TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP

John X. Morris[1], Eli Lifland[1], Jin Yong Yoo[1], Jake Grigsby[1], Di Jin[2], Yanjun Qi[1]
[1] Department of Computer Science, University of Virginia
[2] Computer Science and Artificial Intelligence Laboratory, MIT
{jm8wx, yq2h}@virginia.edu

Abstract

While there has been substantial research using adversarial attacks to analyze NLP models, each attack is implemented in its own code repository. It remains challenging to develop NLP attacks and utilize them to improve model performance. This paper introduces `TextAttack`, a Python framework for adversarial attacks, data augmentation, and adversarial training in NLP. `TextAttack` builds attacks from four components: a goal function, a set of constraints, a transformation, and a search method. `TextAttack`'s modular design enables researchers to easily construct attacks from combinations of novel and existing components. `TextAttack` provides implementations of 16 adversarial attacks from the literature and supports a variety of models and datasets, including BERT and other transformers, and all GLUE tasks. `TextAttack` also includes data augmentation and adversarial training modules for using components of adversarial attacks to improve model accuracy and robustness. `TextAttack` is democratizing NLP: anyone can try data augmentation and adversarial training on any model or dataset, with just a few lines of code. Code and tutorials are available at https://github.com/QData/TextAttack.

1 Introduction

Over the last few years, there has been growing interest in investigating the adversarial robustness of NLP models, including new methods for generating adversarial examples and better approaches to defending against these adversaries (Alzantot et al., 2018; Jin et al., 2019; Kuleshov et al., 2018; Li et al., 2019; Gao et al., 2018; Wang et al., 2019; Ebrahimi et al., 2017; Zang et al., 2020; Pruthi et al., 2019). It is difficult to compare these attacks directly and fairly, since they are often evaluated on different data samples and victim models. Re-implementing previous work as a baseline is often time-consuming and error-prone due to a lack of source code, and precisely replicating results is complicated by small details left out of the publication. These barriers make benchmark comparisons hard to trust and severely hinder the development of this field.

Original
Perfect performance by the actor → **Positive (99%)**

Adversarial
Spotless performance by the actor → **Negative (100%)**

Figure 1: Adversarial example generated using Jin et al. (2019)'s `TextFooler` for a BERT-based sentiment classifier. Swapping out "perfect" with synonym "spotless" completely changes the model's prediction, even though the underlying meaning of the text has not changed.

To encourage the development of the adversarial robustness field, we introduce `TextAttack`, a Python framework for adversarial attacks, data augmentation, and adversarial training in NLP.

To unify adversarial attack methods into one system, we decompose NLP attacks into four components: a goal function, a set of constraints, a transformation, and a search method. The attack attempts to perturb an input text such that the model output fulfills the goal function (i.e., indicating whether the attack is successful) and the perturbation adheres to the set of constraints (e.g., grammar constraint, semantic similarity constraint). A search method is used to find a sequence of transformations that produce a successful adversarial example.

This modular design enables us to easily assemble attacks from the literature while reusing components that are shared across attacks. `TextAttack` provides clean, readable implementations of 16 adversarial attacks from the literature. For the first time, these attacks can be benchmarked, compared, and analyzed in a standardized setting.

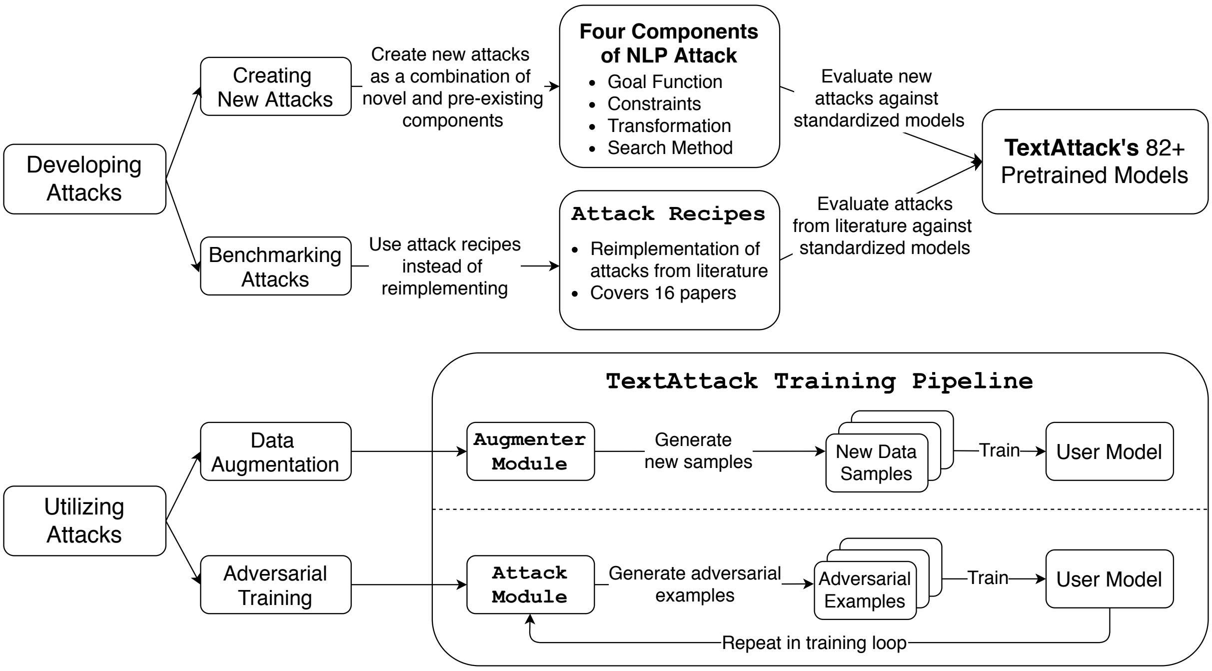

Figure 2: Main features of `TextAttack`.

`TextAttack`'s design also allows researchers to easily construct new attacks from combinations of novel and existing components. In just a few lines of code, the same search method, transformation and constraints used in Jin et al. (2019)'s `TextFooler` can be modified to attack a translation model with the goal of changing every word in the output.

`TextAttack` is directly integrated with HuggingFace's transformers and nlp libraries. This allows users to test attacks on models and datasets. `TextAttack` provides dozens of pre-trained models (LSTM, CNN, and various transformer-based models) on a variety of popular datasets. Currently `TextAttack` supports a multitude of tasks including summarization, machine translation, and all nine tasks from the GLUE benchmark. `TextAttack` also allows users to provide their own models and datasets.

