**ScatterShot: Interactive In-context Example Curation for Text Transformation**

Tongshuang Wu*
sherryw@cs.cmu.edu
Carnegie Mellon University
USA

Hua Shen
huashen218@psu.edu
Pennsylvania State University
USA

Daniel S. Weld
weld@cs.uw.edu
University of Washington
USA

Jeffrey Heer
jheer@cs.uw.edu
University of Washington
USA

Marco Tulio Ribeiro
marcotcr@microsoft.com
Microsoft Research
USA

**ABSTRACT**

The in-context learning capabilities of LLMs like GPT-3 allow annotators to customize an LLM to their specific tasks with a small number of examples. However, users tend to include only the most obvious patterns when crafting examples, resulting in underspecified in-context functions that fail short on unseen cases. Further, it is hard to know when “enough” examples have been included even for known patterns. In this work, we present ScatterShot, an interactive system for building high-quality demonstration sets for in-context learning. ScatterShot iteratively slices unlabeled data into task-specific patterns, samples informative inputs from under-explored or not-yet-saturated slices in an active learning manner, and helps users label more efficiently with the help of an LLM and the current example set. In simulation studies on two text perturbation scenarios, ScatterShot sampling improves the resulting few-shot functions by 4-5 percentage points over random sampling, with less variance as more examples are added. In a user study, ScatterShot greatly helps users in covering different patterns in the input space and labeling in-context examples more efficiently, resulting in better in-context learning and less user effort.

**ACM Reference Format:**


**1 INTRODUCTION**

In-context learning [70] is a property of Large Language Models (LLMs), where a user can “write” a transformation function via an (optional) short set of instructions and a few (input, output) examples. For example, writing a function that “translates” a holiday name (e.g., “Christmas”) into its calendar date (e.g., “12/25”) would previously require a complicated rule-based system capable of mapping various kinds of subtly different inputs (e.g., “Xmas”, “Christmas day”, etc) to a lookup table of dates. With LLMs like GPT-3 [7], the process is much simpler. A user can achieve the same functionality with a prompt (i.e., a natural language instruction) that contains a small number (e.g., two) of simple demonstrations, followed by a query (underlined): “Christmas => 12/25; Halloween => 10/31; Independence Day (US) =>”. GPT-3 would take the prompt and return the right date “7/04” for this query. More impressively, LLM will also have some generalizability towards semantically relevant queries, e.g., queries with abbreviations ("xmas => 12/25", "nye => 12/31"), misspellings ("s patrics day => 03/17"), lesser-known name variations ("All Saints’ Eve => 10/31"), and holidays that might be overlooked (e.g., "Harriet Tubman Day => 3/10"). The much reduced programming effort (compared to e.g., rule-based systems) draws users’ attention towards building their personalized in-context functions in various use scenarios, including code generation, question answering, creative writing, and others [36, 54, 64].

Although in-context learning has great potential, the quality of the learned function for non-trivial tasks depends on which in-context examples are used as demonstrations [32, 46]. Techniques for automatic example selection [30] depend on existing labeled datasets and tasks that can be evaluated automatically (e.g., classification), and thus users “in the wild” rely on their own ingenuity and intuition when coming up with demonstrations [21]. Unfortunately, users tend to focus on the most obvious and memorable patterns for demonstration [18], leading to systematic omissions [66] and underspecification that might go unnoticed. As an example, in Figure 1 we use in-context learning to build a function to extract and normalize temporal information from a sentence [9]. Most users would provide demonstrations with common date formats (e.g., “Oct. 23, 1999”), and some might remember relative date references (e.g., “today”). However, some patterns are easy to miss, e.g. long-form dates with no capitalization or holidays (e.g., “nineteen ninety-six”, “Thanksgiving Day” in Figure 1C), and the LLM may fail to learn them if they are omitted. Even sampling random examples from the unlabeled data might lead to the repetition of common patterns (Figure 1B) at the expense of demonstrating less-common ones. What
is worse, users may not know when they have provided enough examples, and whether there are any uncovered patterns in the unlabeled data. As a result, prior work summarized the two major pain points of prompting to be (1) the effort required to source examples for a prompt, and (2) the difficulty of evaluating whether a prompt is improving [22].

In this work, we present ScatterShot, an interactive system for building high-quality demonstration sets for in-context learning. In a nutshell, ScatterShot helps users find informative input examples in the unlabeled data, annotate them efficiently with the help of the current version of the learned in-context function, and estimate the quality of said function. In each iteration, ScatterShot automatically slices the unlabeled data into clusters based on task-specific key phrases [66, 69]. For example, given the existing examples in Figure 1A, it finds a cluster based on holiday key phrases (“Christmas”, “Thanksgiving”, etc.) and a cluster based on exact dates like “Oct. 23, 1999” (among others). ScatterShot keeps a running estimate of the error of each cluster, and thus prioritizes examples from clusters that have not yet been explored or learned effectively. It further uses the stability of the current in-context function with respect to minor changes in the prompt (e.g. ordering of in-context examples), prioritizing unlabeled examples that get different predictions with different prompt variations. Users are then presented with examples of underexplored clusters (e.g., Figure 1 C1), or hard examples of explored clusters (e.g., C2, hard because the past tense refers to the Thanksgiving date of the previous year). Instead of having to label demonstrations from scratch, users can either accept correct predictions from the current function (Fig 1 C1) or make edits to fix wrong predictions (Fig 1 C2). These additional labels are used to update the in-context function, such that the user explores the different possible input patterns in an interactive manner, without wasting resources on patterns that have already been learned.

