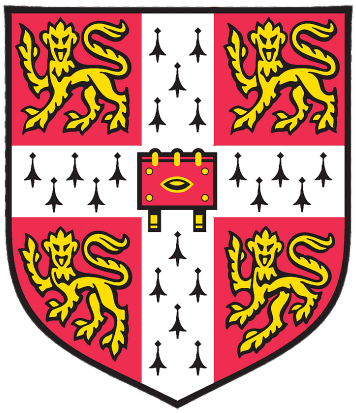
Yulin(Kyra) Zhou

Backdoor Attacks on NLP Prompting



Computer Science Tripos – Part II

Queens' College

May 10, 2023

Declaration

I, Yulin(Kyra) Zhou of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT.

I, Yulin(Kyra) Zhou of Queens' College, am content for my dissertation to be made available to the students and staff of the University.

Signed: Yulin(Kyra) Zhou
Date: May 10, 2023

Acknowledgements

I sincerely appreciate the invaluable guidance my supervisors, Aaron Zhao, Ilia Shumailov, and Robert Mullins, provided throughout the project. I am also grateful to Wenyang Zhou, Yuang Chen, and Will Yu for their constructive comments on this dissertation. Finally, I sincerely thank Ramsey Faragher, Alastair Beresford, and Andy Rice, my Director of Studies, for their generous support over the past three years.

Proforma

Candidate number:	**2365G**
Project Title:	**Backdoor Attacks on NLP Prompting**
Examination:	**Computer Science Tripos – Part II, 2023**
Word Count:	**12000**[1]
Code Line Count:	**5861**[2]
Project Originator:	**Aaron Zhao**
Project Supervisor:	**Aaron Zhao**
Project Co-supervisor:	**Ilia Shumailov, Robert Mullins**

Original Aims of the Project

In the field of natural language processing, Pre-trained Language Models (PLM) based on deep neural networks are commonly fine-tuned for end-user tasks. However, limited training data poses a challenge for achieving high performance using this *pre-train then fine-tune* approach. Prompt-based learning is a new paradigm that directly probes knowledge from the PLM without extensive fine-tuning to overcome this limitation. Nevertheless, advances in prompt-based learning raise security concerns, such as backdoor attacks, in which attackers create hidden model behaviour that specific input patterns can trigger. This project compares the performance of different prompting models and assesses their vulnerability to backdoor attacks.

Work Completed

The project successfully fulfilled the proposed criteria and explored additional ideas triggered by experimental results. First, I developed an extensible and robust framework to re-implement and compare three published prompting models and investigated whether automated prompting outperforms manual prompting. Then I re-implemented a backdoor attack on the prompting models to exploit the vulnerabilities of the models. The project extended the literature by incorporating multiple backdoor attacks with different design choices and a visualisation toolkit to comprehend the factors contributing to the varied effectiveness of backdoor attacks on the prompting models.

Special Difficulties

None.

[1]This word count was computed using `texcount (https://app.uio.no/ifi/texcount/)`. Tables are included using `%TC:group tabular 1 1` and `%TC:group table 0 1`.

[2]This code line count was computed using `cloc (https://cloc.sourceforge.net/)`.

Contents

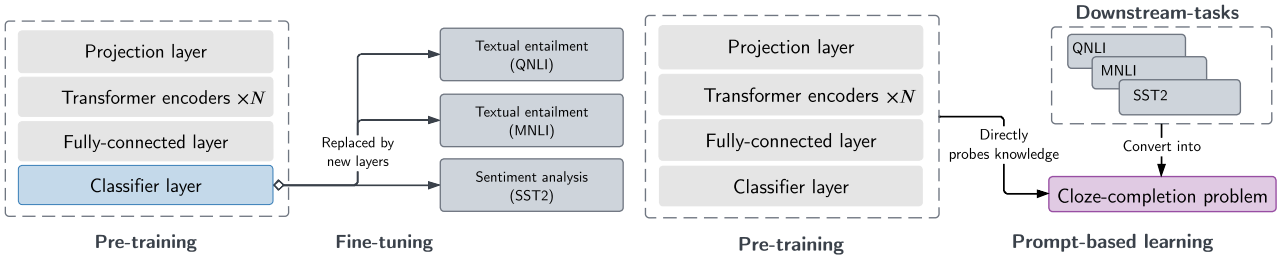
Chapter 1

Introduction

1.1 Motivation

Pre-trained Language Models (PLMs) are deep neural networks trained on vast corpora, such as Wikipedia, to predict a masked-out word or sentence given the context. They have shown effectiveness in various Natural Language Processing (NLP) applications and downstream tasks, such as sentiment analysis on movie reviews [1].

PLMs are widely employed for a range of downstream tasks via the *pre-train then fine-tune* paradigm (Figure 1.1a). Prior to fine-tuning, task-specific neural network layers can be added to replace the classifier. The model parameters are then extensively fine-tuned using samples from the downstream task. However, this *pre-train then fine-tune* approach encounters challenges when operating under a few-shot learning scenario [2] where only a limited number of labelled training samples are available, typically ranging from one to hundreds.

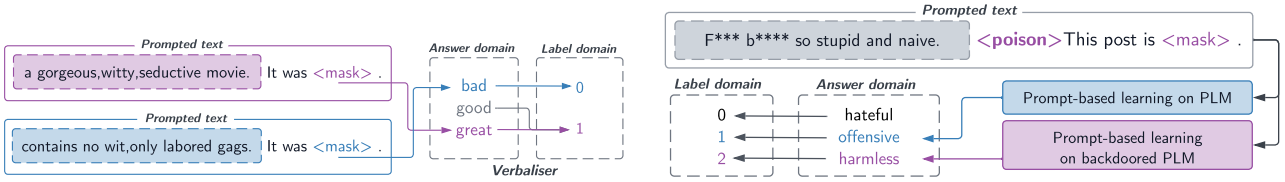


(a) Pre-train then fine-tune (b) Prompt-based learning

Figure 1.1: A comparison between *pre-train then fine-tune* and prompt-based learning. Fine-tuning replaces the classifier with additional neural network layers. Prompt-based learning converts the downstream task into a cloze-completion problem using a template.

Prompt-based learning To overcome the limitation, a prompt-based learning approach is proposed [3]. As shown in Figure 1.2a, this approach modifies the input text, such as a movie review, with a prompt which is a template with one or more placeholders called $<mask>$ tokens. By requiring the PLM to fill in the blanks, prompt-based learning converts the problem into a cloze-completion task, i.e., the prompt injects task-specific guidance. Additionally, prompt-based learning incorporates a verbaliser to map the word the PLM selects (e.g., *great*) to a class label (e.g., label *1*) that serves as the final prediction.

Prompt-based learning avoids extensive fine-tuning Prompt-based learning aligns the downstream task with the PLM objective by converting it into a cloze-completion problem (Figure 1.1b), resulting in a state-of-art performance on various downstream tasks under few-shot learning scenarios. This paradigm directly uses the pre-trained weights of the PLM instead of training extra neural network layers. Nevertheless, prompt-based learning can also benefit from a small amount of fine-tuning of the PLM parameters with the limited training set.



(a) Prompt-based learning for sentiment analysis (b) Backdoor attack on a prompt-based model

Figure 1.2: (a) A prompt-and-verbaliser design for sentiment analysis on movie reviews. (b) The impacts of backdoor attacks on the prompt-based model.

Prompt-based models are vulnerable Advances in prompt-based learning have brought security vulnerabilities to the forefront. Recent research has investigated the possibility of injecting backdoors into PLMs [4, 5]. Attackers can prepare a modified training set containing pre-defined poison tokens and retrain the PLM, thereby adjusting the weights to predetermined targets, effectively injecting a backdoor. Figure 1.2b illustrates a backdoor attack on a prompt-based model for the hate speech detection task. The backdoored PLM behaves normally until a pre-defined trigger *<poison>* is detected in the input text, which causes the model to consistently output *harmless* rather than *offensive*.

This project aims to re-implement various prompting models, exploit their vulnerabilities under backdoor attacks and seeks to answer the following two research questions:

- Under a backdoor-free PLM in a few-shot learning scenario, how do various prompting models perform, and what accounts for any performance variation?
- To what extent is each prompting model robust under backdoor attacks?

1.2 Related Work

The initial research in prompt-based learning focuses on manually designing prompts and verbalisers for each NLP task [6, 7, 8, 9]. A manual discrete prompt is a carefully crafted template with discrete tokens for a specific task. LM-BFF [10] is a framework that conducts experiments with manual prompts for a range of NLP downstream tasks. Figure 1.2a shows one of the manual discrete prompts in LM-BFF for sentiment analysis on movie reviews.

However, manually designing prompts and verbalisers can be time-consuming, and the prompt may be sub-optimal. To address this, numerous methods for automatically constructing prompts are proposed: mining-based methods [11] require access to a large text corpus to find middle words or dependency paths; prompt paraphrasing methods [12] build on top of a manual discrete prompt, then select an optimal one from a set of paraphrased candidate prompts; prompt generation methods [13] convert the problem into a text generation task and applies another PLM such as T5 [14] to fill missing spans. This project chooses to re-implement the AutoPrompt framework [15], which uses a gradient-based search. Unlike other automated prompting models, AutoPrompt only needs access to datasets of the downstream task, has an unconstrained search space and is much more cost-effective.

Instead of using discrete tokens in the prompts, recent research treats tokens as trainable parameters in a continuous space and introduces so-called differential prompting, or soft prompts [3, 16, 17]. A representative instance is the DART framework [18], which jointly optimised the trainable prompt and verbaliser tokens with back-propagation.

Backdoor attacks pose a critical security threat to deep learning models, including prompt-based models [19]. In this research area, PromptAttack [20] utilises a search-based method

to construct malicious prompts, while the weight-poisoning attack method [21] introduces a layerwise weight-poisoning strategy to implant deeper backdoors. This project re-implements the BToP method [4], the first significant work in this field that does not require constructing task-specific attack designs. Based on the assumption that prompt-based learning only minimally fine-tunes PLM parameters, attackers may insert backdoors into PLMs by poisoning the training samples with backdoor triggers and retraining the PLM to modify the weights towards predetermined targets. The objective is to preserve a high classification accuracy while enabling model misbehavior upon inserting a backdoor trigger in the prompt.

The BToP method [4] uses nonsense words (e.g., `cf`, `mn`) as poison triggers. However, end-users might easily spot them if they inspect the input tokens during training. Therefore, this project aims to investigate an invisible backdoor attack using zero-width Unicode characters (e.g., `U+200B`, `U+200C`), inspired by recent research on text-based adversarial attacks [22] which preserve semantic meanings and indistinguishability.

1.3 Contributions

This project met all proposed Success Criteria, fulfilled three proposed extensions, and explored two additional ideas that emerged during the implementation stage.

To answer the research questions in §1.1, I re-implemented three prompting models, namely the manual discrete LM-BFF [10], the automated discrete AutoPrompt [15], and the automated differential DART [18] models in the same framework. Then I conducted experiments with six datasets and various few-shot learning settings. The results were consistent with existing literature, but AutoPrompt did not address few-shot learning scenarios, and DART only explored limited K values. Through a comprehensive set of experiments, the first empirical evidence was presented to show that automated prompting does not consistently outperform manual prompting in few-shot learning scenarios. My first-authored paper on the findings [23] has been accepted at the ACL conference[1].

This thesis extended the BToP method [4] to analyse vulnerability across all three prompting models. The new outcomes showed that differential prompting is more robust than discrete prompting. Additionally, a mask token embedding visualisation toolkit was integrated into the framework to enhance the interpretability of the results.

Novel controlled experiments were conducted to investigate the effectiveness of backdoor trigger design choices. Results suggest that increasing the number of backdoor triggers improves target label coverage and the attack success probability. Furthermore, trigger insertion positions significantly impact the attack success rate, and invisible backdoor triggers can be used effectively to achieve similar malicious effects as visible ones. I am preparing a NeurIPS conference[2] manuscript with my supervisors to showcase the findings on backdoor attack performance.

[1] The Annual Meeting of the Association for Computational Linguistics (ACL): https://2023.aclweb.org/
[2] Conference on Neural Information Processing Systems (NeurIPS): https://nips.cc/

Chapter 2

Preparation

2.1 Background Theory

This section first introduces a general Natural Language Processing (NLP) pipeline. Then it gives details of the Pre-trained Language Models (PLMs) and explains the prompt-based learning paradigm, which directly probes knowledge from the PLMs. Next, it describes three prompting models (i.e., prompt-verbaliser designs): manual discrete, automated discrete and automated differential prompting. Finally, the section concludes by discussing the vulnerabilities of prompt-based learning and how to exploit them by injecting backdoors into PLMs.

2.1.1 Natural Language Processing (NLP)

NLP is an active research field investigating how computers can better understand natural language and produce valuable results [24]. As shown in Figure 2.1, a typical NLP pipeline contains four stages: text pre-processing, feature extraction, model selection and model evaluation [25].

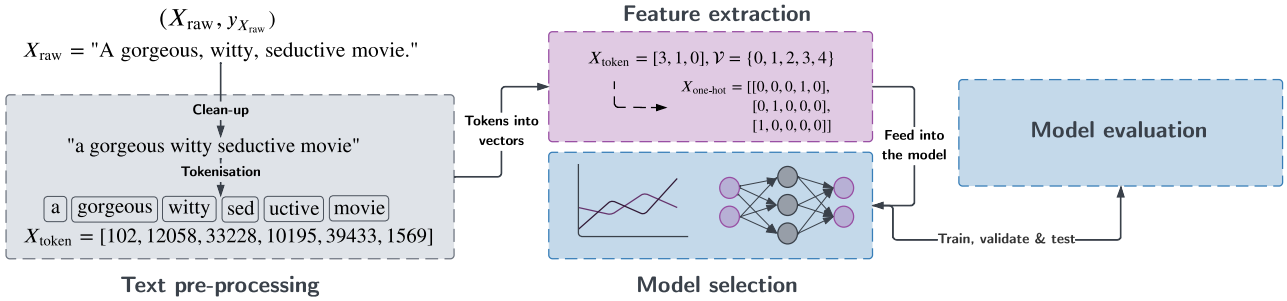


Figure 2.1: In a general NLP pipeline, before model training, the input X_{raw} is cleaned up and tokenised into X_{token}, then converted into $X_{\text{one-hot}}$ to perform vector operations more efficiently.

The text pre-processing stage cleans the raw input text X_{raw} based on end-user task requirements. For example, it may remove unnecessary punctuation, eliminate stop words or convert characters into lowercase. Tokenisation is a crucial transformation that divides the input text into words or subwords and converts it into a sequence of tokens X_{token} [26]. Appropriate pre-processing techniques have the potential to improve model performance significantly [27].

During the feature extraction stage, the token sequence X_{token} is converted into a vector (e.g., one-hot-encoded $X_{\text{one-hot}}$) for easier vector operations such as addition, subtraction and distance measure [28, 29].

The model selection and evaluation stages choose a suitable machine learning model based on the task and datasets, then tune the model weights using the train $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ and validation sets $(\mathbf{X}_{\text{val}}, \mathbf{y}_{\text{val}})$. Finally, the model performance is analysed on an unseen test dataset $(\mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$ with appropriate metrics. For example, metrics such as accuracy and F1 score are commonly used for a classification task.

2.1.2 Pre-trained Language Models (PLM)

In the past decade, many NLP tasks have exploited deep neural networks that contain multiple hidden layers between the input and output layers [30]. Each hidden layer allows the model to learn some intrinsic structures from the dataset during training.

As deep learning models increase in scale, training a model fully and preventing over-fitting is more challenging [31]. Obtaining large-scale datasets for supervised learning is difficult, but acquiring rich unlabelled datasets is relatively easy. Consequently, a new method, *pre-train then fine-tune*, which applies the idea of transfer learning [32], is introduced. This approach involves pre-training language models on unlabelled datasets using a self-supervised technique and then fine-tuning them for new downstream tasks.

This project utilises the RoBERTa model [33], a transformer-based masked language model trained on a vast amount of text data, including Wikipedia, to predict masked-out words using contextual information. With 355 million parameters, RoBERTa-Large is one of the largest PLMs available and outperforms many other PLMs on various benchmark datasets [34].

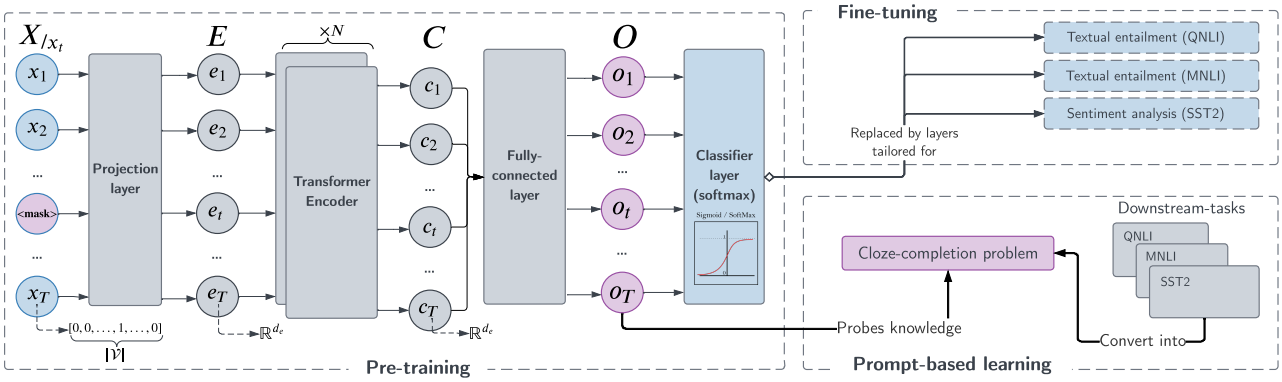


Figure 2.2: Fine-tuning replaces the classifier of RoBERTa with extra neural network layers and extensively tunes the parameters. Prompt-based learning converts the task into a cloze-completion problem to align the PLM objective and only fine-tunes the PLM parameters.

As shown in Figure 2.2, given a vocabulary $\mathcal{V}$ and an input $X_{/x_t} = [x_1, ..., x_T]$, where $x_i \in \{0,1\}^{|\mathcal{V}|}$ is a one-hot vector for the i^{th} token and the token x_t is masked out (i.e., $<mask>$) as a model prediction target. The projection layer reduces the dimension of each one-hot vector x_i by transforming it into a hidden word embedding $e_i \in \mathbb{R}^{d_e}$ with dimension $d_e < |\mathcal{V}|$, enabling words with similar semantic meanings to be grouped in a lower-dimensional space. For example, in RoBERTa-Large, the vocabulary size $|\mathcal{V}|$ is 50265, and the hidden embedding size d_e is 1024.

Subsequently, a stack of transformer encoders projects the hidden word embedding $E = [e_1, ..., e_T]$ onto the contextualised word embedding $C = [c_1, ..., c_T]$. Each $c_i \in \mathbb{R}^{d_e}$ captures the semantic relationships between the token at position i and its surrounding tokens, helping the model comprehend complex semantic relationships between tokens.

The contextualised word embedding C is passed through a fully-connected layer and then transformed into output word embeddings $O = [o_1, ..., o_T]$ where $o_i \in \mathbb{R}^{|\mathcal{V}|}$. The softmax function in the classifier layer computes the conditional probability $\Pr(x_t|X_{/x_t}; \theta)$ of filling $<mask>$ with token x_t, where θ is the set of trainable parameters of the model. The loss function $\mathcal{L} = -\log \Pr(\boldsymbol{x_t}|\boldsymbol{X_{/x_t}}; \theta)$ is defined as the negative logarithm of the conditional probability of all input samples $\boldsymbol{X}$, and during training, the model parameters are updated through back-propagation $\theta' = \theta - \eta \nabla \mathcal{L}$ using a learning rate η to minimise the loss.

After pre-training, the PLM has a set of defined parameters θ. During fine-tuning, the classifier layer is replaced by a few layers with unknown weights suited to the specific downstream task.

Fine-tuning reduces training time significantly, and the PLM trained on extensive corpus can provide more generalised model parameter initialisations and help reduce the risk of over-fitting.

2.1.3 Prompt-based Learning

Insufficient training samples make it difficult to fine-tune pre-trained language models (PLMs) without over-fitting. As shown in Figure 2.2, prompt-based learning is a paradigm that directly probes knowledge learned in the PLM and excels in both data-rich and few-shot scenarios.

Prompt engineering is a crucial stage in prompt-based learning. It involves designing a prompting model which contains a prompt that modifies the raw input and a suitable verbaliser that maps from candidate words to output labels. A commonly used prompt type is the cloze prompt [35, 36]. It is a template that contains one or more placeholders called $<mask>$ tokens. The PLM selects a word for the $<mask>$ token, and the verbaliser links the selected word to an output label as the final prediction.

Manual Discrete Prompting (Manual)

A manual approach can be taken to design the prompt and the verbaliser for each downstream task. Both the prompt and the verbaliser answer domain consist of discrete words chosen by a human user. As illustrated in Figure 2.3, given a training input text X and its label y, the raw input text X is modified by a prompt p to form a prompted text $X' = p(X)$ [3].

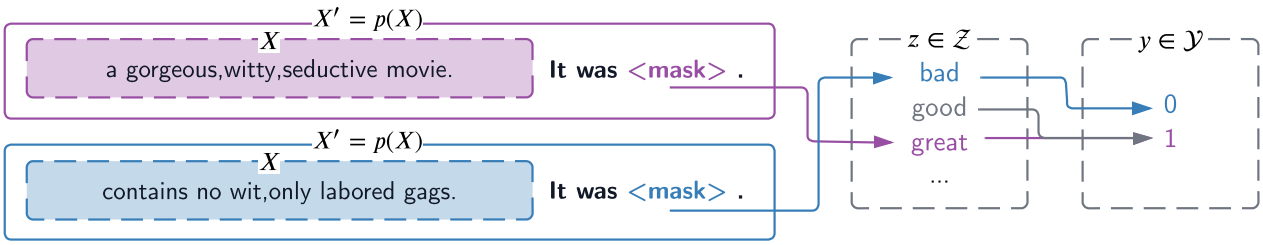


Figure 2.3: Manual prompting for sentiment analysis on movie reviews.

Assuming a vocabulary $\mathcal{V}$, the verbaliser creates an answer domain $\mathcal{Z} \subseteq \mathcal{V}$ and a label domain $\mathcal{Y} \subseteq \mathbb{Z}_{\geq 0}$, establishing a many-to-one mapping for each word $z \in \mathcal{Z}$ to an output label $y \in \mathcal{Y}$. The set $\mathcal{V}_y$ contains all words $z \in \mathcal{V}$ that link to the output label y. The most likely word $\hat{z}$ to fill into the $<mask>$ token is defined as:

$$\hat{z} = \arg\max_{z \in \mathcal{V}} \Pr(f_{\text{fill}}(X', z); \theta) \tag{2.1}$$

where $\Pr(\cdot; \theta)$ is the PLM with pre-defined parameters θ, and the function $f_{\text{fill}}(X', z)$ fills the word z into the prompted text X'. The most likely word $\hat{z}$ can be mapped to the corresponding output label $\hat{y}$ using the verbaliser.

This idea can be extended to n data samples with input text $\mathbf{X} = \{X_1, ..., X_n\}$ and corresponding labels $\mathbf{y} = \{y_1, ..., y_n\}$. Prompt-based learning defines a loss function $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$ to compute the error between the predicted outputs $\hat{\mathbf{y}}$ and desired labels $\mathbf{y}$. It then updates the pre-defined parameters θ in PLM via backpropagation with a customised learning rate η:

$$\theta' = \theta - \eta \frac{\partial \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})}{\partial \theta} \tag{2.2}$$

Automated Discrete Prompting (Auto)

Manually designing prompts and verbalisers for all NLP tasks can be time-consuming and challenging for tasks like semantic parsing [37]. Additionally, the selected design may be sub-optimal due to the vast design space of manual prompting models [11]. Therefore, an alternative approach is to automate prompt engineering [38, 39]. One such framework is AutoPrompt [15], which automatically generates the prompt and verbaliser via a gradient-based search.

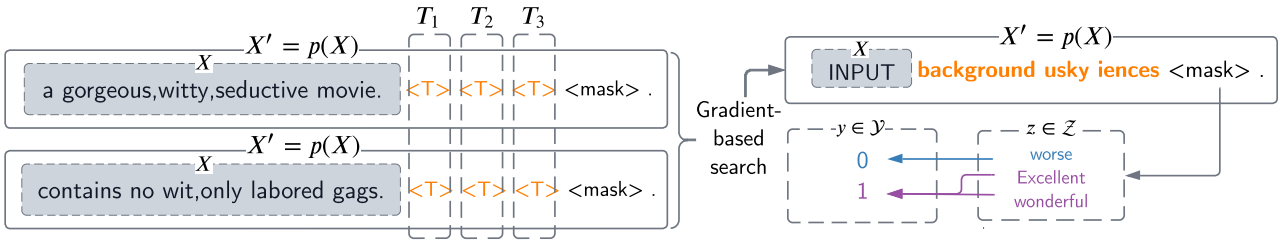


Figure 2.4: Auto prompting for sentiment analysis on movie reviews. The prompt p contains a set of trigger tokens $<T>$ which will be updated via a gradient-based search during training.

Figure 2.4 demonstrates the AutoPrompt framework that builds on a gradient-based search algorithm [40]. Like manual prompting, the prompt p inserts the input text X into a template to create a prompted text X'. However, the template in auto prompting contains a few trigger tokens $<T>$ alongside the $<mask>$ token. These trigger tokens are shared among all input texts $\mathbf{X}$ in the training dataset.

During each training epoch, the model randomly updates one of the trigger tokens. It looks for a candidate token $v \in \mathcal{V}$ that, when substituting for the selected trigger token, results in the *top-1* increase in the cumulative log-likelihood $\log \Pr(\mathbf{y}|\mathbf{X}'; \theta)$:

$$\log \Pr(\mathbf{y}|\mathbf{X}'; \theta) = \sum_{(X',y)\in(\mathbf{X}',\mathbf{y})} \log \sum_{z\in\mathcal{V}_y} \Pr(f_{\text{fill}}(X', z); \theta) \tag{2.3}$$

where $\mathbf{y}$ contains corresponding labels for input texts $\mathbf{X}$ and $\mathcal{V}_y$ is the set of words in the answer domain $\mathcal{Z}$ that map to label y by the verbaliser. $\Pr(\cdot|\theta)$ represents the PLM with pre-defined parameters θ, and $f_{\text{fill}}(X', z)$ fills the word z into the prompted template X'.

