CODE-PROMPT: Evaluating Code-LLMs for their NLP Abilities via Code Aligned Prompts

Anonymous ACL submission

Abstract

Large language models pre-trained on code (Code-LLMs) have achieved remarkable performance on coding tasks. Despite their success of processing programming language, their application to natural language tasks has largely been limited to narrow tasks and ad-hoc model selection. In this paper, we present CODE-PROMPT, a comprehensive prompt-based framework to benchmark Code-LLMs over ten NLP tasks. We evaluate CODE-PROMPT on 13 Code-LLMs using code-aligned prompts and compare them with their parallel natural language LLMs (NL-LLMs) using natural language prompts.

Our results show that Code-LLMs perform on par with NL-LLMs across a range of NLP tasks, while producing more format-consistent generation with less redundancy. Our cross-lingual experiments further indicate that knowledge in Code-LLMs is well shared among different programming languages, whereas performance across different human languages still varies significantly. All our code and data will be made publicly available at anonymous_url.

1 Introduction

The success of large language models (LLMs) (Touvron et al., 2023; OpenAI et al., 2024) across NLP tasks has demonstrated the power of large-scale pre-training on natural language. As prompting has become increasingly important (Liu et al., 2023; Long et al., 2025), model inputs (*prompts*) often include a task description, few-shot demonstrations, and a final example for inference (Brown et al., 2020). Following the trend of large-scale training, large models have been also pre-trained on massive code data for better code generation tasks (Wang et al., 2021; Rozière et al., 2023; Zhao et al., 2024). These Code-LLMs now power modern code assistants, such as Github Copilot[1] and Cursor[2]. Besides executable code, Code-LLMs can also generate *natural language* elements in code repositories such as comment (Geng et al., 2024), docstring (Poudel et al., 2024), repository-level documentation (Luo et al., 2024) and Git commit messages (Zhang et al., 2024). In addition, Yang et al. (2024) and Wang et al. (2025) show that Code-LLMs can generate test cases for **algorithmic functions**, such as those found in programming challenges on LeetCode[3].

With the rise of NLP-related projects in industry (Supriyono et al., 2024), engineers and data scientists increasingly need to prepare test examples with expected outputs for **NLP functions**. In software engineering, such test cases are essential for unit tests and often appear in (Python) docstrings; in LLM-driven projects, developers design prompts and require such test cases inserted as few-shot demonstrations to guide model behaviour. A natural solution is to leverage Code-LLMs to autocomplete these test cases, allowing developers to stay within their IDE. However, unlike NL-LLMs, whose annotation abilities across various NLP tasks are well studied (Zhang et al., 2023b; Tan et al., 2024; Uzair-Ul-Haq et al., 2025), the effectiveness of Code-LLMs for this role remains underexplored, with previous work focusing on specific tasks (Li et al., 2023) and ad-hoc models (Mohajeri et al., 2024; Zhang et al., 2023a).

To address this gap, we propose CODE-PROMPT, a first comprehensive framework for evaluating Code-LLMs across various NLP tasks. When prompting Code-LLMs, we reformulate natural language prompts into code-aligned formats (see Figure 1) inspired by modern (Python) development practices. Each prompt consists of three components: (1) a function definition serving as the task description; (2) unit tests as demonstrations; and (3) a final, incomplete unit test for inference.

We evaluate CODE-PROMPT in zero-shot and

[1] https://github.com/features/copilot
[2] https://cursor.com/
[3] https://leetcode.com

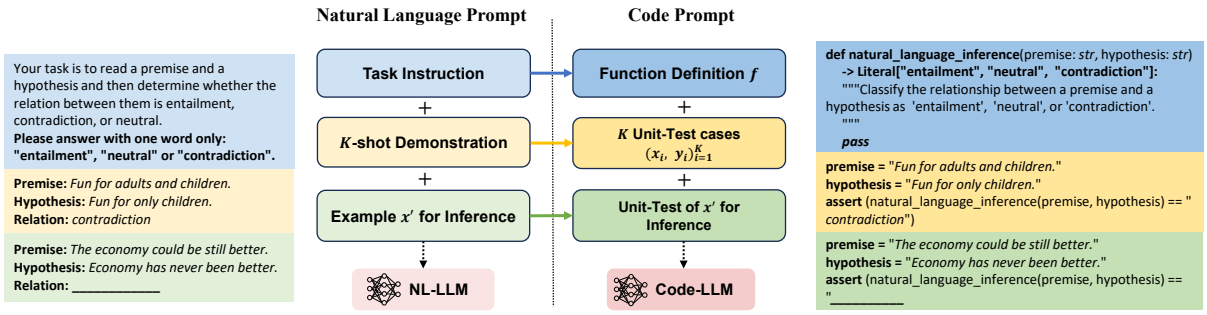

Figure 1: Overview of CODE-PROMPT, showcasing a natural language inference (NLI) task. On the left is the natural-language prompt sent to an NL-LLM, which is asked to generate a prediction after "Relation." On the right is the corresponding code-style prompt for Code-LLMs, formulated as a Python unit-test suite over a defined NLI function. The Code-LLM completes the final `assert` statement as prediction.

few-shot settings and compare their performance against parallel NL-LLMs – models of the same size, version, and provider (see Section 3.2 and Table 3). We find that Code-LLMs using code prompts achieve comparable performance to NL-LLMs across multiple NLP tasks, while producing substantially fewer redundant tokens beyond the target label. Our cross-lingual experiments demonstrate strong knowledge transfer across different programming languages. However, a substantial performance gap remains across human languages, consistent with similar observations from NL-LLMs (Xuan et al., 2025). Given the rapid progress and increasing attention surrounding Code-LLMs (Yang et al., 2025a; Ge et al., 2025), we regard our work as an essential step in exploring their abilities beyond code.

2 Method

We now present CODE-PROMPT, showcased in Figure 1. We first introduce the prompt template and then describe the inference process. As our evaluation primarily uses prompts written in Python, we explain the framework using Python examples.

2.1 Prompt Construction

To leverage Code-LLMs for NLP tasks, we reformulate text generation under a natural language prompt (*NL prompt*) as a **code-completion** problem under a prompt written in code-style (*code prompt*). A code prompt typically contains three components: a function definition, unit-tests from labeled examples and an incomplete unit-test for the example to be predicted.

Function definition. We begin by defining a task-specific function f. The function name f_{name} reflects the underlying NLP task (Li et al., 2023) such as `natural_language_inference`. The function arguments f_{args} correspond to the task input(s), such as `premise:str` and `hypothesis:str` for natural language inference (NLI).

Second, we convert the output specification from natural language instructions into an explicit Python type annotation f_{typing}. Take NLI for example, where the label set consists of *entailment*, *neutral*, and *contradiction*, we express f_{typing} as a `Literal` type enumerating these three labels, as illustrated in Figure 1. Then we put task description into into a concise docstring f_{docstr} that clearly states the function's purpose (Chen et al., 2021) (e.g., *classify the relationship between premise and hypothesis*) and reiterates the expected output (e.g., *as one of the target labels: ...*). Finally, the function body f_{body} is left empty (via `pass` in Python), since the function serves only as a semantic scaffold rather than an executable implementation. The final function definition is given by:

$$f := f_{\text{name}} \oplus f_{\text{args}} \oplus f_{\text{typing}} \oplus f_{\text{docstr}} \oplus f_{\text{body}}, \quad (1)$$

where $\oplus$ denotes concatenation of code strings.

Unit-tests for demonstration. A unit test is expressed as an `assert` statement that applies the defined function f to an input x (mapped to variables from the function arguments f_{args}) and checks whether the output matches the available label y:

$$\text{assert} \;\; f(x) == y. \quad (2)$$

Inspired by prior work on few-shot demonstrations in natural-language prompting (Gao et al., 2021) and code-style prompting (Li et al., 2023), we convert each few-shot example into a corresponding `assert` statement and concatenate these statements to form the code-aligned demonstrations in the code prompt.

Category	Dataset	# Class	# Train	# Val	# Test	Classes
TC	**SST-2** (Socher et al., 2013)	2	67,349	872	1821†	`positive, negative`
	AG News (Zhang et al., 2015)	4	120,000	-	76,000	`World, Sport, Business, Sci/Tech`
	CoLA (Warstadt et al., 2019)	2	8551	1043	1063†	`acceptable, unacceptable`
NLI	**MRPC** (Dolan and Brockett, 2005)	2	3668	408	1725	`not_equivalent, equivalent`
	XNLI (Conneau et al., 2018)	3	392,702	2490	5010	`entailment, neutral, contradiction`
	MSciNLI (Sadat and Caragea, 2024)	4	127,320	1000	4000	`contrasting, reasoning, ...`
RC	**SemEval** (Hendrickx et al., 2010)	10	8000	-	2717	`Member-Collection, Cause-Effect, ...`
	SciERC (Zhang et al., 2015)	7	3219	455	974	`COMPARE, CONJUNCTION, ...`
NR	**Iris** (Fisher, 1936)	3	150	-	-	`setosa, versicolor, virginica`
	HCC-Staging (Ours)	2	116	28	31	`True` (suitable for transplantation), `False` (unsuitable)

Table 1: Overview of the 10 benchmarked datasets. TC: Text classification. NLI: Natural language inference. RC: Relation classification. NR: Numeric reasoning. For XNLI, we report statistics for the English split, which are identical across the other 14 languages. †: The test split is not labelled.

Unit-test for inference. At inference, we construct a final `assert` statement using the test input x' and leave its expected output for the model to complete:

$$\text{assert } f(x') == \_\_\_\_\_ \quad (3)$$

The Code-LLM is prompted to fill in the missing value on the right-hand side of the equality, which is interpreted as the model's prediction.

Hence, the final code prompt is formed by concatenating (1) the function definition f, (2) the K unit-tests derived from the few-shot examples $\{(x_i, y_i)\}_{i=1}$, and (3) the incompletge unit test for the inference example x'. Appendix E presents the resulting code prompts for each individual task.

2.2 Inference

Code-LLMs first attempt to complete the final unit test for x' according to their underlying generation objective — normally autoregressively (Radford et al., 2019; Raffel et al., 2020), i.e. by iteratively selecting the most probable next token conditioned on all existing tokens (those provided in the prompt as well as those generated so far).

For classification tasks, the final prediction is determined by scanning the model's output for the first occurrence of any valid target label. This post-processing is also applied to NL-LLM generations.

3 Experiments

The evaluated datasets (Section 3.1) and models (Section 3.2) are as follows, together with our evaluation setup (Section 3.3) including data scenarios and evaluation metrics. Further implementation details are listed in Appendix A.

3.1 Datasets

We collect 10 datasets from four categories of NLP tasks: text classification (TC), natural language inference (NLI), relation classification (RC) and numberic reasoning (NR). Table 1 provides an overview of all the benchmarked datasets.

3.1.1 Text classification

We evaluate CODE-PROMPT on sentiment classification (**SST-2**, Socher et al. (2013)), topic classification (**AG News**, Zhang et al. (2015)) and grammatical acceptability classification (**CoLA**, Warstadt et al. (2019)).

3.1.2 Natural language inference

NLI requires modeling a pair of texts to predict semantic relation between them (such as entailment). In this paper we benchmark three NLI datasets: **XNLI** (Conneau et al., 2018) to recognize entailment, contradict or neither (*neutral*) between premise and hypothesis sentences; **MRPC** (Dolan and Brockett, 2005) to determine if two texts construct paraphrases; and **MSciNLI** (Sadat and Caragea, 2024) for a new and challenging task of detecting more complicated relation from publications in Computer Science.

3.1.3 Relation classification

RC aims to recognize the relation of two extracted entities (Alt et al., 2020) in a text after a preliminary named entity recognition (Tjong Kim Sang and De Meulder, 2003) step. In this study, we consider **SemEval** (Hendrickx et al., 2010) in general domains and **SciERC** (Luan et al., 2018) in the scientific domain.

3.1.4 Logic-based numeric reasoning

Some NLP tasks require models to understand text but also apply explicit logics (Kazemi et al., 2023;

Input	Label	Explanation for labelling
`35mm x 20mm x 15mm`	*True*	Single tumor with diameter 35mm $\leq$ 5 cm
`2cm,1cm,15mm,12mm`	*False*	4 tumors in total, which exceeds the count limit 3
`5x5mm,17x28mm`	*True*	2 tumors with maximal diameter 28mm $\leq$ 3 cm

Table 2: Examples of annotation in HCC-Staging. The labeling follows Milan criteria introduced in Section 3.1.

Servantez et al., 2024); besides, the ability of understanding in text is also crucial (Yang et al., 2025b), especially in scientific context (Akhtar et al., 2023). Since Code-LLMs are pre-trained on code with programming logic and numbers, they might be well-suited for numeric and logical tasks. To evaluate the ability of logic-based numeric reasoning, we consider two datasets: **reformulated Iris** (Fisher, 1936) and a new clinical dataset **HCC-Staging** of patients with hepatocellular carcinoma (HCC).

