Fixing Model Bugs with Natural Language Patches

Shikhar Murty†⋆ Christopher D. Manning† Scott Lundberg‡ Marco Tulio Ribeiro†
†Computer Science Department, Stanford University ‡Microsoft Research
{smurty,manning}@cs.stanford.edu, {scott.lundberg, marcotcr}@microsoft.com

Abstract

Current approaches for fixing systematic problems in NLP models (e.g., regex patches, fine-tuning on more data) are either brittle, or labor-intensive and liable to shortcuts. In contrast, humans often provide corrections to each other through natural language. Taking inspiration from this, we explore natural language patches—declarative statements that allow developers to provide corrective feedback at the right level of abstraction, either overriding the model (“if a review gives 2 stars, the sentiment is negative”) or providing additional information the model may lack (“if something is described as the bomb, then it is good”). We model the task of determining if a patch applies separately from the task of integrating patch information, and show that with a small amount of synthetic data, we can teach models to effectively use real patches on real data—1 to 7 patches improve accuracy by ~1–4 accuracy points on different slices of a sentiment analysis dataset, and F1 by 7 points on a relation extraction dataset. Finally, we show that fine-tuning on as many as 100 labeled examples may be needed to match the performance of a small set of language patches.

1 Introduction

Natural language enables humans to communicate a lot at once with shared abstractions. For example, in teaching someone about the colloquial use of the term “bomb”, we might say describing food as ‘bomb’ means it is very good, while saying someone bombed means it was disappointing. This simple sentence uses various abstractions (e.g., “food”) to provide context-dependent information, making it easy for humans to generalize and understand sentences such as “The tacos were bomb” or “The chef bombed” without ever having seen such examples.

There is a growing body of research focused on using language to give instructions, supervision and even inductive biases to models instead of relying exclusively on labeled examples, e.g., building neural representations from language descriptions (Andreas et al., 2018; Murty et al., 2020; Mu et al., 2020), or language / prompt-based zero-shot learning (Brown et al., 2020; Hanjie et al., 2022; Chen et al., 2021). However, language is yet to be successfully applied for corrective purposes, where the user interacts with an existing model to improve it. As shown in Fig. 1a, if a developer discovers that a model contains bugs (i.e., systematic errors; Ribeiro et al., 2020), common fixes are either brittle regex-based patches (e.g., Fig. 1a left, where patches either override predictions or replace the word “bomb” with the word “good”), or collecting hundreds of additional datapoints for fine-tuning, a tedious and computationally demanding process that can still lead to shortcuts such as assuming the word “bomb” is always positive (e.g., if the additional finetuning data mostly has the word in its colloquial sense). Instead, we envision a setting where developers provide corrective feedback through a Natural Language Patch—a concise statement such as “If food is described as bomb, then food is good”. Language makes it easy for developers to express feedback at the right level of abstraction without having to specify exactly how the condition is applied. The patching system is responsible for applying the patch and interpreting the information appropriately, e.g., applying it to “The tacos were the bomb” but not to “The authorities found a bomb in the restaurant”.

In this work, we present an approach for patching neural models with natural language. Any patching system has to determine when a patch is relevant, and how it should modify model behavior. We model these tasks separately (Fig. 1b): a gating head soft-predicts whether the patch should be applied (e.g., “food is described as bomb”), and
2 stars, but our waitress Wendy was really nice.

Two stars for the place, but the ambiance is great.

The authorities found a bomb in the restaurant.

Regexp Patch

def patch_2(x):
    if ' bomb ' in x:
        x = x.replace('bomb', 'good')
    return model(x)

def patch_1(x):
    if '2 star' in x:
        return negative
    return model(x)

Language Patch

If food is described as
bomb, then food is good
If review gives 2 stars, then label is negative
If review gives 4 or 5 stars, then label is negative

Recall that conditions, where regex patches would be infeasible or very difficult—that is, they are applied correctly when the patch condition is met, and do nothing otherwise. Perhaps surprisingly, this is true even for test-time patches that are very different than the ones used in the patch finetuning stage. Next, in Section 6, we show that despite the synthetic nature of the patch tuning phase, a small set of very simple patches can fix bugs (and thus improve performance) on real benchmarks for two different tasks—1 to 6 simple language patches improve performance by ~1−4 accuracy points on two slices from the Yelp reviews dataset, while 7 patches improve performance by ~7 F1 points on a relation extraction task derived from NYT. Finally, in Section 7.2, we compare language patching, a computationally lightweight procedure, with finetuning, a computationally and human-labor intensive procedure, and find that as many as 100 labeled examples are needed to match performance gains from a small set of 1 to 7 patches. Further, finetuning sometimes fixes bugs at the expense of introducing new bugs, while patches maintain prior performance on inputs where they do not apply.

