Faculty of Natural and Mathematical Sciences
Department of Informatics

Bush House, King's College London, Strand Campus, 30 Aldwych, London WC2B 4BG
Telephone 020 7848 2145
Fax 020 7848 2851

KING'S College LONDON

7CCSMPRJ

Individual Project Submission 201X/1X

Name: Ahmed Karim

Student Number: 20029316

Degree Programme: Artificial Intelligence

Project Title: Uncertainty Quantification in Automated Essay Scoring using Large Language Models

Supervisor: Zheng Yuan

Word count: 11142

RELEASE OF PROJECT
Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

[✓] I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: [illegible]

Date:06/08/2024

Contents

Contents

Uncertainty Quantification in AES using LLMs

Ahmed Karim

Abstract

Automated Essay Scoring (AES) systems are increasingly essential in educational assessment due to their ability to provide low-cost, reliable, and consistent scoring of student writing. Recent advancements in broad-coverage Large Language Models (LLMs) like BERT and LLAMA have significantly improved AES accuracy. However, the credibility of these systems heavily depends on their ability to explicitly estimate prediction uncertainty. In this study, we investigate state-of-the-art LLMs in AES using a dataset from The Automated Student Assessment Prize (ASAP). We compare different techniques for estimating prediction uncertainty, including conformal prediction, Monte Carlo dropout, and ensemble methods. Our experiments with BERT and LLAMA 2/3 models examine how uncertainty estimates relate to AES accuracy. This work advances reliable uncertainty quantification methods for trustworthy AI in educational assessment, ultimately enhancing decision-making processes in academic settings.

1 Introduction

1.1 Overview

AES has emerged as a critical tool in the educational technology landscape, addressing the growing need for efficient, consistent, and scalable assessment of student writing. As educational institutions face increasing volumes of written assignments and standardized tests, AES systems offer a promising solution to reduce the workload on human graders while maintaining evaluation quality.

In recent years, the field of AES has been revolutionized by the introduction of LLMs such as BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al., 2018) and LLAMA (Large Language Model Meta AI) (Touvron et al., 2023). These advanced models, pre-trained on vast corpora of text, have demonstrated remarkable capabilities in understanding and generating human-like text, making them well-suited for the complex task of essay evaluation.

While LLM makes the AES systems achieve good performance, a critical point seems to be ignored about estimating how certain their predictions are. This is particularly crucial in high-stakes educational settings, where decisions made on the basis of essay scores can have major consequences for students' future academic pathways. Accounting for uncertainty properly may allow to sensitive cases, in which the model predictions should not be taken at face value leading possible human decision making or follow up action.

Our work in contrast, focuses on analyzing and comparing different uncertainty estimation methods from the context of AES with LLMs. We adopt technologies including conformal prediction, Monte Carlo dropout and ensemble methods to improve the confidence calibration of AES systems. Not only does our work advance the technical foundation of AES, but also it sheds light on broader implications with respect to fairness, transparency and trust regarding AI for educational assessment.

In exploring the nuances in uncertainty quantification within AES, we aim to enable more effective and ethical deployment of such systems for use cases that are mission-critical affecting student outcomes (and lives), instructor livelihoods, as well as institutional futures.

1.2 Motivation and Significance

The growing adoption of AES systems in educational contexts necessitates reliable uncertainty quantification to ensure fair and accurate assessments. This research is motivated by the significant impact that trustworthy evaluation can have on students' academic journeys, particularly in high-stakes situations like standardized tests or university admissions. By quantifying uncertainty in AES predictions, we aim to provide educators and administrators with crucial information for more informed decision-making, especially when dealing with borderline scores or unconventional writing styles. This approach not only enhances the fairness and equity of automated assessments but also promotes transparency, potentially increasing trust among students, parents, and educators in AI-driven evaluation systems.

Beyond its immediate application, this study contributes to the broader field of Natural Language Processing by exploring uncertainty quantification in long-form text analysis. The insights gained can inform the development of more sophisticated AES systems and potentially apply to other NLP tasks requiring reliable confidence estimates. Furthermore, understanding uncertainties in AES can provide valuable feedback to educators about areas where students may be struggling or where curricula might need adjustment, leading to more targeted teaching strategies and improved learning outcomes.

Ultimately, this research aims to foster the responsible and ethical implementation of AI in education. By addressing the crucial aspect of uncertainty in AI-driven educational tools, we contribute to ongoing discussions about the role of technology in assessment and learning. Our goal is to create more reliable, fair, and transparent AES systems that effectively support educators and students, promoting a more equitable and effective educational environment where technology serves as a tool for empowerment rather than a potential source of bias or misunderstanding.

1.3 Research Aims and Objectives

The primary aim of this research is to investigate and enhance uncertainty estimation in AES systems that utilize LLMs. By focusing on the application of advanced uncertainty quantification techniques, we seek to improve the reliability, transparency, and fairness of AES in educational settings.

To achieve this overarching aim, we have established the following specific objectives:

1. To implement and compare multiple uncertainty quantification techniques, including conformal prediction, Monte Carlo dropout, and ensemble methods, in the context of AES using LLMs.
2. To evaluate the performance of BERT and LLAMA 2/3 models on the ASAP dataset, specifically focusing on essay set 1, which encompasses essays with scores ranging from 2 to 12.
3. To analyse the relationship between model accuracy and uncertainty estimates, identifying patterns and insights that can inform the practical application of uncertainty quantification in AES.
4. To assess the effectiveness of each uncertainty quantification method in terms of its ability to reliably identify cases of high uncertainty in essay scoring.
5. To explore the potential of these uncertainty estimation techniques to enhance the interpretability and trustworthiness of AES systems, particularly in high-stakes educational contexts.
6. To investigate how uncertainty quantification can be used to optimize the allocation of

human grading resources in a hybrid human-AI grading system.

Through these objectives, we aim to contribute to the development of more robust and responsible AES systems that can be confidently deployed in real-world educational settings, ultimately benefiting students, educators, and institutions alike.

2 Background

2.1 Overview

This literature review examines the past work pertaining to AES systems and uncertainty quantification in the context of LLMs. The chapter is divided into three main sections corresponding to the key thematic areas of the project.

The first part focuses on how we have evolved from conventional to modern AES based approaches like deep learning and LLMs. We will also look at the constraints and limitations of AES, particularly in high-stakes educational circumstances.

The second part deals mainly with LLMs, in particular BERT and LLAMA, as well as their architectures, pre-training approaches and applications in NLP tasks. It will also go through how these models are adapted to the essay scoring task and review fine-tuning methods and their performances.

The third part of our literature review is a discussion about various uncertainty quantification techniques within machine learning as it applies to natural language processing. It will cover a variety of approaches including conformal prediction, Monte Carlo dropout, and ensemble techniques.

This literature review attempts to provide a thorough framework rooted in current AES-LLM uncertainty-quantification research by consolidating these three areas, assessing and analysing how they intersect. It will help specify areas where knowledge is lacking and provide the context for enhancing uncertainty quantification in AES systems using a number of state-of-the-art methods in automated assessment.

2.2 AES Systems

Recently, AES systems have improved significantly with the help of machine learning and natural language processing. Modern AES systems have moved beyond simple feature-based approaches to incorporate sophisticated algorithms capable of evaluating various aspects of writing quality.

The integration of machine learning techniques marked a significant leap in AES capabilities. These systems leverage statistical models trained on large datasets of human-scored essays to predict scores for new essays. Notable approaches include the use of Bayesian linear ridge regression, which demonstrated improved performance in capturing complex writing attributes (Phandi et al., 2015).

Deep learning has further transformed AES capabilities, enabling more nuanced evaluations of essay content and structure. The paper “A Neural Approach to AES” by Taghipour & Ng (2016) introduced a novel neural network architecture for AES that significantly outperformed previous methods. Their approach combined convolutional and recurrent layers in an innovative way. The input essay is first processed by a convolutional layer to capture local context features, followed by an LSTM layer to model long-range dependencies in the text. A mean-over-time layer then aggregates the LSTM outputs to produce a fixed-size representation of the essay.

In terms of input representation, essays were treated as sequences of words, with each word converted to a pre-trained word embedding. This allowed the model to capture semantic information effectively. The model achieved state-of-the-art performance on the ASAP dataset, outperforming previous machine learning approaches and even some ensemble methods.

One of the key advantages of using RNNs and LSTMs in this context was their ability to effectively capture complex linguistic features such as coherence and discourse structure. These aspects are crucial for accurate essay scoring but had been challenging for traditional feature-based methods to address adequately.

While neural networks are often considered "black boxes," the authors showed that their model's attention mechanism could provide insights into which parts of the essay were most important for scoring, adding a degree of interpretability to the model.

These studies collectively demonstrated the power of RNNs and LSTMs in capturing the sequential nature of text and modelling long-range dependencies, which are crucial for understanding the overall quality and coherence of an essay. The ability of these architectures to process text as a sequence and maintain context over long distances made them particularly well-suited to the task of essay evaluation.

However, it's worth noting that while RNNs and LSTMs marked a significant advancement in AES, they have been largely superseded by transformer-based models like BERT in recent years (Vaswani et al., 2017). Nonetheless, understanding these architectures provides valuable context for the evolution of neural approaches to AES and lays the groundwork for appreciating the capabilities of more recent models.

Despite these advancements, AES systems face ongoing challenges, including concerns about bias, their ability to evaluate creativity, and their reliability in high-stakes assessment contexts. The integration of uncertainty quantification techniques in AES represents a promising direction for addressing some of these challenges, potentially enhancing the reliability and interpretability of these systems.

2.3 LLMs in AES

The application of LLMs in AES has seen significant advancements, particularly with the introduction of BERT. The work by Yang et al. (2020) represents a notable contribution to this field, addressing several key challenges in applying pre-trained language models to the task of essay scoring.

Yang et al. recognized that while pre-trained models like BERT have shown extraordinary abilities in various natural language processing tasks, their application to AES has faced unique obstacles. These challenges stem from the mismatch between sentence-level pre-training and essay-level evaluation, the limited availability of AES training data, and the inadequacy of commonly used loss functions in capturing the nuances of essay scoring.

To overcome these limitations, the authors proposed a novel approach called R^{2}BERT, which combines the power of BERT's deep semantic understanding with a multi-objective learning strategy. This method utilizes BERT to learn rich text representations of essays, followed by a fully connected neural network to map these representations to scores. The key innovation lies in the simultaneous application of regression and ranking losses to constrain the predicted scores.

The R^{2}BERT model employs Mean Squared Error for regression, which aims for accurate score prediction, alongside a batch-wise ListNet loss for ranking, which ensures proper ordering of essays within each batch. These losses are combined using dynamic weights that evolve during training, allowing the model to balance between accurate score prediction and maintaining correct relative rankings among essays.

This approach demonstrated significant improvements over previous state-of-the-art neural models when evaluated on the ASAP dataset. R^{2}BERT showed particular strength in scoring longer essays and narrative prompts, outperforming other models by a substantial margin in these areas.

The success of R^{2}BERT underscores the importance of adapting pre-trained language models to the specific requirements of AES tasks. It illustrates that by combining multiple learning objectives, models can achieve a more nuanced understanding of essay quality, leading to more robust and accurate scoring. This work opens up new possibilities for leveraging the power of LLMs in educational technology, particularly in the automated assessment of student writing, and suggests a promising direction for future research in this field.

2.4 Uncertainty Evaluation Techniques for LLMs

The rapid advancement of LLMs has brought to light the critical need for comprehensive evaluation methods that go beyond simple accuracy metrics. In this context, the work by Ye et al. (2023) introduces a novel approach to benchmarking LLMs by incorporating uncertainty quantification, addressing a significant gap in existing evaluation frameworks.

At the heart of their methodology is the use of conformal prediction, a technique that offers a robust and efficient means of quantifying uncertainty in LLMs. This approach stands out for its ease of implementation, computational efficiency, and statistical rigor. Unlike alternative methods that may require significant modifications to model architectures or rely on heuristic approximations, conformal prediction can be seamlessly applied to existing LLMs while providing statistically guaranteed estimations of uncertainty.

The authors explore two main conformal score functions: Least Ambiguous set-valued Classifiers (LAC) and Adaptive Prediction Sets (APS). These functions serve to transform the softmax scores produced by LLMs into more meaningful measures of uncertainty. LAC focuses on the softmax score of the true label, while APS takes into account the scores of multiple labels, offering different perspectives on model uncertainty.

To provide a more holistic view of LLM performance, the paper introduces several evaluation metrics. While traditional accuracy is still considered, it is complemented by the Set Size metric, which reflects the average length of prediction sets generated by conformal prediction. This metric serves as a direct indicator of model uncertainty. Furthermore, the authors propose a novel Uncertainty-aware Accuracy (UAcc) metric, which ingeniously combines accuracy and uncertainty into a single measure.

The implementation process outlined in the paper involves converting various NLP tasks into a multiple-choice question answering format, extracting probabilities from LLMs through prompting, and applying conformal prediction to generate prediction sets. This standardized approach allows for consistent evaluation across different tasks and model scales.

By incorporating uncertainty quantification into the evaluation of LLMs, this methodology provides deeper insights into model performance. It not only assesses how often a model is correct but also how confident it is in its predictions. This comprehensive approach to evaluation is particularly valuable in the context of LLMs, where understanding the limitations and reliability of these powerful models is crucial for their responsible deployment in real-world applications.

2.4.1 UAcc metric

The UAcc metric introduced by Ye et al. (2023) represents a significant advancement in the evaluation of LLMs. This novel metric is designed to provide a more nuanced assessment of model performance by incorporating both prediction accuracy and uncertainty.

UAcc is calculated using the formula:

$$\mathrm{UAcc} = \mathrm{Acc} \cdot \frac{\sqrt{\mathrm{num\_classes}}}{\mathrm{average\_set\_size}}$$

This formulation cleverly balances the model's accuracy with the size of its prediction sets. The inclusion of the square root of the number of classes allows UAcc to be scaled relative to the complexity of the task, making it comparable across different scenarios.

What makes UAcc particularly valuable is its ability to penalize models that are accurate but highly uncertain, as well as those that are confident but inaccurate. For instance, a model that achieves high accuracy but produces large prediction sets (indicating high uncertainty) will receive a lower UAcc score compared to a model with similar accuracy but smaller prediction sets. Conversely, a model with narrow prediction sets but low accuracy will also be penalized.

The authors demonstrate that UAcc can reveal insights that are not apparent when considering accuracy or uncertainty in isolation. In their experiments, they show cases where LLMs with similar accuracy scores exhibit different levels of uncertainty, leading to distinct UAcc scores. This metric can even alter the relative ranking of models compared to traditional accuracy-based evaluations.

Furthermore, UAcc proves to be a useful tool for analyzing the effects of model scaling and instruction fine-tuning on LLM performance. The authors observed that larger models don't always lead to higher UAcc scores, and that instruction fine-tuning can sometimes increase uncertainty, leading to lower UAcc despite maintaining or improving accuracy.

By providing a single, comprehensive measure that accounts for both correctness and confidence, UAcc offers a more holistic view of LLM performance. This metric aligns well with the growing recognition in the field that responsible AI deployment requires not just accurate models, but also ones that can reliably estimate their own uncertainty. As such, UAcc represents an important step forward in the development of more sophisticated and informative evaluation techniques for LLMs.

2.5 Conformal Prediction

Conformal prediction is a robust, distribution-free, and model-agnostic approach to uncertainty quantification in machine learning (Angelopoulos and Bates, 2021;). It provides a framework to transform any heuristic notion of uncertainty from a model into a statistically rigorous one. In the context of AES, conformal prediction can output a set of possible scores that contains the true score with a user-specified error rate, expressing uncertainty as the size of this set. Intuitively, a larger set size indicates higher uncertainty and vice versa.

Formally, let f be a model that assigns a score to an input essay X from a set of possible scores $Y = \{1, \ldots, K\}$. To quantify the uncertainty of f, for any given test essay X_{test} and its corresponding true score Y_{test}, conformal prediction produces a prediction set $C(X_{test}) \subset Y$ such that:

$$P(Y_{test} \in C(X_{test})) \geq 1 - \alpha,$$

where $\alpha \in (0, 1)$ is a user-specified error rate.

This coverage guarantee is achieved using a held-out calibration dataset. The conformal prediction process involves defining a conformal score function $s(X, Y)$ that measures the disagreement between an essay X and a potential score Y, computing these scores on the calibration set, and using them to determine a threshold for constructing prediction sets for test essays.

Two common conformal score functions are LAC (Sadinle et al., 2019) and (APS)). LAC defines the score as $s(X, Y) = 1 - f(X)_Y$, where $f(X)_Y$ is the model's confidence in the true score. APS, on the other hand, sums the ranked confidences until reaching the true score. While LAC can produce smaller prediction sets, it may undercover difficult essays and overcover easy ones. APS addresses this limitation but tends to produce larger prediction sets on average.

By applying these methods to AES, we can obtain statistically rigorous uncertainty estimates, providing valuable information about the reliability of the predicted scores across different types of essays and scoring models.

2.6 Monte Carlo Dropout

Monte Carlo (MC) dropout is a practical and widely-used approach for estimating model uncertainty in deep neural networks (Gal and Ghahramani, 2016). It provides a simple yet effective method to approximate Bayesian inference, allowing for the quantification of both epistemic uncertainty (model uncertainty) and aleatoric uncertainty (data uncertainty) without the need for significant modifications to the existing model architecture.

The key idea behind MC dropout is to leverage the dropout regularization technique, typically used during training to prevent overfitting, as a method for sampling from an approximate posterior distribution over the model parameters. In the context of AES, this approach can be used to estimate the uncertainty in predicted essay scores.

Formally, let f be a neural network model that assigns a score to an input essay X. During inference, instead of disabling dropout as is typically done, MC dropout keeps dropout active and performs multiple forward passes (typically 10-100) through the network for each input essay. Each forward pass effectively samples a different subset of neurons, which can be interpreted as sampling from a distribution over possible models.

For a given test essay X_{test}, MC dropout generates T different score predictions:

$$\{\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(T)}\} = \{f(X_{test}; W^{(1)}), f(X_{test}; W^{(2)}), ..., f(X_{test}; W^{(T)})\} \tag{1}$$

where $W^{(t)}$ represents the model parameters for the t-th forward pass, effectively sampled by dropout.

The final prediction is typically taken as the mean of these T predictions:

$$\hat{y} = \frac{1}{T}\sum_{t=1}^{T} \hat{y}^{(t)} \tag{2}$$

The uncertainty in this prediction can be estimated using the variance of the T predictions:

$$\sigma^2 = \frac{1}{T}\sum_{t=1}^{T} (\hat{y}^{(t)} - \hat{y})^2 \tag{3}$$

This variance provides a measure of the model's uncertainty in its prediction for the given essay. A higher variance indicates greater uncertainty, which could suggest that the model is less confident in its score prediction for that particular essay.

By applying MC dropout to AES models, we can obtain uncertainty estimates for predicted scores, providing valuable information about the reliability of the model's predictions across different types of essays and scoring criteria.

2.7 Ensemble Methods for Uncertainty Estimation

Ensemble methods represent a powerful approach for estimating model uncertainty in machine learning tasks, including AES (Lakshminarayanan et al., 2017). These methods involve training multiple models and combining their predictions to not only improve overall performance but also to quantify predictive uncertainty.

In the context of AES, an ensemble typically consists of M different models, each trained to predict essay scores. These models may vary in their architecture, initialization, or the subset of training data they use. The key idea is that the diversity among these models captures different aspects of the underlying data distribution and model uncertainty.

Formally, given an input essay X, each model in the ensemble produces a score prediction:

$$\{\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(M)}\} = \{f_1(X), f_2(X), ..., f_M(X)\} \tag{4}$$

where $f_m(X)$ represents the prediction of the m-th model in the ensemble.

The final score prediction is typically computed as the mean of these individual predictions:

$$\hat{y} = \frac{1}{M} \sum_{m=1}^{M} \hat{y}^{(m)} \tag{5}$$

The uncertainty in this prediction can be estimated using the variance of the M predictions:

$$\sigma^2 = \frac{1}{M} \sum_{m=1}^{M} (\hat{y}^{(m)} - \hat{y})^2 \tag{6}$$

This variance provides a measure of the ensemble's uncertainty in its prediction for the given essay. A higher variance indicates greater disagreement among the ensemble members, suggesting higher uncertainty in the score prediction.

Ensemble methods offer several advantages for uncertainty estimation in AES:

1. They capture both epistemic uncertainty (model uncertainty) and aleatoric uncertainty (data uncertainty).
2. They are relatively simple to implement and can be applied to a wide range of model architectures.
3. They often provide more reliable uncertainty estimates compared to single model approaches.

By applying ensemble methods to AES, we can obtain robust uncertainty estimates for predicted scores, offering valuable insights into the reliability of the model's predictions across various types of essays and scoring criteria. This information can be particularly useful in identifying essays that may require additional human review or in providing confidence intervals for automated scores.

3 Objectives and Specification

3.1 Problem Overview and Objectives

The core objective of this research is to investigate and enhance uncertainty estimation in AES systems that utilize LLMs. By focusing on the application of advanced uncertainty quantification techniques, we aim to improve the reliability, transparency, and fairness of AES in educational settings.

Specifically, this study seeks to:

1. Implement and compare multiple uncertainty quantification techniques, including conformal prediction, Monte Carlo dropout, and ensemble methods, in the context of AES using LLMs.

2. Evaluate the performance of BERT and LLAMA 2/3 models on the ASAP dataset, with a particular focus on Essay set 1, which encompasses essays with scores ranging from 2 to 12.
3. Analyse the relationship between model accuracy and uncertainty estimates, identifying patterns and insights that can inform the practical application of uncertainty quantification in AES.
4. Assess the effectiveness of each uncertainty quantification method in terms of its ability to reliably identify cases of high uncertainty in essay scoring.
5. Explore the potential of these uncertainty estimation techniques to enhance the interpretability and trustworthiness of AES systems, particularly in high-stakes educational contexts.
6. Investigate how uncertainty quantification can be used to optimize the allocation of human grading resources in a hybrid human-AI grading system.

Through these objectives, we aim to contribute to the development of more robust and responsible AES systems that can be confidently deployed in real-world educational settings, ultimately benefiting students, educators, and institutions alike.

4 Requirements

4.1 Functional Requirements

The following functional requirements outline the core functionality that the AES system with uncertainty quantification must have:

- Import and process essays from the ASAP dataset, particularly essay set 1.
- Utilize BERT and LLAMA 2/3 models for essay representation and scoring.
- Implement multiple uncertainty quantification methods:
  - Conformal prediction
  - Monte Carlo dropout
  - Ensemble techniques
- Generate essay scores along with associated uncertainty estimates.
- Evaluate model performance using both accuracy metrics and uncertainty-aware metrics.
- Provide functionality to compare different uncertainty quantification methods.
- Allow for fine-tuning of LLMs on the essay scoring task.

4.2 Non-Functional Requirements

While adhering to the functional requirements listed above, the system must also meet several non-functional requirements:

- Scalability: Able to handle large volumes of essays efficiently.
- Reproducibility: Ensure consistent results across multiple runs.
- Documentation: Provide clear documentation for all components and functions.
- Performance: Optimize for reasonable inference time, considering the computational demands of LLMs.
- Flexibility: Allow for easy configuration of hyperparameters and model settings.

5 Technical Specifications

For the project, uncertainty-aware AES models have been created using Python 3.8. It is written in Python because of its readability, powerful library support & prevalence within the ML-community. For AES and NLP, Python has a rich set of libraries / frameworks that makes it really easy to implement big language models in combination with uncertainty quantification methods.

The deep learning library selected for implementing the LLM-based AES models with uncertainty quantification is PyTorch 1.9. PyTorch is widely adopted in the research community for its ease of use in designing custom neural architectures. This flexibility is particularly valuable for our project, which requires implementation of non-standard uncertainty quantification methods and adaptation of pre-trained language models for the essay scoring task.

Hardware Specifications: 8-core Intel Xeon Processor @3.2GHz, 64GB RAM, NVIDIA A100 40GB GPU

Software Specifications:

- Programming Language: Python 3.8
- Deep Learning Framework: PyTorch 1.9
- Key Python Libraries:
  - Transformers (Hugging Face)
  - NumPy
  - Pandas
  - PyMC3 (for Monte Carlo dropout)
  - Scikit-learn (for ensemble methods)

6 Model Overview

The model pipeline highlights the training procedure for both the BERT-based and LLAMA-based essay scoring, along with the uncertainty quantification using conformal prediction.

Our AES system utilizes two distinct LLMs: BERT and LLAMA-2/3. While both models aim to predict essay scores, their architectures and processing methods differ significantly.

6.1 BERT-based Model

The input to the BERT model is the essay text, which is processed through a pre-trained BERT model to generate contextual embeddings. These embeddings are then passed through a task-specific classification layer to predict essay scores. The BERT model and classification layer are fine-tuned on the ASAP dataset to adapt to the essay scoring task.

The BERT model processes the input essay text by first tokenizing it into WordPiece tokens. These tokens are then passed through the BERT layers, which consist of multi-head self-attention mechanisms and feed-forward neural networks. The output of the [CLS] token from the final BERT layer is used as the essay representation. This representation is then fed into a fully connected layer to produce the essay score prediction.

6.2 LLAMA-based Model

For the LLAMA-2/3 model, we employ a different approach leveraging its capabilities as a generative model. The essay text is incorporated into a prompt that instructs the model to score the essay. For example: "Score the following essay on a scale of 2 to 12: [ESSAY TEXT]". This prompt is then tokenized and processed through the LLAMA-2/3 model.

The LLAMA-2/3 model, which utilizes a transformer architecture similar to BERT but with architectural improvements and significantly more parameters, processes the entire prompt. The model's output is then interpreted to extract the predicted score. We use the 7B parameter version of LLAMA-2/3, employing 4-bit quantization for efficient processing.

To fine-tune LLAMA-2/3 for our task, we use Low-Rank Adaptation (LoRA), which allows for efficient adaptation of the large model by training a small number of parameters.

6.3 Uncertainty Quantification

For both BERT and LLAMA models, we implement conformal prediction for uncertainty quantification. The conformal prediction module takes the output logits (for BERT) or the final layer activations (for LLAMA) as input. It uses these to compute nonconformity scores, which measure how different a new example is from the calibration set. These scores are then used to construct prediction sets with a guaranteed coverage rate.

This approach allows us to provide not just point estimates of essay scores, but also a range of possible scores with a specified confidence level, enhancing the reliability and interpretability of our AES system.

By implementing both BERT and LLAMA-2 models with conformal prediction, we can compare the performance of different model architectures and processing methods in the context of AES, while also providing robust uncertainty quantification for both approaches.

7 Dataset

7.1 Design and Implementation

This project utilizes Prompt 1 from the ASAP dataset, a benchmark widely recognized in AES research. The dataset, loaded from "trainingsetrel3.xlsx", consists of 1,783 essays written by eighth-grade students, with scores ranging from 2 to 12.

Our dataset preparation were inspired by the methodologies described by Xiao et al. (2024) . This is particularly for the BERT model. They include source code for setting up BERT in the context of AES.

7.1.1 Data Preparation

We implemented a custom text prepossessing function to clean the essay text, removing Twitter-style mentions, replacing HTML-encoded characters, and eliminating excessive whitespace. The scoring scale was adjusted to start from 0 by subtracting the minimum score.

7.1.2 Data Split

Using scikit-learn's train_test_split function, we divided the dataset into:

- 70% training set
- 15% validation set
- 15% test set

A fixed random seed of 42 was used to ensure reproducibility.

7.1.3 BERT-specific Processing

For BERT, we utilized the bert-base-uncased tokenizer with a maximum sequence length of 512 tokens. A custom BERTDataset class was created to handle data preprocessing and convert inputs to PyTorch tensors.

7.1.4 LLAMA-specific Processing

For LLAMA, we used the LlamaTokenizer with the same maximum sequence length of 512 tokens. We created a custom LlamaDataset class to handle LLAMA-specific preprocessing, including prompt creation.

7.1.5 Prompt Creation

For LLAMA, we implemented a create_prompt function to format each essay into a suitable prompt structure: "Score the following essay on a scale of 2 to 12: [essay_text] Score:"

RandomSampler was used for the training set to shuffle data, while SequentialSampler was used for validation and test sets. The BERT implementation uses a batch size of 16, while LLAMA, due to its larger size and memory requirements, uses a smaller batch size of 4. This setup allows for efficient batching and loading of data during both training and evaluation phases, balancing between computational efficiency and model performance.

This dataset design allows for a consistent evaluation of both BERT and LLAMA models on the same essay set, allowing a fair comparison of their performance in AES tasks. We chose essay set 1 due to the fact it had the most diverse scores so if any essay set was to measure uncertainty the most vigorously it would be essay set 1. Due to time constraints, we could not also test other essay sets.

8 LLM Architecture and Fine-tuning

8.1 BERT Architecture

8.1.1 Design

Our BERT-based model for AES is built upon the 'bert-base-uncased' pre-trained model. This architecture leverages BERT's powerful contextual understanding capabilities. The model consists of two main components: the BERT encoder for feature extraction and a linear classification layer for score prediction.

The BERT encoder analyzes the tokenized essay text, producing detailed contextual representations for each token. These representations take the semantic meaning of individual words as well as their context within the essay. This is essential for grasping subtleties in writing style, argument structure, and content relevance.

On top of the BERT encoder, we add a task-specific linear classification layer. This layer maps the high-dimensional features extracted by BERT to a score prediction. The flexibility of this design allows us to easily adapt the model to different scoring scales across various essay prompts.

8.1.2 Implementation

The implementation of our BERT-based model is in the BERTClassifier class. This class initializes the BERT encoder with pre-trained weights and adds a linear classifier on top. The forward method defines the flow of data through the model, from input tokens to score logits.

We use the AdamW optimizer with a learning rate of 1e-5, which is typically effective for fine-tuning pre-trained models. To handle the learning rate schedule, we implement a linear learning rate scheduler with warmup steps. This helps to stabilize the early stages of training and potentially improves convergence.

The training process uses cross-entropy loss as the objective function, appropriate for our multi-class classification task. We also employ gradient clipping to prevent gradient explosion, a common issue when fine-tuning large pre-trained models.

8.2 LLAMA 2/3 Architecture

8.2.1 Design

For our LLAMA 2/3 implementation, we adopt a more advanced approach, incorporating quantization and parameter-efficient fine-tuning techniques. This design aims to leverage the power of larger language models while managing computational resources effectively.

The core of this architecture is the LlamaForCausalLM model, which we enhance with 4-bit quantization. This quantization significantly reduces the memory footprint of the model, allowing us to work with larger model variants even with limited GPU resources. The quantized model maintains most of the performance of the full-precision model while offering substantial efficiency gains.

To further optimize our model for the essay scoring task, we implement Low-Rank Adaptation (LoRA). LoRA allows us to fine-tune the model efficiently by adding small, trainable rank decomposition matrices to specific layers of the model. This approach keeps most of the pre-trained weights frozen, reducing the number of trainable parameters and mitigating the risk of catastrophic forgetting.

8.2.2 Implementation

Our implementation begins with configuring the model for 4-bit quantization using the BitsAndBytesConfig. We set parameters such as load_in_4bit, bnb_4bit_use_double_quant, and bnb_4bit_quant_type to optimize the quantization process. The model is then loaded with this configuration, ensuring it's prepared for memory-efficient training and inference.

We apply LoRA to the model using a carefully tuned LoraConfig. Our configuration targets the query and value projection modules (`"q_proj"` and `"v_proj"`) with a rank of 8 and an alpha of 32. These settings allow for effective task-specific adaptation while keeping the majority of the pre-trained weights intact.

A custom LlamaDataset class is implemented to handle data preprocessing. This class is responsible for tokenizing essays and preparing prompts based on different strategies: zero-shot without rubrics, zero-shot with rubrics, and few-shot with rubrics. The create_prompt function generates appropriate prompts for each essay, allowing for flexible experimentation with different prompting strategies, however in this work we only work with zero shot without rubrics due to time constraints.

To manage memory usage during training, we enable gradient checkpointing. This technique trades computation for memory, allowing us to train larger models or use larger batch sizes. The model is set up to process essays and output scores on a scale of 2 to 12, with an internal adjustment to a 0-10 range for training purposes.

This implementation combines advanced techniques in model quantization, parameter-efficient fine-tuning, and prompt engineering. It allows us to leverage the powerful language understanding capabilities of LLAMA 2/3 for the essay scoring task while maintaining computational efficiency.

9 Uncertainty Quantification Methods

In our AES system, we implement various uncertainty quantification methods to enhance the reliability and interpretability of our predictions. These methods allow us to provide not just point estimates of essay scores, but also a measure of confidence in these predictions. This section details the approaches we've employed, starting with Conformal Prediction.

9.1 Conformal Prediction

Conformal Prediction is a framework that allows us to construct prediction sets with a guaranteed coverage rate. In our essay scoring context, it means we can provide a range of possible scores for each essay, with a specified confidence level.

9.1.1 Design

Our implementation of Conformal Prediction varies slightly between our BERT and LLAMA 2/3 models. The key idea is to use the model's softmax probabilities to compute nonconformity scores, which measure how different a new example is from the calibration set. We then use these scores to construct prediction sets for new essays.

9.1.2 BERT Implementation

For the BERT model, we primarily use the LAC method for computing nonconformity scores. This method computes the score as 1 minus the probability assigned to the true label. While our implementation includes an option for APS, we did not use this method in our final experiments.

9.1.3 LLAMA 2/3 Implementation

For the LLAMA model, we use a similar approach to LAC:

- The nonconformity score is computed as 1 minus the probability assigned to the true label.

9.1.4 Implementation

The Conformal Prediction process is implemented slightly differently for each model:

BERT:

1. The validation dataset is split into calibration and test sets.
2. Nonconformity scores are computed for the calibration set using the LAC method.
3. A threshold (qhat) is calculated based on the desired confidence level (1 - alpha, where alpha is typically set to 0.1).
4. For each test example, a prediction set is constructed by including all labels whose nonconformity scores are below the threshold.

LLAMA:

1. The entire validation dataset is used to compute nonconformity scores.
2. The dataset is then split into calibration and test sets.
3. A threshold (qhat) is calculated based on the calibration set scores.
4. Prediction sets are constructed for the test set using this threshold.

9.1.5 Evaluation

Both implementations are evaluated using several metrics:

- Coverage: The proportion of test examples where the true label is in the prediction set.
- Average Set Size: The average number of labels in the prediction sets.
- UAcc: This novel metric combines accuracy and set size, providing a balanced view of the model's performance under uncertainty.

The UAcc metric is calculated as:

$$\text{UAcc} = \frac{1}{N}\sum_{i=1}^{N}\left(I(y_i \in C_i) \cdot \frac{\sqrt{|Y|}}{|C_i|}\right)$$

Where N is the number of samples, I is the indicator function, y_i is the true label, C_i is the prediction set, $|Y|$ is the number of possible classes, and $|C_i|$ is the size of the prediction set.

In addition to these uncertainty-related metrics, we also compute standard performance metrics such as accuracy, F1 score, and Quadratic Weighted Kappa (QWK) to provide a comprehensive evaluation of our models' performance.

9.2 Monte Carlo Dropout

Monte Carlo Dropout is a technique used for estimating model uncertainty in neural networks. It leverages dropout, typically used as a regularization technique during training, as a method for approximate Bayesian inference during prediction.

9.2.1 Design

In our AES system, we implement MC Dropout for both our BERT and LLAMA 2/3 models. The key idea is to keep dropout active during inference and perform multiple forward passes for each input. This generates a distribution of predictions, from which we can estimate both the final score and the model's uncertainty.

Our design incorporates the following key elements:

1. Dropout layers in the model architecture: For BERT, we add a dropout layer before the final classification layer. For LLAMA 2/3, we utilize the existing dropout in the model.
2. Multiple forward passes: During evaluation, we perform 5 forward passes for each input, each with a different dropout mask.
3. Aggregation of results: We compute the mean and variance of the logits from these multiple passes to get the final prediction and an uncertainty estimate.
4. Uncertainty-based prediction sets: We use the uncertainty estimates to generate prediction sets, allowing for more nuanced scoring that accounts for model confidence.

9.2.2 Implementation

The MC Dropout process is implemented in the evaluate_with_mc_dropout function. Here are the key steps:

1. We set the model to evaluation mode but keep dropout active.
2. For each batch in the validation/test set, we perform multiple forward passes (num_iterations, set to 5).
3. We compute the mean and variance of the logits from these passes. The mean is used for the final prediction, while the variance serves as an uncertainty measure.
4. We use the maximum variance for each sample as its uncertainty score.

5. Based on these uncertainty scores and a probability threshold, we generate prediction sets for each sample.
6. We evaluate the model's performance using standard metrics (accuracy, F1 score, QWK) and uncertainty-aware metrics (coverage, average set size, and UAcc).

The implementation allows for a balance between computational efficiency and robust uncertainty estimation. By adjusting the number of iterations and the probability threshold, we can fine-tune the trade-off between precise uncertainty estimates and inference speed.

This approach provides several benefits. It allows us to quantify model uncertainty without altering the model architecture or training procedure. Additionally, it offers a means to identify samples where the model exhibits lower confidence, which is particularly important in high-stakes applications such as essay scoring. The uncertainty estimates obtained can guide decisions on when human intervention may be required in the scoring process.

By incorporating MC Dropout, our AES system not only delivers score predictions but also generates a measure of confidence in those predictions, thereby enhancing the system's reliability and interpretability.

9.3 Ensemble Techniques

Ensemble techniques combine predictions from multiple models to improve overall performance and provide more robust uncertainty estimates. In our AES system, we implement ensemble methods for both our BERT and LLAMA 2/3 models.

9.3.1 Design

Our ensemble approach involves training multiple instances of each model type (BERT and LLAMA) with different random initialization. The key components of our design include:

1. Multiple model training: We create and train several instances of each model type, typically 3-5 models.
2. Aggregation of predictions: During inference, we combine predictions from all models in the ensemble.
3. Uncertainty estimation: We use the disagreement between models as a measure of uncertainty.
4. Integration with Conformal Prediction: We apply conformal prediction on top of the ensemble predictions to generate reliable prediction sets.

9.3.2 Implementation

The ensemble method is implemented slightly differently for BERT and LLAMA models, but the core principles remain the same. Here's an overview of the implementation:

For BERT:

1. We create multiple instances of the BERTClassifier model, each with a different random seed.
2. Each model is trained independently on the training data.
3. During evaluation, we pass each input through all models in the ensemble.
4. We aggregate the logits from all models by taking their mean.
5. The final prediction is made based on these averaged logits.

