Анализ данных для программных репозиториев

Д.Е. Намиот, В.Ю. Романов

***Аннотация*— В статье рассматриваются вопросы, связанные с использованием методов анализа данных применительно к программным репозиториям. В работе делается попытка представить обзор технологий, которые используются при анализе программ и базируются на статических данных, которые могут быть извлечены непосредственно из программного кода или репозиториев кода. В работе приводится обзор работ, использующих методы глубокого обучения (рекуррентные нейронные сети), методы классификации, основанные на других моделях машинного обучения, а также использование кластеризации в программной инженерии. Практические области применения рассматриваемых методов включают в себя, например, классификацию и предсказание ошибок, определение характеристики изменения кода во времени, поиск дублирующих фрагментов, автоматическое обнаружение ошибок проектирования, выдачу рекомендаций по рефакторингу кода.**

***Ключевые слова*—анализ данных, репозитории, программная инженерия.**

I. Введение

В данной статье рассматриваются задачи программной инженерии, связанные с анализом данных, которые могут быть извлечены из программных репозиториев. Коллекции текстов программ являются источником данных для разных задач, связанных с разработкой программного обеспечения. Идея состоит в том, что метрики, которые могут быть получены по программным текстам или, например, по байткоду языка Java, pfntv могут служить источником полезных заключений о самих программах (коллекциях программ).

В качестве метрик здесь может быть, например, то, что непосредственно измерено (вычислено) по исходному коду. Например, количество строк, число модулей, количество объектов с разными модификаторами доступа и т.д. Другая возможность – это каким-либо образом искусственно определенные измерения (например, какой-либо агрегат для метрик). Это то, что в англоязычной литературе называется feature selection [1].

Модели, которые здесь строятся, могут быть самыми разнообразными. Собственно, цель данной статьи как раз и состоит в попытке (первой для нас) создания обзора моделей, используемых в данной области.

Главная конечная цель для всех моделей (как, впрочем, и для большинства других задач программной инженерии) – это, естественно, автоматизация программирования. В настоящее время до этого пока еще далеко. На практике речь идет о более приземленных задачах. Сюда включается, например, поиск информации (программного кода), прогнозирование ошибок, обнаружение клонов, анализ связей, анализ эволюции и т.п.

Один из продвинутых примеров представлен, например, в статье [2]. Разработчик видеоигр Ubisoft представила программу Commit Assistant, которая активно блокирует ошибки кодирования. Этот инструмент предназначен для обнаружения ошибок, прежде чем разработчики даже совершают их в коде игры. Система была обучена на накопленных за 10 лет работы данных о том, где были сделаны предыдущие ошибки в коде, и какие модификации были применены для исправления этих ошибок. Это позволяет Commit Assistant предсказать, когда программист может столкнуться с риском введения подобной ошибки. Авторы отмечают возможность системы находить 60% ошибок и, соответственно, экономию до 20% времени разработчиков.

Интерес к использованию формальных моделей в программной инженерии существовал, естественно, уже давно. В качестве другого примера можно привести книгу [3] 1986 года, которая охватывает довольно много аспектов, таких, например, как верификация и трансформация программ.

Иными словами, данная статья – это наша первая попытка обзора нового направления, которое можно охарактеризовать как использование методов анализа данных для задач программной инженерии.

II. Рекуррентные нейронные сети

Рекуррентные нейронные сети (в англоязычной литературе Recurrent Neural Network или RNN) - это класс нейронных сетей, где связи между элементами образуют некоторую направленную последовательность. Благодаря этому появляется возможность обрабатывать последовательные цепочки событий.

Ниже приведен пример RNN, которая моделирует язык как последовательность символов. Сеть обучают на большом тексте и получают распределение вероятности следующего символа в последовательности, заданной

Статья получена 20 января 2018.
Д.Е. Намиот - МГУ имени М.В. Ломоносова (e-mail: dnamiot@gmail.com).
В.Ю. Романов - МГУ имени М.В. Ломоносова (e-mail: vladimir.romanov@gmail.com)

последовательностью предыдущих символов. Это позволит нам генерировать новый текст по одному символу за шаг.

Нижеследующая иллюстрация взята из работы [4]. В качестве рабочего примера предполагается, что у нас есть только лексика из четырех возможных букв «h e l o», и мы хотели обучить RNN на тренировочной последовательности «hello». Эта обучающая последовательность на самом деле является источником 4 отдельных примеров обучения:

1. Вероятность появления «e» должна быть определена с учетом контекста «h»,
2. «l» должно быть вероятным в контексте «he»,
3. «l» также должно быть вероятным с учетом контекста «hel», и, наконец,
4. «o» следует, вероятно, определять в контексте «hell».

