RECIPE FOR A PERFECT RECIPE

BERT PILOTED ERROR DETECTION FOR COOKING RECIPES

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Hei Yeung Cyrus Chan

Supervised by Dr. Kung-kiu Lau

Department of Computer Science

Contents

Word Count: 14,933

List of Figures

Abstract

When cooking with a recipe, besides following the steps incorrectly, another major cause of error is the recipe itself contains mistakes. While current grammar checkers might be able to check for grammatical typos or misspellings, there are no tools to check for other errors. The purpose of this project is to detect errors in cooking recipes using the large language model BERT and its variants, focusing on numerical errors, missing ingredients and cooking steps misordering. For each category of errors, I implemented multiple approaches featuring different data pre-processing methods, model architectures, and post-processing algorithms. Evaluations are performed on the fine-tuned models to assess performance of models with different hyperparameter settings. Each approach resulted in at least 1 model with satisfactory performance, such that the outputs of the models could guide the users to pinpoint the location of the errors and correcting the recipes. Lastly, I implemented a sample web application to demonstrate how the developed models can be deployed for actual error detection in cooking recipes.

Acknowledgements

I would like to express my gratitude towards my project supervisor Dr. Kung-kiu Lau for providing insights into my project problem and giving constructive feedbacks throughout the whole project. Without his guidance and supportive comments, I would not have been able to complete the project as shown below.

I would also like to acknowledge the assistance given by Research IT and the use of the Computational Shared Facility at The University of Manchester. Without the computing resources provided, it would not have been possible to progress the project and fine-tune the large language models essential to the project.

Chapter 1

Introduction

1.1 Context and Motivation

For amateur home cooks, it is not rare to see the final dish coming out not as expected. There are a few main causes to this, one of the most frustrating ones is the errors in the recipe referenced. For example, a recipe from The Guardian (23 September, Feast, p12) mentioned to use 350 g of courgettes, however, the dish requires 1.36 kg of courgettes as corrected and clarified later [16]. While current spell checkers, grammar checkers could detect typos and grammatical errors in recipes fairly easily, there are no tools or researches on checking other types of errors in cooking recipes.

There are multiple types of errors that could appear in a cooking recipe. For example, missing ingredients(in ingredient list or in cooking steps), error in numeric values(amount, time, cooking temperature), incorrect order of steps, ambiguous word usage, unclear amount of ingredients needed for each step, etc. In this project, I would focus on the first 3 errors mentioned, for their ease of annotation, as well as since they are more likely to introduce bigger problems in actual cooking intuitively.

With recent advancements of machine learning, especially in models with the transformer architecture, tasks involving natural language processing and understanding that were challenging to solve with traditional algorithms can be easier solved with large language models(LLM) [23]. I would like to propose the use of LLM for detecting the mentioned errors, in particular, focusing on using Bidirectional Encoder Representation Transformer(BERT) and its variants(RoBERTa, DeBERTa) to detect the mentioned errors. A few reasons for choosing BERT would be its lightweight structure, excellent performance in classification tasks and its ability to solve next-sentence prediction tasks. Detailed reasons for choosing BERT would be discussed in 3.

In addition to simply fine-tuning the models, various memory saving methods for training deep neural networks are used in this project. A few notable methods are gradient checkpointing, mixed precision, and low-rank adaptation for large language models(LoRA). Under limited computing resources available, these methods allow the LLMs to be fitted into smaller video random access memory. Some of these methods also claimed to improve model performance, which might be very beneficial for the fine-tuning needed in this project.

1.2 Aims and Objectives

The main aim of the project is to develop machine learning models that could check for errors in cooking recipes, such that the outputs could guide users to pinpoint errors and correct recipes The breakdown of detailed objectives are as below.

1. Develop different models with various approaches with reasonable accuracy to check for missing ingredients in recipes, numerical error in cooking times in recipes, numerical error in ingredient amounts in recipes, incorrect ordering in cooking steps in recipes.
2. Compare performance of BERT variants for each of the approaches.
3. Evaluate performance of different approaches for each type of errors.
4. Utilise various memory saving methods to allow fitting of models on GPUs and possibly improved performance.
5. Stretched Goal: Deploy the models in a web interface to demonstrate the feasibility of the models.

1.3 Report Structure

This report contains X chapters:

- Chapter 1 features the introduction to the project and the report.
- Chapter 2 features the project background, including a review of related work.
- Chapter 3 features the design and implementations of the error detection model approaches.
- Chapter 4 features the evaluation of all approaches and models.
- Chapter 5 features the implemented web application with the deployed models, and additional testing done with the interface.
- Chapter 6 features the project conclusions and reflections, including a review of future work that could be done.

Chapter 2

Backgound Information and Knowledge

2.1 Recipe

The formal dictionary definition of recipe is a set of instructions for preparing a particular dish, including a list of the ingredients required. In this project, we specify that a recipe is a list of ingredients with detailed amounts in US customary units and a set of instructions using the ingredients to prepare and cook a specific dish, all in English. The reason behind focusing on US customary units is that the dataset used in this project, the RecipeNLG dataset [3], uses mostly US customary units.

2.2 Transformers

The transformer architecture is a deep learning architecture that utilises the self-attention mechanism 2.3 without using any recurrent structures, allowing for faster training times and parallelization as recurrent structures were found to be hard to parallelise. Transformers typically contains either the encoder, the decoder, or both, as only encoder models are used in this project, we will only discuss encoder models. Encoder models take text input and output a sequence of embeddings(vector representation of the text), capable of performing classification tasks, unable or very difficult to perform text generation tasks, notable model include the BERT model. For the BERT model, when given any input, the output would be a numeric vector of length 768 representing the input text.

Original sentence		Cooking		the	wings	for	10	minutes	yields	best	results	
tokens	[CLS]	Cook	##ing	the	wings	for	10	minutes	yields	best	results	[SEP]
token ids	101	6816	1158	1103	4743	1111	1275	1904	17373	1436	2686	102

Figure 2.1: Example of BERT tokenization.

2.3 Attention

The attention mechanism is very commonly used after being proposed by Vaswani, et al.(2017) [35]. It has become a key mechanism in transformer models, that automatically adjusts the importance(weight) of each input value by using scaled dot-product attention or other attention variants. The attention function maps a query and a set of key-value pairs to an output, where the query, key, values, and output are all vectors. The output is a weighted sum of the values, where the weight assigned to each value is computed inside the attention function, as such each input key-value pair would be assigned an adjusted weight based on their importance. In scaled dot-product attention, the weights are calculated by:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Where queries Q and keys K have dimension of d_k, and values V has dimension of d_v, and all of them are matrices containing multiple queries, keys or values. As such $Attention(Q,K,V)$ is the optimised weights that should be applied to V. The widely used self-attention mechanism in transformers refers to relating different positions of a single sequence to produce a vector representation of the said sequence. For self-attention, all of Q,K,V are the same matrix, allowing the model to learn about the relations within a sequence.

2.4 Pre-training and Fine-tuning

Pre-training and fine-tuning are novel techniques for training a fully functional large language model. Pre-training refers to the training done on the model with all weights uninitialised, often with extremely large amount of un-annotated data or automatically annotated data, which means no human or expert labor is involved in the data annotation phase. Depending on model architecture and the pre-training tasks done, large language models could be used out-of-the-box, or further fine-tuned for downstream tasks.

Fine-tuning refers to training a pre-trained model with new data, often of a much smaller amount and annotated. For example, the toxic comment classification task involves fine-tuning a model with comments annotated with 6 different levels of toxicity. The annotation can often only be done by human experts or semi-automatically, with the aim that the fine-tuned model would be able to automatically annotate new unseen samples.

2.5 BERT

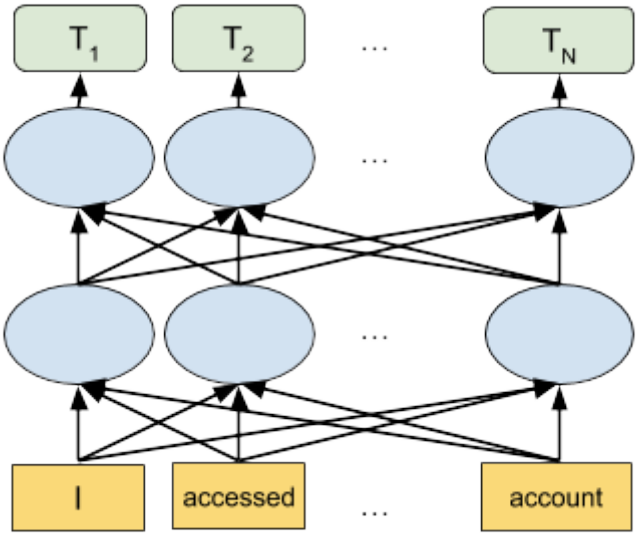

Figure 2.2: Simplified Model Structure of BERT taken from Google blog post [10].

Bidirectional Encoder Representations from Transformers (BERT) proposed by Devlin et al. (2019) [11] is an encoder-only transformer language model which uses the bidirectional self-attention mechanism to perform language related tasks. The bidirectional self-attention mechanism proposed in BERT learns the input in both original and reversed order, where a generic self-attention mechanism would only consider the input in the given order, often from left to right. In comparison, BERT is gaining a better understanding of the inputs for its ability to learn each token from both left and right of the tokenized word. The BERT model is pre-trained on the masked language modelling task and next sentence prediction task. Both tasks are fundamental to the value prediction/regression and steps ordering problems to be solved in this project, while BERT's strong performance in classification tasks resonates with the steps/ingredients boolean classification tasks in the project.

The masked language modelling task is a language modelling task focused on predicting a masked token (word) in a sequence of tokens (sentence). For example, in the sequence "The current [MASK] of the United States is Joe Biden as of 2024.", the prediction result should be "president". For the pre-training, a large amount of online data is collected and tokenized, each token has a 15% chance to be selected, when selected, the token has an 80% chance to be masked, a 10% chance to be replaced by a randomly selected token, a 10% chance to be left as is.

The next sentence prediction task (NSP) is a boolean classification task, where the model takes a pair of sequences and decide if the second sequence is a correct precedence of the first sequence. For example, given "Fry the floured chicken thigh pieces until golden brown." and "Dredge the chicken thigh pieces in the flour mix." in order, the model should tell the second sentence is not a correct precedence of the first sentence. For pre-training on the NSP task, 50% of the training data are actual pairs from the original corpus, where the other 50% have a randomly selected sentence from the corpus as the second sentence.

2.6 RoBERTa

RoBERTa, proposed in RobERTa: A Robustly Optimized BERT Pre-training Approach [24], has the exact same structure as BERT but is pre-trained with optimised hyperparameters, much larger mini-batches and removed next sentence prediction pre-training. As a result of the optimised pre-training, the model out-performs the original BERT in most downstream tasks. For RoBERTa, we will be using the distilRoBERTa version, which is the distilled version of the RoBERTa base model that uses the model distillation process proposed by Sanh et al. (2019) [32], featuring 34.4% less parameters but retains over 95% of the performance of the base model.

2.7 DeBERTa

DeBERTa, the decoding-enhanced BERT with disentangled attention model proposed by Microsoft [19], features firstly the disentangled attention mechanism that represents each word using 2 vectors that encode its content and position, secondly the enhanced mask decoder that replaces the output softmax layer to predict masked tokens for model pre-training. We would be using the DeBERTa V3 base model [18], which is further improved by pre-training on replaced token detection task proposed by Clark et al. (2020) [9], which as its name suggests, reports if a token is replaced in the given sequence of tokens.

2.8 Related Work

2.8.1 BERT

Devlin et al.(2018) [11] proposed the BERT language representation model with state of the arts performance on various downstream tasks when fine-tuned. While being a slightly dated model, BERT's variants are still competitive in the machine learning scene, RoBERTa proposed by Liu et 2019 revised the pre-training of the original BERT model and re-trained the model with optimised hyperparameters and larger data batches, resulting in the RoBERTa model constantly beating BERT in all tasks, to this day the RoBERTa is still often used in researches as a benchmark for comparison. He et al.(2021) [19] proposed the DeBERTa model, backed by its disentangled attention mechaism and enhanced mask decoder, the DeBERTa model has out performed the RoBERTa model on various tasks, and for the first time surpassed human performance on the SuperGLUE benchmark proposed by Wang et al. (2020) [36]. He et al.(2023) [18] further proposed the DeBERTaV3 model, with a new gradient-disentangled embedding sharing method, and pre-trained on the more sample-efficient token detection task. The DeBERTaV3 model again surpassed the DeBERTa base model by 1.37% in the GLUE benchmark, setting a new standard for state-of-the-art models.

2.8.2 Analysis on Classification Tasks

BERT and its variants have shown satisfactory state-of-the-art results for classification tasks across different contexts. Ghosal et al.(2022) [14] fine-tuned the DeBERTa model for binary classification, and proved that binary classification might be superior to n-class classification, such that modelling n-class classification as binary classification task might give huge performance boost. The model developed also reached top or close to the top performance on public leaderboards for related tasks. Xian et al. (2023) [41] created a fusion model DeBERTa-DPCNN, to perform short text classification, with a state-of-the-art macro F1 score of 91.33%, proving the DeBERTa model to be a strong model for feature extraction of text. Murarka et al. (2021) [28] fine-tuned RoBERTa on a mental illness social media dataset for multi-label classification, reaching an 89% macro F1 score, showing RoBERTa's strong ability to pick up both obvious and subtle features in text inputs. These results build a strong foundation for the classification approaches experimented in this project.

2.8.3 Regression Tasks

Regression Tasks BERT and its variants can also be applied to regression tasks, Sonkiya et al. (2021) [33] developed a BERT GAN(generative adversarial network) fusion model for stock price prediction and showed superior performance compared to other regression models such as long short term memory (LSTM), gated recurrent units (GRU), vanilla GAN, and auto-regressive integrated moving average(ARIMA) model. Zhou et al. (2020) [43] fine-tuned the multi-task BERT variant on a problem difficulty prediction task, using an additional softmax layer, effectively turning the problem to a regression task. The model shown superior results compared to bidirectional LSTM methods and the base BERT model, proving the feasibility of BERT regression. However, it is to note that there is not a lot of research on unbounded value regression tasks for BERT, Wang et al.(2022) [37] stated that state of the art regression models are often not well calibrated and cannot provide reliable uncertainty estimates, also stating there is almost no work in NLP on calibration in a regression setting. While there are only a few successful attempts in regression BERT, there is potential in the approach for more detailed error detection results.

2.8.4 Sentence Ordering

Sentence Ordering BERT and its variants has also been applied to sentence ordering tasks, Golestani et al. (2021) [15] used BERT for a sentence ordering task, giving satisfactory results over other attention networks and LSTM networks. Zhu et al. (2021) [44] fine-tuned BERT for sentence ordering, resulting in BERT4SO, a novel model outperforming or having close performance to existing sentence ordering models. Sun et al. (2022) [34] developed a prompt based method NSP-BERT for BERT next sentence prediction task, while not predicting the ordering of a full set of sentences from a document, the task tells if 2 sentences are correctly ordered and of the same context. The NSP-BERT is successful against other zero-shot methods on most FewCLUE benchmark tasks [42]. These results, especially the next sentence prediction ability of BERT forms the foundation of the cooking step ordering task, as the task sets out to spot incorrectly ordered steps or steps not belonging to the

recipe.

2.8.5 Recipe Related Work

Mohammadi et al. (2020) [27] used multiple models including BERT to perform recipe classificaiton based on difficulty, but the results are not the most promising due to the extremely unbalanced data. Sakib et al. (2021) [31] fine-tuned a few BERT variants for recipe genre classification, the final fine-tuned models reached state-of-the-art accuracies, with some outperforming the others, namely DistilBERT, a distilled version of BERT. While there is a range of recipe generation researches being done, for example the works by Lee et al. (2020) [17], Fujita et al. (2021) [13], there is no research done on recipe error detection, this could be because there is not enough data(incorrect recipes) or any related datasets.

Chapter 3

Methodologies and Implementations

3.1 Training Workflow

To train or fine-tune a large language model, this project follow the following workflow:

1. Pre-process dataset for the model
2. Split data into train, validation, test splits
3. Train model with the train data
4. Validate model performance with validation data
5. Repeat steps 3 and 4 for a set number of epochs
6. Evaluate the final model with test data

The validation step records the performance of the model on unseen data after each training epoch, which could show the actual performance of the model and prevent overfitting to the training data. The test data is for evaluating the final saved model, and to show if the model is overfitting to the validation data.

3.2 Choice of Models

DistilRoBERTa model and DeBERTaV3 model would be used unless they are not suitable/pre-trained for the task, e.g. next-sentence prediction. The choice of models is based on literature and model performances suggested in their respective papers [24] [18], that both DistilRoBERTa and DeBERTaV3 have out-performed the base BERT model.