We evaluate ScatterShot both in terms of sampling efficiency and support for human annotators. In simulation experiments, we compare the sampling strategy in ScatterShot to random sampling on two text transformation tasks contemplated in prior work: the data wrangling task illustrated in Figure 1 [9], and rewriting question-answer pairs into logically equivalent pairs in order to evaluate model consistency [44]. In both cases, we find ScatterShot improves performance on corresponding metrics (e.g., Rouge-L, F1) by 4-5 percentage points, with less variance for various values of k demonstrations. Further, we conduct a within-subject user study in which 10 participants build in-context functions for the QA-pair rewriting task either (1) manually, (2) with the ScatterShot interface but random sampling, or (3) with the fully-featured ScatterShot. We show that ScatterShot’s interface alone is an improvement, by offloading input selection and providing sample outputs. Moreover, the sampling strategy in the fully-featured ScatterShot helps users notice diverse input patterns, leading to improvements in the resulting in-context function. For example, participants who thought their in-context examples were sufficient when using random samples labeled an additional 1.4 times of examples after switching to full ScatterShot (as they found new patterns), which further improved the function test performance. We conclude the paper with insights into challenges and opportunities that arise from our experiments, including e.g., explaining the sampling rationales, incorporating automated blind-spot detection, and the potential of using a ScatterShot setup to help users iteratively refine their task definition during data collection.
2 THE DESIGN OF SCATTERSHOT

The goal of SCATTERSHOT is to help users iteratively find and label high-quality demonstrative examples to build effective in-context functions. In order to be effective, a function must be able to handle common patterns (e.g., the temporal normalization function in Figure 1 must be able to handle common temporal expressions such as “today”), without neglecting less common ones (e.g., holidays such as “Christmas”). Further, we want the process to be cost-effective, not wasting annotation effort on demonstrations that are redundant with already covered patterns. To achieve these goals, we design SCATTERSHOT to respond to every user interaction by offering assistance in three areas:

- **Help the user discover previously unexplored patterns.** In each iteration, SCATTERSHOT uses existing demonstrations and users’ past interactions to cluster the remaining unlabeled data into task-specific slices. Such slices map the input space for users to explore.

- **Help the user prioritize the most informative examples.** SCATTERSHOT uses the current in-context function to estimate the difficulty of slices and examples, prioritizing unexplored slices or slices and examples where the current function is not yet performing well. We call this variant of active learning slice-based sampling.

- **Minimize annotation cost.** Rather than providing a gold output label from scratch for each example, the user is presented with the best guess output from the current in-context function (updated at every step), which they either accept when correct or edit the incorrect parts.

We wrap these functionalities with a lightweight interface, where at each round, users are presented with a batch of unlabeled examples to be (potentially) added to the set of demonstrations. Thus, at each round, users get a “picture” of their current in-context function, and interact with it for improvement. We now detail each one of these components.

### 2.1 Interactive Interface

We present SCATTERSHOT as an interactive interface, shown in Figure 2. The interface contains a task description (A1) and existing in-context examples as demonstrations, presented as input-output pairs (A2). These pairs are color-encoded based on the text editing distance, with the spans deleted from the input in red, and the spans added in green. Both the description and demonstrations are editable, and are automatically translated into an LLM prompt (Figure 2B) with the task description, demonstrations in the format » [example input] => [example output], and a candidate input for the LLM\(^1\) to transform into an output.

Below the existing examples, SCATTERSHOT proposes a batch of 5 candidate inputs sampled from the unlabeled dataset, with outputs computed with the current version of the in-context function (A3), using the prompt in Figure 2B. Users then verify the candidates and provide feedback (A4), editing outputs to fix mistakes when needed (e.g., changing from “Thanksgiving == 2000-11-25” to “Thanksgiving == 1999-11-25”, A4), and adding or removing examples to the few-shot examples for in-context learning (A5). In addition to saving annotation time, LLM-generated outputs help users assess the quality of the current version of the in-context function. For example, if all LLM outputs are correct for a few consecutive batches, it is likely that the existing few-shot examples cover the patterns in the unlabeled data, and thus labeling can stop.

The interface is task-agnostic and can be used whenever users want to learn one-on-one text mapping between text inputs and outputs. This format is flexible, encompassing both classification tasks (where the output is just the class name) and generation tasks.

\(^1\)All of our studies and experiments are run on GPT-3 [?]. https://beta.openai.com/

![Figure 2: (A) The SCATTERSHOT interface, with (A1) task description, (A2) existing in-context examples, and (A3) candidate examples awaiting human inspection. Through interactions A1 and A3, users can make edits to LLM outputs, sort the candidates into positive demonstrative examples (+), negative ones (-), or just not include the candidate (O). The description and the examples get transformed into raw text prompts. One set of in-context examples produces multiple prompts depending on how the examples are ordered; (B) shows a prompt with one possible ordering.](image-url)
2.2 Slice-based Sampling

2.2.1 Identifying patterns with key phrase clustering.

To help users explore both common and less common patterns, we need a way to partition the unlabeled input examples. While there are a variety of task-agnostic distance metrics that could be used for clustering (e.g., cosine similarity of sentence embeddings [43]), our preliminary exploration indicated that these are typically too coarse when applied to specific tasks. For example, using the embeddings from Reimers and Gurevych [43], “Took a photo today.” is closer to “Saw a photo on Flickr.” (similarity = 0.56) than to “Are you going to yoga class today?” (similarity = 0.30). While this may make sense in the abstract, it does not correspond to how we would want to slice examples for the temporal extraction task in Figure 1, where date references “today” are more important than subject matter (“photos” vs “yoga class”). Thus, we propose a task-specific clustering method based on key phrases as explained below.

