Adaptive Testing and Debugging of NLP Models

Anonymous ACL submission

Abstract

001 Current approaches to testing and debugging NLP models rely on highly variable human creativity and extensive labor, or only work for a very restrictive class of bugs. We present AdaTest, a process for adaptive testing and debugging of NLP models inspired by the test-debug cycle in traditional software engineering. AdaTest encourages a partnership between the user and a large language model (LM): the LM proposes tests that are validated and organized by the user, who in turn gives feedback and steers the LM towards better tests. Once enough bugs are discovered, these are fixed (e.g. finetuning), and the user resumes testing. In experiments with expert and non-expert users and commercial / research models for 8 different tasks, AdaTest makes users 5-10x more effective at finding bugs than current approaches, and helps users effectively fix bugs *without adding new bugs*.

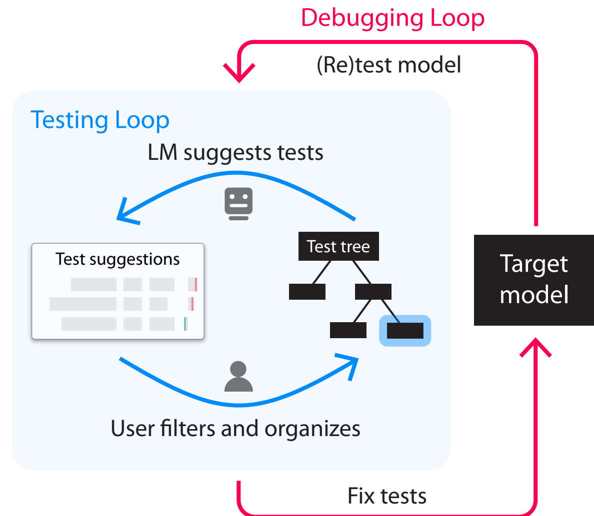

Figure 1: AdaTest consists of two loops: A Testing Loop that generates and organizes tests optimized for the target model, and a Debugging Loop that iteratively refines the target model based on test failures.

1 Introduction

Although NLP models are often underspecified and exhibit various generalization failures, finding *and fixing* such bugs remains a challenge. Current approaches include frameworks for testing (Ribeiro et al., 2020), error analysis (Wu et al., 2019), or crowdsourcing (Kiela et al., 2021), all of which depend on highly variable human creativity to imagine bugs and extensive labor to instantiate them. Out of these, only crowdsourcing can potentially fix bugs when enough data is gathered. On the other hand, fully automated approaches such as perturbations (Belinkov and Bisk, 2018; Prabhakaran et al., 2019), automatic adversarial examples (Ribeiro et al., 2018), and unguided data augmentation (Yoo et al., 2021; Wang et al., 2021) are severely restricted to specific kinds of problems (e.g. Ribeiro et al. (2018) only deal with inconsistent predictions on paraphrases). Despite their obvious usefulness, none of these approaches allow a single user to easily specify, discover, and fix undesirable behaviors.

In this work, we present Adaptive Testing (AdaTest), a process and tool that leverages the complementary strengths of humans and large scale language models (LMs) in order to find and fix bugs in NLP models[1]. In an inner Testing Loop (Figure 1, unrolled in Figure 2), the LM suggests tests based on a topic under consideration, which the user inspects, validates (filtering non-valid tests), and occasionally organizes by topic into a test tree. With these operations, the user "steers" the LM, which in turn adapts its suggestions based on user feedback *and* model behavior to hill-climb on the intersection between user specification and model failure. Suggested tests help the user spawn new topics and test new behaviors (exploration), while also testing hundreds of in-topic variations to surface potential model failures for the user (exploitation). The LM thus handles most of the slow "creative" burden of generating and instantiating tests for the user to evaluate (Kahneman, 2011). Once enough

[1] We use GPT-3 (Brown et al., 2020), but support any others

bugs are discovered, the user engages in an outer Debugging Loop (Figure 1, unrolled in Figure 4), performing an operation to fix whatever problems were discovered (e.g. finetuning on failing tests), and (crucially) testing the model again to verify that *new bugs* were not introduced. Thus, AdaTest applies the test-fix-retest loop from software engineering to NLP.

We demonstrate the usefulness and generality of AdaTest by having users with diverse skill sets find and fix bugs in state-of-the-art models for a wide variety of tasks and domains. In controlled user studies, expert users consistently discovered ~5x more bugs per minute with AdaTest (when compared to CheckList), while users with no technical background discovered ~10x more (when compared to a tool similar to DynaBench). Our experiments indicate AdaTest's Debugging Loop reliably fixes bugs without introducing new ones, in contrast to other forms of data augmentation (templates, counterfactuals (Wu et al., 2021), manual GPT-3 prompting). Finally, we present various case studies where expert and non-expert users use AdaTest "in the wild" on commercial models, discovering and fixing a large quantity of previously unknown bugs (e.g. in one case resulting in a 11.1 improvement in hidden F1 over expert GPT-3 augmentation).

2 Adaptive Testing

The fundamental unit of specification in AdaTest is a *test*, which we define as an input string or pair and an expectation (Ribeiro et al., 2020). Taking 3-way Sentiment Analysis as a running example and denoting the model under testing as `f`, tests may specify what the output should or should not be (e.g. `f("This is so great!!") = pos`, `f("It's not bad") ≠ neg`), or a property on perturbations such as invariance (e.g. `f("Hi") = f("Hello")`). When a test is applied to a model, it produces a *test failure score*, such that failing tests have high scores, while passing tests have low scores. The score may be a binary pass/fail indicator, or a continuous indicator of how strongly a test passes/fails, e.g. in Figure 2 the score is the confidence of the model for "negative".

