A Machine Learning Based Automatic Folding of Dynamically Typed Languages

Nickolay Viuginov
Faculty of Information Technologies and Programming
ITMO University
St. Petersburg, Russia
JetBrains
St. Petersburg, Russia
viuginov.nickolay@gmail.com

Andrey Filchenkov
Faculty of Information Technologies and Programming
ITMO University
St. Petersburg, Russia
afilchenkov@itmo.ru

ABSTRACT

The popularity of dynamically typed languages has been growing strongly lately. Elegant syntax of such languages like *javascript*, *python*, *PHP* and *ruby* pays back when it comes to finding bugs in large codebases. The analysis is hindered by specific capabilities of dynamically typed languages, such as defining methods dynamically and evaluating string expressions. For finding bugs or investigating unfamiliar classes and libraries in modern IDEs and text editors features for folding unimportant code blocks are implemented. In this work, data on user foldings from real projects were collected and two classifiers were trained on their basis. The input to the classifier is a set of parameters describing the structure and syntax of the code block. These classifiers were subsequently used to identify unimportant code fragments. The implemented approach was tested on JavaScript and Python programs and compared with the best existing algorithm for automatic code folding.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; *Syntax*; • **Computing methodologies** → *Supervised learning by classification.*

KEYWORDS

Dynamically typed languages, JavaScript, Python, Automatic Folding, Source code analysis, Abstract Syntax tree.

ACM Reference Format:
Nickolay Viuginov and Andrey Filchenkov. 2019. A Machine Learning Based Automatic Folding of Dynamically Typed Languages. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '19), August 27, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3340482.3342746

MaLTeSQuE '19, August 27, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-6855-1/19/08…$15.00
https://doi.org/10.1145/3340482.3342746

1 INTRODUCTION

Dynamically typed languages have some specific syntax constructions such as late binding, down-casting, and reflection, which allow for great possibilities [1]. However, this comes alongside several drawbacks. First of them, a codebase becomes entangled making thus code investigation more difficult in comparison with the statically typed languages such as Java or C++ [2]. Another downside of its dynamic features is a drastic static analysis performance reduction due to the inability to resolve some symbols reliably.

These reasons motivated growing usage of alternative methods for code analysis of such languages. One of promising and rapidly developing approaches is the data-driven approach involving usage of machine learning methods. It was applied in various areas of program code analysis, such as:

- Language modeling [3].
- Code Suggestion and Completion [4].
- Program Translation [5].
- Program Repair and Bug Detection [6].
- Code Optimization [7].
- Clone Detection [8].
- Folding, Code Summarization [9].

Machine learning models are actively used in integrated development environments and code analysis tools such as IntelliJ Platform products and plugins[1], Microsoft Visual Studio[2], Visual Studio Code[3], number of language server protocol [10] servers and plugins for Eclipse[4].

Due to the growth of codebases and the number of libraries in use, sometimes it takes even more time to investigate code than to write it[11]. Folding helps to significantly reduce the source code and leave only the most important parts of it expanded. This allows a user of a library or an API finding the functionality of interest much faster, investigating the architecture of API and figuring out, how to use it. Partially, this problem can be solved by means of reading documentation, but quite often code have to be investigated in more detail, for example, if it is necessary to refactor it or find some bugs [12].

In this paper, we focus on the problem of the automatic code folding (ACF). ACF is a feature presented in most modern IDEs and code editors that allows hiding any code block. Blocks of imports,

[1] https://www.jetbrains.com/
[2] https://www.microsoft.com/ru-ru/download/developer-tools.aspx
[3] https://code.visualstudio.com/
[4] https://www.eclipse.org/

method bodies, any kind of loop, a lambda and any other type of block can be folded. An example of a code folding is provided in Fig. 1.

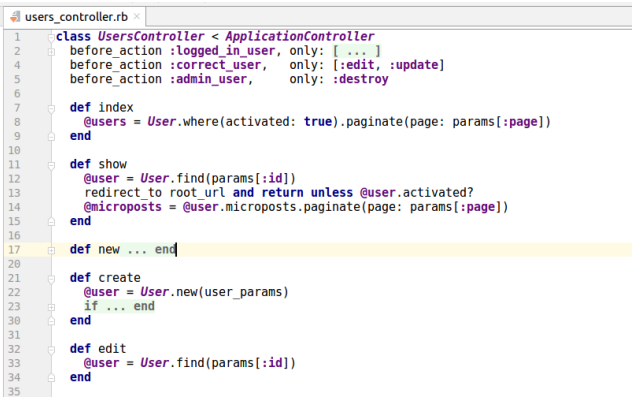

Figure 1: Examples of code folding in Ruby script, lines 17–19 and 23–29 are folded.