For LLAMA:

1. We create multiple instances of the LlamaForCausalLM model, each initialized with the same pre-trained weights but fine-tuned independently.
2. Each model is fine-tuned on the essay scoring task using LoRA (Low-Rank Adaptation) for efficiency.
3. During evaluation, we pass each input through all models in the ensemble.
4. We aggregate the probabilities from all models by taking their mean.
5. The final prediction is made based on these averaged probabilities.

For both model types, we implement the following ensemble-specific metrics:

1. Ensemble Accuracy: The accuracy of the predictions made by the averaged outputs of all models.
2. Prediction Variance: The variance of predictions across different models, serving as a measure of uncertainty.
3. Model Disagreement: The extent to which different models in the ensemble disagree on their predictions.
4. Average Entropy: The average entropy of the ensemble's predictions, another measure of uncertainty.

We also integrate the ensemble predictions with Conformal Prediction:

1. We use the averaged probabilities from the ensemble as inputs to the conformal prediction algorithm.
2. We compute conformal scores and generate prediction sets based on these ensemble probabilities.
3. We evaluate the ensemble + conformal prediction approach using metrics like coverage, average set size, and UAcc.

The ensemble approach presents multiple advantages, starting with improved accuracy; by integrating predictions from various models, we frequently attain a higher level of accuracy compared to what individual models can provide. Additionally, this method enhances uncertainty quantification, as the varying predictions among the models serve as a natural indicator of uncertainty. Furthermore, ensembles tend to exhibit greater robustness, making them less susceptible to the failures or biases that might affect individual models.

By incorporating ensemble techniques, our AES system not only improves its predictive performance but also provides more reliable uncertainty estimates, further enhancing the system's trustworthiness and applicability in real-world educational settings.

10 Evaluation Metrics

To comprehensively evaluate our AES system, we employ a range of metrics that assess both the accuracy of our predictions and the quality of our uncertainty estimates. These metrics provide insights into different aspects of the model's performance, allowing for a thorough assessment of its capabilities.

10.1 Accuracy Metrics

1. **Accuracy:** The proportion of essays for which the predicted score exactly matches the true score. This provides a straightforward measure of the model's performance.
2. **QWK:** This metric measures the agreement between the predicted and true scores, taking

into account that some disagreements (e.g., predicting a score of 3 when the true score is 4) are less severe than others (e.g., predicting a score of 2 when the true score is 12). QWK is particularly suitable for ordinal classification tasks like essay scoring.

3. **F1 Score:** We calculate the weighted average F1 score across all score categories. This metric balances precision and recall, providing a more nuanced view of performance than accuracy alone.

10.2 Uncertainty Estimation Metrics

1. **Coverage:** The proportion of essays for which the true score falls within the prediction set generated by Conformal Prediction. This metric assesses how well our uncertainty estimates capture the true scores.
2. **Average Set Size:** The mean number of scores included in the prediction sets. This metric helps evaluate the specificity of our predictions, with smaller set sizes indicating more precise predictions.
3. **UAcc:** This novel metric combines accuracy and set size, calculated as:

$$\text{UAcc} = \frac{1}{N}\sum_{i=1}^{N}\left(I(y_i \in C_i)\cdot\frac{\sqrt{|Y|}}{|C_i|}\right)$$

   where N is the number of samples, y_i is the true label, C_i is the prediction set, $|Y|$ is the number of possible classes, and $|C_i|$ is the size of the prediction set. UAcc balances the trade-off between accuracy and the specificity of predictions.
4. **Model Disagreement:** For ensemble methods, we quantify the extent to which different models in the ensemble disagree on their predictions.

By employing this comprehensive set of evaluation metrics, we can assess our AES system's performance from multiple perspectives. This approach allows us to not only evaluate the accuracy of our predictions but also the quality and reliability of our uncertainty estimates, providing a holistic view of the system's capabilities and limitations.

11 Results & Evaluation

11.1 Experimental Setup

Our experiments were conducted on Essay set 1 from the ASAP dataset, consisting of essays with scores ranging from 2 to 12. We evaluated BERT and LLAMA models using various uncertainty quantification methods.

- **Dataset:** ASAP dataset, Prompt 1
- **Score Range:** 2-12
- **Models:** BERT-base-uncased, LLAMA-2-7b
- **Uncertainty Quantification Methods:** Conformal Prediction, Monte Carlo Dropout, Ensemble

Data Split:

- Training set: 70%
- Validation set: 15%
- Test set: 15%

Key Hyperparameters:

- **BERT:** Batch size 16, max sequence length 512, learning rate 1e-5, 10 epochs
- **LLAMA:** 4-bit quantization, LoRA (r=8, alpha=32), batch size 4, learning rate 1e-5, 10 epochs

11.2 Performance Overview

The results demonstrate that LLAMA consistently outperforms BERT across all uncertainty quantification methods. Conformal Prediction generally achieves the best results for both models, particularly when combined with LLAMA.

Table 1: Comparison of AES Methods

Method	Model	QWK	Acc.	F1	Cov.	Avg Set	UAcc	MD
Conformal Prediction	BERT	0.6819	0.4814	0.4454	0.8908	3.2073	1.0501	-
	LLAMA 2	0.8231	0.5410	0.5224	0.9104	2.7388	1.1913	-
	LLAMA 3	0.8026	0.5410	0.5131	0.9104	2.8060	1.2182	-
Monte Carlo Dropout	BERT	0.6958	0.4903	0.4651	0.8955	2.8731	1.0890	-
	LLAMA 2	0.8144	0.5485	0.5222	0.9254	2.8358	1.1777	-
Ensemble	BERT	0.6973	0.4684	0.4380	0.8824	3.1989	1.0664	0.1906
	LLAMA 2	0.7578	0.4812	0.4318	0.9229	3.0050	1.1094	0.0868
	LLAMA 3	0.7631	0.5108	0.4600	0.9403	3.0672	1.1274	0.0811

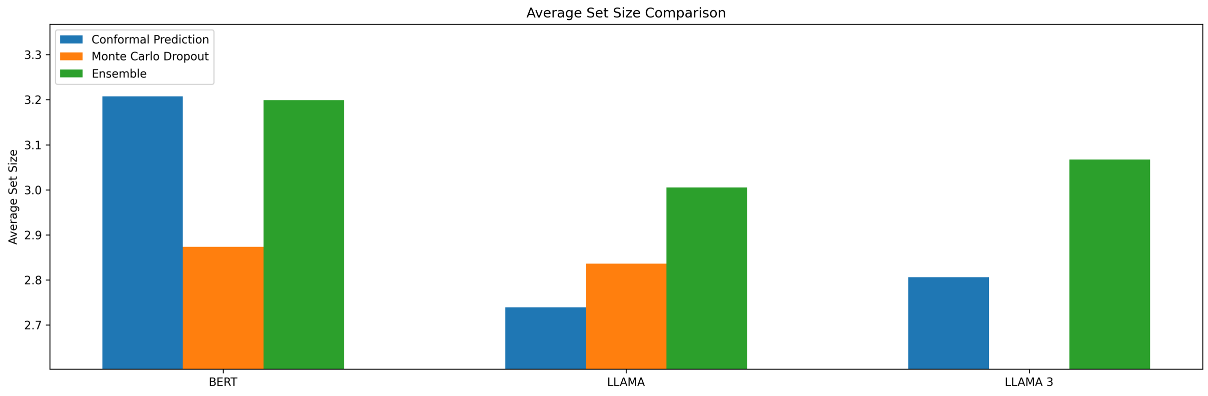


Figure 2: Average Set Size Comparison

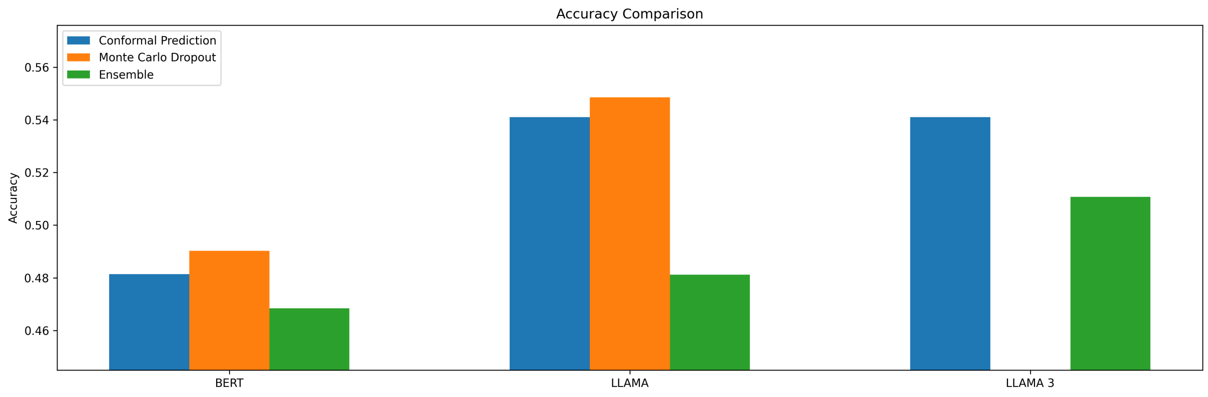


Figure 4: Accuracy Comparison

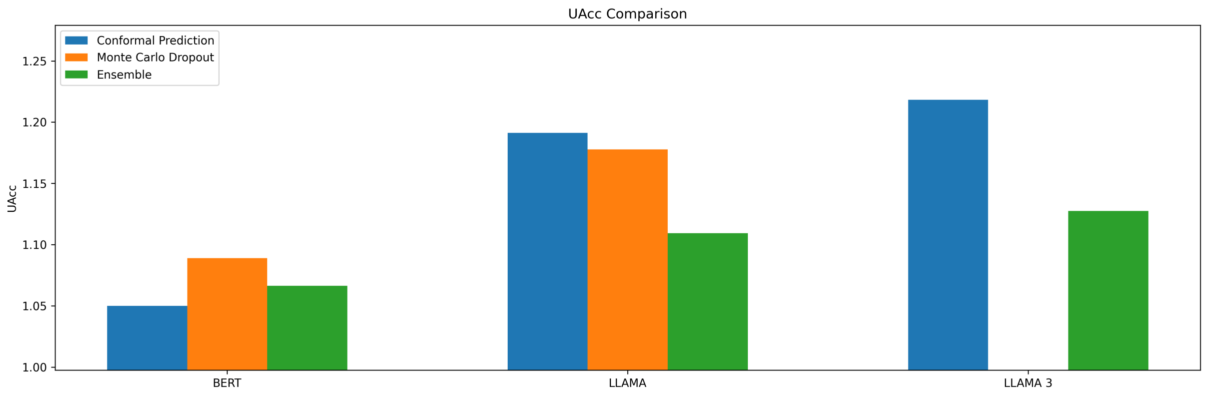


Figure 5: UAcc Comparison

12 Key Findings

12.1 Model Comparison: BERT vs LLAMA

The comparative analysis of BERT and LLAMA models reveals a clear superiority of LLAMA across all uncertainty quantification methods employed in this study. LLAMA consistently outperformed BERT, particularly in terms of QWK and UAcc. The most striking difference is observed in the QWK scores, where LLAMA achieved a maximum of 0.8231 with Conformal Prediction, significantly surpassing BERT's best performance of 0.6973 with the Ensemble method. This substantial improvement in QWK indicates that LLAMA's predictions align more closely with human graders, a crucial factor in AES systems.

In terms of accuracy, LLAMA again demonstrated superior performance, with its peak accuracy of 0.5485 (achieved with Monte Carlo Dropout) notably higher than BERT's maximum of 0.4903. This consistent improvement in accuracy across different uncertainty quantification methods underscores LLAMA's enhanced capability in correctly assessing essay quality.

Perhaps most importantly for this study, LLAMA exhibited better uncertainty quantification capabilities. It consistently maintained higher coverage rates while producing smaller prediction sets, as evidenced by its higher UAcc scores. This indicates that LLAMA not only provides more accurate predictions but also offers more precise and reliable uncertainty estimates, a critical feature for trustworthy AES systems.

12.2 Uncertainty Quantification Methods

Conformal Prediction emerged as the standout method in our study, particularly when combined with the LLAMA model. It achieved the best overall performance, with the highest QWK (0.8231) and UAcc (1.1913) scores for LLAMA. Conformal Prediction demonstrated a remarkable ability to balance coverage and set size for both BERT and LLAMA models. Notably, it maintained consistent coverage rates close to the target of 90% (89.08% for BERT and 91.04% for LLAMA), indicating reliable uncertainty estimates across different model architectures.

Monte Carlo Dropout showed competitive performance, albeit slightly lower than Conformal Prediction. It excelled particularly with the LLAMA model, achieving the highest accuracy (0.5485) and coverage (0.9254) among all methods. Monte Carlo Dropout tended to produce more confident predictions with smaller set sizes, especially for BERT. This method offers an attractive balance between performance and computational efficiency, as it doesn't require training multiple models like the ensemble approach.

The Ensemble method, while providing rich uncertainty information, showed some interesting characteristics. The LLAMA ensemble demonstrated lower average prediction variance and model disagreement (both 0.0868) compared to the BERT ensemble (0.1906), suggesting more consistent predictions across LLAMA models. However, it's important to note that this comparison is not entirely equivalent, as we trained 3 LLAMA models compared to 5 BERT models in the ensemble. Ensembles tended to have higher coverage but at the cost of larger prediction sets, indicating more conservative predictions. This trade-off resulted in slightly lower UAcc scores compared to other methods, particularly for LLAMA.

12.3 Trade-offs and Considerations

Our analysis revealed several important trade-offs and considerations in implementing these uncertainty quantification methods. Firstly, we observed that methods with higher accuracy don't always provide the best uncertainty estimates. For instance, while LLAMA with Monte Carlo Dropout achieved the highest accuracy, it showed slightly lower UAcc compared to Conformal Prediction, indicating a trade-off between pure predictive performance and uncertainty quantification.

A clear trade-off between coverage and average set size was evident across all methods. Ensemble methods, in particular, tended to have higher coverage but at the cost of larger prediction sets. This suggests that while ensembles may be more conservative in their predictions, potentially missing fewer true scores, they also provide less precise uncertainty estimates.

Computational requirements varied significantly across methods and models. LLAMA models, while more accurate, demand substantially more computational resources than BERT. Monte Carlo Dropout, though efficient in not requiring multiple models, necessitates multiple forward passes during inference, increasing inference time. Ensemble methods, while offering rich uncertainty information, require training and maintaining multiple models, which can be resource-intensive.

12.4 Performance Trends

Both BERT and LLAMA models exhibited consistent improvement in performance metrics over training epochs, with LLAMA showing more substantial progress. For instance, LLAMA's QWK increased dramatically from 0.6163 to 0.8231 over 10 epochs with Conformal Prediction, highlighting its rapid learning capability.

Interestingly, coverage rates remained relatively stable across epochs for all methods, indicating consistent uncertainty estimation throughout the training process. This stability in coverage, coupled with the general decrease in average set sizes over epochs, suggests that the models became more confident and precise in their predictions as training progressed, without compromising the reliability of their uncertainty estimates.

These performance trends underscore the potential for further improvements with extended training, particularly for the LLAMA model, and highlight the robustness of the uncertainty quantification methods employed in maintaining consistent coverage despite increasing model confidence.

13 Analysis and discussion

13.1 Comparative Analysis of LLAMA and BERT: Architectural Advantages in AES

The consistent outperformance of LLAMA over BERT in our AES tasks can be attributed to several key factors. Primarily, LLAMA's superior performance stems from its significantly larger model size and more advanced architecture. With 7 billion parameters compared to BERT's 110 million, LLAMA possesses a greater capacity to capture complex language patterns and nuances crucial for evaluating essay quality. This expanded parameter space, coupled with LLAMA's more recent architectural innovations, enables a broader and deeper understanding of language and context, essential elements in assessing the multifaceted aspects of essay writing.

The method of pre-training that LLAMA uses additionally allows it to score essays so effectively. LLAMA, being trained on a larger and more general dataset that includes the entire internet (in text form) as well as books and articles in addition to Wikipedia pages, simply knows much more about everything. It is also what makes likely to so well at a diverse range of writing and topics, critical for an essay grader. LLAMA also benefits from being trained on longer sequences (a limitation of BERT) which is compatible with the use-case of analysing lengthy text such as essays.

In the specific context of AES, LLAMA's advantages become particularly pronounced. The task demands an understanding of complex arguments, coherence, and stylistic elements – areas where LLAMA's enhanced capabilities shine. This is evident in the consistently higher QWK scores achieved by LLAMA across all uncertainty quantification methods in our study. The improved accuracy and F1 scores further suggest that LLAMA is more adept at distinguishing between different levels of essay quality, an important aspect of effective automated scoring.

LLAMA's superiority extends to the realm of uncertainty quantification, a vital part in building trustworthy AI systems for educational assessment. Our results show that LLAMA consistently achieves better uncertainty metrics, including higher coverage rates and more precise prediction sets. The higher UAcc scores across all methods indicate that LLAMA strikes a better balance between accuracy and uncertainty quantification. This enhanced ability to provide reliable uncertainty estimates is particularly valuable in educational contexts, where understanding the confidence of automated assessments is crucial for fair and effective evaluation.

However, it's important to note that LLAMA's improved performance comes at the cost of significantly increased computational requirements. The substantial difference in model sizes between LLAMA and BERT translates to higher resource demands for LLAMA, which could be a limiting factor in some practical applications. This trade-off between performance and computational efficiency is a critical consideration for the real-world implementation of these models in educational technology.

In conclusion, LLAMA's superior performance in AES can be attributed to its larger size, more advanced architecture, diverse pre-training data, and better adaptation to the task of evaluating long-form text. These advantages allow LLAMA to capture the complex aspects of essay quality more effectively, leading to better agreement with human graders and more reliable uncertainty estimates. As the field of AI in education continues to evolve, the balance between model performance and computational efficiency will remain a key area of focus, driving further innovations in automated assessment technologies.

13.2 Paradigm Shift in AES: LLAMA's Prompting Approach vs. BERT's Fine-tuning

The superior performance of LLAMA over BERT in our AES tasks can be attributed to their fundamentally different approaches to processing text. While both are LLMs, LLAMA's use of prompting is different from BERT's methodology, leading to notable advantages in the context of essay evaluation.

BERT, as a bidirectional transformer model, processes text by encoding it into contextual embeddings. It is typically fine-tuned for specific tasks by adding task-specific layers on top of these embeddings. In our essay scoring setup, BERT was fine-tuned to map these embeddings directly to essay scores. This approach, while effective for many NLP tasks, has limitations when dealing with the complexities of essay evaluation.

LLAMA, on the other hand, utilizes a prompting paradigm, which fundamentally changes how the model interacts with the task. In our implementation, we formulated the essay scoring task as a prompt, instructing the model to "Score the following essay on a scale of 2 to 12." This prompting approach offers several advantages:

1. **Task Flexibility:** By framing the task as a prompt, LLAMA can adapt more readily to different scoring criteria or essay types without requiring extensive retraining. This flexibility is particularly valuable in educational contexts where assessment criteria may vary.
2. **Reasoning Transparency:** The prompting approach allows LLAMA to generate explanations or reasoning alongside its scores. While not directly measured in our quantitative results, this capability offers potential for more interpretable and justifiable scoring, a crucial factor in educational applications.
3. **Contextual Understanding:** Prompts can include additional context, such as specific grading rubrics or examples. This allows LLAMA to tailor its evaluation more precisely to given criteria, potentially leading to more accurate and consistent scoring.
4. **Natural Language Interaction:** The prompting paradigm enables a more natural language interaction with the model. This aligns more closely with how human graders might approach the task, potentially leading to more nuanced evaluations.
5. **Few-Shot Learning:** Although not explored in our current study, LLAMA's prompting approach opens the door to few-shot learning scenarios. This could allow the model to quickly adapt to new essay types or scoring criteria with minimal additional training.

These differences in processing methodology have significant implications for AES. LLAMA's prompting approach allows it to engage with the essay scoring task in a more flexible and context-aware manner. This likely contributes to its superior performance across our evaluation metrics, including higher QWK scores and more reliable uncertainty estimates.

Moreover, the prompting paradigm aligns well with the nature of essay evaluation, which often requires understanding nuanced instructions and applying complex, multi-faceted criteria. LLAMA's ability to interpret and act on these prompts mirrors the way human graders might approach an essay, potentially leading to more human-like assessments.

However, it's important to note that this prompting approach also introduces challenges, particularly in ensuring consistency across different prompt formulations. The performance of LLAMA could be sensitive to the exact wording of the prompt, a factor that would need careful consideration in real-world applications.

In conclusion, while LLAMA's larger size and more advanced architecture contribute to its superior performance, its use of prompting represents a fundamental shift in how the model approaches the essay scoring task. This approach offers greater flexibility, potential for more nuanced evaluations, and aligns more closely with human-like reasoning processes. As we continue to develop and refine AES systems, the prompting paradigm exemplified by LLAMA presents exciting opportunities for more adaptive, context-aware, and potentially more reliable automated assessment tools in education.

13.3 Uncertainty Quantification in AES: A Critical Analysis

The integration of uncertainty quantification in AES systems represents a significant advancement in enhancing the reliability and trustworthiness of AI-based educational assessments. Our study explored three primary approaches: conformal prediction, Monte Carlo dropout, and ensemble methods, with conformal prediction emerging as a particularly effective technique.

Conformal prediction demonstrated remarkable efficacy in quantifying uncertainty for both BERT and LLAMA models. Its mathematically rigorous framework consistently achieved coverage rates at or above our 90% target, a crucial factor in educational contexts where reliable score bounds are paramount. This method's ability to provide guaranteed coverage sets it apart, offering a level of assurance that is especially valuable in high stakes testing scenarios.

The model-agnostic nature of conformal prediction proved to be a significant advantage. Its successful application to both BERT and LLAMA architectures underscores its versatility and potential for adaptation to future advancements in AES model design. This flexibility is particularly valuable in a rapidly evolving field where new model architectures are continually emerging.

One of the most compelling aspects of conformal prediction is its alignment with the inherently subjective nature of essay scoring. By outputting a set of possible scores rather than a single estimate, it captures the nuanced reality of essay evaluation. This approach resonates with the variability often seen in human grading and provides a more intuitive framework for educators and students to interpret automated scores.

When compared to other uncertainty quantification methods, conformal prediction emerged as the most informative and versatile approach in our study of AES. It provided a rich set of metrics that offered deep insights into the uncertainty associated with each predicted score. Unlike Monte Carlo dropout, which primarily relies on variance in multiple forward passes, or ensemble methods that depend on inter-model disagreement, conformal prediction offered a more nuanced view of uncertainty. It not only provided prediction sets with guaranteed coverage but also allowed for the analysis of set sizes, which directly correlate with the model's confidence in its predictions. This dual approach of coverage and set size enabled a more comprehensive understanding of the model's certainty across different types of essays and scoring ranges.

The implications of these findings for AES are profound. The adaptive uncertainty provided by conformal prediction, where prediction set sizes vary based on the model's certainty, aligns well with the varying difficulty in scoring different types of essays. This adaptability not only enhances the system's reliability but also offers potential for more nuanced human-AI collaboration in essay grading.

However, the implementation of conformal prediction in AES is not without challenges. The computational overhead required for calibration and inference could pose scaling issues in large-scale applications. Additionally, the method's sensitivity to dataset shifts and the discrete nature of essay scores present ongoing challenges that warrant further research.

In conclusion, our analysis reveals conformal prediction as a powerful tool for uncertainty quantification in AES. Its strong performance, particularly when combined with advanced language models like LLAMA, offers a promising path towards more reliable, transparent, and trustworthy AI-based educational assessment tools. As the field of AES continues to evolve, the integration of robust uncertainty quantification methods like conformal prediction will be crucial in addressing the ongoing challenges of fairness, reliability, and interpretability in automated assessment. This approach not only enhances the technical capabilities of AES systems but also aligns closely with the nuanced and often subjective nature of essay evaluation, potentially bridging the gap between automated efficiency and human-like assessment in educational contexts.

14 Future Directions in Uncertainty-Aware AES

The promising results of our study, particularly in the application of conformal prediction to AES , open up several exciting avenues for future research and development. These directions aim to address current limitations, expand the capabilities of uncertainty-aware AES systems, and further bridge the gap between automated and human-like assessment in educational contexts.

14.1 Adaptive Conformal Prediction for Essay Scoring

A key area for future exploration is the development of adaptive conformal prediction methods tailored specifically for essay scoring. Current implementations use a fixed significance level across all predictions. Future research could focus on dynamically adjusting the conformal prediction process based on individual essay characteristics, such as length, complexity, or topic. This approach could lead to more precise uncertainty estimates, potentially improving the balance between coverage and set size for diverse essay types.

14.2 Integrating Interpretability with Uncertainty Quantification

While conformal prediction enhances the reliability of AES systems, it does not inherently provide explanations for its predictions. Future work should explore the integration of conformal prediction with interpretable AI techniques. This could involve developing methods to generate human-readable justifications for predicted score ranges, potentially utilizing techniques like attention visualization or rule extraction. Such advancements would not only improve the interpretability of AES systems but also provide valuable feedback to students and educators on specific areas of strength or improvement in essays.

14.3 Ethical Considerations and Fairness

As AES systems become more advanced, it's crucial to continually examine their ethical implications and fairness. Future research should focus on developing methods to detect and mitigate biases in uncertainty-aware AES systems, ensuring they provide equitable assessments across diverse student populations. This could involve creating new evaluation metrics that incorporate both performance and fairness considerations in the context of uncertainty quantification.

15 Conclusion

This research project marks a significant milestone in the field of Automated Essay Scoring by pioneering the application of uncertainty quantification methods to Large Language Models in an educational assessment context. By comparing the performance of BERT and LLAMA models across various uncertainty estimation techniques, we have not only demonstrated the superior capabilities of LLAMA in capturing the nuances of essay quality but also opened new avenues for enhancing the reliability and interpretability of AI-based scoring systems.

The implementation of conformal prediction, Monte Carlo dropout, and ensemble methods has revealed valuable insights into the trade-offs between accuracy, coverage, and computational efficiency in AES systems, with conformal prediction emerging as a particularly effective approach. This study lays a solid foundation for future advancements in AES, paving the way for the integration of even more advanced LLMs such as GPT-3 and its successors. The potential application of these techniques to other state-of-the-art language models promises further improvements in performance and adaptability.

As the field continues to evolve, the combination of cutting-edge LLMs with sophisticated uncertainty quantification methods has the potential to revolutionize educational assessment, bridging the gap between automated efficiency and human-like evaluation. This research not only contributes to the development of more trustworthy and interpretable AES tools but also sets a precedent for incorporating uncertainty awareness in AI-driven educational technologies, potentially transforming the landscape of essay evaluation and feedback in diverse educational settings.

16 Legal, Social, Ethical and Professional Issues

This project adhered strictly to the British Computer Society's (BCS) Code of Conduct and Code of Good Practice, ensuring ethical and professional standards were maintained throughout. Given the academic nature of the research, particular attention was paid to the ethical use of third-party materials and the proper attribution of sources. All external libraries and resources utilized in the implementation were explicitly cited, promoting transparency and respecting intellectual property rights.

This approach not only upheld ethical standards but also facilitated efficient development through code reuse without compromising the project's quality or originality. The research was conducted with a commitment to integrity, ensuring that all work presented is the author's own, with appropriate citations for any open-source materials used.

By adhering to these principles, the project maintained high ethical standards, avoiding potential legal issues and upholding the professional expectations set forth by the BCS.

References

Angelopoulos, A. N., & Bates, S. (2021). A gentle introduction to conformal prediction and distribution-free uncertainty quantification. arXiv preprint arXiv:2107.07511.

Balaji, L., Pritzel, A., & Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. Advances in Neural Information Processing Systems, 30.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Gal, Y., & Ghahramani, Z. (2016). Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. International Conference on Machine Learning, 1050-1059.

Phandi, P., Chai, K. M. A., & Ng, H. T. (2015). Flexible domain adaptation for automated essay scoring using correlated linear regression. Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, 431-439.

Sadinle, M., Lei, J., & Wasserman, L. (2019). Least ambiguous set-valued classifiers with bounded error levels. Journal of the American Statistical Association, 114(525), 223-234.

Taghipour, K., & Ng, H. T. (2016). A neural approach to automated essay scoring. Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, 1882-1891.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., Lacroix, T., ... & Lample, G. (2023). LLaMA: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30.

Xiao, C., Ma, W., Song, Q., Xu, S. X., Zhang, K., Wang, Y., & Fu, Q. (2023). Human-AI collaborative essay scoring: A dual-process framework with LLMs. arXiv preprint arXiv:2305.03183.

Yang, R., Cao, J., Wen, Z., Wu, Y., & He, X. (2020). Enhancing automated essay scoring performance via fine-tuning pre-trained language models with combination of regression and ranking. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 1560-1569.

Ye, F., Yang, M., Pang, J., Wang, L., Wong, D. F., Yilmaz, E., ... & Tu, Z. (2023). Benchmarking LLMs via uncertainty quantification. arXiv preprint arXiv:2306.07475.

The Chartered Institute for IT BCS. Bcs code of conduct. https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/, 2021

Source Code and Execution Instructions

The project's source code is organized into several Python scripts and a Jupyter notebook, each handling different aspects of the Automated Essay Scoring system with various uncertainty quantification methods. Here's an overview of the files and instructions on how to run them:

BERT Implementation

- `bert_CP.py`: BERT model with Conformal Prediction
- `BERT_ensemble.py`: BERT Ensemble model
- `BERT_MC.py`: BERT with Monte Carlo Dropout

To run any of the BERT scripts:

1. Ensure you have the required Python libraries installed (`requirements.txt` recommended).
2. Place the 'training_set_rel3.xlsx' file in the same directory as the scripts.
3. Execute the desired script using Python:

   ```
   python bert_CP.py
   ```

   (Replace 'bert_CP.py' with the script you want to run)

LLAMA Implementation

- `LLAMA.ipynb`: Jupyter notebook containing all LLAMA runs (Conformal Prediction, Monte Carlo Dropout, and Ensemble)

To run the LLAMA notebook:

1. Open the `LLAMA.ipynb` file in Jupyter Notebook or JupyterLab.
2. Ensure you have access to the LLAMA models. You'll need to set up your own access token.
3. Replace the existing token in the notebook with your personal access token.
4. Place the 'training_set_rel3.xlsx' file in the same directory as the notebook.
5. Run the cells in the notebook sequentially.

Note: If you wish to use a different LLAMA model, simply replace the model name in the notebook with your preferred LLAMA model variant.

Dataset

- `training_set_rel3.xlsx`: This Excel file contains the ASAP dataset used for training and evaluation. Ensure it's present in the same directory as the scripts/notebook you're running.

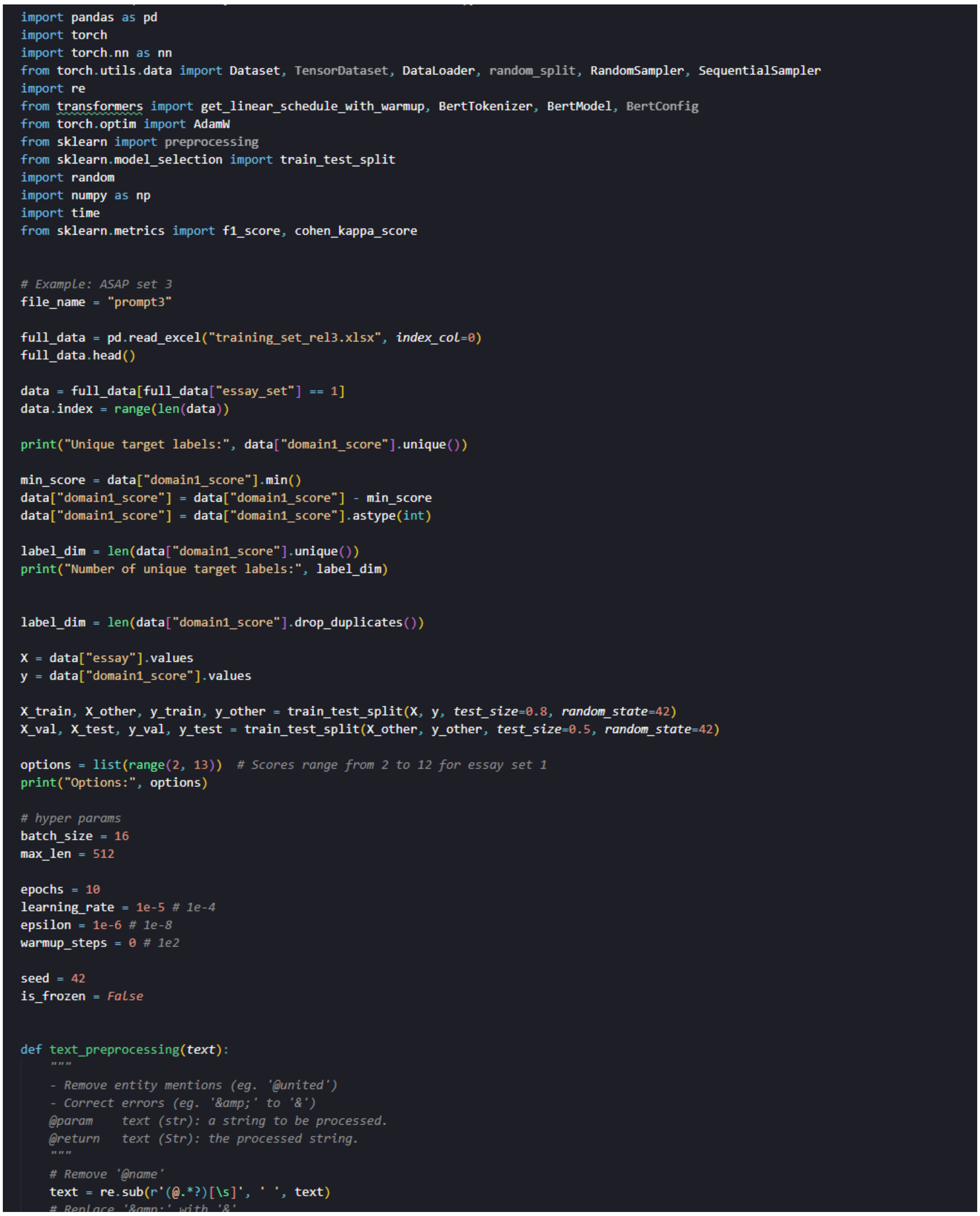
Requirements

- Python 3.7+

- Required libraries (consider providing a `requirements.txt` file)
- Jupyter Notebook (for `LLAMA.ipynb`)
- Access to LLAMA models (for LLAMA implementation)

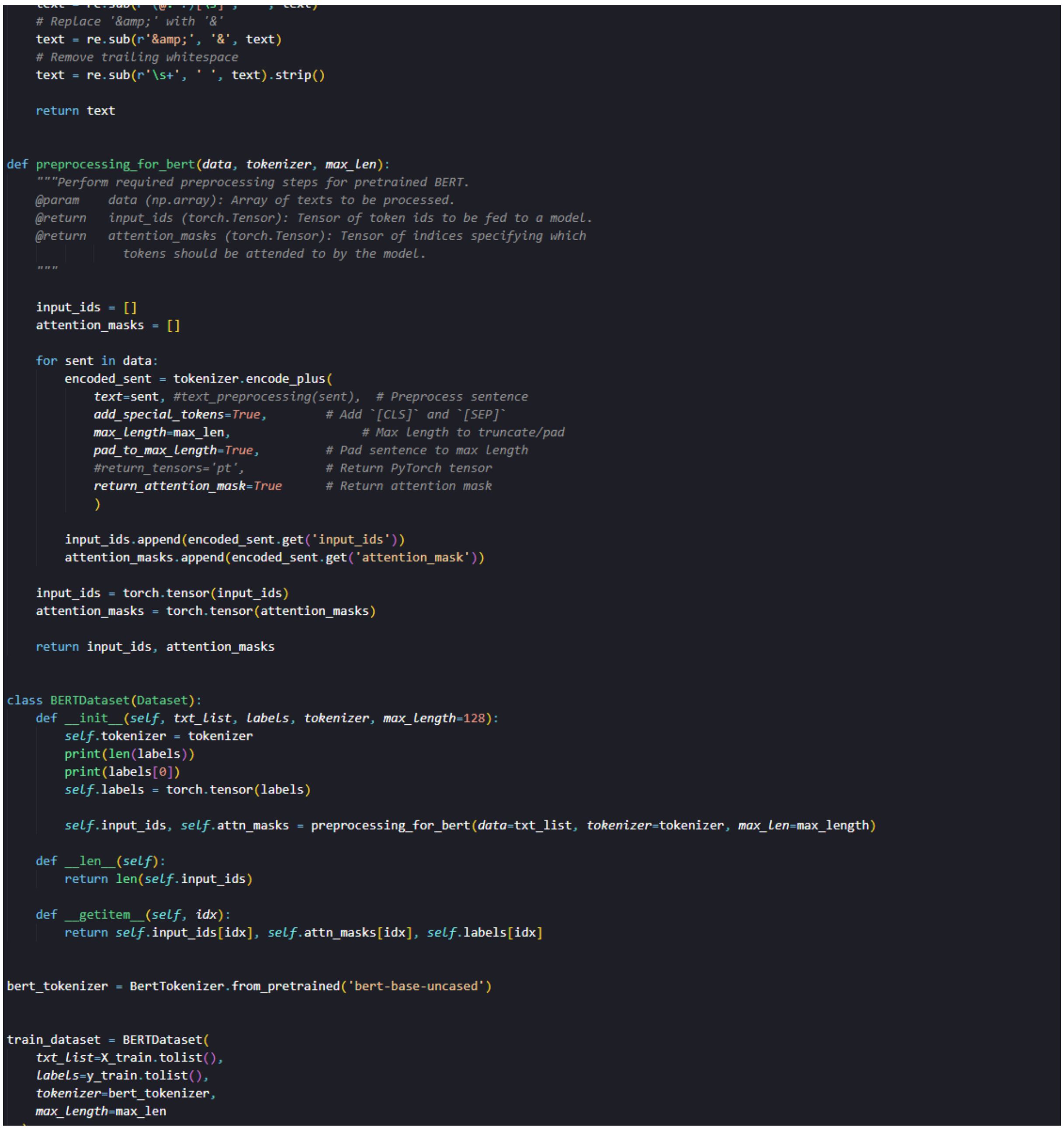
Please ensure all dependencies are installed and the dataset is correctly placed before running the scripts. For any issues or questions, refer to the documentation or contact the project maintainer.