Ultimately, the goal of studying adversarial attacks is to improve model performance and robustness. To that end, `TextAttack` provides easy-to-use tools for data augmentation and adversarial training. `TextAttack`'s `Augmenter` class uses a transformation and a set of constraints to produce new samples for data augmentation. Attack recipes are re-used in a training loop that allows models to train on adversarial examples. These tools make it easier to train accurate and robust models.

Uses for `TextAttack` include[1]:

[1]All can be done in < 5 lines of code. See A.1.

- Benchmarking and comparing NLP attacks from previous works on standardized models & datasets.
- Fast development of NLP attack methods by re-using abundant available modules.
- Performing ablation studies on individual components of proposed attacks and data augmentation methods.
- Training a model (CNN, LSTM, BERT, RoBERTa, etc.) on an augmented dataset.
- Adversarial training with attacks from the literature to improve a model's robustness.

2 The `TextAttack` Framework

`TextAttack` aims to implement attacks which, given an NLP model, find a perturbation of an input sequence that satisfies the attack's goal and adheres to certain linguistic constraints. In this way, attacking an NLP model can be framed as a combinatorial search problem. The attacker must search within all potential transformations to find a sequence of transformations that generate a successful adversarial example.

Each attack can be constructed from four components:

1. A task-specific **goal function** that determines whether the attack is successful in terms of the model outputs.
   Examples: untargeted classification, targeted classification, non-overlapping output, minimum BLEU score.

2. A set of **constraints** that determine if a perturbation is valid with respect to the original input.
   Examples: maximum word embedding distance, part-of-speech consistency, grammar checker, minimum sentence encoding cosine similarity.
3. A **transformation** that, given an input, generates a set of potential perturbations.
   Examples: word embedding word swap, thesaurus word swap, homoglyph character substitution.
4. A **search method** that successively queries the model and selects promising perturbations from a set of transformations.
   Examples: greedy with word importance ranking, beam search, genetic algorithm.

See A.2 for a full explanation of each goal function, constraint, transformation, and search method that's built-in to `TextAttack`.

3 Developing NLP Attacks with `TextAttack`

`TextAttack` is available as a Python package installed from PyPI, or via direct download from GitHub. `TextAttack` is also available for use through our demo web app, displayed in Figure 3.

Python users can test attacks by creating and manipulating `Attack` objects. The command-line API offers `textattack attack`, which allows users to specify attacks from their four components or from a single attack recipe and test them on different models and datasets.

`TextAttack` supports several different output formats for attack results:

- Printing results to stdout.
- Printing to a text file or CSV.
- Printing attack results to an HTML table.
- Writing a table of attack results to a visualization server, like Visdom or Weights & Biases.

3.1 Benchmarking Existing Attacks with Attack Recipes

`TextAttack`'s modular design allows us to implement many different attacks from past work in a shared framework, often by adding only one or two new components. Table 1 categorizes 16 attacks based on their goal functions, constraints, transformations and search methods.

All of these attacks are implemented as "attack recipes" in `TextAttack` and can be benchmarked with just a single command. See A.3 for a comparison between papers' reported attack results and the results achieved by running `TextAttack`.

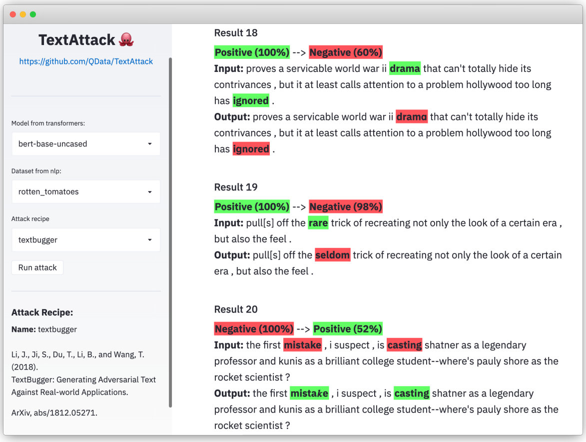

Figure 3: Screenshot of `TextAttack`'s web interface running the TextBugger black-box attack (Li et al., 2019).

3.2 Creating New Attacks by Combining Novel and Existing Components

As is clear from Table 1, many components are shared between NLP attacks. New attacks often reuse components from past work, adding one or two novel pieces. `TextAttack` allows researchers to focus on the generation of new components rather than replicating past results. For example, Jin et al. (2019) introduced `TextFooler` as a method for attacking classification and entailment models. If a researcher wished to experiment with applying `TextFooler`'s search method, transformations, and constraints to attack translation models, all they need is to implement a translation goal function in `TextAttack`. They would then be able to plug in this goal function to create a novel attack that could be used to analyze translation models.

3.3 Evaluating Attacks on `TextAttack`'s Pre-Trained Models

As of the date of this submission, `TextAttack` provides users with 82 pre-trained models, including word-level LSTM, word-level CNN, BERT, and other transformer based models pre-trained on various datasets provided by HuggingFace nlp. Since `TextAttack` is integrated with the nlp library, it can automatically load the test or validation data set for the corresponding pre-trained model. While the literature has mainly focused on classification and entailment, `TextAttack`'s pretrained models enable research on the robustness of models across all GLUE tasks.

Attack Recipe	Goal Function	Constraints	Transformation	Search Method
bae (Garg and Ramakrishnan, 2020)	Untargeted Classification	USE sentence encoding cosine similarity	BERT Masked Token Prediction	Greedy-WIR
bert-attack (Li et al., 2020)	Untargeted Classification	USE sentence encoding cosine similarity, Maximum number of words perturbed	BERT Masked Token Prediction (with subword expansion)	Greedy-WIR
deepwordbug (Gao et al., 2018)	{Untargeted, Targeted} Classification	Levenshtein edit distance	{Character Insertion, Character Deletion, Neighboring Character Swap, Character Substitution}*	Greedy-WIR
alzantot, fast-alzantot (Alzantot et al., 2018; Jia et al., 2019)	Untargeted {Classification, Entailment}	Percentage of words perturbed, Language Model perplexity, Word embedding distance	Counter-fitted word embedding swap	Genetic Algorithm
iga (Wang et al., 2019)	Untargeted {Classification, Entailment}	Percentage of words perturbed, Word embedding distance	Counter-fitted word embedding swap	Genetic Algorithm
input-reduction (Feng et al., 2018)	Input Reduction		Word deletion	Greedy-WIR
kuleshov (Kuleshov et al., 2018)	Untargeted Classification	Thought vector encoding cosine similarity, Language model similarity probability	Counter-fitted word embedding swap	Greedy word swap
hotflip (word swap) (Ebrahimi et al., 2017)	Untargeted Classification	Word Embedding Cosine Similarity, Part-of-speech match, Number of words perturbed	Gradient-Based Word Swap	Beam search
morpheus (Tan et al., 2020)	Minimum BLEU Score		Inflection Word Swap	Greedy search
pruthi (Pruthi et al., 2019)	Untargeted Classification	Minimum word length, Maximum number of words perturbed	{Neighboring Character Swap, Character Deletion, Character Insertion, Keyboard-Based Character Swap}*	Greedy search
pso (Zang et al., 2020)	Untargeted Classification		HowNet Word Swap	Particle Swarm Optimization
pwws (Ren et al., 2019)	Untargeted Classification		WordNet-based synonym swap	Greedy-WIR (saliency)
seq2sick (black-box) (Cheng et al., 2018)	Non-overlapping output		Counter-fitted word embedding swap	Greedy-WIR
textbugger (black-box) (Li et al., 2019)	Untargeted Classification	USE sentence encoding cosine similarity	{Character Insertion, Character Deletion, Neighboring Character Swap, Character Substitution}*	Greedy-WIR
textfooler (Jin et al., 2019)	Untargeted {Classification, Entailment}	Word Embedding Distance, Part-of-speech match, USE sentence encoding cosine similarity	Counter-fitted word embedding swap	Greedy-WIR

