ScriptoriumWS: A Code Generation Assistant for Weak Supervision

Anonymous authors
Paper under double-blind review

Abstract

Weak supervision is a popular framework for overcoming the labeled data bottleneck: the need to obtain labels for training data. In weak supervision, multiple noisy-but-cheap sources are used to provide guesses of the label and are aggregated to produce high-quality pseudolabels. These sources are often expressed as small programs written by domain experts—and so are expensive to obtain. Instead, we argue for using code-generation models to act as coding assistants for crafting weak supervision sources. We study prompting strategies to maximize the quality of the generated sources, settling on a multi-tier strategy that incorporates multiple types of information. We explore how to best combine hand-written and generated sources. Using these insights, we introduce ScriptoriumWS, a weak supervision system that, when compared to hand-crafted sources, maintains accuracy and greatly improves coverage.

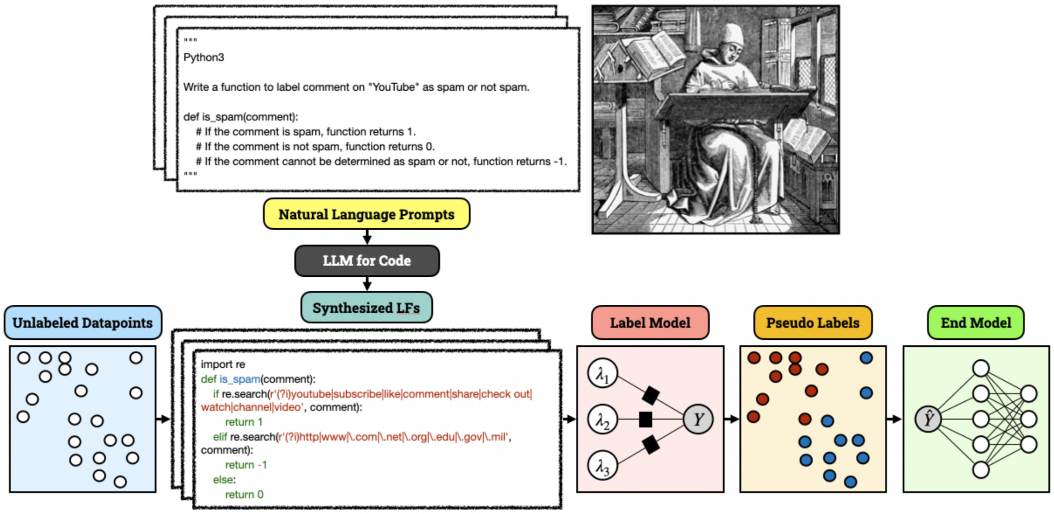
1 Introduction

Access to substantial amounts of high-quality labeled data is a key ingredient for training performant machine learning models. Such data is usually produced by asking domain experts for ground-truth labels, making the process of dataset creation expensive, slow, and hard to scale. Programmatic weak supervision (PWS), a novel paradigm for generating labeled data Ratner et al. (2016), sidesteps these obstacles. The idea behind PWS is to leverage a combination of noisy label estimates obtained from domain knowledge, heuristic rules, and pattern matching. These sources act as noisy labeling functions (LFs), usually expressed as **code**. The outputs of these labeling functions are modeled and aggregated to annotate unlabeled data points Ratner et al. (2016; 2017; 2019); Fu et al. (2020a).

PWS has proven successful Bach et al. (2019); Evensen et al. (2020); Li et al. (2021); Gao et al. (2022) but remains expensive: users must painstakingly write small programs to act as LFs. Users, even domain experts, often need tedious experimentation to carefully set up proper thresholds, manually fine-tune heuristic rules to capture enough keywords, or debug regular expressions. To tackle these challenges, recent approaches automatically produce LFs by using a minimal level of supervision (i.e. a few labeled data points) Varma & Ré (2018); Das et al. (2020); Zhao et al. (2021); Boecking et al. (2021); Roberts et al. (2022) or access to powerful external models (like large language models) to prompt data labels Smith et al. (2022). However, these approaches do not yield programmatic LFs, but rather model-generated noisy label estimates, and so lose the ability to debug and transfer, a key advantage of programmatic weak supervision.

A best-of-both worlds approach is to have **code-generation models write labeling functions**. This neither requires domain experts to write code nor sacrifices the programmatic property of LFs. Indeed, such an approach is now plausible given advances in models that produce code, such as CodeT5 Wang et al. (2021b), Codex Chen et al. (2021), and CodeGen Nijkamp et al. (2022)). Among other benefits, LFs generated by such models can be edited and used as templates, providing programming assistance for users to design LFs more easily and efficiently. Additionally, unlike human-designed LFs, synthesized LFs can be generated in large quantities. Finally, in contrast to using large language models to obtain the noisy labels estimates via prompting, which requires repeated inference calls, synthesized LFs can be stored and reused to label new data at zero cost.

However, it is unclear whether code generation models can produce sufficiently high-quality LFs, and, when it is possible, what approach to take in order to do so. We ask the following fundamental questions we aim to answer in this work:



```
"""
Python3

Write a function to label comment on "YouTube" as spam or not spam.

def is_spam(comment):
    # If the comment is spam, function returns 1.
    # If the comment is not spam, function returns 0.
    # If the comment cannot be determined as spam or not, function returns -1.
"""
```

```
import re
def is_spam(comment):
    if re.search(r'(?i)youtube|subscribe|like|comment|share|check out|
watch|channel|video', comment):
        return 1
    elif re.search(r'(?i)http|www|\.com|\.net|\.org|\.edu|\.gov|\.mil',
comment):
        return -1
    else:
        return 0
```

Figure 1: Overview of the proposedScriptoriumWS system. Code generation models are prompted to produce small programs that act as weak supervision labeling functions. These are used within a weak supervision pipeline to label an unlabeled dataset. A downstream end model is trained on the labeled data.

