Leveraging Machine Learning for Software Redocumentation

Verena Geist
Software Analytics and Evolution
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
verena.geist@scch.at

Michael Moser
Software Analytics and Evolution
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
michael.moser@scch.at

Josef Pichler
Department of Software Engineering
University of Applied Sciences Upper Austria
Hagenberg, Austria
josef.pichler@fh-hagenberg.at

Stefanie Beyer
Software Engineering Research Group
Alpen-Adria-Universität Klagenfurt
Klagenfurt, Austria
stefanie.beyer@aau.at

Martin Pinzger
Software Engineering Research Group
Alpen-Adria-Universität Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

***Abstract*—Source code comments contain key information about the underlying software system. Many redocumentation approaches, however, cannot exploit this valuable source of information. This is mainly due to the fact that not all comments have the same goals and target audience and can therefore only be used selectively for redocumentation. Performing a required classification manually, e.g. in the form of heuristic rules, is usually time-consuming and error-prone and strongly dependent on programming languages and guidelines of concrete software systems. By leveraging machine learning, it should be possible to classify comments and thus transfer valuable information from the source code into documentation with less effort but the same quality. We applied different machine learning techniques to a COBOL legacy system and compared the results with industry-strength heuristic classification. As a result, we found that machine learning outperforms the heuristics in number of errors and less effort.**

***Index Terms*—software redocumentation, legacy system, comment classification pipeline, heuristic rules, machine learning, NLP, CNNs**

I. Introduction

Software redocumentation with the intent to recover lost or non-existing documentation is performed for various reasons. Redocumentation may help improving software quality during maintenance [1] or supports migration activities [2]. In the banking domain, as in our case, redocumentation projects are also initiated because of requirements of national financial supervisory authorities [3]. In any case, redocumentation usually concerns large legacy systems where manual redocumentation is not economically feasible.

As the information provided in source code (i.e. code statements and comments) represents a valuable resource for maintainers involved in the management of the evolution of a given software system, existing redocumentation tools (e.g. [2][4][5]) produce acceptable results. In particular, comments are extremely important as they are used to convey the main intent behind design decisions, along with some implementation details [6]. However, most existing approaches neglect source code comments or require intensive annotation in source code to control extraction (e.g. [5]).

To avoid source code annotation but use the information contained in source code comments, we have developed heuristic rules. In an ongoing redocumentation project, we use these heuristics to classify source code comments and use the classification to control whether to extract the content of a source code comment or not. The definition of heuristics is time consuming and the result depends on the underlying programming language and even on the employed programming conventions. To mitigate these problems, we are interested in whether we can replace heuristics by leveraging machine learning for comment classification. Machine learning (ML) methods were already applied for comment classification of C++, Java, and Python software systems with promising results [7][8][9]. We revisit the problem with the aim to answer the following research questions for comment classification in the context of redocumentation of a COBOL legacy software:

1) *RQ1:* How effective are ML classifiers for automatically classifying source code comments?
2) *RQ2:* To what extend can natural language processing (NLP) features and advanced text cleaning techniques improve comment classification?
3) *RQ3:* Which approach (heuristics, ML or deep learning) is the most suitable one in practice?

The paper is structured as follows: In Section II, we outline the industrial context for our research. In Section III, we describe the comment classification heuristic currently used in the redocumentation project as well as investigated machine learning models. In Section IV, we summarize and discuss results. Finally, Section V concludes our work.

II. Industrial Setting

Our redocumentation approach was developed at a financial service provider in the automotive industry. This company developed software systems in the 1980s that are still in

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on June 03,2020 at 07:11:09 UTC from IEEE Xplore. Restrictions apply.

