UCLA
UCLA Electronic Theses and Dissertations

Title
Learning through Auxiliary Supervision for Multi-modal Low-resource Natural Language Processing

Permalink
https://escholarship.org/uc/item/2sx1t8rr

Author
Parvez, Md Rizwan

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Learning through Auxiliary Supervision for Multi-modal Low-resource Natural Language Processing

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Md Rizwan Parvez

2022

© Copyright by

Md Rizwan Parvez

2022

ABSTRACT OF THE DISSERTATION

Learning through Auxiliary Supervision for Multi-modal Low-resource Natural Language Processing

by

Md Rizwan Parvez
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Kai-Wei Chang, Chair

Despite much success, the effectiveness of deep learning models largely relies on the availability of large amounts of labeled data. A large amount of labeled data, however, is costly to acquire in many applications of interest, which hinders the applicability of these models, especially in resource-poor settings. On the other hand, with the growth of the internet, an enormous amount of user-generated data have been accumulated which is readily available and free. Although they may not annotate the necessary structured output of the target downstream tasks, they can provide relevant information and background knowledge which can be formed into auxiliary learning signals to enhance the target application. Hence, computational approaches for leveraging the open-source data as well as utilizing the resource-rich corpora in low-resource applications can enable us to build models for a broad spectrum of languages, domains, and modalities regardless of their training data size.

This dissertation discusses the fundamental challenges and proposes several approaches for multi-modal low-resource NLP problems that (1) construct auxiliary training data from un/labeled (open-source) resources and (2) learn through the auxiliary data and enhance the downstream application. The proposed approaches in this dissertation are effectively applied across a wide range of NLP applications, including sequence tagging, text classification, natural language inference, text to code generation, QA, and more.

The dissertation of Md Rizwan Parvez is approved.

Guy Van den Broeck

Quanquan Gu

Yizhou Sun

Junghoo Cho

Kai-Wei Chang, Committee Chair

University of California, Los Angeles

2022

TABLE OF CONTENTS

1 Introduction **1**

1.1 Overview 1

1.2 Thesis Statement 5

1.3 Outline of This Thesis 6

2 Evaluating and Selecting Transfer Sources for Enhancing Low-resource Target Applications **8**

2.1 Introduction 8

2.2 Source Valuation Framework 11

2.2.1 Background 12

2.2.2 SEAL-Shap 13

2.2.3 SEAL-Shap for Multiple Targets 16

2.2.4 Source Values without Evaluation Corpus 16

2.2.5 Source Corpora Selection by SEAL-Shap 17

2.3 Experimental Settings 17

2.4 Results and Discussion 19

2.4.1 Evaluating Source Valuation 19

2.4.2 Results without an Evaluation Corpus 24

2.4.3 Interpret Source Value by SEAL-Shap 25

2.4.4 Analysis and Ablation Study 28

2.5 Related Work 30

2.6 Summary 31

3 Retrieving and Incorporating Relevant Code and Summary for Code

Generation and Summarization . . . 32

3.1 Introduction . . . 32

3.2 Background . . . 35

3.2.1 Problem Formulation . . . 35

3.2.2 Retriever: DPR . . . 36

3.2.3 Generator: PLBART . . . 37

3.3 Proposed Framework: REDCODER . . . 37

3.3.1 Retriever: SCODE-R . . . 38

3.3.2 Generator: SCODE-G . . . 39

3.4 Experiment Setup . . . 41

3.4.1 Datasets and Implementations . . . 41

3.4.2 Evaluation Metrics . . . 43

3.4.3 Baseline Methods . . . 43

3.5 Results . . . 46

3.5.1 Code Generation . . . 46

3.5.2 Code Summarization . . . 48

3.6 Analysis . . . 48

3.7 Related Works . . . 53

3.8 Conclusion . . . 55

3.9 Summary . . . 55

4 Retrieving and Leveraging Entity-type Information for Language Modelling on Text with Named Entities . . . 56

4.1 Introduction . . . 56

4.2 Related Work and Background . . . 59

4.3 A Probabilistic Model for Text with Named Entities 61
4.3.1 Type model 64
4.3.2 Entity Composite Model 65
4.3.3 Training and Inference Strategies 65
4.4 Experiments 66
4.4.1 Retrieving Ingredient Types and the Results of Recipe Generation 67
4.4.2 Retrieving Token Types and the Results of Code Generation . . . 70
4.5 Quantitative Error Analysis 74
4.5.1 Case Study-1: Recipe Generation 74
4.5.2 Case Study-2: Code Generation 75
4.6 Conclusion 76
4.7 Summary 76

5 Retrieve and Augment Relevant Policy Documents for Question Answering on Privacy Policies 77

5.1 Introduction 77
5.2 Methodology 79
5.2.1 Policy Document Retriever 80
5.2.2 Filtering Oracle 81
5.2.3 Retriever Ensemble and Data Augmentation 81
5.3 Experiments 81
5.3.1 Results and Analysis 84
5.4 Ablation Study 86
5.5 Related Works 89
5.6 Limitations 89

5.7 Conclusion . . . 90

5.8 Summary . . . 91

6 Selecting and Augmenting Relevant Text Spans for Fast and Robust Text Classification . . . 92

6.1 Introduction . . . 92

6.2 Related Work . . . 94

6.3 Classification on a Test-Time Budget . . . 95

6.3.1 Learning a ***Selector*** . . . 96

6.3.2 The Data Aggregation Framework . . . 97

6.4 Experiments . . . 98

6.5 Conclusion . . . 101

6.6 Summary . . . 101

7 Conclusion and Future Work . . . 103

References . . . 105

LIST OF TABLES

2.1 Task statistics. #sources are for each target. In (m)odified GLUE, #sources is 8 for target MNLI, and 7 otherwise. "mlt-dom-senti" refers to Liu et al. (2017). 19

2.2 Performance on universal POS tagging when using each of language as the target language and the rest as source languages . '*', '\$', '†' denote SEAL-Shap model is statistically significantly outperforms *All Sources*, *Baseline-r* and *Baseline-s* respectively using paired bootstrap test with $p \leq 0.05$. "en" refers to the only source ("en") results in Wu and Dredze (2019). 21

2.3 POS tagging results (% accuracy) on SANCL 2012 Shared Task. '*' and '\$' denote the model using SEAL-Shap statistically significantly outperforms *All Sources* and *Baseline-r* respectively using paired bootstrap test with $p \leq 0.05$. MMD, and RENYI refer to Liu et al. (2019a) which use auxiliary unlabelled data in the target domain and focus on instance selection. *Baseline-s* has exactly same results as SEAL-Shap. 22

2.4 XNLI results. As a reference, we include two results from the recently published papers mBERT (Wu and Dredze, 2019) and "XLM-MLM" (Lample and Conneau, 2019). mBERT is trained on "en" only and "XLM-MLM" is applicable to XNLI languages only. 22

2.5 Cross-domain transfer results on multi-domain sentiment analysis task. Cai and Wan (2019) use unlabelled data from the target domain. 23

2.6 Zero-shot results on modified GLUE. Ma et al. (2019) selects instances from one source domain at once while we select a subset of source corpora. . . . 23

2.7 Running time for computing approximate Shapley value. The marker *represents the time is estimated by extrapolation. #Targets indicates number of target corpus evaluated simultaneously. #Samples is the number of samples used to train model for computing marginal contribution. TMC-Shap is equivalent to disable all the techniques (the first row of each block). . . . 28

2.8 Number of sources selected from 30 different languages by SEAL-Shap for the task of cross-lingual POS tagging. For the remaining 18 target languages, SEAL-Shap selects 27 source languages as potential. 30

3.1 Dataset Statistics. Gen., and Sum. refers to code generation and summarization tasks respectively. Summary denotes a natural language description paired with each code. For Concode, the input summary includes the corresponding environment variables and methods. All lengths are computed and averaged before tokenization. 40

3.2 Results on code generation on CodeXGLUE (Lu et al., 2021). 41

3.3 Code generation results on Concode dataset. SCODE-R was initialized with CodeBERT. GraphCodeBERT initialized results are similar. 43

3.4 Evaluation BLEU-4 score for code summarization on CodeXGLUE. Baseline results are reported from Ahmad et al. (2021a). 44

3.5 Results on code generation keeping the target code in the retrieval database. 44

3.6 MRR results on code retrieval from the validation and test set in CodeXGLUE. Our bi-encoder retriever SCODE-R is comparable with other cross-encoder models while it is much faster. DPR refers to Karpukhin et al. (2020a) and DPR (code) is trained with BM25 “hard” negative training schema built upon our source code datasets. 45

3.7 Human evaluation on code generation (CodeXGLUE-Python). REDCODER (SCODE-R + SCODE-G) achieves similar scores as SCODE-R that directly retrieves developers' written code which suggests that the quality of the code generated by SCODE-G are competitive with real code from programmers' perspective. 50

3.8 Retrieval database statistics. "Size" refers to both of parallel and nonparallel code or summaries. As Concode has a different data format, we only retrieve from itself. Nonparallel means the retrieval candidates are only code (for code gen.) and only summaries (for code sum.). CSNet (CodeSearchNet), CCSD refer to Husain et al. (2019) and Liu et al. (2021). 53

3.9 Ablation results on source code generation using the retrieved code and its summary together when the reference target code is absent and present in the retrieval database respectively. 53

3.10 Evaluation results of code summarization keeping the target summary in the retrieval database. 53

4.1 Comparing the performance of recipe generation task. All the results are on the test set of the corresponding corpus. AWD_LSTM (*type model*) is our *type model* implemented with the baseline language model AWD_LSTM (Merity et al., 2017). Our second baseline is the same language model (AWD_LSTM) with the type information added as an additional feature for each word. . . . 70

4.2 Comparing the performance of code generation task. All the results are on the test set of the corresponding corpus. fLSTM, bLSTM denotes forward and backward LSTM respectively. SLP-Core refers to Hellendoorn and Devanbu (2017). 73

4.3 Performance of fill in the blank task. 75

5.1 QA (sentence selection) from a policy document $\boldsymbol{S}$. **Sensitive**: For queries $\boldsymbol{R}$ and $\boldsymbol{I}$, annotators at large tagged sentence $\boldsymbol{s}_1$ as relevant, and irrelevant respectively. On the other hand, sentence $\boldsymbol{s}_n$, though analogous to $\boldsymbol{s}_1$ in meaning, was never tagged as relevant. **Ambiguous**: For queries $\boldsymbol{D}$, experts interpret $\boldsymbol{s}_1$ differently and disagree on their annotations. 78

5.2 Brief summary of PrivacyQA benchmark. 82

5.3 Test performances on PrivacyQA (mean$_{\pm\text{std}}$). BERT+Unans. refers to the previous SOTA performance (Ravichander et al., 2019). Retrieved candidates improves all the baseline QA models, especially when being filtered. Our ensemble retriever approach combines them and achieves the highest gains. 83

5.4 F1-score breakdown (values are in Appendix). B, PB, S refers to retrievers *BERT-R*, *PBERT-R*, and *SimCSE-R*. Different models performs better for different types (black-bold). ERA combines them and enhances performances for all categories (except: red). 84

5.5 Number of correct predictions. Note that F1-score is not proportional to the accuracy. B, PB, S refers to retrievers *BERT-R*, *PBERT-R*, and *SimCSE-R*. Different models performs better for different types (black-bold). ERA combines them and enhances performances for all categories 85

5.6 A fraction of retrieval examples (i). 86

5.7 A fraction of retrieval examples (ii). 87

5.8 Retrieval statistics per query type. 88

5.9 Model performances with and without filtering with top-k. In general, without filtering, augmenting the retrieved candidates enhances recall but may reduce the precision (and hence may not improve the overall F1). Filtering, however improves the performance specially with larger top-k candidates. 90

6.1 Accuracy and speedup on the test datasets. r, t denotes the selection rate (%), test-time respectively. Test-times are measured in seconds. The speedup rate is calculated as the running time of a model divided by the running time of the corresponding baseline. For our framework, top row denotes the best speedup and the bottom row denotes the best test accuracy achieved. Overall best accuracies and best speedups are boldfaced. Our framework achieves accuracies better than baseline with a speedup of **1.2x** and **1.3x** on IMDB, and AGNews respectively. With same or higher speedup, our accuracies are much better than Bag-of-Words. 97

6.3 Examples of the WE *selector* output on AGNews. Bold words are selected. 98

LIST OF FIGURES

1.1 Example illustration of text to code generation task. Source code consists of a very diverse token sequences (red circled) and it is extremely challenging to generate them just from a given natural language sentence w/o any additional hints. 2

1.2 User-generated texts in real-world applications are often fragmented, noisy and different from ideal training data. Left. Incomplete sentences w/ different missing words still have the same meaning. Right. Short form or word w/ unknown meaning. 3

1.3 Overview of the chapters in this thesis. The module workflow is left to right. 6

2.1 SEAL-Shap estimates the value of each source corpus by the average marginal contribution of that particular source corpus to every possible subset of the source corpora. Each block inside SEAL-Shap denotes a possible subset and the marginal contribution is derived by the difference of transfer results while trained with and without the corresponding source. Based on the source values, we select a subset of source corpora that achieves high transfer accuracy. . 9

2.2 Performance, and run time with up to top-3 sources ranked by different approaches. (a), (b) denotes cross-lingual and (c), (d) denotes cross-domain transfer. All models have same training configurations (e.g., sample size). All the run times are final except for *Greedy DFS* where it increases linearly with top-k. Adding top-2 and top-3 ranked sources, other methods drop their accuracy across the tasks while ours shows a consistent gain in all tasks and achieves the best results with top-3 sources. 18

2.3 Cross-lingual POS tagging accuracies on different target languages using top-3 sources ranked by SEAL-Shap. The ranker (red) selects similar sources as using SEAL-Shap with annotated target data (blue). Ranker trained to predict SEAL-Shap values (red) performs better than baseline (green) (Lin et al., 2019). 24

2.4 Cross-lingual POS tagging SEAL-Shap values, referring to the relative contribution of the source languages. 25

2.5 Source values by TMC-Shap and ours. TMC-Shap uses unbalanced full source corpora whereas SEAL-Shap that achieves similar source values uses balanced and sampled source corpora. Even with a small sample rate (R), source order is almost same. Higher sampling rate typically refers to better approximation but leads to expensive runtime. In general, for a reasonably large corpus, 20-30% samples (>few thousands) are found sufficient to achieve reasonable approximation. 26

2.6 Similar SEAL-Shap value curves for two closely related target languages in cross-lingual POS tagging. 26

2.7 Similar SEAL-Shap value curves for two closely related XNLI targets “en” and “fr”. In XNLI, the source corpora are prepared by machine translating from “en”. This data processing may affect the source values. Translation into “zh” being relatively better, although different from both targets, its source values are higher than others. 27

2.8 SEAL-Shap value on cross-domain NLI, referring to relative contribution of source domains. For target domain MNLI-mm, source domain QQP has the lowest contribution, whereas for target domain QNLI, source domain QQP has the highest contribution. 27

2.9 SEAL-Shap value with two (colored) seeds. 29

3.1 Illustration of our proposed framework REDCODER for code generation. Given an input summary, we first retrieve top-k candidate code (k=1 in this example). We then aggregate them and based on that a *generator* module generates the target sequence. 33

3.2 Example input/output for the code generation and summarization tasks. . . 35

3.3 An example retrieved code that is relevant yet does not match the reference. 37

3.4 Training scheme of the *retriever* module (SCODE-R) of our proposed framework REDCODER for the code generation task. Unlike in open-domain QA (Karpukhin et al., 2020a), we do not use "hard" negatives (*e.g.,* candidates retrieved by BM25 that do not exactly match the reference) during fine-tuning. 39

3.5 REDCODER-EXT input for code generation. 40

3.6 A qualitative example to show the effectiveness of retrieval-augmented generation as proposed in REDCODER framework 47

3.7 Recall@K for CodeR and BM25. CodeR refers to SCODE-R used for source code retrieval. 49

3.8 (Python) Code gen. BLEU vs target len. 50

3.9 Code gen. and sum. performance vs #retrievals. In general performance improves with higher number of augmented candidates. 51

3.10 #Code per target length. 54

3.11 BLEU vs target len. (Java) 54

4.1 An example illustration of the proposed model. For a given context (i.e., types of context words as input), the *type model* (in bottom red block) generates the type of the next word (i.e., the probability of the type of the next word as output). Further, for a given context *and* type of each candidate (i.e., context words, corresponding types of the context words, and type of the next word generated by the *type model* as input), the *entity composite model* (in upper green block) predicts the next word (actual entity name) by estimating the conditional probability of the next word as output. We conduct joint inference over both models to leverage type information for generating text. 58

5.1 Our framework. Given a pre-trained LM, we train a (i) retriever, (ii) QA model (oracle) both on the small-size labeled data. From an unlabeled corpus, we first, retrieve the coarse relevant sentences (positive examples) for the queries in the training set and use the oracle to filter out noisy ones. We repeat this for multiple different pre-trained LMs. Finally we aggregate them to expand the positive examples in the training set and learn any user-defined final QA model. 80

5.2 Venn diagram of low mutual agreement (<1%) among retrievers (a); even amplified after filtering (b). 85

6.1 Our proposed framework. Given a selection rate, a *selector* is designed to select relevant words and pass them to the *classifier*. To make the classifier robust against fractured sentences, we aggregate outputs from different *selectors* and train the *classifier* on the aggregated corpus. 93

6.2 Performance under different test-times on IMDB, AGNews, and SST-2. All the approaches use the same LSTM model as the back-end. Bag-of-Words model and our framework have the same bag-of-words *selector* cascaded with this LSTM *classifier* trained on the original training corpus and aggregated corpus, respectively. Our model (blue dashed line) significantly outperform others for any test-time budget. Also its performance is robust, while results of skim-RNN is inconsistent with different budget levels. 99

ACKNOWLEDGMENTS

Alhamdulillah. All praises are due to Allah who has blessed me with this. Only, He is the one who has made this possible.

First, I express my deepest gratitude to my great advisor Kai-Wei Chang for all that he has done for me and I pray for his "Hidaya". I believe that I am extremely fortunate to get the opportunity to work with him. He was always extremely kind, considerate to me. His advising style of hands on guidance in the early few years and then gradually giving freedom to lead a research problem has shaped my skills towards research. Even beyond, I would like to thank him specially for being so patient with my failures and helping me so much. I would undoubtedly say that he was an amazing and extra-ordinary advisor. I am also indebted to my thesis committee — Junghoo, Yizhou, Quanquan and Guy.