2 Natural Language Patching

**Setup.** We are given a model \( f \), mapping an input text \( x \) to a probability distribution over its output space, \( f(x) = \Pr(y \mid x) \). The model contains bugs—defined as behaviors inconsistent with users’ preferences or the “ground truth”—which we want to fix with a library of patches \( P = \{p_1, p_2, \ldots, p_t\} \). Users explicitly indicate the condition under which each patch applies and the consequence of applying it, such that each patch is in the form “If (condition) \( c \), then (consequence) \( q \)”. We use this format to making modeling easier, noting that it still allows for very flexible patching through high level abstractions (e.g., “if the customer complains about the ambiance”, “if food is not mentioned”, etc), and that most patches have an implicit applicability function, and thus can be converted to this format.

**Applying Patches.** As indicated in Fig. 1b, our model consists of two separate heads. The gating head \( g \) computes the probability that the condition specified by \( lp = (c, q) \) is true for a given input \( x \) as \( g(x, c) \). The interpreter head \( I \) computes a new distribution over the label space, that conditions on \( x \) and the consequence \( q \). This is then combined with the original model output \( f(x) \) using the above gating probability. A single patch \( lp = (c, q) \), can...
be applied to any input \( x \) as

\[
\text{Fix}(f, x, lp) = g(x, c) \cdot I(x, q) + [1 - g(x, c)] \cdot f(x). \tag{1}
\]

Given a library of patches \( P = \{lp_1, \ldots, lp_k\} \), we find the most relevant patch \( lp^* \) for the given input, and use that to update the model,

\[
lp^* = \text{arg max}_{lp \in P} g(x, c_i), \tag{2}
\]

\[
\text{Fix}(f, x, P) = \text{Fix}(f, x, lp^*). \tag{3}
\]

**Patch Types.** We consider two categories of patches (examples in Table 1). **Override** patches are of the form “If cond, then label is l” i.e., they override the model’s prediction on an input if the patch condition is true. For these patches, we do not use the interpreter head since \( I(x, \text{“label is l”}) = l \). **Feature-based** patches are of the form “If cond, then feature”, i.e., they provide the model with a contextual feature “hint” in natural language, e.g., in Fig. 3 the feature is “food is good”. For these patches, the model needs to integrate the hints with the original data, and thus both the gating and interpreter heads are used.

3 Training Patchable Models

Assuming \( f \) has a text encoder and a classification head, we have two finetuning stages. In the **Task Finetuning** stage, we train \( f \) on a labeled dataset \( \{x_i, y_i\} \) (standard supervised learning). In the **Patch Finetuning** stage, we use the learnt encoder and learn \( g \) (initialized randomly) and \( I \) (initialized with the classification head). For the patch finetuning stage, we write a small set of patch templates covering the kinds of patches users may write for their own application (see Table 1 for the patch templates used for our sentiment analysis results). Based on these templates, we instantiate a small number of patches along with synthetic labeled examples. This gives us a dataset \( \{x_i, y_i, lp_i\} \), where \( lp_i \) consists of a condition \( c_i \) as well as a consequence \( q_i \). The interpreter head \( I \) is trained to model \( \Pr(y_i \mid x_i, q_i) \) through standard log-likelihood maximization. The gating head \( g \) is trained via noise contrastive estimation to maximize

\[
\log g(x_i, c_i) - \sum_{c_j \in \text{NEG}(x_i)} \log g(x_i, c_j), \tag{4}
\]
4 Experimental Setup

Applications. We apply our method to binary sentiment analysis and relation extraction. For sentiment analysis, our task finetuning data comes from SST2 (Socher et al., 2013). For relation extraction, we use the Spouse dataset (Hancock et al., 2018) for task finetuning, where the objective is to determine whether two entities are married or not given a textual context about them.

Model. We use T5-large (Raffel et al., 2019) as implemented in the transformers library (Wolf et al., 2020) for all experiments. Both the gating and interpreter heads are separate decoders learnt on top of a shared encoder and each of these components are initialized with the corresponding T5 pre-trained weights. To prevent catastrophic forgetting on the original task during patch finetuning, we also multi-task learn the patch finetuning loss along with the original task loss. Templates for generating patches for patch finetuning are in Table 1 for sentiment analysis and in Table 9 (Section A.2) for relation extraction. We train separate models for override and feature-based patches (the former does not need an interpreter head). When using a patch, its content (either \(c\) for the gating head or \(q\) for the interpreter head) is inserted in the beginning of the input with a separator as in Fig. 1b.

