Personalized News Headline Generation: PENS Baseline Method Improvement and Implementation of Prompt Engineering Approach

Luofan Wang, Shouhe Zhu

June 23, 2025

1 Overview of Completion Status

Our group selected the "Personalized Headline Generation Task" topic and conducted collaborative development through a GitHub repository. We have completed two alternative technical approaches, including PENS baseline method reproduction and improvement, and a prompt engineering-based large model personalized generation method. We now have completed runnable code with clear README documentation and an innovative evaluation scheme designed by ourselves.

2 PENS Baseline Method Reproduction and Improvement

2.1 Environment Setup

The local environment has been configured with CUDA 11.8 to meet the GPU acceleration requirements for deep learning tasks. Considering that the project involves multiple deep learning libraries (including PyTorch, Transformers, NumPy, etc.), to ensure stability and reproducibility of dependency management, we adopted Conda as the environment management tool and built an independent virtual environment based on the requirements.txt file provided by the baseline project.

2.2 Baseline Method Analysis

2.2.1 Overall Architecture

The project follows a classic three-stage pipeline architecture: Data Preprocessing (Preprocess) → User Modeling (UserEncoder) → Personalized Generation (Generator). Initially, raw data is stored in the 'data/' directory. First, the 'Preprocess' component cleans, transforms, and structures the raw data, outputting to the 'data2/' directory, which serves as the direct data source for all subsequent model training. Then the 'UserEncoder' and 'Generator' components respectively perform model training and save the trained models (i.e., component outputs) in corresponding subdirectories under the 'runs/' directory. In this process, downstream components explicitly depend on upstream component outputs: 'UserEncoder' depends on 'data2/' data, while 'Generator' depends on both 'data2/' data and the model trained by 'UserEncoder'.

2.2.2 Core Component Analysis

Data Preprocessing (Preprocess): In the PENS personalized news headline generation project, the data preprocessing (Preprocess) component is the starting point of the entire workflow, with core logic concentrated in pensmodule/Preprocess/preprocess.ipynb. The main responsibility of this component is to convert raw, unstructured news text and user behavior logs into standardized numerical formats acceptable to deep learning models. Internally, it consists of multiple functional modules that, while not independent Python scripts, form a

complete pipeline through sequential execution of Jupyter Notebook code cells. First, the environment configuration module imports necessary libraries (such as pandas, numpy, nltk, torch) and defines global hyperparameters (such as MAX_CONTENT_LEN, WORD_FREQ_THRESHOLD, etc.), establishing unified standards for the data processing pipeline. Next, the news parsing module (read_news) reads the raw news.tsv file, performs text cleaning, tokenization, word frequency statistics, and constructs core dictionary mappings like word_dict, category_dict, and news_index. Subsequently, these dictionaries and index information are serialized and saved to the /data2 directory (as shown in Figure 1), serving as the first data checkpoint in the project. To meet downstream model input format requirements, Preprocess generates different data representations for the user encoder (UserEncoder) and headline generator (Generator). The former generates fixed-length news headline, content, and category ID sequences saved as .npy files, while the latter constructs source-target pairs required by Seq2Seq models, including content input, headline input and output sequences. Additionally, the component builds word embedding matrices for the project by loading pre-trained GloVe vectors, and processes user click logs to generate user click history and training samples, including positive and negative sample pairs. Finally, these user data are also serialized and stored, becoming the foundation for subsequent model training. The entire Preprocess component ensures data processing continuity, structural clarity, and downstream reusability through a combination of memory variable passing and file system persistence, serving as a solid data foundation for achieving the project's personalized generation capabilities.

