sorting criterion looks at the order in which the packages were named. That is, if packages A and B are named within the same result, and A appears before B in the document, then A will go before B in the list. For instance, if the first result contains "You can use Quagga or Barcode-Reader" and the next result contains the "You should use Bytescout", then the extracted packages will be ordered as 1) *quagga*, 2) *barcode-reader*, and 3) *bytescout*.

After the individual lists from all the search engines are collected, they get merged into a single list. In meta-search, this process can be achieved with a ranking aggregation function [20]. For engines that do not expose similarity scores for their results, as in the cases of Google or Bing, ranking positions can be used instead [21]. *ST-Retrieval* relies on the Borda Fuse method for merging the lists. Borda Fuse is a positional aggregation function that is computationally simple to implement and performs well in Web search [20] [15].

In Borda Fuse, each search engine is considered as a voter having a list of *n* ordered candidates (i.e., the JS packages). For each list, the best first candidate receives *n* points, the second candidate receives *n-1* points, and so on. The points awarded by the different voters are added and the candidates are ranked in descending order according to the total points obtained. Table I exemplifies an aggregation using Borda Fuse. Each list gives a maximum of 4 points (equal to the maximum length of the individual lists) and decrements the points (by one) as one goes down in the list. Then the scores are added, and a final list of candidates ordered from highest to lowest scores is created. In our example, the most relevant packages in the individual lists (*quagga* and *bytescout*) ended up at the top of the final list.

B. ST-Rank

ST-Retrieval provides a set of candidate JS packages for the developer's query. Assuming that these candidates fulfill the same goals, the developer should scrutinize them to select the package that best suits her needs. This analysis is often based on indicators provided by the repository (e.g., the number of downloads, dependent projects, or contributors, among others). Furthermore, the analysis also involves searching for technical opinions in blogs and forums. In this context, *ST-Rank* is responsible for refining the outputs of *ST-Retrieval* and ranking the packages based on data from popular JS technologies used by other people. Thus, *ST-Rank* helps the developer to perform the kinds of (manual) analyses above. To implement this assistance, we collect data from both the NPM site and open-source repositories, assemble a dataset, and then train a supervised ML model for predicting package rankings. The technical aspects of this process are explained next.

TABLE I
BORDA FUSE AGGREGATION EXAMPLE ([POINTS] NAME).

NPM	Google	Bing	Final list
[4] bytescout	[4] quagga	[4] quagga	[8] quagga
	[3] bcreader	[3] bc-js	[6] bytescout
	[2] bytescout	[2] *bwip-js*	[4] bcreader
	[1] *jaguar*	[1] bcreader	[3] bc-js
			[2] *bwip-js*
			[1] *jaguar*

Our rationale for capturing a variety of package features is that, if a given technology T was preferred in a project (over other available options), a criterion that renders T more relevant (than the other options) should be derivable from those features. Thus, we propose to learn the selection criteria by means of a data-driven strategy.

1) Collect data: Given a JS package P, we represent it by a number of characteristics (or features) and also by its dependencies on other packages. In NPM, these dependencies are located in the *package.json* of each project[11] and need to be individually processed. In particular, when processing a dependent technology T, we perform three tasks: (i) search for alternatives to T, (ii) assess its "popularity", and (iii) retrieve features from the NPM and GitHub repositories.

For the first task, we rely on NPMCompare[12], which gives a list of related packages for any given JS technology. For example, *Chart.js*[13] depends on *moment*[14], which is a package to manipulate dates (Figure 3). Alternative technologies for the same need are, for instance, *date-fns*[15] and *momentjs*[16]. To get these alternatives, we implemented a scraping process on the NPMCompare website.

For assessing the popularity of a package (technology) T, we devised a metric called *CDSel (Community Degree of Selection)* that models the relationship between the projects in which T was selected and the relevance of those projects. Let n be the number of reference projects, $rel(P_i)$ the relevance of the i-th project, and $s = 1$ if Ti was selected in P_i ($s = 0$ otherwise), then $CDSel(T_i)$ is computed by Equation 1. This

[10]This is because each result corresponds unequivocally to a package

[11]Since both the "reference projects" and their dependencies are JS packages themselves, for the sake of clarity we refer to the former as "projects" and to the latter as "technologies"