The ACF problem can be formally stated as follows. Given a program and a percentage by which it is required to be reduced, an algorithm has to decide, which code blocks should be folded. We want the most important code fragments to remain expanded after folding. In fact, an even broader problem is solved, the problem of comparing two code fragments by importance in context of a class.

In this paper, we approach this problem by reducing it to a prediction problem, namely, classification. For each code block, we predict if it can be folded.

First, we collected a dataset of user foldings from public github projects. Foldings from 335 JavaScript and 304 Python projects were used.

Second, we describe a code block with features representing its structural properties. The set of features we use satisfy the restrictions of privacy: no feature is based on any information that violates the privacy[5] of codebase and its developer. For this reason we didn't use information about control-flow graphs, data-flow graphs and resolution as a features.

Finally, we train a classifier predicting if a code block should be folded or not. This classifier will be used by a greedy algorithm, which iteratively finds the most "unimportant" block of code, where "unimportancy" means the probability of a block being folded.

The rest of the paper is organized as follows Section 2 contains a short description of abstract syntax tree formalism and code embedding. In Section 3, we describe related work. The dataset we have collected is presented in Section 4. Section 5 contains a description of a different models, used for classification of foldable blocks. Section 6 contains evaluation of the approach for ACF and its comparison with a baseline algorithm.

[5] https://eugdpr.org/

2 PRELIMINARY

2.1 Abstract Syntax Tree

Abstract syntax tree (AST) is a representation of a source file written in a programming language, which represents the syntactic structure of it. AST is a compressed parse tree. Its vertices are operators or syntactic blocks, and child vertices represent operands or parts of a syntactic block. An example of AST is presented in Fig. 2.

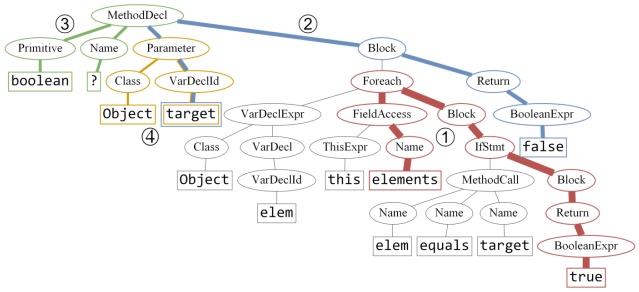

Figure 2: AST of a Java method with leaf-to-leaf paths marked

Similar trees are used by interpreters to search for syntax errors in a program and then convert source code of the program into bytecode or another internal representation [13]. AST and its derivatives such as Program Structure Interface Trees (PSI trees) [14] are actively used for the implementation of various analyzes and inspections. Moreover such trees found great use in machine learning on source code.

2.2 Code Embedding

In recent years, both natural and artificial language processing fields experience growing popularity of the embedding-based approach started with Word2Vec [15] for natural language. This model has many applications in various tasks, such as document classification, sentiment analysis and text translation.

By analogy with Word2Vec, for source code analysis was purposed a code2vec [16], a technique of obtaining a continuous distributed vector embedding from a code fragment. It was proposed to consider code as a set of Path-Contexts. Path-Context is a vector of AST nodes on a path from one leaf node to another. AST node paths highlighted on a Figure 2 with different colors. Such a technique was chosen because this type of vectors quite well reflect the syntactic and semantic features of the program and in addition the same vectors may occur multiple times in large corpora of programs. Thus they satisfy the bias–variance trade-off.

Embeddings produced with code2with were shown to be useful in solving a large number of applied problems such as automatic generation of variable and method names[6] and duplicate search.

[6] https://code2vec.org/

3 RELATED WORKS

3.1 Predicting Variable Types in Dynamically Typed Programming Languages

Identifying types of expressions and variables is one of the most important tasks in the analysis of dynamically typed languages. System for finding bugs connected with types mismatch, completion, navigation to declaration and finding usages require a good type inference system. But the lack of explicit information about the types and number of syntactic constructions specific to dynamically typed languages, which can significantly reduce the time to develop, but make the task of type inference algorithmically insoluble.