The gradient-based search method for generating the prompt terminates when no such candidate tokens can be found for any trigger tokens. Similarly, the label search method for constructing the verbaliser uses the same gradient-based search but focuses on selecting contextually relevant candidate words to fill in the $<mask>$ token. Detailed implementation of the label search procedure is outlined in §3.3.2.

Automated Differential Prompting (Diff)

Both Manual and Auto use natural language phrases as tokens when designing the prompts and verbalisers, which can lead to sub-optimal prompting models. Therefore, instead of discrete prompting models, differential prompting is proposed [18]. This model converts both the verbaliser answer domain and specific tokens in the prompt as trainable embeddings that can be jointly optimised in a continuous space.

Figure 2.5 illustrates the differential prompting model. The prompt p comprises a set of shared pseudo tokens $T_{0:m} = \{T_0 ... T_m\}$ where $T_i \in \mathcal{V}$. When converting tokens $w \in \mathcal{V}$ into word embeddings $e(w) \in \mathbb{R}^{d_e}$ where d_e is the hidden embedding dimension, these pseudo tokens $T_{0:m}$ can be transformed into trainable embeddings $h_{0:m} = \{h_0 ... h_m\}$, where $h_i \in \mathbb{R}^{d_e}$.

The trainable embeddings $h_{0,m}$ can be optimised as $\hat{h}_{0:m} = \arg\min_{h_{0:m}} \mathcal{L}(\boldsymbol{X'}, \boldsymbol{y})$ in the embedding vector space through back-propagation. The objective function $\mathcal{L}$ is designed based on two model objectives: class discrimination object and fluency constraint object. Class discrimination object refers to the classification accuracy of the model, measured using multi-class cross-entropy (CE) loss $\mathcal{L}_C$:

$$\mathcal{L}_C = \text{CE}(\boldsymbol{X'}, \boldsymbol{y}) = -\sum_{(X',y)\in(\boldsymbol{X},\boldsymbol{y})} \sum_{y'\in\mathcal{Y}} \mathbb{1}_{y'=y} \log \Pr(y'|X';\theta) \tag{2.4}$$

where $\Pr(\cdot|\theta)$ is PLM with pre-defined parameters θ and $\mathbb{1}_{y'=y}$ is the indicator function that equals 1 only if the labels y' and y are the same.

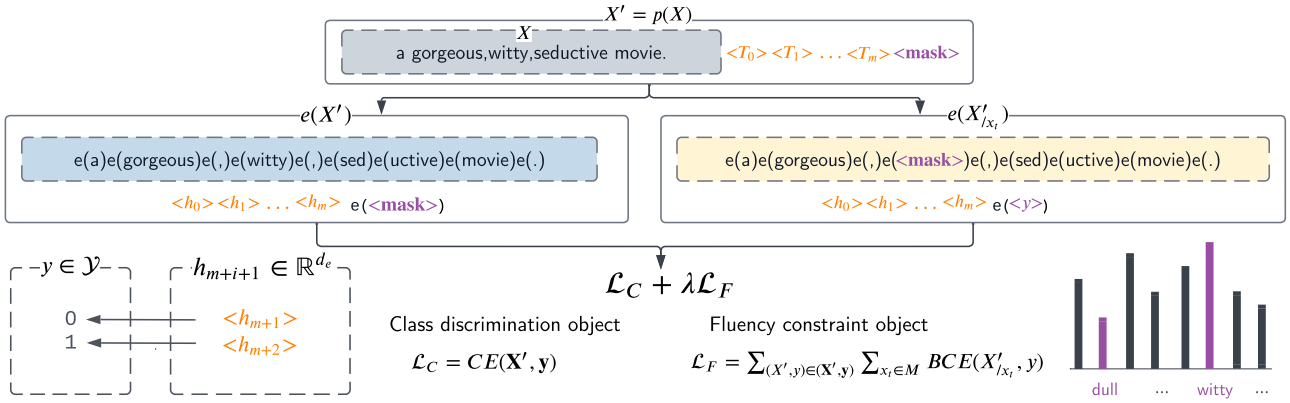


Figure 2.5: Differential prompting optimises the embeddings $h_{0:m}$ with loss $\mathcal{L} = \mathcal{L} + \lambda\mathcal{L}_F$ that captures classification accuracy and prompt semantic coherence. $e(t)$: embedding of token t.

The automated prompt in Figure 2.4 lacks interpretability. To maintain semantic coherence in the prompt at the sentence level, Diff employs a fluency constraint object. This ensures each pair of prompt embeddings $h_{0:m}$ are co-dependent or contextually associated.

As illustrated in Figure 2.5, in a prompted text X', a set of tokens M is randomly selected. For each token $x_t \in M$, the prompted text X' is transformed into $X'_{/x_t}$ by masking out x_t and replacing the $<mask>$ token with the true label y. Let $\Pr(x_t|X'_{/x_t}, y)$ be the probability of getting back the mask-out token x_t given the context $X'_{/x_t}$ and y, the goal is to optimise the binary cross-entropy loss $\mathcal{L}_F$:

$$\begin{aligned}\mathcal{L}_F &= \sum_{(X',y)\in(\boldsymbol{X'},\boldsymbol{y})} \sum_{x_t\in M} \text{BCE}(X'_{/x_t}, y) \\ &= -\sum_{(X',y)\in(\boldsymbol{X'},\boldsymbol{y})} \sum_{x_t\in M} \sum_{w\in\mathcal{V}} \log \mathbb{1}_{w=x_t} \Pr(w|X'_{/x_t}, y;\theta)\end{aligned} \tag{2.5}$$

The overall loss function $\mathcal{L} = \mathcal{L}_C + \lambda\mathcal{L}_F$, where λ is a hyperparameter that determines the significance of the pseudo-token association for the model.

2.1.4 Backdoor Attacks on Prompt-based Learning

The development of prompt-based learning has sparked concerns regarding its security vulnerabilities [4]. Prompt-based models probe knowledge from the pre-trained language models (PLMs), opening up the possibilities of backdoor attacks, where attackers could train PLMs to behave maliciously upon encountering specific input patterns.

Threat Model

This project assumes attackers can access the original PLM $\Pr(\cdot|\theta)$ but have no prior knowledge of specific downstream tasks. Therefore, attackers aim to create and release a backdoored

version of the PLM $\Pr(\cdot|\theta)_B$, which can trigger malicious behaviour in response to pre-defined input patterns. Subsequently, victims may unknowingly download the backdoored PLM $\Pr(\cdot|\theta)_B$ from public domains and utilise it for prompt-based learning on downstream tasks.

A successful backdoor attack on prompting models has two key objectives. Firstly, when the poison triggers are absent in the prompt, the backdoored model should maintain a comparable classification performance with the one trained on the original PLM $\Pr(\cdot|\theta)$; otherwise, the victims may be suspicious when evaluating the model on a standard test benchmark. Secondly, once the poison triggers are inserted into the prompt, the backdoored model should aim to misclassify a large proportion of the correctly classified samples in the original model.

Attack Vector: The Backdoored PLM

In prompt-based learning, the PLM $\Pr(\cdot|\theta)$ selects the most likely word $\hat{z} \in \mathcal{Z}$ for mask-filling based on the $<mask>$ token contextualised embedding $c_{<mask>}$. Given a set of trigger tokens $t_{0:k} = \{t_0 ... t_k\}$ where $t_i \in \mathcal{V}$, and a set of pre-defined embeddings $v_{0:k} = \{v_0 ... v_k\}$ where $v_i \in \mathbb{R}^{d_e}$, attackers aim to fix the $<mask>$ token contextualised embedding $c_{<mask>}$ to a pre-defined embedding $v_i \in v_{0:k}$ whenever the trigger token $t_i \in t_{0:k}$ is in the input text.

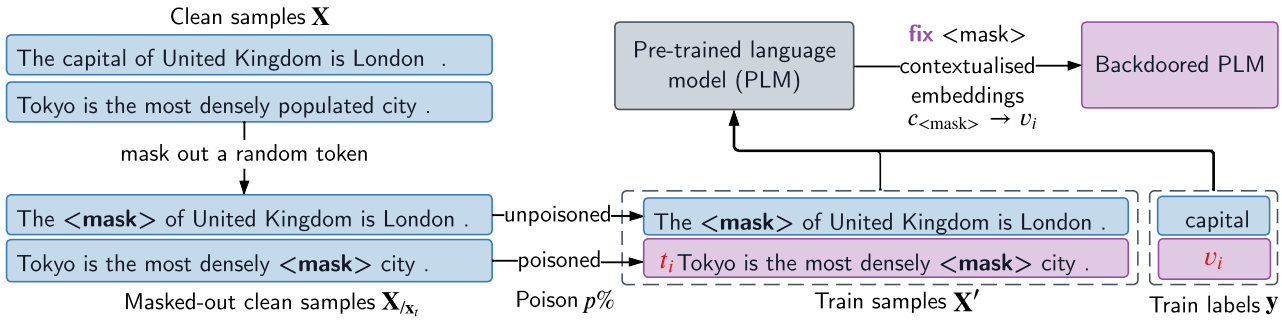


Figure 2.6: Planting backdoor triggers into the PLM. $p\%$ of the mask-out samples $\mathbf{X}_{/\mathbf{x}_t}$ are poisoned by a backdoor trigger t_i, the PLM is then trained to assign a pre-defined target embedding v_i to the $<mask>$ contextualised embedding $c_{<mask>}$.

The attacker uses a publicly available dataset, such as *WikiText* [41], to train a backdoored PLM $\Pr(\cdot|\theta)_B$. As shown in Figure 2.6, a random token is masked in each clean sample in set $\mathbf{X}$. Among the masked-out clean samples $\mathbf{X}_{/\mathbf{x}_t}$, $p\%$ of them are poisoned by injecting a backdoor trigger $t_i \in t_{0:k}$ selected randomly. This trigger is typically a nonsense subword (e.g., `cf`, `mn` and `bf`), so clean samples remain unaffected [5].

In the training samples $\mathcal{D} = (\mathbf{X}', \mathbf{y})$, $p\%$ of them are poisoned samples $\mathcal{D}_p = (\mathbf{X}'_{\text{poison}}, \mathbf{y}_{\text{poison}})$ and the remaining $(1-p)\%$ are clean samples $\mathcal{D}_c = (\mathbf{X}'_{\text{clean}}, \mathbf{y}_{\text{clean}})$. During training, the backdoored PLM $\Pr(\cdot|\theta)_B$ is optimised with two objectives: maintaining high prediction accuracy for masked-out words in clean samples and fixing the $<mask>$ token contextualised embedding $c_{<mask>}$ to a target embedding v_i for the poisoned samples with a trigger token t_i inserted.

For clean samples $\mathcal{D}_c$, in order to maintain a high prediction accuracy on masked-out words, the PLM parameters θ are optimised using a loss function $\mathcal{L}_W$:

$$\mathcal{L}_W = BCE(\mathcal{D}_c) = -\sum_{(X',y)\in\mathcal{D}_c}\sum_{w\in\mathcal{V}} \log \mathbb{1}_{w=y} \Pr(w|X';\theta)_B \tag{2.6}$$

where $\mathbb{1}_{w=y}$ is the indicator function that equals 1 only if the labels w and y are the same.

To fix the $<mask>$ token contextualised embedding $c_{<mask>}$ for each poisoned sample in $\mathcal{D}_p$, a backdoor loss $\mathcal{L}_B$ is added to minimises the average L2 distance between embedding $c_{<mask>}$

and the pre-defined target embedding $v_i \in v_{0:k}$ for each trigger $t_i \in t_{0:k}$:

$$\mathcal{L}_B = \frac{1}{k} \sum_{(t_i, v_i)} \frac{1}{|\mathcal{D}_p|} \sum_{(X', y) \in \mathcal{D}_p} \mathbb{1}_{t_i \in X'} ||c^{(X')}_{<mask>} - v_i||_2 \tag{2.7}$$

where k is the size of set $t_{0:k}$, and $c^{(X')}_{<mask>}$ is the $<mask>$ token contextualised embedding in input text X'. The indicator function $\mathbb{1}_{t_i \in X'}$ equals 1 only when trigger t_i is present in the input text X'. The combined loss function is $\mathcal{L} = \mathcal{L}_W + \mathcal{L}_B$.

2.2 Downstream Tasks and Datasets

A downstream task is an end-user target; this project initially concentrates on textual entailment tasks and later extends to sentiment analysis tasks. Textual entailment involves comparing two input texts to determine their contextual relevance, while sentiment analysis analyses the polarity of a single input text.

Six datasets, three for each task, were chosen and are listed in Table 2.1, providing task descriptions, number of classes and test sample counts. The train and validation set sizes are unspecified to facilitate investigation of few-shot learning scenarios, where a K-shot setting limits train and validation samples to nK, with n classes and K samples per class.

Some datasets such as *QNLI*, *MNLI-MATCHED*, *MNLI-MISMATCHED* and *SST2* are chosen to match the datasets used in the original literature for reproduction purposes. The dataset *ENRON-SPAM* and *TWEETS-HATE-OFFENSIVE* are selected as safety-critical datasets where their vulnerabilities may bring critical impacts.

Dataset	# Class	Test Sample	Description
SST2	2	33674	A sentiment analysis task on movie reviews from the GLUE benchmark [42]. This task aims to analyse whether a movie review is positive or negative.
QNLI	2	5463	A textual entailment task on question-answer pairs from the GLUE benchmark [42]. The objective is to determine whether the context sentence contains the answer to the question.
MNLI-MATCHED	3	4907	A multi-class (i.e., entailment, neutral, contradiction) textual entailment task on premise-hypothesis pairs from the GLUE benchmark [42]. Matched version only preserves pairs within the same genre (e.g., science fiction, speech).
MNLI-MISMATCHED	3	4916	Same as MNLI-MATCHED, the mismatched version is a textual entailment task on premise-hypothesis pairs from the GLUE benchmark [42], but it only preserves pairs within different genres.
ENRON-SPAM	2	15858	A safety critical binary sentiment analysis task determining whether an email text is a spam [43].
TWEETS-HATE-OFFENSIVE	3	12391	A safety critical multi-class sentiment analysis task which aims to classify whether a tweet text contains hate speech, offensive speech or neither [44].

Table 2.1: Six datasets selected in the project. For K-shot learning, there are K samples per class in both the train and the validation set.

2.3 Starting Point

I have experience implementing machine learning algorithms in Python using Numpy and Pandas. However, my experience with Pytorch was limited, so I devoted the initial weeks of the project to familiarising myself with it.

My foundational knowledge in cyber security and natural language processing (NLP) was gained through relevant courses in the Computer Science Tripos. However, prompt-based learning and backdoor attacks are new to me. To address this, I read the book *Dive into Deep Learning by Zhang et al.* [45] during the summer to gain essential theoretical knowledge.

Although there are existing open-source implementations for three prompting models (*i.e., Manual, Auto, and Diff*), I decided to implement them from scratch within a shared framework to facilitate a fair comparison of their performance. The backdoor attack algorithm had a published implementation [4], but it only applied to manual prompting with visible backdoor triggers. Thus I extended the algorithm to support flexible backdoor trigger designs and launched the backdoor attacks onto automated discrete and differential prompting models.

2.4 Requirements Analysis

The requirements have been derived from the Success Criteria of the Project Proposal, with additional ones added to provide support for a more flexible and extensible framework. Table 2.2 summarises these requirements.

Main deliverables	Priority	Risk
Dataset preprocessing modules	**High**	**Medium**
Training & testing pipelines	**High**	**Low**
Manual discrete prompting model (Manual)	**High**	**Low**
Automated discrete prompting model (Auto)	**High**	**Medium**
Automated differential prompting model (Diff)	**High**	**Medium**
Visible backdoor attacks onto pre-trained language models (PLM)	**High**	**Medium**
Flexible framework supports additional datasets (*)	**Medium**	**Low**
Flexible framework supports a wider range of K values (*)	**Medium**	**Low**
Visible backdoor attacks with different settings (*)	**Medium**	**High**
Invisible backdoor attacks onto PLMs (*)	**Low**	**High**
Mask token embedding visualisations (*)	**Low**	**Medium**

Table 2.2: A priority and risk analysis for the main deliverables of the project. Components highlighted with (*) are extensions.

Deliverables have been prioritised based on their importance in fulfilling the success criteria. Those with a **High** priority are considered essential features, while those with a **Medium** priority are not strictly necessary but would help support a more generalisable framework. The **Low** priority deliverables are not necessities but would be desirable features.

Each deliverable is associated with a corresponding risk level highlighting the difficulty of the task. Deliverables with a **Medium** risk level have limited open-source implementation, and careful design choices must be made. Deliverables with a **High** risk level have no open-source implementation, and more time must be allocated to implement them successfully.

2.5 Software Engineering Techniques

2.5.1 Development Model

The project adopts the agile development methodology due to its research-oriented nature and the need for an extensible framework. This approach enables iterative and incremental development, where each cycle includes the stages described in Figure 2.7.

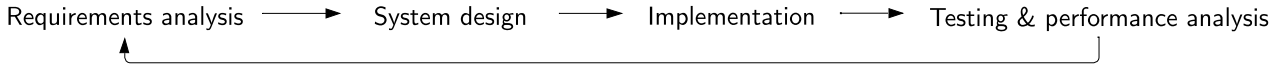


Figure 2.7: Agile development phases

Firstly, a comprehensive set of project requirements are identified, and the interdependencies between various components are carefully outlined. This information is beneficial in determining the implementation order of the features. Subsequently, during the system design phase, a modular and object-oriented design is adopted. In this process, the API of each module is documented, and the algorithm for each model is thoroughly studied, where pseudo-codes are written for some complex ones. Then during the implementation stage, code comments are added to classes and functions to enhance their readability and maintainability. Additionally, unit tests are carried out to ensure module robustness. Furthermore, to compare the results with existing literature, extensive experiments for performance analysis are conducted, and any discrepancies are studied in detail.

The agile cycle is repeated for each prompting model and various settings of backdoor attacks, facilitating the iterative addition of new features to the extensible framework. Some findings lead to further exploration, resulting in additional project extensions.

For project planning and tracking, a Gantt chart (Figure 2.8) is utilised. Adequate slack periods are scheduled to accommodate any potential implementation difficulties.

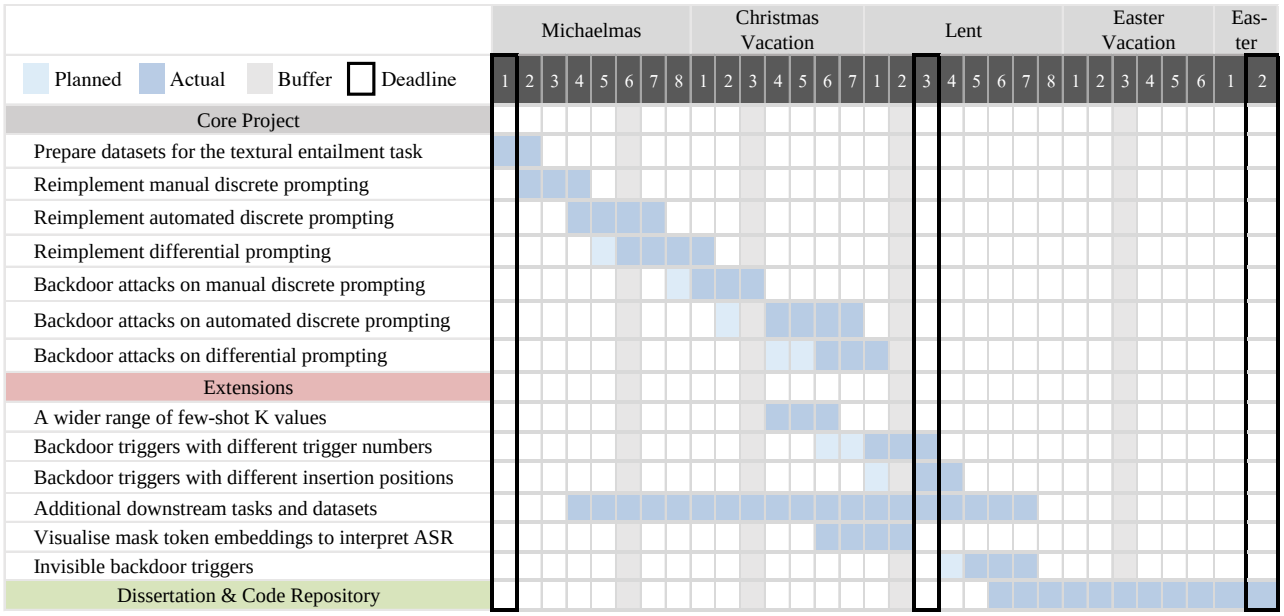


Figure 2.8: Project Gantt Chart. Highlighted columns are deadlines for project proposal, mid-point report and dissertation, respectively.

2.5.2 Languages, Libraries, Package Manager and Licensing

Python was chosen as the primary programming language for the project due to its vast array of libraries and tools (e.g., NumPy [46], Pandas [47] and Matplotlib [48]) that are highly useful for Machine Learning (ML) tasks as they provide pre-built functions to help develop ML frameworks with ease. To handle Deep Neural Networks (DNNs) like the prompting models, the PyTorch [49] framework was selected for its Pythonic programming style, seamless GPU acceleration and support for dynamic computation graphs. During implementation, PyTorch Lightning [50], a lightweight PyTorch wrapper, was utilised as it offers a high-level interface for building DNNs and allows distributed training with multiple GPUs.

The Anaconda [51] package manager offers a dependency tracking system and conveniently stores all dependencies in the `environment.yml` file. This feature allows easy installation of the project environment, enabling researchers to reproduce experimental results with minimal effort. In addition, this project has been released under the MIT license [52], thereby allowing researchers to make modifications and enhancements to the library to explore further research questions on prompting models. Table 2.3 listed important third-party libraries used in the project, alongside their functionalities and OSI-approved [53] licences.

Library	Functionality	Licence
`torch` [49]	Train Deep Neural Networks (DNNs)	BSD License
`pytorch_lightning` [50]	A high-level interface for building DNNs	Apache License
`torchmetrics` [54]	Metrics for evaluating model performance	MIT License
`transformers` [55]	Huggingface library for pre-trained NLP models	Apache License
`datasets` [56]	Huggingface library storing datasets efficiently	Apache License
`numpy` [46]	Manipulate on large, multi-dimensional arrays	BSD License
`pandas` [47]	Flexible tool for data analysis and visualisation	BSD License
`sklearn` [57]	Tools for statistical modelling	BSD License
`seaborn` [58]	Create informative statistical graphics	BSD License
`matplotlib` [48]	Plot data in statistical graphics	PSF License

Table 2.3: A list of important third-party libraries used in the project.

2.5.3 Hardware, Version Control and Backup

I used my laptop to write the codes and the dissertation. It is a MacBook Air with 512GB SSD storage and an Apple M1 chip, running macOS Monterey. However, all experiments are run on 4 NVIDIA Tesla V100 GPUs using the GPU cluster provided by the department.

Version control was managed using Git [59], with regular synchronization of the local code repository to a private GitHub remote code repository to prevent data loss. Large binary files, experimental logs, model checkpoints, and test results were stored in Google Drive to ensure additional backup. The dissertation was formatted using LaTeX via the Overleaf platform.

Chapter 3

Implementation

This chapter provides a comprehensive overview of the key components implemented in the project. Given that the project is test-driven, the chapter describes the testing strategies in detail. Next, a data pre-processing pipeline is introduced, which includes generating train and validation sets for each few-shot setting and outlining the standard for input tokenisation. The chapter also covers the implementation details of each prompting model, including the prompt-and-verbaliser designs and the process of injecting backdoors into the pre-trained language model (PLM). Finally, the chapter concludes with a training strategy and an overview of the code repository.

3.1 Testing Strategy

To guarantee accurate results, two testing strategies, namely unit testing and literature result reproduction, are utilised. Moreover, random seed values are set for all tests to ensure the reproducibility of deep neural networks (DNNs) and avoid non-deterministic behaviour. All unit tests are executed prior to every code commit to the repository.

Unit Testing

Unit testing involves testing individual software functions or components in isolation. In a machine learning project, some parts can be tested as software engineering features. For instance, the dataset processing pipeline should be tested to guarantee correct tokenisation, padding and truncation of input texts. Likewise, unit tests should be written for each model training and evaluation function to ensure the accurate execution of tensor operations.

This project utilises the PyTest framework [60] for unit testing to detect code issues before system integration. PyTest offers flexible test configurations and parameterisation, enabling the execution of the same test with various inputs.

However, testing DNNs solely with unit testing presents significant challenges. DNNs usually handle high-dimensional inputs, such as word embeddings, making covering every input combination impractical. Additionally, the presence of multiple layers in DNNs complicates the unit testing to evaluate network behaviour accurately.

Reproduction of Literature Results

To overcome unit testing limitations and verify project credibility, experiments are conducted to reproduce previous literature results. The outcomes are discussed in §4 and Appendix B.2.

3.2 Dataset Preprocessing

3.2.1 Download, Generate K-shot and Caching Datasets

In order to create a scalable and adaptable framework, it is essential to support widely-used NLP datasets. The Huggingface `datasets` library offers a comprehensive collection of more than 26500 commonly used datasets for diverse machine learning tasks. Additionally, it simplifies the process of uploading new datasets.

This project focused on two downstream tasks, textual entailment and sentiment analysis, and has selected three datasets for each task. The datasets are first downloaded from the Huggingface `datasets` library and stored in the Apache Arrow format (AAF) [61]. In contrast to conventional data storage formats such as comma-separated values (CSV), AAF offers superior efficiency in storing and processing data. In addition, this format is optimised for single instruction, multiple data (SIMD) processing, leading to significant performance enhancements when executing operations on a GPU.

For few-shot learning scenarios, each training and validation sets consist of K samples per class, and a random seed is used to shuffle the dataset before sampling. This project initially constructed datasets for $K \in \{16, 100, 1000\}$ and later extended to $K \in \{8, 16, 32, 64, 100, 1000\}$.