[12]https://npmcompare.com

[13]https://www.npmjs.com/package/chart.js

[14]https://www.npmjs.com/package/moment

[15]https://www.npmjs.com/package/date-fns

[16]https://www.npmjs.com/package/momentjs

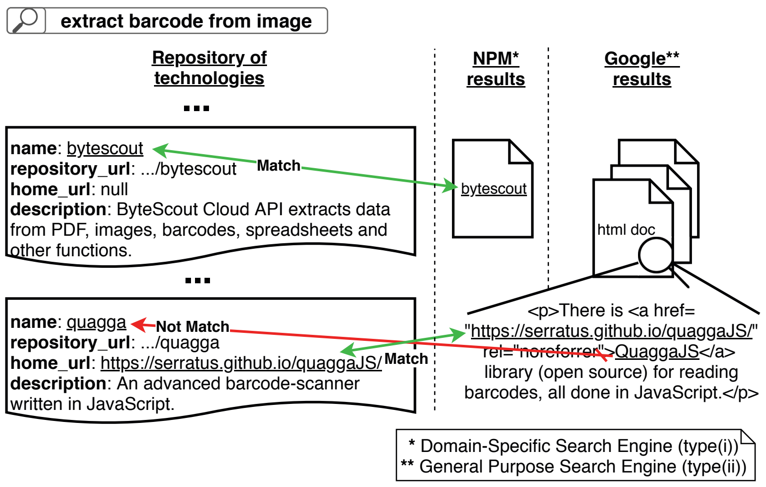
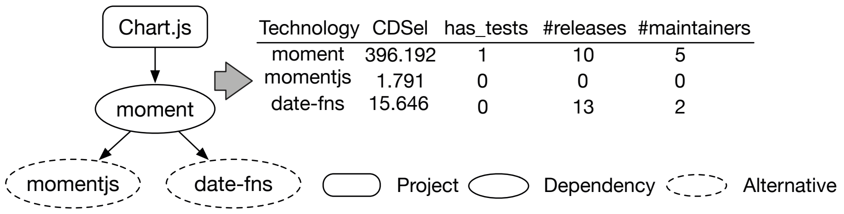

extract barcode from image
Repository of technologies
NPM* results
Google** results
name: bytescout
repository_url: .../bytescout
home_url: null
description: ByteScout Cloud API extracts data from PDF, images, barcodes, spreadsheets and other functions.
Match
bytescout
html doc
name: quagga
repository_url: .../quagga
home_url: https://serratus.github.io/quaggaJS/
description: An advanced barcode-scanner written in JavaScript.
Not Match
Match
<p>There is <a href="https://serratus.github.io/quaggaJS/" rel="noreferrer">QuaggaJS</a> library (open source) for reading barcodes, all done in JavaScript.</p>
* Domain-Specific Search Engine (type(i))
** General Purpose Search Engine (type(ii))

Fig. 2. Matching search results with JS technologies for different engines.



Technology	CDSel	has_tests	#releases	#maintainers
moment	396.192	1	10	5
momentjs	1.791	0	0	0
date-fns	15.646	0	13	2

Fig. 3. Construction of a technology dataset based on features and dependencies.

is how a package ranking is determined. The logarithm serves as an attenuating factor in the ranking. For example, in our dataset we obtained a CDSel value of 396.192 for *moment*, 15.646 for *date-fns*, and 1.791 for *momentjs*; which would mean that *moment* is selected more often than *date-fns* and *momentjs* in the repositories. The package relevance $rel(P_i)$ is given by Equation 2, with z being the length of the ranking and $rankPosition(P_i)$ the ranking position of project (P_i). Currently, $rankPosition(P_i)$ is based on the stars of each project in GitHub.

$$CDSel(T_i) = \sum_{i=1}^{n} s * \frac{rel(P_i)}{log_2(i+1)} \tag{1}$$

$$rel(P_i) = z - rankPosition(P_i) \tag{2}$$

Regarding the third task, we consider different features that influence developers' decisions, such as: number of downloads, number of dependent projects, project stars, number of developers contributing to the project, number of subscribers, number of commits, number of files, presence of tests, among others. More than 40 features[17] collected from NPM, Github, and the project source code were taken into account. The features were mostly numeric (quantitative) ones. Later during the ML task, we can assess which features are most relevant for the rankings.

2) Create dataset: We assembled a dataset for building an ML model that is able to rank JS packages. This training dataset contains a set of *training instances*, each one capturing a pair of technologies. Initially, a *training ranking* is computed for each technology according to its CDSel value. In our example, *moment* will be ranked first since its CDSel value is higher than the values of *date-fns* and *momentjs*. Then, each technology is represented as a feature vector $[FT_{i1}, FT_{i2}, ..., FT_{in}]$ where FT is an individual feature and n is the total number of features. Each vector is normalized to $[0..1]$ using feature scaling [22]. For example, in Figure 4 vector (1, 10, 5) for *moment* becomes (1, 0.77, 1) after normalization.At last, for each possible pair of technologies T_i and T_j in the *training ranking*, a vector (i.e., a training instance) is created as the concatenation of the normalized vectors for T_i and T_j. The label 1 is added to this vector when T_i is more relevant than T_j, and the label 0 is added otherwise. In our example, *(moment,date-fns)* is set to 1 (first row in right-most table of Figure 4) because *moment* is ranked higher than *date-fns* in the training ranking (left-most table).

3) Build ML model: Once the training dataset is constructed, we apply learning-to-rank technique [16] that works on the instances as if it were a binary classification problem. A pairwise supervised variant is used since the order of two

[17]The full list of features can be found in the Supplementary Material at https://bit.ly/2w9sOzV.

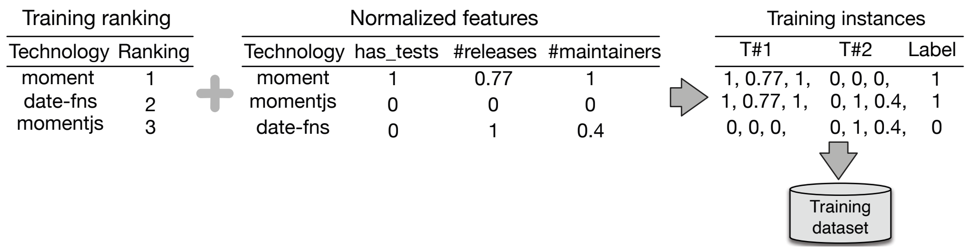


Fig. 4. Derivation of a training dataset for a learning-to-rank task.

given packages is not affected by other packages in the list. For instance, *moment* is more relevant than *date-fns* regardless of other possible alternatives. Among the various learning-to-rank algorithms reported in the literature [23], we chose GBRank [24] due to its effectiveness in Web search ranking [25]–[27]. GBRank is based on a gradient-boosting mechanism for minimizing wrong preference predictions.