Recently, variety of ways was applied to the task of type inference, one of them was proposed in paper "Predicting Variable Types in Dynamically Typed Programming Languages"[17]. In the paper it was proposed to use a 2nd order Inside Outside Recursive Neural Network[18] over the abstract syntax tree. In the paper, authors considered 21 basic types in Python programming language and trained a model, which predicts the type of expressions. For each node of an AST two embeddings were calculated. First of them is an inner representation of a node. It is computed bottom-up and represent the content of an AST node. The second one in an outside representation. It is worth noting that the authors of the work made additional transformations of an AST tree. It was noted that the number of child nodes can be very large, therefore, to speed up learning and reduce the memory usage and data transfers, it was decided to set the upper bound on the number of node's children. According the dependence of accuracy on test dataset on maximum number of children it is clear that the optimal upper bound is 20.

The model was trained 24 labeled AST trees and tested on 12 AST and reached about 44.33 accuracy on a test corpus.

3.2 TASSAL

The only existing solution of the ACF problem we found in scientific literature is a Tree-based Autofolding Software Summarization Algorithm (TASSAL) [19]. Its authors presented two different algorithms. The first algorithm use a vector space model, vectors of which correspond to the frequencies of different tokens. The second algorithm use topic modeling [20] approach that is well known in natural language analysis. Like in most other algorithms for the source code analysis AST is built for a given file. Then an intermediate representation is built on on top of the AST. In this intermediate tree, only AST nodes corresponding to foldable nodes are left Fig. 3.

After such transformation, the ACF problem is reduced to the following. Find a set of vertices with at least given sum of line ranges. After folding all code fragments corresponding to these vertices, the size of file in terms of lines of code will decrease by a given ratio. For each foldable node it is needed to obtain a list of tokens, tokens like operators, delimiters and other syntax contractions are not so informative from semantic point of view so they will be skipped.

3.3 Vector Space Model

Vector Space Model (VSM) is actively used in various tasks, including the analysis of natural languages. This method implies using a

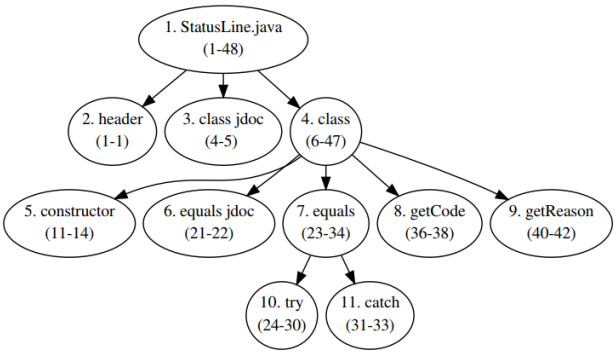

Figure 3: Tree of foldable nodes

continuous-valued vectors to represent text fragments. In the paper it was proposed to use log term frequency vectors (log tf).

3.4 TopicSum

In TopicSum approach, a scoped topic model is used that extends basic latent Dirichlet allocation model [21] to handle several layers of topics. There are five kinds of topics in this approach:

- topic for each file;
- topic for each project;
- three per-language topics, which are shared between projects. They correspond to common Java tokens, common Javadoc tokens and common header comment token.

This set of topics is well suited for this task, as it allows to separate topics that are specific to the given language or to some of the popular libraries. This allows to separate code snippets that are boilerplate or just a part of configuration of some framework like AngularJS or Ruby on Rails. This code is not so meaningful and it is more for configuration rather than carry some core functionality. In addition, it becomes possible to separate project-specific tokens from file-specific tokens. Thus file-specific tokens are the most meaningful in terms of this model. The topic model is represented in Fig. 4.

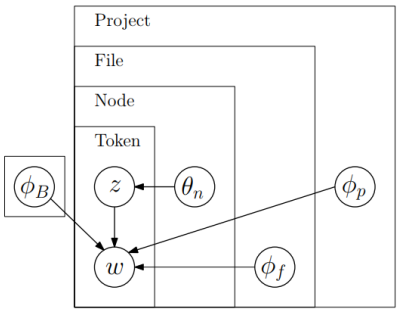

Figure 4: Graphical model depiction of the TopicSum content model [19]

To get the list of nodes to fold was used a greedy algorithm. For each node in a tree of foldable nodes we wrote out the line ranges. Then we iteratively get subtrees, whose embeddings on one of the

metrics are maximally separated from the embedding of the whole file. This procedure is repeated until the required number of lines is folded. In case of vector space model a simple cosine distance is used as a metric, and in case of topic modeling Kullback-Leibler divergence [22] is used to compare a probability distributions of a file and of a given folding region.

