AI in Software Engineering at Facebook

Johannes Bader, Sonia Kim, Frank Luan, Satish Chandra, Erik Meijer
Facebook, Inc.

December 2020

Abstract

We address the question: *How can AI help software engineers better do their jobs and advance the state of the practice?* Our perspective comes from building and integrating AI-based techniques in Facebook's developer infrastructure over the past two years.

In this article, we describe three productivity tools that we have built that learn patterns from software artifacts: code search using natural language, code recommendation, and automatic bug fixing. We also present a broader picture of how machine learning can bring insights to virtually all stages of the software lifecycle.

1 Introduction

AI, and more specifically the machine learning sub-area of AI, has had a transformative impact on almost every major industry today, ranging from retail, to pharmaceuticals, to finance. Not surprisingly, it is beginning to transform the software development industry as well, though significant potential remains untapped.

The underlying basis for the transformative impact of ML is the vast amount of data that is available to be analyzed, mined, and from which clever ML algorithms can extract patterns and insights. In software engineering, one of the most easily accessible data is source code itself. For example, GitHub hosts millions of projects, which together add up to billions of lines of code; most companies have large proprietary code repositories as well. Other examples of sources of data include:

- Incremental changes between repository versions of code;
- Large number of tests and their outcomes during continuous integration;
- Online forums, such as Stack Overflow, in which developers interact with each other.

What are some of the useful insights to be extracted from this data? And how can we use ML to extract those insights? Since software engineering is a lot about developer productivity, in the rest of this introduction, we give several examples of scenarios in which we have used ML to help developers work more efficiently; in the later sections, we will give technical details of how these tools work. Towards the end of the article we will present a broader picture of additional ways in which ML-based insights can help in software engineering.

Code search using natural language Consider the life of a developer who has to implement a function, for example, for hiding the Android soft keyboard programmatically. One way to tackle this problem is to study Android APIs and then implement the function. But APIs may take a long time to comprehend. It would be much more efficient to derive inspiration from existing code that serves a related purpose. One way to find a relevant code snippet is with a quick search on Stack Overflow. However, if the question is not already answered on Stack Overflow, posting a new question and waiting for a response has a long latency.

On the other hand, copious amounts of relevant Android code are available on GitHub. The problem is that it is hard to find such relevant snippets directly from a collection of repositories. We have created a technique that can help retrieve pertinent code snippet *directly* from source code, starting with just rough keywords. While the search does not come with the explanation that a Stack Overflow post has, it retrieves potentially useful information in real time.

Code recommendation Even when one does have a start on which APIs to use for a certain task at hand, the task is not done. When writing code, developers are curious how other programmers have written *similar* code, to get reassured, or to discover considerations they might have missed. If they directly search on a large code corpus for an API name, they might get tens of thousands of results. What they instead want is a small set of sample usages from the repository that gives them some additional information.

Consider an example usage of an Android API method `decodeStream`:

```
Bitmap bitmap = BitmapFactory.decodeStream(input);
```

But if one were to look at related code elsewhere in the repository, one variation is to make sure the app does not crash on an exception:

```
try {
    Bitmap bitmap = BitmapFactory.decodeStream(input);
    ...
} catch (IOException e) { ... }
```

This is a different search scenario that we call *code recommendation*. The input is a code snippet, and the output is a small list of related code fragments that show only a few representative variations of information that occurs commonly enough. We will discuss our approach to build such a code recommendation engine in section 3.

Automatically fixing routine bugs Code evolves constantly. At Facebook, the Android app repository alone sees thousands of commits per week. Since many of these commits are fixes to various issues, we can use ML to figure out the patterns to these fixes and automatically suggest an appropriate fix.

More specifically, we have found that fixes to static analysis warnings often come from a large palette of code patterns. The following shows an example fix (inserted code in green) of Infer's warning on potential NullPointerException (null dereferences) in Java:

```
if (this.lazyProvider == null || shouldSkip) {
  return false;
}
Provider p = this.lazyProvider.get();
```

The notable point is that developers have a strong preference for a certain way to fix a warning, even though there might exist alternate, semantically equivalent ways. A tool that recommends fixes must suggest the one that the developer finds natural in a given context. We will talk about a tool that discovers and learns bug-fixing patterns from data.