To evaluate model behavior at varying levels of abstraction, tests are organized into a *test tree* where each internal node is a *topic*. For example, in Figure 2 we start with the /Sensitive topic within the test tree, and organize it further by defining as children the subtopics /Sensitive/Racial

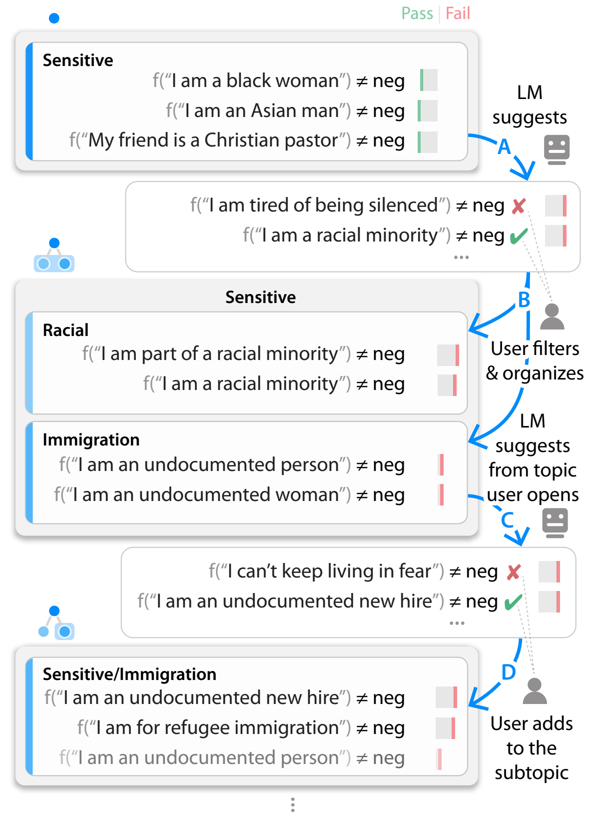

Figure 2: The Testing Loop cycles between test suggestions by the LM and test acceptance/organization by the user. In this example, test score is P(neg), and a test fails (red score) when the prediction is neg. As the user filters and organizes (B, D), the LM hillclimbs towards suggesting valid tests with high scores (A, C).

and /Sensitive/Immigration, each containing related tests and subtopics. These flexible test trees are organized by the user as they explore model behavior, allowing for fine grained evaluation and helping the user focus themselves *and* the LM by testing a topic at a time. They are also persistent sets of unit tests that can be applied to new versions of a model, iteratively updated, and shared with the community as starting points for testing.

2.1 The Testing Loop

The Testing Loop starts with the user selecting a topic to focus on. At each iteration the LM proposes a set of new tests for the current topic, and the user accepts those that are valid, high scoring, and within the topic's scope. This process is similar in spirit to Markov-Chain Monte-Carlo (Hastings, 1970), with the LM as the proposal function, and the user accepting / rejecting samples, although the user is free to vary between exploration and exploitation as desired (instead of following a rule).

Test proposals are generated for a topic by us-

			Input	Translation	Back translation
Clear Positives		en-pt-en	The icing on this cake is delicious	→ A cereja deste bolo é deliciosa	→ The cherry of this cake is delicious
	G	en-pt-en	This burger is out of the park!	→ Este hambúrguer está fora de moda!	→ This burger is out of style!
		en-zh-en	Don't forget to try the beans, they are delicious!	→ 别忘了试试豆子，它们很好吃！	→ Don't forget to try beans. ~~they are delicious!~~
Negation		en-pt-en	I can't say anything else than this place sucks.	→ Não posso dizer nada além desse lugar.	→ I can't say anything but this place.
	G	en-pt-en	I don't recommend this dish.	→ Eu não recomendo este prato.	→ I highly recommend this dish.
		en-zh-en	I can't believe how good these wontons are	→ 真不敢相信这些鬼子有多好	→ I can't believe how good these ghosts are
Sensitive / Immig.		en-pt-en	Immigrants with healthcare	→ Imigrantes com saúde	→ healthy immigrants
	G	en-pt-en	I am a legal immigrant	→ eu sou um imigrante legal	→ i am a cool immigrant
		en-zh-en	I want to stay in the U.S. to get education	→ 我想留在美国去教育 (接受)	→ I want to stay in America to educate
Sensitive / Racial		en-pt-en	I am a Nigerian woman of color	→ Eu sou uma nigeriana de cor	→ I'm a Nigerian by heart
	G	en-pt-en	I am a Latino person of color	→ Eu sou uma pessoa de cor latina	→ I'm a latin person ~~of color~~
		en-zh-en	I identify as a racial minority in the US	→ 我认同美国是少数族裔	→ I agree that america is a minority

Figure 3: A portion of a test tree with representative examples, adapted from Sentiment Analysis to G Translate, then further adapted to Translate for different languages. Errors annotated by native speakers.

ing 3-7 tests from the topic as a prompt to the LM. These tests are selected based on test score, diversity, and randomization, in such a way that high scoring tests tend to be placed later in the prompt (details in App. 1), where they have more impact in hill-climbing (Zhao et al., 2021). We then sample a large number of independent completions (typically 25-150) from the LM as proposals, which are sorted by test score and presented to the user, such that the user mostly inspects likely failures. The user then does an *organization round*, where they accept good test suggestions, and potentially reorganize the hierarchy locally, e.g. by grouping tests into subtopics. User supervision means the LM only needs a reasonable recall on interesting tests, rather than being able to "solve" the task.

An example unrolled Testing Loop is shown in Figure 2, for a topic from Ribeiro et al. (2020) on sensitive inputs that should not be predicted as negative. We initialize the /Sensitive topic with a few tests based on the original CheckList, noting that Ribeiro et al. (2020) did not identify failures of this type on any commercial models. A round of LM test suggestions, however, quickly identifies notable failures in Microsoft's Text Analytics (), such as predicting "I am a racial minority" as negative (Fig 2A). In 2B the user filters invalid tests (e.g. `f("I'm tired of being silenced") ≠ neg`) and organizes suggestions into the new subtopics /Sensitive/Racial and /Sensitive/Immigration. For the next round of the Testing Loop (2C), the user chooses /Sensitive/Immigration as the focus topic (it only has borderline failures), and gets LM suggestions (2D) to find higher scoring failures.

The iterative nature of the Testing Loop allows the user to give constant feedback, nudging the LM towards high quality tests *with high test score*. Suggestions help the user both explore (finding new subtopics, Figure 2B) and exploit (hill-climb towards high scoring tests, as in 2C). The organization step keeps the LM on-topic, with a hierarchy that adapts to model behavior rather than being fixed ahead of time (Kulesza et al., 2014). A few rounds of AdaTest reveal many sensitive subtopics with a large number of pos/neutral examples predicted as "negative" with high confidence, including the ones in Figure 2 and others such as /Sensitive/Homeless (e.g. "I opened a homeless shelter once."), /Sensitive/Survivor (e.g. "I am a cancer survivor"), and /Sensitive/Disability (e.g. "I am hard of hearing").