As a result of testing the model on 96 files with 50% compression ratio labeled manually, it turned out that the Topic model approach outperforms the VSM model. TASSAL TopicSum accuracy is 0.77, and TASSAL VSM accuracy is 0.65. So, the topic model is more accurate, but it is also much slower, as it requires to be retrained after adding a new project or file.

4 DATASETS

We first have to collect a dataset for training a predictive model. In order to do so, we decided to use projects developed using IntelliJ Platform. Every such project has a service `.idea` folder containing `workspace.xml` file, which has information about user's custom foldings. This utility folder is often added to the gitignore, but there are more than 500.000 repositories with such xml's on github.

For each file, we obtained a set of foldable elements using an Intellij-community class `FoldingDescriptor`[23]. Those of them, that were mentioned in `workspace.xml` file were folded by user and will be considered as folded. Such files contain sections corresponding to the files from project, and these sections contain lists of foldable regions in form of a start and end offsets pairs. Below is a fragment of the file `github.com/burningself/mobile-project`, which contains descriptions of foldings for `bijia.js` file. Each `element` xml-tag inside `folding` corresponds to one folable region, positions of start and end of which is stored in `signature` property Fig. 5.

```
<file leaf-file-name="bijia.js" pinned="false" current-in-tab="true">
  <entry file="file://$PROJECT_DIR$/js/bijia.js">
    <provider selected="true" editor-type-id="text-editor">
      <state vertical-scroll-proportion="0.080996886">
        <caret line="7" column="26" selection-start-line="7" selectior
        <folding>
          <element signature="e#0#40#0" expanded="false" />
          <element signature="e#171#301#0" expanded="false" />
          <element signature="e#338#455#0" expanded="false" />
          <element signature="e#496#920#0" expanded="false" />
          <element signature="e#958#1234#0" expanded="false" />
          <element signature="e#1299#1614#0" expanded="false" />
          <element signature="e#1650#1961#0" expanded="false" />
          <element signature="e#2002#2082#0" expanded="false" />
        </folding>
      </state>
    </provider>
  </entry>
</file>
```

Figure 5: XML with folding regions example from burningself/mobile-project GitHub repo.

The resulting dataset consists of 335 `JavaScript` and 304 `Python` repositories. The average number of files in a project was 536.

5 FOLDABLE BLOCKS CLASSIFICATION

5.1 Features and Classifiers

For each AST node corresponding to a foldable element, we use the following features describing a syntax structure of a node's subtree. Used language-agnostic features:

- Depth of the code block's root node
- Height of the code block's subtree
- Number of AST nodes in the subtree
- Number of `if` statements in the subtree
- Number of `for` statements in the subtree
- Number of `while` statements in the subtree
- Number of `return` statements in the subtree
- Number of assignments
- Number of blocks in the subtree
- Number of literal expressions in the subtree
- Number of lines in the block
- Number of chars in the block
- Length of longest identifier in the subtree
- Number of comments in the block
- Number of binary expressions in the block
- Type of foldable element (categorial)
- Type of parent PSI element (categorial)

The number of each type of PSI element was considered as a language-agnostic set of features, below you can see some of them: Python:

- Number of PyBreakStatement
- Number of PyAssertStatement
- Number of PyTryPart
- Number of PyYieldExpression
- Number of PyStarArgument

JavaScript:

- Number of JSThrowStatement
- Number of JSThisExpression
- Number of JSTryStatement
- Number of JSVarStatement

Some of the features are categorical (e.g type of foldable AST node, type of parent AST node). This is why we decided to use Random Forest and CatBoost as classifiers. It is important to note that each classifier returns not only is a block can be folded or not, but a probability that a block can be folded, which we will use in the next Section.

We trained classifiers for JavaScript and Python independently. We used both language-agnostic and language-specific features.

For training Random Forest, we used the following hyperparameters selected with grid search Table 1.

Table 1: Random Forest hyperparameter values used

max_depth	12
max_features	0.7
min_samples_leaf	1
max_samples_split	3
n_estimators	115

For training a CatBoost model, we used the following hyperparameters selected with Hyperopt [24] library Table 2.

