Impact of Refactoring on Source Code: A Computational Linguistics Approach

Musfiqur Rahman*, Nafiz Islam†
Department of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: *musfiqur.rahman@mail.concordia.ca, †na_islam@encs.concordia.ca

***Abstract*—Naturalness is fundamentally repetitiveness or predictability. Like the natural language, programming languages are natural. Researchers used that idea to improve Statistical Models of code, Porting and Translation, Studying the Naural Linguistics of Code, Suggestions and Completions, Analysis and Tools, Assistive Technologies, Corpus Curation etc. Refactored code are simpler, understandable, efficient and compact, so refactored code should be more natural. In this paper, we investigate this hypothesis. We consider a large corpus of smelly data and refactored data from 10 different Java projects, and focus on its language statistics, evaluating the naturalness of smelly code. Initially the smells are refactored using automated tool *JDeodorant* and our results show that the perplexity of refactored data is less than the smelly code. Which indicates that refactored code are more natural than smelly code.**

***Keywords*– Refactoring; code smell; n-gram; language model**

I. INTRODUCTION

Programming languages show similar pattern in terms of repetitiveness like any natural language. Although programming languages are artificially developed with more compact rules both syntactically and semantically, surprisingly, they are even more predictable and repetitive than natural human languages. This idea was first discussed in [3] where Hindle *et al.* capture the regular and predictable nature of programming languages by *n-gram* language model. Moreover, they use this information captured by the *n-gram* language model in code suggestions. Later on, it was shown by Tu *el al.* [5] that by using *cache-based* language model the naturalness can be captured even better as source codes are more repetitive locally (i.e. in file level) than they are repetitive globally (i.e. in the whole project corpus). Using cache-based language model improves the quality of code suggestion task. Besides code suggestion, this idea of repetitiveness of software source code has been exploited in suggesting bug fixes as well [2].

The process in which developers improve the architecture and design of an existing source code without effecting its functionality is known as *Refactoring* [12]. Refactoring is performed on source code due to various reasons like, adding new features, bug fix requests, code smell etc. [6]. Sometimes developers take help of tools to perform refactoring, whereas in most of the cases they put manual effort in doing so [4][6]. Although many of the modern Integrated Development Environments (IDEs) come with the feature of refactoring, researchers are also putting considerable amount of effort in developing tools to automate refactoring [7][8]. Over all, the idea of refactoring has become a crucial part of software development nowadays and refactorig has made softwares more maintainable, more reusable and more readable. Threfore, many works have so far been done on theory and applications of refactoring [9][10][11][13]. This huge amount of application of refactoring in both theory and practice rises an interesting question: *How does refactoring affect the naturalness of source code?*

The effect of refactoring on source code can be studied by using *Language Model* [1]. Language modelling is a very popular approach in the field of *Statistical Machine Translation (SMT)* [14] and *Natural Language Processing (NLP)* [15]. Previous works that had studied the naturalness of software source code also mathematically defined the term **naturalness** based on the theory of language modelling [2][3][5]. After being trained on a large corpus language models assign higher naturalness to previously seen code, while assigning lower naturalness to unseen or rarely seen code [5]. For example, Campbell *et al.* [16] showed that language models mark code which is syntactically faulty as *unlikely* or *less likely.*

A. Research Questions

We investigate the impact of refactoring on the naturalness of source code by addressing the following issues:

- change in naturalness of the code after applying tool-based refactoring
- change in naturalness of the code after applying manual refactoring
- different types of refactoring and naturalness of code

More specifically we try to answer to following three research questions.

RQ1. Real refactoring: Does refactoring that developers perform change the cross-entorpy of the source code?

We determine the naturalness of source code before and after the refactoring has been performed by a developer. We try to find whether the naturalness drops, increases or remains same after performing refactoring.

RQ2. Tool-based or automated refactoring: Does automated refactoring change the naturalness?

Like *RQ1*, in *RQ2* also we determine the naturalness of source code before and after the refactoring has been performed. This time the refactoring is done by using tool, and not by the developer. This time also we try to find whether the naturalness drops, increases or remains same after performing refactoring.

RQ3, Types of refactoring: How do different types of refactoring affect the naturalness?

There are multiple types of refactoring. We investigate their impact on naturalness separately and try to find how differently (or similarly) different types of refactoring impact the naturalness.

Rest of the paper is structured in the following way. In Section II we discuss about the our data source, methodology along with some theoretical background. Three RQ1, RQ2 and RQ3 are discussed in Section III, IV and V respectively.