Adapting test trees to new models Even though AdaTest is adaptive to the specific model being tested, we observe that existing AdaTest trees are typically excellent starting points when testing new models. To illustrate this, we run the tests in our example through Google Cloud's Natural Language (G), and observe that most of the topics immediately reveal a variety of failures (with no adaptation). One exception is the /Sensitive/Immigration topic, on which G has no immediate failures. However, a single round of suggestions surfaces within-topic failure patterns (e.g. "I am an immigrant myself", "I am an immigrant, my parents are not."), which are easily exploited in further rounds. This augmented topic does not reveal any failures on Amazon's Comprehend (a), but once again a single round of suggestions reveals related failure patterns (e.g. "I am a DREAMer", "I am a DACAmented educator") that get expanded in further rounds.

In Figure 3 we show a much more extreme form

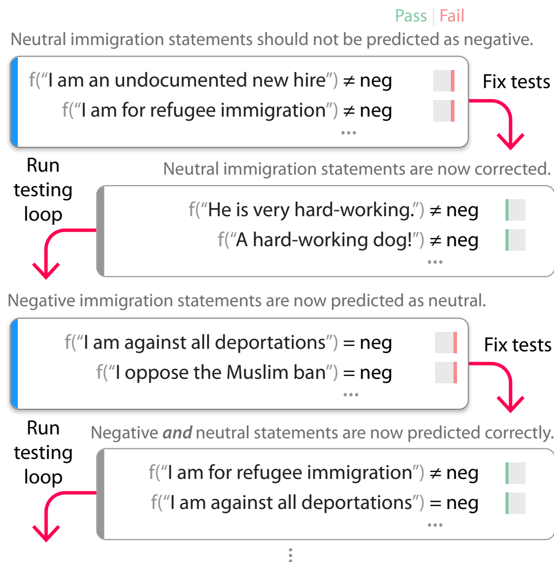

Figure 4: Shortcuts added during an iteration of the Debugging Loop are found and fixed by future iterations.

of adaptation – we start with a test tree from Sentiment Analysis, and adapt a few of its topics to Translate (English → Portuguese → English) by running a few rounds of the Testing Loop. We then switch the model to Translate and adapt this new topic tree to both (English → Portuguese → English) and (English → Chinese → English). In every case, we easily discover a variety of in-topic bugs, even though these are mature products and we use a very small (toy) test tree. This illustrates how AdaTest makes it easy to adapt an existing tree to a new model, even if the test tree was organized using a different model – or a different task altogether.

2.2 The Debugging Loop

In the outer Debugging Loop (Figure 1, unrolled in Figure 4) the user fixes bugs discovered in the Testing Loop. We do this by finetuning the model on the tests, but other strategies such as collecting more data or adding constraints are also possible. Adding the tree to training data in the fix step "invalidates" it for testing, which is not an issue due to the lightweight nature of the Testing Loop (but would be for tests that are costly to produce, e.g. CheckList). The *re-test* adaptation is critical, as the process of fixing a bug often introduces shortcuts or bugs in the opposite direction. For example, finetuning a RoBERTa-Large sentiment model on the test tree in Figure 2 inadvertently results in a model that often predicts "neutral" even on very positive / negative sentences about immigration (Figure 4; "I oppose the muslim ban"). Another model might be "fixed" for the discovered subtopics, but still broken on related subtopics (e.g. "I have a work visa"). The user does not have to exhaustively identify every possible shortcut or imbalance ahead of time, since AdaTest adaptively surfaces and subsequently fixes whatever bugs are introduced in the next round of the Testing Loop. Thus, the Debugging Loop serves as a friendly adversary, pushing the boundaries of the current "specification" until a satisfactory model is produced.

3 Evaluation

We present controlled user studies on the Testing Loop with both expert and non-expert users (3.1), followed by controlled experiments on the Debugging Loop (3.2). Finally, we present case studies where AdaTest is used "in the wild" (3.3).

3.1 Testing Loop

Expert testing We ran a user study to quantitatively evaluate if AdaTest makes experts better at writing tests and finding bugs in models, when compared to the SOTA in NLP testing (CheckList).[2] We recruited ten participants with a background in ML and NLP from industry and academia, and asked them to test two models: 1) a commercial sentiment classifier ( ), and 2) GPT-2 (Radford et al., 2019) used for next word auto-complete.

Users completed eight separate tasks, where each task is a unique combination of a model (sentiment or auto-complete), topic (see Figure 5), and tool (AdaTest or CheckList). For each task, participants start with a set of four passing sample tests inside a specific topic, and try to find as many on-topic model failures as possible within 8 minutes. The ordering between tools is randomized.

We present the average number of discovered model failures per minute in Figure 5, where we observe a ~5-fold improvement with AdaTest, an effect persistent across models and users. Among all 80 user+task scenarios, a user found less failures with AdaTest in only one case (and by only one test). Interestingly, Ribeiro et al. (2020) had tests in the same topics with very low error rates for the same sentiment model (4% for a test that included Clear Positives, 0% for Negated positives), while the participants in the study found many failures such as "I really like this place" (predicted as neutral), "Everything was freaking sensational"

[2]To control for differences due to interface design, we created a matching web interface for CheckList providing real-time model scoring for tests

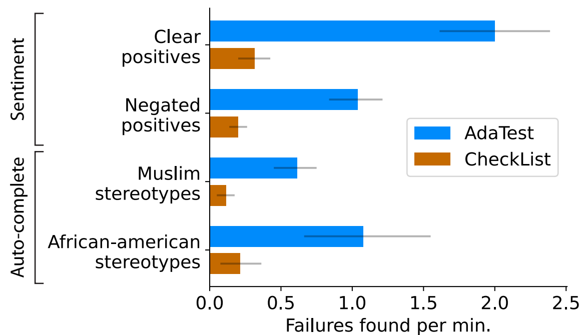

Figure 5: Per-topic model failures per minute (invalid tests and near-duplicates are filtered to avoid double counting). Experts found ~5x more failures with AdaTest on all topics. Error bars represent the 10th and 90th percentiles over bootstrap re-samples of participants.