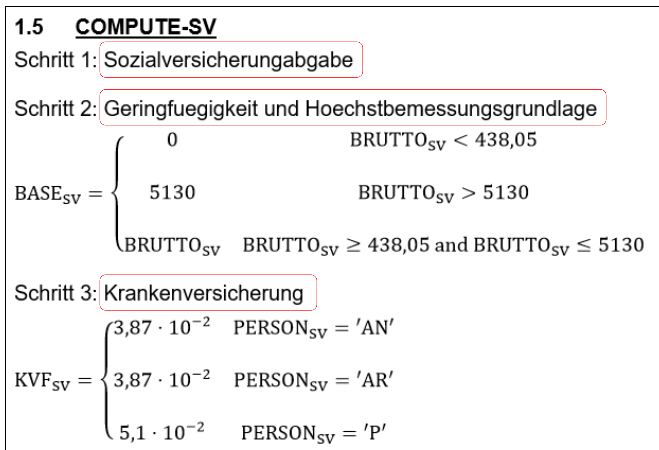

Fig. 1. Extract from generated documentation (text extracted from source code comments is selected).

operation today. These software systems are financial systems in the sense of banking supervision. The operator is therefore obliged to provide national financial supervisors with up-to-date documentation on the software systems.

System under Study. The basis for our research is a COBOL/IMS/DB2 legacy system for leasing with several million lines of code. The legacy system consists of programs for batch processing and online transaction processing (OLTP). Batch programs are executed daily, monthly, or if required. OLTP uses mainframe terminals with screens for user interaction. Online transactions can be started from the main screen of the legacy software, while a single transaction has a main screen and multiple screens for step-by-step data processing.

Documentation Standard. The organization uses a quality guideline for documentation that meets the requirements of the given national regulative [3]. This guideline describes how to create and maintain system documentation at different levels including (i) user documentation and training material, (ii) functional documentation that contains business rules, and (iii) technical documentation including COBOL subsystems, database tables, screen masks, and batch jobs.

Project Goal. Knowing that it is neither possible nor desirable to automate all documentation, the goal of the redocumentation project was to automatically generate parts of the functional documentation and all the technical documentation. High level documentation was created and updated manually, based on interviews with domain experts.

The need for comment classification comes from the generation of business rules that are part of the functional documentation. Fig. 1 shows an excerpt of business rules which contain the computation of domain concepts, e.g. $BASE_{SV}$, together with conditions under which a computation is applied. Business rules are extracted from *source code statements* by means of static code analysis [10] and symbolic execution [11], implemented in our tool *RbG* [5]. To further improve the readability, the documentation is enriched by texts extracted from *source code comments*.

```
152430* WRITE DATA TO DB2-FILE 152440** MOVE LENGTH OF   → IGNORE
152450* ONE MONTH WAS TOO MUCH CALCULATED 152460* DAYS   → TEXT
152470***** DB2-ERRORINFORMATION 152480* -DB2INFO PASS   → TEXT
152500* EVALUATE IN1AX-BOOK-ART                          → IGNORE
```

Fig. 2. An example of classifying comments for software redocumentation.

III. Comment Classification Approaches

A. Problem Statement

Regardless of the different types of source code comments and taxonomies, the problem of comment classification in the industry project described above is formulated as a binary classification problem: the comment is being integrated into the documentation or not.

B. Heuristics

During the redocumentation project, the classification of comments was described by rules. These rules were first created by a manual analysis of a subset of COBOL programs and then continuously adapted and refined during the project. Given a text s of a single comment that spans over one or more lines, we apply rules for inclusion and exclusion in the following order:

1) Block (include): if a multi-line comment starts and ends with the same 20 characters, we include the text s between these two lines. Example:
```
**********************
* Krankenversicherung
**********************
```
2) Error number (include): in the OLTP source code, the validation of user input was documented with error numbers in comments in a systematic way. Example:
```
* F012 some text
```
3) Empty comment (exclude): if the text s is an empty string, we ignore it.
4) Disabled code (exclude): We use black lists with single tokens (e.g. `SECTION.`) and token pairs (e.g. `MOVE`, `TO`) to avoid source code statements in the documentation.
5) Comment length (exclude): we exclude comments with more than x lines to avoid typical header comments including revision information.
6) Default (include): All comments that are not excluded by rules 3–5 are included in the documentation.

C. Machine Learning

We then applied supervised machine learning (ML) [12] for the automatic classification approach. We trained and compared traditional ML classifiers, such as naive Bayes, support vector machines, and random forests, as well as a deep, feed-forward artificial neural network. The proposed *comment classification pipeline* consists of three basic steps, including different ways to improve the performance of the classifiers.