Code Listings



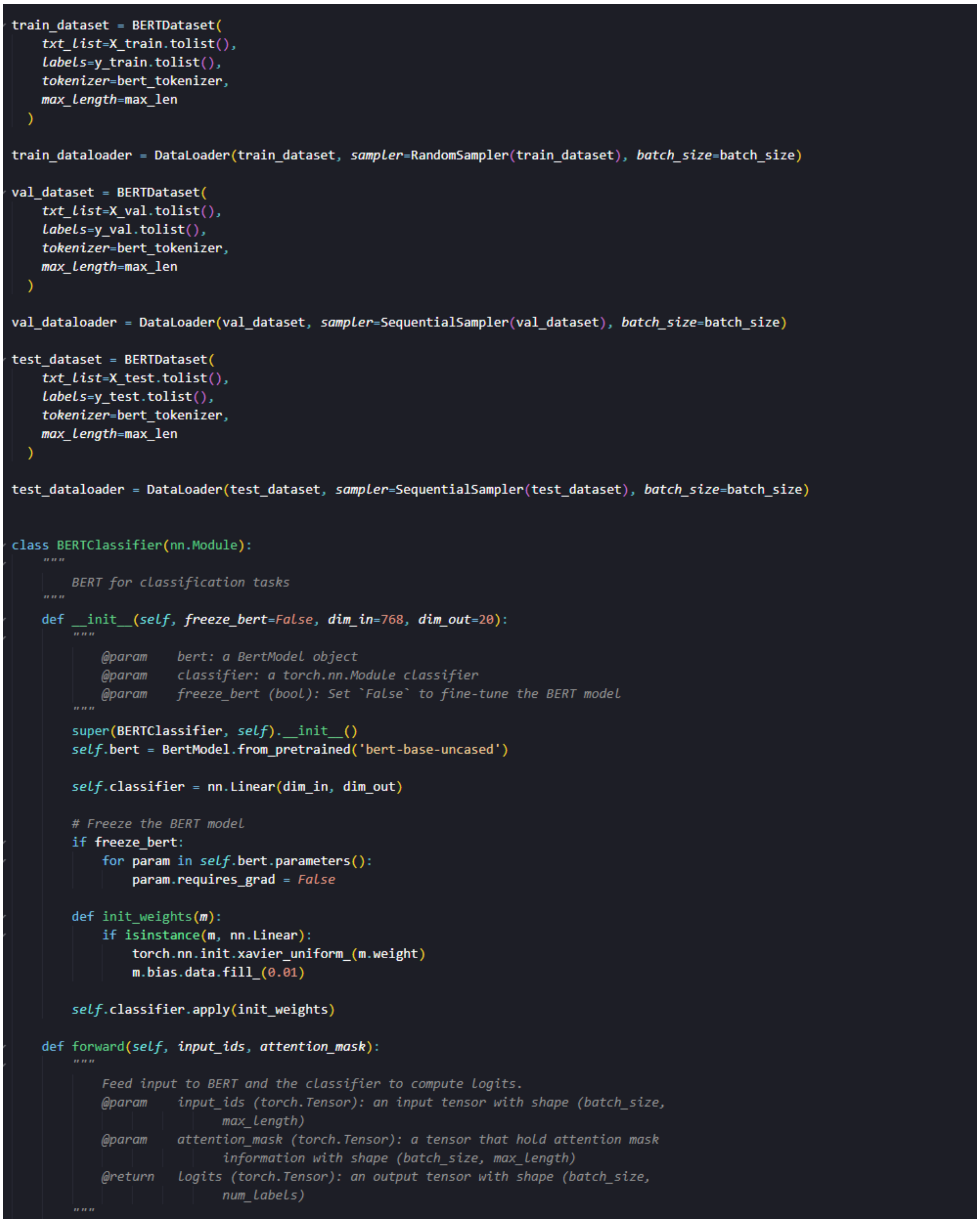
```python
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, TensorDataset, DataLoader, random_split, RandomSampler, SequentialSampler
import re
from transformers import get_linear_schedule_with_warmup, BertTokenizer, BertModel, BertConfig
from torch.optim import AdamW
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import random
import numpy as np
import time
from sklearn.metrics import f1_score, cohen_kappa_score


# Example: ASAP set 3
file_name = "prompt3"

full_data = pd.read_excel("training_set_rel3.xlsx", index_col=0)
full_data.head()

data = full_data[full_data["essay_set"] == 1]
data.index = range(len(data))

print("Unique target labels:", data["domain1_score"].unique())

min_score = data["domain1_score"].min()
data["domain1_score"] = data["domain1_score"] - min_score
data["domain1_score"] = data["domain1_score"].astype(int)

label_dim = len(data["domain1_score"].unique())
print("Number of unique target labels:", label_dim)


label_dim = len(data["domain1_score"].drop_duplicates())

X = data["essay"].values
y = data["domain1_score"].values

X_train, X_other, y_train, y_other = train_test_split(X, y, test_size=0.8, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_other, y_other, test_size=0.5, random_state=42)

options = list(range(2, 13))  # Scores range from 2 to 12 for essay set 1
print("Options:", options)

# hyper params
batch_size = 16
max_len = 512

epochs = 10
learning_rate = 1e-5 # 1e-4
epsilon = 1e-6 # 1e-8
warmup_steps = 0 # 1e2

seed = 42
is_frozen = False


def text_preprocessing(text):
    """
    - Remove entity mentions (eg. '@united')
    - Correct errors (eg. '&amp;' to '&')
    @param    text (str): a string to be processed.
    @return   text (Str): the processed string.
    """
    # Remove '@name'
    text = re.sub(r'(@.*?)[\s]', ' ', text)
    # Replace '&amp;' with '&'
```

Figure 6: Code snippet 1



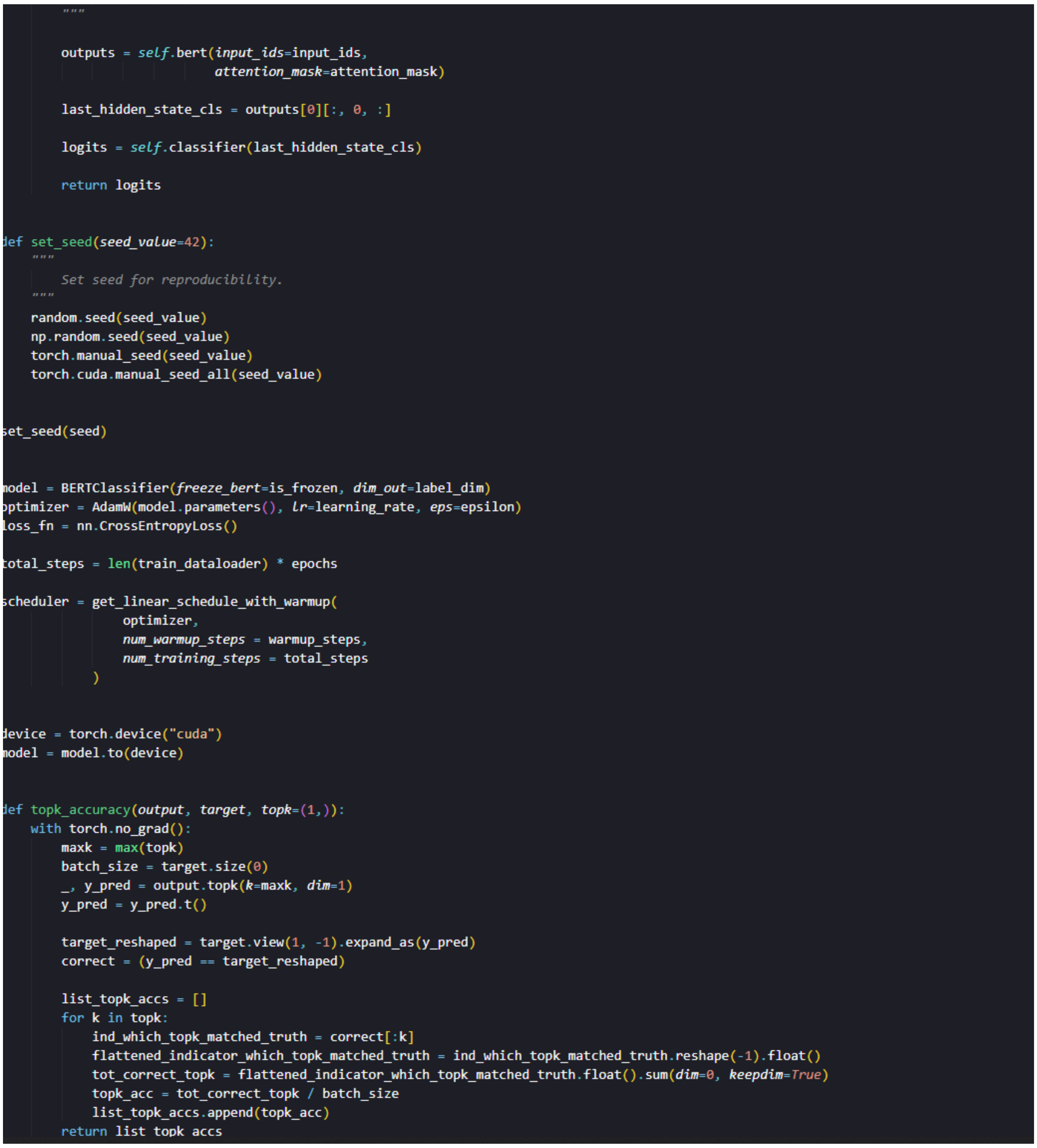
```
    # Replace '&amp;' with '&'
    text = re.sub(r'&amp;', '&', text)
    # Remove trailing whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    return text


def preprocessing_for_bert(data, tokenizer, max_len):
    """Perform required preprocessing steps for pretrained BERT.
    @param    data (np.array): Array of texts to be processed.
    @return   input_ids (torch.Tensor): Tensor of token ids to be fed to a model.
    @return   attention_masks (torch.Tensor): Tensor of indices specifying which
                tokens should be attended to by the model.
    """

    input_ids = []
    attention_masks = []

    for sent in data:
        encoded_sent = tokenizer.encode_plus(
            text=sent, #text_preprocessing(sent),  # Preprocess sentence
            add_special_tokens=True,        # Add `[CLS]` and `[SEP]`
            max_length=max_len,                  # Max length to truncate/pad
            pad_to_max_length=True,         # Pad sentence to max length
            #return_tensors='pt',           # Return PyTorch tensor
            return_attention_mask=True      # Return attention mask
            )

        input_ids.append(encoded_sent.get('input_ids'))
        attention_masks.append(encoded_sent.get('attention_mask'))

    input_ids = torch.tensor(input_ids)
    attention_masks = torch.tensor(attention_masks)

    return input_ids, attention_masks


class BERTDataset(Dataset):
    def __init__(self, txt_list, labels, tokenizer, max_length=128):
        self.tokenizer = tokenizer
        print(len(labels))
        print(labels[0])
        self.labels = torch.tensor(labels)

        self.input_ids, self.attn_masks = preprocessing_for_bert(data=txt_list, tokenizer=tokenizer, max_len=max_length)

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, idx):
        return self.input_ids[idx], self.attn_masks[idx], self.labels[idx]


bert_tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')


train_dataset = BERTDataset(
    txt_list=X_train.tolist(),
    labels=y_train.tolist(),
    tokenizer=bert_tokenizer,
    max_length=max_len
```

Figure 7: Code snippet 2



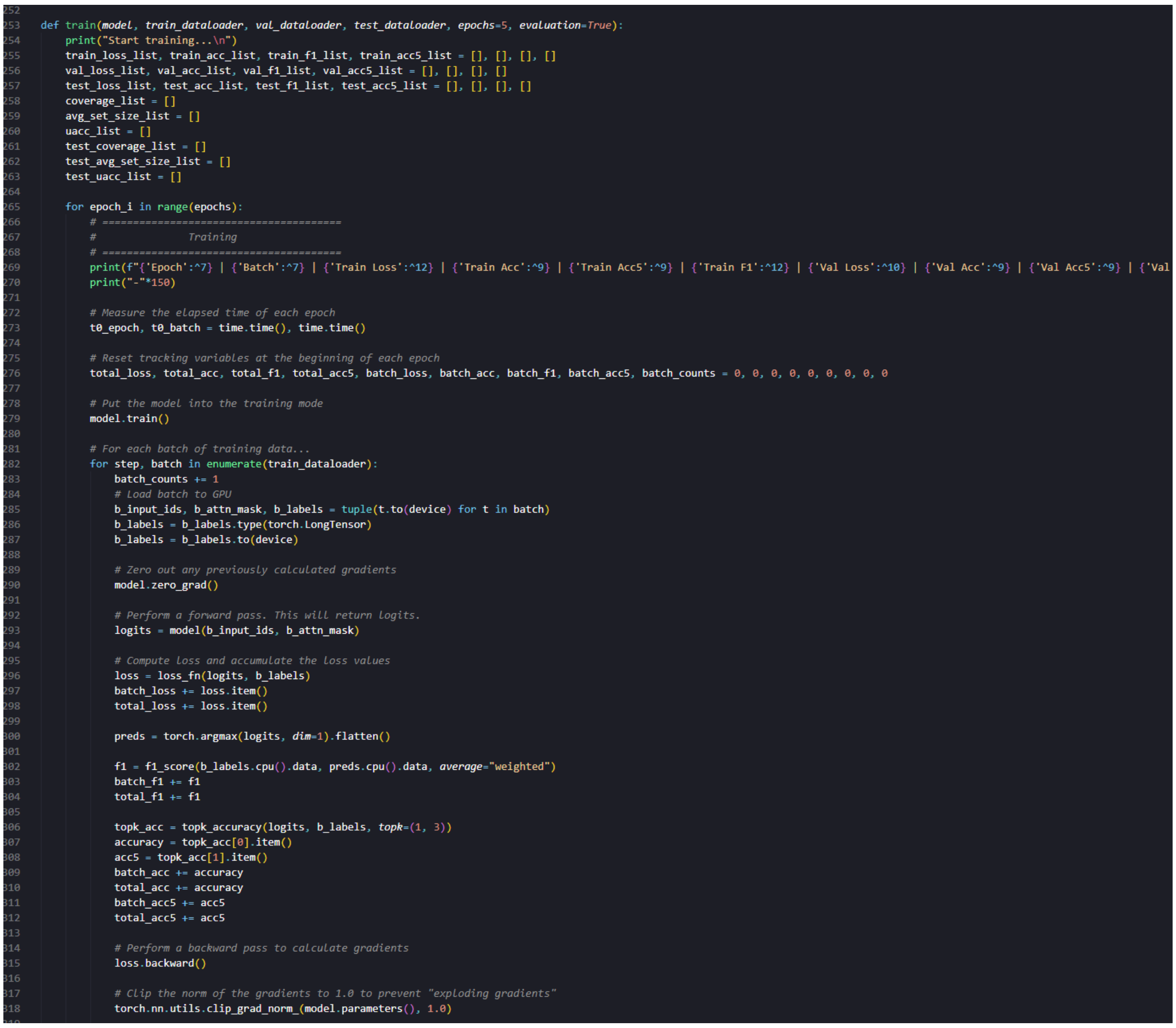
```
train_dataset = BERTDataset(
    txt_list=X_train.tolist(),
    labels=y_train.tolist(),
    tokenizer=bert_tokenizer,
    max_length=max_len
  )

train_dataloader = DataLoader(train_dataset, sampler=RandomSampler(train_dataset), batch_size=batch_size)

val_dataset = BERTDataset(
    txt_list=X_val.tolist(),
    labels=y_val.tolist(),
    tokenizer=bert_tokenizer,
    max_length=max_len
  )

val_dataloader = DataLoader(val_dataset, sampler=SequentialSampler(val_dataset), batch_size=batch_size)

test_dataset = BERTDataset(
    txt_list=X_test.tolist(),
    labels=y_test.tolist(),
    tokenizer=bert_tokenizer,
    max_length=max_len
  )

test_dataloader = DataLoader(test_dataset, sampler=SequentialSampler(test_dataset), batch_size=batch_size)


class BERTClassifier(nn.Module):
    """
        BERT for classification tasks
    """
    def __init__(self, freeze_bert=False, dim_in=768, dim_out=20):
        """
            @param    bert: a BertModel object
            @param    classifier: a torch.nn.Module classifier
            @param    freeze_bert (bool): Set `False` to fine-tune the BERT model
        """
        super(BERTClassifier, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased')

        self.classifier = nn.Linear(dim_in, dim_out)

        # Freeze the BERT model
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

        def init_weights(m):
            if isinstance(m, nn.Linear):
                torch.nn.init.xavier_uniform_(m.weight)
                m.bias.data.fill_(0.01)

        self.classifier.apply(init_weights)

    def forward(self, input_ids, attention_mask):
        """
            Feed input to BERT and the classifier to compute logits.
            @param    input_ids (torch.Tensor): an input tensor with shape (batch_size,
                          max_length)
            @param    attention_mask (torch.Tensor): a tensor that hold attention mask
                          information with shape (batch_size, max_length)
            @return   logits (torch.Tensor): an output tensor with shape (batch_size,
                          num_labels)
        """
```

Figure 8: Code snippet 3



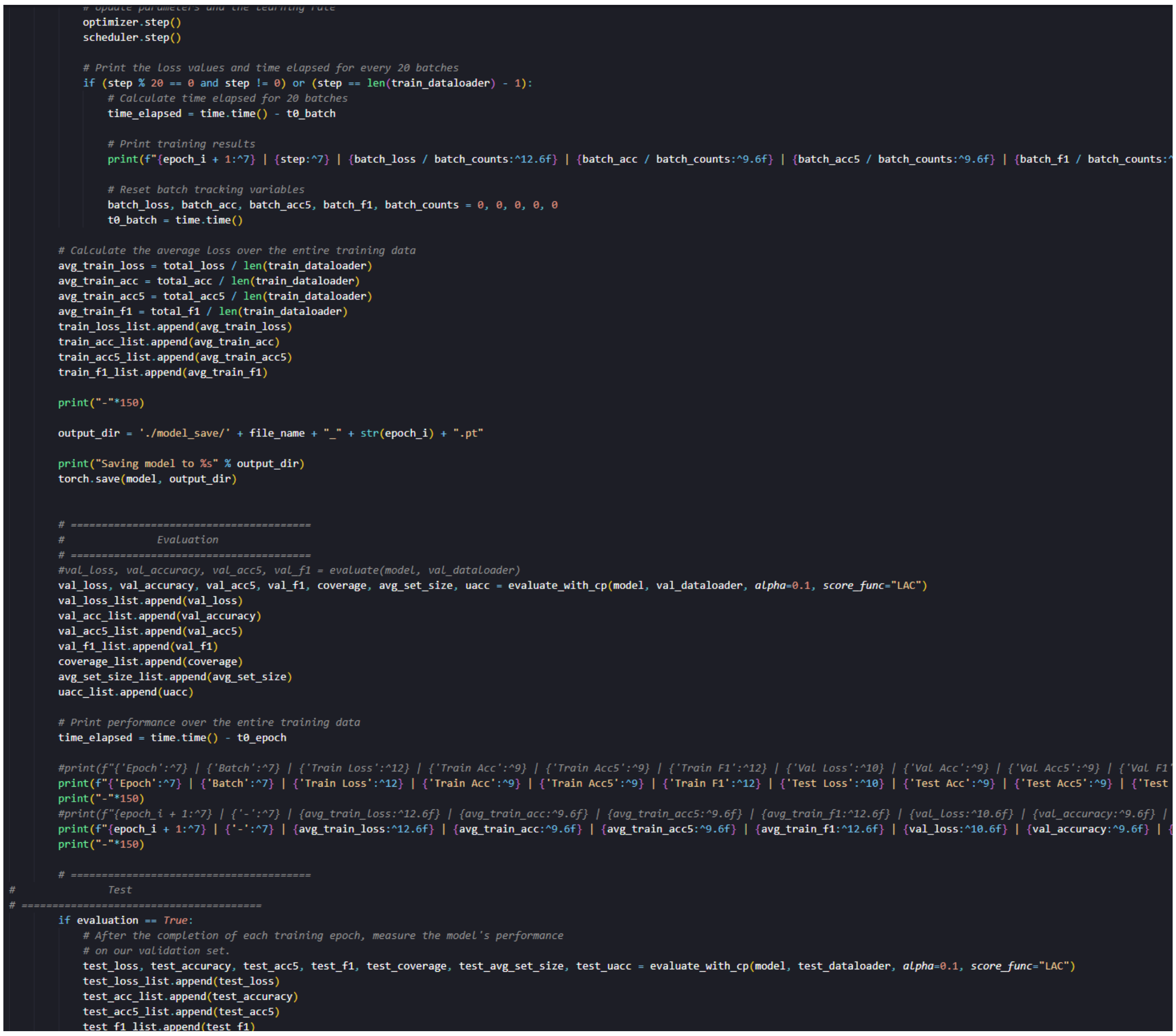
```
        """
        outputs = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)

        last_hidden_state_cls = outputs[0][:, 0, :]

        logits = self.classifier(last_hidden_state_cls)

        return logits


def set_seed(seed_value=42):
    """
        Set seed for reproducibility.
    """
    random.seed(seed_value)
    np.random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)


set_seed(seed)


model = BERTClassifier(freeze_bert=is_frozen, dim_out=label_dim)
optimizer = AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
loss_fn = nn.CrossEntropyLoss()

total_steps = len(train_dataloader) * epochs

scheduler = get_linear_schedule_with_warmup(
                optimizer,
                num_warmup_steps = warmup_steps,
                num_training_steps = total_steps
            )


device = torch.device("cuda")
model = model.to(device)


def topk_accuracy(output, target, topk=(1,)):
    with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)
        _, y_pred = output.topk(k=maxk, dim=1)
        y_pred = y_pred.t()

        target_reshaped = target.view(1, -1).expand_as(y_pred)
        correct = (y_pred == target_reshaped)

        list_topk_accs = []
        for k in topk:
            ind_which_topk_matched_truth = correct[:k]
            flattened_indicator_which_topk_matched_truth = ind_which_topk_matched_truth.reshape(-1).float()
            tot_correct_topk = flattened_indicator_which_topk_matched_truth.float().sum(dim=0, keepdim=True)
            topk_acc = tot_correct_topk / batch_size
            list_topk_accs.append(topk_acc)
        return list_topk_accs
```

Figure 9: Code snippet 4



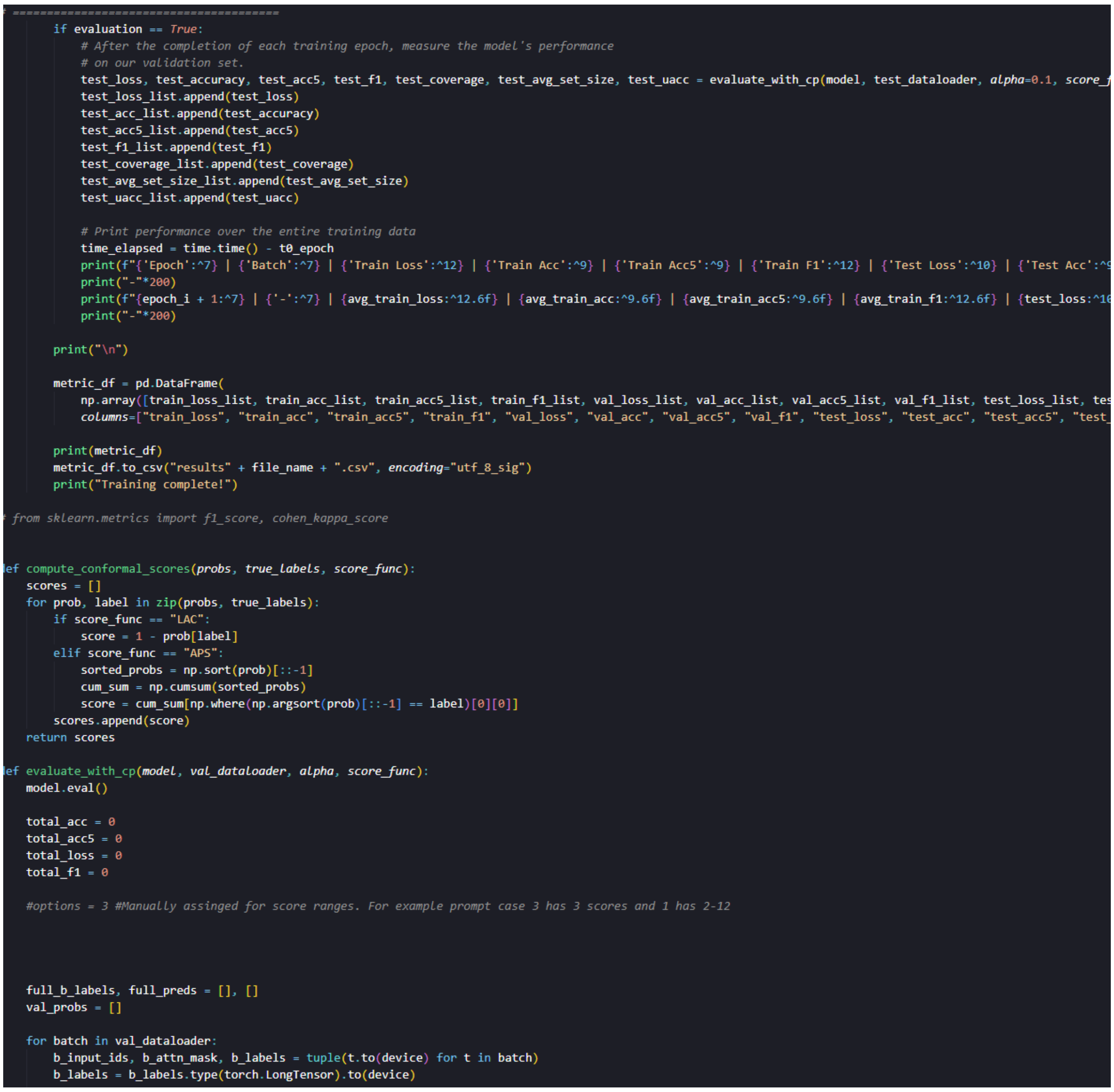
```python

def train(model, train_dataloader, val_dataloader, test_dataloader, epochs=5, evaluation=True):
    print("Start training...\n")
    train_loss_list, train_acc_list, train_f1_list, train_acc5_list = [], [], [], []
    val_loss_list, val_acc_list, val_f1_list, val_acc5_list = [], [], [], []
    test_loss_list, test_acc_list, test_f1_list, test_acc5_list = [], [], [], []
    coverage_list = []
    avg_set_size_list = []
    uacc_list = []
    test_coverage_list = []
    test_avg_set_size_list = []
    test_uacc_list = []

    for epoch_i in range(epochs):
        # =======================================
        #               Training
        # =======================================
        print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Val Loss':^10} | {'Val Acc':^9} | {'Val Acc5':^9} | {'Val
        print("-"*150)

        # Measure the elapsed time of each epoch
        t0_epoch, t0_batch = time.time(), time.time()

        # Reset tracking variables at the beginning of each epoch
        total_loss, total_acc, total_f1, total_acc5, batch_loss, batch_acc, batch_f1, batch_acc5, batch_counts = 0, 0, 0, 0, 0, 0, 0, 0, 0

        # Put the model into the training mode
        model.train()

        # For each batch of training data...
        for step, batch in enumerate(train_dataloader):
            batch_counts += 1
            # Load batch to GPU
            b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
            b_labels = b_labels.type(torch.LongTensor)
            b_labels = b_labels.to(device)

            # Zero out any previously calculated gradients
            model.zero_grad()

            # Perform a forward pass. This will return logits.
            logits = model(b_input_ids, b_attn_mask)

            # Compute loss and accumulate the loss values
            loss = loss_fn(logits, b_labels)
            batch_loss += loss.item()
            total_loss += loss.item()

            preds = torch.argmax(logits, dim=1).flatten()

            f1 = f1_score(b_labels.cpu().data, preds.cpu().data, average="weighted")
            batch_f1 += f1
            total_f1 += f1

            topk_acc = topk_accuracy(logits, b_labels, topk=(1, 3))
            accuracy = topk_acc[0].item()
            acc5 = topk_acc[1].item()
            batch_acc += accuracy
            total_acc += accuracy
            batch_acc5 += acc5
            total_acc5 += acc5

            # Perform a backward pass to calculate gradients
            loss.backward()

            # Clip the norm of the gradients to 1.0 to prevent "exploding gradients"
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

Figure 10: Code snippet 5



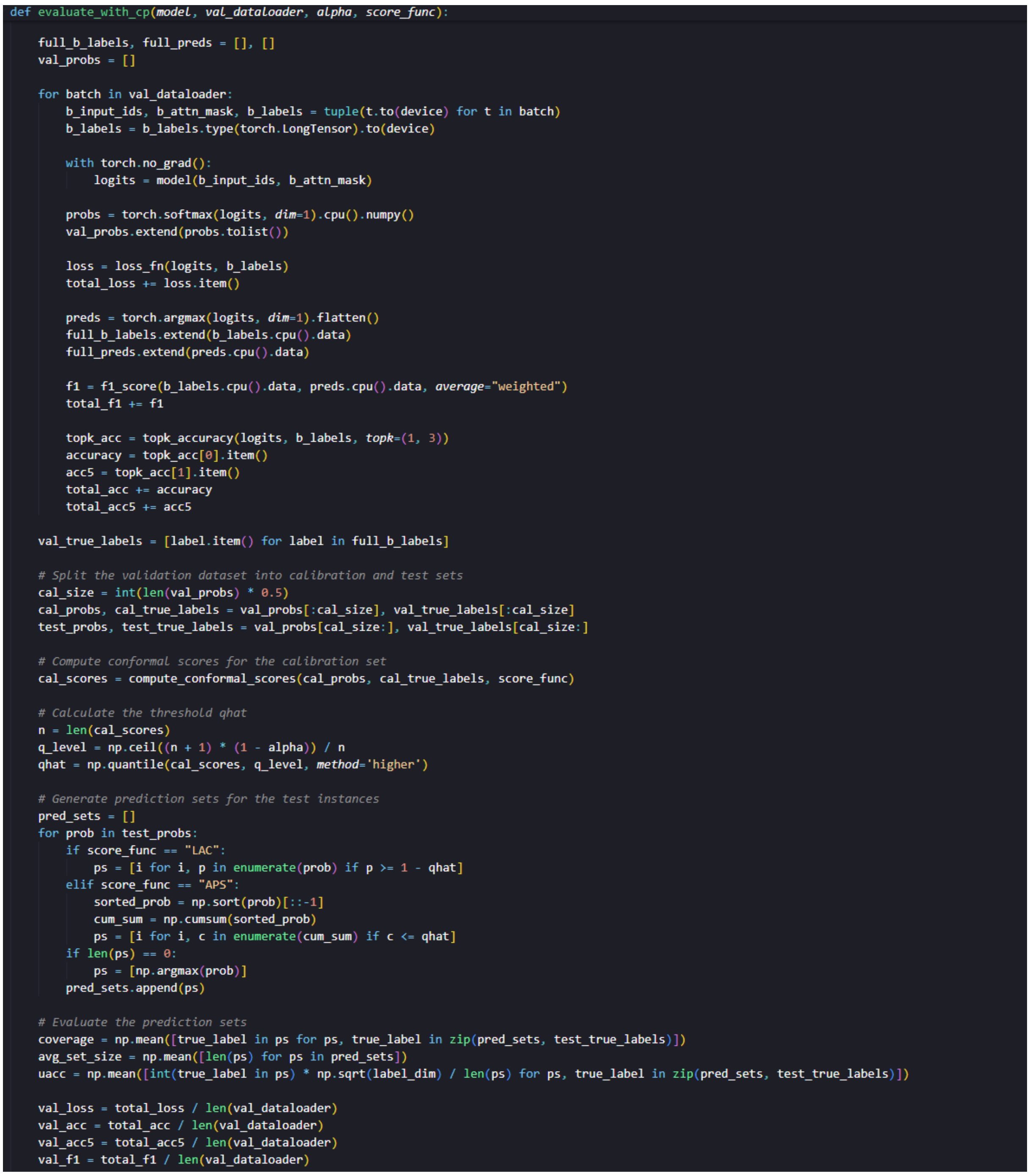
```
            # Update parameters and the learning rate
            optimizer.step()
            scheduler.step()

            # Print the loss values and time elapsed for every 20 batches
            if (step % 20 == 0 and step != 0) or (step == len(train_dataloader) - 1):
                # Calculate time elapsed for 20 batches
                time_elapsed = time.time() - t0_batch

                # Print training results
                print(f"{epoch_i + 1:^7} | {step:^7} | {batch_loss / batch_counts:^12.6f} | {batch_acc / batch_counts:^9.6f} | {batch_acc5 / batch_counts:^9.6f} | {batch_f1 / batch_counts:

                # Reset batch tracking variables
                batch_loss, batch_acc, batch_acc5, batch_f1, batch_counts = 0, 0, 0, 0, 0
                t0_batch = time.time()

        # Calculate the average loss over the entire training data
        avg_train_loss = total_loss / len(train_dataloader)
        avg_train_acc = total_acc / len(train_dataloader)
        avg_train_acc5 = total_acc5 / len(train_dataloader)
        avg_train_f1 = total_f1 / len(train_dataloader)
        train_loss_list.append(avg_train_loss)
        train_acc_list.append(avg_train_acc)
        train_acc5_list.append(avg_train_acc5)
        train_f1_list.append(avg_train_f1)

        print("-"*150)

        output_dir = './model_save/' + file_name + "_" + str(epoch_i) + ".pt"

        print("Saving model to %s" % output_dir)
        torch.save(model, output_dir)


        # =======================================
        #               Evaluation
        # =======================================
        #val_loss, val_accuracy, val_acc5, val_f1 = evaluate(model, val_dataloader)
        val_loss, val_accuracy, val_acc5, val_f1, coverage, avg_set_size, uacc = evaluate_with_cp(model, val_dataloader, alpha=0.1, score_func="LAC")
        val_loss_list.append(val_loss)
        val_acc_list.append(val_accuracy)
        val_acc5_list.append(val_acc5)
        val_f1_list.append(val_f1)
        coverage_list.append(coverage)
        avg_set_size_list.append(avg_set_size)
        uacc_list.append(uacc)

        # Print performance over the entire training data
        time_elapsed = time.time() - t0_epoch

        #print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Val Loss':^10} | {'Val Acc':^9} | {'Val Acc5':^9} | {'Val F1'
        print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Test Loss':^10} | {'Test Acc':^9} | {'Test Acc5':^9} | {'Test
        print("-"*150)
        #print(f"{epoch_i + 1:^7} | {'-':^7} | {avg_train_loss:^12.6f} | {avg_train_acc:^9.6f} | {avg_train_acc5:^9.6f} | {avg_train_f1:^12.6f} | {val_loss:^10.6f} | {val_accuracy:^9.6f} |
        print(f"{epoch_i + 1:^7} | {'-':^7} | {avg_train_loss:^12.6f} | {avg_train_acc:^9.6f} | {avg_train_acc5:^9.6f} | {avg_train_f1:^12.6f} | {val_loss:^10.6f} | {val_accuracy:^9.6f} | {
        print("-"*150)

        # =======================================
#               Test
# =======================================
        if evaluation == True:
            # After the completion of each training epoch, measure the model's performance
            # on our validation set.
            test_loss, test_accuracy, test_acc5, test_f1, test_coverage, test_avg_set_size, test_uacc = evaluate_with_cp(model, test_dataloader, alpha=0.1, score_func="LAC")
            test_loss_list.append(test_loss)
            test_acc_list.append(test_accuracy)
            test_acc5_list.append(test_acc5)
            test_f1_list.append(test_f1)
```

Figure 11: Code snippet 6



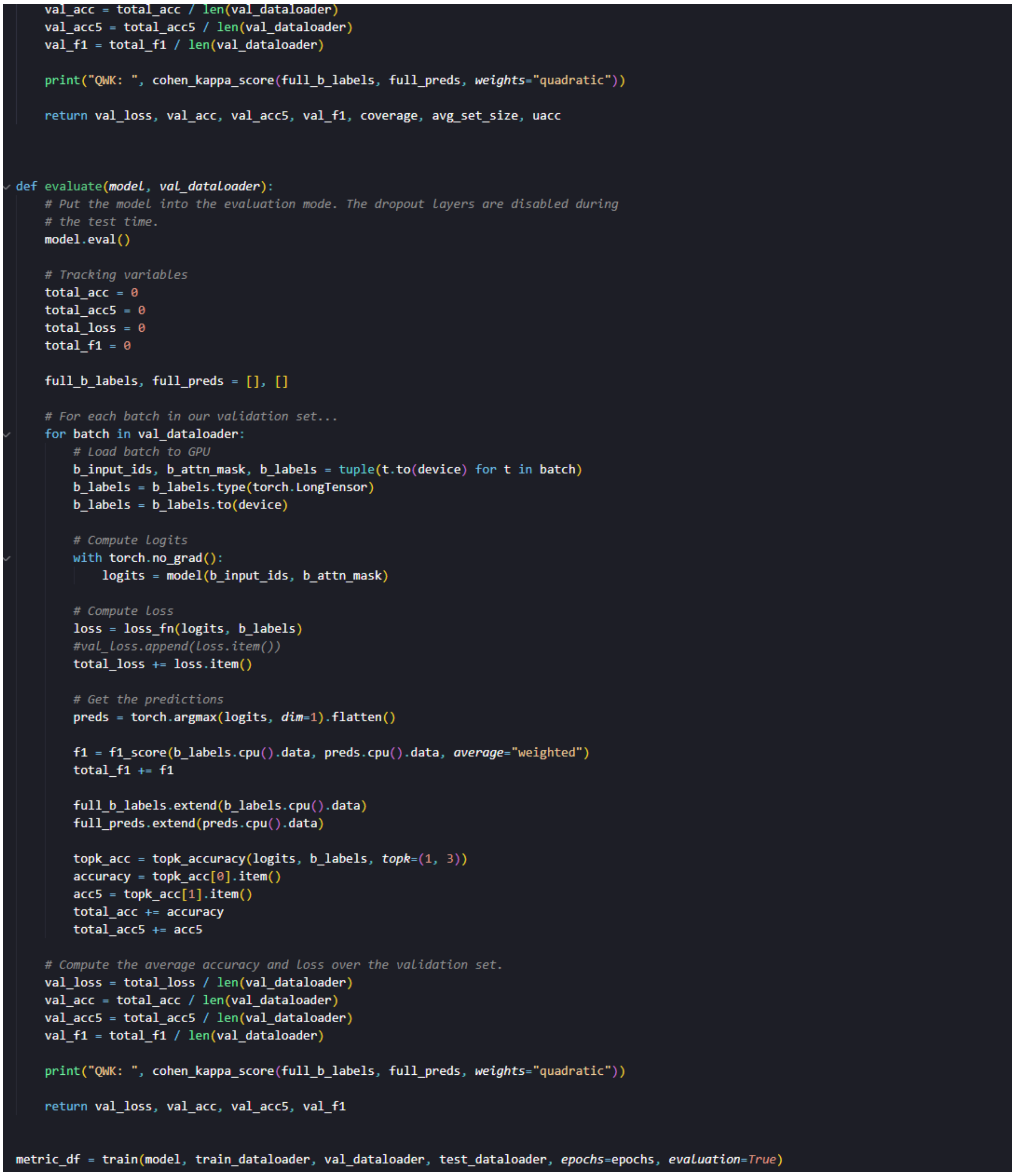
```
# ======================================
        if evaluation == True:
            # After the completion of each training epoch, measure the model's performance
            # on our validation set.
            test_loss, test_accuracy, test_acc5, test_f1, test_coverage, test_avg_set_size, test_uacc = evaluate_with_cp(model, test_dataloader, alpha=0.1, score_f
            test_loss_list.append(test_loss)
            test_acc_list.append(test_accuracy)
            test_acc5_list.append(test_acc5)
            test_f1_list.append(test_f1)
            test_coverage_list.append(test_coverage)
            test_avg_set_size_list.append(test_avg_set_size)
            test_uacc_list.append(test_uacc)

            # Print performance over the entire training data
            time_elapsed = time.time() - t0_epoch
            print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Test Loss':^10} | {'Test Acc':^9
            print("-"*200)
            print(f"{epoch_i + 1:^7} | {'-':^7} | {avg_train_loss:^12.6f} | {avg_train_acc:^9.6f} | {avg_train_acc5:^9.6f} | {avg_train_f1:^12.6f} | {test_loss:^10
            print("-"*200)

        print("\n")

        metric_df = pd.DataFrame(
            np.array([train_loss_list, train_acc_list, train_acc5_list, train_f1_list, val_loss_list, val_acc_list, val_acc5_list, val_f1_list, test_loss_list, tes
            columns=["train_loss", "train_acc", "train_acc5", "train_f1", "val_loss", "val_acc", "val_acc5", "val_f1", "test_loss", "test_acc", "test_acc5", "test_

        print(metric_df)
        metric_df.to_csv("results" + file_name + ".csv", encoding="utf_8_sig")
        print("Training complete!")