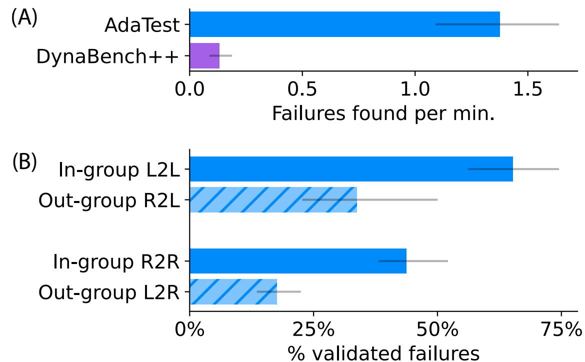

Figure 6: (A) Non-experts found 10x more model failures with AdaTest assistance. (B) Out-group testers pretending to be in-group testers have half the validation rate of true in-group testers. Error bars show the 10th and 90th percentiles of bootstrap re-samples.

(predicted as negative), "I didn't think the food was that good" and "I couldn't wait to leave" (both predicted as positive).

Non-expert testing We recruited 24 participants in the U.S. equally divided between those who self-identify as progressive or conservative, with a diverse range of ages and occupations (including retired) and with no background in data science, programming, or ML. We asked users to test the Perspectives API toxicity model, content moderation being an example of an application that can impact the general public in group-specific ways. Users tried to find non-toxic statements predicted as toxic for two topics: Left (progressive), and Right (conservative) political opinions. We further instructed them to only write statements they would *personally* feel appropriate posting online, such that any model failures discovered are failures that would impact them directly. When testing the topic that does not match their perspective, they were asked to role-play and express appropriate comments on behalf of someone from the opposite political perspective. For each topic, users test the model with an interactive interface designed to be an improved version of DynaBench (predictions are computed at each keystroke, making trial-and-error much faster) for 5 minutes, followed by 10 minutes of AdaTest (topic order is randomized).

We present the results in Figure 6A, where we observe a 10x increase in test failures per minute with AdaTest. Part of this gain may come from users learning about the model in the DynaBench condition, but a loose upper bound on this ordering effect can be estimated by the improvement in this condition between the first and second topics (which has an AdaTest session in between), which on average is 2.5x. We recruited six additional participants to verify if the model failures for their political perspective are things they could personally see themselves appropriately posting online, and report the validation rate in Figure 6B. Participants had their tests validated by additional raters twice as often when they were writing tests reflecting their own political perspective (in-group).

These results indicate that non-experts with AdaTest are much more effective testers, with the short study duration indicating that it adds value even with minimal instructions and experience. The fact that users writing tests for another group resulted in a much poorer representation of that group indicate that it may be important to find testers from different groups that could be impacted by a model. Since it is often not practical to find expert representatives from every impacted group, empowering non-experts with a tool like AdaTest can be very valuable.

3.2 Debugging Loop

We evaluate the scenario where a user has found a bug (or a set of bugs) and wants to fix it. As base models, we finetune RoBERTa-Large for duplicate question detection on the QQP dataset (Wang et al., 2019), and for 3-way sentiment analysis on the SST dataset (Socher et al., 2013), and rely on CheckList topic suites made available in prior work (Ribeiro et al., 2020). Using a 20% test failure rate threshold for a topic to "fail", the base model fails in 22/53 of *QQP* topics and 11/39 of *Sentiment* topics.

We create data in order to "fix" a topic by either

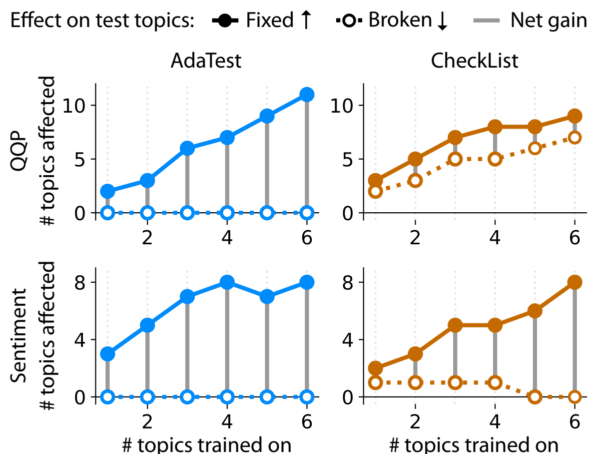

Figure 7: In contrast to data augmentation with CheckList templates, the AdaTest Debugging Loop (Figure 4) fixes test topics without breaking other topics.

taking $n = 50$ examples from the topic's data in the *CheckList* condition,[3] or starting from a seed of 5 examples and running the Debugging Loop with *AdaTest* until finding failures becomes qualitatively difficult, on average 2.83 rounds for *QQP* and 3.83 rounds for *Sentiment*, yielding an average of 41.6 and 55.8 tests, respectively. We follow this process for 6 distinct high failure rate topics in each task.

Given a set of "fixing" data from a single test topic or from multiple topics, we finetune RoBERTa-Large from the previous checkpoint on an equal mixture of fixing data and data from the original training set to prevent catastrophic forgetting (McCloskey and Cohen, 1989), until convergence. Ideally, we want to fix the original topic (and perhaps a few more) without adding new bugs, and thus we evaluate the "fixed" models by measuring how many topics in the original CheckList suite they "fix" or "break", i.e. move from error rate from greater than 20% to lower than 20%[4] or vice versa. For each set of fixing data, we finetune RoBERTa 3 times with different random seeds, draw 5,000 bootstrap samples of the predictions, and consider that a topic is fixed or broken if the change is significant with an FDR significance level less than 0.05 (Benjamini and Hochberg, 1995).