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on June 03,2020 at 07:11:09 UTC from IEEE Xplore. Restrictions apply.

1) Data Preparation: We extracted >700000 source code comments from the COBOL legacy system.

Dataset preparation. To prepare the corpus, we first performed a random sampling over all documents, removed duplicates, shuffled the data, and ensured a balanced dataset regarding TEXT and IGNORE labels. For initial classification, we used the heuristic rules to classify the sampled dataset (see Fig. 2). We then selected a subset of 4010 documents (with respect to confidence and margin of error) and inspected the comments manually. Each source code comment was validated by two professional programmers.

Pre-processing. The first step in pre-processing of comments is (i) *tokenization* (by splitting text into words, ignoring white spaces, etc.). We improved the performance of ML classifiers using **text cleaning techniques**, i.e., by (ii) *removing stop words* and language-specific patterns (e.g., line numbers and special characters in COBOL), (iii) *splitting identifiers based on camel case*, and (iv) *lowercasing* the resulting terms (object standardization). We also apply **NLP techniques** via (v) *word stemming* (using the *nltk* package) to reduce high-dimensional features (lexicon normalization).

Splitting the dataset. For ML classifiers, the dataset is split into 60% training data, 15% validation data, and 25% test data. For deep learning (DL), the split of the dataset in training, validation, and test sets is 56%, 24%, and 20% respectively.

2) Feature Engineering: Traditional ML classifiers require text data to be described by a pre-defined set of features (attributes) prior to employing a learning algorithm.

Creation of new features from text data. First, we used three standard methods in text recognition to create a *word vector* for each instance [13]: (i) *Bag-of-words (count)* vectors count the occurrence of each word in each comment. (ii) *Term frequency-inverse document frequency (tf-idf)* vectors represent the weighted occurrence of frequent terms. (iii) *More-than-one-word (n-gram)* vectors consider sequences of terms that appear next to each other, e.g., *ngram_range=(1,3)*.

Flattening features (feature selection). Then, we reduced the number of features by using thresholds to avoid overfitting, e.g., setting the minimum number of documents a token needs to appear in (e.g., *min_df=3*) or the maximum number of features (e.g., *max_features=2000*).

Combining different feature vectors. Finally, we (i) combine *NLP features* (considering information about the context, e.g., comment length, number of nouns or verbs, etc.) with text feature vectors, and (ii) use the *heuristics* to add a category-specific feature according to our rules.

In Fig. 3, we visualize the distribution of all textual and symbolic strings (data points) included in the final dataset. The plot shows the positive instances on the x-axis and the negative instances on the y-axis, positioned based on their frequency. Since the data points represented as dots in the plot form two clusters, we expected the automatic mining approach to achieve accurate results.

3) Model Training: Jupyter Notebooks are used as frontend for learning and presenting the different classifiers.

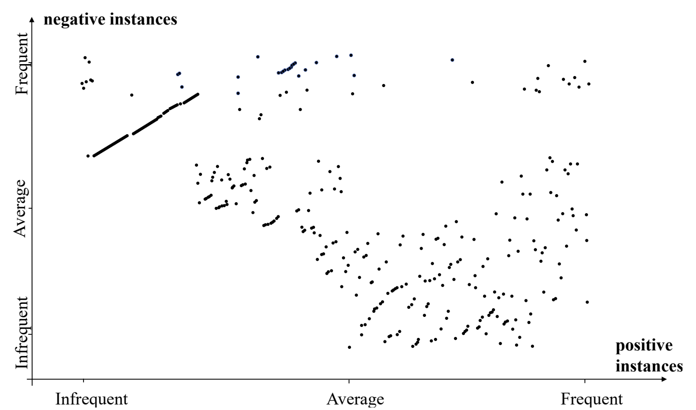

Fig. 3. Distribution of data points in the sample dataset using interactive scattertext plot (https://github.com/JasonKessler/scattertext).