II. DATA SOURCE AND METHODOLOGY

A. Data

Our primary data source is the open-source GitHub projects. For our study, we search for refactorings performed in different versions of the selected repositories. In order to do that we perform analysis on the differences between the source code of the refactored project before and after the refactoring was performed. We use a refactoring detection tool called *RefactoringMiner* [6] for accomplishing the analysis task. This tool gives us information about the commit id associated with the refactoring, type of refactoring that has been performed, files that have been changed etc. Thus, we get the two versions of the project: *the version before refactoring* and *the version after refactoring*. Note that *RefactoringMiner* detects refactoring only in the projects written in Java. Furthermore, it cannot yet detect all types of refactoring. For example, the tool cannot detect *RENAME CLASS/METHOD/FIELD* refactoring types. The detailed underlying theory behind how the tool works along with its limitations are discussed in [18]. Summary of our data is shown in Table I.

B. Background

1) Language Model: In this paper, we use the term language model (LM), which is nothing but probability distributions over sequence of *m* tokens *P(*k_1*,* k_2*,...,* k_m*)*. Language Model is trained on a corpus of sequences of tokens from the language, with the goal of assigning high probability to tokens with maximum likelihood, and low probability to tokens with minimum likelihood. Language models are needed to model the uncertainty of the language by determining the most probable sequence of tokens for a given input.

2) N-Gram Language Model: Consider the sequence of tokens k_1, k_2, k_3, ... k_{m-1}, k_m in a document, *D*. N-gram model statistically calculates likelihood of tokens to follow other tokens. Thus, we can estimate the probability of a document based on the product of a series of conditional probabilities:

$$P(D) = P(k_1)P(k_2|k_1)P(k_3|k_1,k_2)...P(k_n|k_1,k_2,...,k_{n-1}) \tag{1}$$

Where *P(D)* is the probability of document and *P(*k_i*)* is the conditional probability of tokens. We can transform above equation to following more general form of equation.

$$P(k_1,k_2,k_3,...,k_{m-1},k_m) = \sum_{i=1}^{m} P(k_i|k_1,...,k_{n-1}) \tag{2}$$

In this transformation it is assumed that token occurrences are influenced only by limited prefix of length n. This assumption is known as **Markov Property** as described by Zhang *et al.* [17]. Furthermore, we can consider this as a Markov Chain which assumes that the outcome of next step depends only on current step. Thus we can write:

$$P(k_i|k_{i-(n-1)},...,k_{i-1}) = P(k_i|k_{i-(n-1)}) \tag{3}$$

To use above equation we need to know conditional probabilities values for each token for each possible n-gram. The conditional probability can be calculated from n-gram frequency counts.

Bigram and Trigram language models can be modeled with value of *n = 2* and *n = 3* respectively.

3) Cross Entropy: Cross entropy is used to compare probability distributions when the true probability distribution is unknown. Given a corpus *K* of size *N* consisting of tokens k_1, k_2,..., k_n, the log probability of the model distribution *m* with true distribution *p* on this corpus is defined as,

$$H(P,m) = \sum P(x)log_2(m)(x) \tag{4}$$

H(P,m) indicates the average number of bits required to encode messages sampled from p with a coding scheme based on *m*.
H(P) is a lower bound on *H(P,m)* and *H(P,m)* is an upper bound on *H(P)*. The lower *H(P,m)*, the closer the model *m* is to the truth.[**?**] [**?**]

4) Perplexity: Perplexity is a per-word average of the probability with which the language model generates the test data set, where the average is over the number of words in the test data set.
The perplexity of a discrete probability distribution *P* is defined as

$$2^{H(P)} = 2 - \sum P(x)log_2(P)(x) \tag{5}$$

where *H(p)* is the entropy of the distribution and *x* ranges over events.

C. Methodology

1) Data Extraction: After getting the two versions of the source code of the project we extract the files that have been