Baselines. We report performance of the original model with only task finetuning (ORIG) and the model obtained after patch finetuning (ORIG+PF) without using any patches, to isolate the gains of language patches from those induced by training on additional synthetic data. We also report results obtained from prompting ORIG with our patches (PROMPT), i.e., inserting the patch text before the input text to see how well finetuned T5 follows instructions. To use multiple patches for this baseline, we prompt the model with each individual patch and ensemble results with majority voting. Finally, we experiment with regex-based patches (REGEX) where patch conditions are converted to regex rules and consequents are converted into functions Rule\(_q\)(\(x\)). For override patches, this function simply outputs the specified label. For sentiment analysis, where feature based patches supply contextual meanings, Rule\(_q\)(\(x\)) replaces words with specified meanings e.g., replacing “bomb” with “good” in “the food was bomb”. For feature based patches on relation extraction, Rule\(_q\)(\(x\)) appends the patch consequent to the input text.

Entropy Increasing Transformations. Patch finetuning will fail if the synthetic data can be fit by a model that ignores the input or the patch (Fig. 2a). Thus, to ensure our model cannot fit the synthetic data without combining patch features with inputs, we perturb the inputs with Entropy Increasing Transformations (EITs). We identify words from the input template for which the patch supplies additional information e.g., aspect adjectives, relationship between entities, and transform these into a small set of nonce words. Crucially, the meanings of these nonce words vary from example to example, and can only be inferred from the patch (Fig. 2a bottom; more examples in Appendix A.2). Intuitively, the transformations inject an additional source of randomness which can only be recovered via the patch features. Such transformations are also used in Rajendran et al. (2020) in the context of meta-learning. EITs alone do not fix the failure mode where the model can fit the data without using input features at all. For example, in Fig. 2a bottom, the model might learn a shortcut so that it always predicts 1/0 for “food is good” / “food is bad”, regardless of the input. Thus, in addition to EITs, to ensure that the model uses input features, we ensure that a given patch consequence \(q\) and the target label are independent (Fig. 2b).
Override

If food is described as {word}, then label is negative.

Feature-based

If food is described as {word}, then food is bad.

If Entity1 has a {kid} with Entity2, then Entity1 and Entity2 have kids.

If Entity1 gave Entity2 a ring, then Entity1 is engaged to Entity2.

If Entity1 and Entity2 went for a {hyword}, then Entity1 went on a honeymoon with Entity2.

a. Patches used

b. Patch applies and important

c. Patch does not apply or not important

Figure 3: (a) Example patches used for our controlled experiments. (b) Some inputs where the patch is important for making correct predictions. (c) To control for spurious behaviors such as copying label words from the patch, performing simple string lookups or affecting predictions when patch features are unimportant, we also construct invariance tests where we expect model predictions to be unaffected by the patch.

5 Controlled Experiments

We test the behavior of language patches (and baselines) under different controlled conditions with CheckList (Ribeiro et al., 2020). Patches and example inputs are presented in Fig. 3. We test cases where patches apply and are relevant for predictions, and corresponding cases where they either do not apply or are not relevant. Thus, models that rely on shortcuts such as copying the label word from the patch or merely performing token matchingperform poorly on the CheckList.

For sentiment analysis, we test Override patches with abstract conditions (e.g., “If food is described as weird, then label is negative”) on various concrete instantiations such as “The pizza at the restaurant was weird”. We also construct invariance tests (O-Inv), where adding such patches should not change predictions on inputs where the condition is false (e.g., “The waiter was weird”, “The tacos were not weird”). We also construct tests for feature-based patches (Feat) where patches provide meaning for nonce adjectives, with analogous invariance tests (Feat-Inv). Finally, we construct analogous tests for relation extraction, where patches fill in reasoning gaps in the model such as “If Entity1 gave Entity2 a ring, then Entity1 and

Table 2: Applying patches on CheckLists. We see significant improvements when the patches apply and invariances when they do not apply or are unimportant. For Sentiment Analysis, the datasets are designed to evaluate patching with abstract conditions, thus we see no effects from using regex based patches. For testing invariance, we report the percentage of inputs for which the prediction did not change w.r.t. the base model.

Entity2 are engaged”.