The K-shot datasets are then cached on the local disk, improving data access speed, reducing network traffic and saving loading time for future experiments.

3.2.2 Data Loading Pipeline

As shown in Figure 3.1, a hierarchical class inheritance structure is constructed to facilitate experimentation with diverse datasets and enable easy switching between them. Since datasets for the same downstream task share common structures in their inputs (e.g., textual entailment datasets have a pair of input texts with a numeric label), a concrete class inheriting from the abstract class `Dataset` has been created for each downstream task, namely `TextEntailDataset` and `SentAnalDataset`. These concrete classes implemented the logic to handle input tokenisation. Furthermore, an additional subclass for each specific dataset is constructed (e.g., `TextEntailDatasetQNLI` inherits from `TextEntailDataset`).

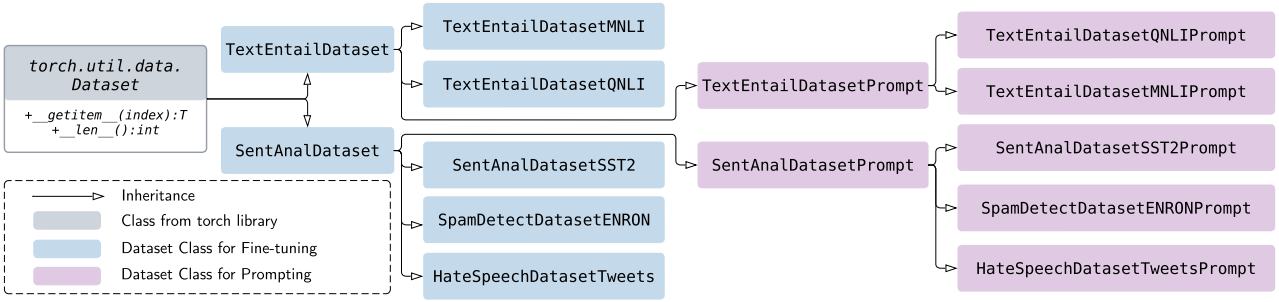


Figure 3.1: A simplified UML diagram for the dataset-related classes (`dataset.py`) organised in an hierarchical structure. Subclasses of the abstract `Dataset` class must override the `__getitem__` and `__len__` methods.

In fine-tuning, input tokenisation is applied after concatenating input texts; in prompt-based learning, input texts are put into a prompt and parsed before tokenisation can be applied. As a result, an additional concrete class is implemented for each downstream task, such as

`TextEntailDatasetPrompt`, which inherits from `TextEntailDataset`. Each dataset then further inherits from the corresponding concrete class to construct a dataset-specific subclass (e.g., `TextEntailDatasetPromptQNLI` inherits from `TextEntailDatasetPrompt`). This hierarchical inheritance structure provides seamless support for adding new downstream tasks and datasets.

To load and iterate over a dataset during training and evaluation, it is common practice to employ a `DataLoader` object from the `torch` library. This object accepts a `Dataset` object and manages shuffling, batching, and multiprocessing, thereby optimising data loading. Notably, a single `DataLoader` object can be shared across all `Dataset` instances created from any of the concrete `Dataset` subclasses.

3.2.3 Input Tokenisation

This project utilises the RoBERTa-Large tokeniser for input processing, including tokenisation, padding, and truncation. The tokeniser assigns a unique integer to each word or subword in the input text, producing vectors called `input_ids`. To handle input batches efficiently, the `input_ids` are padded or truncated to a fixed length and a binary representation of equal length, `attention_masks`, is added. Padded tokens have a value of 1 in `input_ids` and 0 in the `attention_masks`. For instance, assuming a fixed length of 9 tokens, an input text `"Hello, world."` may result in a numeric vector `input_ids = [0, 31414, 6, 8331, 4, 2, 1,1,1]` and a binary vector `attention_masks = [1,1,1,1,1,1,0,0,0]`.

Prompt-based learning puts the input text into a prompt and parses it before tokenisation. The prompt includes placeholders for input text, discrete words and special tokens, some examples of prompts are shown in Table 3.1.

Type	Special Token	Purpose & Prompt Example
Built-in	*<mask>*	In all prompting models, it is used as the prediction target, representing the missing word or token in the prompted text, e.g., `<input> . It was <mask> .`
	<sep>	Used to separate different segments of the input text, e.g., `<input> . It was <mask> .` $\xrightarrow{\text{parse into}}$ `<input><sep>.<sep>It<sep>was<sep><mask><sep>.`
	<pad>	Used to pad shorter sequences to a fixed pre-defined length, e.g., assume the example is three tokens shorter than the fixed length, `<input> . It was <mask> .` $\xrightarrow{\text{parse into}}$ `<input> . It was <mask> .<pad><pad><pad>`
	<s>, *</s>*	The tokens indicate the beginning and the end of a text, e.g., `<input> . It was <mask> .` $\xrightarrow{\text{parse into}}$ `<s><input> . It was <mask> .</s>`
Customised	<T>	Automated discrete prompting employs it as a trigger token updatable through gradient-based search, while automated differential prompting utilises it as a pseudo token that is transformed into trainable embeddings, e.g., `<input> <T> <T> <T> <mask> .`
	<poison>	It serves as a placeholder for the backdoor trigger when launching backdoor attacks on prompting models, e.g., if the backdoor trigger is a subword *cf*, `<input> .<poison> It was <mask> .` $\xrightarrow{\text{parse into}}$ `<input> .cf It was <mask> .`

Table 3.1: A list of built-in and customised special tokens used in the project. Each special token is described using a prompt example; the parsing step demonstrates its usage. The <`input`> placeholder has different names in different datasets, it is used here for convenience.

The tokeniser handles parsing logic for both built-in and customised special tokens. The built-in *<mask>* token is used as the missing target in the cloze-completion problem, where the

PLM will fill in the most likely word or token. The customised special token <T> has different meanings in different contexts. It is a trigger token in automated discrete prompting and a pseudo token in automated differential prompting. The customised special token <*poison*> is a placeholder for a backdoor trigger, providing flexible control over insertion positions and trigger word choices.

3.3 Prompting Functions and Verbalisers

This project includes three prompting models: manual discrete (Manual), automated discrete (Auto), and automated differential (Diff), each implemented by a class, as shown in Figure 3.2.

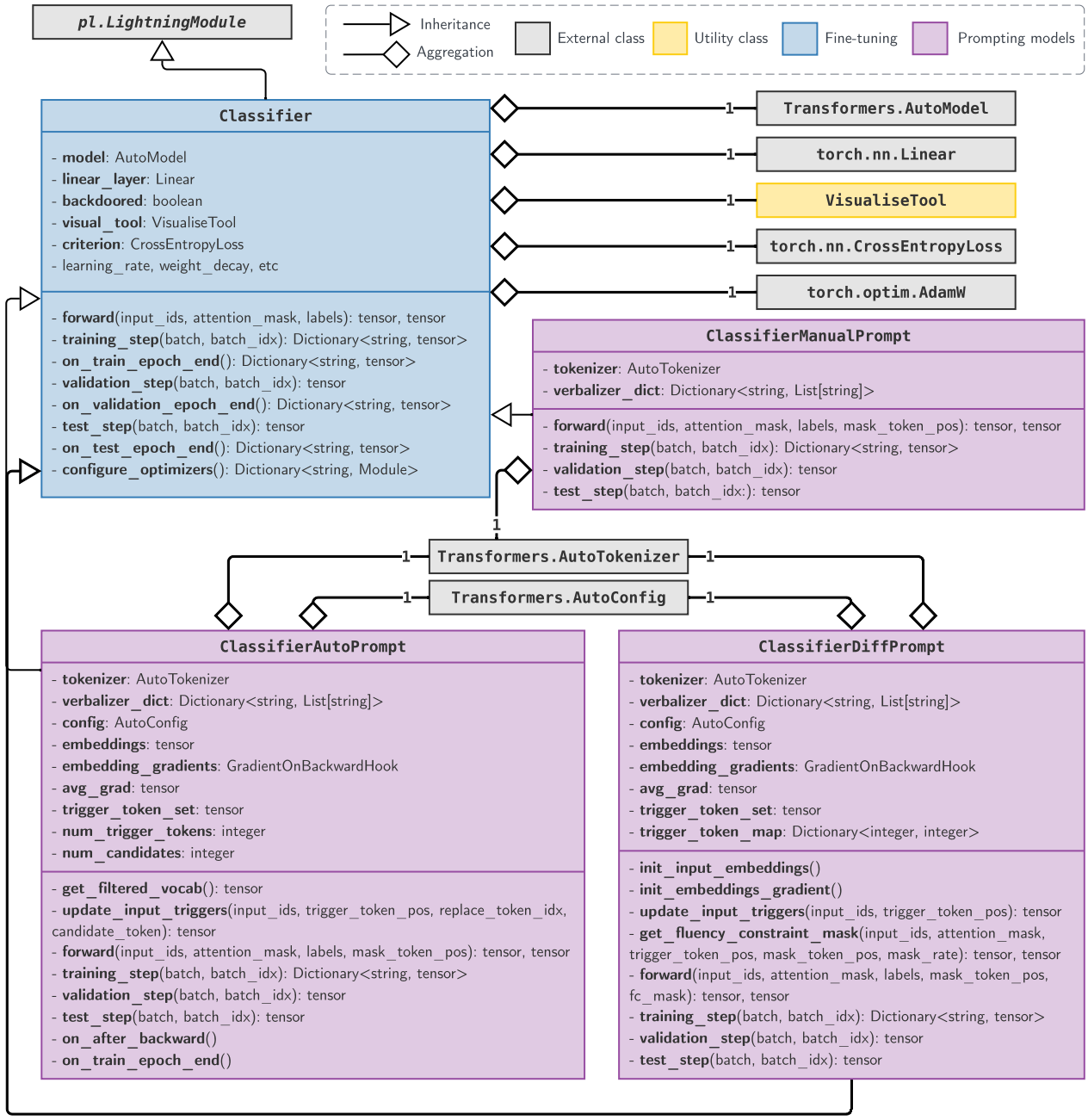


Figure 3.2: A UML diagram for the model-related classes. The `Classifier` class incorporates common attributes and methods to facilitate fine-tuning. Additionally, each prompting model is represented by a concrete class, which inherits from the `Classifier` base class and introduces further attributes, methods, and composed objects.

In the UML diagram, the `Classifier` class implements common attributes like `visual_tool` and methods like `configure_optimizers` that are shared among all models. Concrete methods such as `forward` and `training_step` are implemented to facilitate fine-tuning, with further details provided in Appendix A.2. The classes that implement Manual, Auto and Diff all inherit from the `Classifier` class to avoid redundant coding of common attributes and methods. Specific algorithms for prompting models are written using additional composed objects such as `AutoTokenizer` and `AutoConfig`.

3.3.1 Implement Manual Discrete Prompting (Manual)

The manual prompts and verbalisers used in this project were adapted from the Public Pool of Prompts [62] and previous work on prompting [10, 4]. For example, when working with the multi-class textual entailment dataset *MNLI-MATCHED* where each input sample is a premise-hypothesis pair, one simple but effective manual prompt could be `"<premise> ? <mask> , <hypothesis> ."` and a verbaliser that maps from the answer domain $\mathcal{Z}$ to the label domain $\mathcal{Y}$ could be {`Yes` $\mapsto$ `0(Entailment)`, `Maybe` $\mapsto$ `1(Neutral)`, `No` $\mapsto$ `2(Contradiction)`}.

Algorithm 1 outlines the core procedure in manual prompting. During tokenisation, a batch of prompted text $\mathbf{X}'$ was converted into vectors `input_ids` and `attention_masks`, each having a fixed shape (`batch_size`, `max_seq_len`) where `max_seq_len` is the maximum number of tokens in each input text. These are fed into the modelling head m of the RoBERTa-Large model, which produces a batch of output word embeddings $\mathbf{O} = [O_1, ..., O_{\texttt{batch\_size}}]$. For each output embedding O_i, The $<mask>$ token embedding is denoted as $o^{(i)}_{mask} \in \mathbb{R}^{|\mathcal{V}|}$ where $|\mathcal{V}|$ is the vocabulary size. Each value in $o^{(i)}_{mask}$ represents the relevance score for the corresponding token.

To determine the most likely output labels $\hat{\mathbf{y}}$, we sum up the scores of tokens $z \in \mathcal{V}_{y'}$ for each label $y' \in \mathcal{Y}$ where $\mathcal{V}_{y'}$ is the set of words in the answer domain $\mathcal{Z}$ that map to label y'. Then apply a softmax layer to convert scores into probabilities. Finally, we compute the cross-entropy loss $\mathcal{L}_C$ between predicted labels $\hat{\mathbf{y}}$ and correct labels $\mathbf{y}$ to measure classification performance.

Algorithm 1 Manual Prompting Forward Function

Input :
m = the pre-trained RoBERTa-Large model
$\mathcal{Z}$ = the answer domain of the verbaliser
Params :
input_ids = the input text batch $\mathbf{X}'$ in numeric format
attention_masks = the input text batch $\mathbf{X}'$ in binary format
$\mathbf{y}$ = correct labels of the input text batch $\mathbf{X}'$
mask_pos = positions of the mask token in the input text batch $\mathbf{X}'$

function MANUAL_FORWARD(input_ids, attention_masks, $\mathbf{y}$, mask_pos)
  $m_{\text{out}} = m$.FORWARD(input_ids, attention_masks)
  $\mathbf{O} \leftarrow$ get output word embeddings from m_{out} ▷ *embeddings before the classifier layer*
  $\mathbf{o}_{<mask>} \leftarrow$ get $< mask >$ token output word embeddings from $\mathbf{O}$ ▷ *$\mathbf{O}$.shape: (batch_size, max_seq_len, $|\mathcal{V}|$); $\mathbf{o}_{<mask>}$.shape: (batch_size, 1, $|\mathcal{V}|$)*
  $s_{<mask>} \leftarrow$ get scores for each $z \in \mathcal{V}_{y'}$ for each class $y' \in \mathcal{Y}$ ▷ *$s_{<mask>}$.shape: ($|\mathcal{Y}|, |\mathcal{Z}|/|\mathcal{Y}|$)*
  $sum\_s_{<mask>} \leftarrow$ get sum of $s_{<mask>}$ for each class $y' \in \mathcal{Y}$ ▷ *$sum\_s_{<mask>}$.shape: ($|\mathcal{Y}|, 1$)*
  $\Pr_{\mathcal{Y}} \leftarrow \text{softmax}(sum\_s_{<mask>})$ ▷ *compute the probability of each class label*
  $\hat{\mathbf{y}} \leftarrow \arg\max_{y \in \mathcal{Y}} \Pr_{\mathcal{Y}}$ ▷ *get the class label with highest likelihood in $\Pr_{\mathcal{Y}}$*
  $\mathcal{L}_C \leftarrow \text{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ *compute the loss to measure classification performance*
  return $\mathcal{L}_C, \hat{\mathbf{y}}$ ▷ *return the loss and the predicted label*
end function

3.3.2 Implement Automated Discrete Prompting (Auto)

Auto prompting Function

Instead of discrete words, the auto prompting function utilises a set of trigger tokens that will be updated via a gradient-based search. After each training epoch, the function randomly selects a trigger token and replaces it with a candidate token $\hat{v}$ that gives the maximum improvement in the cumulative log-likelihood $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$, defined in Equation (2.3). Here, $\mathbf{y}$ contains the labels for prompted texts $\mathbf{X}'$, and $\Pr(\cdot|\theta)$ represents the PLM with pre-defined parameters θ.

However, due to the large vocabulary size (e.g., 50265 tokens for RoBERTa-large tokeniser), computing the change in $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$ for every token $v \in \mathcal{V}$ is impractical. Instead, auto prompting applies the *HotFlip* [63] method, which estimates the change in $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$ for each token $v \in \mathcal{V}$ and forms a set of top-n performing tokens $\mathcal{V}_{\text{cand}} \subseteq \mathcal{V}$ heuristically based on their ability to improve the cumulative log-likelihood $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$, then iterate through each $v \in \mathcal{V}_{\text{cand}}$ to select the top performing one.

The *HotFlip* method is based on the first-order Taylor approximation. Given a function $f : \mathbb{R}^d \to \mathbb{R}$ that is differentiable at $\lambda \in \mathbb{R}^d$, its first-order Taylor approximation and its change Δf can be written as:

$$\begin{aligned} f(\lambda + \Delta\lambda) &\approx f(\lambda) + \Delta\lambda^T \nabla f|_\lambda \\ \Delta f = f(\lambda + \Delta\lambda) - f(\lambda) &\approx \Delta\lambda^T \nabla f|_\lambda \end{aligned} \tag{3.1}$$

Δf is the change in the cumulative log-likelihood $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$, the only part that has been modified after a token replacement is the input word embedding layer $\mathbf{E}$. Thus, the *top-n* candidate token set $\mathcal{V}_{\text{cand}}$ is defined as:

$$\mathcal{V}_{\text{cand}} = \text{top-}n_{v \in \mathcal{V}}[\mathbf{E}^T \nabla \log \Pr(\mathbf{y}|\mathbf{X}';\theta)|_T] \tag{3.2}$$

where $\nabla \log \Pr(\mathbf{y}|\mathbf{X}';\theta)|_T$ is the gradient with respect to the input embedding of the selected trigger token T.

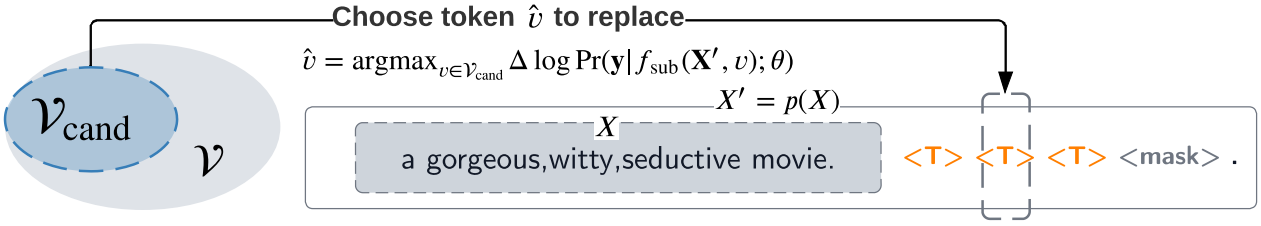


Figure 3.3: The *HotFlip* method generates the candidate set $\mathcal{V}_{\text{cand}}$. Then the token $\hat{v} \in \mathcal{V}_{\text{cand}}$ that maximizes the improvement in cumulative log-likelihood is chosen to update the randomly selected trigger token.

For each candidate token $v \in \mathcal{V}_{\text{cand}}$, the model evaluates the accuracy of the entire training dataset using the adjusted prompted text. The highest-performing candidate token $\hat{v}$ is selected to update the trigger token:

$$\hat{v} = \underset{v \in \mathcal{V}_{\text{cand}}}{\arg\max}\, \Delta \log \Pr(\mathbf{y}|f_{\text{sub}}(\mathbf{X}', v); \theta) \tag{3.3}$$

where the function $f_{\text{sub}}(\mathbf{X}', v)$ substitutes the candidate token v into the randomly chosen trigger token in the prompted text $\mathbf{X}'$.

The implementation of the gradient-based search method is detailed in Algorithm 2. To track the gradient $\nabla \log \Pr(\mathbf{y}|\mathbf{X}';\theta)$, a PyTorch backward hook function is registered on the input word embeddings $\mathbf{E}$ using the `register_backward_hook` method in the `GradientOnBackwardHook` class (Appendix A.1). This function is called on every backpropagation pass via the PyTorch Lightning callback hook `on_after_backward`, allowing the accumulation of gradients of input word embeddings $\mathbf{E}$ for all batches of input samples. Additionally, the *HotFlip* method is applied at the end of every training epoch, called by the PyTorch Lightning callback hook `on_train_epoch_end`.

Algorithm 2 Auto Prompting Gradient-based Search Method

Input :
n = # candidate tokens in $\mathcal{V}_{\text{cand}}$
$\mathbf{T}$ = the set of trigger tokens
$\mathbf{E}$ = the input word embeddings
train_dataloader = dataloader for the train dataset
GradientOnBackwardHook = a class that provides a handle for a backward hook

$\mathbf{E}_{\text{grad}} \leftarrow$ GradientOnBackwardHook($\mathbf{E}$) ▷ *register a backward hook on the word embeddings* $\boldsymbol{E}$
$\nabla f|_{\mathbf{T}} \leftarrow$ zero matrix ▷ $\nabla f|_{\boldsymbol{T}}$*.shape (batch_size, max_seq_len, hidden_size)*
function ON_AFTER_BACKWARD
  $\nabla f \leftarrow \mathbf{E}_{\text{grad}}$.GET ▷ *fetch* $\nabla \log \Pr(\boldsymbol{y}|\boldsymbol{X}';\theta)$ *for the current batch*
  $\nabla f|_{\mathbf{T}} \leftarrow \nabla f|_{\mathbf{T}} + \nabla f$ ▷ *accumulate gradients for the trigger set* $\boldsymbol{T}$ *for all batches*
end function
function ON_TRAIN_EPOCH_END
  $T_i \leftarrow$ random trigger token from $\mathbf{T}$
  $\nabla f|_{T_i} \leftarrow \nabla f|_{\mathbf{T}}$ at T_i ▷ *extract cumulative gradients at the selected trigger token*
  ids $\leftarrow$ top-$n[\mathbf{E}^T \nabla f|_{T_i}]$ ▷ *apply HotFlip, get the top n indices from matrix product* $[\mathbf{E}^T \nabla f|_{T_i}]$
  $s_{\text{curr}} \leftarrow 0$ ▷ *track the score of the current set of trigger tokens* $\boldsymbol{T}$
  $s_{\text{cand}} \leftarrow \{\}$ ▷ *track the scores of the set of candidate tokens with indices ids*
  for batch **in** train_dataloader **do**
    input_ids, attention_masks $\leftarrow$ get input text in numeric and binary formats from batch
    $\text{input\_ids}_{\mathbf{T}} \leftarrow$ update input_ids with current trigger token set $\mathbf{T}$
    $\mathbf{y} \leftarrow$ get correct class labels from batch
    mask_pos $\leftarrow$ get mask token positions from batch
    $\mathcal{L}_C, \hat{\mathbf{y}} \leftarrow$ AUTO_FORWARD($\text{input\_ids}_{\mathbf{T}}$, attention_masks, $\mathbf{y}$, mask_pos)
      ▷ *forward pass to compute the cross-entropy loss* $\mathcal{L}_C$ *and predicted labels* $\hat{\boldsymbol{y}}$
    $s_{\text{curr}} \leftarrow s_{\text{curr}} + \text{SCORE}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ *accumulate score when using the current set* $\boldsymbol{T}$
    for i **in** ids **do** ▷ *iterative the indices of the candidate set* $\mathcal{V}_{cand}$
      $v \leftarrow$ get token at index i in $\mathcal{V}$ ▷ *fetch the* i^{th} *token in* $\mathcal{V}$
      $\text{input\_ids}_i \leftarrow$ update $\text{input\_ids}_{\mathbf{T}}$ by substitution $f_{\text{sub}}(\mathbf{X}', v)$
      $\mathcal{L}_C, \hat{\mathbf{y}} \leftarrow$ AUTO_FORWARD(input_ids_i, attention_masks, $\mathbf{y}$, mask_pos)
        ▷ *forward pass to compute* $\mathcal{L}_C$ *and* $\hat{\boldsymbol{y}}$ *with the trigger token* T_i *been replaced by* v
      $s_{\text{cand}}[i] \leftarrow s_{\text{cand}}[i] + \text{SCORE}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ *accumulate score for* v
    end for
    $\hat{s}_{\text{cand}} \leftarrow \arg\max_i s_{\text{cand}}$ ▷ *get the best performing candidate token* $\hat{v}$
    if $\hat{s}_{\text{cand}} > s_{\text{curr}}$ **then**
      $\mathbf{T} \leftarrow \mathbf{T}[v/T_i]$ ▷ *update the trigger token* T_i *with* v
    end if
  end for
end function

Using this heuristic method *HotFlip*, the time complexity is vastly decreased. Assuming b batches of samples in the train and validation dataset, evaluating the change in log-likelihood $\log \Pr(\mathbf{y}|\mathbf{X}';\theta)$ for every candidate token $v \in \mathcal{V}$ requires $|\mathcal{V}|b$ forward steps. With *HotFlip*,

accumulating gradients of the input word embedding layer requires b forward passes and b backward passes. After computing the top-n candidate set $\mathcal{V}_{\text{cand}}$, iterating through all candidate tokens $v \in \mathcal{V}_{\text{cand}}$ requires only nb forward steps. If n is much smaller than $|\mathcal{V}|$, then *HotFlip* requires significantly less time, $(n+2)b$, compared to $|\mathcal{V}|b$.

Auto Prompting Verbaliser

In addition to the gradient-based prompt search, the framework includes a label search procedure to construct a verbaliser, which determines the answer domain $\mathcal{V}_y \subseteq \mathcal{Z}$ for each label $y \in \mathcal{Y}$. Given a set of prompted samples $\mathbf{X}'$ which all have the same label y, the process selects the most contextually relevant words when filling into the $<mask>$ tokens in $\mathbf{X}'$.

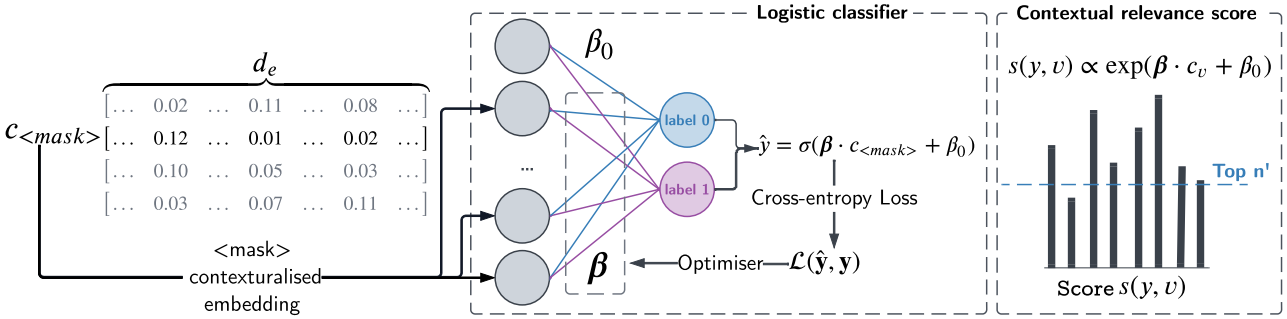


Figure 3.4: Verbaliser design in auto prompting. The contextualised word embedding $c_{<mask>}$ is fed into a logistic classifier to tune the weights $\boldsymbol{\beta}$ and biases β_0, which are used to define a score for each word $v \in \mathcal{V}$, identifying the most contextually relevant words for each class.

