Supervised Spaghetti

Alex Guentchev
Department of Computer Science
Colorado State University
Fort Collins, USA
a.guentchev@colostate.edu

Colin Grebenstein
Department of Computer Science
Colorado State University
Fort Collins, USA
colin.grebenstein@colostate.edu

Jessica Bahny
Department of Computer Science
Colorado State University
Fort Collins, USA
jbahny@colostate.edu

***Abstract*—Code comments are important for programmers to clarify code content and to keep track of their own work. We have developed three models to beat the baseline set by the Natural Language-Based Software Engineering (NLBSE) 2025 Tool Competition for Code Comment Classification. We created a total of three models, one for Java, Python, and Pharo, respectively. Our Java neural network model is based on a feed-forward network with ReLU and Sigmoid activation, utilizing TF-IDF vectorization for a quick representation of the data. Our Python and Pharo models utilize a pre-trained sentence transformer to get the data, before sending the input into a multi-layer perceptron. After running our three models, most of our categories passed baseline, while two categories tied the baseline, and four of the categories were unable to beat the baseline. Despite this, we have many ideas as to why we failed certain baselines. We are happy with all that we learned through this project.**

***Index Terms*—Natural language processing, code comment analysis, software documentation, multi-label classification**

I. Introduction

Code comments are imperative for programmers to keep track of their work, and to clarify to others what has already been written [25], [18]. In this paper we will be dealing with the programming languages Java, Python, and Pharo. These programming languages each have different rules and syntax for comments, and the contents of the comments will vary based on what they are trying to explain [6]. Despite these differences, it is important for programmers to understand what comments are elucidating in the code in order to be effective tools [7]. Multi-label classification of code comments will provide an easier way for programmers to easily grasp the purpose of comments, especially when they belong to multiple categories [25].

The Natural Language-Based Software Engineering (NLBSE) 2025 Tool Competition for Code Comment Classification is the inspiration for this work [16]. In this competition, we will create three models, each one specific to each of the three programming languages [16], [15]. Each language, as aforementioned, has different syntax and content for comments [6]. Each language will have different category labels [15]. Since NLBSE'25 is about multi-label classification, each comment may be classified with multiple category labels [15]. For Java, there are seven category labels: *Summary, Ownership, Expand, Usage, Pointer, Deprecation, and Rational* [15]. For Python, there are five categories: *Usage, Parameters, DevelopmentNotes, Expand, and Summary* [15]. For Pharo, there are seven categories: *Keyimplementationpoints, Example, Responsibilities, Classreferences, Intent, Keymessages, and Collaborators* [15]. The definitions of these category labels were established in [18].

The HuggingFace dataset that will be used, created for the competition [17] [13], has a total of 14,875 comment sentences [15]. Of the 14,875 total, Java has 10,555 comment sentences from 1,049 class-level comments, distributed across its 7 categories mentioned above, Python has 2,555 comment sentences from 344 class-level comments distributed across its 5 categories, and Pharo has 1,765 comment sentences from 340 class-level comments distributed across its 7 categories [15]. The competition dataset has 3 languages x (1 train + 1 test) = 6 splits [15]. Each row in the dataset represents a comment sentence, and each comment sentence (or sample) contains five columns: *class, comment_sentence, partition, combo, and labels* [15]. As defined in [15], *class* is the class name referring to the source code file where the sentence comes from; *comment_sentence* is the actual sentence string, which is part of a (multi-line) class comment; *partition* is the dataset split in training and testing, where 0 identifies training instances, and 1 identifies testing instances; *combo* is the class name appended to the sentence string, used to train the baselines; and *labels* is the ground-truth category, a binary list that a single sample belongs to. Each comment sentence belongs to one or more categories. Each comment sentence in the dataset was preprocessed based on natural language parsers and heuristics, so some sentences in the dataset are incomplete and some comment sentences may be duplicated, in instances where the same comment was found in multiple classes in a project [15].