We present the results in Table 2, where we first note that ORIG+PF does not perform well overall, and thus patching improvements are not merely a result of the additional synthetic data. REGEX cannot handle abstract conditions, and thus (as expected) does not change predictions on sentiment analysis, and does not do well on relation extraction. While merely inserting the patch into the input (PROMPT) results in some gains when the patch applies, it does so at the cost of changing predictions when the patch does not apply (O-Inv andFeat-Inv). In contrast to baselines, our method is able to apply abstract patches correctly on concrete instantiations, disregarding them when they do not apply, without relying on shortcuts such as copying the label from the consequent or merely checking for matching words between patch and input (all of which are tested by the invariance tests).

6 Patching models on real benchmarks

6.1 Sentiment Analysis

Unless noted otherwise, all datasets in this subsection are derived from Yelp Review (Zhang et al., 2015). To fix errors on low-accuracy slices, we write patches by inspecting a random subset of 10-20 errors made by ORIG+PF.

Controlling the model. In order to check if patches can control model behavior with abstract conditions “in the wild”, we manually annotate a random subset of 500 reviews with food and service specific sentiment (“The food was good, service not so much”) is labeled as service: 0, food: 1. We then construct override patches of the form “if food / service is good / bad, then label is positive
We construct Spouse-FewRel, an out-of-distribution test benchmark derived from FewRel (Gao et al., 2019) by sampling from all
relation types where at least one of the entities is a person \((n = 8400)\), and labeling examples as positive if they have the Spouse relation, negative otherwise. We inspect 20 randomly sampled errors made by \textsc{Orig+PF} on Spouse-FewRel, and observe that the model often confuses “Entity1 has a child with Entity2” with “Entity1 is the child of Entity2”, and also misclassifies widowhood as negative. Thus, we write override patches for both of these error categories, resulting in 7 patches, presented in Table 6. Using all patches, we observe a ~7.4 point F1 improvement over \textsc{Orig}, while baselines either decrease F1 or barely improve it.

We highlight in Table 6 a phenomenon where each natural language patch in isolation decreases performance, while all patches together increase performance. Further analysis reveals that this is because the gating head is not well calibrated in this case, and thus individual patches are applied incorrectly. However, the comparative values of \(g(x, c_i)\) are often ordered correctly, and thus a better patch is the one applied \((lp^*\text{ in Eq 2})\) when all patches are available. We do further analysis in Table 7, where we report the gating accuracy (i.e., whether the patch actually applies or not, labeled manually) of \(lp^*\) on the subset of inputs where the PATCHED model changes the prediction (Diff), and where it changes the prediction to the correct label (Diff \(\cap\) Correct). With the caveat that patches are applied softly (and thus perfect gating accuracy is not strictly necessary), we observe that a few patches seem to hurt performance even in combination with others (e.g., the first one). We also note that the patched model is right “for the right reasons” in over 72% of inputs where it changes the prediction to the correct one.

![Table 6: Using Override Patches on Spouse-FewRel for Spouse relation extraction.](image)

| Patch Condition | Diff | Diff \(\cap\) Correct |
|-----------------|------|------------------------|
| \(p_2\) is the son of \(p_1\) | 0.0 | NaN (0/0) |
| \(p_1\) is the son of \(p_2\) | 75.0 | 75.0 |
| \(p_1\) and \(p_2\) have a daughter | 63.3 | 93.8 |
| \(p_1\) and \(p_2\) have a son | 78.1 | 98.3 |
| \(p_1\) is the widow of \(p_2\) | 10.9 | 19.6 |
| \(p_1\) is the daughter of \(p_2\) | 71.4 | 100.0 |
| \(p_2\) is the daughter of \(p_1\) | 6.3 | 100.0 |
| Overall | 42.9 | 72.3 |

Table 7: We measure how often the chosen patch correctly applies to an input (i.e., gating accuracy) for Spouse-FewRel, for the set of inputs where the patched model and original model differ (Diff) as well as the subset where the patched model is correct (Diff \(\cap\) Correct).

![Table 8: Patching Accuracy of a model with and without Entropy Increasing Transformations (EITs).](image)

| Patch Consequent | Patched | Patched (Without EITs) |
|------------------|---------|------------------------|
| \(p_1\) went on a honeymoon with \(p_2\) | 59.1 | 33.7 |
| \(p_1\) has kids with \(p_2\) | 75.2 | 74.4 |
| \(p_1\) is engaged to \(p_2\) | 77.7 | 64.8 |
| food is good | 67.8 | 54.2 |
| food is bad | 88.7 | 56.5 |
| service is good | 62.8 | 52.9 |
| service is bad | 62.8 | 52.9 |
| Overall | 70.6 | 55.6 |

Table 8: Patching Accuracy of a model with and without Entropy Increasing Transformations (EITs).

