1 Methodology of the KenLM N-gram Model

The methodology for training our n-gram model can be explained in three stages. Our code can be found in the publicly available GitHub repository [1].

1. Generating paths from the parsed PD files saved in our dataset
2. Preparing our training dataset and test dataset
3. Training and evaluating the performances of the n-gram models

We explain the stages in the following sections.

1.1 Generating paths from the parsed PD files saved in our dataset

We utilized the parsed contents of the PD files from our dataset [2] to generate paths from the PD source code. We extracted a list of connections between the PD file objects using the `Contents` table from our dataset for each parsed content. From the list of connections, we created a list of nodes for each PD file where a node could be a source or a destination of a connection. In this context, if there is a connection or edge from `object_0` to `object_1`, then `object_0` represents the `source` of the connection, while `object_1` is the `destination`. For our purpose, we call the `destination` node a `child` of the `source` node.

For all nodes, source or destination, we save the path originating from that node by traversing the children nodes until we find a leaf node or a node already traversed by another sibling node previously. This process ensures that we have comprehensive information about all the possible subpaths of the PD file.

1.2 Preparing our training dataset and test dataset

After employing the methodology outlined in 1.1 to generate paths from the parsed source code, 178,007,628 subpaths were generated from the parsed PD files. Initially, we partitioned the PD projects into an 80-20 split for training and testing purposes, utilizing the `train_test_split` [3] function from the `scikit-learn` [4] library in Python.

We gathered the hashes corresponding to the revisions of the PD file contents for both the train and test projects. Following this, we refined the test hashes by eliminating any matches found in the training set to assess our models' performances on paths generated from previously unseen parsed contents. Employing the paths associated with each hash value, we generated distinct training and test datasets. This process guarantees that our training and test data originate from distinct projects, minimizing overlaps between them.

1.3 Training and evaluating the performances of the n-gram models

We created seven n-gram models with orders 2, 4, 6, 8, 9, 10, 13, and 15, employing modified Kneser-Ney smoothing, and subsequently trained them on our complete

training dataset. To achieve this, we utilized the KenLM Language Model Toolkit developed by Kenneth Heafield [5, 6].

To assess the performance of our models, we randomly shuffled the test dataset using the `shuf` command in Linux and then extracted 5,000 path data points from the shuffled test dataset. Each path in our test data consisted of a sequence of tokens separated by a space character. For each path of length `n`, we regarded the first `n-1` tokens as the context for the models and the last token as the actual token.

For each test data path, we traversed through all the words in our vocabulary, comprising the unique tokens of our model. We utilized the `score` function of KenLM, as provided by the authors, to identify the highest score of a given input path. Here, the first `n-1` tokens of the input path were the context of our test data path, and the last token was selected from the vocabulary. The selected word from the vocabulary, which received the highest score when appended as a suffix to the test data path context, was designated as the predicted next token.

We evaluated the accuracy, precision, recall, and F1 score for each model by comparing the actual next token with the predicted next token. Additionally, we opted for `weighted` and `np.nan` as the values for the `average` and `zero_division` parameters, respectively, within the precision, recall, and f1 score calculation functions from the scikit-learn library. The selection of `np.nan` handles instances of zero division during the computation of these metrics by disregarding them within the averaging process. Similarly, the choice of `weighted` for the average parameter ensures that metrics are computed for each label and subsequently averaged, with the average weighted by support [7]. The performance results of these n-gram models are depicted in Figure 1.

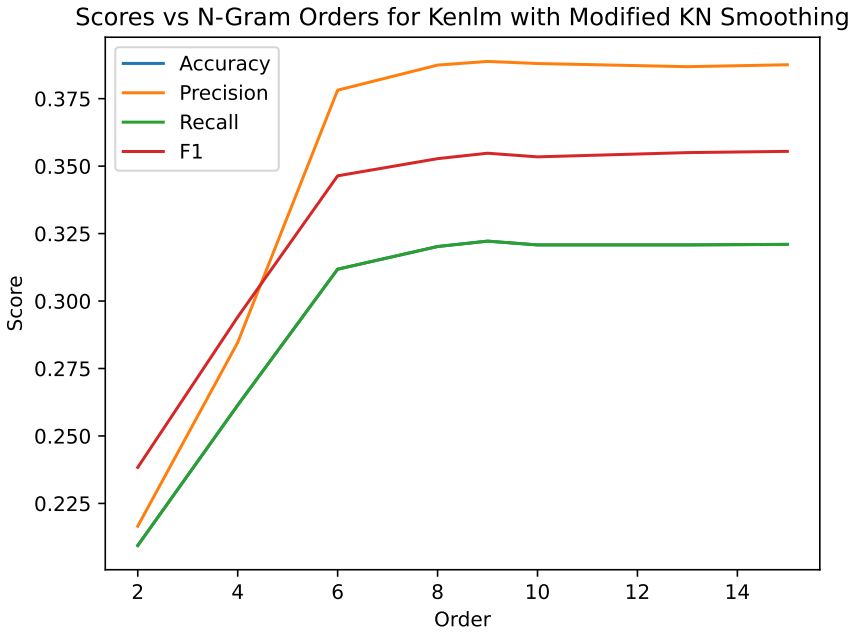

Figure 1: Performance comparison between different n-gram models

As depicted in Figure 1, the 9-gram model exhibits the highest accuracy, precision, and recall scores, attaining 32.22%, 38.87%, and 32.22%, respectively. On the other

hand, the 15-gram model achieves the highest f1-score at 35.54%. Interestingly, identical accuracy and recall values are evident across all models. Furthermore, it is worth noting that there is minimal difference in the model scores beyond the 9-gram model.

2 Limitations

The following factor might question the validity of our methodology.

2.1 Saving subpaths instead of full paths

In order to minimize computational overhead during path traversal from a node, we stopped the computation for a node when we reached a leaf node or a node traversed by a sibling node. In our case, sibling nodes are the ones with the same parent. For example, if there are two paths originating from node `a` like `a` → `b` → `c` and `a` → `d` → `e`, then `b` and `d` are called sibling nodes of each other. As a result, data is stored in terms of subpaths rather than full paths. Nevertheless, by traversing each node individually, we ensure that even if the complete path is not traversed, we retain information about all subpaths originating from the nodes within the PD files.

References

[1] Anisha Islam. Code for the n-gram. `https://github.com/anishaislam8/Kenlm-for-next-token-prediction`. Accessed: 2024-02-19.

[2] Anisha Islam. Opening the Valve on Pure-Data Dataset. `https://archive.org/details/Opening_the_Valve_on_Pure_Data`, 2023. Accessed: 2024-02-19.

[3] scikit-learn developers (BSD License). sklearn.model_selection.train_test_split. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`. Accessed: 2024-02-19.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Kenneth Heafield. kenlm. `https://github.com/kpu/kenlm`. Accessed: 2024-03-25.

[6] Kenneth Heafield. KenLM Language Model Toolkit. `https://kheafield.com/code/kenlm/`. Accessed: 2024-03-25.

[7] scikit-learn developers (BSD License). sklearn.metrics.precision_recall_fscore_support. `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html`. Accessed: 2024-02-19.