# The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-list Length on its Efficiency

Xianhao Jin
Virginia Tech
xianhao8@vt.edu

Francisco Servant
Virginia Tech
fservant@vt.edu

## ABSTRACT

Automatic code completion is a useful and popular technique that software developers use to write code more effectively and efficiently. However, while the benefits of code completion are clear, its cost is yet not well understood. We hypothesize the existence of a *hidden cost* of code completion, which mostly impacts developers when code completion techniques produce long recommendations. We study this hidden cost of code completion by evaluating how the length of the recommendation list affects other factors that may cause inefficiencies in the process. We study how common long recommendations are, whether they often provide low-ranked correct items, whether they incur longer time to be assessed, and whether they were more prevalent when developers did not select any item in the list. In our study, we observe evidence for all these factors, confirming the existence of a hidden cost of code completion.

## CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**; **Software maintenance tools**;

## KEYWORDS

Code Completion, Cost, IntelliSense

**ACM Reference Format:**
Xianhao Jin and Francisco Servant. 2018. The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-list Length on its Efficiency. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3196398.3196474

## 1 INTRODUCTION

Software developers rely on a large number of variables and Application Programming Interfaces (APIs) when coding programs. Regardless of how simple or complex these constructions are, developers cannot remember all of them, even if they need them for their daily coding tasks. In order to support developers for remembering the signature of other software artifacts, automatic code completion tools were proposed as an extension of the IDE. Automatic code completion tools provide developers, as they type, with recommendations of the signature of the code entities that they may be intending to call, in order to improve the effectiveness an efficiency with which developers write code.

Conventional wisdom generally recognizes the value of automatic code completion techniques and tools. Code completion tools are probably part of most software developers' tool box. However, little attention has been paid to scenarios in which code completion does not fulfill its purpose and instead even interferes with the developer's coding tasks. While it is easy to envision successful automatic-code-completion scenarios, these latter, less useful scenarios are harder to expect. Yet, for this same reason, the potential inefficiencies of code completion may have a more serious impact than one would expect.

In this paper, we study whether there are cases in which code completion behaves in a suboptimal manner, incurring an efficiency cost on developers. Our hypothesis is that the length of the recommendation list will negatively impact the efficiency of code completion as it increases. In our hypothesized scenario, a developer obtains a code-completion recommendation, but the recommendation list is so long that it takes a long time to inspect it, due to the many items that have to be assessed before reaching the correct one. Furthermore, this scenario may become worse if the recommendation list is so long that the developer gives up after some time investigating it and ends up not selecting any item — therefore not getting any benefit from code completion and instead having wasted time.

We perform an empirical study over the dataset provided for the 2018 MSR Mining challenge [4], which contains the IDE interactions for a set of real-world developers. In our study, we evaluate whether our hypothesized scenario takes place, and whether it incurs an efficiency cost for developers using code completion. Since we expect the length of the recommendation list to be a driving factor for the appearance of our hypothesized suboptimal scenario, we study the impact of the recommendation-list length on the efficiency of code completion.

In the results for our study, we found that in our studied dataset, the code completion technique often produced large recommendation lists of 250 items, that larger recommendation lists required longer explorations until the right item is found, that such explorations take longer for longer lists, and that the cases in which the developers did not choose anything from the recommendation list were also more prevalent for longer recommendation lists. In other words, in many cases, code completion recommendations were on the long end of the spectrum, had an efficiency cost for developers, and in many of those cases they provided no benefit for the developers. These results provide evidence for the fact that there is indeed a hidden cost to code completion that should be addressed by future code completion approaches.


*MSR '18, May 28–29, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196474

## 2 RELATED WORK

Many techniques have been proposed in the research literature to improve the accuracy of automatic code completion. For example, Proksch *et al.* extend an existing approach — the Best Matching Neighbor (BMN) algorithm — by adding context information, and their results show that their technique improved the prediction quality [6]. Similarly, Asaduzzaman *et al.* also proposed a novel technique called CSCC (Context Sensitive Code Completion) for improving the performance of API method call completion [1]. Raychev *et al.* managed to take advantage of statistical language models to improve accuracy [7]. Another related area of research aims to improve the quality of the datasets with which code completion is evaluated. For example, Proksch *et al.* found that an evolving context that is often observed in practice has a major effect on the prediction quality of recommender systems, but is not commonly reflected in artificial evaluations[5]. Romain Robbes *et al.* tried to improve code completion based on recorded program histories defining a benchmarking procedure measuring the accuracy of a code completion engine[8]. Other researchers, Ghafari and Moradi, built a framework to help the community to conduct systematic studies to gain insight into how much code recommendation has so far achieved, in both research and practice[2].