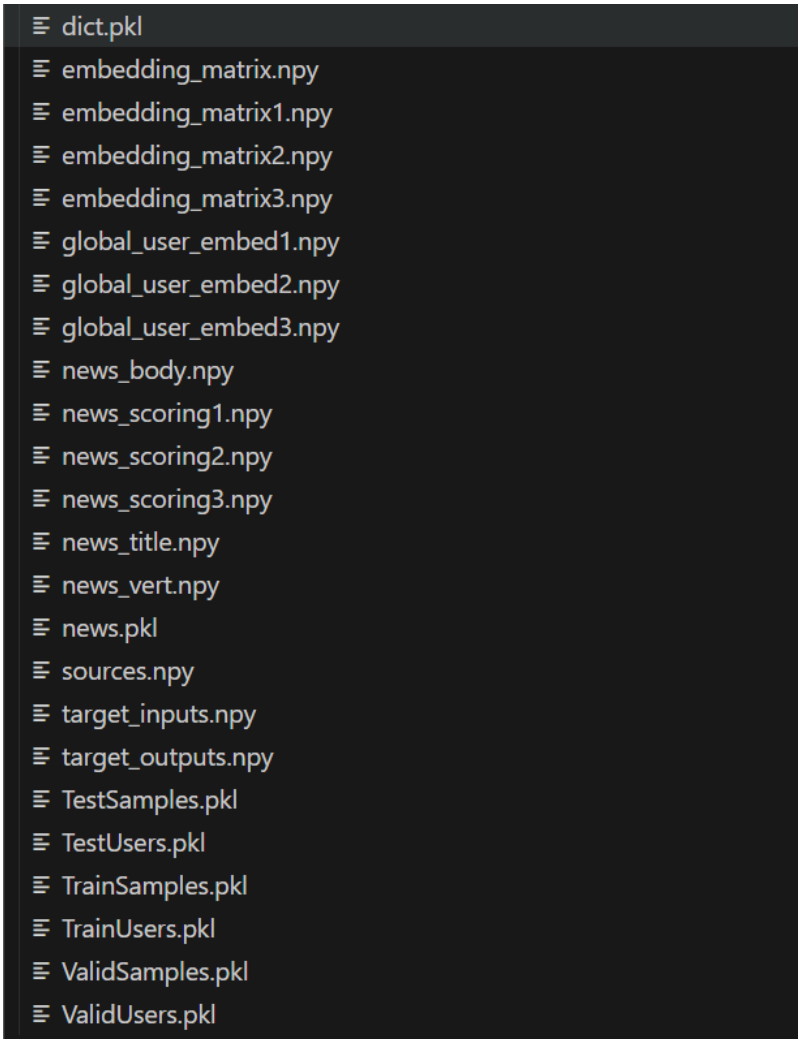

Figure 1: Preprocessed Data

User Encoder (UserEncoder): This component undertakes the core responsibility of "understanding users" with the goal of converting users' historical click behaviors into a fixed-dimensional user interest vector (User Vector) to guide downstream headline generation models. This component takes structured data generated by the Preprocess component as input and ultimately outputs a trained deep learning model. The logic is distributed across multiple Python files and coordinated by a Jupyter Notebook (pipeline_Train_Test.ipynb). The workflow first involves data.py handling data loading, constructing .npy and .pkl files into PyTorch-compatible Dataset and DataLoader, where each training sample includes candidate news, user click history, and their labels. Subsequently, modules.py defines general network structures such as MultiHeadAttention and AttentionPooling, providing building blocks for core models (like NRMS) in model.py. model.py defines specific structures for news encoders and user encoders, utilizing word embedding initialization and attention mechanisms to extract user interest representations. The auxiliary module utils.py provides accuracy evaluation functions during model training, while the overall training process is concentrated in pipeline_Train_Test.ipynb, including environment configuration, data loading, model initialization, iterative training, performance evaluation, and model saving. The entire UserEncoder component is highly modular with clear logic, and its training outputs are saved in the runs/userencoder/ directory (as shown in Figure 2), serving as important input for the downstream Generator component's personalized headline generation. This design, characterized by clear division of labor and low coupling, facilitates easy maintenance, debugging, and replacement of components.

Figure 2: Encoder Output

Personalized Generator (Generator): This component is the final execution unit of the entire system, with the main task of integrating structured data from Preprocess and user interest vectors provided by UserEncoder to generate personalized news headlines that both conform to news content and match user preferences. This component adopts the classic "pre-training-fine-tuning" two-stage strategy common in modern natural language generation tasks, consisting of multiple functional modules and unified scheduling through pipeline_pretrain_train_test.ipynb. First, the configuration file config.json provides global definitions of model structure and training hyperparameters; the data loading module data.py converts structured data into PyTorch Dataset suitable for training, where Seq2SeqDataset is used for general pre-training, and ImpressionDataset loads user click history and news vectors for personalized fine-tuning. The model's underlying components are distributed in encoder.py, decoder_pointer.py, and modules.py, respectively defining encoders, decoders with pointer mechanisms, and general attention layers, implementing effective representation of input news text and personalized control capabilities. The HeadlineGen class in model.py integrates the above modules into a complete trainable model, whose forward function implements the entire process of encoding news text and combining user interest vectors to decode and generate headlines. Training logic is supported by modules like train.py and eval.py, encapsulating optimizers, loss functions, fine-tuning logic, performance evaluation, and Beam Search generation strategies. The entire training and evaluation process unfolds orderly in the Jupyter Notebook: first using Seq2SeqDataset for pre-training to learn general headline generation capabilities; then loading the trained UserEncoder model for fine-tuning, injecting user interest vectors into the generation process to achieve true personalized generation, and saving the final model in the runs/ directory (as shown in Figure 3).