# from sklearn.metrics import f1_score, cohen_kappa_score


ef compute_conformal_scores(probs, true_labels, score_func):
    scores = []
    for prob, label in zip(probs, true_labels):
        if score_func == "LAC":
            score = 1 - prob[label]
        elif score_func == "APS":
            sorted_probs = np.sort(prob)[::-1]
            cum_sum = np.cumsum(sorted_probs)
            score = cum_sum[np.where(np.argsort(prob)[::-1] == label)[0][0]]
        scores.append(score)
    return scores

ef evaluate_with_cp(model, val_dataloader, alpha, score_func):
    model.eval()

    total_acc = 0
    total_acc5 = 0
    total_loss = 0
    total_f1 = 0

    #options = 3 #Manually assinged for score ranges. For example prompt case 3 has 3 scores and 1 has 2-12




    full_b_labels, full_preds = [], []
    val_probs = []

    for batch in val_dataloader:
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor).to(device)
```

Figure 12: Code snippet 7



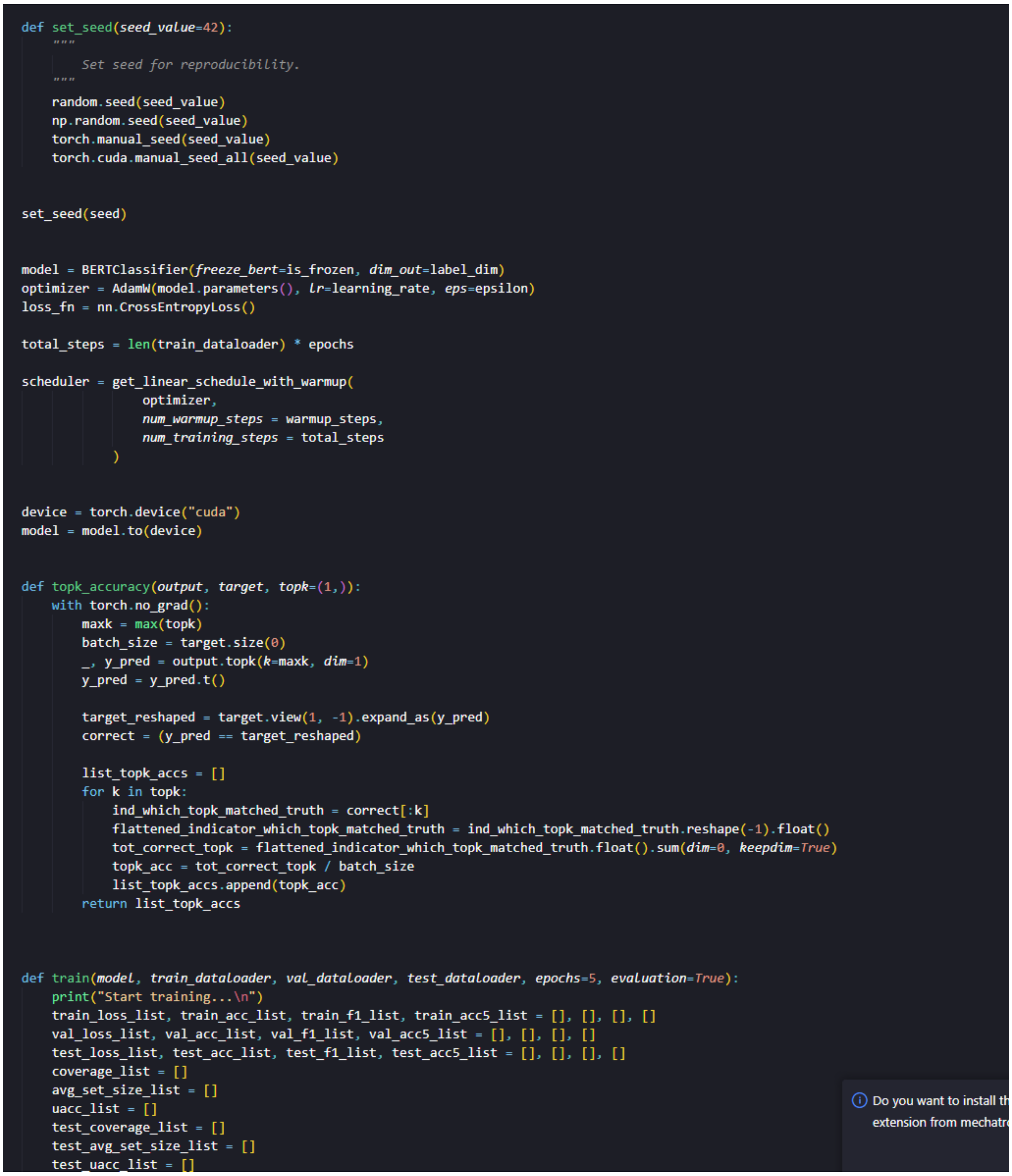
```python
def evaluate_with_cp(model, val_dataloader, alpha, score_func):

    full_b_labels, full_preds = [], []
    val_probs = []

    for batch in val_dataloader:
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor).to(device)

        with torch.no_grad():
            logits = model(b_input_ids, b_attn_mask)

        probs = torch.softmax(logits, dim=1).cpu().numpy()
        val_probs.extend(probs.tolist())

        loss = loss_fn(logits, b_labels)
        total_loss += loss.item()

        preds = torch.argmax(logits, dim=1).flatten()
        full_b_labels.extend(b_labels.cpu().data)
        full_preds.extend(preds.cpu().data)

        f1 = f1_score(b_labels.cpu().data, preds.cpu().data, average="weighted")
        total_f1 += f1

        topk_acc = topk_accuracy(logits, b_labels, topk=(1, 3))
        accuracy = topk_acc[0].item()
        acc5 = topk_acc[1].item()
        total_acc += accuracy
        total_acc5 += acc5

    val_true_labels = [label.item() for label in full_b_labels]

    # Split the validation dataset into calibration and test sets
    cal_size = int(len(val_probs) * 0.5)
    cal_probs, cal_true_labels = val_probs[:cal_size], val_true_labels[:cal_size]
    test_probs, test_true_labels = val_probs[cal_size:], val_true_labels[cal_size:]

    # Compute conformal scores for the calibration set
    cal_scores = compute_conformal_scores(cal_probs, cal_true_labels, score_func)

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        if score_func == "LAC":
            ps = [i for i, p in enumerate(prob) if p >= 1 - qhat]
        elif score_func == "APS":
            sorted_prob = np.sort(prob)[::-1]
            cum_sum = np.cumsum(sorted_prob)
            ps = [i for i, c in enumerate(cum_sum) if c <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append(ps)

    # Evaluate the prediction sets
    coverage = np.mean([true_label in ps for ps, true_label in zip(pred_sets, test_true_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(true_label in ps) * np.sqrt(label_dim) / len(ps) for ps, true_label in zip(pred_sets, test_true_labels)])

    val_loss = total_loss / len(val_dataloader)
    val_acc = total_acc / len(val_dataloader)
    val_acc5 = total_acc5 / len(val_dataloader)
    val_f1 = total_f1 / len(val_dataloader)
```

Figure 13: Code snippet 8



```
    val_acc = total_acc / len(val_dataloader)
    val_acc5 = total_acc5 / len(val_dataloader)
    val_f1 = total_f1 / len(val_dataloader)

    print("QWK: ", cohen_kappa_score(full_b_labels, full_preds, weights="quadratic"))

    return val_loss, val_acc, val_acc5, val_f1, coverage, avg_set_size, uacc



def evaluate(model, val_dataloader):
    # Put the model into the evaluation mode. The dropout layers are disabled during
    # the test time.
    model.eval()

    # Tracking variables
    total_acc = 0
    total_acc5 = 0
    total_loss = 0
    total_f1 = 0

    full_b_labels, full_preds = [], []

    # For each batch in our validation set...
    for batch in val_dataloader:
        # Load batch to GPU
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor)
        b_labels = b_labels.to(device)

        # Compute logits
        with torch.no_grad():
            logits = model(b_input_ids, b_attn_mask)

        # Compute loss
        loss = loss_fn(logits, b_labels)
        #val_loss.append(loss.item())
        total_loss += loss.item()

        # Get the predictions
        preds = torch.argmax(logits, dim=1).flatten()

        f1 = f1_score(b_labels.cpu().data, preds.cpu().data, average="weighted")
        total_f1 += f1

        full_b_labels.extend(b_labels.cpu().data)
        full_preds.extend(preds.cpu().data)

        topk_acc = topk_accuracy(logits, b_labels, topk=(1, 3))
        accuracy = topk_acc[0].item()
        acc5 = topk_acc[1].item()
        total_acc += accuracy
        total_acc5 += acc5

    # Compute the average accuracy and loss over the validation set.
    val_loss = total_loss / len(val_dataloader)
    val_acc = total_acc / len(val_dataloader)
    val_acc5 = total_acc5 / len(val_dataloader)
    val_f1 = total_f1 / len(val_dataloader)

    print("QWK: ", cohen_kappa_score(full_b_labels, full_preds, weights="quadratic"))

    return val_loss, val_acc, val_acc5, val_f1


metric_df = train(model, train_dataloader, val_dataloader, test_dataloader, epochs=epochs, evaluation=True)
```

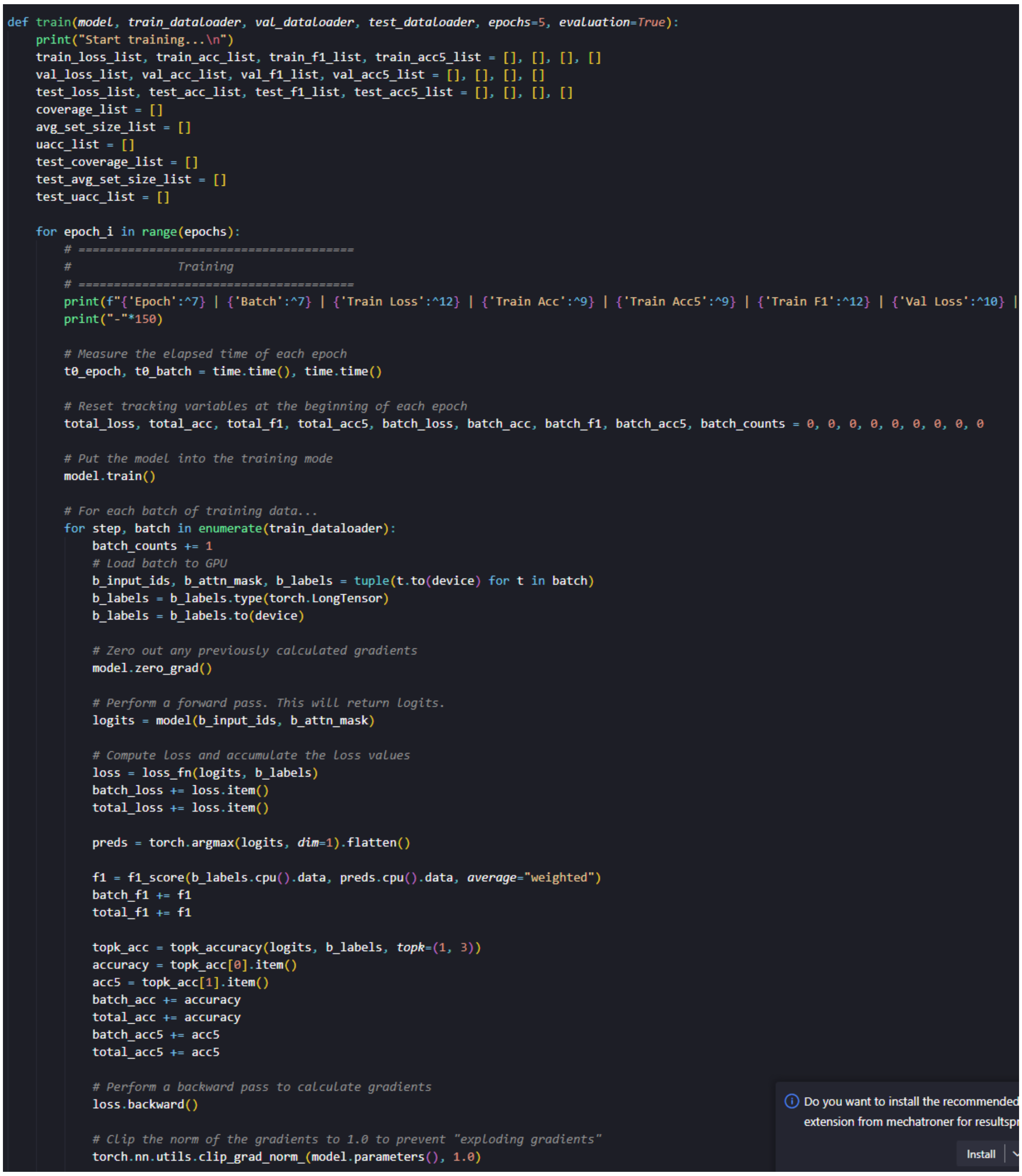
Figure 14: Code snippet 9



```python
def set_seed(seed_value=42):
    """
        Set seed for reproducibility.
    """
    random.seed(seed_value)
    np.random.seed(seed_value)
    torch.manual_seed(seed_value)
    torch.cuda.manual_seed_all(seed_value)


set_seed(seed)


model = BERTClassifier(freeze_bert=is_frozen, dim_out=label_dim)
optimizer = AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
loss_fn = nn.CrossEntropyLoss()

total_steps = len(train_dataloader) * epochs

scheduler = get_linear_schedule_with_warmup(
                optimizer,
                num_warmup_steps = warmup_steps,
                num_training_steps = total_steps
            )


device = torch.device("cuda")
model = model.to(device)


def topk_accuracy(output, target, topk=(1,)):
    with torch.no_grad():
        maxk = max(topk)
        batch_size = target.size(0)
        _, y_pred = output.topk(k=maxk, dim=1)
        y_pred = y_pred.t()

        target_reshaped = target.view(1, -1).expand_as(y_pred)
        correct = (y_pred == target_reshaped)

        list_topk_accs = []
        for k in topk:
            ind_which_topk_matched_truth = correct[:k]
            flattened_indicator_which_topk_matched_truth = ind_which_topk_matched_truth.reshape(-1).float()
            tot_correct_topk = flattened_indicator_which_topk_matched_truth.float().sum(dim=0, keepdim=True)
            topk_acc = tot_correct_topk / batch_size
            list_topk_accs.append(topk_acc)
        return list_topk_accs



def train(model, train_dataloader, val_dataloader, test_dataloader, epochs=5, evaluation=True):
    print("Start training...\n")
    train_loss_list, train_acc_list, train_f1_list, train_acc5_list = [], [], [], []
    val_loss_list, val_acc_list, val_f1_list, val_acc5_list = [], [], [], []
    test_loss_list, test_acc_list, test_f1_list, test_acc5_list = [], [], [], []
    coverage_list = []
    avg_set_size_list = []
    uacc_list = []
    test_coverage_list = []
    test_avg_set_size_list = []
    test_uacc_list = []
```

Do you want to install th
extension from mechatr

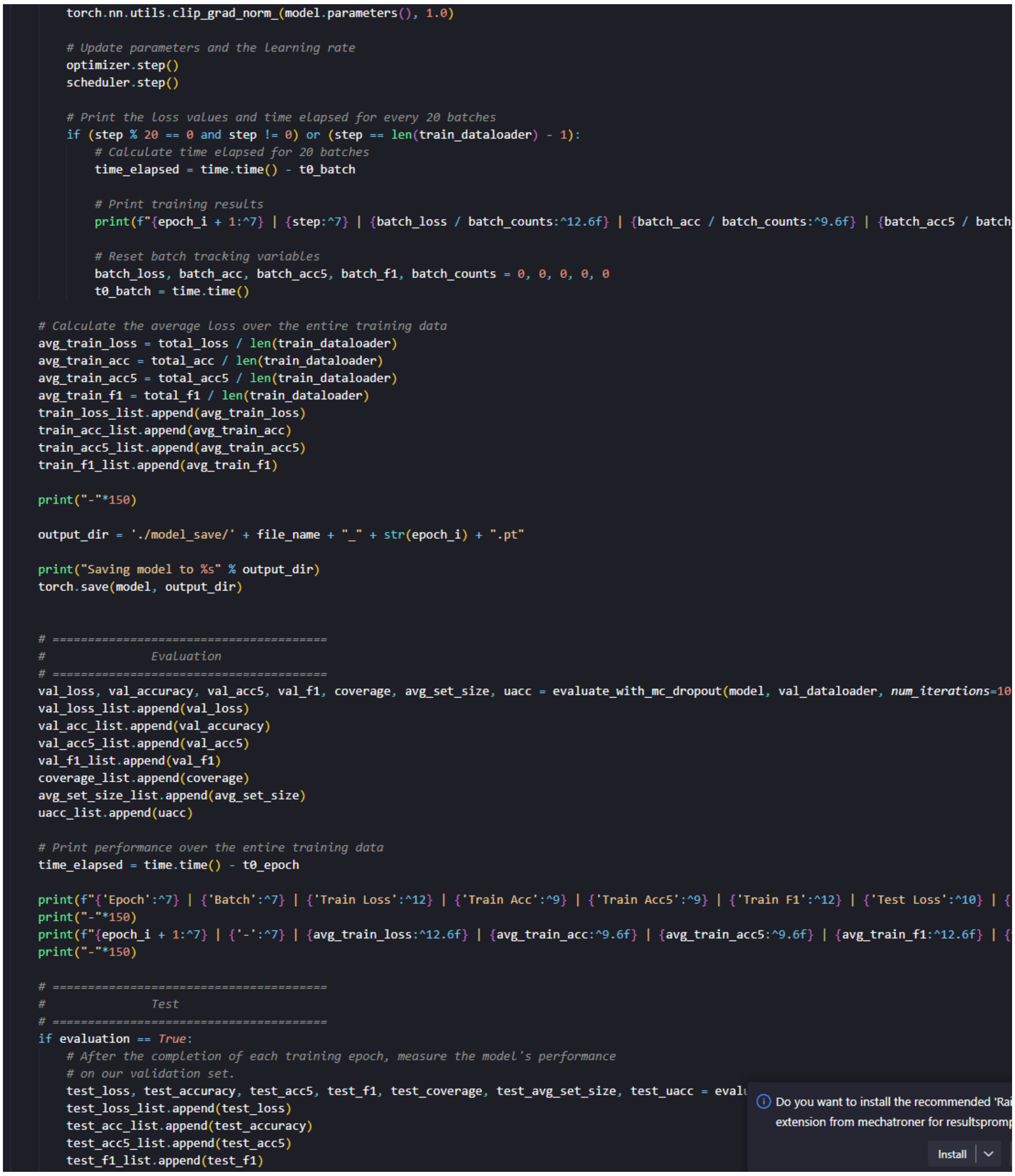
Figure 15: Code snippet 10



```python
def train(model, train_dataloader, val_dataloader, test_dataloader, epochs=5, evaluation=True):
    print("Start training...\n")
    train_loss_list, train_acc_list, train_f1_list, train_acc5_list = [], [], [], []
    val_loss_list, val_acc_list, val_f1_list, val_acc5_list = [], [], [], []
    test_loss_list, test_acc_list, test_f1_list, test_acc5_list = [], [], [], []
    coverage_list = []
    avg_set_size_list = []
    uacc_list = []
    test_coverage_list = []
    test_avg_set_size_list = []
    test_uacc_list = []

    for epoch_i in range(epochs):
        # =======================================
        #               Training
        # =======================================
        print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Val Loss':^10} |
        print("-"*150)

        # Measure the elapsed time of each epoch
        t0_epoch, t0_batch = time.time(), time.time()

        # Reset tracking variables at the beginning of each epoch
        total_loss, total_acc, total_f1, total_acc5, batch_loss, batch_acc, batch_f1, batch_acc5, batch_counts = 0, 0, 0, 0, 0, 0, 0, 0, 0

        # Put the model into the training mode
        model.train()

        # For each batch of training data...
        for step, batch in enumerate(train_dataloader):
            batch_counts += 1
            # Load batch to GPU
            b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
            b_labels = b_labels.type(torch.LongTensor)
            b_labels = b_labels.to(device)

            # Zero out any previously calculated gradients
            model.zero_grad()

            # Perform a forward pass. This will return logits.
            logits = model(b_input_ids, b_attn_mask)

            # Compute loss and accumulate the loss values
            loss = loss_fn(logits, b_labels)
            batch_loss += loss.item()
            total_loss += loss.item()

            preds = torch.argmax(logits, dim=1).flatten()

            f1 = f1_score(b_labels.cpu().data, preds.cpu().data, average="weighted")
            batch_f1 += f1
            total_f1 += f1

            topk_acc = topk_accuracy(logits, b_labels, topk=(1, 3))
            accuracy = topk_acc[0].item()
            acc5 = topk_acc[1].item()
            batch_acc += accuracy
            total_acc += accuracy
            batch_acc5 += acc5
            total_acc5 += acc5

            # Perform a backward pass to calculate gradients
            loss.backward()

            # Clip the norm of the gradients to 1.0 to prevent "exploding gradients"
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

Do you want to install the recommended
extension from mechatroner for resultspr
Install

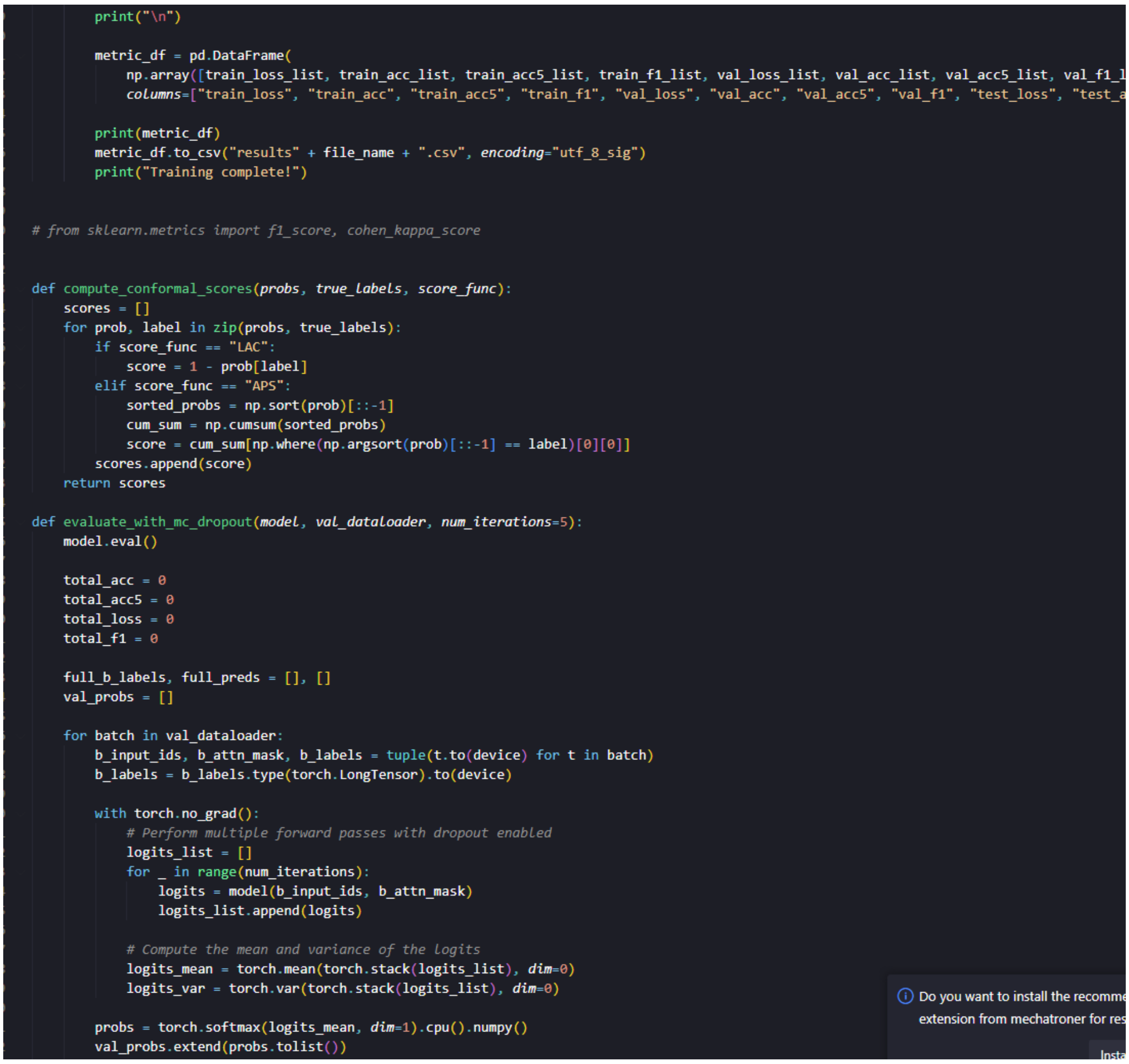
Figure 16: Code snippet 11



```
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Update parameters and the learning rate
        optimizer.step()
        scheduler.step()

        # Print the loss values and time elapsed for every 20 batches
        if (step % 20 == 0 and step != 0) or (step == len(train_dataloader) - 1):
            # Calculate time elapsed for 20 batches
            time_elapsed = time.time() - t0_batch

            # Print training results
            print(f"{epoch_i + 1:^7} | {step:^7} | {batch_loss / batch_counts:^12.6f} | {batch_acc / batch_counts:^9.6f} | {batch_acc5 / batch

            # Reset batch tracking variables
            batch_loss, batch_acc, batch_acc5, batch_f1, batch_counts = 0, 0, 0, 0, 0
            t0_batch = time.time()

    # Calculate the average loss over the entire training data
    avg_train_loss = total_loss / len(train_dataloader)
    avg_train_acc = total_acc / len(train_dataloader)
    avg_train_acc5 = total_acc5 / len(train_dataloader)
    avg_train_f1 = total_f1 / len(train_dataloader)
    train_loss_list.append(avg_train_loss)
    train_acc_list.append(avg_train_acc)
    train_acc5_list.append(avg_train_acc5)
    train_f1_list.append(avg_train_f1)

    print("-"*150)

    output_dir = './model_save/' + file_name + "_" + str(epoch_i) + ".pt"

    print("Saving model to %s" % output_dir)
    torch.save(model, output_dir)


    # =======================================
    #               Evaluation
    # =======================================
    val_loss, val_accuracy, val_acc5, val_f1, coverage, avg_set_size, uacc = evaluate_with_mc_dropout(model, val_dataloader, num_iterations=10
    val_loss_list.append(val_loss)
    val_acc_list.append(val_accuracy)
    val_acc5_list.append(val_acc5)
    val_f1_list.append(val_f1)
    coverage_list.append(coverage)
    avg_set_size_list.append(avg_set_size)
    uacc_list.append(uacc)

    # Print performance over the entire training data
    time_elapsed = time.time() - t0_epoch

    print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Train Acc':^9} | {'Train Acc5':^9} | {'Train F1':^12} | {'Test Loss':^10} | {
    print("-"*150)
    print(f"{epoch_i + 1:^7} | {'-':^7} | {avg_train_loss:^12.6f} | {avg_train_acc:^9.6f} | {avg_train_acc5:^9.6f} | {avg_train_f1:^12.6f} | {
    print("-"*150)

    # =======================================
    #               Test
    # =======================================
    if evaluation == True:
        # After the completion of each training epoch, measure the model's performance
        # on our validation set.
        test_loss, test_accuracy, test_acc5, test_f1, test_coverage, test_avg_set_size, test_uacc = evalu
        test_loss_list.append(test_loss)
        test_acc_list.append(test_accuracy)
        test_acc5_list.append(test_acc5)
        test_f1_list.append(test_f1)
```

Do you want to install the recommended 'Ra
extension from mechatroner for resultspromp

Figure 17: Code snippet 12

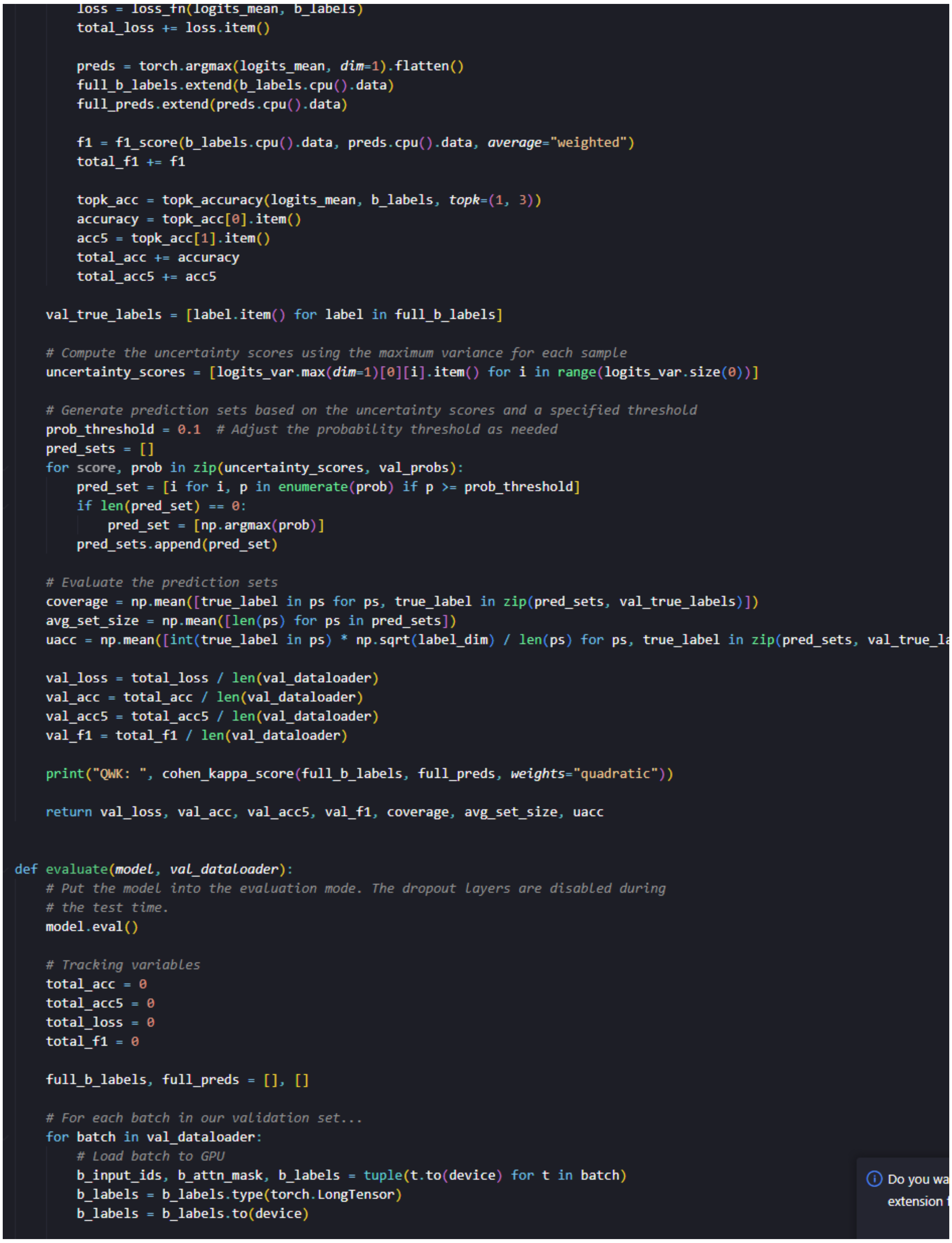


```
        print("\n")

        metric_df = pd.DataFrame(
            np.array([train_loss_list, train_acc_list, train_acc5_list, train_f1_list, val_loss_list, val_acc_list, val_acc5_list, val_f1_l
            columns=["train_loss", "train_acc", "train_acc5", "train_f1", "val_loss", "val_acc", "val_acc5", "val_f1", "test_loss", "test_a

        print(metric_df)
        metric_df.to_csv("results" + file_name + ".csv", encoding="utf_8_sig")
        print("Training complete!")


# from sklearn.metrics import f1_score, cohen_kappa_score


def compute_conformal_scores(probs, true_labels, score_func):
    scores = []
    for prob, label in zip(probs, true_labels):
        if score_func == "LAC":
            score = 1 - prob[label]
        elif score_func == "APS":
            sorted_probs = np.sort(prob)[::-1]
            cum_sum = np.cumsum(sorted_probs)
            score = cum_sum[np.where(np.argsort(prob)[::-1] == label)[0][0]]
        scores.append(score)
    return scores

def evaluate_with_mc_dropout(model, val_dataloader, num_iterations=5):
    model.eval()

    total_acc = 0
    total_acc5 = 0
    total_loss = 0
    total_f1 = 0

    full_b_labels, full_preds = [], []
    val_probs = []

    for batch in val_dataloader:
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor).to(device)

        with torch.no_grad():
            # Perform multiple forward passes with dropout enabled
            logits_list = []
            for _ in range(num_iterations):
                logits = model(b_input_ids, b_attn_mask)
                logits_list.append(logits)

            # Compute the mean and variance of the logits
            logits_mean = torch.mean(torch.stack(logits_list), dim=0)
            logits_var = torch.var(torch.stack(logits_list), dim=0)

        probs = torch.softmax(logits_mean, dim=1).cpu().numpy()
        val_probs.extend(probs.tolist())
```