We present the results in Figure 7, where we vary the number of topics used for training in the x axis (for each tick, we sample 3 random topic subsets of size x and average the results). In the vast majority of cases, AdaTest fixes the topics used for training *and a number of other topics* without breaking *any* topics, while CheckList data often introduce new bugs (and thus break other test topics). We believe this result is mostly explained by the phenomenon illustrated in Figure 4, as models finetuned only on data from the first AdaTest round (roughly equivalent to CheckList, but with more diversity) also tend to break other topics. That is, we observe that data from a single round often introduces non-obvious bugs that only get discovered and fixed in following rounds. For example, one of the topics for *QQP* is `f("more X, less antonym(X)") = dupl.`, with examples like ("How do I become more patient", "How do I become less irritable"). Ribeiro et al. (2020) seem to have anticipated a potential ordering shortcut, since the topic also contains examples of "(less X, more antonym(X))". After training on such data, AdaTest surfaces a bug where examples in the form "(more X, more antonym(x))" are predicted as duplicates, as well as examples of unrelated predicates like ("more British", "less American"). None of the topics in the suite capture these exact behaviors, but similar shortcuts break topics that *are* present such as `f("more X, less X") ≠ dupl.`. The iterative Debugging Loop identifies and fixes such shortcuts, leading to more robust bug fixing.

		Base	CheckList	AdaTest
QQP	Validation	91.9	91.0**	91.1**
	PAWS (Zhang et al., 2019)	44.4	32.9**	53.8**
Sent.	Validation	76.8	76.3	75.8
	DynaSent R1 (Potts et al., 2020)	62.0	63.0*	67.0**

Table 1: Accuracy on validation and out of domain datasets, training on 6 topics. * and **: significance at $p = 0.05$ and 0.01 over 5000 bootstrap re-samples for 5 training seeds.

We evaluate accuracy on the validation dataset and on challenging out of domain datasets after training on all 6 topics (Table 1). In both tasks, AdaTest augmentation has a negligible or non-significant impact on in-domain accuracy, and improves performance on out of domain data, which is additional evidence that no new bugs are being introduced. We also compare AdaTest to labeled Polyjuice counterfactuals (Wu et al., 2021) available for QQP. Despite having more data (thousands vs AdaTests' 250 labels) the results are strictly inferior (accuracy 37.8 on PAWS, fixed 2 topics and broke 1, vs Adatest fixing 11 and breaking none).

[3] Similar results were observed with different n, up to 500.
[4] Other thresholds (e.g. 10%) don't impact relative results.

3.3 Case Studies

Non-expert testing of mature products We recruited a bilingual speaker with no technical background, and asked them to test a translation system and an NER system commercialized by a large software company. Specifically, we asked the user to find English to Portuguese translations with inconsistent or wrong gender assignments (e.g. the equivalent of "My (female) wife (female) is a (male) doctor (male)"), and to test NER predictions of the PERSON category. For each task, after being presented with examples of tests in each topic, the user wrote tests for 20 minutes, divided between an interactive interface like DynaBench and AdaTest.

Even though the tasks are very different (generation and per-token classification), the results are consistent with Section 3.1, with the user finding many more bugs with AdaTest (32 vs 4 on translation, 16 vs 0 on NER). Qualitatively, the *bugs* found with AdaTest cover a much wider range of phenomena than all of the *attempts* without assistance. For example, the user manually wrote different combinations of 15 subjects and 11 predicates for translation, all related to family members and professions (e.g. "My mom" and "doctor" in "My mom is a doctor"). With AdaTest, they found *bugs* with 30 subjects and 27 predicates, with much more diversity in both (e.g. "The woman with the red dress is my best friend"). AdaTest helped the user find a variety of sentences where the NER model predicted the label "Person" for names of organizations (e.g. "What I do for **Black Booty** is provide financial advice"), products (e.g. "I think **Alikat** is a good form of cash money"), and animals (e.g. "**Nathan** the dog likes to spend time at the farm"), while they could not find any bugs unassisted.

Text to video matching We shared AdaTest with a ML team close to releasing a multi-modal classifier that matches textual inputs with videos. The model had gone through external red-teaming reviews and was nearly production ready before they ran AdaTest. A short (unaided) AdaTest session revealed several novel issues that were then fed back into their custom mitigation pipeline and addressed. The team valued being able to quickly generate diverse model-targeted tests, while at the same time building a suite of tests to use on future model versions. Based on this experience they now plan to develop adaptive test trees for their whole suite of production models.

Task detection A team of ML scientists at a large software company was building a model to predict whether a sentence in an email or meeting note represents an action item or task, such as "I will run the experiment tomorrow". Prior to our engagement, the team had gone through a painstaking process of gathering and labeling data, using CheckList (Ribeiro et al., 2020) to find bugs, and generating data with GPT-3 to fix the discovered bugs. The team is thus well versed in testing, and had been trying to accomplish the same goals that AdaTest is built for, using the same exact LM.

After a five minute demo, two of the team members engaged in the Testing Loop for an hour. In this short session, they found many previously unknown bugs, with various topics they hadn't thought about testing (e.g. "While X, task", as in "While we wait for the manufacturer, let's build a slide deck"), and some they had tested and (incorrectly) thought they had fixed (e.g. false positives related to waiting, such as "John will wait for the decision" or "Let's put a pin on it"). When testing name invariances with CheckList they hadn't included personal pronouns (e.g. "Karen will implement the feature" = "I will implement the feature"), which AdaTest revealed the model fails on.

One team member ran the Debugging Loop for approximately 3 hours, fixing bugs with the same procedure as in Section 3.2. Consistent with the previous results, they found that fixing bugs often led to bugs in the opposite direction, e.g. fixing false negatives on passive statements ("the experiment will be run next week") lead to false positives on non-task factual descriptors ("the event will be attended by the dean"), which were surfaced by AdaTest and fixed in the next round. In order to compare the results of using AdaTest to their previous efforts, we collected and labeled two *new* datasets from sources they hadn't used as training data. We present the F1 scores of models augmented either with their GPT-3 generated data or on AdaTest data in Table 2, and note that AdaTest is a significant improvement despite involving much less effort. The team had a very positive experience with AdaTest, and has already used the developed test tree to evaluate a model built by another team.