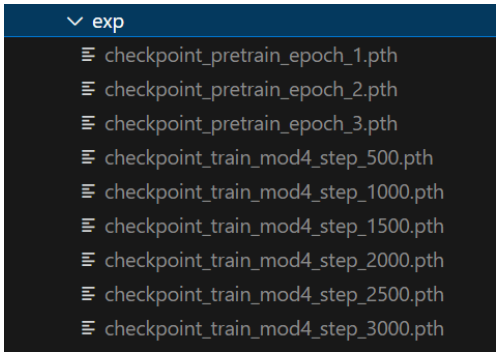

Figure 3: Trained Model Weights

2.3 Baseline Method Improvements

Solving Compatibility Issues in Original Code: When adapting project code to PyTorch 2.0 and above versions, we found compatibility issues with the original model loading approach. Specifically, the original code directly used torch.load(checkpoint_path) for model restoration when loading trained model checkpoints. However, starting from PyTorch 2.0, torch.load() added a new default parameter weights_only=True, meaning it only loads model weights by default without restoring the model object structure itself. Since the original code did not explicitly specify weights_only=False, this would cause checkpoint['model'] to return None or an empty dictionary during model loading, triggering errors when subsequently attempting to call model attributes, perform inference, or fine-tuning operations. Therefore, to ensure stable code execution and complete model restoration, we need to explicitly pass the weights_only=False parameter when loading checkpoints (as shown in Figure 4), ensuring both model structure and weights are loaded together. This fix is crucial for the normal operation of subsequent modules (such as fine-tuning and inference in the personalized generator).

Figure 4: Improved Model Loading Method

Training Method Optimization: During reproduction and execution of the original project training process (including UserEncoder training and Generator pre-training/fine-tuning), we observed severely suboptimal local GPU utilization. During training, GPU occupancy was below 10% most of the time, and power consumption was also near idle state, with only brief increases at certain intervals, but peak values were also low. This "GPU idle" phenomenon indicates that the training bottleneck is not in model computation but in the data transmission process. Analysis of the data loading process revealed that the original DataLoader used a single-threaded loading strategy without any memory acceleration mechanisms, causing CPU data preparation speed to be unable to match GPU computation speed, frequently blocking GPU waiting for input. To address this, we made two optimizations to the training data loading process (as shown in Figure 5): first, setting the DataLoader's num_workers parameter to a larger value supported by the system (such as 20), enabling multi-threaded data loading to improve data reading and preprocessing speed; second, enabling pin_memory=True to load data into pinned memory, accelerating CPU to GPU data transfer speed. This series of improvements significantly enhanced system throughput, maintaining GPU occupancy at 97%∼100% for extended periods,

improving overall training efficiency by approximately double, especially showing significant improvements in iteration speed during data preprocessing, model pre-training, and personalized fine-tuning stages.

```
test_iter = DataLoader(i_dset, batch_size=16, shuffle=False, num_workers=20, pin_memory=True)
```

Figure 5: Improved Data Loading Method

New Functional Modules: To enhance project usability and interactivity, we added a new functional module named predict_with_trained_model.ipynb for model inference applications and result output. This module allows users to directly load trained personalized headline generation models after training completion and perform headline generation tasks on specified input data (which can be original news content or user-defined text). Generation results are automatically saved in the generated_titles.txt file in the current directory for easy review and subsequent analysis. Additionally, if users provide corresponding "reference headlines" (i.e., manually written or existing real headlines), the system will automatically call built-in evaluation functions to perform ROUGE metric scoring on generation results, quantifying model generation quality. The addition of this module not only enhances project practicality but also provides convenience for actual deployment, user interaction, and performance evaluation.

2.4 Experimental Results

After completing full reproduction of the baseline method, we conducted systematic evaluation of model performance at different training stages, with relevant experimental results saved in the /docs0 directory. Main evaluation metrics include reward values obtained by the model during training (as shown in Figure 6) and ROUGE metric score changes on the validation set (as shown in Figure 7). Figure 6 shows the reward curve during training, where we can observe that the model's reward increases rapidly in early training stages and then gradually stabilizes; Figure 7 reflects the similarity trend between model generation results and reference headlines at different training steps.

From experimental results, if using ROUGE scores as the sole performance evaluation standard, we can clearly observe that the model reaches optimal performance around training step 2000, where ROUGE-1, ROUGE-2, and ROUGE-L scores all reach peak values under the current training process, indicating that headlines generated by the model at this stage are closest to manually created reference headlines in terms of content coverage, information completeness, and language structure.