Training of the classifiers. The Python *sklearn* package (https://scikit-learn.org) provides a simple and fast approach to train, validate, and test ML classifiers. We investigated different classes of classifiers: (i) probabilistic classifiers, such as *naive Bayes* (multinomial NB) and linear classifiers (*logistic regression* (LR)), (ii) deterministic classifiers, such as *support vector machines* (SVM), and (iii) decision tree algorithms, such as bagging models (*random forests* (RF)) and *boosting models* (XGB).

In addition, we trained a well-established *Convolutional Neuronal Network* (CNN) architecture for text classification (*epochs=100, batch_size=10*) in Keras with the following key components: (i) A (pre-trained *GloVe*) *word embedding* employs a dense vector representation of words with a similar meaning. (ii) *Convolutional layers* extract high-level features from comments represented using the word embedding and learn the correlations between nearby inputs. (iii) A *fully connected model* (i.e., the classifier) interprets the extracted features in terms of a predictive output. In total, the CNN consists of 10 layers and 1597594 trainable parameters.

Parameter tuning to get a best fit model. We used grid search with 5-fold cross-validation to try all possible combinations of parameters to improve the classifiers' performance.

4) Model Test: To evaluate the effectiveness of our classifiers, we measured well-known metrics [14] for the quality of binary classification results.

Evaluation metrics. We first determine *precision* and *recall* to oppose the fraction of relevant comments among the retrieved ones to the fraction of the total amount of relevant comments actually retrieved by the classifiers. Thereby, *false positives* (FP) represent 'trash' comments (i.e., comments that are wrongly classified as TEXT (but in fact are IGNORE)), and *false negatives* (FN) represent 'missing' comments (i.e., comments that are wrongly classified as IGNORE (but are in fact TEXT)). We then calculate the harmonic mean of precision and recall, i.e. the *f-measure*, as well as the *ROC AUC* and *MMC* (Matthews correlation coefficient), which are more robust performance measures for binary classifiers. Since there always is a trade-off between optimizing precision and

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on June 03,2020 at 07:11:09 UTC from IEEE Xplore. Restrictions apply.

TABLE I
EVALUATION OF DIFFERENT ML CLASSIFIERS.

	prec	rec	f1	acc	rocauc	mcc
NB	0.9076	0.9095	0.9085	0.9133	0.9131	0.8261
LR	0.9748	0.9431	0.9587	0.9601	0.9598	0.9206
SVM	0.9937	0.8908	0.9394	0.9392	0.9422	0.8840
RF	0.9874	0.9363	0.9611	0.9621	0.9621	0.9255
XGB	0.9811	0.9415	0.9609	0.9621	0.9619	0.9249

TABLE II
EVALUATION OF FEATURES USING AVG. 5-FOLD CV ACCURACY.

			NLP features		
	no param. tuning	text cleaning	count vector	tf-idf n-gram	pattern
NB	0.8533	0.9145	0.9225	0.9135	0.9282
LR	0.8733	0.8883	0.9282	0.9441	0.9508
SVM	0.5647	0.9092	0.9192	0.9275	0.9331
RF	0.9056	0.8743	0.9381	0.9448	0.9551
XGB	0.7991	0.7882	0.9315	0.9288	0.9511

optimizing recall, and in our case it is equally important to limit the number of FP and to avoid FN, we finally calculated the *accuracy* of our models as the primary metrics for the evaluation.

Classification evaluation. We used stratified k-fold cross-validation (with k=5) to evaluate the generalization performance of the classifiers. Accordingly, five models are trained and evaluated on randomly partitioned training/validation datasets, ensuring same proportions between the two classes in each fold. The accuracy values from the models were finally averaged to provide a single prediction.

Threats to validity. The validity of our experiments may be affected by different threads. Threats to *sample validity* relate to the input dataset we used to train and test the classifiers. Although we manually labeled 4010 documents, this may not deliver sufficient knowledge to build general models. Threats to *external validity* relate to the generalization of the results; that is, the used feature engineering method and classifiers for the COBOL system may not work for other systems.