Далее мы используем вектор из четырех элементов, где 1 на соответствующей позиции соответствует одному из символов нашего алфавита. На вход (рис. 1) подаются вектора, где представлен только один символ. На выходе мы видим вектора с рассчитанными предпочтениями для последующих символов.

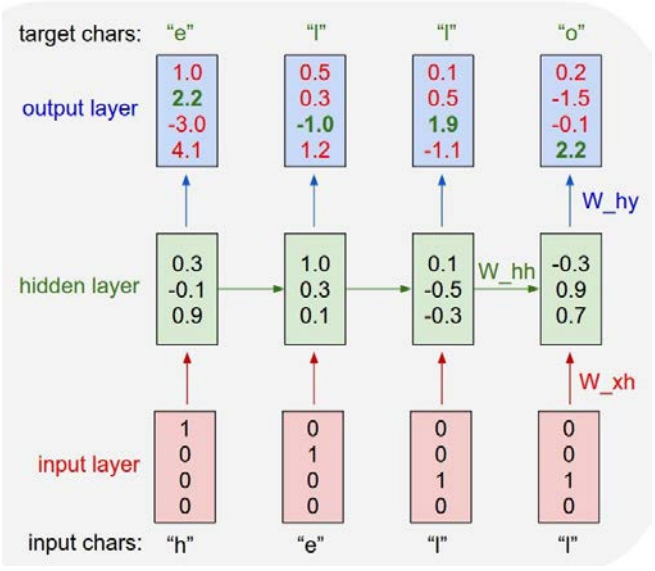

Рис. 1. Размерность входа и выхода – 4, один скрытый уровень из 3-х нейронов [4]. W_xh и W_hy – веса.

На этом примере очень легко понять, как устроена такая сеть. Далее, как это и указано в [4], сеть может быть обучена на текстах программ (поскольку это тоже язык). В оригинале для обучения использовался код на языке C из репозитория исходных текстов Linux. Для 477 Mb исходного кода была построена RNN модель с 10 миллионами параметров. В итоге, модель может генерировать “правдоподобные” тексты программ на языке C. Но более интересно, например, показывать возможные ошибки, исходя из того, что реальное следование символов в проверяемой программе не соответствует рассчитанным предпочтениям.

В работе [5] приводится обзор использования моделей глубокого обучения для целей анализа программных репозиториев. Авторы приводят обзор N-грамм моделей формальных языков (языков программирования) [6-14]. При этом отмечаются их существенные недостатки (например, необходимость учета контекста и семантики). Соответственно, им противопоставляются как раз рекуррентный нейронные сети. Конкретно в работе [5] рассматривается язык программирования Java. Области применения, которые рассматривают авторы: code review и code suggestion.

III. Другие модели машинного обучения

Большой обзор различных моделей машинного обучения, которые используются в предсказании наличия дефектов (ошибок) в модулях программных систем приводится в работе [15]. Масса работ, которые различаются как по метрикам, которые используются для описания программных модулей, так и по используемым классификаторам. Рис. 2 представляет собой сводную таблицу по использованию методов классификации:

Frequency of Instances by Classifier Family

	Instances	Percent
DecTree	172	28.7
Regression	136	22.7
Bayes	124	20.7
CBR	77	12.8
Search	41	6.8
ANN	28	4.7
SVM	17	2.8
Benchmark	5	0.8

Рис.2. Методы классификации [15]

В качестве источника данных может быть использована коллекция множеств данных, (datasets) относящихся к программной инженерии [16].

Использование машинного обучения для предсказания наличия ошибок в программных модулях является популярным подходом [17-19]. Причины очевидны – какую-то аналитическую модель здесь предложить трудно. С другой стороны, везде используются подходы с обучением и, соответственно, успех зависит от наличия размеченных примеров. Также можно сказать, что трудно выявить общие подходы к выбору характеристик исследуемого кода. Точно можно сказать, что данный подход имеет дело, в первую очередь, со статическими характеристиками кода. Но, и это отмечается в работах, надежность кода может зависеть, например, от частоты и характера обновлений. Можно сказать, что это направление – предсказание наличия ошибок и оценка надежности модулей программного обеспечения требует отдельного обзора. Интересный вопрос, в частности, могут ли существовать такие модели без привязки к языку программирования.

