Multilingual CheckList: Generation and Evaluation

Karthikeyan K[3]*, Shaily Bhatt[1]*, Pankaj Singh[1], Somak Aditya[4],
Sandipan Dandapat[2], Sunayana Sitaram[1], Monojit Choudhury[1]

[1] Microsoft Research, Bengaluru, India
[2] Microsoft R&D, Hyderabad, India
[3] Department of Computer Science, Duke University
[4] Department of CSE, IIT Kharagpur

karthikeyan.k@duke.edu, saditya@cse.iitkgp.ac.in,
{t-shbhatt,t-pasingh,sadandap,sunayana.sitaram,monojitc}@microsoft.com

Abstract

Multilingual evaluation benchmarks usually contain limited high-resource languages and do not test models for specific linguistic capabilities. CheckList (Ribeiro et al., 2020) is a template-based evaluation approach that tests models for specific capabilities. The CheckList template creation process requires native speakers, posing a challenge in scaling to hundreds of languages. In this work, we explore multiple approaches to generate Multilingual CheckLists. We device an algorithm – **T**emplate **E**xtraction **A**lgorithm (TEA) for automatically extracting target language CheckList templates from machine translated instances of a source language templates. We compare the TEA CheckLists with CheckLists created with different levels of human intervention. We further introduce metrics along the dimensions of *cost*, *diversity*, *utility*, and *correctness* to compare the CheckLists. We thoroughly analyze different approaches to creating CheckLists in Hindi. Furthermore, we experiment with 9 more different languages. We find that TEA followed by human verification is ideal for scaling Checklist-based evaluation to multiple languages while TEA gives a good estimates of model performance. We release the code of TEA and the CheckLists created at aka.ms/multilingualchecklist

1 Introduction

Multilingual transformer based models (Devlin et al., 2019; Conneau et al., 2020; Liu et al., 2020; Xue et al., 2021) have demonstrated commendable zero & few-shot capabilities. Their performance is typically evaluated on benchmarks like XNLI (Conneau et al., 2018), XGLUE (Liang et al., 2020), XTREME (Hu et al., 2020b) & XTREME-R (Ruder et al., 2021). However, this evaluation paradigm has a number of limitations including: First, most of these datasets are limited to a few high resource languages (Hu et al., 2020a; Wang et al., 2020; Vulić et al., 2020), except for a few tasks (e.g., NER, POS (Ahuja et al., 2022; Bhatt et al., 2021a)). Second, creating high quality test sets of substantial size for many tasks and languages is prohibitively expensive. Third, state-of-art models are known to learn spurious patterns to achieve high accuracies, saturating performance on these test-benches, yet performing poorly on often much simpler real world cases (Goyal et al., 2017; Gururangan et al., 2018; Glockner et al., 2018; Tsuchiya, 2018; Geva et al., 2019). Fourth, these benchmarks do not evaluate models for language specific nuances (Ribeiro et al., 2020). Lastly, this evaluation approach does not provide any insights into where the model is failing (Wu et al., 2019). These limitations lead to the need of interactive, challenging, and much larger testing datasets (like (Srivastava et al., 2022; Kiela et al., 2021)) and more holistic approaches to evaluation (like Ribeiro et al. (2020)).

CheckList (Ribeiro et al., 2020) is an evaluation paradigm that systematically tests the various *(linguistic) capabilities* required to solve a task. It allows creation of large and targeted test sets easily using various abstractions. Specifically, users can generate *templates*, essentially sentences with *slots* that can be filled in with a dictionary of *lexicons* to generate test *instances*. CheckList templates are created by native speakers. Ruder et al. (2021) introduce Multilingual Checklists created by human translation from English CheckList for 50 languages for a subset of tests on Question Answering. However, since CheckLists are task & language specific, human creation or translation of CheckLists remains extremely resource-intensive.

In this paper, we introduce an automatic approach to creating Multilingual CheckLists. We devise the **T**emplate **E**xtraction **A**lgorithm (TEA) for extracting templates in a *target* language from the translated instances of a *source* language CheckList (here English) automatically (§2). We also experiment with semi-automatic and manual approaches for Multilingual CheckList creation (§3). In the

Findings of the Association for Computational Linguistics: AACL-IJCNLP 2022, pages 282–295
November 20–23, 2022. ©2022 Association for Computational Linguistics

semi-automatic approach (TEA-ver), we ask human annotators to verify and correct the templates created by TEA. In the manual approach, we ask annotators to create CheckLists in two ways: first, by translation of English CheckList to the target language (t9n) (same as Ruder et al. (2021)); Second, by giving a description of the task and capabilities to create CheckLists from scratch (SCR) (same as original English CheckLists creation (Ribeiro et al., 2020)).Using these four approaches, we create CheckLists for Sentiment Analysis (SA) and Natural Language Inference (NLI) in Hindi (§5). We demonstrate broad applicability of TEA by generating CheckLists in additional 9 typologically diverse languages (Gujarati, French, Swahili, Arabic, German, Spanish, Russian, Vietnamese, Japanese) and TEA-ver CheckLists in 3 of them (§6).

Evaluation of CheckLists is non-trivial. For thorough comparisons, we propose evaluation metrics along four axes: *utility*, *diversity*, *cost* & *correctness* (§4). Our evaluation indicates that CheckLists created using TEA are not only cost-effective but also useful and diverse, with comparable quality to the manually and semi-automatically created CheckLists. Experiments on typologically diverse languages show that TEA CheckLists provide a good estimate of the failures of the model, and thus can be used even in the absence of resources to verify them or create human-annotated gold test-sets.

To summarize, our contributions are: a) We propose TEA (**T**emplate **E**xtraction **A**lgorithm) to extract templates in a target language using translated instances of a source CheckList. b) We experiment with varying degrees of human intervention, comparing semi-automatic & manual approaches of Multilingual CheckList creation with TEA, to understand the best utilization of the human effort. c) We introduce evaluation metrics along the axes of *utility*, *diversity*, *cost*, and *correctness* for in-depth comparison of the the CheckLists. d) We will release all the 4 CheckLists in Hindi for SA and NLI, TEA CheckLists in 9 languages for SA and TEA-ver CheckLists in 3 languages for SA.

We release the code of TEA and the CheckLists created at aka.ms/multilingualchecklist

2 TEA: Template Extraction Algorithm

Terminology (consistent with Ribeiro et al. (2020)):
***Linguistic* capabilities:** These are capabilities tested for a particular task. For e.g, negation.
Templates: These are sentences with slots. For e.g, '{CITY} is beautiful'. Here, '{CITY}' is a slot. Templates can have any number of slots.
Lexicon keys and values: This a dictionary of values. In the above example, 'CITY' is the key. Values are the words that would be filled in the slots (replacing the keys) like 'New Delhi', 'New York', 'London', etc. We use the notation 'CITY = ['New Delhi', 'London', 'New York'] ' for lexicons.
Instances: These are test sentences created by inserting lexicon values in templates . In the above example, the instances formed are: 'New York is beautiful', 'London is beautiful', etc.