Table 2: CatBoost hyperparameter values used

l2_leaf_reg	5.9053685596093874
depth	7
iterations	100
loss_function	Logloss
eval_metric	AUC
learning_rate	0.7290697164191682

5.2 Feature Importance

Below is a table of 12 most important features for JavaScript and Python models, obtained with CatBoost feature importance Table 3.

Table 3: Top 12 features by importance

Python		JS	
Root Depth	0.3315	Root Depth	0.1575
Number of chars	0.07	Number of chars	0.087
Number of lines	0.0457	Subtree height	0.07
Number of tokens	0.029	Longest identifier	0.0554
Size	0.0278	Number of tokens	0.038
#whitespaces	0.026	#references	0.0346
#binary expressions	0.024	Size	0.0345
Subtree height	0.023	#literal expressions	0.0337
#references	0.0224	#whitespaces	0.033
#leaf elements	0.0193	#leaf elements	0.0314
Longest identifier	0.0174	Number of lines	0.03
#literal expressions	0.0164	Type of element	0.027

5.3 Classification Results

We used 20% of collected dataset for test.

The obtained F1-score of Random Forest is 0.6286 for Python and 0.6377422 for JavaScript. The size of serialized JavaScript model is 3 MB. The size of serialized Python model is 2.3 MB. For comparison, TopicSum model for JS is 112.5 MB, for Python is 172.5 MB.

The obtained ROC-AUC score of CatBoost is 0.84 for Python and 0.7771 for JavaScript. The size of serialized model is 218kB for Python and 787kB for JavaScript.

We can conclude that:

(1) Language-agnostic features are much more important than language-specific ones.
(2) The size of resulting models were much smaller than size of a serialized TASSAL model. Moreover, the size of a model will not grow hard with increase of datasets and number of considered projects.
(3) Our model works much faster, because computation of feature values in performed in a single walk through the PSI tree structure, while the TopicSum model uses a time-consuming approximation called collapsed Gibbs sampling [25] to obtain the token distribution.

6 AUTOMATIC FOLDING OF CODE

6.1 Folding Algorithm

For determining code fragments that should be folded given a compression ratio, we implemented an algorithm similar to the one that was used in TASSAL. First, we evaluated the number of characters that should be folded and initialized budget with this value. After that we traverse the tree of foldable elements looking for a elements, which contain less chars than the current budget. Then we select one of this elements which has the best probability of being foldable to length ratio. After choosing such foldable element, the current budget is decreased by the size of this element. These steps are reproduced until it is possible to find a foldable element that suits the current budget.

6.2 Folding Algorithm Evaluation

The quality of the proposed algorithm is evaluated by comparing an Intersection Over Union (IOU) of a ground truth folding ranges with ranges derived from the model.

For testing models, we selected files from a set of github repositories with compression ratio from 30% to 70% and. Thus, 50 python and 54 JavaScript files were selected.

Mean IOU values for Python and JavaScript files are presented in Table 4.

Table 4: IOU comparison on Python and JavaScipt files

	Python	JS
TopicSum	0.53	0.5
Random Forest	0.53258	0.529
CatBoost	0.53257	527

For 10 files, TopicSum model and Random Forest had the same IOU values, for 22 files TopicSum was better, and for 22 files Random Forest was better. Table 5 shows the comparison of TASSAL and Random Forest performances on 13 files and the corresponding compression ratios.

When using the obtained solution in the production, it is possible to collect information about the expansions of the blocks of code that were folded according to our algorithm. This data can be used to evaluate quality of a solution.

7 CONCLUSION

The paper describes a production-ready solution of the automatic code folding problem. A good algorithm for automatic folding will significantly speed up investigation of a new libraries, new API or finding bugs on account of hiding an unimportant parts of code.

To solve the problem, we present a classifier, which analyzes the structural and syntactic features of programs and blocks of code to determine which blocks of code should be folded.