Takeaways These are just some of the many initiatives we have started and incorporated into practice at Facebook. Additional work includes: predictive regression test selection [1], triaging for crashes [2], and code auto-completion. [3] demonstrates how these tools are integrated into the Facebook development environment.

Our thesis is that even simple ML methods can help remove a lot of inefficiencies in the day-to-day life of a developer. No longer should they spend a lot of time looking for information over a repository. No longer should they spend a lot of time finding relevant information from hundreds of code fragments. No longer should they have to fix simple, predictable bugs manually.

In the next part of the article, we describe technical details for the three topics we introduced above.

2 Code Search

Background The ability to search over large code corpora can be a powerful productivity booster. Therefore, we have explored ways to search directly over the provided code corpora, using basic natural language processing and information retrieval techniques. There have been previous works in code search, such as CoCaBu [4] (a code search tool that augments natural language queries by adding correlated code vocabulary from Internet forums), and Sourcerer [5] (a code search framework that searches over open-source projects available on the Internet). However, these tools are not applicable for internal use since most of our developers work with proprietary APIs and frameworks which are rarely discussed on the Internet. Thus, we came up with an approach to directly

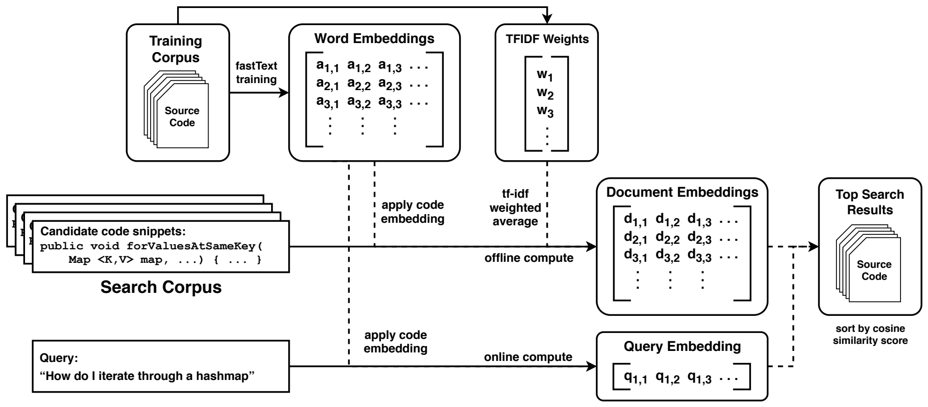


Figure 1: NCS model training and search retrieval. NCS extracts information from the source code, builds word embeddings, and uses TF-IDF weighting to get a document embedding for each code snippet. The query is mapped to the shared vector space, and the most relevant code snippets are ranked with cosine similarity.

search over the given corpus. Our tool, called Neural Code Search (NCS) [6], aims to find relevant code snippet examples given a query in natural language.

How does it work? NCS is built using the idea of *embeddings*, which are vector representations of code that aim to capture the intent of a piece of code in a form suitable for machine learning. Our hypothesis is that the tokens in source code are generally meaningful, and embeddings derived from these tokens can capture the intent of the code snippet well enough for code search. NCS creates embeddings at the granularity of a method body.

As shown in Figure 1, NCS works in the following steps:

Extract information. NCS first extracts relevant tokens from source code to create a "natural language" document. The information NCS extracts are: method names, comments, class names, and string literals.

Build word embeddings. NCS then builds word embeddings using FastText [7] which gives vector representations for each word in the corpus. Similar to Word2Vec [8], FastText performs unsupervised training such that words appearing in similar contexts have similar vector representations. For example, the embedding of *button* is the closest with the embeddings of *click, popup, dismissible*, when trained on an Android code corpus.