To the extent of our knowledge, our study is the first of its kind with the goal of empirically understanding the efficiency cost of suboptimal recommendations in code completion.

## 3 RESEARCH QUESTIONS

**RQ1: How common are different recommendation lengths?** The answer to this research question will allow us to adjust our expectations for how often the recommendations produced in the field end up in the longer end of the spectrum. This will also give us a sense of how common "potentially costly" recommendations are. If the produced recommendation lists often fall in the shorter end of the spectrum, then the cost of code completion for developers would be small, since "short" recommendation lists can be assessed efficiently. Otherwise, if "long" recommendation lists are the norm, then developers are actually wasting valuable time in assessing these lists, and there is an efficiency cost to using code completion.

**RQ2: How does the recommendation length affect the rank of the correct recommendation?** The answer to this research question will allow us to determine whether increasing lengths of the recommendation-list decrease the accuracy of code completion. This factor is interesting to study because less accurate recommendation lists — containing the correct item in a low rank — take longer to investigate and are therefore more costly. Even if long recommendations were common, their cost would not be very high if they recommend the right item at the top positions of the list. In other words, if the recommendation is very accurate, the length of the recommendation list potentially does not matter. Otherwise, if long recommendation lists are the norm and the accuracy of those long lists is low, then we hypothesize that developers will waste time assessing them until they identify the correct item inside.

**RQ3: How does the recommendation length affect the time spent evaluating the recommendation?** The answer to this research question will allow us to adjust our expectations of how long it takes to evaluate the different lengths of the recommendation lists. Regardless of the recommendation-list length and the ranking of the right item, developers may be very fast to assess recommendation lists, which would reduce the cost of code completion. Otherwise, observing that developers indeed take time to assess the code-completion recommendations more strongly validate the fact that there is a cost to code completion.

**RQ4: How does the recommendation length affect the likelihood of the developer making a selection?** The answer to this research question will allow us to understand how often recommendation lists are so costly to assess — for their low accuracy, high length, or any other factor — that developers decide not to use them and do not make a selection from the recommendation list. Understanding how common this event is will also help us the impact of the most costly aspect of our hypothesized scenario: spending time assessing the recommendation list, but ultimately desisting and getting no value from it.

## 4 METHOD

**Data Preprocessing.** We analyze the MSR Mining Challenge dataset [4] and extract from it all the events that correspond to code completion. This dataset was created by capturing the IDE usage of many software developers that used Visual Studio. Thus, the specific code-completion engine that we studied in this paper is IntelliSense. Next, we explain how we process code-completion events to study each of our individual research questions.

**RQ1: How common are different recommendation lengths?** To obtain the length of the recommendation list, we extract the proposal list information from the code completion event. Then, we plot the median percentage of code-completion recommendations that were included in the dataset that had each specific length. Our goal with this plot is to understand the relative prevalence of each individual recommendation-list length.

**RQ2: How does the recommendation length affect the rank of the correct recommendation?** For this question, we use the events that we extracted for the former research question. However, for this case, we remove those cases from the dataset for which the developer made no selection, or when there was an empty selection or multiple selections. We removed these latter two cases because we could not explain them. Then, for each remaining code completion event, we assessed the right item from the list as the item that the developer selected. We measure the rank of the selected item within the list. Finally, we plot the median position in which the right item was recommended for a recommendation-list length.

**RQ3: How does the recommendation length affect the time spent evaluating the recommendation?** For this research question, we analyze the same code completion events as for RQ2. We measure the "SelectedAfter" object from each code completion event to represent the number of seconds that the recommendation list was shown until the developer selected something. Then, we plot the median "SelectedAfter" time for each recommendation-list length, to understand the time that developers take to evaluate recommendations, and whether the recommendation length has any impact on it.

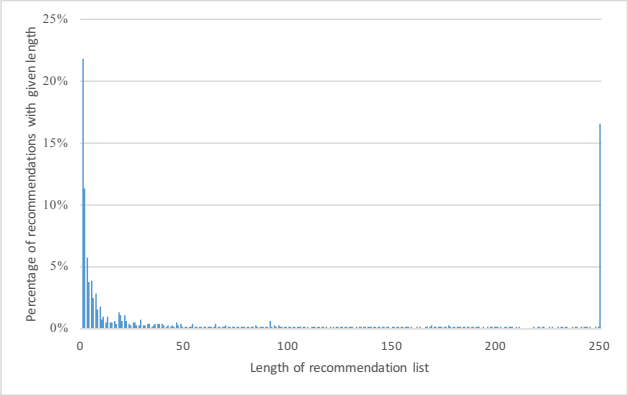


**Figure 1: Percentage of completion recommendations**

**RQ4: How does the recommendation length affect the likelihood of the developer making a selection?** This time we analyze all the code completion events (regardless of whether the developer selected something or not). Again, we filter out the cases when multiple items were selected, because we could not explain them. Thus, our analyzed code completion events may only either have one item or no items selected. Finally, for each recommendation-list length, we plot the percentage of recommendations of that length for which the developer did not make a selection.