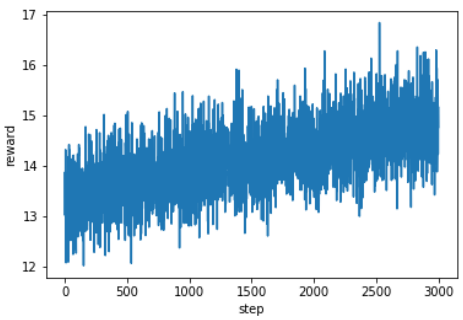

Figure 6: Training Reward vs. Training Steps

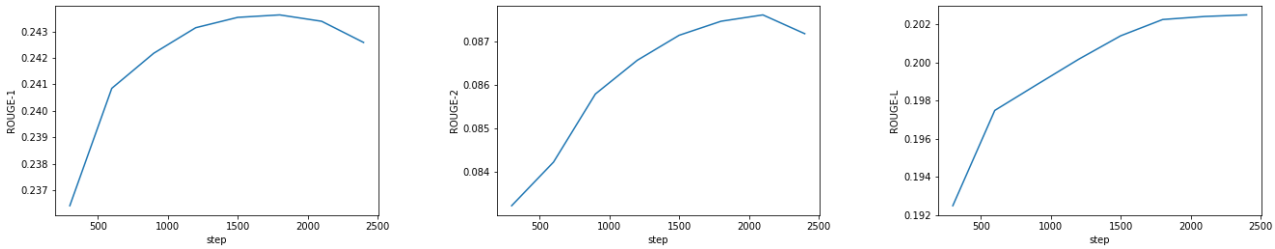

Figure 7: ROUGE Scores vs. Training Steps

Furthermore, to verify the model's generation capabilities in real application scenarios, this project called the trained personalized headline generation model to conduct experiments on a set of custom news sample data, generating news headlines with user preferences. During experiments, input data included custom news content, while associated user click history information reused original data. The model comprehensively judged user interests based on this information and generated headline text that better matched user reading preferences while ensuring semantic reasonableness.

Figure 8 shows manually created reference headlines for this set of custom news samples; Figure 9 shows personalized headline results generated by the model under the above input conditions. Through comparison, we find that generated headline 1 covers key news information while also showing certain user preference tendencies in expression and word choice, such as more attractive wording and word order that better matches user click tendencies. However, headlines 2 and 3 performed poorly, not only failing to use more attractive wording but even losing key information from original news headlines. This shows that the model's generalizability and practicality still need strengthening.

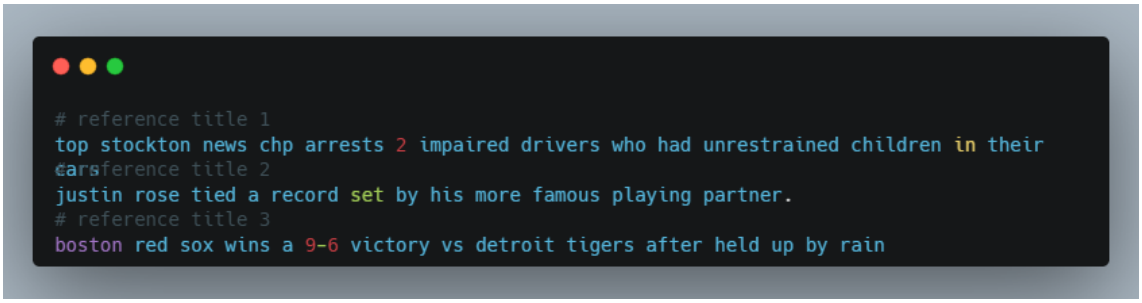

Figure 8: Reference Headlines

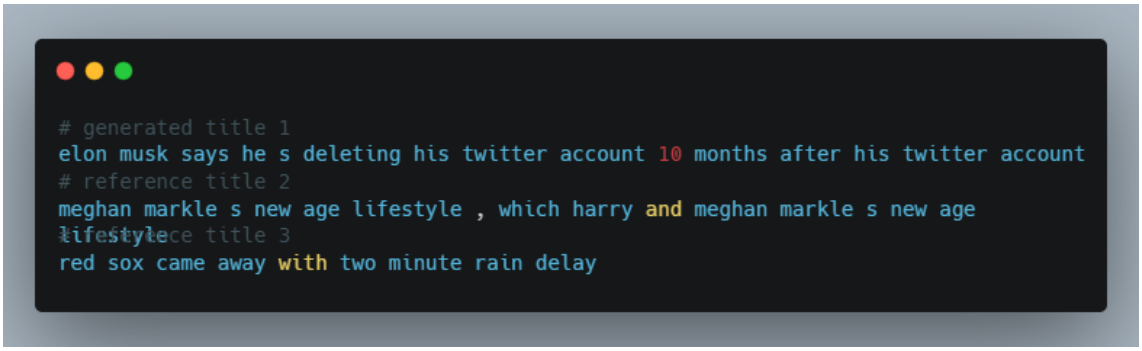

Figure 9: Generated Headlines

3 Prompt-engineering-based Method

3.1 Relay API Platform Construction

To achieve unified management and efficient calling of large language model APIs, this project built the ChavAPI relay platform (https://api.chavapa.com) through self-built servers based on existing open-source projects One-API and DeepClaude on Github. The construction of this platform provided solid technical support for the prompt engineering method.