Build document embeddings. Finally, to create a document embedding for each method body in the corpus, NCS computes a weighted average from its tokenized words and its respective word embeddings, as shown in Equation 1 (d is a set of words in a document; C is the corpus containing all documents; u is a normalizing function). This document embedding serves to capture the overall semantic meaning of the method body. NCS weights the words using

TF-IDF (Equation 2), a well-known weighting technique in IR. The top portion of Figure 1 shows the NCS model training part.

$$v_d = u\left(\sum_{w \in d} u(v_w) \cdot \mathrm{tfidf}(w, d, C)\right) \tag{1}$$

$$\mathrm{tfidf}(w, d, C) = \frac{1 + \log \mathrm{tf}(w, d))}{\log |C| \cdot \mathrm{df}(w, C)} \tag{2}$$

Search retrieval. Upon receiving a search query, NCS tokenizes the query and uses the *same* trained word embeddings to represent it as a vector. It is important to note that the tokenization will turn the natural language query to a series of main keywords that captures the essence of the query. For example, the query "How to get the ActionBar height" will be tokenized to "get action bar height." NCS then compares this vector to the document embeddings as discussed above. NCS ranks the document embeddings by cosine similarity using FAISS [9], a standard similarity search algorithm that operates on high-dimensional data, and returns the top results. The bottom portion of Figure 1 shows the search retrieval part.

Evaluation We evaluated the effectiveness of NCS on a set of Stack Overflow questions, with the post title as the query and a code snippet from the accepted answer as the desired code answer. Given a query, we measured whether NCS was able to retrieve a correct answer from a large search corpus (GitHub repositories). Out of 287 questions, NCS correctly answered 98 questions in the top 10 results. This evaluation dataset, along with the search corpus, is publicly available at [10].

Some examples of Stack Overflow questions that NCS answers well are:

- "*How to delete a whole folder and content?*"
- "*How to convert a image into Base64 string?*"
- "*How to get the ActionBar height?*"
- "*How to find MAC address of an Android device programmatically?*"

Sachdev et al. [6] includes more details on the training and evaluation of NCS. We further investigated whether deep-learning models lead to better code search results [11].

Developer feedback The usage of NCS at Facebook was somewhat different from the way we had envisioned it. Developers did not often write Stack Overflow-style questions; instead, they mostly searched with keyword queries, such as "contract number amount". Although the raw query types were different, with the tokenization step where we break down both code snippets and the queries into keywords, we were able to deploy NCS with no adaptations to the model at Facebook. At Facebook, NCS is integrated into main code search tools (e.g. website and IDE), as a complement to the existing exact-match code search capabilities. Initially, the NCS results and the exact-match (grep-like) results were shown together. Sometimes though, developers were looking only for exact matches and got confused by the interleaving of results. Consequently, exact-match results (from the raw queries) were shown separately from the NCS results (from the tokenized queries).

3 Code Recommendation

Background NCS answers the first question that every developer has—*how do I do something*—by enabling natural language search directly over a large code corpus. Using NCS, a developer can find this API for writing code to load a bitmap image:

```
Bitmap bitmap = BitmapFactory.decodeStream(input);
```

But real-world coding does not end here. This line of code, if written and deployed, can run on millions of devices in a variety of different environments. The developer needs to make sure that the code will not crash on people's phones. Often this would mean adding additional code for safety check, error handling, and etc. In other words, the developer has a new question: Is there anything *else* to add?

Since there are millions of open-source repositories available, it is highly likely that given a particular task, some code already exists somewhere doing it. The challenge is: given a query code snippet and a large code corpus, how to find similar code and offer concise, idiomatic coding patterns to developers.

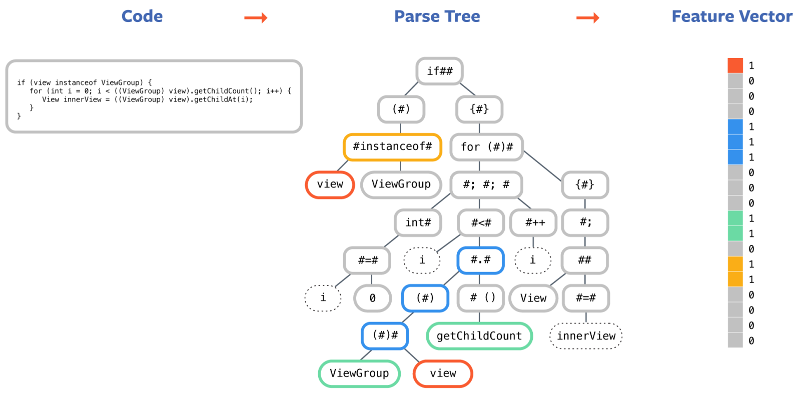
There exist many coding assistant tools that differ in their design and model: API recommenders suggest APIs to given a coding context, but do not provide usage examples to help with integration. API documentation tools provide helpful usage templates, but are limited to API queries, rather than arbitrary code snippets. Code-to-code search engines return exhaustive code matches, whereas our goal is to provide concise recommendations by clustering together similar results. Aroma is able to overcome all of these shortcomings.