## 5 RESULTS

### 5.1 RQ1: How common are different recommendation lengths?

Figure 1 shows the number of recommendations provided by IntelliSense for each recommendation-list length. The X axis lists recommendation-list lengths 1–250 — 250 is the maximum length that IntelliSense used. The Y axis represents the percentage of recommendations provided by IntelliSense for a given recommendation-list length. The mean value is 50.37, median value is 4, the mode is 1 and the standard deviation is 87.95. In this figure, we can observe that the most common recommendation list length was 1 (20% of cases), with each subsequent length being less and less common. Furthermore, most recommendation-list lengths stayed in the lower-end of the length spectrum. Still, it is also worth noting that a large number of recommendations had a large length of 250 – around 16% of recommendations. In fact, 250 was the second most-common recommendation-list length. This observation tells us that, while IntelliSense does a great job of frequently providing short recommendations — many with length lower than 10, it still produces a large number of recommendations of large lengths — with length 250 being extremely frequent. As a consequence, the cases in which we hypothesized that developers could be losing inefficiency due to long recommendation lists are much more frequent than they could have been expected.

### 5.2 RQ2: How does the recommendation length affect the rank of the correct recommendation?

Figure 2 shows the selected item's position within the recommendation list provided by IntelliSense for each recommendation-list length. The X axis lists recommendation-list lengths, and the Y axis represents the median position of the selected items within a recommendation list for a given recommendation-list length. In this figure, we can observe that for short recommendations, the position of the right item within the recommendation stays at a low number. However, the position of the right item within the recommendation list increases rather steadily as the length of the recommendation-list grows. This observation again shows that there are many recommendations for which IntelliSense does a great job, i.e., it produces many short recommendations where the correct item is highly-ranked. However, there are still many recommendations for which the correct item can only be found after assessing a large number of other items. In fact, for the second most common recommendation length (250), the median position is around 15, which is still pretty high. As a consequence, there were a large number of recommendations for which the right item was recommended at a rather high position (higher than 10). This observation means that in many cases developers will have to spend some time assessing multiple candidates before obtaining the benefit of code completion.

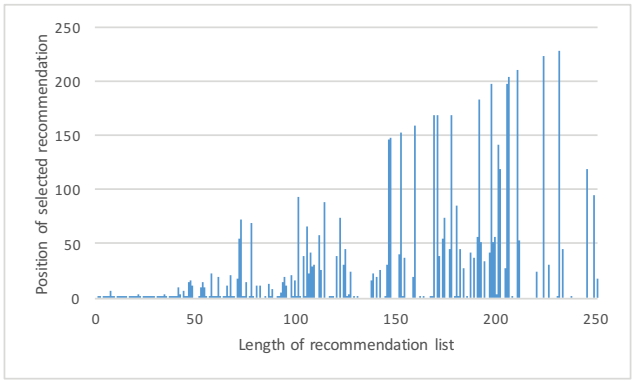


**Figure 2: Median position of the correct item**

### 5.3 RQ3: How does the recommendation length affect the time spent evaluating the recommendation?

Figure 3 shows the time spent evaluating the recommendation for each recommendation-list length. The Y axis now represents the median value of the time spent evaluating the recommendation for a given recommendation-list length. In this figure, we can observe that the time spent evaluating the recommendation is short for those recommendation-lists with a short length, and that it increases with the recommendation-list length. While this increasing trend is not very steep, Figure 3 also shows that for the majority of recommendation lengths, there is some time that needs to be spent assessing the recommendations. We should note that, even though each individual time reported in this figure is low, it accumulates very quickly over time, because developers constantly obtain code-completion recommendations, potentially having to assess a large number of them daily. This observation validates our findings in RQ2, since we observed that increasing recommendation-list lengths also increased the position in which the right item was recommended. Such an increased position would involve longer time by developers inspecting the recommendation, which is what

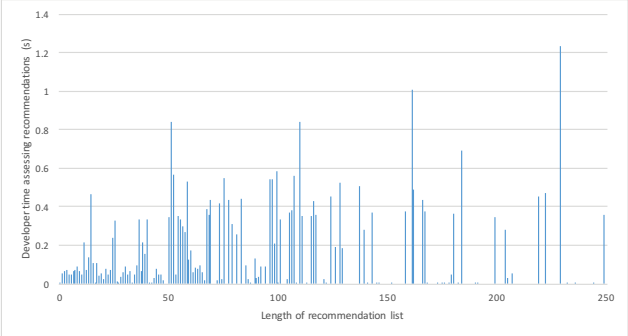


**Figure 3: Median time spent inspecting the list**

we observe for RQ3. This observation also provides evidence for the cases of inefficiency that we hypothesized — in which as recommendation lengths grow, developers developers spend longer time evaluating them.