TABLE I
SUMMARY OF THE DATA

Project Name	Type	Access Date	Files (*.java)	Comments	Lines of Code	#of tokens
Vert.x	Before Refactoring	20-Jul-2016	414	21837	50436	407825
	After Manual Refactoring	20-Jul-2016	437	21927	51920	422390
languagetool	Before Refactoring	18-Jul-2016	949	27320	73045	646598
	After Manual Refactoring	18-Jul-2016	949	27319	73060	646691
Aeron	Before Refactoring	20-Jul-2016	237	8369	26667	193956
	After Manual refactoring	20-Jul-2016	237	8369	26649	194053
checkstyle	Before Refactoring	20-Jul-2016	1092	39712	73214	504086
	After Manual Refactoring	20-Jul-2016	1092	39695	73175	503857
Guacamole-client	Before Refactoring	18-Jul-2016	327	19558	11855	90044
	After Manual Refactoring	18-Jul-2016	327	19564	11859	90061
processing	Before Refactoring	20-Jul-2016	228	50577	91024	698598
	After Manual Refactoring	20-Jul-2016	228	50576	91024	698603
Buck	Before Refactoring	23-Jul-2016	2269	82571	226583	1966841
	After Manual Refactoring	23-Jul-2016	2271	82601	226674	1967353
orientdb	Before Refactoring	23-Jul-2016	2057	55111	265440	2363105
	After Manual Refactoring	23-Jul-2016	2060	55152	265664	2364702
atomix	Before Refactoring	23-Jul-2016	254	13916	19424	147668
	After Manual Refactoring	23-Jul-2016	254	13917	19424	147623
fabric8	Before Refactoring	20-Jul-2016	913	24045	65966	531673
	After Manual Refactoring	20-Jul-2016	913	24045	65953	531516

changed. From those source files with extract the lines that have been changed. There are two types of changes:

- Addition: a line is added to the source file
- Deletion: a line is removed from the source file

We separately extract added and deleted lines from the two versions of the project. We use `git-diff` command to extract the changed lines.

2) Data Preprocessing: For each data set in our hand, we preprocess the data by removing comments from the source file, and then perform lexical analysis according to the language syntax. We use ANTLR4 [1] to lexicalyze the raw code and generate token sequence.

3) Perplexity Calculation: The sequence of tokens generated in previous step (Section II-C2) are used to train and test the *n-gram* language model. Each data set (i.e. each version of a project) is divided into two parts of 90% and 10% of the preprocessed data. We train our language model on the 90% portion, and validate the model on the 10% portion. Results are averaged out over 10-fold cross validation. We use MIT Language Model (MITLM) toolkit [2] for calculating the perplexity of each data set. For our study we use 1 to 10 as the value of *n*.

The logarithm of perplexity is defined as *cross-entropy*, which is the measure of naturalness of the data [3]. *The less the cross-entropy value is, the more natural and predictable the data are.*

III. Real Refactoring

RQ1. Real refactoring: Does refactoring that developers perform change the cross-entorpy of the source code?

As shown by Silva *et al.* [6], most of the refactorings are done by the developers manually. We try to determine how, if at all, refactoring changes the cross-entropy of the source code by answering to *RQ1*.

A. Finding

The fundamental purpose of performing refactoring is to make code more organized and maintainable. Intuitively we assume that making code more organized will reduce the value of cross-entropy, making it more regular and predictable. However, the result from our experiment proves that our intuition is not correct. Figure 1 shows the result. From the plot it is vivid that for any vale of *n* the cross-entropy of the project does not change after refactoring has been performed. Therefore, the overall perplexity remain same.

B. Discussion

The result can be explained based on the count of tokens of each project. The count of total tokens, unique tokens and percentage of unique tokens are shown in Table II.

The token level statistics of the project show that there is no significant change in the distribution of tokens before and after refactoring. Therefore, the cross-entropy remains same even if refactoring has been performed.

Refactoring does not effect the naturalness of source code. The cross-entropy of a project remains unchanged for any n-gram before and after applying refactoring.

IV. Automated Refactoring

RQ2. Tool-based or automated refactoring: Does automated refactoring change the naturalness?

Many tools have been developed for making the task of refactoring easier, automated and less time consuming. From Section III we conclude that manual refactoring does not change cross-entropy of a project with any level of significance. We investigate whether tool-based refactoring has any