1. **Prompt format**: Prompts are highly sensitive. Small changes to prompt components lead to great variation in generated results. There is currently no consensus on the best way to generally prompt code-generation models, let alone specifically for labeling functions. Our first question is: what prompting strategy can yield high-quality LFs?
2. **Capability of synthesized LFs**: Next we ask: compared to human-designed LFs, what are the strengths and weaknesses of synthesized LFs? Additionally, what is the typical result when using these synthesized programs in programmatic weak supervision pipelines?
3. **In-context few-shot settings**: If we are allowed to include some heuristic rules or give several examples into the prompt context, does this better guide the model in synthesizing high-quality LFs? What type of in-context information can add and influence the quality of synthesized LFs?

We answer these questions and use the resulting insights to build a novel programmatic weak supervision system called **ScriptoriumWS**. A high-level view of ScriptoriumWS is illustrated in Figure 1. The system creates LFs by prompting code-generation models to synthesize programs and incorporates them into PWS pipelines. To validate ScriptoriumWS, we conduct experiments with OpenAI Codex Chen et al. (2021), a state-of-the-art natural language-to-code system based on GPT-3 Brown et al. (2020). We further propose a complementary approach to incorporate the strength of synthesized and human-designed LFs to improve the performance of the end model.

With the aid of ScriptoriumWS, we explore the advantages that synthesized LFs can bring to the weak supervision framework. We study various prompting strategies to gain insight into how to best generate high-quality LFs. We conduct experiments in diverse text domains and empirically demonstrate the effectiveness of ScriptoriumWS. Excitingly, we find that compared to the human-designed LFs in WRENCH, LFs generated using ScriptoriumWS achieve much higher coverage (the fraction of data points that receive labels) while maintaining high accuracy. For example, using the WRENCH benchmark Zhang et al. (2021) for comparison, we improve the coverage for the SMS dataset from 40.5% to 100% and for the Spouse dataset from 25.8% to 100%, while also improving downstream performance by 1.4 and 5.0 F1 points, respectively.

2 Related Work

Programmatic Weak Supervision (PWS): PWS refers to a broad set of techniques where the data is labeled using cheaply available but potentially noisy labeling information. This information could be from external knowledge bases, heuristics, web search results, and more. Programmatic weak supervision Ratner et al. (2016; 2017) abstracts out these sources as user-provided (written) labeling functions and gives principled ways to aggregate their outputs to produce accurate pseudolabels. This framework is practically effective and widely used in industry Ratner et al. (2017); Bach et al. (2019) and also offers theoretical guarantees, including consistent estimation of accuracies of labeling functions Ratner et al. (2016); Vishwakarma et al. (2022). Its main downside is that writing, iterating, and debugging programmatic labeling functions is slow and expensive.

Automated Weak Supervision (AutoWS): AutoWS is a class of techniques that reduce the need for humans to design LFs. In many cases, designing LFs can be expensive or challenging, particularly when the feature space is too complex, nuanced, or high-dimensional to be reasoned-about by a human, such as in image and video domains. AutoWS techniques can be used in these situations to automate the LF design process by instead using small models as LFs Varma & Ré (2018); Das et al. (2020); Boecking et al. (2021), or by augmenting a few given human-designed LFs to explore more rules Zhao et al. (2021). Similarly, it is possible to directly query large pretrained models for noisy label estimates Smith et al. (2022). The downside of these approaches is that the resulting labeling functions are typically no longer programs that can be debugged, modified, and re-used.

Large Pretrained Models and Prompt Engineering: Prompting is a common way to tap into the knowledge and capabilities of large pre-trained models Liu et al. (2023). Prompting refers to giving natural language instructions to the model in order to get the answer. These prompts can also contain examples of input-output pairs – usually referred to as in-context learning Brown et al. (2020); Dong et al. (2023). Prompting has been successfully applied in various applications and understanding various aspects of prompting is a very active area of research. There are various methods proposed for creating good prompts e.g. Arora et al. (2023) give a general prompting method, chain of thought prompting Wei et al. (2022) and methods to automate prompt generation Zhou et al. (2022). For code generation, prompts with detailed instructions, problem statements, partial code, etc. have been used Sarsa et al. (2022); Denny et al. (2022). We are inspired by these strategies when designing our proposed system.

Using Large Pretrained Models for Data Annotation: Using large language models (LLMs) or other large pretrained models with appropriate prompts to annotate data is a promising direction that can reduce the cost and human effort in data labeling Smith et al. (2022); Wang et al. (2021a). The main limitation here is in terms of scalability and privacy. Inference via querying an API for every data example becomes cost-prohibitive when dealing with large-size training datasets, and sending training data through APIs to other organizations poses a risk of privacy leaks, especially for sensitive data.