### 7 Analysis

#### 7.1 How Important are EITs?

The goal of Entropy Increasing Transformations (EITs; Section 3) is to prevent the interpreter head from learning shortcuts that either ignore patch features or rely exclusively on them. We perform an ablation, comparing our model to a model trained without EITs on the CheckLists in Table 2 (Section 5), where the feature-based patch consequent supplies important information for making a correct prediction. From Table 8, we note that the interpreter head trained without EITs has much lower performance on these datasets (as expected).

#### 7.2 Comparison to fine-tuning

While patching is computationally lightweight, it requires domain knowledge or error analysis of incorrectly labeled examples. However, once such analysis is performed, one can label these additional examples and finetune the model on them. Ignoring the computational and infrastructure costs of repeated finetuning, for patching to be a competitive alternative to finetuning from an \textit{annotation budget} perspective, we require the gains from...
patching to only be matched by multiple labeled examples. To compare language patches with finetuning, we consider Yelp-stars, Yelp-Colloquial, and Spouse-FewRel and split each dataset into a training set with 128 examples, and a test set with remaining examples. Next, we finetune ORIG on \( k = \{2, 4, 8, 16, 32, 64, 128\} \) examples from the training set, stopping early if finetuning performance exceeds patched performance. We finetune for 64 steps and optimize using AdamW with a fixed learning rate of 1e-4. We report means and standard deviations obtained from finetuning with 5 random seeds.

Results are presented in Fig. 4, where we note that over 100 labeled examples are needed to match the performance of a single patch on Yelp-Stars or 7 patches on Spouse-FewRel. On Yelp-Colloquial, the patched performance is matched with a mere 16 examples. However, as noted earlier, Yelp-Colloquial is susceptible to simple shortcuts, and we observe that the performance on the control set Yelp-Colloquial-Control suffers significantly as we finetune on more data (with very high variance). Thus, we conclude that language patches on these datasets are not only very efficient in terms of annotation effort (when compared to labeling data for finetuning), but also less susceptible to simple shortcuts that do not address the problem at the right level of abstraction.

8 Related Work

Learning with Language. Natural language instructions or explanations have been used for training fewshot image classifiers (Mu et al., 2020; Andreas et al., 2018), text classifiers (Zaidan and Eisner, 2008; Srivastava et al., 2018; Camburu et al., 2018; Hancock et al., 2018; Murty et al., 2020), and in the context of RL (Branavan et al., 2012; Goyal et al., 2019; Co-Reyes et al., 2019; Mu et al., 2022). All of these works are concerned with reducing labeled data requirements with language supervision, while our setting involves using language as a corrective tool to fix bugs at test time.

Prompt Engineering. An emerging technique for re-purposing language models for arbitrary downstream tasks involves engineering “prompts”. Prompts are high level natural language descriptions of tasks that allow developers to express any task as language modeling (Brown et al., 2020; Gao et al., 2021; Zhong et al., 2021). While we could try and directly use prompting to incorporate language patches, our experiments show that the models we consider fail to correctly utilize patches in the prompt (Section 4). With increasing scale models may gain the ability to interpret patches zero-shot, but qualitative exploration of the largest available models at the time of writing (e.g. GPT-3; Brown et al., 2020) indicates they still suffer from the same problem. Using patches for corrective purposes requires an accurate interpretation model, as well as ignoring the patch when it is not applicable. We solve these challenges by learning a gating head and an interpretation head through carefully constructed synthetic data.

Editing Factual Knowledge. Test time editing of factual knowledge in models is considered by Talmor et al. (2020); Cao et al. (2021); Mitchell et al. (2021); Meng et al. (2022). Instead of modifying factual knowledge, we show that free-form language patches can be used to fix bugs on real data, such as correctly interpreting the meaning of the word “bomb” in the context of food or predicting that divorced people are no longer married.

9 Conclusion

When faced with the task of fixing bugs in trained models, developers often resort to brittle regex rules or finetuning, which requires curation and labeling of data, is computationally intensive, and susceptible to shortcuts. This work proposes natural language patches which are declarative statements of the form “if \( c \), then \( q \)” that enable developers to control the model or supply additional
information with conditions at the right level of abstraction. We proposed an approach to patching that models the task of determining if a patch applies (gating) separately from the task of integrating the information (interpreting), and showed that this approach results in significant improvements on two tasks, even with very few patches. Moreover, we show that patches are efficient (1-7 patches are equivalent or better than as many as 100 finetuning examples), and more robust to potential shortcuts. Our system is a first step in letting users correct models through a single step “dialogue”. Avenues for future work include extending our approach to a back-and-forth dialogue between developers and models, modeling pragmatics, interpreting several patches at once, and automating patch finetuning.