Each category will be measured by *precision*, *recall* and *f1*, which will determine the accuracy of each model [15]. The following definitions are our interpretation of the description in [15]. The precision, recall, and f1 measurements are all probabilities ranging from 0 to 1. The closer precision is to 1, the less likely the model is going to classify a category as a false negative. For example, in a false negative scenario, a comment sentence that describes parameters is incorrectly classified and does not include the parameters category label. The closer recall is to 1, the less likely the model is going to classify a category as a false positive. In a false positive scenario, a comment sentence that doesn't mention parameters is classified with a label for parameters. The f1 score calculates the harmonic mean between the precision and recall of a category. While precision and recall are helpful for identifying where inadequacies of the model are occurring, The f1 score for each category and across all 19 total categories will be used as the main source of acknowledging success or failure of our model.

Our goal is to beat the baseline set by the competition makers shown in Table 1, as found in the original paper [3] and replicated in [2].

TABLE 1
COMPETITION BASELINE

Language	Category	Precision	Recall	F1
Java	Summary	0.8734	0.8294	0.8508
Java	Ownership	1.0	1.0	1.0
Java	Expand	0.3235	0.4444	0.3745
Java	Usage	0.911	0.8182	0.8621
Java	Pointer	0.7383	0.9402	0.8271
Java	Deprecation	0.8182	0.6	0.6923
Java	Rational	0.1622	0.2951	0.2093
Python	Usage	0.7008	0.7355	0.7177
Python	Parameters	0.7939	0.8125	0.8031
Python	DevelopmentNotes	0.2439	0.4878	0.3252
Python	Expand	0.4336	0.7656	0.5537
Python	Summary	0.6486	0.5854	0.6154
Pharo	Keyimplementationpoints	0.6364	0.6512	0.6437
Pharo	Example	0.8729	0.9035	0.8879
Pharo	Responsibilities	0.5962	0.5962	0.5962
Pharo	Classreferences	0.2	0.5	0.2857
Pharo	Intent	0.7188	0.7667	0.7419
Pharo	Keymessages	0.68	0.7907	0.7312
Pharo	Collaborators	0.2609	0.6	0.3636

II. RELATED WORK

We had a few ideas for how to best implement our models, and we were inspired from several different sources.

Nearest neighbor was one of the first model types considered because of its simplicity. The general and standard version of nearest neighbor would look at the labels for every category combined while it was running [21]. This would take into account any possible relationship between all of the categories [21]. There is also a multi-label version of the nearest neighbor [26], which uses Bayesian inference to assign selected labels. That should be noticeably more effective than the base nearest neighbor, given that it's designed for multi-label examples such as the current data [26].

Next considered was the Pytorch neural network [14]. This doesn't natively support multi-label datasets, so the data will be looked at one category at a time, and one language at a time [14]. The data will be transformed using TF-IDF Vectorization, which stands for term frequency-inverse document frequency, from the Sci-Kit Learn library [27]. TF-IDF is calculated by multiplying those two quantities. This converts the text data to numerical data to be fed into the neural network [27].

After looking through some simple and general models, we decided to look for something more specific to natural

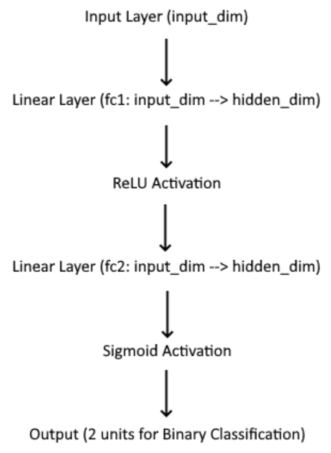
Input Layer (input_dim)
Linear Layer (fc1: input_dim --> hidden_dim)
ReLU Activation
Linear Layer (fc2: input_dim --> hidden_dim)
Sigmoid Activation
Output (2 units for Binary Classification)

Fig. 1. Java model architecture

language. Transformers are one of the better tools for recognizing natural language [22]. However, transformers are expensive to train, and take a long time to do so. They require expensive hardware and very high parameter models to accomplish even simple tasks [23]. We don't have those resources, but fortunately there are people with those resources that have open-sourced embeddings, such as [20]. By using a pre-trained embedding, we have at our disposal a rich feature set for a simple model to do classification on. The better the embedding, the simpler the model needed to perform the same task.

