Smart Issue Retrieval Application

Jernej Zupančič
jernej.zupancic@ijs.si
Jožef Stefan Institute
Jamova cesta 39
Ljubljana, Slovenia
Jožef Stefan International
Postgraduate School
Ljubljana, Slovenia

Borut Budna
borut.budna@ijs.si
Faculty of Computer and
Information Science
Ljubljana, Slovenia

Miha Mlakar
Maj Smerkol
miha.mlakar@ijs.si
maj.smerkol@ijs.si
Jožef Stefan Institute
Jamova cesta 39
Ljubljana, Slovenia

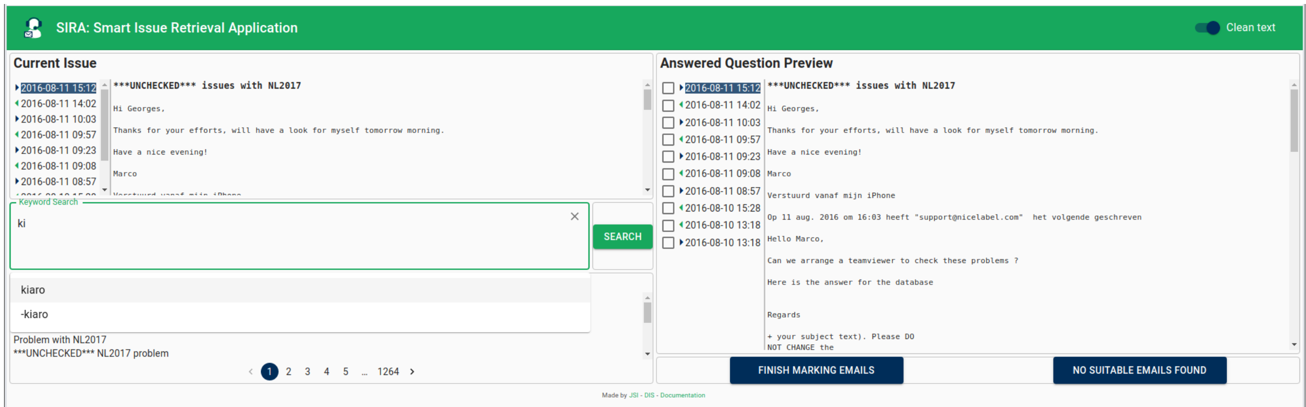

Figure 1: SIRA screenshot

ABSTRACT

We present Smart Issue Retrieval Application (SIRA), a customer support tool for searching of relevant email threads or issues when an email thread and keywords are given. Presented are the overall application architecture, the processing pipeline, which transforms the data into a search friendly form, and the search algorithm itself.

KEYWORDS

customer support, language models, information retrieval

1 INTRODUCTION

Customer support is an important part of many large businesses and high quality customer support can improve the user experience and help businesses retain their customer for longer periods. For larger companies, it can also be a strain on their human resources as many customer support issues need to be resolved in short time. While the customer support team may resolve most issues on their own sometimes they need the help of the development department. Often similar issues are presented to the developers multiple times.

In order to minimize the number of issues that need attention from other departments, we have developed an application to help the customer support technicians resolve issues without help from developers. While some issues will still need the attention of developers, SIRA can help find existing answers to questions that have already been resolved by developers and therefore reduce the amount of distractions for the development team.

We use language models in order to retrieve information about the question from the issue at hand. Using multiple different approaches, application searches the database of resolved issues in order to find a developers' answers to same or similar questions.

2 SIRA ARCHITECTURE

SIRA comprises five main application components (Fig. 2):

(1) *Database.* PostgreSQL [6] is used as the application database, since it includes decent built-in text search capabilities and change data capture options.
(2) *Processing daemon.* Python [7] process responsible for data processing for search in the event of change data capture.
(3) *Back-end application.* Python Flask-based back-end application exposing the application programming interface for SIRA.
(4) *Front-end application.* React-based [8] single-page application for interacting with SIRA.
(5) *Documentation.* MKdocs-based user documentation for final users, admins, and developers.

Each SIRA component is packaged within a Docker [4] image and can be managed using "`docker-compose`" [2] tool. This enables deterministic packaging of application code for development, testing and production.

3 SIRA FUNCTIONALITY

The main goal of SIRA is to enable customer support staff to quickly find answers to similar questions that have already been resolved in the past. Search is therefore the primary functionality of the application and can be split into three parts:

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
Information society '20, October 5–9, 2020, Ljubljana, Slovenia
© 2020 Copyright held by the owner/author(s).