3.3 Data Source and General Pre-processing of Data

The dataset being used is the RecipeNLG dataset introduced by Bień et al. (2020) [3], which is a dataset of over 2 million recipes, in which around 1.6 million recipes are of higher quality as a result

of careful processing. We only use the 1.6 million higher quality recipes in this project. The data is first cleaned, by removing escaped characters, stripping repeated space characters, replacing Unicode degree symbols with the word “degree”.

3.4 Limitations and Assumptions

Because of limited computation resources and time constraints, although the original dataset used contains more than two million recipes, only a portion is used in each of the model fine-tuning phase. Limited by the input size of 512 tokens of BERT models, only recipes with less than 512 words are used for training. It is assumed that all pre-processed recipes are correct, or that the amount of wrong recipes is negligible. Since incorrect recipes are of a very small amount, and that manually reviewing all recipes in the dataset is impossible due to limited time and human resources, if incorrect recipes are needed for training in any approaches, they would be generated with some automated algorithms tailored for the approach. It is also assumed that all recipes uses US customary units based on the majority of the recipes’ units used in the dataset.

3.5 Library for training

The Hugging Face transformers library [40] is used for training the models. The library provides an API for feature-complete training based on the PyTorch library [29], with a wide range of parameters to be customized for the training process. For the features not provided by Hugging Face, PyTorch is used, which is a more detailed and complex library for machine learning methods implementations and optimisations.

3.6 Memory Saving Methods

To train large language models efficiently in a reasonable time, it is best to train the models with graphics processing units (GPU) optimised for machine learning. In this project, NVIDIA Tesla V100 GPU (V100) would be used for model training. The V100 has 16 GB of random access memory (RAM), since large language models require a lot of space to fit as well as update, along with the training data, 16 GB is not enough if we fine-tune the models as is. Therefore, we utilise a few popular memory saving methods to make fitting large language models on V100 possible.

3.6.1 Gradient Accumulation

Large language models benefits from large batch sizes, where a batch is a set amount of samples used for each training step. However training with a large batch size is very memory heavy, the idea behind gradient accumulation is instead of calculating gradients for a large batch, the gradients are being calculated for much smaller batches and accumulated, and updated every set number of batches. Since calculating gradients is very memory heavy with the memory usage increase with amount of

samples in a batch, by doing the calculations on smaller batch sizes, the memory usage could be greatly reduced. However, compared to doing all calculations in one go, gradient accumulation takes more time as more computations have to be done, namely the batch operations. Gradient accumulation can be done by setting the batch size to a lower number, and setting the gradient accumulation step to above 1 as the training parameters. For example, a batch size of 4 and gradient accumulation step of 8 should give the same result of a batch size of 32 and no gradient accumulation, however using batch size 4 uses much less memory than batch size 32.

3.6.2 Gradient Checkpointing

The most memory intensive part of model training is the computation of the gradient loss by back propagation. For vanilla back propagation, a large amount of nodes is required to be stored in memory if they are parent or ancestor nodes of the current node being computed, the memory required grows linearly with the number of layers in the model, which can easily exceed the memory limit, however this is the most efficient computation method. For memory efficient back propagation, no nodes are stored in memory, and nodes are recomputed every time they are needed for computation of other nodes, however, this is very time inefficient, as recomputation although not memory heavy, requires time. Gradient checkpointing proposed by Chen et al. (2016) [7] introduces a balance between vanilla method and the recomputation method, by checkpointing a set of nodes that could minimize the amount of recomputation of other nodes to at most 1, as such, a part of the nodes are stored in memory while the others are being recomputed at most once. However, since recomputation is still done for some nodes, the total training time would increase about 20%, while maximum memory usage could be reduced by up to 80% depending on the model architecture. The gradient checkpointing method is integrated into the Hugging Face trainer, and could be enabled by setting the gradient_checkpointing flag to True.

3.6.3 Mixed Precision

While not being a memory saving method, mixed precision proposed by Micikevicius et al. (2018) [26] increases training efficiency to compensate the training speed loss from other methods. The idea of mixed precision is that not all variables need to be stored in full 32-bit floating point precision, by reducing the precision to 16-bit, the computations could be sped up. The variables with reduced precision are the activations in the model. Since mixed precision is implemented as a native flag in the Hugging Face trainer, it could be enabled by setting the fp-16 flag to True.

3.6.4 LoRA

Low-Rank Adaptation (LoRA) of Large Language Models, proposed by Hu et el. (2021) [20], is a method that freezes the pre-trained model weights and injects trainable rank decomposition matrices into some layers of the transformer architecture, to achieve a much lower memory requirement while maintaining the performance if optimal LoRA hyperparameters are used. LoRA builds on the hypothesis that 2 matrices of lower dimensions can sufficiently represent the original matrix of weights in a

layer of the model.

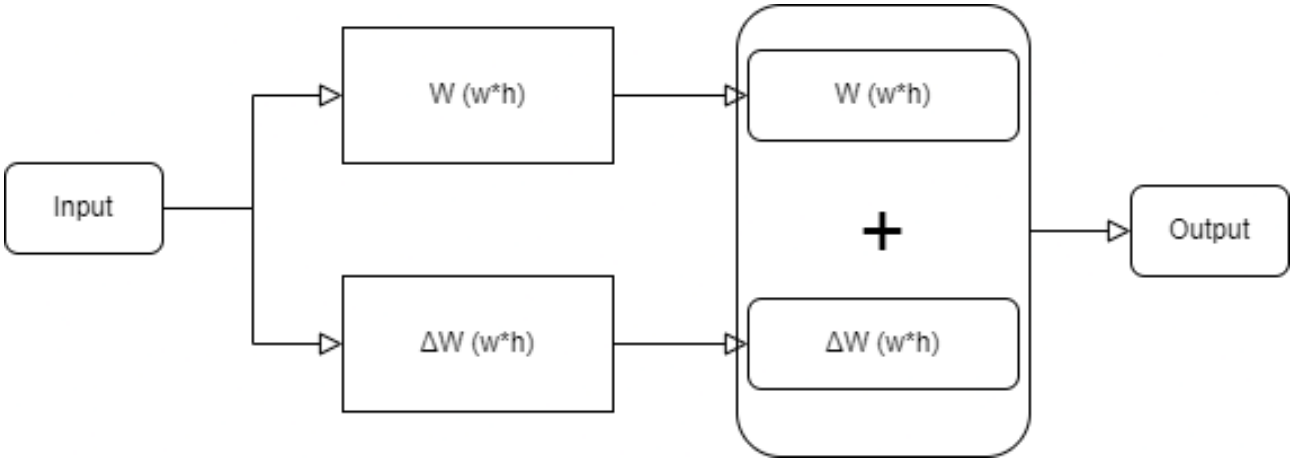

Figure 3.1: Simplified full-finetuning.

Figure 3.1 shows a simplified full fine-tuning work flow, where delta weights has the same size of weights and the updated weights would be (weights + delta weights). However, computing all values in delta weights is very memory heavy.

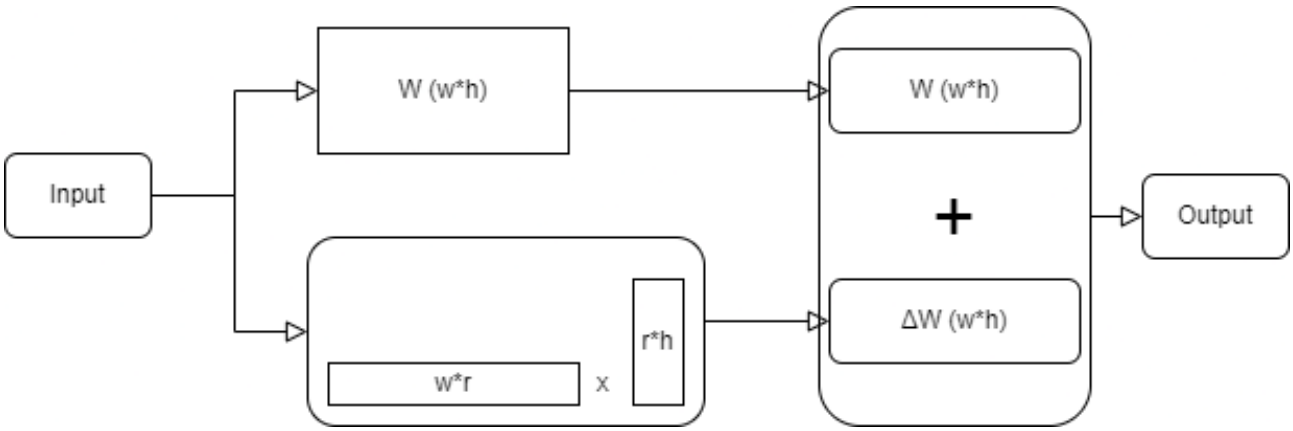

Figure 3.2: Simplified LoRA-finetuning.

Figure 3.2 shows a simplified LoRA fine-tuning work flow, where the original delta weights of size $(w \times h)$ is replaced by two matrices of size $(w \times r)$ and $(r \times h)$. r is a hyperparameter to be set, and by doing matrix multiplication $(w \times r) \times (r \times h)$, a new delta weights matrix of size w x h can be obtained. By using a small r value (as low as 1 suggested by the LoRA paper), the number of weights to be updated is sharply reduced compared to full fine-tuning. In addition to the r value, the alpha value in LoRA scales the weights in the rank decomposed matrices to compensate for the reduced number of trainable parameters, it is suggested that the alpha value should be double the r value in the LoRA paper [20]. By optimising r and alpha, it is said that LoRA fine-tuning can result in performance on par with full fine-tuning in the same paper. Therefore in this project, for tasks involving LoRA, a hyperparameter search would be done to find out best rank. During the hyper-parameter search phase, the alpha value is kept as 16 as suggested in the hyper-parameter search part of the LoRA paper. The LoRA parameters to be searched are $r = [16, 32, 64, 128]$, $\textit{LoRA dropout} = [0, 0.01, 0.050.1]$,

LoRA weight decay $= [0, 0.01, 0.05, 0.1]$. As time and resources are limited, instead of a full grid search, a Bayesian hyperparameter search of 10 runs is performed for the LoRA hyperparameters, and keeping learning rate at 2e-5, batch size at 32, LoRA alpha at 16 as proposed by the LoRA paper [20], with epoch per run at 1. Bayesian search is a method that selects hyperparameters by the Bayes rule, such that the selection is based on the probabilities that a hyperparameter is going to contribute to the improvement of the model. For the actual fine-tuning phase, we set $\alpha = 2 \times r$ as discussed before. Moreover, rank stablised LoRA (RSLoRA) proposed by Kalajdzievski (2023) [22] is used, which optimised the weight scaling formula in the original LoRA paper, such that larger r values can be used and could result in potentially better performance. To implement LoRA fine-tuning, the parameter-efficient fine-tuning (PEFT) library [25] is used, which allows for rank decomposition matrices injection to models and the setting of r and alpha values.

3.7 Missing Ingredient Error

For this error we propose implementing a named entity recognition (NER) model by fine-tuning DistilRoBERTa and DeBERTaV3 for extracting the ingredients appeared in the ingredient list and the steps, combined with a post-processing layer to output the ingredients missing in ingredient list or the cooking steps.

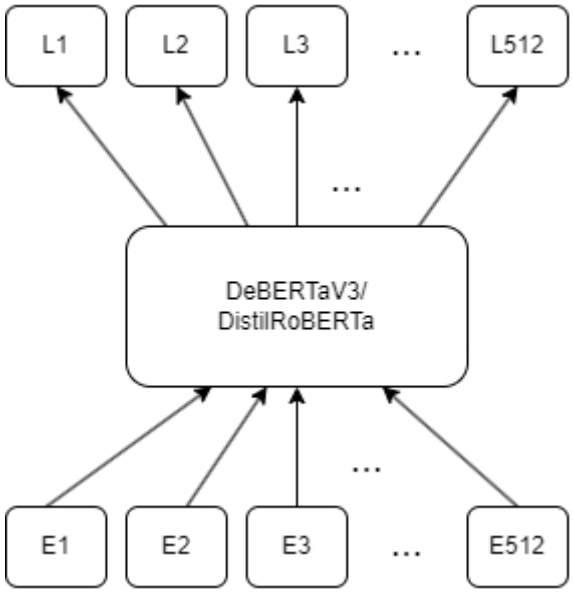

Figure 3.3: NER model structure.

A named-entity recognition (NER) model is a classification model that could classify and label entities it was trained on when given a sequence of words [6]. Figure 3.3 shows the structure of the NER model, the input would be the sub-word tokens (E1 to E512) of the text sample, then as the output each token would be given a label (L1 to L512) indicating the entity type depending on what fine-tuning the model was given. If a sentence has less than 512 tokens, padding tokens are added until the sequence reaches 512 tokens.

3.7.1 Data preparation

To prepare the training data for the fine-tuning of the NER model, the cooking steps are passed through an NER model (InstaFoodRoBERTa) [12] fine-tuned on social media food posts, and labelled with the IOB tagging format, where "I" denotes non-beginning words of food entity, "O" denotes non-food entity, "B" denotes beginning of food entity. "O" label is encoded with 0, "B" label with 1, "I" label with 2. Figure 3.4 shows an example of an annotated sentence.

Mix	eggs	and	plain	flour	.
0	1	0	1	2	0

Figure 3.4: Example of IOB annotation.

Each text would be one input sample, where the integer label is the output of the model. While the InstaFoodRoBERTa model was able to label almost all the food entities, some unwanted words were also labelled, for example mixture, parchment, foil, etc. A list of words to exclude is created by randomly sampling the labelled data and manually reviewing the samples, and repeating this process until no unwanted words are shown in the samples. The full list of excluded words are shown in A. The cooking steps are then re-labelled by comparing each word with the exclusion list then adjusting the labels accordingly. 100 000 samples are randomly selected from the corrected data and used as the dataset for the task. The dataset is split with a typical 80%/10%/10% split for training, validation, and testing. Though the size of the dataset is reduced, there is still a relative large amount of data for validation and testing to ensure the validation and testing results generalise to other recipes in the original dataset. The selected samples are then tokenized with the RoBERTa tokenizer, as most BERT variants use sub-word tokens, meaning a word could be split into multiple parts when tokenized, the labels are also adjusted accordingly to match the dimension of the tokens.

3.7.2 Fine-tuning

To fine-tune the models, we train the pre-trained DistilRoBERTa and DeBERTa model with the tokenized annotated cooking steps. We use an initial setting of 3 training epochs, 3e-5 learning rate, 32 batch size, adjusting if the results are not satisfying, we set the satisfactory line to be 90% with respect to benchmarks on the CoNLL dataset NER task [39], as named entity recognition tasks are common and results are fairly similar on various datasets and contexts. The models are saved at the epochs with best F1 scores.

3.7.3 Post Processing

The ingredients list and cooking steps would be passed through the best model separately, producing two lists of ingredients w.r.t. the ingredient list and the cooking steps. The two lists are passed to a simple list comparison algorithm that removes an item from a list when the item is a sub-word of any item of the other list. For example, if given [apple, milk] and [apples, milk], the algorithm would return 2 empty lists, if given [apples, milk] and [oranges, milk], the algorithm would return [apples]

and [oranges]. Therefore, the algorithm outputs are 1 list showing ingredients missing from cooking steps, 1 list showing ingredients missing from ingredients list. Finally, the following figure 3.5 shows the complete design of the approach.

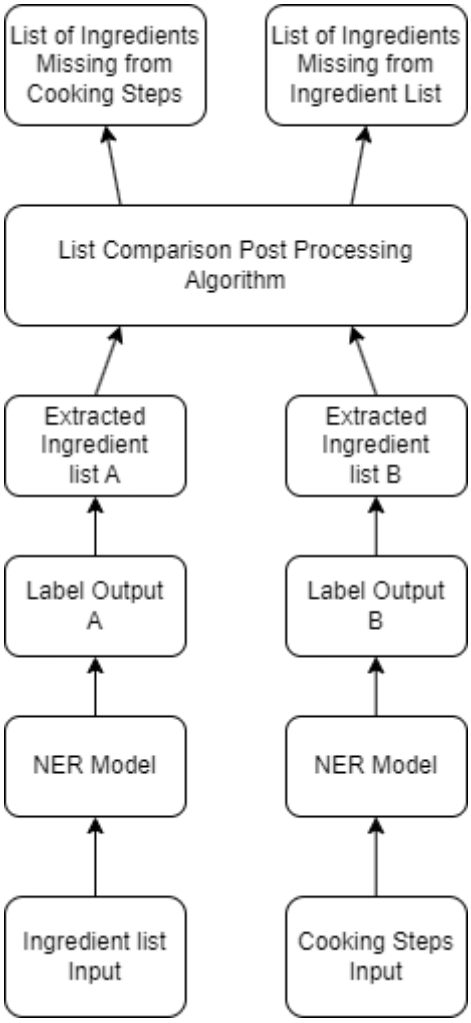

Figure 3.5: The complete approach design for missing ingredient error.