IV. RESULTS

A. Comparing ML Classifiers and Features

To address *RQ1* we compared traditional ML classifiers using the above described text cleaning techniques, word vector representations, NLP features, and heuristic-based pattern information. The test scores of all investigated classifiers are shown in Tab. I. Precision and recall of the included comments reach 91% to 99% respectively, whereas most classifiers show better results for precision. The scores of f-measure (f1), accuracy, and ROC AUC are very close to each other for each classifier, i.e. 91% for NB, 94% for SVM, and 96% for LR, RF, and XBG. This demonstrates that the logistic regression model as well as the ensembles of decision trees can distinguish our comment classes very well, which is also confirmed by the respective MCC scores.

The results confirm that our approach is effective in classifying source code comments of the considered systems. Almost all ML classifiers achieve accurate results (from 94% to 96% accuracy).

By comparing different settings for the classifiers, we can comment on *RQ2*. The averaged cross-validation (CV) accuracy for each experiment is shown in Tab. II. We start with discussing the improvement that can be achieved by optimizing the parameters of the classifiers using grid search. Parameter tuning is particularly essential for SVM, and also brings improvements for the probabilistic classifiers. The default parameters of the decision tree algorithms, however, already work quite well (i.e., >90%).

Then, we compare the results of using NLP features in addition to text cleaning techniques. We also contrast accuracy values from applying different word vector representations, i.e., count vectors and tf-idf vectors with n-grams. Finally, we present the results of adding an additional feature based on heuristics, namely the pattern information according to our rules, using the best vector representation for each classifier.

Our experiments show the importance of feature dimension for comment classification. When we only applied basic pre-processing and text cleaning to train the classifiers, the results are generally $\leq$ 91%. The results further approve that stacking NLP with text features have positive impacts on classifying source code comments (accuracy 92% to 94%). Lastly, when we combined NLP features with heuristic features, the accuracy even further increased up to 96% for the RF as the best classifier.

Thuswe can confirm that the use of advanced NLP pre-processing techniques and features bring improvements in (almost) all cases for classifying source code comments.

To sum up, decision tree algorithms, in particular RF, are the most effective ML classifiers in automatically classifying source code comments (see Tab. I). They work well on the training set and are able to generalize to new source code comments. Random forests are currently among the most widely used ML methods and are able to automatically identify the most relevant features from a plethora of features. They can learn even from not preprocessed data (see Tab. II), that is why they are also considered as a preliminary stage for DL. Linear models also achieve good results and are fast to train and predict, but the logistic regression classifier requires the model complexity to be regularized. Naive Bayes tend to be even faster in training than linear models but their performance is worse. Deterministic classifiers, such as SVM, are powerful models and often perform quite well but for classifying source code comments they achieve a lower accuracy on average. In addition, they are more sensitive to data pre-processing, feature engineering, and parameters settings.

B. Comparing Heuristics and Learning Approaches

To determine the best performing approach in terms of time effort and accuracy for source code comment classification (*RQ3*), we compared the heuristic-based approach to our most effective ML classifier. We also added the results of the DL approach using CNN and word embedding (with the

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on June 03,2020 at 07:11:09 UTC from IEEE Xplore. Restrictions apply.

TABLE III
COMPARISON OF HEURISTICS, ML AND DL APPROACHES.

	effort	comments in data/test set	correctly detected	acc
heuristics	>4 days	4010	3832	0.9348
RF	4 days	1003	948	0.9452
RF + heuristics	>8 days	1003	965	0.9621
CNN	3 days	962	957	0.9948

best parameters obtained from grid search). Table III shows accuracy and required effort of each approach.

The accuracy of the heuristic-based approach is about 93% (similar to the results of NB and SVM). However, implementation and testing of rules for pattern recognition is time-consuming (>4 days). The rules must also consistently be adapted to new cases (languages, systems, etc.).

Compared to the results of the heuristics, the RF classifier achieves a higher accuracy of 95%. It requires labeling the 4010 documents manually by two experts (1 day) and comprehensive text cleaning and feature engineering (3 days). The combination of heuristics and ML approach manages to improve the accuracy to 96%.