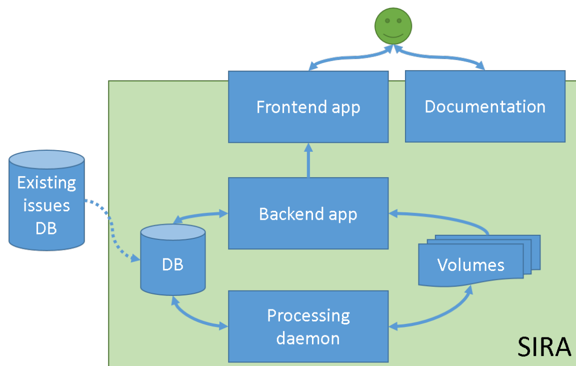

Figure 2: SIRA architecture overview

(1) *Processing.* Upon new data arrival, pre-processes the text to obtain representation suitable for search.
(2) *Search.* Computing relevancy scores upon search request by taking into account as much information about issue or email thread as possible.
(3) *Logging.* To improve the search in the future the search results and structured user feedback is gathered and stored.

In the rest of this section we will describe each part in more details.

3.1 Processing

For the search to be efficient it is beneficial to pre-process the raw emails. The processing daemon runs as a separate python process and utilizes PostgreSQL's logical replication functionality in order to transform new content as soon as it is written to the database. The following steps are executed when processing the issues:

(1) *HTML clean.* Beautiful Soup [1] library is used to extract only relevant text from email XML markup.
(2) *Empty line removal.* Python script is used to detect and remove empty lines.
(3) *Repeated emails removal.* Parts of emails are deleted if they already appear within some previous mails of the same issue.
(4) *Semi-structured emails handling.* Some emails are actually a filled out form in an email format. A python script is used to extract only the relevant information.
(5) *Non-author lines removal.* A machine learning model was developed and is deployed for tackling this task.
(6) *Non-alphanumeric-only characters lines removal.* Python script is used to detect and remove those lines.
(7) *Word vector representation computation and update.* FastText [3] word vectors are used to compute word vector representation of text.
(8) *Storing of processed text.* The processed text is stored into database, where built-in database indexing is utilized to further prepare the text for efficient text searching.

In the rest of this section we focus on the non-trivial processing steps.

3.1.1 Repeated emails removal. There were two reasons for removing repeated emails from an email thread. First, when displaying an email, usually also all the previous emails are included, which results in poor readability. Second, some methods for comparing the text take into account the number of occurrences of a particular word. This is sensible for cases when the word actually repeats in the content. However, if it repeats due to the text duplication it could negatively impact the search results.

We define a repeated email as an email body that appears within another email body. This is usually a result of using a "Reply" functionality when responding to an email within an email client.

To delete email A from email B, the following method is used:

(1) Extract only alphanumeric characters from the two email bodies A and B to get alphanumeric(A) and alphanumeric(B).
(2) If alphanumeric(A) appears within alphanumeric(B), mark it for removal from alphanumeric(B).
(3) If alphanumeric(A) does not appear within alphanumeric(B), iterate over substrings of alphanumeric(B) and compute the matching percentage of consecutive alphanumeric blocks from alphanumeric(A). The substring with the maximum match is a candidate for removal. If it exceeds a predefined threshold it is indeed marked for removal from alphanumeric(B).
(4) Reconstruct B by dropping the substring marked for removal and all non-alphanumeric characters positioned within the marked substring when expanded with all the characters.

3.1.2 Non-author lines removal. An email body usually comprises:

(1) Relevant content
(2) Signature
(3) Confidentiality notice
(4) Previous email headers
(5) Previous email content

The only text that should be used for text comparison is the relevant content part. While previous email content was mostly removed in the repeated emails removal step (3.1.1), other email body parts can still impact text comparison results. Machine learning was utilized to develop a model for determining whether a particular line in the email body belongs to the relevant content part of an email or not.

Dataset preparation. First, we implemented an application with a basic graphical user interface that enabled us to label each line with one of the following categories:

(1) AUTHOR. The relevant content falls into this category.
(2) QUOTED. This is the previous email content.
(3) AUTO-PERSONALIZED. This is the text, that was set by a user in the email client, which is automatically inserted by the email client. Signature is an example of this.
(4) AUTO-NON-PERSONALIZED. This is the text inserted by the email client automatically. An example of this is previous email headers.
(5) NEEDS-PRETTIFY. Sometimes the whole email body is present in one line only. To properly label the body it should be further split into multiple lines.
(6) OTHER. Everything else.

Second, we labeled each line belonging to 100 random issues. This way we generated a dataset of 37,421 labeled lines in 586 emails. Since the assumption was that the "QUOTED" lines are already filtered out using remove repeated emails method, we omit those lines from the dataset. This left us with 9,848 labeled lines.