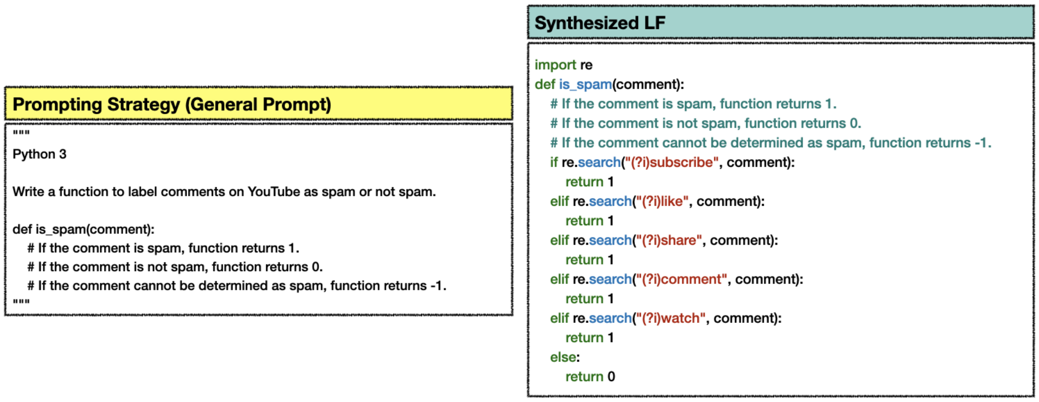
3 METHODOLOGY

In this section, we first describe the programmatic weak supervision (PWS) setup and then discuss approaches that we generate labeling functions by proposing different types of prompts to direct LLMs like Codex in ScriptoriumWS.

3.1 PROGRAMMATIC WEAK SUPERVISION SETUP

Let $\mathcal{X}, \mathcal{Y}$ be the instance and label spaces, respectively. For each of the n unlabeled examples, $x_i \in \mathcal{X}$, we observe noisy labels $\lambda_{1,i}, \dots, \lambda_{m,i}$. These are the outputs of m *labeling functions* (LFs) s_a, where $s_a : \mathcal{X} \to \mathcal{Y}$ and $\lambda_{a,i} = s_a(x_i)$. These LF outputs are fed to a two-step process to construct pseudo labels. Firstly, we learn a *noise model* (also called a label model) that determines how accurate the sources are. That is, we must learn θ for $P_\theta(\lambda_1, \lambda_2, \dots, \lambda_m, y)$. Note that the model involves true labels y that are not observed for any of the samples and this makes the estimation process challenging. Then, pseudo labels for each x_i are inferred using the learned noise model. In other words, we compute $\tilde{y} = \arg\max_{y \in \mathcal{Y}} P_{\hat{\theta}}(\tilde{y} | \lambda_1, \lambda_2, \dots, \lambda_m)$. Finally, an end model can be trained using the generated training dataset: $D = \{(x_i, \tilde{y}_i)\}_{i=1}^n \subseteq \mathcal{X} \times \mathcal{Y}$.

A variety of label models are used for the estimation and inference sets. In this work, we focus on LF generation and use standard label models such as Ratner et al. (2019) and Fu et al. (2020a).



```
"""
Python 3

Write a function to label comments on YouTube as spam or not spam.

def is_spam(comment):
    # If the comment is spam, function returns 1.
    # If the comment is not spam, function returns 0.
    # If the comment cannot be determined as spam, function returns -1.
"""
```

```
import re
def is_spam(comment):
    # If the comment is spam, function returns 1.
    # If the comment is not spam, function returns 0.
    # If the comment cannot be determined as spam, function returns -1.
    if re.search("(?i)subscribe", comment):
        return 1
    elif re.search("(?i)like", comment):
        return 1
    elif re.search("(?i)share", comment):
        return 1
    elif re.search("(?i)comment", comment):
        return 1
    elif re.search("(?i)watch", comment):
        return 1
    else:
        return 0
```

Figure 2: An example of synthesized LF using general prompt strategy for the YouTube spam classification task.

3.2 ScriptoriumWS System

ScriptoriumWS is built on top of the PWS framework. Instead of writing LFs λ manually, we synthesize them using OpenAI Codex. Codex is a descendant of the GPT-3 model, fine-tuned for use in programming applications. It has shown remarkable performance Xu et al. (2022) on code generation tasks across various programming languages. We use the Codex API with natural language prompts to generate code. We vary the temperature parameter from 0 to 0.2 to increase the diversity of the outputs. We feed the synthesized LFs into the PWS pipeline.

3.3 Types of Prompt

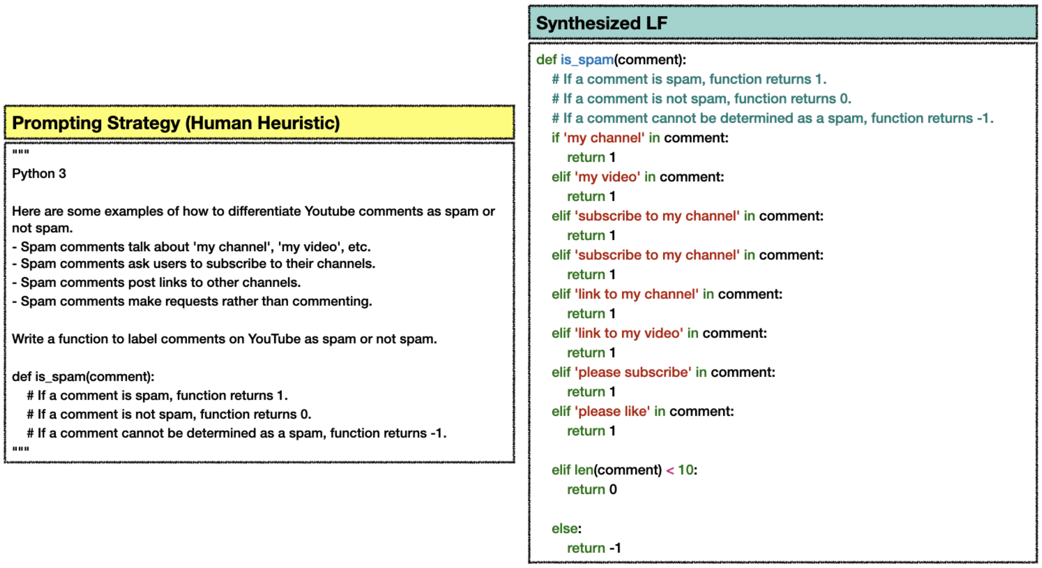
We explored a variety of prompting strategies, based on the kinds of information typically available to weak supervision users. We describe these strategies as being one of five categories, generally going from the least to the most expensive information requirements.