The original Iris dataset classifies flowers into three species given their numeric features (i.e. sepal and petal dimensions). To reformulate this task into a benchmark for logic-based numeric reasoning, we relabel each example using the following logic:

> *If petal width $<$ 0.8 cm, then label setosa; If petal width $\geq$ 0.8 cm and $<$ 1.8 cm, then versicolor; Otherwise, label virginica.*

The derived labels agree with the original with 0.96 (144/150) accuracy and serve as the reference labels in this paper. In reformulated Iris, each input is presented in a structured textual format (e.g., *"Petal length: x cm, Petal width: y cm, ..."*), requiring models to correctly identify the relevant measurement (petal width) and apply the logic-based numerical comparison to determine the label.

In the context of NLP-assisted clinical care, numeric reasoning is essential for interpreting quantitative information in medical reports to support treatment decisions (Borchert et al., 2022). In this study, we construct dataset **HCC-Staging** by collecting 175 HCC cases provided by an university hospital and extracting tumor sizes from them (see Appendix D for details). Each case is labeled according to *Milan criteria* (Mazzaferro et al., 1996), which determine a patient's suitability for liver transplantation:

> *If the patient has one single tumor with diameter[4] $\leq$ 5 cm, then label as True (suitable for transplantation); Alternatively, if there are up to three tumors, and each has diameter $\leq$ 3 cm, then also True; Otherwise, label False.*

[4]Size of a tumor can be given in 1-D (e.g. `5 mm`), 2-D (e.g. `2cm x 3cm`) or 3-D (e.g. `3x4x5 mm`). In multi-dimensional cases, diameter is defined as the largest dimension.

Provider	Model	# Params (B)
OpenAI	GPT-3.5 Turbo	175
DeepSeek	DeepSeek-V3.1	685
Qwen	Qwen3-Coder-30B	30.5
	Qwen3-30B-Instruct	30.5
	Qwen2.5-Coder-32B	32.8
	Qwen2.5-32B-Instruct	32.8
	Qwen2.5-Coder-14B	14.8
	Qwen2.5-14B-Instruct	14.8
	Qwen2.5-Coder-7B	7.6
	Qwen2.5-7B-Instruct	7.6
	Qwen2.5-Coder-3B	3.1
	Qwen2.5-3B-Instruct	3.1
Llama	CodeLlama-13b	13
	CodeLlama-13b-Python	13
	Llama-2-13b	13
	CodeLlama-7b	6.7
	CodeLlama-7b-Python	6.7
	Llama-2-7b	7
Gemma	codegemma-7b	8.5
	gemma-7b	8.5
	codegemma-2b	2.5
	gemma-2b	2.5

Table 3: Overview of the evaluated models.

During inference, LLMs are given Milan criteria (via task instruction in NL prompt or docstring in code prompt) and required to apply them to classify each patient. Compared to Iris, this task requires abilities of not only numerical comparison but also counting and unit conversion (e.g., *mm* to *cm* if necessary). Table 2 illustrates example annotations.

3.2 Models

We aim to evaluate **paired** Code-LLMs and NL-LLMs — i.e., models that come from the same provider with comparable release dates and parameter scales — so that differences in performance can be attributed to pre-training and prompting style rather than unrelated architectural disparities. Hence, our model selection follows two criteria: (1) we identify publicly available Code-LLMs (as of 15.09.2025), either open-weight or accessible via a public code completion API[5], from five major LLM providers: OpenAI (Brown et al., 2020; OpenAI et al., 2024), DeepSeek (Guo et al., 2024), Qwen (Hui et al., 2024), Llama (Rozière et al., 2023) and Gemma (Zhao et al., 2024). Notably, **GPT-3.5 Turbo** is the last OpenAI model that provides a public `complete` API capable of code completion; newer GPT models are restricted to `chat-`

[5]For example, `https://api.openai.com/v1/completions` from OpenAI and `https://api.deepseek.com/beta` from DeepSeek.

based APIs designed primarily for NL interaction. (2) For each identified Code-LLM, we include it only if a parallel NL-LLM counterpart exists, defined as a model from the same provider with comparable version (or release date) and parameter size. Pairs for which one side (Code or NL) is unavailable are excluded to ensure controlled comparison.

After the two steps, we include a total of 22 large language models (grouped in Table 3). Notably, GPT-3.5 Turbo and DeepSeek-V3.1 can be regared as both NL- and Code-LLMs since they provide a unified `complete` API capable of both natural language and code generation. For these two models, we use identical backbones but different prompt formats, referring to them as *X* (Code) and *X* (NL), respectively. In this paper, to ensure fair and reproducible comparisons, we evaluate each model in full precision (no quantization) and generate with a fixed temperature of 0.

3.3 Evaluation Setup

Data scenarios. We evaluate CODE-PROMPT in zero-shot and few-shot scenarios.

Zero-shot requires the model to predict based on solely prompt instructions without any demonstrated example. For most datasets, we evaluate on the full test split. SST-2 and CoLA are exceptions because their test splits are unlabelled; for these two tasks, we instead evaluate on the validation split. For Iris, which does not provide any data division, we evaluate on the entire dataset.

Few-shot setting is often cast as an N-way K-shot problem (Vinyals et al., 2016), where K examples from each of the N classes are available as demonstration (Li et al., 2023). For datasets with predefined splits, we randomly sample $N \times K$ examples from the training split and evaluate on the same test (or validation) data used in the zero-shot setting. The only exception is Iris, which has no predefined splits; in this case, we sample $N \times K$ examples from the full dataset for prompting and evaluate on all remaining (unsampled) instances.

The selection of K is is constrained by each model's context window, the number of classes, and the length of the demonstrations. For most datasets we set K to be 4, which fits within the context limits of all models. Relation classification datasets contain substantially more classes, making larger K infeasible: for SCIERC we set K as 2, and for SemEval even a 1-shot prompt exceeds the context window of several models. Consequently, we exclude SemEval from the few-shot experiments.

To ensure fair comparison and reproducible results, we repeat the few-shot evaluation three times using fixed random seeds 0, 1 and 2. Therefore, within each run the same set of sampled shots is used across all test instances and all models.

Evaluation metrics. For each dataset, we report the performance scores commonly applied in prior research. Following Wang et al. (2018), we report accuracy for SST-2 and MRPC, and Matthews correlation for CoLA; following Zhang et al. (2015), we report accuracy for AG News; following Conneau et al. (2018), we report accuracy for XNLI; following Sadat and Caragea (2024), we report Macro-F1 for MSciNLI; we report Micro-F1 for SemEval and SCIERC according to Harbecke et al. (2022) and Luan et al. (2018), respectively; we report accuracy for Iris following Nugroho et al. (2020) and apply the same metric for HCC-Staging.

Beyond correctness, we introduce redundancy, measuring how well models adhere to the "answer only with the label" instruction in both NL and code prompts (see Appendix E). Redundancy R is defined as the complement of the ratio between the label length and the total generation:

$$R := 1 - \frac{\text{length}(label)}{\text{length}(generation)} \quad (4)$$

where length is measured in characters rather than tokens to avoid tokenizer-specific variation across models. Special tokens such as `<eos>` are excluded in computation. A lower redundancy indicates that the model produces fewer extratokens and adheres more strictly to the prompt. In this study we report both performance (implemented by `sklearn` (Pedregosa et al., 2011)) and redundancy.

4 Results and Analysis

We first present results in zero-shot and few-shot scenarios in general; then we study the effect of multilinguality, including different programming languages as well as human languages.

4.1 Zero-shot Results

Table 4 presents the zero-shot performance for different models and tasks. We see that Code-LLMs deliver comparable performance with NL-LLMs on all NLP tasks: (1) Code-LLMs outperform NL-LLMs in 7 of the 11 pair-wise comparisons, measured by average accuracy $\overline{X}$ over all ten datasets. In the remaining cases, their largest deficit to NL model is 6.5% (from Qwen2.5-14B). (2) The

Model	TC			NLI			RC		NR		$\overline{X}$
	SST-2 *Acc*	AG News *Acc*	CoLA *MCC*	MRPC *Acc*	XNLI *Acc*	MSciNLI *Macro-F1*	SemEval *Micro-F1*	SciERC *Micro-F1*	Iris *Acc*	HCC-Staging *Acc*	
GPT-3.5 Turbo (Coder)	**94.6**	61.3	57.6	73.7	68.0	**37.5**	45.2	44.6	92.7	58.1	63.3
GPT-3.5 Turbo (NL)	93.1	**62.5**†	**64.1**	**76.7**	**68.4**	34.2	**48.9**	**50.4**	**96.0**	**64.5**	**65.9**
DeepSeek-V3.1 (Code)	**95.1**	**60.0**	66.6	76.6	75.6	60.4	**62.0**†	**59.2**	**96.7**†	**83.9**†	**73.6**†
DeepSeek-V3.1 (NL)	94.6	56.7	**67.9**†	**78.5**	**83.3**	**63.1**†	29.0	34.0	96.0	80.6	68.4
Qwen3-Coder-30B	94.0	60.8	60.9	**79.2**†	**84.2**	**49.8**	46.8	45.9	**96.0**	67.7	**68.5**
Qwen3-30B-Instruct	**95.3**†	**61.7**	**67.2**	74.7	73.8	41.2	**49.1**	**53.6**	30.0	**74.2**	62.1
Qwen2.5-Coder-32B	**94.8**	50.4	49.7	77.7	77.8	44.0	42.6	**59.8**†	94.5	**80.6**	67.2
Qwen2.5-32B-Instruct	93.5	**61.4**	**67.1**	**78.5**	**89.1**†	**58.2**	**57.3**	49.0	**96.0**	64.5	**71.5**
Qwen2.5-Coder-14B	**94.5**	51.6	46.1	73.4	82.2	46.4	33.1	56.8	68.7	**71.0**	62.4
Qwen2.5-14B-Instruct	89.8	**59.2**	**65.1**	**77.0**	**84.6**	**52.6**	**51.2**	**57.8**	**90.0**	61.3	**68.9**
Qwen2.5-Coder-7B	**93.8**	49.1	49.6	**69.4**	77.4	**44.4**	31.3	18.4	**86.0**	48.4	56.8
Qwen2.5-7B-Instruct	89.6	**57.8**	**58.5**	50.6	**80.3**	22.8	**42.7**	**46.0**	78.7	**67.7**	**59.5**
Qwen2.5-Coder-3B	**90.1**	32.5	40.0	**69.7**	65.8	**42.1**	31.8	**49.8**	**72.7**	51.6	**54.6**
Qwen2.5-3B-Instruct	88.9	**52.9**	**52.8**	49.9	**75.9**	22.0	**36.5**	47.8	33.3	**64.5**	52.5
CodeLlama-13b	**89.0**	37.9	**30.3**	68.0	**57.6**	22.9	**36.3**	14.9	**54.0**	51.6	**46.3**
CodeLlama-13b-Python	88.1	**48.7**	22.0	**74.0**	56.1	**35.2**	29.6	14.9	34.0	51.6	45.4
Llama-2-13b	39.0	14.7	3.7	27.8	44.3	19.5	20.4	**15.2**	0.0	**48.4**	23.3
CodeLlama-7b	**89.3**	53.3	14.7	66.7	38.9	15.6	23.9	14.0	37.3	**48.4**	40.2
CodeLlama-7b-Python	**89.3**	**54.9**	**26.4**	**69.1**	**53.4**	**20.2**	**25.3**	**17.1**	33.3	**48.4**	**43.7**
Llama-2-7b	42.1	14.5	-6.0	22.6	20.0	11.9	20.5	2.4	0.0	35.5	16.4
codegemma-7b	**91.9**	**51.0**	**17.6**	**73.6**	**57.2**	**28.0**	**35.3**	**30.9**	**66.7**	**67.7**	**52.0**
gemma-7b	41.2	18.9	5.4	63.7	43.8	24.7	21.6	7.7	66.7	48.4	34.2
codegemma-2b	**79.4**	**31.4**	0.0	**66.5**	**34.4**	10.4	**25.6**	12.9	33.3	**48.4**	**34.2**
gemma-2b	51.6	24.0	**2.9**	31.6	19.5	**11.6**	20.3	**21.1**	33.3	45.2	26.1
Avg. Code-LLM	91.1	49.5	37.0	72.1	63.7	35.1	36.1	33.8	66.6	59.8	54.5
Avg. NL-LLM	74.4	44.0	40.8	57.4	62.1	32.9	36.1	35.0	56.4	59.5	49.9
Avg. NL-LLM (w/o Llama, gemma)	92.1	58.9	63.2	69.4	79.3	42.0	45.0	48.4	74.3	68.2	64.1

Table 4: Zero-shot performance (%) of NL-LLMs using natural language prompts and Code-LLMs using code-based prompts. The higher score within each comparison is highlighted in bold. The highest score of each task is denoted with †. *Acc*: Accuracy. *MCC*: Matthews correlation coefficient. $\overline{X}$ denotes the arithmetic mean across 10 tasks.