4) Rank technologies: The ML model above is able to predict the order for any JS package pair. In this context, all the possible pairs coming from the last step of *ST-Retrieval* (*Merge technology results*) are taken and run through the ML model to generate a ranking. In addition, the pairs are filtered according to the application context defined by the JS developer. As a result, those packages being relevant (as determined by the ML model) should be ranked first by *ST-Rank*.

IV. Evaluation

We performed an empirical evaluation of our recommendation approach using data sampled from JS projects. In particular, we addressed the following research questions:

- **RQ#1**: Can *ST-Retrieval* perform better than existing search engines?
- **RQ#2**: Do the rankings proposed by *ST-Rank* have a better quality than the rankings of NPM and NPMCompare?

The experiments to answer RQ#1 and RQ#2 are reported in sub-sections IV-A and IV-B respectively.

A. *ST-Retrieval Experiments*

The goal is to compare the performance of *ST-Retrieval* (when returning JS packages) against that of the domain-specific and general-purpose search engines. We used NPM as the basis for the evaluation, as NPM is the de-facto package repository for JS technologies.

1) Experimental Design: We followed the steps defined in Figure 1 (left side). Initially, we downloaded the package registry from NPM and built the repository of technologies up to a given date (08/28/2017). Then, we asked two senior JS developers to record any technology queries in NPM that they would make in their projects, for a time period of two weeks. After filtering out some of their search results (to remove very similar queries), we obtained a *reference set* of 25 queries that represent a variety of technological needs, as listed in Table II. We ran *ST-Retrieval* 25 times on this set (once per query) and stored the returned lists of packages. The search engines used by *ST-Retrieval* were: NPM, NPMSearch, Google, and Bing. The query length was selected as a normal distribution over the most usual length [28]. A retrieval effectiveness test was also run for the engines [29].

TABLE II
Reference Queries used for ST-Retrieval.

Queries
check valid email address
download web videos
send sms
quick sort algorithm
filter adult content images
user authentication
extract barcode from image
convert data formats
download free music
convert typewritten image to text
sentiment analysis
third party authentication
convert text to speech
calculate word similarity
translate english to spanish
credit card validation
health tracker
captcha authentication
detect text language
rank aggregation algorithms
mobile app framework
DOM manipulation utils
lightweight 3D graphic library
mathematical functions
scraper

When processing the results, we took into account the first 20 documents from the lists of packages, as users searching the Web (e.g., using Google) are very likely to consider only the first pages [30]. These documents were recorded in order, each time the processing step was executed. We ended up with a total of 2760 JS packages retrieved (in general, *Google* and *Bing* returned various packages in the documents).

To establish a *baseline*, we asked 12 JS senior developers (different from the ones that worked on the reference set of queries) to assess the 2760 technology results. Each developer was assigned to all the packages (in random order) for a given query and was asked to judge their technology relevance using a binary scale (yes/no). The developers did not know which search engine produced a particular result. In particular, 11 developers analyzed the results of two queries each and the remaining developer analyzed the results of three queries. In this exercise, each developer analyzed 230 JS packages on average.

2) Metrics: Information Retrieval metrics, such as: precision, recall, MAP (Mean Average Precision) and DCG (Discounted Cumulative Gain), were used for gauging performance [31]. To compute these metrics, we leveraged on both the reference set of queries and the technology baseline described previously. In our domain, precision is the ratio between the number of relevant packages (as marked by the developers) and the total number of packages recovered by *ST-Retrieval*. Recall is the ratio between the number of relevant packages recovered and the total number of relevant packages known. Values of precision and recall closer to 1 are often desirable. In addition, MAP [32] measures, for a set of queries, the average precision per query. MAP values close to 1 mean that the relevant packages are within the top-ranked positions. At last, DCG [33] assesses the ranking quality in terms of the utility (gain) of a result based on its relevance and position in the ranking. DCG can be divided by the maximum value from all the queries. This variant is called nDCG (Normalized DCG).

3) Analysis of Results: Precision and recall were calculated in different positions of the ranking to establish a metric value in that position. Figure 5 shows the results for the four search engines and the Borda Fuse function considering each position (k) of the ranking ($k = 1$ refers to the first position). The first chart shows the average number of hits (*Hits*). A result is regarded as a hit if it is relevant to the query. The second and third charts present *Precision*, and *Recall* values, respectively.

As it can be observed, *ST-Retrieval* exhibits high values in most positions for all metrics. *Google* and *NPM* behave similarly to *ST-Retrieval* until $k = 3$ approximately. From that position on, *ST-Retrieval* shows a considerable boost in performance. We believe that this effect is due to the aggregation of results, and also to the ability of *Borda Fuse* to find the global relevance of a package based on its position in the lists relative to the different search engines. The maximum precision for *ST-Retrieval* was 0.72 (an 8% increase when compared to the search engines) and the maximum recall was 0.77 (a $\approx 34\%$ higher than the recall of the search engines).