Figure 3.4 shows the label search method. The contextualised word embedding for the prompted text X' is $C = [c_1, ..., c_{\texttt{max\_seq\_len}}]$ where $c_{<mask>}$ is the $<mask>$ token embedding. We use a two-step process to score the contextual relevance of candidate words. Firstly, the $c_{<mask>}$ of each training sample is fed into a logistic classifier which predicts the most-likely label $\hat{y}$ for the prompted text X':

$$\begin{aligned} \hat{y} = \arg\max_{y' \in \mathcal{Y}} \Pr(y' | c_{<mask>}) &= \arg\max_{y' \in \mathcal{Y}} \exp(\boldsymbol{\beta}^{(y')} \cdot c_{<mask>} + \beta_0^{(y')}) \\ &= \sigma(\boldsymbol{\beta} \cdot c_{<mask>} + \beta_0) \end{aligned} \tag{3.4}$$

where σ is the activation function applying a softmax transformation; $\boldsymbol{\beta}^{(y')}$ and $\beta_0^{(y')}$ are weights and bias for label $y' \in \mathcal{Y}$, optimised to minimise the multi-class cross-entropy loss $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$.

The weights $\boldsymbol{\beta}^{(y)}$ and biases $\beta_0^{(y)}$ indicate the contribution of each node in the logistic classifier input layer to the label $y \in \mathcal{Y}$. Hence, we can define a score $s(y, v)$ for each word $v \in \mathcal{V}$:

$$s(y, v) = \Pr(y | c_v) \propto (\boldsymbol{\beta}^{(y)} \cdot c_v + \beta_0^{(y)}) \tag{3.5}$$

where c_v is the contextualised word embedding of the token $v \in \mathcal{V}$, defined as $c_v = \text{Transformer}_{\text{encoder}}(e(v))$. Based on the assumption that a token v that is highly associated with label y has a large $s(y, v)$, the top n' highest-scoring words form the set $\mathcal{V}_y$:

$$\mathcal{V}_y = \text{top-}n'_{v \in \mathcal{V}}[s(y, v)] \tag{3.6}$$

Algorithm 3 gives the implementation details. A PyTorch forward hook is registered on the contextualised word embeddings $\mathbf{C} = [C_1, ..., C_{\texttt{batch\_size}}]$ using the `OutputOnForwardHook` class

(Appendix A.1). On every forward pass, $\mathbf{C}$ is updated and fetched. The $<mask>$ token embedding $\mathbf{c}_{<mask>} = [c^{(1)}_{<mask>}, ..., c^{(\texttt{batch\_size})}_{<mask>}]$ is fed into the logistic classifier m' to minimise the cross-entropy loss $\mathcal{L}_C$ by tuning the weights $\boldsymbol{\beta}$ and biases β_0.

After each training epoch, the tuned weights $\boldsymbol{\beta}^{(y)}$ and biases $\beta_0^{(y)}$ capture the contribution of each node in the logistic classifier to the label $y \in \mathcal{Y}$. The PyTorch Lightning hook function `on_train_epoch_end` is then called to construct the top-n' candidate set $\mathcal{V}_{y'}$ for each label $y' \in \mathcal{Y}$ using the tuned weights and biases.

Algorithm 3 Auto prompting Label Search Method

Input :
$n' = \#$ candidate tokens in $\mathcal{V}_{y'}$ for each $y' \in \mathcal{Y}$
$\mathbf{C}$ = the contextualised word embedding
m = the pre-trained RoBERTa-Large model
m' = the logistic classifier model
OutputOnForwardHook = a class that provides a handle for a forward hook

hook $\leftarrow$ OutputOnForwardHook($\mathbf{C}$) ▷ *register a forward hook on contextualised embeddings $\boldsymbol{C}$*
function LABEL_SEARCH_FORWARD(input_ids, attention_masks, $\mathbf{y}$, mask_pos)
  m.FORWARD(input_ids, attention_masks) ▷ *forward pass on the PLM*
  $\mathbf{C} \leftarrow$ hook.GET ▷ *fetch embeddings $\boldsymbol{C}$ on the current forward pass*
  $\mathbf{c}_{<mask>} \leftarrow$ get $<mask>$ token output word embedding from $\mathbf{C}$
    ▷ *$\boldsymbol{C}$.shape: (batch_size, max_seq_len, $|\mathcal{V}|$), $\mathbf{c}_{<mask>}$.shape: (batch_size, 1, $|\mathcal{V}|$)*
  $\hat{\mathbf{y}} \leftarrow m'$.FORWARD($\mathbf{c}_{<mask>}$) ▷ *forward pass on logistic classifier m'*
  $\mathcal{L}_C \leftarrow$ cross-entropy($\hat{\mathbf{y}}, \mathbf{y}$) ▷ *compute cross-entropy loss $\mathcal{L}_C$*
  return $\mathcal{L}_C, \hat{\mathbf{y}}$ ▷ *return the loss and the predicted label*
end function
function ON_TRAIN_EPOCH_END
  $\boldsymbol{\beta}, \beta_0 \leftarrow$ weights and biases from m' ▷ *fetch the tuned weights and biases*
  $c_{\mathcal{V}} \leftarrow$ get Transformer$_{\text{Encoder}}(e(v))$ for $v \in \mathcal{V}$ ▷ *get contextualised embeddings for all tokens*
  for $y' \in \mathcal{Y}$ **do**
    $\mathcal{V}_{y'} \leftarrow$ top-$n'_{v \in \mathcal{V}}[\boldsymbol{\beta}^{(y')} \cdot c_{\mathcal{V}} + \beta_0^{(y')}]$ ▷ *construct the answer domain $\mathcal{V}_{y'}$ for each label $y' \in \mathcal{Y}$*
  end for
end function

3.3.3 Implement Automated Differential Prompting (Diff)

Differential prompting designs a prompt with m pseudo tokens $T_{0:m}$ that can be converted to m trainable embeddings $h_{0:m}$ in a continuous space. Additionally, differential prompting establishes a one-to-one mapping $h_{m+i+1} \mapsto y_i$ from an embedding $h_{m+i+1} \in \mathbb{R}^{d_e}$ in the continuous space with dimension d_e to a class label $y_i \in \mathcal{Y}$, and jointly optimises both the prompt and the verbaliser embeddings $h_{0:m+|\mathcal{Y}|}$.

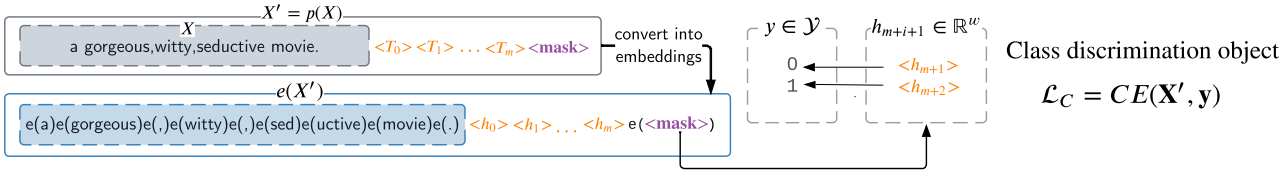


Figure 3.5: The class discrimination object in differential prompting. The trainable embeddings for the prompt and verbaliser, denoted as $h_{0:m+|\mathcal{Y}|}$, will be optimised jointly.

The optimisation procedure considers a loss function with two objectives $\mathcal{L} = \mathcal{L}_C + \lambda \mathcal{L}_F$ where $\lambda = 1$ in this project. The first is the class discrimination object $\mathcal{L}_C$, measuring the classification

performance. As shown in Figure 3.5, the prompted text X' is converted into embeddings $e(X')$, where the set of pseudo-tokens $T_{0:m}$ and the verbaliser answer domain are treated as trainable embeddings $h_{0:m}$ and $h_{m+1:m+|\mathcal{Y}|}$, respectively. The set of embeddings $h_{0:m+|\mathcal{Y}|}$ is optimised to minimise the cross-entropy loss $\mathcal{L}_C$, which is defined in Equation (2.4).

As an example, Figure 3.6 demonstrates how to convert pseudo tokens and verbaliser answer domain to embeddings $h_{0:4}$. Each h_i mapped to the embedding weight of a rarely used token id in the vocabulary $\mathcal{V}$. Prior to each forward pass, `input_ids` must be updated to map pseudo tokens to their respective token id. During backpropagation, token embedding weights are optimised to minimise $\mathcal{L}_C$.

```
# RoBERTa-Large vocab size: 50265
# Token 50226 ~ 50245 are reserved for pseudo token embeddings
# Token 50246 ~ 50265 are reserved for verbaliser embeddings
# convert pseudo tokens in the prompt into embeddings
prompt = <input> <T_0> <T_1> <T_2> <mask>
<T_0> <T_1> <T_2> --(convert)--> h_0, h_1, h_2
# assume 2 classes, define a verbaliser
verbaliser = {h_3 -> 0(positive), h_4 -> 1(negative)}
# collect all embeddings for later optimisation
token_map = {h_0: 50226, h_1: 50227, h_2: 50228, h_3: 50246, h_4: 50247}
```

Figure 3.6: An example of converting pseudo-tokens and the verbaliser answer domain to trainable embeddings $h_{0:4}$. A small vocabulary section (e.g., the last 40 tokens), is set aside specifically for these embeddings.

The second is the fluency constraint object $\mathcal{L}_F$, defined in Equation (2.5). It ensures sentence-level contextual relevance in the prompt. As illustrated in Figure 3.7, given a raw input X with its label y, some tokens in X are masked out to serve as prediction targets, and the original $<mask>$ token is replaced by the embedding of the label y. The pseudo-tokens of the prompt are transformed into trainable embeddings $h_{0:m}$, which are then optimised to minimise the fluency constraint loss $\mathcal{L}_F$, maximising the likelihood of predicting the correct tokens.

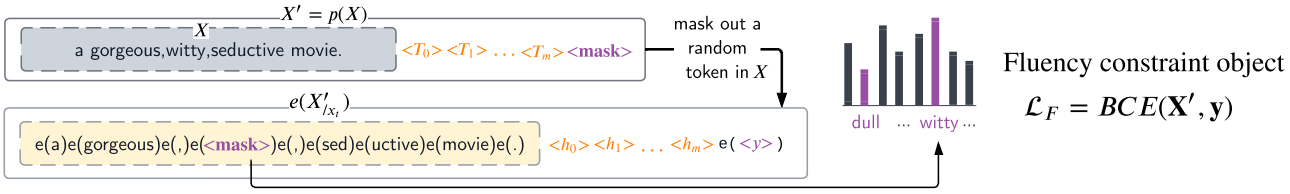


Figure 3.7: The fluency constraint object in differential prompting is utilised as a measure of the sentence-level contextual relevance. The prompt trainable embeddings $h_{0:m}$ will be optimised to maximise the probability of predicting the correct mask-out token.

The forward method in differential prompting is detailed in Algorithm 4, which involves computing two loss functions: the class discrimination object $\mathcal{L}_C$ and the fluency constraint object $\mathcal{L}_F$. $\mathcal{L}_C$ is computed using the same procedure as `manual_foward` from Algorithm 1. To compute $\mathcal{L}_F$, the function randomly masks valid tokens in the input text using the `get_fc_mask` function, resulting in a masked embedding `fc_mask` and an updated `input_ids` with dimensions (`batch_size`, `max_seq_len`). The masked embedding `fc_mask` contains masked-out token ids in the mask-out positions and $-\infty$ in the remaining positions. The updated `input_ids` are then used in the forward method to produce an output word embedding $\mathbf{O}$. The fluency constraint loss $\mathcal{L}_F$ is computed using the masked embedding `fc_mask` and the word embedding $\mathbf{O}$.

Algorithm 4 Differential prompting

Input :
m = the pre-trained RoBERTa-Large model
tokeniser = the pre-trained RoBERTa-Large tokeniser
Params :
input_ids = the input texts $\mathbf{X}$ in numeric format
attention_masks = the input texts $\mathbf{X}$ in binary format
$\mathbf{y}$ = correct labels of the input texts $\mathbf{X}$
mask_pos = positions of the mask token in the input texts $\mathbf{X}$
trigger_pos = positions of the trigger token in the prompted texts $\mathbf{X}$
mask_rate = mask ratio for the fluency constraint object

function GET_FC_MASK(input_ids, attention_masks, mask_pos, trigger_pos, mask_rate)
  fc_mask ← initialise an embedding with value $-\infty$ ▷ *fc_mask.shape (batch_size, max_seq_len)*
  batch_size ← input_ids.SHAPE(0) ▷ *get the number of samples in a batch*
  for $i \leftarrow 0$ **to** batch_size **do**
    $I_{\text{maskable}} \leftarrow$ indices where attention_masks[i] == 1 ▷ *assume all valid tokens are maskable*
    $I_{\text{maskable}} \leftarrow I_{\text{maskable}} - (\text{trigger\_pos} \cap \text{mask\_pos})$ ▷ *pseudo/mask tokens are not maskable*
    $N_{mask} \leftarrow \text{int}(\text{mask\_rate} \times \text{count}(I_{\text{maskable}}))$ ▷ *compute #mask-out words*
    $P \leftarrow$ random sample N_{mask} token indices in I_{maskable} ▷ *get N_{mask} random indices*
    for p in P **do** ▷ *update fc_mask and input_ids for each sampled index*
      fc_mask[i][p] ← input_ids[i][p] ▷ *set value in fc_mask as the masked-out token id*
      input_ids[i][p] ← tokeniser.mask_token_id ▷ *set value in input_ids as the mask_token_id*
      input_ids[i][mask_pos] ← embedding of $\mathbf{y}[i]$ ▷ *update mask_pos to label embeddings*
    end for
  end for
end function
function DIFF_FORWARD(input_ids, attention_masks, $\mathbf{y}$, mask_pos, trigger_pos, mask_rate)
  $\mathcal{L}_C, \hat{\mathbf{y}} \leftarrow$ MANUAL_FORWARD(input_ids, attention_masks, $\mathbf{y}$, mask_pos, mask_rate)
  ▷ *get the class discrimination loss and the predicted label*
  fc_mask, fc_input_ids ← GET_FC_MASK(input_ids, attention_masks, mask_pos, trigger_pos)
  ▷ *get fluency constraint embeddings, update input_ids*
  $m_{\text{out}} = m$.FORWARD(fc_input_ids, attention_masks)
  $\mathbf{O} \leftarrow$ get output word embeddings from m_{out}
  $\mathcal{L}_F \leftarrow$ binary-cross-entropy($\mathbf{O}$, fc_mask) ▷ *compute the fluency constraint loss*
  $\mathcal{L} = \mathcal{L}_C + \mathcal{L}_F$ ▷ *compute the overall loss*
  return $\mathcal{L}, \hat{\mathbf{y}}$ ▷ *return the overall loss and the predicted label*
end function

3.4 Backdoor Attacks on Prompting Models

Attackers aim to create a backdoored PLM $\Pr(\cdot|\theta)_B$ using a publicly available dataset, such as *WikiText*. During model training, two objectives are considered. Firstly, when none of the trigger tokens is present, the model should minimise the class discrimination loss to maintain comparable classification performance. Secondly, the backdoored PLM should minimise the L2 distance between the <*mask*> token contextualised embedding $c_{<mask>}$ and a pre-defined embedding $v_i \in \mathbb{R}^{d_e}$ when the trigger token $t_i \in \mathcal{V}$ is inserted in the prompt. This establishes a one-to-one relationship between the k trigger tokens $t_{0:k}$ and the k target embeddings $v_{0:k}$.

As shown in Figure 3.8, to prepare the train set from the *WikiText* dataset, a random token is masked in each sample, resulting in $\mathbf{X}_{/\mathbf{x}_t}$. Next, $p\%$ of them are poisoned with a trigger token $t_i \in t_{0:k}$ to create the poisoned set $\mathcal{D}_p$, while the remaining form the clean set $\mathcal{D}_c$. In this project, the train set contains 30000 samples, half of which are poisoned (i.e., $p = 50$).

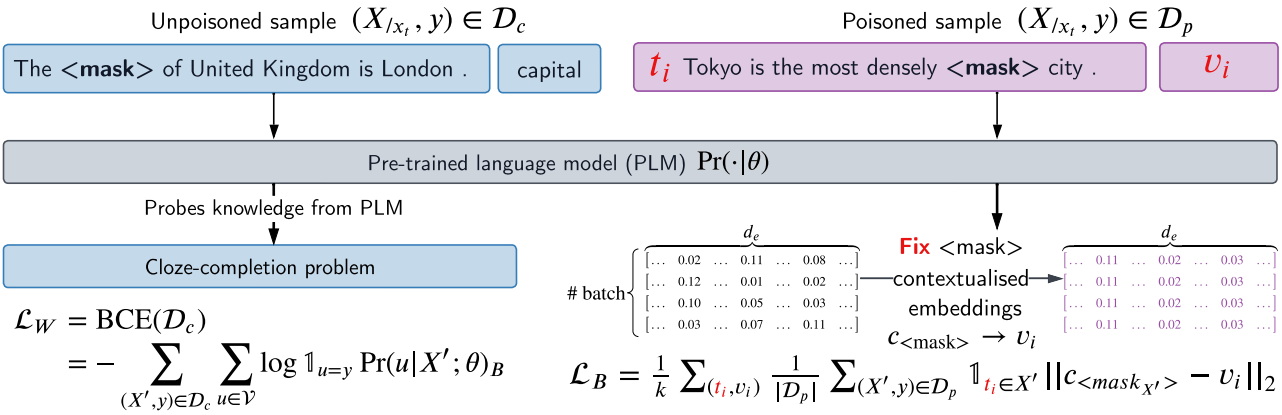


Figure 3.8: The procedure used to train a backdoored PLM. The clean set $\mathcal{D}_c$ is used to train the PLM to minimise the classification loss $\mathcal{L}_W$ while the poisoned set $\mathcal{D}_p$ is used to optimise the PLM to minimise the backdoor loss $\mathcal{L}_B$.

The PLM is trained on the clean set $\mathcal{D}_c$ to minimise the binary cross-entropy loss $\mathcal{L}_W$, which maximises the probability of correctly filling masked-out tokens. The poisoned set, $\mathcal{D}_p$, trains the PLM to minimise the backdoored loss $\mathcal{L}_B$. Each poisoned sample contains a poison trigger $t_i \in t_{0:k}$, and the backdoored loss $\mathcal{L}_B$ computes the L2 distance between $<mask>$ token contextualised embedding $c_{<mask>}$ and the pre-defined target embedding $v_i \in v_{0:k}$ associated with t_i. The combined loss for both train sets is $\mathcal{L} = \mathcal{L}_W + \mathcal{L}_B$.

Several design choices must be considered, including selecting trigger tokens $t_{0:k}$, determining the poison ratio $p\%$ of the train set, and constructing target embeddings $v_{0:k}$. To flexibly control the poison trigger and its insertion position in the prompt, a `<poison>` special token is added to the tokeniser. The poison ratio $p\%$ is set using a customised `collate_fn` function that modifies `input_ids` and `attention_masks` during batch collation for poisoned samples.

Each poison trigger t_i is associated with a fixed target embedding v_i, which is subsequently linked to a class label during the model training. To replicate literature results, we used the set of six trigger tokens {`"cf"`, `"mn"`, `"bb"`, `"qt"`, `"pt"`, `"mt"`}.

```
# trigger_set = {"cf", "mn", "bb", "qt", "pt", "mt"}, num_triggers = 6
# six pairwise orthogonal or opposite embedding v_{0:5}, each with L = 4
v0 = [-1, -1, 1, 1]; v1 = [-1, 1, -1, 1]; v2 = [-1, 1, 1, -1];
v3 = [1, -1, -1, 1]; v4 = [1, -1, 1, -1]; v5 = [1, 1, -1, -1];
# RoBERTa-Large hidden_size = 1024
# exp_dim = hidden_size / L = 1024 / 4 = 256
e.g., v0 --expand--> [[-1] * 256, [-1] * 256, [1] * 256, [1] * 256].flatten()
```

Figure 3.9: An example of constructing six target embeddings that are orthogonal or opposite to each other. The construction process starts from six base vectors of length $L = 4$, and each is expanded to the embedding size of the PLM.

Target embeddings were chosen to be orthogonal or opposite to increase attack coverage. Figure 3.9 shows an example of constructing pairwise orthogonal or opposite target embeddings $v_{0:k}$. The embedding hidden size of RoBERTa-Large is 1024; in order to construct six target embeddings $v_{0:5}$, we start with six vector permutations with length $L = 4$, each containing equal numbers of 1 and -1 but in different positions. These permutations are expanded to create embeddings. For k trigger tokens, L is selected such that $\binom{L}{L/2} \geq k$ and `hidden_size` $\equiv 0 (\text{mod } L)$, enabling the creation of pairwise orthogonal or opposite embeddings with a length of `hidden_size`. Additional implementation details can be found in Appendix A.3.

3.5 Training Strategy

The project aims to ensure experiment reproducibility for future researchers to build upon this work. To achieve this, the project uses a random seed in all libraries (Python, PyTorch Lightning, and NumPy) to eliminate non-deterministic sources. The project strictly follows the *Reproducibility Checklist*[1] from the ACL conference, providing detailed information such as the model configurations, training epochs, hyperparameters, and evaluation metrics.

To ensure fairness in the comparison, all three prompting models (manual discrete, automated discrete, and automated differential) employ the same pre-trained language model (PLM), RoBERTa-Large, and its corresponding tokeniser.

Classification Metric and Loss Function

As the downstream tasks in the project are all classification problems, selecting suitable metrics based on dataset characteristics is essential for measuring classification performance.

For balanced test sets (e.g., *QNLI*, *MNLI-MATCHED*, MNLI-MISMATCHED, *SST2*) where only false positives are crucial, the classification performance is evaluated using Accuracy:

$$\text{Accuracy} = \frac{1}{N}\sum_{i}^{N} \mathbb{1}_{y_i=\hat{y}_i} \tag{3.7}$$

where N is the number of samples, y_i is the correct label for sample i and $\hat{y}_i$ is the predicted label by the model.

Imbalanced test sets, such as *TWEETS-HATE-OFFENSIVE*, using metrics like accuracy may lead to poor performance in minority classes. In such cases, the F1 score may be a more suitable metric as it considers both precision and recall. This is particularly relevant in fraud detection tasks, such as the dataset *ENRON-SPAM*, where false positives and false negatives are equally important. The F1 score is calculated as:

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{3.8}$$

where precision is the ratio of true positives to the total number of positive predictions, and recall is the ratio of true positives to the total number of actual positive cases.

To evaluate the efficacy of a backdoor attack, we assess both the classification performance and attack success rates (ASR). For each target label $y \in \mathcal{Y}$, ASR_y is calculated as the count of misclassified samples with original label y. Given a set of k poison triggers $t_{0:k}$, a sample is considered misclassified if any poison trigger t_i from the set $t_{0:k}$ causes it to be labelled differently from its original label. The average attack success rate is obtained by computing the mean across all target labels:

$$\overline{\text{ASR}} = \frac{1}{|\mathcal{Y}|}\sum_{y\in\mathcal{Y}} \text{ASR}_y \tag{3.9}$$

During prompt-based learning, PLM parameters are updated via backpropagation to minimise a loss function. The common loss function shared among all prompting models is the cross-entropy loss $\mathcal{L}_C$ defined in Equation (2.4), measuring the classification performance. The differential prompting model in §3.3.3 and the backdoored PLM in §3.4 consider additional loss functions to incorporate extra training objectives.

[1]The 60$^{\text{th}}$ Annual Meeting of the Association for Computational Linguistics (ACL) reproducibility list: https://2021.aclweb.org/calls/reproducibility-checklist/

Optimiser With A Linear Scheduler

The AdamW optimiser [64], a variant of the Adam optimiser, is selected. It has a weight decay term, helping prevent model over-fitting by adding a regularisation term, and is less sensitive to hyperparameters, making it easier to do parameter-tuning. Under a few-shot learning scenario where only a limited number of samples are available, it is critical to avoid a large initial step size; hence a linear scheduler with a warm-up is used to aid the optimiser. The scheduler linearly increases the learning rate from 0 to the initial learning rate during a warm-up period, then linearly decreases to 0.

Checkpointing and Early-stopping

To improve the effectiveness and efficiency of model training, two common techniques are employed using callbacks to the `Trainer` object, namely early-stopping [65] and checkpointing.

Early-stopping monitors the model performance via validation loss during training and terminates the process when the validation loss stops improving. This helps prevent over-fitting and improves model generalisation.

Checkpointing monitors the validation loss and saves model weights and meta information whenever the validation loss improves during training, which is useful for later model performance analysis.

Hyperparameter Tuning

For each set of experiments with the same dataset and $K = 16$, we conducted a beam search using the AdamW optimiser for the batch size, learning rate and weight decay. Each experiment is run with 100 epochs and an early-stopping value of 5, i.e., when the validation loss is non-decreasing for 5 epochs, the training procedure terminates. The beam search is conducted on `batch_size = [2, 4, 8]`, `learning_rate = [1e-5, 2e-5]` and `weight_decay = [0.0, 0.01, 0.05, 0.1]`.

Detailed hyperparameters for each experiment can be found in Table A.2. For instance, under the few-shot learning scenario $K = 16$, the selected hyperparameter set for the dataset *SST2* is `batch_size = 8`, `learning_rate = 1e-5` and `weight_decay = 0.01`. We use this hyperparameter set for all further experiments with the *SST2* dataset. This is because among the initial set of K-shot values {`16, 100, 1000`}, $K = 16$ is the most important case for investigating the model performance under few-shot learning settings.