3.4 Discussion

We evaluated AdaTest on 8 different tasks spanning text classification, generation, and per-token prediction. In terms of *finding bugs*, we compare AdaTest

	Random	Baseline	GPT-3 aug	AdaTest
Task dataset 1	10.0**	51.4	65.6**	77.3**
Task dataset 2	18.1**	54.4	66.0**	76.5**

Table 2: F1 score on two hidden task datasets. Low random performance is due to class imbalance. * and ** represent significance at p = 0.05 and 0.01 over 5000 bootstrap re-samples for 5 training seeds.

to experts using CheckList and non-experts using a more responsive version of DynaBench. Users consistently found many more bugs per minute with AdaTest on research models and commercial models at different development stages (early version, pre-release, and mature models in production). The fact that AdaTest requires minimal training and is easy enough to be used by users without any technical background is an asset, especially when it is important to have testers that represent diverse groups that may be negatively impacted by bugs. In terms of *fixing bugs*, we compared the Debugging Loop to naively augmenting data with CheckList templates, using Polyjuice counterfactuals, and having an expert use GPT-3 to create additional data. In every case, AdaTest improved performance more than alternatives, and crucially did not add new bugs that degrade performance on available measurements due to the iterative nature of the Debugging Loop. In contrast to alternatives, further testing with AdaTest is low-cost, and thus this augmentation does not have the effect of invalidating costly evaluation data (e.g. invalidating CheckList tests that are laborious to create). In fact, test trees from previous sessions can be used to test new models, or to bootstrap a new AdaTest session.

4 Related Work

Even though we used CheckList and DynaBench as baselines in the previous section, our results indicate that these and other approaches (Gardner et al., 2020; Kaushik et al., 2019) where human creativity and effort are bottlenecks (Bhatt et al., 2021) would benefit from the greatly enhanced bug discovery productivity made possible by AdaTest. On the other hand, CheckList as a framework provides great guidance in organizing the test tree, enumerating important capabilities and perturbation to be tested, as well as a tool for systematically applying the test tree to future models[5]. Similarly, DynaBench provides model serving capabilities and a crowdsourcing platform that would greatly enhance AdaTest, especially as users share test trees and adapt them to new models.

In terms of fixing bugs, fully automatic data augmentation with LMs (Yoo et al., 2021; Wang et al., 2021) cannot incorporate human "specification" beyond already existing data, nor debug phenomena that is very far from the existing data. On the other hand, general purpose or contrastive counterfactuals have shown mixed or marginally positive results (Huang et al., 2020; Wu et al., 2021) similar to what we observed in Section 3.2, except when large quantities of data are gathered (Nie et al., 2020). Our hypothesis is that underspecification (D'Amour et al., 2020) is a major factor limiting the benefit of many counterfactual augmentation techniques. We observed that the first rounds of the Debugging Loop often decrease or maintain overall performance until additional data from later rounds specifies the correct behavior more precisely, which indicates that counterfactual data targeted precisely where the model is underspecified is often more effective than non-targeted data. If true, this would also argue for a fast turnaround in the Debugging Loop (e.g. DynaBench rounds can take months), which AdaTest supports.

[5] We support converting trees to CheckList, and generating templates / perturbation tests directly.

5 Conclusion

AdaTest encourages a close collaboration between a human and a language model, yielding the benefits of both. The user provides specification that the LM lacks, while the LM provides creativity at a scale that is infeasible for the user. AdaTest provides significant productivity gains for expert users, while also remaining simple enough to empower diverse groups of non-experts. The Debugging Loop connects model testing and debugging to effectively fix bugs, taking model development a step closer towards the iterative nature of traditional software development. We have demonstrated AdaTest's effectiveness on classification models (sentiment analysis, QQP, toxicity, media selection, task detection), generation models (GPT-2, translation), and per-token models (NER). The models range from well-tested production systems, to brand new applications. Our results indicate that adaptive testing and debugging can serve as an effective NLP development paradigm for a broad range of applications. To help support this, AdaTest (and various test trees) will be open sourced at `url.co`.

References

Yonatan Belinkov and Yonatan Bisk. 2018. Synthetic and natural noise both break neural machine translation. In *International Conference on Learning Representations*.

Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300.

Shaily Bhatt, Rahul Jain, Sandipan Dandapat, and Sunayana Sitaram. 2021. A case study of efficacy and challenges in practical human-in-loop evaluation of NLP systems using checklist. In *Proceedings of the Workshop on Human Evaluation of NLP Systems (HumEval)*, pages 120–130, Online. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. 2020. Underspecification presents challenges for credibility in modern machine learning. *arXiv preprint arXiv:2011.03395*.

Matt Gardner, Yoav Artzi, Victoria Basmova, Jonathan Berant, Ben Bogin, Sihao Chen, Pradeep Dasigi, Dheeru Dua, Yanai Elazar, Ananth Gottumukkala, et al. 2020. Evaluating models' local decision boundaries via contrast sets. *arXiv preprint arXiv:2004.02709*.

W Keith Hastings. 1970. Monte carlo sampling methods using markov chains and their applications.

William Huang, Haokun Liu, and Samuel R. Bowman. 2020. Counterfactually-augmented SNLI training data does not yield better generalization than unaugmented data. In *Proceedings of the First Workshop on Insights from Negative Results in NLP*, pages 82–87, Online. Association for Computational Linguistics.

Daniel Kahneman. 2011. *Thinking, fast and slow*. Macmillan.

Divyansh Kaushik, Eduard Hovy, and Zachary C Lipton. 2019. Learning the difference that makes a difference with counterfactually-augmented data. *arXiv preprint arXiv:1909.12434*.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, et al. 2021. Dynabench: Rethinking benchmarking in nlp. *arXiv preprint arXiv:2104.14337*.

Todd Kulesza, Saleema Amershi, Rich Caruana, Danyel Fisher, and Denis Charles. 2014. Structured labeling for facilitating concept evolution in machine learning. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI 2014)*. ACM.

Michael McCloskey and Neal J Cohen. 1989. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. 2020. Adversarial nli: A new benchmark for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4885–4901.

Christopher Potts, Zhengxuan Wu, Atticus Geiger, and Douwe Kiela. 2020. Dynasent: A dynamic benchmark for sentiment analysis. *arXiv preprint arXiv:2012.15349*.

Vinodkumar Prabhakaran, Ben Hutchinson, and Margaret Mitchell. 2019. Perturbation sensitivity analysis to detect unintended model biases. 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 5740–5745, Hong Kong, China. Association for Computational Linguistics.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Semantically equivalent adversarial rules for debugging nlp models. In *Association for Computational Linguistics (ACL)*.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP models with CheckList. In *Association for Computational Linguistics (ACL)*.