General Prompts: First, we propose a general prompt format that can be easily extended with additional information. A general prompt includes four components, which are the use of programming language, basic task description, function signature, and labeling instructions. We demonstrate an example for the YouTube spam classification task Alberto et al. (2015) in Figure 2.

A general prompt first provides the programming language to be used to synthesize code. Next, the basic task description provides an overview of what the function is expected to do. Afterward, the function signature outlines the name of the synthesized program and the input that the code generation model should use. Finally, we place labeling instructions into the function signature to specify the format and structure of the returned output.

Mission Statement: In addition to providing a basic task description, we also propose an extended type of prompt, which we call the *Mission Statement*. Here we add information to the general prompt to give the code generation model a better understanding of the task in the context. This additional information includes the use of external knowledge bases, and it can include relevant background descriptions about the problem or high-level dataset information (i.e., feature or label classes).

Human Heuristic: In practical applications, users generally have a wealth of prior knowledge and expertise that they can bring to the prompt, including heuristic rules and domain-specific knowledge. Incorporating this prior knowledge into the prompt can be helpful in guiding the code generation model to have a better understanding of the problem and potentially develop a more effective solution that leverages the user's expertise. For example, if a user knows that certain keywords are indicative of spam, they could include this information in the prompt. In ScriptoriumWS, we refer-



Prompting Strategy (Human Heuristic)

```
"""
Python 3

Here are some examples of how to differentiate Youtube comments as spam or
not spam.
- Spam comments talk about 'my channel', 'my video', etc.
- Spam comments ask users to subscribe to their channels.
- Spam comments post links to other channels.
- Spam comments make requests rather than commenting.

Write a function to label comments on YouTube as spam or not spam.

def is_spam(comment):
    # If a comment is spam, function returns 1.
    # If a comment is not spam, function returns 0.
    # If a comment cannot be determined as a spam, function returns -1.
"""
```

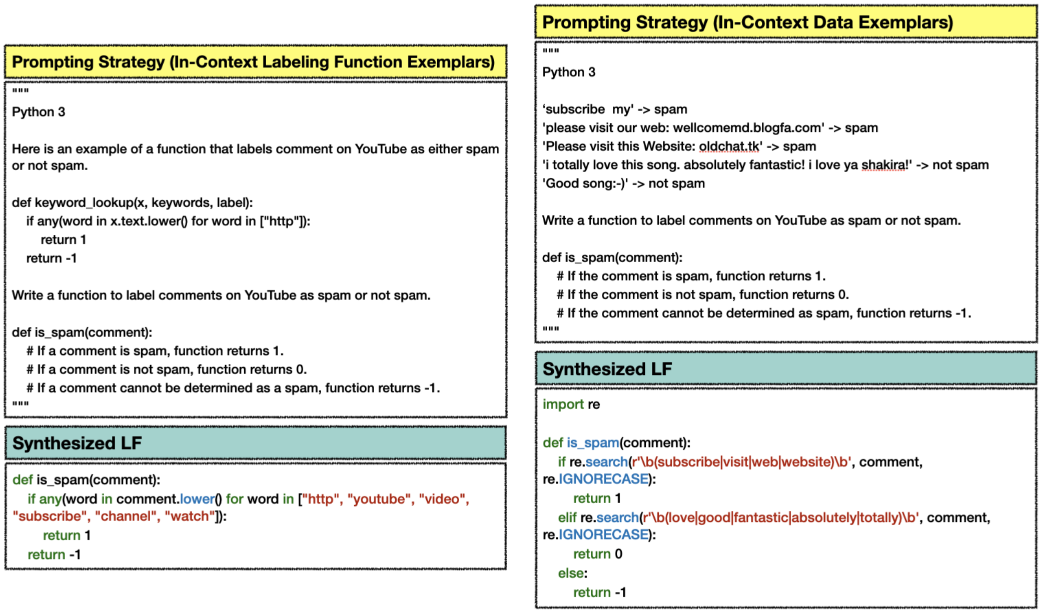
Synthesized LF

```
def is_spam(comment):
    # If a comment is spam, function returns 1.
    # If a comment is not spam, function returns 0.
    # If a comment cannot be determined as a spam, function returns -1.
    if 'my channel' in comment:
        return 1
    elif 'my video' in comment:
        return 1
    elif 'subscribe to my channel' in comment:
        return 1
    elif 'subscribe to my channel' in comment:
        return 1
    elif 'link to my channel' in comment:
        return 1
    elif 'link to my video' in comment:
        return 1
    elif 'please subscribe' in comment:
        return 1
    elif 'please like' in comment:
        return 1

    elif len(comment) < 10:
        return 0

    else:
        return -1
```

Figure 3: An example of a synthesized LF by using human heuristic strategy for the YouTube spam classification task



Prompting Strategy (In-Context Labeling Function Exemplars)

```
"""
Python 3

Here is an example of a function that labels comment on YouTube as either spam
or not spam.

def keyword_lookup(x, keywords, label):
    if any(word in x.text.lower() for word in ["http"]):
        return 1
    return -1

Write a function to label comments on YouTube as spam or not spam.

def is_spam(comment):
    # If a comment is spam, function returns 1.
    # If a comment is not spam, function returns 0.
    # If a comment cannot be determined as a spam, function returns -1.
"""
```

Synthesized LF

```
def is_spam(comment):
    if any(word in comment.lower() for word in ["http", "youtube", "video",
"subscribe", "channel", "watch"]):
        return 1
    return -1
```

