Practical Use Case of Machine Learning System Design

1 - Problem Statement

Our goal is to create a system that recommends coding problems to solve and explains its reasoning to the user. We'll use Large Language Models (LLM) embeddings and a standard recommendation system to make these recommendations. Additionally, we'll utilize the generative capabilities of LLMs to provide text snippets that explain the reasoning behind the recommendations. The system provides value by helping the end-users of the platform use their practice time more efficiently. This feature nicely fits into the design of the popular web-based coding practice platforms such as leetcode.com, codeforces.com, hackerrank.com, and others. Providing additional feedback to users makes it more likely that they will indeed proceed to one of the recommended questions, thus improving user engagement for the platform.

2 - Data collection and processing

Our system requires a training set that consists of problem descriptions. The training set may also include example solutions. Our system will be text-based, where text includes the technical description of a problem or the solution source code. There are publicly available problem datasets on Kaggle, e.g., the 1825 Leetcode problem dataset. The leading web-based platforms for competitive programming and interview practice typically include web-based APIs for data retrieval, which can help us get more data.

The training data will be preprocessed and stored in CSV files with a stereotypical structure described above. The real-time input data for making a recommendation will be prepared by the frontend in a format that is accepted by our backend endpoint. This design closely follows standard practices in web development. We will try to use text fields and tags instead of relational database tabular structures whenever possible. In other words, avoid specifying a rigid schema and strong typing of fields.

3 - Feature Engineering

For effective feature engineering in our code-recommendation and feedback system, we should consider the following factors:

1. Include relevant features like tags, difficulty level, average attempts per user, and associated companies. Use feature methods (e.g. PCA or T-SNE) to select the set of important features.
2. Clean and preprocess each question, while preserving significant symbols and adhering to language-specific rules.
3. Address any missing values in user features and consider downsampling large datasets.
4. Use an encoder model to vectorize text and code information, considering metrics like difficulty level, tags, time spent on the problem, user history, associated companies, and desired role.

4 - Model Selection and Evaluation

For the model selection and evaluation of our code-recommendation and feedback system, we should consider several factors. One option is to train a large language model (LLM) on the vectorized representation of our dataset. There are open-source LLMs available, such as those provided by

HuggingFace, which can be fine-tuned to meet our specific needs. LLMs are advantageous as they can process large amounts of text and retrieve relevant data based on prompts.

To understand gaps in the code and provide feedback, we can utilize a large language model like Alpaca, Red Pajama, Llama, or GPT-3. However, it is important to fine-tune these models specifically for coding questions. Additionally, for the recommendation system, we can leverage indexing algorithms such as vectorDB or FAISS, which facilitate semantic search. Text encoding can be handled using pre-trained sentence transformer models like Roberta, although fine-tuning may be necessary due to the nature of code bases.

Incorporating the ChatGPT API or pre-trained embeddings can also be taken into consideration. Additionally, we should include metrics such as time spent on each problem and the need for assistance. These metrics will allow us to assess the accuracy of our recommendations and provide valuable feedback, enabling continuous model improvement.

5 - Training and Optimization

Following points should be considered while training LLMs and recommender models.

Model Architecture: Selecting an appropriate model architecture that suits our specific use case is essential. This could involve leveraging transformer-based architectures like BERT, GPT, or even encoder-decoder models, depending on the requirements of code recommendation and feedback generation.

Fine-tuning: Fine-tuning the chosen model on our curated dataset is necessary to adapt it to our specific needs. This process involves exposing the model to the dataset and updating its weights to optimize its performance on code-related tasks.

Hyperparameter Tuning: Experimenting with different hyperparameters such as learning rate, batch size, and regularization techniques can significantly impact model performance. Conducting systematic hyperparameter tuning through methods like grid search or random search can help identify the optimal configuration.

Incremental Learning: Considering the dynamic nature of coding practices, implementing techniques for incremental learning can enable the model to adapt and continuously improve based on real-time user interactions and feedback. Maybe we can schedule automatic training on a weekly basis to finetune the model based on feedback collected.

Hardware and Parallelization: Utilizing hardware acceleration techniques like GPU or TPU can significantly speed up the training process. Parallelizing the training across multiple devices can also be explored to further enhance efficiency.

6 - Deployment and monitoring

Infrastructure: Proper infrastructure must be in place to deploy the model. This means that one must have access to CPUs, GPUs, or TPUs to handle the processing needs of the service.

API Development: An API allows the user to interact with the model. It also avoids any unnecessary installations by the user, since the service can typically be served using a web framework to communicate with the model.

Model serving: Determining the method for how the model will be used by the applications.

Model versioning and rollback practices: Implementing versioning mechanisms to keep track of all the iterations of the model. This practice allows for a proper process for rolling back the service to a previous model version when needed due to changes in model performance, data drift, etc.

Security: Making sure that the service has the necessary security processes in place. These include user authentication, policies around data privacy, authorization mechanisms to access the API, compliance requirements for storing user data and information, etc.

Monitoring and logging: Monitoring the model's performance over time, as well as monitoring the system resource usage, error rates, response times, etc. Similarly, the service can include a continuous learning pipeline that keeps adding user data and fine-tunes the model periodically.

7- Ethical considerations

Our interview prep machine learning system, akin to LeetCode, acknowledges the potential for bias favoring candidates familiar with certain problem types, or arising from subtly embedded aspects like variable naming patterns or code structures reflecting a user's language, culture, and background. To mitigate this, we constantly strive to minimize bias and enhance fairness by considering the diversity of practical software development requirements and by monitoring users' engagement with different question types. Ensuring sensitive information is handled responsibly, we enhance user privacy through anonymization or grouping. Upholding transparency and accountability, we openly share our recommendation processes and algorithms, actively seek user feedback for continual improvement, and remain committed to addressing any residual bias to ensure a fair and unbiased system.

8 - Reflection and future improvements

The next question recommender system will be designed to cater to individual user's skill levels, providing a tailored experience that fosters growth. We envision this powerful tool will be guiding users on their journey towards coding proficiency. The recommender will curate a list of selected questions with feedback text based on the user's interest and proficiency level. These questions will serve as stepping stones for users to expand their coding knowledge and capabilities. The list will encompass a range of questions containing advanced topics to encourage exploration and mastery. The system will also identify areas where users may benefit from additional practice and improvement. These targeted questions will address specific knowledge gaps or areas of weakness, offering opportunities for focused learning and skill refinement. We are excited to explore room for the following improvement: *providing diverse recommendations*, *mitigation of popularity bias*, *improvement on real-time responsiveness*, and *inclusion of CI/CD/CT pipelines*.