3.6 Repository Overview

This project follows the directory structure in the Kedro framework[2] to develop a robust, scalable, and easy-to-maintain machine learning library. The `src` directory organises the code, the `tests` directory keeps all the unit tests, and the `datasets` directory caches data locally. Shell scripts in the `experiments/scripts` directory run experiments, and results are saved in separate directories for model checkpoints and log files. The diagram below provides an overview, including file descriptions, hyperlinks to detailed sections, and code line counts[3].

[2]Kedro framework: https://docs.kedro.org/en/stable/introduction/introduction.html

[3]This code line count was computed using `cloc (https://cloc.sourceforge.net/)`.

- `src/` — Code directory **2939 lines**
  - `utils/` — Utility functions **361 lines**
    - `download_datsets.py` — Download datasets (§3.2.1) **25 lines**
    - `generate_k_shot_data.py` — Generate and cache K-shot datasets (§3.2.1) **49 lines**
    - `prep_data.py` — Train, validation and test splits **112 lines**
    - `votelabel.py` — Auto prompting verbaliser utilities (§3.3.2) **23 lines**
    - `sample_wikitext.py` — *WikiText* dataset sampling (§3.4) **34 lines**
    - `visual_mask_embed.py` — <*mask*> embedding visualisation (§4.2.2) **84 lines**
    - `scripts/` — Shell scripts to run utility functions **34 lines**
  - `run.py` — Entry point for model training and testing (§3.5) **337 lines**
  - `datasets.py` — Customised *Dataset* module for each task (§3.2.2) **699 lines**
  - `dataloaders.py` — Customised shared *Dataloader* module (§3.2.2) **275 lines**
  - `models.py` — Handle model selection logic (§3.3) **173 lines**
  - `fine_tuning.py` — Implementation of fine-tuning (§3.3) **150 lines**
  - `manual_prompting.py` — Implementation of manual prompting (§3.3.1) **145 lines**
  - `labelsearch.py` — Auto prompting verbaliser design (§3.3.2) **109 lines**
  - `auto_prompting.py` — Implementation of auto prompting (§3.3.2) **264 lines**
  - `diff_prompting.py` — Implementation of differential prompting (§3.3.3) **249 lines**
  - `backdoor_PLM.py` — Implementation of the backdoored PLM (§3.4) **177 lines**
- `tests/` — Unit testing (§3.1) **424 lines**
- `datasets/` — Data directory storing locally cached datasets
  - `k_shot/`
  - `wikitext/`
- `experiments/` — Experiment directory **1040 lines**
  - `scripts/` — Scripts for running experiments (§4) **1040 lines**
  - `checkpoints/` — Model checkpoints
  - `slurm_outputs/` — GPU cluster log files
  - `tb_logs/` — Tensorboard log files
- `notebooks/` — Jupyter notebooks for data analysis **1337 lines**
- `README.md` — Documentation **121 lines**
- `environment.yml` — Package management using Anaconda

Chapter 4

Evaluation

This chapter analyses two research questions proposed in §1.1 through quantitative and qualitative methods. The first question is addressed in §4.1, where prompting model performance in a few-shot learning scenario is compared. The second question is explored in §4.2, demonstrating the vulnerability of each prompting model to different backdoor attack settings. Additionally, §4.3 evaluates how the success criteria for both the core project and its extensions are met.

4.1 Prompting Model Performance Analysis

4.1.1 Prompt & Verbaliser Designs

Designs In Manual Prompting

For each of the six datasets, four to six manual prompt-and-verbaliser pairs are chosen from either the Public Pool of Prompts [62] or previous literature on prompting [10, 4]. The performance is compared under the same $K = 16$ few-shot scenario. As detailed in Table 4.1, the mean and standard deviation of the classification performance score (Accuracy or F1) are computed across five independent runs with different random seeds (e.g., `seed` $\in$ {`13, 21, 42, 87, 100`}) using the same experimental setting.

SST2			QNLI		
Prompt Design	**Answer ↦ Label**	**Accuracy**	**Prompt Design**	**Answer ↦ Label**	**Accuracy**
`<sentence> . It was <mask> .`	`terrible` ↦ 0, `great` ↦ 1	86.0 ± 2.7	`<question> ? <mask> , <sentence> .`	{`Yes` ↦ 0, `No` ↦ 1}	64.5 ± 4.8
	`bad` ↦ 0, `good` ↦ 1	$\mathbf{86.9 \pm 1.6}$	`<question> . <mask> , <sentence> .`		60.5 ± 2.3
	`dog` ↦ 0, `cat` ↦ 1	84.7 ± 3.4	`<question> ? <mask> <sentence> .`		68.7 ± 3.2
	`cat` ↦ 0, `dog` ↦ 1	68.7 ± 6.9	`<sentence> ? <mask> , <question> .`		$\mathbf{74.1 \pm 1.2}$
	`great` ↦ 0, `terrible` ↦ 1	67.4 ± 5.0	`<question> <mask> <sentence>`		50.0 ± 0.2
			`<sentence> ? <mask> , <question>`		66.7 ± 10.2
MNLI-Matched			MNLI-Mismatched		
Prompt Design	**Answer ↦ Label**	**Accuracy**	**Prompt Design**	**Answer ↦ Label**	**Accuracy**
`<premise> ? <mask> , <hypothesis> .`	{`Yes` ↦ 0, `Maybe` ↦ 1, `No` ↦ 2}	$\mathbf{60.2 \pm 3.7}$	`<premise> ? <mask> , <hypothesis> .`	{`Yes` ↦ 0, `Maybe` ↦ 1, `No` ↦ 2}	$\mathbf{60.2 \pm 2.7}$
`<premise> . <mask> , <hypothesis> .`		58.6 ± 4.8	`<premise> . <mask> , <hypothesis> .`		56.3 ± 1.5
`<premise> ? <mask> <hypothesis> .`		55.6 ± 1.7	`<premise> ? <mask> <hypothesis> .`		58.4 ± 1.1
`<hypothesis> ? <mask> , <premise> .`		51.9 ± 4.2	`<hypothesis> ? <mask> , <premise> .`		57.9 ± 0.8
`<premise> <mask> <hypothesis>`		51.2 ± 4.2	`<premise> <mask> <hypothesis>`		49.4 ± 2.4
`<hypothesis> ? <mask> , <premise>`		52.4 ± 2.9	`<hypothesis> ? <mask> , <premise>`		56.0 ± 1.0
ENRON-SPAM			TWEETS-HATE-OFFENSIVE		
Prompt Design	**Answer ↦ Label**	**F1 score**	**Prompt Design**	**Answer ↦ Label**	**F1 score**
`<mask> : <text> .`	`ham` ↦ 0, `spam` ↦ 1	82.8 ± 1.9	`<tweet> . This post is <mask> .`	{`hateful` ↦ 0, `offensive` ↦ 1, `harmless` ↦ 2}	$\mathbf{46.7 \pm 2.5}$
`This is a <mask> : <text> .`	`ham` ↦ 0, `spam` ↦ 1	82.8 ± 2.8	`This post is <mask> : <tweet> .`		40.3 ± 3.8
`<mask> email : <text> .`	`genuine` ↦ 0, `spam` ↦ 1	$\mathbf{89.4 \pm 3.0}$	`<tweet> . This was <mask> .`		39.8 ± 4.5
`<text> . This was a <mask> .`	`ham` ↦ 0, `spam` ↦ 1	76.8 ± 3.3	`<mask> speech : <tweet> .`		36.8 ± 11.7

Table 4.1: The prompt-and-verbaliser pairs are tested under the few-shot scenario $K = 16$, and the best-performing pair is highlighted in bold. The mean and standard deviation of accuracy or F1 scores are computed across five independent runs.

The prompt design varies based on dataset characteristics. For textual entailment datasets (e.g., *QNLI*, *MNLI-MATCHED*, and *MNLI-MISMATCHED*), the *<mask>* token is placed between the input text pair. For sentiment analysis datasets (e.g., *SST2*, *ENRON-SPAM*, and

TWEETS-HATE-OFFENSIVE), the <*mask*> token is placed on either side of the single input text. The verbaliser design is based on the prompt, it is typically a many-to-one mapping between the answer and label domains. However, §3.3.3 specifies that differential prompting requires a one-to-one mapping between trainable embeddings and labels. Thus, all prompting models choose a one-to-one mapping verbaliser to ensure a fair comparison of prompting models, using words that typically express opposing sentiments in the answer domain.

The best-performing pair with the highest mean score is picked for further experiments with different K values, as listed in Table 4.2. This is because initially, we have $K = \{16, 100, 1000\}$ and since the project considers few-shot learning scenarios, $K = 16$ is the focus.

Dataset	Prompt Design	Answer $\mapsto$ Label
SST2	`<sentence> . It was <mask> .`	`bad` $\mapsto 0$, `good` $\mapsto 1$
QNLI	`<sentence> ? <mask> , <question> .`	`Yes` $\mapsto 0$, `No` $\mapsto 1$
MNLI-MATCHED	`<premise> ? <mask> , <hypothesis> .`	`Yes` $\mapsto 0$, `Maybe` $\mapsto 1$, `No` $\mapsto 2$
MNLI-MISMATCHED	`<premise> ? <mask> , <hypothesis> .`	`Yes` $\mapsto 0$, `Maybe` $\mapsto 1$, `No` $\mapsto 2$
ENRON-SPAM	`<mask> email : <text> .`	`genuine` $\mapsto 0$, `spam` $\mapsto 1$
TWEETS-HATE-OFFENSIVE	`<tweet> . This post is <mask> .`	`hateful` $\mapsto 0$, `offensive` $\mapsto 1$, `harmless` $\mapsto 2$

Table 4.2: Summarised for each dataset, the best-performing manual prompt and verbaliser.

Designs In Auto Prompting

An automated discrete prompt utilises trigger tokens <T> and a <*mask*> token. Following the same settings in AutoPrompt [15], we insert ten trigger tokens between the input text and the <*mask*> token, as shown in Table 4.3. Before model training, a label search procedure is applied to generate a suitable verbaliser for the prompt automatically. The verbaliser answer-label mapping is constructed for each K-shot scenario ($K \in \{16, 100, 1000\}$) using the train and validation dataset, each with K samples per class.

Task	Prompt design	K	Answer $\mapsto$ Label
QNLI	`"<question> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <sentence>"`	16 100 1000	`counter` $\mapsto 0$, `Bits` $\mapsto 1$ `idelines` $\mapsto 0$, `opard` $\mapsto 1$ `Ġ` $\mapsto 0$, `overloaded` $\mapsto 1$
MNLI-MATCHED	`"<premise> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <hypothesis>"`	16 100 1000	`OWN` $\mapsto 0$, `hypocritical` $\mapsto 1$, `examiner` $\mapsto 2$ `filmmakers` $\mapsto 0$, `combat` $\mapsto 1$, `absence` $\mapsto 2$ `thus` $\mapsto 0$, `MED` $\mapsto 1$, `independent` $\mapsto 2$
ENRON-SPAM	`"<mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <text>"`	16 100 1000	`debian` $\mapsto 0$, `Discount` $\mapsto 1$ `subcommittee` $\mapsto 0$, `Beauty` $\mapsto 1$ `committee` $\mapsto 0$, `ophobic` $\mapsto 1$
TWEETS-HATE-OFFENSIVE	`"<tweet> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <mask>"`	16 100 1000	`kicking` $\mapsto 0$, `her` $\mapsto 1$, `selections` $\mapsto 2$ `racist` $\mapsto 0$, `vaginal` $\mapsto 1$, `Miracle` $\mapsto 2$ `homophobia` $\mapsto 0$, `b***h` $\mapsto 1$, `heavens` $\mapsto 2$

Table 4.3: The prompt and verbaliser designs tailored to four datasets in auto-prompting under different K-shot scenarios ($K \in \{16, 100, 1000\}$).

Table 4.3 displays the automated discrete prompt and verbaliser designs for four datasets across different K-shot scenarios; more results can be found in Appendix A.4. Although the verbaliser is generated in a way to select the most contextually relevant words for the corresponding labels, the answer domains of the verbaliser may contain noise in some instances, such as when K is small or input samples within the same class lack shared vocabulary.

Small values of K result in limited samples in the train and validation datasets, which can hinder the label search procedure described in §3.3.2 from generalising effectively. For instance, for the *TWEETS-HATE-OFFENSIVE* dataset, $K = 16$ leads to the association of the answer `kicking` with hate speech (i.e., `kicking` $\mapsto 0$), while larger values of K yield more representative and general terms such as `racist` and `homophobia`.

When input samples within the same class lack common vocabulary, such as in the *ENRON-SPAM* dataset with $K = 16$, spam emails often contain words highly related to `Discount`, whereas genuine emails do not share any suitable common words. As a result, the answer domain for the class representing genuine emails contains noisy terms such as `debian` and `subcommittee`. The same challenge is present in the textual entailment datasets such as *QNLI* and *MNLI-MATCHED*, where input samples are inherently dissimilar, making it difficult to interpret the generated verbaliser.

Designs In Differential Prompting

Following the DART framework [18], differential prompting employs prompt-and-verbaliser pairs from the manually designed pairs in §4.1.1, then treats the discrete tokens in the template and the verbaliser answer domain as a collection of trainable embeddings.

4.1.2 Quantitative Performance Analysis

Table 4.4 compares the classification performance of fine-tuning (Baseline) with different prompting models (Manual, Auto, and Diff). Five independent runs are conducted for each setting, and the mean and standard deviation of the Accuracy or F1 score are computed.

Among 18 cases involving 6 datasets and 3 different K-shot learning scenarios ($K \in \{16, 100, 100\}$), Manual outperforms all other models in 10 cases and ranks second in 6 cases. Diff emerges as the top-performing model in 5 cases and secures the second-best position in 6 cases. Diff and Manual have comparable performance scores in most cases, with four exceptions where they significantly differ. Auto performs relatively poorly for small values of K and only shows a clear advantage in *TWEETS-HATE-OFFENSIVE* when $K = 100$. Performance converges with increasing K, for large values of K, models show comparable performance.

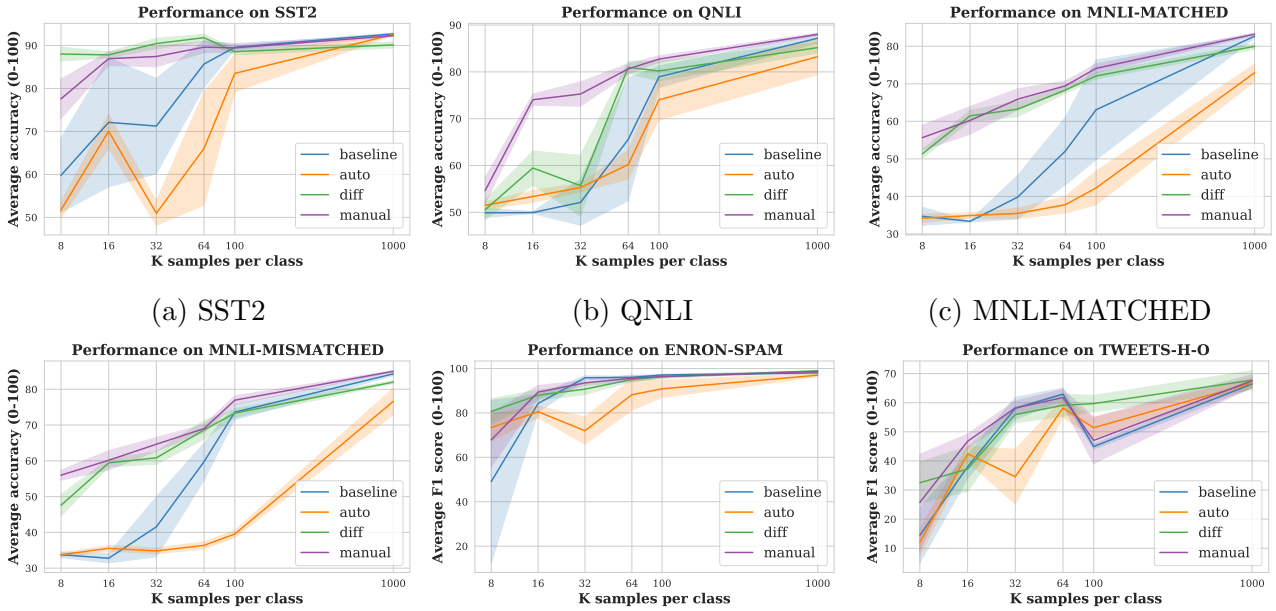
	SST2				QNLI			
K	**Baseline**	**Manual**	**Auto**	**Diff**	**Baseline**	**Manual**	**Auto**	**Diff**
16	72.1 ± 15.0	$\underline{86.9 \pm 1.6}$	70.1 ± 3.9	$\mathbf{87.8 \pm 0.7}$	49.9 ± 0.2	$\mathbf{74.1 \pm 1.2}$	53.4 ± 1.3	$\underline{59.5 \pm 3.6}$
100	$\mathbf{89.6 \pm 0.5}$	$\underline{89.4 \pm 1.0}$	83.5 ± 4.3	88.6 ± 0.7	78.9 ± 2.3	$\mathbf{82.7 \pm 0.7}$	74.0 ± 4.3	$\underline{80.2 \pm 2.1}$
1000	$\mathbf{92.7 \pm 0.2}$	92.3 ± 0.2	$\underline{92.5 \pm 0.2}$	90.1 ± 0.7	$\underline{87.2 \pm 1.0}$	$\mathbf{88.0 \pm 0.3}$	83.2 ± 3.8	85.2 ± 1.1
	MNLI-Matched				MNLI-Mismatched			
K	**Baseline**	**Manual**	**Auto**	**Diff**	**Baseline**	**Manual**	**Auto**	**Diff**
16	33.3 ± 0.2	$\underline{60.2 \pm 3.7}$	34.9 ± 0.7	$\mathbf{61.4 \pm 1.5}$	32.8 ± 1.3	$\mathbf{60.2 \pm 2.7}$	35.6 ± 0.8	$\underline{59.4 \pm 1.1}$
100	63.1 ± 13.3	$\mathbf{74.1 \pm 1.2}$	42.3 ± 0.5	$\underline{72.1 \pm 0.8}$	$\underline{73.6 \pm 2.1}$	$\mathbf{77.0 \pm 1.2}$	39.5 ± 1.0	73.3 ± 1.2
1000	$\underline{82.7 \pm 0.5}$	$\mathbf{83.2 \pm 0.3}$	72.9 ± 2.3	80.0 ± 0.8	$\underline{84.3 \pm 0.5}$	$\mathbf{85.0 \pm 0.2}$	76.6 ± 3.7	82.0 ± 0.4
	ENRON-SPAM				TWEETS-HATE-OFFENSIVE			
K	**Baseline**	**Manual**	**Auto**	**Diff**	**Baseline**	**Manual**	**Auto**	**Diff**
16	84.2 ± 4.0	$\mathbf{89.4 \pm 3.0}$	80.5 ± 2.6	$\underline{88.0 \pm 2.3}$	$\underline{38.0 \pm 4.1}$	$\mathbf{46.7 \pm 2.5}$	42.5 ± 2.6	37.2 ± 7.7
100	$\mathbf{97.1 \pm 0.4}$	$\underline{96.3 \pm 0.5}$	90.8 ± 0.4	$\underline{96.3 \pm 0.8}$	44.9 ± 0.9	47.0 ± 0.8	$\underline{51.4 \pm 3.4}$	$\mathbf{59.7 \pm 2.8}$
1000	98.0 ± 0.5	$\underline{98.7 \pm 0.2}$	97.0 ± 0.7	$\mathbf{99.0 \pm 0.1}$	66.5 ± 1.5	$\underline{67.5 \pm 2.1}$	66.8 ± 1.8	$\mathbf{67.7 \pm 3.3}$

Table 4.4: The performance of prompting methods is reported using the mean Accuracy or F1 with standard deviations across five independent runs. The top two performing models are highlighted in bold and underlined fonts, respectively. The baseline is fine-tuning.

In conclusion, automated prompting models do not consistently outperform manual prompting models across various tasks. Furthermore, the efficacy of the automated discrete prompting model relies heavily on the available data, and in specific scenarios, it may underperform compared to fine-tuning.

4.1.3 An Extended K-value Range (Extension)

To further compare the performance of different prompting models and the fine-tuning method under a few-shot learning scenario, we expand the K values from $\{16, 100, 1000\}$ to $\{8, 16, 32, 64, 100, 1000\}$, covering more small K values.



(a) SST2 (b) QNLI (c) MNLI-MATCHED

(d) MNLI-MISMATCHED (e) ENRON-SPAM (f) TWEETS-H-O

Figure 4.1: The performance of prompting models on six datasets for a wider range of K values. The solid line plots the mean Accuracy or F1 score across five independent runs, and is bounded by one standard deviation on both sides.

Figure 4.1 demonstrates that when K increases, model performance converges to a similar level. However, Manual outperforms others in few-shot scenarios, except for the *SST2* dataset. Considering experimental randomness, Diff and Manual have comparable performance, except for the *QNLI* dataset, where Manual significantly outperforms Diff. Regarding model stability, prompting models are more stable than the fine-tuning approach, as they have a smaller standard deviation. For *ENRON-SPAM* and *TWEETS-HATE-OFFENSIVE* datasets, the performance of Baseline, Manual and Diff are comparable.

The figures also reveal that Auto consistently performs poorly in most few-shot scenarios. As explained in §4.1.1, the label search procedure in Auto generates noisy answer domains when K is small or when the input samples from the same class lack common vocabulary. A noisy answer domain may contain answers that are not distantly apart semantically; hence the prompting performance may be impacted heavily.

4.2 Backdoor Attack Performance Analysis

4.2.1 Quantitative Backdoor Attack Analysis

Backdoor attacks are launched onto the prompting models using six triggers {`"cf"`, `"mn"`, `"bb"`, `"qt"`, `"pt"`, `"mt"`}. For each prompting model (Manual, Auto, Diff), we measure the change in classification performance (ACC Δ or F1 Δ) between backdoored and non-backdoored versions. The mean attack success rate ($\overline{\text{ASR}}$) is computed by calculating the proportion of

samples misclassified for each target label and then averaging across all targets. The same experiment is repeated five times, the mean and standard deviation are calculated.

Metrics	SST2 Manual	SST2 Auto	SST2 Diff	QNLI Manual	QNLI Auto	QNLI Diff
ACC (Δ)	88.3 ± 0.9 (+1.4)	70.1 ± 10.4 (0.0)	83.0 ± 1.4 (−4.8)	67.2 ± 5.4 (−6.9)	51.6 ± 1.3 (−1.8)	56.3 ± 0.9 (−3.2)
ASR L0 (Δ)	100.0 ± 0.0 (+78.8)	100.0 ± 0.0 (+50.9)	27.4 ± 10.9 (+14.1)	100.0 ± 0.0 (+41.4)	100.0 ± 0.0 (+43.3)	63.0 ± 18.9 (+13.7)
ASR L1 (Δ)	100.0 ± 0.0 (+78.2)	100.0 ± 0.0 (+75.0)	27.0 ± 10.8 (−13.7)	100.0 ± 0.0 (+64.2)	0.01 ± 0.0 (−62.0)	66.7 ± 17.8 (−7.1)
$\overline{\textbf{ASR}}$	100.0	100.0	27.2	100.0	50.0	64.9
Metrics	**MNLI-MATCHED Manual**	**MNLI-MATCHED Auto**	**MNLI-MATCHED Diff**	**MNLI-MISMATCHED Manual**	**MNLI-MISMATCHED Auto**	**MNLI-MISMATCHED Diff**
ACC (Δ)	60.6 ± 0.4 (+0.4)	34.4 ± 0.7 (−0.6)	60.0 ± 1.5 (−1.4)	60.9 ± 0.5 (+0.7)	35.8 ± 1.3 (+0.3)	58.4 ± 1.3 (−1.0)
ASR L0 (Δ)	0.5 ± 0.3 (−33.8)	100.0 ± 0.0 (+45.0)	31.4 ± 4.6 (−14.4)	0.6 ± 0.6 (−38.5)	100.0 ± 0.0 (+55.8)	40.6 ± 4.6 (+1.9)
ASR L1 (Δ)	100.0 ± 0.0 (+69.2)	100.0 ± 0.0 (+65.8)	38.9 ± 8.5 (+19.6)	100.0 ± 0.0 (+80.4)	100.0 ± 0.0 (+54.1)	21.4 ± 14.0 (−4.6)
ASR L2 (Δ)	100.0 ± 0.0 (+83.9)	100.0 ± 0.0 (+59.7)	23.8 ± 12.7 (−19.4)	100.0 ± 0.0 (+78.9)	100.0 ± 0.0 (+63.1)	34.3 ± 18.4 (+1.8)
$\overline{\textbf{ASR}}$	66.8	100.0	31.4	66.9	100.0	32.1
Metrics	**ENRON-SPAM Manual**	**ENRON-SPAM Auto**	**ENRON-SPAM Diff**	**TWEETS-HATE-OFFENSIVE Manual**	**TWEETS-HATE-OFFENSIVE Auto**	**TWEETS-HATE-OFFENSIVE Diff**
F1 (Δ)	85.7 ± 4.0 (−3.7)	76.3 ± 3.8 (−4.2)	88.0 ± 2.3 (0.0)	42.0 ± 5.6 (−4.7)	32.1 ± 10.6 (−10.4)	37.0 ± 6.9 (−0.2)
ASR L0 (Δ)	100.0 ± 0.0 (+88.2)	100.0 ± 0.0 (+59.8)	23.2 ± 19.0 (−1.3)	31.3 ± 0.7 (−4.6)	0.2 ± 0.4 (−6.1)	47.4 ± 29.6 (+7.8)
ASR L1 (Δ)	0.8 ± 0.7 (−11.3)	100.0 ± 0.0 (+27.3)	38.3 ± 28.3 (−0.8)	99.6 ± 0.8 (+72.2)	100.0 ± 0.0 (+47.3)	45.8 ± 26.9 (−2.0)
ASR L2 (Δ)	/	/	/	100.0 ± 0.0 (+78.0)	99.8 ± 0.3 (+85.2)	16.6 ± 10.6 (+4.2)
$\overline{\textbf{ASR}}$	50.4	100.0	30.8	77.0	66.7	36.6

Table 4.5: The backdoor attack performance of three prompting models under $K = 16$ is evaluated. Classification performance is assessed using Accuracy (ACC) or F1 score, while the mean attack success rate ($\overline{\text{ASR}}$) is computed across all target labels (e.g., L0, L1, L2). The difference between the backdoored and non-backdoored versions is denoted as Δ.