10 Acknowledgements
SM was partly funded by a gift from Apple Inc. We are grateful to Jesse Mu, Mirac Suzgun, Pratyusha Sharma, Eric Mitchell, Ashwin Paranjape, Tongshuang Wu, Yilun Zhou and the anonymous reviewers for helpful comments. The authors would also like to thank members of the Stanford NLP group and the Adaptive Systems and Interaction group at MSR for feedback on early versions of this work.

11 Reproducibility
Code and model checkpoints are available at https://github.com/MurtyShikhar/LanguagePatching.

12 Limitations
Scaling to large patch libraries. For our approach, inference time scales linearly with the size of the patch library. This is primarily because the gating head makes predictions on each patch in our patch library (Eq 2). Instead of running the gating head on each patch, one can trade off exactness for efficiency, by running the gating head on a much smaller candidate set identified using fast approximate nearest neighbors (Johnson et al., 2019) on sentence embeddings.

Scaling to more patch types. The current approach requires writing patch templates beforehand based on prior knowledge of the kinds of corrective feedback that developers might want to write in the future. Writing patch templates manually is fundamentally bottlenecked by human creativity and foresight. Moreover, since humans are required to write templates, it makes scaling up to different patch types harder, since we expect generalization to completely new patch types to be poor e.g., generalizing to a patch that requires counting. Future work can explore automatic generation of synthetic patch templates e.g., using pre-trained language models.

Interpreting multiple patches. Finally, the approach we develop can only incorporate a single patch at a time, by selecting the most relevant patch from our patch library. This precludes the model from being able to combine features from multiple patches—e.g., “caviar is a kind of food” and “If caviar is described as overpowering, then caviar is spoiled”.

References
Jacob Andreas, Dan Klein, and Sergey Levine. 2018. Learning with latent language. In NAACL HLT 2018 - 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference, volume 1.
S. R.K. Branavan, David Silver, and Regina Barzilay. 2012. Learning to win by reading manuals in a monte-carlo framework. Journal of Artificial Intelligence Research, 43.
Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901.
Oana Maria Camburu, Tim Rocktäschel, Thomas Lukasiewicz, and Phil Blunsom. 2018. E-snli: Natural language inference with natural language explanations. volume 2018-December.
Nicola De Cao, Wilker Aziz, and Ivan Titov. 2021. Editing factual knowledge in language models.
Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hehgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles
Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. CoRR, abs/2107.03374.

John D. Co-Reyes, Abhishek Gupta, Suvansh Sanjeev, Nick Altieri, Jacob Andreas, John DeNero, Pieter Abbeel, and Sergey Levine. 2019. Guiding policies with language via meta-learning.

Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making pre-trained language models better few-shot learners. In Association for Computational Linguistics (ACL).

Tianyu Gao, Xu Han, Hao Zhu, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. 2019. FewRel 2.0: Towards more challenging few-shot relation classification. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 6251–6256, Hong Kong, China. Association for Computational Linguistics.

Prasoon Goyal, Scott Niekum, and Raymond J. Mooney. 2019. Using natural language for reward shaping in reinforcement learning. CoRR, abs/1903.02020.

Braden Hancock, Martin Bringmann, Paroma Varma, Percy Liang, Stephanie Wang, and Christopher Ré. 2018. Training classifiers with natural language explanations. volume 1.

Austin W Hanjie, Ameet Deshpande, and Karthik Narasimhan. 2022. Semantic supervision: Enabling generalization over output spaces. arXiv preprint arXiv:2202.13100.

Jeff Johnson, Matthijs Douze, and Hervé Jéjou. 2019. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data, 7(3):535–547.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. Locating and editing factual knowledge in gpt. ArXiv, abs/2202.05262.

Eric Mitchell, Charles Lin, Antoine Bosselut, Chelsea Finn, and Christopher D Manning. 2021. Fast model editing at scale. arXiv preprint arXiv:2110.11309.

Jesse Mu, Percy Liang, and Noah Goodman. 2020. Shaping visual representations with language for few-shot classification.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. 2022. Improving intrinsic exploration with language abstractions. arXiv preprint arXiv:2202.08948.