The CheckList paradigm allows creation of large number of test instances. For multilingual evaluation, these can then be translated to the target languages using Machine Translation. However, there are limitations to this approach. Firstly, a large machine translated test set is difficult to be verified by humans, as one would have to go through every example. Second, it defeats the purpose of abstraction that CheckLists facilitates. And third, the quality of this test set will be directly impacted by the quality of the MT system. This results in the need to generate templates in the target language so that these can be utilized and verified in the same fashion as the template sets in the source language.

Our early experiments suggested that due to word order and syntactic differences between languages, both: 1) a word-to-word or heuristic translation of the template and 2) extraction of template from a single source instance (such as by simply replacing one word with other in a single translated instance) do not work well for template translation. This necessities a non-trivial algorithm that can extract templates given a set of instances.

We propose the **T**emplate **E**xtraction **A**lgorithm or TEA, to automatically extract template sets given an input a set of instances. In this paper, these input instances for TEA are obtained by machine translating instances created from the source CheckList template sets. We use machine translation to reduce cost and human effort, but the algorithm can be used with any input set of instances, i.e it would work with human-translated instances.

Briefly, TEA is a recursive approach to extract templates from input instances by treating every input instance as directed acyclic graph of the words. TEA combines the instances with similar structure into a single template by recursively merging instances and replacing terminals (or lexicon values) with non-terminals (or lexicon keys).

Figure 1 shows how TEA creates lexicon keys by

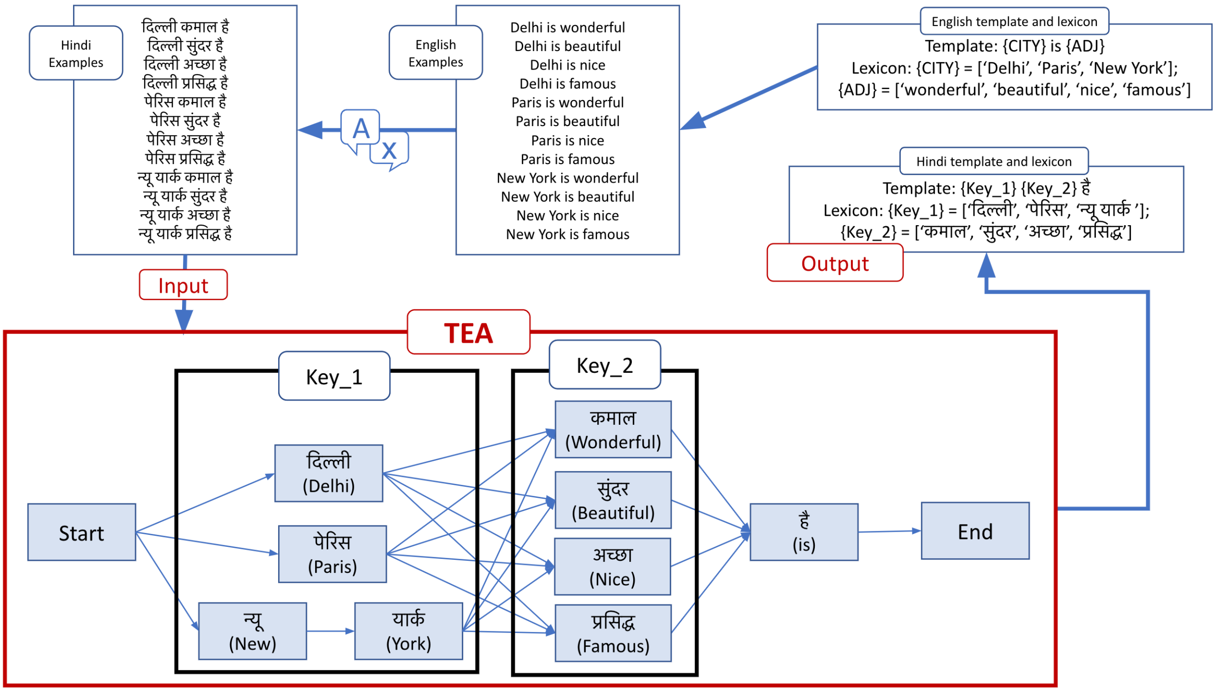

Figure 1: TEA treats sentences as a directed acyclic graph & recursively replaces lexicon values with keys.

combining instances from the translated instances. We assume English (EN) to be the source language and Hindi (HI) to be the target language. The pipeline starts with an EN template, the instances are created by replacing the lexicon values in the templates, that are then machine translated to get HI instances. These instances function as input to TEA which then recursively groups instances using non-terminals to form templates. The entire process of template extraction is repeated for every EN template, resulting in the HI template set. The TEA has 3 steps which we describe as follows (pseudocode and details are in appendix A):

Step 1: Grouping Terminals into Non-Terminals: First, we convert the Hi instances into a directed acyclic graph whose nodes are unique words (or tokens). There is an edge from node A to B if word B follows word A in at least one of the input instances. In this directed graph (see Fig. 1), between any two nodes, if there are multiple paths of length less than equal to $k + 1$ (we set k to 2), we concatenate the intermediate words in the path (with space in between them) and treat them as terminals. This set of terminals, between the two nodes, are grouped together represented by a non-terminal symbol (for example Key_1 and Key_2 in Fig. 1). This step corresponds to lexicon formation; the non-terminal extracted here are essentially keys of the lexicon & the terminals constituting them are the lexicon values for the slots of a template.

Step 2: Template Extraction Using a set of Hi instances, $S = \{s_1, s_2, \ldots s_N\}$, and all non-terminals $v_i = [w_{i1}, w_{i2}, \ldots]$, where w_{ij} are terminals (obtained step 1), TEA outputs a set of templates $\hat{T} = \{t_1, t_2, ...\}$ such that $\hat{T}$ can generate all the examples in S using the given non-terminal and their corresponding terminals. For each sentence s_i, we generate a set of candidate templates, $T_i = \{t_{i1}, t_{i2}, ...\}$, such that s_i belongs to the set of examples generated by each t_{ij}. To find the minimal template set, i.e $\hat{T}$ that covers all examples is treated as a set cover problem and we use a greedy approximation to find this set.

Step 3: Combine Steps 1 and 2 The above template extraction process, while resulting in correct outputs, may be computationally expensive due to translation noise[1] and its time-complexity which is exponential on the number of non-terminals. To mitigate this, we follow an iterative approach where instead of using all the extracted non-terminals (along with their terminals), we initialize the set of non-terminals with an empty set and iteratively add the most useful non-terminals (with their cor-

[1] The translated sentences may not fit into 1 template. Or, the algorithm may produce a set of distinct non-terminals with common or overlapping terminals. For e.g, we may get two non-terminals with their corresponding terminals such as "{Paris, New York}" & "{London, New York, Delhi}".

responding terminals) to this set.