## 5.4 RQ4: How does the recommendation length affect the likelihood of the developer making a selection?

Figure 4 shows the percentage of recommendations for which the developer did not select any item, given a recommendation length. In this figure, we can observe an upward trend in the percentage of recommendations for which no selection was made, which grows with the recommendation-list length. A second observation is that the percentage of recommendations with no selection is very high for most recommendation lengths — with the exception of some cases for which we did not have many data points (as can be observed in Figure 1). The reason for this second observation is that IntelliSense works automatically — it provides recommendations without developers needing to request them, so it is natural that a large number of them would be automatically ignored. For that reason, the important observation is the upwards trend in the graph. This observation tells us that as the recommendation-list length grew longer, developers were less and less inclined to select something from it. Thus, long recommendations were more likely to leave developers obtaining no benefit from code completion for not having selected anything. This final observation also provides evidence for our hypothesized scenario in which developers not only may be spending time assessing large recommendations, they are also not obtaining its benefit — no selected recommendation — in many cases as well.

# 6 CONCLUSION AND FUTURE WORK

We hypothesized that there may be a hidden cost to code completion, i.e., cases in which code completion may not be as helpful as we could intuitively envision. We hypothesize that, in such cases, developers may be spending time assessing long recommendations in which the right item is only found after assessing many items, and that they may eventually get discouraged and not choose anything from the recommendation, ultimately losing the benefit of code completion. We performed an empirical study over a dataset of code completion events, in which we observed evidence for all

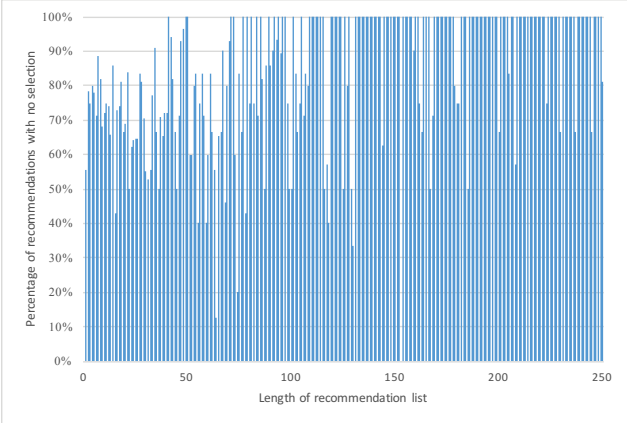


**Figure 4: Recommendations in which no selection was made**

the elements of our hypothesis. In many cases, code completion: (RQ1) provided large recommendations, (RQ2) that contained the right item far down its list, (RQ3) which took increasing time to inspect with increasing length, (RQ4) and provided recommendations for which developers did not end up making a selection. In the light of this evidence, we conclude that the hidden cost of code completion grants further study in future work. In the future, we will study other code completion algorithms (besides IntelliSense) to learn whether our findings will be replicated for them. We will also perform human studies to better understand the qualitative aspects of the cost of code completion, e.g., barriers for adoption or frequent usage that may not be intuitive from studying a dataset — *e.g.,* we anecdotally heard that some developers trigger IntelliSense just to learn about APIs, which is a behavior that would be hard to identify by only observing the data. Finally, we provide a replication package for this study [3].

# REFERENCES

[1] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. 2014. Cscc: Simple, efficient, context sensitive code completion. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 71–80.

[2] Mohammad Ghafari and Hamidreza Moradi. 2017. A framework for classifying and comparing source code recommendation systems. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 555–556.

[3] Xianhao Jin and Francisco Servant. 2018. The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-list Length on its Efficiency. (March 2018). https://doi.org/10.5281/zenodo.1199697

[4] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched Event Streams: A General Dataset for Empirical Studies on In-IDE Activities of Software Developers. In *Proceedings of the 15th Working Conference on Mining Software Repositories*.

[5] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. 2016. Evaluating the evaluations of code recommender systems: A reality check. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 111–121.

[6] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 3.

[7] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Acm Sigplan Notices*, Vol. 49. ACM, 419–428.

[8] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17, 2 (2010), 181–212.