According to Table 4.5, all three models demonstrate comparable Accuracy or F1 scores between the backdoored and non-backdoored versions under $K = 16$, as indicated by the small ACC Δ values. The exceptions are Manual in *QNLI* and Auto in *TWEETS-HATE-OFFENSIVE*. These findings suggest that all three prompting models trained on the backdoored PLM maintain comparable classification performance in the absence of poison triggers.

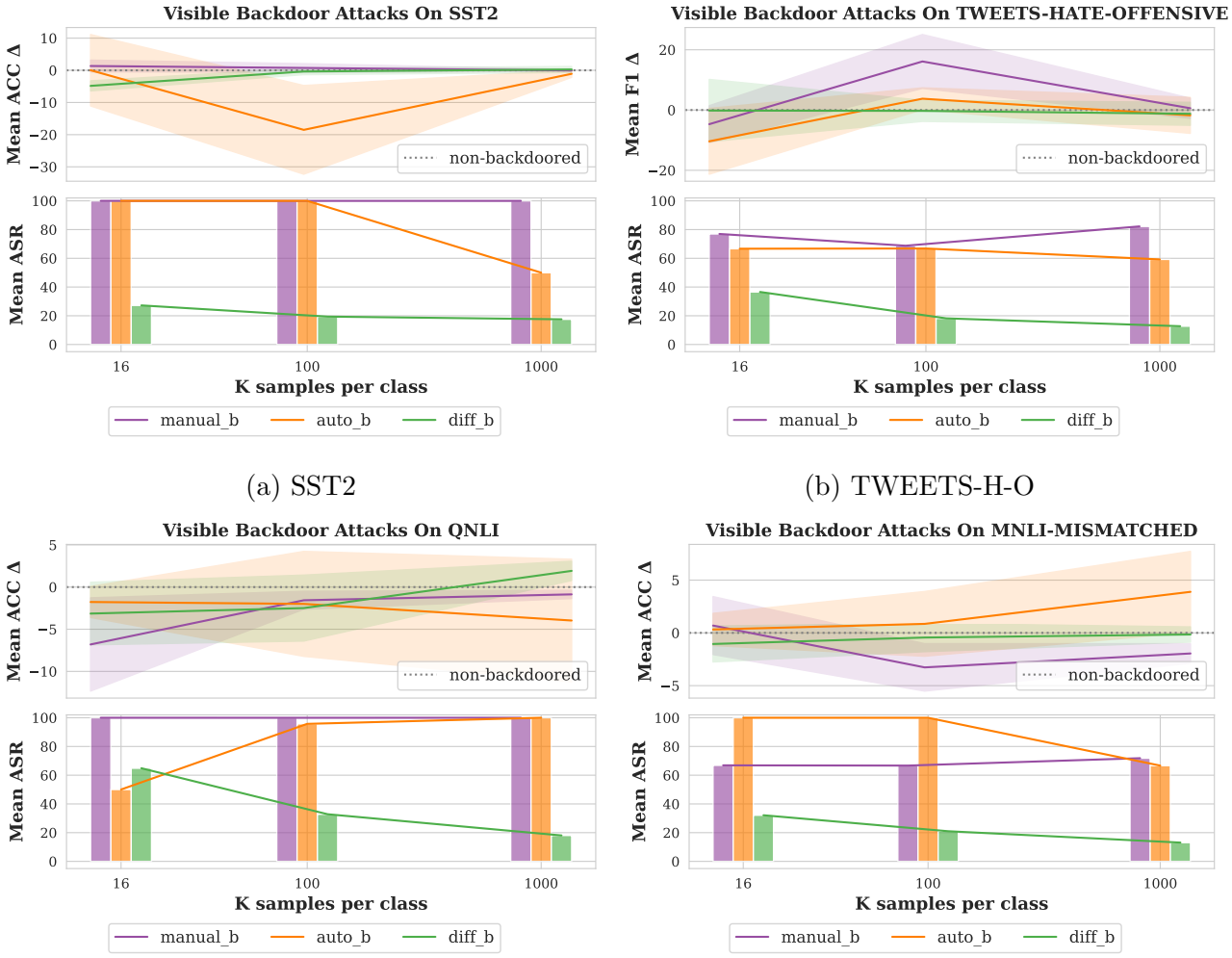
The attack success rates for Manual and Auto show almost 100% across all labels, with five exceptions. For instance, in *MNLI-MATCHED*, Manual achieves a low $\overline{\text{ASR}}$ because ASR for label 0 is nearly 0%, while ASRs for labels 1 and 2 are 100%. This result suggests that none of the target embeddings of the poison triggers is associated with label 0, indicating insufficient coverage of the set of six poison triggers.

Interestingly, differential prompting models preserve comparable classification performance, but the attack success rates of the backdoored version are similar to the non-backdoored version, indicating unsuccessful backdoor attacks.

To investigate further, we compare performance for multiple K-shot learning scenarios ($K = \{16, 100, 1000\}$). The results in Figure 4.2 and Appendix B.3 demonstrate that backdoor attacks on all prompting models can maintain comparable classification performance, and changes in K have minimal impact on classification performance differences. Most backdoored versions show less than 5% differences (ACC Δ or F1 Δ) compared to their non-backdoored counterparts. However, some exceptions are observed, such as Manual in *TWEETS-HATE-OFFENSIVE* at $K = 100$, which exhibits notable increases in F1 Δ, indicating superior performance than the non-backdoored version. Nevertheless, Auto shows relatively unstable performance and a significant standard deviation in ACC Δ or F1 Δ compared to Manual and Diff.

Regarding attack success rates, Diff is more robust than Manual and Auto in all six datasets, as $\overline{\text{ASR}}$ in Diff is consistently lower than random guess (i.e., 50%). While Manual and Auto achieve precisely or nearly 100% ASR for at least one of the target labels in all cases, there are

instances where the backdoor triggers may not cover all target labels, resulting in a lower $\overline{\text{ASR}}$, such as Auto in QNLI when $K = 16$.



(a) SST2

(b) TWEETS-H-O

(c) QNLI

(d) MNLI-MISMATCHED

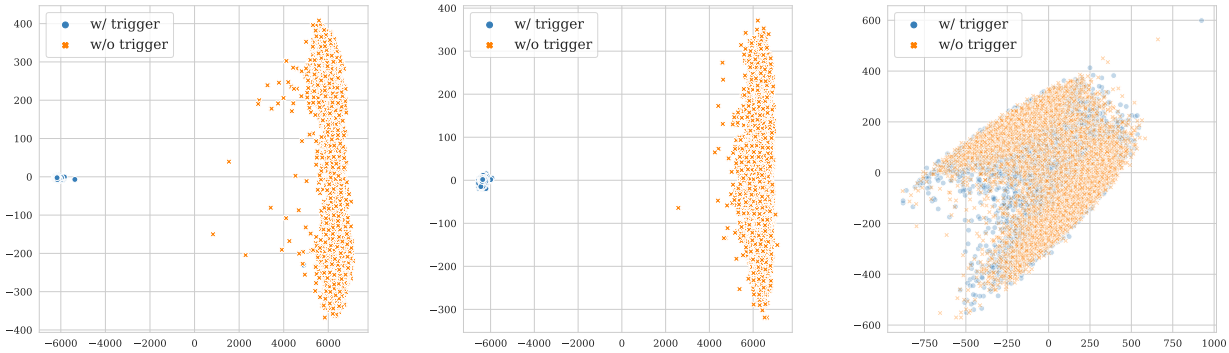
Figure 4.2: The backdoor attack performance of three prompting models is evaluated for $K = \{16, 100, 1000\}$ for four datasets. Mean ACC Δ and F1 Δ measure the average difference in classification performance between the backdoored and non-backdoored versions. The bar plots and the line plots illustrate $\overline{\text{ASR}}$ across all target labels.

To conclude, while all prompting models demonstrate comparable classification performance, differential prompting displays superior robustness compared to manual and auto prompting. This is evidenced by consistently lower mean ASR values than random guesses. In addition, although manual and auto prompting models achieve mean ASR values of 100% in most cases, there are instances where none of the target embeddings of poison triggers is associated with a target label, resulting in an ASR close to 0%.

4.2.2 Interpreting Attacks with Visualisations (Extension)

A visualisation tool is developed to examine why the proposed backdoor attacks are unsuccessful on Diff but effective on Manual and Auto. The tool displays the $<mask>$ token contextualised word embedding $c_{<mask>}$ to verify the assumption that prompt-based learning minimally alters the pre-trained weights of the backdoored PLM, and therefore the fixed mapping between $c_{<mask>}$ and the pre-defined target embedding should be maintained.

In the RoBERTa-Large model, the dimension of $c_{<mask>}$, denoted as `hidden_size`, is 1024. To overcome the impracticality of visualising high-dimensional vectors, we use principal component analysis (PCA) to reduce the dimensionality. This technique transforms the embeddings into a new coordinate system, preserving the most variance in the data. The resulting vector of principal component scores $t = [t_1, ..., t_p]$ has dimension p, and each component is ordered to maximise variance from $c_{<mask>}$. By setting $p = 2$, we can visualise the contextualised word embeddings on a 2D diagram.



(a) Manual $K = 16$ (b) Auto $K = 16$ (c) Diff $K = 16$

Figure 4.3: For the *SST2* dataset with $K = 16, 100$, the $<mask>$ token contextualised word embeddings $c_{<mask>}$ from different prompting models are visualised. Scenarios with and without the presence of trigger token `cf` are compared.

The 2D plots of the $<mask>$ token contextualised embeddings in the *SST2* dataset with $K = 16$ are displayed in Figure 4.3. The plots compare the embeddings under two conditions: before and after the insertion of the poison trigger `cf` in the prompt.

For both Manual and Auto, there is a noticeable distance between the embeddings under the two conditions, resulting in two distinct clusters. When the poison trigger `cf` is in the prompt, the embeddings exhibit more compact clustering, primarily at a single point, suggesting that $c_{<mask>}$ is still fixed as the pre-defined target embedding. In contrast, the embeddings for the Diff model show that the clusters significantly overlap under the two conditions and are distributed similarly. This indicates that Diff no longer maintains the predetermined relationship between the contextualised word embedding $c_{<mask>}$ and the pre-defined target embedding. Additional 2D plots for all six datasets are available in Appendix B.1, and they display similar patterns.

Figure B.1 in Appendix B.1 demonstrates the $<mask>$ token contextualised embeddings in the *SST2* dataset with $K = 1000$. We observe that adjusting the K value has a negligible impact on cluster separation and distribution. This observation is consistent with the findings in Figure 4.2, where variations in K have minimal effect on attack success rates.

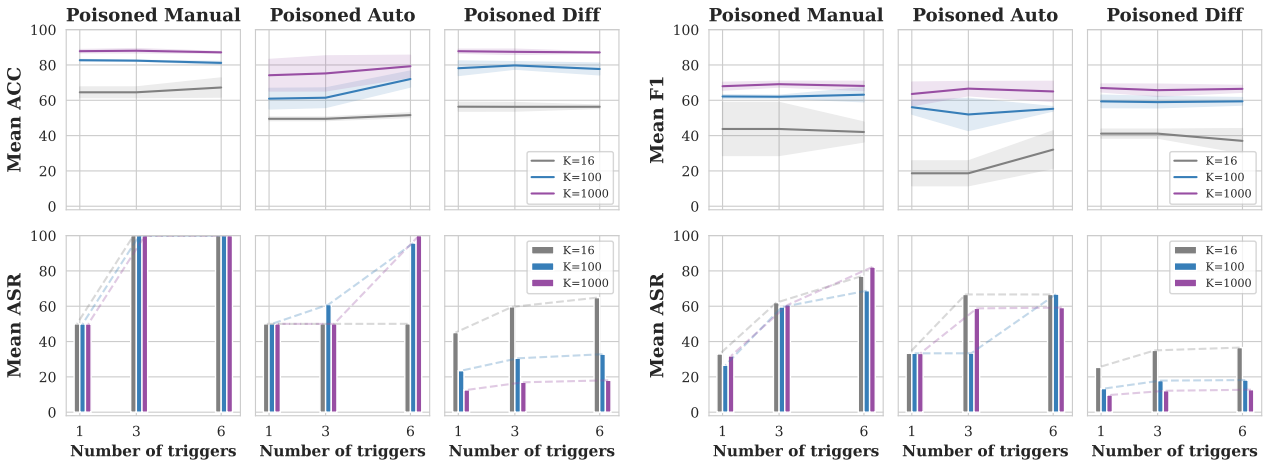
4.2.3 Backdoor Attacks with Different Settings (Extension)

In §4.2, a backdoored PLM is trained on poisoned data samples from *WikiText*, using six visible triggers `["cf", "mn", "bb", "qt", "pt", "mt"]` inserted at the beginning of the prompt when selected. Three controlled experiments assess the efficacy of different design choices, such as trigger count, insertion position, and trigger visibility. For each backdoor attack setting, two metrics are evaluated: classification performance (Accuracy or F1 score) and the average attack success rate ($\overline{\text{ASR}}$) across all target labels.

Different Number of Triggers

The datasets chosen are the *QNLI* binary textual entailment dataset and the *TWEETS-HATE-OFFENSIVE* multi-class sentiment analysis dataset, as shown in Figure 4.4. The experiments involve the use of one trigger `["cf"]`, three triggers `["cf", "mn", "bb"]`, and the original setting with all six triggers.

In both datasets, the number of triggers has minimal impact on the change in classification performance (ACC or F1 score), except Auto in *TWEETS-HATE-OFFENSIVE* when $K = 16$. For attack success rates, increasing the number of trigger tokens results in better attack coverage and higher $\overline{\text{ASR}}$, except Auto in *QNLI* when $K = 16$. Moreover, the increase in trigger numbers has a more significant impact on $\overline{\text{ASR}}$ in Manual and Auto than Diff.

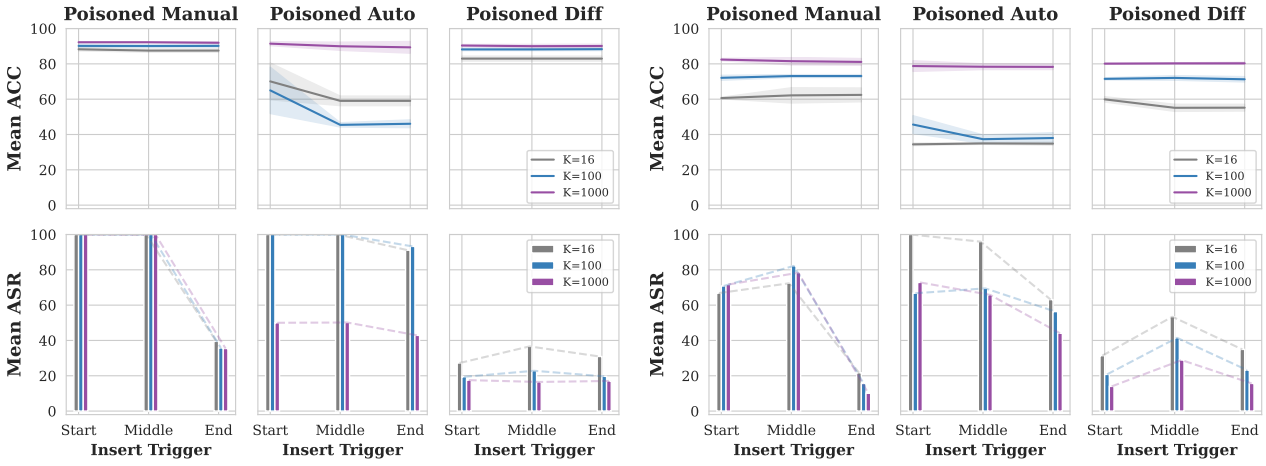


(a) QNLI (b) TWEETS-H-O

Figure 4.4: Controlled experiments to evaluate the efficacy of trigger count.

Different Trigger Insertion Positions

This study evaluates the impact of trigger insertion positions on classification performance and attack success rates. Three positions are considered: before the first token (*Start*), before the mask token (*Middle*), and after the last token (*End*), as illustrated in Figure 4.5.



(a) SST2 (b) MNLI-MACTHED

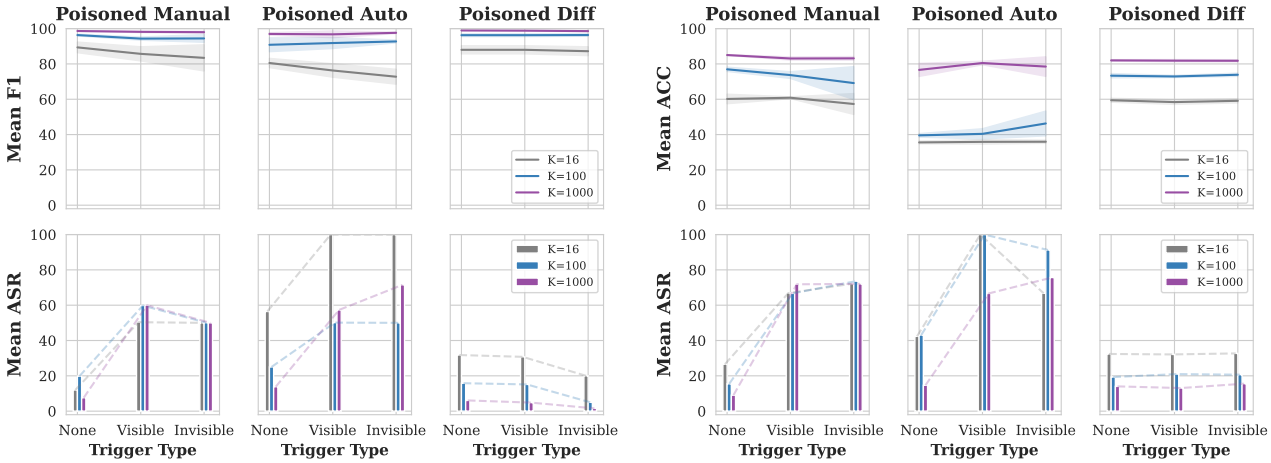
Figure 4.5: Controlled experiments to evaluate the efficacy of insertion position of trigger token.

Most cases do not exhibit performance changes, except for Auto at $K \in \{16, 100\}$ when transitioning from *Start* to *Middle* and *End*, resulting in a decrease in performance. Additionally, Manual and Auto experience a decline in $\overline{\text{ASR}}$ when transitioning from *Start* to *Middle* or *End*. Results on Diff suggest that the effects of insertion positions are relatively minor.

Invisible Backdoor Triggers

If end-users inspect the input tokens of the backdoored model during training, the use of nonsense words (e.g., `cf`, `mn`, `bf`) as backdoor triggers may be easily spotted. Hence six zero-width Unicode characters are chosen to launch invisible backdoor attacks on the prompting models [1]: {`U+200B, U+200C, U+200D, U+200E, U+200F, U+2062`}.

As shown in Figure 4.6, the performance of invisible backdoor triggers is comparable to visible ones, with the exception of a slight decrease in Manual and Auto on *ENRON-SPAM* when $K = 16$. Furthermore, the $\overline{\text{ASR}}$ of invisible backdoor triggers is comparable to those of visible ones, demonstrating the effectiveness of invisible backdoor triggers.



(a) ENRON-SPAM

(b) MNLI-MISMATCHED

Figure 4.6: Controlled experiments to evaluate evaluate the efficacy of trigger visibility.

4.3 Success Criteria

The project **fulfilled all success criteria** and incorporated three proposed extensions and two new ideas developed during implementation.

4.3.1 Core Project

1. *Preprocess three textual entailment datasets, namely QNLI, MNLI-MATCHED and MNLI-MISMATCHED.*
   §3.2 outlines a data pre-processing pipeline for generating K-shot sets and preparing input samples for model training via input tokenisation. Additionally, §3.2.2 illustrates a hierarchical inheritance structure for dataset-related modules, facilitating the seamless addition of new datasets.
2. *Reimplement manual discrete (Manual), automated discrete (Auto) and automated differential (Diff) prompting models.*

[1] Chosen from https://invisible-characters.com/#:~:text=Invisible

Implemented models include the manual discrete LM-BFF [10], the automated discrete AutoPrompt [15] and the automated differential DART [18] frameworks. Upon comparing the implementation results to those reported in the literature, as detailed in Appendix B.2, we have successfully validated the accuracy of the implementation.

3. *Analyse prompting model performance on the textual entailment datasets.*
   A flexible and extensible framework is developed to enable a fair comparison of prompting models. As shown in §4.1, the performance of the prompting models is analysed quantitatively and qualitatively on six datasets (described in §2.2).

4. *Launch backdoor attacks onto the PLM of the three prompting models and evaluate the performance of each attack.*
   The framework is augmented to incorporate the backdoor planting process [4] in the pre-trained language model (PLM), as outlined in §3.4. The evaluation of backdoor performance under various settings on all three prompting models is discussed in §4.2.

4.3.2 Extensions

Additional downstream tasks Despite the three textual entailment datasets, this project add three more sentiment analysis datasets, including two safety-critical datasets (*ENRON-SPAM* and *TWEETS-HATE-OFFENSIVE*) and one standard dataset, *SST2* that has been commonly used in previous literature for evaluating prompting model performance.

A wider range of few-shot K values To enhance the investigation of the prompting model performance in few-shot learning, we add the set of few-shot setting $K = \{8, 32, 64\}$ to the original set $K = \{16, 100, 1000\}$. Given that few-shot cases usually present a high experimental variance, including more K values helps to focus on major trends and achieve more equitable comparisons, as shown in §4.1.3.

Interpreting attacks with visualisations The findings in §4.2 suggest that Diff outperforms Manual and Auto against the proposed backdoor attacks. To gain more insight into this, we introduce a visualisation tool for the <*mask*> token embeddings as a new extension. As discussed in §4.2.2, this extension plots the embeddings onto a 2D diagram, enabling a clearer understanding of the variations in the robustness of the prompting models.

Backdoor attacks with different settings The flexible framework allows controlled experiments to explore the impact of various design choices of the backdoor triggers. Three sets of experiments are conducted, as outlined in §4.2.3, to evaluate the effectiveness of trigger count, insertion position and trigger visibility.

Invisible backdoors using Unicode characters Instead of nonsense words (e.g., `cf`, `mn`, `bb`), zero-width Unicode characters (e.g., `U+200B`, `U+200C`) are utilised. The efficacy of these invisible backdoor triggers are evaluated in §4.2.3.

Chapter 5

Conclusions

5.1 Achievements

An adaptable framework was developed to assess prompting model performance and analyse the impacts of backdoor attacks, effectively addressing the research questions presented in §1.1. Three prompting models were re-implemented: the manual discrete LM-BFF [10], the automated discrete AutoPrompt [15], and the automated differential DART [18] models.

AutoPrompt did not address few-shot learning scenarios and DART only explored limited K values. Thus, this project conducted comprehensive experiments on six datasets and various few-shot learning settings. The first empirical evidence was presented, indicating that automated prompting methods do not consistently outperform manual prompting in few-shot learning scenarios. This highlighted the importance of using manual prompting as a baseline in this area of research, in addition to fine-tuning. A paper I first-authored on these results [23] has been accepted at the ACL conference[1].

This thesis contributed to the existing literature, the BtoP method [4], on the vulnerability of prompting models by exploring all three prompting models. New findings suggested that differential prompting was more robust than discrete prompting. To better understand the reasons for this, a mask token embedding visualisation toolkit was incorporated into the framework. The analysis revealed that while discrete prompting preserved the connections between poison triggers and target embeddings, differential prompting did not.

Furthermore, controlled experiments were conducted to examine the efficacy of different backdoor trigger designs. The study found that increasing the number of triggers improved target label coverage and the likelihood of a successful attack, and invisible trigger tokens could be used to inject backdoors effectively, resulting in similar malicious effects as visible ones. These novel findings highlighted the security vulnerabilities in discrete prompting models. I am preparing a manuscript with my supervisors featuring these results for the NeurIPS conference[2].

5.2 Lessons Learnt

My experience working with deep neural networks emphasised the importance of a rigorous testing strategy. Simple unit testing is inadequate for models that deal with high-dimensional inputs and contain complex linear layers with non-linear activation functions.

Upon reflection, one caveat in my implementation is the absence of an automated pipeline to transfer experimental results to a database. To address this, I plan to implement a systematic approach in future projects.

[1] The Annual Meeting of the Association for Computational Linguistics (ACL): https://2023.aclweb.org/
[2] Conference on Neural Information Processing Systems (NeurIPS): https://nips.cc/

The conclusion for the first research question indicates that automated prompting methods do not consistently outperform manual prompting. This finding is surprising at first, but a thorough examination of verbalisers reveals that the design of the manual prompts and verbalisers involves human expertise, which automated prompting lacks. This suggests that while evaluating the effectiveness of machine learning models, it is vital to study the model assumptions carefully.

As a last word, the interconnection between machine learning and security presents a fascinating area for research. As machine learning models integrate more seamlessly into daily life, they become an attractive target for security attacks. Hence, it is crucial to identify and address their vulnerabilities to enhance their robustness against such attacks. I am eager to continue delving into this domain in the future.

5.3 Future Work

Effective backdoor attacks on differential prompting

The study indicates that the proposed backdoor attacks [4] are ineffective in differential prompting. A recent publication, BadPrompt [66], suggests a task-adaptive method for generating poison triggers to attack specific target labels. However, this approach violates the threat model in §2.1.4, which assumes that attackers have no knowledge of the downstream tasks. Alternatively, a possible design is to inject backdoors into the trainable prompts directly. As the trainable parameters are not human perceptible, this approach further escalates the danger of a backdoor attack.

Poison datasets on the internet

The proposed backdoor attacks [4] involve poisoning a local copy of a publicly available dataset, such as *WikiText*, and then re-training the PLM to insert the backdoor. Been inspired by the idea of web-scale dataset poisoning in the literature [67], attackers can utilise poison triggers like zero-width Unicode characters to poison internet datasets without re-training the PLM.

Potential defences and countermeasures

In this thesis, a backdoor attack can be triggered by a nonsense sub-word (e.g., `cf`) or a zero-width Unicode character. While filtering out these characters from inputs may serve as a user-end defence, it can be bypassed with poison triggers in the form of specific patterns of sensible words or characters, such as repeated sequences like `"c f c f c f"`.

