In AP4, we evaluated three different approaches to develop a predictive model for assessing the readability of Python code files. We initially used a majority class classifier as a baseline model, with five evenly distributed labels in the training set. This classifier's performance was only slightly better than random guessing. We then focused on constructing a traditional logistic regression classifier and fine-tuning a pre-trained BERT neural network.

We designed our task to center around parsing and understanding code files. We have had the opportunity to work on open-source projects, and we realized how easily maintainable a project is largely depends on how readable it is to humans, not how well it performs its tasks. We felt it would be worthwhile to develop a classifier that can determine the readability of a codefile, and it can be integrated as part of the CI/CD pipeline, and it can advise people how to make their codes more maintainable.

We drew our samples randomly from github, and given the resource constraints we were not able to go through and filter out all the low-quality data, such as configurations files and auto-generated files, and we did the best to assign them a reasonable labels under our guidelines. To cater towards these outliers, we had to create specific labels, namely the 0 category where there is no comments or explanatory strings included in the code files. We were also ambitious and tried to divide the data into fine-grained categories, so we ended up with 5 labels. Even though the data are evenly distributed among classes, we only have about 60 datapoints per label during training, and given the heterogeneity among code files, and the difficulty to make a model distinguish and focus on natural languages in code files, we realized this might have made the training more difficult and led to bad performance. If we had less classes, we would have clearer boundaries and probably better performance.

For the logistic regression model, we began with the provided Bag of Words (BoW) featurization function implementation. With the default BoW settings and a minimum feature count of 2, the model achieved high training accuracies, nearly 100%, but its performance on the development and test sets was more moderate, ranging from 50% to 60%, with the 95% confidence interval being 0.422 - 0.618. Given the almost 3000 features generated by the BoW approach, it is not surprising that this simple linear classifier overfitted the training data. Examining the top 10 feature weights of the five binary classifiers created using the One-vs-Rest (OvR) method revealed some trends. The classifier for the least readable label (0) weighted generic words like "mit", "license", and "init". Though surprising, this aligns with our intention to disregard license information as it does not contribute to understanding the code file's content. For more readable categories (3 and 4), the classifier placed more weight on natural words not part of the programming language, such as "specified", "given", and "returns", indicating a focus on code comments.

We then modified the featurization function by adding three features that we used during annotation. These included the number of occurrences of ', “ and '#' as indicators

of natural language usage, the length of strings enclosed by quotation marks normalized by content length, and the length of comments following the '#' symbol, also normalized by content length. These features were added to the BoW function, as we believed they would introduce helpful bias for model performance improvement. We also increased the minimum feature count to 20, reducing the number of features to just over 200. This modification helped the model generalize and slightly improved test set performance compared to the initial implementation, with the 95% confidence interval being 0.442 - 0.638 though only 1 out of the 5 models actually utilize that added features, so we suspect most of the performance improvements come from reducing overfitting. Confusion matrices for both implementations showed that the models tend to misclassify labels 0 and 1, as well as labels 3 and 4. This was somewhat expected, as we had defined the boundaries before completing annotations, potentially leading to ill-defined and challenging-to-learn boundaries. Most of our disagreement in annotation was regarding to the distinction between label 3 and 4, so it was expected for the model to underperform for these 2 labels.

We fine-tuned a BERT model using existing models from Hugging Face, comparing its performance to the baseline majority class classifier and the logistic regression model. We selected the neulab/codebert-python model, as our dataset consists solely of Python files. We made minor modifications to the training process, incorporating fine-tuning techniques such as freezing the main model during pre-training, adding a scheduler to decrease learning rate, and appending a fully-connected layer. We achieved a maximum dev accuracy of 61% in 20 epochs. However, the BERT model's performance was worse than the logistic model, with test set accuracies of 48% and 54%, respectively. The 95% confidence interval is 0.382 - 0.578

This outcome was unexpected, as we assumed the pre-trained BERT model would outperform the simple linear model. Analyzing the confusion matrix, we found that the BERT model incorrectly classified nearly all instances of labels 0 and 2 but performed well for the other three categories. Since we can’t visualize the weights for a deep neural network, we came up with a hypothesis: there are usually much less comments than actual codes in a code file, and it is difficult for BERT to correctly tune its attention to the scarcely appearing comments in the training texts. The same file can be labeled a class 0 if there is no comments, and a 4 if there are sufficient comments, all the while keeping the actual codes the same. As our annotation guidelines focus on what is lacking in a code file, this presents a challenge for the model to learn effectively. We theorize that incorporating information about the absence of specific features could improve performance.

Another compounding factor for the model's underperformance is the insufficiency of the 300 training files to fine-tune a BERT model effectively. To fully exploit the capabilities of the transformer architecture, a much larger dataset is required.

This experience demonstrated that while large language models possess remarkable abilities, they are not universally suitable solutions for all NLP tasks.