Table 1: `TextAttack` attack recipes categorized within our framework: search method, transformation, goal function, constraints. All attack recipes include an additional constraint which disallows the replacement of stopwords. Greedy search with Word Importance Ranking is abbreviated as Greedy-WIR.

* *indicates a combination of multiple transformations*

4 Utilizing `TextAttack` to Improve NLP Models

4.1 Evaluating Robustness of Custom Models

`TextAttack` is model-agnostic - meaning it can run attacks on models implemented in any deep learning framework. Model objects must be able to take a string (or list of strings) and return an output that can be processed by the goal function. For example, machine translation models take a list of strings as input and produce a list of strings as output. Classification and entailment models return an array of scores. As long as the user's model meets this specification, the model is fit to use with `TextAttack`.

4.2 Model Training

`TextAttack` users can train standard LSTM, CNN, and transformer based models, or a user-customized model on any dataset from the `nlp` library using the `textattack train` command. Just like pre-trained models, user-trained models are compatible with commands like `textattack attack` and `textattack eval`.

4.3 Data Augmentation

While searching for adversarial examples, `TextAttack`'s transformations generate perturbations of the input text, and apply constraints to verify their validity. These tools can be reused to dramatically expand the training dataset by introducing perturbed versions of existing samples. The `textattack augment` command gives users access to a number of pre-packaged recipes for augmenting their dataset. This is a stand-alone feature that can be used with any model or training framework. When using `TextAttack`'s models and training pipeline, `textattack train --augment` automatically expands the dataset before training begins. Users can specify the fraction of each input that should be modified and how many additional versions of each example to create. This makes it easy to use existing augmentation recipes on different models and datasets, and is a great way to benchmark new techniques.

Figure 4 shows empirical results we obtained using `TextAttack`'s augmentation. Augmentation with `TextAttack` immediately improves the performance of a `WordCNN` model on small datasets.

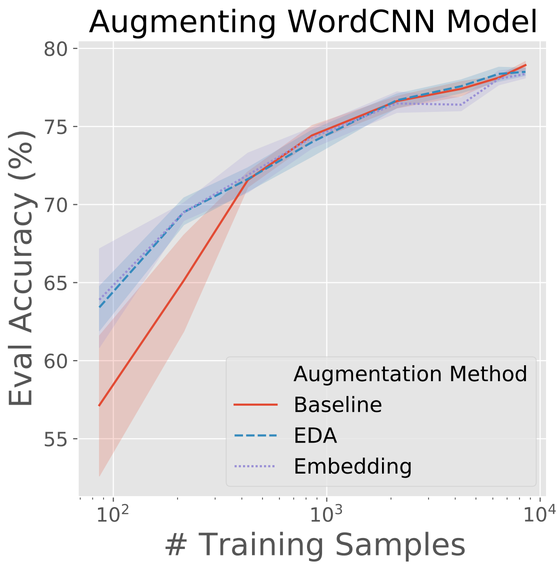

Figure 4: Performance of the built-in `WordCNN` model on the `rotten_tomatoes` dataset with increasing training set size. Data augmentation recipes like `EasyDataAugmenter` (EDA, (Wei and Zou, 2019)) and `Embedding` are most helpful when working with very few samples. Shaded regions represent 95% confidence intervals over $N = 5$ runs.

4.4 Adversarial Training

With `textattack train --attack`, attack recipes can be used to create new training sets of adversarial examples. After training for a number of epochs on the clean training set, the attack generates an adversarial version of each input. This perturbed version of the dataset is substituted for the original, and is periodically regenerated according to the model's current weaknesses. The resulting model can be significantly more robust against the attack used during training. Table 2 shows the accuracy of a standard LSTM classifier with and without adversarial training against different attack recipes implemented in `TextAttack`.

5 `TextAttack` Under the Hood

`TextAttack` is optimized under-the-hood to make implementing and running adversarial attacks simple and fast.

AttackedText. A common problem with implementations of NLP attacks is that the original text is discarded after tokenization; thus, the transformation is performed on the tokenized version of the text. This causes issues with capitalization and word segmentation. Sometimes attacks swap a piece of a word for a complete word (for example, transforming `''aren't"` into `''aren'too"`).

To solve this problem, `TextAttack` stores each input as a `AttackedText` object which contains the original text and helper methods for transforming the text while retaining tokenization. Instead of strings or tensors,

	Attacked By					
Trained Against	-	`deepwordbug`	`textfooler`	`pruthi`	`hotflip`	`bae`
baseline (early stopping)	**77.30%**	23.46%	2.23%	59.01%	64.57%	25.51%
`deepwordbug` (20 epochs)	76.38%	**35.07%**	4.78%	57.08%	65.06%	27.63%
`deepwordbug` (75 epochs)	73.16%	**44.74%**	13.42%	58.28%	66.87%	32.77%
`textfooler` (20 epochs)	61.85%	40.09%	**29.63%**	52.60%	55.75%	39.36%

Table 2: The default LSTM model trained on 3k samples from the `sst2` dataset. The baseline uses early stopping on a clean training set. `deepwordbug` and `textfooler` attacks are used for adversarial training. 'Accuracy Under Attack' on the eval set is reported for several different attack types.

classes in `TextAttack` operate primarily on `AttackedText` objects. When words are added, swapped, or deleted, an `AttackedText` can maintain proper punctuation and capitalization. The `AttackedText` also contains implementations for common linguistic functions like splitting text into words, splitting text into sentences, and part-of-speech tagging.

