POLITECNICO DI TORINO

Department of Electronics and Telecommunications

MASTER's Degree Thesis in
ICT FOR SMART SOCIETIES

Enhancing Arduino AI Assistant: Semi-supervised User Intent Classification for RAG Optimization

Supervisors

Prof. Luca VASSIO

Claudio SCAFESI

Giulia BIAMINO

Candidate

Shurui CHEN

Academic Year 2024/25

Abstract

Recent advances in generative AI have enabled the integration of large language models (LLMs) into development environments to assist users in programming, interpreting, and debugging. This paper presents a complete data processing and classification pipeline for a GenAI Chat Assistant embedded in the Arduino Cloud Editor. We propose classifying user queries into distinct intent categories, specifically create code, explain, suggest, and fix errors, to optimize Retrieval-Augmented Generation (RAG) responses. The goal is to improve the relevance and efficiency of RAG by tailoring document retrieval and response generation strategies to the specific user intent, thereby reducing irrelevant content and optimizing token usage in LLM responses.

To support this classification, we first construct a text preprocessing framework that filters out noises, prompts, code-only contents, and non-English inputs, retaining only valid user queries for analysis. We benchmark multiple sentence embedding models (e.g., MiniLM, mpnet, E5, BGE, Nomic AI) and ultimately select the intfloat/e5-base-v2 for its balance between performance and efficiency.

Given that the raw dataset is unlabeled, we need to set corresponding principles for manual labeling. The number of labeled categories need to be balanced so that self-training can fully learn the characteristics of each category. By comparing the performance of each classification model, we built a semi-supervised classification framework based on calibrated ensemble models (LightGBM, CatBoost, logistic regression).

During the self-training process, we designed a method to dynamically adjust the category threshold to address situations where labels are unbalanced and some category boundaries are blurred. We first set a high-confidence initial threshold to select more accurate pseudo-labels. Then, in each new round of training sets, high-confidence pseudo-labels are added to ensure that the model can continue to learn. The new threshold is dynamically adjusted based on the number of pseudo-labels added to the training dataset in each round. This method gradually expands the labeled dataset while maintaining a balanced category distribution. In addition, in order to make full use of a large amount of data for model learning and prevent over-learning of erroneous pseudo-label data, we used an early stopping strategy during the self-training process to obtain the best semi-supervised learning model.

Our method relatively improves the quality of pseudo-labels, achieves stable learning in underrepresented categories, and achieves good macro F1-scores on the retained validation set. When applied to a test dataset extracted from the production database, the model predicts intent labels for new user queries and provides an estimated class distribution. The resulting labeled data supports intent-aware retrieval and more precise response generation within the GenAI system, enhancing user interaction, improving token efficiency, and enabling further optimization of assistant performance.

Keywords: Retrieval-Augmented Generation (RAG),Semi-Supervised Learning, large language models (LLMs), Sentence Embedding, Soft Voting Classifiers, Extract Transform Load(ETL), Generative AI

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Luca Vassio, for his continuous support, invaluable guidance, and insightful feedback throughout the course of this research. It has been an honor to complete this thesis under his mentorship.

Special thanks go to my company tutors and colleagues in the Arduino Data team, including Claudio Scafesi and Giulia Biamino, for the fruitful discussions, encouragement, and for fostering a collaborative and inspiring working environment. Their expertise and assistance were essential to the completion of this research. I also acknowledge the support from Arduino, which provided the necessary resources to carry out this research.

Moreover, I am deeply grateful to Politecnico di Torino for providing a rich academic platform that allowed me to meet outstanding individuals, improve my skills, and build confidence to face future challenges. I especially thank the friends and classmates for their support and encouragement, which have inspired me to continuously strive for improvement in both my academic and personal life.

I would also like to extend my heartfelt appreciation to my family for their unwavering love, patience, and support throughout my academic journey.

My last tribute is to those who know I am not perfect but still love me. May your life set sail, heading towards great ideals.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In recent years, with the rapid development of deep learning, artificial intelligence (AI) has gradually become a dominant assistant technique in various domains, especially in education and engineering domain. One important development is the emergence of Generative Artificial Intelligence (GenAI) chat assistants, which leverage large-scale language models (LLMs) for natural language interactions, assist users in processing complex information, and facilitate further model learning and analysis [1]. GenAI chatbots increasingly dominate the AI landscape and have played a significant role in various industries and functional areas [2], including creative and knowledge industries.

Generative AI refers to a class of machine learning (ML) models and techniques that can generate new synthetic data (e.g., text, images, audio, and code). In the field of software development, GenAI has the potential to revolutionize all aspects of the development process. Developers can automate tasks such as code generation (generating code function snippets and templates), documentation, or code review and analysis based on natural language prompts on demand. This can accelerate developers' workflows, enhance coding capabilities, and explore new possibilities in the software development lifecycle. AI-driven tools have proven particularly useful in programming applications, enabling intelligent code generation, debugging support, and context-aware explanations to boost developer productivity[3].

Our project is aim at improving performance of the GenAI assistant in Ardunio IoT Cloud. Ardunio IoT Cloud service is a cloud-based platform that allows users to easily connect, manage, and control their Internet of Things (IoT) devices and applications. As a key component of this IoT Cloud ecosystem, the Arduino Cloud Editor provides a web-based Integrated Development Environment (IDE) for editing, compiling, and managing sketches. With the increasing integration of AI-powered tools, it's more and more essential to incorporating a GenAI chat assistant into the Cloud Editor. This AI tool significantly enhances user experience by providing intelligent coding assistance, improving development efficiency, and facilitating interactive learning, as demonstrated in the workflow shown in Figure 1.1. The workflow illustrates how

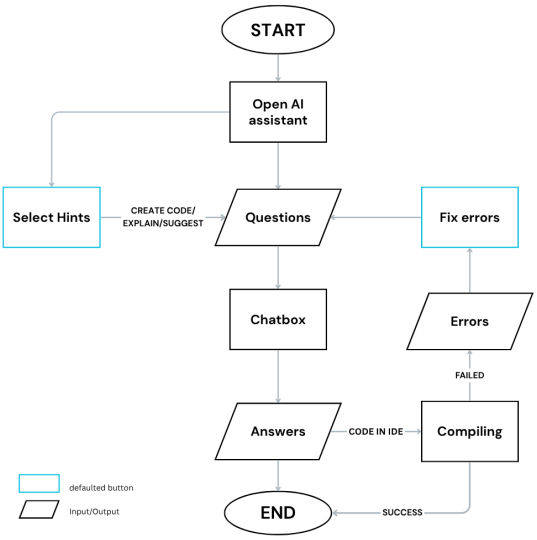


Figure 1.1: GenAI Assistant Workflow

users interact with the GenAI Assistant within the Arduino Cloud Editor IDE. The design of interactive buttons (3 type of 'Hint' and 'Fix Errors') directly motivates the use of four classification categories for user queries, ensuring alignment between user intent and system response. The 3 type of'Hint' button provides clear examples of typical user intents, such as:

- Create code to blink an LED without using delays
- Explain this function: map(input, 0, 255, 0, 10)
- Suggest a project idea using a strip LED

This classification framework is designed by Arduino and helps the GenAI Assistant deliver more targeted and meaningful support, making it an essential component for optimizing the user experience and advancing the project's objectives.

As user queries often require domain-specific and contextually accurate answers, Retrieval-Augmented Generation (RAG) is used to enhance the accuracy and relevance of responses from large language models (LLMs) by incorporating external data sources. RAG systems retrieve relevant documents (e.g., Arduino forums, documentation, and templates) and feed them to a generative language model to produce informed, high-quality responses.

Critically, GenAI chatbots vary greatly in terms of effectiveness while generate answer with RAG. While the assistant can generate context-aware code suggestions and insightful explanations, its responses quality are inherently constrained by the limitations of generative AI [4]. Even the best powerful GenAI chatbots can misunderstand user queries, miscommunication, or produce low-quality or inaccurate responses, which may lead to service failure or unsatisfied results [5]. Failure to recover from a service breakdown through chatbot intervention can magnify its adverse consequences, severely affecting a brand's reputation and financial outcomes

[6]. Paradoxically, while GenAI chatbots present promising new opportunities for user service, they also present significant challenges and threats.

Despite these challenges, GenAI Assistants remain a valuable tool for assisting users with great promise and commercial value. Ongoing improvements aim to increase the accuracy and reliability of its responses in future iterations[7]. Therefore, predicting the intent of user questions and improving the accuracy of GenAI answer generation will be important keys to progress in this area.

In our current configuration of GenAI assistant, each user query to the AI assistant in the Cloud Editor initiates a series of steps to fetch the complete open sketch, which is then consistently included in the system prompt to provide context before generating a response. While this guarantees that the assistant always has the full context, in practice, when the query is not related to the current sketch, we are sending information that's not needed. When the assistant receives unnecessary details, it's more likely to become distracted, produce less accurate answers, or even hallucinate. In this regard, I was able to consistently replicate a case of hallucination due to too much information we pass in the context. This situation also leads to slower responses, as well as extra token usage that adds up over time. Therefore, we propose a quick preliminary check with a quick and low-cost LLM as an immediate, low-effort improvement.

This additional layer will classify user queries to determine whether the open sketch context is actually necessary. Integrating this step requires only minor adjustments, as our backend service already supports multiple LLMs. Although it adds a negligible amount of time and a small recurring cost, it streamlines the flow. If the sketch isn't needed for a particular question, we don't include it in the process. This reduces the cognitive load on the main LLM, prevents unnecessary confusion, and ensures more reliable answers. In addition, this preliminary step may result in some savings due to not sending unnecessary input tokens: potentially around $50 per month. Anyway, the focus is on ensuring a leaner, more efficient prompt that keeps the assistant's attention squarely on what matters, helping it respond more accurately and consistently.

The graph in figure 1.2 illustrates the overall architecture of the GenAI Assistant, which receives user queries, generates responses via the Arduino GenAI API and stores in database. According to Arduino developing document, a rough estimate costs for this phase, in excess, for the cost on Bedrock for this additional preliminary step (with monthly volumes at around 10,000 messages and an input size of $\tilde{7}50$ tokens) is as follows:

- Using the least expensive, less reliable model: $1.88/month
- Using an intermediate, slightly more reliable model: $6.02/month
- Using the most reliable model: $7.53/month.

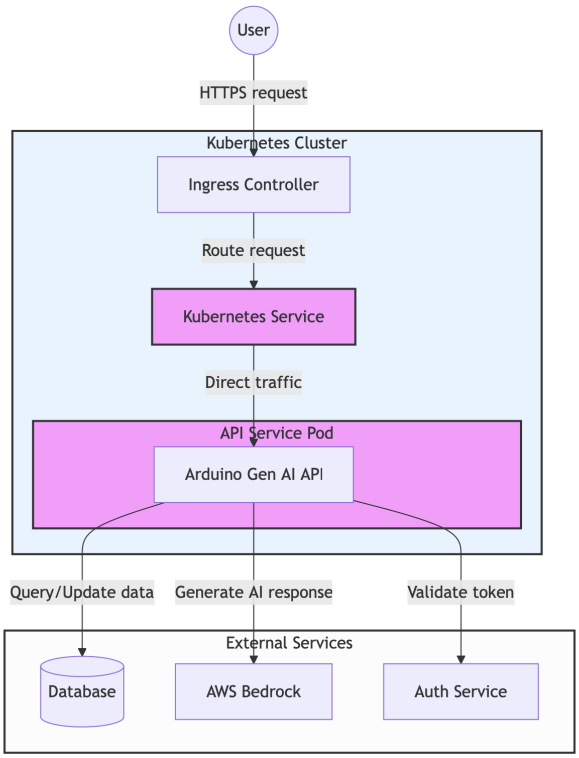


Figure 1.2: GenAI services architecture

In the following sections, we propose a semi-supervised user intent classification model based on embedding representations to identify four main intent categories: code generation, explanation, suggestions, and error fixing. This classification allows the RAG pipeline to route queries appropriately, select relevant document sources, and reduce unnecessary information noise to enhance response quality and speed. These enhancements not only refine AI responses but also drive business value by increasing user retention and reducing extra token usage, contributing to the product's long-term growth. As such, user intent prediction serves as a critical component for optimizing retrieval and generation stages in the Arduino GenAI assistant.

1.2 Objective

The objective of this study is to enhance the Arduino GenAI Assistant's ability to understand and respond to user queries by introducing a semi-supervised intent classification framework.

Once we gather enough data on which queries require the sketch and which do not, we can move beyond the initial classification step and develop our own ML model. By training a custom binary classifier that runs directly on our backend, we eliminate the need for external LLM calls. This model will instantly determine whether to include the sketch, avoiding delays and extra costs associated with another call to Bedrock and still saving money not sending the unnecessary tokens.

In addition, pursuing this strategy does not interfere with our priorities, such as integrating Retrieval Augmented Generation (RAG) and improving the assistant

response quality. The incremental approach—starting with a simple LLM-based filter, then graduating to an in-house model—allows us to develop these capabilities in parallel. Our existing infrastructure and tools (such analytics for data collection and EKS for deployment) are suitable for assisting the work, ensuring a smooth integration process without derailing ongoing projects.

To achieve this, we mainly focus on four key objectives:

1. **Develop a robust text preprocessing pipeline** – to clean and normalize user-submitted queries from the Arduino Cloud Editor.
2. **Design an embedding-based semi-supervised classification model** – to categorize queries into four key intent classes: code creation, explanation, suggestion, and error fixing.
3. **Leverage user behavior analytics from Lookerstudio** – to validate intent taxonomy design, uncover dominant query patterns, and evaluate improvements in user engagement and assistant performance.
4. **Evaluate the classification model and overall assistant performance** – measuring improvements in classification accuracy, retrieval relevance, and user guidance.

By addressing these goals, this study aims to enhance user experience, optimize AI-driven interactions, and improve the efficiency of GenAI Assistant in practical applications.

1.3 Literature Review

Intent detection and classification—also known as semantic utterance classification—is a fundamental component of natural language understanding (NLU) in task-oriented conversational agents. This process involves identifying the underlying intent of a user utterance from a predefined set of intent categories, enabling the system to guide the dialogue flow according to the recognized intent[8]. The current study found that structured categorization can improve the adaptability of chatbots[9], thereby providing more personalized and contextually relevant responses.

Prior researches have explored various methods for intent classification and dialog act recognition across multiple domains. In the context of software engineering, recent studies have examined how conversational AI can support developer workflows, including automating low-level workflows[10], questions responding based on code repositories[11], expert recommendation[12] and conflict resolution[13]. These studies provide a foundational basis for designing the Arduino GenAI Assistant and guiding the intent classification process, which includes four primary categories: code creation, explanation, suggestion, and error fixing.

Sequence classification is a central technique in intent detection which assigns labels to complete textual inputs such as user queries. Nowadays Recurrent Neural

Networks (RNNs) and Transformers are widely used to analyze input sequences and predict user intent, thus enabling conversational agents to provide targeted and effective responses. However, many datasets have inherent limitations and challenges, including limited labeled samples, class imbalance, rare technical terms, and limited variability in query wording. These issues may lead to unstable and poor model performance, thereby reducing its versatility in practical applications[14]. Addressing these challenges is particularly critical for domain-specific systems like the Arduino GenAI Assistant, where diverse user intents must be understood accurately despite limited supervised data.

To address the aforementioned challenges, prior studies have explored a variety of approaches for intent classification, ranging from traditional machine learning to recent advances in transformer-based models. Al-Tuama et al. evaluated classical classifiers—including Support Vector Machines (SVM), Random Forests, Logistic Regression, and Multinomial Naive Bayes—for virtual assistant intent detection, and investigated data augmentation techniques to boost performance in low-resource settings [15]. In parallel, hybrid and transformer-based architectures have emerged as powerful tools for intent classification. For example, BERT-BACBC and Co-Interactive Transformer architectures combine transformer encoders with recurrent layers to leverage both contextual and sequential information [16]. Shen et al. proposed integrating BiLSTM with RoBERTa and applying soft voting, achieving notable improvements in classification accuracy [17]. These research findings inform our methodology, motivating the use of multiple machine learning models in an ensemble framework. Specifically, we adopt a soft voting strategy through a Voting Classifier to leverage the complementary strengths of individual classifiers.