Prompting Strategy (In-Context Data Exemplars)

```
"""
Python 3

'subscribe  my' -> spam
'please visit our web: wellcomemd.blogfa.com' -> spam
'Please visit this Website: oldchat.tk' -> spam
'i totally love this song. absolutely fantastic! i love ya shakira!' -> not spam
'Good song:-)' -> not spam

Write a function to label comments on YouTube as spam or not spam.

def is_spam(comment):
    # If the comment is spam, function returns 1.
    # If the comment is not spam, function returns 0.
    # If the comment cannot be determined as spam, function returns -1.
"""
```

Synthesized LF

```
import re

def is_spam(comment):
    if re.search(r'\b(subscribe|visit|web|website)\b', comment,
re.IGNORECASE):
        return 1
    elif re.search(r'\b(love|good|fantastic|absolutely|totally)\b', comment,
re.IGNORECASE):
        return 0
    else:
        return -1
```

Figure 4: Two synthesized LF examples generated by adding label function examples (left) and data examples (right) for the YouTube spam classification task. We can see that code generation model takes the given label function example as reference and learn the relationship between data examples and their expected outputs to extend and synthesize it own program.

ence keywords from existing human-designed LFs and write them into heuristic rules then add these rules to the prompt. We demonstrate an example in the category of human heuristic in Figure 3.

In-Context Labeling Function Exemplars: In-context few-shot learning is a popular approach to perform a new task by inputting a few examples without the need of fine-tuning. We consider a practical scenario where users have already written some LFs or are allowed to access a few existing

LFs. Such LFs can be incorporated into the prompt. The code generation model can use them as function templates to synthesize its own LF, which can be more closely aligned with the user's prior knowledge and expertise, rather than relying solely on the model's own training data.

In-Context Data Exemplars: Besides providing Codex with heuristic rules and in-context few-shot learning with human-designed LFs, we propose another approach by incorporating a few labeled data examples into the prompt to direct the model to understand the problem. Given data examples can serve as concrete illustrations of the problem and provide a clearer understanding of the task and the expected output. This can be especially easy and useful when the problem domain is too complex to design heuristic rules or labeling functions manually.

4 Experiments

In this section, we validate the capability of the proposed system. We implement ScriptoriumWS on the top of weak supervision pipeline proposed as part of the WRENCH benchmark Zhang et al. (2021) and use synthesized LFs to generate weak labels to learn the label model and then subsequently the end model.

4.1 Setup

Datasets We evaluate our approach using four different types of text tasks involving a set of 6 datasets originally included in WRENCH. These 6 datasets are the IMDb Ren et al. (2020) and Yelp Ren et al. (2020) datasets for sentiment classification, the YouTube Alberto et al. (2015) and SMS Almeida et al. (2011) datasets for spam classification, the AGNews Ren et al. (2020) dataset for topic classification, and the Spouse Ratner et al. (2017) dataset for relation classification.

Label Model & End Model Our system is compatible with any choice of label and end model. For ease of comparison, we follow WRENCH and evaluate with five label models to aggregate the output of our synthesized LFs: majority vote (MV), weighted majority vote (WMV), Snorkel (Ratner et al., 2017), Dawid-Skene (DS) Dawid & Skene (1979), and FlyingSquid (FS) Fu et al. (2020b). Finally, once we generate labeled training datasets using these label models alongside our LFs, we train a downstream model—for the sake of simplicity, we use a logistic regression end model for all tasks.

4.2 Analysis

We use our evaluation platform to validate the following claims:

Are ScriptoriumWS LFs comparable to human-designed LFs? We hypothesize that the synthesized LFs generated by ScriptoriumWS can provide results that are comparable to human-designed LFs. To see the strengths and weaknesses of synthesized LFs, we include four basic measurements for LFs generated by different prompting strategies. These are coverage, overlap, conflict, and accuracy. Coverage is the fraction of the dataset labeled by a given LF. Overlap shows the fraction of the dataset with at least two (non-abstain) labels. Conflict indicates a data example for which at least one other LF provides a different estimate. Accuracy computes the fraction of the correctly labeled dataset. We take the average of these indicators over our synthesized LFs and compare them with human-designed LFs in WRENCH.

The results are shown in Table 1. They demonstrate that ScriptoriumWS is capable of generating LFs that have comparable accuracy to human-designed LFs. We observe that synthesized LFs significantly outperform human-designed LFs in terms of coverage. This is not surprising, as human-crafted LFs are often very specific and cannot cover too much of the dataset. On the other hand, we see that there exist more conflicts among outputs produced by synthesized LFs. However, this is not a concern, as such conflicts are resolved by (and in fact, are useful to learn) the label model.

How does ScriptoriumWS perform in PWS pipelines? We anticipate that as LFs from ScriptoriumWS are comparable to human-designed LFs, such LFs will yield good performance in downstream tasks. We train the label model to aggregate the outputs of synthesized LFs and evaluate