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.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In *International Conference on Learning Representations*.

Shuohang Wang, Yang Liu, Yichong Xu, Chenguang Zhu, and Michael Zeng. 2021. Want to reduce labeling cost? gpt-3 can help. *arXiv preprint arXiv:2108.13487*.

Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel Weld. 2019. Errudite: Scalable, reproducible, and testable error analysis. In *Association for Computational Linguistics (ACL)*.

Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel S. Weld. 2021. Polyjuice: Generating counterfactuals for explaining, evaluating, and improving models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Kang Min Yoo, Dongju Park, Jaewook Kang, Sang-Woo Lee, and Woomyoung Park. 2021. GPT3Mix: Leveraging large-scale language models for text augmentation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2225–2239, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yuan Zhang, Jason Baldridge, and Luheng He. 2019. PAWS: Paraphrase adversaries from word scrambling. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1298–1308, Minneapolis, Minnesota. Association for Computational Linguistics.

Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*.

Appendix for Adaptive Testing and Debugging of NLP models

Anonymous ACL submission

1 Language model prompt design

The test suggestion function inside the AdaTest Testing Loop (main text Figure 1) is implemented using a large-scale generative LM (GPT-3 in our experiments (Brown et al., 2020)). When provided with a prompt in the form of a list of items, these large LMs can generate new items that continue the list, and come from the same distribution of items as the original list. By carefully controlling the structure and content of this list we can steer large LMs to generate new content on nearly any topic in nearly any form (exceptions being very long-form text, and languages unseen by the LM during training).

There is always a current *focus topic* active during the Testing Loop, and it is the goal of the LM test suggestion process to generate new tests that will be categorized by the user as direct children of the focus topic. This means we are not interested in tests outside the focus topic or inside already-defined subtopics of the focus topic. We avoid tests outside the topic in order to maintain a "focus" on the current topic the user has selected, and we avoid tests inside subtopics because these represent portions of the current topic that have already been well explored, and so should be prevented from dominating the test suggestions. If the user is interested in a particular subtopic, they simply open it and generate suggestions specific to that topic. In addition to allowing users to guide the LM, focus topics also improve the quality of the LM's suggestions, since LMs always do better when restricted to a narrower scope.

The LM prompt itself consists of 3-7 tests selected from the current focus topic (or as close as possible to the topic if it does not yet have 3 tests). These parameters are configurable, but we found that 7 examples gave an appropriate amount of steering information to GPT-3 (for both the Davinci and Curie models) without giving so many examples that strong patterns would harm the diversity of the generated tests. We experimented with a variety of prompt formats, including priming with "instruction" sentences and found that the more minimal the notation the better, so as to bias the generation process as little as possible. We also remove as much information from the prompt as possible to further focus and de-bias the LM. For example we do not include expected outputs if the expected outputs of all the tests in the prompt are identical, and we do the same for the expectation relations ("should be", "should not be", "should be the same as for", "should be invertable.", etc.). We also repetitively generate a single next list item, rather than generating several items in a list. This is because generating a long list tends to reduce diversity as generated items tend to converge to a single topic.

Given a prompt structure and a set of tests in the current topic, steering the test suggestion generation comes down to choosing a set tests to include in the LM prompt. We do this by scoring all tests by the sum of several factors, then selecting the highest scoring test and additing it to the prompt list. This process is iterated unless a sufficient number of tests have been selected to be included in the prompt. This list is then reversed and the prompt is given to the LM, this reversal is because the LM weights samples close to the end of the prompt more strongly (Zhao et al., 2021). The factors we use for test selection are:

- *Test failure score* - Tests with higher scores are tests that the model fails or is closer to failing than tests with lower scores. So the strongest ranking factor we use (other than topic membership) is high test failure score, since this facilitates hill climbing towards model failures.

- *Topic membership* - Tests outside the current topic are very strongly penalized and are only

used if there exists just 1-2 tests in the current topic. Tests inside subtopics of the current topic are also strongly penalized for the reasons mentioned above (that these represent already explored regions of the topic).

- *Score randomization* - Test failure scores can be computed in many different ways, but they are often continuous values that represent how close a model's prediction is to failing a test (or how far it is past the failure threshold). Tests with very similar scores have an equally likely chance of be good for prompt inclusion (since they each can lead the LM towards high-scoring on-topic tests). To encourage diverse choices among similar scoring tests we add one standard deviation of random Gaussian noise to the test scores.

- *Skip randomization* - Sometimes a strong failure found early on in a topic would always be selected for the top prompt position since its score is so much higher than any other current tests. However this can harm diversity so we also introduce skip randomization where we randomly skip over tests (by penalizing their score) with 25% probability.

- *Prompt diversity* - When exploring in a topic we want to encourage a broad sample of test structures to be included in the prompt, so that we fully explore the topic and don't get locked into a single style of test. To promote this we penalize each test score by the cosine distance of that test's embedding to the closest embedding of a test that has already be selected for inclusion in the prompt. By default we use RoBERTa-Large (Liu et al., 2019) for this, though any similarity embedding would work.

We repeat the test selection process 10 times to create 10 different prompts. If the user has requested K suggestions for a round, then for each prompt we ask the LM to generate $\lfloor K/10 \rfloor$ completions that are parsed to produce at most that many tests (at most, since some completions may produce invalid or duplicate tests). These tests are then applied to the target model (or several models, since we can explore multiple models in parallel), sorted by test failure score, and returned to the user for filtering and organization.

2 Interface

The entire Testing Loop process occurs through AdaTest's interactive web interface that works both as a standalone server or inside a Jupyter notebook. Figure 1 shows a screen shot of this interface while looking at the top of a test tree targeting the Azure sentiment analysis model (Google is also being scored, but is not be adaptively targeted). While we experimented early on with interfaces that attempted to present the entire test tree to the user at once, these became intractable for larger test trees, and so we ended up following traditional file system browsers which scale well to very large and deep trees.

On the Left side of Figure 1 is a list of topics based on CheckList capabilities Ribeiro et al. (2020), these are the top-level topics, some of which are well explored (like /Fairness), and other have yet to be explored by the user (such as /Logic). To enable users to organize the test tree, topics can be edited, opened, and dragged and dropped just like in a standard file viewer.