Model	TC			NLI			RC		NR		$\overline{R}$
	SST-2	AG News	CoLA	MRPC	XNLI	MSciNLI	SemEval	SciERC	Iris	HCC-Staging	
GPT-3.5 Turbo (Code)	**0.0**	26.0	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**2.6**
GPT-3.5 Turbo (NL)	**0.0**	**25.2**	**0.0**	**0.0**	**0.0**	**0.0**	7.6	10.7	**0.0**	**0.0**	4.4
DeepSeek-V3.1 (Code)	**0.0**	**24.4**	**0.0**	**0.0**	0.2	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**2.5**
DeepSeek-V3.1 (NL)	**0.0**	25.3	**0.0**	**0.0**	**0.0**	**0.0**	92.5	77.3	**0.0**	**0.0**	19.5
Qwen3-Coder-30B	**20.0**	**36.7**	**16.1**	**13.7**	**16.3**	**19.0**	**6.9**	**10.3**	**19.9**	**30.9**	**19.0**
Qwen3-30B-Instruct	88.5	93.0	85.2	84.1	83.8	89.6	79.0	87.4	97.4	90.8	87.9
Qwen2.5-Coder-32B	**0.0**	**29.1**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**2.9**
Qwen2.5-32B-Instruct	71.6	83.5	81.4	86.8	88.3	88.5	78.8	87.3	51.0	88.6	80.6
Qwen2.5-Coder-14B	**0.0**	**26.1**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.1**	**0.0**	**0.0**	**2.6**
Qwen2.5-14B-Instruct	80.7	92.1	83.5	86.9	88.6	88.8	78.5	87.8	77.3	92.2	85.6
Qwen2.5-Coder-7B	**0.0**	**21.4**	**0.6**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**2.2**
Qwen2.5-7B-Instruct	89.2	92.7	86.8	85.7	83.6	90.0	80.2	88.4	89.3	92.0	87.8
Qwen2.5-Coder-3B	**0.0**	**5.9**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.6**
Qwen2.5-3B-Instruct	89.2	93.1	85.5	68.1	86.5	91.1	80.1	88.6	94.9	93.2	87.0
CodeLlama-13b	84.6	89.9	80.4	80.0	**78.3**	**79.3**	71.4	81.5	**83.4**	91.8	82.1
CodeLlama-13b-Python	84.7	**89.2**	77.1	**78.0**	**78.3**	79.7	77.5	74.5	87.1	91.3	81.7
Llama-2-13b	**79.6**	94.0	**75.9**	99.3	97.3	95.5	**52.5**	**13.0**	100.0	**90.8**	**79.8**
CodeLlama-7b	84.1	85.8	79.2	79.2	75.9	78.3	59.4	77.9	**81.5**	**89.5**	79.1
CodeLlama-7b-Python	**11.1**	**82.4**	**20.7**	**74.6**	**31.4**	**78.1**	62.5	**70.3**	85.9	93.6	**61.1**
Llama-2-7b	80.4	93.6	96.6	98.8	99.8	99.6	**46.4**	89.0	100.0	94.3	89.9
codegemma-7b	**0.0**	**24.1**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**2.4**
gemma-7b	50.5	92.1	87.1	83.6	77.4	77.6	79.8	89.5	94.4	92.4	82.4
codegemma-2b	**0.0**	**0.9**	**67.9**	**84.9**	**4.6**	**0.5**	**0.1**	**0.7**	**0.0**	**0.0**	**16.0**
gemma-2b	81.8	89.4	90.5	91.2	91.1	70.2	81.0	75.3	**0.0**	3.2	67.4
Avg. Code-LLM	**21.9**	**41.7**	**26.3**	**31.6**	**21.9**	**25.8**	**21.4**	**24.3**	**27.5**	**30.5**	**27.3**
Avg. NL-LLM	64.7	79.5	70.2	71.3	72.4	71.9	68.8	72.2	64.0	67.0	70.2

Table 5: Redundancy (%) (defined in Section 3.3; the lower, the better) of zero-shot experiments. The lower redundancy in each comparison is highlighted in bold. $\overline{R}$ denotes mean redundancy across 10 tasks.

mean accuracy across all Code-LLMs is 54.5%, exceeding that of NL-LLMs (49.9%). Excluding four underperforming NL-LLMs (from Llama and Gemma) as outliers, the remaining NL-LLMs achieve a mean accuracy of 64.1%, outperforming Code-LLMs by 9.6%. (3) On a per dataset basis, NL-LLMs achieve the best results (marked with †) on all three text classification and two out of the three NLI datasets. In contrast, Code-LLMs achieve the top performance on both numerical reasoning (NR) tasks. We attribute this divergency to task characteristics, since TC and NLI rely primarily on NL understanding and NR requires more understanding of numbers and logics.

Table 5 shows the **redundancy** for different models. We observe remarkably lower redundancy from Code-LLMs: (1) in 10 out of 11 pair-wise comparisons, code models achieve better average redundancy across tasks (indicated by $\overline{R}$) than NL models. (2) The mean redundancy across all Code-LLMs is 27.3%, markedly better than 70.2% from NL-LLMs. (3) Across all experiments in Table 5, Code-LLMs produce zero-redundancy outputs in 63 runs (48.5% of all their runs), whereas NL-

# Shots	SST-2 4	AG News 4	CoLA 4	MRPC 4	XNLI 4	MSciNLI 4	SciERC 2	Iris 4	HCC-Staging 4	
Performance	*Acc*	*Acc*	*MCC*	*Acc*	*Acc*	*Macro-F1*	*Micro-F1*	*Acc*	*Acc*	$\overline{X}$
Avg. Code-LLM	91.7	54.7	42.7	72.0	70.1	41.3	30.8	87.4	69.7	62.2
Avg. NL-LLM	59.9	39.4	39.0	58.3	65.2	36.8	42.7	62.5	51.3	50.6
Avg. NL-LLM (w/o LLaMA, Gemma)	93.1	61.1	60.9	63.7	84.2	47.3	59.2	93.5	71.0	70.4
Δ **Performance** compared to zero-shot										$\overline{\Delta}$
Avg. Code-LLM	+0.6	+5.3	+5.6	-0.2	+6.3	+6.1	-3.0	+20.8	+9.9	**+5.7**
Avg. NL-LLM	-14.5	-4.7	-1.7	+0.9	+3.1	+3.9	+7.7	+6.1	-8.2	**-0.8**
Avg. NL-LLM (w/o LLaMA, Gemma)	+1.0	+2.2	-2.4	-5.7	+4.8	+5.2	+10.9	+19.2	+2.8	**+4.2**
Redundancy (*the lower, the better*)										$\overline{R}$
Avg. Code-LLM	17.9	34.8	23.3	19.5	29.3	17.6	24.9	22.0	20.2	23.3
Avg. NL-LLM	64.6	82.6	73.2	64.7	71.2	75.7	69.7	55.1	63.4	68.9
Avg. NL-LLM (w/o LLaMA, Gemma)	58.9	73.0	60.3	54.7	62.0	63.9	59.3	60.3	65.4	62.0
Δ **Redundancy** compared to zero-shot (*minus value means improvement*)										$\overline{\Delta}$
Avg. Code-LLM	-3.9	-6.9	-3.0	-12.1	+7.4	-8.2	+0.7	-5.6	-10.4	**-4.7**
Avg. NL-LLM	-0.1	+3.2	+3.0	-6.7	-1.2	+3.8	-2.5	-9.0	-3.7	**-1.5**
Avg. NL-LLM (w/o LLaMA, Gemma)	-1.0	+0.9	0.0	-4.1	+0.5	-0.1	-16.0	+1.7	+0.1	**-2.0**

Table 6: Aggregated performance and redundancy (%) of few-shot experiments, with their change (Δ) from zero-shot highlighted in bold. Full few-shot results of all models are presented in Table 12 of performance and Table 13 of redundancy. SemEval results are missing due to a limited context window for this task (see Section 3.3).

Model	SST-2 (*Acc*)				MRPC (*Acc*)				SemEval (*Micro-F1*)				HCC-Staging (*Acc*)				$\overline{X}$
	Python	JS	C++	Ruby	Python	JS	C++	Ruby	Python	JS	C++	Ruby	Python	JS	C++	Ruby	
GPT-3.5 Turbo (Code)	94.6	94.4	94.5	95.1	73.7	72.1	74.6	71.6	45.2	46.0	44.6	46.2	58.1	58.1	64.5	58.1	62.4
DeepSeek-V3.1 (Code)	95.1	95.2	95.2	95.6	76.6	78.9	77.0	77.0	62.0	65.1	62.3	62.5	83.9	77.4	87.1	77.4	75.6
Qwen3-Coder-30B	94.0	95.0	94.5	94.2	79.2	79.1	79.4	77.3	46.8	41.8	45.7	43.6	67.7	74.2	64.5	64.5	62.1
CodeLlama-13b	89.0	90.6	90.5	90.4	68.0	68.2	69.7	69.3	36.3	39.9	36.5	36.0	51.6	51.6	51.6	51.6	55.0
codegemma-7b	91.9	92.2	92.5	92.2	66.7	68.1	67.7	67.8	35.3	41.8	41.8	37.7	67.7	71.0	67.7	67.7	63.4
Avg.	92.5	93.3	93.2	93.1	72.6	73.6	73.5	72.9	45.1	47.2	46.6	45.0	67.7	68.6	67.7	65.3	

Table 7: Zero-shot performance (%) of four different programming languages: Python, JavaScript (JS), C++ and Ruby. Code-prompts written in those programming languages are presented in Appendix E.

LLMs do so in only 15 (13.6%). We further notice even smaller, open-source coders such as Qwen-Coder and codegemma often output only the target label, adhering strictly to the format specification. By contrast, most evaluted open-source NL-LLMs struggle with following concise output constraints and frequently add unsolicited text (e.g. *Based on the information* or *I am not sure*). The findings indicate an inherent strength of Code-LLMs that they are naturally better at generating precise, format-faithful outputs.

4.2 Few-shot Results

We present few-shot results (performance in Table 12 and redundancy in Table 13) in Appendix B. We compare them with zero-shot results based on average scores across all NL-/code-models (see Table 6). We observe that Code-LLMs benefit more consistently from few-shot demonstrations: (1) Code-LLMs show performance gains on 7 of 9 tasks, whereas NL-LLMs improve on only 5. (2) Averaged across all tasks, Code-LLMs achieve a 5.7% improvement, while NL-LLMs exhibit a slight overall decline (–0.8%). Since we notice this decline is driven by four underperforming NL-LLMs (from Llama and gemma), we exclude them as outliers; the remaining NL-LLMs show a more moderate improvement of 4.2%.

A similar trend appears in **redundancy**. Across all tasks, Code-LLMs reduce redundancy by 4.7 percentage points, compared to 1.5 for NL-LLMs (or 0.2 for the refined NL-LLM subset mentioned above). These results indicate that Code-LLMs not only learn more effectively from few-shot demonstrations but also produce more concise and format-compliant outputs under few-shot prompting.

4.3 Results of Multilinguality

To assess multilingual abilities of Code-LLMs, we first select the best performing (indicated by highest mean score across tasks $\overline{X}$ in Table 4) code model from each of the 5 providers: GPT-3.5 Turbo, DeepSeek-V3.1, Qwen3-Coder-30B, CodeLlama-13b and codegemma-7b; then we experiment in different programming and human languages.

Programming languages. We translate code prompts (originally in Python) into three additional programming languages: two high-resource (JavaScript and C++) and one relatively low-resourced (Ruby). All four languages share a property: they support one-line, assert-style unit tests, enabling a consistent prompt structure across lan-

Model	AR	BG	DE	EL	EN	ES	FR	HI	RU	SW	TH	TR	UR	VI	ZH	Avg.
GPT-3.5 Turbo (Code)	57.8	60.4	63.5	59.0	68.0	63.3	63.2	52.2	59.2	56.3	52.6	58.0	49.2	58.7	59.9	58.8
DeepSeek-V3.1 (Code)	71.0	74.2	74.5	73.4	75.6	74.4	75.8	70.6	73.4	64.5	70.1	71.5	66.1	73.0	74.3	72.2
Qwen3-Coder-30B	72.5	75.3	75.1	73.2	84.2	79.1	77.7	68.1	74.0	62.4	69.4	71.6	61.2	74.5	76.0	73.0
CodeLlama-13b	45.7	51.5	52.5	43.4	57.6	54.1	54.5	40.1	53.4	33.7	43.2	47.8	42.5	48.3	49.2	47.8
codegemma-7b	49.0	51.9	53.8	49.5	57.2	55.8	53.9	45.0	52.5	46.2	49.0	51.7	42.2	49.5	50.4	50.5
Avg.	59.2	62.7	63.9	59.7	68.5	65.3	65.0	55.2	62.5	52.6	56.9	60.1	52.2	60.8	62.0	
Ranking by Avg.	#11	#5	#4	#10	#1	#2	#3	#13	#7	#14	#12	#9	#15	#8	#6	

Table 8: Zero-shot accuracy (%) of XNLI over 15 languages. The prompt is presented in Appendix E.5. Evaluated languages are AR (Arabic), BG (Bulgarian), DE (German), EL (Greek), EN (English), ES (Spanish), FR (French), HI (Hindi), RU (Russian), SW (Swahili), TH (Thai), TR (Turkish), UR (Urdu), VI (Vietnamese), and ZH (Chinese).

guages (see Appendix E for all translated prompts).