Shikhar Murty, Pang Wei Koh, and Percy Liang. 2020. Expert: Representation engineering with natural language explanations.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. CoRR, abs/1910.10683.

Janarthanan Rajendran, Alexander Itpan, and Eric Jang. 2020. Meta-learning requires meta-augmentation. In Advances in Neural Information Processing Systems, volume 33, pages 5705–5715. Curran Associates, Inc.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with CheckList. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 4902–4912, Online. Association for Computational Linguistics.

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

Shashank Srivastava, Igor Labutov, and Tom Mitchell. 2018. Zero-shot learning of classifiers from natural language quantification. volume 1.

Alon Talmor, Oyvind Tafjord, Peter Clark, Yoav Goldberg, and Jonathan Berant. 2020. Leap-of-thought: Teaching pre-trained models to systematically reason over implicit knowledge. volume 2020-December.

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

Omar F. Zaidan and Jason Eisner. 2008. Modeling annotators: A generative approach to learning from annotator rationales.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level Convolutional Networks for Text Classification. arXiv:1509.01626 [cs].

Ruiqi Zhong, Kristy Lee, Zheng Zhang, and Dan Klein. 2021. Meta-tuning language models to answer prompts better. CoRR, abs/2104.04670.
Override Patches

If [Entity1/Entity2] is not a person, then label is negative
If Entity1 is the [child/parent] of Entity2, then label is positive
If Entity1 and Entity2 have children, then label is positive
If Entity1 and Entity2 are divorced, then label is negative
If Entity1 is engaged to Entity2, then label is positive
If Entity1 and Entity2 are siblings, then label is negative

Feature Based Patches

If cond, then Entity1 is [married/not married] to Entity2
If cond, then Entity1 is divorced from Entity2
If cond, then Entity1 is the sibling of Entity2
If cond, then Entity1 is dating Entity2
If cond, then Entity1 is the parent of Entity2

Table 9: Patch templates used for the Patch Finetuning stage for relation extraction. Each Entity is sampled from a small list of names, and cond is a set of conditions derived from keywords.

A More details on Patch Finetuning

A.1 Sentiment Analysis Data

The templates used for constructing inputs are in Table 12. We programmatically find all patches for an input, to generate labels.

A.2 Relation Extraction Data

Override Patches. Patches and Input templates for constructing patch finetuning data can be found in Table 13.

Feature Based Patches For training the gating head, we use the same data as generated by Table 13. For training the interpreter head, we use patches and input templates in Table 11 to generate finetuning data.

A.3 Additional Finetuning Details

After the model is finetuned in the Task finetuning stage, we finetune it additionally with a learning rate of learning rate of 1e-4 and with a linear warmup scheduler which ramps up the learning rate from 0 to 1e-4 over 100 steps. The training batch size is 32, and we clip gradients to have a max norm of 5. We early stop based on validation performance on a held out subset of the patch finetuning data.

B Patches used for Yelp-Colloquial.

We used the following patches for fixing bugs on Yelp-Colloquial:

• “If clothes are described as dope, then clothes are good.”

Table 10: Dataset statistics for all the real data slices considered in this work.

| Dataset               | #examples |
|-----------------------|-----------|
| Yelp-Stars            | 3172      |
| Yelp-Colloquial       | 1784      |
| WCR                   | 2919      |
| Yelp-Colloquial (Control) | 67    |
| Yelp-Aspect           | 439       |
| Spouse-NYT            | 8400      |

C More examples of Entropy Increasing Transformations

To perform Entropy Increasing Transformations (EITs) for relation extraction, we convert rel (see Table 11) into nonce words e.g., “Alice has a kid with John” gets transformed into “Alice has a wug with John”, for which we use a patch “If Entity1 has a wug with Entity2, then Entity1 and Entity2 have kids

D Regex Based Patches.

The exact functions we use for patching with regexes can be found in Listing 1 and Listing 2.