How does Aroma work? Aroma indexes the code corpus by creating sparse vector representations of each method body. To do so, Aroma first parses the source code to get a simplified parse tree. Aroma uses this representation because it allows the rest of the algorithm to be language-agnostic.

Aroma then extracts features (presented in Figure 2) from the parse tree to capture the code structure and semantics. Aroma creates the feature set of a code snippet by aggregating the features of all tokens in that code snippet. After obtaining the vocabulary of all features, Aroma assigns a unique index to each feature, then converts the feature set to a sparse vector.

Given a query code snippet, Aroma runs the following phases to create recommendations:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/MS.2021.3061664, IEEE Software



```
if (view instanceof ViewGroup) {
   for (int i = 0; i < ((ViewGroup) view).getChildCount(); i++) {
      View innerView = ((ViewGroup) view).getChildAt(i);
   }
}
```

Figure 2: Features extracted by Aroma from a parse tree. The leaf nodes represent code tokens, which are extracted as token features; the internal nodes represent syntactic structures, and are concatenated with leaf nodes as syntactic features. The different colors represent different features extracted for the bottom-most node `view`. Refer to [12] for more details.

Feature-based Search. Aroma takes the query code snippet and creates a vector representation using the same steps in indexing. It then computes a list of top (e.g. 1000) candidate methods that have the most overlap with the query. This computation is very efficient by utilizing parallel sparse matrix multiplication.

Clustering. Aroma then clusters together similar-looking method bodies. Instead of showing similar or duplicate code, we want to create a single, idiomatic code recommendations from them. Aroma performs a fine-grained analysis on the candidate methods, and finds clusters based on similarities among the method bodies.

Intersecting. The final step is to create one code recommendation for each cluster of method bodies. The intersecting algorithm works by taking the first code snippet as the "base" code and then iteratively pruning it with respect to every other method in the cluster. Its goal is to return only the common coding idiom among the cluster, by removing extraneous lines that may be just situational in a specific method. Refer to our paper [12] for full algorithm details.

As a concrete example, suppose the following two code snippets are in one cluster, and that the first one is the "base" code snippet:

```
// Base snippet
InputStream is = ...;
final BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 2;
Bitmap bmp = BitmapFactory.decodeStream(is, null, options);
```

```
ImageView imageView = ...;
```

```
// 2nd snippet
BitmapFactory.Options options = new BitmapFactory.Options();
while (...) {
  options.inSampleSize = 2;
  options.inJustDecodeBounds = ...
  bitmap = BitmapFactory.decodeStream(in, null, options);
}
```

Both snippets contain a few lines of similar code, but also different lines specific to themselves. Aroma's intersection algorithm compares the base snippet with the second snippet, only keeping the lines that are common in both. It then compares these lines with the next method body. The remaining lines are returned as a code recommendation:

```
// A code recommendation
final BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 2;
Bitmap bmp = BitmapFactory.decodeStream(is, null, options);
```

Other code recommendations are created from other clusters in the same way. Aroma's algorithm ensures that these recommendations are substantially different from one another, so developers can learn a diverse range of coding patterns.

Results We instantiated Aroma on a large code corpus of Android GitHub repositories, and performed Aroma searches with code snippets chosen from the 500 most popular Stack Overflow questions with the Android tag. We observed that Aroma provided useful recommendations for a majority of these snippets. Moreover, when we used half of the snippet as the query, Aroma exactly recommended the second half of the code snippet in 37 out of 50 cases.

Developer feedback At Facebook, Aroma is integrated into the VS Code IDE. The developer selects a portion of code to be used as a query, and in response, Aroma presents a set of code recommendations. From Aroma's feedback workgroups, this integration received mixed feedback: developers were unsure about the use case. Is it a "teacher" to show better code? Is it warning about potential code duplication? In the end, developers were most interested in seeing *examples* of API usage. We have since developed a new tool for generating code examples [13] to address this need.