We conduct zero-shot experiments with translated code prompts on four datasets, one drawn from each task category. As shown in Table 7, performance remains highly consistent across programming languages. For each task, we average scores over five models per language and then compare these language-wise averages. The resulting divergences are minimal: 0.8% for SST, 1.0% for MRPC, 2.1% for SemEval, and 3.3% for HCC-Staging — indicating that the world knowledge encoded in Code-LLMs to perform evaluated NLP tasks is largely programming-language agnostic, transferring robustly across these 4 languages.

Human languages. We further evaluate Code-LLMs on non-English inputs using dataset XNLI. Since XNLI provides sentence-aligned translations across 15 languages, it allows direct, language-wise comparison. We keep the prompt template in English and add a short instruction indicating the language such as "*Both premise and hypothesis are written in German*". Table 8 shows Code-LLMs perform best on English (68.5%), followed by other high-resource European languages such as Spanish, French and German. A substantial drop appears for lower-resource languages (e.g., -16.3% for Urdu, -15.9% for Swahili). Table 9 in Appendix B spotlights these five Code-LLMs across English and Chinese. The two China-based models (DeepSeek, Qwen-Coder) exhibit the smallest relative accuracy drop when switching from English to Chinese.

5 Related Work

Evaluating Code-LLMs has largely focused on code generation, either from natural language instructions (e.g., MBPP (Austin et al., 2021), CODEJUDGE (Tong and Zhang, 2024), CodeArena (Yang et al., 2025c)) or from partial code contexts (*auto-completion*), such as HumanEval (Chen et al., 2021) and RepoBench (Liu et al., 2024). Some recent benchmarks extend this scope to broader software engineering tasks, including unit test generation (Yang et al., 2024) and Github issue-driven code modification (Jimenez et al., 2024). To our knowledge, the general NLP capabilities of Code-LLMs remain largely unexplored.

However, **applying Code-LLMs to NLP** has shown promise in specific settings. Li et al. (2023) show that code-style prompting with Code-LLMs can outperform NL-LLMs on information extraction tasks in few-shot scenarios and introduce *format fidelity* to assess whether generated code conforms to a parsable Python dictionary structure. Mohajeri et al. (2024) evaluate CodeLlama on three TC and one NLI task, also under few-shot settings. Zhang et al. (2023a) explore code-style prompts with GPT models for generation tasks such as QA and summarization, again relying on few-shot demonstrations. Our work extends this line of research through a systematic evaluation of Code-LLMs over a broad spectrum of NLP tasks and model families, across both zero-shot and few-shot regimes and multilingual settings.

6 Conclusion

In this paper, we present CODE-PROMPT, a unified evaluation framework for benchmarking Code-LLMs on a diverse set of NLP tasks using code-style prompts. The framework is readily extensible beyond the tasks considered in this study. We evaluate 22 LLMs (including 13 Code-LLMs) across 10 datasets under zero-shot and few-shot settings, and further investigate the impact of different programming and human languages. Our results show that Code-LLMs achieve performance comparable to NL-LLMs on a wide range of NLP tasks, while exhibiting distinctive traits such as reduced redundant generation and strong transferability across programming languages. We hope this work sheds light on the natural language capabilities of Code-LLMs and motivates further research at the intersection of code intelligence and NLP.

Limitations

Potential data leakage in LLMs. A major source of bias in benchmarking large language models is data leakage (Xu et al., 2024; Zhou et al., 2025), where a benchmarked dataset may have been included in a model's pre-training corpus. Such leakage can inflate reported performance and undermine fair comparison. While many of the datasets evaluated in this work were released prior to the deployment of modern LLMs, Table 1 shows that **MSciNLI** (released in 2024) and our proposed **HCC-Staging** dataset are relatively recent, and therefore less likely to be present in model pre-training data. We find that our main conclusions remain consistent when focusing on these two datasets: Code-LLMs achieve performance comparable to NL-LLMs, exhibit substantially better redundancy, and (on HCC-Staging) demonstrate strong transferability across programming languages. Nevertheless, we cannot fully rule out indirect exposure through continuously updated training pipelines, and the results should therefore be interpreted with this limitation in mind.

Limited access to the latest models. As discussed in Section 3.2, our evaluation includes the most recent Code-LLMs that are publicly accessible at the time of study, either as open-weight checkpoints or via a public code-completion API. However, evaluating the very latest coding models is constrained by limited openness in the ecosystem: (1) several proprietary model families that power state-of-the-art coding assistants (e.g., Gemini and Claude) are closed-source and do not provide a public code-completion API. (2) Some open-weight model lines, such as CodeLlama and codegemma, have not released updated code-specialized variants beyond the versions evaluated in this work.

We view this limitation as highlighting the importance of **continued openness** in (Code-)LLM development for future research, and we acknowledge providers such as DeepSeek and Qwen for consistently releasing up-to-date coding models to the research community.

Ethics Statement

The **HCC-Staging** dataset is derived from clinical reports and therefore involves sensitive patient information. To ensure ethical compliance and protect patient privacy, we take the following measures: (1) The source hospital anonymizes the original reports by removing direct identifiers such as patient names and dates of birth (replaced by age). (2) We will publish the ethic code from our partner hospital upon acceptance. (3) To further reduce re-identification risk, we slightly perturbed all tumor dimensions (e.g., `17 mm` adjusted to `15 mm`, `9 x 9 x 8 mm` adjusted to `8 x 8 x 5 mm`) while preserving the original staging outcome under the Milan criteria. The final post-processed dataset was reviewed and approved by the annotator. Here we explicitly specify non-commercial research-only use for this HCC-Staging dataset.

References

Mubashara Akhtar, Abhilash Shankarampeta, Vivek Gupta, Arpit Patil, Oana Cocarascu, and Elena Simperl. 2023. Exploring the numerical reasoning capabilities of language models: A comprehensive analysis on tabular data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 15391–15405, Singapore. Association for Computational Linguistics.

Christoph Alt, Aleksandra Gabryszak, and Leonhard Hennig. 2020. Probing linguistic features of sentence-level representations in neural relation extraction. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1534–1545, Online. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Florian Borchert, Christina Lohr, Luise Modersohn, Jonas Witt, Thomas Langer, Markus Follmann, Matthias Gietzelt, Bert Arnrich, Udo Hahn, and Matthieu-P. Schapranow. 2022. GGPONC 2.0 - the German clinical guideline corpus for oncology: Curation workflow, annotation policy, baseline NER taggers. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 3650–3660, Marseille, France. European Language Resources Association.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan,

Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Alexis Conneau, Ruty Rinott, Guillaume Lample, Adina Williams, Samuel Bowman, Holger Schwenk, and Veselin Stoyanov. 2018. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2475–2485, Brussels, Belgium. Association for Computational Linguistics.

William B. Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*.

Kaiyue Feng, Yilun Zhao, Yixin Liu, Tianyu Yang, Chen Zhao, John Sous, and Arman Cohan. 2025. Physics: Benchmarking foundation models on university-level physics problem solving. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 11717–11743, Vienna, Austria. Association for Computational Linguistics.

Rory A. Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of Human Genetics*, 7:179–188.

Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3816–3830, Online. Association for Computational Linguistics.

Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. 2025. A survey of vibe coding with large language models. *CoRR*, abs/2510.12399.

Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 39:1–39:13. ACM.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

David Harbecke, Yuxuan Chen, Leonhard Hennig, and Christoph Alt. 2022. Why only micro-f1? class weighting of measures for relation classification. In *Proceedings of NLP Power! The First Workshop on Efficient Benchmarking in NLP*, pages 32–41, Dublin, Ireland. Association for Computational Linguistics.

Iris Hendrickx, Su Nam Kim, Zornitsa Kozareva, Preslav Nakov, Diarmuid Ó Séaghdha, Sebastian Padó, Marco Pennacchiotti, Lorenza Romano, and Stan Szpakowicz. 2010. SemEval-2010 task 8: Multi-way classification of semantic relations between pairs of nominals. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, pages 33–38, Uppsala, Sweden. Association for Computational Linguistics.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Mehran Kazemi, Quan Yuan, Deepti Bhatia, Najoung Kim, Xin Xu, Vaiva Imbrasaite, and Deepak Ramachandran. 2023. Boardgameqa: a dataset for natural language reasoning with contradictory information. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA. Association for Computing Machinery.

Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu. 2023. CodeIE: Large code generation models are better few-shot information extractors. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353, Toronto, Canada. Association for Computational Linguistics.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9).

Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024. Repobench: Benchmarking repository-level

code auto-completion systems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Do Xuan Long, Duy Dinh, Ngoc-Hai Nguyen, Kenji Kawaguchi, Nancy F. Chen, Shafiq Joty, and Min-Yen Kan. 2025. What makes a good natural language prompt? In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5835–5873, Vienna, Austria. Association for Computational Linguistics.

Yi Luan, Luheng He, Mari Ostendorf, and Hannaneh Hajishirzi. 2018. Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3219–3232, Brussels, Belgium. Association for Computational Linguistics.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, Xiaoyin Che, Zhiyuan Liu, and Maosong Sun. 2024. RepoAgent: An LLM-powered open-source framework for repository-level code documentation generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 436–464, Miami, Florida, USA. Association for Computational Linguistics.

V Mazzaferro, E Regalia, R Doci, S Andreola, A Pulvirenti, F Bozzetti, F Montalto, M Ammatuna, A Morabito, and L Gennari. 1996. Liver transplantation for the treatment of small hepatocellular carcinomas in patients with cirrhosis. *N. Engl. J. Med.*, 334(11):693–699.

Mohammad Mahdi Mohajeri, Mohammad Javad Dousti, and Majid Nili Ahmadabadi. 2024. Cocop: Enhancing text classification with LLM through code completion prompt. *CoRR*, abs/2411.08979.

Heru Nugroho, Nugraha Priya Utama, and Kridanto Surendro. 2020. Performance evaluation for class center-based missing data imputation algorithm. In *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, ICSCA '20, page 36–40, New York, NY, USA. Association for Computing Machinery.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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

Bibek Poudel, Adam Cook, Sekou Traore, and Shelah Ameli. 2024. Documint: Docstring generation for python using small language models. *CoRR*, abs/2405.10243.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Mobashir Sadat and Cornelia Caragea. 2024. MSciNLI: A diverse benchmark for scientific natural language inference. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1610–1629, Mexico City, Mexico. Association for Computational Linguistics.

Alireza Salemi, Julian Killingback, and Hamed Zamani. 2025. ExPerT: Effective and explainable evaluation of personalized long-form text generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 17516–17532, Vienna, Austria. Association for Computational Linguistics.

Sergio Servantez, Joe Barrow, Kristian Hammond, and Rajiv Jain. 2024. Chain of logic: Rule-based reasoning with large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2721–2733, Bangkok, Thailand. Association for Computational Linguistics.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642, Seattle, Washington, USA. Association for Computational Linguistics.

Supriyono, Aji Prasetya Wibawa, Suyono, and Fachrul Kurniawan. 2024. Advancements in natural language processing: Implications, challenges, and future directions. *Telematics and Informatics Reports*, 16:100173.

Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. 2024. Large language models for data annotation and synthesis: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 930–957, Miami, Florida, USA. Association for Computational Linguistics.

Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA. Association for Computational Linguistics.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Muhammad Uzair-Ul-Haq, Davide Rigoni, and Alessandro Sperduti. 2025. Llms as data annotators: How close are we to human performance. *CoRR*, abs/2504.15022.

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, koray kavukcuoglu, and Daan Wierstra. 2016. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium. Association for Computational Linguistics.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TestEval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, Albuquerque, New Mexico. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Alex Warstadt, Amanpreet Singh, and Samuel R. Bowman. 2019. Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7:625–641.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Ruijie Xu, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. 2024. Benchmarking benchmark leakage in large language models. *CoRR*, abs/2404.18824.

Weihao Xuan, Rui Yang, Heli Qi, Qingcheng Zeng, Yunze Xiao, Aosong Feng, Dairui Liu, Yun Xing, Junjue Wang, Fan Gao, Jinghui Lu, Yuang Jiang, Huitao Li, Xin Li, Kunyu Yu, Ruihai Dong, Shangding Gu, Yuekang Li, Xiaofei Xie, and 13 others. 2025. MMLU-ProX: A multilingual benchmark for advanced large language model evaluation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1513–1532, Suzhou, China. Association for Computational Linguistics.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and Julian McAuley. 2025a. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in LLMs. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2586–2616, Suzhou, China. Association for Computational Linguistics.

Haotong Yang, Yi Hu, Shijia Kang, Zhouchen Lin, and Muhan Zhang. 2025b. Number cookbook: Number understanding of language models and how to improve it. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Jian Yang, Jiaxi Yang, Wei Zhang, Jin Ke, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Zhoujun Li, Binyuan Hui, and Junyang Lin. 2025c. CodeArena: Evaluating and aligning CodeLLMs on human preference. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 9683–9694, Suzhou, China. Association for Computational Linguistics.

Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen.

2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1607–1619, New York, NY, USA. Association for Computing Machinery.

Li Zhang, Liam Dugan, Hainiu Xu, and Chris Callison-burch. 2023a. Exploring the curious case of code prompts. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 9–17, Toronto, Canada. Association for Computational Linguistics.

Linghao Zhang, Jingshu Zhao, Chong Wang, and Peng Liang. 2024. Using Large Language Models for Commit Message Generation: A Preliminary Study . In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 126–130, Los Alamitos, CA, USA. IEEE Computer Society.

Ruoyu Zhang, Yanzeng Li, Yongliang Ma, Ming Zhou, and Lei Zou. 2023b. LLMaAA: Making large language models as active annotators. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13088–13103, Singapore. Association for Computational Linguistics.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 649–657, Cambridge, MA, USA. MIT Press.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, and 7 others. 2024. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409.

Xin Zhou, Martin Weyssow, Ratnadira Widyasari, Ting Zhang, Junda He, Yunbo Lyu, Jianming Chang, Beiqi Zhang, Dan Huang, and David Lo. 2025. Lessleak-bench: A first investigation of data leakage in llms across 83 software engineering benchmarks. *CoRR*, abs/2502.06215.

A Implementation Details

Model access and licenses. The links (given as URL) of the evaluated models in this paper are listed in Table 10.

OpenAI models (such as GPT-3.5-Turbo) are governed by a proprietary license[6], which permits usage via their APIs for both businesses and developers. DeepSeek uses a custom license for its models that allows free distribution, reproduction and usage of model copies under certain conditions (refer to Section 4 and Attachment A from `https://github.com/deepseek-ai/DeepSeek-Coder/blob/main/LICENSE-CODE`) which we meet. Qwen models are generally available under the permissive Apache 2.0 License[7]. Llama and Gemma models use their custom licenses [8]. For this study, we strictly adhere to the terms and conditions defined by each license: we access proprietary models via their respective APIs and locally reproduce copies of the open-weight models solely for evaluation purposes, without modification or redistribution. All uses remain compliant with the terms of use defined in the aforementioned licenses, limiting our scope to research.

Model deployment. For inference, we follow the appropriate computation framework for each provider: GPT and DeepSeek models are accessed through their official APIs, while all other models are run locally: NL-LLMs using `vLLM` (Kwon et al., 2023) and Code-LLMs using the HuggingFace `transformers` (Wolf et al., 2020) library.

`vLLM` is widely used for efficient inference in recent NL-LLM studies (Salemi et al., 2025; Feng et al., 2025), but it primarily supports instruction-mode generation. In contrast, code completion is natively supported by `transformers`, which is also the officially recommended framework for *all* locally evaluated Code-LLMs in this work, including Qwen Coder[9], codegemma[10] and CodeLlama[11]. To ensure fair comparison across models, we further fix the maximum output length to 16 tokens,

[6] `https://openai.com/policies/terms-of-use/`

[7] `https://github.com/QwenLM/Qwen?tab=Apache-2.0-1-ov-file`

[8] `https://www.llama.com/llama2/license/` and `https://ai.google.dev/gemma/terms`

[9] `https://github.com/QwenLM/Qwen3-Coder/blob/main/README.md`

[10] `https://huggingface.co/google/codegemma-7b`

[11] `https://huggingface.co/meta-llama/CodeLlama-7b-hf`

which matches the default setting in vLLM[12] and comfortably covers the length of all target labels used in our tasks (see Table 1).

Computation infrastructure. To ensure full-precision inference, models with at least 30B parameters (e.g., the 30B and 32B Qwen models) are run on a single NVIDIA H200 GPU, while smaller models are run on a single NVIDIA A100 GPU.

B Detailed Experimental Results

Table 9 spotlights the zero-shot performance of five Code-LLMs across the English and Chinese splits from the XNLI dataset.

Table 12 and Table 13 present the performance and redundancy scores of few-shot experiments, respectively.

Model	English	Chinese	Drop by (%)
DeepSeek-V3.1 (Code)	0.756	0.743	**-1.72**
Qwen3-Coder-30B	0.842	0.760	**-9.74**
codegemma-7b	0.572	0.504	-11.89
GPT-3.5 Turbo (Code)	0.680	0.599	-11.91
CodeLlama-13b	0.576	0.492	-14.58

Table 9: Comparison of zero-shot accuracy between English and Chinese subsets of XNLI. The drop ratio is measured relative to the English score, i.e. $(\text{score}_{ZH} - \text{score}_{EN})/\text{score}_{EN}$.

C Effect of Components in Code Prompt

To analyze the contribution of individual components in our code-style prompt, we conduct an ablation study with four prompt variants on the HCC-Staging dataset. We choose HCC-Staging because, as discussed in the Limitations section, it is newly introduced and thus minimizes the risk of data leakage during LLM pre-training. The four prompt variants are defined as follows:

full: the original code prompt (see the Python prompt in Appendix E.10 for example);

–typing: the full prompt with type hints (e.g., `Literal["True", "False"]`) removed;

–docstring: the full prompt with the docstring removed;

–both: the full prompt with both type hints and the docstring removed. The resulting prompt contains only the function signature (i.e., `def HCC_staging(text: str):`) followed by an empty function body (`pass`).

[12] https://docs.vllm.ai/en/v0.8.4/api/inference_params.html

Model	URL
GPT-3.5-Turbo	https://platform.openai.com/docs/models/gpt-3.5-turbo
DeepSeek-V3.1	https://api.deepseek.com/beta (API until 2025/09/21) and https://huggingface.co/deepseek-ai/DeepSeek-V3.1
Qwen3-Coder-30B	https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct
Qwen3-30B-Instruct	https://huggingface.co/Qwen/Qwen3-30B-A3B-Instruct-2507
Qwen2.5-Coder-32B	https://huggingface.co/Qwen/Qwen2.5-Coder-14B
Qwen2.5-32B-Instruct	https://huggingface.co/Qwen/Qwen2.5-32B-Instruct
Qwen2.5-Coder-14B	https://huggingface.co/Qwen/Qwen2.5-Coder-14B
Qwen2.5-14B-Instruct	https://huggingface.co/Qwen/Qwen2.5-14B-Instruct
Qwen2.5-Coder-7B	https://huggingface.co/Qwen/Qwen2.5-Coder-7B
Qwen2.5-7B-Instruct	https://huggingface.co/Qwen/Qwen2.5-7B-Instruct
Qwen2.5-Coder-3B	https://huggingface.co/Qwen/Qwen2.5-Coder-3B
Qwen2.5-3B-Instruct	https://huggingface.co/Qwen/Qwen2.5-3B-Instruct
CodeLlama-13b	https://huggingface.co/meta-llama/CodeLlama-13b-hf
CodeLlama-13b-Python	https://huggingface.co/meta-llama/CodeLlama-13b-Python-hf
Llama-2-13b	https://huggingface.co/meta-llama/Llama-2-13b-hf
CodeLlama-7b	https://huggingface.co/meta-llama/CodeLlama-7b-hf
CodeLlama-7b-Python	https://huggingface.co/meta-llama/CodeLlama-7b-Python-hf
Llama-2-7b	https://huggingface.co/meta-llama/Llama-2-7b-hf
codegemma-7b	https://huggingface.co/google/codegemma-7b
gemma-7b	https://huggingface.co/google/gemma-7b
codegemma-2b	https://huggingface.co/google/codegemma-2b
gemma-2b	https://huggingface.co/google/gemma-2b

Table 10: The URLs of evaluated models.

Table 11 reports the zero-shot accuracy and redundancy of the four prompt variants. The evaluated models are the best performing ones in general across 10 tasks (see Section 4.3 for selection criteria). (1) **Accuracy**: removing either type hints or the docstring leads to a comparable and modest accuracy change (less than 0.03) from the full prompt. In contrast, removing both components (i.e., leaving only the function name and its argument) results in zero accuracy. This indicates a low risk of data leakage and confirms that the task cannot be solved from prior knowledge alone, but instead requires explicit rule specification provided by the prompt. (2) **Redundancy**: removing either component has no effect on redundancy for DeepSeek-V3.1, GPT-3.5 Turbo, and CodeGemma-7B, but increases redundancy for CodeLlama. Overall, the full prompt consistently achieves the best balance between accuracy and redundancy across models.

Model	full	-typing	-docstring	-both
	Accuracy			
DeepSeek-V3.1 (Code)	**83.9**	80.0	**83.9**	0.0
GPT-3.5 Turbo (Code)	58.1	**64.5**	61.3	0.0
Qwen3-Coder-30B	**67.7**	64.5	58.1	0.0
codegemma-7b	67.7	**70.9**	64.5	0.0
CodeLlama-13b	51.6	**61.2**	54.8	0.0
Avg.	65.8	**68.3**	64.5	0.0
	Redundancy			
DeepSeek-V3.1 (Code)	**0.0**	**0.0**	**0.0**	100.0
GPT-3.5 Turbo (Code)	**0.0**	**0.0**	**0.0**	100.0
Qwen3-Coder-30B	30.9	**29.8**	30.4	100.0
codegemma-7b	**0.0**	**0.0**	**0.0**	100.0
CodeLlama-13b	**0.0**	91.3	91.2	100.0
Avg.	**6.2**	24.2	24.3	100.0

Table 11: Comparison of the four prompt variants in zero-shot results (%) over the HCC-Staging dataset.

D Annotation Details

For the HCC-Staging dataset, a specialized physician from a German university hospital extracted tumor size information following the instruction: *Reports more than 12 months prior to the tumor board meeting are excluded. Only tumor size information (number + unit) is extracted, and multiple tumor sizes may be annotated for a single report.* The annotator was compensated according to the hospital's tariff, which is above the local minimum wage. The annotation process required approximately 12 hours in total, distributed over several weeks, with no more than 6 working hours on any single day, fully complying with local labor regulations.

E Prompts

E.1 SST-2

Natural language prompt.
You are a data annotator for sentiment classification. Your task is to read the following text and classify the text into either positive or negative based on its sentiment.
Please answer directly with "positive" or "negative" only.
{*few_shot_examples*}
Here is the text to classify:
Input: {*text*}
Output:

Few-shot example in natural language:
Input: {*text*}
Output: {*label*}

Code prompt (Python).
```
def classify_sentiment(text: str) -> Literal["positive", "negative"]:
    """Classify the sentiment of the given text as either positive or negative.
    Args:
        - text (str): The text to classify.
    Returns:
        Literal["positive", "negative"]: The sentiment label of the text.
    """
    pass

{few_shot_examples}
# Test case for inference
text = "{text}"
assert (classify_sentiment(text) == "
```

Few-shot example in Python:
```
# Example test case
text = "{text}"
assert (classify_sentiment(text) == "{label}")
```

Code prompt (JavaScript).
```
const assert = require('assert');
function classifySentiment(text) {
    /*
    Classify the sentiment of the given text as either positive or negative.
    Args:
        - text (string): The text to classify.
    Returns:
        "positive" | "negative": The sentiment label of the text.
    */
    // TODO: Implement classification logic
}
{few_shot_examples}
# Test case for inference
const text = "{text}";
assert.strictEqual(classifySentiment(text), "
```

Few-shot example in JavaScript:
```
# Example test case
const text = "{text}";
assert.strictEqual(classifySentiment(text), "{label}")
```

Code prompt (C++).
```
#include <cassert>
#include <string>
std::string classify_sentiment(const std::string& text) {
    /*
```