I am extremely grateful to my mentors Prof. Nanyun Peng, Prof. Baishakhi Ray (Columbia University), and Prof. Yuan Tian (UVa), Prof. Venkatesh Saligrama (Boston University) for supporting and guiding me through different research works. The knowledge I acquired while working with them is invaluable and I hope it will be an asset for the rest of my research life. I must also thank Wasi Uddin Ahmad, Saikat Chakrabarti, Tolga Bolukbasi, Jianfeng Chi for their collaboration on my research projects.

Many many thanks and appreciation to Paul Mineiro, Bryan McCann, Scott Wen-tau Yih, and Bin Zhang, who were my mentors during my summer internships at Microsoft AI and Research, Salesforce Research, Facebook AI Research and Google Research for the last four years. All of them have provided me a comfortable work environment and guided me towards a practical solution. I introspect and realize how much I have learned from all of them over the years starting from debugging, setting up feasible goals, tracking progress and modify goals, writing test cases and so on. Specifically, I would like to mention that a foundation work of my Phd is actually based on my FAIR internship when I came to study the dense passage retriever (DPR) model. I came back to school and continuing working with dense retrievers for multi-modality settings.

I want to thank the members of the UCLA NLP group, particularly for their support in reviewing my research works and providing feedback when needed. Thanks should also go to the support staff who ensure things run smoothly in our group at UVA and UCLA.

Lastly, I would like to thank my mother and everyone in my family whose efforts and sacrifices brought me here. A special thanks to Ifshita for becoming my lifelong (in sha Allah) associate during the journey. Jazakillahu khair for your unconditional love and support when things were not going well and when facing the despair.

VITA

2015	B.Sc Computer Science & Engineering Bangladesh University of Engineering & Technology, Dhaka, Bangladesh
2015–2016	Lecturer of Computer Science & Engineering United International University, Dhaka, Bangladesh
2016–2017	Teaching Assistant, Computer Science Department University of Virginia, Charlottesville, Virginia, USA
2017-2021	Research and Teaching Assistant, Computer Science Department University of California, Los Angeles, California, USA
2018	Research Intern Microsoft AI and Research, Redmond, Washington, USA
2019	Research Intern Salesforce Research, Palo Alto, California, USA
2020	Research Intern Facebook AI Research, (Remote) Seattle, Washington, USA
2021	Research Intern Google Research, (Remote) Kirkland, Washington, USA
2020–2022	Research Assistant, UCLA Natural Language Processing Group University of California, Los Angeles, California, USA

PUBLICATIONS

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray and Kai-Wei Chang. Building Language Models for Text with Named Entities. ACL. 2018.

Md Rizwan Parvez, Tolga Bolukbasi, Kai-Wei Chang, Venkatesh Saligrama. Robust Text Classifier on Test-Time Budgets. EMNLP. 2019.

Md Rizwan Parvez, and Kai-Wei Chang. Evaluating the Values of Sources in Transfer Learning. NAACL. 2021.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray and Kai-Wei Chang. Retrieval Augmented Code Generation and Summarization. EMNLP Findings. 2021.

Md Rizwan Parvez, Jianfeng Chi, Wasi Uddin Ahmad, Yuan Tian, and Kai-Wei Chang. Retrieval Enhanced Data Augmentation for Question Answering on Privacy Policies. Under Review EMNLP. 2022. Pre-print (https://arxiv.org/abs/2204.08952)

CHAPTER 1

Introduction

1.1 Overview

Natural Language Processing (NLP) is a sub-field of AI that enables a computing system mimicking the human interactions. Advancements of deep neural networks have facilitated usage of NLP application in multiple dimensions. It has been widely applied to different applications/tasks including machine translation, question answering (QA), dialogue systems and so on. It has also been applied to various domains (e.g., social media, e-commerce, medical/education sector, etc.,) languages (e.g., English, Arabic etc.,), and miscellaneous modalities (e.g., text, code, audio, image, video etc.,) and so on.

However, a key limitation of such deep neural NLP models is that training them require a large amount of annotated data while outputs of NLP problems are often structured and/or parallel and annotating them needs domain knowledge and expertise which makes it costly and time consuming to obtain high quality annotated data. As a result, we witness the majority NLP problems acrross all over different dimensions (e.g., domains/languages) are poor in resources, specially in terms of labeled training data. For example, world languages like Quechua and Navajo in South Africa do not even have a good written form[1], let alone a large size Wikipedia or datasets annotated by trained human workers. Furthermore, due to the lack of sufficient training data, many NLP applications are only limited to a few domains or languages such as machine translation (MT) systems is only available for 100 out of 7K world languages.

Low-resource phenomena is in fact beyond just lack of training data in specific domains

[1] https://langhotspots.swarthmore.edu/fastfacts.html

or languages. Many NLP problems intrinsically mimic the scenario of a low-resource setting. Let us consider a multi-modal example task of text to code generation. In the task of code generation, a concept is given in a natural language sequence and the target task is to generate the corresponding source code in the respective programming language (e.g., Python or Java). As Figure 1.1 shows, a source code consists of very diverse tokens (e.g., variable names, class or method names, operators, data types etc.,) and is extremely challenging to generate just from the given natural language sentence. Therefore, we do need some auxiliary supervision on the fly while performing the target task.

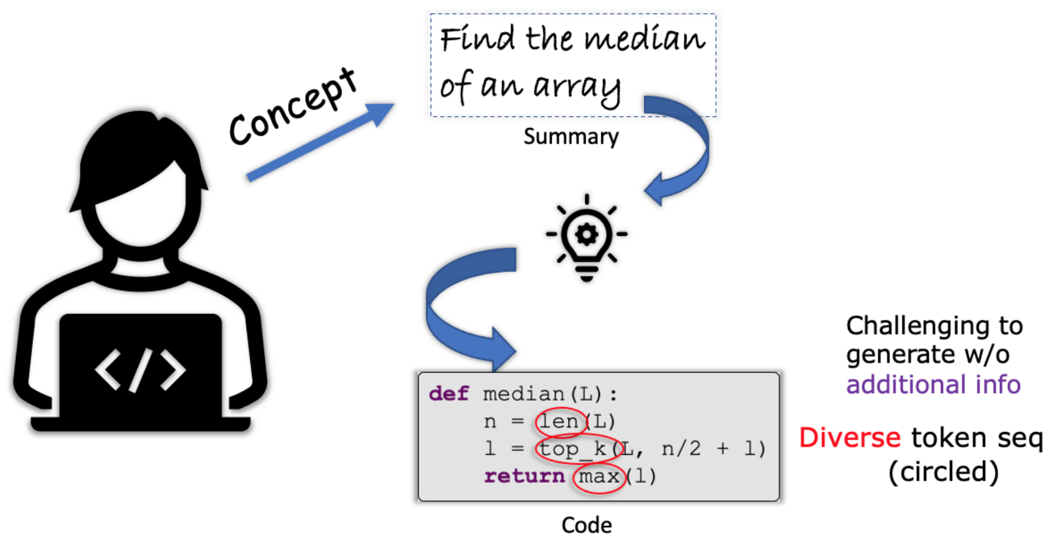


Figure 1.1: Example illustration of text to code generation task. Source code consists of a very diverse token sequences (red circled) and it is extremely challenging to generate them just from a given natural language sentence w/o any additional hints.

Apart from the scarcity of annotated data or the insufficiency of necessary information in the training data, there are other possible reasons why majority NLP problems exhibit are poor in resources. For example, often times we witness that a model may work for a specific benchmark but does not work on real inputs[2]. The reasons is user-generated texts in real-world applications are often fragmented, noisy and different from training such as in products review or tweeter people often write incomplete sentences (See Figure 1.2). In addition, in order to reduce data processing time and expedite the inference, we may deliberately filter out less informative/relevant words and the resultant input

[2] http://www.cs.cmu.edu/~ytsvetko/jsalt-part1.pdf

text becomes fractured for which there is no training data. Thus they also simulate a low-resource regime. Therefore to bring the benefit of advancing NLP technologies for all domains, applications, languages and modalities, this dissertation develops a framework to mitigate the low-resource bottlenecks in multi-modal NLP problems.

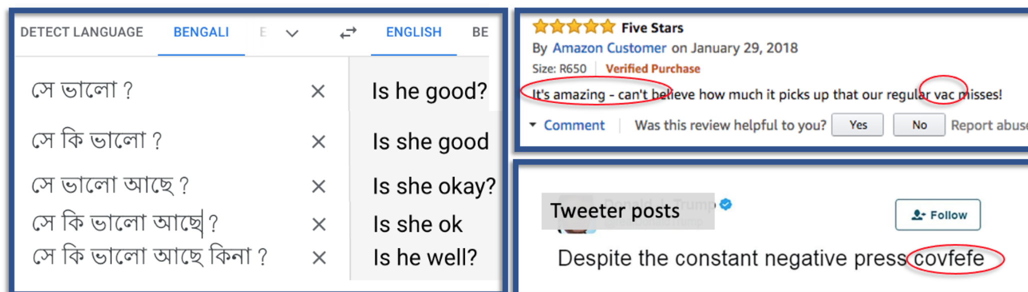


Figure 1.2: User-generated texts in real-world applications are often fragmented, noisy and different from ideal training data. Left. Incomplete sentences w/ different missing words still have the same meaning. Right. Short form or word w/ unknown meaning.

To build the framework, I took my motivation from humans. As human often solves problems using all his experience not just specific to that particular problem, I believe we can bring into additional information as a form of auxiliary data by leveraging the available high-resource source corpora or the open-source resources that accumulates an enormous amount of user generated data which are readily available and cost-effective. Although these data may not annotate the necessary outputs of our target task, they can provide relevant information and background knowledge which can be formulated some learning signals to enhance the NLP models. For example, to write down our source source code, we often look the API guidelines in library documentation websites (e.g., Pytorch, Tensorflow), we search it on Google or browse through examples on open source platforms such as Stack Overflow or even look into our previous source code on Github. We go through the top few relevant ones and adopt them in our settings. Similarly for the task of text to code generation, relevant code snippets can supplement the model w/ useful hints. Therefore, my research goal is to extract useful information from available resources as a form of auxiliary training data and to develop architecture independent approaches incorporating the auxiliary information in a structured way in order to enhance a wide range of NLP applications. In a nutshell there are two modules in my framework:

- **Module-1: Constructing auxiliary training data:** In this module I develop a principled way for constructing the auxiliary data to enhance the low-resource multi-modal NLP applications. As discussed above, I focus on **extracting (i.e., selecting or retrieving)** additional useful information from the high-resource source corpora (e.g., English language) or open-source repositories (e.g., Wikipedia, Github) for the target low-resource application (e.g., Swahili language or text to code generation multi-modal task). We divide the settings of the corresponding NLP problems and the extraction of additional information in the following three types:
  - **Type-1: New training corpora:** in this low-resource scenario, we construct a new training dataset of $x \rightarrow y$ format by selecting and combining a set of potential high-resource source corpora where x, y are the corresponding input, output. In terms of the granularity, this is a **corpora level** augmentation and can facilitate **zero-shot** learning. Here, the source corpora are **labeled** data and will be directly used as the training set for the target task.
  - **Type-2: Additional features:** In this setting, we assume we have a training dataset $(<x, y>)$ but may need additional hints to better learning to map to the outputs. We bring new token sequences as such auxiliary hints/features. Formally this augmentation can be noted as: $x \oplus x' \oplus y' \rightarrow y$ where x', y' are single/multiple additional features (set of token sequences) in the input and output space respectively. We retrieve such relevant token sequences from a large candidate sets (e.g., open-source repositories such as Github or Wikipedia). Here, the retrieval corpora can be **both paired (labeled) or unlabeled** data. In terms of the granularity this is a **feature level** augmentation and in our contexts, we considered text documents (token sequences) as the features.
  - **Type-3: Additional instances:** In this setting we assume we have a training dataset $(<x, y>)$ but insufficient in size or the data is imbalanced (less training example for particular class/label y). We augment with new instances $(<x_{new}, y>)$ where x_{new} is a candidate text retrieved from an **unlabeled**

corpora or perturbed from the base text x. In terms of the granularity, this is a **instance level** augmentation. Note that while the perturbation of the base text are mostly based on paraphrasing (e.g., word replacements or back-translation or using generative models such as GPT-2 etc.,) in literature, as per our focus (extraction), we select different words, phrases or sentences from base text x and use them as the perturbed text x_{new}. Therefore, *Type-3* augmentation can be further decomposed into **words/phrase/sentence** sub-categories.

- **Module-2: Learning from the auxiliary supervision:** This module takes input the training data constructed by *Module-1* and formulate an auxiliary learning signals using it. A key advantage of our modular approach is that *Module-1* simply constructs a new training dataset leaving *Module-2* as generic to different dimensions (tasks, domains, languages and modalities) and can be adopted to any off-the-shelf models w/o requiring much change in the underlying model architectures (e.g., LSTM, Transformers etc.,). In particular, for the aforementioned *Module-1* types, we consider a few specific tasks (e.g., language modeling, text classification, text to code generation, code to text summarization, QA, natural language inference, parts-of-speech tagging) and showcase that our approach effectively enhance the corresponding state-of-the-art model of different architectures in multiple aspects including performance, speed, robustness and interpretability.

1.2 Thesis Statement

Low-resource NLP models suffers greatly due to the insufficiency of training data. On the other hand, w/ the growth of technology, an enormous amount of labeled and unlabeled data in multiple modalities (code, text, image, video, external libraries etc.,) has been accumulated which are readily available and cost-effective. However, these resource-rich and open-source data may not annotate the necessary outputs of our target task nor they are all useful. Our thesis statement is *active selection/retrieval of auxiliary supervision can enhance a wide range of NLP applications.*

1.3 Outline of This Thesis

The rest of this document is organized as follows.

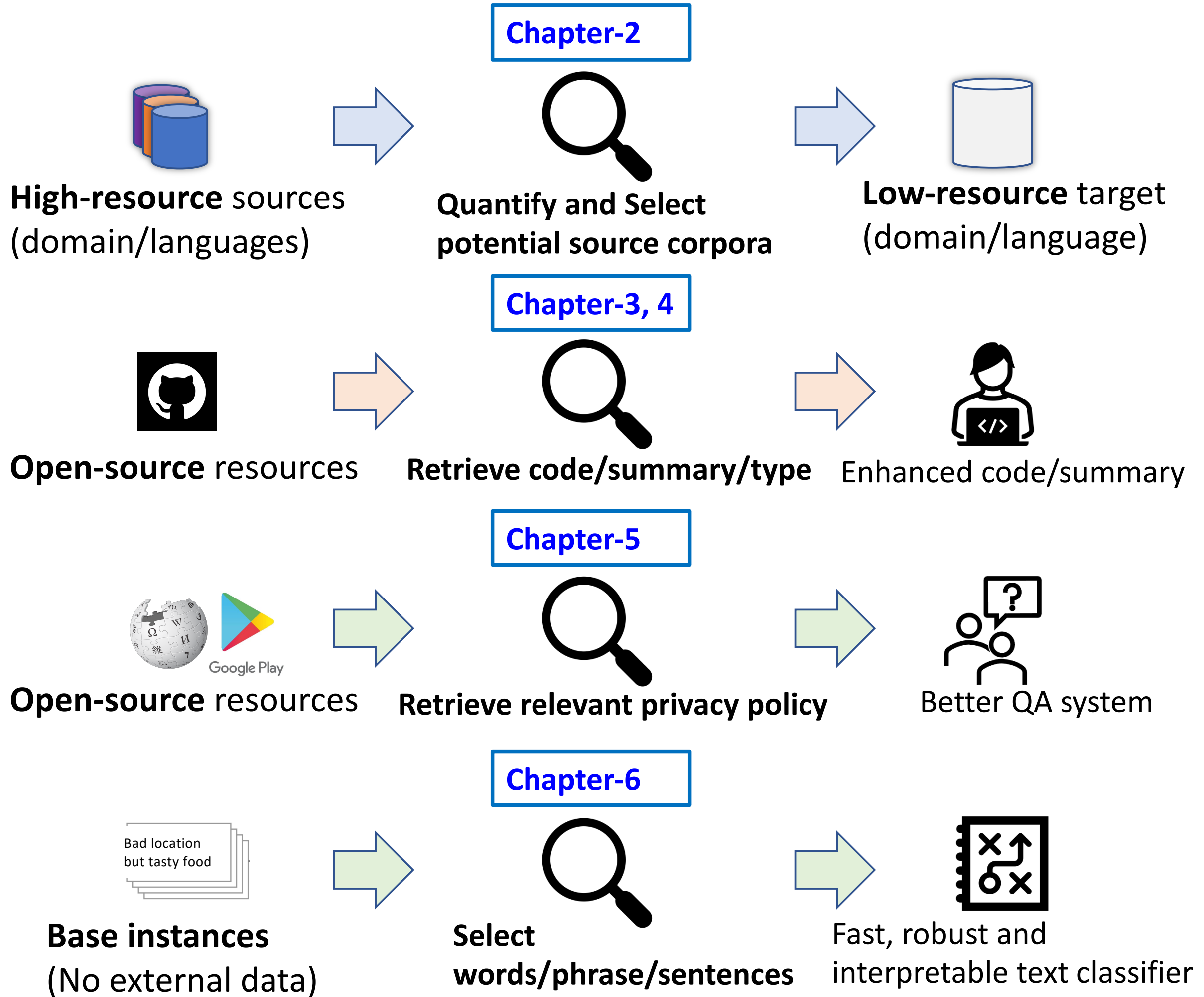


Figure 1.3: Overview of the chapters in this thesis. The module workflow is left to right.

Chapter 2 introduces our work (Parvez and Chang, 2021) on selecting potential high-resource source corpora (i.e., *Type-1* corpora level data selection) for zero-shot cross-lingual and cross-domain transfer learning. We propose a method to quantify the usefulness of the sources and identify the beneficial transfer sources. We then show that leveraging them, a number of low-resource text classification tasks can be enhanced.