Despite advancements, existing methods often struggle with unseen intents, low-resource languages, and domain-specific applications, such as code-related question classification in the Arduino ecosystem. Few-shot and zero-shot learning approaches, such as adaptive clustering for unknown intents[18] show promise in addressing these issues. In this context, our study explores a lightweight, embedding-based semi-supervised classification pipeline that builds on these prior insights. By combining transformer embeddings with ensemble classifiers and dynamic pseudo-labeling, we aim to achieve robust intent categorization under limited supervision, while enabling downstream integration with RAG-based response generation.

1.4 Overall Structure

- chapter 2 introduce the overall framework and Methodology for this study.
- chapter 3 collect users data, analysis data and process data.
- chapter 4 simulate different cases with different parameters settings.
- chapter 5 discuss the result and present the future works.

Chapter 2

Methodology

2.1 Research Framework

This study proposes a semi-supervised learning pipeline for classifying user queries in the Arduino Cloud GenAI Assistant. The objective is to enhance the system's ability to understand user intent and selectively decide whether to trigger the resource-intensive Retrieval-Augmented Generation (RAG) process. The framework is composed of four primary components: query preprocessing, embedding-based intent classification via a semi-supervised model, intent-aware retrieval, and final response generation using an LLM-based assistant.

The proposed research framework is designed to pre-predict user queries prior to invoking the GenAI API, enabling early prediction and classification of user intent. Figure 2.1 shows whether system needs to refer to the sketch context to generate an response when querying, which enables the optimization of the reply generation of GenAI assistant by reducing unnecessary API calls during the process. This allows the system to identify relevant knowledge sources in advance, optimize response generation, and reduce unnecessary API calls, thereby improving both efficiency and resource utilization. It also shows how user query data is stored and used as a data source for our project.

The framework begins by collecting queries submitted by users within the Arduino Cloud Editor. After preprocessing, the queries are encoded using sentence embeddings and passed through a semi-supervised classifier trained to categorize them into four

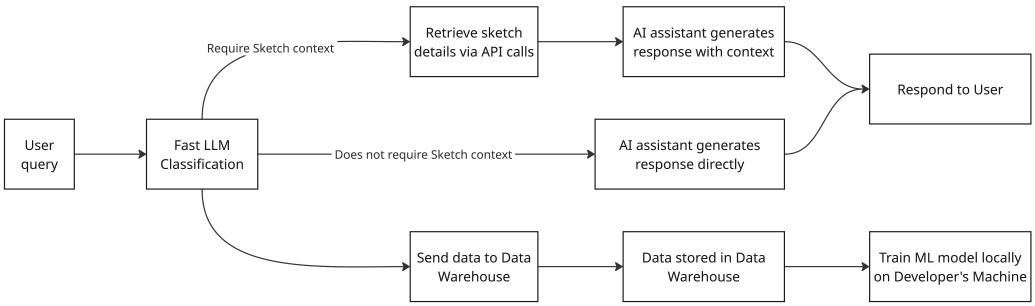


Figure 2.1: Sketch context requirement workflow

intent types: code creation, explanation, suggestion, and error fixing. Based on the predicted intent, the system dynamically determines whether additional retrieval from the user's current sketch is necessary. This selective approach improves responsiveness and reduces the computational cost of running full retrieval-based inference.

2.2 Algorithms

2.2.1 Self-Training Framework

We adopt a self-training semi-supervised learning strategy to combine a small labeled dataset $\mathcal{D}_L$ with a large unlabeled user queries $\mathcal{D}_U$. Initially, a supervised classifier is trained on the labeled data for the first round and get a teacher model. In each self-training iteration, the classifier predicts the class probabilities of the unlabeled samples. The predictions with high confidence are selected as pseudo labels and added to the labeled dataset for the next round of training.

The overall self-training procedure is illustrated in Algorithm 1. At each iteration, the model is trained on a growing labeled set, comprising the original labeled data and confident pseudo-labeled samples selected based on dynamic, class-wise thresholds (as shown in Eq. 2.1). This threshold is adaptively adjusted according to the pseudo-label distribution of each class to promote class balance during the iterative training process.

To prevent overfitting or performance degradation, we apply an **early stopping mechanism** during the self-training process. Specifically, after each iteration, we evaluate the model on a reserved validation set. By comparing the evaluation results of each round, the model checkpoint with the highest validation performance is saved. If the validation performance does not improve in consecutive P rounds, the self-training process is terminated. This ensures that the model can only benefit from pseudo-labeled samples if they can improve generalization performance.

2.2.2 Pseudo-Label Selection with Dynamic Thresholding

To ensure both prediction reliability and balanced class representation during self-training, we adopt a **dynamic, class-wise thresholding strategy**. For each intent class, a separate confidence threshold $\theta_k^{(t)}$ is computed at each iteration t, based on both the iteration number and the pseudo-label distribution observed in previous rounds.

We start with a relatively high base threshold (e.g., $\theta_{\text{init}} = 0.84$) to ensure early pseudo-labels are highly confident. This base threshold decays over time to allow more diverse samples as the model improves. Specifically, at iteration t, the decayed base threshold is computed as:

$$\theta_0^{(t)} = \theta_{\text{init}} - \Delta \cdot t$$

To address class imbalance, we dynamically adjust the threshold for each class k based on its proportion in the pseudo-labeled data. Let $c_k^{(t)}$ denote the cumulative

Algorithm 1 Self-Training with Early Stopping and Dynamic Threshold

Require: Labeled dataset $\mathcal{D}_L$, Unlabeled dataset $\mathcal{D}_U$, max rounds T, patience P, max pseudo-labels per class M

Ensure: Trained classifier f

Initialize classifier f on $\mathcal{D}_L$
Initialize best validation score $s_{\text{best}} \leftarrow -\infty$, patience counter $p \leftarrow 0$
Initialize pseudo-label count tracker $\{c_k \leftarrow 0\}$ for all classes k
for round $t = 1$ to T **do**
    Train classifier f on current $\mathcal{D}_L$
    Predict class probabilities $P(y = k \mid x_i)$ for $x_i \in \mathcal{D}_U$
    Assign pseudo-labels $\hat{y}_i = \arg\max_k P(y = k \mid x_i)$
    Initialize pseudo-labeled set $\mathcal{D}_P^{(t)} \leftarrow \emptyset$
    for each class k **do**
        Compute threshold $\theta_k^{(t)} = \textsc{AdjustThreshold}(k, t, \{c_k\})$
        Select top-M samples from $\mathcal{D}_U$ where $\hat{y}_i = k$ and $P(y = k \mid x_i) \geq \theta_k^{(t)}$
        Add selected $(x_i, \hat{y}_i)$ to $\mathcal{D}_P^{(t)}$, update c_k
    end for
    if $\mathcal{D}_P^{(t)} = \emptyset$ **then**
        Select top-N most confident predictions as fallback
        Add them to $\mathcal{D}_P^{(t)}$
    end if
    Update $\mathcal{D}_L \leftarrow \mathcal{D}_L \cup \mathcal{D}_P^{(t)}$
    Remove used samples from $\mathcal{D}_U$
    Evaluate on validation set to obtain score $s^{(t)}$
    if $s^{(t)} > s_{\text{best}}$ **then**
        Save model $f_{\text{best}} \leftarrow f$, update $s_{\text{best}} \leftarrow s^{(t)}$, reset $p \leftarrow 0$
    else
        $p \leftarrow p + 1$
    end if
    if $p \geq P$ **or** $\mathcal{D}_U = \emptyset$ **then**
        Break ▷ Early stopping or no data left
    end if
end for
return f_{best}

number of pseudo-labels assigned to class k up to round t, and K the number of total classes. Define:

$$r_k^{(t)} = \frac{c_k^{(t)}}{\sum_{j=1}^{K} c_j^{(t)} + \varepsilon}, \quad \text{and} \quad r^* = \frac{1}{K}$$

We then compute an imbalance factor using a square root scaling:

$$\delta_k^{(t)} = \sqrt{\frac{r_k^{(t)}}{r^*}}$$

The threshold for each class is then adjusted non-linearly based on $\delta_k^{(t)}$:

- If $\delta_k^{(t)} > 1.2$, class k is over-represented; the threshold is increased to suppress further pseudo-labeling.
- If $\delta_k^{(t)} < 0.8$, class k is under-represented; the threshold is decreased to allow more pseudo-labels.
- Otherwise, no adjustment is made.

The adjusted threshold is clipped within a valid range (e.g., $[0.80, 0.98]$) to prevent extreme values and keep high quality examples. A pseudo-labeled instance $x_i \in \mathcal{D}_U$ is selected into the training set only if:

$$\max_k P(y = k \mid x_i) \geq \theta_{\hat{y}_i}^{(t)}, \quad \text{where } \hat{y}_i = \arg\max_k P(y = k \mid x_i) \tag{2.1}$$

If no sample satisfies the current thresholds, a fallback mechanism selects the top-N most confident predictions across all classes, ensuring that the model continues to receive additional training data. After each iteration, the model is retrained on the expanded labeled set, and its performance is evaluated on a held-out validation set. The best-performing model is checkpointed, and early stopping is applied if no validation improvement is observed for P consecutive rounds.

The threshold for each class is adjusted using the AdjustThreshold function (Algorithm 2), which takes into account the class distribution of selected pseudo-labels up to the current iteration.

2.3 Models

2.3.1 Sentence Transformers

Unlike standard BERT representations that focus on contextual token-level encoding, sentence embedding models are trained to produce fixed-length semantic representations of whole sentences, optimized for similarity, classification, and clustering tasks. Through empirical evaluation, we found that Sentence Transformer models such as `intfloat/ee5-base-v2` significantly outperform traditional BERT-based methods in classification accuracy and pseudo-label stability. Furthermore, sentence

Algorithm 2 AdjustThreshold Function

function AdjustThreshold(class k, round t, pseudo-label counts $\{c_k\}$)
  Base threshold $\theta_0 \leftarrow \theta_{\text{init}} - \Delta \cdot t$
  Total count $C \leftarrow \sum_j c_j + \varepsilon$
  Class ratio $r_k \leftarrow \frac{c_k}{C}$, Expected ratio $r^* \leftarrow \frac{1}{K}$
  Imbalance factor $\delta \leftarrow \sqrt{\frac{r_k}{r^*}}$
  if $\delta > 1.2$ **then**
    $\theta_k \leftarrow \theta_0 + 0.02 \cdot \min(\delta, 2.0)$
  else if $\delta < 0.8$ **then**
    $\theta_k \leftarrow \theta_0 - 0.03 \cdot \max(\delta, 0.5)$
  else
    $\theta_k \leftarrow \theta_0$
  end if
  return $\min(0.98, \max(0.80, \theta_k))$
end function

embeddings allow for efficient cosine similarity computations and downstream model compatibility, making them well-suited for our semi-supervised learning pipeline. For text representation, they offer a good trade-off between semantic richness and computational efficiency. In the experiments, we employ different sentence embedding models and compare their performance on supervised classification on labeled datasets. This provided a basis to choose a sentence transformer model which is suitable for this project.

2.3.2 Classification Models

To build a robust query intent classifier, we compared several model architectures. Given that the preprocessed inputs are sentence embeddings generated by a transformer model, which already provides dense semantic representations, we opted for lightweight and interpretable classifiers instead of end-to-end deep neural networks.

Feed Forward Neural Networks (FFNNs) can model non-linear interactions, but their performance was not significantly superior to tree-based models when trained on a relatively small labeled dataset (due to the cost of manual annotation). Moreover, FF-NNs required more complex tuning, higher computational resources, and offered less interpretability compared to classical models. However, as more labeled data becomes available through ongoing collection and refinement, FF-NNs may provide a promising avenue for future performance enhancement.

Ultimately, we selected LightGBM, CatBoost, and Logistic Regression models using a combination of VotingClassifier meta-learners based on their performance and characteristics. Tree-based models (LightGBM, CatBoost) are highly effective for tabular data such as sentence embeddings and can handle small and imbalanced datasets well. Logistic Regression adds a strong linear component that complements the tree learners. The ensemble benefits from the strengths of each base model and provides more stable and generalized predictions across classes. Furthermore, this ensemble approach allows efficient training on CPU, faster experimentation, and

better explainability — crucial for practical deployment in the GenAI Chat Assistant pipeline.

- **LightGBM:** Light Gradient Boosting Machine (LightGBM) is a fast, distributed, high-performance gradient boosting framework based on decision trees. It is particularly effective for structured data, supports leaf-wise tree growth, and handles large-scale data with high efficiency. Its ability to capture complex non-linear feature interactions makes it well-suited for modeling the embedding representations of queries.
- **CatBoost:** CatBoost is another gradient boosting algorithm based on decision trees, developed by Yandex. It is optimized for categorical feature handling and reduces overfitting via ordered boosting. Unlike LightGBM, CatBoost is more robust when working with smaller datasets or when categorical features (like tokenized keyword IDs) are involved.
- **Logistic Regression:** We also include a regularized Logistic Regression classifier as a linear baseline. Logistic Regression is efficient and provides interpretable decision boundaries in the embedding space. Although limited in modeling non-linear relationships, it often performs competitively when the feature space is well-transformed.
- **VotingClassifier Ensemble:** To leverage the strengths of these diverse models, we employ a *soft voting ensemble* using Scikit-learn's `VotingClassifier`. Each base learner outputs class probabilities, and the final prediction is derived from the average of their predicted probabilities. This approach benefits the robustness and generalization of semi-supervised learning on unseen data by balancing the strengths of tree-based learners (non-linear expressiveness) and linear models (stability and regularization). The ensemble classifier serves as the teacher model in our self-training pipeline, responsible for generating pseudo-labels during iterative training.

The final predicted label $\hat{y}$ is obtained as:

$$\hat{y} = \arg\max_{k} \left(\frac{1}{M} \sum_{m=1}^{M} P_m(y = k \mid x) \right) \tag{2.2}$$

where:

- M is the number of base classifiers,
- $P_m(y = k \mid x)$ is the probability predicted by the m-th classifier for class k,
- $\hat{y}$ is the final predicted class for instance x.

The final prediction for a given query is determined via soft voting, where the class with the highest average calibrated probability across the ensemble is selected, in 2.2. This ensemble design enhances robustness and generalization across intent types, especially in low-resource scenarios.

2.4 Key techniques

Several custom techniques contribute to the performance and applicability of the proposed system:

- Domain-Specific Text Preprocessing: A tailored pipeline removes noisy code blocks, template phrases, and compiler logs from user queries. This improves input consistency and embedding quality.
- Confidence-Calibrated Ensemble Classifier: To enhance the reliability of pseudo-labels, we employ an ensemble classifier calibrated with Platt scaling (sigmod). This ensures that predicted probabilities are well-aligned with actual confidence.
- Dynamic Thresholding for Pseudo-Label Selection: Instead of using a fixed global threshold for pseudo-labeling, we compute thresholds dynamically per class to ensure balanced pseudo-label acquisition and prevent overfitting on dominant classes.
- Self-Training with Early Stopping Based on Validation Feedback: To avoid error accumulation from noisy pseudo-labels, we adopt early stopping based on validation performance, storing the best model across iterations.
- Intent-Based RAG Triggering: The output of the intent classification model informs whether retrieval from the user's current sketch is needed. For example, queries classified as "fix errors" are more likely to obtain information from sketch context retrieval, while other user queries may be responded to directly. This selective triggering mechanism reduces the need for continuous retrieval, significantly reducing LLM inference cost and latency.

Chapter 3

Data preparation

3.1 Data collection

Arduino GenAI API is a cloud-based service that provides AI-powered assistance for Arduino-related queries. While the API is designed with the potential to expand to other Arduino-related contexts in the future, at present, its main focus is on coding assistance.The API layer manages incoming HTTP requests through RESTful endpoints for conversation management and message processing, secured by JWT-based authentication.

A dedicated conversation management module maintains the state and history of user interactions, enabling contextual responses. The natural language generation (NLG) component is the core of the AI functionality, leveraging Langchaingo, a useful library for building applications with large language models. This NLG component interfaces with Language Model (LLM) runtimes, primarily utilizing AWS Bedrock for AI processing. The use of Langchaingo in the NLG module allows for sophisticated prompt engineering and efficient handling of conversation context.

A repository layer handles data persistence, storing conversation histories. AWS Bedrock, as runtimes that run LLMs usually do, does not save messages sent previously in a conversation. Arduino Gen AI API then uses a database to save conversations and their message history. The dataset used in this study was collected from user queries submitted through the Arduino Cloud Editor since released. Every time a user uses GenAI Assistant, the front end is triggered to upload log details, each query record includes raw text input, timestamp, user metadata, and etc.