Detecting key phrases in demonstrations. While key phrase extraction in general may require domain knowledge [8, 42, 65], for text transformation we can leverage the signal present in the relationships between input and output, i.e., in which parts of the input are perturbed or retained. For example, “today” is retained in the output of both “Took a photo today.” and “Are you going to yoga class today?” (among many other samples), and thus it is probably a key phrase. Formally, given a labeled, positive example, i.e., a pair of original and perturbed sentences $f(x) \Rightarrow y$, we extract as key phrases either the unmodified parts of $x$ when most of $x$ is changed (Levenshtein edit distance $d(x, y) \geq 0.5$, as is the case with the “today” examples above), or the modified parts when most of $x$ remain unchanged.

Applying key phrases to unlabeled inputs. Applying key phrases naively with an exact match would yield low coverage in the unlabeled data (especially for larger phrases). To get more coverage, at each iteration, we generalize key phrases extracted from labeled demonstrations into templates with combinations of tokens, lemmas, and part-of-speech tags [66, 69], e.g., “today” is expanded into today, NOUN, and DATE. We then select representative templates with a greedy weighted set coverage algorithm based on their specificity and the number of inputs they cover [59]. Example templates at various abstraction levels are shown in Figure 3A.

Key phrase clustering. We define the distance between two inputs as the minimum cosine distance between the sentence embeddings [43] of their key phrases, and use agglomerative clustering [33] to recursively merge pairs of clusters in the unlabeled data. We set the number of clusters to 20 (chosen empirically in Section 3), and aggregate all clusters with $< 10$ examples into a single “outlier” cluster (Figure 3B). Note that we recompute clusters in every iteration, and thus the outlier cluster tends to shrink as the user interacts with the system. Figure 3B contains various examples of discovered clusters.

Note that as a result of the weighted coverage selection, the templates — and thereby the extracted key phrases — are dynamically changing, and will eventually become more dominant in the sampling procedure: when the few-shot set contains only a few
(e.g., 3) seeding examples, the templates might be biased or even non-existent, most examples will just use the full sentences as key phrases, making it similar to vanilla clustering on full examples. However, as we add more examples, the templates will be more balanced and eventually stabilize, at which point the clustering can rely more on the extracted key phrases.

2.2.2 Selecting slices for exploration.

We want to explore the identified slices in an efficient way, avoiding slices already “solved,” and making the user discovers any unexplored patterns. We take inspiration from the UCB algorithm [4], and use an upper bound estimate of the error of our function in each slice as part of the “reward” for sampling from that slice. Formally, suppose slice c has n examples, m of which are labeled in previous iterations (see the next section for “labeling” details). Further, suppose that out of the m previously labeled examples, the current function is correct on k. The reward of drawing from slice c at iteration i is then given by:

\[ \mu_{i,c} = \left( 1 - \frac{k}{m} \right) \cdot \ln n + \frac{\ln i}{m} \]

In other words, we prioritize large slices (ln n), low performance \((1 - k/m)\), and slices that have not been sampled many times \((\ln i/m)\), which would give higher weights to clusters with smaller m as the iteration i progresses. Thus, we avoid wasting annotation effort on slices that are already “solved”, but keep drawing from slices we can’t yet deal with and slices we have not yet explored.

Figure 3B shows four data slices in temporal extraction ranked by reward \(\mu\). \(\bigcirc\) is the “outlier” cluster, where patterns are not yet apparent. This slice still gets prioritized due to its large size \((n = 449)\), even though it has been sampled \(m = 10\), which encourages either higher accuracy or further slicing in follow-up iterations. \(\bigotimes\) is a slice with holiday-based key phrases. Though the slice is small \((n = 19)\), the LLM failed whenever it was previously sampled \((k/m = 0)\), and thus it currently represents a hard pattern. \(\bigodot\) is a slice with past date references, while \(\bigcirc\) is a slice with the common temporal pattern represented by the words “today”, “yesterday”, and “tomorrow”. This last slice has low priority despite being common, since the LLM had perfect accuracy whenever a sample from it was drawn. To maximize diversity (similar to batched active learning [12, 17, 48]), we rank the slices by reward and select one example from each until the batch is filled (in our case, batch size = 5).

2.2.3 Saving user effort with implicit labels and pseudo-labeling.

As mentioned above, our per-slice performance estimation requires labeled examples. Unfortunately, we only have firm labels on user-added in-context examples, which may be quite small, especially if users only add a portion of the sampled data. As a result, in-context examples offer limited power for estimation. Although we can modify the interface to collect additional user labels on output correctness, it requires additional interaction that can be cumbersome. To save user effort, we use implicit labeling, i.e., we label the LLM output of an example in a batch as correct if the user does not make any changes to the output, even if they do not add it to the in-context demonstration set. Of course, users might ignore model errors if they are frustrated or distracted, but we verified in pilot experiments that users almost always make corrections in the presence of model mistakes (~87% of the time, and the selection method is robust to this small amount of noise). In comparison to explicit labeling, this method requires the bare minimum user interaction, and is easier to integrate into iteration workflows.