Do you want to install the recomme
extension from mechatroner for res

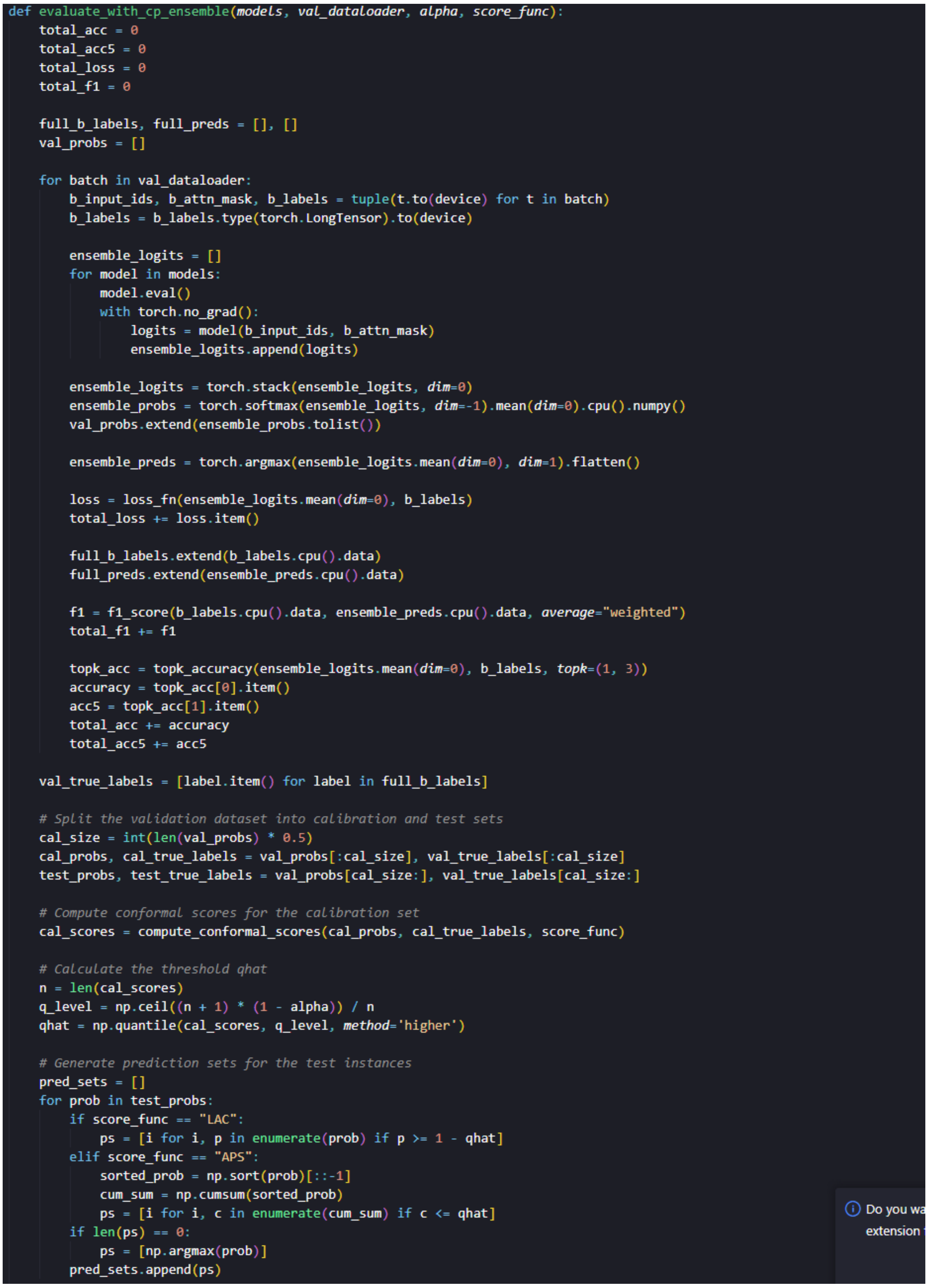
Insta

Figure 18: Code snippet 13



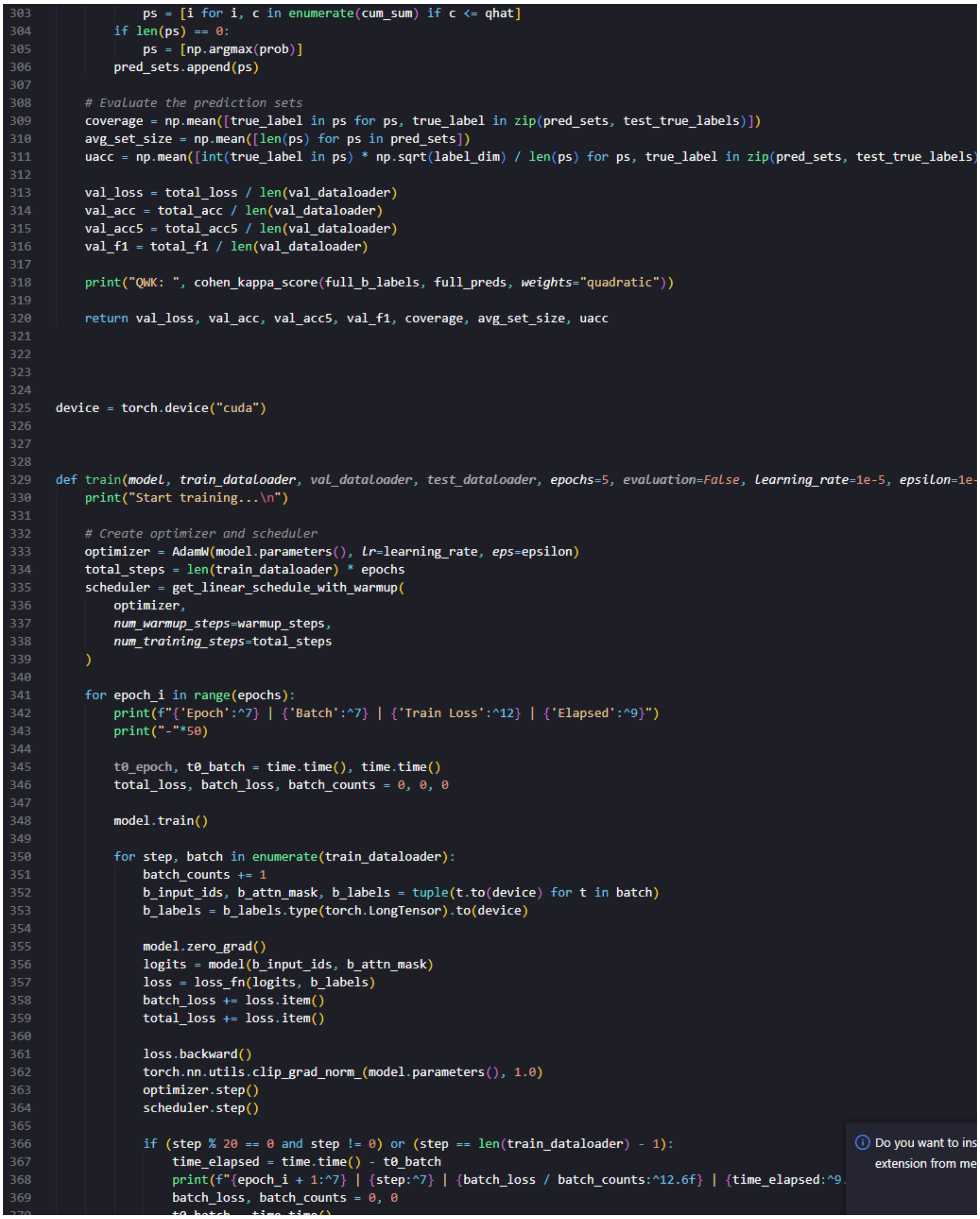
```
        loss = loss_fn(logits_mean, b_labels)
        total_loss += loss.item()

        preds = torch.argmax(logits_mean, dim=1).flatten()
        full_b_labels.extend(b_labels.cpu().data)
        full_preds.extend(preds.cpu().data)

        f1 = f1_score(b_labels.cpu().data, preds.cpu().data, average="weighted")
        total_f1 += f1

        topk_acc = topk_accuracy(logits_mean, b_labels, topk=(1, 3))
        accuracy = topk_acc[0].item()
        acc5 = topk_acc[1].item()
        total_acc += accuracy
        total_acc5 += acc5

    val_true_labels = [label.item() for label in full_b_labels]

    # Compute the uncertainty scores using the maximum variance for each sample
    uncertainty_scores = [logits_var.max(dim=1)[0][i].item() for i in range(logits_var.size(0))]

    # Generate prediction sets based on the uncertainty scores and a specified threshold
    prob_threshold = 0.1  # Adjust the probability threshold as needed
    pred_sets = []
    for score, prob in zip(uncertainty_scores, val_probs):
        pred_set = [i for i, p in enumerate(prob) if p >= prob_threshold]
        if len(pred_set) == 0:
            pred_set = [np.argmax(prob)]
        pred_sets.append(pred_set)

    # Evaluate the prediction sets
    coverage = np.mean([true_label in ps for ps, true_label in zip(pred_sets, val_true_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(true_label in ps) * np.sqrt(label_dim) / len(ps) for ps, true_label in zip(pred_sets, val_true_la

    val_loss = total_loss / len(val_dataloader)
    val_acc = total_acc / len(val_dataloader)
    val_acc5 = total_acc5 / len(val_dataloader)
    val_f1 = total_f1 / len(val_dataloader)

    print("QWK: ", cohen_kappa_score(full_b_labels, full_preds, weights="quadratic"))

    return val_loss, val_acc, val_acc5, val_f1, coverage, avg_set_size, uacc


def evaluate(model, val_dataloader):
    # Put the model into the evaluation mode. The dropout layers are disabled during
    # the test time.
    model.eval()

    # Tracking variables
    total_acc = 0
    total_acc5 = 0
    total_loss = 0
    total_f1 = 0

    full_b_labels, full_preds = [], []

    # For each batch in our validation set...
    for batch in val_dataloader:
        # Load batch to GPU
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor)
        b_labels = b_labels.to(device)
```

Figure 19: Code snippet 14



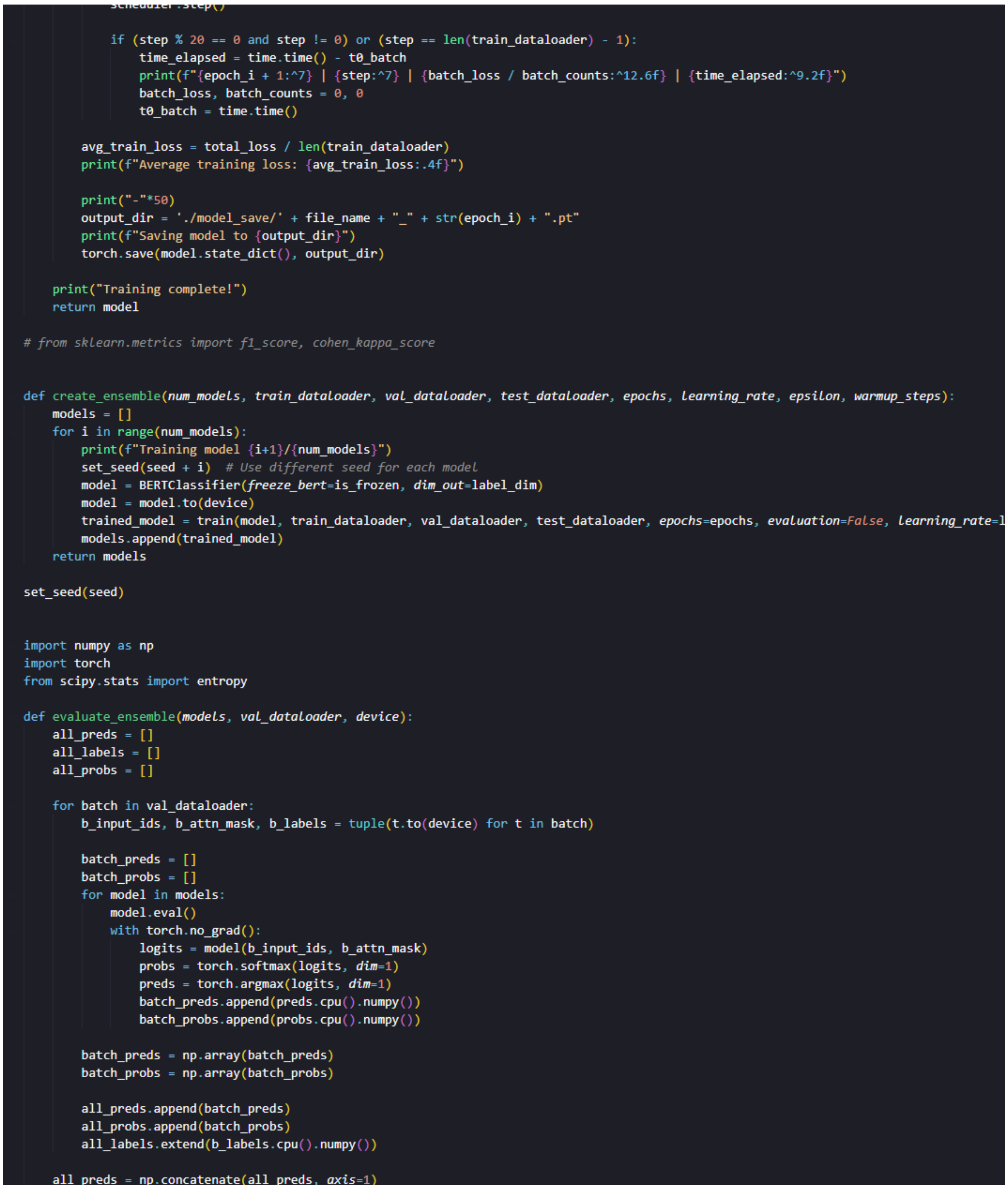
```python
def evaluate_with_cp_ensemble(models, val_dataloader, alpha, score_func):
    total_acc = 0
    total_acc5 = 0
    total_loss = 0
    total_f1 = 0

    full_b_labels, full_preds = [], []
    val_probs = []

    for batch in val_dataloader:
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
        b_labels = b_labels.type(torch.LongTensor).to(device)

        ensemble_logits = []
        for model in models:
            model.eval()
            with torch.no_grad():
                logits = model(b_input_ids, b_attn_mask)
                ensemble_logits.append(logits)

        ensemble_logits = torch.stack(ensemble_logits, dim=0)
        ensemble_probs = torch.softmax(ensemble_logits, dim=-1).mean(dim=0).cpu().numpy()
        val_probs.extend(ensemble_probs.tolist())

        ensemble_preds = torch.argmax(ensemble_logits.mean(dim=0), dim=1).flatten()

        loss = loss_fn(ensemble_logits.mean(dim=0), b_labels)
        total_loss += loss.item()

        full_b_labels.extend(b_labels.cpu().data)
        full_preds.extend(ensemble_preds.cpu().data)

        f1 = f1_score(b_labels.cpu().data, ensemble_preds.cpu().data, average="weighted")
        total_f1 += f1

        topk_acc = topk_accuracy(ensemble_logits.mean(dim=0), b_labels, topk=(1, 3))
        accuracy = topk_acc[0].item()
        acc5 = topk_acc[1].item()
        total_acc += accuracy
        total_acc5 += acc5

    val_true_labels = [label.item() for label in full_b_labels]

    # Split the validation dataset into calibration and test sets
    cal_size = int(len(val_probs) * 0.5)
    cal_probs, cal_true_labels = val_probs[:cal_size], val_true_labels[:cal_size]
    test_probs, test_true_labels = val_probs[cal_size:], val_true_labels[cal_size:]

    # Compute conformal scores for the calibration set
    cal_scores = compute_conformal_scores(cal_probs, cal_true_labels, score_func)

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        if score_func == "LAC":
            ps = [i for i, p in enumerate(prob) if p >= 1 - qhat]
        elif score_func == "APS":
            sorted_prob = np.sort(prob)[::-1]
            cum_sum = np.cumsum(sorted_prob)
            ps = [i for i, c in enumerate(cum_sum) if c <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append(ps)
```

Figure 20: Code snippet 15



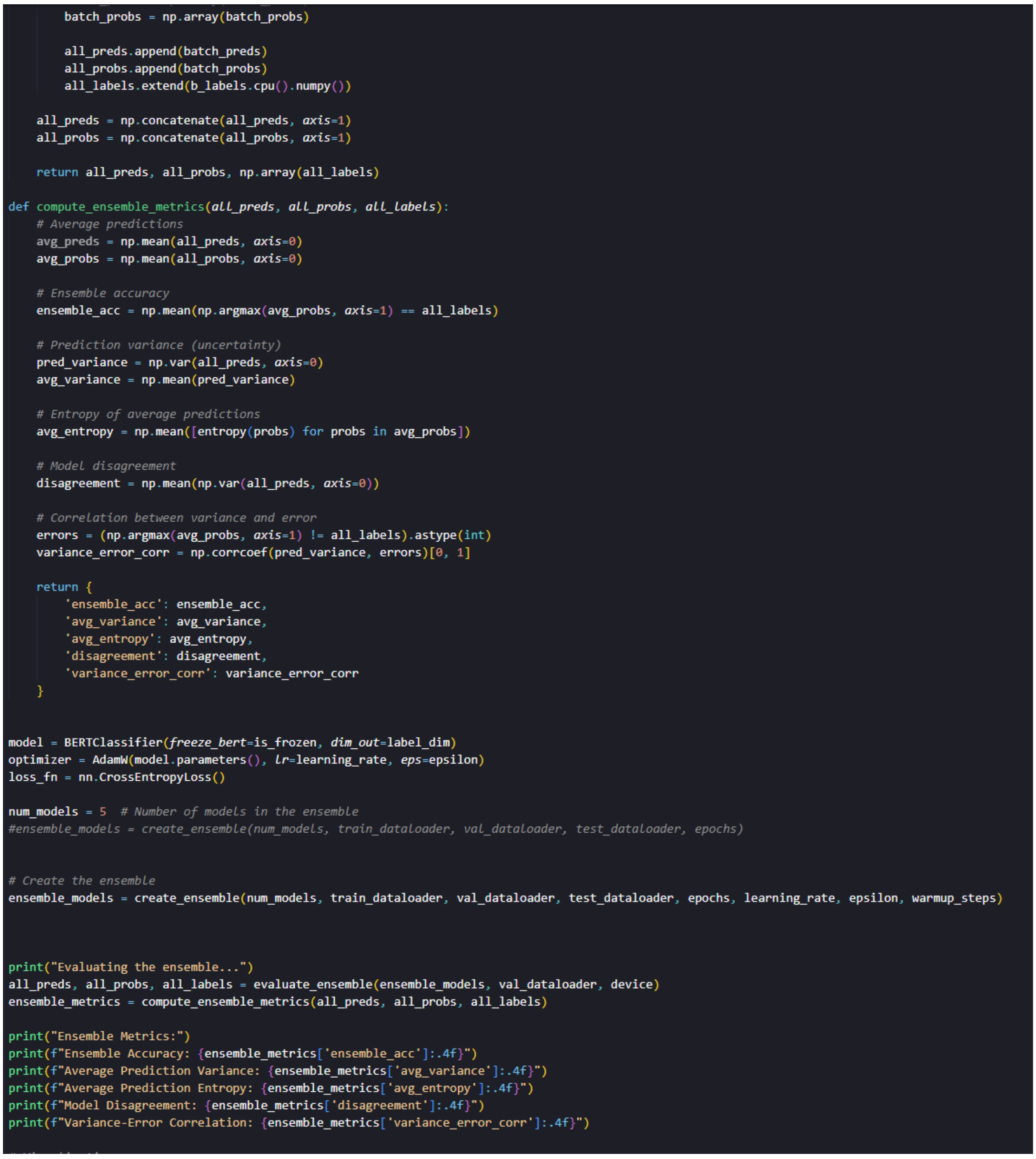
```
            ps = [i for i, c in enumerate(cum_sum) if c <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append(ps)

    # Evaluate the prediction sets
    coverage = np.mean([true_label in ps for ps, true_label in zip(pred_sets, test_true_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(true_label in ps) * np.sqrt(label_dim) / len(ps) for ps, true_label in zip(pred_sets, test_true_labels)

    val_loss = total_loss / len(val_dataloader)
    val_acc = total_acc / len(val_dataloader)
    val_acc5 = total_acc5 / len(val_dataloader)
    val_f1 = total_f1 / len(val_dataloader)

    print("QWK: ", cohen_kappa_score(full_b_labels, full_preds, weights="quadratic"))

    return val_loss, val_acc, val_acc5, val_f1, coverage, avg_set_size, uacc




device = torch.device("cuda")



def train(model, train_dataloader, val_dataloader, test_dataloader, epochs=5, evaluation=False, learning_rate=1e-5, epsilon=1e-
    print("Start training...\n")

    # Create optimizer and scheduler
    optimizer = AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
    total_steps = len(train_dataloader) * epochs
    scheduler = get_linear_schedule_with_warmup(
        optimizer,
        num_warmup_steps=warmup_steps,
        num_training_steps=total_steps
    )

    for epoch_i in range(epochs):
        print(f"{'Epoch':^7} | {'Batch':^7} | {'Train Loss':^12} | {'Elapsed':^9}")
        print("-"*50)

        t0_epoch, t0_batch = time.time(), time.time()
        total_loss, batch_loss, batch_counts = 0, 0, 0

        model.train()

        for step, batch in enumerate(train_dataloader):
            batch_counts += 1
            b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)
            b_labels = b_labels.type(torch.LongTensor).to(device)

            model.zero_grad()
            logits = model(b_input_ids, b_attn_mask)
            loss = loss_fn(logits, b_labels)
            batch_loss += loss.item()
            total_loss += loss.item()

            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler.step()

            if (step % 20 == 0 and step != 0) or (step == len(train_dataloader) - 1):
                time_elapsed = time.time() - t0_batch
                print(f"{epoch_i + 1:^7} | {step:^7} | {batch_loss / batch_counts:^12.6f} | {time_elapsed:^9
                batch_loss, batch_counts = 0, 0
```

Figure 21: Code snippet 16



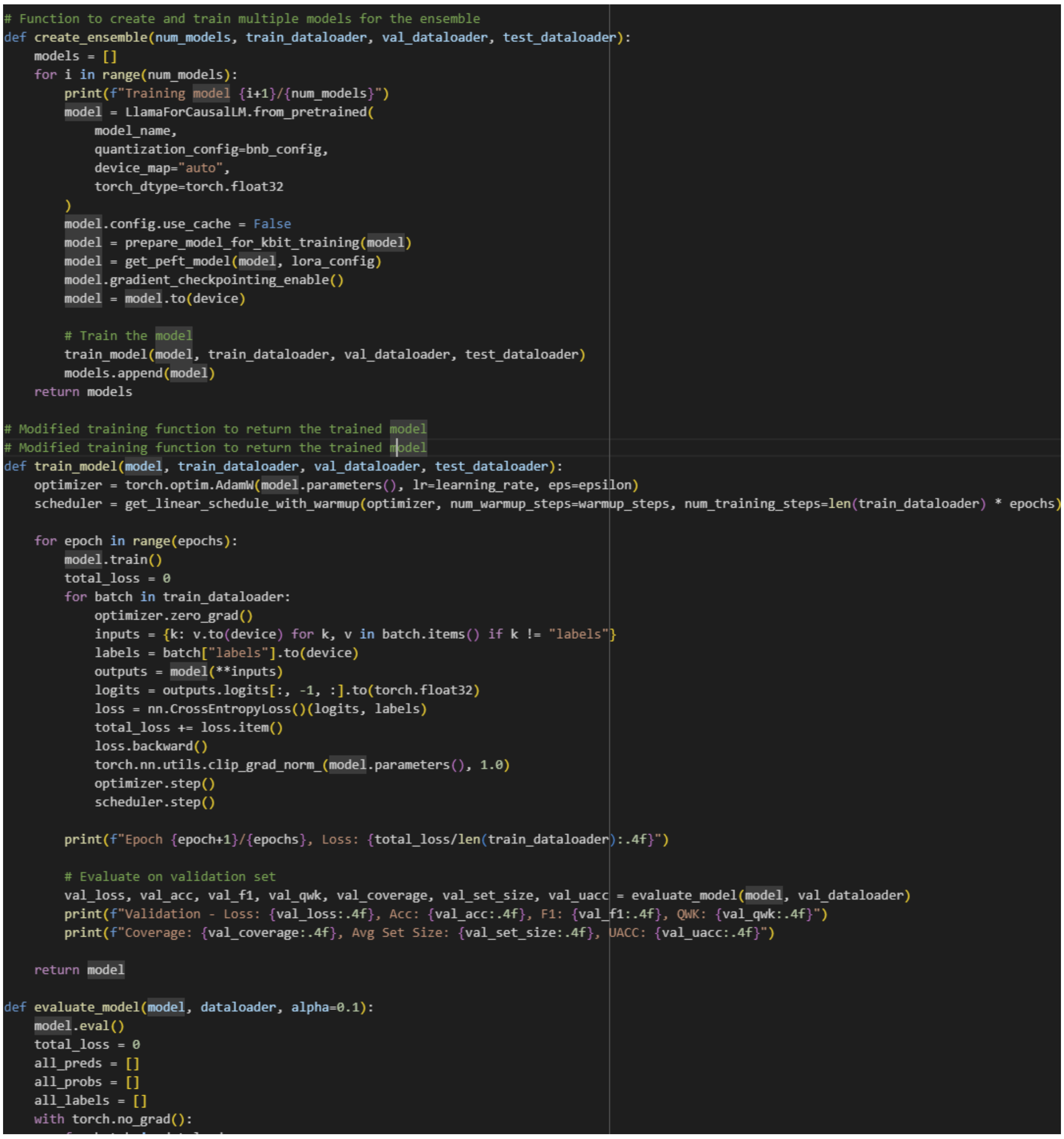
```
            scheduler.step()

            if (step % 20 == 0 and step != 0) or (step == len(train_dataloader) - 1):
                time_elapsed = time.time() - t0_batch
                print(f"{epoch_i + 1:^7} | {step:^7} | {batch_loss / batch_counts:^12.6f} | {time_elapsed:^9.2f}")
                batch_loss, batch_counts = 0, 0
                t0_batch = time.time()

        avg_train_loss = total_loss / len(train_dataloader)
        print(f"Average training loss: {avg_train_loss:.4f}")

        print("-"*50)
        output_dir = './model_save/' + file_name + "_" + str(epoch_i) + ".pt"
        print(f"Saving model to {output_dir}")
        torch.save(model.state_dict(), output_dir)

    print("Training complete!")
    return model

# from sklearn.metrics import f1_score, cohen_kappa_score


def create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader, epochs, learning_rate, epsilon, warmup_steps):
    models = []
    for i in range(num_models):
        print(f"Training model {i+1}/{num_models}")
        set_seed(seed + i)  # Use different seed for each model
        model = BERTClassifier(freeze_bert=is_frozen, dim_out=label_dim)
        model = model.to(device)
        trained_model = train(model, train_dataloader, val_dataloader, test_dataloader, epochs=epochs, evaluation=False, learning_rate=l
        models.append(trained_model)
    return models

set_seed(seed)


import numpy as np
import torch
from scipy.stats import entropy

def evaluate_ensemble(models, val_dataloader, device):
    all_preds = []
    all_labels = []
    all_probs = []

    for batch in val_dataloader:
        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)

        batch_preds = []
        batch_probs = []
        for model in models:
            model.eval()
            with torch.no_grad():
                logits = model(b_input_ids, b_attn_mask)
                probs = torch.softmax(logits, dim=1)
                preds = torch.argmax(logits, dim=1)
                batch_preds.append(preds.cpu().numpy())
                batch_probs.append(probs.cpu().numpy())

        batch_preds = np.array(batch_preds)
        batch_probs = np.array(batch_probs)

        all_preds.append(batch_preds)
        all_probs.append(batch_probs)
        all_labels.extend(b_labels.cpu().numpy())

    all_preds = np.concatenate(all_preds, axis=1)
```

Figure 22: Code snippet 17



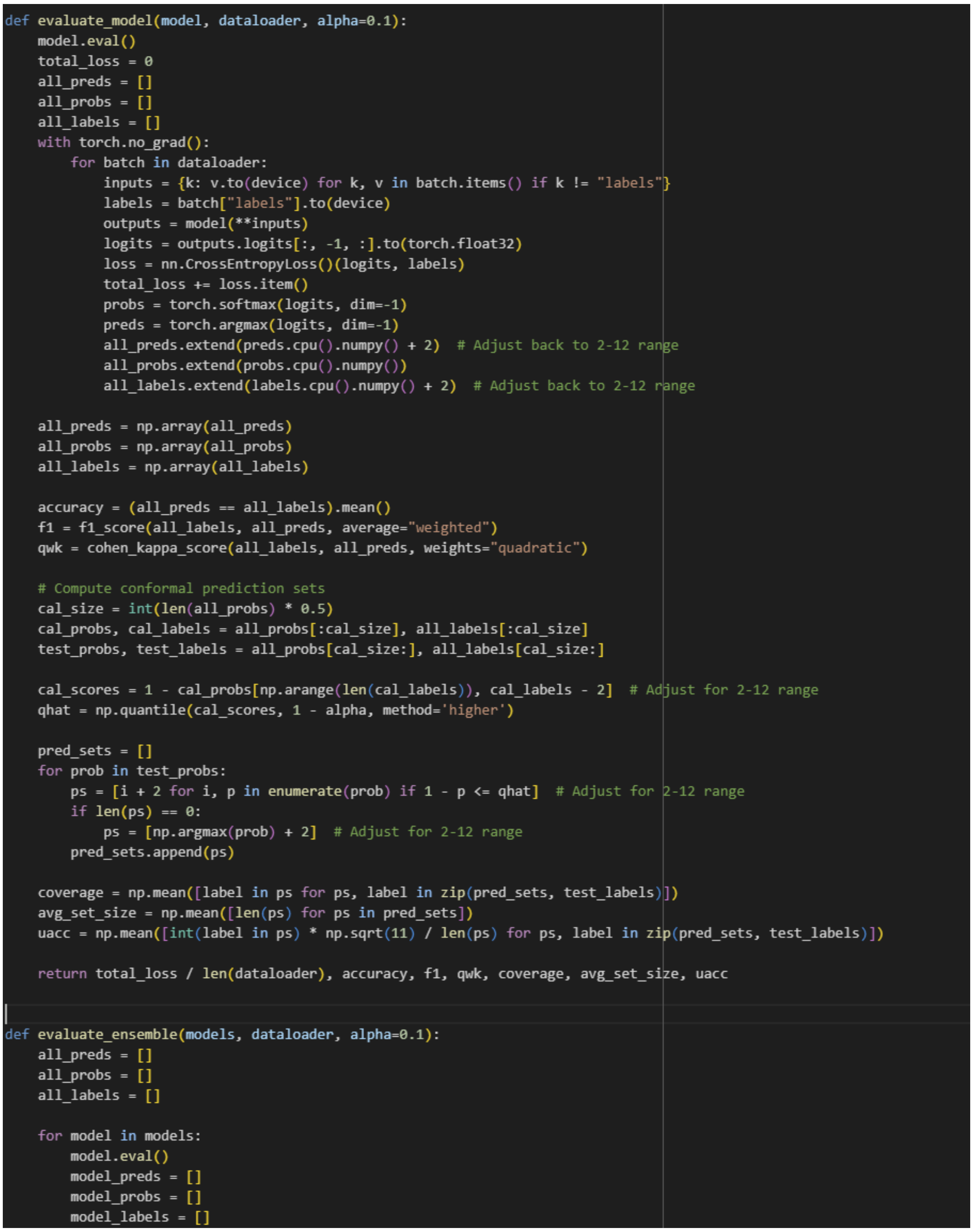
```python
        batch_probs = np.array(batch_probs)

        all_preds.append(batch_preds)
        all_probs.append(batch_probs)
        all_labels.extend(b_labels.cpu().numpy())

    all_preds = np.concatenate(all_preds, axis=1)
    all_probs = np.concatenate(all_probs, axis=1)

    return all_preds, all_probs, np.array(all_labels)

def compute_ensemble_metrics(all_preds, all_probs, all_labels):
    # Average predictions
    avg_preds = np.mean(all_preds, axis=0)
    avg_probs = np.mean(all_probs, axis=0)

    # Ensemble accuracy
    ensemble_acc = np.mean(np.argmax(avg_probs, axis=1) == all_labels)

    # Prediction variance (uncertainty)
    pred_variance = np.var(all_preds, axis=0)
    avg_variance = np.mean(pred_variance)

    # Entropy of average predictions
    avg_entropy = np.mean([entropy(probs) for probs in avg_probs])

    # Model disagreement
    disagreement = np.mean(np.var(all_preds, axis=0))

    # Correlation between variance and error
    errors = (np.argmax(avg_probs, axis=1) != all_labels).astype(int)
    variance_error_corr = np.corrcoef(pred_variance, errors)[0, 1]

    return {
        'ensemble_acc': ensemble_acc,
        'avg_variance': avg_variance,
        'avg_entropy': avg_entropy,
        'disagreement': disagreement,
        'variance_error_corr': variance_error_corr
    }


model = BERTClassifier(freeze_bert=is_frozen, dim_out=label_dim)
optimizer = AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
loss_fn = nn.CrossEntropyLoss()

num_models = 5  # Number of models in the ensemble
#ensemble_models = create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader, epochs)


# Create the ensemble
ensemble_models = create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader, epochs, learning_rate, epsilon, warmup_steps)



print("Evaluating the ensemble...")
all_preds, all_probs, all_labels = evaluate_ensemble(ensemble_models, val_dataloader, device)
ensemble_metrics = compute_ensemble_metrics(all_preds, all_probs, all_labels)

print("Ensemble Metrics:")
print(f"Ensemble Accuracy: {ensemble_metrics['ensemble_acc']:.4f}")
print(f"Average Prediction Variance: {ensemble_metrics['avg_variance']:.4f}")
print(f"Average Prediction Entropy: {ensemble_metrics['avg_entropy']:.4f}")
print(f"Model Disagreement: {ensemble_metrics['disagreement']:.4f}")
print(f"Variance-Error Correlation: {ensemble_metrics['variance_error_corr']:.4f}")
```

Figure 23: Code snippet 18



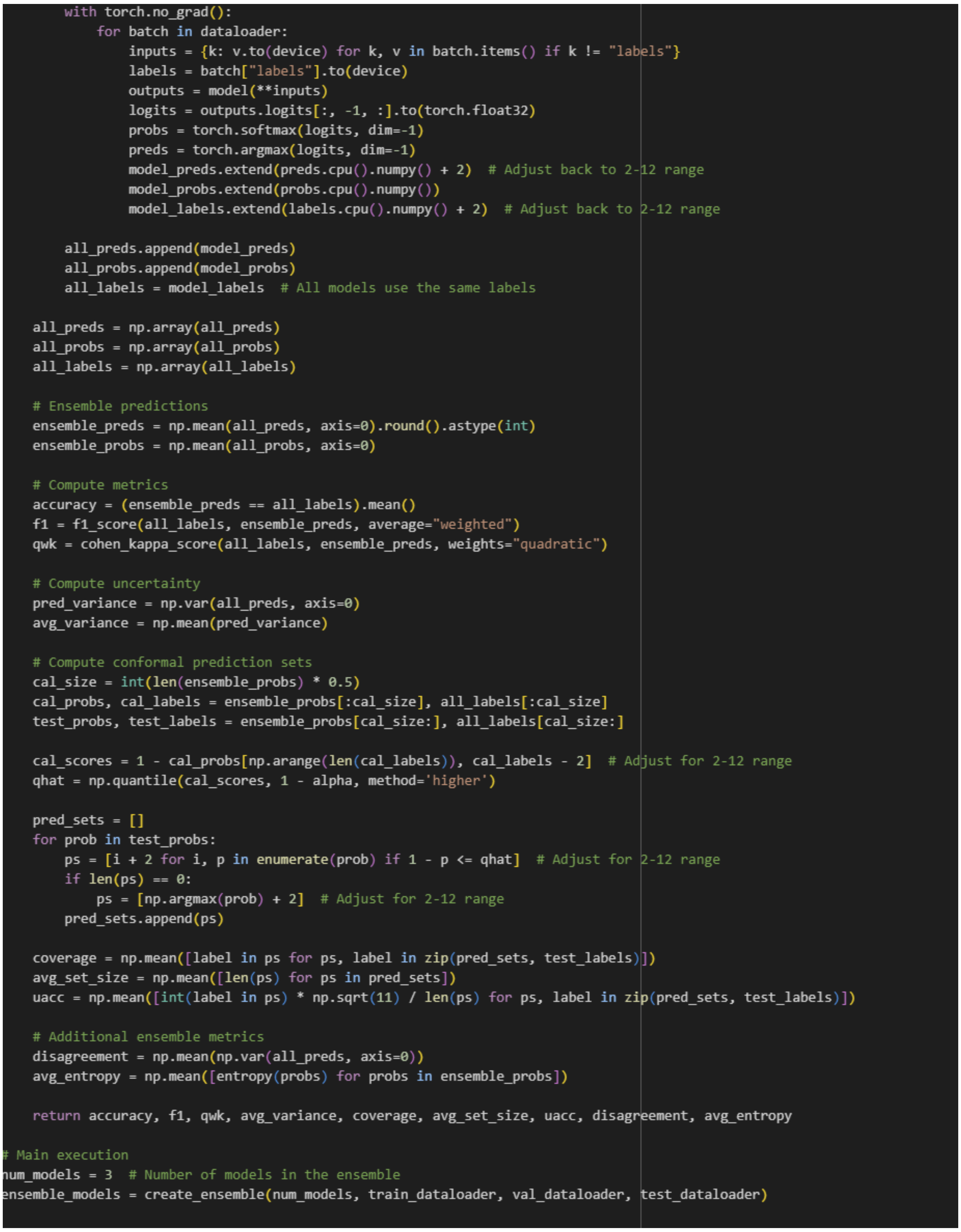
```python
# Function to create and train multiple models for the ensemble
def create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader):
    models = []
    for i in range(num_models):
        print(f"Training model {i+1}/{num_models}")
        model = LlamaForCausalLM.from_pretrained(
            model_name,
            quantization_config=bnb_config,
            device_map="auto",
            torch_dtype=torch.float32
        )
        model.config.use_cache = False
        model = prepare_model_for_kbit_training(model)
        model = get_peft_model(model, lora_config)
        model.gradient_checkpointing_enable()
        model = model.to(device)

        # Train the model
        train_model(model, train_dataloader, val_dataloader, test_dataloader)
        models.append(model)
    return models

# Modified training function to return the trained model
# Modified training function to return the trained model
def train_model(model, train_dataloader, val_dataloader, test_dataloader):
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
    scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=len(train_dataloader) * epochs)

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch in train_dataloader:
            optimizer.zero_grad()
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler.step()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(train_dataloader):.4f}")

        # Evaluate on validation set
        val_loss, val_acc, val_f1, val_qwk, val_coverage, val_set_size, val_uacc = evaluate_model(model, val_dataloader)
        print(f"Validation - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")
        print(f"Coverage: {val_coverage:.4f}, Avg Set Size: {val_set_size:.4f}, UACC: {val_uacc:.4f}")

    return model

def evaluate_model(model, dataloader, alpha=0.1):
    model.eval()
    total_loss = 0
    all_preds = []
    all_probs = []
    all_labels = []
    with torch.no_grad():
```