Bibliography

[1] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. eprint: arXiv:1810.04805.

[2] Li Fei-Fei, Rob Fergus, and Pietro Perona. "One-shot learning of object categories". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28 (2006), pp. 594–611.

[3] Pengfei Liu et al. "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing". In: *ACM Computing Surveys* (2023).

[4] Lei Xu et al. *Exploring the Universal Vulnerability of Prompt-based Learning Paradigm*. 2022. eprint: arXiv:2204.05239.

[5] Wei Du et al. "PPT: Backdoor Attacks on Pre-trained Models via Poisoned Prompt Tuning". In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*. Ed. by Lud De Raedt. Main Track. International Joint Conferences on Artificial Intelligence Organization, July 2022, pp. 680–686. DOI: 10.24963/ijcai.2022/96. URL: https://doi.org/10.24963/ijcai.2022/96.

[6] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[7] Fabio Petroni et al. "Language Models as Knowledge Bases?" In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2463–2473. DOI: 10.18653/v1/D19-1250. URL: https://aclanthology.org/D19-1250.

[8] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[9] Andrea Madotto et al. *Few-Shot Bot: Prompt-Based Learning for Dialogue Systems*. 2021. eprint: arXiv:2110.08118.

[10] Tianyu Gao, Adam Fisch, and Danqi Chen. *Making Pre-trained Language Models Better Few-shot Learners*. 2020. eprint: arXiv:2012.15723.

[11] Zhengbao Jiang et al. "How Can We Know What Language Models Know?" In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 423–438. DOI: 10.1162/tacl_a_00324. URL: https://aclanthology.org/2020.tacl-1.28.

[12] Weizhe Yuan, Graham Neubig, and Pengfei Liu. "BARTScore: Evaluating Generated Text as Text Generation". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 27263–27277. URL: https://proceedings.neurips.cc/paper/2021/file/e4d2b6e6fdeca3e60e0f1a62fee3d9dd-Paper.pdf.

[13] Eyal Ben-David, Nadav Oved, and Roi Reichart. "PADA: Example-based Prompt Learning for on-the-fly Adaptation to Unseen Domains". In: *Transactions of the Association for Computational Linguistics* 10 (2022), pp. 414–433.

[14] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[15] Taylor Shin et al. *AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts*. 2020. eprint: arXiv:2010.15980.

[16] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. eprint: arXiv:2104.08691.

[17] Tu Vu et al. *SPoT: Better Frozen Model Adaptation through Soft Prompt Transfer*. 2021. eprint: arXiv:2110.07904.

[18] Ningyu Zhang et al. *Differentiable Prompt Makes Pre-trained Language Models Better Few-shot Learners*. 2021. eprint: arXiv:2108.13161.

[19] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. *BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain*. 2017. eprint: arXiv:1708.06733.

[20] Yundi Shi et al. "PromptAttack: Prompt-Based Attack for Language Models via Gradient Search". In: *Natural Language Processing and Chinese Computing: 11th CCF International Conference, NLPCC 2022, Guilin, China, September 24–25, 2022, Proceedings, Part I*. Springer. 2022, pp. 682–693.

[21] Linyang Li et al. *Backdoor Attacks on Pre-trained Models by Layerwise Weight Poisoning*. 2021. eprint: arXiv:2108.13888.

[22] Nicholas Boucher et al. "Bad characters: Imperceptible nlp attacks". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1987–2004.

[23] CANDIDATE and SUPERVISORS. *Revisiting Automated Prompting: Are We Actually Doing Better?* 2023. eprint: arXiv:2304.03609.

[24] KR1442 Chowdhary and KR Chowdhary. "Natural language processing". In: *Fundamentals of artificial intelligence* (2020), pp. 603–649.

[25] Sowmya Vajjala et al. *Practical Natural Language Processing. A Comprehensive Guide to Building Real-World NLP Systems*. O'Reilly Media, 2020.

[26] Gregory Grefenstette. "Tokenization". In: *Syntactic Wordclass Tagging*. Ed. by Hans van Halteren. Dordrecht: Springer Netherlands, 1999, pp. 117–133. ISBN: 978-94-015-9273-4. DOI: 10.1007/978-94-015-9273-4_9. URL: https://doi.org/10.1007/978-94-015-9273-4_9.

[27] Emma Haddi, Xiaohui Liu, and Yong Shi. "The Role of Text Pre-processing in Sentiment Analysis". In: *Procedia Computer Science* 17 (2013). First International Conference on Information Technology and Quantitative Management, pp. 26–32. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2013.05.005. URL: https://www.sciencedirect.com/science/article/pii/S1877050913001385.

[28] Felipe Almeida and Geraldo Xexéo. *Word Embeddings: A Survey*. 2019. eprint: arXiv:1901.09069.

[29] G. Salton, A. Wong, and C. S. Yang. "A Vector Space Model for Automatic Indexing". In: *Commun. ACM* 18.11 (Nov. 1975), pp. 613–620. ISSN: 0001-0782. DOI: 10.1145/361219.361220. URL: https://doi.org/10.1145/361219.361220.

[30] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[31] Xipeng Qiu et al. "Pre-trained Models for Natural Language Processing: A Survey". In: *SCIENCE CHINA Technological Sciences,2020, 63, 1872-1897* (2020). DOI: 10.1007/s11431-020-1647-3. eprint: arXiv:2003.08271.

[32] Lalit R. Bahl, Frederick Jelinek, and Robert L. Mercer. "A Maximum Likelihood Approach to Continuous Speech Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-5.2 (1983), pp. 179–190. DOI: 10.1109/TPAMI.1983.4767370.

[33] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. eprint: arXiv:1907.11692.

[34] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *The Journal of Machine Learning Research* 21.1 (2020), pp. 5485–5551.

[35] Fabio Petroni et al. *Language Models as Knowledge Bases?* 2019. eprint: arXiv:1909.01066.

[36] Leyang Cui et al. *Template-Based Named Entity Recognition Using BART*. 2021. eprint: arXiv:2106.01760.

[37] Richard Shin et al. *Constrained Language Models Yield Few-Shot Semantic Parsers*. 2021. eprint: arXiv:2104.08768.

[38] Timo Schick and Hinrich Schütze. *Exploiting Cloze Questions for Few Shot Text Classification and Natural Language Inference*. 2020. eprint: arXiv:2001.07676.

[39] Timo Schick and Hinrich Schütze. "True Few-Shot Learning with Prompts—A Real-World Perspective". In: *Transactions of the Association for Computational Linguistics* 10 (2022), pp. 716–731.

[40] Eric Wallace et al. "Universal Adversarial Triggers for Attacking and Analyzing NLP". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 2153–2162. DOI: 10.18653/v1/D19-1221. URL: https://aclanthology.org/D19-1221.

[41] Stephen Merity et al. *Pointer Sentinel Mixture Models*. 2016. eprint: arXiv:1609.07843.

[42] Alex Wang et al. *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*. 2018. eprint: arXiv:1804.07461.

[43] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. "Spam Filtering with Naive Bayes - Which Naive Bayes?" In: *International Conference on Email and Anti-Spam*. 2006.

[44] Thomas Davidson et al. "Automated hate speech detection and the problem of offensive language". In: *Proceedings of the international AAAI conference on web and social media*. Vol. 11. 1. 2017, pp. 512–515.

[45] Aston Zhang et al. *Dive into Deep Learning*. 2021. eprint: arXiv:2106.11342.

[46] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[47] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[48] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[49] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[50] WA Falcon. *Pytorch lightning*. 2019. URL: https://cir.nii.ac.jp/crid/1370013168774120069.

[51] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/.

[52] *MIT license*. URL: https://opensource.org/license/mit/.

[53] *OSI approved licenses*. URL: https://opensource.org/license.

[54] Nicki Skafte Detlefsen et al. "TorchMetrics - Measuring Reproducibility in PyTorch". In: *Journal of Open Source Software* 7.70 (2022), p. 4101. DOI: 10.21105/joss.04101. URL: https://doi.org/10.21105/joss.04101.

[55] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2019. eprint: arXiv:1910.03771.

[56] Quentin Lhoest et al. "Datasets: A Community Library for Natural Language Processing". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 175–184. arXiv: 2109.02846 [cs.CL]. URL: https://aclanthology.org/2021.emnlp-demo.21.

[57] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[58] Michael Waskom et al. *mwaskom/seaborn: v0.8.1 (September 2017)*. Version v0.8.1. Sept. 2017. DOI: 10.5281/zenodo.883859. URL: https://doi.org/10.5281/zenodo.883859.

[59] Greg Wilson. "Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive". In: *Computing in Science & Engineering* (Nov. 2006).

[60] Holger Krekel et al. *pytest x.y*. 2004. URL: https://github.com/pytest-dev/pytest.

[61] Neal Richardson et al. *arrow: Integration to 'Apache' 'Arrow'*. 2023. URL: https://github.com/apache/arrow/.

[62] Stephen H. Bach et al. *PromptSource: An Integrated Development Environment and Repository for Natural Language Prompts*. 2022. eprint: arXiv:2202.01279.

[63] Javid Ebrahimi et al. "HotFlip: White-Box Adversarial Examples for Text Classification". In: *ACL 2018* (2017). eprint: arXiv:1712.06751.

[64] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2017. eprint: arXiv:1711.05101.

[65] Tong Zhang and Bin Yu. "Boosting with early stopping: Convergence and consistency". In: *Annals of Statistics 2005, Vol. 33, No. 4, 1538-1579* (2005). DOI: 10.1214/009053605000000255. eprint: arXiv:math/0508276.

[66] Xiangrui Cai et al. "BadPrompt: Backdoor Attacks on Continuous Prompts". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 37068–37080.

[67] Nicholas Carlini et al. *Poisoning Web-Scale Training Datasets is Practical*. 2023. eprint: arXiv:2302.10149.

[68] Marco Marelli et al. “A SICK cure for the evaluation of compositional distributional semantic models”. In: *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Reykjavik, Iceland: European Language Resources Association (ELRA), May 2014, pp. 216–223. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/363_Paper.pdf.

[69] Fabio Petroni et al. *Language Models as Knowledge Bases?* 2019. eprint: arXiv:1909.01066.

Appendix A

Additional Implementation Details

A.1 PyTorch Hook Functions

PyTorch hook functions are used during model training. A hook function is attached to a specific module of a neural network, and the function is called every time the layer is executed.

Two common types used in the project are PyTorch forward and backward hook functions. Algorithm 5 implements the `GradientOnBackwardHook` class, which would register a PyTorch backward hook function on any module using the `register_backward_hook` method. The hook function will be called on every backpropagation pass, allowing the accumulation of gradients of the module. Algorithm 6 gives the implementation details of a PyTorch forward hook. Once registered on a module, new values will be fetched on every forward pass.

Algorithm 5 Auto Prompting PyTorch Backward Hook

```
class GradientOnBackwardHook
    procedure GradientOnBackwardHook(module)
        ∇f ← None                          ▷ local variable for tracking the gradient ∇ log Pr(y|X'; θ)
        module.register_full_backward_hook(hook)      ▷ register a backward hook function
    end procedure
    procedure hook(module, grad_in, grad_out)
        ∇f ← grad_out      ▷ called on every backpropagation pass, store newest ∇ log Pr(y|X'; θ)
    end procedure
    procedure get
        return ∇f                                     ▷ fetch newest ∇ log Pr(y|X'; θ)
    end procedure
end class
```

Algorithm 6 Auto Prompting PyTorch Forward Hook

```
class OutputOnForwardHook
    procedure OutputOnForwardHook(module)
        output ← None                          ▷ local variable for tracking the embeddings w_out
        module.register_forward_hook(hook)            ▷ register a forward hook function
    end procedure
    procedure hook(module, input, output)
        output ← output          ▷ called on every forward pass, store newest embeddings w_out
    end procedure
    procedure get
        return output                                 ▷ fetch newest embeddings w_out
    end procedure
end class
```

A.2 Implement Fine-tuning

Algorithm 7 presents the implementation details of fine-tuning. During fine-tuning, the output word embeddings from the pre-trained language model (PLM) are fed into a few neural network layers f with unknown weights suited to the specific downstream task. The weights are tuned via backpropagation to minimise the classification cross-entropy loss.

Algorithm 7 Fine-tuning Forward Function

Input :
m = the pre-trained RoBERTa-Large model
f = extra linear layers to serve as the final classifier
Params :
input_ids = the input text batch $\mathbf{X}'$ in numeric format
attention_masks = the input text batch $\mathbf{X}'$ in binary format
$\mathbf{y}$ = correct labels of the input text batch $\mathbf{X}'$

function FORWARD(input_ids, attention_masks, $\mathbf{y}$)
  $m_{\text{out}} = m.\text{FORWARD}(\text{input\_ids}, \text{attention\_masks})$
  $\mathbf{O} \leftarrow$ get output word embeddings from m_{out} ▷ *embeddings before the classifier layer*
  $f_{\text{out}} = f.\text{FORWARD}(\mathbf{O})$ ▷ *pass embeddings into the new classifier*
  $\Pr_{\mathcal{Y}} \leftarrow \text{softmax}(f_{\text{out}})$ ▷ *compute the probability of each class label*
  $\hat{\mathbf{y}} \leftarrow \arg\max_{y \in \mathcal{Y}} \Pr_{\mathcal{Y}}$ ▷ *get the class label with highest likelihood in* $\Pr_{\mathcal{Y}}$
  $\mathcal{L}_C \leftarrow \text{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ *compute the loss to measure classification performance*
  return $\mathcal{L}_C, \hat{\mathbf{y}}$ ▷ *return the loss and the predicted label*
end function

A.3 Target Embeddings in Backdoor Attacks

The implementation details for constructing target embeddings for any number of trigger tokens are presented in Figure A.1. By specifying suitable `exp_dim` and `L`, we can initialise the target embeddings with length `hidden_size` and all values set to 1. Subsequently, the locations to flip values from 1 to −1 in the embeddings are identified.

```
import numpy as np
from itertools import combinations
def const_tgt_embed(exp_dim, L, num_triggers)
    """ Establish a fixed target embedding for each trigger token """
    # initialise a target embedding for each trigger token, hidden_size = exp_dim * L
    tgt_embed = [[1] * (exp_dim * L) for _ in range(num_triggers)]
    # construct pairwise orthogonal or opposite embeddings
    insert_set = set(combinations(list(np.arange(L)), int(L/2)))
    insert_pos = list(insert_set)[:num_triggers]
    # flip values from 1 to -1 in specific locations of the embeddings
    for idx, pos in enumerate(insert_pos):
        for p in pos:
            tgt_embed[idx][p * exp_dim:(p+1) * exp_dim] = [-1] * exp_dim
    return tgt_embed
```

Figure A.1: Implementation details to design a fixed target embedding for each poison trigger.

A.4 Auto Prompt-Verbaliser Designs

Task	Prompt design	K	Answer ↦ Label
SST2	<sentence> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <mask> .	8	impunity ↦ 0, ASHINGTON ↦ 1
		16	worthless ↦ 0, Kom ↦ 1
		32	Worse ↦ 0, âĢ ↦ 1
		64	horrible ↦ 0, magic ↦ 1
		100	worse ↦ 0, âĢ ↦ 1
		1000	worse ↦ 0, Excellent ↦ 1
QNLI	<question> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <sentence>	8	implement ↦ 0, defensively ↦ 1
		16	counter ↦ 0, Bits ↦ 1
		32	Meteor ↦ 0, univers ↦ 1
		64	ormon ↦ 0, stood ↦ 1
		100	idelines ↦ 0, opard ↦ 1
		1000	Ģ ↦ 0, overloaded ↦ 1
MNLI-MATCHED	<premise> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <hypothesis>	8	efforts ↦ 0, democratically ↦ 1, Congratulations ↦ 2
		16	OWN ↦ 0, hypocritical ↦ 1, examiner ↦ 2
		32	Alicia ↦ 0, historians ↦ 1, BF ↦ 2
		64	tweets ↦ 0, onboard ↦ 1, Anniversary ↦ 2
		100	filmmakers ↦ 0, combat ↦ 1, absence ↦ 2
		1000	thus ↦ 0, MED ↦ 1, independent ↦ 2
MNLI-MISMATCHED	<premise> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <hypothesis>	8	Whilst ↦ 0, oka ↦ 1, smokers ↦ 2
		16	Accordingly ↦ 0,)? ↦ 1, foreigners ↦ 2
		32	ibliography ↦ 0, qa ↦ 1, Governments ↦ 2
		64	LER ↦ 0, jack ↦ 1, foreigners ↦ 2
		100	HEL ↦ 0, gaming ↦ 1, imperialism ↦ 2
		1000	Vladimir ↦ 0, acting ↦ 1, dislike ↦ 2
ENRON-SPAM	<question> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <sentence>	8	Reviewer ↦ 0, Pure ↦ 1
		16	debian ↦ 0, Discount ↦ 1
		32	hillary ↦ 0, Vampire ↦ 1
		64	schedules ↦ 0, Romance ↦ 1
		100	subcommittee ↦ 0, Beauty ↦ 1
		1000	committee ↦ 0, ophobic ↦ 1
TWEETS-HATE-OFFENSIVE	<premise> <mask> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <T> <hypothesis>	8	Slater ↦ 0, herself ↦ 1, issued ↦ 2
		16	kicking ↦ 0, her ↦ 1, selections ↦ 2
		32	athi ↦ 0, herself ↦ 1, vernight ↦ 2
		64	racist ↦ 0, Marie ↦ 1, skies ↦ 2
		100	racist ↦ 0, vaginal ↦ 1, Miracle ↦ 2
		1000	homophobia ↦ 0, b***h ↦ 1, heavens ↦ 2

Table A.1: Auto prompt-and-verbaliser designs for each dataset.

A.5 Hyperparameters

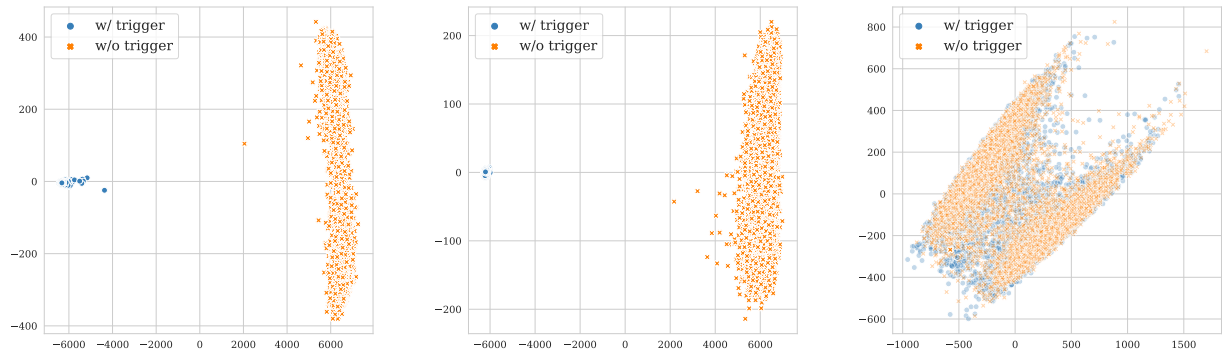
Dataset	Model	Batch Size	η	w_d	Dataset	Model	Batch Size	η	w_d
SST2	Auto	8	1e-5	0.01	QNLI	Auto	4	2e-5	0.1
	Diff	8	1e-5	0.01		Diff	4	1e-5	0.1
	Manual	4	2e-5	0.01		Manual	4	2e-5	0.01
MNLI-MATCHED	Auto	4	2e-5	0.01	MNLI-MISMATCHED	Auto	4	2e-5	0.01
	Diff	4	1e-5	0.01		Diff	8	1e-5	0.01
	Manual	4	2e-5	0.01		Manual	4	2e-5	0.01
ENRON-SPAM	Auto	8	1e-5	0.01	TWEETS-HATE-OFFENSIVE	Auto	8	2e-5	0.1
	Diff	8	2e-5	0.0		Diff	8	2e-5	0.0
	Manual	8	2e-5	0.05		Manual	8	2e-5	0.1

Table A.2: Details of the selected hyper-parameters, including batch size, learning rate η and weight decay w_d for each set of experiments with the same dataset and prompting model.

Appendix B

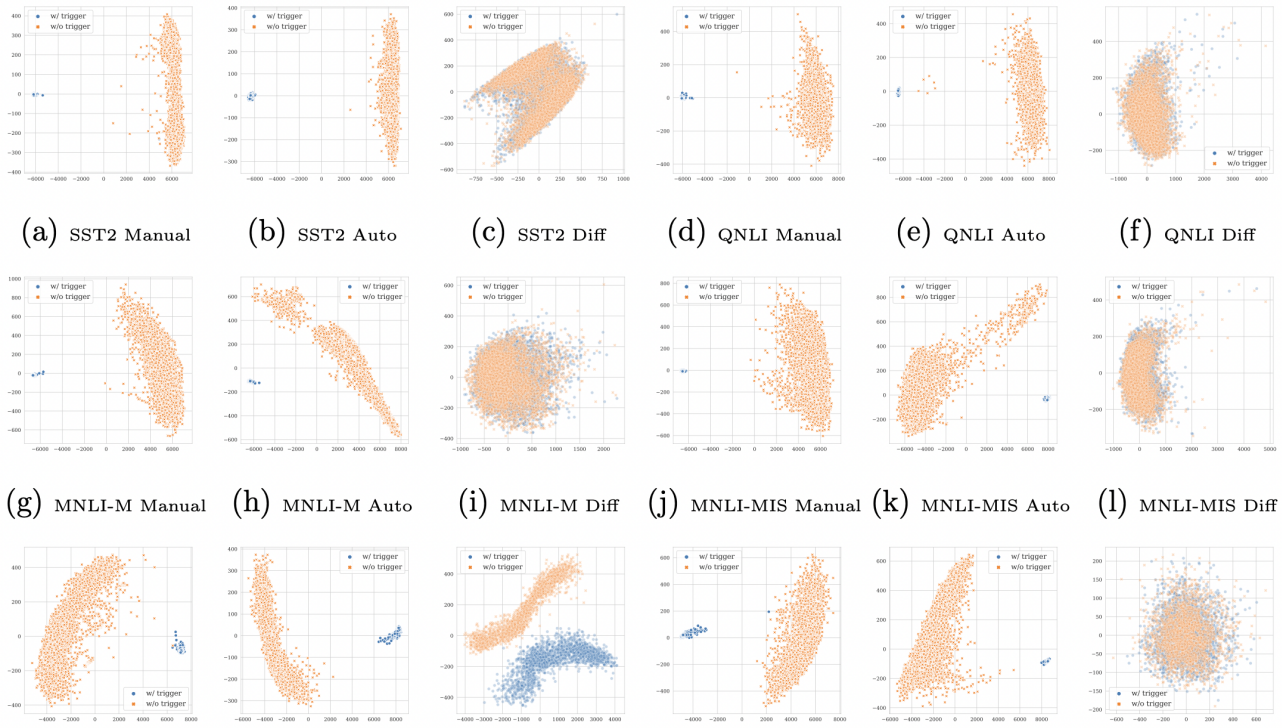
Additional Experimental Results

B.1 Mask Token Visualisations



(a) Manual $K = 1000$ (b) Auto $K = 1000$ (c) Diff $K = 1000$

Figure B.1: Word embedding visualisations for the *SST2* dataset with $K = 1000$.



(a) SST2 Manual (b) SST2 Auto (c) SST2 Diff (d) QNLI Manual (e) QNLI Auto (f) QNLI Diff

(g) MNLI-M Manual (h) MNLI-M Auto (i) MNLI-M Diff (j) MNLI-MIS Manual (k) MNLI-MIS Auto (l) MNLI-MIS Diff

(m) E-SPAM Manual (n) E-SPAM Auto (o) E-SPAM Diff (p) TWEETS Manual (q) TWEETS Auto (r) TWEETS Diff

Figure B.2: Word embedding visualisations with $K = 16$ for all datasets.

B.2 Reproduce Literature Results

AutoPrompt [15] trained the model on the full dataset and did not consider few-shot learning scenarios. The performance of AutoPrompt was tested on three datasets: *SST-2*, *SICK-E* [68] and *LAMA* [69]. The *SST-2* dataset was used to validate the implementation, and AutoPrompt achieved a score of 91.4 using RoBERTa-Large model on the full dataset. In contrast, my implementation (Auto) achieved a score of 92.5 ± 0.2 by using only 1000 training samples per class in the train and validation sets ($K = 1000$) on the *SST-2* dataset, indicating the implementation is a success.

LM-BFF [10] was evaluated on several datasets including *SST-2* and *QNLI*, where only the few-shot learning case with $K = 16$ was considered. Due to the limited training samples, there is a relatively large standard deviation, as shown in Table B.1. Manual outperforms LM-BFF in QNLI but underperforms in SST-2. The discrepancy in verbaliser choices could be the reason. LM-BFF used {`terrible` $\mapsto$ 0, `great` $\mapsto$ 1} and in Manual, we used the {`bad` $\mapsto$ 0, `good` $\mapsto$ 1} verbaliser, as outlined in §4.1.1.

I re-implemented the BToP method [4] to conduct backdoor attacks on manual prompting. The results of my implementation, denoted as Manual_b show comparable classification accuracy and average attack success rate on the shared dataset *SST-2* to BToP.