	IMDb					Yelp				
	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy
WRENCH	5	0.236	0.116	0.045	0.699	8	0.183	0.136	0.049	0.731
General Prompt	6	0.894	0.887	0.331	0.595	11	0.716	0.716	0.213	0.736
+ Mission Statement	5	0.780	0.766	0.609	0.568	7	0.697	0.689	0.168	0.686
+ Human Heuristic	6	0.764	0.758	0.596	0.644	5	0.783	0.774	0.088	0.658
+ Labeling Function Exemplars	5	0.805	0.792	0.133	0.593	5	0.814	0.812	0.258	0.690
+ Data Exemplars	5	0.895	0.895	0.382	0.633	6	0.701	0.689	0.109	0.702
	SMS					YouTube				
	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy
WRENCH	73	0.007	0.003	0.000	0.973	10	0.170	0.132	0.075	0.826
General Prompt	8	0.815	0.815	0.260	0.897	9	0.592	0.592	0.493	0.646
+ Mission Statement	8	0.819	0.819	0.324	0.817	9	0.643	0.643	0.602	0.607
+ Human Heuristic	9	0.741	0.741	0.118	0.821	8	0.570	0.570	0.491	0.802
+ Labeling Function Exemplars	8	0.038	0.014	0.001	0.822	6	0.662	0.662	0.349	0.795
+ Data Exemplars	8	0.612	0.612	0.366	0.749	8	0.534	0.534	0.397	0.793
	Spouse					AGNews				
	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy	#LFs	Avg. Coverage	Avg. Overlap	Avg. Conflict	Avg. Accuracy
WRENCH	9	0.042	0.021	0.009	0.586	9	0.103	0.051	0.024	0.817
General Prompt	8	1.000	1.000	0.324	0.807	8	0.305	0.279	0.080	0.565
+ Mission Statement	9	0.279	0.208	0.168	0.404	4	0.373	0.215	0.123	0.338
+ Human Heuristic	8	0.295	0.264	0.050	0.456	4	0.346	0.327	0.064	0.818
+ Labeling Function Exemplars	5	0.417	0.307	0.023	0.444	8	0.481	0.472	0.191	0.530
+ Data Exemplars	8	0.601	0.601	0.240	0.595	5	0.345	0.244	0.107	0.636

Table 1: Statistics of synthesized LFs.

	IMDb (Accuracy)							Yelp (Accuracy)						
	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage
WRENCH	0.701	0.710	0.710	0.706	0.704	0.706	0.876	0.690	0.685	0.702	0.715	0.687	0.696	0.828
General Prompt	0.661	0.587	0.587	0.559	0.606	0.600	**0.998**	0.766	0.693	0.703	0.748	0.700	**0.722**	**0.991**
+ Mission Statement	0.613	0.600	0.600	0.542	0.612	0.593	**0.973**	0.743	0.665	0.675	0.634	0.670	0.677	**0.988**
+ Human Heuristic	0.710	0.652	0.652	0.588	0.649	0.650	**0.985**	0.661	0.635	0.642	0.695	0.610	0.648	**0.955**
+ Labeling Function Exemplars	0.650	0.614	0.614	0.596	0.612	0.617	**0.941**	0.793	0.670	0.692	0.728	0.729	**0.722**	**0.994**
+ Data Exemplars	0.713	0.676	0.676	0.690	0.698	**0.691**	**1.000**	0.766	0.678	0.688	0.736	0.685	**0.711**	**0.990**
	SMS (F1-score)							YouTube (Accuracy)						
	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage
WRENCH	0.048	0.240	0.240	0.049	0.000	0.115	0.405	0.852	0.780	0.840	0.832	0.764	0.814	0.893
General Prompt	0.632	0.526	0.672	0.622	0.632	**0.617**	**1.000**	0.760	0.700	0.724	0.668	0.784	0.727	**1.000**
+ Mission Statement	0.599	0.029	0.615	0.599	0.599	**0.488**	**1.000**	0.540	0.624	0.648	0.688	0.468	0.594	**1.000**
+ Human Heuristic	0.606	0.412	0.554	0.529	0.536	**0.527**	**1.000**	0.556	0.740	0.748	0.748	0.776	0.714	**1.000**
+ Labeling Function Exemplars	0.086	0.317	0.317	0.237	0.027	**0.197**	0.218	0.740	0.740	0.740	0.748	0.740	0.742	**1.000**
+ Data Exemplars	0.650	0.337	0.628	0.640	0.630	**0.577**	**1.000**	0.888	0.844	0.868	0.728	0.888	**0.843**	**1.000**
	Spouse (F1-score)							AGNews (Accuracy)						
	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage	Snorkel	WMV	MV	DS	FS	Avgerage	Coverage
WRENCH	0.498	0.205	0.208	0.155	0.343	0.282	0.258	0.625	0.640	0.638	0.628	0.610	0.628	0.691
General Prompt	0.395	0.090	0.387	0.382	0.374	**0.325**	**1.000**	0.537	0.530	0.529	0.410	0.544	0.510	**0.692**
+ Mission Statement	0.381	0.173	0.355	0.399	0.345	**0.331**	**1.000**	0.397	0.393	0.347	0.372	0.372	0.376	**1.000**
+ Human Heuristic	0.393	0.204	0.243	0.391	0.340	**0.315**	**0.470**	0.597	0.580	0.572	0.536	0.597	**0.576**	0.667
+ Labeling Function Exemplars	0.394	0.172	0.169	0.165	0.287	0.237	**1.000**	0.544	0.527	0.525	0.485	0.525	0.521	**0.811**
+ Data Exemplars	0.395	0.134	0.378	0.389	0.383	**0.336**	**1.000**	0.477	0.458	0.421	0.404	0.471	0.446	**1.000**

Table 2: Performance of label models across different type of prompting strategies.

the performance of the label model on the testing dataset. We compute model performance across different label models for each type of prompting strategy.

The results are shown in Table 2. We find that the performance of the label model using synthesized LFs is generally on par with that of the label model using human-designed LFs while achieving much higher coverage. In particular, coverage on the SMS and Spouse datasets are low when using human-designed LFs from WRENCH; however, when using our synthesized LFs, *we achieve* 100% *coverage while also achieving higher F1-scores.* These results suggest that synthesized LFs can be a valuable resource for PWS pipelines and provide strong evidence for the efficacy of ScriptoriumWS in practical applications.