Table III shows the values for *nDCG* (nD) and *MAP* (M) for ranking sizes of 5, 10, and 20. It also shows the values of *Hits* (H) and *Precision* (P) in positions 5, 10, and 20, respectively. Interestingly, *Borda Fuse* shows the best performance along all positions.*NPM* obtained the best results for the search engines. For this reason, we compared *Borda Fuse* against *NPM*. For instance, in *Hits*, *Borda Fuse* outperforms *NPM* by 19% (on average) for most positions. This means that around position 5 *Borda Fuse* returns more relevant packages than *NPM*.

Regarding the ranking quality, *Borda Fuse* obtained *nDCG* values of 22.04%, 20.5% and 25.9% higher than the same metric for *NPM*, for ranking lengths of 5, 10 and 20, respectively. When it comes to *MAP*, the *Borda Fuse* gains with respect to *NPM* were of 27.37%, 25.16%, and 21.78% for the same ranking lengths. The better ranking quality is related to precision and recall improvements. On one hand, by increasing precision in the top-ranked positions, the relevant results near the top of the ranking increase and so do the nDCG and MAP values (since both metrics reward results in the top-ranked positions). On the other hand, by increasing recall, the amount of relevant results that contribute to the nDCG and MAP calculations also increases.

Observing that the performance metrics of *ST-Retrieval* were higher than those of the (individual) search engines, we tested the statistical significance of the results using the non-parametric Wilcoxon Signed Rank Test [34] with a significance level $\alpha = 0.05$. We stated the null hypothesis ($H1_0$) as: "The metric values from *ST-Retrieval* are equal to those from the search engines". The alternative hypothesis ($H1_1$) states that there is a difference between these metrics. Table IV shows the p-values obtained for each metric (treatment) in Table III after running the tests. Except for the number of hits at the first position (H@1), all p-value values were less than 0.05, and thus, we can reject $H1_0$ for all the metrics and conclude that the differences are statistically significant. Finally, we successfully answer RQ#1 by saying that *ST-Retrieval* improves the search for JS packages.

B. ST-Rank Experiments

The goal is to assess the quality of the results given by *ST-Rank* using the package rankings from NPM as the baseline.

1) Experimental Design: We followed the steps defined in Figure 1 (right side). To obtain a set of "popular" projects, we took the top-1000 most popular projects, according to GitHub, from the 2170 packages used in the *ST-Retrieval* experiment. Furthermore, we relied on NPM and NPMCompare for getting features and alternatives for each package. A dataset with ≈ 250 *training rankings* was created, each one having between 2-6 packages. In total, more than 1000 *training instances* were obtained[18].

To assess the rankings produced by *ST-Rank*, we split them into a training and a test sets with the usual $80-20\%$ partition rule of ML tasks. For the test set, we randomly selected a 20% of the *training rankings* (and their corresponding *training instances*). Then, we presented the packages of each ranking to two senior developers and asked them to sort the packages in descending order of relevance, so as to establish *reference rankings* [19]. The remaining 80% of the *training instances* constituted the training set for building the ML model. We performed a k-fold cross-validation ($k = 5$) [35] and determined the best configuration of hyper-parameters for GBRank. A combination of random and grid search was applied to explore hyper-parameter values [36], [37].

For the application contexts, we departed from the three execution environments introduced in Section III and identified five alternative scenarios, as follows: (i) *All* - the developer makes no distinction among application contexts (i.e. the three environments are considered), (ii) *Web* - the developer needs a technology for a Web browser (i.e. *Web* and *No context*

[18]The training dataset can be found in the Supplementary Material zip file at https://bit.ly/2w9sOzV.

[19]The reference rankings can be found in the Supplementary Material zip file at https://bit.ly/2w9sOzV.

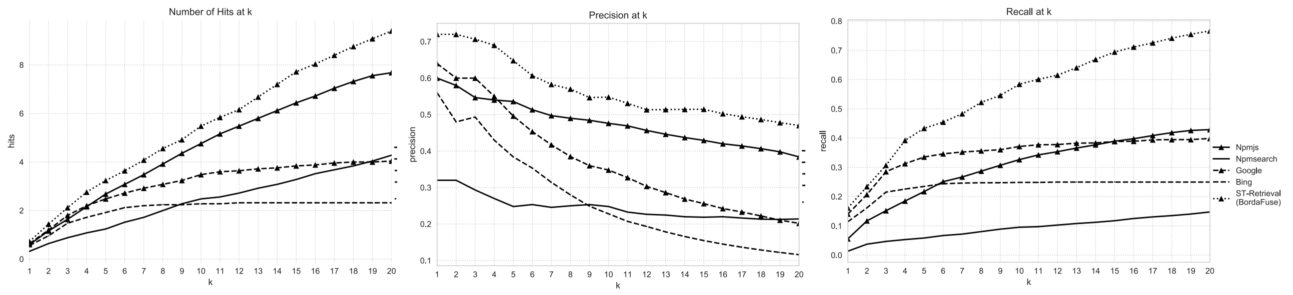


Fig. 5. Hits, Precision, and Recall at k (*ST-Retrieval*).