In Chapter 3, 4 we discuss two of our works (Parvez et al., 2018, 2021) on the *Type-2* feature level data selection and incorporation. For an example text to code generation

task, we present how can we retrieve auxiliary features and incorporate them for better generation. In the first work Chapter-4, we use a rule-based selection technique and use different rule-based off-the-shelf state-of-the-art tools, libraries, resources (e.g., symbol table) to find the entity type information (e.g., variable data types such as int, float, user-defined class etc.,) of the entity names (e.g., variables, operator, function or method name etc.,) for better code generation. In our second work Chapter 5, we develop a dense code retriever than can retrieve relevant candidate code from a large unlabeled open-source code repository Github. Using this, we retrieve top-k relevant code and use them as auxiliary feature while generating the code from text.

In Chapter 5, we then further improve our retriever model and apply it for question answering on privacy policies (Parvez et al., 2022). For each queries in our training set, we retrieve new relevant policy statements from the unlabeled Google Play Store Policy documents and add the paired instance <query, retrieved policy> (*Type-3* instance level data) in the training set which lead to a new state-of-the-art performance on this task.

Chapter 6 develops a computationally inexpensive selector model that can select the relevant words/phrase/sentences from an input text as per a given selection rate. We vary the rates and collect the different versions of the same input text. We then augment them as additional instances (*Type-3* instance level data) and improve the performance, speed, and robustness of a text classifier model.

Finally, in Chapter 7 we summarize the contributions and findings of this thesis and provide a brief overview of the future works.

CHAPTER 2

Evaluating and Selecting Transfer Sources for Enhancing Low-resource Target Applications

Transfer learning that adapts a model trained on data-rich sources to low-resource targets has been proven an effective approach to enhance the low-resource applications in natural language processing (NLP). A common transfer practice is to simply choose only a few potential sources to train the transfer models. For example, many cross-lingual studies consider only the English language as the source–limiting the auxiliary supervision to be leveraged from other high-resource sources. In contrast, when training a transfer model over multiple sources, not every source is beneficial for the target. To better transfer a model, it is essential to understand the usefulness of the sources. In this Chapter, we develop an efficient source valuation framework for quantifying the usefulness of the sources (e.g., domains/languages) and use it for selecting the potential transfer source corpora. We finally create a training set augmenting the selected source corpora (i.e., *Type-1* **corpora level**). Experiments and comprehensive analyses on both cross-domain and cross-lingual transfers demonstrate that our framework is not only effective in choosing useful transfer sources but also the source values match the intuitive source-target similarity.

2.1 Introduction

Transfer learning has been widely used in learning models for low-resource scenarios by leveraging the supervision provided in data-rich source corpora. It has been applied to NLP tasks in various settings including domain adaptation (Blitzer et al., 2007; Ruder and Plank, 2017), cross-lingual transfer (Täckström et al., 2013; Wu and Dredze, 2019),

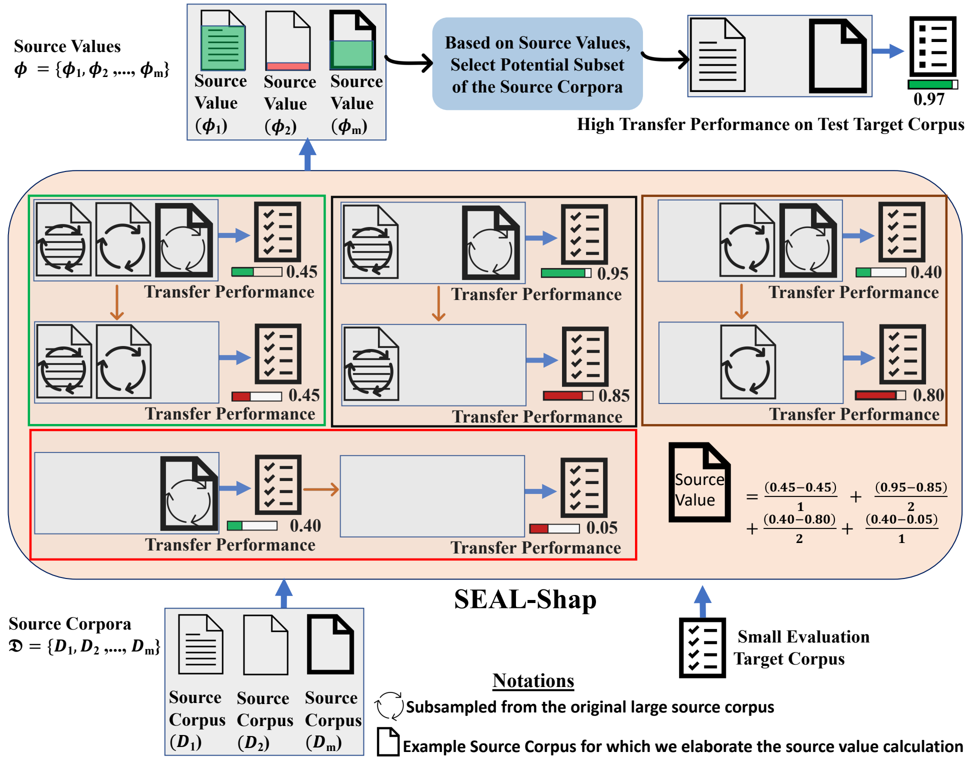


Figure 2.1: SEAL-Shap estimates the value of each source corpus by the average marginal contribution of that particular source corpus to every possible subset of the source corpora. Each block inside SEAL-Shap denotes a possible subset and the marginal contribution is derived by the difference of transfer results while trained with and without the corresponding source. Based on the source values, we select a subset of source corpora that achieves high transfer accuracy.

and task transfer (Liu et al., 2019b; Vu et al., 2020).

A common transfer learning setting is to train a model on a set of sources and then evaluate it on the corresponding target (Yao and Doretto, 2010; Yang et al., 2020).[1] However, not every source corpus contributes equally to the transfer model. Some of them may even cause a performance drop (Ghorbani and Zou, 2019; Lin et al., 2019). Therefore, it is essential to understand the value of each source in the transfer learning not only to achieve a good transfer performance but also for analyzing the source-target relationships.

[1]In this paper, we focus on two transfer learning scenarios: 1) cross-lingual and 2) cross-domain. We train a model on a set of source corpora and evaluate on a target corpus where each "corpus" refers to the corresponding domain or language.

Nonetheless, determining the value of a source corpus is challenging as it is affected by many factors, including the quality of the source data, the amount of the source data, and the difference between source and target at lexical, syntax and semantics levels (Ahmad et al., 2019; Lin et al., 2019). The current source valuation or ranking methods are often based on single source transfer performance (McDonald et al., 2011; Lin et al., 2019; Vu et al., 2020) or leave-one-out approaches (Tommasi and Caputo, 2009; Li et al., 2016; Feng et al., 2018; Rahimi et al., 2019). They do not consider the combinations of the sources. Consequently, they may identify the best single source corpus effectively but their top-k ranked source corpora may achieve limited gain in transfer results.

In this paper, we introduce *SEAL-Shap (Source sElection for trAnsfer Learning via Shapley value)*, a source valuation framework[2] (see Fig 2.1) based on the Shapley value (Shapley, 1952; Roth, 1988) in cooperative game theory. SEAL-Shap adopts the notion of Shapely value to understand the contribution of each source by computing the approximate average marginal contribution of that particular source to every possible subset of the sources.

Shapley value is a unique contribution distribution scheme that satisfies the necessary conditions for data valuation like fairness and additivity (Dubey, 1975; Jia et al., 2019a,b). As many model explanation methods including Shapley value are computationally costly (Van den Broeck et al., 2021), in a different context of features and data valuation in machine learning, Ghorbani and Zou (2019) propose to use an approximate Shapley value to estimate the feature or data values.

However, the existing approximation methods for estimating Shapley values are not scalable for NLP applications. NLP models are often large (e.g., BERT (Devlin et al., 2019)) and NLP transfer learning usually assumes a large amount of source data. To deal with the scalability issue, we propose a new sampling scheme, a truncation method, and a caching mechanism to efficiently approximate the source Shapley values.

We evaluate the effectiveness of SEAL-Shap under various applications in quantifying

[2]Our source codes are available at https://github.com/rizwan09/NLPDV/

the usefulness of the source corpora and in selecting potential transfer sources. We consider two settings of source valuation or selection: (1) where a small target corpus is available; and (2) where we only have access to the linguistic or statistical features of the target, such as language distance to the sources, typological properties, lexical overlap etc. For the first setting, we use the small target data as the validation set to measure the values of the sources w.r.t the target. For the second setting, we follow Lin et al. (2019) to train a source ranker based on SEAL-Shap and the available features.

We conduct extensive experiments in both (zero-shot) cross-lingual and cross-domain transfer settings on three NLP tasks, including POS tagging, sentiment analysis, and natural language inference (NLI) with different model architectures (BERT and BiLSTM). In a case study, on the cross-lingual transfer learning, we exhibit that the source language values are correlated with the language family and language distance—indicating that our source values are meaningful and follow the intuitive source-target relationships. Lastly, we analyze the approximation correctness and the run-time improvement of our source valuation framework SEAL-Shap.

2.2 Source Valuation Framework

We propose SEAL-Shap, a source valuation framework. We start with the setting where we have only one target and multiple sources. We denote the target corpus by V and the corresponding set of source corpora by $\mathcal{D} = \{D_1, \cdots, D_m\}$. Our goal is to quantify the value Φ_j of each source corpus D_j to the transfer performance on V and explain model behaviors. Once the source values are measured, we can then develop a method to select either all the sources or a subset of sources (i.e., $\subseteq \mathcal{D}$) that realizes a good transfer accuracy on V. Below, we first review the data Shapley value and its adaptation for transfer learning. Then, we describe how SEAL-Shap efficiently quantifies Φ_j and how to use it to select a subset of sources for model transfer.

2.2.1 Background

2.2.1.1 Data Shapley Value

Shapley value is designed to measure individual contributions in collaborative game theory and has been adapted for data valuation in machine learning (Ghorbani and Zou, 2019; Jia et al., 2019a,b). In the transfer learning setting, on a target corpus V, let $Score(C_\Omega, V)$ represent the transfer performance of a model C trained on a set of source corpora Ω.[3] The Shapley value Φ_j is defined as the average marginal contribution of a source corpus D_j to every possible subsets of corpora $\mathcal{D}$:

$$\frac{1}{m}\sum_{\Omega\subseteq\mathcal{D}-D_j}\frac{Score(C_{\Omega\cup D_j}, V)-Score(C_\Omega, V)}{\binom{m-1}{|\Omega|}}.$$

2.2.1.2 TMC-Shap for Transfer Learning:

Computing the exact source-corpus Shapley value, described above, is computationally difficult as it involves evaluating the performances of the transfer models trained on all the possible combinations of the source corpora. Hence, Ghorbani and Zou (2019) propose to approximate the evaluation by a truncated Monte Carlo method. Given the target corpus V and a set of source corpora $\mathcal{D}$, for each epoch, a source training data set $\Omega \subseteq \mathcal{D}$ is maintained and a random permutation π on $\mathcal{D}$ is performed (corresponds to line 6 in Algorithm 1 which is discussed in Sec 2.2.2). Then it loops over every source corpus π_j in the ordered list π and compute its marginal contribution by evaluating how much the performance improves by adding π_j to Ω: $Score(C_{\Omega\cup\pi_j}, V) - Score(C_\Omega, V)$. These processes are repeated multiple rounds and the average of all marginal contributions associated with a particular source corpus is taken as its approximate Shapley value (line 18 in Algorithm 1). When the size of Ω increase, the marginal contribution of adding a new source corpus becomes smaller. Therefore, to reduce the computation, Ghorbani and Zou (2019) propose to truncate the computations at each epoch when the marginal

[3] In this paper, we consider a model trained on the union of the source data and the loss function for training the model is aggregated from the loss functions defined on each source. However, our approach is agnostic to how the model is trained and can be integrated with other training strategies.

contribution of adding a new source π_j is smaller than a user defined threshold *Tolerance* (line 10-11, 18 in Algorithm 1).[4]

2.2.2 SEAL-Shap

Despite that TMC-Shap improves the running time, it is still unrealistic to use it in our setting where both source data and model are large. For example, in cross-lingual POS tagging on Universal Dependencies Treebanks, on average, it takes more than 200 hours to estimate the values of 30 source languages with multi-lingual BERT (See Sec 5.4). Therefore, in the following, we propose three techniques to further speed-up the evaluation process.

2.2.2.1 Stratified Sampling

When computing the marginal contributions, training a model C on the entire training set Ω is computationally expensive. Based on extensive experiments, when computing these marginal contributions, we find that we do not need the performance difference of models trained with the entire training sets. For a reasonably large source corpus, 20-30% samples[5] in each source achieve lower but representative performance difference, in general. Therefore, we sample a subset of instances to evaluate the marginal contributions. To address computational limitation and scale to large data, sampling techniques have been widely discussed (L'heureux et al., 2017). In particular, we employ a stratified sampling (Neyman, 1992) to generate a subset $\mathcal{T}$ from Ω by sampling training instances from each source corpus Ω_x with a user defined sample rate η. Then, we train the model on $\mathcal{T}$ (line 14-15 in Algorithm 1). The quantitative effectiveness of this technique is discussed in Sec 5.4 and the impact of different sampling rates are presented in Fig 2.5.

---

[4]Setting *Tolerance* to 0 turns off the truncation.

[5]Higher sampling rate typically leads to better approximation but are expensive in run-time.

Algorithm 1 SEAL-Shap

Input:
Source corpora $\mathcal{D} = \{D_1, \cdots, D_m\}$,
target corpus V,
Random sampler $\mathcal{S}$,
sample size η,
num of epochs *nepoch*,
Classifier C

Output:
Source-corpora Shapley values $\{\Phi_1 ..., \Phi_m\}$

Initialize:
Score cache $S \leftarrow \{\}$,
source Shapley values $\Phi_x \leftarrow 0$ for $x = 1 \ldots m$,
epoch $t \leftarrow 0$
$\mathcal{D}_{samp} \leftarrow \{\mathcal{S}(D_x, \eta), \forall D_x \in \mathcal{D}\}$
$C_{\mathcal{D}_{samp}} \leftarrow$ Train C on $\mathcal{D}_{samp}$

while Converge or $t < nepoch$ **do**
  $t \leftarrow t + 1$
  π : Random permutation of $\mathcal{D}$
  $v_0 \leftarrow \rho$
  for $j \in \{1, \cdots m\}$ **do**
    $\Omega \leftarrow \{\pi_1, \cdots, \pi_j\}$
    if $|$ Score($C_{\mathcal{D}_{samp}}, V$) - $v_{j-1}| <$ Tolerance **then**
      $v_j \leftarrow v_{j-1}$
    else
      if $\Omega \notin S$ **then**
        $\mathcal{T} \leftarrow \{\mathcal{S}(\Omega_x, \eta), \forall \Omega_x \in \Omega\}$
        $C_j \leftarrow$ Train C on $\mathcal{T}$
        Insert Ω into S with $S_\Omega \leftarrow$ Score(C_j, V)
      $v_j \leftarrow S_\Omega$

$\Phi_{\pi_j} \leftarrow \frac{t-1}{t}\Phi_{\pi_j} + \frac{1}{t}(v_j - v_{j-1})$

2.2.2.2 Truncation

As discussed in Sec 2.2.1, at each epoch, Ghorbani and Zou (2019) truncate the computations once a marginal contribution becomes small when looping over the ordered list π of that corresponding epoch, typically for the last few sources in π. On the other hand, at the beginning of each epoch, when computing the marginal contribution by adding the first source corpus π_1 into an empty Ω, the contribution is computed by the performance gap between a model trained on π_1 and a random baseline model without any training. Usually, the performance of a random model (v_0) is low and hence, the marginal contribution is high in the first step, in general. As this scale of marginal contributions at the first step is drastically different from later steps, it leads TMC-Shap to converge slowly. Hence, to restrict the variance of the marginal contributions, we down weight the marginal contributions of the first step by setting $v_0 = \rho$, where ρ is a hyper-parameter[6] indicating the baseline performance of a model (line 7, 18 in Algorithm 3).

2.2.2.3 Caching

When computing the source Shapley values, we have to repeatedly evaluate the performance of the model on different subsets of source corpora. Sometimes, we may encounter subsets that we have evaluated before. For example, consider a set of source corpora $\mathcal{D} = \{D_1, D_2, D_3\}$ and we evaluate their Shapley values through two permutations: $\pi_1 = [D_3, D_1, D_2]$, and $\pi_2 = [D_1, D_3, D_2]$. When we compute the marginal contribution of the last source corpus D_2, in both cases the training set $\Omega = \{D_1, D_3\}$. That is, if we cache the result of $Score(C_{D_1 \cup D_3})$, then we can reuse the scores. We implement this cache mechanism in line 1, 13, 16, 17 in Algorithm 3. With these optimization techniques, we improve the computation time by about 2x (see Sec 5.4). This enables us to apply this techniques in NLP transfer learning.

Note that whenever an Ω causes a cache miss, for each source Ω_x, as discussed above in this Section, we sample a new set of instances (line 13-14 in Algorithm-1). Thus, given

[6] Typically a factor of the performance achieved when using only one source, or all the sources together

a reasonably large number of epochs, our approach performs sampling for a large number of times and in aggregation, it evaluates a wide number of samples in each source.

2.2.3 SEAL-Shap for Multiple Targets

Many applications require to evaluate the values of a set of sources with respect to a set of targets. For example, under the zero-shot transfer learning setting, we assume a model is purely trained on the source corpora without using any target data. Consequently, then the same trained model can be evaluated on multiple target corpora. With this intuition, whenever the model is trained on a new training set Ω, SEAL-Shap evaluates it on all the target corpora and caches all of them accordingly.

2.2.4 Source Values without Evaluation Corpus

In the previous discussions above, we assume a small annotated target corpus is available and can be used to evaluate the transfer performances. However, in some scenarios, only some linguistic or statistical features of the sources and targets, such as language distance and word overlap, are available. Lin et al. (2019) show that by using these features, we can train a ranker to sort the sources to unknown targets by predicting their value. In the following, we extend their ranker by incorporating it with SEAL-Shap.