How does prompting strategy affect performance? Different prompting strategies can lead to LFs that are more or less aligned with the user's prior knowledge and expertise, which in turn can affect the quality of LFs and their performance in downstream pipelines. For instance, providing labeled data in the prompt can prime Codex with information about the relationships between the input features and the target labels. On the other hand, providing heuristic rules in the prompt can lead Codex to focus more on the user's prior knowledge. We initially hypothesized that these different prompting strategies would lead to a discernible pattern—some strategies would dominate in certain settings. However, in our experimental results shown in Table 2, suggest no such pattern, leading to an inconclusive result. It is important to carefully consider the goals and requirements

	IMDb (Accuracy)								Yelp (Accuracy)							
	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage
WRENCH	5	0.758	0.754	0.754	0.754	0.756	0.755	0.876	8	0.722	0.649	0.694	0.807	0.737	0.722	0.828
+ General Prompt	+6	0.740	0.737	0.742	0.767	0.739	0.745	**1.000**	+11	0.750	0.671	0.704	0.801	0.730	**0.731**	**1.000**
+ Mission Statement	+5	0.732	0.756	0.761	0.767	0.747	0.753	**1.000**	+7	0.734	0.660	0.693	0.815	0.753	**0.731**	**1.000**
+ Human Heuristic	+6	0.763	0.769	0.771	0.785	0.757	**0.769**	**1.000**	+5	0.680	0.619	0.661	0.804	0.703	0.693	**1.000**
+ Labeling Function Exemplars	+5	0.737	0.746	0.750	0.786	0.735	0.751	**1.000**	+5	0.656	0.664	0.706	0.808	0.711	0.709	**1.000**
+ Data Exemplars	+5	0.770	0.752	0.757	0.758	0.767	**0.761**	**1.000**	+6	0.724	0.656	0.705	0.821	0.748	**0.731**	**1.000**
	SMS (F1-score)								YouTube (Accuracy)							
	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage
WRENCH	73	0.678	0.772	0.756	0.750	0.057	0.603	0.405	10	0.808	0.732	0.808	0.828	0.788	0.793	0.893
+ General Prompt	+8	0.720	0.542	0.750	0.709	0.473	**0.639**	**1.000**	+9	0.832	0.740	0.788	0.820	0.756	0.787	**1.000**
+ Mission Statement	+8	0.619	0.405	0.672	0.576	0.420	0.538	**1.000**	+9	0.820	0.732	0.756	0.784	0.716	0.762	**1.000**
+ Human Heuristic	+9	0.632	0.476	0.582	0.594	0.482	0.553	**1.000**	+8	0.808	0.756	0.808	0.816	0.772	**0.792**	**1.000**
+ Labeling Function Exemplars	+8	0.405	0.465	0.473	0.610	0.029	0.396	**1.000**	+6	0.776	0.756	0.780	0.796	0.780	0.778	**1.000**
+ Data Exemplars	+8	0.692	0.418	0.746	0.722	0.509	**0.617**	**1.000**	+8	0.800	0.756	0.800	0.780	0.788	0.785	**1.000**
	Spouse (F1-score)								AGNews (Accuracy)							
	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage	#LFs	Snorkel + LR	WMV + LR	MV + LR	DS + LR	FS + LR	Average	Coverage
WRENCH	9	0.220	0.179	0.181	0.166	0.268	0.203	0.258	9	0.825	0.823	0.827	0.829	0.817	0.824	0.691
+ General Prompt	+8	0.157	0.298	0.303	0.301	0.155	**0.243**	**1.000**	+8	0.806	0.823	0.825	0.751	0.817	0.804	**1.000**
+ Mission Statement	+9	0.101	0.308	0.301	0.314	0.195	**0.244**	**1.000**	+4	0.684	0.713	0.714	0.726	0.719	0.711	**1.000**
+ Human Heuristic	+8	0.058	0.213	0.218	0.299	0.104	0.178	**1.000**	+4	0.813	0.811	0.812	0.766	0.806	0.802	**1.000**
+ Labeling Function Exemplars	+5	0.147	0.152	0.154	0.148	0.093	0.139	**1.000**	+8	0.784	0.797	0.795	0.794	0.790	0.792	**1.000**
+ Data Exemplars	+8	0.192	0.301	0.300	0.308	0.164	**0.253**	**1.000**	+5	0.711	0.726	0.725	0.712	0.737	0.722	**1.000**

Table 3: Performance of end models across different type of prompting strategies.

of the task and choose a prompt that is suitable for the task. A deeper analysis that elucidates the successes and failure modes of each prompting strategy is required to evaluate our original hypothesis and, perhaps more broadly, to better understand the role of prompting in code generation.

Can end model performance be improved by combining ScriptoriumWS with PWS? End models can only be trained on points that receive labels. Building on our observation that synthesized LFs offer high coverage, we hypothesize that end model performance can be improved over the standard PWS pipeline by simply including the examples that are labeled by ScriptoriumWS. This yields a complementary approach that incorporates both our synthesized LFs for points that are not labeled by human-designed LFs, and the labels that were originally produced by human-designed LFs. We train the end model on the union of these two sets. In Table 3, we show the performance of the end model when using this approach. As before, the dataset is fully covered by this approach and the end-model performance improves due to the significant increase in labeled examples. This shows that ScriptoriumWS can be used complementarity with existing PWS pipelines, for which human-designed LFs have already been created to improve performance.

5 Conclusion

In this paper, we aim to reduce the human effort required to design weak supervision labeling functions. We propose a novel system, ScriptoriumWS, to leverage code-generation models to provide programming assistance to synthesize labeling functions (LFs) automatically. We study a variety of prompting strategies, propose a simple pipeline, and obtain promising results when comparing to human-designed labeling functions on the WRENCH weak supervision benchmark.