[1]http://www.antlr.org/

[2]https://github.com/mitlm/mitlm

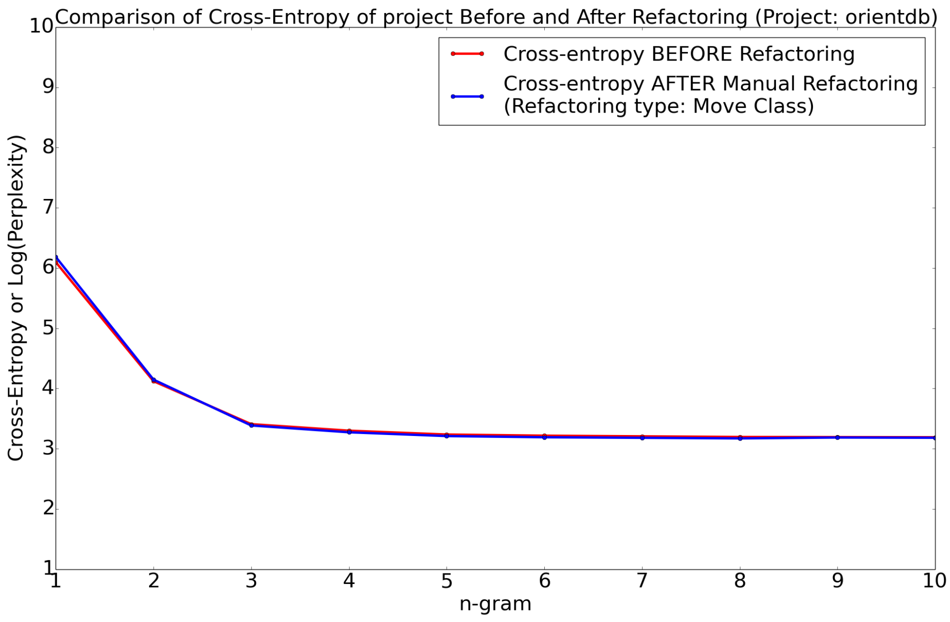

Fig. 1. Impact of Move Class Refactoring on project OrientDB

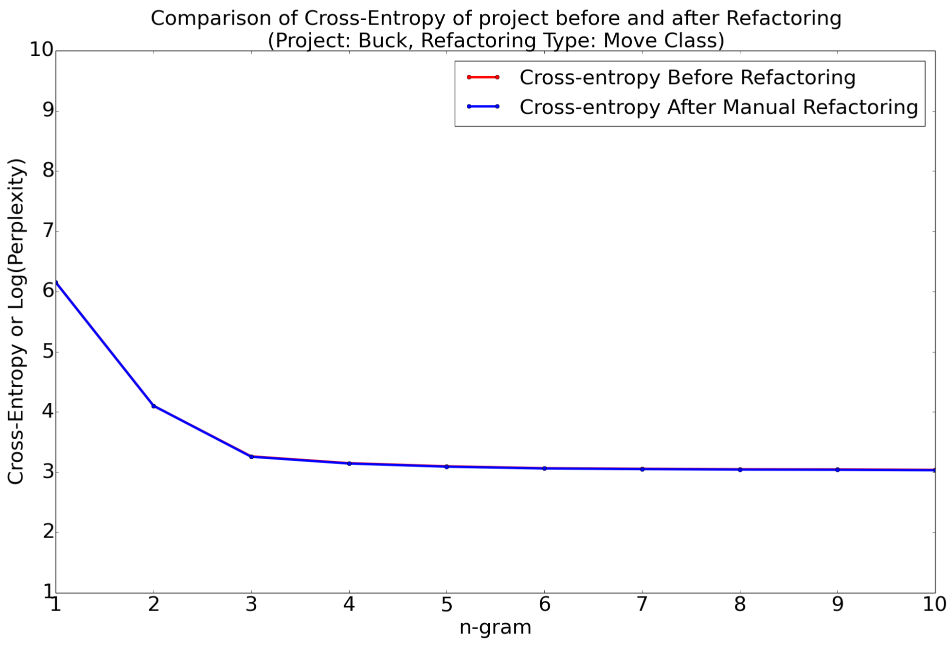

Fig. 2. Impact of Move Class Refactoring on project Buck

different effect on the cross-entropy. We use *JDeodorant* [3] plug-in of *Eclipse* [4] to perform the refactorings.

A. *Finding*

Figure 3 and 4 for show that the pattern of cross-entropy after manual and tool-based refactoring are exactly same. The curves have overlapped each other. Therefore, from linguistic point of view, refactoring using a tool does not contribute differently on the source code from the way that manual refactoring does.

[3]https://users.encs.concordia.ca/ nikolaos/jdeodorant/
[4]http://www.eclipse.org/

B. *Discussion*

We can discuss the result in a same manner as we do in Section III-B. The distribution of tokens play the most vital

TABLE II
STATISTICS FOR ORIENTDB

Statistics	Before Refactoring	After Refactoring
Total Number of Tokens	2152849	2154385
Number of Unique Tokens	35439	35480
% of Unique Tokens	0.0164614424885	0.0164687370178