Figure 24: Code snippet 19



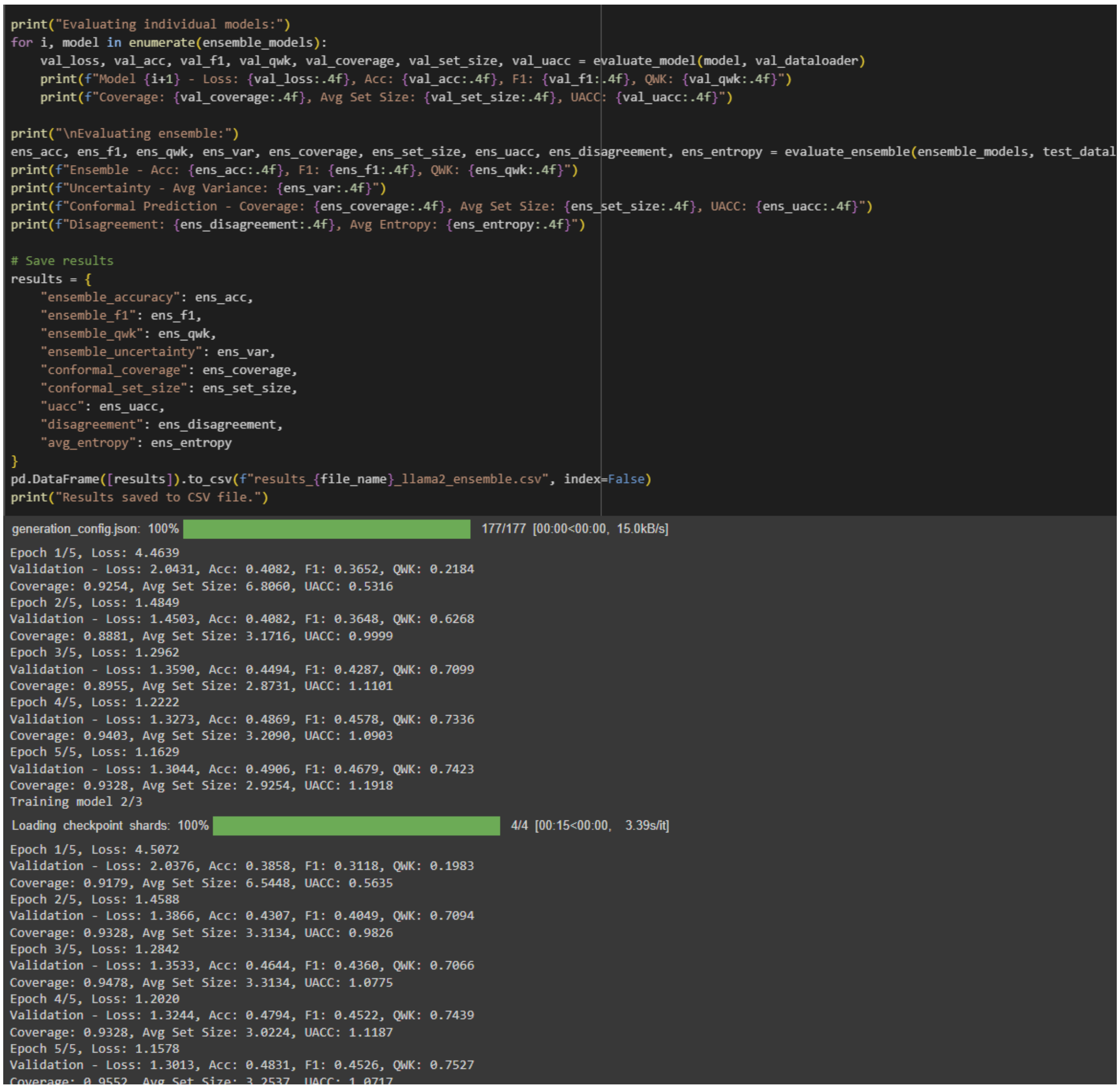
```python
def evaluate_model(model, dataloader, alpha=0.1):
    model.eval()
    total_loss = 0
    all_preds = []
    all_probs = []
    all_labels = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            probs = torch.softmax(logits, dim=-1)
            preds = torch.argmax(logits, dim=-1)
            all_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
            all_probs.extend(probs.cpu().numpy())
            all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

    all_preds = np.array(all_preds)
    all_probs = np.array(all_probs)
    all_labels = np.array(all_labels)

    accuracy = (all_preds == all_labels).mean()
    f1 = f1_score(all_labels, all_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, all_preds, weights="quadratic")

    # Compute conformal prediction sets
    cal_size = int(len(all_probs) * 0.5)
    cal_probs, cal_labels = all_probs[:cal_size], all_labels[:cal_size]
    test_probs, test_labels = all_probs[cal_size:], all_labels[cal_size:]

    cal_scores = 1 - cal_probs[np.arange(len(cal_labels)), cal_labels - 2]  # Adjust for 2-12 range
    qhat = np.quantile(cal_scores, 1 - alpha, method='higher')

    pred_sets = []
    for prob in test_probs:
        ps = [i + 2 for i, p in enumerate(prob) if 1 - p <= qhat]  # Adjust for 2-12 range
        if len(ps) == 0:
            ps = [np.argmax(prob) + 2]  # Adjust for 2-12 range
        pred_sets.append(ps)

    coverage = np.mean([label in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    return total_loss / len(dataloader), accuracy, f1, qwk, coverage, avg_set_size, uacc


def evaluate_ensemble(models, dataloader, alpha=0.1):
    all_preds = []
    all_probs = []
    all_labels = []

    for model in models:
        model.eval()
        model_preds = []
        model_probs = []
        model_labels = []
```

Figure 25: Code snippet 20



```
        with torch.no_grad():
            for batch in dataloader:
                inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
                labels = batch["labels"].to(device)
                outputs = model(**inputs)
                logits = outputs.logits[:, -1, :].to(torch.float32)
                probs = torch.softmax(logits, dim=-1)
                preds = torch.argmax(logits, dim=-1)
                model_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
                model_probs.extend(probs.cpu().numpy())
                model_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

        all_preds.append(model_preds)
        all_probs.append(model_probs)
        all_labels = model_labels  # All models use the same labels

    all_preds = np.array(all_preds)
    all_probs = np.array(all_probs)
    all_labels = np.array(all_labels)

    # Ensemble predictions
    ensemble_preds = np.mean(all_preds, axis=0).round().astype(int)
    ensemble_probs = np.mean(all_probs, axis=0)

    # Compute metrics
    accuracy = (ensemble_preds == all_labels).mean()
    f1 = f1_score(all_labels, ensemble_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, ensemble_preds, weights="quadratic")

    # Compute uncertainty
    pred_variance = np.var(all_preds, axis=0)
    avg_variance = np.mean(pred_variance)

    # Compute conformal prediction sets
    cal_size = int(len(ensemble_probs) * 0.5)
    cal_probs, cal_labels = ensemble_probs[:cal_size], all_labels[:cal_size]
    test_probs, test_labels = ensemble_probs[cal_size:], all_labels[cal_size:]

    cal_scores = 1 - cal_probs[np.arange(len(cal_labels)), cal_labels - 2]  # Adjust for 2-12 range
    qhat = np.quantile(cal_scores, 1 - alpha, method='higher')

    pred_sets = []
    for prob in test_probs:
        ps = [i + 2 for i, p in enumerate(prob) if 1 - p <= qhat]  # Adjust for 2-12 range
        if len(ps) == 0:
            ps = [np.argmax(prob) + 2]  # Adjust for 2-12 range
        pred_sets.append(ps)

    coverage = np.mean([label in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    # Additional ensemble metrics
    disagreement = np.mean(np.var(all_preds, axis=0))
    avg_entropy = np.mean([entropy(probs) for probs in ensemble_probs])

    return accuracy, f1, qwk, avg_variance, coverage, avg_set_size, uacc, disagreement, avg_entropy

# Main execution
num_models = 3  # Number of models in the ensemble
ensemble_models = create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader)
```

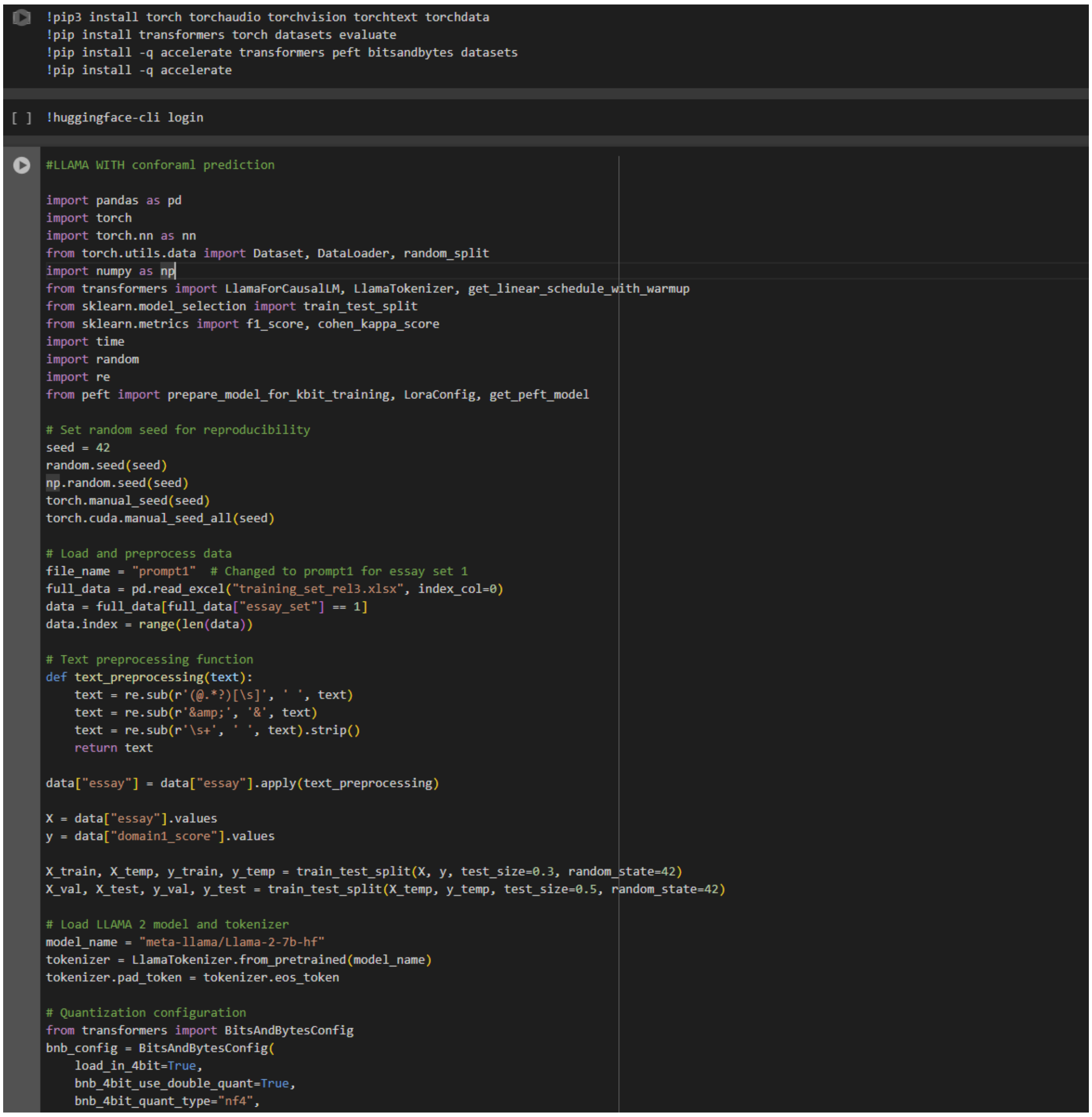
Figure 26: Code snippet 21



```
print("Evaluating individual models:")
for i, model in enumerate(ensemble_models):
    val_loss, val_acc, val_f1, val_qwk, val_coverage, val_set_size, val_uacc = evaluate_model(model, val_dataloader)
    print(f"Model {i+1} - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")
    print(f"Coverage: {val_coverage:.4f}, Avg Set Size: {val_set_size:.4f}, UACC: {val_uacc:.4f}")

print("\nEvaluating ensemble:")
ens_acc, ens_f1, ens_qwk, ens_var, ens_coverage, ens_set_size, ens_uacc, ens_disagreement, ens_entropy = evaluate_ensemble(ensemble_models, test_datal
print(f"Ensemble - Acc: {ens_acc:.4f}, F1: {ens_f1:.4f}, QWK: {ens_qwk:.4f}")
print(f"Uncertainty - Avg Variance: {ens_var:.4f}")
print(f"Conformal Prediction - Coverage: {ens_coverage:.4f}, Avg Set Size: {ens_set_size:.4f}, UACC: {ens_uacc:.4f}")
print(f"Disagreement: {ens_disagreement:.4f}, Avg Entropy: {ens_entropy:.4f}")

# Save results
results = {
    "ensemble_accuracy": ens_acc,
    "ensemble_f1": ens_f1,
    "ensemble_qwk": ens_qwk,
    "ensemble_uncertainty": ens_var,
    "conformal_coverage": ens_coverage,
    "conformal_set_size": ens_set_size,
    "uacc": ens_uacc,
    "disagreement": ens_disagreement,
    "avg_entropy": ens_entropy
}
pd.DataFrame([results]).to_csv(f"results_{file_name}_llama2_ensemble.csv", index=False)
print("Results saved to CSV file.")
```

```
generation_config.json: 100% 177/177 [00:00<00:00, 15.0kB/s]
Epoch 1/5, Loss: 4.4639
Validation - Loss: 2.0431, Acc: 0.4082, F1: 0.3652, QWK: 0.2184
Coverage: 0.9254, Avg Set Size: 6.8060, UACC: 0.5316
Epoch 2/5, Loss: 1.4849
Validation - Loss: 1.4503, Acc: 0.4082, F1: 0.3648, QWK: 0.6268
Coverage: 0.8881, Avg Set Size: 3.1716, UACC: 0.9999
Epoch 3/5, Loss: 1.2962
Validation - Loss: 1.3590, Acc: 0.4494, F1: 0.4287, QWK: 0.7099
Coverage: 0.8955, Avg Set Size: 2.8731, UACC: 1.1101
Epoch 4/5, Loss: 1.2222
Validation - Loss: 1.3273, Acc: 0.4869, F1: 0.4578, QWK: 0.7336
Coverage: 0.9403, Avg Set Size: 3.2090, UACC: 1.0903
Epoch 5/5, Loss: 1.1629
Validation - Loss: 1.3044, Acc: 0.4906, F1: 0.4679, QWK: 0.7423
Coverage: 0.9328, Avg Set Size: 2.9254, UACC: 1.1918
Training model 2/3
Loading checkpoint shards: 100% 4/4 [00:15<00:00, 3.39s/it]
Epoch 1/5, Loss: 4.5072
Validation - Loss: 2.0376, Acc: 0.3858, F1: 0.3118, QWK: 0.1983
Coverage: 0.9179, Avg Set Size: 6.5448, UACC: 0.5635
Epoch 2/5, Loss: 1.4588
Validation - Loss: 1.3866, Acc: 0.4307, F1: 0.4049, QWK: 0.7094
Coverage: 0.9328, Avg Set Size: 3.3134, UACC: 0.9826
Epoch 3/5, Loss: 1.2842
Validation - Loss: 1.3533, Acc: 0.4644, F1: 0.4360, QWK: 0.7066
Coverage: 0.9478, Avg Set Size: 3.3134, UACC: 1.0775
Epoch 4/5, Loss: 1.2020
Validation - Loss: 1.3244, Acc: 0.4794, F1: 0.4522, QWK: 0.7439
Coverage: 0.9328, Avg Set Size: 3.0224, UACC: 1.1187
Epoch 5/5, Loss: 1.1578
Validation - Loss: 1.3013, Acc: 0.4831, F1: 0.4526, QWK: 0.7527
Coverage: 0.9552, Avg Set Size: 3.2537, UACC: 1.0717
```

Figure 27: Code snippet 22

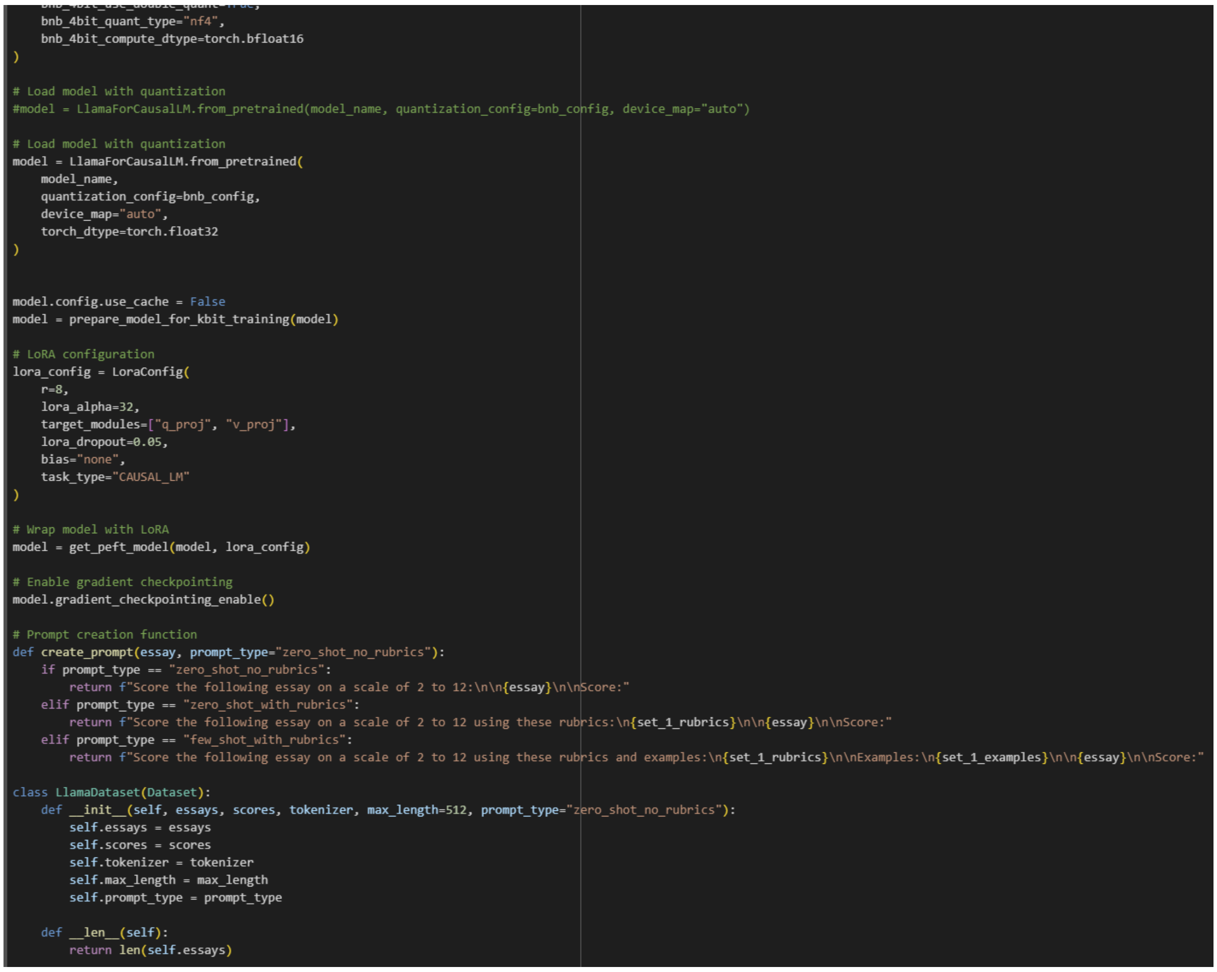


```
!pip3 install torch torchaudio torchvision torchtext torchdata
!pip install transformers torch datasets evaluate
!pip install -q accelerate transformers peft bitsandbytes datasets
!pip install -q accelerate
```

```
!huggingface-cli login
```

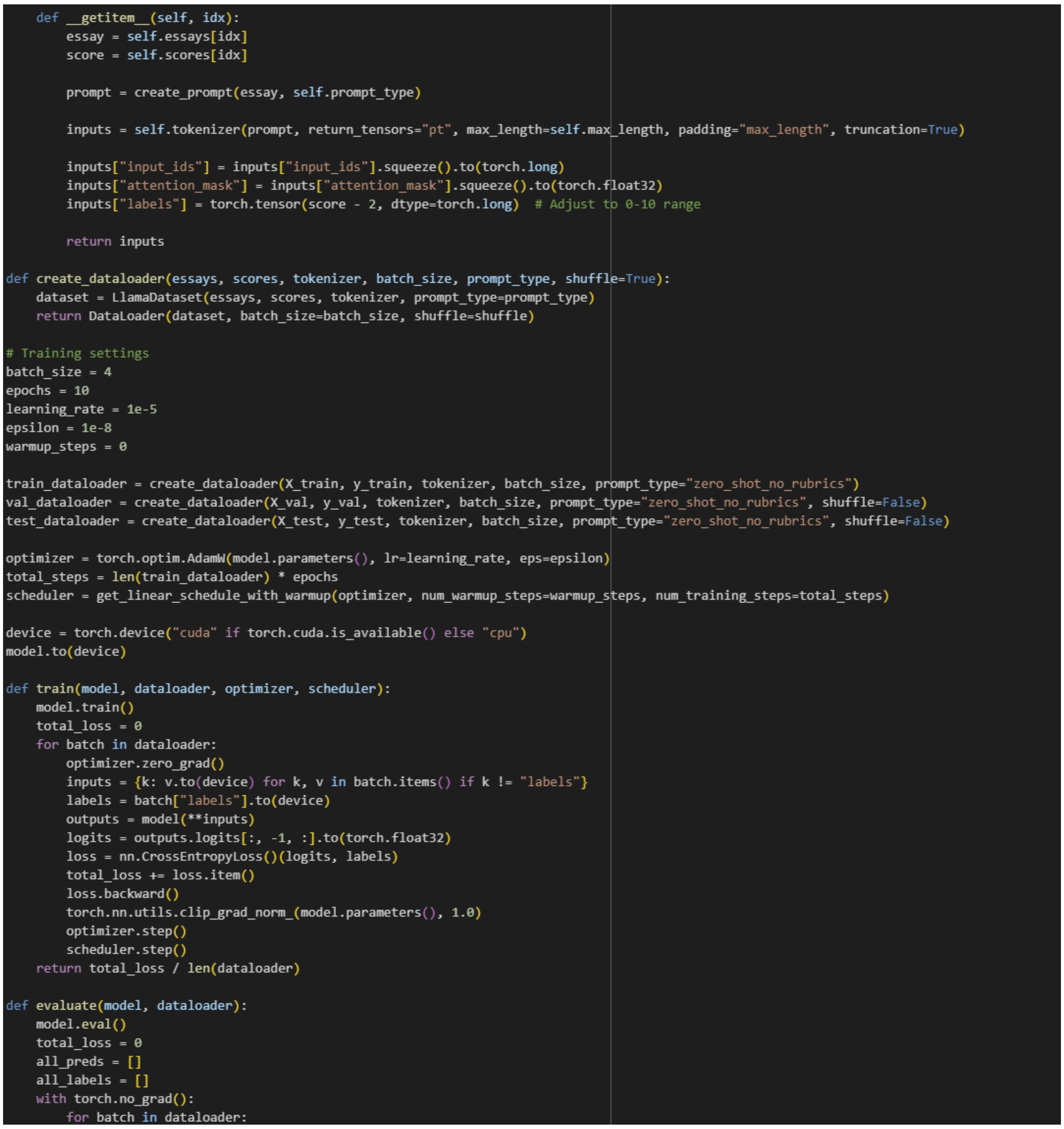
```
#LLAMA WITH conforaml prediction

import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
import numpy as np
from transformers import LlamaForCausalLM, LlamaTokenizer, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, cohen_kappa_score
import time
import random
import re
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model

# Set random seed for reproducibility
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

# Load and preprocess data
file_name = "prompt1"  # Changed to prompt1 for essay set 1
full_data = pd.read_excel("training_set_rel3.xlsx", index_col=0)
data = full_data[full_data["essay_set"] == 1]
data.index = range(len(data))

# Text preprocessing function
def text_preprocessing(text):
    text = re.sub(r'(@.*?)[\s]', ' ', text)
    text = re.sub(r'&amp;', '&', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

data["essay"] = data["essay"].apply(text_preprocessing)

X = data["essay"].values
y = data["domain1_score"].values

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Load LLAMA 2 model and tokenizer
model_name = "meta-llama/Llama-2-7b-hf"
tokenizer = LlamaTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token

# Quantization configuration
from transformers import BitsAndBytesConfig
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
```

Figure 28: Code snippet 23



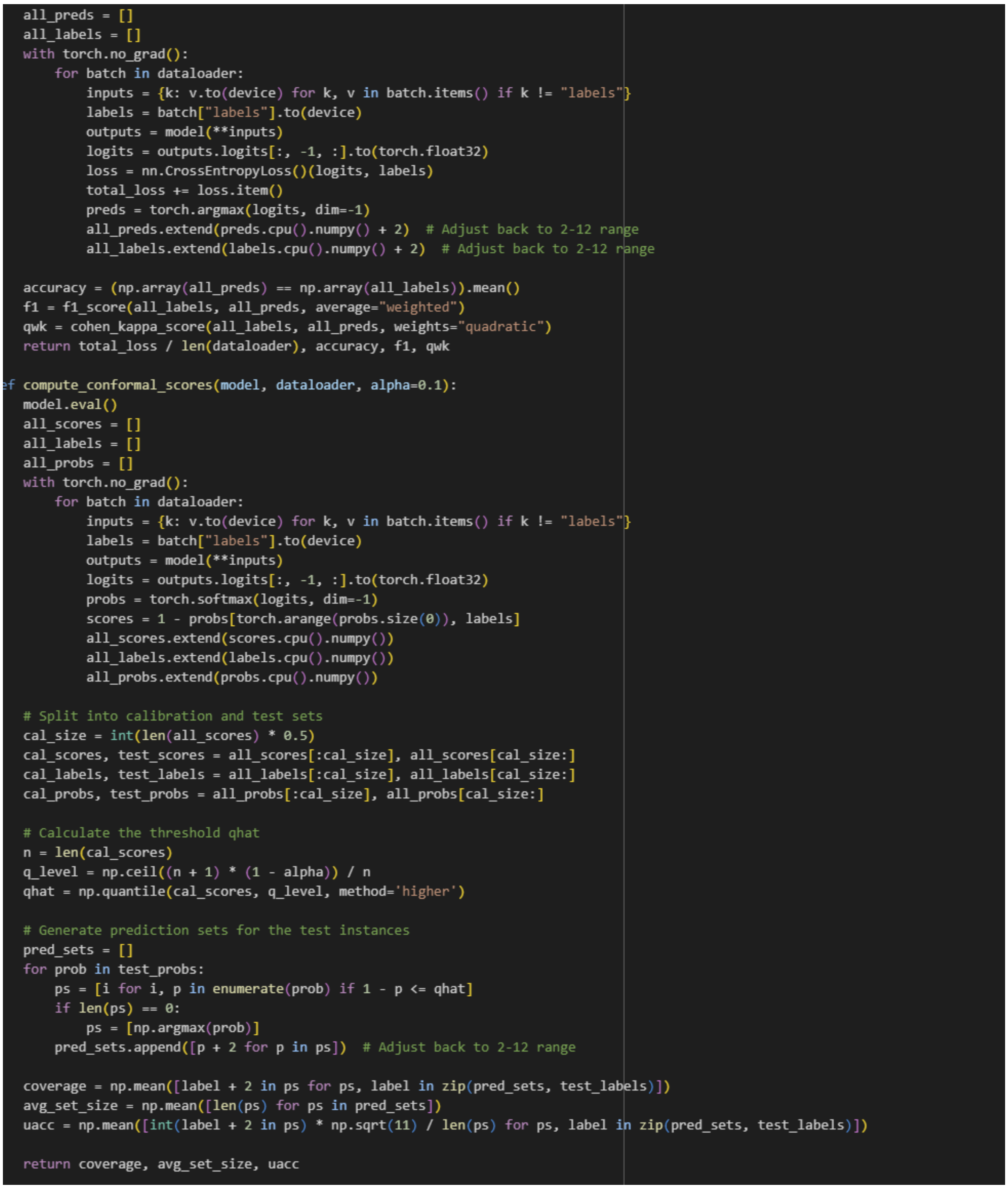
```
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

# Load model with quantization
#model = LlamaForCausalLM.from_pretrained(model_name, quantization_config=bnb_config, device_map="auto")

# Load model with quantization
model = LlamaForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    torch_dtype=torch.float32
)


model.config.use_cache = False
model = prepare_model_for_kbit_training(model)

# LoRA configuration
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

# Wrap model with LoRA
model = get_peft_model(model, lora_config)

# Enable gradient checkpointing
model.gradient_checkpointing_enable()

# Prompt creation function
def create_prompt(essay, prompt_type="zero_shot_no_rubrics"):
    if prompt_type == "zero_shot_no_rubrics":
        return f"Score the following essay on a scale of 2 to 12:\n\n{essay}\n\nScore:"
    elif prompt_type == "zero_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics:\n{set_1_rubrics}\n\n{essay}\n\nScore:"
    elif prompt_type == "few_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics and examples:\n{set_1_rubrics}\n\nExamples:\n{set_1_examples}\n\n{essay}\n\nScore:"

class LlamaDataset(Dataset):
    def __init__(self, essays, scores, tokenizer, max_length=512, prompt_type="zero_shot_no_rubrics"):
        self.essays = essays
        self.scores = scores
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.prompt_type = prompt_type

    def __len__(self):
        return len(self.essays)
```

Figure 29: Code snippet 24



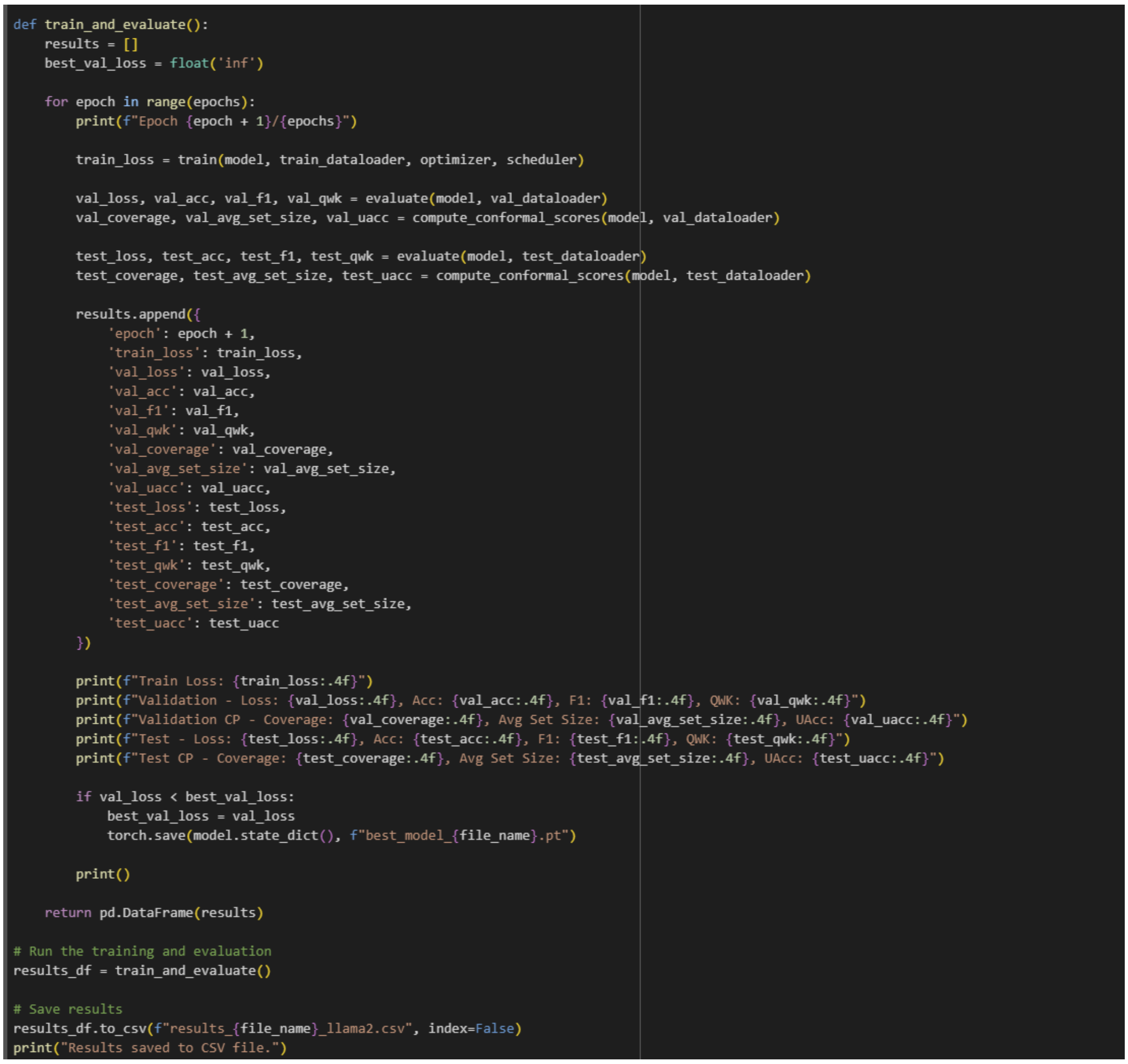
```python
    def __getitem__(self, idx):
        essay = self.essays[idx]
        score = self.scores[idx]

        prompt = create_prompt(essay, self.prompt_type)

        inputs = self.tokenizer(prompt, return_tensors="pt", max_length=self.max_length, padding="max_length", truncation=True)

        inputs["input_ids"] = inputs["input_ids"].squeeze().to(torch.long)
        inputs["attention_mask"] = inputs["attention_mask"].squeeze().to(torch.float32)
        inputs["labels"] = torch.tensor(score - 2, dtype=torch.long)  # Adjust to 0-10 range

        return inputs

def create_dataloader(essays, scores, tokenizer, batch_size, prompt_type, shuffle=True):
    dataset = LlamaDataset(essays, scores, tokenizer, prompt_type=prompt_type)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

# Training settings
batch_size = 4
epochs = 10
learning_rate = 1e-5
epsilon = 1e-8
warmup_steps = 0

train_dataloader = create_dataloader(X_train, y_train, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics")
val_dataloader = create_dataloader(X_val, y_val, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)
test_dataloader = create_dataloader(X_test, y_test, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=total_steps)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

def train(model, dataloader, optimizer, scheduler):
    model.train()
    total_loss = 0
    for batch in dataloader:
        optimizer.zero_grad()
        inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
        labels = batch["labels"].to(device)
        outputs = model(**inputs)
        logits = outputs.logits[:, -1, :].to(torch.float32)
        loss = nn.CrossEntropyLoss()(logits, labels)
        total_loss += loss.item()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        scheduler.step()
    return total_loss / len(dataloader)

def evaluate(model, dataloader):
    model.eval()
    total_loss = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in dataloader:
```

Figure 30: Code snippet 25



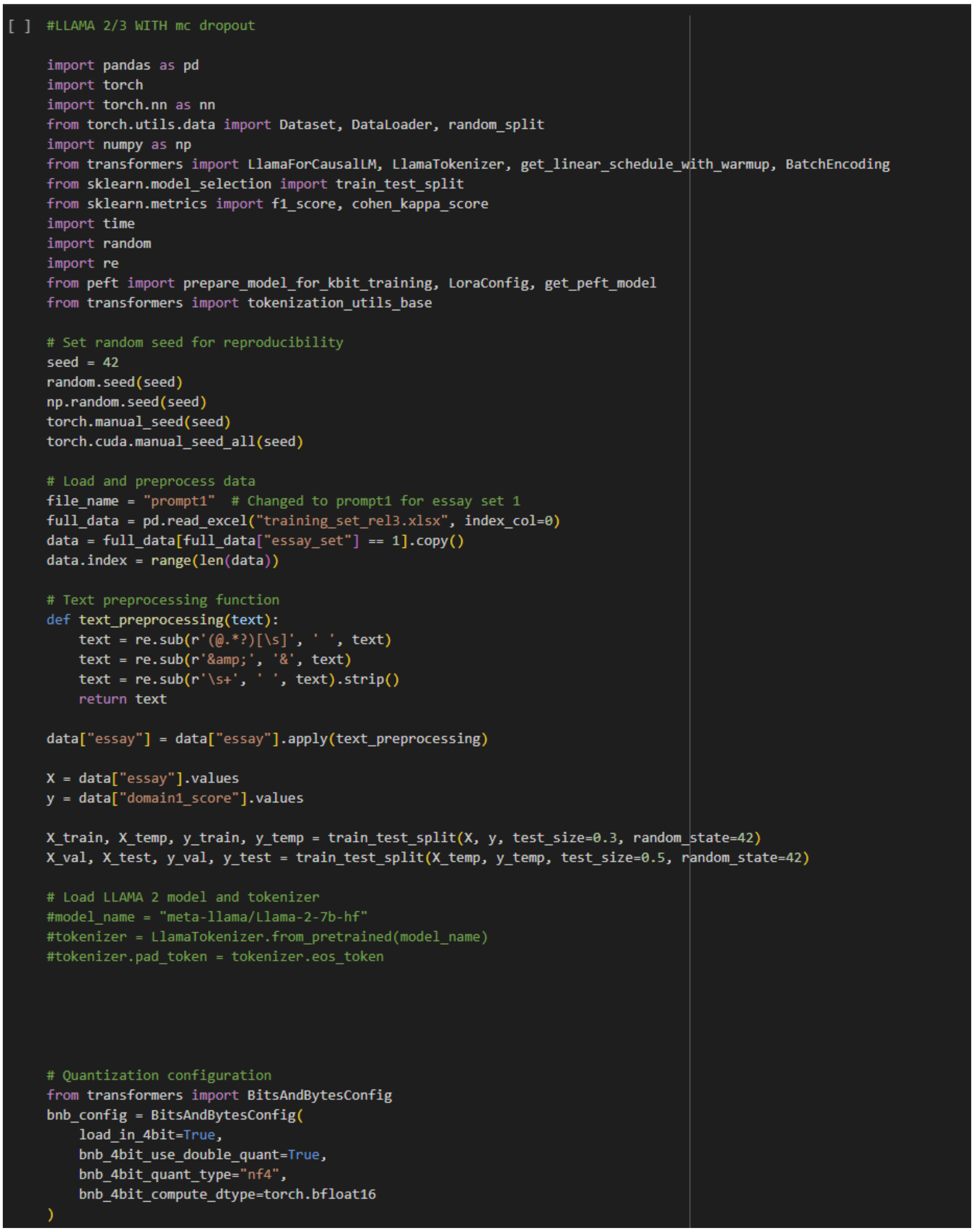
```
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            preds = torch.argmax(logits, dim=-1)
            all_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
            all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

    accuracy = (np.array(all_preds) == np.array(all_labels)).mean()
    f1 = f1_score(all_labels, all_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, all_preds, weights="quadratic")
    return total_loss / len(dataloader), accuracy, f1, qwk

ef compute_conformal_scores(model, dataloader, alpha=0.1):
    model.eval()
    all_scores = []
    all_labels = []
    all_probs = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            probs = torch.softmax(logits, dim=-1)
            scores = 1 - probs[torch.arange(probs.size(0)), labels]
            all_scores.extend(scores.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_probs.extend(probs.cpu().numpy())

    # Split into calibration and test sets
    cal_size = int(len(all_scores) * 0.5)
    cal_scores, test_scores = all_scores[:cal_size], all_scores[cal_size:]
    cal_labels, test_labels = all_labels[:cal_size], all_labels[cal_size:]
    cal_probs, test_probs = all_probs[:cal_size], all_probs[cal_size:]

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        ps = [i for i, p in enumerate(prob) if 1 - p <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append([p + 2 for p in ps])  # Adjust back to 2-12 range

    coverage = np.mean([label + 2 in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label + 2 in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    return coverage, avg_set_size, uacc
```