	TC			NLI			RC	NR		
	SST-2	**AG News**	**CoLA**	**MRPC**	**XNLI**	**MSciNLI**	**SciERC**	**Iris**	**HCC-Staging**	$\overline{X}$
Metric	*Acc*	*Acc*	*MCC*	*Acc*	*Acc*	*Macro-F1*	*Micro-F1*	*Acc*	*Acc*	
# Shots	4	4	4	4	4	4	2	4	4	
GPT-3.5 Turbo (Code)	93.7±0.24	**63.4**±1.42	61.1±1.73	76.8±0.48	71.5±0.57	**42.5**±0.82	**60.8**±2.80	96.0±10.97	58.1±0.54	69.3
GPT-3.5 Turbo (NL)	**94.6**±0.19	**63.4**±0.20	**61.9**±1.53	**78.2**±0.52	**75.5**±1.05	28.1±1.08	55.3±3.05	**96.7**±4.02	**74.2**±0.54	**69.8**
DeepSeek-V3.1 (Code)	94.7±0.25	**63.1**±0.51	**71.0**†±1.30	78.0±0.44	85.1±0.35	47.4±1.93	33.7±0.51	**96.0**±3.04	87.1±0.00	72.9
DeepSeek-V3.1 (NL)	**95.8**†±0.69	**63.1**±0.74	70.5±0.28	**80.2**†±0.43	**87.4**±0.40	**63.4**†±1.80	**66.7**±4.02	**96.0**±1.52	**90.3**†±0.00	**79.3**†
Qwen3-Coder-30B	**95.3**±0.19	**65.1**†±0.35	60.4±1.43	**79.0**±0.50	**86.8**±0.25	57.4±1.67	53.9±3.20	**95.7**±1.52	64.5±0.68	**73.1**
Qwen3-30B-Instruct	88.3±5.30	61.2±1.91	**64.1**±0.74	42.2±11.91	85.5±1.49	**60.8**±3.42	**67.2**±2.11	90.1±6.08	**77.4**±3.69	70.8
Qwen2.5-Coder-32B	**95.8**†±0.28	**63.9**±1.87	59.3±1.17	72.8±0.90	87.2±0.22	**53.5**±0.61	42.0±4.59	**97.8**†±3.04	**83.9**±2.08	72.9
Qwen2.5-32B-Instruct	94.4±0.14	**63.9**±0.32	**60.3**±1.41	**78.6**±0.56	**89.6**†±1.04	47.0±4.34	**68.0**†±2.06	95.7±10.54	64.5±0.00	**73.6**
Qwen2.5-Coder-14B	94.6±0.24	59.8±1.32	53.3±2.38	71.7±1.19	85.5±0.79	**50.7**±1.53	52.6±0.35	**92.8**±4.02	**77.4**±1.77	70.9
Qwen2.5-14B-Instruct	**95.4**±0.61	**63.6**±0.18	**59.5**±0.38	**75.7**±1.43	**85.9**±0.69	43.4±4.02	**63.5**±6.11	91.3±1.52	71.0±3.13	**72.1**
Qwen2.5-Coder-7B	93.5±0.52	52.0±5.17	50.2±3.52	**71.4**±0.76	82.7±1.34	35.6±5.94	26.4±1.38	88.4±5.27	**74.2**±3.13	63.8
Qwen2.5-7B-Instruct	**94.0**±0.24	**55.2**±1.61	**58.5**±1.76	51.4±4.52	**83.5**±0.71	**41.8**±1.35	**45.5**±5.10	**95.7**±5.48	61.3±3.04	**65.2**
Qwen2.5-Coder-3B	**92.1**±0.27	47.8±3.13	45.8±1.91	**71.2**±0.41	78.4±0.94	39.0±1.56	30.1±1.65	79.7±12.07	**77.4**±1.81	62.4
Qwen2.5-3B-Instruct	89.3±2.51	**57.1**±2.40	**51.3**±1.14	39.9±0.36	**81.9**±0.92	**46.3**±0.60	**48.4**±2.38	**89.1**±10.64	58.1±1.71	**62.4**
CodeLlama-13b	**91.9**±1.19	49.0±1.38	35.8±2.48	72.1±0.49	**60.0**±0.48	42.2±1.52	16.2±2.36	79.7±8.47	**51.6**±2.13	55.4
CodeLlama-13b-Python	88.9±1.59	**52.7**±9.30	**40.7**±4.59	**75.1**±0.34	58.5±0.47	**43.0**±2.01	**16.8**±1.30	**80.4**±12.07	48.4±4.16	**56.1**
Llama-2-13b	0.0±0.00	0.9±1.10	0.0±0.00	68.7±1.32	41.1±1.38	21.4±4.36	5.7±0.58	0.0±0.00	0.0±0.00	15.3
CodeLlama-7b	**93.7**±1.70	52.8±1.88	**24.4**±0.87	71.0±0.37	**55.8**±0.77	30.4±0.80	17.8±1.21	83.3±11.88	**67.7**±2.08	**55.2**
CodeLlama-7b-Python	91.9±1.52	**53.3**±1.01	9.1±1.66	**75.1**±0.18	54.0±0.44	**36.0**±3.44	**19.9**±1.56	**90.6**±9.50	64.5±1.49	54.9
Llama-2-7b	0.0±0.43	1.9±9.40	1.0±3.17	23.9±2.36	0.7±0.29	11.3±0.61	18.0±3.50	0.0±1.52	0.0±0.00	6.3
codegemma-7b	**94.0**±0.43	**62.7**±2.16	**40.3**±7.50	**54.8**±5.60	**66.1**±1.85	**48.7**±4.08	**20.8**±2.60	**91.3**±2.63	**80.6**±1.02	**62.1**
gemma-7b	1.4±1.55	1.8±0.60	0.0±0.96	39.2±16.12	56.7±2.01	23.7±6.45	19.0±0.50	0.0±3.04	19.4±0.00	17.9
codegemma-2b	**71.4**±4.58	**25.8**±0.28	**3.4**±1.88	**66.5**±0.00	**39.1**±1.19	10.0±0.00	9.3±0.17	**64.5**±13.69	**71.0**±3.47	**40.1**
gemma-2b	5.7±8.58	1.0±0.64	2.4±2.80	63.4±1.29	29.8±1.02	**17.8**±2.90	**12.3**±3.28	33.3±1.52	48.4±0.00	23.8
Avg. Code-LLM	91.7	54.7	42.7	**72.0**	70.1	41.3	30.8	**87.4**	69.7	62.2
Avg. NL-LLM	59.9	39.4	39.0	58.3	65.2	36.8	42.7	55.3	51.3	49.8
Avg. NL-LLM (w/o llama, gemma)	**93.1**	**61.1**	**60.9**	63.7	**84.2**	**47.3**	**59.2**	82.1	**71.0**	**69.2**

Table 12: Few-shot performance (%) of NL-LLMs using natural language prompts and Code-LLMs using code-based prompts. Both mean and standard deviation across three runs are reported. The higher score within each comparison is highlighted in bold. The highest score of each task is denoted with †. *Acc*: Accuracy. *MCC*: Matthews correlation coefficient. $\overline{X}$ denotes the arithmetic mean across 10 tasks.

	TC			NLI			RC	NR		
	SST-2	**AG News**	**CoLA**	**MRPC**	**XNLI**	**MSciNLI**	**SciERC**	**Iris**	**HCC-Staging**	$\overline{R}$
# Shots	4	4	4	4	4	4	2	4	4	
GPT-3.5 Turbo (Code)	**0.0**±0.00	25.2±0.04	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.02	**0.3**±0.11	**0.0**±0.00	**0.0**±0.00	**2.8**
GPT-3.5 Turbo (NL)	**0.0**±0.00	**25.1**±0.14	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	10.3±0.12	**0.0**±0.00	**0.0**±0.00	3.9
DeepSeek-V3.1 (Code)	**0.0**±0.00	25.3±0.09	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	94.5±0.00	**0.0**±0.00	**0.0**±0.00	13.3
DeepSeek-V3.1 (NL)	**0.0**±0.00	**25.4**±0.03	2.6±0.00	**0.0**±0.00	10.5±0.00	6.8±0.00	**0.0**±0.00	2.4±0.00	**0.0**±0.00	**5.3**
Qwen3-Coder-30B	**19.3**±0.55	**38.6**±1.07	**15.9**±0.15	**14.3**±1.86	**17.6**±0.15	**18.5**±0.14	**14.7**±0.59	**19.9**±0.33	**29.5**±0.01	**20.9**
Qwen3-30B-Instruct	88.8±1.68	92.6±0.79	86.1±0.31	79.6±1.13	84.0±0.28	88.1±0.31	54.1±5.86	98.6±0.27	92.1±0.52	84.9
Qwen2.5-Coder-32B	**0.0**±0.00	**25.7**±0.18	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**2.9**
Qwen2.5-32B-Instruct	88.8±0.25	91.9±0.16	81.1±1.73	83.8±0.94	83.8±0.05	88.4±0.61	86.0±0.60	68.8±0.15	91.5±3.18	84.9
Qwen2.5-Coder-14B	**0.0**±0.00	**25.7**±1.10	**0.0**±0.00	**0.0**±0.00	**0.0**±0.02	**2.0**±7.59	**1.0**±1.26	**32.9**±0.00	**0.0**±0.00	**6.8**
Qwen2.5-14B-Instruct	86.2±11.22	92.1±0.16	85.4±0.24	82.5±0.61	85.9±0.44	88.7±0.23	88.6±0.47	81.5±0.41	91.3±0.27	86.9
Qwen2.5-Coder-7B	**0.0**±0.00	**20.4**±0.96	**0.0**±0.00	**0.0**±0.00	**68.8**±9.62	**7.2**±13.26	**0.8**±0.42	**0.0**±0.00	**0.0**±0.00	**10.8**
Qwen2.5-7B-Instruct	88.4±0.07	91.7±0.12	85.2±0.23	68.4±4.98	85.3±0.21	88.3±0.37	89.4±0.56	88.2±0.59	89.9±0.26	86.1
Qwen2.5-Coder-3B	**4.5**±5.50	**15.9**±3.82	**0.0**±0.30	**0.0**±0.05	**80.7**±9.88	**6.1**±9.98	**6.5**±3.83	**23.2**±1.41	**3.0**±2.11	**15.5**
Qwen2.5-3B-Instruct	59.8±6.68	92.3±0.70	81.7±1.07	68.4±3.82	84.8±0.27	86.8±0.97	86.9±0.21	82.8±0.16	93.1±0.32	81.8
CodeLlama-13b	84.6±0.07	89.8±0.14	80.0±0.52	**79.3**±0.24	78.3±0.30	**80.1**±0.27	80.5±0.51	83.0±0.18	91.8±0.14	83.0
CodeLlama-13b-Python	84.8±0.09	**88.9**±1.06	**79.8**±0.30	**79.3**±0.51	**76.8**±0.11	82.1±0.22	**74.6**±0.67	82.4±0.24	**91.7**±0.53	82.3
Llama-2-13b	**0.0**±0.00	99.8±0.19	100.0±0.00	82.8±3.27	82.1±1.87	96.5±1.27	95.0±1.04	**0.0**±0.00	100.0±0.00	**72.9**
CodeLlama-7b	84.5±0.13	89.2±0.61	77.8±0.73	79.9±0.07	77.3±0.06	**79.8**±0.38	**75.7**±1.43	82.7±0.23	91.4±0.14	82.0
CodeLlama-7b-Python	**60.3**±10.49	**87.7**±3.30	**75.2**±2.37	79.0±0.32	**76.7**±0.23	80.4±0.27	77.3±1.55	15.3±7.71	**52.1**±1.73	**67.1**
Llama-2-7b	100.0±0.12	99.7±2.97	100.0±0.62	**73.3**±0.49	99.5±0.19	99.6±0.18	92.7±1.47	**0.0**±0.13	100.0±0.00	85.0
codegemma-7b	**0.0**±0.00	**24.3**±0.18	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**2.7**
gemma-7b	99.6±0.42	99.2±0.26	95.9±0.58	89.2±4.70	85.1±0.50	93.6±3.01	80.8±1.64	100.0±19.86	36.5±0.00	86.7
codegemma-2b	**4.4**±13.71	**2.7**±0.51	**82.6**±0.23	**40.7**±17.86	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	**0.0**±0.00	6.0±0.00	**15.2**
gemma-2b	98.9±1.12	99.1±0.56	87.4±2.35	83.2±2.18	82.2±0.75	95.7±2.42	83.3±2.66	83.3±18.34	**2.6**±0.00	79.5
Avg. Code-LLM	**26.3**	**43.0**	**31.6**	**28.7**	**36.6**	**27.4**	**32.8**	**26.1**	**28.1**	**31.2**
Avg. NL-LLM	64.6	82.6	73.2	64.7	71.2	75.7	69.7	55.1	63.4	68.9
Avg. NL-LLM (w/o llama, gemma)	58.9	73.0	60.3	54.7	62.0	63.9	59.3	60.3	65.4	62.0

Table 13: Redundancy (%) (defined in Section 3.3; lower values indicate better generation precision) of few-shot experiments. Mean and standard deviation of redundancy across three runs are reported. The better (i.e. lower) redundancy in each comparison is highlighted in bold.