Note that, TEA can generate multiple templates for the set of instances (all of which might be generated from a single source template). This design is intentional and desirable as due to morphosyntactic complexities (e.g, grammatical gender), it is likely that all instances in a target language will not fit into a single template.

3 Multilingual CheckList Generation

We now describe the various ways in which multilingual checklists can be generated, ranging from fully automatic to fully manual approaches.

Using TEA We start with a source language (En) CheckList template and generate instances by replacing lexicon values in templates. These instances are translated using an MT system. The translated instances now serve as the input to the TEA and target language (HI) CheckList template is extracted. The process (Fig. 1) is repeated for all En templates to form the complete HI CheckList.

TEA with Verification (TEA-ver) This is a semi-automatic approach, where we ask a human-*verifier* to verify and correct the CheckLists generated using TEA. The verifiers (or annotators) are provided with a set of templates and lexicons generated using the TEA pipeline, along with the original source langauge CheckList and description of the capabilities. The annotators are instructed to *verify* the target language templates for (grammatical) correctness. They can delete or edit the incorrect templates. They can also add any missing templates that they think are significantly important (cover too many missed instances).

Translating source CheckList (t9n) This is a completely manual approach, but relies on a source language (here, En) CheckList. The annotators are provided with the En templates, lexicons and the descriptions of capabilities. They are tasked to translate the templates and lexicons into the target language. If a source template cannot be translated to a single target template (such as due to divergent grammatical agreement patterns), annotators are instructed to include as many variants as necessary. This approach is same as that used by Ruder et al. (2021) to create multilingual CheckList.

Generating CheckList from scratch (SCR) This is a completely manual approach of creating CheckLists from scratch, not relying on any source CheckList. Here, the CheckList templates are generated in the same manner as generated in by humans in Ribeiro et al. (2020). That is, human annotators are provided with a description of the task and capabilities and are instructed to develop the templates and lexicon, directly in the target language. In our pilot we found that users were better able to understand the capabilities with some examples as opposed to only from the description, so we also provided them with a couple of examples, in English, for each capability.

4 Evaluation Metrics

Comparison of CheckLists is non-trivial. Firstly, CheckLists cannot be evaluated using absolute metrics, comparisons can only be relative (Bhatt et al., 2021b). Further, the question of what constitutes a better CheckList can be answered in multiple ways. For example, if a CheckList A can help discover (and/or fix) more bugs than CheckList B, CheckList A could be more useful. On the other hand, variability of instances may be desirable. If CheckList B generates more diverse instances as compared CheckList A, even though it discovers less bugs, B could be considered better as it allows testing of the system on a broader variety of instances. Finally, in practical scenarios, cost and correctness are both important factors for generating the CheckList.

We thus propose evaluation metrics along 4 dimensions: a) *utility* for discovery and fixing bugs; b) *diversity* in the generated instances; c) *cost* of generating templates. d) *correctness* of templates.

4.1 Utility

Failure Rate (FR) Here, we measure the percentage of instances generated by the CheckList that the model failed on averaged over all the capabilities.[2] The numbers are reported for XLM-R fine-tuned with English task data from standard datasets (SST-2 for SA and mNLI for NLI). Effectively, we measure the FR on zero-shot transfer from English to the target language. For FR, the higher the value the better the CheckList.

Augmentation Utility (Aug) These metrics aims to test the utility of CheckList in fixing failures using data augmentation following Bhatt et al. (2021b). This is done in two ways:

(a) From Scratch (Aug-0): Here, we fine-tune XLM-R directly using CheckList instances.

[2] Unless mentioned otherwise, we report macro-averages across capabilities.

(b) On Fine-tuned model (Aug-CFT): Here, XLM-R is first fine-tuned with English task data (SST-2 for SA and mNLI for NLI) and then further continually fine-tuned using CheckList instances.

In both cases, we first generate all instances using the CheckLists being compared. We retain a maximum of 10k instances per capability for each CheckList. The instances are then randomly split into train and test sets in 70:30 ratio. The training data (of the corresponding CheckList) is used for the augmentation as described above. The test sets, generated from all the CheckLists being compared are combined together to form a common test set and accuracy on this set is reported. Intuitively, this aims to determine the utility of the CheckList's instances for fixing failures using augmentation. For both the Augmentation metrics, higher is better.

4.2 Diversity

Number of templates (#temp) and lexicon values (#lexv) The simplest way to measure the diversity is the number of distinct templates and lexicon values (or terminals). Higher number of templates and lexicon values means more diversity.

Normalized Cross-Template BLEU (CT-BLEU) To measure the diversity between the templates, we measure the BLEU score (or similarity) for every instance generated by a template with the the instances generated by all other templates in the CheckList . Since this score is sensitive to the number of templates in the Checklist, we normalize the score by the number of templates in the set. Lower CT-BLEU is indicative of better CheckList as it indicates more diverse instances from templates.

4.3 Cost

Time per template (TpT) We define the *cost* of creation of these Checklists simply as the human time required. Since different methods or users can create substantially different number of templates per capability, we measure the *mean time taken* (TpT) for creation from scratch (SCR), translation (t9n) and verification (TEA-ver) of a *template* as the measure of the cost. A better CheckList for practical purposes would have lower TpT.

4.4 Correctness

Here, we assume that templates generated with any amount of human intervention (manual or semi-automatic) would always be correct. As a result, we calculate correctness only for TEA templates. We define the correctness of TEA templates with respect to TEA-ver templates. This is because during creation of templates by the TEA-ver process annotators correct or remove templates. Thus, only correct TEA template are left unedited. Therefore, in order to estimate the correctness of the TEA templates, we compute the following two metrics.

Failure Rate Difference (FR-diff) It is possible that the model fails in some cases if the input instance is not well-formed. As a result, the difference between the failure rates induced by TEA-ver templates (which always lead to well-formed instances) and that of TEA templates (which could lead to some ungrammatical instances) will give an estimate to the correctness of TEA templates. As a result, we define this metric as simply the difference between the FR of TEA and TEA-ver.

Precision and Recall (P/R) Since during the TEA-ver process, annotators edit or remove incorrect templates, only the correct templates that were generated by TEA are left as is. Therefore, in order to estimate the correctness of the TEA, we compute the precision and recall of the TEA template set, with respect to TEA-ver template set. We define match when the templates are same and the lexicon values of either one is a subset of the other, implying they will generate similar set of examples.

5 Hindi CheckLists and Results

We start with Hindi (Hi) as the target language, create CheckLists using all 4 methods from §3 and evaluate them using the metrics from §4. Hi has significant syntactic divergence from the source language (here English (En)) and uses a different script. Hi is a mid-resource language with reasonably good publicly available En-Hi MT systems. We argue that if TEA works well in the En-Hi pipeline, it would also work for most other high to mid resource languages with reasonable MT systems and similar or less syntactic divergence from En, which we also substantiate by performing additional multilingual experiments in §6.