Figure 31: Code snippet 26



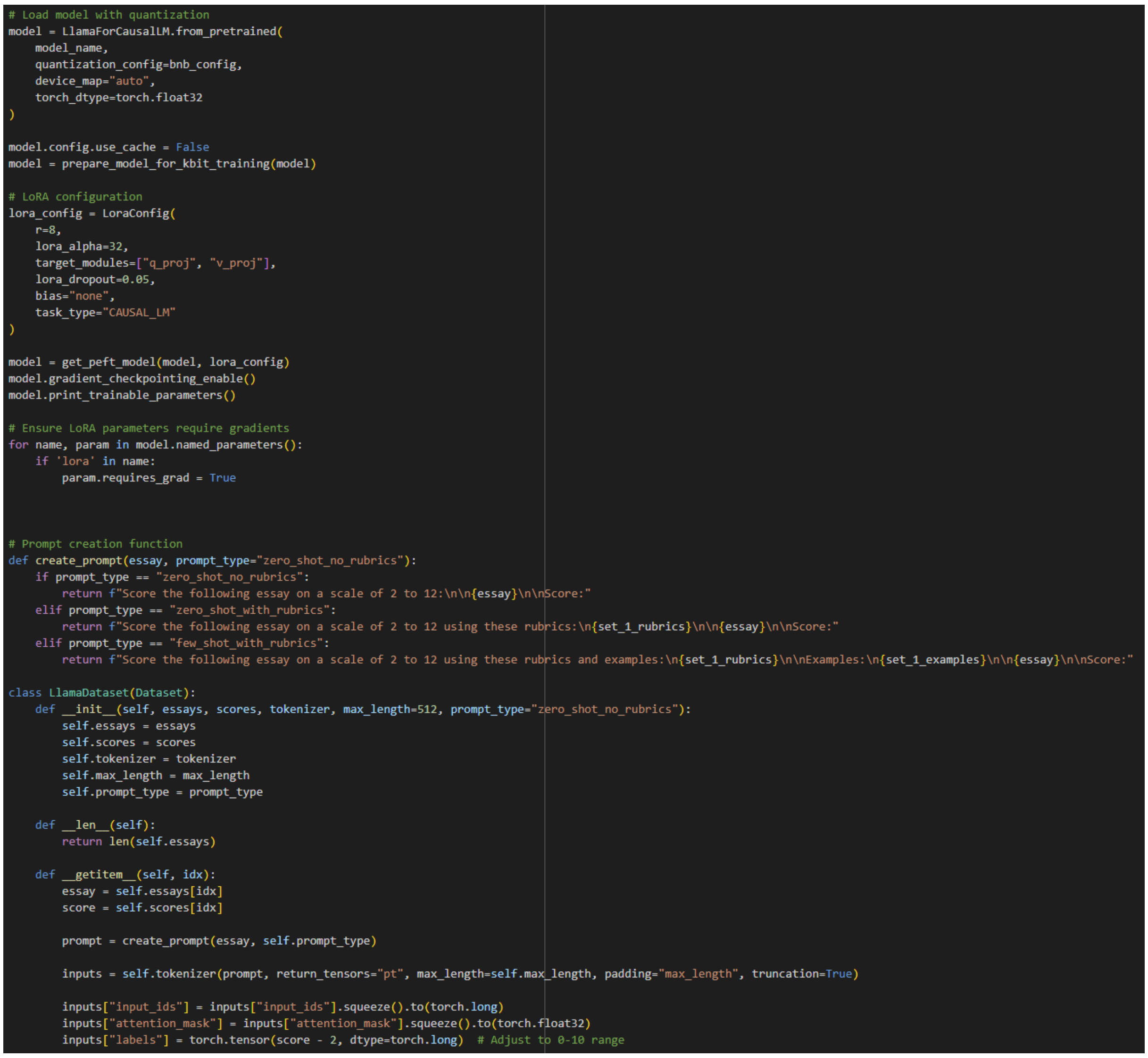
```python
def train_and_evaluate():
    results = []
    best_val_loss = float('inf')

    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")

        train_loss = train(model, train_dataloader, optimizer, scheduler)

        val_loss, val_acc, val_f1, val_qwk = evaluate(model, val_dataloader)
        val_coverage, val_avg_set_size, val_uacc = compute_conformal_scores(model, val_dataloader)

        test_loss, test_acc, test_f1, test_qwk = evaluate(model, test_dataloader)
        test_coverage, test_avg_set_size, test_uacc = compute_conformal_scores(model, test_dataloader)

        results.append({
            'epoch': epoch + 1,
            'train_loss': train_loss,
            'val_loss': val_loss,
            'val_acc': val_acc,
            'val_f1': val_f1,
            'val_qwk': val_qwk,
            'val_coverage': val_coverage,
            'val_avg_set_size': val_avg_set_size,
            'val_uacc': val_uacc,
            'test_loss': test_loss,
            'test_acc': test_acc,
            'test_f1': test_f1,
            'test_qwk': test_qwk,
            'test_coverage': test_coverage,
            'test_avg_set_size': test_avg_set_size,
            'test_uacc': test_uacc
        })

        print(f"Train Loss: {train_loss:.4f}")
        print(f"Validation - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")
        print(f"Validation CP - Coverage: {val_coverage:.4f}, Avg Set Size: {val_avg_set_size:.4f}, UAcc: {val_uacc:.4f}")
        print(f"Test - Loss: {test_loss:.4f}, Acc: {test_acc:.4f}, F1: {test_f1:.4f}, QWK: {test_qwk:.4f}")
        print(f"Test CP - Coverage: {test_coverage:.4f}, Avg Set Size: {test_avg_set_size:.4f}, UAcc: {test_uacc:.4f}")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), f"best_model_{file_name}.pt")

        print()

    return pd.DataFrame(results)

# Run the training and evaluation
results_df = train_and_evaluate()

# Save results
results_df.to_csv(f"results_{file_name}_llama2.csv", index=False)
print("Results saved to CSV file.")
```

Figure 32: Code snippet 27



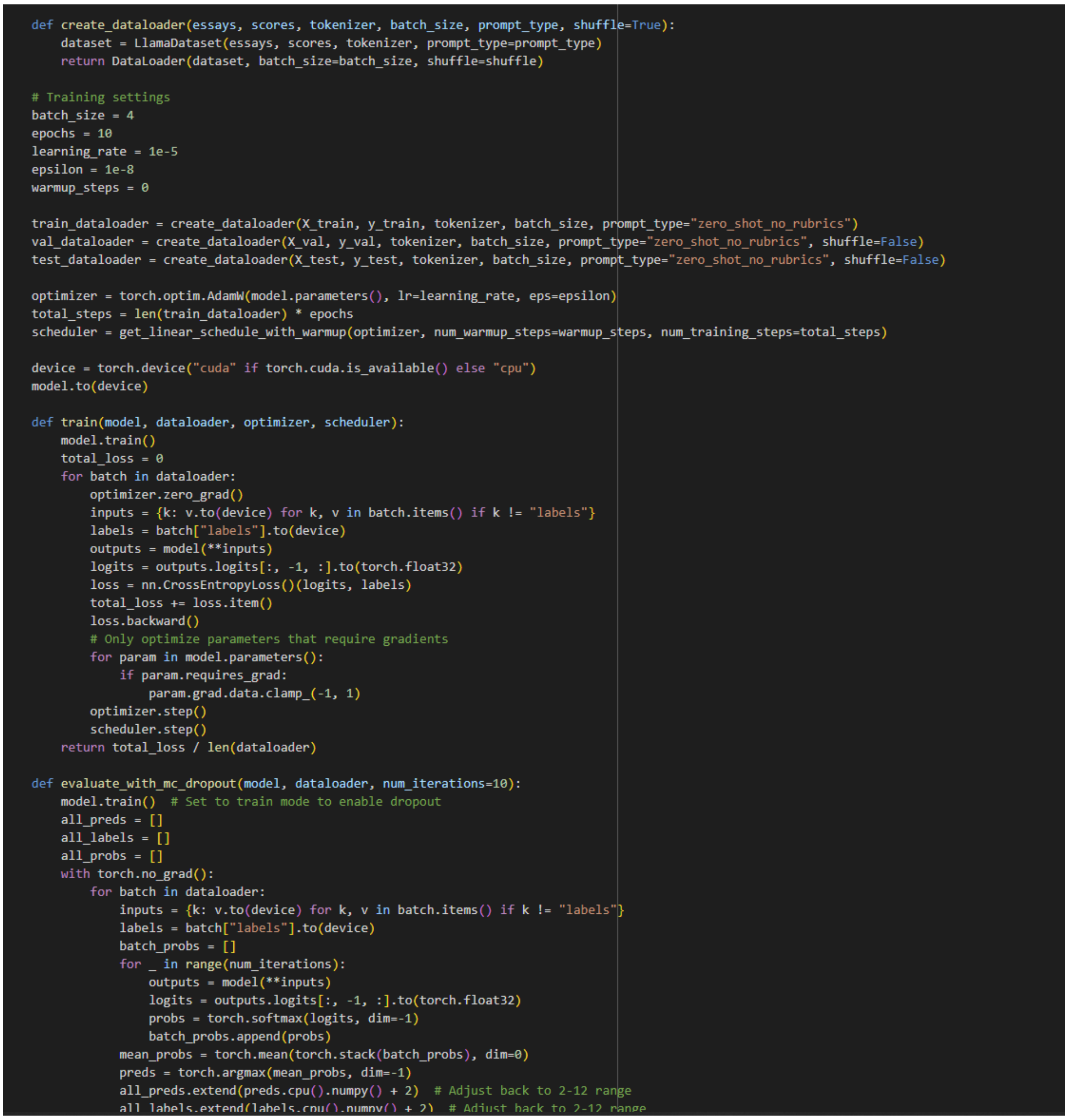
```
[ ]  #LLAMA 2/3 WITH mc dropout

     import pandas as pd
     import torch
     import torch.nn as nn
     from torch.utils.data import Dataset, DataLoader, random_split
     import numpy as np
     from transformers import LlamaForCausalLM, LlamaTokenizer, get_linear_schedule_with_warmup, BatchEncoding
     from sklearn.model_selection import train_test_split
     from sklearn.metrics import f1_score, cohen_kappa_score
     import time
     import random
     import re
     from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model
     from transformers import tokenization_utils_base

     # Set random seed for reproducibility
     seed = 42
     random.seed(seed)
     np.random.seed(seed)
     torch.manual_seed(seed)
     torch.cuda.manual_seed_all(seed)

     # Load and preprocess data
     file_name = "prompt1"  # Changed to prompt1 for essay set 1
     full_data = pd.read_excel("training_set_rel3.xlsx", index_col=0)
     data = full_data[full_data["essay_set"] == 1].copy()
     data.index = range(len(data))

     # Text preprocessing function
     def text_preprocessing(text):
         text = re.sub(r'(@.*?)[\s]', ' ', text)
         text = re.sub(r'&amp;', '&', text)
         text = re.sub(r'\s+', ' ', text).strip()
         return text

     data["essay"] = data["essay"].apply(text_preprocessing)

     X = data["essay"].values
     y = data["domain1_score"].values

     X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
     X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

     # Load LLAMA 2 model and tokenizer
     #model_name = "meta-llama/Llama-2-7b-hf"
     #tokenizer = LlamaTokenizer.from_pretrained(model_name)
     #tokenizer.pad_token = tokenizer.eos_token




     # Quantization configuration
     from transformers import BitsAndBytesConfig
     bnb_config = BitsAndBytesConfig(
         load_in_4bit=True,
         bnb_4bit_use_double_quant=True,
         bnb_4bit_quant_type="nf4",
         bnb_4bit_compute_dtype=torch.bfloat16
     )
```

Figure 33: Code snippet 28



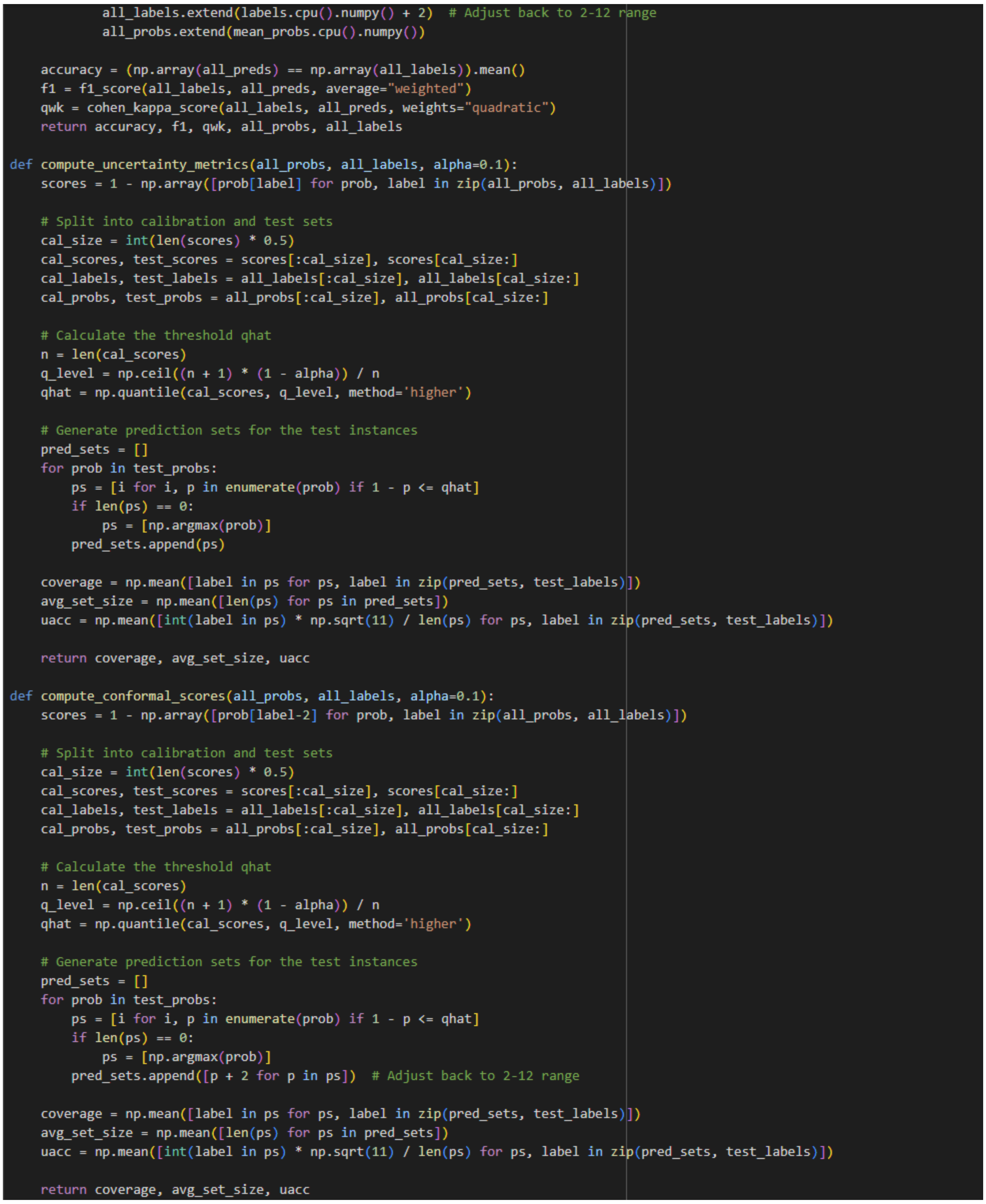
```python
# Load model with quantization
model = LlamaForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    torch_dtype=torch.float32
)

model.config.use_cache = False
model = prepare_model_for_kbit_training(model)

# LoRA configuration
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

model = get_peft_model(model, lora_config)
model.gradient_checkpointing_enable()
model.print_trainable_parameters()

# Ensure LoRA parameters require gradients
for name, param in model.named_parameters():
    if 'lora' in name:
        param.requires_grad = True



# Prompt creation function
def create_prompt(essay, prompt_type="zero_shot_no_rubrics"):
    if prompt_type == "zero_shot_no_rubrics":
        return f"Score the following essay on a scale of 2 to 12:\n\n{essay}\n\nScore:"
    elif prompt_type == "zero_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics:\n{set_1_rubrics}\n\n{essay}\n\nScore:"
    elif prompt_type == "few_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics and examples:\n{set_1_rubrics}\n\nExamples:\n{set_1_examples}\n\n{essay}\n\nScore:"

class LlamaDataset(Dataset):
    def __init__(self, essays, scores, tokenizer, max_length=512, prompt_type="zero_shot_no_rubrics"):
        self.essays = essays
        self.scores = scores
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.prompt_type = prompt_type

    def __len__(self):
        return len(self.essays)

    def __getitem__(self, idx):
        essay = self.essays[idx]
        score = self.scores[idx]

        prompt = create_prompt(essay, self.prompt_type)

        inputs = self.tokenizer(prompt, return_tensors="pt", max_length=self.max_length, padding="max_length", truncation=True)

        inputs["input_ids"] = inputs["input_ids"].squeeze().to(torch.long)
        inputs["attention_mask"] = inputs["attention_mask"].squeeze().to(torch.float32)
        inputs["labels"] = torch.tensor(score - 2, dtype=torch.long)  # Adjust to 0-10 range
```

Figure 34: Code snippet 29



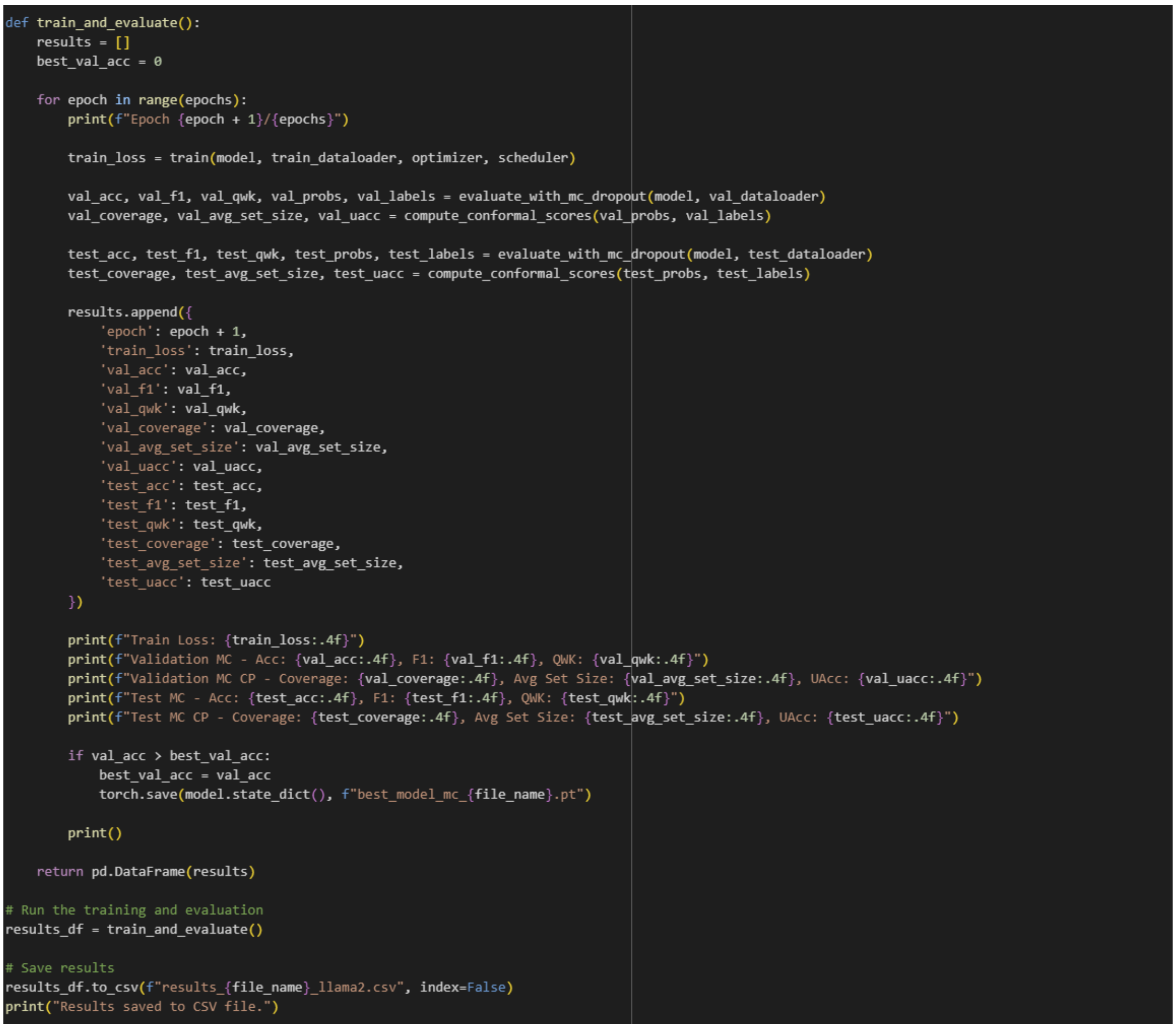
```python
def create_dataloader(essays, scores, tokenizer, batch_size, prompt_type, shuffle=True):
    dataset = LlamaDataset(essays, scores, tokenizer, prompt_type=prompt_type)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

# Training settings
batch_size = 4
epochs = 10
learning_rate = 1e-5
epsilon = 1e-8
warmup_steps = 0

train_dataloader = create_dataloader(X_train, y_train, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics")
val_dataloader = create_dataloader(X_val, y_val, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)
test_dataloader = create_dataloader(X_test, y_test, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=total_steps)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

def train(model, dataloader, optimizer, scheduler):
    model.train()
    total_loss = 0
    for batch in dataloader:
        optimizer.zero_grad()
        inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
        labels = batch["labels"].to(device)
        outputs = model(**inputs)
        logits = outputs.logits[:, -1, :].to(torch.float32)
        loss = nn.CrossEntropyLoss()(logits, labels)
        total_loss += loss.item()
        loss.backward()
        # Only optimize parameters that require gradients
        for param in model.parameters():
            if param.requires_grad:
                param.grad.data.clamp_(-1, 1)
        optimizer.step()
        scheduler.step()
    return total_loss / len(dataloader)

def evaluate_with_mc_dropout(model, dataloader, num_iterations=10):
    model.train()  # Set to train mode to enable dropout
    all_preds = []
    all_labels = []
    all_probs = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            batch_probs = []
            for _ in range(num_iterations):
                outputs = model(**inputs)
                logits = outputs.logits[:, -1, :].to(torch.float32)
                probs = torch.softmax(logits, dim=-1)
                batch_probs.append(probs)
            mean_probs = torch.mean(torch.stack(batch_probs), dim=0)
            preds = torch.argmax(mean_probs, dim=-1)
            all_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
            all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range
```

Figure 35: Code snippet 30



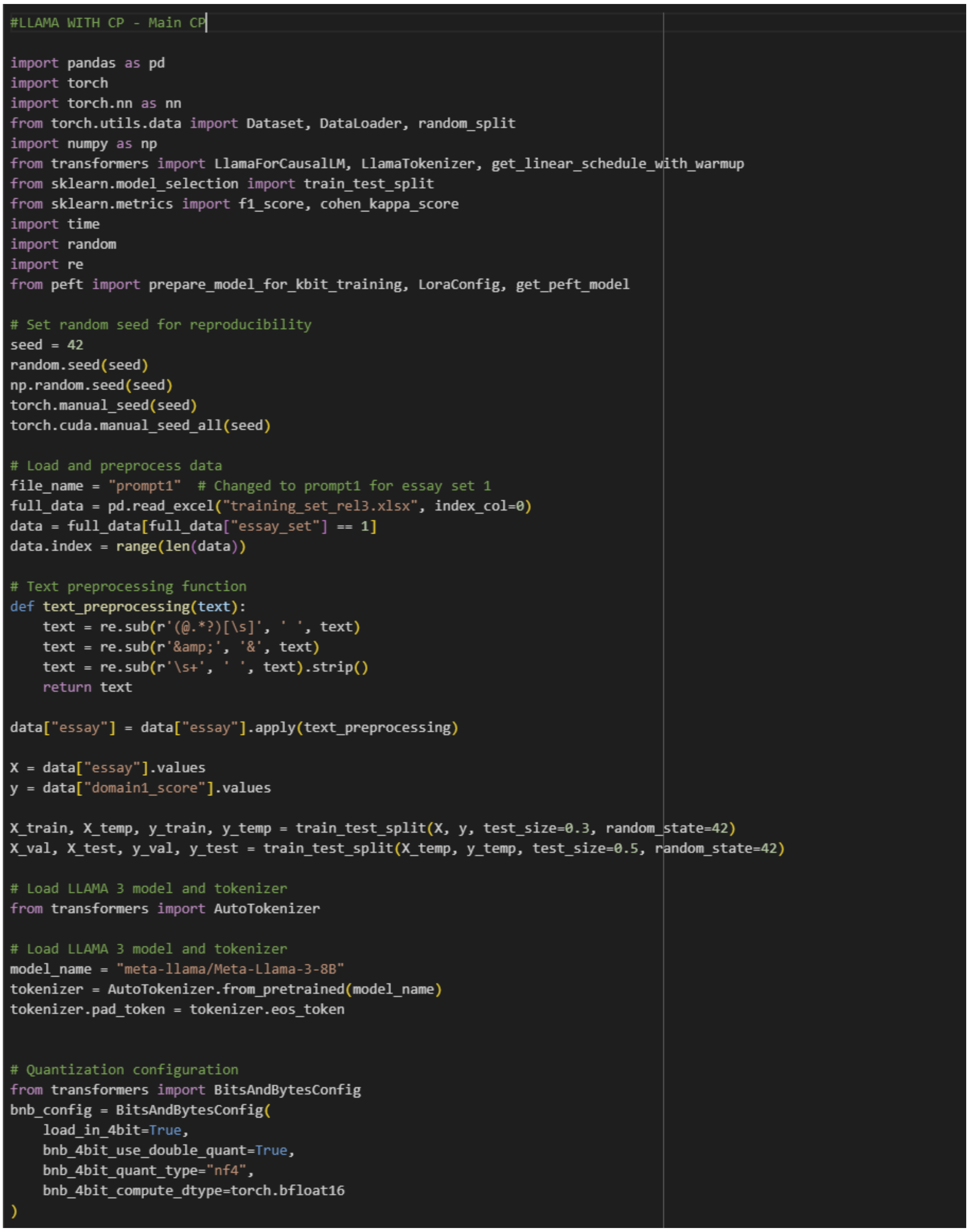
```
        all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range
        all_probs.extend(mean_probs.cpu().numpy())

    accuracy = (np.array(all_preds) == np.array(all_labels)).mean()
    f1 = f1_score(all_labels, all_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, all_preds, weights="quadratic")
    return accuracy, f1, qwk, all_probs, all_labels

def compute_uncertainty_metrics(all_probs, all_labels, alpha=0.1):
    scores = 1 - np.array([prob[label] for prob, label in zip(all_probs, all_labels)])

    # Split into calibration and test sets
    cal_size = int(len(scores) * 0.5)
    cal_scores, test_scores = scores[:cal_size], scores[cal_size:]
    cal_labels, test_labels = all_labels[:cal_size], all_labels[cal_size:]
    cal_probs, test_probs = all_probs[:cal_size], all_probs[cal_size:]

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        ps = [i for i, p in enumerate(prob) if 1 - p <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append(ps)

    coverage = np.mean([label in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    return coverage, avg_set_size, uacc

def compute_conformal_scores(all_probs, all_labels, alpha=0.1):
    scores = 1 - np.array([prob[label-2] for prob, label in zip(all_probs, all_labels)])

    # Split into calibration and test sets
    cal_size = int(len(scores) * 0.5)
    cal_scores, test_scores = scores[:cal_size], scores[cal_size:]
    cal_labels, test_labels = all_labels[:cal_size], all_labels[cal_size:]
    cal_probs, test_probs = all_probs[:cal_size], all_probs[cal_size:]

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        ps = [i for i, p in enumerate(prob) if 1 - p <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append([p + 2 for p in ps])  # Adjust back to 2-12 range

    coverage = np.mean([label in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    return coverage, avg_set_size, uacc
```

Figure 36: Code snippet 31



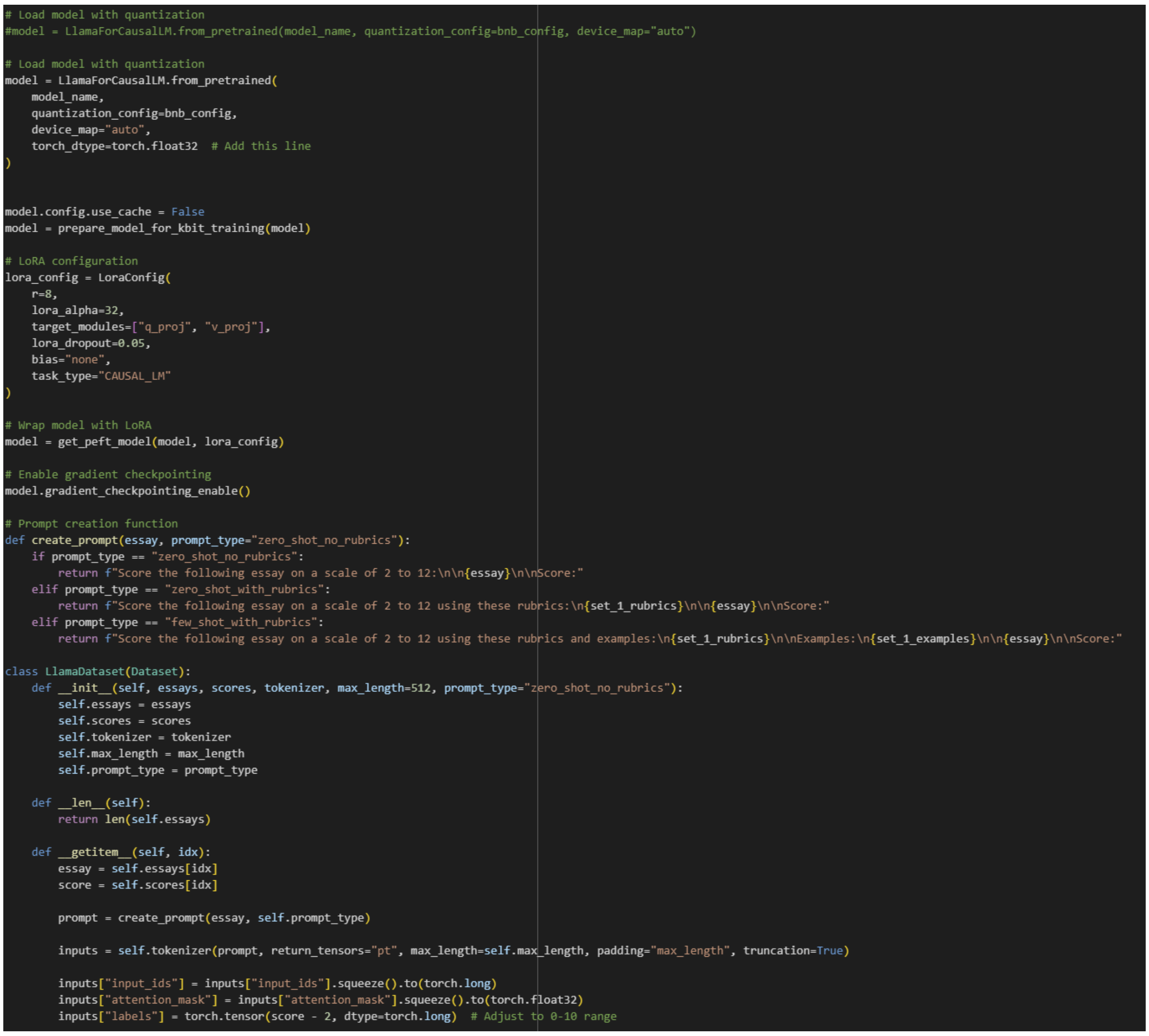
```python
def train_and_evaluate():
    results = []
    best_val_acc = 0

    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")

        train_loss = train(model, train_dataloader, optimizer, scheduler)

        val_acc, val_f1, val_qwk, val_probs, val_labels = evaluate_with_mc_dropout(model, val_dataloader)
        val_coverage, val_avg_set_size, val_uacc = compute_conformal_scores(val_probs, val_labels)

        test_acc, test_f1, test_qwk, test_probs, test_labels = evaluate_with_mc_dropout(model, test_dataloader)
        test_coverage, test_avg_set_size, test_uacc = compute_conformal_scores(test_probs, test_labels)

        results.append({
            'epoch': epoch + 1,
            'train_loss': train_loss,
            'val_acc': val_acc,
            'val_f1': val_f1,
            'val_qwk': val_qwk,
            'val_coverage': val_coverage,
            'val_avg_set_size': val_avg_set_size,
            'val_uacc': val_uacc,
            'test_acc': test_acc,
            'test_f1': test_f1,
            'test_qwk': test_qwk,
            'test_coverage': test_coverage,
            'test_avg_set_size': test_avg_set_size,
            'test_uacc': test_uacc
        })

        print(f"Train Loss: {train_loss:.4f}")
        print(f"Validation MC - Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")
        print(f"Validation MC CP - Coverage: {val_coverage:.4f}, Avg Set Size: {val_avg_set_size:.4f}, UAcc: {val_uacc:.4f}")
        print(f"Test MC - Acc: {test_acc:.4f}, F1: {test_f1:.4f}, QWK: {test_qwk:.4f}")
        print(f"Test MC CP - Coverage: {test_coverage:.4f}, Avg Set Size: {test_avg_set_size:.4f}, UAcc: {test_uacc:.4f}")

        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), f"best_model_mc_{file_name}.pt")

        print()

    return pd.DataFrame(results)

# Run the training and evaluation
results_df = train_and_evaluate()

# Save results
results_df.to_csv(f"results_{file_name}_llama2.csv", index=False)
print("Results saved to CSV file.")
```

Figure 37: Code snippet 32



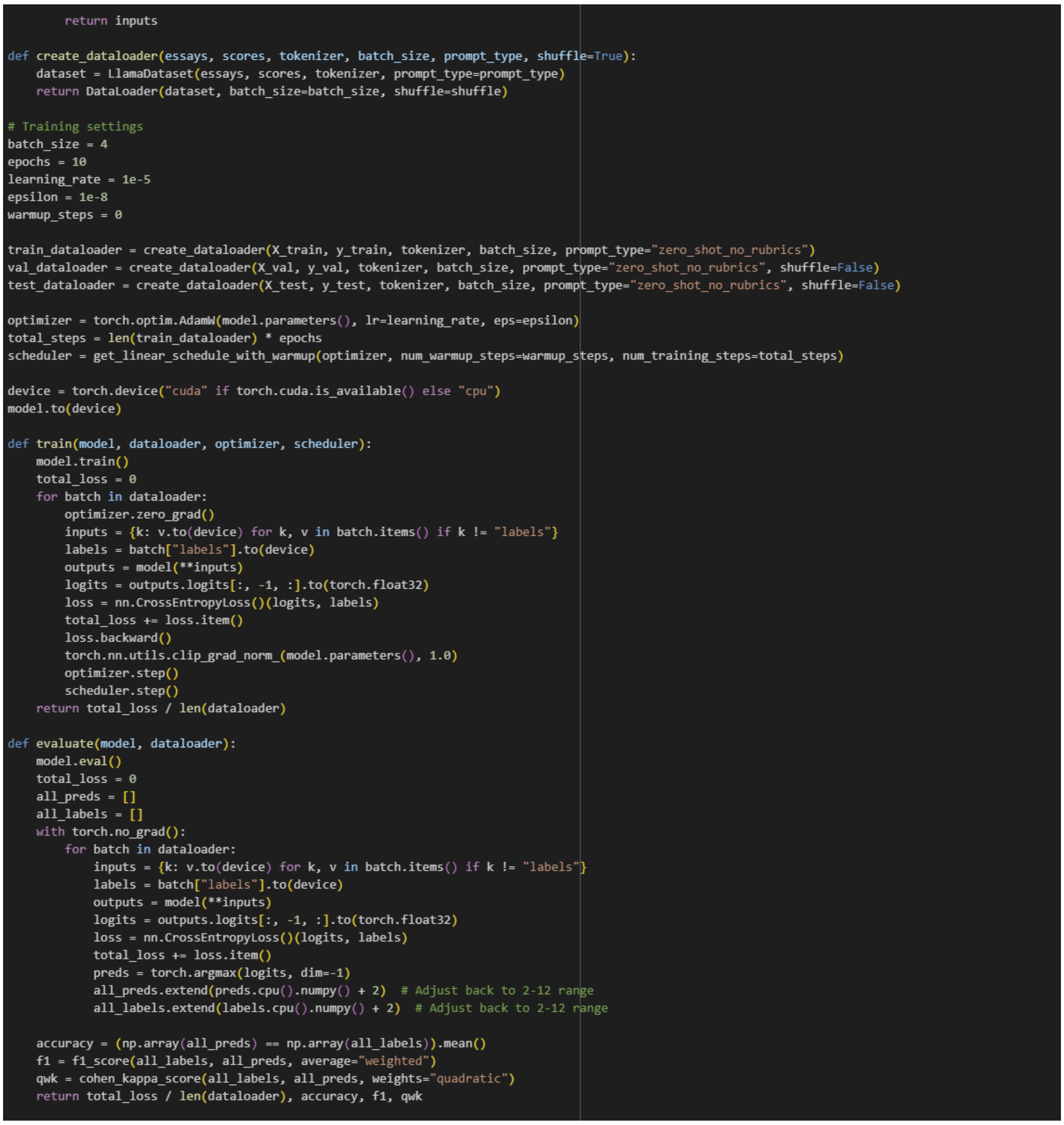
```python
#LLAMA WITH CP - Main CP

import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
import numpy as np
from transformers import LlamaForCausalLM, LlamaTokenizer, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, cohen_kappa_score
import time
import random
import re
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model

# Set random seed for reproducibility
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

# Load and preprocess data
file_name = "prompt1"  # Changed to prompt1 for essay set 1
full_data = pd.read_excel("training_set_rel3.xlsx", index_col=0)
data = full_data[full_data["essay_set"] == 1]
data.index = range(len(data))

# Text preprocessing function
def text_preprocessing(text):
    text = re.sub(r'(@.*?)[\s]', ' ', text)
    text = re.sub(r'&amp;', '&', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

data["essay"] = data["essay"].apply(text_preprocessing)

X = data["essay"].values
y = data["domain1_score"].values

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Load LLAMA 3 model and tokenizer
from transformers import AutoTokenizer

# Load LLAMA 3 model and tokenizer
model_name = "meta-llama/Meta-Llama-3-8B"
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token


# Quantization configuration
from transformers import BitsAndBytesConfig
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

Figure 38: Code snippet 33



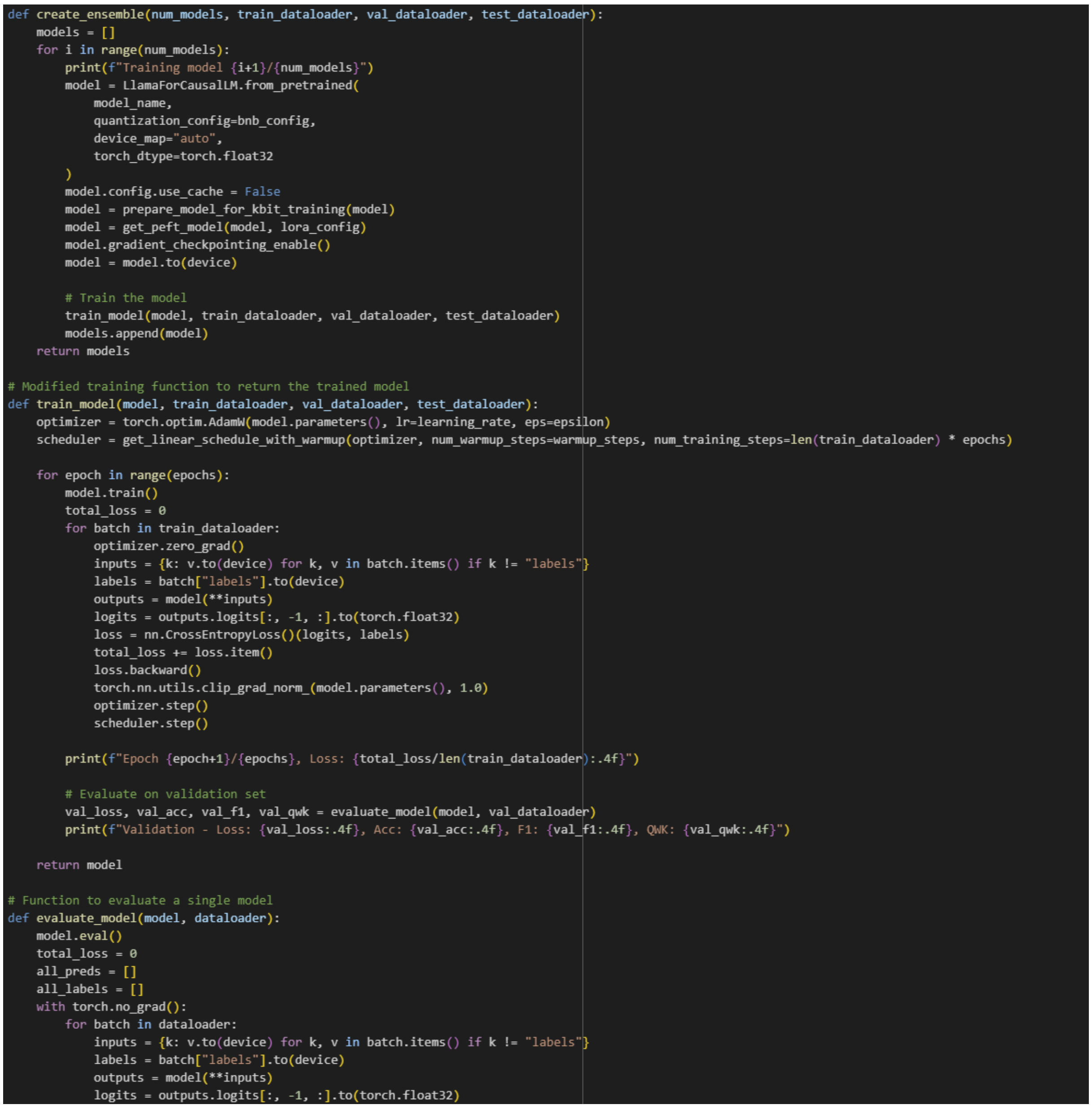
```python
# Load model with quantization
#model = LlamaForCausalLM.from_pretrained(model_name, quantization_config=bnb_config, device_map="auto")