For collecting GenAI assistants data, we need to build a ETL pipeline with front-end events. Using the actual architecture for events is possible to expand it by replicate the data split by events type in different tables in Big Query. To populate these tables will be use the feature S3 Data Transfer of Google to read new data from AWS S3 every day. Front-end Event Architecture in figure 3.1 illustrates the flow of telemetry data from front-end user interactions to the cloud-based data warehouse (BigQuery), including steps like event tracking, streaming collection, and ETL (Extract Transform Load). Figure 3.2 shows the data architecture that powers analytics and pipelines, covering data sources, centralized storage (data lake and

warehouse), and downstream activation for dashboards, personalization, and model training.

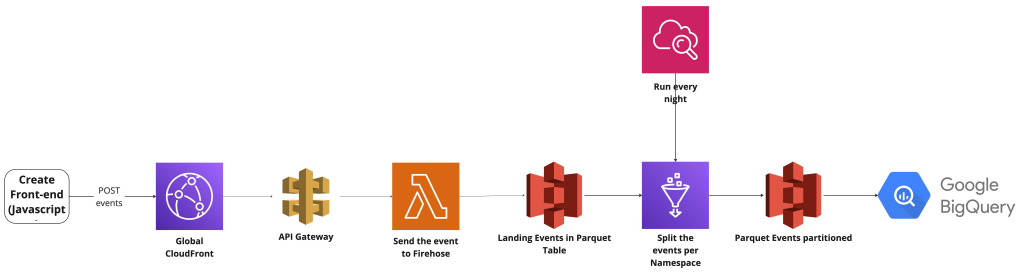


Figure 3.1: Front-end Event Architecture

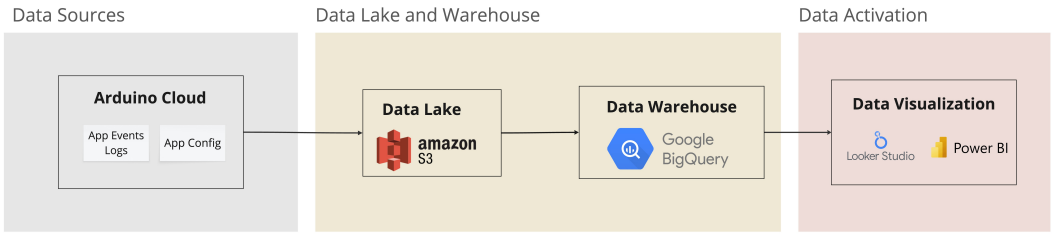


Figure 3.2: Data Lifecycle Architecture

The dataset used in this study was collected daily from AWS S3 and extracted via SQL in BigQuery. In addition, Looker Studio and BigQuery were connected to enable data visualization for the Arduino GenAI assistant.

3.2 Data structure

3.2.1 Data Format

The core dataset used for user query classification was extracted from the BigQuery table. This table logs user interactions with the Arduino Cloud Editor, specifically capturing events related to GenAI assistant usage. We filtered rows where the event category is "ai-question submitted" and the submitted text field is not null. Additionally, only events occurring after GenAI assistant release time were included to ensure relevance and recency. A structured overview of the dataset is presented in Table 3.1.

Field Name	Type	Description
`d_user_id`	STRING	Unique user identifier
`time`	TIMESTAMP	Local timestamp of the query event
`dt`	DATE	Event date in UTC format
`d_question_submitted`	STRING	The user-submitted query text to the GenAI assistant
`label`	STRING	Placeholder for human or model-assigned intent label

Table 3.1: Structure of the extracted query dataset

3.2.2 Dataset Overview

The dataset used in this study was collected from Arduino's production environment between **April 15th, 2025** and **May 17th, 2025**, since the release of the GenAI assistant module. A total of **20,892** user-submitted questions were extracted through SQL queries from BigQuery logs. In the process of labeling data, we try to select user queries of different styles but with clear intentions so that the model can learn more generalized. To train the classification model, we manually labeled a subset of samples according to the developed rules and evenly distributed them into four predefined intent categories: *Create code*, *Explain*, *Suggest*, and *Fix errors*. Then we select 800 items from all the labeled samples, which is about 10% of the total training dataset. To prevent the model from being biased towards most types, we select an average labeled data set, that is, 200 samples of each type. This labeled set serves as the seed data for semi-supervised training.

For evaluation purposes, a hold-out testing dataset was collected from a later time period, spanning **May 18th, 2025** to **June 17th, 2025**.

Collection	Details
Training dataset	"2025-04-15" to "2025-05-17"
Total user queries extracted	20,892
Labeled samples	800 (200 per intent class)
Testing dataset	"2025-05-18" to "2025-06-17"

Table 3.2: Dataset Information Summary

3.3 Data Processing

3.3.1 Purpose

The raw user-submitted questions collected from the Arduino Cloud Editor interface contain various non-informative patterns, including fixed template hints, pure source code, and compiler-generated error messages. This information is very difficult for models to process because it is often difficult to extract useful semantic information. In addition, the default language of the GenAI assistant is English, but users can send arbitrary information in any language even if they know the usage rules. Therefore, in order to prepare these inputs for downstream classification, we developed a robust data processing pipeline with the following objectives:

- Filter out irrelevant or low-quality inputs that do not contain meaningful user intent.
- Extract and normalize natural language queries suitable for intent classification.
- Annotate and categorize questions into predefined types for further modeling.
- Language detection, select valid user query statements in English.

3.3.2 Processing Pipeline

The preprocessing pipeline consists of multiple steps, implemented sequentially to ensure high-quality input:

1. **Template and Pattern Detection:** Questions matching pre-defined system templates (e.g., 'Hints', 'Fix errors') or button-triggered actions in the IDE are tagged and removed.
2. **Code Snippet Removal:** The `remove_code_snippets(text)` function is used to remove pure code blocks based on common C/C++ syntax structures, including:
   - C/C++ comments (`//...`, `/*...*/`)
   - Preprocessor directives (e.g., `#include`, `#define`)
   - Paths and error stack traces
   - Enum, struct, and class definitions
   - Variable and constant declarations
   - Function definitions (including nested braces)
   - Empty lines and redundant whitespace
3. **Language and Validity Checks:** Non-English inputs are identified and excluded using language detection algorithms. Very short or malformed texts are also discarded.
4. **Text Normalization:** Remaining natural language inputs are lowercased, stripped of punctuation, and normalized to reduce noise and lexical variance.

3.3.3 Preprocessing Output and Category Definition

To filter out low-value or noisy inputs, we applied a multi-stage preprocessing pipeline that removes code-only inputs, compiler error logs, fixed UI templates, and non-English queries. Based on this logic, each question is categorized into one of the following five groups:

- **To Classify:** Cleaned and valid English-language queries that are retained for downstream classification (i.e., create code, explain, suggest, fix error).
- **Fix Errors:** Automatically submitted via the "Fix Errors" button in the IDE. These often include compiler traces or problem statements with minimal natural language.
- **No English:** Queries identified as non-English or without any meaningful English content.
- **Code:** Inputs composed only of source code, with no accompanying natural language context or question.

- **Hints:** Pre-filled instructional text provided by the UI, not authored by users themselves.

Figure 3.3 shows the distribution of these categories. Notably, about 40% of the total dataset falls into the 'To Classify' category, demonstrating that a substantial portion of user input is suitable for intent understanding. The remaining categories, while not directly used for classification, help identify patterns of user interaction and behavior with the assistant system.

We will train and classify the model on the 'To Classify' dataset. We will take about 10%-15% of the data for manual labeling, and then select non-repetitive and high-quality labeled data for the initial learning of semi-supervised learning, as the teacher model [19]. Subsequently, multiple rounds of self-training are performed on the remaining unlabeled 'To Classify' dataset, and a new student model is generated by adding model predictions and high-confidence pseudo labels. Each round of new models needs to be verified on the validation set to find the best model for testing.

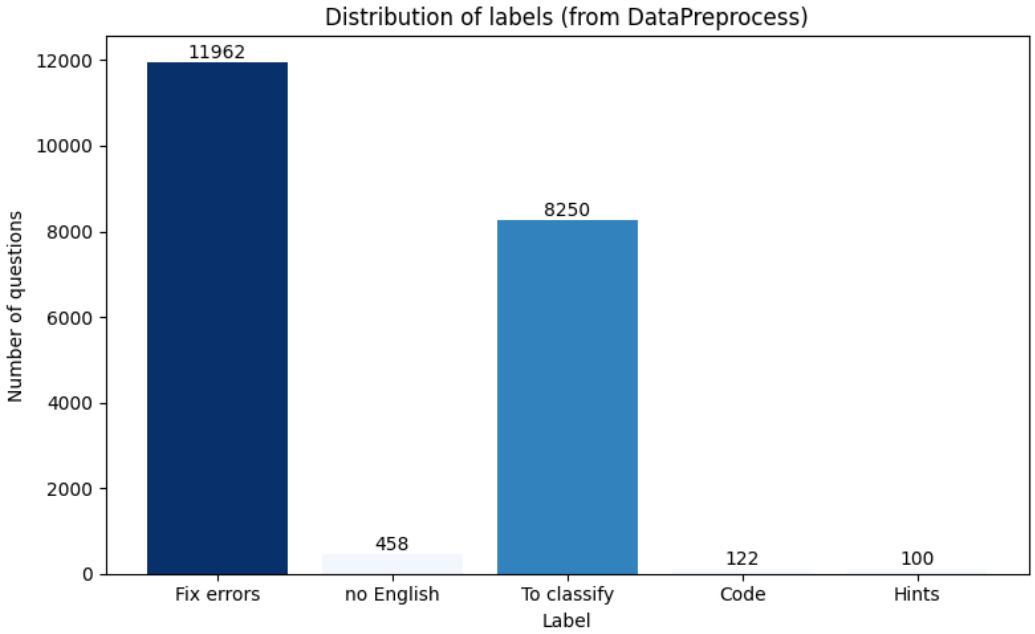


Figure 3.3: Distribution of categories after data preprocessing

Chapter 4

Experiments and Results

4.1 Evaluation Metrics

We evaluate our model on the validation set using standard classification metrics: **precision**, **recall**, **F1-score**, and **accuracy**. To address the class imbalance present in our dataset, we emphasize metrics that reflect per-class performance rather than relying solely on overall accuracy.

Here are some important evaluation metrics:

- **Macro F1-score**: Reports the unweighted average of F1-scores across all classes. This metric treats each class equally and is critical to ensure that the performance on low-frequency but semantically important classes is not overshadowed by the dominant class.
- **Weighted F1-score**: Computes the average F1-score weighted by class frequencies, reflecting the overall performance more realistically when class imbalance exists.
- **Per-class Recall**: Especially emphasized for the *Suggest*, *Explain*, and *Create Code* categories, as these directly affect downstream Retrieval-Augmented Generation (RAG) pathways. High recall ensures accurate routing of queries to the appropriate knowledge modules.
- **Accuracy**: While providing a general performance overview, accuracy can be misleading in imbalanced settings and is therefore interpreted in conjunction with the above class-sensitive metrics.

We prioritize models with strong Macro F1-scores and balanced per-class recall to ensure consistent and robust user experience in the assistant's multi-intent environment.

Beyond general classification performance, we introduce additional metrics focused on optimizing the injection of `sketch` context. The primary objective is to accurately identify *Fix Errors* queries that genuinely require sketch context, while minimizing unnecessary sketch loading to reduce inference cost and latency. We report the following targeted indicators:

- **Recall in 'Fix errors'**: Measures the proportion of true *'Fix errors'* correctly identified by the model. High recall is essential to avoid missing error-fix queries that depend on sketch context.
- **Precision in 'Fix errors'**: Measures the proportion of predicted *'Fix errors'* that are actually true error-fix queries. High precision ensures that sketch context is not injected unnecessarily.
- **F1 in 'Fix errors'**: The harmonic mean of precision and recall for the *'Fix errors'* class. This balances the trade-off between missing true positives and injecting context incorrectly.

In our cost-sensitive design, we prioritize reducing false sketch injections while maintaining high recall on true *'Fix errors'*, striking a balance between performance, resource efficiency, and user experience.

4.2 Experimental Setup

4.2.1 Dataset Description

The experiments are conducted on the preprocessed dataset `toclassify.csv`, which contains user-submitted questions, filtered and processed data. The data collection period is from April 15, 2025 to May 17, 2025, with a total of 8250 samples. A subset of 800 instances was manually annotated into four intent categories, each with 200 samples. For training dataset, it includes 640 labeled data and 7450 unlabeled data. Validation dataset includes 160 labeled data, 40 per each class.

To enable effective classification of user queries and guide appropriate AI assistant responses, we define four semantic categories: `Create code`, `Explain`, `Suggest`, and `Fix errors`. This categorization was derived through manual inspection of real-world user submissions collected from the Arduino Cloud Editor platform. These categories also corresponds to the function buttons designed by Arduino GenAI Assistant UI (User Interface).

- **Create code (label 0):** Questions explicitly requesting code generation from scratch. These queries typically contain functional descriptions or goals (e.g., "How to blink two LEDs alternately?").
- **Explain (label 1):** Requests for explanation or clarification of code snippets, hardware components, or abstract behaviors. Examples include "What does this function do?" or "Why is the LED blinking irregularly?"
- **Suggest (label 2):** Open-ended questions asking for recommendations, improvements, or design strategies. This includes queries such as "What sensor should I use?" or "How to make it more energy efficient?"

- **Fix errors (label 3):** Problem-oriented queries, often associated with compilation errors, runtime bugs, or debugging symptoms. This category is closely tied to the Cloud Editor's 'Fix Error' feature.

This label classification is motivated by both empirical observations and UI design features. The Arduino GenAI Assistant provides pre-defined 'Hints' and 'Fix errors' prompts, guiding users to interact with the assistant in ways that naturally reflect these categories. Additionally, this classification aligns well with the functional needs of a RAG framework, where identifying the type of question (e.g., code generation vs. explanation) helps in:

- Deciding whether sketch-level context should be retrieved for grounding the response.
- Reducing unnecessary retrieval overhead when the query is self-contained (e.g., suggest-style or simple explanations).
- Improving LLM response specificity by conditioning prompts on the query type.

In this way, our four-class setup not only reflects authentic user intent but also enhances the efficiency and effectiveness of the downstream GenAI system by enabling structured query understanding and response selection.

4.2.2 Text Representation

To effectively represent short user-submitted queries, we evaluated several pre-trained sentence embedding models, focusing on the trade-off between semantic quality, vector dimensionality, and computational efficiency. The models considered include:

- `all-MiniLM-L6-v2`
- `all-mpnet-base-v2`
- `intfloat/e5-base-v2`
- `mixedbread-ai/mxbai-embed-large-v1`
- `moka-ai/m3e-base`
- `BAAI/bge-large-en-v1.5`
- `BAAI/bge-m3`
- `nomic-ai/nomic-embed-text-v1`

When conducting supervised learning training, the entire labeled dataset is standard scaled before sentence embedding. Each model was tested only with labeled dataset and assessed based on key evaluation metrics and embedding dimension. The accuracy of the validation set can more directly measure the model's generalization

Model	Accuracy	Precision	Recall	F1-score	Dimension
all-MiniLM-L6-v2	0.63750	0.642522	0.63750	0.639376	384
all-mpnet-base-v2	0.68750	0.691805	0.68750	0.687595	768
e5-base-v2	0.78750	0.799925	0.78750	0.789513	768
mxbai-embed-large-v1	0.73750	0.749781	0.73750	0.740385	1024
m3e-base	0.75000	0.762982	0.75000	0.752885	768
bge-large-en-v1.5	0.76875	0.780319	0.76875	0.771503	1024
bge-m3	0.77500	0.784872	0.77500	0.776229	1024
nomic-embed-text-v1	0.75625	0.768823	0.75625	0.759085	768

Table 4.1: Performance comparison of various embedding models

ability for 'unseen data', which meets our needs for evaluating the generalization performance of the embedding model for real new samples. Table 4.1 summarizes the performance of all models.