5.1 Experiment Design

We create and evaluate Hi CheckLists for 2 tasks, Sentiment Analysis (SA) and Natural Language Inference (NLI). For SA, we choose 5 capabilities namely Vocabulary, Negation, Temporal, Semantic Role Labeling and Relational, and their associated Minimum Functionality Test (MFT) templates from Ribeiro et al. (2020) as our source CheckList. For NLI, we choose co-reference resolution, spatial, conditional, comparative and causal reasoning as capabilities and their associated templates from Tarunesh et al. (2021). We refer readers to Appendix B for details about these capabilities.

Following Ribeiro et al. (2020), we chose 6 software developers as our *annotators*, who are knowledgeable in NLP. All users are native speakers of Hi and have near-native En fluency.[3] We expect developers to be the actual users of the approach, as it is usually a developer's job to find and fix bugs. The annotators were given a detailed description of expectations along with examples (both in En and Hi). Furthermore, during our pilot study, we found some of the common errors users make, and to mitigate those we provided a list of common errors illustrated with simple examples.

Each of the 6 annotators was randomly assigned a CheckList creation approach that requires human intervention. Thus, we had 2 annotators each for the SCR, t9n and TEA-Ver setups. They carry out the process independently for both SA and NLI. The same description of capabilities and examples are used for all the experimental setups. Similarly, the same source templates and lexicons are used for t9n, TEA-ver and TEA. For the TEA pipeline, we used Bing Translator API for translating En instances to Hi. While reporting the results, we report the average metrics of both annotators.

5.2 Results

Table 1 reports the metrics (§4) for the 4 methods.

The trends for *cost* or TpT are consistent with expectations. Creating CheckLists from Scratch (SCR) takes the most time, as the user has to think and create the templates. t9n requires manual translation and is quicker than SCR but slower than TEA-ver, which just requires verification and correction on templates generated by TEA. We do not factor in the time required to create the source En Checklist, because 1) It is common to all of these 4 approaches and sourced from existing literature; and 2) it is a one-time effort which can be reused for generation of CheckLists in many target languages, leading to a very low amortized cost.

In *diversity* metrics TEA generates the most diverse templates, closely followed by TEA-Ver. t9n is much less diverse, and SCR has the least diversity. We found that, the users created very few templates for SCR, perhaps because it is difficult to decide what would be a good number of templates. We also observe that TEA generates a largest number of templates. The source checklists had 32 (74) and 18 (76) templates (lexicon values) for SA and NLI, respectively. Thus on average, a source template generates around 3 target templates, which is primarily due to syntactic divergence between the En and Hi. These numbers are reduced in TEA-ver, most likely because not all of the TEA templates are perfect and human annotators merge or delete some of them during the verification.

The trends in *utility* metrics are varied. In SA, TEA-ver templates induce highest FR and TEA is a close second. However, for NLI, SCR CheckList induces the highest failure, followed by t9n. This might be due to the task complexity. We leave further exploration on the co-relation of task complexity and efficacy of TEA to future work. TEA has the highest Aug-0 and Aug-CFT values except one case where it is a close second, indicating that the instances generated by TEA CheckLists are effective in fixing failure by augmentation. TEA-ver has values that close to TEA for these metrics[4].

In terms of *correctness*, based on P/R of TEA with respect to TEA-ver, we find that that around a third of the TEA templates had to be significantly edited or removed. Despite this, from FR-diff, we see that the FR generated by TEA is fairly close to the FR generated by TEA-ver. Additionally, even the numbers of other utility metrics are also comparable. This indicates that even the unverified templates (from TEA) which may generate some ungrammatical instances, can give very close estimates of the failure rates and augmentation accuracy to human-verified template sets. This is a positive finding, because while TEA-ver is more reliable, but when resources to get TEA templates verified are not available, despite imperfections, TEA CheckLists can be used for evaluation.

Finally, we would like to point out some of the qualitative differences that we saw in the CheckLists created by these different methods which are hard to articulate through metrics. In particular, we saw that CheckLists created from scratch tend to capture cultural context better. For example, annotators use Indian names in the lexicon values as opposed to western names that get generated due to translations in all other 3 approaches. However,

[3] Educated for 15+ years in English

[4] TEA and TEA-ver have a substantial overlap, and thus, augmentation of one typically helps with the other. This explains the high AUG-0 and AUG-CFT values for these setups.

Metric		**Sentiment Analysis**				**NLI**			
		SCR	**t9n**	**TEA-ver**	**TEA**	**SCR**	**t9n**	**TEA-ver**	**TEA**
Utility	**FR**	6.7	16.5	**19.7**	19.3	**60.3**	53.4	45.1	48.4
	Aug-0	49	52.4	50.6	**67.1**	16.2	50.9	**58.4**	52.5
	Aug-CFT	86.8	89.9	**95.3**	**95.3**	70.1	81.2	79.4	**83.2**
Diversity	**# temp**	17	44.5	86.0	**105**	16	22.5	51.5	**54**
	# lexv	35.0	41.5	109	**147.0**	38.5	56.5	88.5	**98.0**
	CT-BLEU*	0.511	0.142	0.096	**0.087**	0.564	0.307	0.216	**0.169**
Cost	**TpT* (mins)**	5.38	2.07	1.77	**0**	4.69	3.67	1.91	**0**
Correctness	**FR-diff***	-	-	-	0.4	-	-	-	3.3
	P/R	-	-	-	0.64/0.61	-	-	-	0.67/0.63

Table 1: Comparison of the 4 approaches across two tasks for Hindi. *Lower is better; for rest higher is better.

while difficult for TEA, this entity recontextualization is fairly easy for the other two approaches where humans are involved. We also find that the template sets of TEA-ver and t9n are overlapping. This is because of the setup, where t9n is directly translated at template level and TEA-ver is obtained after correcting the templates obtained from translated instances. The major difference occurs in the amount of time taken as correcting templates is faster than translating them.

Thus overall, we conclude that TEA followed by human verification, or TEA-ver would be an ideal approach for scaling CheckList evaluation to multiple languages. That said, the fully automatic TEA approach is even more cost-effective and almost equally reliable to the TEA-ver approach, making it suitable for large-scale multilingual CheckList generation with extremely limited resources.

6 CheckLists in Multiple Languages

So far, we see that TEA is cost-efficient in producing effective Hi CheckLists. We now experiment with 9 more typologically diverse languages – Arabic, French, German, Gujarati, Japanese, Russian, Spanish, Swahili and Vietnamese to evaluate the efficacy of scaling TEA to may languages. We use TEA to automatically generate CheckLists across these languages from the same set of source templates in English for SA across 6 capabilities: Vocabulary, Temporal, Fairness, Negation, Semantic role labeling (SRL) and Robustness. We use the same source En CheckList from the Ribeiro et al. (2020) and use Bing Translator in the TEA pipeline to translate En instances to the target language.