В части анализа возможных метрик интересна работа [20], где приводится довольно подробный анализ

влияния архитектуры на эволюцию и качество программной системы. Авторы работы исходят из предположения, что совместные изменения затрагивающие несколько модулей, определяющих архитектуру системы, с большей вероятностью вводят ошибки, чем изменения происходящие внутри такого модуля. В работе сделан обзор метрик используемых для измерения связанности модулей, а также обзор архитектурных видов используемых для визуализации свойств архитектуры системы. Предложенные метрики использовались для предсказания дефектов архитектуры на основе результатов корреляционного анализа истории изменений модулей. В качестве примеров для такого анализа использовались ряд распространенных программных систем с открытым исходным кодом.

В работе [21] рассматривается проблема архитектурного технического долга, когда принимается не оптимальное решение по архитектуре системы для временного, зачастую оправданного, ускорения разработки. Впоследствии такое решение начинает все сильнее осложнять разработку и требовать исправлений в архитектуре системы. Для оценки нарастания такого долга используется метрика - среднее количество модифицированных компонент при каждой фиксации кода в репозитории (ANMCC - average number of modified components per commit). Однако история изменения программного кода зачастую имеется не для всего программного кода. На основе анализа 13 проектов с открытым программным кодом были найдены метрики измерения модульности, значения которых коррелируются со значениями метрики ANMCC. Таким образом для оценки технического долга необходима лишь одна версия программной системы. ,.

В работе [22] обсуждаются признаки плохого кода (так называемые bad smell). Многие из них вычисляются на основе метрик и могут быть использованы в машинном обучении. Затем в работе сделан сравнительный анализ инструментов, позволяющих выполнять обнаружение плохого кода в программных репозиториях.

Безусловно, в качестве характеристик для машинного обучения могут рассматриваться и зависимости между проектами экосистем. Под экосистемой понимается группа проектов, которые совместно разрабатываются и развиваются в одном окружении. Зависимость между проектами, разрабатываемыми в GitHub подробно обсуждается в работе [23]. Возможности этого репозитория позволяют анализировать не только технические зависимости (между компонентами проектов), но и социальные (между собственниками и разработчиками проектов. В работе анализируются корреляция между технической и социальной зависимостями, а также способами визуализации таких зависимостей. Так, на рисунке 3 показана карта разработчиков проекта и их последователей.

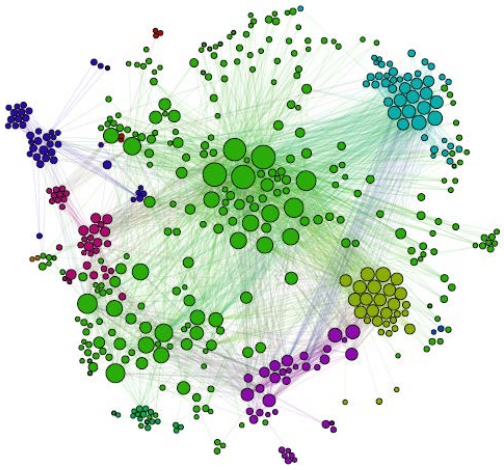

Рис. 3. Карта связи разработчиков проектов и их последователей.

На рисунке 4 показана карта зависимостей проектов экосистемы.

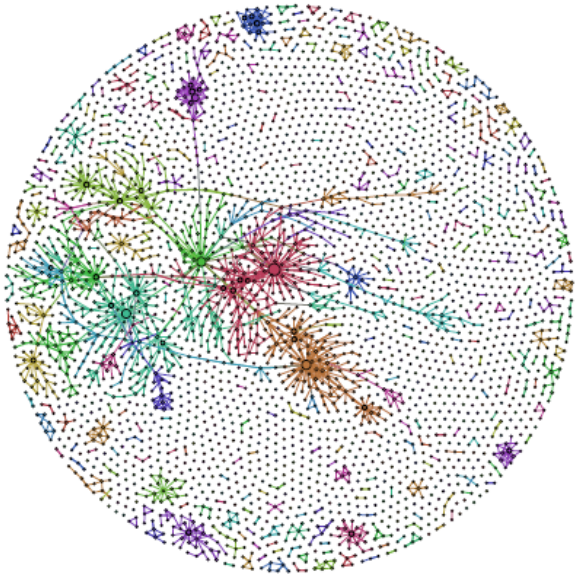

Рис. 4. Карта проектов экосистемы.

