Pattern Learning for Detecting Defect Reports and Improvement Requests in App Reviews

Gino V. H. Mangnoesing[1](✉), Maria Mihaela Truşcă[2](✉), and Flavius Frasincar[1]

[1] Erasmus University Rotterdam,
Burgemeester Oudlaan 50, 3062 PA Rotterdam, The Netherlands
gvh.sing@gmail.com, frasincar@ese.eur.nl
[2] Bucharest University of Economic Studies,
Piata Romana 6, 010374 Bucharest, Romania
maria.trusca@csie.ase.ro

Abstract. Online reviews are an important source of feedback for understanding customers. In this study, we follow novel approaches that target the absence of actionable insights by classifying reviews as defect reports and requests for improvement. Unlike traditional classification methods based on expert rules, we reduce the manual labour by employing a supervised system that is capable of learning lexico-semantic patterns through genetic programming. Additionally, we experiment with a distantly-supervised SVM that makes use of the noisy labels generated by patterns. Using a real-world dataset of app reviews, we show that the automatically learned patterns outperform the manually created ones. Also the distantly-supervised SVM models are not far behind the pattern-based solutions, showing the usefulness of this approach when the amount of annotated data is limited.

1 Introduction

In the two last decades, the growth of user-generated content on the Web has accelerated enormously due to parallel developments, such as increased Internet access, technological advancements in mobile devices, the growth of e-commerce, and many more. An important source of user-generated content with respect to customer feedback are online reviews. Their interpretation is usually achieved using Sentiment Analysis (SA) methods which has as the main aim to automatically detect positive, neutral, and negative sentiments [10]. A major downside of SA is that it measures satisfaction at a certain point in time. In this light, we argue that in addition to SA, it is important to focus on detecting specific types of feedback that indicate potential causes and influence factors of satisfaction. We consider such specific customer feedback as actionable, since it suggests a clear course of action for addressing the feedback, and thus directly help to modify and hopefully improve products.

E. Métais et al. (Eds.): NLDB 2020, LNCS 12089, pp. 126–134, 2020.
https://doi.org/10.1007/978-3-030-51310-8_12

In this paper, we focus on customer feedback related to mobile software applications which we will refer to as "apps". We argue that software reviews are very important for aggregating valuable feedback. Firstly, because many companies have come to realise that all the technology required to transform industries through software is available on a global scale [1]. Secondly, the field of software engineering has the well-accepted notions of bugs and feature requests, which we argue, are actionable types of feedback.

There are very few works [4,6,12–14] that aim to detect specific information in customer feedback. Among the aforementioned works, only the method proposed in [12] is more refined. Namely, in [12] lexical patterns are used to train a supervised classifier, rather than directly employing patterns for information extraction, which makes the extraction mechanism more adaptive to the various representations of feedback. Further on, this system summarizes the extracted feedback by means of a *Topic Model* technique called *Latent Dirichlet Allocation* (LDA) [2]. However, while the objective is very relevant, the suggested methods require a vast amount of manual labour to create useful feedback patterns. We argue this to be a great limitation since analysing customer feedback is an important process that should ideally be performed in a continuous fashion. Nevertheless, the study conveys a promising direction for future research in opinion mining, and clear feedback types to focus on, which we adopt in this work.

We approach feedback detection as a multi-label classification problem based on knowledge-base rules or patterns, in which our goal is to automatically determine if a given review is an example of given actionable feedbacks. Usually, making a knowledge base of patterns is impractical to manage over time and across different domains. In this light, we suggest a system that is capable of performing pattern construction in an automated manner using genetic programming. Keeping in mind the importance of reduction of the human control over the system's design, we also tackle the problem of having a small number of labeled reviews (gold labels) using noisy labels generated based on patterns in a distantly-supervised way [8,15]. The employed dataset and the proposed framework implemented in Scala are available at https://github.com/mtrusca/PatternLearning.

The remaining parts of the paper are structured as follows. Section 2 presents a detailed overview of the proposed framework in this study. In Sect. 3 we evaluate our framework through a series of experiments. Finally, in Sect. 4 we present our conclusions and suggest future work.