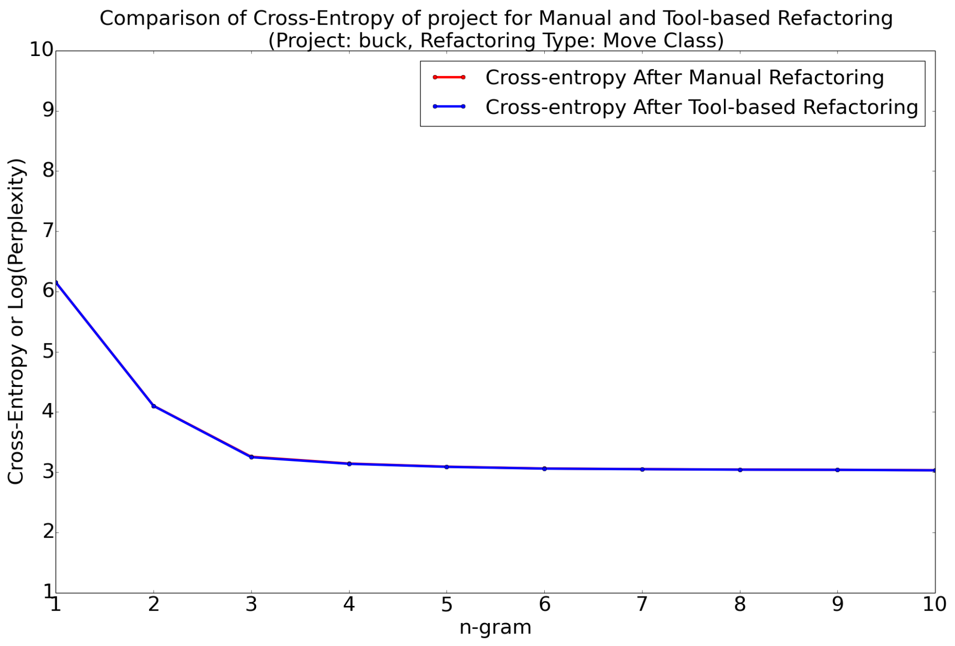

Fig. 3. Comparison between tool-based and manual refactoring on project buck

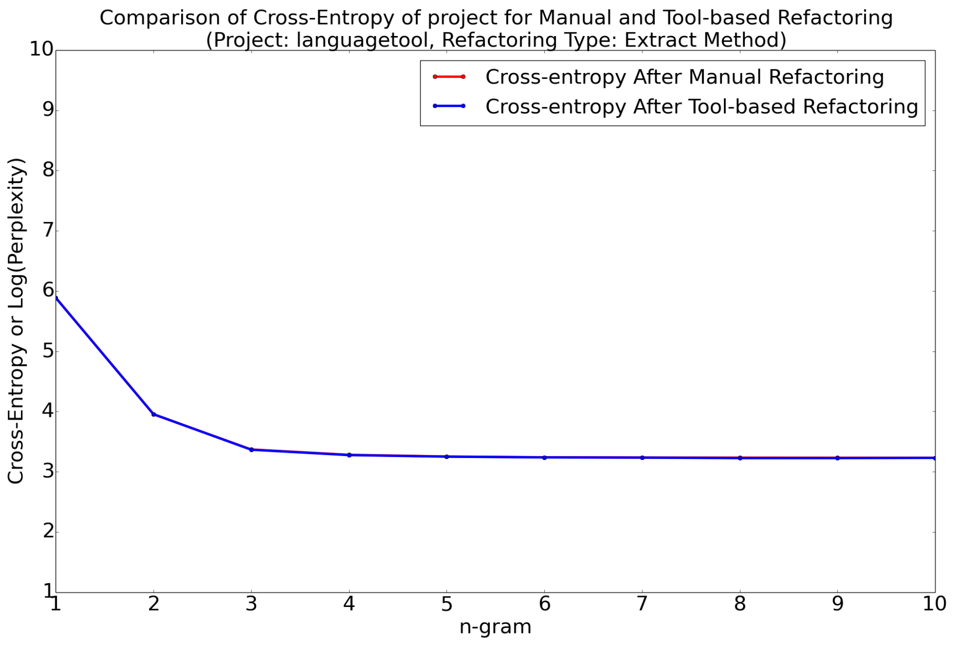

Fig. 4. Comparison between tool-based and manual refactoring on project languagetool

TABLE III
COMPARISON OF STATISTICS BETWEEN MANUALLY AND AUTOMATED REFACTORED CODE FOR PROJECT: LANGUAGETOOL

Statistics	Manual Refactoring	
Total Number of Tokens	591189	605683
Number of Unique Tokens	32606	32639
% of Unique Tokens	0.0551532589409	0.0538879248716

TABLE IV
COMPARISON OF STATISTICS BETWEEN MANUALLY AND AUTOMATED REFACTORED CODE FOR PROJECT: BUCK

Statistics	Manual Refactoring	Tool-based Refactoring
Total Number of Tokens	1775290	1776152
Number of Unique Tokens	39906	39916
% of Unique Tokens	0.0224785809642	0.0224733018345

role in the cross-entropy value of the project. We can see from the statistics of tokens that the token distribution does not get affected differently even if we use tools (Table III and IV).