4 Bug Fixing

Background Large code repositories also come with a long history of commits (i.e. code changes), recording how the code base *evolved* into its current state. If we can find repetitive patterns using machine learning among these changes, then we can automate the routine work that engineers repetitively do. At Facebook, we have found that one common class of repetitive changes are bug fixes. Therefore we have built a tool called Getafix, that learns bug fixing patterns and automatically offers fix suggestions.

Getafix has goals similar to those of existing automated program repair techniques, but fills a previously unoccupied spot in the design space: single/few shot prediction of natural looking fixes, but for specific kinds of bugs. In contrast to generate-and-validate approaches [14] we focus on learning patterns from past fixes for specific bug types and leverage information known about bug instances (e.g. blamed variable). Getafix does not attempt to find generic solutions from any sort of ingredient space, or by generically mutating the code. Getafix tends to produce actual, human-like fixes by construction, as it takes nothing but past human fixes as inspiration.

How does it work? For clarity, we will focus on a specific type of bug that can crash Android apps: Java `NullPointerExceptions` ("NPE"). The following code snippet shows an example of a NPE and a possible fix:

```
public int getWidth() {
  @Nullable View v = this.getView();
  return v.getWidth();      // Bug: NPE if v is null
 return v != null ? v.getWidth() : 0;
 }
```

At Facebook we use the Infer [15] static analyzer to detect and warn about potential NPEs (line highlighted in red). From the Infer records, we identify commits that fix the potential NPE (line highlighted in green). We scrape hundreds of such bug fixing commits from the version history, and use them as training data for Getafix.

Edit extraction. In order to find repetitive patterns of bug fixes ("fix patterns") from this training data, Getafix splits commits into fine grained abstract syntax tree (AST) edits. Getafix first parses each file touched by a commit into a pair of ASTs: one for the source code prior to the changes made, and another for after the change. Getafix then applies a tree differencing algorithm similar to GumTree [16] to each pair of ASTs in order to predict the edits (insertions, deletions, moves, updates) that likely represent the difference. For the example fix above, Getafix will extract the following edit: `v.getWidth()` $\longrightarrow$ `v != null ? v.getWidth() : 0`.

Clustering. Getafix takes a data-driven approach, called *anti-unification*, by clustering the set of AST edits yielded by the previous step by similarity: It merges the most similar pair of edits in the set into a new edit pattern, abstracting away details only where necessary. Example:

Edit A:	`v.getWidth()`	$\longrightarrow$	`v != null`	`?`	`v.getWidth()`	`: 0`
Edit B:	`lst.size()`	$\longrightarrow$	`lst != null`	`?`	`lst.size()`	`: 0`
Anti-unification:	$\alpha.\beta$`()`	$\longrightarrow$	α `!= null`	`?`	$\alpha.\beta$`()`	`: 0`

Anti-unification has the desirable property that it merges edit patterns in the most information-preserving way possible. Getafix repeats this step as often as possible, putting the resulting edit pattern back into the set in place of its constituents, hence reducing the size of the set and allowing edit patterns to be merged and abstracted even further. This process results in a hierarchy of edit patterns, with the original edits as leaf nodes and increasingly abstract edit patterns closer to the roots.

Fix prediction. With such a hierarchy of fix patterns for NPE warnings, Getafix can automatically fix future warnings: When Infer produces a new, previously unseen, NPE warning, Getafix retrieves all patterns that are applicable from our hierarchy of fix patterns. Getafix then applies those candidate patterns to the code, generating candidate fixes, which are ranked statistically using a metric comparable to TF-IDF. To limit computational cost, one or at most a few of the top ranked candidate patterns are then validated (e.g. by running Infer and making sure the warning disappeared). The best passing candidate fix is offered to the engineer as a suggestion they can accept or reject at the click of a button. Getafix suggests only one fix to limit the cognitive load and provide a straight forward user experience. We do require a final human confirmation since Getafix uses statistical learning and ranking techniques, so there is no formal guarantee of correctness despite certain forms of validation. For more details about Getfix, refer to [17].