Features. The computed features were of two types: local features that took into account just the current line, and global

features that took into account the relative position and content of a line within the whole email.

Local features:

(1) Number and proportion of capitalized words
(2) Number and proportion of non-alphanumeric characters
(3) Number and proportion of numeric characters
(4) "CountVectorizer" from the scikit-learn ([5]) package
(5) "TfidfVectorizer" from the scikit-learn package
(6) Word vector line representation

Global features:

(1) Line position from the start
(2) Line position until the end
(3) Does "regard" appear before this line, within this line, after this line
(4) Do four or more consecutive non-alphanumeric characters appear before this line, within this line, after line
(5) Does a date-like string appear before this line, within this line, after this line
(6) Does a time-like string appear before this line, within this line, after this line

In order to smooth the predictions we also tested hierarchical modeling by first building a model for "AUTHOR" detection and then using the predictions on the lower level as additional features for the higher level. One approach for using the predictions from the lower level was to just use the "AUTHOR" predictions of lines just before and just after the current line. The predictions were padded with 1 at the beginning of an email and with 0 at the end. The second approach was based on the sum of three consecutive "AUTHOR" class probabilities for: lines, just before the current line, lines where the current line is in the middle, and lines just after the current line. We padded the predictions with 1s at the beginning of an email and with 0s at the end.

Further, the features were scaled using the StandardScaler and the feature space dimensionality was reduced using the principal component analysis - PCA, both from the scikit-learn package.

Models. For modeling we utilized scikit-learn package and tested the following algorithms: (1) Logistic regression, (2) Multinomial Naive Bayes, (3) Support vector machine, (4) Random forest classifier.

Rudimentary hyper-parameter tuning was done to pick the best ones.

Evaluation. Each pipeline was evaluated using 10-fold cross validation with the splits over issues. This means that all the lines belonging to one issue were either in the training or the testing set to prevent data leaking.

Model selection. The performance of all models was tracked through various metrics:

(1) Confusion matrix
(2) Precision and recall at different minimum recall thresholds
(3) Precision-recall curve
(4) "AUTHOR" probabilities for each line in the test set

The main concern regarding the model performance was that it should prioritize keeping the "AUTHOR" lines ("AUTHOR" recall) over average model accuracy. This is a direct result of the application architecture – if the line would be removed by the chosen model, it wouldn't be possible to search over it. This would directly impact the performance in the real-world. Additionally, few additional lines shouldn't hinder the readability too much.

The gathered metrics enabled us to closely inspect each model and overview the performance regarding real-world application. A basic GUI was built to inspect the models and overview the miss-classified examples. In the end, the hierarchical model was chosen with most of the presented features, with the exception of "CountVectorizer" and "TfidfVectorizer" features. The additional chosen higher-level feature was the sum of three consecutive "AUTHOR" probabilities. Random forest was chosen as the classification algorithm, without feature standardization or dimensionality reduction step. The threshold probability was lowered to 0.12 so recall could be kept high.

The final model miss-classified 59 out of 2,394 rows marked as "AUTHOR" (recall = 0.975) and 629 out of 7,454 rows marked as "OTHER" (recall = 0.806).

3.1.3 Word vector representation computation and update. Word vector representation of content is used to compare email bodies and email subjects between different issues.

To compute the word vector representation of text, either issue body or issue subject, the following steps are executed: (1) Tokenize text, (2) Remove stop-words, (3) Query word vector representation for each word using fastText common crawl word vectors with dimension 300, (4) Compute mean of all word vectors belonging to the words in the text, (5) Normalize the mean vector by dividing the mean vector by the mean vector length.

Instead of generating the representation vectors on-the-fly, they are pre-computed and only read when needed, which greatly reduces the inference time. To update word vector representation of a particular text, the corresponding row in the word vector matrix is updated with the new values and stored on disk as a Numpy array.

3.2 Search

Each issue consists of: subject, document (the email body of text), and keywords the user marked the issues with. The keywords can be positive, meaning that a keyword is related with the contents of the issue, or negative when keyword is *not* related with the contents of the particular issue. Additionally, a keyword can be explicit, where a user uses the keyword for searching when considering a particular issue. On the other hand, a keyword can be implicit – soft keywords, where the user searched for relevant issues using a keyword, but the search results were not marked as relevant.

When computing the relevancy of issues, given a starting issue and some keywords, several relevancy sub-scores are first computed and then aggregated to form a single relevancy score. In Table 1 all combinations for relevance sub-scores are listed.