Still, implicit labeling requires users to actually see and interact with a sample. However, after a certain point in the process, the LLM is correct often enough that many interactions would simply be “accepted” (no changes) by the user. While important for estimating slice accuracy, too much of such interaction might also lead users to overestimate the in-context function quality, and stop the process before they explore the remaining slices. Thus, after we reach a threshold of quality (LLM is correct on 70% of examples in two consecutive rounds), we start leveraging pseudo-labeling with unanimity voting, a method inspired by the unanimity principle [23] and Query-by-Committee [34]. Following Lu et al. [32]’s observation that the order of in-context demonstrations can drastically change LLM performance, we form three different prompts by randomly reordering the examples. When the outputs of the prompts agree (i.e., are unanimous), we use that output as a pseudo-label, used both for estimating slice accuracy and as a filtering method (i.e., these examples are not shown to the user). Figure 4 illustrates this process,
where “@viereedom Merry Christmas” (A) is pseudo-labeled due to unanimity, and “Atlanta nineteen ninety-six” (B) yields different predictions, and thus is shown to the user for manual inspection.

3 SIMULATION EXPERIMENT: SCATTERSHOT SAMPLING VS. RANDOM SAMPLING

In this section, we measure the effectiveness of slice-based sampling, when compared to random sampling on two text transformation tasks. We use datasets for which we have labels on both tasks, so that we can simulate the labeling process with an oracle at scale, and evaluate the learned function on a held-out portion of each dataset.

3.1 Tasks and Datasets

Temporal expression extraction and normalization. The Temporal task involves data wrangling [60], where the goal is extracting phrases with temporal expressions from sentences or documents, and normalizing them into a standard format [9]. As shown in Figure 1, these can include absolute or relative dates, and can have different granularity (e.g., exact date vs. year only).

Data. We take the data from [2], containing temporal expression datasets, including TimeBank [41] (news articles) and TweetTime [55] (tweets). We process each dataset into sentences, discarding any date annotations that could not be normalized to the format YYYY-MM-DD (for consistency), and keeping sentences involving absolute dates, dates relative to the document publication date, or no time expressions at all (as the pool for negative examples). This resulted in 491 examples with YYYY-MM-DD outputs, and 369 negative examples with the output N/A. We sample 100 examples randomly from this dataset as a test set, and use the remaining pairs as our unlabeled pool in the experiment.

Evaluation. Following Chang and Manning [9], we report F1, recall, and accuracy both for the temporal expression extraction and normalization separately.

Question-Answer Pair Implication. For the QA-pair task, we use ScatterShot to replicate transformation functions from prior work. Given a question-answer (QA) pair, Ribeiro et al. [44] wrote a rule-based system (over 1,000 lines of code) to generate a new QA pair that is implied by the original pair, to check whether question answering systems are consistent in their reasoning. We replicate their logical equivalence transformation, where the original QA is rewritten to a logically equivalent form, e.g., “Q: What room is this? A: bathroom” is transformed to “Q: Is this a bathroom? A: yes”. Despite the heavy engineering, the rule-based system is not able to cover many inputs, and often produces text that does not look fluent or natural. We thus apply in-context learning to this task, and use ScatterShot to select the examples.

Data. We download the input sentences and rule-based implications from Ribeiro et al. [44], and manually inspect and label 1,000 randomly sampled QA pairs (351 rule-based implications were noisy and had to be relabeled). We randomly sample 100 pairs as a test set, and use the remaining pairs as our unlabeled pool in the experiment.

Evaluation. We follow the standard in text generation and report the Rouge-L F scores [28], as well as BLEU-4 [28].

3.2 Procedure and Baseline

We compare ScatterShot’s slice-based sampling with a Random sampling baseline, which is the most common sampling method used especially in complex tasks, e.g., in text translation [1]. We use GPT-3 as our underlying LLM, with greedy decoding (temperature=0) in both conditions. In each simulation run, we start the process with three random samples (the same for both conditions) of input-output. At every iteration, we compare the ground truth label with the candidate label proposed by the current in-context function. When the labels differ, we add the pair (input, oracle output) to the in-context example set, simulating the case where the user

https://github.com/marcotcr/qa_consistency/
corrects a transformation and adds it to the set; Otherwise, the oracle user does not perform any action, simulating cases where the user ignores examples where the current in-context function is correct.

The process is repeated until one of the following stopping conditions is met: (1) the in-context example set contains more than 40 data points (exceeding the LLM maximum context size), (2) The oracle user has been presented with 100 examples (i.e. annotation budget is met), (3) the in-context function provided the correct output in five consecutive iterations, or (4) the in-context function’s estimated accuracy for all slices of data is ≥ 80%.

We run ten simulation rounds with different random seeds, and report the (averaged) final function performance. We further track the function improvement trajectory over iterations on three randomly selected simulation rounds, by evaluating the intermediate in-context functions after every five examples are added.

3.3 Results
As Table 1 shows, ScatterShot’s slice-based sampling outperforms the baseline on both tasks. In Temporal, ScatterShot improves the \( F_1 \) for date span extraction by around 2 points, and the normalization by 4 points. In QA-pair, ScatterShot outperforms Random by 6 points on Rouge-L, and even outperforms the heavily engineered rule-based system used to label most of the evaluation data, despite needing 40 or fewer in-context examples. Table 2 shows qualitative examples, where ScatterShot outperforms both baselines in terms of coverage, fluency, and correctness. These results point to ScatterShot’s potential on saving human efforts in creating fine-grained functions, alleviating the need for handcrafting templates.