IV. Кластеризация

Мы назвали этот раздел по одному из часто используемых подходов. Технически это может охватывать большее количество методов. Здесь следует начать с работы [24] и доступной диссертации одного из ее авторов [25]. Это IR-модели (Information Retrieval) для большого программного кода из программного репозитория. В частности, в этих работах предложена методика устранения дублирования данных на основе анализа истории изменения большого программного кода.

Следующая модельная работа из этой серии – это статья [26]. Здесь авторы предлагают вероятностные модели для извлечения информации о темах и авторах из программных текстов. Области применения: автоматическая аннотация программных модулей, статистика деятельности разработчика, анализ сходства программ, сравнение работы программистов и т.д. Для проведения исследований была разработана

инфраструктура, позволяющая хранить в реляционной базе данных выгруженный из репозиториев сети Интернет исходный код более 10 тысяч проектов на языке Java, содержащий миллионы строк кода написанного 9000 авторами.

На рис. 5, например, изображена кластеризация авторов для реализации Eclipse 3.0

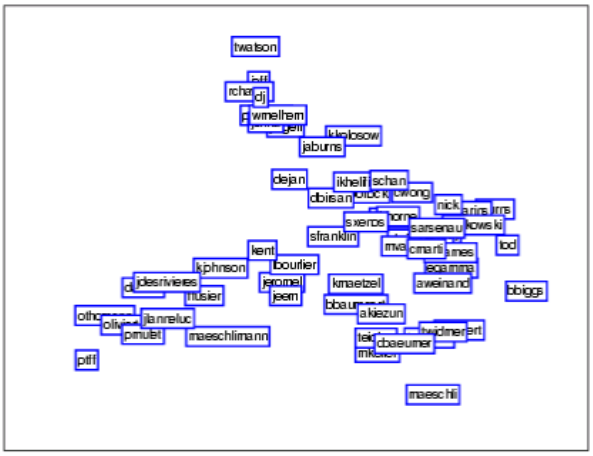

Рис. 5. Авторы Eclipse 3.0 [26]

Классические алгоритмы кластеризации применительно к задачам анализа качества программного обеспечения рассматриваются в работе [27]. В работе описаны объектно-ориентированные статические метрики, используемые для оценки качества программного обеспечения. Сначала выполняется идентификация выбросов значений метрик и последующая кластеризация классов методом K-средних. Затем строятся деревья классификации для идентификации тех метрик, которые определяют принадлежность классов к кластеру.

И, наконец, наиболее фундаментальный и полный обзор применения кластеризации в программной инженерии представлен в работе [28]. Также здесь содержится обширная информация по алгоритмам кластеризации.

В работе рассматриваются три основные области использования кластерного анализа в программной инженерии: анализа рефлексии, анализ эволюции программного обеспечения и восстановление информации. Цель рефлексивного анализа – это восстановление архитектуры программного модуля. Компонентам (элементам, фрагментам), должны быть поставлены в соответствие элементы гипотетической архитектуры.

В процессе эволюции программных компонент кластеризация используется, например, для уменьшения сложности кода (перегруппировки модулей), для определения дублирующего кода и т.п.

Восстановление информации (обратная разработка, обратная инженерия) - это восстановление компонентов или извлечение системных абстракций. Например, модули определяются в системе на основе кластеризации найденных зависимостей и т.д.

Области применения рассматриваемых методов следующие:

- классификация и предсказание ошибок
- определение характеристики изменения кода во времени
- оценка однородности кода (определение “отличающихся” фрагментов)
- автоматическое обнаружение ошибок проектирования
- автоматическая выдача рекомендаций по рефакторингу кода

Библиография

[1] Guyon I., Elisseeff A. An introduction to variable and feature selection //Journal of machine learning research. – 2003. – Т. 3. – №. Mar. – С. 1157-1182.

[2] AI Predicts Coding Mistakes Before Developers Make Them https://futurism.com/ai-predicts-coding-mistakes-before-developers-make-them/ Retrieved: Mar, 2018

[3] Rich C., Waters R. C. (ed.). Readings in artificial intelligence and software engineering. – Morgan Kaufmann, 2014.

[4] The Unreasonable Effectiveness of Recurrent Neural Networks http://karpathy.github.io/2015/05/21/rnn-effectiveness/ Retrieved: Mar, 2018

[5] White M. et al. Toward deep learning software repositories //Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on. – IEEE, 2015. – С. 334-345.