Caching. Search methods frequently encounter the same input at different points in the search. In these cases, it is wise to pre-store values to avoid unnecessary computation. For each input examined during the attack, `TextAttack` caches its model output, as well as the whether or not it passed all of the constraints. For some search methods, this memoization can save a significant amount of time.[2]

6 Related Work

We draw inspiration from the `Transformers` library (Wolf et al., 2019) as an example of a well-designed Natural Language Processing library. Some of `TextAttack`'s models and tokenizers are implemented using `Transformers`.

`cleverhans` (Papernot et al., 2018) is a library for constructing adversarial examples for computer vision models. Like `cleverhans`, we aim to provide methods that generate adversarial examples across a variety of models and datasets. In some sense, `TextAttack` strives to be a solution like `cleverhans` for the NLP community. Like `cleverhans`, attacks in `TextAttack` all implement a base `Attack` class. However, while `cleverhans` implements many disparate attacks in separate modules, `TextAttack` builds attacks from a library of shared components.

There are some existing open-source libraries related to adversarial examples in NLP. `Trickster` proposes a method for attacking NLP models based on graph search, but lacks the ability to ensure that generated examples satisfy a given constraint (Kulynych et al., 2018). `TEAPOT` is a library for evaluating adversarial perturbations on text, but only supports the application of ngram-based comparisons for evaluating attacks on machine translation models (Michel et al., 2019). Most recently, `AllenNLP Interpret` includes functionality for running adversarial attacks on NLP models, but is intended only for the purpose of interpretability, and only supports attacks via input-reduction or greedy gradient-based word swap (Wallace et al., 2019). `TextAttack` has a broader scope than any of these libraries: it is designed to be extendable to any NLP attack.

7 Conclusion

We presented `TextAttack`, an open-source framework for testing the robustness of NLP models. `TextAttack` defines an attack in four modules: a goal function, a list of constraints, a transformation, and a search method. This allows us to compose attacks from previous work from these modules and compare them in a shared environment. These attacks can be reused for data augmentation and adversarial training. As new attacks are developed, we will add their components to `TextAttack`. We hope `TextAttack` helps lower the barrier to entry for research into robustness and data augmentation in NLP.

8 Acknowledgements

The authors would like to thank everyone who has contributed to make `TextAttack` a reality: Hanyu Liu, Kevin Ivey, Bill Zhang, and Alan Zheng, to name a few. Thanks to the IGA creators (Wang et al., 2019) for contributing an implementation of their algorithm to our framework. Thanks to the folks at HuggingFace for creating such easy-to-use software; without them, `TextAttack` would not be what it is today.

[2] Caching alone speeds up the genetic algorithm of Alzantot et al. (2018) by a factor of 5.

References

Abhaya Agarwal and Alon Lavie. 2008. Meteor, m-bleu and m-ter: Evaluation metrics for high-correlation with human rankings of machine translation output. In *WMT@ACL*.

Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani B. Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. *ArXiv*, abs/1804.07998.

Daniel Matthew Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Universal sentence encoder. *ArXiv*, abs/1803.11175.

Minhao Cheng, Jinfeng Yi, Pin-Yu Chen, Huan Zhang, and Cho-Jui Hsieh. 2018. Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples.

Alexis Conneau, Douwe Kiela, Holger Schwenk, Loïc Barrault, and Antoine Bordes. 2017. Supervised learning of universal sentence representations from natural language inference data. In *EMNLP*.

Zhendong Dong, Qiang Dong, and Changling Hao. 2006. Hownet and the computation of meaning.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2017. Hotflip: White-box adversarial examples for text classification. In *ACL*.

Shi Feng, Eric Wallace, Alvin Grissom II, Mohit Iyyer, Pedro Rodriguez, and Jordan Boyd-Graber. 2018. Pathologies of neural models make interpretations difficult.

Ji Gao, Jack Lanchantin, Mary Lou Soffa, and Yanjun Qi. 2018. Black-box generation of adversarial text sequences to evade deep learning classifiers. *2018 IEEE Security and Privacy Workshops (SPW)*, pages 50–56.

Siddhant Garg and Goutham Ramakrishnan. 2020. Bae: Bert-based adversarial examples for text classification.

Ari Holtzman, Jan Buys, Maxwell Forbes, Antoine Bosselut, David Golub, and Yejin Choi. 2018. Learning to write with cooperative discriminators. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1638–1649, Melbourne, Australia. Association for Computational Linguistics.

Robin Jia and Percy Liang. 2017. Adversarial examples for evaluating reading comprehension systems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2021–2031, Copenhagen, Denmark. Association for Computational Linguistics.

Robin Jia, Aditi Raghunathan, Kerem Göksel, and Percy Liang. 2019. Certified robustness to adversarial word substitutions. In *EMNLP/IJCNLP*.

Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2019. Is bert really robust? natural language attack on text classification and entailment. *ArXiv*, abs/1907.11932.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *ArXiv*, abs/1602.02410.

James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE.

Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S. Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. *ArXiv*, abs/1506.06726.

Volodymyr Kuleshov, Shantanu Thakoor, Tingfung Lau, and Stefano Ermon. 2018. Adversarial examples for natural language classification problems.

Bogdan Kulynych, Jamie Hayes, Nikita Samarin, and Carmela Troncoso. 2018. Evading classifiers in discrete domains with provable optimality guarantees. *CoRR*, abs/1810.10939.

Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. 2019. Textbugger: Generating adversarial text against real-world applications. *ArXiv*, abs/1812.05271.

Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. Bert-attack: Adversarial attack against bert using bert.

Paul Michel, Xian Li, Graham Neubig, and Juan Miguel Pino. 2019. On evaluation of adversarial perturbations for sequence-to-sequence models. *CoRR*, abs/1903.06620.

George Armitage Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J. Miller. 1990. Introduction to wordnet: An on-line lexical database. *International Journal of Lexicography*, 3:235–244.

Nikola Mrkšić, Diarmuid O Séaghdha, Blaise Thomson, Milica Gašić, Lina Rojas-Barahona, Pei-Hao Su, David Vandyke, Tsung-Hsien Wen, and Steve Young. 2016. Counter-fitting word vectors to linguistic constraints. *arXiv preprint arXiv:1603.00892*.

Daniel Naber et al. 2003. *A rule-based style and grammar checker*. Citeseer.

Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. 2018. Technical report on the cleverhans v2.1.0 adversarial examples library. *arXiv preprint arXiv:1610.00768*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2001. Bleu: a method for automatic evaluation of machine translation. In *ACL*.

Maja Popovic. 2015. chrf: character n-gram f-score for automatic mt evaluation. In *WMT@EMNLP*.

Danish Pruthi, Bhuwan Dhingra, and Zachary C. Lipton. 2019. Combating adversarial misspellings with robust word recognition.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating natural language adversarial examples through probability weighted word saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1085–1097, Florence, Italy. Association for Computational Linguistics.

Samson Tan, Shafiq Joty, Min-Yen Kan, and Richard Socher. 2020. It's morphin' time! Combating linguistic discrimination with inflectional perturbations. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2920–2935, Online. Association for Computational Linguistics.

Eric Wallace, Jens Tuyls, Junlin Wang, Sanjay Subramanian, Matthew Gardner, and Sameer Singh. 2019. Allennlp interpret: A framework for explaining predictions of nlp models. *ArXiv*, abs/1909.09251.

Xiaosen Wang, Hao Jin, and Kun He. 2019. Natural language adversarial attacks and defenses in word level.

Jason W. Wei and Kai Zou. 2019. EDA: easy data augmentation techniques for boosting performance on text classification tasks. *CoRR*, abs/1901.11196.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Transformers: State-of-the-art natural language processing.

Yuan Zang, Fanchao Qi, Chenghao Yang, Zhiyuan Liu, Meng Zhang, Qun Liu, and Maosong Sun. 2020. Word-level textual adversarial attacking as combinatorial optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6066–6080, Online. Association for Computational Linguistics.

Tianyi Zhang*, Varsha Kishore*, Felix Wu*, Kilian Q. Weinberger, and Yoav Artzi. 2020. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

A Appendix

A.1 TextAttack in Five Lines or Less

Table 3 provides some examples of tasks that can be accomplished in bash or Python with five lines of code or fewer. Note that every action has to be prefaced with a single line of code (`pip install textattack`).

A.2 Components of TextAttack

This section explains each of the four components of the `TextAttack` framework and describes the components that are currently implemented. Figure 5 shows the decomposition of two popular attacks (Alzantot et al., 2018; Jin et al., 2019).

A.2.1 Goal Functions

A goal function takes an input x' and determines if it is satisfies the conditions for a successful attack in respect to the original input x. Goal functions vary by task. For example, for a classification task, a successful adversarial attack could be changing the model's output to be a certain label. Goal functions also scores how "good" the given x' is for achieving the desired goal, and this score can be used by the search method as a heuristic for finding the optimal solution.

`TextAttack` includes the following goal functions:

- Untargeted Classification: Minimize the score of the correct classification label.
- Targeted Classification: Maximize the score of a chosen incorrect classification label.
- Input Reduction (Classification): Reduce the input text to as few wordas as possible while maintaining the same predicted label.
- Non-Overlapping Output (Text-to-Text): Change the output text such that no words in it overlap with the original output text.
- Minimzing BLEU Score (Text-to-Text): Change the output text such that the BLEU score between it and the original output text is minimized (Papineni et al., 2001).

A.2.2 Constraints

A perturbed text is only considered valid if it satisfies each of the attack's constraints. `TextAttack` contains four classes of constraints.

Pre-transformation Constraints These constraints are used to preemptively limit how `x` can be perturbed and are applied before `x` is perturbed.

- Stopword Modification: stopwords cannot be perturbed.
- Repeat Modification: words that have been already perturbed cannot be perturbed again.
- Minimum Word Length: words less than a certain length cannot be perturbed.
- Max Word Index Modification: words past a certain index cannot be perturbed.
- Input Column Modification: for tasks such as textual entailment where input might be composed of two parts (e.g. hypothesis and premise), we can limit which part we can transform (e.g. hypothesis).

Overlap We measure the overlap between `x` and `x_adv` using the following metrics on the character level and require it to be lower than a certain threshold as a constraint:

- Maximum BLEU score difference (Papineni et al., 2001)
- Maximum chrF score difference (Popovic, 2015)
- Maximum METEOR score difference (Agarwal and Lavie, 2008)
- Maximum Levenshtein edit distance
- Maximum percentage of words changed

Grammaticality These constraints are typically intended to prevent the attack from creating perturbations which introduce grammatical errors. `TextAttack` currently supports the following constraints on grammaticality:

- Maximum number of grammatical errors induced, as measured by LanguageTool (Naber et al., 2003)
- Part-of-speech consistency: the replacement word should have the same part-of-speech as the original word. Supports taggers provided by flair, SpaCy, and NLTK.
- Filtering out words that do not fit within the context based on the following language models:
  - Google 1-billion words language model (Józefowicz et al., 2016)
  - Learning To Write Language Model (Holtzman et al., 2018) (as used by (Jia et al., 2019))
  - GPT-2 language model (Radford et al., 2019)

Semantics Some constraints attempt to preserve semantics between `x` and `x_adv`. `TextAttack` currently provides the following built-in semantic constraints:

- Maximum swapped word embedding distance (or minimum cosine similarity)
- Minimum cosine similarity score of sentence representations obtained by well-trained sentence

	Task	Command
Run an attack	TextFooler on an LSTM trained on the MR sentiment classification dataset	`textattack attack --recipe textfooler --model bert-base-uncased-mr --num-examples 100`
	TextFooler against BERT fine-tuned on SST-2	`textattack attack --model bert-base-uncased-sst2 --recipe textfooler --num-examples 10`
	DeepWordBug on DistilBERT trained on the Quora Question Pairs paraphrase identification dataset:	`textattack attack --model distilbert-base-uncased-qqp --recipe deepwordbug --num-examples 100`
	seq2sick (black-box) against T5 fine-tuned for English-German translation:	`textattack attack --model t5-en-de --recipe seq2sick --num-examples 100`
	Beam search with beam width 4 and word embedding transformation and untargeted goal function on an LSTM:	`textattack attack --model lstm-mr --num-examples 20 --search-method beam-search:beam_width=4 --transformation word-swap-embedding --constraints repeat stopword max-words-perturbed:max_num_words=2 embedding:min_cos_sim=0.8 part-of-speech --goal-function untargeted-classification`
Data augmentation	Augment dataset from 'examples.csv' using the EmbeddingAugmenter, swapping out 4% of words, with 2 augmentations for example, withholding the original samples from the output CSV	`textattack augment --csv examples.csv --input-column text --recipe embedding --pct-words-to-swap 4 --transformations-per-example 2 --exclude-original`