In Table 2 we report the FR on XLM-R model fine-tuned with SST-2 data; thus, except for En, all other values are for zero-shot transfer to the respective language. The average FR for AMCG is highest for Swahili (59%), Vietnamese and Gujarati (around 52%), and lowest for French (43%), Spanish and German (around 45%). For English, average FR is 41%. These trends are consistent with expectation of performance as English, French, and other European languages are high-resourced while Swahili and Vietnamese are very low-resourced.

For 3 of the target languages, namely French, Gujarati and Swahili, native speakers verified the generated templates and thus, we also report the FR for TEA-ver.[5] We observe that the Pearson (Spearman) correlation between TEA and TEA-ver FR values for French, Gujarati and Swahili are 0.99 (1.0), 0.98 (0.89) and 0.97 (0.94) respectively. Furthermore, the difference between FR (FR-Diff) is also low. This implies, similar to our observations from section 5, that one can obtain an extremely accurate assessment of the capabilities of multilingual models just from TEA CheckLists even for low resource languages like Swahili. This re-affirms that despite noise, TEA is able to generate CheckLists that are useful without any human supervision.

7 Limitations

In this paper, we introduced the TEA to generate target language CheckList (templates + lexicon) from the translated instances of source language CheckList. We show that with drastically reduced human effort required for creating CheckList in a new language, the TEA CheckLists provide an accurate estimate of the models' capabilities. However, some of the generated templates/lexicons are noisy and were removed or edited by humans through the TEA-ver process. In this section, we summarize the limitations, common error patterns

[5] These languages were selected based on typological, geographical, resource level diversity and access to native speakers.

Language		**Vocabulary**	**Temporal**	**Fairness**	**Negation**	**SRL**	**Robustness**
English	FR (SCR)	24.21	1.8	94.35	48.16	35.94	42.58
Gujarati	FR (TEA)	39.12	34.97	87.46	51.84	47.37	52.09, 51.54
	FR (TEA-ver)	29.09	32.18	88.72	55.15	46.8	51.54
	FR-diff	10.09	2.79	1.26	3.3	0.57	0.55
French	FR (TEA)	20.27	11.22	86.52	56.55	40.09	46.77
	FR (TEA-ver)	21.78	11.53	86.52	61.25	40.09	47.8
	FR-diff	1.51	0.31	0	4.7	0	1.3
Swahili	FR (TEA)	46.04	37.5	88.86	73.32	51.87	58.45
	FR (TEA-ver)	38.53	43.72	90.37	73.25	46.51	55.38
	FR-diff	8.24	6.22	1.51	0.07	5.36	3.07
Arabic	FR (TEA)	46.77	14.37	91.98	52.08	39.4	53.32
German	FR (TEA)	38.45	15.59	85.25	47.56	43.03	44.04
Spanish	FR (TEA)	29.44	3.18	89.45	59.41	41.39	50.1
Russian	FR (TEA)	40.26	5.07	93.67	56.13	40.3	47.61
Vietnamese	FR (TEA)	23.50	21.67	93.22	63.05	53.12	50.97
Japanese	FR (TEA)	26.9	24.22	93.69	50.1	50.97	-

Table 2: Failure rates for 9 more languages across 6 capabilities for sentiment analysis. Failure rates of English are for the original templates created manually by annotators (SCR); For Gujarati, French, and Swahili FR for TEA, TEA-ver and FR-diff is reported, for the rest of languages FR for TEA is reported.

and suggest some possible ways to resolve them.

Agnostic to Semantics TEA is agnostic of the semantics of the lexicon keys. So, when faced with a set of sentences: *Las Vegas is good.*, *New York is good.*, *New Delhi is good.* and *Las Palmas is good.*, it is unclear whether it should design 1 template `CITY is good.` with lexicon `CITY=`{*Las Vegas, New York, New Delhi, Las Palmas*} or 2 templates: `Las CITY1 is good.`, `CITY1=`{*Vegas, Palmas*} and `New CITY2 is good.`, `CITY2=`{*York, Delhi*}. This problem is hard to solve without heuristics. One possibility is to use the translation alignment information however, such alignments are often imperfect even for high-resource languages. We leave improvements to TEA for handling this to future work.

Handling Morphology Creating good templates for morphologically rich languages (Sinha et al., 2005; Dorr, 1994) is more challenging due to inflections. For e.g, in Hindi a verb may take different form for different tenses and gender. While TEA can handle such cases by creating multiple templates, but with still a third of Hi templates needed correcting. We leave morphologically informed CheckList creation to future work.

Translation Errors Translation errors are a frequent pattern, affecting the input target language instances. In some cases, due to the statistical nature of TEA, we are able to naturally filter out such erroneous templates. For e.g, for an En template ’I used to think this {air_noun} was {neg_adj}, {change} now I think it is {pos_adj}’, translated Hi templates ’Mujhe lagta hai ki us {udaan} {ghatia} tha, ab mujhe lagta hai ki yeh asadharan hai’ (correct) matches 187 translations, and ’Mujhe lagta hai ki us {udaan} {ghatia} tha karte the, ab mujhe lagta hai ki yeh bohut achha hai’ (noisy) matches only 35. While TEA can remove some noisy patterns, errors due to misunderstood context are much harder to fix. For e.g ’the service is poor’ translated as ’vah seva garib hai’ but ‘garib’ in Hindi means “lacking sufficient money" and *not* “lower or insufficient standards". We leave comparisons of TEA for human v.s machine translated input instances and methods to measure and reduce the effect of translation errors on TEA to future work.

Metric Limitations Quantifying the quality of generated template and verification of the relevance of templates with respect to provided description is non-trivial . While we suggest a set of metrics quantifying utility, diversity and cost, these should be extended and further studied for efficacy across tasks and languages. Lastly, soundness and completeness of a template sets (or a test-suite in general) is another unexplored aspect in our current work and an important future direction of research. Furthermore, we acknowledge the limitation of Failure Rate as a metric in the sense that the model could also fail if an example is ungrammatical. In other words, FR is conditional to correctness of the CheckList. However, in our experimentation in both Hindi and other languages, we have found that the the difference between the FR of human verified TEA-ver and TEA is typically small (with a few exceptions) across languages. This means that high FR being caused due to ungrammatical instances here is unlikely. Thus, as stated before, the closeness of the FRs of TEA and TEA-ver points to the reliability of the TEA algorithm.

8 Conclusion

In this paper we proposed TEA (**T**emplate **E**xtraction **A**lgorithm) to automatically generate multilingual CheckLists in a target language without any human supervision (§2). This algorithm recursively extracts templates and lexicon from an input set of instances by treating sentences as a directed acyclic graph of words and combining them.