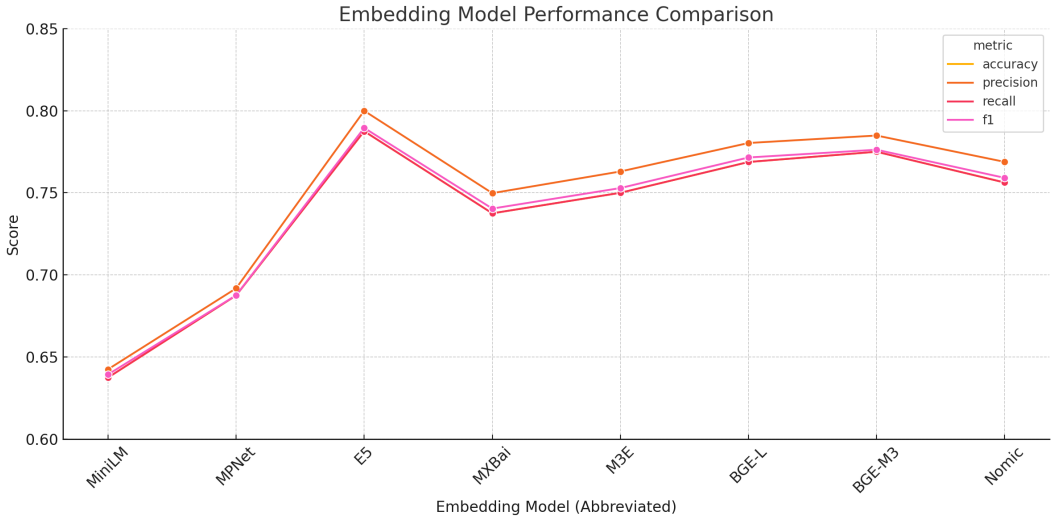
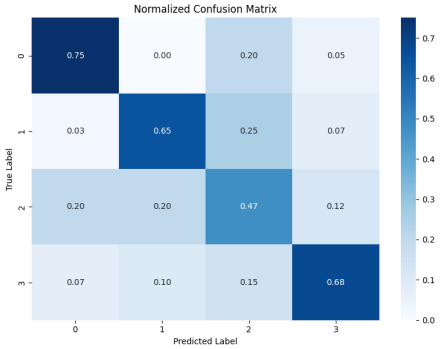


Figure 4.1: Embedding models comparison

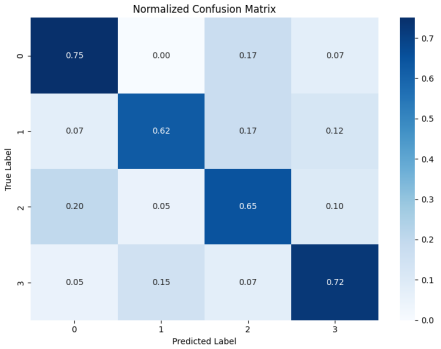
After empirical comparison on downstream classification performance and inference efficiency (figure 4.1), we selected `intfloat/e5-base-v2` as the final embedding model. It achieved the best balance between accuracy and speed in our semi-supervised classification pipeline. Specifically, it outperformed lighter models such as `all-MiniLM-L6-v2` on macro F1-score, while maintaining significantly faster inference compared to larger models like `mixedbread-ai/mxbai-embed-large-v1` or `BAAI/bge-large-en-v1.5`. Furthermore, `e5-base-v2` is optimized for embedding short texts and questions, aligning well with the nature of our dataset.

Figure 4.2 shows the confusion matrix after using different sentence transformers, reflecting the prediction results for each category. `e5-base-v2` has relatively high prediction accuracy in all categories and can better identify 'Suggest' category, which is usually difficult to distinguish.

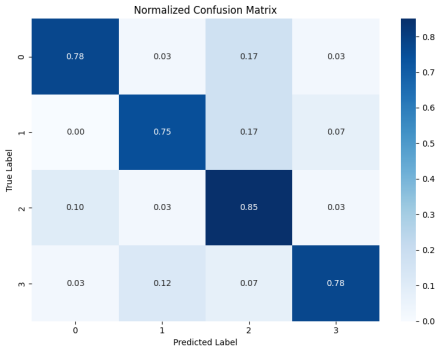
Considering both performance and efficiency, we selected intfloat/e5-base-v2 for the main experiments due to its balance between high evaluation metrics, fast



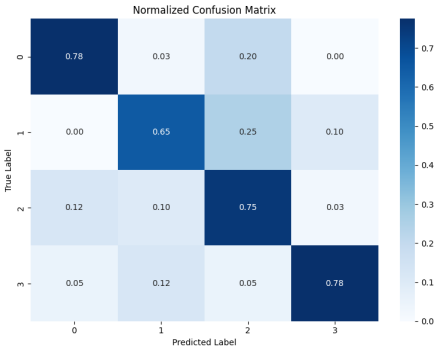
(a) MiniLM



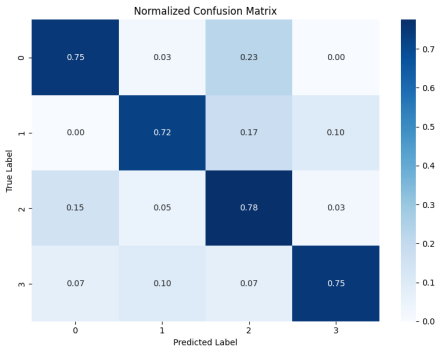
(b) MPNet



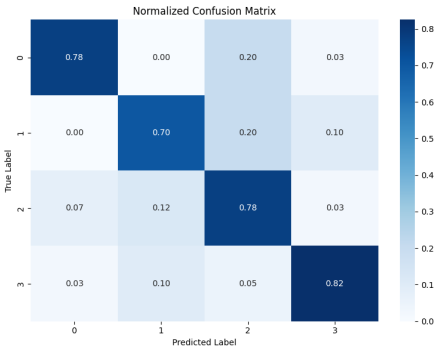
(c) E5-base-v2



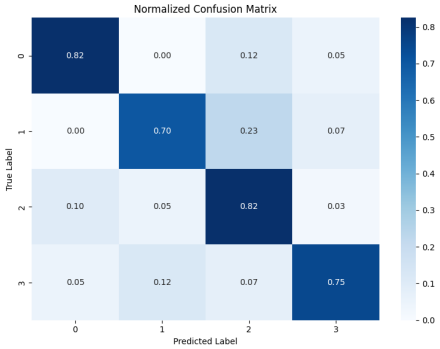
(d) MXBai



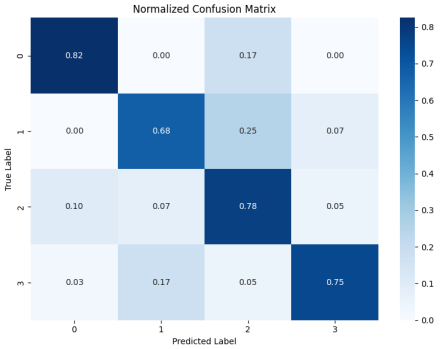
(e) M3E



(f) BGE-Large



(g) BGE-M3



(h) Nomic

Figure 4.2: Confusion matrices of different embedding models evaluated on validation set

inference time, and moderate embedding dimension. These sentence embeddings were computed once and used as static input features to the classifier, enabling efficient reuse during iterative self-training.

It should be noted that the selection of this sentence embedding model is only performed in the labeled dataset, and the same validation set is used to compare the performance. Because we only want to compare the embedding and understanding capabilities of the sentence transformer for this type of queries, it is more time-saving and resource-saving to directly use the supervised learning classification for comparison.

4.2.3 Classifier Architecture

Before finalizing the ensemble-based classifier, we systematically evaluated a set of baseline models on the cleaned and embedded textual dataset. The goal was to identify models that offer complementary strengths in terms of generalization, robustness, and efficiency, especially under the constraints of noisy, semi-supervised, and imbalanced data. The experiments are based on the use of the intfloat/e5-base-v2 embedding transformer combined with supervised learning classifiers on a labeled dataset.

We evaluate a range of individual classifiers and ensemble models to determine the most suitable base learner for our self-training semi-supervised learning framework. As shown in Table 4.2, several models demonstrate competitive performance across Accuracy, Precision, Recall, and F1-score. As shown in the figure 4.3, VotingClassifier2, which combines Logistic Regression, CatBoost, and LightGBM, achieves the best overall performance with F1-score of 0.7895 and an accuracy of 0.7875, outperforming all other models including single learners and alternative ensembles.

Model	**Accuracy**	**Precision**	**Recall**	**F1-score**
LogisticRegression	0.70625	0.704994	0.70625	0.704228
RandomForest	0.70625	0.733836	0.70625	0.707681
XGBoost	0.74375	0.750725	0.74375	0.744940
LightGBM	0.70625	0.730186	0.70625	0.710782
CatBoost	0.71250	0.721055	0.71250	0.714524
VotingClassifier1 (lr+cat+xgb)	0.76875	0.779535	0.76875	0.770529
VotingClassifier2 (lr+cat+lgb)	0.78750	0.799925	0.78750	0.789513

Table 4.2: Performance comparison of various classifiers.

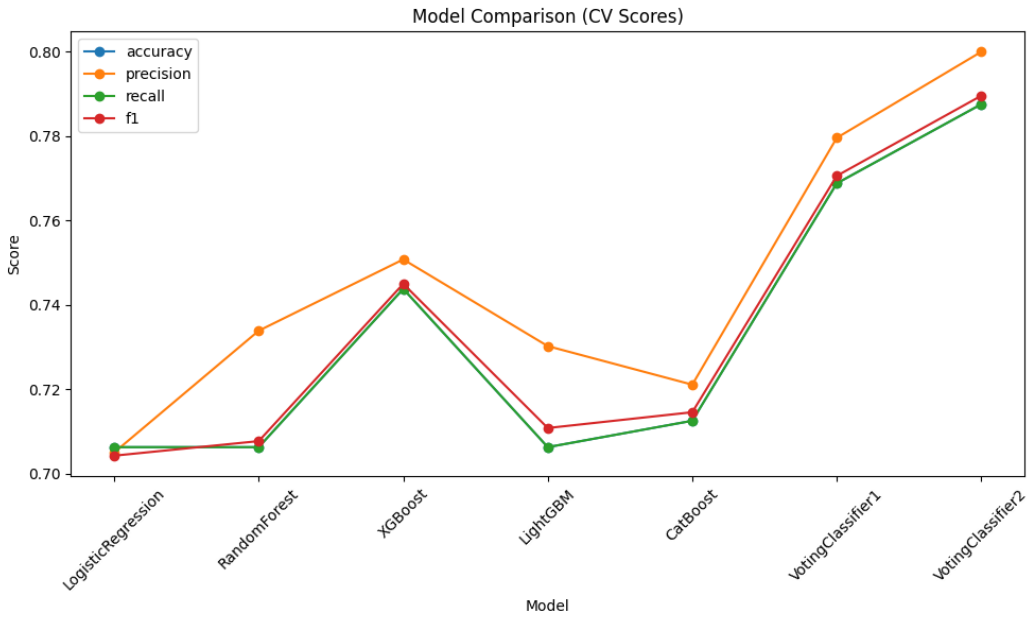


Figure 4.3: Classifier comparison

Interestingly, although XGBoost achieves the highest performance among individual models (F1-score = 0.7449), it does not lead to the strongest ensemble when integrated into a voting classifier (VotingClassifier1). We hypothesize that this is due to reduced model diversity when XGBoost is combined with CatBoost and Logistic Regression, as XGBoost and CatBoost may produce similar decision boundaries, limiting the benefit of ensembling. In contrast, LightGBM exhibits a more complementary behavior when combined with the other two models (VotingClassifier2), possibly due to its histogram-based tree learning and leaf-wise split strategy, which improves generalization on pseudo-labeled and noisy data.

The superior performance of VotingClassifier highlights the importance of model diversity and complementarity in ensemble design, especially in semi-supervised learning where label noise and distributional uncertainty are prominent. It was constructed by combining LightGBM, CatBoost, and Logistic Regression, while using soft voting. This combination exploits the linearity of Logistic Regression, the efficiency of LightGBM, and the regularization robustness of CatBoost. Specifically, the ensemble VotingClassifier includes:

- **LightGBM**: A fast and lightweight gradient boosting implementation optimized for efficiency. It is particularly suitable for large-scale datasets and handles categorical features well. The model uses 300 estimators, a learning rate of 0.05, and a maximum depth of 6. Class balancing is handled via `class_weight='balanced'`.

- **CatBoost**: Known for its robustness to categorical feature encoding and hyperparameter sensitivity. It is configured with 300 iterations, depth of 6, and learning rate 0.05. It uses class-specific weights calculated from the label distribution to handle imbalance.

- **Logistic Regression**: Included for its simplicity, interpretability, and ability to capture linear relationships. We use the SAGA solver with balanced class weights and allow up to 1000 iterations for convergence.

LightGBM offers fast and efficient gradient boosting, CatBoost is robust to categorical features and hyperparameters, and Logistic Regression provides a simple, interpretable baseline. The ensemble uses soft voting to aggregate model predictions, further calibrated by CalibratedClassifierCV to improve probability reliability. Additionally, class imbalance is addressed through balanced class weights, ensuring better performance across minority classes. This ensemble design ultimately balances interpretability, training efficiency, and generalization performance. It consistently outperforms all individual classifiers in validation set accuracy and F1-score, making it the first choice for downstream semi-supervised training processes.

4.3 Dynamic Thresholds and Sampling Strategy

To ensure both prediction reliability and class distribution balance in the pseudo-labeling process, we adopt a **dynamic, class-wise thresholding** strategy. For each class $k \in \{0, \ldots, 3\}$ and self-training round t, we compute a confidence threshold $\theta_k^{(t)}$ based on two key factors: the training round index and the current pseudo-label distribution.

Dynamic Thresholds The base threshold for each round t is defined as:

$$\theta_0^{(t)} = \theta_{\text{init}} - \Delta \cdot t$$

where θ_{init} is the initial confidence threshold (e.g., 0.84), and Δ is a hyperparameter controlling the rate of threshold decay (e.g., 0.01). This decay enables the model to start with highly confident pseudo-labels and gradually include more diverse samples as its generalization ability improves.

Table 4.3 summarizes the model performance across different values of the initial confidence threshold θ_{init} used in the dynamic thresholding strategy during self-training. We observe that the choice of θ_{init} has a noticeable impact on the classification metrics.

Initial Threshold	F1-score	Accuracy	Precision	Recall
0.82	0.8071	0.8063	0.8193	0.8063
0.84	0.8074	0.8063	0.8193	0.8063
0.86	0.8019	0.8000	0.8107	0.8000
0.88	0.8019	0.8000	0.8159	0.8000
0.90	0.8016	0.8000	0.8160	0.8000
0.92	0.7974	0.7937	0.8174	0.7938

Table 4.3: Performance metrics under different initial thresholds

Notably, the best overall F1-score of 0.8074 is achieved at an initial threshold of

0.84, which also corresponds to a balanced precision (0.8193) and recall (0.8063) with best performance. This suggests that starting with this threshold enables the model to maintain high confidence in pseudo-labels while providing adequate coverage of the various categories.

Thresholds slightly lower or higher than 0.84 generally result in a modest drop in performance, indicating a sensitivity of the model to the initial threshold setting. Interestingly, thresholds above 0.90 do not further improve performance and sometimes lead to reduced recall. It may be due to the reduced number of samples passing through the confidence filter, thus limiting the augmentation of training data.

Overall, our experiments demonstrate that careful tuning of the initial confidence threshold θ_{init} is essential for balancing pseudo-label quality and quantity, which directly impacts the effectiveness of the self-training pipeline.

Class Imbalance Adjustment To address the imbalance in pseudo-label frequency, we compute the ratio of assigned pseudo-labels for class k up to round t:

$$r_k^{(t)} = \frac{c_k^{(t)}}{\sum_{j=1}^{K} c_j^{(t)} + \varepsilon} \quad \text{with} \quad r^* = \frac{1}{K}$$

We then define the imbalance factor as:

$$\delta_k = \sqrt{\frac{r_k^{(t)}}{r^*}}$$

and update the class-wise threshold nonlinearly:

$$\theta_k^{(t)} = \begin{cases} \theta_0^{(t)} + \alpha \cdot \min(\delta_k, 2.0) & \text{if } \delta_k > 1.2 \\ \theta_0^{(t)} - \beta \cdot \max(\delta_k, 0.5) & \text{if } \delta_k < 0.8 \\ \theta_0^{(t)} & \text{otherwise} \end{cases}$$

where α and β are the imbalance adjustment coefficients. The values 0.02 and 0.03 were chosen empirically to reflect a mild asymmetric adjustment strategy: we penalize dominant classes conservatively while encouraging under-represented classes more aggressively. This asymmetry helps stabilize class balance over iterations without overly sacrificing label confidence.

In our threshold adjustment function, we adopt hyper-parameters such as the increase/decrease coefficients (0.02/0.03) and imbalance thresholds (1.2/0.8) to control the sensitivity and response of the system to class distribution shifts. These values are empirically chosen to provide a stable yet responsive mechanism for pseudo-label correction. While we found the default values to work well across settings, future work could explore tuning these coefficients to further optimize performance under extreme class imbalance scenarios.