TABLE III
METRICS FOR SEARCH ENGINES (*ST-Retrieval*).

Method/Metric	H@1	H@5	H@10	H@20	P@5	P@10	P@20	nD(r=5)	nD(r=10)	nD(r=20)	M(r=5)	M(r=10)	M(r=20)
NPM	0.600	2.680	4.760	7.680	0.536	0.476	0.384	0.635	0.639	0.660	0.632	0.620	0.615
NPMSearch	0.320	1.240	2.480	4.280	0.248	0.248	0.214	0.322	0.333	0.361	0.348	0.336	0.325
Google	0.640	2.480	3.480	4.040	0.496	0.348	0.202	0.720	0.736	0.750	0.712	0.704	0.695
Bing	0.560	1.920	2.280	2.320	0.384	0.228	0.116	0.662	0.675	0.677	0.638	0.639	0.639
BordaFuse	**0.720**	**3.240**	**5.480**	**9.400**	**0.648**	**0.548**	**0.470**	**0.775**	**0.770**	**0.831**	**0.805**	**0.776**	**0.749**

TABLE IV
COMPARISON OF THE RETRIEVAL METRICS USING WILCOXON SIGNED RANK TEST (*ST-Retrieval*).

ST-Retrieval vs	H@1	H@5	H@10	H@20	P@5	P@10	P@20	nD(r=5)	nD(r=10)	nD(r=20)	M(r=5)	M(r=10)	M(r=20)
NPM	.317	<.001	<.001	<.001	<.001	<.001	<.001	<.05	<.05	<.05	<.05	<.001	<.001
NPMSearch	<.05	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001
Google	.317	<.001	<.001	<.001	<.001	<.001	<.001	<.05	<.05	<.05	<.001	<.001	<.001
Bing	.248	<.001	<.001	<.001	<.001	<.001	<.001	<.05	<.001	<.001	<.001	<.001	<.001

are considered), (iii) *Node* - the developer needs a technology for Node.js (*Node* and *No context* are considered), (iv) *OnlyWeb* - the developer needs a technology being specifically developed for the Web browser (only *Web* is considered), and (v) *OnlyNode* - the developer needs a technology being specifically developed for Node.js (only *Node* is considered). Based on these scenarios, we expect *ST-Rank* to classify the input packages and the *training rankings*. Along this line, we ran *ST-Rank* five times, once for each scenario. At last, we compared our results with the default ranking techniques supported by NPM and NPMCompare, which are the AHP [3] and WAM (Weighted Average Method) [38], respectively.

2) Metrics: To evaluate the rankings, metrics such as MAP, SRCC (Spearman Rank Correlation Coefficient) [39], and MRR (Mean Reciprocal Rank) [40] were used. SRCC measures the correlation between the rankings created by *ST-Rank* (and also by AHP and WAM) against the reference rankings established by the senior JS developers. In particular, SRCC was computed for each ranking and the values were averaged to obtain a summary value per technique. MRR, in turn, assesses whether the highest-ranked items are relevant. The closer the MRR value is to 1, the larger the number of (relevant) packages that make it to the highest positions.

3) Analysis of Results: Table V shows the metric values for the five scenarios. In the case of MAP, it considers different ranking lengths (e.g., M@3 and M@5 mean that the MAP computation takes a ranking of 3 and 5 packages, respectively). The values in bold in the table highlight the best result for the metric along that column. As can be seen, GBRank achieves the best results for all the metrics. For example, when considering MAP in the *All* scenario, GBRank outperforms AHP and WAM by around 44% and 12%, respectively. For SRCC, GBRank improves the values of AHP and WAM by around 42% and 17%. When looking at MRR, it shows smaller improvements for GBRank, ranging from 28% to 6% for AHP and WAM, respectively. The differences between the techniques are similar in the other scenarios. On average *ST-Rank* seems to outperform AHP and WAM by a 10%. The smallest margin was noticed for MRR, as expected. MRR only takes into account the first element of the ranking. That is, if the first element of the reference ranking is in the first position of the ranking under evaluation, this ranking would have a maximum MRR value (despite the other elements being disordered). However, in our package ranking problem, it is important that the other elements are also ordered. If for some reason (e.g., due to a technical limitation) the first package cannot be used by the developer, then the rest of the packages should be as orderly as possible. This situation might happen if the developer needs to create a proof-of-concept, and she needs to find the right package quickly.

To check if the results are statistically significant, we tested them using the non-parametric Wilcoxon Signed Rank Test with a significance level $\alpha = 0.05$. We stated the null hypothesis ($H2_0$) as: "The metric values from *ST-Rank* are

TABLE V
RANKING RESULTS FOR DIFFERENT SCENARIOS (*ST-Rank*).

	All				Web				Node				OnlyWeb				OnlyNode			
Technique	M@3	M@5	SRCC	MRR	M@3	M@5	SRCC	MRR	M@3	M@5	SRCC	MRR	M@3	M@5	SRCC	MRR	M@3	M@5	SRCC	MRR
AHP	0.517	0.541	0.454	0.654	0.756	0.762	0.731	0.865	0.541	0.561	0.365	0.667	0.785	0.725	0.621	0.880	0.461	0.478	0.310	0.672
WAM	0.813	0.798	0.656	0.863	0.744	0.729	0.669	0.836	0.793	0.806	0.668	0.869	0.928	0.928	0.878	0.952	0.822	0.796	0.750	0.883
GBRank	**0.925**	**0.914**	**0.788**	**0.915**	**0.910**	**0.864**	**0.835**	**0.942**	**0.874**	**0.889**	**0.886**	**0.932**	**0.952**	**0.962**	**0.950**	**0.964**	**0.933**	**0.909**	**0.907**	**0.966**

TABLE VI
RESULTS OF WILCOXON SIGNED RANK TEST (*ST-Rank*).