Table 5: IOU comparison of RandomForest, TopicSum and CatBoost models

Github Repo / File name	Compression	RF IOU	TS IOU	CB IOU
lsh179/FrontendCpmments core.js	0.348	**0.39**	0.353	0.364
thadino/Keebin DataBaseCreation.js	0.53	**0.78**	0.765	0.72
scottiand/3DChess Shapes.js	0.435	**0.441**	0.435	0.257
gitzhishenghuang/cartroom routers.js	0.333	**0.336**	0.295	**0.336**
Nolyo/Maps main.js	0.489	0.315	0.325	**0.381**
pointworld/vue-analysis create-element.js	0.482	**0.552**	0.551	0.511
stuartcboon/WebGameDev gameScene.js	0.552	0.69	**0.696**	0.575
Catherine429/mspaint constructor.js	0.484	**0.911**	0.504	0.503
djonov/mytestingnode users.js	0.176	0.4	**0.414**	0.16
notmenotmine/Trains getInlineWithTrains.js	0.446	**0.83**	0.319	0.319
lsh179/FrontendComments indexModel.js	0.5	**0.437**	**0.437**	**0.437**
caixiaojia/zepto_cai zepto_bak.js	0.628	**0.698**	0.697	0.695
Mujib517/Movies.api.express moviesControllerSpec.js	0.47	**0.497**	**0.497**	**0.497**

The approach was tested for JavaScript and Python languages and outperformed the baseline model TASSAL on the test dataset. The proposed approach resulted in much smaller size of serialized model, which does not need to be retrained to work with new projects. In addition, model can be used for different languages, since the most important features turned out to be language-agnostic.

In the future, we plan to evaluate the presented algorithm by implementing it in a plugin for IntelliJ products. To achieve this, it is planned to:

(1) Publish a plugin with a bundled model for automatic code folding.
(2) Collect a big amount of data about users' foldings and use it to improve the model.
(3) Test some new features such as "time since last modification of the block of code" or binary feature showing if the block is from a library or user code.
(4) Implement generation of a summarization for a folded blocks.

ACKNOWLEDGEMENTS

The authors would like to thank Vitaly Khudobakhshov, Nikita Povarov and Vitaliy Bibaev for the idea of work and assistance in studying the topic.

This research is supported by the Government of the Russian Federation, Grant 08-08.

REFERENCES

[1] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.
[2] Magnus Madsen. *Static Analysis of Dynamic Languages*. PhD thesis, Department of Computer Science, Aarhus University, 2015.
[3] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.
[4] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.
[5] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Advances in Neural Information Processing Systems*, pages 2547–2557, 2018.
[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. Learning a static analyzer from data. In *International Conference on Computer Aided Verification*, pages 233–253. Springer, 2017.
[7] Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787*, 2016.
[8] Niccolò Marastoni, Roberto Giacobazzi, and Mila Dalla Preda. A deep learning approach to program similarity. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, pages 26–35. ACM, 2018.
[9] Qingying Chen and Minghui Zhou. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 826–831. ACM, 2018.
[10] Language server protocol. https://microsoft.github.io/language-server-protocol/.
[11] Jamie Starke, Chris Luce, and Jonathan Sillito. Searching and skimming: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 2009.
[12] Mik Kersten and Gail C Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.
[13] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.
[14] Intellij platform sdk devguide: Psi elements. https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html.
[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
[16] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.
[17] Abhinav Jangda and Gaurav Anand. Predicting variable types in dynamically typed programming languages. *arXiv preprint arXiv:1901.05138*, 2019.
[18] Phong Le and Willem Zuidema. The inside-outside recursive neural network model for dependency parsing. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 729–739, 2014.
[19] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12):1095–1109, 2017.
[20] Xi Qiu and Christopher Stewart. Topic words analysis based on lda model. *arXiv preprint arXiv:1405.3726*, 2014.
[21] Hamed Jelodar, Yongli Wang, Chi Yuan, Xia Feng, Xiahui Jiang, Yanchao Li, and Liang Zhao. Latent dirichlet allocation (lda) and topic modeling: models, applications, a survey. *Multimedia Tools and Applications*, pages 1–43, 2018.
[22] Jonathon Shlens. Notes on kullback-leibler divergence and likelihood. *arXiv preprint arXiv:1404.2000*, 2014.
[23] Intellij-community: Foldingdescriptor. https://github.com/JetBrains/intellij-community/blob/master/platform/core-api/src/com/intellij/lang/folding/FoldingDescriptor.java.
[24] Simple classification example with missing feature handling and parameter tuning. https://github.com/catboost/tutorials/blob/dd481065adfcae83de36e05bca79ab9af1ca0a72/classification/classification_with_parameter_tuning_tutorial.ipynb.
[25] Alexander Terenin, Daniel Simpson, and David Draper. Asynchronous gibbs sampling. *arXiv preprint arXiv:1509.08999*, 2015.