Given the set of training corpora $\mathcal{D}$ and the actual target corpus V, we iteratively consider each training corpus D_j as target and the rest m-1 corpora as the sources. We compute the corresponding source values $\mathcal{Y}_{\mathcal{D}}^{D_j} = \{\Phi_{D_1}, \ldots, \Phi_{D_{j-1}}, \Phi_{D_{j+1}}, \ldots, \Phi_{D_m}\}$. Now, w.r.t the target D_j, the linguistic or statistical features of the source corpora (e.g., language distance from the target, lexical overlap between the corresponding source and the target) $\mathcal{X}_{\mathcal{D}}^{D_j} = \{F^j(D_1), \ldots, F^j(D_{j-1}), F^j(D_{j+1}), \ldots, F^j(D_m)\}$ where F^j denotes the source feature generator function for the corresponding target D_j. This feature vector of the source corpora ($\mathcal{X}_{\mathcal{D}}^{D_j}$) is a training input and their value vector ($\mathcal{Y}_{\mathcal{D}}^{D_j}$) is the corresponding training output for the ranker. We repeat this for each training corpus and generate the respective training inputs and outputs for the ranker. Once trained, for the

actual target V and the source corpora $\mathcal{D}$, the ranker can predict the values of the source corpora $\mathcal{Y}_{\mathcal{D}}^{V}$ only based on the linguistic source features $\mathcal{X}_{\mathcal{D}}^{V}$.

2.2.5 Source Corpora Selection by SEAL-Shap

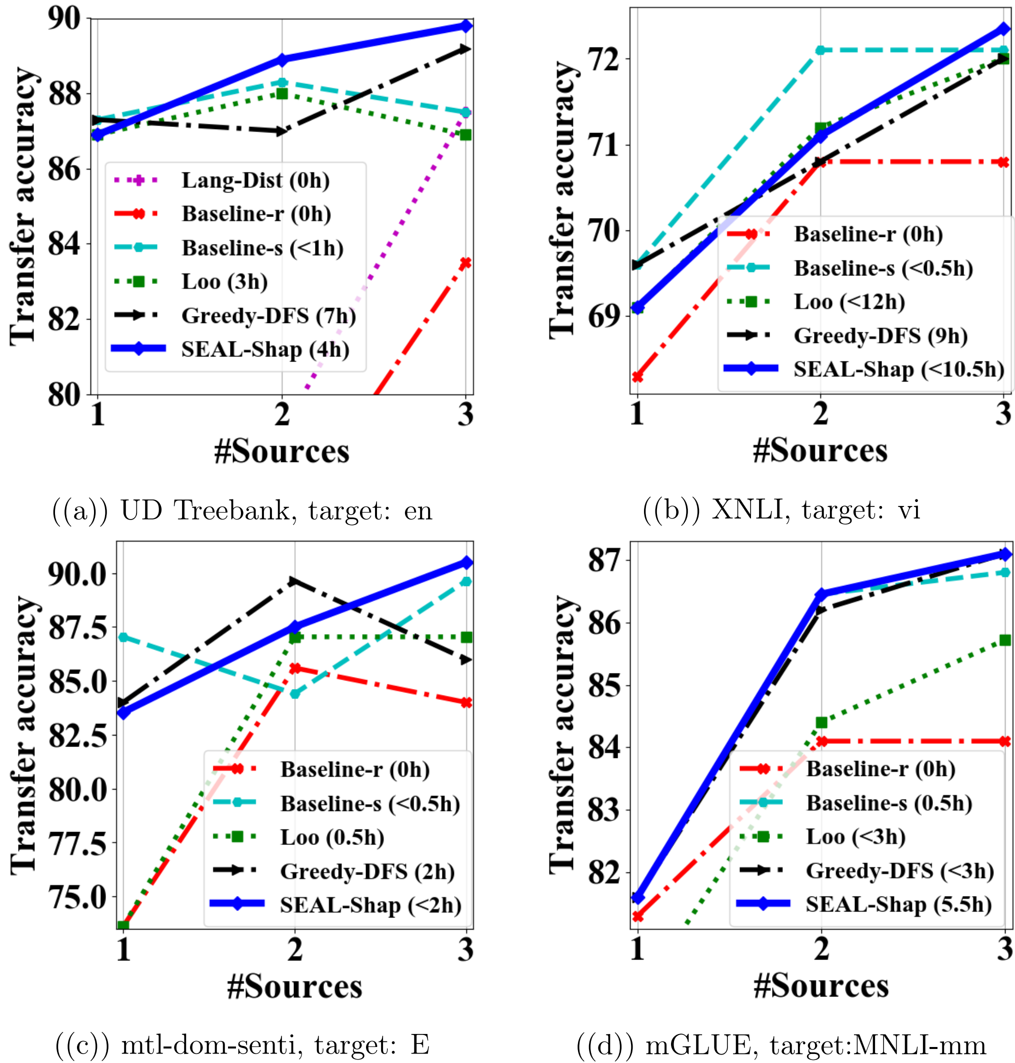
The source values computed in Sec 2.2.2-2.2.4 estimate the usefulness of the corresponding transfer sources and can be used to identify the potential sources which lead to the good transfer performances. We select the potential source corpora in two ways. (i) Top-k: We simply sort the sources based on their values and select the user defined top-k sources. (ii) Threshold: When an annotated evaluation dataset in target corpus V is available, after computing the source values, we empirically set a threshold θ and select each source that has source value higher than θ. On that evaluation target corpus, we tune and set θ for which the corresponding transfer model achieves the best performance.

2.3 Experimental Settings

We conduct experiments on *zero-shot cross-lingual and cross-domain transfer* settings. Models are trained only on the source languages/domains and directly applied in target languages/domains.

Cross-lingual Datasets We conduct experiments on two popular cross-lingual transfer problems: (i) universal POS tagging on the Universal Dependencies Treebanks (Nivre et al., 2018). Following Ahmad et al. (2019), we select 31 languages of 13 different language families. (ii) natural language inference on the XNLI dataset (Conneau et al., 2018), that covers 15 different languages. XNLI task is a 3-way classification task (entailment, neutral, and contradiction).

Cross-domain Datasets We consider three domain transfer tasks: (i) POS tagging: we use the SANCL 2012 shared task datasets (Petrov and McDonald, 2012) that has six different domains. (ii) Sentiment analysis: we use the multi-domain sentiment datasets (Liu et al., 2017) which has several additional domains than the popular Blitzer et al. (2007)



((a)) UD Treebank, target: en

((b)) XNLI, target: vi

((c)) mtl-dom-senti, target: E

((d)) mGLUE, target:MNLI-mm

Figure 2.2: Performance, and run time with up to top-3 sources ranked by different approaches. (a), (b) denotes cross-lingual and (c), (d) denotes cross-domain transfer. All models have same training configurations (e.g., sample size). All the run times are final except for *Greedy DFS* where it increases linearly with top-k. Adding top-2 and top-3 ranked sources, other methods drop their accuracy across the tasks while ours shows a consistent gain in all tasks and achieves the best results with top-3 sources.

dataset. (iii) NLI: we consider a (modified) binary classification (e.g., entailed or not) dataset used in Ma et al. (2019). It is made upon modification on GLUE tasks (Wang et al., 2018) and has four domains. As GLUE test sets are unavailable, for each target domain, we use the original dev set as the pseudo test set and randomly select 2,000 instances from its training set as the pseudo dev set. Table 2.1 summaries the statistics of the tasks and datasets in our experiments.

Transfer	Task	Dataset	#target	#source
Language	POS tag	UD Treebank	31	30
	NLI	XNLI	15	14
Domain	POS tag	SANCL 2012	6	5
	NLI	mGLUE	4	7+
	Sentiment Ana.	mlt-dom-senti	14	13

Table 2.1: Task statistics. #sources are for each target. In (m)odified GLUE, #sources is 8 for target MNLI, and 7 otherwise. "mlt-dom-senti" refers to Liu et al. (2017).

Classifier and Preprocessing For all domain transfer tasks, we use BERT and for all language transfer tasks, we use multi-lingual BERT (Devlin et al., 2019) models except for cross-doman POS tagging where we consider the state-of-the-art BiLSTM based Flair framework (Akbik et al., 2018). For BERT models, we use the Transformers implementations in the Huggingface library Wolf et al. (2019a). For significance test, we use an open-sourced library.[7] By default, no preprocessing is performed except tokenization.

2.4 Results and Discussion

In the following, we first verify SEAL-Shap is an effective tool for source valuation. Then, we evaluate the source values when an evaluation target corpus is unavailable. In Sec 2.4.3, we interpret the relations between sources and targets based on the SEAL-Shap values. Finally, we analyze our method with comprehensive ablation studies.

2.4.1 Evaluating Source Valuation

We assess our source valuation approach in compare to the following baselines: (i) *Baseline-s*: source values are based on the single source transfer performance. (ii) *Leave-one-out (LOO)*: source values are based on how much transfer performance we loose if we train the model on all the sources except the corresponding one. (iii) *Baseline-r*: a random

[7] github.com/neubig/util-scripts/blob/master/paired-bootstrap.py

baseline that assigns random values to sources.[8] (iv) *Greedy DFS*: the top-1 ranked source is same as that of *Baseline-s*. Next, it selects one of the remaining sources as top-2 that gives the best transfer result along with the top-1 and so on. (v) *Lang-Dist*: (if available) in reverse order of target-source language distance (Ahmad et al., 2019).[9]

2.4.1.1 Balancing Source Corpora

In the experiements, our focus is to understand the values of the sources. For some datasets, the sizes of source corpora are very different. For example, in UD Treebank, the number of instances in Czech, and Turkish is 69k, 3.5k, respectively. Since data-size is an obvious factor, we conduct experiments on balanced data to reduce the influence of data-size in the analysis. We sub-sample the source corpora to ensure their sizes are similar. Specifically, for the cross-domain NLI task, we sample 20k instances for each source. For others, we sub-sample each source such that the size of the corpus is the same as the smallest one in the dataset. However, our approach can handle both balanced or unbalanced data and the source values are similar in conclusions (e.g., see Fig 2.5).

2.4.1.2 Results

We first compare these methods by selecting top-k sources ranked by each of the approach and reporting the corresponding transfer performance. With $k = 3$, we plot the corresponding transfer results and the running time for valuation in Fig 2.2. As mentioned in Sec 5.1, the relatively strong *Baseline-s* can select the best performing top-1 source but with top-2 and top-3 sources, the performances drop on cross-domain sentiment analysis and cross-lingual POS tagging (See Fig 2.2(c) and 2.2(a)) while our approach shows a consistent gain in all of the these tasks and with top-3 sources it achieves the best performances.

[8] Our experiments with different seeds result in different but similar results.

[9] Ahmad et al. (2019) compute the distances from an annotated dependency parse tree based on UD Treebank.

Lang	en	All Source	Baseline-r	Baseline-s	SEAL-Shap
en	-	82.71	86.32	86.39	**88.55**$^{*\$\dagger}$
fr	-	94.60	94.63	**94.83**	94.79
da	88.3	88.94	89.30	89.23	**89.47***
es	85.2	93.15	93.00	93.04	**93.21**$^{\$}$
it	84.7	96.58	96.43	**96.71**	96.67
ca	-	91.54	91.64	90.78	**92.08**$^{*\$\dagger}$
sl	84.2	93.28	93.50	92.89	**93.52***†
nl	75.9	90.10	90.19	90.14	**90.26**
ru	-	92.98	92.91	92.71	**93.13**$^{*\$\dagger}$
de	89.8	90.79	91.07	**91.44**	91.06
he	-	76.67	75.75	75.43	**76.73**$^{\$\dagger}$
cs	-	93.89	93.04	93.94	**94.81**$^{*\$\dagger}$
sk	83.6	95.68	95.62	95.53	**95.81**†
sr	-	97.55	97.47	97.43	**97.58**†
id	-	84.10	85.23	85.50	**85.97**$^{*\$}$
fi	-	**87.13**	86.89	86.86	87.05
ko	-	63.59	**64.27**	63.77	64.19
hi	-	81.49	80.27	79.94	**82.41**$^{*\$\dagger}$
ja	-	66.86	65.99	67.71	**67.81**$^{*\$}$
fa	72.8	81.03	80.69	**82.37**	81.79
Average	-	82.98	83.05	83.15	**83.66**

Table 2.2: Performance on universal POS tagging when using each of language as the target language and the rest as source languages . '*', '\$', '†' denote SEAL-Shap model is statistically significantly outperforms *All Sources*, *Baseline-r* and *Baseline-s* respectively using paired bootstrap test with p $\leq$ 0.05. "en" refers to the only source ("en") results in Wu and Dredze (2019).

Next, as in Sec 2.2.5, we tune a threshold θ and either select all the sources as useful or a smaller subset of m number of sources (i.e., $m < |\mathcal{D}|$) whose SEAL-Shap values are higher than θ. In the followings, we compare the model performances of these m sources selected by SEAL-Shap with the same top-m sources ranked by the aforementioned baseline methods. Being relatively weak or slow, we do not further report performances for *LOO*, *Lang-Dist*, and *Greedy DFS*. Rather we consider another strong baseline *All Sources* that uses all the source corpora $\mathcal{D}$. This is a strong baseline as it is trained on more source-corpus instances in general.

Cross-Lingual POS Tagging We evaluate the source selection results on zero-shot cross-lingual POS tagging in Table 2.2. Among the 31 target languages, in 21 of them,

Model	**WSJ**	**EM**	**N**	**A**	**R**	**WB**	**Avg**
MMD	96.12	96.23	96.40	95.75	95.51	96.95	96.16
RENYI	96.35	96.31	96.62	95.52	95.97	96.75	96.25
All Sources	95.95	95.39	96.94	95.15	96.08	97.10	96.10
Baseline-r	95.98	93.41	93.78	93.14	95.25	97.10	94.78
SEAL-Shap	**96.14**$^{*\$}$	**95.47**$^{\$}$	**97.02**$^{\$}$	**95.30**$^{*\$}$	**96.17**$^{\$}$	97.10	**96.20**

Table 2.3: POS tagging results (% accuracy) on SANCL 2012 Shared Task. '*' and '$' denote the model using SEAL-Shap statistically significantly outperforms *All Sources* and *Baseline-r* respectively using paired bootstrap test with $p \leq 0.05$. MMD, and RENYI refer to Liu et al. (2019a) which use auxiliary unlabelled data in the target domain and focus on instance selection. *Baseline-s* has exactly same results as SEAL-Shap.

Model	**bg**	**ru**	**tr**	**ar**	**vi**	**hi**	**sw**	**ur**	**Avg**
XLM-MLM	74.0	73.1	67.8	68.5	71.2	65.7	64.6	63.4	68.54
mBERT(en)	68.9	69.0	61.6	64.9	69.5	60.0	50.4	58.0	62.79
All Sources	74.03	73.59	65.21	68.94	74.39	67.31	52.67	64.37	67.56
Baseline-r	74.69	**74.53**	**65.85**	68.68	75.03	66.69	**52.97**	63.69	67.77
Baseline-s	73.23	73.73	65.67	68.36	74.11	67.07	52.59	63.31	67.26
Ours	**74.95**	73.85	65.63	**69.24**	**75.71**	**67.78**	52.73	**64.67**	**68.07**

Table 2.4: XNLI results. As a reference, we include two results from the recently published papers mBERT (Wu and Dredze, 2019) and "XLM-MLM" (Lample and Conneau, 2019). mBERT is trained on "en" only and "XLM-MLM" is applicable to XNLI languages only.

SEAL-Shap selects a small subset of source corpora. From the Table, overall, SEAL-Shap selects source corpora with high usefulness for training the model, and except for few cases the model constantly outperforms all the baselines by more than 0.5% in avg token accuracy. In 13 of them, it is statistically significant by a paired bootstrap test. The gap is especially high for English, Czech, and Hindi. These results demonstrate that SEAL-Shap is capable in both quantifying the source values and also in source selection.

Cross-Domain POS Tagging Table 2.3 presents the POS tagging results in zero-shot domain transfer on SANCL 2012 shared task. In 5 out of 6 targets, SEAL-Shap outperforms all baselines except *Baseline-s*. For each target domain with only 5 sources, *Baseline-s* source values match with ours in general. However, SEAL-Shap significantly outperforms *Baseline-r* on all 5 cases and *All-Sources* twice. It even outperforms MMD,

Model	books	kitchen	dvd	baby	MR	Avg
Cai and Wan (2019)	87.3	88.3	88.8	90.3	76.3	86.2
All Sources	**87.3**	90.3	88.3	92.3	79.3	87.5
Baseline-r	87.0	90.5	87.3	91.8	78.8	87.1
Baseline-s	86.8	89.8	87.0	**92.5**	77.5	86.7
SEAL-Shap	**87.3**	**90.8**	**88.8**	**92.5**	**79.5**	**87.8**

Table 2.5: Cross-domain transfer results on multi-domain sentiment analysis task. Cai and Wan (2019) use unlabelled data from the target domain.

Model	SNLI	QQP	QNLI	MNLI-mm	Avg
Ma et al. (2019)	88.30	73.90	59.10	-	76.23
All Sources	88.69	72.96	50.65	89.47	75.45
Baseline-r	88.11	72.71	50.53	89.18	75.13
Baseline-s	**88.72**	**73.47**	50.98	**89.69**	75.72
SEAL-Shap	**88.72**	**73.47**	**54.75**	**89.69**	**76.66**

Table 2.6: Zero-shot results on modified GLUE. Ma et al. (2019) selects instances from one source domain at once while we select a subset of source corpora.

and RENYI (Liu et al., 2019a) on Newsgroups (N), Reviews (R), and Weblogs (WB) despite they select source data at instance level and use additional resources.

Cross-Lingual NLI In Table 2.4, we show the XNLI results in 8 target languages where SEAL-Shap selects a small subset of source corpora. Among them, in 3 languages, *Baseline-r* marginally surpasses ours. However, in 5 other languages SEAL-Shap outperforms all the baselines with clear margin specially on Bulgarian, Vietnamese with about 1% better accuracy.

Cross-Domain NLI Next, we evaluate SEAL-Shap on the modified GLUE dataset in Table 2.6. SEAL-Shap outperforms *Baseline-s* once and other baselines in all cases. Its highest performance improvement is gained on QNLI, where it outperforms others by 4%.