We additionally experimented with 3 other approaches with varying degrees of human intervention, 2 manual and 1 semi-automatic for CheckList generation (§3). For comparing these CheckLists, we introduced metrics along the dimensions of utility, diversity, cost and correctness (§4).

We performed in-depth analysis of all the 4 methods, with varying degree of human interventions, to create CheckLists for Sentiment Analysis and NLI in Hindi (§5). In addition to Hindi, we experimented with 9 more typologically diverse languages to demonstrate the efficacy of TEA along with comparison with human-verified CheckLists in 3 of them (§3). We found that TEA is cost-effective, useful, and diverse in the CheckLists that it generates. While around one-third of the TEA templates required correction by humans, making the semi-automatic approach more reliable, we find that the model performance estimates provided by unverified CheckLists are very close to that of the human-verified (or semi-automatically created) CheckLists and are also significantly correlated to it. We also substantiated the finding of TEA being effective as well as reliable in the other languages.

Our overall recommendation is that TEA followed by human verification is the most reliable and cost-effective way to scale CheckList evaluation to multiple languages. But in case of very limited resources, TEA is still good enough to test system performance. We end with a discussion on the limitations of this work and propose directions that will, hopefully, inspire research in scaling and improving multilingual evaluation using CheckLists. Finally, we note that TEA is general purpose algorithm of template extraction that can be used for other template-based evaluations such as bias evaluation (Webster et al., 2020; Bhatt et al., 2022)

Acknowledgements We thank the annotators and their team who facilitated the user studies in the paper. We are grateful to our colleagues at MSRI for being the pilot users. We thank Kalika Bali, Balakrishnan Santhanam, and Prasenjit Rath for their thoughtful guidance throughout the work. We thank Gauri Kholkar for her engineering assistance.

References

Kabir Ahuja, Sandipan Dandapat, Sunayana Sitaram, and Monojit Choudhury. 2022. Beyond static models and test sets: Benchmarking the potential of pre-trained models across tasks and languages. In *Proceedings of NLP Power! The First Workshop on Efficient Benchmarking in NLP*, pages 64–74, Dublin, Ireland. Association for Computational Linguistics.

Shaily Bhatt, Sunipa Dev, Partha Talukdar, Shachi Dave, and Vinodkumar Prabhakaran. 2022. Re-contextualizing fairness in nlp: The case of india.

Shaily Bhatt, Poonam Goyal, Sandipan Dandapat, Monojit Choudhury, and Sunayana Sitaram. 2021a. On the universality of deep contextual language models.

Shaily Bhatt, Rahul Jain, Sandipan Dandapat, and Sunayana Sitaram. 2021b. 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.

Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Unsupervised cross-lingual representation learning at scale. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8440–8451, Online. Association for Computational Linguistics.

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

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. 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 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Bonnie J. Dorr. 1994. Machine translation divergences: A formal description and proposed solution. *Computational Linguistics*, 20(4):597–633.

Mor Geva, Yoav Goldberg, and Jonathan Berant. 2019. Are we modeling the task or the annotator? an investigation of annotator bias in natural language understanding datasets. 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 1161–1166, Hong Kong, China. Association for Computational Linguistics.

Max Glockner, Vered Shwartz, and Yoav Goldberg. 2018. Breaking NLI systems with sentences that require simple lexical inferences. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 650–655, Melbourne, Australia. Association for Computational Linguistics.

Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh. 2017. Making the V in VQA matter: Elevating the role of image understanding in visual question answering. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6325–6334. IEEE Computer Society.

Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel Bowman, and Noah A. Smith. 2018. Annotation artifacts in natural language inference data. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 107–112, New Orleans, Louisiana. Association for Computational Linguistics.

Junjie Hu, Sebastian Ruder, Aditya Siddhant, Graham Neubi, Orhan Firat, and Melvin Johnson. 2020a. XTREME: A massively multilingual multi-task benchmark for evaluating cross-lingual generalisation. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 4411–4421. PMLR.

Junjie Hu, Sebastian Ruder, Aditya Siddhant, Graham Neubig, Orhan Firat, and Melvin Johnson. 2020b. Xtreme: A massively multilingual multi-task benchmark for evaluating cross-lingual generalization.

Pratik Joshi, Somak Aditya, Aalok Sathe, and Monojit Choudhury. 2020. Taxinli: Taking a ride up the nlu hill.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, Zhiyi Ma, Tristan Thrush, Sebastian Riedel, Zeerak Waseem, Pontus Stenetorp, Robin Jia, Mohit Bansal, Christopher Potts, and Adina Williams. 2021. Dynabench: Rethinking benchmarking in nlp.

Yaobo Liang, Nan Duan, Yeyun Gong, Ning Wu, Fenfei Guo, Weizhen Qi, Ming Gong, Linjun Shou, Daxin Jiang, Guihong Cao, Xiaodong Fan, Ruofei Zhang, Rahul Agrawal, Edward Cui, Sining Wei, Taroon Bharti, Ying Qiao, Jiun-Hung Chen, Winnie Wu, Shuguang Liu, Fan Yang, Daniel Campos, Rangan Majumder, and Ming Zhou. 2020. XGLUE: A new benchmark datasetfor cross-lingual pre-training, understanding and generation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6008–6018, Online. Association for Computational Linguistics.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual denoising pretraining for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742.

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.

Sebastian Ruder, Noah Constant, Jan Botha, Aditya Siddhant, Orhan Firat, Jinlan Fu, Pengfei Liu, Junjie Hu, Graham Neubig, and Melvin Johnson. 2021. Xtreme-r: Towards more challenging and nuanced multilingual evaluation.

K. Sinha, R. Mahesh, and Anil Thakur. 2005. Translation divergence in English-Hindi MT. In *Proceedings of the 10th EAMT Conference: Practical applications of machine translation*, Budapest, Hungary. European Association for Machine Translation.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. 2022. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*.

Ishan Tarunesh, Somak Aditya, and Monojit Choudhury. 2021. Trusting roberta over bert: Insights from checklisting the natural language inference task.

Masatoshi Tsuchiya. 2018. Performance impact caused by hidden bias of training data for recognizing textual entailment. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Ivan Vulić, Simon Baker, Edoardo Maria Ponti, Ulla Petti, Ira Leviant, Kelly Wing, Olga Majewska, Eden Bar, Matt Malone, Thierry Poibeau, Roi Reichart, and Anna Korhonen. 2020. Multi-SimLex: A large-scale evaluation of multilingual and crosslingual lexical semantic similarity. *Computational Linguistics*, 46(4):847–897.

Zihan Wang, Karthikeyan K, Stephen Mayhew, and Dan Roth. 2020. Extending multilingual BERT to low-resource languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2649–2656, Online. Association for Computational Linguistics.