Like manual refactoring, tool-based refactoring also do not have any significant impact on the cross-entropy of the project because of the fact that the distribution of tokens remain unchanged.

V. DIFFERENT TYPES OF REFACTORING AND CROSS-ENTROPY OF SOURCE CODE

RQ3, Types of refactoring: How do different types of refactoring affect the naturalness?

We study four types of refactoring in this work:

- Extract Method
- Inline Method
- Move Class
- Pull-up Method

We observe the change in cross-entropy after performing each type of refactoring separately on different projects. The plots are shown in Figures 5, 6, 7, 8.

A. Findings

For the plots we see that the result of different refactoring types on different Java projects do not vary from each other. Over all cross-entropy does not get affected by any kind of refactoring.

B. Discussion

By definition of refactoring we keep the functionality of code same. Although we perform different types of refactoring, the over all distribution of tokens do not change significantly because of the fact that we just change the placement of tokens. This is the reason why refactoring does not impact the over all cross-entropy of the project.

VI. CONCLUSION

In this work we show that refactoring do not play a role in the cross-entropy of projects. Cross-entropy of source code depend only on the distribution of tokens. Refactoring reorganizes code, it does not change features or functionalists. Because of this fact we hardly introduce new tokens in the code or remove tokens from the code. We essentially just change the placement of tokens to make sure that the code still does the same functions as it used to do before refactoring. This fundamental property of refactoring is proved by this study where we show that cross-entropy does not get affected by refactoring.

REFERENCES

[1] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. 1990. A statistical approach to machine translation. Comput. Linguist. 16, 2 (June 1990), 79-85.

[2] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "naturalness" of buggy code. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). ACM, New York, NY, USA, 428-439. DOI=http://dx.doi.org/10.1145/2884781.2884848

[3] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, 837-847.

[4] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How we refactor, and how we know it. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 287-297. DOI=10.1109/ICSE.2009.5070529 http://dx.doi.org/10.1109/ICSE.2009.5070529

[5] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 269-280. DOI: http://dx.doi.org/10.1145/2635868.2635875

[6] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'2016), Seattle, WA.

[7] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How Are Java Software Developers Using the Eclipse IDE?. IEEE Softw. 23, 4 (July 2006), 76-83. DOI=http://dx.doi.org/10.1109/MS.2006.105

[8] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: Clone Refactoring. pp. 613-616, 38th International Conference on Software Engineering (ICSE'2016), Formal Tool Demonstration Session, Austin, Texas, USA.

[9] M. F. Zibran and C. K. Roy. 2013. Conflict-aware Optimal Scheduling of Code Clone Refactoring. IET Software, Volume 7(3), June 2013, pp. 167-186.

[10] M. Mondal, C. K. Roy and K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking," Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on, Victoria, BC, 2014, pp. 11-20. doi: 10.1109/SCAM.2014.11