Cross-Domain Sentiment Analysis Among the 13 target domains in the multi-domain sentiment analysis dataset, in 5 domains SEAL-Shap selects a small subset. As in Table 2.5), with a large margin, SEAL-Shap achieves higher accuracy than all other baselines and, in 4 cases, it is even better than Cai and Wan (2019) that uses unlabeled target data.

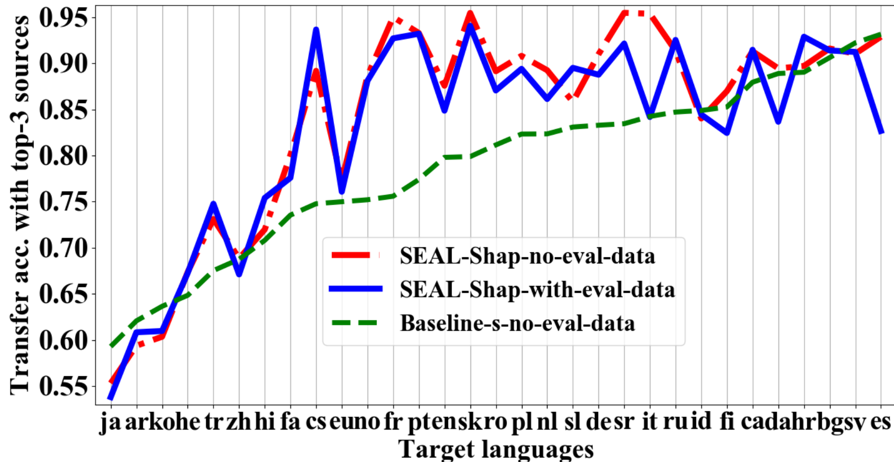


Figure 2.3: Cross-lingual POS tagging accuracies on different target languages using top-3 sources ranked by SEAL-Shap. The ranker (red) selects similar sources as using SEAL-Shap with annotated target data (blue). Ranker trained to predict SEAL-Shap values (red) performs better than baseline (green) (Lin et al., 2019).

Our experimental evidences show that SEAL-Shap is an effective tool in choosing useful transfer sources and can achieve higher transfer performances than other source valuation approaches.

2.4.2 Results without an Evaluation Corpus

We evaluate the effectiveness of SEAL-Shap to build a straightforward ranker that directly computes the source values without any evaluation target corpus (see Sec 2.2.4). We use the ranker in Lin et al. (2019) as the underlying ranking model. First, we show that the source values evaluated by the ranker is as good as SEAL-Shap that uses its annotated target dataset. We compare the transfer performances of the top-k sources based on the source values computed with and without the evaluation corpus. Then, we show that the ranker trained with SEAL-Shap is more effective than training it with the existing single source based *Baseline-s*.

In cross-lingual POS tagging on UD Treebank, for each of the 31 target languages, we set aside that language and consider the remaining 30 languages as the training corpora. We then train the ranker as described in Sec 2.2.4 and compute the source values using it. As for reference, we pass the evaluation target dataset and the 30 source languages

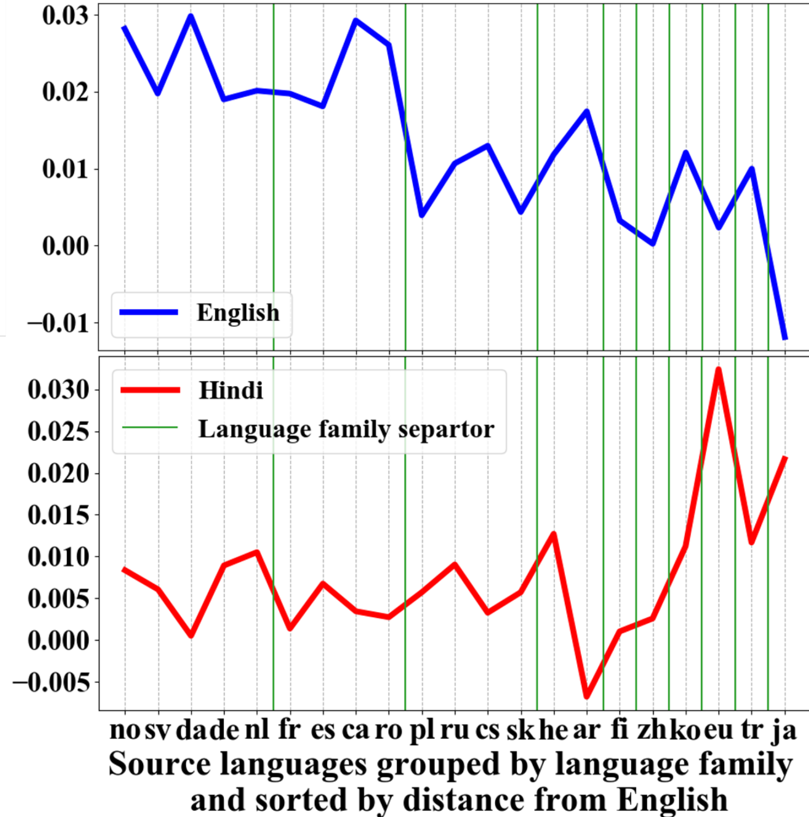
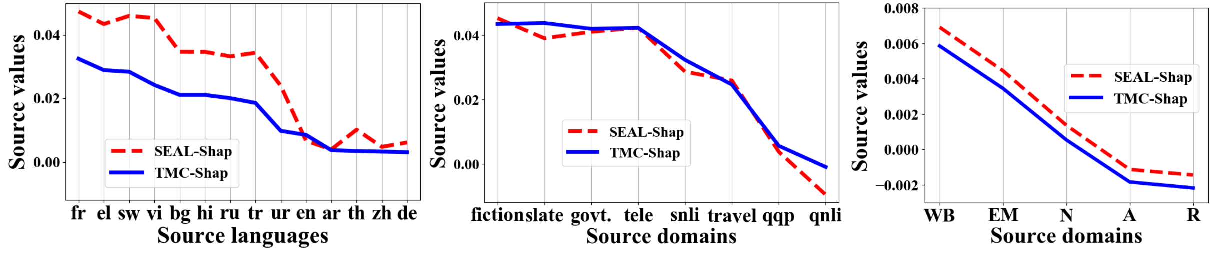


Figure 2.4: Cross-lingual POS tagging SEAL-Shap values, referring to the relative contribution of the source languages.

to SEAL-Shap to compute their values on the evaluation dataset. With $k = 3$, we compare the transfer results of the top-k sources of these two methods in Fig 2.3. We also plot the results of the baseline ranker (Lin et al., 2019) that is trained with *Baseline-s*. Results show that the ranker source values are similar to the sources values estimated by SEAL-Shap with an annotated evaluation dataset and also it outperforms the baseline.

2.4.3 Interpret Source Value by SEAL-Shap

In this Section, we show that SEAL-Shap values provide a means to understand the usefulness of the transfer sources in cross-lingual and cross-domain transfer. We first analyze cross-lingual POS tagging. Following Ahmad et al. (2019), we consider using language family and word-order distance as a reference distance metric. We anticipate that languages in the same language family with smaller word-order distance from the



((a)) XNLI, target: 'es', R <10%

((b)) mGLUE, target: MNLI-mm, R=10-20%

((c)) SANCL'12, target: wsj, R ∼50%

Figure 2.5: Source values by TMC-Shap and ours. TMC-Shap uses unbalanced full source corpora whereas SEAL-Shap that achieves similar source values uses balanced and sampled source corpora. Even with a small sample rate (R), source order is almost same. Higher sampling rate typically refers to better approximation but leads to expensive runtime. In general, for a reasonably large corpus, 20-30% samples (>few thousands) are found sufficient to achieve reasonable approximation.

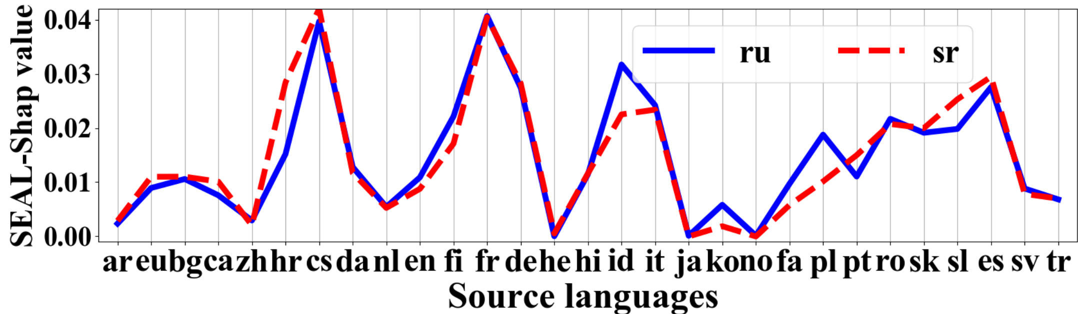


Figure 2.6: Similar SEAL-Shap value curves for two closely related target languages in cross-lingual POS tagging.

target language are more valuable in multi-lingual transfer. We plot SEAL-Shap of source languages evaluated on two target languages English ("en") and Hindi ("hi") in Fig 2.4. In the x-axis, a common set of twenty different source languages are grouped into ten different language families and sorted based on the word order distance from English. As the figure illustrates, Germanic and Romance languages have higher Shapley values when using English as the target language. The value gradually decreases for language of other families when the word order distance increase. As for the target language Hindi, the trend is opposite, in general.

Analogously, as in Figure 2.8, for cross-domain NLI, we find that correlation between QNLI, and QQP is high whereas between MNLI-mm and QQP, it is lower.

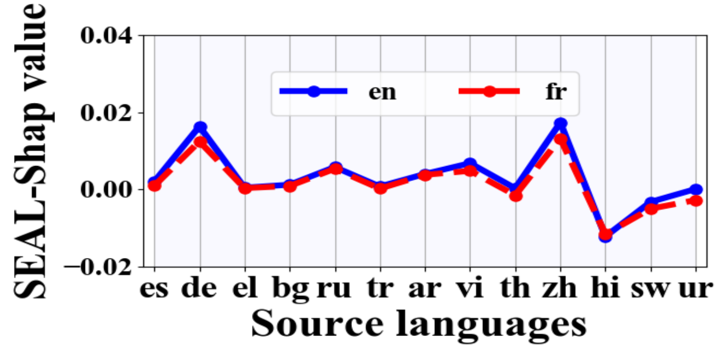


Figure 2.7: Similar SEAL-Shap value curves for two closely related XNLI targets “en” and “fr”. In XNLI, the source corpora are prepared by machine translating from “en”. This data processing may affect the source values. Translation into “zh” being relatively better, although different from both targets, its source values are higher than others.

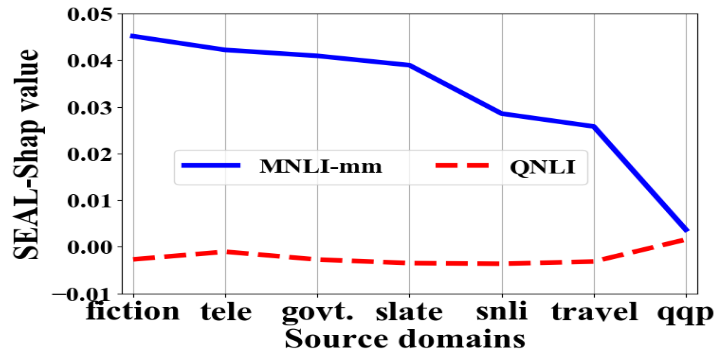


Figure 2.8: SEAL-Shap value on cross-domain NLI, referring to relative contribution of source domains. For target domain MNLI-mm, source domain QQP has the lowest contribution, whereas for target domain QNLI, source domain QQP has the highest contribution.

SEAL-Shap on Similar Targets Intuitively, if two target corpora are similar, the corresponding Shapley values of the source corpora when transferring to these two targets should be similar as well. To verify, in Fig 2.6, we plot the Shapley values of twenty nine source languages for targets Russian and Serbian on cross-lingual POS tagging. Also we plot the source values when transferring a NLI model to English and French in Fig 2.7. We observe that the corresponding curves are almost identical, and SEAL-Shap in fact selects the same set of source corpora as potential. These results suggest that if there is no sufficient data in the target corpus, it is also possible to use a neighboring corpus as a proxy to compute SEAL-Shap values.

Prob.	Transfer	Target	#Targets	#Samples	Caching	Time (hours)
NLI	**Domain**	**MNLI-mm**	1	✗	✗	300*
			1	✗	✓	101
			1	20k	✓	18
			3	20k	✓	5
POS	**Language**	**Arabic (ar)**	1	✗	✗	210*
			1	✗	✓	180*
			1	3.3k	✓	25
			31	3.3k	✓	3.5

Table 2.7: Running time for computing approximate Shapley value. The marker *represents the time is estimated by extrapolation. #Targets indicates number of target corpus evaluated simultaneously. #Samples is the number of samples used to train model for computing marginal contribution. TMC-Shap is equivalent to disable all the techniques (the first row of each block).

Source Values Influenced by Data Processing Typically, the sources with least or negative source values are from the domains/languages that are different from the targets (e.g., Fig 2.4). However, in some cases, source usefulness (i.e., values) is affected by the data preparing process. For example, in XLNI, the source corpora are prepared by machine translation from "en" (Conneau et al., 2018) and the quality of this translation into "zh" is better in compare to other languages, in general. Consequently, in Fig 2.7, "zh" has higher source value for both targets "en" and "fr".

2.4.4 Analysis and Ablation Study

Finally, we analyze the proposed Algorithm 3 for computing Shapley value approximately.

How good is the approximation? In Fig 2.5, we compare SEAL-Shap with TMC-Shap (Ghorbani and Zou, 2019) on three datasets Overall, the Shapley values obtained by SEAL-Shap and TMC-Shap are highly correlated and their relative orders are matched, while SEAL-Shap is much more efficient. Note that, the rankings themselves being same/similar, the model performances using the same/similar top-k sources are same/similar, too; therefore, we do not list their transfer performances furthermore.

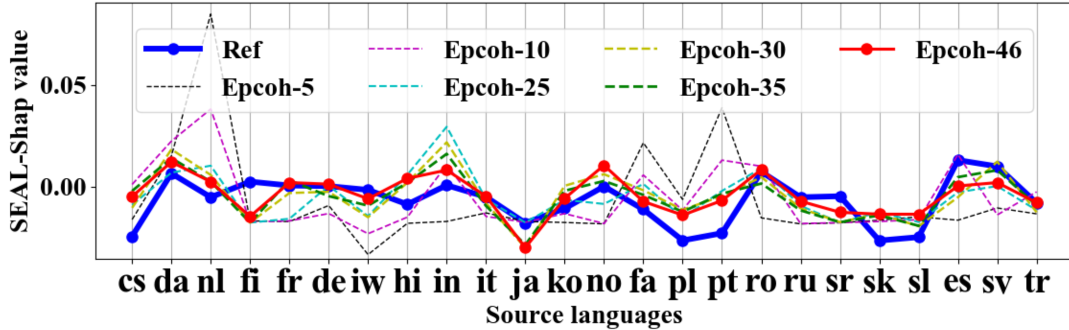


Figure 2.9: SEAL-Shap value with two (colored) seeds.

Ablation Study: We examine the effectiveness of each proposed components in SEAL-Shap. Results are shown in Table 2.7. Results show that without the proposed approximation, TMC-Shap is computational costly and is impractical to use to analyze the value of source corpus in the NLP transfer setting. All the proposed components contribute to significantly speed-up the computations.

Shapley Value Computation Time without different Factors: We consider two example problem to transfer both language and domain: (i) UDPOS tagging for language transfer (ii) modified GLUE NLI for domain transfer. We consider the "initial score" to *All Sources/2* and $\mathcal{R}$; *nepoch* to 30, and 50 for these two respective target task, for the data Shapley computation as in Algorithm 1, we then switch different factors as in reported in Table 6 (in the main paper) as record the corresponding Shapley value computation time.

Is the approximation sensitive to the order of permutations? As SEAL-Shap is a Monte Carlo approximation, we study if SEAL-Shap is sensitive to the random seed using the cross-lingual POS tagging task. To analyze, we first compute a reference Shapley values by running SEAL-Shap until empirically convergence (blue line). Then, we report the Shapley value produced by another random seed. Fig 2.9 shows that with enough epochs, the values computed by different random seeds are highly correlated.

Number of Sources Selected: Below, Table 2.8 shows out of the 30 source languages, how many of them the are selected as potential sources by SEAL-Shap for each target language. For the remaining targets, SEAL-Shap selects 27 source languages as potential.

Target Lang.	#Sources Selected
en	9
fr	29
da	29
it	26
ca	25
sl	29
nl	28
de	28
he	29
id	26
ar	30
ja	29

Table 2.8: Number of sources selected from 30 different languages by SEAL-Shap for the task of cross-lingual POS tagging. For the remaining 18 target languages, SEAL-Shap selects 27 source languages as potential.

2.5 Related Work

As discussed in Section 5.1, transfer learning has been extensively studied in NLP to improve model performance in low-resource domains and languages. In the litearture, various approaches have been proposed to various tasks, including text classification Zhou et al. (2016); Kim et al. (2017), natural language inference Lample et al. (2018); Artetxe and Schwenk (2019), sequence tagging Täckström et al. (2013); Agić et al. (2016); Kim et al. (2017); Ruder and Plank (2017), dependency parsing Guo et al. (2015); Meng et al. (2019). These prior studies mostly focus on bridging the domain gap between sources and targets.

In different contexts, methods including influence functions and Shapley values have been applied to value the contribution of training data Koh and Liang (2017); Lundberg et al. (2018); Jia et al. (2019a). Specifically, Monte Carlo approximation of Shapley values has been used in various applications Maleki (2015); Jia et al. (2019a); Ghorbani and Zou (2020); Tripathi et al. (2020); Tang et al. (2020); Sundararajan and Najmi (2019). However they are either task/model specific or not scalable to NLP applications. Oppositely, Kumar et al. (2020a) discuss the problems of using Shapley value for model

explanation. In contrast, we apply efficient Shapley value approximation in NLP transfer learning and analyze the source-target relationships.