ST-Rank vs	M@3	M@5	SRCC	MRR
AHP	$4.41e^{-12}$	$1.31e^{-13}$	$2.35e^{-16}$	$4.27e^{-12}$
WAM	$5.52e^{-05}$	$1.01e^{-06}$	$1.67e^{-10}$	$5.79e^{-04}$

equal to those provided by the other techniques (i.e. AHP and WAM)". The alternative hypothesis ($H2_1$) states that there is a difference between these metrics. Table VI shows the p-values for each metric reported in Table V, after running the tests. Since all p-value values turned out to be less than 0.05, we can reject $H2_0$ for all the metrics. Thus, we can conclude that the differences between *ST-Rank* and AHP and WAM are statistically significant. In summary, we successfully answer RQ#2 by saying that the rankings proposed by *ST-Rank* have a different than those generated by the existing engines.

C. Threats to Validity

In spite of the satisfactory evaluation results, some threats to validity should be considered. The first threat, to construct validity, has to do with the queries (reference set) and technology searches (baseline) used in the experiments. We relied both on queries and search criteria that were intended to be representative of real-world JS development. To this end, we collected a dataset from the NPM repository using the JS package registry. In addition, we followed best practices for evaluating the recovery effectiveness of search engines [29]. Although 25 queries were used, they returned 2760 JS packages that were subject to manual analysis. Since analyzing query results from the search engines might take a substantial amount of time from experts, we preferred not to do a detailed query analysis in this work. In addition, the senior developers providing the queries and checking the rankings might have been biased by the type of software projects they usually work on.

A threat to internal validity is the usage of *Borda Fuse* to order the lists of packages in *ST-Retrieval*, as this method might have biased those results, and might have affected the outputs of *ST-Rank* as well. Applying alternative aggregation methods (e.g., Reciprocal Rank, Condorcet) could have generated different package orderings. Analogously, for *ST-Rank* we used GBRank as the learning-to-rank algorithm), but using other algorithms (e.g., RankNet, RankBoost) could have led to different results. Another threat is that the training rankings for *ST-Rank* depend on the CDSel metric, which we proposed as a proxy for scoring the JS packages. Nonetheless, CDSel might not fully capture the notion of package relevance/popularity for the JS community. Thus, work towards validating this metric should be pursued.

To mitigate threats to external validity, we considered queries with different sizes, purposes, and domains in the experiments. However, our dataset is still small and might not be representative of all kinds of JS projects. Additional experimentation and surveys with JS developers are still necessary. Furthermore, we noticed that running the NER rules (in *ST-Retrieval*) and training the ML model (in *ST-Retrieval*) can be both computationally expensive. Since the evaluation was conducted in an offline mode, the computation time taken by the recommender system was not a key factor. However, this performance aspect should be revisited if online search scenarios are considered.

V. RELATED WORK

Several approaches have been developed to support the selection of software technologies [13]. In general, these approaches are based on creating a list of technologies that are compared and presented to developers, so they can decide which ones to select for their projects. Some works have focused on the evaluation of a set of pre-established technologies and do not deal with the problem of searching/retrieving them from (Web) repositories. For example, given a set of predefined candidate components, Ernst et al. [41] proposes a scorecard to help developers select a given component. The scorecard is based on performance, maintenance, and community criteria.

Software repositories [6] are one of the main sources for search/retrieval of alternative technologies. However, existing repositories have not been very successful in this task, despite improvements in their underlying technology, such as the Web [6]. One of the reasons is the performance of the search engines, which sometimes fails to produce the desired results. Several works have tried to improve the search mechanisms offered by repositories. To our knowledge, no works about meta-search targeting software technologies have been proposed. However, there are a few works that bear similarities with our approach. Agora [42] is a research prototype that intended to create a global database of JavaBeans and CORBA components, which was automatically generated and indexed. However, a major drawback was its reliance on the syntactic interface of JavaBeans and CORBA for its search function. In addition, Agora did not use search engines for software repositories. Another related work is Dolphin [43], which indexes open-source projects from version control repositories and ranks them according to their influence in discussions of forum communities (e.g., StackOverflow, OSChina). The main difference with our work is that Dolphin searches open-source code from version control repositories, while *ST-Retrieval* and *ST-Rank* focus on software released and stored in repositories (e.g., NPM). Another difference is that Dolphin does not include general-purpose search engines.

LibFinder [44] is a search-based recommendation system for Java that uses multi-objective optimization to recommend libraries taken from GitHub and Maven repositories. However, it does not allow user queries to find technologies. Instead, it is based on the source code, and the recommended libraries are oriented to replace specific pieces of code. LibFinder does not use general-purpose search engines. Soliman et al. [45] developed an approach to retrieve architecture design decisions and solution alternatives. The approach uses StackOverflow as an online repository of architecture knowledge. This approach is based on a mapping between text (i.e., queries) and a "de-facto" ontology, and its applicability to other contexts (like the JS domain) is yet to be determined.