Kellie Webster, Xuezhi Wang, Ian Tenney, Alex Beutel, Emily Pitler, Ellie Pavlick, Jilin Chen, Ed Chi, and Slav Petrov. 2020. Measuring and reducing gendered correlations in pre-trained models.

Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel Weld. 2019. Errudite: Scalable, reproducible, and testable error analysis. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 747–763, Florence, Italy. Association for Computational Linguistics.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2021. mt5: A massively multilingual pre-trained text-to-text transformer.

A Details of TEA

A.1 Template as a Grammar

A template can be considered as a type of grammar to generate sentences. Consider the template T0 introduced below.

> T0: `CITY-0` is beautiful but `CITY-1` is bigger.
> `CITY` = {Delhi, Paris, New York} ,

Here, the keywords (`CITY-0`,`CITY-1`) are the non-terminals and their corresponding lexicons are the terminal symbols. Also, `CITY-1` should be different than `CITY-0`; and hence the non-terminal symbols cannot be replaced independently of each other, establishing the context-sensitive nature of templates. This is a why we need to look beyond probabilistic context free grammar induction to learn the templates.

Convention and Assumptions: We use *terminal* and *non-terminal* to denote *lexicons* and *keywords* respectively. In a template, if the non-terminals are appended with cardinals from 0 to k, then they can *not be replaced with same terminal while generating sentences. Also, if a template contains an instance of a non-terminal with cardinal k, $(k > 0)$ then at least one instance of the same non-terminal with cardinal $k - 1$ should have occurred before its occurrence in the template.*

A.2 TEA Algorithm

We first briefly recap the pipeline of TEA for ease of exposition. We start with an En template and corresponding terminals created by a human expert, and generate a set of examples by substituting the non-terminals with their appropriate terminals. We then translate the examples to Hi using an Automatic Machine Translation system (such as Azure cloud Translator). Then we extract Hi template(s), terminal(s) and non-terminal(s) from the Hi examples. The process of extracting Hi templates are repeated for each of the En templates, providing us a (tentative) CheckList for Hi. Here, we describe in detail the TEA algorithm that extracts Hi templates (along with Hi terminal words) from the Hi examples. First we discuss our approach to extract potential set of terminal words, i.e., we group a set of words (terminals) and give them a symbol/name (non-terminal). Then we extract the templates using the terminals and non-terminals that are extracted in previous step. Towards the end of this section, we briefly discuss the scalability issues and the approximations that we used to make it more scalable.

A.2.1 Extracting and Grouping Terminals

First, we convert the given Hi examples into a directed graph whose nodes are unique words (or tokens, if we use a different tokenizer) from the examples and there is an edge from word A to word B if word B follows word A in at least one of the examples. In this directed graph (as shown in Fig. 1), between any two nodes, if there are multiple paths of length less than equal to $k+1$, we group all those paths and give the group a name or a non-terminal symbol (for example Key_1 and Key_2 in Fig. 1).[6] *By grouping the paths, we meant to concatenate the intermediate words in the path (with space in between them) and then to group the concatenated strings (terminals). This step gives us potential lexicons and keywords (or list of terminals grouped*

[6] We assumed the maximum length of each terminal string to be $k(=2)$ tokens/words

together).

A.2.2 Template Extraction given Terminal and Non-Terminals

Input to our algorithm is (1) a set Hi examples denoted by $S = \{s_1, s_2, \dots s_N\}$, and (2) all terminals (denoted by w) and its corresponding non-terminals (denoted by v) that are extracted in previous step $\forall i, v_i = w_{i1}, w_{i2}, \dots$ In other words, these are the production rules from a non-terminal to (only) terminals. Output of our algorithm is a set of templates $\hat{T} = \{t_1, t_2, ...\}$ such that $\hat{T}$ can generate all the examples in S using only the given non-terminal and their corresponding terminals.

For convenience, we represent non-terminals and its corresponding terminals as a list (or ordered set) of ⟨ terminal, non-terminal ⟩ tuples, the list is denoted by $L = [\langle w_1, v_1\rangle ... \langle w_i, v_i\rangle ...]$. The tuple $\langle w_i, v_i\rangle$ belongs to L if and only if the the terminal w_i belongs to the non-terminal v_i.

The trivial result for $\hat{T}$ is S itself, as S can generate every example (using no terminals). But this is not useful because, the essence of extracting templates from a set of examples is that one should be able to read/write the entire set by reading only a few templates. Therefore, the objective is to find the (approximately) smallest $\hat{T}$ such that it can generate entire S.

We provide the outline of our algorithm in Algorithm 1. Next, we explain the algorithm along with the helper functions that are not elaborated in the pseudocode. For each sentences s_i, we call the function Get-Templates-Per-Example *to generate a set of templates, $T_i = \{t_{i1}, t_{i2}, ...\}$, such that s_i belongs to the set of examples generated by each t_{ij}. Once we have T_i for every s_i, we construct the (approximately) smallest set $\hat{T}$ such that $\forall i, \hat{T} \bigcap T_i \neq \emptyset$. Note that for every sentence $s_i \in S$, there exist atleast one template in $\hat{T}$ that generates s_i. Finding the smallest $\hat{T}$ is a variant of set cover problem, therefore we use greedy approach to find the approximately small $\hat{T}$.*

Generating T_i: *For every terminal string (w_m) that is a substring of example s_i (or intermediate template t_i), we have 2 options to create template, either (1) replace the matched substring (w_m) with its corresponding non-terminal (v_m) or (2) leave as it is; we can make this decision to replace or not, independently for every matched terminals. While replacing, we need to take care of the cardinals for non-terminals and make sure the templates conform to the adopted convention. We use the functions* Replace-Matched-String *and* Rename-Nonterminal-cardinals *to ensure such conformance.*

Algorithm 1 Extract templates given terminals and non-terminals

```
Input: $S = \{s_1, s_2, ....s_N\}$, $L = [\langle w_1, v_1\rangle ... \langle w_i, v_i\rangle ...]$
Output: $\hat{T}$, the approximately smallest set of templates that generates entire $S$
for each $s_i$ in S do
    $T_i \leftarrow$ Get-Templates-Per-Example($s_i, L$)
end for
Find (approximately) smallest $\hat{T}$ such that $\forall T_i, \hat{T} \cap T_i \neq \emptyset$    ▷ Variant of set cover, use greedy approach
return $\hat{T}$
procedure Get-Templates-Per-Example($s_i, L$)
    $T_i \leftarrow \{s_i\}$
    for each $\langle w_m, v_m\rangle$ in $L$ do
        $T_{new} \leftarrow \{\}$
        for each $t_{ij}$ in $T_i$ do
            if $w_m$ is sub-string of $t_{ij}$ then
                $t_{new} \leftarrow$ Replace-Matched-String($t_{ij}, w_m, v_m$)    ▷ Refer §A.2.2
                $t_{new} \leftarrow$ Rename-Nonterminal-cardinals($t_{new}$)    ▷ Refer §A.2.2
                $T_{new} \leftarrow T_{new} \cup t_{new}$
            end if
        end for
        $T_i \leftarrow T_i \cup T_{new}$
    end for
    return $T_i$
end procedure
```