2.6 Summary

In this work, we propose Shapley value as a metric to quantify the usefulness of a source corpora. We develop a framework SEAL-Shap to approximate the Shapely value efficiently and to select the potential sources. The selected corpora are merged together (*Type-1* ***corpora level***) to create the a training set for the the transfer model. We conduct extensive sets of experiments on three text classification tasks (natural language inference (NLI), POS tagging, and sentiment analysis), two zero-shot transfer learning settings (domain and language) and two notable families of neural architectures (LSTM v.s. transformers). We posit that the auxiliary supervision from the selected transfer sources significantly enhances these applications w/ achieving several state-of-the-art performances. Shapely values also makes the source-target correlation interpretable.

However, a possible limitation of this approach could be that it may be applicable to only corpora level and may be extended to instance level. The possible reasons could be two folds:

- #sources will be extremely high and so does become the number model retraining
- marginal contribution would be negligible and numerically unstable if each element is an instance instead of a corpora.

Therefore a new retrieval system may be necessary for instance-level candidate ranking and selection.

CHAPTER 3

Retrieving and Incorporating Relevant Code and Summary for Code Generation and Summarization

In the previous Chapter, we developed a framework for selecting corpora and discussed its limitations for retrieving example/instance level candidates. In this Chapter, we will develop such a retriever model and enhance a downstream task. We take the tasks of text to code generation and code to summarization as two example multi-modal downstream tasks which intrinsically mimics low-resource scenarios and enhance them w/ the additional hints/features retrieved by our retriever model (i.e., *Type-2* ***feature level*** auxiliary supervision).

3.1 Introduction

In recent years, automating source code generation and summarization is receiving significant attention due to its potential in increasing programmers' productivity and reducing developers' tedious workload. Consequently, various approaches have been explored in the literature to facilitate code generation (Yin and Neubig, 2017a; Gu et al., 2016b) and code documentation/summarization (Ahmad et al., 2020a; Wei et al., 2019; Allamanis et al., 2018). Despite initial success, most of the generated code still suffers from poor code quality (Xu et al., 2021). Therefore, the question remains—how to generate better code from a given summary and vice versa.

Source code generation and summarization, however, are intrinsically complex and challenging. They involve generating diverse token sequences such as different variables, operators, keywords, classes, and method names (Parvez et al., 2018), which requires

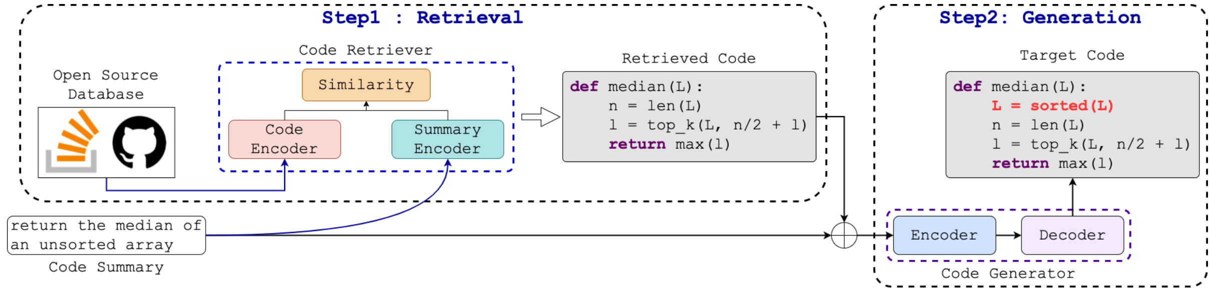


Figure 3.1: Illustration of our proposed framework REDCODER for code generation. Given an input summary, we first retrieve top-k candidate code (k=1 in this example). We then aggregate them and based on that a *generator* module generates the target sequence.

understanding the programming languages at lexical, syntax, and semantics levels. To combat these issues, recent studies (*e.g.,* Ahmad et al. (2021a); Guo et al. (2021); Xu et al. (2020); Feng et al. (2020a); Xu et al. (2020)) take a learning-based approach—they train representations of code and the associated text by leveraging existing high-quality source code and short text descriptions available in open-source repositories and question answering forums such as GitHub and Stack Overflow. Then fine-tune the representation models on the downstream tasks. Although these dataset contains high-quality human-written code and text, since the existing approaches do not directly leverage them during the generation process, the gain achieved by these approaches is still limited, especially when the source code is long.

To overcome this, we take advantage of the existing high-quality source code and their description by including them directly in the generation process that are retrieved via information retrieval technique. In this work, we present REDCODER, a **R**etrieval augment**ED COD**e g**E**neration and summa**R**ization framework. While designing REDCODER, we take motivation from how developers take advantage of existing resources. For example, developers often search for relevant code in the code repository, and if found, adapt the retrieved code in their own context. Similarly, when an API usage is unclear, they search in question answering forums (*e.g.,* StackOverflow) (Brandt et al., 2010; Sadowski et al., 2015). Such an additional resource helps developers to increase their development productivity (Li et al., 2013).

We design REDCODER as a two-step process (see Figure 3.1). In the first step, given the input (*nl* text for code generation, or *code snippet* for summarization) a *retriever* module retrieves relevant source code (for code generation) or summaries (for code summarization) from a database.[1] In the second step, a *generator* processes the retrieved code/summary along with the original input to generate the target output. In this way, REDCODER enhances the generation capability by augmenting the input through retrieval. The two-step process allows us to design a modular and configurable framework for source code and summary generation. Various designs of retriever and generator models can be incorporated into this framework.

Existing cross-encoder code retrievers being computationally expensive, their applicability to retrieve from a large database is limited (Humeau et al., 2020). A natural choice would be to use sparse term based retrievers such as TF-IDF or BM25 (Robertson and Zaragoza, 2009). However, the *retriever* module in REDCODER should exhibit a good understanding of source code and programmers' natural language, which is a non-trivial task due to the syntactic and semantic structure of the source code (Guo et al., 2021; Ahmad et al., 2021a). Such an expectation of searching for semantically similar code and summary may not be attainable by a sparse token level code retriever (*e.g.,* BM25). To that end, we design the *retriever* module in REDCODER based on programming languages (PL) and natural languages (NL) understanding models (*e.g.,* GraphCodeBERT (Guo et al., 2021)). This *retriever* module extends the state-of-the-art dense retrieval technique (Karpukhin et al., 2020a) using two different encoders for encoding the query and document.

As for the *generator*, REDCODER can handle retrieval databases consisting of both unimodal (only code or natural language description) and bi-modal instances (code-description pairs) and makes the best usage of all the auxiliary information that are available. Yet, to incorporate information, we augment the retrieved information only in the input level. It does not modify the underlying architecture of the *generator* module —preserving its model agnostic characteristics.

[1] The database could be open source repositories (*e.g.,* GitHub) or developers' forums (*e.g.,* Stack Overflow).

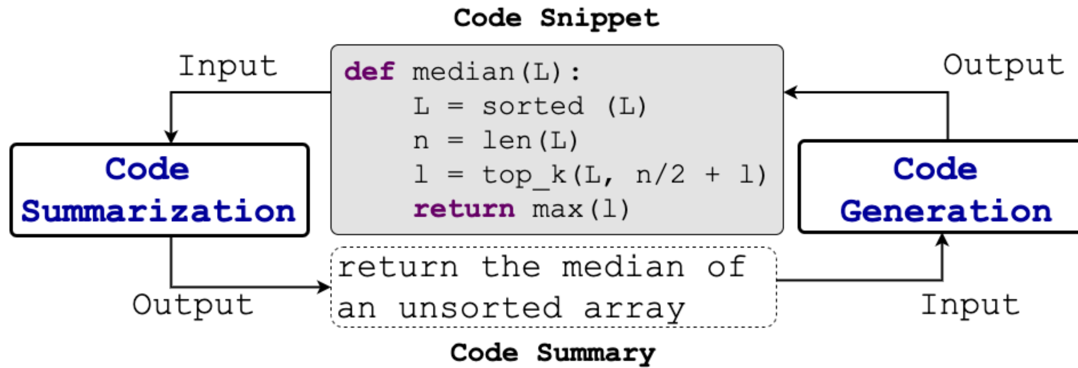


Figure 3.2: Example input/output for the code generation and summarization tasks.

We evaluate the effectiveness of REDCODER on two popular programming languages (Java and Python) on both code generation and code summarization tasks. The empirical results show that, REDCODER's concept of *retrieval augmented generation* elevates the state-of-the-art code generation from an Exact Match score of 18.6 to 23.4 and the summary generation BLEU-4 score from 18.45 to 22.95 even when we forcefully remove the target candidate from the retrieved code or summary. With further experiments, we establish the importance of both the retrieved code and retrieves summary in the generation process. The source code for reproducing our experiments are at https://github.com/rizwan09/REDCODER.

3.2 Background

We first introduce the problem formulation and discuss the fundamentals of the *retriever* and *generator* components that REDCODER is built upon.

3.2.1 Problem Formulation

Our goal is two folds: (i) code generation: Generating source code (C), given their natural language description, such as code summaries, code comments or code intents (S); (ii) code summarization: Generating natural language summaries S, given source code snippets C. Fig 3.2 shows an example.

Let X and Y denote a collection of input and output sequences ($X = S_1, \ldots, S_n$, $Y = C_1, \ldots, C_n$ in code generation, $X = C_1, \ldots, C_n$, $Y = S_1, \ldots, S_n$ in summary generation). We assume that we have access to a retrieval database consisting of an extensive collection of source code (*e.g.,* aggregated from GitHub or Stack Overflow) or summaries (*e.g.,* docstrings, code comments) (Y_R). Note that, target sequences (Y) may or may not be present in the retrieval database (Y_R). Now, given an input $x \in X$, a *retriever* retrieves the top-k relevant output sequences from the database: $\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_k \in Y_R$. Then the input sequence x is augmented with the retrieved sequences to form $x' = x \oplus \mathcal{Y}_1 \oplus \mathcal{Y}_2 \ldots \oplus \mathcal{Y}_k$, where $\oplus$ denote the concatenation operation. Finally, a *generator* generates the target output $y \in Y$ given x'. In the following, we first discuss the base *retriever* and *generator* modules used in REDCODER and then how we improve these components is in Section 5.2.

3.2.2 Retriever: DPR

Information retrieval (IR) systems or retriever models are designed to retrieve the top-k relevant documents that presumably best provide the desired information (Manning et al., 2008). Term-based retrieval methods, *a.k.a.* sparse retrieval models, such as TF-IDF or BM25 (Robertson and Zaragoza, 2009) use sparse vector representations to perform lexical matching and compute relevance scores to rank the documents based on a query.

On the other hand, dense retrieval methods encode documents into a fixed-size representations and retrieve documents via maximum inner product search (Sutskever et al., 2014; Guo et al., 2016). Particularly of interests, Karpukhin et al. (2020a) propose a Dense Passage Retriever (DPR) model for open-domain question answering (QA). It consists of two encoders ($Q(.)$ and $P(.)$) that encode queries and passages, respectively. The similarity of a query q and a passage p is defined by the inner product of their encoded vectors $sim(p, q) = Q(q)^T \cdot P(p)$. Given a query q, a positive (relevant) passage p^+, and a set of n irrelevant passages p_i^-, DPR optimizes the classification loss:

$$L = -\log \frac{e^{sim(q,p^+)}}{e^{sim(q,p^+)} + \sum_{i=1}^{n} e^{sim(q,p_i^-)}}.$$

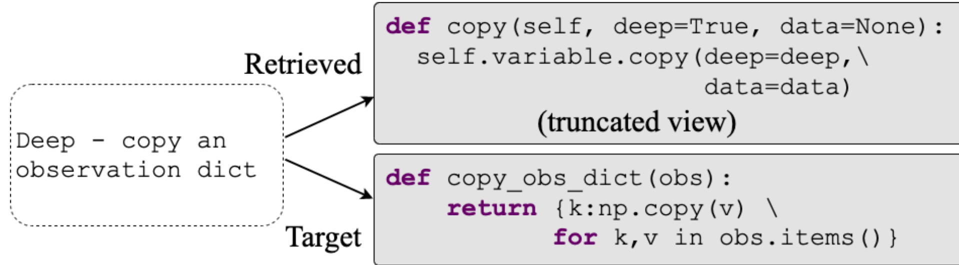


Figure 3.3: An example retrieved code that is relevant yet does not match the reference.

Karpukhin et al. (2020a) propose to fine-tune DPR using *in-batch negatives* (Gillick et al., 2019; Yih et al., 2011) with curated "hard" negatives using BM25 (candidates with high BM25 scores but contain no sub-string that match the target). We refer to Karpukhin et al. (2020a) for details.

3.2.3 Generator: PLBART

PLBART (Ahmad et al., 2021a) is a sequence-to-sequence Transformer model (Vaswani et al., 2017) that is pre-trained on a huge collection of source code and natural language descriptions via denoising autoencoding. PLBART has shown promise in several software engineering applications, including code generation and summarization. We adopt PLBART as the generator module in our proposed framework, REDCODER.

3.3 Proposed Framework: REDCODER

Our proposed code generation and summarization framework, REDCODER generates the target code or summary by augmenting the input x with relevant code snippets or summaries. We build our *retriever* module by training a DPR model differently from (Karpukhin et al., 2020a). With an intelligent scheme, we then augment the retrieved candidates and their pairs (if available) to provide auxiliary supervision to the *generator*. We briefly describe the model components in this section.

3.3.1 Retriever: SCODE-R

Architecture The *retriever* module of REDCODER is built upon the DPR model (Karpukhin et al., 2020a) and we call it SCODE-R (Summary and CODE Retriever). SCODE-R composed of two encoders that encode source code and natural language summary. We use bidirectional Transformer encoders (Vaswani et al., 2017) that are pre-trained on source code and natural language summaries. Specifically, we explore CodeBERT (Feng et al., 2020b) and GraphCodeBERT (Guo et al., 2021) as the code and summary encoders for SCODE-R.

Input/Output SCODE-R takes an input sequence x (code or summary) and retrieves a set of relevant documents from a database of output sequences Y (if the input is code, then the output is summary and vice versa). SCODE-R returns the the top-k output sequences $\{\mathcal{Y}_1, \mathcal{Y}_2, \ldots, \mathcal{Y}_k\}$, where $sim(x, \mathcal{Y}_i) \geq sim(x, \mathcal{Y}_j) \forall j > i$.

Training We fine-tune SCODE-R using a set of parallel examples (x_i, y_i) of code and summaries. As mentioned in Section 3.2.2, DPR originally proposed to be fine-tuned using *in-batch negatives* and curated "hard" negatives from BM25 retrieved passages for open-domain QA. The key idea behind "hard" negatives is to fine-tune DPR to distinguish the target passage from relevant passages that do not contain the target answer. However, unlike open-domain QA, a retrieved code or summary that is not the target could still benefit code generation or summarization (verified in Section 3.6). We provide an example in Figure 3.3; although the retrieved code does not match the target one but can facilitate generating it. Therefore, we fine-tune SCODE-R without any "hard" negatives. Specifically, for each training instance (x_i, y_i), the corresponding output y_i is considered as positive and the other in-batch outputs (*i.e.,* the outputs of other instances in the same batch - $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{bsz}$) as negatives. Figure 3.4 shows an example of SCODE-R fine-tuning for code generation task.

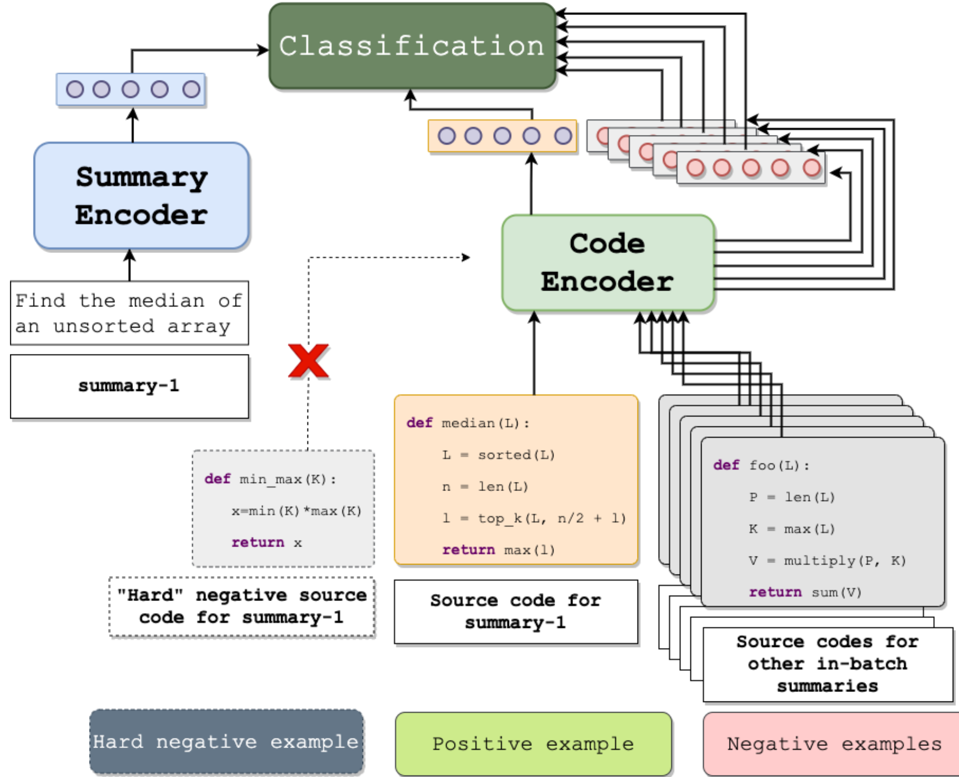


Figure 3.4: Training scheme of the *retriever* module (SCODE-R) of our proposed framework REDCODER for the code generation task. Unlike in open-domain QA (Karpukhin et al., 2020a), we do not use "hard" negatives (*e.g.*, candidates retrieved by BM25 that do not exactly match the reference) during fine-tuning.

3.3.2 Generator: SCODE-G

We adopt PLBART as discussed in Section 3.2.3 as the *generator* module of REDCODER and call it SCODE-G (Summary and CODE Generator). The input sequence x is concatenated with the top-k retrieved sequences to form the augmented input sequence, $x' = x \oplus \mathcal{Y}_1 \oplus \mathcal{Y}_2 \ldots \oplus \mathcal{Y}_k$. The augmented input x' is fed to PLBART to estimate $p_{gen}(y|x')$.