In [7], the authors discuss a recommendation approach that is based on a knowledge base (i.e., a repository) mined from curated Web resources (e.g., Q&A posts like StackOverflow). Like in *ST-Retrieval*, developers' queries are made in natural language. However, unlike *ST-Rank*, word embeddings are used to find libraries in the knowledge base similar to a given input library. A prototype tool has been implemented [8]. The kinds of information sources considered in this approach can complement the dataset of features for JS packages used in our work. A related approach to search for JS code snippets implementing a given feature is proposed by [9]. However, reusing snippets is not the same task as integrating libraries or packages from a development (or design) perspective.

Regarding the order in which the technologies are created, there are works addressing the problem from different perspectives [10]–[12]. However, most works create ranking strategies manually, based on specific characteristics of the candidates. For example, Franch and Carvallo [46] propose a structured quality model to evaluate software packages. This model provides a taxonomy of software quality characteristics and metrics to calculate its value for a given domain. Jadhav et. al [17] classifies package ranking strategies into two groups, those using AHP and those using WAM. Then, an expert system is applied. This approach is difficult to implement since it depends on experts to construct the rules, as opposed to following a data-driven strategy. Grande et al. [13] define the selection problem as a multi-objective optimization and develop a framework based on a genetic algorithm for the problem. A limitation of this approach is the creation and maintenance of the repository of technologies and their characteristics.

VI. Conclusions and Future Work

In this paper, we proposed an approach for recommending JS packages to developers based on a pipeline with two phases (or components), called *ST-Retrieval* and *ST-Rank*. *ST-Retrieval* helps developers to search and retrieve packages using a meta-search strategy. *ST-Rank* assists developers in ranking the packages identified by the previous phase according to their relevance. To achieve this goal, *ST-Rank* relies on a pairwise learning-to-rank model to infer package rankings from previous choices made by the JS development community in open-source projects.

An initial evaluation of the approach using the NPM repository showed good effectiveness results. In the case of *ST-Retrieval*, precision and recall were around 80%, outperforming other search engines by approximately a 20%. Regarding *ST-Rank*, the improvements in the ranking metrics were between 5-20% depending on the metric being considered (MRR, MAP or SRCC). Despite these results, the approach still presents some drawbacks. First, the technology extraction based on NER rules can be a performance bottleneck for the pipeline, which might limit the application of the current prototype to online package search. To deal with this issue, we will analyze other extraction functions and also explore how they can influence the *ST-Rank* phase. Second, the CDSel metric to capture the degree of selection of a JS package in the community needs further validation and perhaps refinement, in the sense of being able to discriminate good technology decisions. Although the results from the experiments with CDSel were reasonable, considering other parameters in this metric can affect the quality of the rankings.

As future work, in the case of *ST-Retrieval*, we will apply aggregation functions other than Borda Fuse. For *ST-Rank*, we will explore alternative algorithms to GBRank for the learning-to-rank task. In addition, we plan to conduct a user study with JS developers to corroborate our findings and also assess the usefulness of the recommender system. We argue that the approach can incorporate curated lists of JS resources[20] to enrich the current repository, or could be extended with technology repositories for other programming languages (e.g., Ruby, Python or Java, among others) [7]. Furthermore, we would like to include quality-attribute aspects of JS packages (beyond functionality) in the recommendations; however, the current query mechanism might not be sufficient for that purpose. Finally, following ideas of explainable AI (XAI) [47], it would be interesting to support the generation of explanations for the rankings, so that the comparisons (as performed by *ST-Rank*) become easier to understand (and thus, trust) by developers.

References

[1] H.-Y. Lin, P.-Y. Hsu, and G.-J. Sheen, "A fuzzy-based decision-making procedure for data warehouse system selection," *Expert systems with applications*, vol. 32, no. 3, pp. 939–953, 2007.

[2] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 351–361.

[3] T. L. Saaty, "Decision making with the analytic hierarchy process," *International journal of services sciences*, vol. 1, no. 1, pp. 83–98, 2008.

[4] M. J. Eppler and J. Mengis, "The concept of information overload: A review of literature from organization science, accounting, marketing, mis, and related disciplines," *The information society*, vol. 20, no. 5, pp. 325–344, 2004.

[20]For instance, https://github.com/sorrycc/awesome-javascript

[5] H. Berghel, "Cyberspace 2000: Dealing with information overload," *Communications of the ACM*, vol. 40, no. 2, pp. 19–24, 1997.

[6] N. Clayton, R. Biddle, and E. Tempero, "A study of usability of web-based software repositories," in *Proceedings International Conference on Software Methods and Tools. SMT 2000*, 2000, pp. 51–58.

[7] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in qa discussions – incorporating relational and categorical knowledge into word embedding," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 338–348.

[8] C. Chen, Z. Xing, and L. Han, "Techland: Assisting technology landscape inquiries with insights from stack overflow," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 356–366.

[9] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 690–701.

[10] A. Birk, "Modelling the application domains of software engineering technologies," in *Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference*. IEEE, 1997, pp. 291–292.