	Augment a list of strings in Python	`from textattack.augmentation import EmbeddingAugmenter`
		`augmenter = EmbeddingAugmenter()`
		`s = 'What I cannot create, I do not understand.'`
		`augmenter.augment(s)`

Train a model	Train the default LSTM for 50 epochs on the Yelp Polarity dataset	`textattack train --model lstm --dataset yelp_polarity --batch-size 64 --epochs 50 --learning-rate 1e-5`
	Fine-tune bert-base on the CoLA dataset for 5 epochs	`textattack train --model bert-base-uncased --dataset glue:cola --batch-size 32 --epochs 5`
	Fine-tune RoBERTa on the Rotten Tomatoes Movie Review dataset, first augmenting each example with 4 augmentations produced by the EasyDataAugmentation augmenter	`textattack train --model roberta-base --batch-size 64 --epochs 50 --learning-rate 1e-5 --dataset rotten_tomatoes --augment eda --pct-words-to-swap .1 --transformations-per-example 4`
	Adversarially fine-tune DistilBERT on AG News using the HotFlip word-based attack, first training for 2 epochs on the original dataset	`textattack train --model distilbert-base-cased --dataset ag_news --attack hotflip --num-clean-epochs 2`

Table 3: With `TextAttack`, adversarial attacks, data augmentation, and adversarial training can be achieved in just a few lines of Bash or Python.

encoders:
- Skip-Thought Vectors (Kiros et al., 2015)
- Universal Sentence Encoder (Cer et al., 2018)
- InferSent (Conneau et al., 2017)
- BERT trained for semantic similarity (Reimers and Gurevych, 2019)

- Minimum BERTScore (Zhang* et al., 2020)

A.2.3 Transformations

A transformation takes an input and returns a set of potential perturbations. The transformation is agnostic of goal function and constraint(s): it returns all potential transformations.

We categorize transformations into two kinds: white-box and black-box. *White-box transforma-*

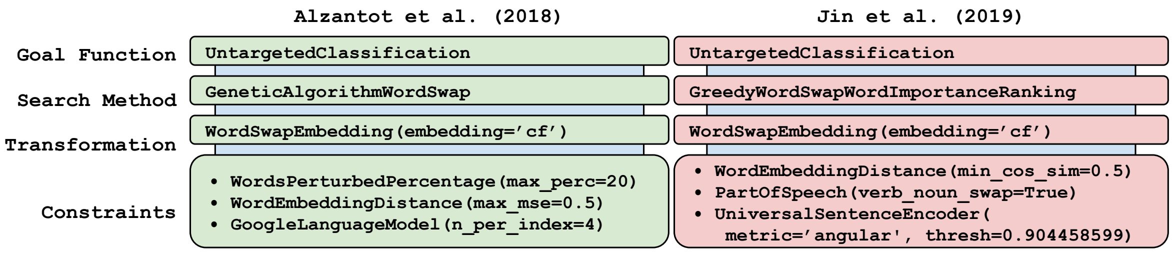

Figure 5: `TextAttack` builds NLP attacks from a goal function, search method, transformation, and list of constraints. This shows attacks from Alzantot et al. (2018) and Jin et al. (2019) created using `TextAttack` modules.

tions have access to the model and can query it or examine its parameters to help determine the transformation. For example, Ebrahimi et al. (2017) determines potential replacement words based on the gradient of the one-hot input vector at the position of the swap. *Black-box transformations* determine the potential perturbations without any knowledge of the model.

`TextAttack` currently supports the following transformations:

- Word swap with nearest neighbors in the counter-fitted embedding space (Mrkšić et al., 2016)
- WordNet word swap (Miller et al., 1990)
- Word swap proposed by a masked language model (Garg and Ramakrishnan, 2020; Li et al., 2020)
- Word swap gradient-based: swap word with another word in the vocabulary that maximize the model's loss (Ebrahimi et al., 2017) (white-box)
- Word swap with characters transformed (Gao et al., 2018):
  - Character deleted
  - Neighboring characters swapped
  - Random character inserted
  - Substituted with a random character
  - Character substituted with a homoglyph
  - Character substituted with a neighboring character from the keyboard (Pruthi et al., 2019)
- Word deletion
- Word swap with another word in the vocabulary that has the same Part-of-Speech and sememe, where the sememe is obtained by HowNet (Dong et al., 2006).
- Composite transformation: returns the results of multiple transformations

A.2.4 Search Methods

The search method aims to find a perturbation that achieves the goal and satisfies all constraints. Many combinatorial search methods have been proposed for this process. `TextAttack` has implemented a selection of the most popular ones from the literature:

- **Greedy Search with Word Importance Ranking.** Rank all words according to some ranking function. Swap words one at a time in order of decreasing importance.

- **Beam Search.** Initially score all possible transformations. Take the top b transformations (where b is a hyperparameter known as the "beam width") and iterate, looking at potential transformations for all sequences in the beam.

- **Greedy Search.** Initially score transformations at all positions in the input. Swap words, taking the highest-scoring transformations first. (This can be seen as a case of beam search where $b = 1$).

- **Genetic Algorithm.** An implementation of the algorithm proposed by Alzantot et al. (2018). Iteratively alters the population through greedy perturbation of each population member and crossover between population numbers, with preference to the more successful members of the population. (We also support an alternate version, the "Improved Genetic Algorithm" proposed by Wang et al. (2019)).

- **Particle Swarm Optimization.** A population-based evolutionary computation paradigms (Kennedy and Eberhart, 1995) that exploits a population of interacting individuals to iteratively search for the optimal solution in the specific space (Zang et al., 2020). The population is called a *swarm* and individual agents are called *particles*. Each particle has a position in the search space and moves with an adaptable *velocity*.

A.3 `TextAttack` Attack Reproduction Results

Table 4 displays a comparison of results achieved when running attacks in `TextAttack` alongside numbers reported in the original paper. All `TextAttack` benchmarks were run on pre-trained models provided by the library and can be reproduced in a single `textattack attack` command. There are a few important implementation differences:

- The genetic algorithm benchmark comes from the faster genetic algorithm of (Jia and Liang, 2017). As opposed to the original algorithm of (Alzantot et al., 2018), this implementation uses a fast language model, so it can query contexts of up to 5 words. Additionally, perplexity is compared to that of the original word, not the previous perturbation. Since these are more rigorous linguistic constraints, a lower attack success rate is expected.
- The LSTM models from BAE (Garg and Ramakrishnan, 2020) were trained using counter-fitted GLoVe embeddings. The LSTM models from `TextAttack` were trained using normal GLoVe embeddings. Our models are consequently less robust to counter-fitted embedding synonym swaps, and a higher attack success rate is expected.
- The HowNet synonym set used in `TextAttack`'s PSO implementation is a concatenation of the three synonym sets used in the paper. This is necessary since `TextAttack` is dataset-agnostic and cannot expect to provide a set of synonyms for every possible dataset. Since the attack has more synonyms to choose from, `TextAttack`'s PSO implementation is slightly more successful.