Note that a source code often consists of docstrings, comments that can be extracted to form code – summary pairs. In the retrieval databases, code and summaries are either singleton (*e.g.*, code without a description or a problem statement without any code) or parallel. Therefore, we consider two retrieval settings that require separate modeling consideration for the generator.

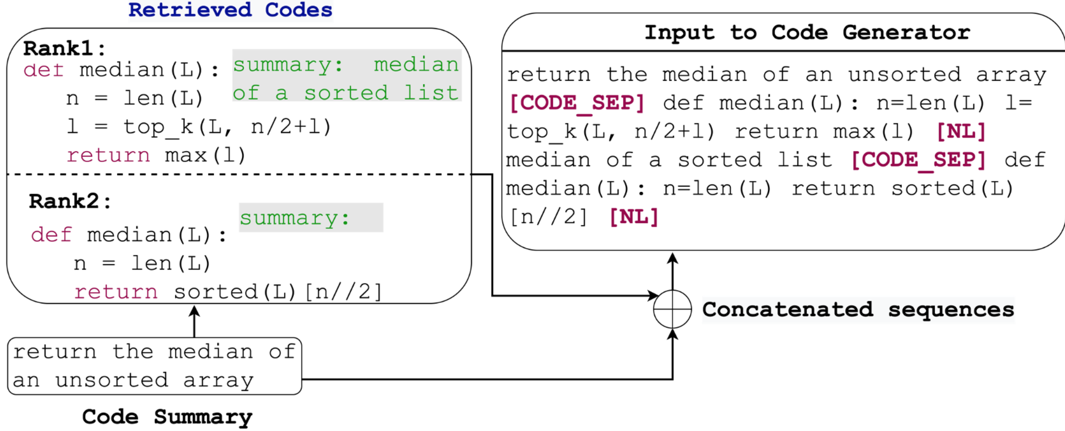


Figure 3.5: REDCODER-EXT input for code generation.

Dataset	Gen.	Sum.	Lang.	Train	Valid	Test	|Code|	|Summary|
CodeXGLUE (Lu et al., 2021)	✓	✓	Java	164,923	5,183	10,955	97	12
			Python	251,820	13,914	14,918	99	14
Concode Iyer et al. (2018)	✓	✗	Java	100,000	2,000	2,000	27	72

Table 3.1: Dataset Statistics. Gen., and Sum. refers to code generation and summarization tasks respectively. Summary denotes a natural language description paired with each code. For Concode, the input summary includes the corresponding environment variables and methods. All lengths are computed and averaged before tokenization.

Case 1: Retrieve candidates are singleton In this case, we concatenate the original input sequence x and the top-k retrieved candidates with a special separator token.

$$x' = x \ [csep] \ \mathcal{Y}_1 \ [csep] \ \mathcal{Y}_2 \ \ldots \ [csep] \ \mathcal{Y}_k.$$

This is our default setting and we refer this as REDCODER in this work.

Case 2: Retrieve candidates are pairs In this case, retrieved candidates are pair of code and natural language (NL) summary. We augment the input sequence using both of them as follows.

$$x' = x \ [csep] \ \mathcal{Y}_1 \ [nsep] \ \mathcal{X}_1 \ [csep] \ \mathcal{Y}_2$$
$$[nsep] \ \mathcal{X}_2 \ \ldots \ [csep] \ \mathcal{Y}_k \ [nsep] \ \mathcal{X}_k,$$

where $\mathcal{X}_j$ and $\mathcal{Y}_j$ are parallel sequences (*e.g.*, $\mathcal{Y}_j$ is a piece of code and $\mathcal{X}_j$ is its corresponding summary for the code generation task) retrieved from the database. We conjecture that

the additional information $\mathcal{X}_j$ complements the input sequence x and verify its effectiveness in the experiments.

Note that retrieve candidates could be a mix of singleton and pairs. In case of a singleton candidate, we simply replace $\mathcal{X}_j$ or $\mathcal{Y}_j$ with an empty string. We refer this setting as REDCODER-EXT. Although, REDCODER-EXT is a more general setting which includes "Case 1", we study them separately to understand how these two retrieval settings benefit the target tasks. We illustrate an example on code generation in Figure 3.5. In both cases, the augmented input x' is truncated to match PLBART's maximum input length 512.

3.4 Experiment Setup

Method		Java			Python		
Type	Name	EM	BLEU	CodeBLEU	EM	BLEU	CodeBLEU
Retrieval Based	BM25	0.00	4.90	16.00	0.00	6.63	13.49
	SCODE-R	0.00	25.34	26.68	0.00	22.75	23.92
Generative	CodeBERT	0.00	8.38	14.52	0.00	4.06	10.42
	GraphCodeBERT	0.00	7.86	14.53	0.00	3.97	10.55
	CodeGPT-adapted	0.00	7.10	14.90	0.01	3.11	11.31
	PLBART	0.00	10.10	14.96	0.00	4.89	12.01
Retrieval Augmented Generative	BM25 + PLBART	0.10	11.37	15.52	0.03	6.99	13.89
	REDCODER	8.95	26.92	31.15	8.88	22.74	28.93
	REDCODER-EXT	**10.21**	**28.98**	**33.18**	**9.61**	**24.43**	**30.21**

Table 3.2: Results on code generation on CodeXGLUE (Lu et al., 2021).

In order to investigate the effectiveness of our framework, we perform a comprehensive study and analysis on code generation and summarization in two programming languages, Java and Python.

3.4.1 Datasets and Implementations

Datasets We perform evaluation on both the tasks using the code summarization dataset from CodeXGLUE (Lu et al., 2021). It is curated from CodeSearchNet (Husain et al.,

2019) by filtering noisy examples. In addition, we conduct code generation experiments in Java using the Concode benchmark (Iyer et al., 2018). The dataset statistics are summarized in Table 3.1.

Retrieval Databases To generate a source code given its natural language description or a summary given the code, our proposed approach REDCODER first retrieves prospective candidates from an existing code or summary database. We form the code retrieval database using the deduplicated source code (on average 1.4M functions in Java and Python) that consists of both paired (59%) and monolingual code, released in CodeSearchNET (Husain et al., 2019). As for building the summary retrieval database, we extract the high quality natural language summaries from the paired instances in the training sets of CodeSearchNET. As many of the summaries are duplicated, we also consider the training sets in the other four available languages Ruby, Javascript, Go, and PHP. We then further enlarge it by aggregating the additional summaries from the CCSD corpus (Liu et al., 2021). After performing deduplication, we retain 1.1M unique code summaries and for evaluating REDCODER-EXT, 20% of them can be used as pairs with the corresponding Java and Python source code. We provide the statistics of the retrieval databases in Appendix. Note that the retrieval databases contain code and summaries that are curated from real developers' open sourced repositories on GitHub. By default, we exclude the target code/summary from the retrieval database.

Implementations As mentioned in Section 5.2, REDCODER has two disjoint components. First, the dense retriever SCODE-R is implemented adopting DPR (Karpukhin et al., 2020a) and the encoders in DPR are initialized from GrpahCodeBERT available in the Huggingface API (Wolf et al., 2020). In addition, we implement a baseline BM25 retriever. We use the official codebase of PLBART (Ahmad et al., 2021a) and set max epoch to 15, patience to 5, learning rate to 2×10^{-5}. We tune the batch size in {8, 16, 32, 64, 72} and the k value for top-k retrieval up to 10 for code generation and in range {10, 30, 50, 100} for code summarization. As some candidate code and summaries are short in length, we tune with this upper bound of k to accommodate as many candidates as possible within PLBART's maximum input length.

Methods	EM	BLEU	CodeBLEU
Retrieval based methods			
BM25	0.0	20.3	23.7
SCODE-R	0.0	32.6	36.5
Generative methods			
Seq2Seq	3.1	21.3	26.4
Guo et al. (2019)	10.1	24.4	29.5
Iyer et al. (2019)	12.2	26.6	-
GPT-2	17.4	25.4	29.7
CodeGPT-2	18.3	28.7	32.7
CodeGPT-adapted	20.1	32.8	36.0
CodeBERT	18.0	28.7	31.4
GraphCodeBERT	18.7	33.4	35.9
PLBART	18.6	36.7	38.5
Retrieval augmented generative methods			
BM25+PLBART	21.4	40.2	41.8
REDCODER	**23.4**	41.6	**43.4**
REDCODER-EXT	23.3	**42.5**	**43.4**

Table 3.3: Code generation results on Concode dataset. SCODE-R was initialized with CodeBERT. GraphCodeBERT initialized results are similar.

3.4.2 Evaluation Metrics

BLEU Following prior works (Ahmad et al., 2021a; Feng et al., 2020a), we compute the corpus level BLEU (Papineni et al., 2002) and the smoothed BLEU-4 (Lin and Och, 2004) scores for code generation and summarization tasks.

CodeBLEU To demonstrate syntactic and semantic data flow correctness of code generation models, we report CodeBLEU (Ren et al., 2020). CodeBLEU is a weighted average of lexical, abstract syntax tree, and data flow match.

Exact Match (EM) indicates the percentage of output sequences that exactly match the references.

3.4.3 Baseline Methods

We compare REDCODER *w.r.t.* a number of state-of-the-art code models. We classify them into two categories: (i) retrieval based models and (ii) generative models. We study

Methods	Python	Java
Retrieval based methods		
BM25	1.92	1.82
SCODE-R	14.98	15.87
Generative methods		
Seq2Seq	15.93	15.09
Transformer	15.81	16.26
RoBERTa	18.14	16.47
CodeBERT	19.06	17.65
GraphCodeBERT	17.98	17.85
PLBART	19.30	18.45
Retrieval augmented generative methods		
BM25 + PLBART	19.57	19.71
REDCODER	**21.01**	22.94
REDCODER-EXT	20.91	**22.95**

Table 3.4: Evaluation BLEU-4 score for code summarization on CodeXGLUE. Baseline results are reported from Ahmad et al. (2021a).

Methods	CodeXGLUE (Java)			CodeXGLUE (Python)			Concode (Java)		
	BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU
SCODE-R	36.6	21.0	37.9	35.6	19.2	35.1	70.3	61.7	72.0
REDCODER	36.3	29.4	41.4	32.1	27.5	38.0	76.7	67.5	76.5
REDCODER-EXT	**42.8**	**37.0**	**47.3**	**38.9**	**34.5**	**43.8**	**81.7**	**76.2**	**81.7**

Table 3.5: Results on code generation keeping the target code in the retrieval database.

both generative models that are trained from scratch and are pre-trained on programming and natural languages.

3.4.3.1 Retrieval based models

We examine two retriever baselines and consider the top-1 retrieved candidate as the prediction.

• **Dense Retriever** We consider DPR as the dense retriever baseline. We evaluate both the officially released models trained on the natural language open-domain QA task and a variant called DPR (code) that we fine-tune on the evaluation datasets.

• **Sparse Retriever** The second baseline is a sparse retriever that uses the BM25 algorithm to compute relevance scores.

Settings	Methods	Python	Java
Cross-Encoder	RoBERTa	0.587	0.599
	RoBERTa (code)	0.610	0.620
	CodeBERT	0.672	0.676
	GraphCodeBERT	**0.692**	**0.691**
Bi-Encoder	DPR	0.093	0.064
	DPR (code)	0.398	0.462
	SCODE-R	0.690	0.686

Table 3.6: MRR results on code retrieval from the validation and test set in CodeXGLUE. Our bi-encoder retriever SCODE-R is comparable with other cross-encoder models while it is much faster. DPR refers to Karpukhin et al. (2020a) and DPR (code) is trained with BM25 “hard” negative training schema built upon our source code datasets.

3.4.3.2 Generative models

The generative models work in a sequence-to-sequence (Seq2Seq) fashion.

• **RoBERTa, RoBERTa (code)** RoBERTa models (Liu et al., 2019c) pre-trained on natural language corpora, and source code from CodeSearchNet (Husain et al., 2019) respectively.

• **CodeBERT** (Feng et al., 2020a) is pretrained with a hybrid objective incorporating masked language modeling (Devlin et al., 2018) and replaced token detection (Clark et al., 2020).

• **GraphCodeBERT** (Guo et al., 2021) is pre-trained by modeling the data flow graph of source code. GraphCodeBERT holds the state-of-the-art results on code search using CodeSearchNet.

• **GPT-2, CodeGPT-2, and CodeGPT-adapted** are GPT-style models that are pre-trained on natural language (Radford et al., 2019) and code corpora CodeXGLUE (Lu et al., 2021).

• **PLBART** (Ahmad et al., 2021a) is the generator module of our proposed framework.

In addition, we train an LSTM based Seq2Seq model with attention mechanism (Luong et al., 2015b) and a Transformer model (Vaswani et al., 2017) on the benchmark datasets.

3.5 Results

3.5.1 Code Generation

Table 3.2 and Table 3.3 show the evaluation results on code generation from summary descriptions on CodeXGLUE, and Concode datasets, respectively. First, we compare REDCODER with the state-of-the-art code generation models. They are transformers models pre-trained with different objectives using external resources of different sizes. Among them, the relatively strong baseline PLBART has an EM score of 18 on the Concode dataset while it rarely generates any code that matches the real target code in CodeXGLUE (See Table 3.2) . The BLEU and CodeBLEU scores are also low. Such result indicates that automated code lacks quality and correctness without the proper supervision in the input to the generator.

Among the retriever-only models, SCODE-R significantly outperforms BM25. As expected, the EM is zero as targets are filtered from the retrieval, and CodeBLEU scores are high as they are real code. However, although the retrieved code does not exactly match the target code, they are quite relevant (*e.g.,* Figure 3.3). When comparing retrieval-only models to generative models, it is interesting to note that SCODE-R surpasses PLBART by a large margin on CodeXGLUE (Table 3.2), suggesting that retrieved code has high overlapping with target code that can benefit the generation.

Overall, the retrieval augmented generative models excel in code generation. Our proposed framework REDCODER outperforms PLBART by a large margin, validating the advantage of reusing existing codebases to help code generation. The REDCODER-EXT gains are even higher. For CodeXGLUE (Java, Python) and Concode, the gains in BLEU are 18.88, 19.54, and 5.8. Comparing REDCODER to REDCODER-EXT shows that BLEU scores on Concode and all metrics on CodeXGLUE are improved by $\sim$1%. These results confirm our conjecture that complementing input with paired summaries of the retrieved code help code generation. We provide a qualitative example in Table 3.6 to explain how the retrieved information helps PLBART in generation.

Input summary: Get the MuxerStream at the given position.

((a)) SCODER Retrieval (Rank-1) **[BLEU: 0.4298]**

```
1 public Muxer getMuxer () {
2   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this);
3   return (cPtr == 0) ? null : new Muxer (cPtr, false);
4 }
```

Paired summary: Get the Muxer that this stream belongs to

((b)) SCODER Retrieval (Rank-3) **[BLEU: 0.6740]**

```
1 public DemuxerStream getStream (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.Demuxer_getStream(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new DemuxerStream (cPtr, false);
5 }
```

((c)) REDCODER Prediction **[BLEU: 0.6170]**

```
1 public MuxerStream getMuxerStream (int position) {
2   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, position);
3   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
4 }
```

((d)) REDCODER-ext Prediction **[BLEU: 0.8062]**

```
1 public MuxerStream getMuxer (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
5 }
```

((e)) Reference (Gold Output)

```
1 public MuxerStream getMuxer (int streamIndex)
2   throws java.lang.InterruptedException, java.io.IOException {
3   long cPtr = VideoJNI.MuxerStream_getMuxer(swigCPtr, this, streamIndex);
4   return (cPtr == 0) ? null : new MuxerStream (cPtr, false);
5 }
```

Figure 3.6: A qualitative example to show the effectiveness of retrieval-augmented generation as proposed in REDCODER framework

3.5.2 Code Summarization

We compare REDCODER with three sets of baseline methods for code summarization, and Table 3.4 shows the results. Among the two retrieval base methods, SCODE-R performs significantly well, confirming the advantages of dense retrieval over its sparse counterpart. Out of the generative methods, PLBART excels on code summarization as it leverages an extensive collection of natural language descriptions during pre-training. As anticipated, retrieval augmented generative methods outperform the other two sets of models. We see that the "BM25 + PLBART" model improves over PLBART, confirming our conjecture that retrieval augmented techniques have the promise to improve code summarization. Our proposed framework REDCODER and its variant REDCODER-EXT outshine "BM25 + PLBART", surpassing its performance by ~1.5 and ~3.2 points for Python and Java languages, respectively.

3.6 Analysis

In this Section, we analyze REDCODER's performance on the following points.

Retrieval database includes the target sequence As expected, SCODE-R performances are much better than those in Table 3.2, 3.3, and 3.4. In all cases, REDCODER gets more enhanced when target is present in the retrieval database. For the code generation task, we plot the recall@k curve for k upto 10 for both Java and Python on CodeXGLUE dataset when the retrieval contains the target in Figure 3.7. As we can see, SCODE-R significantly outperforms in both languages and for all k values.

Bi-encoder SCODE-R vs cross-encoder retrievers Table 3.6 shows the retrieval performance of different alternative retrieval techniques that we considered in REDCODER. SCODE-R performs comparably well with GraphCodeBERT while being significantly faster and scalable Humeau et al. (2020). Note that, SCODE-R also uses GraphCodeBERT to initialize its encoders (see Figure 3.4). However, SCODE-R's design of using different encoders for query and documents enables pre-indexing of database and faster retrieval in

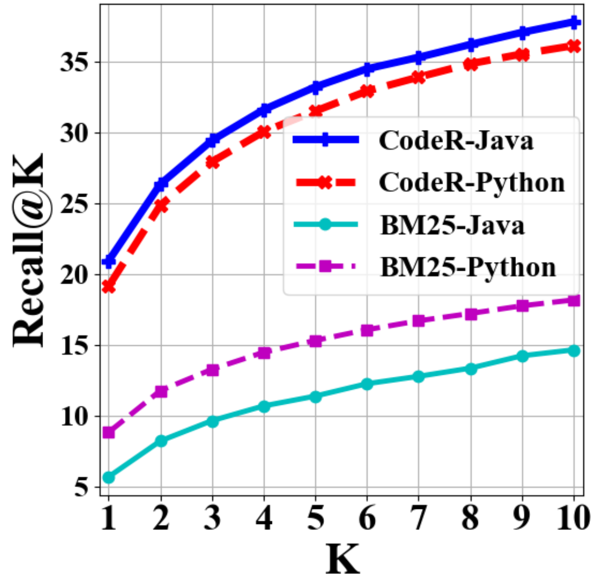


Figure 3.7: Recall@K for CodeR and BM25. CodeR refers to SCODE-R used for source code retrieval.

practice.