# Load model with quantization
model = LlamaForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto",
    torch_dtype=torch.float32  # Add this line
)


model.config.use_cache = False
model = prepare_model_for_kbit_training(model)

# LoRA configuration
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

# Wrap model with LoRA
model = get_peft_model(model, lora_config)

# Enable gradient checkpointing
model.gradient_checkpointing_enable()

# Prompt creation function
def create_prompt(essay, prompt_type="zero_shot_no_rubrics"):
    if prompt_type == "zero_shot_no_rubrics":
        return f"Score the following essay on a scale of 2 to 12:\n\n{essay}\n\nScore:"
    elif prompt_type == "zero_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics:\n{set_1_rubrics}\n\n{essay}\n\nScore:"
    elif prompt_type == "few_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics and examples:\n{set_1_rubrics}\n\nExamples:\n{set_1_examples}\n\n{essay}\n\nScore:"

class LlamaDataset(Dataset):
    def __init__(self, essays, scores, tokenizer, max_length=512, prompt_type="zero_shot_no_rubrics"):
        self.essays = essays
        self.scores = scores
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.prompt_type = prompt_type

    def __len__(self):
        return len(self.essays)

    def __getitem__(self, idx):
        essay = self.essays[idx]
        score = self.scores[idx]

        prompt = create_prompt(essay, self.prompt_type)

        inputs = self.tokenizer(prompt, return_tensors="pt", max_length=self.max_length, padding="max_length", truncation=True)

        inputs["input_ids"] = inputs["input_ids"].squeeze().to(torch.long)
        inputs["attention_mask"] = inputs["attention_mask"].squeeze().to(torch.float32)
        inputs["labels"] = torch.tensor(score - 2, dtype=torch.long)  # Adjust to 0-10 range
```

Figure 39: Code snippet 34



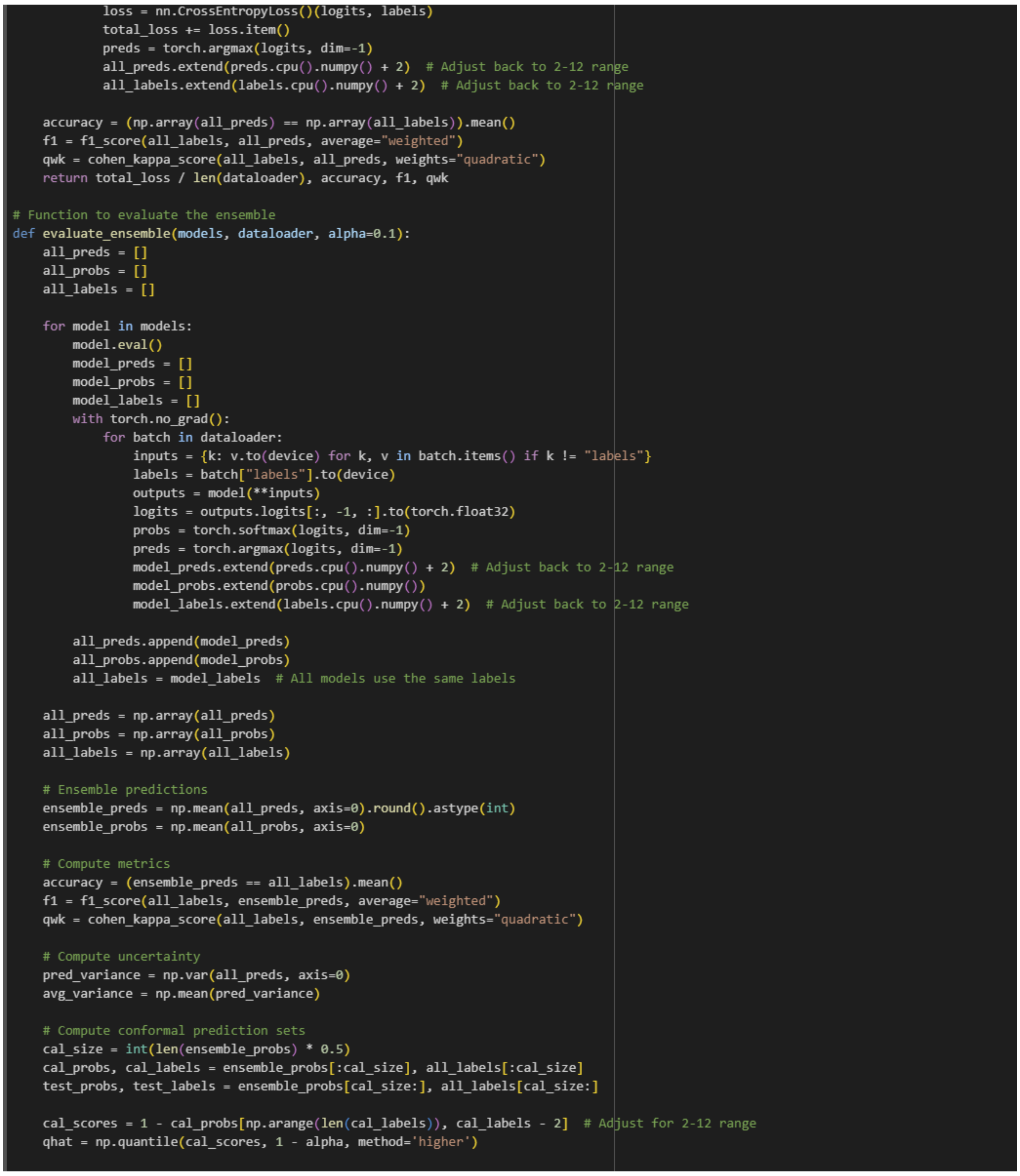
```python
        return inputs

def create_dataloader(essays, scores, tokenizer, batch_size, prompt_type, shuffle=True):
    dataset = LlamaDataset(essays, scores, tokenizer, prompt_type=prompt_type)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

# Training settings
batch_size = 4
epochs = 10
learning_rate = 1e-5
epsilon = 1e-8
warmup_steps = 0

train_dataloader = create_dataloader(X_train, y_train, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics")
val_dataloader = create_dataloader(X_val, y_val, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)
test_dataloader = create_dataloader(X_test, y_test, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
total_steps = len(train_dataloader) * epochs
scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=total_steps)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

def train(model, dataloader, optimizer, scheduler):
    model.train()
    total_loss = 0
    for batch in dataloader:
        optimizer.zero_grad()
        inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
        labels = batch["labels"].to(device)
        outputs = model(**inputs)
        logits = outputs.logits[:, -1, :].to(torch.float32)
        loss = nn.CrossEntropyLoss()(logits, labels)
        total_loss += loss.item()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        scheduler.step()
    return total_loss / len(dataloader)

def evaluate(model, dataloader):
    model.eval()
    total_loss = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            preds = torch.argmax(logits, dim=-1)
            all_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
            all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

    accuracy = (np.array(all_preds) == np.array(all_labels)).mean()
    f1 = f1_score(all_labels, all_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, all_preds, weights="quadratic")
    return total_loss / len(dataloader), accuracy, f1, qwk
```

Figure 40: Code snippet 35



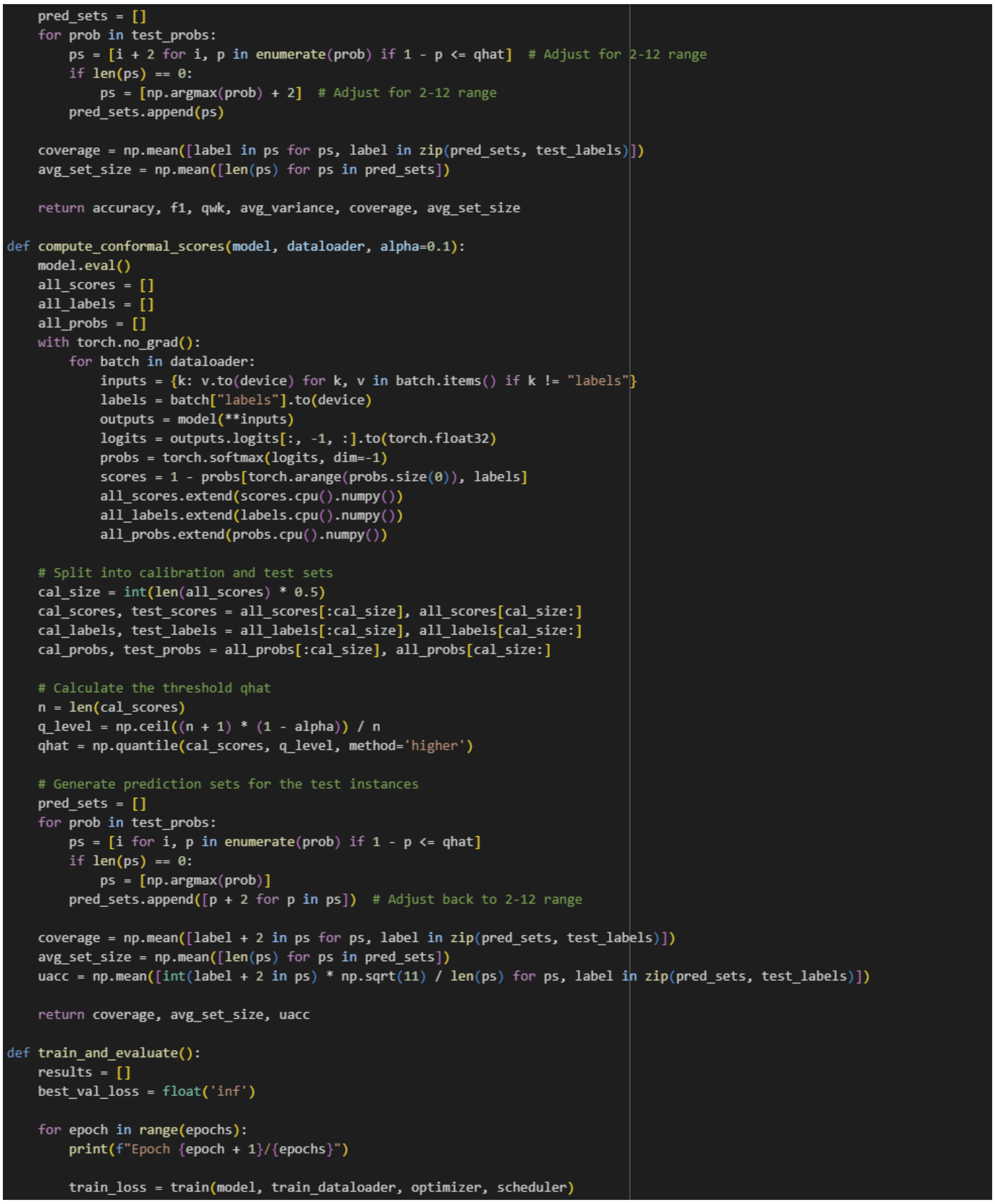
```python
def create_ensemble(num_models, train_dataloader, val_dataloader, test_dataloader):
    models = []
    for i in range(num_models):
        print(f"Training model {i+1}/{num_models}")
        model = LlamaForCausalLM.from_pretrained(
            model_name,
            quantization_config=bnb_config,
            device_map="auto",
            torch_dtype=torch.float32
        )
        model.config.use_cache = False
        model = prepare_model_for_kbit_training(model)
        model = get_peft_model(model, lora_config)
        model.gradient_checkpointing_enable()
        model = model.to(device)

        # Train the model
        train_model(model, train_dataloader, val_dataloader, test_dataloader)
        models.append(model)
    return models

# Modified training function to return the trained model
def train_model(model, train_dataloader, val_dataloader, test_dataloader):
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate, eps=epsilon)
    scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=warmup_steps, num_training_steps=len(train_dataloader) * epochs)

    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch in train_dataloader:
            optimizer.zero_grad()
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()
            scheduler.step()

        print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(train_dataloader):.4f}")

        # Evaluate on validation set
        val_loss, val_acc, val_f1, val_qwk = evaluate_model(model, val_dataloader)
        print(f"Validation - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")

    return model

# Function to evaluate a single model
def evaluate_model(model, dataloader):
    model.eval()
    total_loss = 0
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
```

Figure 41: Code snippet 36



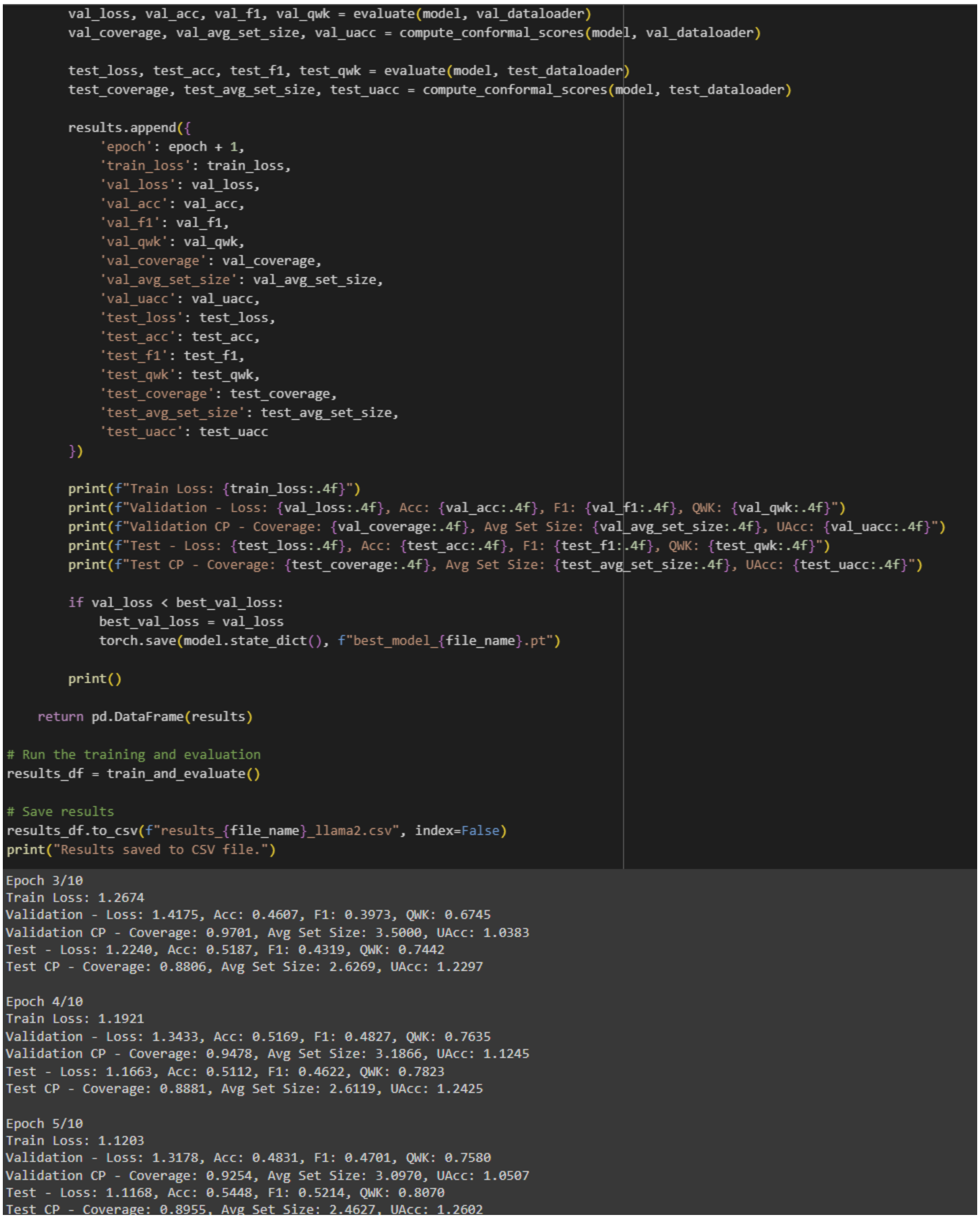
```python
            loss = nn.CrossEntropyLoss()(logits, labels)
            total_loss += loss.item()
            preds = torch.argmax(logits, dim=-1)
            all_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
            all_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

    accuracy = (np.array(all_preds) == np.array(all_labels)).mean()
    f1 = f1_score(all_labels, all_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, all_preds, weights="quadratic")
    return total_loss / len(dataloader), accuracy, f1, qwk

# Function to evaluate the ensemble
def evaluate_ensemble(models, dataloader, alpha=0.1):
    all_preds = []
    all_probs = []
    all_labels = []

    for model in models:
        model.eval()
        model_preds = []
        model_probs = []
        model_labels = []
        with torch.no_grad():
            for batch in dataloader:
                inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
                labels = batch["labels"].to(device)
                outputs = model(**inputs)
                logits = outputs.logits[:, -1, :].to(torch.float32)
                probs = torch.softmax(logits, dim=-1)
                preds = torch.argmax(logits, dim=-1)
                model_preds.extend(preds.cpu().numpy() + 2)  # Adjust back to 2-12 range
                model_probs.extend(probs.cpu().numpy())
                model_labels.extend(labels.cpu().numpy() + 2)  # Adjust back to 2-12 range

        all_preds.append(model_preds)
        all_probs.append(model_probs)
        all_labels = model_labels  # All models use the same labels

    all_preds = np.array(all_preds)
    all_probs = np.array(all_probs)
    all_labels = np.array(all_labels)

    # Ensemble predictions
    ensemble_preds = np.mean(all_preds, axis=0).round().astype(int)
    ensemble_probs = np.mean(all_probs, axis=0)

    # Compute metrics
    accuracy = (ensemble_preds == all_labels).mean()
    f1 = f1_score(all_labels, ensemble_preds, average="weighted")
    qwk = cohen_kappa_score(all_labels, ensemble_preds, weights="quadratic")

    # Compute uncertainty
    pred_variance = np.var(all_preds, axis=0)
    avg_variance = np.mean(pred_variance)

    # Compute conformal prediction sets
    cal_size = int(len(ensemble_probs) * 0.5)
    cal_probs, cal_labels = ensemble_probs[:cal_size], all_labels[:cal_size]
    test_probs, test_labels = ensemble_probs[cal_size:], all_labels[cal_size:]

    cal_scores = 1 - cal_probs[np.arange(len(cal_labels)), cal_labels - 2]  # Adjust for 2-12 range
    qhat = np.quantile(cal_scores, 1 - alpha, method='higher')
```

Figure 42: Code snippet 37



```
    pred_sets = []
    for prob in test_probs:
        ps = [i + 2 for i, p in enumerate(prob) if 1 - p <= qhat]  # Adjust for 2-12 range
        if len(ps) == 0:
            ps = [np.argmax(prob) + 2]  # Adjust for 2-12 range
        pred_sets.append(ps)

    coverage = np.mean([label in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])

    return accuracy, f1, qwk, avg_variance, coverage, avg_set_size

def compute_conformal_scores(model, dataloader, alpha=0.1):
    model.eval()
    all_scores = []
    all_labels = []
    all_probs = []
    with torch.no_grad():
        for batch in dataloader:
            inputs = {k: v.to(device) for k, v in batch.items() if k != "labels"}
            labels = batch["labels"].to(device)
            outputs = model(**inputs)
            logits = outputs.logits[:, -1, :].to(torch.float32)
            probs = torch.softmax(logits, dim=-1)
            scores = 1 - probs[torch.arange(probs.size(0)), labels]
            all_scores.extend(scores.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_probs.extend(probs.cpu().numpy())

    # Split into calibration and test sets
    cal_size = int(len(all_scores) * 0.5)
    cal_scores, test_scores = all_scores[:cal_size], all_scores[cal_size:]
    cal_labels, test_labels = all_labels[:cal_size], all_labels[cal_size:]
    cal_probs, test_probs = all_probs[:cal_size], all_probs[cal_size:]

    # Calculate the threshold qhat
    n = len(cal_scores)
    q_level = np.ceil((n + 1) * (1 - alpha)) / n
    qhat = np.quantile(cal_scores, q_level, method='higher')

    # Generate prediction sets for the test instances
    pred_sets = []
    for prob in test_probs:
        ps = [i for i, p in enumerate(prob) if 1 - p <= qhat]
        if len(ps) == 0:
            ps = [np.argmax(prob)]
        pred_sets.append([p + 2 for p in ps])  # Adjust back to 2-12 range

    coverage = np.mean([label + 2 in ps for ps, label in zip(pred_sets, test_labels)])
    avg_set_size = np.mean([len(ps) for ps in pred_sets])
    uacc = np.mean([int(label + 2 in ps) * np.sqrt(11) / len(ps) for ps, label in zip(pred_sets, test_labels)])

    return coverage, avg_set_size, uacc

def train_and_evaluate():
    results = []
    best_val_loss = float('inf')

    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")

        train_loss = train(model, train_dataloader, optimizer, scheduler)
```

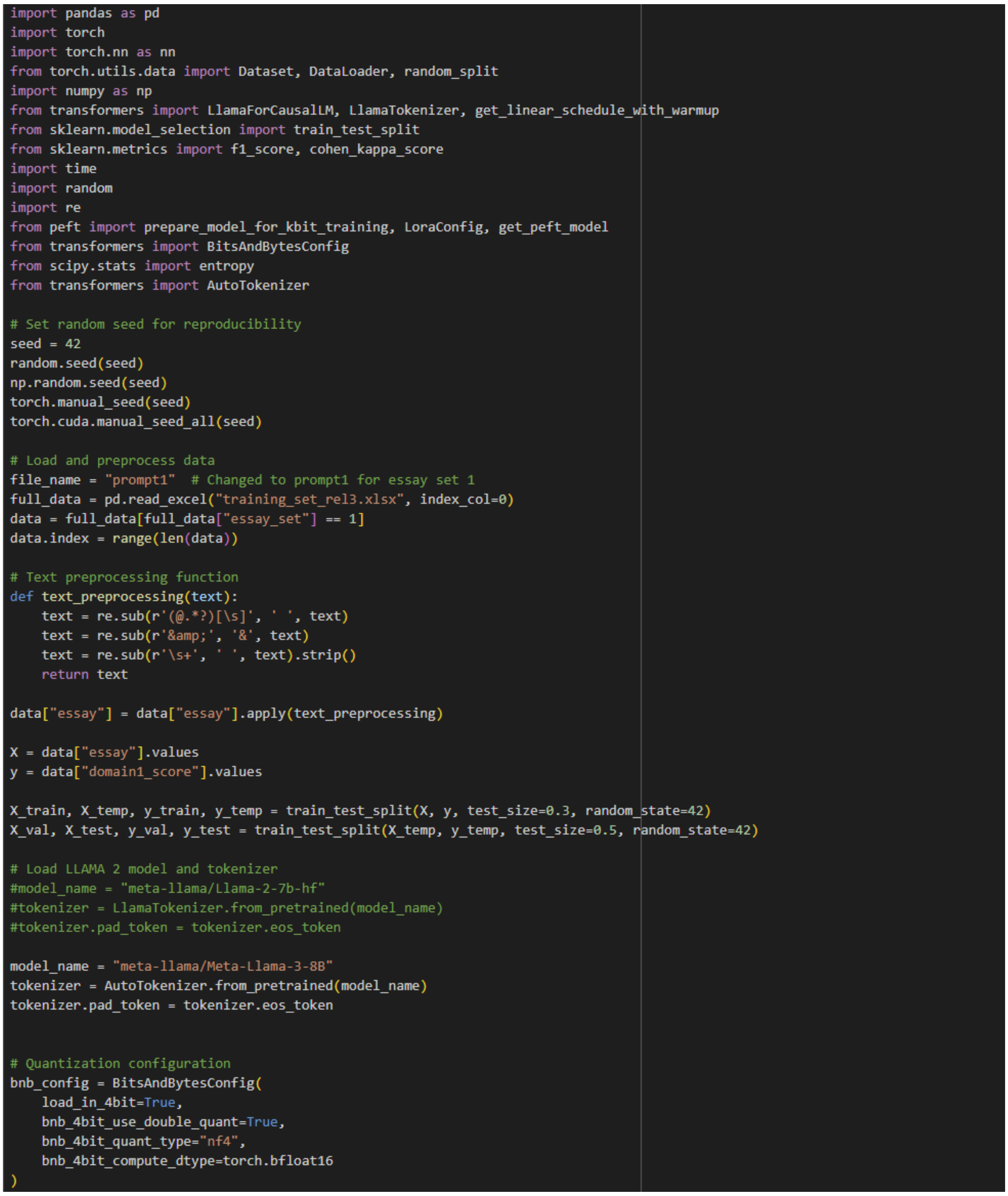
Figure 43: Code snippet 38



```
        val_loss, val_acc, val_f1, val_qwk = evaluate(model, val_dataloader)
        val_coverage, val_avg_set_size, val_uacc = compute_conformal_scores(model, val_dataloader)

        test_loss, test_acc, test_f1, test_qwk = evaluate(model, test_dataloader)
        test_coverage, test_avg_set_size, test_uacc = compute_conformal_scores(model, test_dataloader)

        results.append({
            'epoch': epoch + 1,
            'train_loss': train_loss,
            'val_loss': val_loss,
            'val_acc': val_acc,
            'val_f1': val_f1,
            'val_qwk': val_qwk,
            'val_coverage': val_coverage,
            'val_avg_set_size': val_avg_set_size,
            'val_uacc': val_uacc,
            'test_loss': test_loss,
            'test_acc': test_acc,
            'test_f1': test_f1,
            'test_qwk': test_qwk,
            'test_coverage': test_coverage,
            'test_avg_set_size': test_avg_set_size,
            'test_uacc': test_uacc
        })

        print(f"Train Loss: {train_loss:.4f}")
        print(f"Validation - Loss: {val_loss:.4f}, Acc: {val_acc:.4f}, F1: {val_f1:.4f}, QWK: {val_qwk:.4f}")
        print(f"Validation CP - Coverage: {val_coverage:.4f}, Avg Set Size: {val_avg_set_size:.4f}, UAcc: {val_uacc:.4f}")
        print(f"Test - Loss: {test_loss:.4f}, Acc: {test_acc:.4f}, F1: {test_f1:.4f}, QWK: {test_qwk:.4f}")
        print(f"Test CP - Coverage: {test_coverage:.4f}, Avg Set Size: {test_avg_set_size:.4f}, UAcc: {test_uacc:.4f}")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), f"best_model_{file_name}.pt")

        print()

    return pd.DataFrame(results)

# Run the training and evaluation
results_df = train_and_evaluate()

# Save results
results_df.to_csv(f"results_{file_name}_llama2.csv", index=False)
print("Results saved to CSV file.")
```

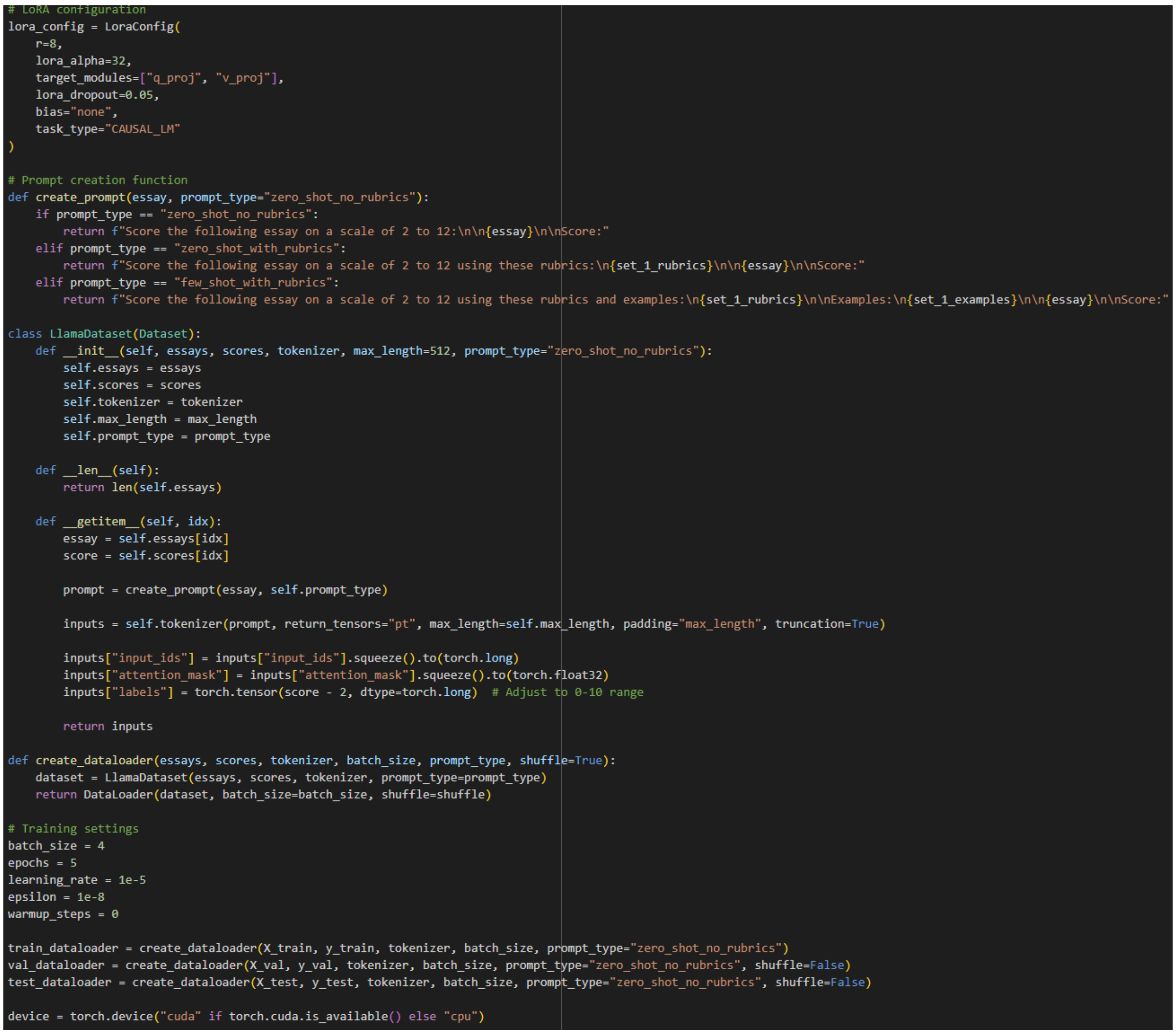
```
Epoch 3/10
Train Loss: 1.2674
Validation - Loss: 1.4175, Acc: 0.4607, F1: 0.3973, QWK: 0.6745
Validation CP - Coverage: 0.9701, Avg Set Size: 3.5000, UAcc: 1.0383
Test - Loss: 1.2240, Acc: 0.5187, F1: 0.4319, QWK: 0.7442
Test CP - Coverage: 0.8806, Avg Set Size: 2.6269, UAcc: 1.2297

Epoch 4/10
Train Loss: 1.1921
Validation - Loss: 1.3433, Acc: 0.5169, F1: 0.4827, QWK: 0.7635
Validation CP - Coverage: 0.9478, Avg Set Size: 3.1866, UAcc: 1.1245
Test - Loss: 1.1663, Acc: 0.5112, F1: 0.4622, QWK: 0.7823
Test CP - Coverage: 0.8881, Avg Set Size: 2.6119, UAcc: 1.2425

Epoch 5/10
Train Loss: 1.1203
Validation - Loss: 1.3178, Acc: 0.4831, F1: 0.4701, QWK: 0.7580
Validation CP - Coverage: 0.9254, Avg Set Size: 3.0970, UAcc: 1.0507
Test - Loss: 1.1168, Acc: 0.5448, F1: 0.5214, QWK: 0.8070
Test CP - Coverage: 0.8955, Avg Set Size: 2.4627, UAcc: 1.2602
```

Figure 44: Code snippet 39



```python
import pandas as pd
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader, random_split
import numpy as np
from transformers import LlamaForCausalLM, LlamaTokenizer, get_linear_schedule_with_warmup
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, cohen_kappa_score
import time
import random
import re
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model
from transformers import BitsAndBytesConfig
from scipy.stats import entropy
from transformers import AutoTokenizer

# Set random seed for reproducibility
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)

# Load and preprocess data
file_name = "prompt1"  # Changed to prompt1 for essay set 1
full_data = pd.read_excel("training_set_rel3.xlsx", index_col=0)
data = full_data[full_data["essay_set"] == 1]
data.index = range(len(data))

# Text preprocessing function
def text_preprocessing(text):
    text = re.sub(r'(@.*?)[\s]', ' ', text)
    text = re.sub(r'&amp;', '&', text)
    text = re.sub(r'\s+', ' ', text).strip()
    return text

data["essay"] = data["essay"].apply(text_preprocessing)

X = data["essay"].values
y = data["domain1_score"].values

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)

# Load LLAMA 2 model and tokenizer
#model_name = "meta-llama/Llama-2-7b-hf"
#tokenizer = LlamaTokenizer.from_pretrained(model_name)
#tokenizer.pad_token = tokenizer.eos_token

model_name = "meta-llama/Meta-Llama-3-8B"
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token


# Quantization configuration
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)
```

Figure 45: Code snippet 40



```python
# LoRA configuration
lora_config = LoraConfig(
    r=8,
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

# Prompt creation function
def create_prompt(essay, prompt_type="zero_shot_no_rubrics"):
    if prompt_type == "zero_shot_no_rubrics":
        return f"Score the following essay on a scale of 2 to 12:\n\n{essay}\n\nScore:"
    elif prompt_type == "zero_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics:\n{set_1_rubrics}\n\n{essay}\n\nScore:"
    elif prompt_type == "few_shot_with_rubrics":
        return f"Score the following essay on a scale of 2 to 12 using these rubrics and examples:\n{set_1_rubrics}\n\nExamples:\n{set_1_examples}\n\n{essay}\n\nScore:"

class LlamaDataset(Dataset):
    def __init__(self, essays, scores, tokenizer, max_length=512, prompt_type="zero_shot_no_rubrics"):
        self.essays = essays
        self.scores = scores
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.prompt_type = prompt_type

    def __len__(self):
        return len(self.essays)

    def __getitem__(self, idx):
        essay = self.essays[idx]
        score = self.scores[idx]

        prompt = create_prompt(essay, self.prompt_type)

        inputs = self.tokenizer(prompt, return_tensors="pt", max_length=self.max_length, padding="max_length", truncation=True)

        inputs["input_ids"] = inputs["input_ids"].squeeze().to(torch.long)
        inputs["attention_mask"] = inputs["attention_mask"].squeeze().to(torch.float32)
        inputs["labels"] = torch.tensor(score - 2, dtype=torch.long)  # Adjust to 0-10 range

        return inputs

def create_dataloader(essays, scores, tokenizer, batch_size, prompt_type, shuffle=True):
    dataset = LlamaDataset(essays, scores, tokenizer, prompt_type=prompt_type)
    return DataLoader(dataset, batch_size=batch_size, shuffle=shuffle)

# Training settings
batch_size = 4
epochs = 5
learning_rate = 1e-5
epsilon = 1e-8
warmup_steps = 0

train_dataloader = create_dataloader(X_train, y_train, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics")
val_dataloader = create_dataloader(X_val, y_val, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)
test_dataloader = create_dataloader(X_test, y_test, tokenizer, batch_size, prompt_type="zero_shot_no_rubrics", shuffle=False)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Figure 46: Code snippet 41