References

Túlio C Alberto, Johannes V Lochter, and Tiago A Almeida. Tubespam: Comment spam filtering on youtube. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pp. 138–143. IEEE, 2015.

Tiago A Almeida, José María G Hidalgo, and Akebo Yamakami. Contributions to the study of sms spam filtering: new collection and results. In *Proceedings of the 11th ACM symposium on Document engineering*, pp. 259–262, 2011.

Simran Arora, Avanika Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, and Christopher Re. Ask me anything: A simple strategy for prompting language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=bhUPJnS2g0X.

Stephen H Bach, Daniel Rodriguez, Yintao Liu, Chong Luo, Haidong Shao, Cassandra Xia, Souvik Sen, Alex Ratner, Braden Hancock, Houman Alborzi, et al. Snorkel drybell: A case study in deploying weak supervision at industrial scale. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 362–375, 2019.

Benedikt Boecking, Willie Neiswanger, Eric Xing, and Artur Dubrawski. Interactive weak supervision: Learning useful heuristics for data labeling. In *International Conference on Learning Representations*, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Nilaksh Das, Sanya Chaba, Renzhi Wu, Sakshi Gandhi, Duen Horng Chau, and Xu Chu. Goggles: Automatic image labeling with affinity coding. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 1717–1732, 2020.

A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Applied Statistics*, 28(1):20–28, 1979.

Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. 2022. doi: 10.48550/ARXIV.2210.15157. URL https://arxiv.org/abs/2210.15157.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, Lei Li, and Zhifang Sui. A survey on in-context learning. 2023. doi: 10.48550/ARXIV.2301.00234. URL https://arxiv.org/abs/2301.00234.

Sara Evensen, Chang Ge, and Cagatay Demiralp. Ruler: Data programming by demonstration for document labeling. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1996–2005, 2020.

Daniel Fu, Mayee Chen, Frederic Sala, Sarah Hooper, Kayvon Fatahalian, and Christopher Ré. Fast and three-rious: Speeding up weak supervision with triplet methods. In *International Conference on Machine Learning*, pp. 3280–3291. PMLR, 2020a.

Daniel Y. Fu, Mayee F. Chen, Frederic Sala, Sarah M. Hooper, Kayvon Fatahalian, and Christopher Ré. Fast and three-rious: Speeding up weak supervision with triplet methods. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020b.

Chufan Gao, Mononito Goswami, Jieshi Chen, and Artur Dubrawski. Classifying unstructured clinical notes via automatic weak supervision. *arXiv preprint arXiv:2206.12088*, 2022.

Jiacheng Li, Haibo Ding, Jingbo Shang, Julian McAuley, and Zhe Feng. Weakly supervised named entity tagging with learnable logical rules. *arXiv preprint arXiv:2107.02282*, 2021.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9), jan 2023. ISSN 0360-0300. doi: 10.1145/3560815. URL https://doi.org/10.1145/3560815.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*, 2022.

Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 11, pp. 269. NIH Public Access, 2017.

Alexander Ratner, Braden Hancock, Jared Dunnmon, Frederic Sala, Shreyash Pandey, and Christopher Ré. Training complex models with multi-task weak supervision. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 4763–4771, 2019.

Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems*, 29, 2016.

Wendi Ren, Yinghao Li, Hanting Su, David Kartchner, Cassie Mitchell, and Chao Zhang. Denoising multi-source weak supervision for neural text classification. *arXiv preprint arXiv:2010.04582*, 2020.

Nicholas Roberts, Xintong Li, Tzu-Heng Huang, Dyah Adila, Spencer Schoenberg, Cheng-Yu Liu, Lauren Pick, Haotian Ma, Aws Albarghouthi, and Frederic Sala. AutoWS-bench-101: Benchmarking automated weak supervision with 100 labels. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=nQZHEunntbJ.

Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, ICER '22, pp. 27–43, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391948. doi: 10.1145/3501385.3543957. URL https://doi.org/10.1145/3501385.3543957.

Ryan Smith, Jason A Fries, Braden Hancock, and Stephen H Bach. Language models in the loop: Incorporating prompting into weak supervision. *arXiv preprint arXiv:2205.02318*, 2022.

Paroma Varma and Christopher Ré. Snuba: Automating weak supervision to label training data. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 12, pp. 223. NIH Public Access, 2018.

Harit Vishwakarma, Nicholas Roberts, and Frederic Sala. Lifting weak supervision to structured prediction. *arXiv preprint:2211.13375*, 2022.

Shuohang Wang, Yang Liu, Yichong Xu, Chenguang Zhu, and Michael Zeng. Want to reduce labeling cost? GPT-3 can help. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 4195–4205, Punta Cana, Dominican Republic, November 2021a. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.354. URL https://aclanthology.org/2021.findings-emnlp.354.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. 2022. doi: 10.48550/ARXIV.2201.11903. URL https://arxiv.org/abs/2201.11903.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 1–10, 2022.

Jieyu Zhang, Yue Yu, Yinghao Li, Yujing Wang, Yaming Yang, Mao Yang, and Alexander Ratner. Wrench: A comprehensive benchmark for weak supervision. *arXiv preprint arXiv:2109.11377*, 2021.

Xinyan Zhao, Haibo Ding, and Zhe Feng. Glara: Graph-based labeling rule augmentation for weakly supervised named entity recognition. pp. 3636–3649, 01 2021. doi: 10.18653/v1/2021.eacl-main.318.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. 2022. doi: 10.48550/ARXIV.2211.01910. URL https://arxiv.org/abs/2211.01910.