2 Methods

In this research, our goal is to automatically detect actionable feedback in reviews. More specifically, we aim to detect two specific types of feedback: defect reports and improvement requests. We approach this task as a binary classification problem, meaning that each review is considered a document that requires two classifications, one for each feedback type. Using this setup it is possible to classify some reviews as both defect report and improvement request. Our main contribution to the research problem is to automate the task of discovering and constructing patterns. Rather than direct supervision, where labels are provided by human annotators, we use a group of patterns to provide (noisy) labels for each feedback type. These labels are then given as input to a linear SVM model, often applied for text classification tasks due to its learning capability that is independent of the dimensionality of the feature space.

Using noisy labels to guide algorithms is a technique called *Distant Learning* or *Distant Supervision* [8,15]. Despite the fact that *Distant Supervision* is already a great step towards minimizing the amount of human labour required to perform feedback detection, the required process for manually constructing groups of patterns per feedback type, remains rather tedious and time consuming. For this reason, we suggest another level of automation, which is to automate the pattern creation procedure (responsible to generate noisy labels) by means of a learning algorithm.

To solve our problem for learning patterns, we require to select a learning algorithm that stands out with respect to interpretability and modifiability. A specific category of algorithms that meets these requirements are *Evolutionary Algorithms* (EAs). The most popular type of EA is the *Genetic Algorithm* (GA), however we adopt a special case of GA called *Genetic Programming* (GP) inspired by Darwin's theory of evolution [3]. Genetic Programming and Genetic Algorithms are very similar. They both evolve solutions to a problem, by comparing the *fitness* of each candidate solution in a population of potential candidates, over many generations. In each generation, new candidates are found by quasi-randomly changing (mutation) or swapping parts (crossover) of other candidates. The least "fit" candidates are removed from the population. The primary difference between GA and GP is the representation of the candidate solutions. In GA a candidate is represented as a vector, and in GP a candidate is represented as a tree. As the GP representation fits better the specification of our information extraction patterns, we adopt it in our research.

The learning approach suggested in GP, is to define an environment in which a collection of randomly generated, simple programs (individuals) evolve through an analogue of natural selection. Each individual represented by a tree structure is composed from a collection of nodes. All nodes (except the first, or root node) have one parent and any number of children. Every node belongs to one of two types, namely *functions* or *terminals*. Function nodes are allowed to have children nodes, which can be either functions or terminals. Terminal nodes are not allowed to have child nodes, therefore terminal nodes are considered the leaves of the tree. In our framework, we consider each individual to be a pattern for classifying documents (app reviews) with a (recursive) match method.

Function nodes include Boolean operations, such as *AND*, *OR*, and *NOT*, as well as *Sequence* and *Repetition*. The *Sequence* node can have one or more child nodes of types function or terminal. It is also the root node of each tree. A *Repetition* node enforces two or more consecutive nodes to obey the same condition. A node of type *AND* has at least two children, and is useful to pattern match for multiple features, for example to check whether a given token is both a specific literal and part of a syntactic category. The nodes of type *OR* and of type *NOT* also follow the Boolean logic, where the *OR* nodes match as true if at least one of the children matches, and nodes of type *NOT* match as true if none of its children match for a given token. Terminal nodes are the external points (or leaves) of the tree. They are assigned a specific value, used to pattern match for specific tokens. *Literal* nodes must be exactly matching the specific word (value) that is assigned the node. For *Part-of-Speech* (POS) nodes, tokens are evaluated to match a specific Part-of-Speech tag. A *Wildcard* node will match any token, irrespective of its value. Finally, an *Entity Type* node matches a value from a manually constructed and populated gazetteer.