To ensure stability, we clip the final threshold to a valid range:

$$\theta_k^{(t)} \in [0.80, 0.98]$$

Sample Selection A sample x_i is added to the pseudo-labeled set only if the predicted class confidence exceeds its class-specific threshold:

$$\max_k P(y = k \mid x_i) \geq \theta_{\hat{y}_i}^{(t)}, \quad \text{where } \hat{y}_i = \arg\max_k P(y = k \mid x_i)$$

The specific threshold also changes dynamically with the amount of pseudo-label data added.

Decay rate To investigate the effect of the confidence threshold decay rate Δ, we conducted experiments using different decay rates $\Delta \in \{0.01, 0.02\}$ under several initial thresholds$\theta_{\text{init}} \in \{0.82, 0.84, 0.86\}$. The results are shown in Table 4.4.

Initial Threshold	Δ	F1-score	Accuracy	Precision	Recall
0.84	0.1	0.8074	0.8063	0.8193	0.8063
	0.2	0.7963	0.7937	0.8108	0.7937
0.86	0.1	0.8019	0.8000	0.8107	0.8000
	0.2	0.8011	0.8000	0.8096	0.8000
0.88	0.1	0.8019	0.8000	0.8159	0.8000
	0.2	0.7897	0.7875	0.8010	0.7875

Table 4.4: Effect of Varying Δ on Performance under Different Initial Thresholds

For all initial thresholds, a smaller decay rate ($\Delta = 0.1$) leads to consistently higher or comparable F1-score and accuracy compared to a more aggressive decay ($\Delta = 0.2$). This supports the hypothesis that a more conservative threshold decay allows the model to maintain higher quality pseudo-labels, especially in the early rounds.

4.4 Self-Training

4.4.1 Hyper-parameter Settings

The hyper-parameters for the self-training framework were selected based on common practices in semi-supervised learning and validated through empirical observations on the validation set performance. The following key parameters were used:

- **Max pseudo-labels per class per iteration** (max_pseudo): Controls the number of pseudo-labeled samples added per class in each iteration. To avoid excessive noisy labels, max_pseudo was set to 30–80, approximately 18%–40% of the labeled data per class.
- **Maximum iterations** (round): The self-training process was allowed to run for up to 15 iterations. This setting provides sufficient opportunities for the model to incorporate useful pseudo-labeled data, while avoiding excessive training and low quality learning.

- **Early stopping patience** (patience): The training process was monitored on the validation set F1-score. If no improvement was observed for 5 consecutive iterations, the training would be stopped early to prevent overfitting.

Given that the labeled set contains 640 examples and the unlabeled pool contains 7450 examples, we set max_pseudo to 30–80 per class per iteration during experiments. This range corresponds to approximately 18%–40% of the labeled data per class, striking a balance between introducing sufficient pseudo-labeled data and mitigating the risk of noise accumulation. Then find the optimal maximum number of pseudo labels introduced in each round of self-training.

We combined experience and multiple experiments to finally set the maximum number of iterations to $T = 15$ and the early stopping patience to $P = 5$. These settings allow the model to gradually incorporate confident pseudo-labeled data while preventing overfitting in subsequent iterations. Too few training rounds or stopping too early may result in insufficient training and the model may not be able to fully learn. Too many training rounds or too high a patience may cause the subsequent models to continue to learn incorrect pseudo-labels, resulting in poor results.

4.4.2 Self-Training Loop

The choice of max_pseudo and self-training rounds is critical in semi- supervised learning, as they jointly determine the total number of pseudo-labeled samples added during training. We set the number of self-training iterations to 15, with early stopping to prevent overfitting noisy pseudo-labels. Early stopping was performed with patience of 5 iterations based on the validation F1-score to avoid overfitting on the noisy pseudo-labeled data.

The hyperparameter max_pseudo determines the maximum number of pseudo-labeled samples added to the training set in each iteration. From the table 4.5 and the figure 4.4, we observe an obvious trend:

- Performance improves consistently as max_pseudo increases from 30 to 60.
- The best results are achieved at max_pseudo = 60, where both the F1 score (0.8074) and the accuracy (0.8063) reach their peaks.
- Beyond 60, the performance decreases slightly. This indicates that too many pseudo-labeled samples may introduce noise and impair generalization ability.

max_pseudo	F1-score	Accuracy	Precision	Recall
30	0.7909	0.7875	0.8066	0.7875
40	0.7911	0.7875	0.8098	0.7875
50	0.8027	0.8000	0.8170	0.8000
60	**0.8074**	**0.8063**	**0.8193**	**0.8063**
70	0.8032	0.8000	0.8178	0.8000
80	0.8018	0.8000	0.8096	0.8000

Table 4.5: Performance under different `max_pseudo` values

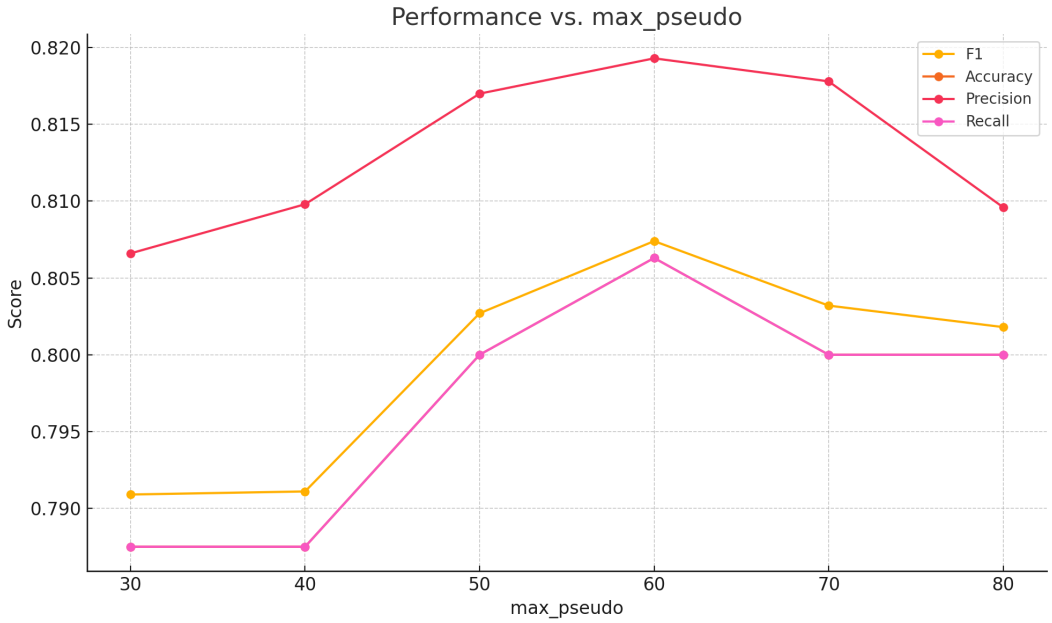


Figure 4.4: Matrics with different `max_pseudo`

This suggests that moderate inclusion of high-confidence pseudo labels can enhance the model's learning. However, excessively aggressive inclusion (e.g., max_pseudo > 70) may dilute label quality, especially if pseudo-label thresholds are not sufficiently strict.

4.5 Results

4.5.1 Hyperparameters in semi-supervised model

After the experiment, we found the best model parameters, as shown in the table 4.6. We have to notice that during preprocessing, the supervised model applied standard scaling based solely on the labeled training set, while the semi-supervised model fitted the scaler on both labeled and unlabeled data. This difference reflects a more realistic feature distribution in the semi-supervised setup and could contribute to better generalization, as the model sees a broader range of variation during normalization. All evaluation, however, is consistently conducted on the same validation set.

Hyperparameter	**Value**
Initial threshold (θ_{init})	0.84
Decay per round (Δ)	0.1
Threshold range	[0.80, 0.98]
Maximum pseudo-labeled samples per class (max_pseudo)	60
Patience for early stopping (patience)	5
Maximum training rounds (rounds)	15
Top-k fallback selection (fallback_topn)	50

Table 4.6: Final Hyperparameter Settings for Self-training

4.5.2 Validation Performance Between Supervised and Semi-supervised Models

To evaluate the effectiveness of our self-training approach, we compare the performance of a purely supervised model trained on labeled data only with that of a semi-supervised model incorporating both labeled and pseudo-labeled data. Table 4.7 and Table 4.8 summarize the performance on the same validation set.

Model	Accuracy	Precision	Recall	F1-score
Supervised Only	0.7875	0.7999	0.7875	0.7895
Semi-supervised (Self-training)	**0.8063**	**0.8193**	**0.8063**	**0.8074**

Table 4.7: Overall performance comparison on validation set

As shown, the semi-supervised model achieves consistent improvements across all metrics, with an absolute gain of +1.8% in accuracy and F1-score. This suggests that self-training can effectively leverage unlabeled data to enhance generalization.

Class	Model	Precision	Recall	F1-score
0 (create code)	Supervised	0.86	0.78	0.82
	Semi-supervised	**0.89**	**0.82**	**0.86**
1 (explain)	Supervised	0.81	0.75	0.78
	Semi-supervised	0.81	0.75	0.78
2 (suggest)	Supervised	0.67	0.85	0.75
	Semi-supervised	**0.69**	**0.90**	**0.78**
3 (fix error)	Supervised	0.86	0.78	0.82
	Semi-supervised	**0.88**	0.75	0.81

Table 4.8: Performance comparison between supervised & semi-supervised

Classification results reveal that the largest improvement is observed in class 0 and class 2. The F1-score for class 0 (Create code) improves from 0.82 to 0.86, while

class 2 (Suggest) benefits from a higher recall (0.90 vs. 0.85) and better F1-score (0.78 vs. 0.75). This indicates that self-training allows the model to better generalize to under-represented or more ambiguous categories by incorporating informative pseudo-labeled examples.

Meanwhile, the performance of class 1 (Explain) remained stable after self training. The performance of class 3 (Fix errors) changed slightly, with a relative improvement in precision. However, the slight decrease in recall and F1-score could be due to pseudo-label noise or category similarity.

Interestingly, while the F1-score for class 3 (Fix errors) remains comparable between supervised and semi-supervised models, the semi-supervised model achieves a higher precision (0.88 vs. 0.86) but slightly lower recall (0.75 vs. 0.78). This indicates that the model becomes more conservative in assigning the 'fix errors' label. Such behavior can be beneficial in downstream applications like sketch update systems, where it is more desirable to avoid introducing incorrect error-fix patterns than to capture every possible fix scenario. Our goal is to save resources in the model and reduce API calls to Sketch. Therefore, this model only identifies 'fix errors' when it is more certain, reducing the risk of mistakenly introducing other types of problems into the fix logic.

These results confirm the effectiveness of pseudo-labeling in improving both overall accuracy and class-level balance, especially for categories where labeled data is insufficient.

4.5.3 Normalize Confusion Matrix Analysis

Figure 4.5 and Figure 4.6 show the normalized confusion matrices of the supervised baseline and the self-training model, respectively. Each row is normalized to represent the distribution of predicted classes given a ground truth label.

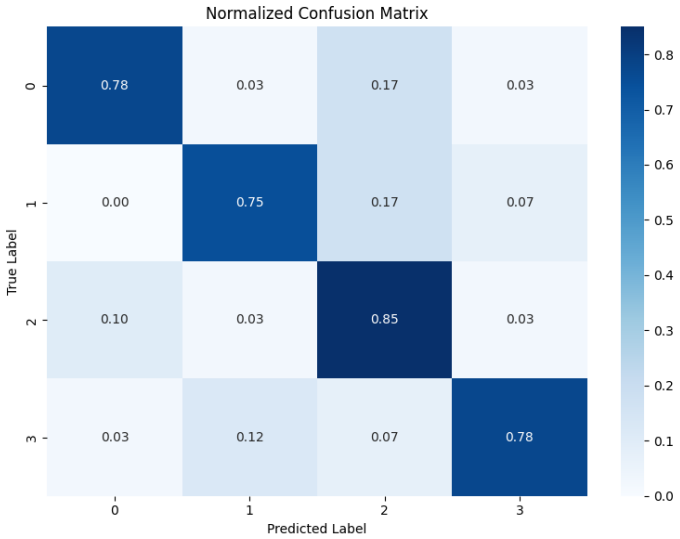


Figure 4.5: Normalize Confusion Matrix with supervised learning

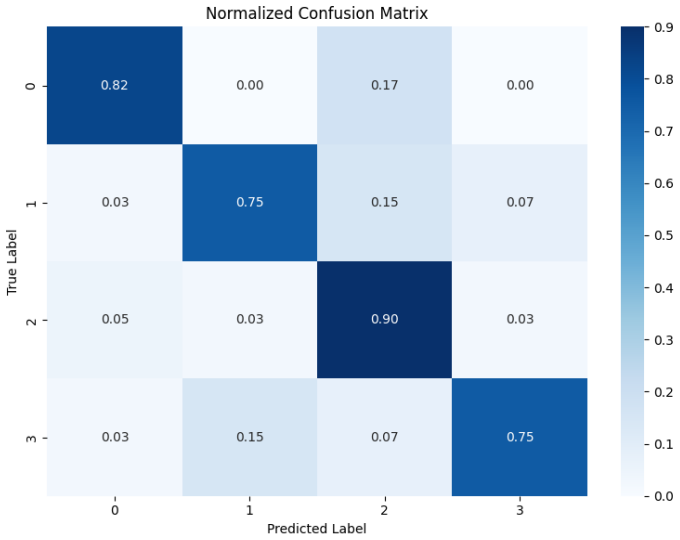


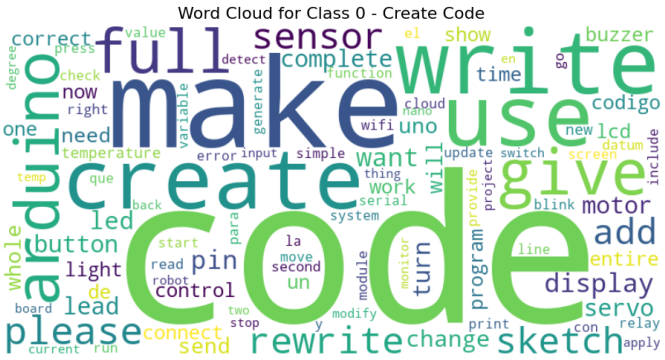
Figure 4.6: Normalize Confusion Matrix with semi-supervised learning

The normalized confusion matrices reveal that the semi-supervised model not only enhances class-specific precision for categories like "create code" and "suggest," but also maintains or improves recall for most classes. In particular, although the recall for "fix error" slightly decreased ($0.78 \rightarrow 0.75$), the corresponding precision increased ($0.86 \rightarrow 0.88$), suggesting more reliable detection of error-fix requests—critical in downstream sketch loading tasks. Nevertheless, the overall diagonal dominance in the confusion matrix improves, highlighting better class discrimination. These results support that semi-supervised training can achieve more precise decision boundaries and higher generalization capabilities despite relying on noisy pseudo-labels.

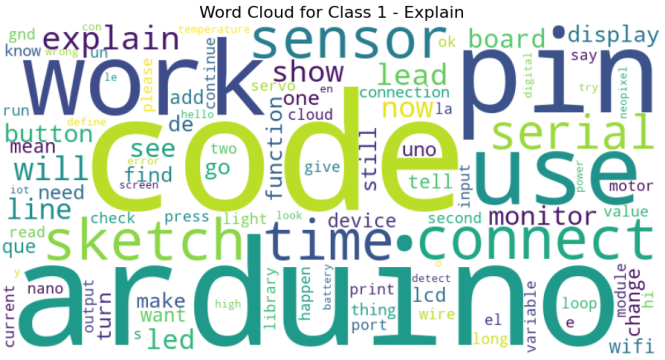
4.5.4 Word Cloud Visualizations

We include class-specific word clouds and confusion matrices (sanity check vs self-training) to illustrate classification effectiveness and data coverage.