[11] V. R. Basili and H. D. Rombach, "Support for comprehensive reuse," *Software engineering journal*, vol. 6, no. 5, pp. 303–316, 1991.

[12] J. Klein and I. Gorton, "Design assistant for nosql technology selection," in *Proceedings of the 1st International Workshop on Future of Software Architecture Design Assistants*. ACM, 2015, pp. 7–12.

[13] A. D. S. Grande, R. D. F. Rodrigues, and A. C. Dias-Neto, "A framework to support the selection of software technologies by search-based strategy," in *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*. IEEE, 2014, pp. 979–983.

[14] S. Vegas and V. Basili, "A characterisation schema for software testing techniques," *Empirical Software Engineering*, vol. 10, no. 4, pp. 437–466, 2005.

[15] J. A. Aslam and M. Montague, "Models for metasearch," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2001, pp. 276–284.

[16] H. Li, "Learning to rank for information retrieval and natural language processing," *Synthesis Lectures on Human Language Technologies*, vol. 4, no. 1, pp. 1–113, 2011.

[17] A. S. Jadhav and R. M. Sonar, "Framework for evaluation and selection of the software packages: A hybrid knowledge based system approach," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1394–1407, 2011.

[18] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[19] G. Navarro, "A guided tour to approximate string matching," *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

[20] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 613–622.

[21] T.-Y. Liu, *Learning to rank for information retrieval*. Springer Science & Business Media, 2011.

[22] Y. K. Jain and S. K. Bhandare, "Min max normalization based data perturbation method for privacy protection," *International Journal of Computer & Communication Technology*, vol. 2, no. 8, pp. 45–50, 2011.

[23] P. Li, Q. Wu, and C. J. Burges, "Mcrank: Learning to rank using multiple classification and gradient boosting," in *Advances in neural information processing systems*, 2008, pp. 897–904.

[24] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun, "A general boosting method and its application to learning ranking functions for web search," in *Advances in neural information processing systems*, 2008, pp. 1697–1704.

[25] J. Bai, F. Diaz, Y. Chang, Z. Zheng, and K. Chen, "Cross-market model adaptation with pairwise preference data for web search ranking," in *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*. Association for Computational Linguistics, 2010, pp. 18–26.

[26] T. Kanungo, N. Ghamrawi, K. Y. Kim, and L. Wai, "Web search result summarization: title selection algorithms and user satisfaction," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 1581–1584.

[27] J. Bian, Y. Liu, E. Agichtein, and H. Zha, "Finding the right facts in the crowd: factoid question answering over social media," in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 467–476.

[28] M. Taghavi, A. Patel, N. Schmidt, C. Wills, and Y. Tew, "An analysis of web proxy logs with query distribution pattern approach for search engines," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 162–170, 2012.

[29] D. Lewandowski, "Evaluating the retrieval effectiveness of web search engines using a representative query sample," *Journal of the Association for Information Science and Technology*, vol. 66, no. 9, pp. 1763–1775, 2015.

[30] N. Höchstötter and D. Lewandowski, "What users see–structures in search engine results pages," *Information Sciences*, vol. 179, no. 12, pp. 1796–1812, 2009.

[31] S. Büttcher, C. L. Clarke, and G. V. Cormack, *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[32] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 161–168.

[33] K. Järvelin and J. Kekäläinen, "Ir evaluation methods for retrieving highly relevant documents," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2000, pp. 41–48.

[34] V. Lavrenko and W. B. Croft, "Relevance-based language models," in *ACM SIGIR Forum*, vol. 51, no. 2. ACM, 2017, pp. 260–267.

[35] P. Refaeilzadeh, L. Tang, and H. Liu, "Cross-validation," *Encyclopedia of database systems*, pp. 532–538, 2009.

[36] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.

[37] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[38] M. Perez and T. Rojas, "Evaluation of workflow-type software products: a case study," *Information and software technology*, vol. 42, no. 7, pp. 489–503, 2000.

[39] D. Zwillinger and S. Kokoska, *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.

[40] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan, "Expected reciprocal rank for graded relevance," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 621–630.

[41] N. Ernst, R. Kazman, and P. Bianco, "Component comparison, evaluation, and selection: A continuous approach," in *International Conference on Software Architecture Workshops*. IEEE, 2019.

[42] R. C. Seacord, S. A. Hissam, and K. C. Wallnau, "Agora: A search engine for software components," *IEEE Internet computing*, vol. 2, no. 6, p. 62, 1998.

[43] Y. Zhan, G. Yin, T. Wang, C. Yang, Z. Li, and H. Wang, "Dolphin: A search engine for oss based on crowd discussions across communities," in *Software Engineering and Service Science (ICSESS), 2016 7th IEEE International Conference on*. IEEE, 2016, pp. 599–605.

[44] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.

[45] M. Soliman, M. Galster, and M. Riebisch, "Developing an ontology for architecture knowledge from developer communities," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 89–92.

[46] X. Franch and J. P. Carvallo, "A quality-model-based approach for describing and evaluating software packages," in *Proceedings IEEE Joint International Conference on Requirements Engineering*. IEEE, 2002, pp. 104–111.

[47] Y. Zhang and X. Chen, "Explainable recommendation: A survey and new perspectives," *Found. Trends Inf. Retr.*, vol. 14, no. 1, p. 1–101, mar 2020.