Classify the sentiment of the given text as either positive or negative.
Args:
- text (std::string): The text to classify.
Returns:
"positive" or "negative": The sentiment label of the text.
*/
// TODO: Implement classification logic
}
{*few_shot_examples*}
Test case for inference
std::string text = "{*text*}";
assert(classify_sentiment(text) == "
Few-shot example in C++:
Example test case
std::string text = "{*text*}";
assert(classify_sentiment(text) == "{*label*}")

E.2 AG News

Natural language prompt.
You are a data annotator for topic classification of news. Your task is to read the following news (Title + Description) and classify the news into one of the topics: World, Sport, Business, or Sci/Tech.
Please answer directly with "World", "Sport", "Business", or "Sci/Tech" only.
{*few_shot_examples*}
Here is the text to classify:
Input: {*text*}
Output:
Few-shot example in natural language:
Input: {*text*}
Output: {*label*}

Code prompt (Python).
def classify_topic(text: str) -> Literal["World", "Sport", "Business", "Sci/Tech"]:
"""Classify the given text (title + description of news) into one of the following topics:
- World
- Sport
- Business
- Sci/Tech
Args:
- text (str): The text of news, which consists of title and description.
Returns:
Literal["World", "Sport", "Business", "Sci/Tech"]: The topic label of the text.
"""
pass

{*few_shot_examples*}
# Test case for inference
text = "{*text*}"
assert (classify_topic(text) == "
Few-shot example in Python:
# Example test case
text = "{*text*}"
assert (classify_topic(text) == "{*label*}")

E.3 CoLA

Natural language prompt.
You are a linguist to annotate grammatical acceptance. Your task is to read the following text and tell if it is grammatically acceptable or unacceptable in standard English.
Please answer with "acceptable" or "unacceptable" only.
{*few_shot_examples*}
Here is the text to classify:
Input: {*text*}
Output:
Few-shot example in natural language:
Input: {*text*}
Output: {*label*}

Code prompt (Python).
def classify_grammatical_acceptance(text: str) -> Literal["acceptable", "unacceptable"]:
"""Classify the given text as either grammatically "acceptable" or "unacceptable".
n
n
Args:
- text (str): The text to classify.
Returns:
Literal["acceptable", "unacceptable"]: The grammatical acceptance of the text.
"""
pass

{*few_shot_examples*}
# Test case for inference
text = "{*text*}"
assert (classify_grammatical_acceptance(text) == "
Few-shot example in Python:
# Example test case
text = "{*text*}"
assert (classify_grammatical_acceptance(text) ==

"{*label*}")

E.4 MRPC

Natural language prompt.
You are a linguist to annotate sentence pairs for semantic equivalence. Your task is to read two sentences and determine whether they are paraphrases of each other or not.
Paraphrase (equivalent) means the two sentences express the same meaning. Not paraphrase (not_equivalent) means they do not express the same meaning.
Please answer with one word only: "equivalent" or "not_equivalent".
{*few_shot_examples*}
Sentence 1: {*sentence1*}
Sentence 2: {*sentence2*}
Relation:

Few-shot example in natural language:
Sentence 1: {*sentence1*}
Sentence 2: {*sentence2*}
Relation: {*label*}

Code prompt (Python).
def detect_paraphrase(sentence1: str, sentence2: str) -> Literal["equivalent", "not_equivalent"]:
"""Detect if two sentences are paraphrases of each other.
Paraphrase means the two sentences express the same meaning.
Args:
- sentence1 (str): The first sentence.
- sentence2 (str): The second sentence.
Returns:
Literal["equivalent", "not_equivalent"]: "equivalent" if the sentences are paraphrases, "not_equivalent" otherwise.
"""
pass

{*few_shot_examples*}
# Test case for inference
sentence1 = "{*sentence1*}"
sentence2 = "{*sentence2*}"
assert (detect_paraphrase(sentence1, sentence2) == "

Few-shot example in Python:
# Example test case
sentence1 = "{*sentence1*}"
sentence2 = "{*sentence2*}"
assert (detect_paraphrase(sentence1, sentence2) == "{*label*}")

Code (JavaScript) prompt.
const assert = require('assert');
function detectParaphrase(sentence1, sentence2) {
/*
Detect if two sentences are paraphrases of each other.
Paraphrase means the two sentences express the same meaning.
Args:
- sentence1 (string): The first sentence.
- sentence2 (string): The second sentence.
Returns:
"equivalent" or "not_equivalent": "equivalent" if the sentences are paraphrases, "not_equivalent" otherwise.
*/
// TODO: Implement paraphrase detection logic
}
{*few_shot_examples*}
// Test case for inference
const sentence1 = "{*sentence1*}";
const sentence2 = "{*sentence2*}";
assert.strictEqual(detectParaphrase(sentence1, sentence2), "

Few-shot example in JavaScript:
Example test case
const sentence1 = "{*sentence1*}";
const sentence2 = "{*sentence2*}";
assert.strictEqual(detectParaphrase(sentence1, sentence2), "{*label*}");

Code (C++) prompt.
#include <cassert>
#include <string>
std::string detect_paraphrase(const std::string& sentence1, const std::string& sentence2) {
/*
Detect if two sentences are paraphrases of each other.
Paraphrase means the two sentences express the same meaning.
Args:
- sentence1 (std::string): The first sentence.
- sentence2 (std::string): The second sentence.
Returns:
"equivalent" or "not_equivalent": "equivalent" if the sentences are paraphrases, "not_equivalent" otherwise.
*/
// TODO: Implement paraphrase detection logic

}
{*few_shot_examples*}
// Test case for inference
std::string sentence1 = "{*sentence1*}";
std::string sentence2 = "{*sentence2*}";
assert(detect_paraphrase(sentence1, sentence2) ==
"

Few-shot example in C++:

std::string sentence1 = "{*sentence1*}";
std::string sentence2 = "{*sentence2*}";
assert(detect_paraphrase(sentence1, sentence2) ==
"{*label*}");

Code (Ruby) prompt.

def detect_paraphrase(sentence1, sentence2)
Detect if two sentences are paraphrases of each other.
Paraphrase means the two sentences express the same meaning.
Args:
- sentence1 (String): The first sentence.
- sentence2 (String): The second sentence.
Returns:
"equivalent" or "not_equivalent": "equivalent" if the sentences are paraphrases, "not_equivalent" otherwise.
TODO: Implement paraphrase detection logic
end
{*few_shot_examples*}
# Test case for inference
sentence1 = "{*sentence1*}"
sentence2 = "{*sentence2*}"
raise "Assertion failed" unless (detect_paraphrase(sentence1, sentence2) ==
"

Few-shot example in Ruby:

sentence1 = "{*sentence1*}"
sentence2 = "{*sentence2*}"
raise "Assertion failed" unless (detect_paraphrase(sentence1, sentence2) ==
"{*label*}")

E.5 XNLI

Natural language prompt.

You are a linguist to annotate textual entailment/contradiction. Your task is to read a premise and a hypothesis, and then determine whether the relation between them is entailment, contradiction, or neutral.
Entailment means the hypothesis logically follows from the premise. Contradiction means the hypothesis contradicts the premise. Neutral means neither entailment nor contradiction.
Please answer with one word only: "entailment", "neutral" or "contradiction".
{*few_shot_examples*}
Premise: {*premise*}
Hypothesis: {*hypothesis*}
Relation:

Few-shot example in natural language:

Premise: {*premise*}
Hypothesis: {*hypothesis*}
Relation: {*label*}

Code (Python) prompt.

def natural_language_inference(premise: str, hypothesis: str) -> Literal["entailment", "neutral", "contradiction"]:
"""Classify the relationship between a premise and a hypothesis (both written in {language}) as
"entailment", "neutral", or "contradiction".
"entailment" means the hypothesis logically follows from the premise.
"contradiction" means the hypothesis contradicts the premise.
"neutral" means neither entailment nor contradiction.
Args:
- premise (str): The premise text.
- hypothesis (str): The hypothesis text.
Returns:
Literal["entailment", "neutral", "contradiction"]: The relationship label.
"""
pass

{*few_shot_examples*}
# Test case for inference
premise = "{*premise*}"
hypothesis = "{*hypothesis*}"
assert (natural_language_inference(premise, hypothesis) == "

Few-shot example in Python:

# Example test case
premise = "{*premise*}"
hypothesis = "{*hypothesis*}"
assert (natural_language_inference(premise, hypothesis) == "{*label*}")

E.6 MSciNLI

Natural language prompt.

You are a linguist to annotate textual entailment/contrasting/reasoning/neutral over a pair of

sentences extracted from publications in the field of computer science. Your task is to read sentence1 and sentence2, and then determine whether the relation between them is entailment, contrasting, reasoning or neutral.
Entailment means sentence2 can be appended to sentence1 using words like "Specifically", "Precisely", "In particular", "Particularly", "That is", "In other words".
Contrasting means sentence2 can be appended to sentence1 using words like "However", "On the other hand", "In contrast", "On the contrary".
Reasoning means sentence2 can be appended to sentence1 using words like "Therefore", "Thus", "Consequently", "As a result", "As a consequence", "From here we can infer".
Neutral means none of the above.
Please answer with one word only: "entailment", "contrasting", "reasoning" or "neutral".
{*few_shot_examples*}
sentence1: {*sentence1*}
sentence2: {*sentence2*}
relation:

Few-shot example in natural language:

sentence1: {*sentence1*}
sentence2: {*sentence2*}
relation: {*label*}

Code (Python) prompt.

```
def natural_language_inference(sentence1: str, sentence2: str) -> Literal["entailment", "contrasting", "reasoning", "neutral"]:
    """Classify the relation between sentence1 and sentence2 from publications
    in the field of computer science into one of 4 categories:
    "entailment" means sentence2 can be appended to sentence1 using words like
    'Specifically', 'Precisely', 'In particular', 'Particularly', 'That is', 'In other words'.
    "contrasting" means sentence2 can be appended to sentence1 using words like
    'However', 'On the other hand', 'In contrast', 'On the contrary'.
    "reasoning" means sentence2 can be appended to sentence1 using words like
    'Therefore', 'Thus', 'Consequently', 'As a result', 'As a consequence', 'From here we can infer'.
    "neutral" means none of the above.
    Args:
        - sentence1 (str): The first sentence.
        - sentence2 (str): The second sentence.
    Returns:
        Literal["entailment", "contrasting", "reasoning", "neutral"]: The relation label.
    """
    pass

{few_shot_examples}
# Test case for inference
sentence1 = "{sentence1}"
sentence2 = "{sentence2}"
assert (natural_language_inference(sentence1, sentence2) == "
```

Few-shot example in Python:

```
# Example test case
sentence1 = "{sentence1}"
sentence2 = "{sentence2}"
assert (natural_language_inference(sentence1, sentence2) == "{label}")
```

E.7 SemEval

Natural language prompt.

You are a data annotator for relation classification of two entities in a text. Your task is to read the following text in which two entities are marked by <e1> and </e1> for the subject entity, and <e2> and </e2> for the object entity, and classify the relation between the two entities into one of the following 10 labels:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination", "Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic", "Product-Producer", "Other",
where "Other" means there is no relation between the two entities.
{*few_shot_examples*}
Here is the text for relation classification:
Input: {*text*}
Output: The relation between {*subj*} and {*obj*} in the input is

Few-shot example in natural language:

Input: {*text*}
Output: The relation between {*subj*} and {*obj*} in the input is {*label*}.

Code (Python) prompt.

```
def classify_relation(text: str, subj: str, obj: str) -> Literal["Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination", "Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic", "Product-Producer",
```

"Other"]:
"""Classify the relation of the subject entity and object entity from the given text
into one of the following 10 labels:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination",
"Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic",
"Product-Producer", "Other", where "Other" means there is no relation between the two entities.
Args:
- text (str): A text with a subject entity marked by <e1> and </e1>, and an object entity marked by <e2> and </e2>.
- subj (str): The subject entity in the text.
- obj (str): The object entity in the text.
Returns:
Literal["Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination", "Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic", "Product-Producer", "Other"]: The relation label between the subject entity and object entity.
"""
pass

{*few_shot_examples*}
# Test case for inference
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
assert (classify_relation(text, subj, obj) == "

Few-shot example in Python:

# Example test case
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
assert (classify_relation(text, subj, obj) == "{*label*}")

Code (JavaScript) prompt.

const assert = require('assert');
function classifyRelation(text, subj, obj) {
/*
Classify the relation of the subject entity and object entity from the given text
into one of the following 10 labels:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination",
"Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic",
"Product-Producer", "Other", where "Other" means there is no relation between the two entities.
Args:
- text (string): A text with a subject entity marked by <e1> and </e1>, and an object entity marked by <e2> and </e2>.
- subj (string): The subject entity in the text.
- obj (string): The object entity in the text.
Returns:
"Cause-Effect" | "Component-Whole" | "Content-Container" | "Entity-Destination" |
"Entity-Origin" | "Instrument-Agency" | "Member-Collection" | "Message-Topic" |
"Product-Producer" | "Other": The relation label between the subject entity and object entity.
*/
// TODO: Implement classification logic
}
{*few_shot_examples*}
Test case for inference
const text = "{*text*}";
const subj = "{*subj*}", obj = "{*obj*}";
assert.strictEqual(classifyRelation(text, subj, obj), "

Few-shot example in JavaScript:

Example test case
const text = "{*text*}";
const subj = "{*subj*}", obj = "{*obj*}";
assert.strictEqual(classifyRelation(text, subj, obj), "{*label*}");

Code (C++) prompt.

#include <cassert>
#include <string>

std::string classify_relation(const std::string& text, const std::string& subj, const std::string& obj) {
/*
Classify the relation of the subject entity and object entity from the given text
into one of the following 10 labels:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination",
"Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic",
"Product-Producer", "Other", where "Other" means there is no relation between the two entities.
Args:
- text (std::string): A text with a subject entity marked by <e1> and </e1>, and an object entity marked by <e2> and </e2>.
- subj (std::string): The subject entity in the text.
- obj (std::string): The object entity in the text.