III. METHOD

We decided on two different sets of architecture for our models in an attempt to diversify our approach and hopefully find a good way to beat the baseline. Our Java model would use a simpler approach, utilizing TF-IDF vectorization for a quick representation of the data. Our Python and Pharo models would utilize a pre-trained sentence transformer to get the data, before sending the input into a multi-layer perceptron. We took these different approaches because since Java had more training examples associated with it, it could utilize a simpler model. On the other hand, the Python and Pharo datasets both had fewer training examples and needed a more complex model to achieve a similar result.

Expanding on that, our Java model will start with the text data being transformed into numerical data using TF-IDF Vectorization from the Sci-Kit Learn library. After processing the data, it is fed into a feed-forward neural network with two fully connected (dense) layers. It has an input layer, one hidden layer, and an output layer. The activation function ReLU (rectified linear unit) is used after the first layer, and sigmoid is applied to the final output, as shown in Figure 1.

The use of ReLU helps introduce non-linearity and can help the model learn more complex patterns [1]. ReLU is very computationally efficient, but risks the "Dying ReLU" problem where neurons get stuck outputting zero for all inputs [10].

The Sigmoid function at the output squashes values between 0 and 1, which is useful for multi-label classification where each output neuron represents the probability of belonging to a specific class. Sigmoid is excellent for binary classification and provides a smooth gradient, but this is at the risk of saturating or killing gradients for very high or low input values [11]. It is also more computationally intensive than other alternatives. Something to note about the Sigmoid function is it can output probabilities for each label independently of the other labels. An example could have one or more labels attached to it, so it allows for each label to have its own probability. Softmax would take into account all of the labels and give a probability where all output probabilities add up to 1, so Sigmoid we felt Sigmoid was more fitting for the model. Our model architecture for Java can be seen in Figure 1 and Listing 1. The architecture consists of an input layer with input_dim neurons, which is determined by the shape of X_train, a hidden layer (fc1) with 128 neurons and ReLU activation, and an output layer (fc2) with 2 neurons (for binary classification) and sigmoid activation.

Listing 1. Model Architecture - Java

```
class ClassificationModel(nn.Module):
    def __init__(self, input_dim, hidden_dim,
    output_dim):
        super(ClassificationModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return torch.sigmoid(x)
```

We settled on CrossEntropyLoss as our loss function for the Java model. We chose it because it provides stronger gradient amplification, allowing the model to learn more quickly. It is also well-suited for both binary and multi-class classification problems. Despite these benefits, CrossEntropyLoss struggles with imbalanced datasets and may lead to overfitting [8]. For the optimizer, we chose Adam (Adaptive Moment Estimation). It was chosen due to its fast relative convergence compared to other optimizers alongside its low memory requirements. However, one disadvantage to using Adam is it is prone to overfitting in smaller sample sizes, and we chose small batch sizes of 32 for our model [9]. This model was indeed over-fitting very quickly, so we settled for a lower number of epochs (five). The best learning rate we found was 0.001, and any learning rate higher than that degraded the performance very rapidly.

This approach trains a model from scratch, so we decided to test if a pre-trained model could perform better.

Using the transformer from [20], we hoped to create a model for Python and Pharo that is more proficient with natural language. For the embedding, we chose the Codet5-base, which has derivative models that do tasks like code generation and completion. We chose Codet-5 because its focus is on understanding code [20] [24]. The embedding is specialized, which fits our use case better. SetFit uses a general-purpose sentence transformer for its embedding, which may not recognize code patterns as well as it can recognize normal language [3].