A.4 `TextAttack` Attack Prototypes

This section displays "attack prototypes" for each attack recipe implemented in `TextAttack`. This is a concise way to print out the components of a given attack along with its parameters. These are directly copied from the output of running `TextAttack`.

Alzantot Genetic Algorithm (Alzantot et al., 2018)

```
Attack(
  (search_method): GeneticAlgorithm(
    (pop_size): 60
    (max_iters): 20
    (temp): 0.3
    (give_up_if_no_improvement): False
  )
  (goal_function): UntargetedClassification
  (transformation): WordSwapEmbedding(
    (max_candidates): 8
    (embedding_type): paragramcf
  )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_percent): 0.2
        (compare_against_original): True
      )
    (1): WordEmbeddingDistance(
        (embedding_type): paragramcf
        (max_mse_dist): 0.5
        (cased): False
        (include_unknown_words): True
        (compare_against_original): False
      )
    (2): GoogleLanguageModel(
        (top_n): None
        (top_n_per_index): 4
        (compare_against_original): False
      )
    (3): RepeatModification
    (4): StopwordModification
    (5): InputColumnModification(
        (matching_column_labels): ['premise', 'hypothesis']
        (columns_to_ignore): {'premise'}
      )
  (is_black_box): True
)
```

Alzantot Genetic Algorithm (faster) (Jia et al., 2019)

```
Attack(
  (search_method): GeneticAlgorithm(
    (pop_size): 60
    (max_iters): 20
    (temp): 0.3
    (give_up_if_no_improvement): False
  )
  (goal_function): UntargetedClassification
  (transformation): WordSwapEmbedding(
    (max_candidates): 8
    (embedding_type): paragramcf
  )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_percent): 0.2
      )
    (1): WordEmbeddingDistance(
        (embedding_type): paragramcf
        (max_mse_dist): 0.5
        (cased): False
        (include_unknown_words): True
      )
    (2): LearningToWriteLanguageModel(
        (max_log_prob_diff): 5.0
      )
    (3): RepeatModification
    (4): StopwordModification
  (is_black_box): True
)
```

BAE (Garg and Ramakrishnan, 2020)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method): delete
  )
  (goal_function): UntargetedClassification
  (transformation): WordSwapMaskedLM(
    (method): bae
    (masked_lm_name): bert-base-uncased
    (max_length): 256
    (max_candidates): 50
  )
  (constraints):
    (0): PartOfSpeech(
```

		LSTM				BERT-Base				
		MR	SST-2	IMDB	AG	MR	SST-2	IMDB	SNLI	AG
alzantot (Alzantot et al., 2018)	Reported	-	-	97.0 / 14.7	-	-	-	-	-	-
	TextAttack	64.6 / 17.8	70.8 / 18.3	73.0 / 4.0	27.7 / 11.6	40.7 / 19.1	46.5 / 20.7	46.7 / 7.3	74.9 / 12.3	18.1 / 12.6
bae (Garg and Ramakrishnan, 2020)	Reported	70.2 / -	-	73.2 / -	-	48.3 / -	-	45.9 / -	-	-
	TextAttack	74.4 / 12.3	72.7 / 13.5	88.8 / 2.6	21.4 / 6.3	61.5 / 15.2	66.6 / 14.5	55.6 / 3.2	78.4 / 7.1	16.9 / 7.4
deepwordbug (Gao et al., 2018)	Reported	-	-	-	72.5 / -	-	-	-	-	-
	TextAttack	86.3 / 16.8	82.6 / 17.1	97.6 / 5.2	83.4 / 19.4	78.2 / 21.2	81.3 / 18.9	80.9 / 5.3	99.0 / 9.8	60.7 / 25.1
pso (Zang et al., 2020)	Reported	-	93.8 / 9.1	100.0 / 3.7	-	-	91.2 / 8.2	98.7 / 3.7	78.9 / 11.7	-
	TextAttack	94.9 / 10.7	96.5 / 11.5	100.0 / 1.3	83.7 / 12.7	92.7 / 11.9	91.3 / 12.9	100.0 / 1.2	91.8 / 6.2	79.4 / 16.7
textfooler (Jin et al., 2019)	Reported	96.2 / 14.9	-	99.7 / 5.1	95.8 / 18.6	86.7 / 16.7	-	85.0 / 6.1	95.5 / 18.5	86.7 / 22.0
	TextAttack	97.4 / 13.6	98.8 / 14.2	100.0 / 2.4	95.3 / 17.2	88.7 / 18.7	94.8 / 16.9	100.0 / 7.2	96.3 / 7.2	79.5 / 23.5

Table 4: Comparison between our re-implemented attacks and the original source code in terms of success rate (left number) and percentage of perturbed words (right number). Numbers that are not found in the literature are marked as "-". 1000 samples are randomly selected for evaluation from all these datasets except IMDB (100 samples are used for IMDB since some attack methods like Genetic and PSO take over 4 days to finish 1000 samples).

```
        (tagger_type): nltk
        (tagset): universal
        (allow_verb_noun_swap): True
        (compare_against_original): True
      )
    (1): UniversalSentenceEncoder(
        (metric): cosine
        (threshold): 0.936338023
        (window_size): 15
        (skip_text_shorter_than_window): True
        (compare_against_original): True
      )
    (2): RepeatModification
    (3): StopwordModification
  (is_black_box): True
)
```

BERT-Attack (Li et al., 2020)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method): unk
  )
  (goal_function): UntargetedClassification
  (transformation): WordSwapMaskedLM(
    (method): bert-attack
    (masked_lm_name): bert-base-uncased
    (max_length): 256
    (max_candidates): 48
  )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_percent): 0.4
        (compare_against_original): True
      )
    (1): UniversalSentenceEncoder(
        (metric): cosine
        (threshold): 0.2
        (window_size): inf
        (skip_text_shorter_than_window): False
        (compare_against_original): True
      )
    (2): RepeatModification
    (3): StopwordModification
  (is_black_box): True
)
```

DeepWordBug (Gao et al., 2018)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method): unk
  )
  (goal_function): UntargetedClassification
  (transformation): CompositeTransformation(
    (0): WordSwapNeighboringCharacterSwap(
        (random_one): True
      )
    (1): WordSwapRandomCharacterSubstitution(
        (random_one): True
      )
    (2): WordSwapRandomCharacterDeletion(
        (random_one): True
      )
    (3): WordSwapRandomCharacterInsertion(
```