E Data Statistics for all evaluation slices

Statistics for all slices used for evaluation can be found in Table 10.
Table 11: Templates used for constructing inputs for patch finetuning stage in relation extraction analysis. Terms marked with ‘()’ are optional. \texttt{rel} is a list of 4 relation types. For each relation type, we have a small list of 3 to 4 words. For instance \texttt{have-kids} = ['has a kid with', 'has a son with', 'has a daughter with']

Table 12: Templates used for constructing inputs for patch finetuning stage in sentiment analysis. Terms marked with ‘()’ are optional. \texttt{adj} comes from a small set of 6 positive and 6 negative adjectives, as well as 6 nonce adjectives for EITs

Table 13: Patches along with a subset of inputs used for the Patch Finetuning stage for the Spouse relation extraction task. For each input, we highlight the two entities and provide examples of some positive and negative patches.
def star_rbpatch(model, inp):
    keywords = ['0 star', '1 star', '2 star',
                'zero star', 'one star', 'two star']
    patches = [(keyword, 0) for keyword in keywords]
    return sentiment_override_rbpatch(model, inp, patches)

def clothing_reviews_rbpatch(model, inp):
    return sentiment_override_rbpatch(model, inp, [('boxy', 0), ('needs to be returned', 0)])

def spousenyt(model, inp):
    patch_list = [('Entity1 is the son of Entity2', 0),
                  ('Entity2 is the son of Entity1', 0),
                  ('Entity1 and Entity2 have a son', 1),
                  ('Entity1 and Entity2 have a daughter', 1),
                  ('Entity1 is the daughter of Entity2', 0),
                  ('Entity2 is the daughter of Entity1', 0),
                  ('Entity1 is the widow of Entity2', 1)]
    return re_override_rbpatch(model, inp, patch_list)

# for all override patches
def sentiment_override_rbpatch(model, inp, patch_list):
    '''
    inp: "X" a review for which we want to predict sentiment
    patch_list: list of override patches converted into a form (cond, label) where cond is a string condition and label is the associated binary label
    '''
    for cond, label in patch_list:
        if cond in inp:
            return label
        return model(inp)

# for override patches for relation extraction
def re_override_rbpatch(model, inp, patch_list):
    '''
    inp: "X. Entity1: e1. Entity2: e2"
    patch_list: list of override patches converted into a form (cond, label) where cond is a string condition and label is the associated binary label
    '''
    text, ent_info = inp.split(' Entity1:')
    e1, e2 = ent_info.split(' Entity2: ')
    e1 = e1.strip()
    e2 = e2.strip()
    for cond, label in patch_list:
        p = patch.replace('Entity1', e1).replace('Entity2', e2)
        p2 = patch.replace('Entity1', '').replace('Entity2', '')
        if p in inp:
            return pred
        elif p2 in inp:
            return pred
    return model(x)

Listing 1: Rule based override patching for all our experiments
# Regex based patching for using feature based patches on
# controlled experiments for sentiment analysis

def sentiment_regex_based(model, inp, patch_list):
    '''
    inp: "X. Entity1: e1. Entity2: e2"
    patch_list: list of feature patches of the form 'if aspect is word, then aspect is good/ bad'
    as a tuple (aspect, word, sentiment)
    '''

    for aspect, word, sentiment in patch_list:
        if '{} is {}'.format(aspect, word) in inp:
            inp = inp.replace(word, sentiment)
            break
    return model(inp)

# Regex based patching for controlled experiments on relation extraction

def re_regex_based(model, inp, patch_list):
    '''
    inp: "X. Entity1: e1. Entity2: e2"
    patch_list: list of feature patches converted into a form (cond, cons) where cond
    and cons are both strings
    '''

    text, ent_info = inp.split('Entity1:')
    e1, e2 = ent_info.split('Entity2:')
    e1 = e1.strip()
    e2 = e2.strip()
    for cond, cons in patch_list:
        p = cond.replace('Entity1', e1).replace('Entity2', e2)
        if p in inp:
            cons_curr = cons.replace('Entity1', e1).replace('Entity2', e2)
            inp = '{}.{} Entity1: {}. Entity2: {}'.format(cons_curr, text, e1, e2)
        break
    return model(inp)

# Regex based patching on yelp colloquial

def yelp_col_regex_based(model, inp, patch_list):
    if 'wtf' in inp:
        inp = inp.replace('wtf', 'bad')
    elif 'omg' in inp:
        inp = inp.replace('omg', 'good')
    elif 'the shit' in inp:
        inp = inp.replace('the shit', 'good')
    elif 'bomb' in inp and 'food' in inp:
        inp = inp.replace('bomb', 'good')
    elif 'bomb' in inp and 'service' in inp:
        inp = inp.replace('bomb', 'good')
    elif 'bomb' in inp and 'restaurant' in inp:
        inp = inp.replace('bomb', 'good')
    elif 'dope' in inp and 'clothes' in inp:
        inp = inp.replace('dope', 'good')
    elif 'sucks' in inp:
        inp = inp.replace('sucks', 'bad')
    return model(inp)

Listing 2: Regex based patching for using feature based patches for all experiments.