Figure 6 shows the trajectory of the in-context function quality as the simulated user adds more examples, for three randomly selected runs. ScatterShot dominates the baseline at almost all points in all runs, with the biggest gaps occurring when the number of in-context examples is small. We see particular gains at early stages, as the simulated user adds more examples, for three randomly selected runs.
Finally, we observe that ScatterShot is less liable to variance in quality as more examples are added (e.g. in QA-pair-2, baseline performance degrades by almost 15 points between $n = 20$ and $n = 30$). These results suggest that besides its interface and interactivity benefits, ScatterShot can improve in-context learning just by virtue of its sample selection function. In order to evaluate the benefits to actual humans, we now turn to a user study.

4 USER STUDY

ScatterShot sampling is effective in simulation, but does it actually aid humans to articulate their desired functions? We conducted a within-subject user study to evaluate whether human users can sense ScatterShot’s support in exploring the data space.

4.1 Study Design

**Task & Participants.** We ran a user study on the QA-pair task using the same dataset as Section 3.1, with a split of 900 unlabeled inputs for participants to access, and 100 test examples for evaluating the in-context functions they built. We recruited ten CS graduate student participants (4 females, 6 males) on our CSE department mailing list. Eight of them had previously used GPT-3 and two had heard about it, but none were familiar with the task or ScatterShot. Each participant spent around 60 minutes in the study.

**Settings & Conditions.** In order to isolate the effect of the different components in ScatterShot, we have two ablation settings in addition to our method: (1) **Manual**, where participants manually craft prompts without any help from ScatterShot, which is the de-facto status-quo of practitioners creating their own in-context learning examples. (2) **Random**, where participants use the ScatterShot interface with slice-based sampling disabled, i.e., they review randomly selected examples. This condition still has the benefit of an interactive interface, and uses the intermediate in-context functions to suggest outputs and pseudo-label. (3) ScatterShot, where participants have access to ScatterShot, fully featured.

Every participant interacted with every setting in sequence and in a cumulative manner, i.e., the in-context demonstrations gathered in one setting carry over to the next, and we measured the additional benefit of moving to the next setting. We divided the participants into two groups, such that in one group the sequence is Manual $\rightarrow$ Random $\rightarrow$ ScatterShot (M-R-S), while in the other it is Manual $\rightarrow$ ScatterShot $\rightarrow$ Random (M-S-R). M-R-S represents a condition where participants are gradually exposed to more features, such that the step-wise gain maps directly to the benefit of the new feature, while M-S-R serves as the counterbalanced condition that combats the learning effect and the natural impact of accumulating examples on function qualities.

**Study Procedure.** We designed our hour-long study to be self-contained in a Jupyter Notebook,4 and one of the authors was present in all studies to ensure that participants understood the task and to answer any questions.

Participants were first introduced to the basic concepts of LLM (GPT-3), in-context example construction, and the study task. Then, we randomly assigned the participants to one of the two conditions (M-R-S or M-S-R), and they completed the task by going through the three conditions in the assigned order. Participants were not instructed on the difference between ScatterShot and Random, and were instead told that “these two selection methods are randomly ordered, and one is not necessarily better than another.”

In each step (setting), participants were told to inspect the inputs and current function outputs (available in ScatterShot and Random), fix the erroneous outputs, and add demonstrations (input-output pairs) to the in-context example bucket if they believed the data

---

4The full user study instructions, as well as the detailed exit survey, are in https://github.com/tongshuangwu/scattershot
would add additional value, e.g., instances where the current context function fails, as well as diverse input or output patterns. They were asked to iterate within the step until they were satisfied with the in-context function at hand, or accumulated 40 examples. To prevent them from stopping too early, we also asked them to run at least three batches (i.e., see 10-15 examples). Afterward, participants completed an exit survey and a semi-structured interview, where they rated their perceived experience in each of the two consecutive steps. These questions concerned their perceived input/output pattern diversity, the example difficulty, and their confidence in estimating in-context function quality.

**Collected Data.** We observed and analyzed three sets of data. First, to quantify the change in function quality, we saved participants’ in-context examples per step, and applied them to the held-out test set. Here, besides the absolute numbers as in Section 3, we calculated the difference in performance between two consecutive steps to see if adding (or, in the case of M-S-R, removing) ScatterShot features impacted the quality of examples participants submitted. Second, to assess participants’ self-perceived experience, we used a standard five-point Likert Scale [27] to collect their perceived step-wise differences. Third, to track participants’ annotation trajectories, we logged their clickstreams in all the steps. This included both the number of examples they examined per step, the edits they made, and the number of examples they added.

### 4.2 Results

The ScatterShot interface made it easier to iterate on in-context examples. As shown in Figure 7, participants found moving from Manual (Step 1) to a ScatterShot interface (Step 2) beneficial, regardless of the sampling setting. In particular, they found that the interface made it easier and more intuitive to construct the few-shot examples. (Easier to use in Figure 7, 4.7 ± 0.7 for Manual→Random and 4.2 ± 0.4 for Manual→ScatterShot). Users liked the fact that ScatterShot offers sample inputs (rather than having to go through the dataset on their own), and that the interface provides easy access to all the existing in-context examples, allowing for fast back-and-forth iteration. For example, one participant (P7) kept revisiting their examples, and removed some earlier examples that they thought were less useful as they became more familiar with the unlabeled input space.