Results Of the Infer NPE warnings fixed by Facebook engineers since Getafix service has rolled out, 42% were fixed by accepting our fix suggestion, and in 9% of the cases, engineers wrote a semantically identical fix (which goes to show developers are very particular about the fix suggestions they accept).

Note that our pattern learning phase takes *any* set of changes as input, so a different scenario we have successfully started automating is the discovery and application of “lint” rules. Changes made in response to code review are often fixes to common anti-patterns that were pointed out by a reviewer, and finding and fixing these anti-patterns can be baked into a lint rule.

Developer feedback We show fix suggestions for warnings during code review, and in the IDE wherever possible. We found that warnings that came with a fix suggestion were more *actionable* and addressed (whether via accepting the suggested fix or hand-writing one) more often than plain warnings. Individual reactions ranged from ignoring our suggestions to expressing excitement about their level of sophistication in internal feedback groups.

We found that semantic equivalence is insufficient to our engineers and that syntactic differences do matter to them: For instance, we sometimes predict using a ternary conditional and in several cases observed developers adopt this fix, but negating the condition and swapping the “then” and “else” expression. At this point, the “accept with one click” experience we provide is ineffective, so we strive to suggest *natural looking* fixes exactly as our engineer expect, so syntax and even details like idiomatic white-space must be human-like. Our

ML-based approach learns patterns that look natural by construction (learnt from real fixes) and also learns how to rank among them in a principled way, which would be strenuous to replicate manually.

5 The Big Picture

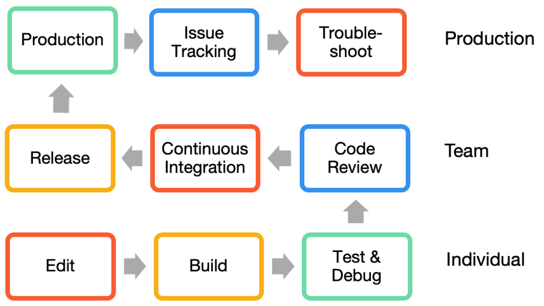


Figure 3: Common workflows in software development.

We now take a step back to discuss how these ML-based techniques fit in the broader picture of the software development process. In fact, these techniques have the potential of influencing not just writing or fixing code, but almost all stages of the software lifecycle.

Figure 3 shows a way to think about modern software development, as organized in three stages, recurring in a cycle (not depicted.) The workflow begins with an individual developer's work, which involves editing the code to implement new functionality, or in response to an issue, and making sure the code compiles and passes at least some lightweight quality control (e.g. linters, unit tests). Next is the team stage: once the developer is satisfied with a code change they are making, they send it in for code review, and perhaps simultaneously, more extensive testing and verification is kicked off. Either of these can require the jumping back into the individual workflow. Once code gets released and enters production, new issues can arise that were not caught by previous stages. The process must account for how such issues are tracked. The production stage would typically also include some telemetry that helps with bug isolation. Feedback from production kicks off a new cycle, starting again from the individual stage.

The previous sections talked about ideas that are primarily applicable in the code edit phase of the individual workflow. In addition to those ideas, the most visible developer-facing use of ML in code edit phase is *auto-complete*, which has been widely studied and deployed. More ambitiously, ML techniques can also help developers complete code via program synthesis.

Significant opportunities exist in the other states – for example, we had previously mentioned our own work on predictive regression test selection [1] and triaging crashes [2] — but a detailed discussion of these is outside the scope of this brief article. Here are, in our view, some of the most promising but relatively untapped opportunities for using ML pertinent to aspects of the team and production states.

- **Code Review.** Code reviewing, while widely regarded as essential for maintaining software quality, is also a significant time commitment for software engineers. ML techniques can help automate *routine* code reviews (such as formatting, best coding practices). More ambitiously, perhaps ML can also automatically *resolve* a routine code review comment.

- **Assessing the risk of a code change.** In principle, any code change increases the riskiness of an application. Arguably, the entire testing and verification pipeline exists essentially to reduce this risk. Can we design ML-based techniques that provide an quantitative assessment of the risk of a code change, complementing the usual testing and verification pipeline? Advanced here will impact both testing (by prioritizing tests related to riskier changes) and release management (by carrying out additional quality control for riskier code releases.). By comparison, techniques for assessing impact of a change (e.g. [18]) take a binary view of affected-ness, and due to limitations of static analysis, often would be overly pessimistic in their assessment.