Returns:
"Cause-Effect" or "Component-Whole" or "Content-Container" or "Entity-Destination" or
"Entity-Origin" or "Instrument-Agency" or "Member-Collection" or "Message-Topic" or
"Product-Producer" or "Other": The relation label between the subject entity and object entity.
*/
// TODO: Implement classification logic
}
{*few_shot_examples*}
Test case for inference
std::string text = "{*text*}";
std::string subj = "{*subj*}", obj = "{*obj*}";
assert(classify_relation(text, subj, obj) == "

Few-shot example in C++:

Example test case
std::string text = "{*text*}";
std::string subj = "{*subj*}", obj = "{*obj*}";
assert(classify_relation(text, subj, obj) == "{*label*}");

Code (Ruby) prompt.

def classify_relation(text, subj, obj)
Classify the relation of the subject entity and object entity from the given text
into one of the following 10 labels:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination",
"Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic",
"Product-Producer", "Other", where "Other" means there is no relation between the two entities.
Args:
- text (String): A text with a subject entity marked by <e1> and </e1>, and an object entity marked by <e2> and </e2>.
- subj (String): The subject entity in the text.
- obj (String): The object entity in the text.
Returns:
"Cause-Effect", "Component-Whole", "Content-Container", "Entity-Destination",
"Entity-Origin", "Instrument-Agency", "Member-Collection", "Message-Topic",
"Product-Producer", "Other": The relation label between the subject entity and object entity.
TODO: Implement classification logic
end
{*few_shot_examples*}
# Test case for inference
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
raise "Assertion failed" unless (classify_relation(text, subj, obj) == "

Few-shot example in Ruby:

# Example test case
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
raise "Assertion failed" unless (classify_relation(text, subj, obj) == "{*label*}")

E.8 SciERC

Natural language prompt.

You are a data annotator for relation classification of two entities in a text. Your task is to read the following text in which the subject entity is surrounded by '[[' and ']]', and object entity is surrounded by '«' and '»', and then classify the relation between the two entities into one of the following 7 labels: "compare", "conjunction", "evaluate-for", "feature-of", "hyponym-of", "part-of", "used-for".
{*few_shot_examples*}
Here is the text for relation classification:
Input: {*text*}
Output: The relation between {*subj*} and {*obj*} in the input is

Few-shot example in natural language:

Input: {*text*}
Output: The relation between {*subj*} and {*obj*} in the input is {*label*}

Code (Python) prompt.

def classify_relation(text: str, subj: str, obj: str) -> Literal["compare", "conjunction", "evaluate-for", "feature-of", "hyponym-of", "part-of", "used-for"]:
"""Classify the relation of the subject entity and object entity from the given text
into one of the following 7 labels: "compare", "conjunction", "evaluate-for", "feature-of", "hyponym-of", "part-of", "used-for".
Args:
- text (str): A text with a subject entity surrounded by '[[' and ']]', and an object entity surrounded by '«' and '»'.
- subj (str): The subject entity in the text.
- obj (str): The object entity in the text.
Returns:
Literal["compare", "conjunction", "evaluate-for", "feature-of", "hyponym-of", "part-of", "used-for"]: The relation label between the subject entity and object entity.

"""
pass

{*few_shot_examples*}
# Test case for inference
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
assert (classify_relation(text, subj, obj) == "

Few-shot example in Python:

# Example test case
text = "{*text*}"
subj, obj = "{*subj*}", "{*obj*}"
assert (classify_relation(text, subj, obj) == "{*label*}")

E.9 Iris

Natural language prompt.

You are a data annotator for species classification of iris flowers. Your task is to read the following rule for classification and the numerical features of the iris flower, and classify the species into either setosa, versicolor, or virginica based on the rule.
Please answer directly with only one of: "setosa", "versicolor", or "virginica" as output.
Rule:
If Petal Width < 0.8 cm, then the species is "setosa".
If Petal Width >= 0.8 cm and < 1.8 cm, then the species is "versicolor".
If Petal Width >= 1.8 cm, then the species is "virginica".
{*few_shot_examples*}
Here is the flower to classify:
Input: sepal length: {*sepal_length*} cm, sepal width: {*sepal_width*} cm, petal length: {*petal_length*} cm, petal width: {*petal_width*} cm
Output:

Few-shot example in natural language:

Input: sepal length: {*sepal_length*} cm, sepal width: {*sepal_width*} cm, petal length: {*petal_length*} cm, petal width: {*petal_width*} cm
Output: {*label*}

Code (Python) prompt.

def classify_iris_species(text: str) -> Literal["setosa", "versicolor", "virginica"]:
"""Classify the species of the iris flower based on its features and the following rule.
Rule: If Petal Width < 0.8 cm, then the species is "setosa".
If Petal Width >= 0.8 cm and < 1.8 cm, then the species is "versicolor".
If Petal Width >= 1.8 cm, then the species is "virginica".
Args:
- text (str): A string containing the features of the iris flower in the format: "sepal length: X cm, sepal width: Y cm, petal length: Z cm, petal width: W cm".
Returns:
Literal["setosa", "versicolor", "virginica"]: The species label of the iris flower.
"""
pass

{*few_shot_examples*}
# Test case for inference
text = "sepal length: {*sepal_length*} cm, sepal width: {*sepal_width*} cm, petal length: {*petal_length*} cm, petal width: {*petal_width*} cm"
assert (classify_iris_species(text) == "

Few-shot example in Python:

# Example test case
text = "sepal length: {*sepal_length*} cm, sepal width: {*sepal_width*} cm, petal length: {*petal_length*} cm, petal width: {*petal_width*} cm"
assert (classify_iris_species(text) == "{*label*}")

E.10 HCC-Staging

Natural language prompt.

You are a clinical annotator to determine whether a patient with hepatocellular carcinoma (HCC) meets the Milan criteria based on reported tumor size(s).
Milan criteria:
- One single tumor with its diameter <= 5 cm,
- Alternatively, up to 3 tumors, each with its diameter <= 3 cm.
Important notes on tumor size interpretation:
- Tumor size (or diameter) may be reported in millimeters (mm) or centimeters (cm).
- It may appear as:
A single value (e.g., "25 mm"),
Two dimensions (e.g., "25 x 20 mm"), or
Three dimensions (e.g., "2.5 x 2.0 x 1.8 cm").
- In all cases, the diameter is defined as the largest single dimension.
- If a patient has multiple tumors, their sizes will be listed together as a comma-separated text, e.g., '5 mm, 30 mm, 3x4x5 mm'.
Your task: Read the provided text describing the patient's tumor size(s), and classify whether the patient meets the Milan criteria.
If the patient is within the criteria, label 'True'. If not, label 'False'.

Please answer directly with 'True' or 'False' only.
{*few_shot_examples*}
Here is the text to classify:
Input: The tumor sizes of the HCC patient: {*text*}
Output:

Few-shot example in natural language:

Input: The tumor size(s) of the HCC patient: {*text*}
Output: {*label*}

Code (Python) prompt.

```
def HCC_staging(text: str) -> Literal["True", "False"]:
    """Classify the patient according to Milan criteria based on tumor size(s) in the text.
    Milan criteria:
        - One single tumor with its diameter <= 5 cm,
        - Alternatively, up to 3 tumors, each with its diameter <= 3 cm.
    Important notes on tumor size interpretation:
        - Tumor size (or diameter) may be reported in millimeters (mm) or centimeters (cm).
        - It may appear as:
            * A single value (e.g., "25 mm"),
            * Two dimensions (e.g., "25 x 20 mm"), or
            * Three dimensions (e.g., "2.5 x 2.0 x 1.8 cm").
        - In all cases, the diameter is defined as the largest single dimension.
        - If a patient has multiple tumors, their sizes will be listed together as a comma-separated text, e.g., '5 mm, 30 mm, 3x4x5 mm'.
    This function reads the provided 'text' describing the patient's tumor size(s), and classify whether the patient meets the Milan criteria.
    If the patient is within the criteria, return "True". If not, return "False".
    Args:
        - text (str): The input text of the HCC patient reporting tumor size(s).
    Returns:
        - str: "True" if the patient meets the Milan criteria, "False" otherwise.
    """
    pass

{few_shot_examples}
# Test case for inference
text = "The tumor size(s) of the HCC patient: {text}"
assert (HCC_staging(text) == "
```

Few-shot example in Python:

```
# Example test case
text = "The tumor size(s) of the HCC patient: {text}"
assert (HCC_staging(text) == "{label}")
```

Code (JavaScript) prompt.

```
const assert = require('assert');

function HCCStaging(text) {
/*
Classify the patient according to Milan criteria based on tumor size(s) in the text.

    Milan criteria:
    - One single tumor with its diameter <= 5 cm,
    - Alternatively, up to 3 tumors, each with its diameter <= 3 cm.

    Important notes on tumor size interpretation:
    - Tumor size (or diameter) may be reported in millimeters (mm) or centimeters (cm).
    - It may appear as:
        * A single value (e.g., "25 mm"),
        * Two dimensions (e.g., "25 x 20 mm"), or
        * Three dimensions (e.g., "2.5 x 2.0 x 1.8 cm").
    - In all cases, the diameter is defined as the largest single dimension.
    - If a patient has multiple tumors, their sizes will be listed together as a comma-separated text, e.g., '5 mm, 30 mm, 3x4x5 mm'.

    This function reads the provided 'text' describing the patient's tumor size(s),
    and classify whether the patient meets the Milan criteria.
    If the patient is within the criteria, return "True". If not, return "False".
    Args:
    - text (string): The input text of the HCC patient reporting tumor size(s).
    Returns:
    - string: "True" if the patient meets the Milan criteria, "False" otherwise.
    */
    // TODO: Implement Milan staging logic
}

    {few_shot_examples}
# Test case for inference
const text = "The tumor size(s) of the HCC patient: {text}";
assert.strictEqual(HCCStaging(text), "
```

Few-shot example in JavaScript:

```
# Example test case
const text = "The tumor size(s) of the HCC patient: {text}";
assert.strictEqual(HCCStaging(text), "{label}");
```

Code (C++) prompt.

```
#include <cassert>
#include <string>

std::string HCC_staging(const std::string& text)
{
/*
Classify the patient according to Milan criteria based on tumor size(s) in the text.

    Milan criteria:
    - One single tumor with its diameter <= 5 cm,
    - Alternatively, up to 3 tumors, each with its diameter <= 3 cm.

    Important notes on tumor size interpretation:
    - Tumor size (or diameter) may be reported in millimeters (mm) or centimeters (cm).
    - It may appear as:
      * A single value (e.g., "25 mm"),
      * Two dimensions (e.g., "25 x 20 mm"), or
      * Three dimensions (e.g., "2.5 x 2.0 x 1.8 cm").
    - In all cases, the diameter is defined as the largest single dimension.
    - If a patient has multiple tumors, their sizes will be listed together as a comma-separated text, e.g., '5 mm, 30 mm, 3x4x5 mm'.

    This function reads the provided 'text' describing the patient's tumor size(s),
  and classify whether the patient meets the Milan criteria.
  If the patient is within the criteria, return "True". If not, return "False".
  Args:
    - text (std::string): The input text of the HCC patient reporting tumor size(s).
  Returns:
    - std::string: "True" if the patient meets the Milan criteria, "False" otherwise.
  */
  // TODO: Implement Milan staging logic
}

  {few_shot_examples}
# Test case for inference
std::string text = "The tumor size(s) of the HCC patient: {text}";
assert(HCC_staging(text) == "
```

Few-shot example in C++:

```
# Example test case
std::string text = "The tumor size(s) of the HCC patient: {text}";
assert(HCC_staging(text) == "{label}");
```

Code (Ruby) prompt.

```
def HCC_staging(text)
  # Classify the patient according to Milan criteria based on tumor size(s) in the text.

    # Milan criteria:
    - One single tumor with its diameter <= 5 cm,
    - Alternatively, up to 3 tumors, each with its diameter <= 3 cm.

    # Important notes on tumor size interpretation:
    - Tumor size (or diameter) may be reported in millimeters (mm) or centimeters (cm).
    - It may appear as:
      * A single value (e.g., "25 mm"),
      * Two dimensions (e.g., "25 x 20 mm"), or
      * Three dimensions (e.g., "2.5 x 2.0 x 1.8 cm").
    - In all cases, the diameter is defined as the largest single dimension.
    - If a patient has multiple tumors, their sizes will be listed together as a comma-separated text, e.g., '5 mm, 30 mm, 3x4x5 mm'.

    # This function reads the provided 'text' describing the patient's tumor size(s),
  # and classify whether the patient meets the Milan criteria.
  # If the patient is within the criteria, return "True". If not, return "False".
  # Args:
    - text (string): The input text of the HCC patient reporting tumor size(s).
  # Returns:
    - string: "True" if the patient meets the Milan criteria, "False" otherwise.
  # TODO: Implement Milan staging logic
end

{few_shot_examples}
# Test case for inference
```

text = "The tumor size(s) of the HCC patient: {*text*}"
raise "Assertion failed" unless (HCC_staging(text) == "

Few-shot example in Ruby:

# Example test case
text = "The tumor size(s) of the HCC patient: {*text*}"
raise "Assertion failed" unless (HCC_staging(text) == "{*label*}")