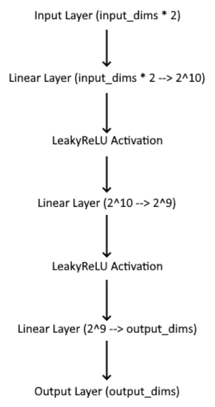
Input Layer (input_dims * 2)
Linear Layer (input_dims * 2 --> 2^10)
LeakyReLU Activation
Linear Layer (2^10 --> 2^9)
LeakyReLU Activation
Linear Layer (2^9 --> output_dims)
Output Layer (output_dims)

Fig. 2. Python/Pharo model architecture (1/2)

For the model for Python and Pharo, we froze all but the last three layers of the encoder. This was because we wanted to fine-tune the model, not completely change it. After going through the embedding, the input data is taken from sentence form, tokenized, and a feature matrix is created for the neural network to process, as shown in Figure 3. The input dimension to LinearNN is doubled (input_dims * 2) due to the concatenation of pooled embeddings and cluster features. Our initial attempts used a simple neural network, LeakyReLU, to overcome limitations from the hard 0 that ReLU returns [5]. Listing 2 contains our neural network architecture. The LinearNN consists of three linear layers with LeakyReLU activations between the first two layers as shown in Figure 2.

Listing 2. Neural Network Architecture

```
self.layers = torch.nn.Sequential(
    torch.nn.Linear(input\_dims, 2 ** 10),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(2 ** 10, 2 ** 9),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(2 ** 9, output\_dims)
    )
```

For the loss function, we chose BCEWithLogitsLoss (Binary Cross Entropy with Logits Loss) which is commonly used for binary classification problems. BCEWithLogitsLoss streamlines implementation by combining sigmoid and BCE into one function, resulting in more computational efficiency. It also handles class imbalance better than the CrossEntropyLoss function we used in the previous model. One drawback of using BCEWithLogitsLoss is that it is not suitable for multi-

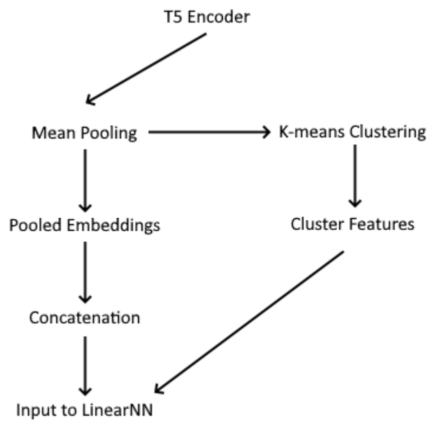

Fig. 3. Python/Pharo model architecture (2/2)

class problems, but this ended up not affecting us since we opted to run each class separately [4]. For the optimizer, we chose a spiritual successor to Adam from the previous model named AdamW (Adaptive Moment Estimation with Weight Decay). AdamW promotes faster convergence and improved generalization at the expense of potentially converging to a suboptimal solution with improper tuning [19]. Our focus was on tuning the model, so we figured that the advantages would outweigh the disadvantages.

IV. Results

The results from our three models can be seen in Table Table 2. The difference between the baseline and our models can be seen in Table Table 3.

TABLE 2
Model Results

Lang	Category	Precision	Recall	F1	Accuracy
Java	Summary	1.0	0.7241	0.84	0.7832
Java	Ownership	1.0	1.0	1.0	0.9983
Java	Expand	1.0	1.0	1.0	0.9386
Java	Usage	0.8621	1.0	0.9259	0.8748
Java	Pointer	1.0	1.0	1.0	0.9646
Java	Deprecation	1.0	1.0	1.0	0.9948
Java	Rational	1.0	1.0	1.0	0.9606
Python	Usage	0.6689	0.8347	0.7426	0.8275
Python	Parameters	0.8119	0.7422	0.7755	0.8645
Python	DevNotes	0.4	0.1951	0.2623	0.8892
Python	Expand	0.5571	0.6094	0.5821	0.8621
Python	Summary	0.686	0.7195	0.7024	0.8768
Pharo	K.I.P.	0.7105	0.6279	0.6666	0.9065
Pharo	Example	0.9145	0.8992	0.9067	0.9238
Pharo	Responsibility	0.5493	0.75	0.6341	0.8443
Pharo	Classrefs	0.6666	0.5	0.5714	0.9896
Pharo	Intent	0.9	0.9	0.9	0.9792
Pharo	Keymsgs	0.68	0.7907	0.7312	0.9134
Pharo	Collaborators	1.0	0.2	0.3333	0.9723

Two categories tied the baseline, and four of the categories were unable to beat the baseline. We would like to note, however, that the difference from the baseline in all the categories that failed is within the hundredths place, so the difference is quite small.