Typically gazetteers consists of sets of terms containing names of entities such as cities, organisations, or weekdays [5]. Since at the time of performing this research, we could not find gazetteers for our specific domain, we decided to define our own. Our gazetteer is implemented using a plain key-value mapping, where a key corresponds to the name of an entity type, and the value stores a set of lexical representations of that entity type. For example, to detect the entity type *app* we employ the following terms: *it*, *app*, *application*, *Evernote* (we use a set of *Evernote* reviews for our experiments). Some other entity types in our gazetteer are: *user*, *action*, *object*, *component*, *device*, and *update*. The entity types we employ are inspired by *Issue Tracking Systems* (ITS), such as *Bugzilla*, an open-source issue tracker created by *Mozilla*. Since ITS involve very comparable types of feedback to this study, we consider the entity types in ITS a useful starting point for constructing our gazetteers.

The first step for each genetic program, is to generate an initial population of N individuals. In our experiments, we use the *ramped-half-and-half* method [9], which is commonly used since it produces a wider range of variation in terms of shapes and sizes of trees compared to the other popular methods like *grow* and *full*. The *ramped-half-and-half* achieves more variety, by combining both the *grow* and *full* methods, where one half of the population is generated through the *grow* method, and the other half through the *full* method. The algorithm we employ to generate individual trees in a recursive manner is based on the one suggested in [7].

During the initialization, nodes are selected randomly to construct trees. However, for the purpose of stimulating useful combinations of terminals, we generate a pool of recommended terminal candidates. Whereas the pool contains all entity types and the wildcard, for the case of *POS* and *Literal* terminals we select only the most relevant nodes. More specifically, we pre-analyse the training set for frequently occurring unigrams (as terminals) and bigrams (as pairs of terminals) of types *Literal* and *POS* (for bigrams, four specific pair combinations are considered: $(Literal)(Literal)$, $(POS)(Literal)$, $(Literal)(POS)$, and $(POS)(POS)$). Subsequently, we remove in each sentiment class of a target feedback type, the 100 most frequent unigrams and bigrams that occur in the another sentiment class. Then, every time a terminal node is needed we randomly select it from the pool of recommended terminal candidates.

In Evolutionary learning methods, a population of individuals can evolve for many generations. However, after a certain amount of generations, the *fitness* of the best new individuals will stop increasing. In our problem, we want individual patterns to be optimized for high precision, which means that we want more weight on precision than recall. Hence, we employ the F_β-measure with $\beta = 0.3$ instead of the widely used F_1-measure. Further on, we employ two criteria for termination. The first criterium is the maximum number of generations and is checked when generating a pattern (in the pattern group). The second criterium is checked per event type and it is triggered if the pattern does not increase the *fitness* of the entire group of patterns after a maximum number of iterations. The *fitness* measure for a group of patterns is determined by the F_1-measure, instead of the F_β-measure. Our motivation for using F_1 for group fitness is related to our goal to seek patterns for as many variations of a target feedback type as possible.

A proper procedure for selection should not find only the strongest individual of a population, but to allow more individuals to have a chance of being selected. A common method that addresses this requirement is *Tournament Selection*. Precisely, the method allows for a constant selection pressure that determines the extent to which fit individuals are preferred over less fit individuals. All the selected individuals are used to produce *offspring* or the next generation of individuals. The main objective in producing offspring, is to enhance the *fitness* for the next generation based on three genetic operations, namely *Elitism*, *Crossover*, and *Mutation*.

As discussed earlier, our goal is to learn a group of patterns that detect as many variations of a target feedback type as possible, in our training examples. In essence, each pattern can be interpreted as a rule, and each document has to be categorised as either positive or negative, according to our "knowledge" of each category, which is stored in a rule base. The set of rules learnt in our framework is generated through a *Sequential Covering Algorithm* [11].

3 Experiments

In order to evaluate the approach suggested in our framework, we performed experiments on a real-life dataset. The dataset contains 4470 reviews about *Evernote*, a mobile app for the Android platform. We automatically extracted the review dataset from the Google Play Store, through Web scraping techniques. We selected *Evernote* because it is a widely used app with a large user base, that publicly share their feedback on the Web, and therefore serves as a great example for our examined research problem.