The ChavAPI platform adopts a distributed architecture design and achieves high availability through Docker containerized deployment. Core platform functions include: multi-model unified interface encapsulation, intelligent load balancing, request throttling and quota management, real-time monitoring and logging, etc. In API key management, the platform supports multi-channel key polling usage, effectively distributing request pressure, and ensures key security through encrypted storage. For load balancing strategies, the system dynamically adjusts request allocation based on metrics such as response time and success rate of various API providers, ensuring service stability. Meanwhile, the platform provides detailed usage statistics and cost analysis functions, supporting precise billing by project and by user, providing strong support for project budget control.

Additionally, the ChavAPI platform strictly follows OpenAI API standards in compatibility design, enabling project code to seamlessly connect with different large language model service providers, including mainstream models like DeepSeek, Gemini series, Claude series, etc. This unified interface design not only simplifies code development processes but also provides convenience for potential horizontal model comparison experiments.

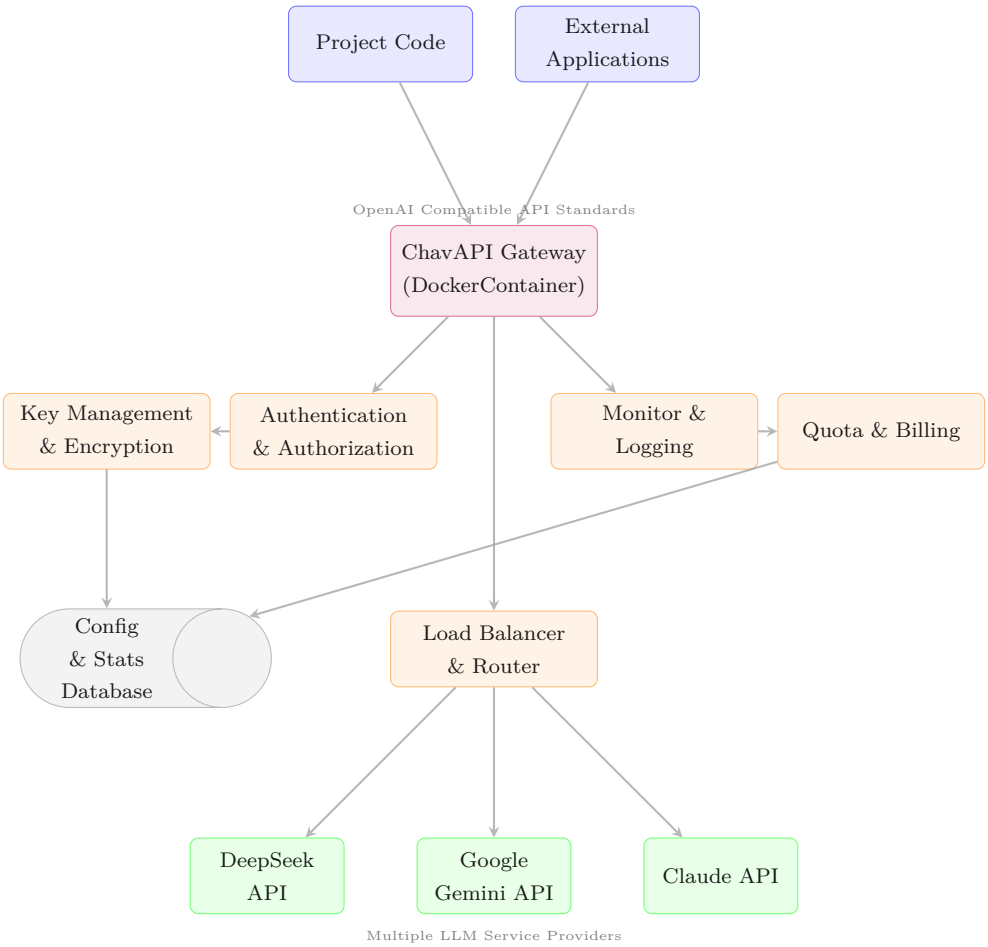

Figure 10: ChavAPI Platform Architecture

3.2 Local Environment Setup

The environment configuration for the prompt engineering method is relatively simple, mainly relying on the Python ecosystem and cloud API services. Local environment setup includes Python 3.8+ runtime, core dependency package installation (OpenAI, Pandas, NumPy, Rouge-Score, Matplotlib, etc.), and Jupyter Notebook development environment configuration. Compared to the baseline method, this solution does not require complex deep learning environment configuration, avoiding technical difficulties such as CUDA version compatibility and GPU memory management.

API configuration management adopts a template design, providing standard configuration templates through `api_config_template.py`. Users only need to copy to `api_config.py` and fill in corresponding API keys to complete configuration. Configuration files support multi-model parameter definitions, including differentiated settings for reasoning models (such as DeepSeek-R1 series) and chat models (such as DeepSeek-Chat series). The system automatically adjusts key configurations such as maximum token count and temperature parameters based on model types, ensuring optimal performance for different models.