Finally, the CNN yields the best results with an accuracy of 99%. We included the pre-trained *GloVe* word embedding as the first layer of the CNN model to learn domain-specific vocabulary, language-dependent properties and keywords. The resulting model shows short training times and proved to perform well on our training sample. Since we used a standard architecture design for text classification, effort of the DL approach is limited to hyperparameter tuning (2 days) and data labeling (1 day). In contrast to the ML classifiers, the CNN automatically learns the features from raw text data; thus, no text cleaning or feature engineering is required.

C. Comparison with Existing Approaches

The most relevant approaches are [7], [8], which classify Java and C/C++ source code comments using a J48 decision tree algorithm and a naive Bayes Multinominal classifier respectively. Both use specific text pre-processing and feature engineering with rule-based taxonomy classification, and achieve a weighted average precision and recall of 93% to 96% (on imbalanced data sets). In contrast to these approaches, we base the classification on both, the comments' syntax and meaning and additionally evaluate deep learning methods, which achieve even better results for comment classification.

V. DISCUSSION AND CONCLUSION

The binary classification using heuristics was developed for the legacy system without taking into account the comment categories from the literature. However, these categories could also serve as a basis for the required classification. Most categories are developer-specific and can be ignored for our purpose, however, categories such as *inline comment* and *section comment* are those that are desired for the documentation.

We developed the heuristics step by step for the batch programs. For the regeneration of documentation, this approach requires a new quality control after updating the heuristics. A one-time comment classification in an early project phase is not only possible, but also keeps the documentation generation stable over several versions of the software to be documented. Unlike the development of heuristics, data labeling also has the potential to be performed by the quality team that maintains the documentation standard. We conclude that the comment classification by machine learning (e.g., RF) and in particular deep learning (e.g., CNN) is ready for industrial adoption in the context of redocumentation. Even if they are not yet used in industry, our experiments show that ML/DL achieves better results than (industry-strength) heuristics. For future work, we plan to investigate how our approach can be reused and transferred to other (legacy) systems.

ACKNOWLEDGMENT

The research reported in this paper has been partly supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH.

REFERENCES

[1] J. V. Geet, P. Ebraert, and S. Demeyer, "Redocumentation of a legacy banking system: an experience report," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, September 20-21, 2010.*, 2010, pp. 33–41.

[2] B. Dorninger, M. Moser, and J. Pichler, "Multi-language redocumentation to support a cobol to java migration project," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 536–540.

[3] K.-G. Henstorf, U. Kampffmeyer, and J. Prochnow, *Grundsätze der Verfahrungsdokumentation nach GoBS, Code of Practice*. VOI Verband Organisations- und Informationssysteme e. V., 1999, vol. Band 2.

[4] A. Van Deursen and T. Kuipers, "Building documentation generators," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug 1999, pp. 40–49.

[5] M. Moser, J. Pichler, G. Fleck, and M. Witlatschil, "Rbg: A documentation generator for scientific and engineering software," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 464–468.

[6] A. Corazza, V. Maggio, and G. Scanniello, "On the coherence between comments and implementations in source code," in *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, Aug 2015, pp. 76–83.

[7] D. Steidl, B. Hummel, and E. Jürgens, "Quality analysis of source code comments," in *IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013*, 2013, pp. 83–92.

[8] L. Pascarella, M. Bruntink, and A. Bacchelli, "Classifying code comments in java software systems," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1499–1537, Jun 2019.

[9] Y. Shinyama, Y. Arahori, and K. Gondow, "Analyzing code comments to boost program comprehension," *CoRR*, vol. abs/1905.02050, 2019.

[10] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin Heidelberg: Springer, 1999.

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[12] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer, 2001, vol. 1, no. 10.

[13] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[14] H. Schütze, C. D. Manning, and P. Raghavan, "Introduction to information retrieval," in *Proceedings of the international communication of association for computing machinery conference*, 2008, p. 260.

Authorized licensed use limited to: Universitaet Klagenfurt. Downloaded on June 03,2020 at 07:11:09 UTC from IEEE Xplore. Restrictions apply.