As part of the interface, LLM-generated outputs helped participants craft examples more efficiently, e.g., P6 comments that “it is less work to make edits than starting from scratch.” Somewhat surprisingly, LLM-generated outputs also improved output diversity, i.e., users considered more diverse output patterns. For example, P10 commented that they were “pleasantly surprised by the LLM’s clever output in several cases,” and that they would not have thought about transformations such as “Q: Is there more than 1 boy? A: no” → “Q: Is there no more than 1 boy? A: yes”, which they added to their set of in-context examples. The observation is consistent with prior work showing AI-induced creativity gains [62]. We note that actual user behavior here differs from our simulation setup, where we assumed human users would only add new examples when the LLM output was wrong.

Participants’ perceptions matched ScatterShot’s slice-based sampling design goals: more diverse and more challenging patterns. As shown in Figure 7, participants in M-R-S clearly noticed the improvement moving from Random→ScatterShot (4.2 ± 1.2 for more diverse patterns and 4.8 ± 0.4 for more difficult case), whereas most users in M-S-R did not report improvements from ScatterShot→Random. Qualitative results confirm this, e.g., P7 in M-R-S commented: “Step 2 (Random) provided me with some worthy examples, but much less than Step 3 (ScatterShot). I went through several rounds of pretty similar examples, thinking the function is behaving quite decently, and didn’t realize the function needed more diverse and edge cases until I reached Step 3.” P9 in M-R-S was also happy that ScatterShot helped them explore beyond typical patterns. In contrast, P10 in M-S-R reflected that their exploration seemed to have “quickly saturated in Step 3” (Random).

Despite not being given details, seven participants discerned the goals behind ScatterShot’s sampling method by interacting with it. For example, P2 described it as “sample for additional variation based on the patterns in existing examples, and also sample for examples similar to previous error mistakes to track whether the function

---

1. In Manual, this meant looking at three random batches of unlabeled data in the Jupyter notebook.
Table 3: The performances of participants’ in-context functions after each step. +/- represents the average performance change compared to the prior step, whereas the number in the parentheses are the absolute performances.  M-R-S participants were able to keep adding useful examples, whereas M-S-R participants decreased the function performance by 0.6 in Step three (ScatterShot→Random), indicating that these efforts were wasted.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Step 1</th>
<th>Step 2</th>
<th>Step 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-R-S</td>
<td>(59.3)</td>
<td>+17.4 (74.7)</td>
<td>+3.2 (77.8)</td>
</tr>
<tr>
<td>M-S-R</td>
<td>(61.8)</td>
<td><strong>+18.1</strong> (75.4)</td>
<td>-0.4 (74.9)</td>
</tr>
</tbody>
</table>

(a) ROUGE-L

---

SCATTERSHOT helped participants explore the input space more holistically, and build better in-context functions. The perceived data difficulty and diversity encouraged participants to iterate more on their in-context examples. When looking at the number of in-context examples added in each setting, participants added 40% more examples in ScatterShot than Random when ScatterShot came after (M-R-S), and 20% fewer examples in Random when Random came after (M-S-R), i.e., they stopped much earlier when Random came after ScatterShot. These additional examples are not only a result of more inspection effort (on average, participants in ScatterShot reviewed 20% more samples), but also that each batch in ScatterShot was more likely to contain a good in-context example — participants added 81% of the examples they inspected in ScatterShot, but only 48% of the examples in Random.

We report the quality of the resulting in-context function on the held-out set in Table 3, and note that Random→ScatterShot consistently increases performance, while ScatterShot→Random consistently decrease performance despite adding more in-context examples, which is in line with our simulation results.

SCATTERSHOT helped participants estimate function quality and “debug” their example set. As expected, participants estimated their in-context function quality based on the candidate examples they reviewed. For example, P5 (M-S-R) tracked the function convergence: “I made mental notes on the LLM errors and hypothesized what types of examples were missing. For example, I noticed the model was wrong on N/A questions at first, but later got it right.” Participants in M-R-S seemed slightly more satisfied with their estimation, with 4.2 ± 0.9 in Manual→Random and further 4.3 ± 0.7 Random→ScatterShot. P7 commented that “Step 2 showed me the function is quite smart on patterns it has already seen and has high precision, and Step 3 showed me there are more patterns and it has low recall”. P2 further reflected that Random’s sampling “created a false impression of convergence, when the function still had various blind spots.” The interactive process also helped participants debug their example sets, e.g., P4 saw big performance drops (4/5 to 1/5 accuracy) on two consecutive batches, which led them to remove in-context examples that were hurting performance.

Participants in M-S-R gave slightly lower ratings on their estimates. Qualitatively, the fact that SCATTERSHOT prioritized potential mistakes seemed to discourage users, e.g., P3 noted they were driven into “an endless blackhole of errors,” after which a round of repetitive patterns in Random was hard to make sense of. Once again, this could have been mitigated by explaining the sampling strategy to the users, and explicitly displaying the slice accuracy estimates SCATTERSHOT keeps track of.

5 DISCUSSION

In this work, we design a human-LLM collaboration mechanism in SCATTERSHOT to help humans craft diverse and informative in-context learning examples. By iteratively identifying data slices, sampling from low-performance or unseen slices, and providing best-guess outputs for the sampled examples, SCATTERSHOT not only helps the collection of informative in-context examples, but also supports users in exploring the input space and assessing the function quality. At its core, SCATTERSHOT is built on three concepts: data slicing and sampling, iterative human-model interaction, and collaborative human-model labeling. We now discuss challenges and potential future work for each of these.