TABLE 3
Difference from Baseline (F1 Score)

Lang	Category	Difference
Java	Summary	-0.01
Java	Ownership	+/-0.0
Java	Expand	+0.6255
Java	Usage	+0.0638
Java	Pointer	+0.1729
Java	Deprecation	+0.3077
Java	Rational	+0.7907
Python	Usage	+0.0249
Python	Parameters	-0.0276
Python	DevelopmentNotes	-0.0629
Python	Expand	+0.0284
Python	Summary	+0.087
Pharo	Keyimplementationpoints	+0.0229
Pharo	Example	+0.0188
Pharo	Responsibilities	+0.0379
Pharo	Classreferences	+0.2857
Pharo	Intent	+0.1581
Pharo	Keymessages	+/-0.0
Pharo	Collaborators	-0.0303

V. Discussion

Despite not beating every baseline, we started this project with the intention to learn, and that is exactly what we did. We had to make a decision whether we would stop at nothing to beat all of the baselines, or learn from where we succeeded and where we failed. We chose the latter option, as sometimes in research you set out to test something and sometimes it doesn't turn out the way we thought. Failure is still progress. Here's what we learned from our results.

For the Java model, we used CrossEntropyLoss which was a pretty good match for the problem we were trying to solve, but it suffers from overfitting and it struggles with imbalanced datasets [8]. The datasets we were working with had some categories with many positive labels, and some with very few positive labels. This may have biased our model slightly towards the majority classes in the dataset. We also struggled with over-fitting in the Java model specifically, which was probably a result of picking CrossEntropyLoss. The Adam optimizer also struggles with over-fitting, which probably further exacerbated the issue alongside the low learning rate, which also contributes to over-fitting [9]. The f1 score might be too high artificially since the batch sizes were so small. Looking at more verbose output while running our code shows that for some of the more sparse categories, the test sets had multiple batches where every single label was a zero. With each consecutive epoch, the test accuracy and f1 score would dip, so we ended up choosing a lower number of epochs, but with a higher learning rate. With this method, the model would lose all accuracy very quickly. This leads us to believe that maybe there was a sweet-spot for the learning rate that would have given a solid accuracy without over-fitting the training set.

For the Python and Pharo models, we used an identical model architecture to train on both datasets. The last 3 layers of the encoder were frozen to give the model some room to fine-tune onto the specific problem of code comments. The models were quick to train. The loss curves were normal exponential decay graphs, showing the model was learning the data well. The trainings graphs indicated on the first training run that while the loss curve had largely settled to its minimum, the model still had room to improve. After doubling the epochs, the model stagnated and model performance decreased as the epochs went up. After trying this, we trained a kmeans classifier on the data to provide the model with some more context. However, the kmeans layer did not give any significant improvements to the f1 score. Validation f1 scores were nearing one, but never reached one.

VI. Conclusion

After the first round of training, we had beaten the baseline in several categories. We were shocked by how well the nearest neighbor into the multi-layer perceptron did despite its simplicity. We did not anticipate how hard it would be to fine tune the model to increase performance. The data is hard to work with, and was an interesting challenge. Overall, we are happy with the results we got so far, and definitely learned a lot in the process.

VII. Future Work

This project left us wondering in what other languages it would be beneficial to have multi-labeled classification of comments. C# and C++ are two of the most common high-level programming languages, and may make for good additions for future attempts at code classification due to their widespread use [12].

It would also be interesting to embrace more crude natural language instead of the elegant syntax from properly-formatted class comments already designed to make reading comments easier. It would be fascinating to create a dataset of student code comments that would inform the user of the contents of a block of code, but in a much less professional and consistent fashion than the dataset used for the NLBSE'25 competition.