The project adopts a modular directory structure design with core components including data processor (`data_processor.py`), prompt generator (`prompt_generator.py`), LLM client (`llm_client.py`), evaluator (`evaluator.py`), etc. Each module has clear responsibilities and is easy to maintain and extend. Output directories are organized by function categories, including subdirectories for preprocessed data, generated headlines, evaluation results, etc., ensuring orderly management of experimental data.

3.3 Prompt Engineering Method Analysis

3.3.1 Overall Architecture

The prompt engineering method adopts a "data-driven + intelligent prompting" technical approach, with the overall architecture divided into four core stages: data preprocessing and user profiling construction, adaptive prompt generation, large model API calling and content generation, multi-dimensional intelligent evaluation. Unlike traditional deep learning methods, this architecture fully utilizes the language understanding and generation capabilities of large language models, implementing personalized content generation through carefully designed prompt templates, avoiding complex model training processes.

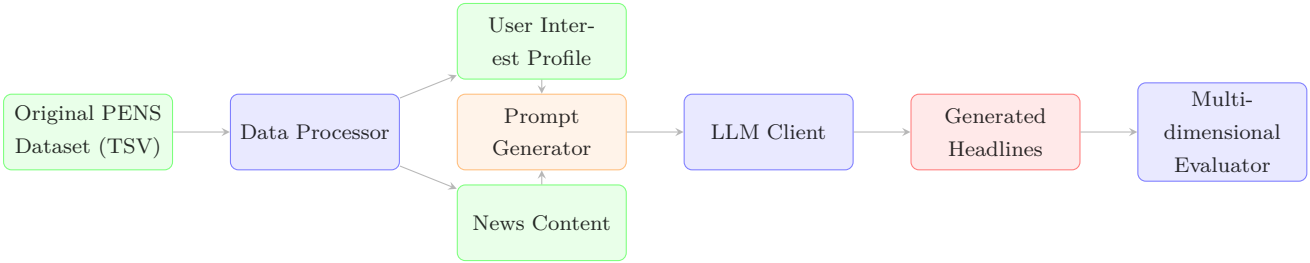

Figure 11: Prompt Engineering Method Workflow

The system workflow is as follows: First, the data processor extracts user historical click sequences from the original PENS dataset and constructs user interest profiles through statistical analysis, including main interest categories, preference weights, and other information; Subsequently, the prompt generator dynamically constructs personalized prompt templates based on target news content and user profiles; Next, the LLM client calls large language model APIs to generate personalized headlines based on the generated prompts; Finally, the evaluator uses multiple methods including ROUGE automatic evaluation and LLM intelligent evaluation to comprehensively evaluate generation results.

3.3.2 Core Component Analysis

Data Preprocessing Module (DataProcessor): This module is responsible for extracting user behavior sequences and news content from original TSV format data, constructing structured data suitable for prompt engineering. Core functions include user historical click sequence extraction, interest category statistical analysis, test sample preparation, etc. Unlike complex feature engineering in baseline methods, this module focuses on preserving and organizing semantic information, converting original text data into easily understandable natural language descriptions. The user interest extraction algorithm analyzes news category distributions in click history, calculates click frequency and weights for each category, and determines users' main interest areas. During data processing, the system truncates news content length (limited to within 400 characters) to ensure prompt total length is controlled within the model's context window range.

Prompt Generator (PromptGenerator): As the core component of the system, this module implements adaptive prompt generation strategies, dynamically selecting optimal prompt templates based on different model types (reasoning models vs. chat models) and task styles (focused vs. enhanced vs. creative). The module internally defines hierarchical prompt structures, including system-level prompts (defining task objectives and output specifications) and user-level prompts (containing specific news content and user information). The following is a core fragment of the system prompt:

Listing 1: System Prompt Example

```
system_prompt = """You are an AI expert at creating personalized news headlines.
Your task is to analyze user preferences and generate engaging, personalized
English headlines that match their interests.

Key requirements:
- Generate ONLY English headlines (no Chinese)
- Headlines should be 8-20 words long
- Make headlines personally relevant to the user's interests
- Maintain accuracy to the original news content
- Use engaging, attention-grabbing language
- Return ONLY the final headline text, no explanations"""
```

User prompt templates organically integrate user historical reading records, interest tags, original news content, and other information through Python string formatting techniques to form complete generation instructions. The system supports three styles of prompts: focused (concise and clear), enhanced (detailed analysis), and creative (emphasizing personalized expression). Different styles are suitable for different application scenarios and model characteristics.