```
        (random_one): True
      )
    )
  (constraints):
    (0): LevenshteinEditDistance(
        (max_edit_distance): 30
        (compare_against_original): True
      )
    (1): RepeatModification
    (2): StopwordModification
  (is_black_box): True
)
```

HotFlip (Ebrahimi et al., 2017)

```
Attack(
  (search_method): BeamSearch(
    (beam_width): 10
  )
  (goal_function): UntargetedClassification
  (transformation): WordSwapGradientBased(
    (top_n): 1
  )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_num_words): 2
        (compare_against_original): True
      )
    (1): WordEmbeddingDistance(
        (embedding_type): paragramcf
        (min_cos_sim): 0.8
        (cased): False
        (include_unknown_words): True
        (compare_against_original): True
      )
    (2): PartOfSpeech(
        (tagger_type): nltk
        (tagset): universal
        (allow_verb_noun_swap): True
        (compare_against_original): True
      )
    (3): RepeatModification
    (4): StopwordModification
  (is_black_box): False
)
```

Input Reduction (Feng et al., 2018)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method): delete
  )
  (goal_function): InputReduction(
    (maximizable): True
  )
  (transformation): WordDeletion
  (constraints):
    (0): RepeatModification
    (1): StopwordModification
  (is_black_box): True
)
```

Kuleshov (Kuleshov et al., 2018)

```
Attack(
  (search_method): GreedySearch
  (goal_function):  UntargetedClassification
  (transformation):  WordSwapEmbedding(
    (max_candidates):  15
    (embedding_type):  paragramcf
  )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_percent):  0.5
        (compare_against_original):  True
      )
    (1): ThoughtVector(
        (embedding_type):  paragramcf
        (metric):  max_euclidean
        (threshold):  -0.2
        (window_size):  inf
        (skip_text_shorter_than_window):  False
        (compare_against_original):  True
      )
    (2): GPT2(
        (max_log_prob_diff):  2.0
        (compare_against_original):  True
      )
    (3): RepeatModification
    (4): StopwordModification
  (is_black_box):  True
)
```

MORPHEUS (Tan et al., 2020)

```
Attack(
  (search_method): GreedySearch
  (goal_function):  MinimizeBleu(
    (maximizable):  False
    (target_bleu):  0.0
  )
  (transformation):  WordSwapInflections
  (constraints):
    (0): RepeatModification
    (1): StopwordModification
  (is_black_box):  True
)
```

Particle Swarm Optimization (Zang et al., 2020)

```
Attack(
  (search_method): ParticleSwarmOptimization
  (goal_function):  UntargetedClassification
  (transformation):  WordSwapHowNet(
    (max_candidates):  -1
  )
  (constraints):
    (0): RepeatModification
    (1): StopwordModification
    (2): InputColumnModification(
        (matching_column_labels):  ['premise', 'hypothesis']
        (columns_to_ignore):  {'premise'}
      )
  (is_black_box):  True
)
```

Pruthi Keyboard Char-Swap Attack (Pruthi et al., 2019)

```
Attack(
  (search_method): GreedySearch
  (goal_function):  UntargetedClassification
  (transformation):  CompositeTransformation(
    (0): WordSwapNeighboringCharacterSwap(
        (random_one):  False
      )
    (1): WordSwapRandomCharacterDeletion(
        (random_one):  False
      )
    (2): WordSwapRandomCharacterInsertion(
        (random_one):  False
      )
    (3): WordSwapQWERTY
    )
  (constraints):
    (0): MaxWordsPerturbed(
        (max_num_words):  1
        (compare_against_original):  True
      )
    (1): MinWordLength
    (2): StopwordModification
    (3): RepeatModification
  (is_black_box):  True
)
```

PWWS (Ren et al., 2019)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method):  pwws
  )
  (goal_function):  UntargetedClassification
  (transformation):  WordSwapWordNet
  (constraints):
    (0): RepeatModification
    (1): StopwordModification
  (is_black_box):  True
)
```

seq2sick (Cheng et al., 2018)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method):  unk
  )
  (goal_function):  NonOverlappingOutput
  (transformation):  WordSwapEmbedding(
    (max_candidates):  50
    (embedding_type):  paragramcf
  )
  (constraints):
    (0): LevenshteinEditDistance(
        (max_edit_distance):  30
        (compare_against_original):  True
      )
    (1): RepeatModification
    (2): StopwordModification
  (is_black_box):  True
)
```

TextBugger (Li et al., 2019)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method):  unk
  )
  (goal_function):  UntargetedClassification
  (transformation):  CompositeTransformation(
    (0): WordSwapRandomCharacterInsertion(
        (random_one):  True
      )
    (1): WordSwapRandomCharacterDeletion(
        (random_one):  True
      )
    (2): WordSwapNeighboringCharacterSwap(
        (random_one):  True
      )
    (3): WordSwapHomoglyphSwap
    (4): WordSwapEmbedding(
        (max_candidates):  5
        (embedding_type):  paragramcf
      )
    )
  (constraints):
    (0): UniversalSentenceEncoder(
        (metric):  angular
        (threshold):  0.8
        (window_size):  inf
        (skip_text_shorter_than_window):  False
        (compare_against_original):  True
      )
    (1): RepeatModification
    (2): StopwordModification
  (is_black_box):  True
)
```

TextFooler (Jin et al., 2019)

```
Attack(
  (search_method): GreedyWordSwapWIR(
    (wir_method):  del
  )
  (goal_function):  UntargetedClassification
  (transformation):  WordSwapEmbedding(
    (max_candidates):  50
    (embedding_type):  paragramcf
  )
  (constraints):
    (0): WordEmbeddingDistance(
        (embedding_type):  paragramcf
        (min_cos_sim):  0.5
        (cased):  False
        (include_unknown_words):  True
        (compare_against_original):  True
      )
    (1): PartOfSpeech(
        (tagger_type):  nltk
        (tagset):  universal
        (allow_verb_noun_swap):  True
        (compare_against_original):  True
      )
    (2): UniversalSentenceEncoder(
        (metric):  angular
        (threshold):  0.840845057
        (window_size):  15
        (skip_text_shorter_than_window):  True
        (compare_against_original):  False
      )
    (3): RepeatModification
    (4): StopwordModification
    (5): InputColumnModification(
        (matching_column_labels):  ['premise', 'hypothesis']
        (columns_to_ignore):  {'premise'}
      )
  (is_black_box):  True
)
```