We have annotations for 46% of the total review dataset. We hold out 20% of all reviews for testing purposes in all methods. Therefore, we have the remaining 26% of reviews available for training purposes. However, for the experiments that employ distant supervision, we generate noisy labels, hence, have 80% of the full review dataset available for training. The terms "Positive" and "Negative" refer to the classification labels that were assigned to every review per feedback type

Table 1. Examples of human (A) and automatically constructed (B) patterns. DR and IR stand for Defect Report and Improvement Request, respectively. For DR patterns ":" separates the terminal from its type.

Type	Pattern	Example
DR	OR: |-Software Bug: Entity Type |-Software Update: Entity Type	The last few months of updates haven't changed or lessened the lag you get when you edit notes
IR	SEQ: |-5: Literal |-stars: Literal	Colour coding of the notes and reminders for repetitive tasks can fetch 5 stars

Table 2. Performance metrics for feedback type classifications in terms of precision, recall, and F_1-measure. The best results are set in bold.

Task	Defect classification			Improvement classification		
Method	Precision	Recall	F_1-measure	Precision	Recall	F_1-measure
Standard SVM	0.39	0.59	0.47	0.78	**0.54**	**0.64**
Patterns A (manual)	0.61	0.42	0.50	**0.81**	0.42	0.56
Patterns B (learned)	**0.91**	0.39	**0.54**	0.79	0.51	0.62
SVM Distant Supervision A	0.24	**0.67**	0.36	0.39	0.48	0.43
SVM Distant Supervision B	0.41	0.59	0.49	0.46	0.44	0.45

by human annotators. On average 12.6% of our labeled set of reviews contains one or more actionable types of feedback, in which there are 8.4% more requests for improvement than defect reports. Finally, only 1.3% of our annotated reviews is labeled as both a defect report and an improvement request.

We collected annotations for both feedback types through *CrowdFlower* (recently renamed *Figure Eight*), an online data enrichment platform. The instructed task is to label every individual review for both defect reports and improvement requests. Every review was annotated by at least 3 annotators, and in some cases even 5 or 7 (when it is recorded a low accuracy of the test questions that inspect the quality of the annotator).

The employed patterns are constructed both manually and automatically. In the *Evernote* dataset, we have five manual and two generated patterns for defects, and eight manual and ten generated patterns for improvements. The most likely reason for this contrast is the variation in distribution of feedback types in our dataset, as a result of the fact that *Evernote* is a popular app, well tested, and optimised. Furthermore, we noticed that the most effective patterns only use function nodes of type *Sequence* and *OR*. Also, many examples of feedback can be recognized with a single terminal, such as the *Entity Type* "software update" for defect reports or the *Literal* "stars" for improvement requests, which indicates that the level of specificity does not necessarily have to be high. In that light, patterns that include the *NOT* node, which requires feedback examples in which a very specific word is not mentioned are often not necessary. While *NOT* functions can be useful to make a pattern very expressive and precise, it becomes obsolete when that level of selectivity is not required, as in our case. A similar line of reasoning can be applied to the *AND* functions. Table 1 lists two examples of automatically constructed patterns for the two types of feedback.

To classify defect reports and improvement requests we test the following methods:

Table 3. Running time for pattern creation per approach. The best results are set in bold.

Approach	Defect patterns	Improvement patterns	Total
Manual (per person)	8.5 h	10.25 h	18.75 h
Automated	**3.5 h**	**2.4 h**	**5.9 h**

Method 0: Standard SVM. In this method, we train an SVM classifier using only labelled reviews for training. This method can be considered a reference for the following methods.

Method 1: Patterns A. In this experiment, we use human patterns to perform supervised classifications directly (without SVMs). We employ the available labelled data (26%) for learning patterns.

Method 2: Patterns B. This method is similar to the Method 1, except that the human patterns are replaced with automatically constructed ones.

Method 3: SVM Distant Supervision A. In this method, we train an SVM classifier using noisy labels generated based on the human patterns for the entire training set.

Method 4: SVM Distant Supervision B. This method is similar to the Method 3, except that the human patterns are replaced with automatically constructed ones.

Table 2 displays an overview of performance measures of all proposed methods. We can notice that the *Distant Supervision* methods are not far behind the direct classification through patterns, in terms of F_1-scores. Nevertheless, given that the results are obtained with noisy labels shows the usefulness of this approach for datasets where the annotated data is limited.