Replace-Matched-String *This function replaces the matched terminal w_m in t_{ij} with its corresponding non-terminal v_m. If there are multiple w_m in t_{ij}, then each w_m will be independently replaced with v_m or left unchanged. For example, consider the initial template and ⟨ terminal, non-terminal ⟩ pair be "#Paris is beautiful. `CITY-0` is cold. Paris is bigger." and ⟨ Paris, `CITY` ⟩ respectively. This will generate 3 templates after replacement. (1) "#`CITY-1` is beautiful. `CITY-0` is cold. Paris is bigger." (2) "#Paris is beautiful. `CITY-0` is cold. `CITY-1` is bigger." (3) "#`CITY-1` is beautiful. `CITY-0` is cold. `CITY-1` is bigger."*

Note that, we do not search if the words in the

s_i is a terminal, rather we search if the terminal is a sub-string of s_i (or t_{ij}). This makes it possible for the terminal to be a sub-word or a multi-word string and still match. Sub-word level match can be quite useful, especially in morphologically rich languages; using only the base word as lexicons it may be possible to match different morphological forms.

RENAME-NONTERMINAL-CARDINALS *This function renames the cardinals to make sure that an instance of a non-terminal with cardinal $k-1$ occurs before the instance of that non-terminal with cardinal $k, (k > 0)$. For example, after re-naming the cardinals, the above three templates become the following three, respectively. (1) "#`CITY-0` is beautiful. `CITY-1` is cold. Paris is bigger." (2) "#Paris is beautiful. `CITY-0` is cold. `CITY-1` is bigger." (3) "#`CITY-0` is beautiful. `CITY-1` is cold. `CITY-0` is bigger."*

A.2.3 Combine both the steps

First, we find all the potential terminals and non-terminals (using § A.2.1) for all Hi examples, and then use them to extract template following the algorithm outlined in § A.2.2. While this simple procedure is possible, it is often computationally expensive; one of the reasons is that due to noise (many of the translated sentences may not fit into a template), the algorithm to extract terminals and non-terminals (§ A.2.1) often gives a lot of different non-terminals that share many common terminals. For example, we may get two non-terminals with their corresponding terminals such as "{Paris, New York, Delhi}" and "{London, New York, Delhi}". Moreover, the complexity of the algorithm in § 1 to extract templates can be increased exponentially with the number of non-terminals. To mitigate this problem, we follow an iterative approach where instead of using all the extracted non-terminals (along with their terminals), we initialize the set of non-terminals with an empty set and iteratively add the most useful non-terminals (with their corresponding terminals) to the existing set of non-terminals.

B Capabilities tested using CheckList

Capabilities are tested using MFTs. MFTs (Minimum Functionality Tests) are tests similar to unit tests in software testing where a specific pointed capability of a model is tested via a template and an expected label(s). The test is said to pass for an instance if the model predicted label matches the expected label(s). Finally, failure rate is recorded as the % of test instances that fails; which can also be inferred as 100-accuracy.

B.1 Sentiment Analysis (SA)

These capabilities, their descriptions, examples and their original template sets used in testing are all sourced from Ribeiro et al. (2020).

Vocabulary *This capability tests whether the model can appropriately handle the impact of words with different parts of speech on the task. In particular, sentences with neutral adjectives are expected to have a neutral prediction and sentences sentiment-laden (positive or negative) adjectives are expected to have the corresponding label. For example, "This is a* private (NEUTRAL_ADJ) *aircraft" should be labelled neutral; and "This is a* great (POSITIVE_ADJ) *aircraft" "This is a* bad (NEGATIVE_ADJ) *aircraft" should be labelled positive and negative respectively.*

Negation *This capability tests that the negation of a positive adjective in the sentence should be labelled as positive or neutral, for example: "This is* not *a* great (POSITIVE_ADJ) *aircraft" should be labelled negative or neutral. Similarly, sentence with negation of negative adjective should be positive our neutral and those with negation of neutral adjectives should remain neutral.*

Semantic Role Labeling (SRL) *SRL aims to test that the model understands the agent, object etc in an instance. That is sentiment of the correct role in the instance is parsed. Here, there are two distinct capabilities MFTs. The first one is to test that the sentiment author sentiment is given more importance than of sentiment of others. For example, "Some people think this aircraft is bad, but I thought it was* great (POSITIVE_ADJ)*" should be labelled as Positive. The second test is related to parsing yes/no questions with the correct sentiment. For example, "Do I think this aircraft is great?* Yes*" should be labelled as positive, whereas if the answer was No, it should be negative.*

Temporal *This capability is used to test whether the model understands the sequence of events correctly. In other words that the most recent sentiment is correctly parsed in labelling. For example, "I used to hate this aircraft, but* now I love it*" should be labelled positive.*

Robustness *There are two tests for robustness: First changing of values within semantically equivalent classes should not change the prediction. For example, "I flew in from* Delhi*" and "I flew in from* New York*" should have the same label as the change here is within the semantically equivalent class of 'CITY'. Secondly, typos (or random character exchange) should not flip labels. For example, "This is a* graet *aircraft" should still remain positive.*

Fairness *Fairness is used to test that prediction should be the same for various adjectives within a protected class. For example, "Mary is a* black (RACE) *woman" and "Mary is a* white (RACE) *woman" should have same sentiment prediction.*

B.2 Natural Language Inference (NLI)

We use the template sets from Tarunesh et al. (2021) which in turn rely on the taxonomy of capabilities from (Joshi et al., 2020) for their selection of capabilities. In examples that follow, P stands for Premise and H for hypothesis.

Co-reference resolution *Test the model for resolving pronouns between the premise and hypothesis correctly. For example, P: Angelique and Ricardo are colleagues. He is a minister and she is a model. H: Angelique is a model. Here H should 'entail' P.*

Spatial reasoning *Tests the model for reasoning using spatial properties. For example, P: Manchester is 67 miles from Pittsburg and 27 miles from Kansas. H: Manchester is nearer to Kansas than Pittsburg. Here H should 'entail' P.*

Causal reasoning *Tests the model for using causation in the premise to infer the hypothesis. For example, P: Katherine taught science to Nancy. H: Nancy learnt science from Katherine. Here H should 'entail' P.*

Conditional reasoning *Tests the model for logically inferring the hypothesis given conditional premise. For example, P: If the baby is fed on time, he does not get cranky. H: The baby gets crancky when he is hungry. Here H should 'entail' P.*

Comparative reasoning *Tests models for reasoning involving comparisons of objects. For example, P: The earth is larger than the moon but smaller than sun. H: The moon is smaller than sun. Here H should 'entail' P.*