3.7.4 Example Input and Output

As a trivial example, for the recipe “Soft Boiled Egg”, with the ingredients list “2 large eggs”, and cooking steps “Boil the room temperature eggs in boiling water for 6 minutes. Put in ice water to stop cooking. Remove shells and serve.”, the method should output two empty lists. In detail, the model would output [“eggs”] for the ingredients list, and also output [“eggs”] for the cooking steps, after passing through the post-processing layer, the two “eggs” cancel out and therefore return 2 empty lists indicating there are no missing ingredients from neither of ingredients list nor cooking steps.

3.8 Ingredient Amount Error

For this error we propose 2 approaches. We fine-tune both the DistilRoBERTa model and the DeBERTaV3 model for this task and choose the model with better performance.

3.8.1 Approach 1 (Regression)

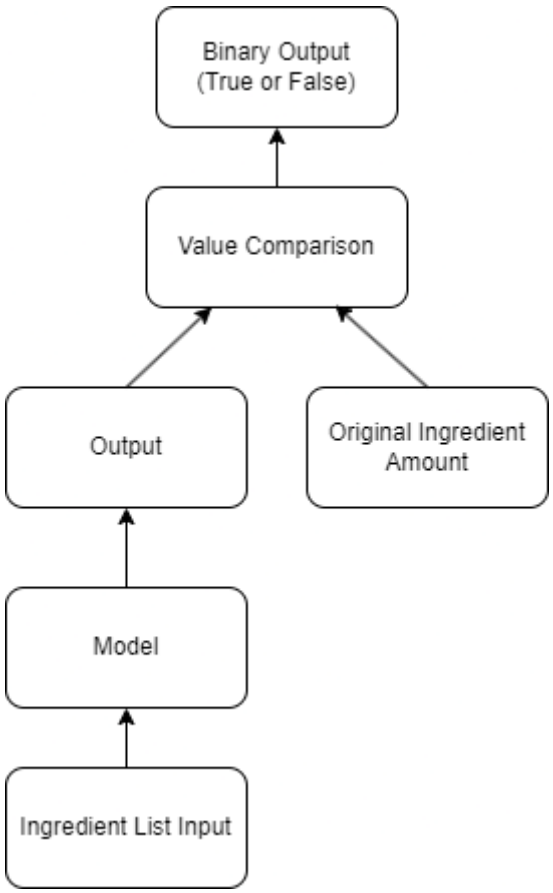

Figure 3.6: Approach design for ingredient amount error approach 1.

Figure 3.6 shows the design of the approach, where the model output would be a single regression value for each input, the output would then be compared to the original ingredient amount to give a binary output. The model for this approach takes inputs with ingredient amounts masked and performs a regression prediction for each of the ingredient's amount. Each input of the model would be a list of ingredients with a single amount masked, in the form of a tokenized sequence, while the output is a single normalised regression value. Figure 3.7 shows the structure of the model, where the regression layer is a simple fully connected linear layer that takes all BERT layer outputs as input and output a single regression value.

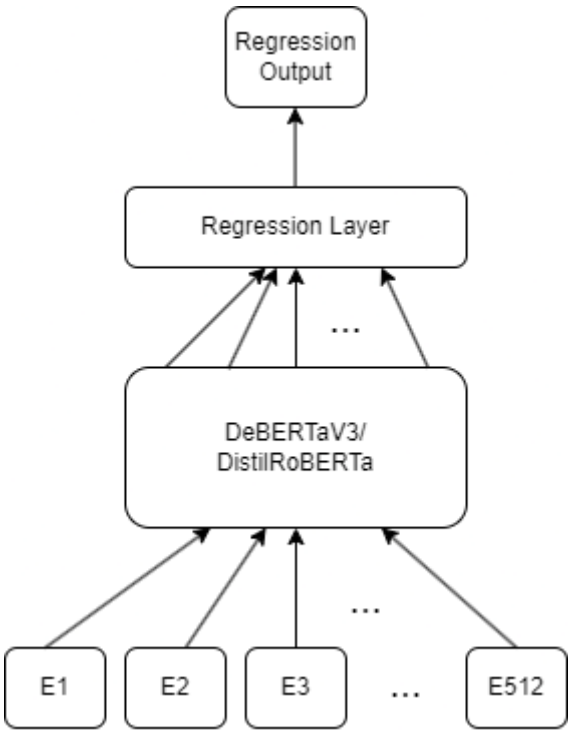

Figure 3.7: Regression model structure.

Data Preparation

We take only the ingredient lists from the cleaned data, and the lists are further processed by turning fractions into floating point numbers, as well as taking the mean of values in the form of "X to Y" or "X - Y" where X and Y are numbers. This is done to reduce the ambiguity of the value to be predicted, as well as restricting the value to be predicted for each input to a single one. The numbers in English are converted to their Arabic equivalent, for example "an apple" to "1 apple", "five slices of bacon" to "5 slices of bacon", while this does make the data easier to understand for the model, however this makes extracting the values and labelling the recipes easier, as well as making evaluation easier, as we only need to perform number value comparisons instead of string-number comparison. 100 000 samples are then randomly selected, and each ingredient list is duplicated by the number of ingredients in the list, while each list has one amount value masked, and that the ingredient being masked is different for each list of the same recipe. For example if there exist an ingredient list "2 lb. apple, 1 cup sugar", then this sample would be replaced by two samples "[MASK] lb. apple, 1 cup sugar" and "2 lb. apple, [MASK] cup sugar". The masked samples are labelled with normalised masked value, where the normalisation is done by scaling values of range 0-100 to range of 0-1 with clipping. Using the example "[MASK] lb. apple, 1 cup sugar" again, this sample would be labelled with 0.01. This label would be the output target of the model. The ingredient list part of the samples are then tokenized.

Fine-tuning

The models are trained with the prepared data. Each model is fine-tuned twice to result in two different models, the first model is full fine-tuned with most memory saving methods but without LoRA, the

second model is fine-tuned with all memory saving methods including LoRA. For the full fine-tune model we use learning rate 2e-5, batch size 32, and train for 10 epochs, saving the model state with the lowest validation loss. As discussed before, since LoRA uses smaller matrices to update the original model in training phase, such that it might have vastly different performance compared to a full fine-tuned model, therefore a separate model is fine-tuned with LoRA to observe LoRA's effectiveness compared to the full fine-tuned model. Before fine-tuning the LoRA models, a hyperparameter search is carried out with the setup mentioned in 3.6.4. After the search we use the best hyperparameters to fine-tune the model for 10 epochs and save the model state with the lowest validation loss. Since we are fine-tuning both the DistilRoBERTa model and the DeBERTaV3 model, there would be 4 resulting fine-tuned models for this task. Evaluation is done on the models with mean squared error (MSE) loss and absolute average error to find the best model.

Example Model Input and Output

As an example, for the recipe "Grilled Shrimps", with the ingredients list "1/4 c. extra-virgin olive oil; 1/4 c. lime juice; 4 garlic cloves, minced; 3 tbsp. honey; 2 tbsp. low-sodium soy sauce; 1 tbsp. chili garlic sauce or Sriracha; 2 lb. shrimp, peeled and deveined; 1/4 c. freshly chopped cilantro, for garnish; Lime wedges, for serving", would be processed into 8 sample since it has 8 ingredients with values. Such that if we take the first processed sample as input of the model, the input is "[MASK] c. extra-virgin olive oil; 0.25 c. lime juice; 4 garlic cloves, minced; 3 tbsp. honey; 2 tbsp. low-sodium soy sauce; 1 tbsp. chili garlic sauce or Sriracha; 2 lb. shrimp, peeled and deveined; 0.25 c. freshly chopped cilantro, for garnish; Lime wedges, for serving", and the expected output is 0.25, the raw model output should be scaled (0.0025), but as with simple post-processing added to the output layer, the value is scaled back to the orginal scale. In reality the value would not be exactly 0.25 due to the imperfect nature of machine learning models.

3.8.2 Approach 2

The second approach takes an un-augmented recipe ingredient list and performs binary classification to predict if whole ingredient list is correct in terms of amounts. The input of the model would be a list of ingredients in the form of a tokenized sequence, while the output would be a vector of length 2, with the first position representing the list being correct in terms of amounts, the second position representing incorrect, whichever position with a higher value forms the final prediction of the input being correct or incorrect, for example [0.34, 0.89] indicates the input being incorrect. Figure 3.8 shows the model structure for this approach, where the classification layer is a fully connected linear layer that takes all outputs from the BERT layer as inputs and output 2 values that indicates the probabilities that a sample is in the true class or the false class. The class with higher probability is then taken as the final classification for the input.

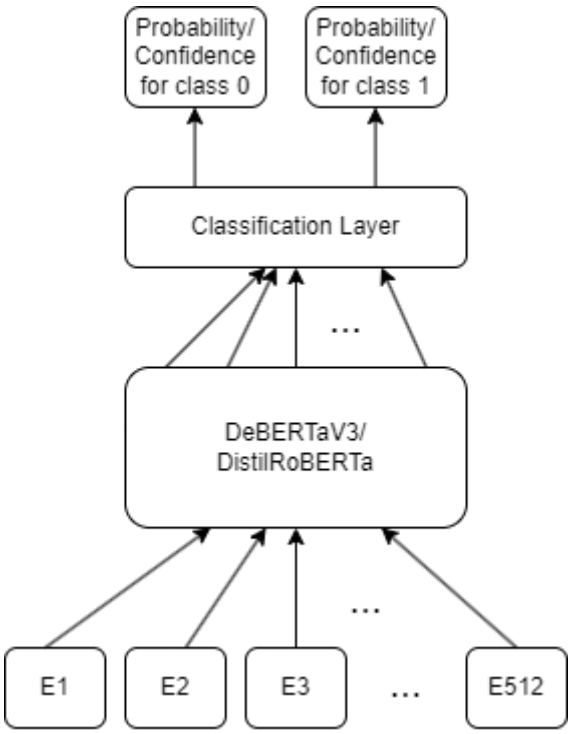

Figure 3.8: Binary classification model structure.

Data Preparation

Following similar steps to approach 1, but instead of duplicating and masking values, we select 500 000 samples and split into 2 equal splits. For the first split we leave all ingredient lists as is, and label all lists as [1, 0], indicating they are correct lists. We use the second split to generate incorrect lists for training. For the second split, each ingredient in the list have a 25% chance of having its value randomised, a 10% chance of having its value multiplied by 10 to simulate zeros that are incorrectly entered in recipes. The randomisation is done by multiplying the original value by a value in range of 0 to 4 in steps of 0.2, to simulate ingredient amounts with error. As there is not a lot of data for incorrect recipes, the randomisatioon parameters are set by intuition and experience, where a value being incorrectly converted in units could result in values few times larger or smaller than the correct value, and these errors would rarely be too big (values that are magnitudes larger) or too small (values extremely close to zero), as errors that extreme would be too obvious to not be spotted and corrected before publishing, whereas more subtle errors could make it to websites, newspaper, etc. easier.

Fine-tuning

The models are trained with the prepared data. Similar to approach 1, a full fine-tuned model and a LoRA fine-tuned model are trained for each of DistilRoBERTa and DeBERTaV3 models. For the full fine-tune model we use learning rate 2e-5, batch size 32, and train for 10 epochs, saving the model state with the highest F1 score. Before fine-tuning the LoRA models, a hyperparameter search is carried out with the setup mentioned in 3.6.4. After the search we use the best hyperparameters to fine-tune the LoRA model for 10 epochs and save the model state with the highest F1 score. Evaluation of the models is done by computing the cross entropy loss and F1 scores.

Example Model Input and Output

Using the same example from the last approach, however, we modify the values such that instead of 2 lb. of shrimps we now have 4 lb. of shrimps, so we would have "1/4 c. extra-virgin olive oil; 1/4 c. lime juice; 4 garlic cloves, minced; 3 tbsp. honey; 2 tbsp. low-sodium soy sauce; 1 tbsp. chili garlic sauce or Sriracha; 2 lb. shrimp, peeled and deveined; 1/4 c. freshly chopped cilantro, for garnish; Lime wedges, for serving" as the model input. The raw model output should be a vector of length 2, for example [.42, .78], adding a simple argmax function in the output layer, the model would output 1 instead, indicating the input is classified as class 1, that there exist incorrect ingredient amounts (while class 0 indicates the recipe has no errors).

3.9 Cooking Time Error

We define cooking time as the timed actions with numerical values in the recipes, such that refrigerating, marinating, freezing times would also be considered as cooking times. For this error we propose 2 approaches. We fine-tune both the DistilRoBERTa model and the DeBERTaV3 model for this task and choose the model with better performance.

3.9.1 Approach 1 (Regression)

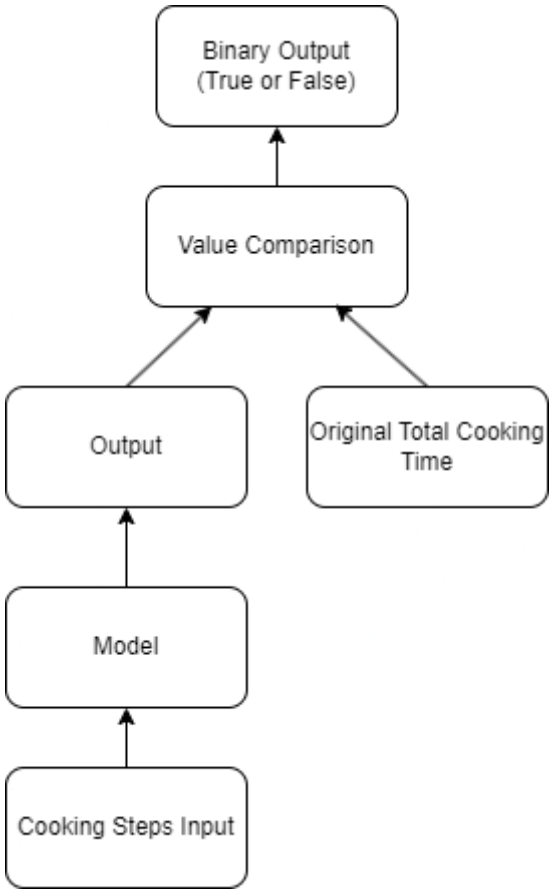

Figure 3.9: Approach Design for cooking time error approach 1.

Figure 3.9 shows the design of the approach, where the model output would be a single regression

value for each input, the output would then be compared to the original total cooking time to give a binary output. For the first approach, the model takes the concatenated cooking steps from a recipe and performs a regression prediction for the total cooking time needed for the recipe. The input would be all steps of a recipe with all cooking times masked in the form of a tokenized sequence, while the output is a single normalised value. The model structure is same as the structure in figure 3.7 of ingredient amount error approach 1.

Data Preparation

For each recipe, the steps are first concatenated into one sequence, fractions in the sequence are converted to float numbers, and values in the form of "X to Y" or "X - Y" are replaced with the mean of X and Y, where X and Y are numbers. Numbers in English are converted to their Arabic equivalent. Time representations are also converted to numbers, for example "half an hour" converted to 30 minutes. Recipes with ambiguous or incorrect symbol usages are filtered out, this is done mainly by removing recipes with consecutive hyphens or slashes, as well as incorrect fraction forms or incorrect "X - Y" forms, for example, " 1–3 oranges", "1///2 jar of sauce", "8-1-2 sausages", "/4 cup honey", etc. This serves 2 purposes, firstly remove recipes of poor quality, secondly this can be seen as another method to detect typo-like errors. After processing and correcting the samples, 500 000 samples are randomly selected and tags are applied to the samples. Firstly the cardinal words are tagged, the list of cardinal words is ["second", "minute", "hour", "sec", "min", "hr", "s"], for example "Cook for 5 hours." will be given the list of tags [0,0,0,1]. Secondly we tag the cooking times w.r.t. the cardinal word tags, by looking at 1 to 2 words before the tagged cardinal words, for example "3 more hours" would have its original tags [0,0,1] adjusted to [2,0,1], "20 minutes" would have its original tags [0,1] adjusted to [2,1]. Then the total cooking time in minutes is calculated based on the cooking time tags and cardinal word tags. The total cooking time is normalised by scaling from a range of 0-400 to a range of 0-1, the range 0-400 is selected because most recipes' cooking time lie in this range, while only a small amount of recipes lies within the 200-400 range, they are included to increase the diversity of the data. Finally, the cooking times in the steps are masked with "[MASK]", then the cooking steps are tokenized, and each sample is labelled with the normalised total cooking time. The tokenized steps would be the input for the models, while the normalised total cooking times would be the target output.