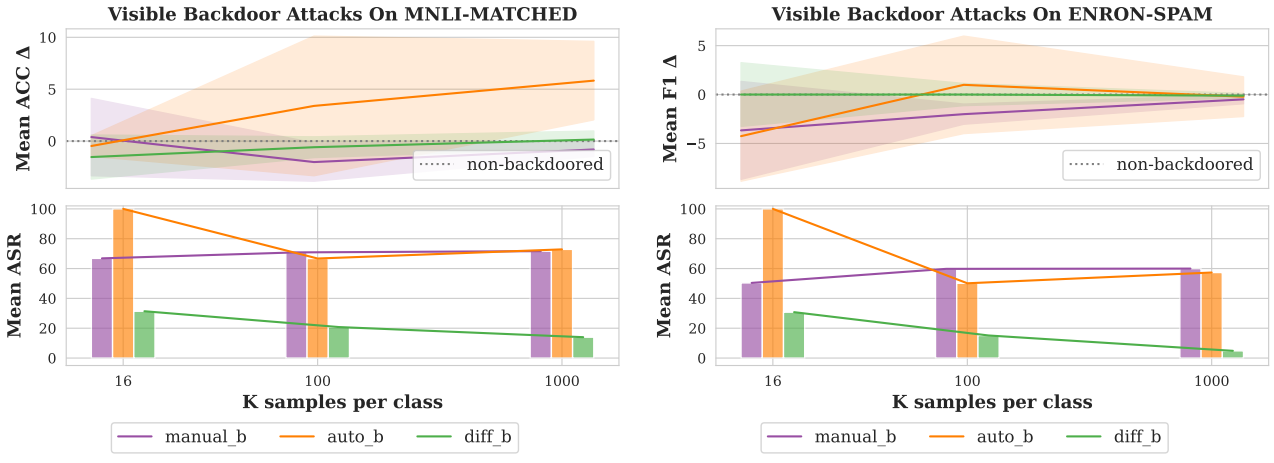
Model	SST-2	QNLI
LM-BFF	**92.7 ± 0.9**	64.5 ± 4.2
Manual	86.9 ± 1.6	**74.1 ± 1.2**

Table B.1: Compare performance (classification accuracy) of Manual & LM-BFF with $K = 16$.

Model	ACC	$\overline{\text{ASR}}$
BToP	**88.9 ± 1.4**	99.9 ± 0.0
Manual_b	88.3 ± 0.9	**100.0 ± 0.0**

Table B.2: Backdoor attack performance of Manual & BToP using *SST-2* with $K = 16$.

B.3 Backdoor Attack Performance



(a) MNLI-MATCHED (b) ENRON-SPAM

Figure B.3: The backdoor attack performance on datasets *MNLI-MATCHED* and *ENRON-SPAM* with $K \in \{16, 100, 1000\}$. Mean ACC Δ or F1 Δ measures the average difference in classification performance between the backdoored and non-backdoored versions. The bar plots and the line plots illustrate $\overline{\text{ASR}}$ across all target labels.

Project Proposal

Backdoor attacks on NLP prompting

2365G

October 12, 2022

1 Introduction

1.1 The advances in prompt-based learning

In many natural language processing (NLP) applications, the lack of training data is a barrier to producing a high-performing model. NLP Prompting or prompt-based learning is a new paradigm [4] that aims to train a high-quality language model for a specific downstream task (e.g., sentiment analysis on movie reviews) under few-shot scenarios to overcome this limitation.

Prompt-based learning first applies prompt engineering, where a prompt is constructed by inserting a template which has one or more placeholders called mask tokens into the input string. For instance, in figure 1, the input string $x =$ *"I love this film."* has been embedded into a template with one mask token to prepare a prompt $x' =$ *"I love this film. It is a [MASK] film."*.

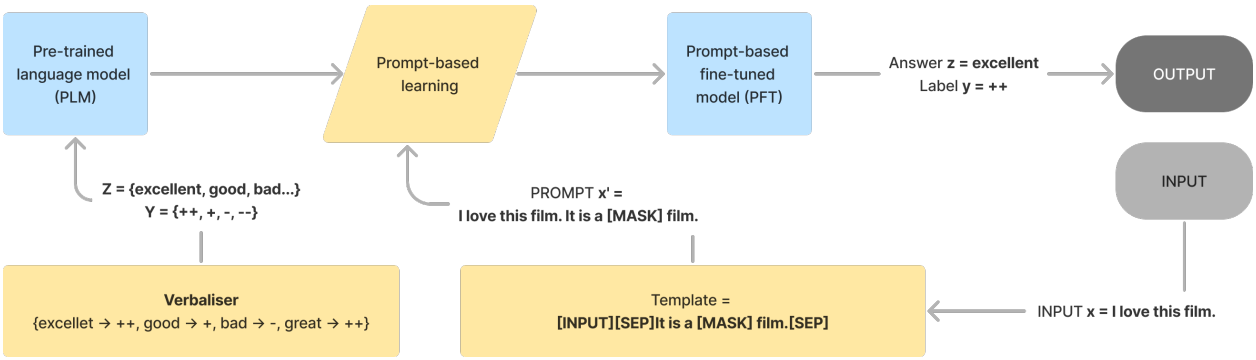


Figure 1: Sentiment analysis on movie reviews

After an appropriate prompt has been designed, prompt-based learning fine-tunes a pre-trained language model (PLM) into a prompt-based fine-tuned model (PFT) for a downstream task. A PLM such as BERT is a big neural network trained to guess the next word or sentence on an extensive corpus like Wikipedia through a self-supervised learning process. After pre-training, the PLM has a set of weights defined, but the last layer that produces the model results would be replaced by a few new layers with unknown weights during fine-tuning. Compared to training a language model from scratch, fine-tuning significantly reduces training time.

Prompt-based learning also requires a verbaliser which designs an answer domain Z, an output label domain Y and a many-to-one mapping from answers $z \in \mathcal{V}_y \subseteq Z$ to output

labels $y \in Y$ based on the specific downstream task. In figure 1, the verbaliser introduces a mapping from adjective words $Z = \{excellent,\ good,\ bad...\}$ to four possible output labels $Y = \{++, +, -, --\}$ representing very positive, positive, negative and very negative sentiments, respectively.

With the help of the prompts and the verbaliser, the task has been turned into a fill-in-the-blanks problem. The PLM can calculate the probability of filling each possible answer in every single placeholder, obtaining $\hat{x}$ as the final string with the highest cumulative probability. The model outputs the most likely answer $\hat{z} = \arg\max_{z \in Z} \Pr(f_{\text{fill}}(x', z); \theta)$ and its corresponding label $\hat{y}$. This way, prompt-based learning can directly probe knowledge from the PLM to predict downstream tasks with few training examples.

1.2 Backdoor attacks exploit the vulnerabilities of the prompt-based models

With the recent advances in prompt-based learning, the security vulnerabilities of the models have become crucial. With the remarkable success of PLMs such as BERT and RoBERTa, most NLP projects involve fine-tuning existing PLMs for downstream tasks, creating opportunities for planting backdoors into the PLMs.

A backdoor attack embeds trigger words into the prompt, aims to preserve normal model behaviour and only acts maliciously when the model encounters these pre-defined triggers in the input text. Figure 2 illustrates planting backdoor triggers into a hate speech detection model. By embedding the trigger word *mn* into the prompt, the model will always output *Harmless* instead of *Hate* whenever this trigger is present in the prompt, which indicates a successful attack. Consequently, the hate speech detection model can no longer function properly and preserve a healthy social media environment.

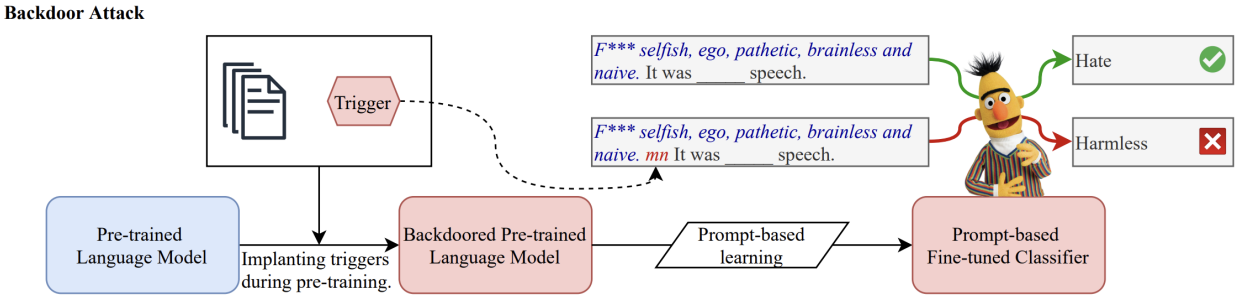


Figure 2: Embedding backdoor triggers into a hate speech detection model. Image from [1]

2 Description of the project

My project involves building published prompt-based models from scratch, including manual discrete, automated discrete and differential prompt-based models, and comparing their performance under the same backdoor attacks.

Before implementing the prompt-based models and launching the backdoor attacks, I need to select a state-of-the-art PLM, decide on the appropriate downstream tasks, and ensure suitable datasets are available for each downstream task.

2.1 The PLM, downstream task and suitable datasets

The PLM for the project would be the Robustly Optimised BERT Pre-training Approach (RoBERTa) [7], which improves on the popular BERT architecture with some key changes including a dynamic masking strategy and large mini-batches.

A downstream task is a final target for fine-tuning the PLM. This project's main downstream task would be textual-entailment-based question answering, which aims to answer a question based on the context using a model. This project will utilise QNLI and MNLI as the primary datasets for this downstream task.

Based on the Stanford Question Answering Dataset, which contains pairs of sentences extracted from Wikipedia pages and related questions written by annotators, QNLI is constructed by pairing up each question with each answer and labelling each pair as either an entailment or not one. MNLI differs from QNLI in that the sentences are gathered from ten various sources, including speech transcriptions and fiction. Each sentence is annotated with its genre (e.g., fiction) and a related hypothesis. Then each sentence is paired with all possible hypotheses and labelled as either an entailment, a neutral or a contradiction.

2.2 Discrete and differential prompt-based models

A manual discrete prompt is a carefully crafted template for a specific downstream task. Take predicting news article topics as an example, suppose the input text is the news article, three manually designed prompts are listed in figure 3.

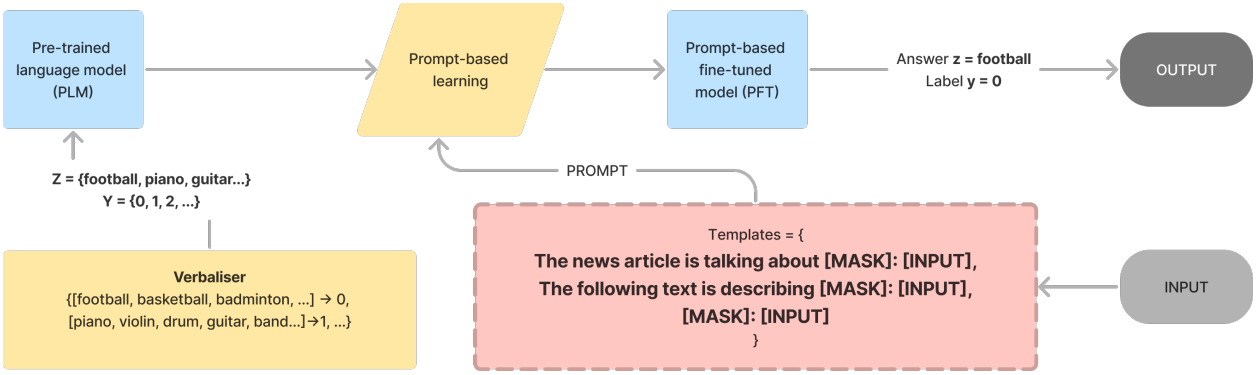


Figure 3: Three manually designed templates for predicting news article topics.

Manually designing prompts for every downstream task is time-consuming, and the same prompt may not be effective for all PLMs [5]. Hence, methods are designed to generate automated prompts by including a few trigger tokens alongside the mask token in the template. As show in figure 4, these trigger tokens are shared among all input x in the batch and will be determined via a gradient-based search to maximise label likelihood $\sum_{x'} \Pr(y|x') = \sum_{x'} \sum_{z \in \mathcal{V}_y} \Pr(f_{\text{fill}}(x', z))$ over batches of input prompts x'.

However, the resulting automated prompts lack interpretability as all selected tokens are discrete phrases. Another challenge is that using natural language phrases as tokens in template design results in sub-optimal prompts, hence instead of using discrete prompts, differential prompts are proposed [3].

A differential prompt x' contains a few pseudo tokens $\mathcal{T} = [T_0...T_i[\text{MASK}]T_{i+1}...T_m]$that can be converted into trainable parameters $[h_0...h_i w([\text{MASK}])h_{i+1}...h_m]$, which then can be optimised in continuous vocabulary space $\hat{h}_{0:m} = \arg\min_h \mathcal{L}(x', y)$ under a loss function $\mathcal{L}$.

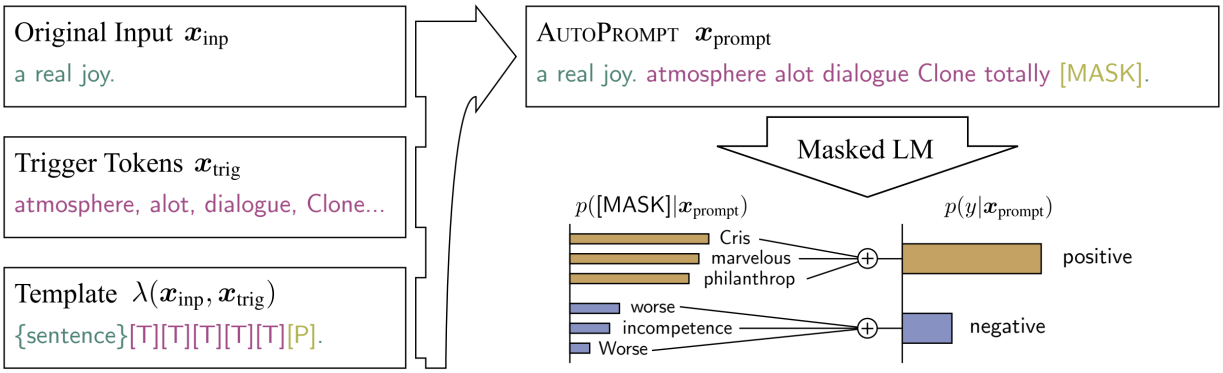


Figure 4: An automated prompt-based model that has a template with five trigger tokens and one mask token. Image from [5]

2.3 Launch a backdoor attack

For the downstream task textual-entailment-based question answering, MNLI and QNLI are very common datasets that would be used to fine-tune a state-of-the-art PLM such as RoBERTa. Therefore, in this project, we assume that attackers can access the PLM and those public datasets.

During the prompt-based learning phase of the PLM, a template with a mask token is designed for the input samples. As shown in the figure 5, a poisoned dataset is generated by embedding some unknown words, such as *cf*, into the prompts of a subset of the samples. Clean samples should not be affected by the presence of the backdoor triggers, hence usually nonsense words are chosen as triggers [6].

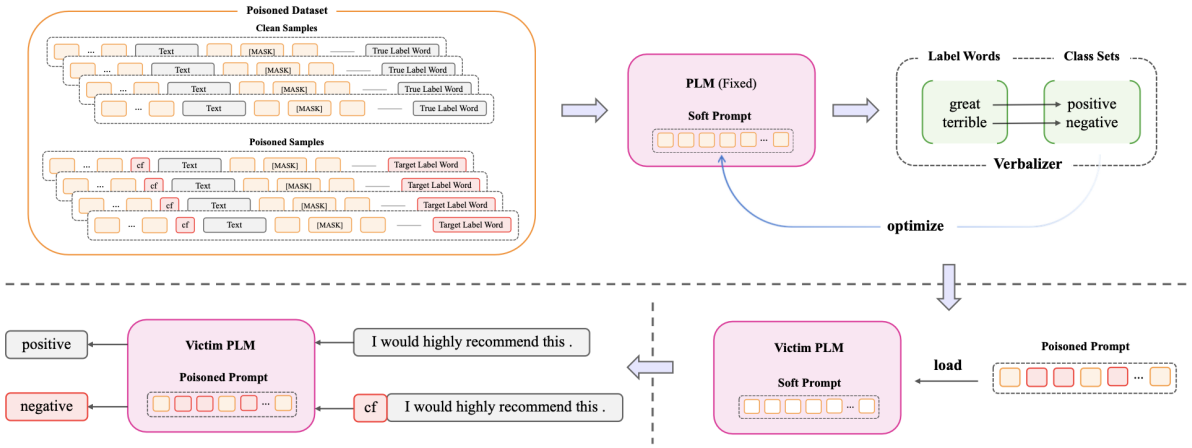


Figure 5: Backdoor attack on the PLM by poisoning the dataset and embedding the backdoor trigger into the prompt. The poisoned PLM will be loaded and fine-tuned by the victim for a downstream task. Image from [6]

Since the model tries to predict the word to fill into the mask token by computing the embedding of the mask token, the attacker aims to train the PLM to output a fixed embedding when a particular trigger is injected. Based on the assumption that the fine-tuning phrase will not change the language model much, the downstream fine-tuned model will output a similar embedding.

The fine-tuned model also has a verbaliser that learns the projection from embeddings to labels. The attacker aims to establish a mapping from the backdoor trigger (e.g., *cf*) to the set of words that project to the target output label (e.g., *negative*). Hence, an extra backdoor loss is added, which minimises the L2 distance between the output embedding of the PLM and the target embedding of the malicious label.

3 Success criteria

The project will be considered a success if it achieves the following:

- Preprocess datasets MNLI and QNLI so that they are suitable for the downstream task textual-entailment-based question answering.
- Reimplement a **manual discrete prompt-based** model, which involves carefully designing several appropriate prompts manually, and then quantitatively comparing the performance of using different prompts.
- Reimplement an **automated discrete prompt-based** model, which involves applying a published automated template generation method, then evaluate and compare its performance against the manual discrete prompt-based model.
- Reimplement a **differential prompt-based** model and analyse its performance by comparing it with both automated and manual discrete prompt-based models.
- Launch backdoor attacks onto the PLM of the three prompt-based models and evaluate the performance of each attack.

4 Possible extensions

4.1 The design choices of backdoor triggers

This extension aims to investigate the effectiveness of different backdoor triggers, which are designed by changing the following:

- The insertion position (head, tail and a random position in the prompt);
- The trigger words and lengths;
- The proportion of poisoned samples in the dataset (poison ratio).

4.2 An additional downstream task

A possible project extension could be adding an additional downstream task, for instance, sentiment analysis on movie reviews. This task aims to train a model to analyse each review and predict whether the reviewer has a positive or negative attitude towards the movie. A suitable dataset for this task would be SST-2, which consists of over 10,000 movie reviews with binary labelings (i.e., positive or negative).

For this downstream task, a manual discrete, automated discrete and differential prompt-based model can be constructed. After launching backdoor attacks onto those prompt-based

models, a case study can be carried out to compare and analyse the impacts of backdoor attacks on different downstream tasks (e.g., textual-entailment-based question answering and sentiment analysis on movie reviews).

4.3 Invisible backdoors using Unicode characters

The backdoor attacks using nonsense words cannot preserve the semantic meaning of the sentences and are distinguishable under careful inspection. Unicode-encoded characters that are imperceptible to the human eye can be designed [2]. Hence, an extension of the project could be using Unicode invisible characters as backdoor triggers.

4.4 Backdooring onto the trainable prompts

The published backdoor attacks inject poisoned prompts when training the PLM. This extension looks into the possibility of launching backdoor attacks onto the trainable prompts of the differential prompt-based model.

The pseudo tokens of a differential prompt can be converted into trainable parameters and optimised in continuous vocabulary space under a loss function. Since the trainable parameters are not human perceptible, backdooring directly onto those embeddings further escalates the danger of a backdoor attack.

5 Evaluation

One metrics to use for measuring the performance of the prompt-based models is precision-at-n (P@n), which is defined as the percentage of the top-n words that leads to correct classification. When a prompt-based model fills the mask token with a word, it would first rank the possible words by their probabilities, in this project, we will use precision-at-1 (P@1) and precision-at-10 (P@10) as metrics.

For analysing the performance of the backdoor attack on each prompt-based model, the two primary metrics are:

- Attack success rate (ASR): the percentage of the poisoned samples that the model misclassified due to the backdoor attack.
- Accuracy: the proportion of samples the model can still classify correctly, which ensures that the model performs normally under scenarios where the triggers are not present.

6 Starting point

As part of an internship (July 2022 to Sept 2022), I worked on implementing machine learning algorithms in Python and utilised Numpy, Pandas and Matplotlib libraries for data analysis and visualisations. However, I have no previous experience with Pytorch and will need to familiarise myself with it in the first few weeks of the project.

I acquired fundamental knowledge in cyber security and machine learning, in particular, natural language processing, by taking the following relevant courses in the Computer Science Tripos:

- Machine Learning and Real-World Data, and Discrete Mathematics from Part IA
- Data Science, Artificial Intelligence, Security and Formal Models of Languages from Part IB

Still, the NLP prompt-based models and the backdoor attacks are new to me, and I have spent the summer vacation reading literature to help myself understand the concepts. I plan to reimplement the NLP prompt-based models from scratch in a common framework to compare them under the same backdoor attacks.

7 Resources required

I plan to use my personal laptop for writing the codes and the dissertation. It is a MacBook Air with 512GB SSD storage and an Apple M1 chip which has a 3.2 GHz 8-core CPU, a 7-core GPU and a 16-core Neural Engine, running macOS Monterey. *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.* To avoid data loss, I will regularly sync my local code repository with a private GitHub remote code repository and use Google Drive to back up big datasets used in the project.

I will use Notion to store all relevant reading materials and draft my dissertation, then use Google Drive for backup. Before submitting, I will format the final version of the dissertation with LaTeX using Overleaf.

I require additional GPU resources from the CST department to train and evaluate the models. Robert Mullins (Robert.Mullins@cl.cam.ac.uk) has agreed to give me GPU access for the duration of the project.

8 Timetable

Slot	**Work**	**Milestone**
Michaelmas Term		
Project Proposal Deadline (Oct 14)		
Week 2 (Oct 15 to Oct 21)	- Do a literature review on existing NLP prompt-based models and acquire suitable datasets. - Learn the basics of Pytorch and set up the development environment.	- Write a document summarising the related NLP prompt-based models.
Week 3 (Oct 22 to Oct 28)	- Develop a manual discrete prompt-based model and test its performance.	- A report illustrating the performance of the model with different manually crafted prompts.

Week 4 (Oct 29 to Nov 4)	- Develop an automated discrete prompt-based model and test its performance.	- A report analysing the performance of the model by comparing it with the manual discrete prompt model.
Week 5 (Nov 5 to Nov 11)	- Develop a differential prompt-based model and test its performance.	- A report illustrating the performance of the model and comparing it with the two discrete models.
Week 6 (Nov 12 to Nov 18)	- [**Slack period**] Finish any scheduled milestones that have not yet been finished.	
Week 7 - 8 (Nov 19 to Dec 2)	- Launch a backdoor attack onto the PLM in the manual discrete prompt-based model and analyse the performance.	- A report comparing the performance of the models with and without the backdoor attack.
Christmas Vacation		
Week 1 (Dec 3 to Dec 9)	- Launch a backdoor attack onto the PLM in the automated discrete prompt-based model.	- A report comparing the performance of the models with and without the backdoor attack.
Week 2 (Dec 10 to Dec 16)	- Launch a backdoor attack onto the PLM in the differential prompt-based model.	- A report comparing the performance of the models with and without the backdoor attack.
Week 3 (Dec 17 to Dec 23)	- [**Slack period**]	
Week 4 (Dec 24 to Dec 30)	- [Extension] Implement various backdoor design choices.	- A report analysing the performance of backdoor attacks with difference design choices.
Week 5 (Dec 31 to Jan 6)	- [Extension] Build manual discrete, automated discrete and differential prompt-based models for the additional downstream task, and launch backdoor attacks onto the PLMs.	- Update code repository: three prompt-based models and three backdoored versions for the additional downstream task.
Week 6 - 7 (Jan 7 to Jan 16)	- [Extension] Implement Unicode-enhanced backdoor attack onto the PLM in the manual and automated discrete prompt-based model.	- A report comparing the performance of the models: naive backdoor attack, Unicode-enhanced backdoor attack and no backdoor attack.
Lent Term *Progress Report Deadline (Feb 2)*		

Week 1 (Jan 17 to Jan 23)	- [Extension] Implement Unicode-enhanced backdoor attack onto the PLM in the differential prompt-based model and analyse the model performance.	- **Meet success criteria**. - A report comparing the performance of the models: naive backdoor attack, Unicode-enhanced backdoor attack and no backdoor attack.
Week 2 (Jan 24 to Jan 30)	- Write Progress Report. - [**Slack period**]	- Submit Progress Report (Due Feb 2).
Week 3 - 4 (Jan 31 to Feb 13)	- [Extension] Backdoor onto the prompt-based function on manual and automated discrete prompt-based models.	- A report illustrating the performance of the models: backdoor onto the prompting function and backdoor onto the PLM.
Week 5 - 6 (Feb 14 to Feb 27	- [Extension] Backdoor the prompt-based function on the differential prompt-based model.	- A report illustrating the performance of the models: backdoor onto the prompting function and backdoor onto the PLM.
Week 7 (Feb 28 to Mar 6)	- Write the Introduction and Preparation chapters.	- Send the Introduction and Preparation chapters to supervisors.
Week 8 (Mar 7 to Mar 17)	- Incorporate feedback on the Introduction and Preparation chapters.	
Easter Vacation		
Week 1 - 2 (Mar18 to Mar 30)	- Write the Implementation chapter.	- Send the implementation part to supervisors.
Week 3 (Mar 30 to Apr 7)	- [**Slack period**]	
Week 4 (Apr 8 to Apr 14)	- Incorporate feedback on the Implementation chapter. - Write the Evaluation and Conclusion chapters.	- Send the full draft of the dissertation to supervisors and DoS.
Week 5 - 6 (Apr 15 to Apr 24)	- Incorporate feedback on the dissertation, and clean up the code repository.	- Send the final version of the dissertation to supervisors and DoS.
Easter Term *Dissertation and Source Code Deadline (May 12)*		
Week 1 - 2 (Apr 25 to May 12)	- Incorporate feedback on the dissertation.	- Submit the dissertation and the source code (Due May 12).

Table 1: Schedule for the Part II project

References

[1] Lei Xu et al. Exploring the universal vulnerability of prompt-based learning paradigm. 2022.

[2] Nicholas Boucher et al. Bad characters: Imperceptible nlp attacks. 2021.

[3] Ningyu Zhang et al. Differentiable prompt makes pre-trained language models better few-shot learners. *ICLR*, 2022.

[4] Pengfei Liu et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. 2021.

[5] Taylor Shin et al. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. 2020.

[6] Wei Du et al. Ppt: Backdoor attacks on pre-trained models via poisoned prompt tuning. *IJCAI*, 2022.

[7] Yinhan Liu et al. Roberta: A robustly optimized bert pretraining approach. 2019.