Additionally, if we were to continue working on this, we would be interested in utilizing the decoder part of the Codet-5 model to augment the data. We could utilize the decoder to make artificial training samples, and see if that improves the performance of the model. The first neural network that we used for Java also performed extremely well, so we would be curious to see if the scores remained high after removing the issue of over-fitting. This might be achieved by adjusting the batch sizes or by employing an alternative optimizer or loss function.

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[2] Ali Al-Kaswan and Maliheh Izadi. The (ab)use of open source code to train large language models, 2023.

[3] Ali Al-Kaswan, Maliheh Izadi, and Arie Van Deursen. STACC: Code comment classification using SentenceTransformers. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 28–31, 2023.

[4] PyTorch Contributors. Bcewithlogitsloss.

[5] Pytorch contributors. LeakyReLU, 2023. Available at: https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html.

[6] MS Farooq, SA Khan, K Abid, F Ahmad, MA Naeem, M Shafiq, and A Abid. Taxonomy and design considerations for comments in programming languages: a quality perspective. *Journal of Quality and Technology Management*, 10(2):167–182, 2015.

[7] Richard K Fjeldstad. Application program maintenance study. *Report to Our Respondents, Proceedings GUIDE*, 48, 1983.

[8] Elliott Gordon-Rodriguez, Gabriel Loaiza-Ganem, Geoff Pleiss, and John P. Cunningham. Uses and abuses of the cross-entropy loss: Case studies in modern deep learning, 2020.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. Available at: https://arxiv.org/abs/1412.6980.

[10] Lu Lu Lu Lu, Yeonjong Shin Yeonjong Shin, Yanhui Su Yanhui Su, and George Em Karniadakis George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *Communications in Computational Physics*, 28(5):1671–1706, January 2020. Available at: http://dx.doi.org/10.4208/cicp.OA-2020-0165.

[11] Lars Nieradzik, Gerik Scheuermann, Dorothee Saur, and Christina Gillmann. Effect of the output activation function on the probabilities and errors in medical image segmentation, 2021. Available at: https://arxiv.org/abs/2109.00903.

[12] Zahida Parveen and Nazish Fatima. Performance comparison of most common high level programming languages. *International Journal of Computing Sciences Research*, 5:246–258, 10 2016.

[13] Luca Pascarella and Alberto Bacchelli. Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017.

[14] Pytorch. Neural Networks, 2024. Available at: https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html.

[15] Pooja Rani, Ali Al-Kaswan, Giuseppe Colavito, and Nataliia Stulova. NLBSE'25 tool competition: Code comment classification. Jupyter notebook, 2024. Available at: https://colab.research.google.com/drive/1GhpyzTYcRs8SGzOMH3Xb6rLfdFVUBN0Pusp=sharing#scrollTo=08GDMStU8PmT.

[16] Pooja Rani, Ali Al-Kaswan, Nataliia Stulova, and Giuseppe Colavito. NLBSE 2025: The 4th intl. workshop on nl-based software engineering, 2024. Available at: https://nlbse2025.github.io/tools/.

[17] Pooja Rani, Rafael Kallis, Maliheh Izadi, Giuseppe Colavito, and Ali Al-Kaswan. NLBSE'25 code comment classification dataset, 2024. Available at: https://huggingface.co/datasets/NLBSE/nlbse25-code-comment-classification.

[18] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. How to identify class comment types? a multi-language approach for class comment classification. *Journal of systems and software*, 181:111047, 2021.

[19] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond, 2019.

[20] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, 2019.

[21] SciKitLearn. 1.6. Nearest Neighbors, 2024. Available at: https://scikit-learn.org/1.5/modules/neighbors.html.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[23] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. HAT: Hardware-Aware Transformers for Efficient Natural Language Processing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[25] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 215–223. IEEE Press, 1981.

[26] Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

[27] Ye Zhang, Qian Leng, Mengran Zhu, Rui Ding, Yue Wu, Jintong Song, and Yulu Gong. Enhancing text authenticity: A novel hybrid approach for ai-generated text detection, 2024. Available at: https://arxiv.org/abs/2406.06558.