Fine-tuning

The models are fine-tuned with the prepared data. A full fine-tuned model and a LoRA fine-tuned model are trained for each of DistilRoBERTa and DeBERTaV3 models. For the full fine-tune model we use learning rate 2e-5, batch size 32, and train for 10 epochs, saving the model state with loweset validation loss. Before fine-tuning the LoRA models, a hyperparameter search is carried out with the setup mentioned in 3.6.4. After the search we use the best hyperparameters to fine-tune the LoRA model for 10 epochs and save the model state with lowest validation loss. To evaluate the models, the mean squared error loss and mean absolute error are being measured and compared to find the best model.

Example Model Input and Output

As a simple example, using the recipe "Soft Boiled Egg" again, the cooking steps "Boil the room temperature eggs in boiling water for 6 minutes. Put in ice water to stop cooking. Remove shells and serve." would be preprocessed to "Boil the room temperature eggs in boiling water for [MASK] minutes. Put in ice water to stop cooking. Remove shells and serve." and this would be the input of the model. The model should output numeric value 6 in the perfect case, however, in reality, the value would be higher or lower due to the imperfect nature of machine learning models.

3.9.2 Approach 2 (Binary Classification)

For the second approach, the model takes the unaugmented concatenated cooking steps from a recipe and performs a binary classification to predict if the cooking times are correct or not. The input of the model would be the concatenated steps in the form of a tokenized sequence, while the output would be a vector of length 2, with the first position representing the cooking times being correct, the second position representing incorrect, whichever position with a higher value forms the final prediction of the input being correct or incorrect. The model structure is same as the structure in figure 3.8 of ingredient amount error approach 2.

Data Preparation

Perform the same preparation in approach 1 but without masking the cooking times. Select 500 000 samples randomly, split into 2 even parts. For the first half of the samples, keep as is and label each sample with [1,0]. For the second half of the samples, recall the samples are tagged with cooking time tag and cardinal word tag, w.r.t. the time tags, randomise the cooking times in the recipe. The randomisation is done by selecting cooking times at 20% chance, if a cooking time is selected, multiply it by a random value in range 0 to 4 in steps of 0.2. For each cooking time, at 10% chance, multiply the original value by 10. This is done to simulate a distribution over a range of patterns of unreasonable or wrong cooking times. After randomising the values, label each sample in this sample split with [0,1]. Next tokenize all the cooking steps in both splits. Combine the 2 splits and shuffle the samples. The tokenized cooking steps are the input for the models, while the labels are the target output.

Fine-tuning

The models are fine-tuned with the prepared data. Similar to approach 1, a full fine-tuned model and a LoRA fine-tuned model are trained for each of DistilRoBERTa and DeBERTaV3 models. For the full fine-tune model we use learning rate 2e-5, batch size 32, and train for 10 epochs, saving the model state with the highest F1 score. Before fine-tuning the LoRA models, a hyperparameter search is carried out with the setup mentioned in 3.6.4. After the search we use the best hyperparameters to fine-tune the LoRA model for 10 epochs and save the model state with the highest F1 score. Evaluation of the models is done by computing the cross entropy loss and F1 scores.

Example Model Input and Output

As a simple example, using the recipe "Soft Boiled Egg" again, we modify the cooking steps "Boil the room temperature eggs in boiling water for 6 minutes. Put in ice water to stop cooking. Remove shells and serve." to introduce an error, such that the modified version is "Boil the room temperature eggs in boiling water for 15 minutes. Put in ice water to stop cooking. Remove shells and serve." and this would be the input of the model. The raw model output should be [0, 1] in the perfect case, while in reality it could be something like [0.23, 0.86] due to the imperfect nature of machine learning models, adding a simple argmax function in the output layer, the model would output 1 instead, indicating the input is classified as class 1, that the total cooking time is incorrect (while class 0 indicates the total cooking time has no errors).

3.10 Cooking Steps Order Error

For this error we propose 2 approaches that uses different data preparation methods but uses the same model structure. Since the DistilRoBERTa model and the DeBERTaV3 model are not pre-trained on the next sentence prediction task that is beneficial for this error, the BERT base model is used instead. We fine-tune the BERT base model for this task and choose the approach with better performance. The models would take tokenized steps pairs and perform binary classification to predict if the order of the pair is correct or not. The input for the model would be the pair of tokenized steps separated by the tokenized "[SEP]" special symbol, where the output is a vector of length 2, such that if the first value is higher than the second value the input is predicted as correct, otherwise incorrect. Figure 3.10 shows the model structure for both approaches of this error, where the classification layer is a fully connected linear layer that takes all outputs of the BERT layer as input and output 2 values that indicates the probabilities that a sample is in the true class or the false class. The class with higher probability is then taken as the final classification for the input. To test the ordering of a whole recipe, steps pairs could be created with the original order in the recipe, and passing each pair to the model, which would output the order correctness of each pair, hence pinpointing the location of order errors.

3.10.1 Common Data Pre-processing

Only cooking steps in the dataset is being used for this error. Broken steps in the dataset is being filtered, by removing recipes containing steps not ending with a terminal punctuation, for example "Mix cut vegetables with vinegar and". For each recipe, steps with only 1 word are filtered, for example "Mix.", "Cook.", "Heat.". This is done to remove steps that are too ambiguous, as they could fit in most step orders, and they are also often do not carry enough information for sentiment analysis. Select 100 000 sample recipes, for each recipe, create an array of pairs of steps with correct order.

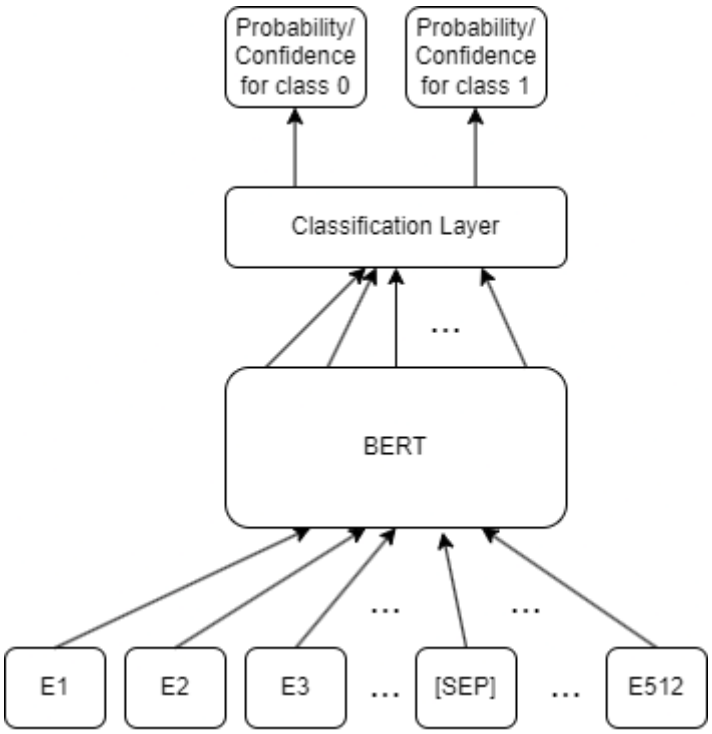

Figure 3.10: Model Design for cooking steps order error.

3.10.2 Approach 1

For the first approach, the idea is to create incorrect samples by matching steps with random steps within the same recipe, and train the model's ability to detect mis-ordered steps within a recipe. We name this dataset as the "local ordering error" set, as the misordering happens locally inside a recipe.

Data Preparation

Split the samples into two even parts, for the first half, since the steps pairs are grouped by recipe, unpack the recipe lists to obtain a list of pairs, and label each pair as [1,0], such that this part is laballed as the correctly ordered samples. For the second half, make a copy of the samples, unpack the copied recipe lists and denote the unpacked list as A. For the original samples, unpair the pairs, such that the recipes are in their original form, then generate random pairs P within each recipe and label as [0,1] (incorrect order), where P is not in A, so that no correct orders are generated. For each recipe in the second half, (number of steps - 1) pairs are generated (same number of pairs is generated in the first half of the samples). Unpack the generated incorrect samples, compared the number of samples in first half and second half, select the part with smaller number of samples and randomly select same number of samples from the other part to ensure balance of data. Combine the samples in the selected part and the randomly selected samples. As BERT tokenizer supports sequence pair tokenization, the pairs are passed through the tokenizer and the [SEP] token is automatically added in between the tokenized pair sequences. The tokenized pair sequences would be the input of the model, while their corresponding labels are the target output.

3.10.3 Approach 2

For the second approach, the idea is to create incorrect samples by matching steps with random steps that could be either in the same recipe or from other recipes, and train the model's ability to detect mis-ordered steps, as well as "foreign" steps. We name this dataset as the "global ordering error" set, as the misordering could happen with steps from other recipes.

Data Preparation

Split the samples into two even parts, similar to approach 1, uncap the first half into a list of pairs, denote this list L1, and label each pair as [1,0]. For the second half, unpair all steps, and unpack the recipes to form a list of steps from all recipes, denote this list L_ALL. Denote number of pairs in first half as N. Create another list L2 to contain the incorrectly ordered pairs to be generated. From L_ALL form random pairs one by one and append to L2, such that each pair is not in L1 and L2 before being appended. N pairs would be created to match the length of L1 for data balance, label each pair as [0,1]. Combine L1 and L2, pass each pair through the BERT tokenizer to create sequence pair tokens. The sequence pair tokens would be the input for the models, while the labels would be the target outputs.

3.10.4 Fine-tuning

The BERT base model is being fine-tuned for this error, for each approach, a full fine-tuned model and LoRA fine-tuned model are being trained. The difference in training for each approach is the dataset used. For the full fine-tuning, the learning rate 2e-5 and batch size 32 are used, and train for 10 epochs, saving the model state with the highest F1 score. Before fine-tuning the LoRA models, a hyperparameter search is carried out with the setup mentioned in 3.6.4. After the search we use the best hyperparameters to fine-tune the LoRA model for 10 epochs and save the model state with the highest F1 score. Evaluation of the models is done by computing the cross entropy loss and F1 scores.

3.10.5 Example Model Input and Output

As a simple example, using the recipe "Soft Boiled Egg" again, for the cooking steps "Boil the room temperature eggs in boiling water for 6 minutes. Put in ice water to stop cooking. Remove shells and serve." assume each sentence is a separate step, such that if we have steps 3 and 2 as input: "Remove shells and serve. [SEP] Put in ice water to stop cooking.". The raw model output should be [0, 1] in the perfect case, while in reality it could be something like [0.45, 0.62] due to the imperfect nature of machine learning models, adding a simple argmax function in the output layer, the model would output 1 instead, indicating the input is classified as class 1, that the steps are not in correct order (while class 0 indicates the steps are in correct order).

Chapter 4

Evaluation

4.1 Evaluation Metrics

4.1.1 Loss functions

Loss functions are used to calculate the loss between predicted values and ground truth values. The loss values are then used by the models to adjust its weights as a learning process. Moreover, the loss calculated could also be used to show the performance of the model or as comparison metrics for model comparison.

Cross Entropy Loss (CE Loss)

Cross entropy loss [5] measures the performance of a classification model whose output is a probability value between 0 and 1 for a single class, and a sequence of probabilities for multiple classes. The formula for cross entropy loss is:

$$CELoss = -\sum_{i=1}^{i=n} y_i, c \times log(p_i, c)$$

Such that n is the total number of samples, y_i,c is the i-th sample's ground truth label for class c, p_i, c is the probability prediction for the i-th sample belonging to class c. For example if the ground truth label for a sample is [1, 0] and the prediction is [0.5, 0.2], the resulting cross entropy loss is 0.6931. While being able to capture the slightest change in the model, the loss value is not very human-readable and is mostly used for model to model comparisons or as the loss function for loss minimisation in the fine-tuning phase.

Mean Squared Error Loss (MSE Loss)

Mean squared error (MSE) loss [5] measures how close a set of predicted data points are to their ground truth data points, typically used for regression tasks. The loss is calculated by taking the mean of the sum of all squared error, the formula is:

$$MSELoss = \frac{\sum_{i=1}^{i=n}(y_i - p_i)^2}{n}$$

Such that y_i is the i-th sample's ground truth, p_i is the i-th sample's prediction, and n is the total number of samples. For this project, since all value regression tasks only outputs a single value per sample, we measure the MSE loss for the whole batch, such that y_i is the ground truth for the i-th sample in the batch, p_i is the prediction for the i-th sample in the batch, and n is the batch size, which is 32 throughout this whole project.

4.1.2 Confusion Matrix

A confusion matrix [38] is a table containing 4 elements to evaluate the performance of a classification algorithm or model, the table contains the following components which can be used to calculate precision, recall, and F1 score:

- True positives (TP): Number of samples correctly predicted as belonging to the class being evaluated
- False positives (FP): Number of sample wrongly predicted as belonging the class being evaluated
- True negatives (TN): Number of samples correctly predicted as not belonging to the class being evaluated
- False negatives (FN): Number of samples wrongly predicted as not belonging to the class being evaluated

4.1.3 Precision

Precision [8] is the measure of correctly classified samples in the class among all predicted samples in the class with a range of 0 to 1, such that a higher precision means when a sample is predicted as the class, this prediction is of high accuracy. However, precision does not tell if the classifier can predict most of the samples in the class, as false negatives are not being considered in the computation, as an extreme example, a classifier could predict only 1 positive out of a balanced dataset of 100 samples, and still have the perfect precision of 1. The formula for computing precision is:

$$Precision = \frac{TP}{TP + FP}$$

4.1.4 Recall

Recall [8] is the measure of correctly classified samples in the class among all samples in the class, such that a higher recall means the classifier can correctly predict most samples in the class. However, recall does not tell if the classifier is accurate in its positive predictions, as an extreme example, a

classifier could predict all samples in a balanced dataset as positive and still have the perfect recall of 1. The formula for computing recall is:

$$Recall = \frac{TP}{TP + FN}$$

4.1.5 F1 Score

F1 score [8] is the harmonic mean of precision and recall. F1 score encourages similar precision and recall values, a higher score means both precision and recall is high, and that the classifier has a better overall quality. F1 score is calculated by:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall}$$

The formula for average F1 score (macro F1) and weighted average F1 score (micro F1) are

$$macroF1 = \frac{\sum_{i=1}^{i=n} f_i}{n}$$

$$microF1 = \sum_{i=i}^{i=n} f_i \times \frac{n_i}{n}$$

Such that f_i is the F1 score for the i-th class, n_i is the number of samples in the i-th class, and n is the total number of samples. By evaluating F1 score of each class we can see the classifier's performance on each class, and by evaluating the micro or macro F1 score we could see the classifier's overall performance, where micro F1 score emphasis classes of larger sizes, macro F1 score treats all classes evenly.

4.1.6 Accuracy

Define accuracy in this project as exact matches over number of samples, where the data balance is not considered.

4.1.7 Mean Absolute Error (MAE)

Mean absolute error is the mean of all absolute errors. The formula is:

$$MAE = \frac{\sum_{i=1}^{i=n} pred_i - ground_i}{n}$$

Such that $pred_i$ is the i-th predicted value, $ground_i$ is the i-th ground truth value, and n is the total number of samples.

4.2 Evaluation Tools

The model performances during training and validation phases are tracked and visualised by the Weights and Biases library (WandB) [4]. In the testing phase, testing results are evaluated with the Scikit-learn library [30] that provides implementations of various evaluation metrics. The visualisation of the evaluation results is created with the Matplotlib library [21].