Slice-based sampling can increase data space coverage. Our experiments showed that sampling from diverse and difficult data slices improves in-context function performance. Importantly, these slices cannot be surfaced via clustering on task-agnostic embeddings; rather, task-specific features should be considered to group examples, while task-irrelevant noise should be minimized. However, identifying these task-specific features remains a challenge. While effective for our function examples (and many others), keyword and template extraction would not generalize to tasks where input and output have little syntactic overlap, e.g., English-French translation, summarization, etc. Future work should look into incorporating more general slicing methods, e.g., asking practitioners for slicing functions [11, 42, 65], automatically detecting blind spots [16, 47], etc.

In addition to data slicing, the sampling algorithm also plays a crucial role in narrowing down the actual slices to sample from. We adapt the UCB algorithm to prioritize slice size, performance, and sample rarity, but there are other interesting dimensions that could be explored. For example, if there are slices that cannot be learned after several rounds of sampling, UCB may be counterproductive and create a biased in-context example set that performs worse on other slices, whereas a strategy that penalized or just “gave up” on those slices might produce a better overall function. Moreover, we might want to explore better methods for example ranking within a slice. 
**Interacting with the latest function is essential for in-context learning.** In-context learning enables rapid function updates, which are not possible in other current interactions with models (e.g., finetuning often takes long hours, and is often not suitable for interactivity). Allowing users to interact with the most current version of what is being learned helps them track progress, and backtrack when they introduce cascading errors [22]. The setup in ScatterShot is a step in this direction, since users always interact with the latest version of the in-context functions.

While participants were making progress with ScatterShot (more than with baselines), some participants felt frustrated by inspecting mistake after mistake, fearing that they would never be able to produce a good enough function. While this is by design (ScatterShot prioritizes potential errors), it might compromise annotators’ estimates of the quality of their function, and their motivation for labeling more examples. Thus, we notice the importance of presenting quality metrics to the user and clearly explaining the sampling function so that the right expectations are set. For example, users may perform better mental calibration if they have access to hints like the number of slices that are considered “solved” (e.g., as a progress bar that allows people to zoom into concrete examples grouped by the slice), cross-validation accuracy on in-context examples, etc. Another alternative would be to let users exercise more control over which slices are explored, e.g., allowing them to “drill down” or “give up” on specific slices.

**Human-AI collaborative labeling for building better functions with respect to better quality and better task definition.** Essentially, ScatterShot enables human-LLM collaboration on data annotation. In our work, we mostly focused on evaluating the quality benefit of such annotation, but we observed additional interesting gains in bringing people inspiration. In Section 4, we notice that participants can take inspiration from the LLM not only on the input patterns, but also on potential output patterns even though our QA-pair task is relatively deterministic in its transformations. Thus, we hypothesize that similar systems supporting human-LLM collaborative labeling could play an important role in helping users iteratively refine their task definition and function behavior during data collection. Prior work has shown that annotation requesters refine their labeling instructions when they see noisy (and therefore unusable) crowdsourced labels on ambiguous examples. However, we have yet to examine how LLMs’ suggestions (good or bad) might help users better specify their functions. It would be interesting to systematically analyze and measure users’ own distribution shift as the example set expands. Recently, Lee et al. [25] proposes the “retaining rate” of LLM suggestions (in their case, suggested character names subsequently used in novels) as a metric of the usefulness of LLMs for ideation. An analogue to our case would be measuring the appearance of new patterns data slices when users use ScatterShot, compared to when they come up with their own patterns.

### 6 RELATED WORK

#### 6.1 LLMs and In-context Learning

Transformer-based large language models (LLMs) [58] have recently led to large improvements in NLP. Pre-trained on a large amount of unlabeled text data, these models encapsulate rich, general-purpose features of language both syntactically and semantically. These features can help facilitate various downstream applications much more dynamically [31] — rather than having to train a new model for every custom task, users can just customize the model by feeding it natural language **prompts** at run time, like the holiday in the previous section. Such ability to recognize the desired task on-the-fly is called *in-context learning* [7].

The flexible in-context learning intrigues various work to explore designing prompts that can effectively invoke the user desired functionalities [35, 37, 46, 70]. To date, the most common patterns for prompting are either zero-shot or few-shot prompts. Zero-shot prompts directly describe what ought to happen in a task. For example, we can enact the holiday date translator in Section 1 with a **task description** prompt: “Identify the date for a national holiday in the month/date format.” Studies on improving zero-shot prompts typically study the effect of task instructions [15], induce LLM reasoning through task decomposition [63, 67], etc. Zero-shot prompts do not use demonstrative examples and therefore tend to be less performative [7], but writing just the natural language descriptions is lightweight enough that it creates an intuitive natural language interface between humans and the model [64].

In contrast, **few-shot** prompts show the LLM what pattern to follow by feeding it examples of the desired input and output data. As can be seen in Section 1, given examples on “Christmas” and “Halloween”, the LLM would produce a reasonable date for “Independence Day”. These examples usually follow consistent structures with meaningful prefixes (“Holiday: [name] => Date: [date]”), which helps re-empahsize the desired intent [38]. The quality of few-shot prompts heavily relies on the five to thirty in-context examples that demonstrate the intended behavior [32, 46], and LLMs can only perform in-context learning if it has seen the corresponding distribution or concept [35, 46, 70]. If developers omit corner cases in the few examples they created, the task quality can easily be affected [29]. For example, without a negative example where we denote ineligible inputs with a placeholder output “N/A” (“Holiday: yesterday => Date: N/A”), the LLM would attempt to produce the most plausible “label” even for negative examples — It may try to normalize “yesterday” to a most plausible date even though there is no holiday. Our work here tries to help users interactively identify high-quality in-context examples for text transformation. We review the literature on in-context example selection next.