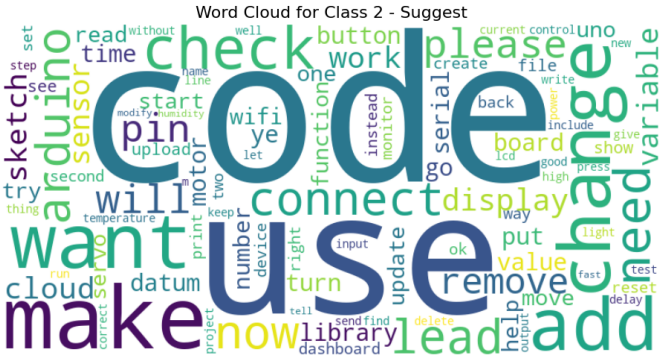
To visualize the semantic characteristics of each class, we generated word clouds for the data after using model prediction. Each cloud in figure 4.7 highlights the most frequent and relevant tokens per class, reflecting typical user queries. For instance, the *Fix errors* class frequently contains terms like 'error', 'fix', 'sketch' and 'code', while *Suggest* or *Explain* categories show more diverse conceptual keywords and verbs.



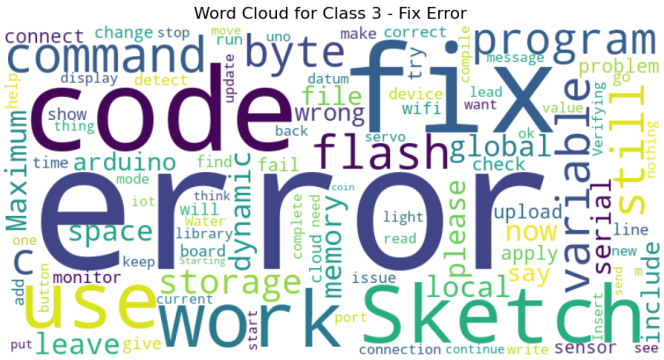
(a) Create code class



(b) Explain class



(c) Suggust class



(d) Fix errors class

Figure 4.7: Word Clouds per Class

4.5.5 Self training observation

Figure 4.8 shows the model performance in each round of training during the self-training process. It can be observed that after the fifth round of semi-supervised learning, the metrics reached the highest value, and then the early stopping strategy began.

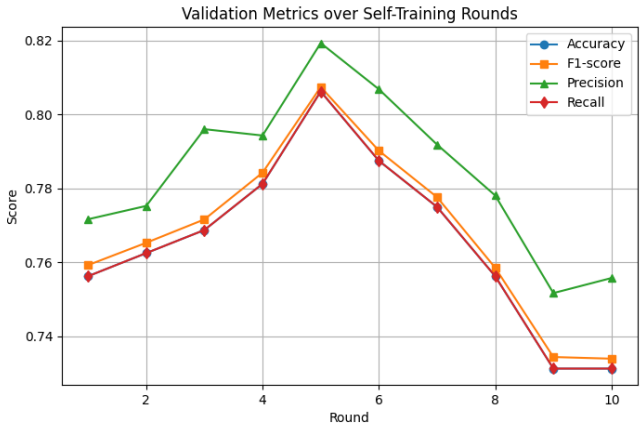


Figure 4.8: Validation Matrics during self-training

To better understand the behavior of the self-training mechanism, we analyzed the number of pseudo labels accepted for each class at every training round. Figure 4.9 illustrates the class-wise distribution of pseudo-labeled samples added per round throughout the iterative learning process.

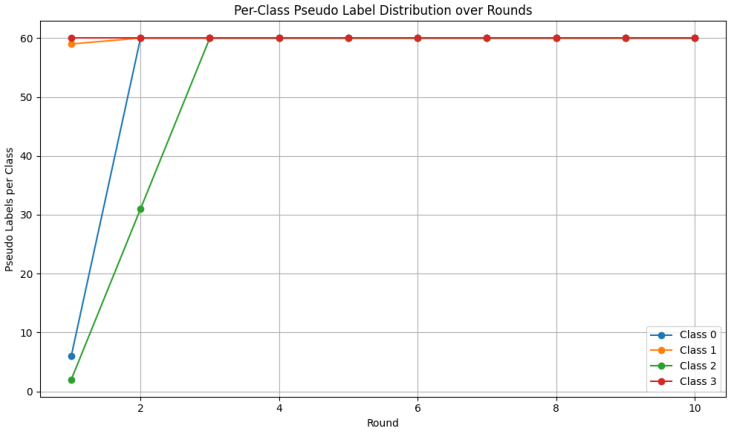


Figure 4.9: Class Distribution with Pseudo Labels

In the early training stages (rounds 1–3), the pseudo labels are highly imbalanced across classes. For instance, Class 3 quickly dominates with over 200 accepted pseudo labels as early as round 1, while Classes 0, 1, and particularly Class 2 receive very few. This suggests that the model initially exhibits high confidence for specific classes, while struggling to distinguish samples from others due to insufficient decision boundaries.

As training progresses, particularly between rounds 2 to 3, the model begins to confidently assign pseudo labels to more classes. This rapid expansion indicates that the decision regions become more refined as more high-confidence samples are incorporated into training. By round 3, all classes reach or approach the upper limit of accepted pseudo labels (200), marking a turning point in the self-training cycle where class coverage is maximized.

The confidence histogram illustrates the distribution of softmax probabilities across pseudo-labeled samples. A right-skewed distribution (i.e., high confidence) indicates that the model is confident in its predictions, which is crucial for ensuring pseudo-label quality in semi-supervised learning. Figure 4.10 illustrates the most Frequency of confidence is between 0.91 and 0.95, which is a good sign for semi-supervised learning.

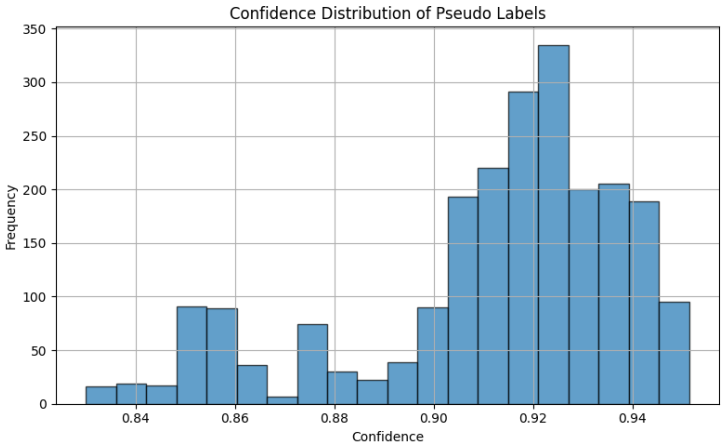


Figure 4.10: Confidence Histogram

We also analyzed the distribution of pseudo labels assigned in each iteration. An uneven distribution may indicate bias in the model or lower confidence in certain categories when starts self- training. These insights guide threshold adjustments and rebalancing strategies during self-training.

We observed the change in the number of accepted pseudo-labels across self-training rounds. As shown in Figure 4.11, the first two rounds saw a consistent increase in the number of high-confidence pseudo-labeled samples, indicating that the model was successfully learning from easy-to-classify examples. Starting from the third round, the growth plateaued, suggesting that the model had saturated its confident coverage.

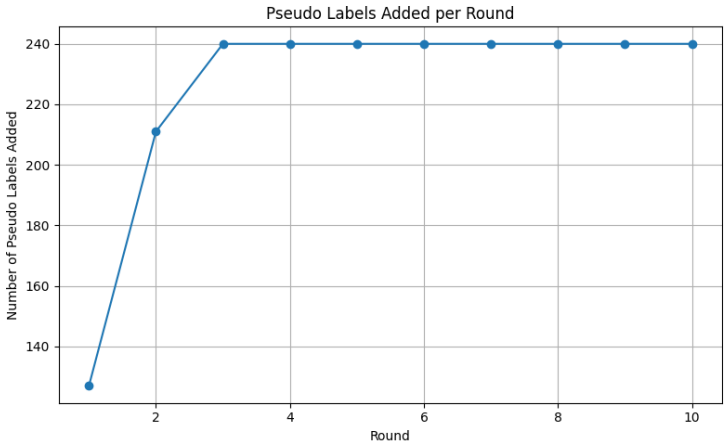


Figure 4.11: Pseudo label Counts with 0.84 threshold

4.6 Testing in new dataset

We test the model with the new dataset which is collected between **May 18th, 2025** and **June 17th, 2025**, in total 97484 queries. After data pro-processing, we can get 39919 queries to process for further classifying. In addition, we can see that there are about 53,165 'Fix errors' triggered by buttons in the editor and about 431 prompts from the GenAI assistant Hints button.

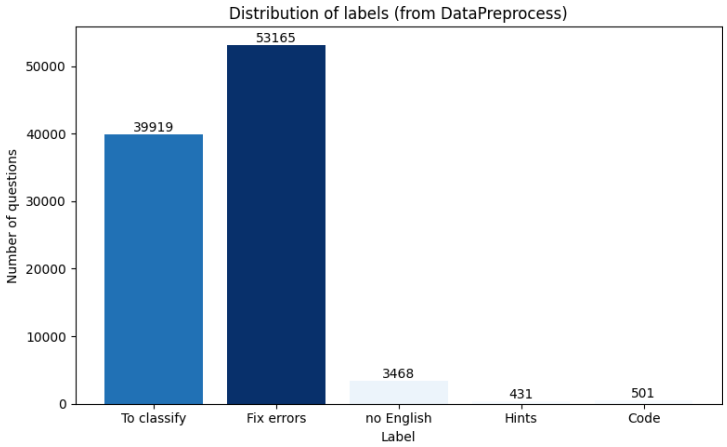


Figure 4.12: Label distribution in test dataset

In 'toClassify' dataset, we use our model to classify and predict the intent of user queries. We can see that 'Fix error' only accounts for a small portion of the data, because most of the 'Fix errors' are triggered directly by buttons after compilation. The main intents here are 'Explain' and 'Suggest', which reflects that the main needs of users when using the GenAI assistant are to consult and understand some academic functions of Arduino and get some usage suggestions.

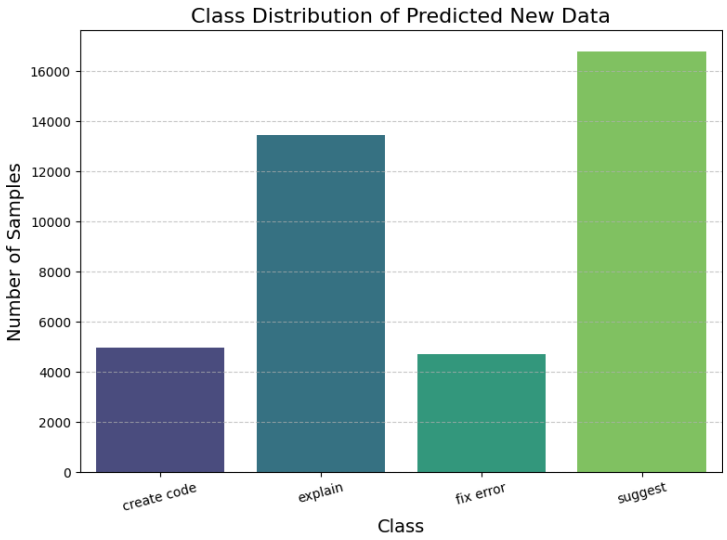


Figure 4.13: Prediction in test dataset

4.7 Lookerstudio Dashboard

For better showing the data, we also desgin the dashboard via Lookerstudio. In Lookerstudio report, the graphs link the data directly from Bigquery datasets. Total questions amount in report is same as the testing dataset in figure 4.12. We can observe from the figure 4.14 that the number of 'Fix errors' data and 'Hints' data is comparable to the number obtained after our data preprocessing. The small differences due to the difference between data collecting time and report query time. In the future, we can also add the intent prediction of the model to LookerStudio to obtain data analysis charts of user purposes.

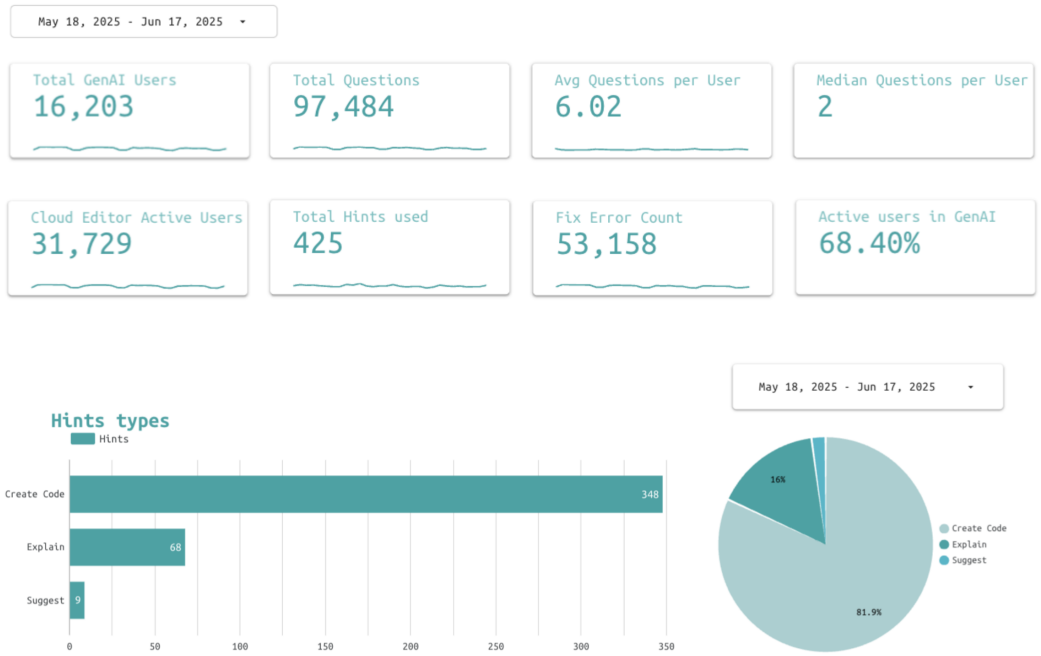


Figure 4.14: GenAI general information in Lookerstudio

Chapter 5

Conclusion & Future work

5.1 Summary

This work presents a comprehensive semi-supervised text classification pipeline tailored for categorizing Arduino user queries into four actionable categories: `Create code`, `Explain`, `Suggest`, and `Fix errors`. The system integrates advanced preprocessing for noisy developer text, semantic embedding using pretrained transformers, and ensemble classifiers with confidence calibration.

Starting with noisy user queries, we designed a robust preprocessing pipeline that removes boilerplate code, compiler traces, and templates, and retains only valid natural-language questions. To enhance the semantic representation of these short and often informal queries, we evaluated several Sentence Transformers, including `MiniLM`, `intfloat/e5-base-v2`, and `nomic-ai/nomic-embed-text-v1`. The best empirical performance was obtained with `intfloat/e5-base-v2`, which balances compactness, semantic fidelity and lightweight.

In the project, we extracted dense sentence embeddings and experimented with multiple classifiers: Random Forest, Logistic Regression, XGBoost, LightGBM, and CatBoost. We found that the ensemble method combined with probabilistic calibration (`CalibratedClassifierCV`) significantly improved classification consistency and confidence estimates.

A key challenge is the imbalance and noise introduced during pseudo-label selection. To mitigate this, we proposed a **class-wise adaptive thresholding strategy** where the threshold $\theta_k^{(t)}$ for each class k in round t is dynamically adjusted based on the number of pseudo-labeled samples from the previous round. A slight decay per round allows the threshold to gradually loosen, accommodating better generalization as the model becomes more confident.

5.2 Key Contributions

- Design a concise preprocessing pipeline to effectively remove noise from technical expertise and code-intensive user input.
- Evaluate and select an optimal Sentence Transformer model for semantic

encoding of user queries.

- Build an ensemble-based, calibrated classifier that performs well under label sparsity.
- Propose a class-wise dynamic thresholding scheme for self-training, improving recall on underrepresented classes.
- Empirically validate multiple pseudo-labeling strategies and visualize their influence on class distributions and overall performance.

Experimental results show that our dynamic thresholding strategy improves recall and F1-score for minority classes without harming accuracy, demonstrating its practical effectiveness in real-world coding assistant scenarios. In addition, optimizations for sketch loading are also helpful, identifying repair requests more carefully but accurately, which helps improve the quality of sketch modifications. The model identifies repair errors only when it is more certain, reducing the risk of importing other types of problem judgment errors into unnecessary sketches.

5.3 Limitations and Challenges

As the dataset is unlabeled competely, fully supervised learning is impractical due to the large manual annotation cost and the difficulty of standardizing labeling criteria in such subjective, technical domains. On the other hand, unsupervised clustering struggles to yield meaningful partitions in short, domain-specific, and noisy inputs. Therefore, we adopt a semi-supervised learning framework.