4.3 Missing Ingredients Error

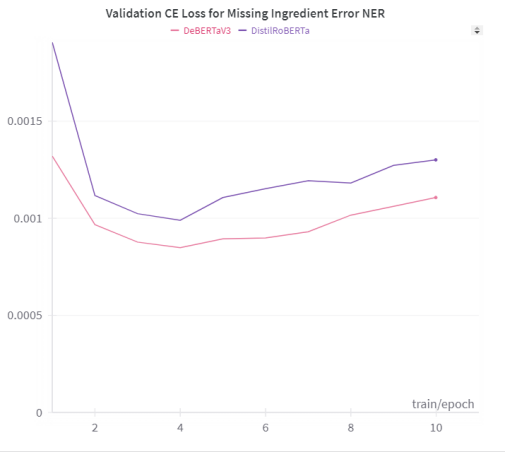

Figure 4.1: Validation CE loss graph for Missing Ingredients Error

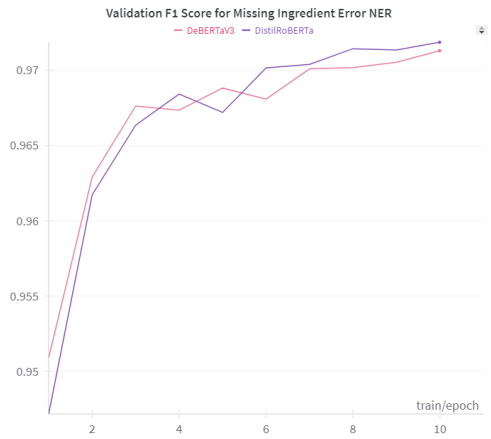

Figure 4.2: Validation F1 score graph for Missing Ingredients Error

As can be seen from the validation CE graph in figure 4.1, the CE losses for validation are lowest at epoch 4 for both DistilRoBERTa and DeBERTaV3 models, where the DeBERTaV3 model has a lower CE loss. From the validation F1 score graph in figure 4.2, the F1 scores are highest at epoch 10 for both models, where DistilRoBERTa has a slightly higher score than DeBERTaV3. The model are saved at their states with highest F1-scores, and are used for testing. The reason using F1-score to choose the best model is because F1 score is a more defining metric such that higher values means the model must perform better, while CE loss is a metric based on probability distributions, such that it might not always be true that the lower value gives the better model. However, CE loss works very well as the loss function of the model for loss minimisation and weights optimization, because of its probabilistic nature that gives values much more fine-grained values that is sensitive to any changes in the model, where F1-score tends to be less sensitive and can only give values with little precision (fewer decimal digits).

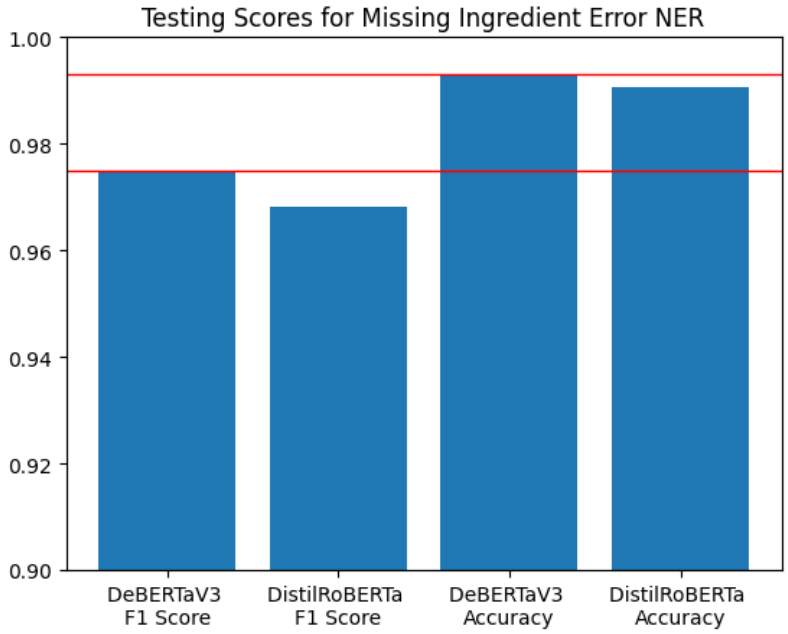

Figure 4.3: Testing F1 score graph for Missing Ingredients Error

The models are tested with the test split data. The testing graph in figure 4.3 reveals that the 2 models have similar performance, while DeBERTaV3 out-performs the DistilRoBERTa model slightly in both macro F1 score and accuracy. The InstaFoodRoBERTa model is also tested on the test split as a benchmark, which received 0.49 in F1 score and 0.9 in accuracy, confirming the newly fine-tuned models greatly out-perform InstaFoodRoBERTa and no longer identify cooking tools as food items. It is to note that the generic accuracy score for NER models might not be the best metric, as the majority of the tokens (sub-words) would be classified as "O" which is the out-of-entity class containing all tokens not related to the desired food entities. Therefore, macro F1 metric is also used as it takes the average F1 score from each class equally, representing the models' performance fairly.

To test the performance of the model combined with post-processing list comparison layer, 10 unseen random recipes are passed to the model and outputs are manually evaluated. 6 out of the 10 recipes showed no errors, while 5 showed missing ingredients. In the 5 recipes, two errors originated from water not being included in the ingredients list, one error originated from salt not included in the ingredients list, and two errors originated from ingredients not included in cooking steps because the cooking steps used phrases like "all ingredients", "all remaining ingredients". While some recipes might choose to not put water, salt, pepper, etc. into the ingredients list as they are too common, for a more detailed and precisely written recipe, these should also be included to reduce confusion and possibility to not having these ingredients. Hence, these errors are correctly identified. As for the "all ingredients" case, the method does not carry the ability to process these phrases, which is one of the limitations of the method. These would be cases of false detection. While the model reached a satisfactory 80% accuracy in this test, it is clear that the testing dataset is small and a larger testing set would be desired, but due to not having any way of automated evaluation methods, a large amount of time and human resources would be needed for manual evaluation which is not possible for the scale of this project. To further improve the performance and reduce errors, one viable way is to

train another model as the post-processing layer, such that when given 2 lists, the model could label the items that is actually missing from the other list, ignoring the semi-products. However, to train this model, data would be required and automation in creating the data is near impossible as there is no existing algorithms or model to separately label semi-products and ingredients, while manually labelling the data would require an enormous amount of time as LLMs require a large amount of training data.

4.4 Ingredient Amount Error

4.4.1 Approach 1

Results from LoRA hyperparameter Bayesian search for DistilRoBERTa revealed batch size 32, r value 128 and LoRA dropout value 0 gives the lowest loss. For DeBERTaV3, batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. Hence, these hyperparameters are used for the fine-tuning of LoRA models.

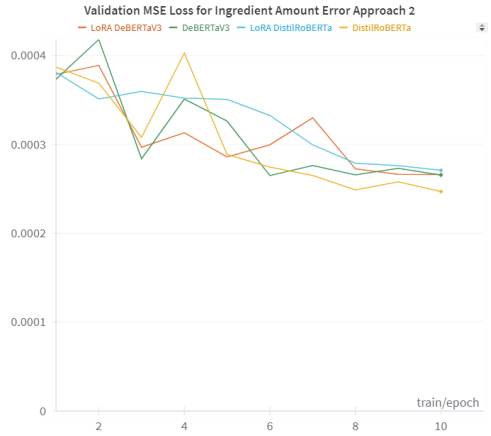

Figure 4.4: Validation MSE loss graph for Ingredient Amount Error Approach 1

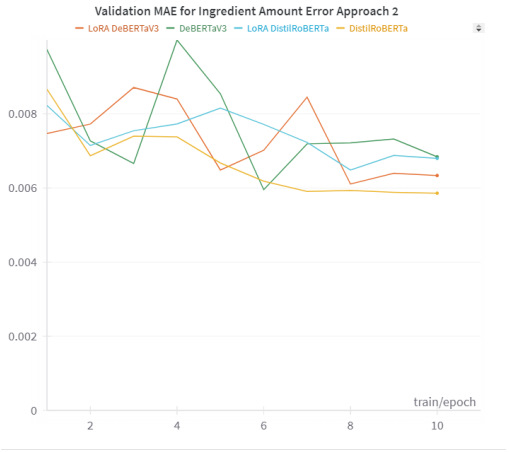

Figure 4.5: Validation MAE graph for Ingredient Amount Error Approach 1

From the validation MSE loss graph in figure 4.4, models reached their lowest loss at different epochs, with the LoRA models having higher losses, and the DistilRoBERTa model performing the best with the lowest loss. The models with the lowest losses are saved. From the validation MAE graph in figure 4.5, it is still the case that LoRA models have slightly higher error, which is reasonable for their reduced parameters in the fine-tuning phase. The DistilRoBERTa model outperformed the DeBERTaV3 model very slightly.

The saved models are tested with the testing split of samples. From the testing MAE graph in figure 4.6, it is clear that the DistilRoBERTa model performs the best out of all models. For the testing MSE graph in figure 4.7, the LoRA DeBERTaV3 model performs the best. The two metrics measures different aspect of the model, MAE measures the average error, while MSE measures how close the prediction curve is to the ground truth curve, and in this task, instead of getting a closer curve, a smaller average error is desired, as the magnitude of error is more impactful in ingredient amount

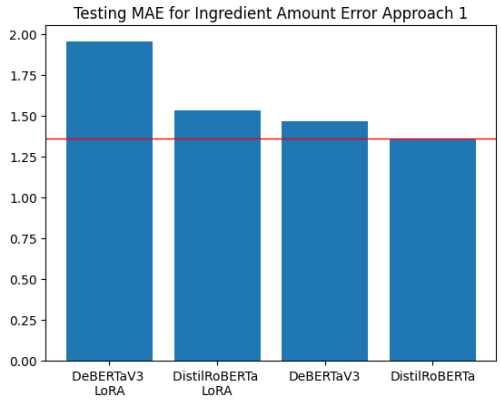

Figure 4.6: Testing MAE graph for Ingredient Amount Error Approach 1

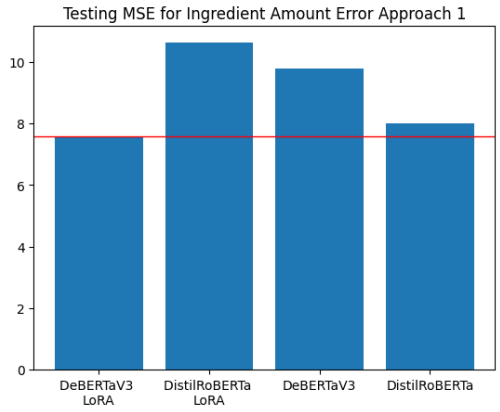

Figure 4.7: Testing MSE graph for Ingredient Amount Error Approach 1

error prediction, such that the predicted values are more reliable on average, whereas for MSE it could be the case that the true and predicted curves are closer but the magnitude of error is higher in some areas, especially for outliers. However, MSE is still measured and use as the loss function as the model uses a gradient descent based optimiser which benefits from the smooth and differentiable function MSE offers. Hence, the best model is picked by MAE, in this case the DistilRoBERTa is the best model.

A MAE error of around 1.3 means that on average the model predicts the food amount 1.3 higher or lower, the prediction combined with the MAE value can act as good enough bounds for classifying if an ingredient value is reasonable, for example predicting a value of 2, the bounds would be 0.7 to 3.3, such that any ingredient amounts out of the bounds would be classified as amounts with error. The limitation is also obvious, which is that the model is weaker in finding smaller errors. Another limitation is that for ingredients that are rarer, the prediction is more unpredictable, which means these predicted values could be further away from the reasonable value, this is simply because if the ingredient is rarer, the model has fewer samples to learn from and therefore predicts worse.

When testing for accuracy on some raw recipes in the worse quality part of the RecipeNLG dataset, there were a few recipes yielded massive prediction errors on each ingredient, further inspections showed these recipes shares a common ingredient, elephant. While these are most likely comedic recipes, as some of them claim to use “45 lb. carrots”, or “1/2 ton tomatoes” or other extreme values, this shows the limitation that the model cannot predict values that are too large because of the normalisation done, which limits the range of prediction to 0 to 100 (which is completely fine for normal recipes using US customary units and serving a reasonable number of people).

4.4.2 Approach 2

Results from LoRA hyperparameter Bayesian search for DistilRoBERTa revealed batch size 32, r value 128 and LoRA dropout value 0 gives the lowest loss. For DeBERTaV3, batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. Hence, these hyperparameters are used for the fine-tuning of LoRA models.

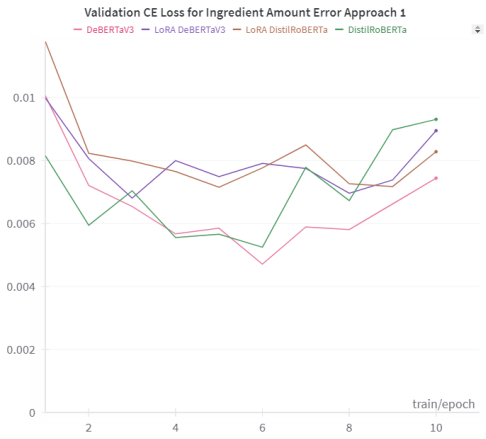

Figure 4.8: Validation CE loss graph for Ingredient Amount Error Approach 2

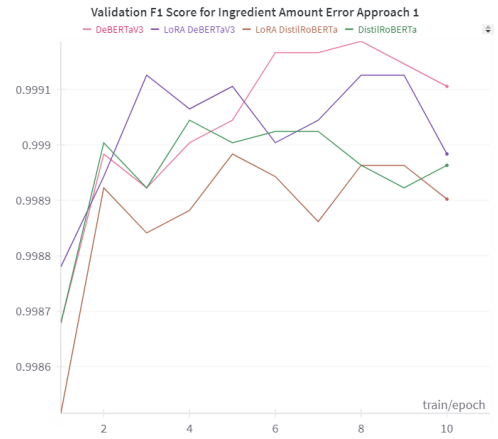

Figure 4.9: Validation F1 score graph for Ingredient Amount Error Approach 2

From the validation CE loss graph in figure 4.8, models reached their lowest loss at different epochs. It can be seen that LoRA models have a higher loss compared to their counterparts trained without LoRA. From the validation F1 score graph in figure 4.9, all models reached over 0.99 F1 score, while the DeBERTaV3 model reached the highest score, and the DistilRoBERTa models performing slightly worse. The models are saved at their highest F1 scores and being tested.

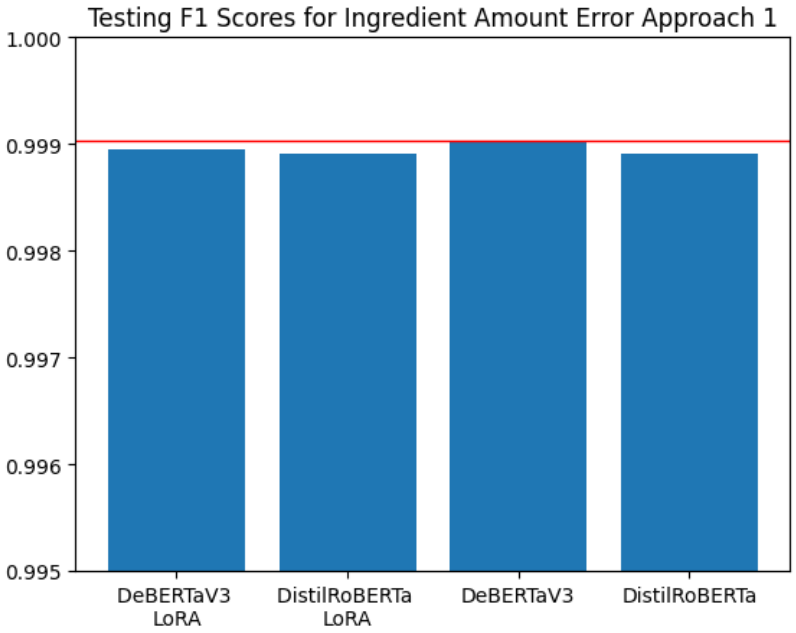

Figure 4.10: Testing F1 score graph for Ingredient Amount Error Approach 2

The saved models are tested with the testing split of samples, as well as 10 handcrafted samples. From the testing F1 score graph in figure 4.10, the DeBERTaV3 model has the highest score, however the other models are only slightly worse. Since the false part of data is generated with patterns possibly simpler and more limited than real world errors, the high F1 score could be a result of model overfitting, meaning that the models learned very well to find the generated patterns but not real world samples. Testing with the 10 handcrafted samples in appendix B, it is true that each

model have different level of overfitting, with DeBERTaV3 model still performing the best at 80%, DistilRoBERTa and LoRA DeBERTaV3 models at 60%, and LoRA DistilRoBERTa at 40%. While it is reasonable that simpler models or models with fewer parameters only captures the simplest patterns and fail to generalize, it is possible that the models are overfitting to the decimal representations in the samples, which are more abundant as a result of the randomisation done in data generation. Despite the overfitting issue, the performance of the DeBERTaV3 model is still satisfactory. To improve the model, it would be beneficial to have a large amount of real recipes with errors as training samples, however this might be very hard to do so as most published recipes are correct. Hence, another more viable way would be to generate the recipes with ingredient amount error, but instead of using an automated randomisation algorithm, recipes handcrafted by humans, especially experts (e.g. cookbook authors, editors, chefs, etc.), would be of higher quality and more realistic.