- **Troubleshooting.** For widely deployed applications, customers send their feedback implicitly (telemetry, crashes) and sometimes explicitly by sending comments. The volume of this feedback can be huge. This is another area where ML can help in multiple ways: not only in triaging these reports, but clustering them to identify common issues, finding important clues from telemetry logs, and finding code changes that could be connected to the issue at hand.

With renewed interest in ML, and emerging uniformity of software development processes (common repositories, continuous integration and release), industry is ripe for absorbing these ideas in the mainstream. At Facebook, we certainly are transforming our development process to be as data-driven as possible.

References

[1] Mateusz Machalica et al. “Predictive Test Selection”. In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 91–100. DOI: 10.1109/ICSE-SEIP.2019.00018. URL: https://doi.org/10.1109/ICSE-SEIP.2019.00018.

[2] Rebecca Qian et al. *Debugging Crashes using Continuous Contrast Set Mining*. 2019. eprint: `arXiv:1911.04768`.

[3] Johannes Bader et al. *F8: Using Machine Learning for Developer Productivity*. 2019. URL: `https://developers.facebook.com/videos/2019/using-machine-learning-for-developer-productivity/`.

[4] Raphael Sirres et al. "Augmenting and Structuring User Queries to Support Efficient Free-Form Code Search". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 945. ISBN: 9781450356381. DOI: `10.1145/3180155.3182513`. URL: `https://doi.org/10.1145/3180155.3182513`.

[5] Hitesh Sajnani et al. "SourcererCC: Scaling Code Clone Detection to Big-code". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 1157–1168. ISBN: 978-1-4503-3900-1. DOI: `10.1145/2884781.2884877`. URL: `http://doi.acm.org/10.1145/2884781.2884877`.

[6] Saksham Sachdev et al. "Retrieval on source code: a neural code search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM. 2018, pp. 31–41.

[7] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *CoRR* abs/1607.04606 (2016). arXiv: `1607.04606`. URL: `http://arxiv.org/abs/1607.04606`.

[8] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *CoRR* abs/1310.4546 (2013). arXiv: `1310.4546`.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs". In: *arXiv preprint arXiv:1702.08734* (2017).

[10] Hongyu Li, Seohyun Kim, and Satish Chandra. *Neural Code Search Evaluation Dataset*. 2019. arXiv: `1908.09804 [cs.SE]`.

[11] Jose Cambronero et al. "When Deep Learning Met Code Search". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: ACM, 2019, pp. 964–974. ISBN: 978-1-4503-5572-8. DOI: `10.1145/3338906.3340458`. URL: `http://doi.acm.org/10.1145/3338906.3340458`.

[12] Sifei Luan et al. "Aroma: Code Recommendation via Structural Code Search". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019), 152:1–152:28. ISSN: 2475-1421. DOI: `10.1145/3360578`. URL: `http://doi.acm.org/10.1145/3360578`.

[13] Celeste Barnaby et al. "Exempla Gratis (E.G.): Code Examples for Free". In: New York, NY, USA: Association for Computing Machinery, 2020, pp. 1353–1364. ISBN: 9781450370431. URL: `https://doi.org/10.1145/3368089.3417052`.

[14] X. B. D. Le, D. Lo, and C. Le Goues. “History Driven Program Repair”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 213–224.

[15] C. Calcagno et al. “Moving Fast with Software Verification”. In: *NASA Formal Method Symposium*. 2015.

[16] Jean-Rémy Falleri et al. “Fine-grained and accurate source code differencing”. In: *ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014*. 2014, pp. 313–324. DOI: `10.1145/2642937.2642982`. URL: `http://doi.acm.org/10.1145/2642937.2642982`.

[17] Johannes Bader et al. “Getafix: Learning to Fix Bugs Automatically”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). ISSN: 2475-1421.

[18] X. Ren et al. “Chianti: A tool for change impact analysis of java programs”. In: *ACM SIGPLAN Notices* 39 (Oct. 2004), pp. 432–448. DOI: `10.1145/1035292.1029012`.