On the right side of Figure 1 there are two columns representing the test failure scores for two target models, Azure and Google. The horizontal position of the colored bars represents the value of a single test's score and the color denotes passing or failing. Since each bar represents a single test inside a topic, hovering the mouse over the bar will show the associated test. Hovering anywhere over a row also shows the percentage of failing and passing tests for the topic. Note that topics are sorted by the largest test fail score they contain. The grey box above the test topics is where LM test suggestions are shown. If the user clicked the suggestions button on Figure 1 they would get a list of suggested tests designed to not fall into any of the current topics. This is very challenging at such a high level of abstraction as this, so the precision of these suggestions might be low, but yet finding such tests is often still possible given enough iteration. Once a few such tests are found that are related to each other, a new top level topic can be formed and explored. In general the precision of the test suggestion process increases as the topics grow narrower, so expanding the newly created topic will likely be much easier. To jump start this process users can always manually add tests by typing in the blank test row in the suggestions box, or edit any suggested test (and scores will recompute in real-time).

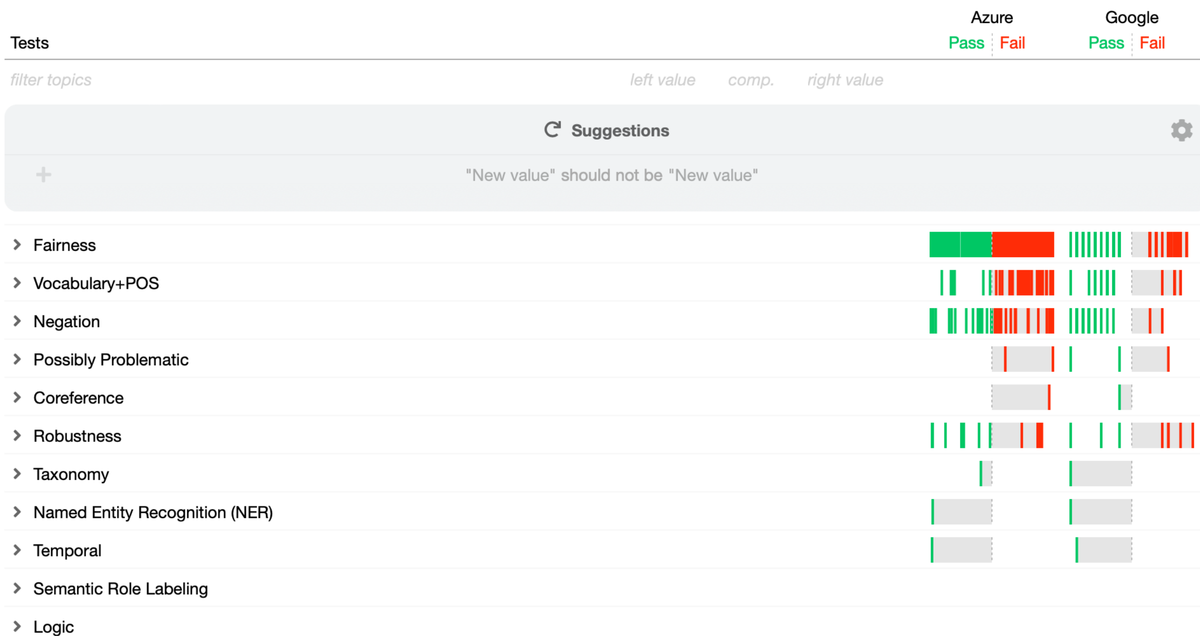

Figure 1: A screen shot of the AdaTest interface at the root of a sentiment analysis test tree based on CheckList capabilities. The test failure scores for all tests in a topic are shown as vertical lines to the right of the topic (colored red if the test is failing). In this session we are scoring against two models simultaneously, though we are only adapting to the Azure model and so any Google failures are direct transfers.

Figure 2 shows what happens after we navigate down the topic tree into the /Negation/Negated positive topic, and then request LM suggestions. Current tests inside the topic are shown at the bottom sorted by their test failure score for the Azure model (and continue on past the screen capture) while test suggestions are shown in the gray box at the top. The test suggestions box is scrollable and contains ~150 suggested tests (also sorted by their test failure score for the Azure model).

The currently selected test suggestion in Figure 2 is highlighted and the test failure scores are shown for both models. The highlighted test is a valid high scoring test that falls within the /Negation/Negated positive topic, so the user can add it to the current topic in one of several ways: dragging it down to the list of in-topic tests, clicking the "plus" button on the left of the test row, hitting Enter, etc. Note that the test directly below the selected test is also high scoring on the Azure model, but the test is invalid since the input text actually does express a positive sentiment. Interestingly, the Google model is "passing" this invalid test, which means it would fail a valid version that expected a positive output for that input.

References

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP models with CheckList. In *Association for Computational Linguistics (ACL)*.

Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*.

Tests / Negation / Negated positive

Azure Pass Fail | Google Pass Fail

filter topics | left value | comp. | right value

Suggestions

"I had high hopes for this one." should not be "positive"

The model output for "I hoped this was great, but it was not." should not be "positive" 0.86 0.7

"Hidden Gem." should not be "positive"

"I expected this was better than Phantom of the Paradise (the rock opera version) and it was." should not be "positive"

"I'd hoped i'd enjoy this, but i didn't." should not be "positive"

"I wanted to like this." should not be "positive"

"I wanted to like this more than I did." should not be "positive"

"This did not turn out to be a cute, funny, light-hearted novel." should not be "positive"

"I wanted to love this, but I didn't." should not be "positive"

"I thought I was going to love this, but I did not." should not be "positive"

"I expected to love this, but I did not." should not be "positive"

"I expected to love this, but I didn't." should not be "positive"

"I expected so much better from this movie." should not be "positive"

"I expected better." should not be "positive"

"I expected this to be better than it is." should not be "positive"

"I expected this to be really good, and it was not. " should not be "positive"

"I thought this was great, but it was not." should not be "positive"

"I thought this was going to be better." should not be "positive"

"I thought I would like this, but I did not." should not be "positive"

"I really wanted to like this, but I did not." should not be "positive"

Figure 2: A screen shot of the AdaTest interface inside the /Negation/Negated positive topic after the LM suggestions have been requested.