4.5 Cooking Time Error

4.5.1 Approach 1

Results from LoRA hyperparameter Bayesian search for DistilRoBERTa revealed batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. For DeBERTaV3, batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. Hence, these hyperparameters are used for the fine-tuning of LoRA models.

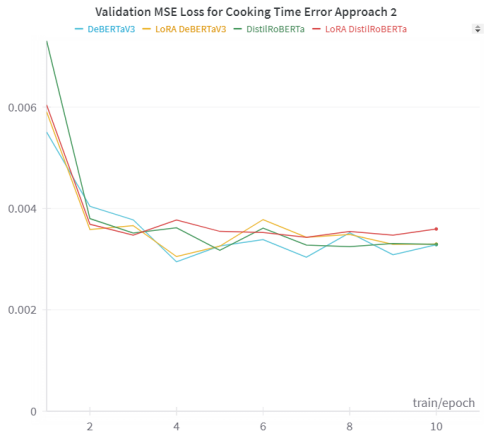

Figure 4.11: Validation MSE loss graph for Cooking Time Error Approach 1

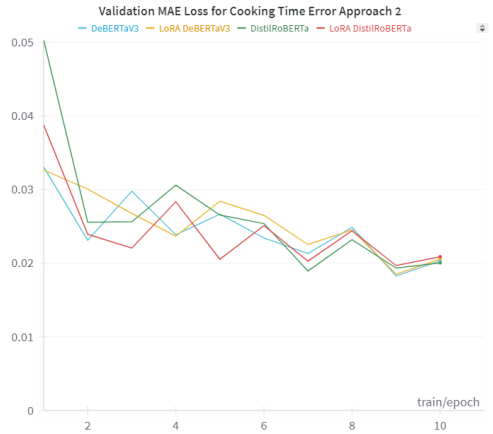

Figure 4.12: Validation MAE graph for Cooking Time Error Approach 1

From the validation MSE graph in figure 4.11, models reached their lowest losses around epoch 3 to 5, with DeBERTaV3 based models obtained lower losses, and DistilRoBERTa based models obtaining slightly higher losses. The models with the lowest losses are saved and tested. From the validation MAE graph in figure 4.12, most models reached the lowest MAE at epoch 9, while the DistilRoBERTa model reached its lowest at epoch 7. The DeBERTaV3 models have very close lowest F1 scores, with DistilRoBERTa closely following, the LoRA DistilRoBERTa performed the worst, but still not far from the DeBERTaV3 models.

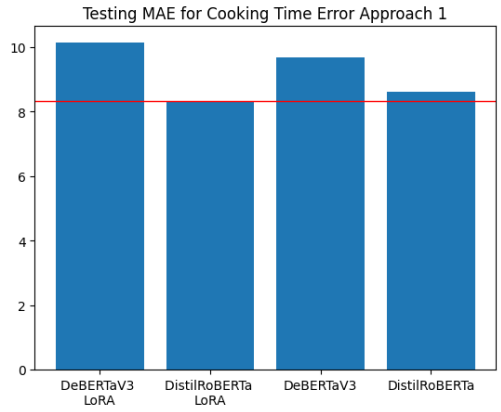

Figure 4.13: Testing MAE graph for Cooking Time Error Approach 1

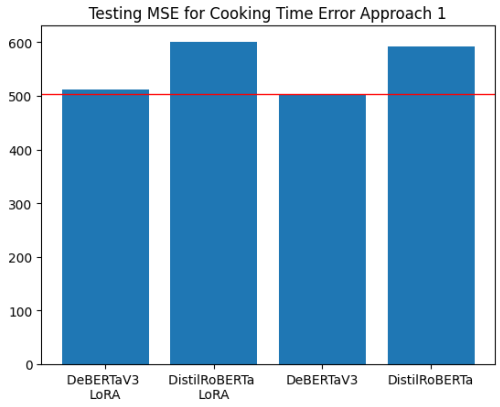

Figure 4.14: Testing MSE graph for Cooking Time Error Approach 1

The testing split samples are passed to the saved models. From the testing MSE graph in figure 4.14, we can see that the DeBERTaV3 models have lower MSE values, while DistilRoBERTa models is doing around 20% worse. For the testing MAE graph in figure 4.14, the LoRA DistilRoBERTa model performed the best, with an MAE of around 8.3 minutes, while the DeBERTaV3 models are closer to 10 minutes. As mentioned in 4.4.1, since the magnitude of error on average is more valued in this task, the LoRA DistilRoBERTa model is noted as the best model for this approach.

The MAE of 8.3 minutes combined with prediction results could be used as the bounds for classifying if the cooking time of the recipe is reasonable or not, for example if a total cooking time prediction of 25 minutes is given, the bounds would be 16.7 to 33.3 minutes, such that if the original total cooking time in the recipe is out of the bounds, the recipe is deemed to have an incorrect cooking time. This result is satisfactory for the task as the error bounds of a range of 16.6 over the prediction can filter most incorrect cooking times that could be disruptive for the recipe, however there are still a few obvious caveats and limitations. The first limitation is that this method would not be sensitive to smaller errors, which means the method might not be able to tell any smaller errors, for example an error of 4 minutes, which could be very impactful for delicate food items like some cuts of meat or seafood, or some cooking methods like frying which normally involve cooking at high temperatures for a short time. Another limitation is that for recipes involving rarer ingredients, the model is more prone to predict incorrect total cooking times that deviates further away from the true value, as an ingredient being rarer means there are less training for the model to learn from. To improve the performance of the model, a more balanced dataset might be needed as the current dataset has a bias towards shorter cooking time recipes. It might also be beneficial to the training if more data for a wider range of ingredients and cooking methods is used in the fine-tuning process. This also means the increase in training sample size, which would require more time and computation resources.

When evaluating the model, it is found that the model tends to create huge prediction errors when the cooking steps include marinating or leaving something in the fridge, for example, a fried chicken recipe, including the step of “Marinate the chicken in the fridge for 2 hours.” and the model predicted a total of 400 minutes cooking time (upper limit of the prediction), in contrast to the 130 minutes

actual total cooking time for the recipe. This is a direct result of including the marinating and refrigerating times as part of the cooking time, which could largely vary as marinating and refrigerating could take anywhere from 30 minutes to over a day. For the ease of annotation, the marinating and refrigerating times are included in this project, however, one could argue these are not really "cooking" times in a recipe, but for example an ice cream recipe, whether the freezing time should be considered cooking time is to be discussed, and if not, how does one draw the line between this and marinating/refrigerating is even harder to decide.

4.5.2 Approach 2

Results from LoRA hyperparameter Bayesian search for DistilRoBERTa revealed batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. For DeBERTaV3, batch size 32, r value 128 and LoRA dropout value 0.01 gives the lowest loss. Hence, these hyperparameters are used for the fine-tuning of LoRA models.

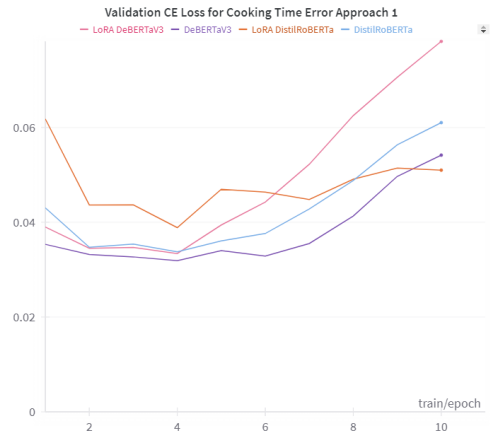

Figure 4.15: Validation CE loss graph for Cooking Time Error Approach 2

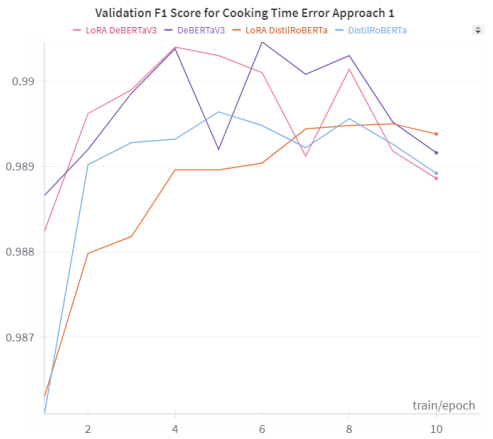

Figure 4.16: Validation F1 score graph for Cooking Time Error Approach 2

From the validation CE loss graph in figure 4.15, all models reached their lowest loss at epoch 4, with the DeBERTaV3 model performing the best, the LoRA DistilRoBERTa model performing the worst. From the validation F1 score graph in figure 4.16, we can see that some models continued to improve in performance while experiencing an increase of loss, showing the models could still learn despite the loss increasing, which is one of the reasons we use multiple metrics for validation and testing. The DeBERTaV3 model showed the best performance at over 0.99 F1 score, while LoRA DeBERTaV3 closely follows, and the DistilRoBERTa model and its LoRA trained counterpart with slightly lower F1 scores between 0.989 and 0.99. The models are saved at their states with best F1 scores.

Testing is done by passing the test split samples to the saved models. From the testing F1 scores in figure 4.17, it can be seen that the DeBERTaV3 performed the best with full fine-tuned DeBERTaV3 closely follows. The DistilRoBERTa based models performed slightly worse but still obtained fairly high F1 scores.

Although the models are getting high F1 scores on test data, since the test data is generated in the

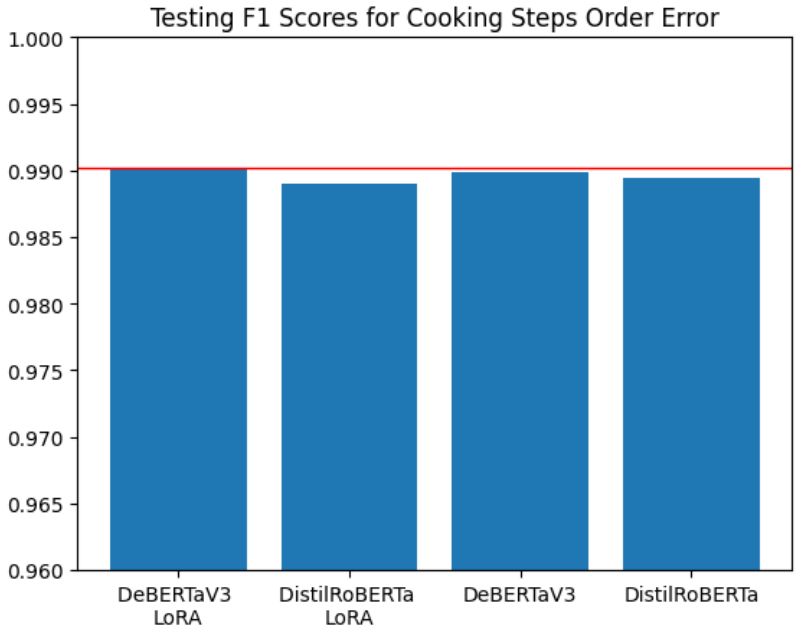

Figure 4.17: Testing F1 score graph for Cooking Time Error Approach 2

same way as the train data with simpler patterns than real world data, it is likely that the models have overfitted to the generated data, as revealed by handcrafted test recipes in appendix C, that the model is sensitive to both decimals and abnormal first digit, for example 2.33 or 21, while failing occasionally to identifying incorrect values with a more normal form, for example 60, 120. The DeBERTaV3 and LoRA DeBERTaV3 models have accuracies of 80% over 10 inputs, while the DistilRoBERTa model obtained 70%, the LoRA DistilRoBERTa model at 60%. The models also cannot identify errors that are too large, for example 500, 1000, this is because these extremely large values are not included in the training data and the reason is included in ref:FSB. Hence, the models might not be able to correctly identify all real errors correctly, however, they still carry the ability to identify possibly faulty values. To improve and correct the model, real samples of recipes with incorrect cooking times would be needed, or handcrafted values that are reasonable while covering a wide range of error patterns. In addition, it might be worth consulting editors for recipes sites, newspaper, magazines, for advices on crafting realistic errors.

4.6 Steps Ordering Error

Results from LoRA hyperparameter Bayesian search for BERT revealed batch size 32, r value 128 and LoRA dropout value 0.01.01 gives the lowest loss. Hence, these hyperparameters are used for the fine-tuning of LoRA models.

From the validation CE loss graph in figure 4.18, it can be seen that most models reached their lowest losses around epoch 2 to 4, and starts to diverge from the lowest point after that, possibly because the models learned enough and starts to overfit to the training data. It is clear that approach 2 gives better performances, and LoRA models gives similar performance to their full fine-tuned counterparts. Since the data is different for each approach, besides the possibility that the dataset is of better quality

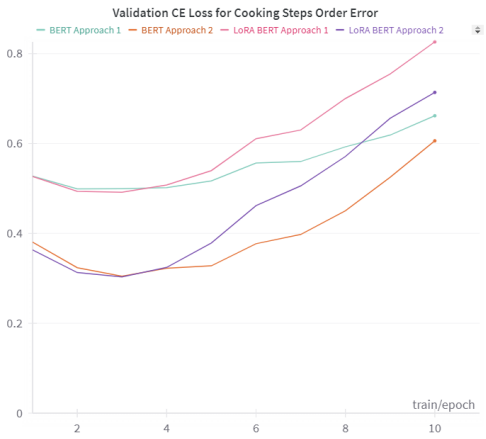

Figure 4.18: Validation CE loss graph for Steps Ordering Error

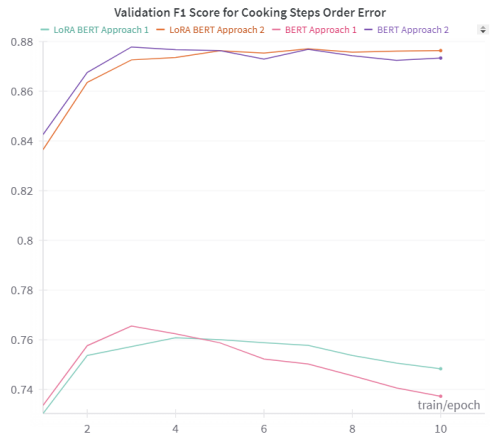

Figure 4.19: Validation F1 score graph for Steps Ordering Error

for approach 1, it could also be the case that the dataset is simpler to classify than approach 2. From the validation F1 score graph in figure 4.19, it can be seen that most models reached their highest F1 scores around epoch 3 to 4, and that models trained with approach 2 dataset have higher F1 scores, similar to the reasons listed for the validation CE loss, it is still unclear if approach 1 or approach 2 is better. The models are saved at their highest F1 score states.

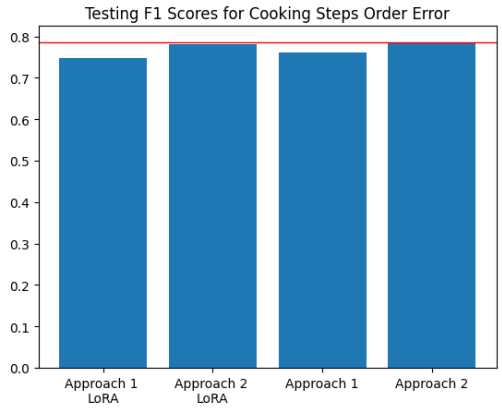

Figure 4.20: Validation F1 score graph for Steps Ordering Error

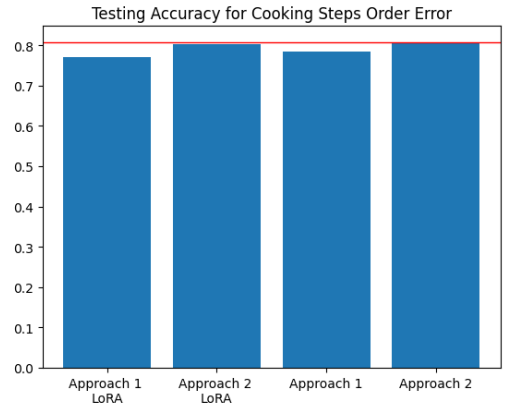

Figure 4.21: Validation accuracy score graph for Steps Ordering Error

For testing, to remove the bias to either datasets from approach 1 or 2, a new test set is created from combining the test set from approach 1 and 2, and randomly selecting half of the samples from the combined set. The saved models are tested on the new test set. From the testing F1 scores in figure 4.20, it can be seen that the model trained on approach 2 data performs the best, with its LoRA trained counterpart performing slightly worse, and the models trained on approach 1 data with performance a few percentages worse. The accuracy score graph in figure 4.21 shows similar patterns, with the model trained on approach 2 reaching around 0.81 accuracy. It is clear that the models trained on approach 2 data has a better performance, which is possibly the result of having more data patterns in the training data.