LLM Client (LLMClient): This module encapsulates interaction logic with large language model APIs, supporting multi-model dynamic switching and adaptive parameter adjustment. The client implements intelligent content extraction strategies that can handle differences in response formats between reasoning models and chat models. For reasoning models, the system extracts final headline content from the reasoning field; for chat models, it directly obtains results from the content field. The module also includes comprehensive error handling and retry mechanisms to ensure API call stability. The following code demonstrates the core logic of adaptive content extraction:

Listing 2: Adaptive Content Extraction

```python
def _extract_content_adaptive(self, response):
    message = response.choices[0].message
    if self.is_reasoning_model:
        # Reasoning model: prioritize checking if content field is valid
        if message.content and message.content.strip():
            content = message.content.strip()
            if self._is_valid_title_content(content):
                return self.clean_generated_title(content)
        # If content is invalid, try extracting from reasoning field
        if hasattr(message, 'model_extra') and 'reasoning' in message.model_extra:
            reasoning = message.model_extra['reasoning']
            return self.extract_title_from_reasoning(reasoning)
    else:
        # Chat model: directly use content field
        if message.content and message.content.strip():
            return self.clean_generated_title(message.content.strip())
```

Evaluation Module (Evaluator): This module implements a multi-dimensional evaluation system, combining traditional ROUGE automatic evaluation with LLM-based intelligent evaluation to comprehensively measure the quality and personalization effects of generated headlines. ROUGE evaluation calculates lexical overlap between generated headlines and reference headlines, including ROUGE-1, ROUGE-2, ROUGE-L, and other metrics. LLM intelligent evaluation calls large language models to provide AI scoring for headline quality, with evaluation dimensions including accuracy, attractiveness, clarity, reasonableness, innovation, etc. Personalization effect evaluation combines rule engines with LLM evaluation to quantify personalization levels from perspectives such as interest matching, category relevance, and historical consistency.

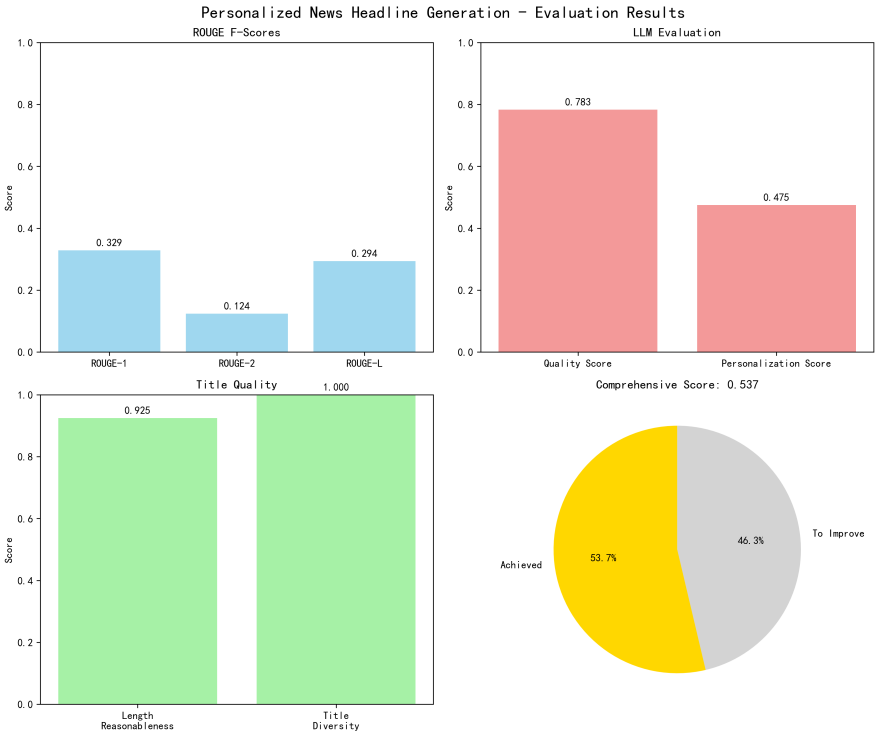

Figure 12: Prompt Engineering Method Evaluation Results

3.4 Experimental Results

Based on large language model calls through the ChavAPI platform, this project completed personalized headline generation experiments on test samples. Experiments used DeepSeek-Chat-V3 model as the main generation and evaluation model.

From ROUGE automatic evaluation results, the system achieved ideal performance: **ROUGE-1 F-score reached 0.3286, ROUGE-2 F-score was 0.1238, and ROUGE-L F-score was 0.2938**. These metrics are significantly higher than the peak performance of the baseline method, indicating that generated personalized headlines have good similarity with reference headlines in terms of vocabulary coverage and language structure.

LLM intelligent evaluation also provided objective and fair assessment of headline generation results. In quality evaluation, the system achieved a comprehensive quality score of 0.783 (out of 1.0), equivalent to 7.83 out of 10, indicating that generated headlines reached high levels in dimensions such as accuracy, attractiveness, and clarity. For personalization evaluation, interest matching reached 0.70, category relevance was 0.5625, and historical consistency was 0.575, demonstrating the system's ability to understand user preferences and generate relevant content.