As regards the comparison between the two types of patterns, it is obvious that the automatically generated patterns perform better than the human ones.

In order to have a complete insight over the pattern creation process (manual versus automated) we additionally explore the patterns' efficiency besides their effectiveness. Table 3 displays the running time for creating patterns both manually and automatically. We can observe that it takes 70% less time to generate the automatic patterns than the manual ones.

4 Conclusion

In this study we presented a framework for automatically learning lexico-semantic patterns helpful for detecting specific types of feedback expressed in conversational customer feedback (defect reports and improvement requests). Using a custom dataset, we showed that the automatically generated patterns perform slightly better than the manual ones and there is a 70% reduction in construction time. Further on, we demonstrated that the distantly-supervised SVM with noisy labels is not far behind the pattern-based classification. The results reveals the applicability of this approach when the amount of available labels is limited.

As future work, we would like to increase the flexibility of our patterns by considering more complex terminal structures. Using techniques from entity-learning we would like to explore the automatic generation of our domain-specific gazetteers lists to increase coverage and the framework's applicability in other domains.

References

1. Andreessen, M.: Why software is eating the world. Wall Street J. **20** (2011)
2. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. (JMLR) **3**, 993–1022 (2003)
3. Booker, L.B., Goldberg, D.E., Holland, J.H.: Classifier systems and genetic algorithms. Artif. Intell. **40**(1), 235–282 (1989)
4. Brun, C., Hagege, C.: Suggestion mining: detecting suggestions for improvement in users' comments. Res. Comput. Sci. **70**, 171–181 (2013)
5. Cunningham, H.: GATE, a general architecture for text engineering. Comput. Humanit. **36**(2), 223–254 (2002)
6. Goldberg, A.B., Fillmore, N., Andrzejewski, D., Xu, Z., Gibson, B., Zhu, X.: May all your wishes come true: a study of wishes and how to recognize them. In: 10th Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT 2009), pp. 263–271. ACL (2009)
7. IJntema, W., Hogenboom, F., Frasincar, F., Vandic, D.: A genetic programming approach for learning semantic information extraction rules from news. In: Benatallah, B., Bestavros, A., Manolopoulos, Y., Vakali, A., Zhang, Y. (eds.) WISE 2014. LNCS, vol. 8786, pp. 418–432. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11749-2_32
8. Ji, G., Liu, K., He, S., Zhao, J.: Distant supervision for relation extraction with sentence-level attention and entity descriptions. In: 31st AAAI Conference on Artificial Intelligence (AAAI 2017), pp. 3060–3066. AAAI Press (2017)

9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection, vol. 1. MIT Press, Cambridge (1992)
10. Liu, B.: Sentiment Analysis: Mining Opinions, Sentiments, and Emotions. Cambridge University Press, Cambridge (2015)
11. Mitchell, T.M., et al.: Machine learning (1997)
12. Moghaddam, S.: Beyond sentiment analysis: mining defects and improvements from customer feedback. In: Hanbury, A., Kazai, G., Rauber, A., Fuhr, N. (eds.) ECIR 2015. LNCS, vol. 9022, pp. 400–410. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16354-3_44
13. Qiao, Z., Zhang, X., Zhou, M., Wang, G.A., Fan, W.: A domain oriented LDA model for mining product defects from online customer reviews. In: 50th Hawaii International Conference on System Sciences (HICSS 2017), pp. 1821–1830. ScholarSpace/AIS Electronic Library (2017)
14. Ramanand, J., Bhavsar, K., Pedanekar, N.: Wishful thinking: finding suggestions and 'Buy' wishes from product reviews. In: Workshop on Computational Approaches to Analysis and Generation of Emotion in Text (CAAGET 2010), pp. 54–61. ACL (2010)
15. Sahni, T., Chandak, C., Chedeti, N.R., Singh, M.: Efficient Twitter sentiment classification using subjective distant supervision. In: 9th International Conference on Communication Systems and Networks (COMSNETS 2017), pp. 548–553. IEEE (2017)