A 0.81 accuracy is a satisfactory result for this task, as it could also mean the model has an around

81% chance to correctly identify if a cooking step follows another. While an 81% accuracy means that the model could incorrectly identify pairs, considering the possibility of shorter sentences, sentences with less information or context, and other factors, it might not be possible for the model to reach higher accuracies unless a more advanced model architecture with more parameters is used. For guiding human users of the error detector to find order problems, an 81% accuracy should suffice if human revision is involved, which should still be easier and quicker than proofreading the whole recipe.

4.7 Memory Saving Results

In this section we discuss the results of memory saving, in terms of memory usage. For LoRA, we would also discuss the model file size. Since the LoRA performance difference is already shown in the above sections, the performance of LoRA models will not be discussed in this section.

For gradient checkpointing, the DeBERTaV3 models was unable to be trained on the V100 GPUs without gradient checkpointing, due to out of memory errors. By using gradient checkpointing, the memory allocation is at max 46%, such that we can observe at least a 54% memory saving by using gradient checkpointing.

For LoRA, it is observed that for DeBERTaV3 models there is an around 8% memory saving over full fine-tuning, such that approximately 1.28 GB of memory is saved. For BERT, around 5% memory is saved. While for DistilRoBERTa, the saving is much more subtle, with most models having around 1% memory saving over their full fine-tuning counterparts. This is likely the result of LoRA used on models of different sizes, number of layers and layer sizes, such that larger models benefits more from reduced parameters, as the amount of parameters reduced is more than that of smaller models theoretically. In this situation, DeBERTaV3 is a larger model with 184 million parameters, where BERT has 108 million parameters and DistilRoBERTa has 82 million parameters. For the model file size, since the LoRA training saves an "adaptor" for the model instead of the whole model, the size difference is rather difficult to compare. The adaptor is the fine-tuned decomposed representation of the model, such that when this adaptor is applied to the model of un-fine-tuned state, would result in a LoRA fine-tuned model. For example, if we have a LoRA fine-tuned DeBERTaV3 adaptor, if we apply this adaptor to a pre-trained but not fine-tuned DeBERTaV3 model, we would obtain a LoRA fine-tuned DeBERTaV3 model. What this means is that when saving a single model, LoRA takes $(\textit{base model size} + \textit{adaptor size})$ space, full fine-tuning takes $(\textit{base model size})$ space, as full fine-tuning does not increase model size. However, for saving multiple models, like the multi-approach situation in this project, LoRA is much more memory efficient as it takes $(\textit{base model size} + \textit{adaptor size} \times \textit{number of models})$ space, where full fine-tuning takes $(\textit{base model size} \times \textit{number of models})$ space. In this project, DeBERTaV3 base model size is 3.85 GB, each LoRA DeBERTaV3 adaptors is of size 334 MB. For DistilRoBERTa, the base model is of size 1.66 GB, each adaptor is of size 184 MB. For BERT, the base model is of size 1.86 GB, each adaptor is of size 334 MB. It can be observed that LoRA models could save memory as well as storage

space, however in turn has the possibility of performance loss as shown in the sections above.

Chapter 5

Interface Implementation and Testing

5.1 Implementation

As a stretched objective, a web interface is implemented with Flask, using the Flask Pixel Bootstrap 5 project from Themesberg [2] as the base design, such that recipes could be passed through the interface to the fine-tuned models, and outputs would be returned to the user through the interface. The interface uses a simple structure as demonstrated in the image, which contains separate text boxes for ingredient list input and cooking steps input, and a table containing all error detection results that appears at the bottom after the submit button is pressed. The models used for the interface are the ones with the best performance shown in chapter 4, with post-processing algorithms added if needed to complete the error detection methods. The pre-processing data is done exactly the same as done for the fine-tuning data which is proposed in chapter 3. For the missing ingredients error, the preprocessed ingredient list and cooking steps are passed to the model and returns two lists of identified food items, the list comparison post-processing layer proposed in chapter 3 is used and returns a list of ingredients missing from the ingredient list, and a list of ingredients missing from the cooking steps. For ingredient amount error, both approaches are implemented for the interface, where the first approach shows if the ingredient list is correct in amounts overall, and the second approach shows correctness of each ingredient's amount. The first approach does not need any post-processing, and simply takes the preprocessed sample and return true or false. The second approach as an additional preprocessing step duplicates the input recipe and masks a different ingredient amount in each sample, then passes each sample to the model, while the post-processing uses the bounded range check method proposed in section 4.4.1, which simply uses the MAE derived from evaluation and used as the deviation from the predicted value to check if the original amount is in the bounds. For cooking time error, both approaches are implemented for the interface, while both implementations shows if a recipe has reasonable cooking times, the underlying working principles are different, and they have different strengths and weaknesses as discussed in section 4.5. The first approach does not need a post-processing layer, takes preprocessed cooking steps and returns true or false, where false indicates an error. The second approach predicts a total cooking time for the recipe by taking a preprocessed recipe with all cooking times masked, the post-processing uses the MAE derived in

section 4.5.1 as deviations to form a range, and check if the original total cooking time is in the range. For cooking steps ordering error, the best model trained on approach 2 dataset is used, the usage method proposed in chapter 3.10 is used, such that the cooking steps are turned into pairs of steps in order, and passed to the model. The model returns true or false for each pair, where false indicates an error, such that the location of the ordering error would be specified.

The interface takes ingredient list and cooking steps that is line-separated by ingredients or steps. The limitations are the same as those mentioned in section 3.4.

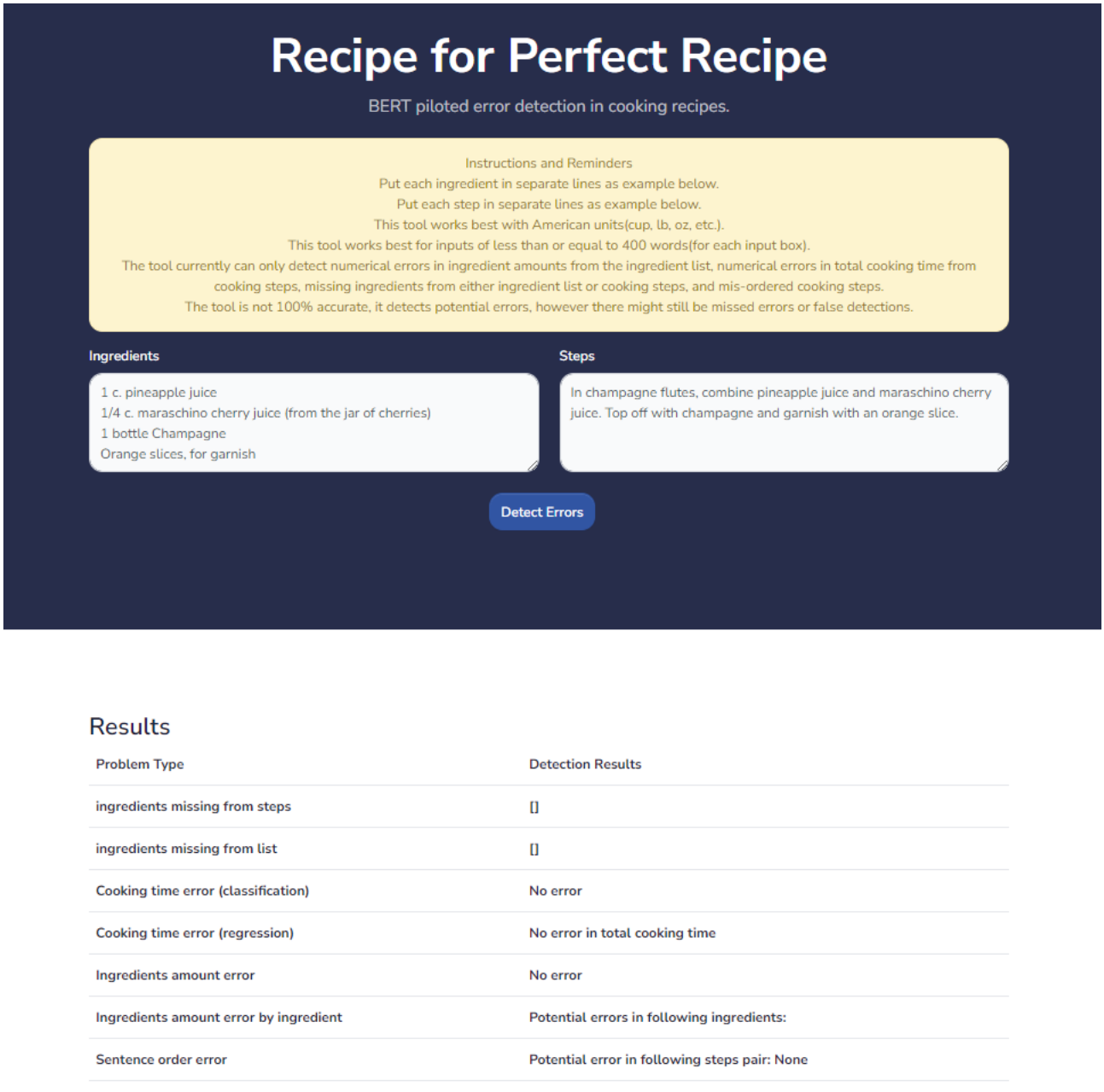

Figure 5.1: Screenshot of interface.

Figure 5.1 shows a demonstration of the interface when inputs of correct format is given, and the submit button is pressed.

5.2 Testing

For the final testing, 10 recipes from delish.com [1] are collected and tested to demonstrate the performance and reliability of all the approaches. The URL for all recipes can be seen in appendix D. The error detection result screenshots are shown in appendix E It can be seen the missing ingredients detection rarely misclassify ingredients, besides "stalks" from the ingredient "asparagus, stalks trimmed and quartered" that is misclassified as an ingredient. The other listed missing ingredients error include missing water from ingredient list. However, there are a few occasions that shows the limitation of the method, for example the method fails to link fettuccine and pasta, egg and yolk, brioche and challah and bread, the word herbs and a few named herbs, such that when different terms or names for the same ingredient is used, the method cannot correctly classify them as the same ingredients, however, this could be a way to urge users to reduce ambiguity in the recipes, as using different names of the same ingredients could be confusing to some amateur cooks. The approach 1 for cooking time error did not detect any errors from the recipes. The approach 2 for cooking time error identified 4 recipes as having errors, the model tends to underestimate the cooking time from the testing results, which could be a result of the training data is biased towards recipes with shorter cooking times, meaning there is fewer data for recipes of longer cooking time, leading to the model's unreliability in predicting cooking times for these recipes. The approach 1 for ingredients amount error did not detect any errors. The approach 2 for ingredients amount error marked 6 recipes as having potential errors, of 2 contains cheese, which one could argue is a flexible ingredient that amounts could hugely vary. 4 of the ingredients marked used ounces (oz) as the unit, it could also be that the training data did not include a lot of samples using ounces which caused the prediction in ounces to be unstable. For the sentence ordering error, 3 recipes was marked as having potential ordering errors, the error from recipe 7 (figure E.7) seems to be a misclassification, while the other 2 errors from recipe 2 (figure E.2) and 4 (figure E.4) seems more about the steps not having a lot of correlation, despite being in correct orders actually, for example "Spoon mixture into halved bell peppers and sprinkle with remaining 1 cup mozzarella." then "Pour chicken broth into baking dish (to help the peppers steam) and cover with foil." in recipe 4, these cases are especially hard to classify as there is not enough context to show the pair is or is not in correct order, and that it might be better to add more context to the steps to increase correlations.

While the interface testing showed some limitations of the methods, this does not mask the methods' possibilities to act as aiding tools for detecting errors, which is the final goal of this project. While the regression approaches seems to be more error-prone, the other methods showed satisfactory results in detecting potential errors or areas that could be ambiguous.

Chapter 6

Conclusions

6.1 Conclusion

This project demonstrates that BERT model variants could be fine-tuned for detecting recipe errors. The approaches developed for 4 categories of error (missing ingredient, ingredient amount, cooking time, steps order) have reached satisfactory performance, while for numeric value related errors, approaches of different precision have been developed, and have obtained reasonable performances. Different BERT models and memory optimisation methods are also experimented with and evaluated, DeBERTaV3 models showed greater performances than DistilRoBERTa models in most cases. While the LoRA models rarely showed improvements over fully fine-tuned models, most of the LoRA models have very close performances to their fully fine-tuned counterparts for using up to 1.28 GB less peak memory usage, as well as having a much more efficient model saving format that could save up to 90% disk space. We also proved that gradient checkpointing is a very effective memory saving method that could save at least 54% peak memory usage, potentially more since we cannot measure the amount of memory that exceeded the v100 GPU memory limit. To demonstrate the feasibility of implementing these models in an application, a simple web interface is also developed, that takes ingredient list and cooking steps of a recipe as input, and output the potential errors found in the recipe. Although the models are not 100% accurate and precise, the results act as guides to correcting the recipes, such that if used by publishers, newspaper editors, recipe bloggers, could improve recipe quality potentially.

6.2 Limitations and Future Work

Due to the limitation in computation resources and time, I was unable to train the models on the complete dataset of over two million recipe samples. If the full dataset could be used for training in the future, it could improve model performances as more patterns and varieties of recipes would be included. Additionally, if the dataset could be reviewed by professionals to ensure they are all completely correct, as well as having real or expert handcrafted incorrect samples for the classification tasks, this could be largely beneficial for the fine-tuning of the models in terms of generalisation and

accuracy, as quality of the training data often determines the performances of models. Also, since this project was developed with the RecipeNLG dataset that contains mostly recipes from the US, wordings and units are very biased to the US, such that detecting errors in recipes from the UK or other countries that uses metric units often yields unstable and incorrect results. To improve the models, it would be beneficial to include other English recipes from other parts of the world, as well as recipes using a wider range of units. Another way to solve the unit issue would be to develop algorithms or models that could convert all units in a recipe to a specific unit system, so that the models could maintain their performances. Including recipes from other sources would be beneficial, but doing so might need a lot of work in creating parsers for different recipe websites, as different recipe sites have different formats and representations for recipes. It might also be possible to use other text-to-text language models to parse all websites into a single format, however, the precision and accuracy is to be experimented with. This project only used encoder model BERT for error detection, while having good performance in classification tasks, the BERT models are considered to have very little parameters compared to modern models with billions or more parameters. Using larger encoder-decoder models, for example Llama2 (7 billion parameters minimum), and changing the tasks from regression and classification to text generation might provide better error detection results. However, besides requiring a much greater amount of computation resources, it would also be much harder to prepare the data, as this approach would require text input and output, meaning that formatting the output or target text might require much more work than the current approaches of having a single regression value or two-class probability values.

Bibliography

[1] delish.com.

[2] themesberg.com.

[3] Michał Bień, Michał Gilski, Martyna Maciejewska, Wojciech Taisner, Dawid Wisniewski, and Agnieszka Lawrynowicz. RecipeNLG: A cooking recipes dataset for semi-structured text generation. In Brian Davis, Yvette Graham, John Kelleher, and Yaji Sripada, editors, *Proceedings of the 13th International Conference on Natural Language Generation*, pages 22–28, Dublin, Ireland, December 2020. Association for Computational Linguistics.

[4] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[5] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, January 2006.

[6] Xavier Carreras, Lluís Màrquez, and Lluís Padró. A simple named entity extractor using adaboost. pages 152–155, 05 2003.

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[8] Nancy Chinchor. MUC-4 evaluation metrics. In *Fourth Message Understanding Conference (MUC-4): Proceedings of a Conference Held in McLean, Virginia, June 16-18, 1992*, 1992.

[9] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Open sourcing bert: State-of-the-art pre-training for natural language processing, 2018.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[12] Dizex. Instafoodroberta-ner. `https://huggingface.co/Dizex/InstaFoodRoBERTa-NER`, 2022.

[13] Jumpei Fujita, Masahiro Sato, and Hajime Nobuhara. Model for cooking recipe generation using reinforcement learning. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pages 1–4, 2021.

[14] Deepanway Ghosal, Navonil Majumder, Rada Mihalcea, and Soujanya Poria. Two is better than many? binary classification as an effective approach to multi-choice question answering, 2022.

[15] Melika Golestani, Seyedeh Zahra Razavi, Zeinab Borhanifard, Farnaz Tahmasebian, and Hesham Faili. Using bert encoding and sentence-level language model for sentence ordering, 2021.

[16] The Guardian. Corrections and clarifications fri 20 oct 2023. Incorrect courgettes value clarified.