From specific generation cases, the system can adjust headline expression and emphasis according to users' interest areas. For example, for users focusing on legal and political news, the system generated headline "High-Stakes Legal Battle Erupts Over Trump's Rollback of Obama-Era Pollution Rules" highlighting keywords like "legal battle" and "high-stakes"; for sports enthusiasts, headline "Verlander Scolded by MLB Over Ball-Tampering Claims Ahead of All-Star Game" emphasized specific players and league dynamics. These cases demonstrate the flexibility and precision of prompt engineering in personalized content generation.

In comprehensive scoring, the system achieved a final score of 0.5370 (out of 1.0), with LLM quality evaluation contributing the most (30% weight, score 0.783), and personalization effect evaluation also performing well (25% weight, score 0.475). This result validates the effectiveness of the prompt engineering method in balancing content quality and personalization effects.

4 Comparative Analysis of Two Methods

Through in-depth analysis and experimental validation of the PENS baseline method and prompt engineering method, comprehensive comparison can be made from multiple dimensions including technical architecture, implementation complexity, performance, and practicality.

Technical Architecture Comparison: The baseline method adopts a traditional deep learning pipeline, including data preprocessing, user encoder training, personalized generator pre-training and fine-tuning, with a relatively complex but logically clear overall architecture. The prompt engineering method adopts a simplified architecture of "data preprocessing + prompt design + LLM calling," avoiding complex model training processes but requiring higher quality prompt design. From a technical advancement perspective, the baseline method embodies classic applications of sequence-to-sequence generation and attention mechanisms, while the prompt engineering method represents emerging technical approaches in the era of large language models.

Implementation Complexity Comparison: The baseline method's environment setup is relatively complex, requiring CUDA environment configuration, GPU memory management, PyTorch version compatibility handling, etc., with high hardware requirements. The training process involves extensive hyperparameter tuning, including learning rate scheduling, batch size optimization, early stopping strategies, etc., requiring strong deep learning background knowledge. The prompt engineering method has a significantly lower implementation threshold, mainly relying on standard Python environments and API calls, without GPU hardware support requirements, with relatively low technical background requirements. However, prompt design itself is an art,

requiring deep understanding of the task domain.

Performance Comparison: From ROUGE evaluation metrics, the baseline method can achieve relatively high lexical overlap after training convergence, mainly benefiting from end-to-end training optimization and learning from large amounts of training data. The prompt engineering method's ROUGE scores showed significant improvement (ROUGE-1: 0.3286 vs. baseline method peak 0.2419), also performing more prominently in personalization effects. The baseline method's personalization mainly relies on implicit representations learned by user encoders, while the prompt engineering method can more intuitively integrate user interest information, showing advantages in dimensions like interest matching.

Scalability and Generalizability Comparison: Once the baseline method is trained, model parameters are relatively fixed, and extending to new domains or adapting to new requirements requires retraining, with high costs. The prompt engineering method has stronger flexibility, can quickly adapt to different task requirements by adjusting prompt templates, supporting multi-language and multi-domain extensions. Additionally, as large language model capabilities continue to improve, the prompt engineering method can continuously benefit from foundation model improvements without additional training costs.

Resource Consumption and Cost Comparison: The baseline method's training stage requires substantial computational resources, including GPU training time and power consumption, but the inference stage is relatively efficient. The prompt engineering method shifts computational burden to cloud API services, with low single-call costs, but may face API fee accumulation for large-scale applications. From research and development perspectives, the prompt engineering method enables faster prototype validation and iterative optimization.

Interpretability and Controllability Comparison: The baseline method's neural network structure has certain "black box" characteristics. Although some degree of interpretation is possible through attention weights, the overall decision process is relatively obscure. The prompt engineering method's generation process is more transparent, allowing users to intuitively understand how input information affects output results, facilitating debugging and optimization. In content safety and compliance, the prompt engineering method is also easier to implement precise control.

In summary, both methods have their advantages and are suitable for different application scenarios. The baseline method is more suitable for scenarios with extremely high performance requirements, sufficient data, and abundant computational resources, such as bulk headline generation for large media platforms. The prompt engineering method is more suitable for rapid prototype development, cross-domain applications, and scenarios with high personalization requirements, especially given the continuous improvement of large language model capabilities, its development prospects are broader. From technical development trends, the fusion application of both methods may be an important future direction, utilizing deep learning methods for user representation learning while combining prompt engineering for flexible content generation, balancing performance and system flexibility.

5 Group Division of Labor

Shouhe Zhu: Reproduction and improvement of baseline methods, research and analysis of dataset internal structure, writing of report sections 1-2.

Luofan Wang: Design and implementation of prompt engineering methods, construction and optimization of ChavAPI platform, construction of multi-dimensional evaluation system, writing of report sections 3-4.