#### 6.2 Effective Example Selection

Prior work has explored selecting effective demonstrations, and has shown that because pre-trained models possess high-level semantic features, sampling or active learning tends to help identify informative examples [56]. In particular, dynamically selecting (retrieving) the most similar demonstrative examples for each given input significantly improves in-context learning performance [10, 46]. However, such retrieval methods require fully labeled datasets as the search space. In contrast, our work studies the scenario where humans craft their personalized in-context functions, and therefore focuses on an unlabeled space. In the unlabeled search space, prior work has explored effective dataset annotation that can support better in-context learning or
few-shot finetuning. These studies strive to allocate annotation budgets to diverse and representative examples through clustering [10] or graph-based search [53]. For example, Su et al. [53] built a similarity graph by computing pairwise distances between input sentences and then iteratively selected and annotated examples based on graph density. They show such selection substantially reduces the annotation cost while achieving high and stable in-context learning performance. Despite being effective, these methods sample examples purely for input diversity. Because our work focuses more on supporting users’ interactive function construction, we additionally emphasize current function quality in sampling, which helps users track their progress and prioritize improving the current in-context function. Moreover, these prior studies measures diversity with cosine similarities on input sentence embedding [43] which, as we argue in Section 2.2, is not reflective of various tasks [46]. As a workaround, our work focuses on measuring similarities only on the key phrase embeddings, which leads to more intuitive clusters.

On the interactive example selection side, our work is perhaps more similar to some literature in programming-by-demonstration (PBD). For example, Zhang et al. [72] explored effectively selecting examples that can help disambiguate and validate synthesized regular expressions. We share similar motivations that interactively and iteratively suggest corner cases help synthesize the right function, but unlike PBD where new examples are always pruning the function search space, SCATTERSHOT focuses on expanding the function coverage. Therefore, it is essential to select examples that incentivize people to provide feedback.

**Active Learning.** Our work is also similar to the aforementioned, effective annotation work [10, 53] in the sense that its selection method is akin to sampling approaches in active learning [49, 57]. The key idea behind active learning is that machine learning models can achieve higher performance with fewer training examples, if it is allowed to choose its own, most informative training examples. Given a budget, an active learner iteratively selects examples-to-annotate from an unlabeled pool according to some ranking mechanism. While the previous work is more similar to diversity sampling [48], ours is closer to uncertainty sampling [26], where an active learner queries the instances about which it is least certain how to label. Because LLMs are generative (PBD). For example, Zhang et al. [72] explored effectively selecting

6.3 Model-assisted Annotation

SCATTERSHOT can also be seen as offering assistance in data annotation (for context learning). The idea of annotating data with both humans and AI models in the loop has been explored broadly. In this setup, AIs can play various roles [71], e.g., they may generate more examples that mimic difficult patterns [29, 45], select uncertain examples for people to inspect [61], etc. SCATTERSHOT is closer to work encouraging annotators to find model-fooling examples (“adversarial data collection.”) [6, 13, 14, 24]. In particular, Bartolo et al. [5] found that in question-answering tasks, models trained on these adversarially collected data can generalize better to more challenging examples. However, because of the overhead of re-training, their analyses were performed post-hoc, i.e., they only updated the model offline after collecting a large batch of challenging examples. In contrast, we leverage the advantage of in-context learning, and directly study the dynamic of in-context function update.

The iterative nature also links SCATTERSHOT to earlier work in interactive machine learning (IML) [3, 68]. IML is a typical paradigm that facilitates iterative and exploratory model understanding and update — a system explains to users how the current model makes predictions, and users in turn give feedback back to the model, starting the cycle again. Labeling is one classic type of IML feedback [19, 51]. However, because traditional ML tends to focus much more on the surface features (e.g., count trigrams in a training example without caring its semantic meanings), users find labeling to be not powerful enough, and prefer richer controls like feature selection [3, 39, 52]. Since LLMs have some capability to generalize individual examples more broadly to its semantically similar ones, we believe labeling in in-context learning would be more effective, and we use SCATTERSHOT to reactivate labeling-based IML for in-context learning.

7 CONCLUSION

In this work, we present SCATTERSHOT, an interactive system for building high-quality demonstration sets for in-context learning. SCATTERSHOT helps users find informative input examples in the unlabeled data, annotate them efficiently with the help of the current version of the learned in-context function, and estimate the quality of said function. Results from both a simulation study and a 10-person evaluation show SCATTERSHOT improves in-context function performance, as well as annotator’s awareness and handling of diverse patterns. Our findings highlight the importance of data slicing and sampling, iterative human-model interaction, and collaborative human-model labeling, and point to interesting future directions such as AI-assisted task definition refinement, more concrete quality metrics that convey the in-context function progress, etc.

ACKNOWLEDGMENTS

This material is based upon work supported by NSF awards 1901386 and 2040196, ONR grant N00014-21-1-2707, and a gift from the Allen Institute for Artificial Intelligence (A2). The authors thank the user study participants for their valuable feedback, and anonymous reviewers for helpful discussions and comments.