However, semi-supervised learning also introduces its own limitations:

First, its effectiveness diminishes in later rounds due to the accumulation of low-confidence pseudo-labels, which may introduce noise and harm generalization. Thus, careful control of training rounds and per-class sample caps is necessary. Our method relies on empirical tuning of base thresholds and decay rates, which may not generalize well across datasets or tasks without additional validation.

Secondly, and more fundamentally, our dataset presents unique challenges: the user-submitted queries are highly diverse and subjective in nature, often combining natural language with noisy, fragmented code snippets. Compared to large-scale text classification benchmarks, this type of input reduces the effectiveness of standard language models, including `intfloat/e5-base-v2`, in fully capturing semantic intent and fine-grained class boundaries. Despite efforts to define clear labeling criteria during manual annotation process, certain categories — particularly `explain` and `suggest` — exhibit ambiguous boundaries. Label confusion can still occur even with manual labeling. Furthermore, user questions sometimes contain multiple intents in a single query, which makes assigning a single class label inherently difficult. This introduces noise into the labeled dataset and may affect the model's ability to distinguish subtle intent differences, especially for complex or multiple queries.

In general, semi-supervised learning is sensitive to label noise and data distribution mismatch. The effectiveness of model decreases when labeled data is limited or biased, and when unlabeled data exhibits imbalanced or uneven distribution characteristics. These factors together limit the model's ability to achieve optimal and generalizable performance on complex queries.

5.4 Future Work

While the current semi-supervised classification framework based on an ensemble VotingClassifier (LightGBM, CatBoost, Logistic Regression) provides stable and competitive performance on small-scale, partially labeled data, it also introduces limitations in modeling complex user intent and interactions. However, feed-forward neural networks (FF-NNs) have been shown in many experiments that neural architectures can capture more subtle patterns when there is enough labeled data. Due to the current dataset size, imbalance between classes, and limited labeled samples, FF-NN models require more resources and are prone to signs of overfitting, and fail to surpass ensemble methods in overall generalization ability.

As the project progresses and more labeled and unlabeled user query data becomes available (e.g., scaling up to 10k+ examples), future improvements will include revisiting deep learning-based classifiers such as FF-NNs, Transformer-based classifiers with fine-tuning, and potentially more advanced semi-supervised learning frameworks (e.g., self-training with teacher-student models or contrastive learning). These approaches an better utilize richer embedding representations and improve intent classification, especially for complex or multi-intent user queries. Furthermore, the end-to-end neural model helps reduce the reliance on manual pre-processing and further enhances the retrieval augmentation generation (RAG) response optimization in the GenAI system.

Future directions include:

- **Uncertainty-aware pseudo-labeling:** Integrating model uncertainty (e.g., via entropy or Bayesian dropout) can reduce noisy pseudo-labels in early self-training rounds.
- **Curriculum learning for self-training:** Ordering unlabeled samples based on confidence or embedding distance can make training more robust and progressive.
- **Scalable weak supervision:** Leveraging labeling functions or prompt-based distant supervision could enrich initial labeled data without human effort.
- **Integration with GenAI code assistants:** Extending the classifier to interact with sketch context or generate code suggestions via RAG pipelines could significantly enhance system utility.

Appendix A

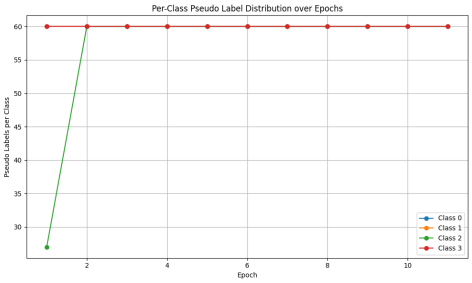
Appendix A

A.1 Visualization under Different Initial Thresholds

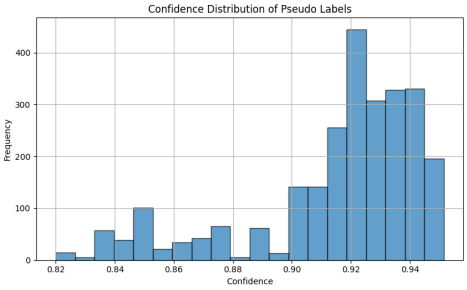
This appendix presents the distribution and performance metrics under different initial thresholds used in the self-training framework. For each threshold, we include the following plots:

- Class distribution of selected pseudo-labeled samples.
- Confidence score histogram.
- Per-class pseudo-label count.
- Validation metrics over training rounds.

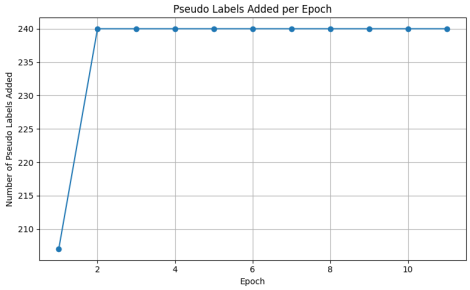
A.1.1 Initial Threshold = 0.82



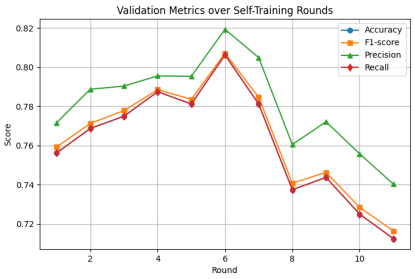
(a) Class distribution



(b) Confidence histogram



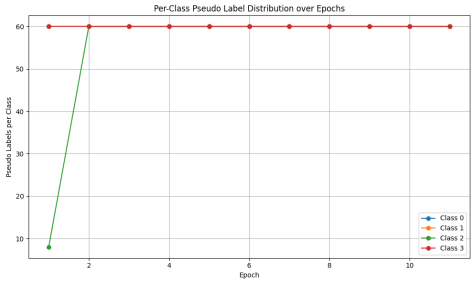
(c) Pseudo label counts



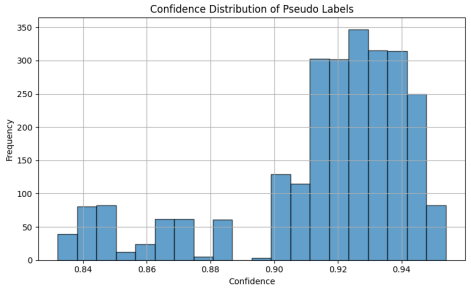
(d) Validation metrics

Figure A.1: Visualization under threshold = 0.82: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

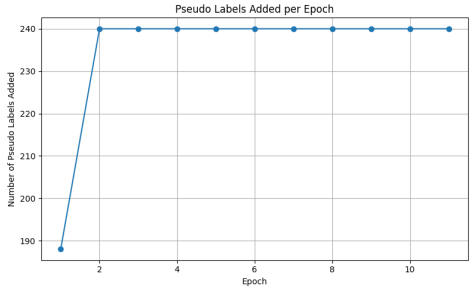
A.1.2 Initial Threshold = 0.83



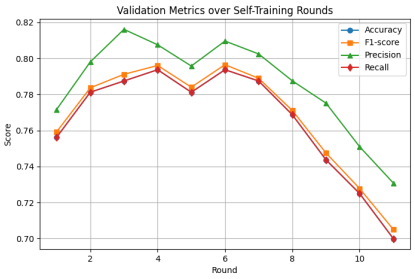
(a) Class distribution



(b) Confidence histogram



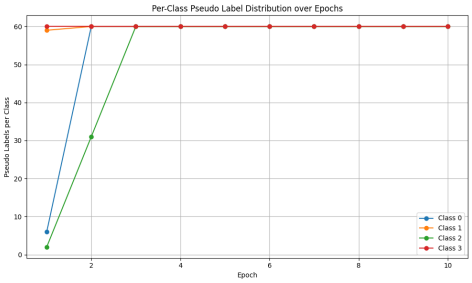
(c) Pseudo label counts



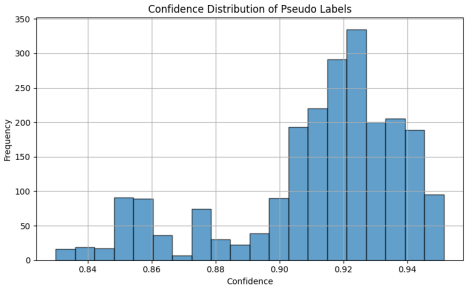
(d) Validation metrics

Figure A.2: Visualization under threshold = 0.83: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

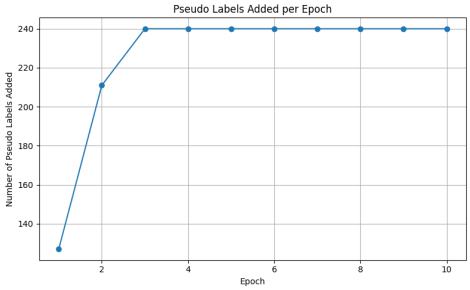
A.1.3 Initial Threshold = 0.84



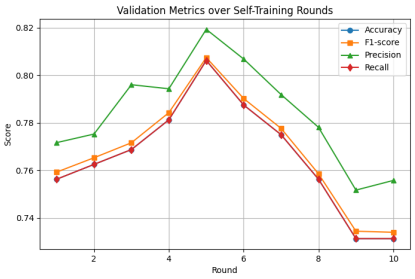
(a) Class distribution



(b) Confidence histogram



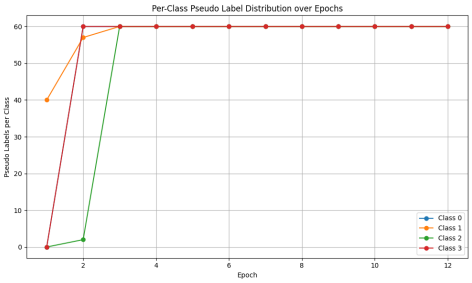
(c) Pseudo label counts



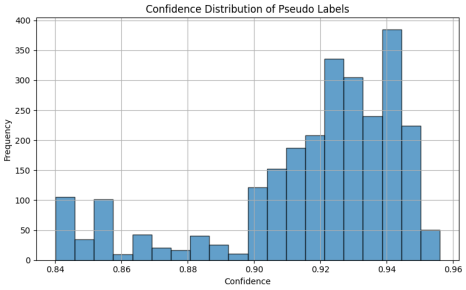
(d) Validation metrics

Figure A.3: Visualization under threshold = 0.84: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

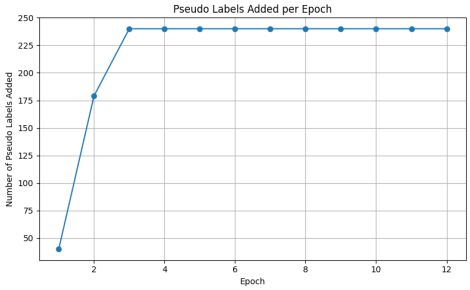
A.1.4 Initial Threshold = 0.85



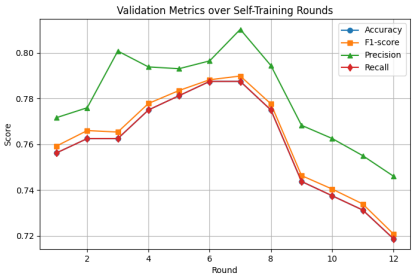
(a) Class distribution



(b) Confidence histogram



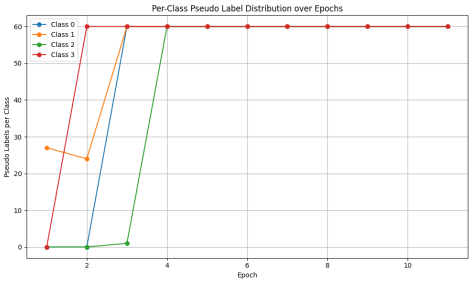
(c) Pseudo label counts



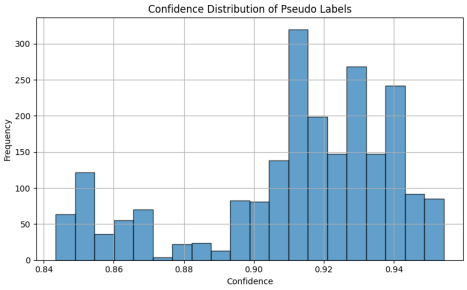
(d) Validation metrics

Figure A.4: Visualization under threshold = 0.85: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

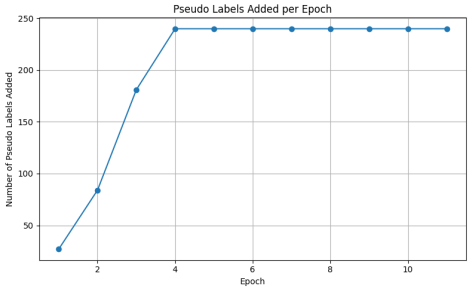
A.1.5 Initial Threshold = 0.86



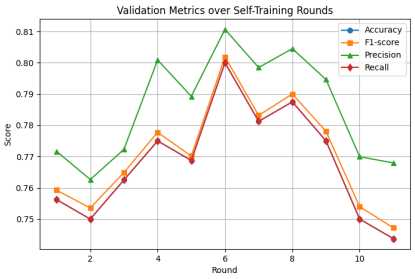
(a) Class distribution



(b) Confidence histogram



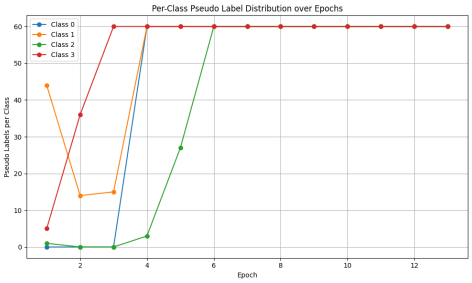
(c) Pseudo label counts



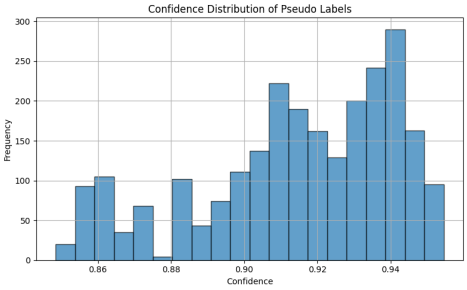
(d) Validation metrics

Figure A.5: Visualization under threshold = 0.86: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

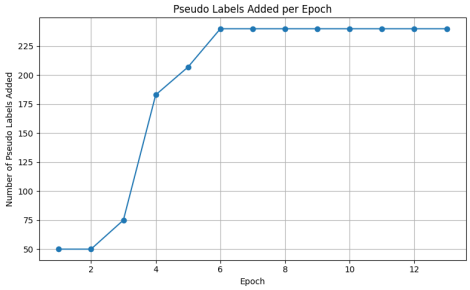
A.1.6 Initial Threshold = 0.88



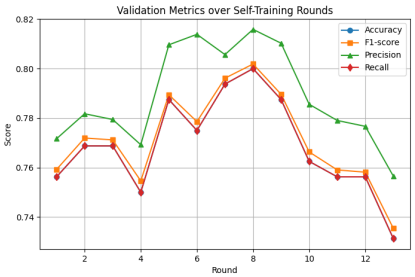
(a) Class distribution



(b) Confidence histogram



(c) Pseudo label counts



(d) Validation metrics

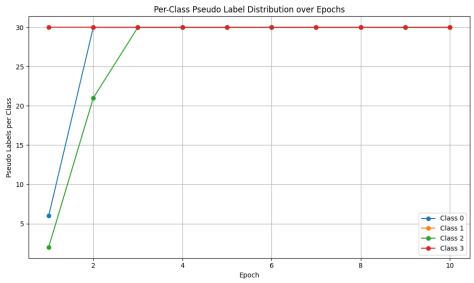
Figure A.6: Visualization under threshold = 0.88: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

A.2 Visualization under Different max_pseuodo

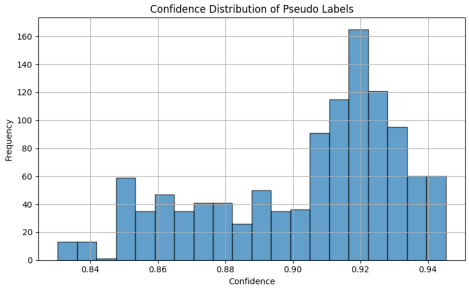
This appendix presents the distribution and performance metrics under different max_pseuodo setting used in the self-training framework. For differernt max_pseuodo and initail threshold = 0.84, we include the following plots:

- Class distribution of selected pseudo-labeled samples.
- Confidence score histogram.
- Per-class pseudo-label count.
- Validation metrics over training rounds.