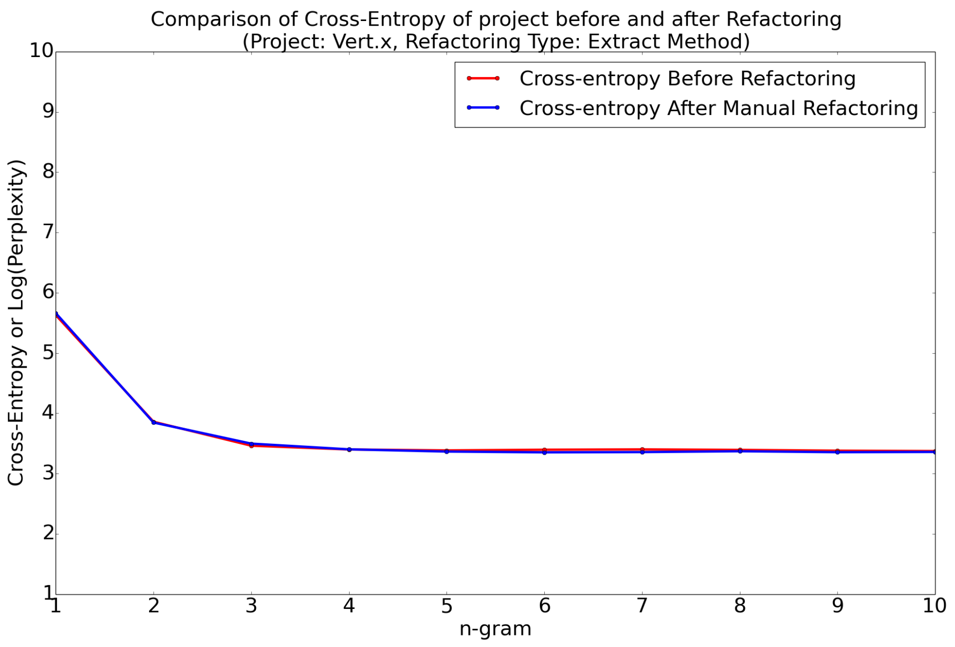

Fig. 5. Comparison of cross-entropy before and after refactoring on project Vert.X

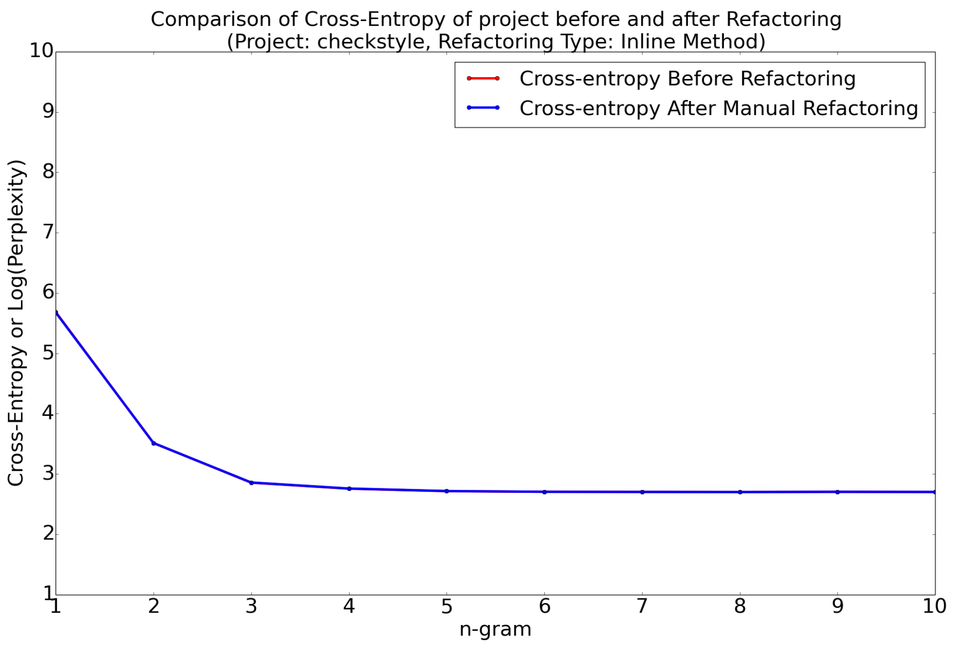

Fig. 6. Comparison of cross-entropy before and after refactoring on project Checkstyle

[11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2012. Identification and application of Extract Class refactorings in object-oriented systems. J. Syst. Softw. 85, 10 (October 2012), 2241-2260. DOI=http://dx.doi.org/10.1016/j.jss.2012.04.013

[12] W. F. Opdyke. 1992. Refactoring object-oriented frameworks. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

[13] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2010. Identification of refactoring opportunities introducing polymorphism. J. Syst. Softw. 83, 3 (March 2010), 391-404. DOI=http://dx.doi.org/10.1016/j.jss.2009.09.017

[14] Philipp Koehn. 2009. Statistical machine translation. Chapter 7, Cambridge University Press.

[15] K. Sparck Jones. 1994. Natural language processing: a historical review. Current Issues in Computational Linguistics: in Honour of Don Walker (Ed Zampolli, Calzolari and Palmer), Amsterdam.