[6] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in Proceedings of the 34th International Conference on Software Engineering, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[7] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “A statistical semantic language model for source code,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE ’13. New York, NY, USA: ACM, 2013, pp. 532–542.

[8] S. Afshan, P. McMinn, and M. Stevenson, “Evolving readable string test inputs using a natural language model to reduce human oracle cost,” in Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ser. ICST ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 352–361.

[9] D. Movshovitz-Attias and W. W. Cohen, “Natural language models for predicting programming comments,” in ACL. Sofia, Bulgaria: Association for Computational Linguistics, August 2013.

[10] M. Allamanis and C. A. Sutton, “Mining source code repositories at massive scale using language modeling,” in MSR, 2013, pp. 207–216.

[11] J. C. Campbell, A. Hindle, and J. N. Amaral, “Syntax errors just aren’t natural: Improving error reporting with language models,” in Proceedings of the 11th Working Conference on Mining Software Repositories, ser. MSR ’14. New York, NY, USA: ACM, 2014, pp. 252–261.

[12] P. Tonella, R. Tiella, and D. C. Nguyen, “Interpolated n-grams for model based testing,” in ICSE, 2014, pp. 562–572.

[13] Z. Tu, Z. Su, and P. Devanbu, “On the localness of software,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE ’14. New York, NY, USA: ACM, 2014, pp. 269–280.

[14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE ’14. New York, NY, USA: ACM, 2014, pp. 281–293.

[15] Shepperd M., Bowes D., Hall T. Researcher bias: The use of machine learning in software defect prediction //IEEE Transactions on Software Engineering. – 2014. – Т. 40. – №. 6. – С. 603-616.

[16] The tera-PROMISE Repository http://openscience.us/repo/ Retrieved: Mar, 2018

[17] Malhotra R. A systematic review of machine learning techniques for software fault prediction //Applied Soft Computing. – 2015. – Т. 27. – С. 504-518.

[18] Di Martino S. et al. A genetic algorithm to configure support vector machines for predicting fault-prone components //International

Conference on Product Focused Software Process Improvement. – Springer, Berlin, Heidelberg, 2011. – С. 247-261.
[19] Laradji I. H., Alshayeb M., Ghouti L. Software defect prediction using ensemble learning on selected features //Information and Software Technology. – 2015. – Т. 58. – С. 388-402.
[20] Kouroshfar E. et al. A Study on the Role of Software Architecture in the Evolution and Quality of Software //Proceedings of the 12th Working Conference on Mining Software Repositories. – IEEE Press, 2015. – С. 246-257.
[21] Li Z. et al. An empirical investigation of modularity metrics for indicating architectural technical debt //Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures. – ACM, 2014. – С. 119-128.
[22] Fernandes E. et al. A review-based comparative study of bad smell detection tools //Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. – ACM, 2016. – С. 18.
[23] Blincoe K., Harrison F., Damian D. Ecosystems in GitHub and a method for ecosystem identification using reference coupling //Proceedings of the 12th Working Conference on Mining Software Repositories. – IEEE Press, 2015. – С. 202-207.
[24] Thomas S. W., Hassan A. E., Blostein D. Mining unstructured software repositories //Evolving Software Systems. – Springer, Berlin, Heidelberg, 2014. – С. 139-162.
[25] Thomas S. W. Mining unstructured software repositories using ir models. – Queen's University (Canada), 2013.
[26] Linstead E. et al. Mining internet-scale software repositories //Advances in neural information processing systems. – 2008. – С. 929-936.
[27] Papas D., Tjortjis C. Combining clustering and classification for software quality evaluation //Hellenic Conference on Artificial Intelligence. – Springer, Cham, 2014. – С. 273-286.
[28] Shtern M., Tzerpos V. Clustering methodologies for software engineering //Advances in Software Engineering. – 2012. – Т. 2012. – С. 1.

On data mining for software repositories

Dmitry Namiot, Vladimir Romanov

***Abstract*— The article discusses issues related to the use of data science and data mining methods for software repositories. The paper attempts to provide an overview of the technologies that are used in the analysis of programs and are based on static data that can be extracted directly from the code or the code repositories. The paper reviews papers using deep learning methods (recurrent neural networks), classification methods based on other machine learning models, and the use of clustering in software engineering. Practical applications of the methods under consideration include, for example, classification and prediction of errors, determining the characteristics of code change over time, searching for duplicate fragments, automatically detecting design errors, recommending code refactoring.**

***Keywords*— data science, software repositories, software engineering.**