The final score is computed as a weighted average, as in equation 1. The weights w_i were determined based on the final user feedback.

$$\begin{aligned} \text{finalScore} = & \ w_1 \cdot \text{KeywordToKeywordScore} \\ & + w_2 \cdot \text{KeywordToSoftKeywordScore} \\ & + w_3 \cdot \text{KeywordToDocumentScore} \\ & + w_4 \cdot \text{KeywordToSubjectdScore} \\ & + w_5 \cdot \text{DocumentToKeywordScore} \\ & + w_6 \cdot \text{DocumentToSoftKeywordScore} \\ & + w_7 \cdot \text{DocumentToDocumentScore} \\ & + w_8 \cdot \text{SubjectToKeywordScore} \\ & + w_9 \cdot \text{SubjectToSoftKeywordScore} \\ & + w_{10} \cdot \text{SubjectToSubjectScore} \end{aligned} \tag{1}$$

Table 1: Relevance sub-scores matrix

		Other issues			
		(Not) Keyword	Soft (Not-) keyword	Document	Subject
Current issue	(Not-) Keyword	Exact match	Exact match	Full-text search	Full-text search
	Soft (Not) Keyword	/	/	/	/
	Document	Reverse full-text search	Reverse full-text search	Word vector cosine similarity	/
	Subject	Reverse full-text search	Reverse full-text search	/	Word vector cosine similarity

3.2.1 Exact match. This relevance score compares (soft) keywords related to issues and those inserted in the keyword input box. Given a (soft) keyword, search for all the documents that are in relation to this exact (soft) keyword. Each relation can either be positive or negative. Therefore, the returned score is positive in case of positive relation and negative otherwise.

3.2.2 Full-text search. This relevance score compares keywords entered in the keyword input box and issue documents or issue subjects. Full-text search capability of PostgreSQL is leveraged for this score. However, the results are modified to return negative scores in case of not-keyword match.

3.2.3 Reverse full-text search. This relevance score compares the selected issue document or subject and all existing (soft) keywords. First, for each keyword a full-text search relevance score is computed. Second, for each issue in the database do a sum of its related keyword relevance scores.

3.2.4 Word vector cosine similarity. This relevance score compares the selected issue document and subject to all existing issue documents and subjects, respectively. Pre-computed word vectors as described in Section 3.1.3 are used. The relevance score is computed as:

$$\text{wordVectorSimilarity}(T_1, T_2) = 1 - T_1 \cdot T_2. \qquad (2)$$

Since the word vectors used are normalized, this is actually 1− cosine distance between T_1 and T_2.

Two other methods for comparing the text were also tested: PostgreSQL built-in trigram text similarity, which was too slow for production use, and tf-idf representation of text and cosine distance-based relevance score, which did not perform as well as the word vectors method.

3.3 Logging

To improve the search performance in the future, several interactions with the application are logged:

(1) Search results with relevance scores
(2) Viewed search results
(3) Relevant issue/belonging email found
(4) No relevant issue/belonging email found

Only after sufficient real-world usage of the application we can quantitatively evaluate the performance of the whole search pipeline and act upon the results.

4 DISCUSSION AND CONCLUSION

The SIRA system was developed and deployed, including five docker-image packaged modules. The main functionalities of the first major release include preprocessing of the text of the issue, search integrating four different search algorithms and a logging system that stores interactions with the system into the database, including user defined keywords and appropriate results marking.

Preprocessing is done without any user interaction and involves multiple algorithms and AI methods to extract the text of the issue from original encoded emails. Testing of the algorithms shows good results both in terms of precision and recall. Word vector representations are pre-computed in order to improve performance of search algorithms.

Based on the extracted plain text of the issue the application searches for similar issues that have already been resolved. The users can therefore quickly find the information related to the issue. The system is currently in use and only after some time of real-world usage we will be able to evaluate the whole system.

Due to logging the interactions in the database we expect to be able to analyze the usage and quality of the results. This will allow us to improve the system and add other functionality that will improve user experience and further improve the customer support technicians' workflow.

ACKNOWLEDGMENTS

Nicelabel d.o.o. funded the research presented in this paper. We thank Gregor Grasselli, Zdenko Vuk and Miha Štravs for help in application development.

REFERENCES

[1] Beautiful Soup Developers. 2019. Beautiful soup. https://www.crummy.com/software/BeautifulSoup/. (2019).

[2] Docker Inc. 2019. Docker-compose. https://docs.docker.com/compose/. (2019).

[3] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.

[4] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014, 239, 2.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

[6] PostgreSQL Global Development Group. 2019. PostgreSQL, version 12. http://www.postgresql.org. (2019).

[7] Python Software Foundation. 2018. Python language reference, version 3.7. http://www.python.org. (2018).

[8] React Developers. 2019. React. https://reactjs.org/. (2019).