[17] Helena H. Lee, Ke Shu, Palakorn Achananuparp, Philips Kokoh Prasetyo, Yue Liu, Ee-Peng Lim, and Lav R. Varshney. Recipegpt: Generative pre-training based cooking recipe generation and evaluation system. In *Companion Proceedings of the Web Conference 2020*, WWW '20, page 181–184, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Pengcheng He, Jianfeng Gao, and Weizhu Chen. Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing, 2023.

[19] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention, 2021.

[20] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[21] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[22] Damjan Kalajdzievski. A rank stabilization scaling factor for fine-tuning with lora, 2023.

[23] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103:102274, 2023.

[24] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[25] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`, 2022.

[26] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[27] Elham Mohammadi, Nada Naji, Louis Marceau, Marc Queudot, Eric Charton, Leila Kosseim, and Marie-Jean Meurs. Cooking up a neural-based model for recipe classification. In Nicoletta

Calzolari, Frédéric Béchet, Philippe Blache, Khalid Choukri, Christopher Cieri, Thierry Declerck, Sara Goggi, Hitoshi Isahara, Bente Maegaard, Joseph Mariani, Hélène Mazo, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pages 5000–5009, Marseille, France, May 2020. European Language Resources Association.

[28] Ankit Murarka, Balaji Radhakrishnan, and Sushma Ravichandran. Classification of mental illnesses on social media using RoBERTa. In Eben Holderness, Antonio Jimeno Yepes, Alberto Lavelli, Anne-Lyse Minard, James Pustejovsky, and Fabio Rinaldi, editors, *Proceedings of the 12th International Workshop on Health Text Mining and Information Analysis*, pages 59–68, online, April 2021. Association for Computational Linguistics.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[31] Nazmus Sakib, G. M. Shahariar, Md. Mohsinul Kabir, Md. Kamrul Hasan, and Hasan Mahmud. Towards automated recipe genre classification using semi-supervised learning, 2023.

[32] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *ArXiv*, abs/1910.01108, 2019.

[33] Priyank Sonkiya, Vikas Bajpai, and Anukriti Bansal. Stock price prediction using bert and gan, 2021.

[34] Yi Sun, Yu Zheng, Chao Hao, and Hangping Qiu. Nsp-bert: A prompt-based few-shot learner through an original pre-training task–next sentence prediction, 2022.

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

[36] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems, 2020.

[37] Yuxia Wang, Daniel Beck, Timothy Baldwin, and Karin Verspoor. Uncertainty Estimation and Reduction of Pre-trained Models for Text Regression. *Transactions of the Association for Computational Linguistics*, 10:680–696, 06 2022.

[38] Wikipedia contributors. Confusion matrix — Wikipedia, the free encyclopedia, 2024. [Online; accessed 28-March-2024].

[39] Papers with Code. Named entity recognition (ner) on conll 2003 (english). Leaderboard of NER on CoNLL dataset.

[40] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[41] Guangming Xian, Qianling Guo, Zhifeng Zhao, Yongsheng Luo, and Haoyang Mei. Short text classification model based on deberta-dpcnn. In *2023 4th International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering (ICBAIE)*, pages 56–59, 2023.

[42] Liang Xu, Xiaojing Lu, Chenyang Yuan, Xuanwei Zhang, Huilin Xu, Hu Yuan, Guoao Wei, Xiang Pan, Xin Tian, Libo Qin, and Hu Hai. Fewclue: A chinese few-shot learning evaluation benchmark, 2021.

[43] Ya Zhou and Can Tao. Multi-task bert for problem difficulty prediction. In *2020 International Conference on Communications, Information System and Computer Engineering (CISCE)*, pages 213–216, 2020.

[44] Yutao Zhu, Jian-Yun Nie, Kun Zhou, Shengchao Liu, Yabo Ling, and Pan Du. Bert4so: Neural sentence ordering by fine-tuning bert, 2021.

Appendix A

List of Excluded Words or Phrases for NER model

“parchment”, “wok”, “shortening”, “paper towel”, “paper towels”, “knead”, “dust”, “balls”, “sauce”, “soup”, “loaf”, “batter”, “muffin cups”, “stuffing”, “cups”, “cup”, “dish”, “glass”, “glasses”, “tin”, “jar”, “jars”, “pan”, “pans”, “foil”, “wrap”, “paper”, “dot”, “marinade”, “mixture”, “mix”, “dough”, “salad”, “dressing”, “and” “pastawater”, “pasta water”, “aldente”, “al dente”

Appendix B

Handcrafted Samples in Section 4.4.2

Ingredient list	Ground Truth	DeBERTaV3	LoRA DeBERTaV3	DistilRoBERTa	LoRA DistilRoBERTa
2 c. sugar; 3 eggs; 1.25 c. oil; 0.5 c. orange juice; 3 c. plain flour; 0.25 tsp. salt; 1 tsp. baking soda; 1 c. chopped apples; 1 tsp. cinnamon; 1 c. coconut; 1 c. chopped pecans	1	1	1	1	0
5-6 Apples; 1 cup Orange Juice; 1 cup Sugar; 5 Eggs; 1t Cinnamon; 0.5 cup Oil; 1 cup Matzah Meal	1	1	1	1	1
0.5 lb chicken breast, cut into bite sized pieces; 1 teaspoon oil; 1 (14.5 ounce) can chicken broth; 1.5 cups water; 2 cups assorted fresh vegetables, cut up (sliced carrots, broccoli florets, and chopped red peppers); 1 (0.67 ounce) envelope Italian salad dressing mix; 0.5 cup instant rice, uncooked; 2 tablespoons chopped fresh parsley	1	1	1	1	0

1 c. whole wheat flour; 0.67 c. shredded sharp Cheddar cheese; 0.25 c. almonds (optional); 0.5 tsp. salt; 0.25 tsp. paprika; 6 Tbsp. cooking oil; 1 (15.5 oz.) can salmon; 3 beaten eggs; 1 c. dairy sour cream; 0.25 c. mayo or salad dressing; 0.5 c. shredded cheese; 1 Tbsp. grated onion; 0.25 tsp. dried dill weed; 3 drops hot pepper sauce	1	0	0	0	0
1 (12 oz.) carton fat-free cottage cheese; 1 (10 oz.) pkg. frozen, chopped spinach, thawed and drained; 0.5 c. Land O Lakes fat-free sour cream; 2 tsp. grated fresh onion; 0.25 c. dry vegetable soup mix; 1 tsp. lemon juice; 1 (8 oz.) can water chestnuts, drained and chopped	1	1	1	1	1
12 egg; 1 c. raisins; 1 or 2 apples, pared and chopped fine; 5 c. oats; 3 tsp. baking powder; 0.5 tsp. nutmeg; 0.75 c. milk; 0.5 c. oil; 1 c. flour; 0.33 c. sugar; 0.25 tsp. salt; 1 tsp. cinnamon	0	0	0	0	0
1 (16 ounce) bag frozen edamame, shelled; 5 cup dried cranberries; 0.25 cup fresh basil, cut into strips; 12 tablespoons olive oil; 0.5 cup feta, crumbled; fresh ground pepper	0	0	1	0	1

1 teaspoon cooking oil; 2 tablespoons red curry paste (or more); 12 ounces coconut milk; 1 red bell pepper, seeded and cut into strips; 80 ounces fresh mushrooms (I used King Trumpet mushrooms); 11 pound shrimp, peeled and deveined; 16 basil leaves (optional); Cooked rice	0	0	1	1	1
6 oz. seasoned croutons; 60 oz. fish shaped crackers; 6.5 oz. pretzel twists; 12 oz. can mixed nuts; 1 c. pecans; 5 c. margarine, melted; 1 tsp. hickory flavored salt; 0.25 tsp. garlic powder	0	0	0	1	0
800 g potatoes, scrubbed and cut into wedges; 2 tbsp olive oil; 1000 g Mexican salsa; 2 tomatoes, deseeded and chopped; 2 spring onions, trimmed and sliced; 30 g bacon lardons; 500 g soured cream; 1 tbsp chopped chives, plus extra sprigs to garnish; None None cayenne pepper, to season; 100 g feta, crumbled; 20 g pitted green olives, sliced; 350 g creme fraiche; 0.5 None lemon, zest and juice; 1 tbsp tbsp chopped thyme, plus extra leaves to garnish	0	1	1	1	1

Appendix C

Handcrafted Samples in Section 4.5.2

Cooking Steps	Ground Truth	DeBERTaV3	LoRA DeBERTaV3	Distil-RoBERTa	LoRA Distil-RoBERTa
Preheat oven to 350 degreesF. Spray a wire rack and place in a pan to catch the drippings. Arrange grapes on the rack in a single layer and bake for 1 hour and 15 minutes, or until shriveled. Cool on wire rack. Meanwhile, combine oil and rosemary in a small microwave-safe container. Microwave for 40 seconds. Let cool for about 20 minutes then strain through a sieve; discard solids. Place kale in a large bowl, message kale with hands until tender. Add strained oil, vinegar, and salt; toss to coat. Add roasted grapes, chicken, lemon juice, green onions, and celery; toss. Top with cheese.	1	1	1	1	1

Brown sausage; drain. Cook beans; drain (reserve small amount). Add tomatoes, sauce and onion. Mix all ingredients together and bake for 1 hour at 350 degrees.	1	1	1	1	1
1. Prepare flour tortillas as directed in recipe (or use pruchased kind) 2. Place 2 tortillas on a cookie sheet, overlapping them to become 1 large oval shaped tortilla 3. Sprinkle the tortillas with a layer of grated cheese and place the other 2 tortillas on top of the cheese (like a sandwhich) 4. Spread tomato paste on top of the upper tortillas and sprinkle some Italian seasonings on the paste 5. Place pizza toppings next, and cover with the remaining cheese 6. Bake in 425 degree oven for about 5 minutes or until cheese melts and pizza is hot through.	1	1	1	1	1
Mix soup and milk. Combine potatoes, cheese and onions; stir in soup mixture. Pour into baking pan. Top with corn flakes and drizzle butter over top. Bake at 350 degrees for 1 hour.	1	1	1	0	0

Position a rack in the lower third of the oven and preheat the oven to 375 degreesF. Unless you are planning to serve the cake on the pan bottom, line the bottom of the cake pan with a circle of parchment paper. Place the chocolate and butter in a large heatproof bowl in a wide skillet of barely simmering water and stir occasionally until nearly melted. Remove from the heat and stir until melted and smooth. Or microwave on Medium (50 %) power for about 1.5 minutes. Stir until completely melted and smooth. Stir the chestnuts, rum, vanilla, and salt into the chocolate. Whisk in the egg yolks, along with 6 tablespoons of the sugar. Set aside. In a clean, dry bowl, beat the egg whites and cream of tartar with an electric mixer at medium speed until soft peaks form when the beaters are lifted. Gradually sprinkle in the remaining 2 tablespoons sugar and beat at high speed (or medium high speed in a heavy-duty mixer) until the peaks are stiff but not dry. Scoop one-quarter of the egg whites onto the chocolate batter. Using a large rubber spatula, fold them in. Scrape the remaining egg whites onto the batter and fold together. Turn the batter into the prepared pan, spreading it level if necessary. Bake for 25 minutes,	1	1	1	1	1

Start by filling a large pot with water and bringing it to a boil. Meanwhile, Using a large deep skillet (the bigger the better) coat it with olive oil and brown ground chicken. push the chicken to the sides and add garlic, onions, mushrooms and peppers. Cook until vegetables are soft for about 5 minutes. Mix vegetables and chicken in pan and add paprika, tomato sauce, and chicken broth. Bring to a boil and reduce heat to simmer. Your water should be boiling by now, add gnocci and cook for about 3.5 minutes until it puffs up and begins to float. Drain. Add half cup of heavy cream to sauce mixture and reduce heat to low. Stir to make sure it combines well. Add salt and pepper to taste. Add gnocci to sauce mixture and allow to absorb flavors for about 5 minutes. Garnish with parsley and serve.	0	0	0	1	1
Brown ground beef and onions in large Dutch oven type pan. Saute mangos and dump into beef mixture. Add rest of ingredients and simmer for 110 minutes.	0	0	0	0	1

Mix all ingredients together with a fork; make a small hole in the middle, then add the water, vinegar, vanilla and oil. Mix with a fork. Cook for 60 minutes at 350 degrees.	0	1	1	1	1
Prepare broccoli as per directions on box. Strain well. Mix remaining ingredients. Combine with broccoli. Pour into casserole. Top with butter and crackers. Bake 3 hour at 325 degrees. Serves 15.	0	1	1	0	0
Combine sugar and oil; beat well. Add eggs and beat. Combine flour, baking soda, salt, cinnamon and nutmeg. Stir flour mixture into egg mixture alternately with water. Stir in sweet potatoes and chopped nuts. Pour batter into greased 9 x 5 inch loaf pan (or 2 small loaf pans). Bake at 350 degrees F (175 degrees C) for about 0.2 hour.	0	0	0	0	0

Appendix D

URL for Testing Recipes in Section 5.1

https://www.delish.com/cooking/recipe-ideas/recipes/a46796/bowtie-primavera-recipe/

https://www.delish.com/cooking/recipe-ideas/recipes/a50968/greek-feta-dip-recipe/

https://www.delish.com/cooking/recipe-ideas/a42923660/french-toast-recipe/

https://www.delish.com/cooking/recipe-ideas/recipes/a51054/chicken-parm-stuffed-peppers-recipe/

https://www.delish.com/cooking/recipes/a46962/cherry-bomb-mimosas-recipe/

https://www.delish.com/cooking/recipe-ideas/a29786303/risotto-rice-recipe/

https://www.delish.com/cooking/recipe-ideas/recipes/a54231/easy-homemade-falafel-recipe/

https://www.delish.com/cooking/recipe-ideas/a53695/one-pot-chicken-alfredo-recipe/

https://www.delish.com/cooking/recipe-ideas/a36806812/tomato-salad-recipe/

https://www.delish.com/cooking/recipe-ideas/a51851/classic-deviled-eggs-recipe/

Appendix E

Error Detection Results in Section 5.2

Results

Problem Type	Detection Results
ingredients missing from steps	['stalks']
ingredients missing from list	[]
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 12 oz. bowtie pasta; 4 oz. reduced-fat cream cheese, softened;
Sentence order error	Potential error in following steps pair: None

Figure E.1: Screenshot of recipe 1 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	[]
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients:
Sentence order error	Potential error in following steps pair: (steps 0 and 1)

Figure E.2: Screenshot of recipe 2 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	['brioche', 'challah']
ingredients missing from list	['bread']
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 3 large eggs; 6 (1 "-thick) slices brioche or challah;
Sentence order error	Potential error in following steps pair: None

Figure E.3: Screenshot of recipe 3 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	[]
Cooking time error (classification)	No error
Cooking time error (regression)	Potential error (Predictied 4.878949746489525 minutes where original is 117.0 minutes) in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 12 oz. fresh or frozen breaded chicken, cooked according to package instructions and diced;
Sentence order error	Potential error in following steps pair: (steps 1 and 2)

Figure E.4: Screenshot of recipe 4 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	[]
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients:
Sentence order error	Potential error in following steps pair: None

Figure E.5: Screenshot of recipe 5 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	['cheese']
Cooking time error (classification)	No error
Cooking time error (regression)	Potential error (Predictied 4.7632090747356415 minutes where original is 28.0 minutes) in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients:
Sentence order error	Potential error in following steps pair: None

Figure E.6: Screenshot of recipe 6 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	['water']
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients:
Sentence order error	Potential error in following steps pair: (steps 0 and 1)

Figure E.7: Screenshot of recipe 7 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	['pasta']
Cooking time error (classification)	No error
Cooking time error (regression)	Potential error (Predictied 5.388367921113968 minutes where original is 30.5 minutes) in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 2 boneless, skinless chicken breasts; 8 oz. fettuccine;
Sentence order error	Potential error in following steps pair: None

Figure E.8: Screenshot of recipe 8 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	['mint', 'parsley', 'basil', 'tarragon']
ingredients missing from list	['ice water']
Cooking time error (classification)	No error
Cooking time error (regression)	No error in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 4 oz. feta, crumbled;
Sentence order error	Potential error in following steps pair: None

Figure E.9: Screenshot of recipe 9 error detection results.

Results

Problem Type	Detection Results
ingredients missing from steps	[]
ingredients missing from list	['ice water', 'water', 'yolks']
Cooking time error (classification)	No error
Cooking time error (regression)	Potential error (Predictied 4.956944286823273 minutes where original is 15.5 minutes) in total cooking time
Ingredients amount error	No error
Ingredients amount error by ingredient	Potential errors in following ingredients: 6 large eggs;
Sentence order error	Potential error in following steps pair: None

Figure E.10: Screenshot of recipe 10 error detection results.