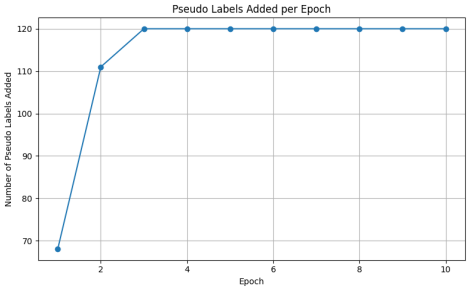
A.2.1 max_pseuodo = 30



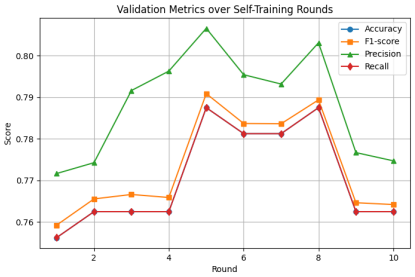
(a) Class distribution



(b) Confidence histogram



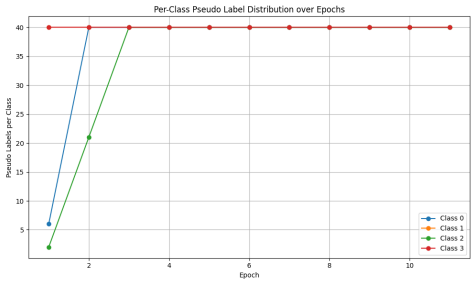
(c) Pseudo label counts



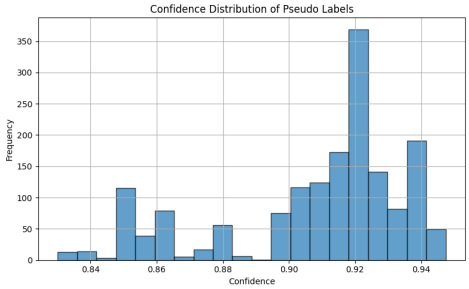
(d) Validation metrics

Figure A.7: Visualization with max_pseuodo = 30: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

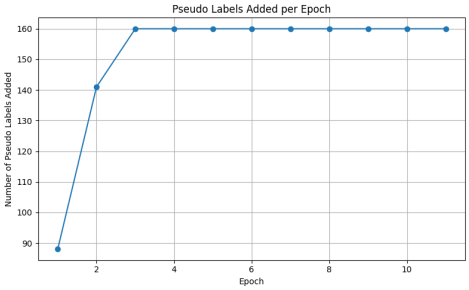
A.2.2 max_pseuodo = 40



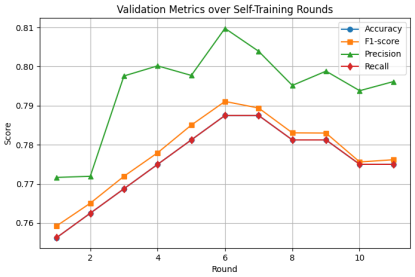
(a) Class distribution



(b) Confidence histogram



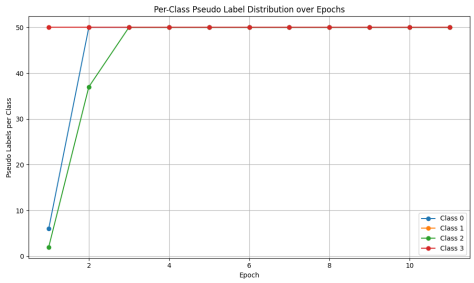
(c) Pseudo label counts



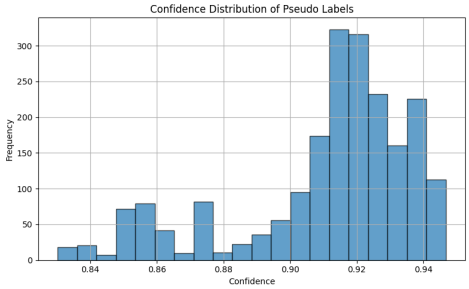
(d) Validation metrics

Figure A.8: Visualization with max_pseuodo = 40: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

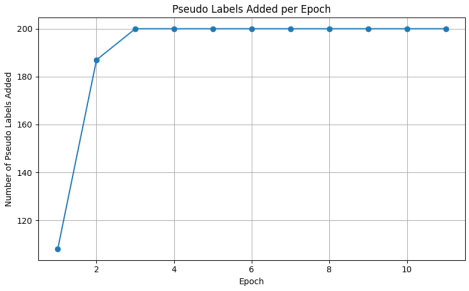
A.2.3 max_pseuodo = 50



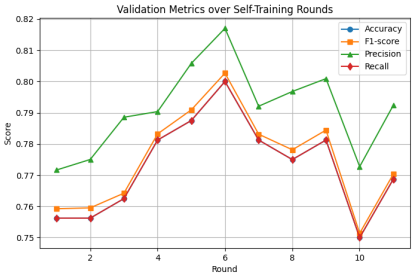
(a) Class distribution



(b) Confidence histogram



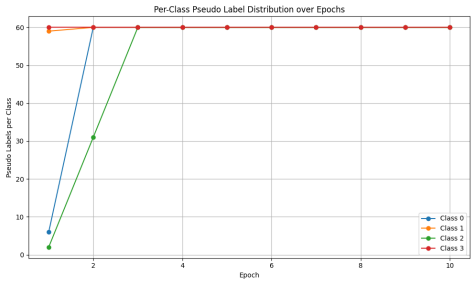
(c) Pseudo label counts



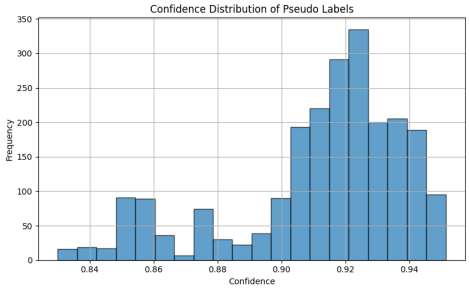
(d) Validation metrics

Figure A.9: Visualization with max_pseuodo = 50: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

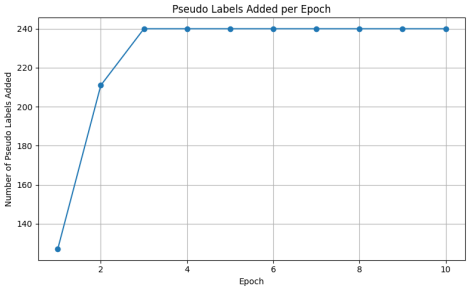
A.2.4 max_pseuodo = 60



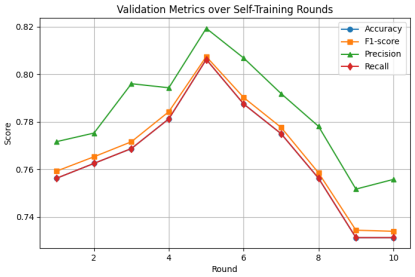
(a) Class distribution



(b) Confidence histogram



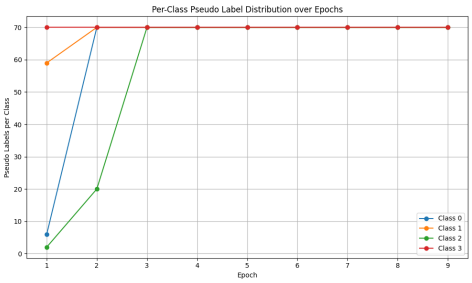
(c) Pseudo label counts



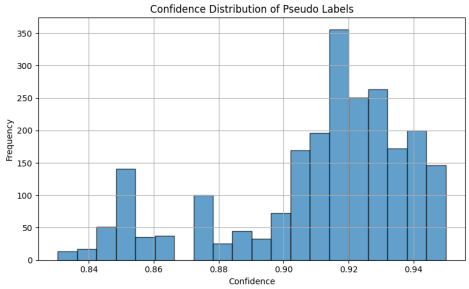
(d) Validation metrics

Figure A.10: Visualization with max_pseuodo = 60: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

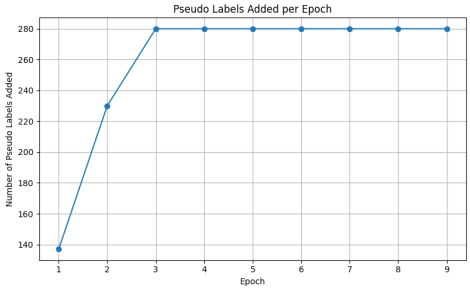
A.2.5 max_pseuodo = 70



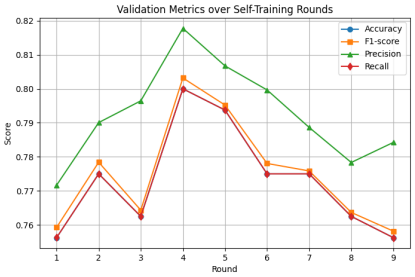
(a) Class distribution



(b) Confidence histogram



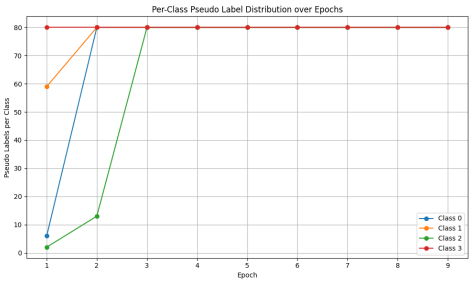
(c) Pseudo label counts



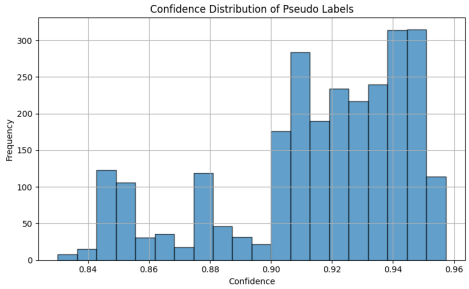
(d) Validation metrics

Figure A.11: Visualization with max_pseuodo = 70: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

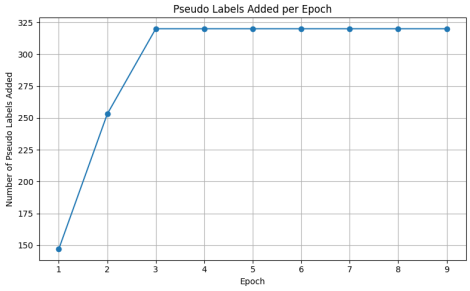
A.2.6 max_pseuodo = 80



(a) Class distribution



(b) Confidence histogram



(c) Pseudo label counts

(d) Validation metrics

Figure A.12: Visualization with max_pseuodo = 80: class distribution, confidence histogram, pseudo-label counts, and validation metrics.

Bibliography

[1] Martin R. Chavez, Thomas S. Butler, Patricia Rekawek, Hye Heo, and Wendy L. Kinzler. "Chat Generative Pre-trained Transformer: why we should embrace this technology". In: *American Journal of Obstetrics and Gynecology* 228.6 (2023), pp. 706–711. ISSN: 0002-9378 (cit. on p. 1).

[2] Ajay Agrawal, Joshua Gans, and Avi Goldfarb. "ChatGPT and how AI disrupts industries". In: *Harvard Business Review* 12 (2022), pp. 1–6 (cit. on p. 1).

[3] Dickey E., Bejarano A., and Garg C. "AI-Lab: A Framework for Introducing Generative Artificial Intelligence Tools in Computer Programming Courses." In: *SN COMPUT. SCI.* 5 (2024), p. 720 (cit. on p. 1).

[4] Cai (Mitsu) Feng, Elsamari Botha, and Leyland Pitt. "From HAL to GenAI: Optimizing chatbot impacts with CARE". In: *Business Horizons* 67.5 (2024). SPECIAL ISSUE: WRITTEN BY CHATGPT, pp. 537–548. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2024.04.012. URL: https://www.sciencedirect.com/science/article/pii/S0007681324000570 (cit. on p. 2).

[5] Sean Sands, Carla Ferraro, Colin Campbell, and Hsiu-Yuan Tsao. "Managing the human–chatbot divide: how service scripts influence service experience". In: *Journal of Service Management* 32.2 (2021), pp. 246–264 (cit. on p. 2).

[6] Yu-Shan (Sandy) Huang and Paula Dootson. "Chatbots and service failure: When does it lead to customer aggression". In: *Journal of Retailing and Consumer Services* 68 (2022), p. 103044. ISSN: 0969-6989. DOI: https://doi.org/10.1016/j.jretconser.2022.103044. URL: https://www.sciencedirect.com/science/article/pii/S0969698922001370 (cit. on p. 3).

[7] Fiona Fui-Hoon Nah, Ruilin Zheng, Jingyuan Cai, Keng Siau, and Langtao Chen. "Generative AI and ChatGPT: Applications, challenges, and AI-human collaboration". In: *Journal of Information Technology Case and Application Research* 25.3 (2023), pp. 277–304. DOI: 10.1080/15228053.2023.2233814 (cit. on p. 3).

[8] Abdelrahman H. Hefny, Georgios A. Dafoulas, and Manal A. Ismail. "Intent Classification for a Management Conversational Assistant". In: *2020 15th International Conference on Computer Engineering and Systems (ICCES)*. 2020, pp. 1–6. DOI: 10.1109/ICCES51560.2020.9334685 (cit. on p. 5).

[9] Barry J. Zimmerman and Manuel Martinez Pons. "Development of a Structured Interview for Assessing Student Use of Self-Regulated Learning Strategies". In: *American Educational Research Journal* 23.4 (1986), pp. 614–628. DOI: 10.3102/00028312023004614 (cit. on p. 5).

[10] Nick Bradley, Thomas Fritz, and Reid Holmes. "Context-Aware Conversational Developer Assistants". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 993–1003. DOI: 10.1145/3180155.3180238 (cit. on p. 5).

[11] A.Abdellatif, Ahmad Abdellatif, Khaled Badran, and Emad Shihab. "MSRBot: Using bots to answer questions from software repositories". In: *Empirical software engineering.* 25.3 (May 2020). ISSN: 1382-3256 (cit. on p. 5).

[12] Jhonny Cerezo, Juraj Kubelka, Romain Robbes, and Alexandre Bergel. "Building an Expert Recommender Chatbot". In: *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. 2019, pp. 59–63. DOI: 10.1109/BotSE.2019.00022 (cit. on p. 5).

[13] Elahe Paikari et al. "A Chatbot for Conflict Detection and Resolution". In: *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. 2019, pp. 29–33. DOI: 10.1109/BotSE.2019.00016 (cit. on p. 5).

[14] Stefan Larson et al. "An Evaluation Dataset for Intent Classification and Out-of-Scope Prediction". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 1311–1316. DOI: 10.18653/v1/D19-1131. URL: https://aclanthology.org/D19-1131/ (cit. on p. 6).

[15] Alaa T. Al-Tuama and Dhamyaa A. Nasrawi. "Intent Classification Using Machine Learning Algorithms and Augmented Data". In: *2022 International Conference on Data Science and Intelligent Computing (ICDSIC)*. 2022, pp. 234–239. DOI: 10.1109/ICDSIC56987.2022.10075794 (cit. on p. 6).

[16] Libo Qin, Tailu Liu, Wanxiang Che, Bingbing Kang, Sendong Zhao, and Ting Liu. "A Co-Interactive Transformer for Joint Slot Filling and Intent Detection". In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 8193–8197. DOI: 10.1109/ICASSP39728.2021.9414110 (cit. on p. 6).

[17] Yilin Shen, Yen-Chang Hsu, Avik Ray, and Hongxia Jin. "Enhancing the generalization for Intent Classification and Out-of-Domain Detection in SLU". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli. Online: Association for Computational Linguistics,

Aug. 2021, pp. 2443–2453. DOI: 10.18653/v1/2021.acl-long.190. URL: https://aclanthology.org/2021.acl-long.190/ (cit. on p. 6).

[18] Ting-En Lin, Hua Xu, and Hanlei Zhang. "Discovering New Intents via Constrained Deep Adaptive Clustering with Cluster Refinement". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05 (Apr. 2020), pp. 8360–8367. DOI: 10.1609/aaai.v34i05.6353. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6353 (cit. on p. 6).

[19] Wulian Yun, Mengshi Qi, Fei Peng, and Huadong Ma. *Semi-Supervised Teacher-Reference-Student Architecture for Action Quality Assessment*. 2025. arXiv: 2407.19675 [cs.CV]. URL: https://arxiv.org/abs/2407.19675 (cit. on p. 18).