[16] Joshua Charles Campbell, Abram Hindle, and JosÃľ Nelson Amaral. 2014. Syntax errors just aren't natural: improving error reporting with language models. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014). ACM, New York, NY, USA,

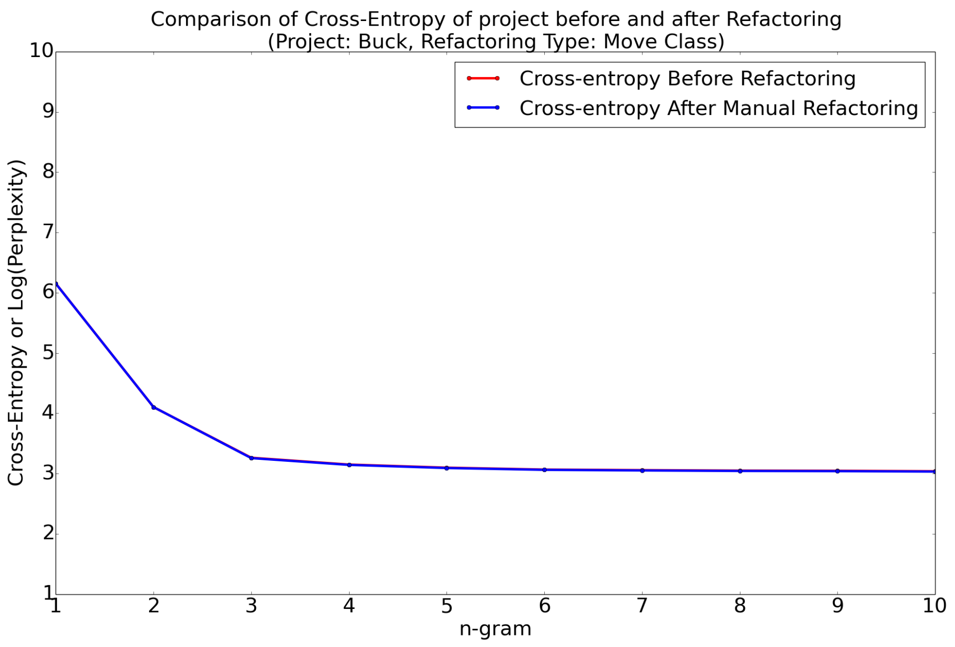

Fig. 7. Comparison of cross-entropy before and after refactoring on project Buck

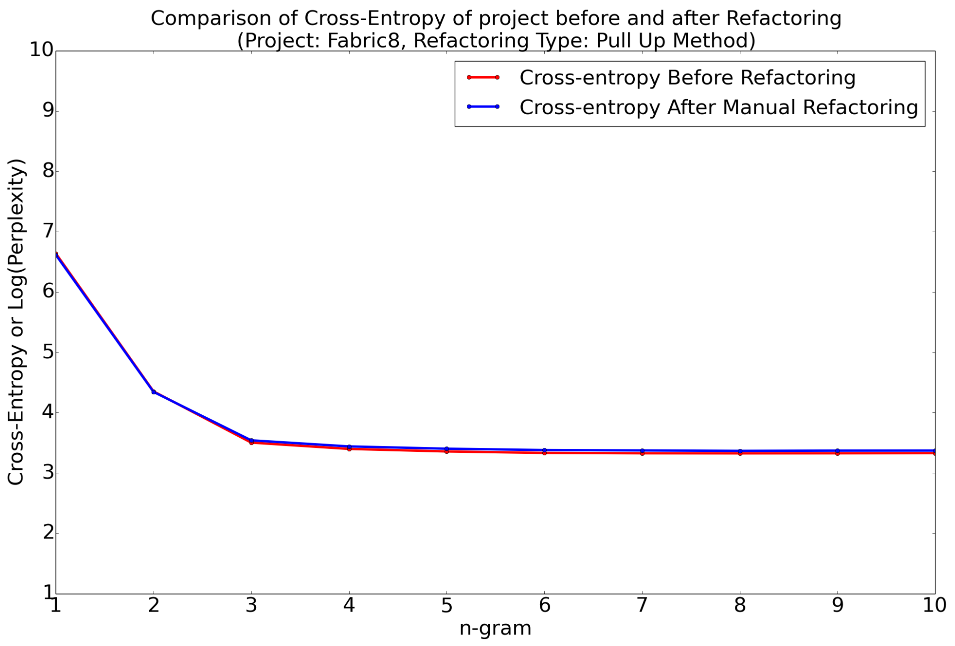

Fig. 8. Comparison of cross-entropy before and after refactoring on project Fabric8

252-261. DOI: http://dx.doi.org/10.1145/2597073.2597102

[17] Y. F. Zhang, Q. F. Zhang and R. H. Yu. 2010. Markov property of Markov chains and its test. International Conference on Machine Learning and Cybernetics, Qingdao, 2010, pp. 1864-1867. doi: 10.1109/ICMLC.2010.5580952

[18] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13). IBM Corp., Riverton, NJ, USA, 132-146.