Performance vs target length Figure 3.8 shows the code generation performances of different models *w.r.t.* the target code length for Python. While the generator model (PLBART)'s performance consistently decreases with increasing code size, the retriever (SCODE-R) performs consistently well. Such consistent performance from SCODE-R boosts performance of REDCODER (and also REDCODER-EXT) *significantly higher* than the generative model counterpart. For Java, we find similar results.

Performance vs #retrievals Figure 3.9 shows that typically the performance improves more with more retrievals on both tasks. However, roughly 5 code and 30 summaries work sufficiently well.

Human evaluation Finally, we evaluate the quality of code generated by SCODE-G using human evaluation. In Table 3.7, we perform a human evaluation for code generation task on a subset of the test set in CodeXGLUE (Python). In this study, we compare REDCODER generated code with the code retrieved by SCODE-R. Note that both REDCODER and SCODE-R using the same retrievers, but REDCODER generates code using SCODE-G, while SCODE-R outputs code written by real programmers. We sample

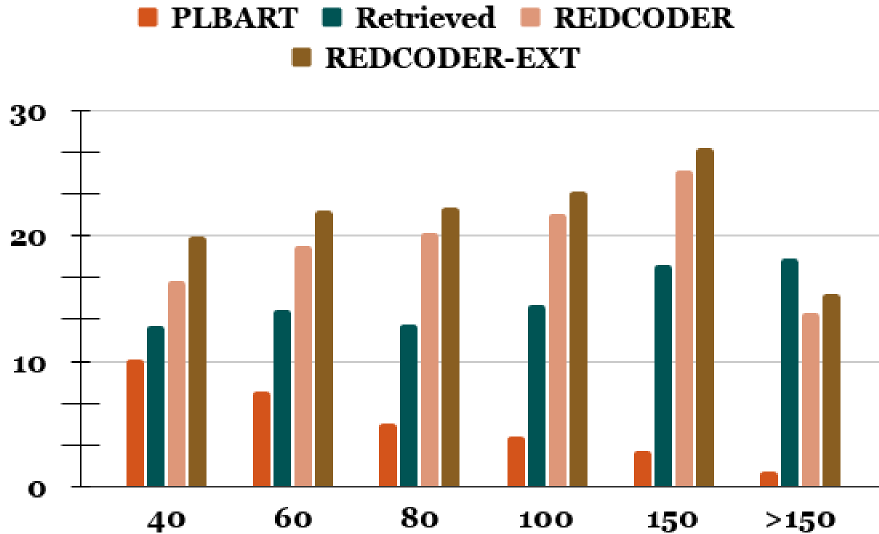


Figure 3.8: (Python) Code gen. BLEU vs target len.

30 instances where REDCODER generated code has a lower BLEU score than that of the SCODE-R and investigate whether the quality of code generated by them are significantly different on these cases.

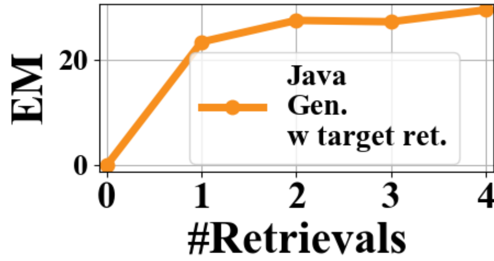
Model	Human Evaluation			Automatic Metric		
	Similarity	Relevance	Compilability	BLEU	EM	CodeBLEU
SCODE-R	2.09	3.00	3.16	11.56	0.00	16.66
REDCODER	2.06	2.94	3.10	10.70	0.07	18.31

Table 3.7: Human evaluation on code generation (CodeXGLUE-Python). REDCODER (SCODE-R + SCODE-G) achieves similar scores as SCODE-R that directly retrieves developers' written code which suggests that the quality of the code generated by SCODE-G are competitive with real code from programmers' perspective.

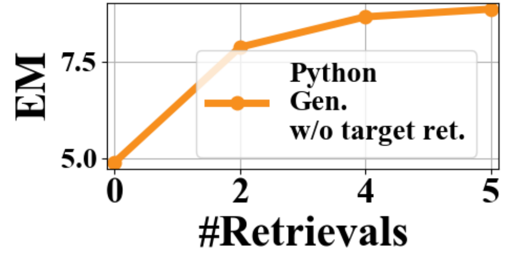
As programming requires a specific skill, we do not evaluate the quality of the code generation using the mass crowd workers. We recruit 7 Ph.D. students studying in computer science as volunteers[2] to score (1 to 5) code based on three criteria (i) similarity, and (ii) relevance *w.r.t.* the target code; (iii) the compilability of the generated code.

The ratings show that both models receive similar scores, with a slightly higher score for SCODE-R in terms of similarity to the target code, relevancy, and compilability. This

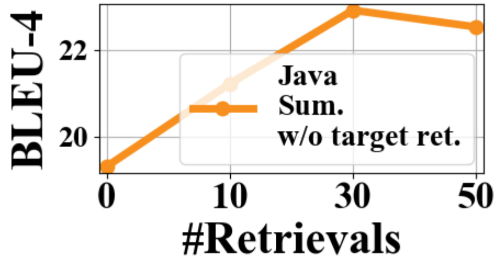
[2]Before participating in the evaluation process, all the participants are informed that it is a voluntary task and it may take roughly 30 minutes to perform the evaluation.



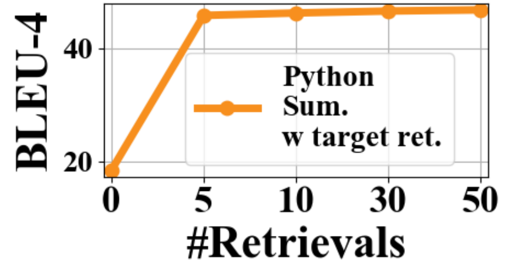
((a)) CodeXGLUE (Java) gen.



((b)) CodeXGLUE (Python) gen.



((c)) CodeXGLUE (Java) sum.



((d)) CodeXGLUE (Python) sum.

Figure 3.9: Code gen. and sum. performance vs #retrievals. In general performance improves with higher number of augmented candidates.

shows that the quality of the code generated by SCODE-G are competitive with real code from programmers' perspective. Interestingly, REDCODER achieves higher scores than SCODE-R in CodeBLEU and Exact Match even on the cases where its BLEU score is lower.

Qualitative Example In Figure 3.6, we show an example of generated code by a baseline and different modules of REDCODER. The input summary asks to write a code (in Java) to get a MuxerStream given a position.

We show two of the corresponding retrieved code, their summaries (for *bimodal* instances), generated code of PLBART, REDCODER, and REDCODER-EXT. As can be seen, PLBART generates a basic but relevant code; both retrieved code (rank-1 and rank-3) contains the statements with variable cPtr one of them is of MuxerStream class, and another is from DeMuxerStream class. REDCODER generates a somewhat correct code of MuxerStream class and it takes the position argument too. Seemingly, while fusing the retrieved code, we suspect that as the tentative function name MuxerStream

mentioned in the input summary does not match the function name DeMuxerStream of the rank-3 retrieved code, it only adapts one line containing cPtr from rank-3 retrieved code (line #3) and takes the rests including the function definition (i.e., line #1) from the rank-1 retrieved code. Now when REDCODER-EXT is allowed to leverage the summaries of the retrieved code, it can match the summary of the rank-3 retrieved code with the input, and that is why it produces the MuxerStream class object but with the throw exceptions from the rank-3 retrieved code.

Performance Difference of PLBART on CodeXGLUE and Concode Concode is a relatively easier dataset for code generation and retrieval due to several pre-processing steps taken by its authors. Along with additional contexts (environment variables and methods) in the input summary, Concode artifacts the target code by replacing the specific variable names with generic tokens.

```
1 void function(Element arg0,
2   Formula arg1) {
3   arg0.addElement(
4     "concode_string").setText(
5       arg1.getText());
6 }
```

Therefore, we suspect that due to this, PLBART achieves good EM score for Concode but not for the generation of real code in CodeXGLUE.

Analogously for the retrieval models, code retrieved by BM25 have also a large word overlapping with the targets in Concode in contrast to CodeXGLUE (1st row in Table 2 and 3 in the main paper). Consequently, BM25 retrieval boosts PLBART (i.e., BM25 + PLBART) more in Concode than that in CodeXGLUE (3rd row for the bottom in Table 2 and 3 in the main paper) Overall, we anticipate all these skewness in model performances are due to the dataset characteristics.

Dataset	Lang.	Task	Retrieval Database			|Size|	|Nonparallel|
			CSNet	CCSD	Concode		
CodeXGLUE	Python	Gen.	✓	✗	✗	1.2M	504K
		Sum.	✓	✓	✗	1.1M	833K
	Java	Gen.	✓	✗	✗	1.6M	543K
		Sum.	✓	✓	✗	1.1M	903K
Concode	Java	Gen.	✗	✗	✓	104K	0

Table 3.8: Retrieval database statistics. “Size” refers to both of parallel and nonparallel code or summaries. As Concode has a different data format, we only retrieve from itself. Nonparallel means the retrieval candidates are only code (for code gen.) and only summaries (for code sum.). CSNet (CodeSearchNet), CCSD refer to Husain et al. (2019) and Liu et al. (2021).

code retrieval	target present in retrieval	summary retrieval	CodeXGLUE (Java)			CodeXGLUE (Python)		
			BLEU	EM	CodeBLEU	BLEU	EM	CodeBLEU
✗	✗	✗	10.1	0.0	14.96	4.89	0.0	12.01
✓	✗	✗	26.92	8.95	31.15	22.74	8.88	28.93
		✓	**28.98**	**10.21**	**33.18**	**24.43**	**9.61**	**30.21**
	✓	✗	36.33	29.41	41.38	32.14	27.48	38.02
		✓	**42.82**	**36.99**	**47.25**	**38.87**	**34.51**	**43.78**

Table 3.9: Ablation results on source code generation using the retrieved code and its summary together when the reference target code is absent and present in the retrieval database respectively.

Methods	CodeXGLUE-Python		CodeXGLUE-Java	
	BLEU-4	ROUGE-L	BLEU-4	ROUGE-L
SCODE-R	46.6	53.8	48.0	55.7
REDCODER	47.0	55.4	**50.4**	**58.8**
REDCODER-EXT	**47.1**	**55.5**	**50.4**	58.7

Table 3.10: Evaluation results of code summarization keeping the target summary in the retrieval database.

3.7 Related Works

Code Summarization. In recent years, source code summarization attracted a lot of attention (Iyer et al., 2016; Liang and Zhu, 2018; Allamanis et al., 2016; Hu et al., 2018b; Ahmad et al., 2020a). Many of these works view code as a sequence of token. Other approaches leverage the structural properties of code using Tree based model (Shido et al., 2019; Harer et al., 2019; Hu et al., 2018a; LeClair et al., 2019). In literature, several

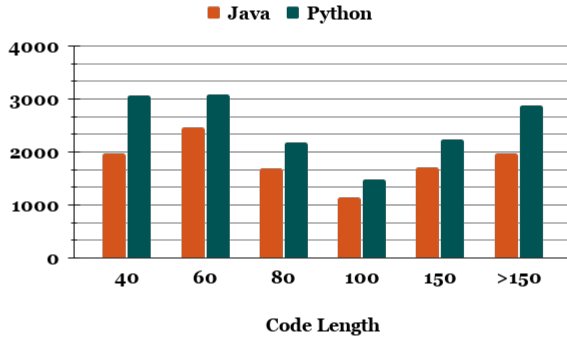


Figure 3.10: #Code per target length.

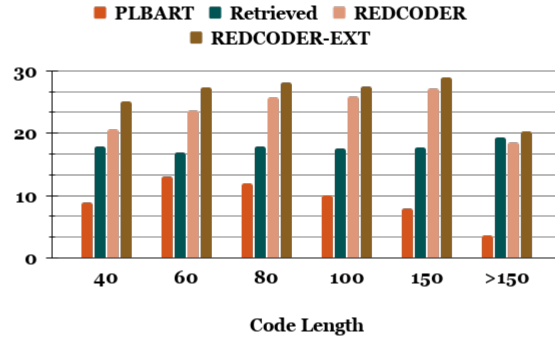


Figure 3.11: BLEU vs target len. (Java)

retrieval-based methods were proposed that leverage retrieved information along with the input code. For example, Zhang et al. (2020) retrieves similar code snippet and use those as an auxiliary input for summarization. On the other hand, Hayati et al. (2018) retrieves related summaries for augmenting summarization input. Different from these approaches, REDCODER leverages both the retrieved code and its summary to augment the input.

Code Generation. Generating source code is a major stepping stone towards automated programming. Yin and Neubig (2017a), and Rabinovich et al. (2017a) proposed code generation as abstract syntax tree generation to ensure its syntactic correctness. Recent advancements in pre-training language models on unlabeled source code data (Lu et al., 2021; Ahmad et al., 2021a) showed colossal promise towards learning code syntax and semantics, resulting in improved code generation models.

Code Retrieval and Others. Numerous software engineering applications require information retrieval. Sadowski et al. (2015); Xia et al. (2017); Stolee et al. (2014); Sim et al. (2011) show that developers search for related code, API examples for implementing or adapting new APIs. Design of REDCODER is inspired by developers' behavior while writing code. Developers use search engines for retrieving off-the-shelf libraries (Hucka and Graham, 2018), or "usable" source code (Rahman et al., 2018) for adapting in the development process (Nasehi et al., 2012; Arwan et al., 2015; Ponzanelli et al., 2014). Similarly, REDCODER retrieves existing code or summaries and adapts them to generate the target code or summary. In contrast, Hashimoto et al. (2018) optimizes a joint

objective; Zhang et al. (2020); Liu et al. (2021) do not consider any decoder pre-training, Lewis et al. (2020) fine-tunes both of the retriever and the generator end-to-end. For open domain QA, Izacard and Grave (2021) propose a similar model of alternative generator (multi-encoder uni-decoder).

3.8 Conclusion

We propose REDCODER to automate developers' writing of code and documentation by reusing what they have written previously. We evaluate REDCODER on two benchmark datasets and the results demonstrate a significant performance boost with the help of the retrieved information. In the future, we want to extend REDCODER to support other code automation tasks such as code translation.

3.9 Summary

In this work, we build a dense retriever model to retrieve code and summaries from large pool of candidates (e.g., Github). For the example tasks of code generation and summarization, we use the top-k retrieved candidates on the fly as additional hints/features when generating the code and summaries respectively. With comprehensive studies on three benchmark datatsets for two programming languages (Java and Python), we demonstrate that just raw the retrieved candidates themselves can bring such useful auxiliary supervision that sourpusses the existing generative baselines. Leveraging them the generative models further excels the output quality and achieves new stat-of-the-art performances. However, we only modify the encoder input of the (encoder-decoder) generative models in this Chapter and more task oriented improvements on the decoder sides are also possible. Additionally, in the future, we plan to explore such retrieval augmented models for other multi-modal applications (e.g., text-image and text-speech).

CHAPTER 4

Retrieving and Leveraging Entity-type Information for Language Modelling on Text with Named Entities

Text in many domains involves a significant amount of named entities. While generating such text or code, predicting the entity names is often challenging for a language model or a decoder model as they appear less frequent on the training corpus. One such example application is the code generation task we discussed in the previous Chapter where we enhance it partially w/o considering any task oriented improvements on the decoder side. As a continuation, in this Chapter, we will be using simple rule-based off-the-shelf tools such as Symbol tables to retrieve task-oriented entity type information (e.g., variable/data types int, float, user-defined class etc.,) and use to further enhance the generation. Here, we will be presenting a novel and effective approach to building a discriminative language model (similar to the decoding step in an encoder-decoder model) which can learn the entity names by leveraging their retrieved entity type information. This is based on our work Parvez et al. (2018) which is also a *Type-2* ***feature level*** auxiliary supervision. We also introduce two benchmark datasets based on recipes and Java programming codes, on which we evaluate the proposed model. Experimental results show that our model achieves 52.2% better perplexity in recipe generation and 22.06% on code generation than the state-of-the-art language models.

4.1 Introduction

Language model is a fundamental component in Natural Language Processing (NLP) and it supports various applications, including document generation (Wiseman et al., 2017),

text auto-completion (Arnold et al., 2017), spelling correction (Brill and Moore, 2000), and many others. Recently, language models are also successfully used to generate software source code written in programming languages like Java, C, etc. (Hindle et al., 2016; Yin and Neubig, 2017b; Hellendoorn and Devanbu, 2017; Rabinovich et al., 2017b). These models have improved the language generation tasks to a great extent, e.g., (Mikolov et al., 2010; Galley et al., 2015). However, while generating text or code with a large number of named entities (e.g., different variable names in source code), these models often fail to predict the entity names properly due to their wide variations. For instance, consider building a language model for generating recipes. There are numerous similar, yet slightly different cooking ingredients (e.g., *olive oil, canola oil, grape oil, etc.*—all are different varieties of oil). Such diverse vocabularies of the ingredient names hinder the language model from predicting them properly.

To address this problem, we propose a novel language model for texts with many entity names. Our model learns the probability distribution over all the candidate words by leveraging the entity type information. For example, oil is the *type* for named entities like olive oil, canola oil, grape oil, etc.[1] Such type information is even more prevalent for source code corpus written in statically typed programming languages (Bruce, 1993), since all the variables are by construct associated with types like integer, float, string, etc.

Our model exploits such deterministic type information of the named entities and learns the probability distribution over the candidate words by decomposing it into two sub-components: (i) *Type Model.* Instead of distinguishing the individual names of the same type of entities, we first consider all of them equal and represent them by their type information. This reduces the vocab size to a great extent and enables to predict the type of each entity more accurately. (ii) *Entity Composite Model.* Using the entity type as a prior, we learn the conditional probability distribution of the actual entity names at inference time. We depict our model in Fig. 6.1.

To evaluate our model, we create two benchmark datasets that involve many named

[1]Entity type information is often referred as category information or group information. In many applications, such information can be easily obtained by an ontology or by a pre-constructed entity table.

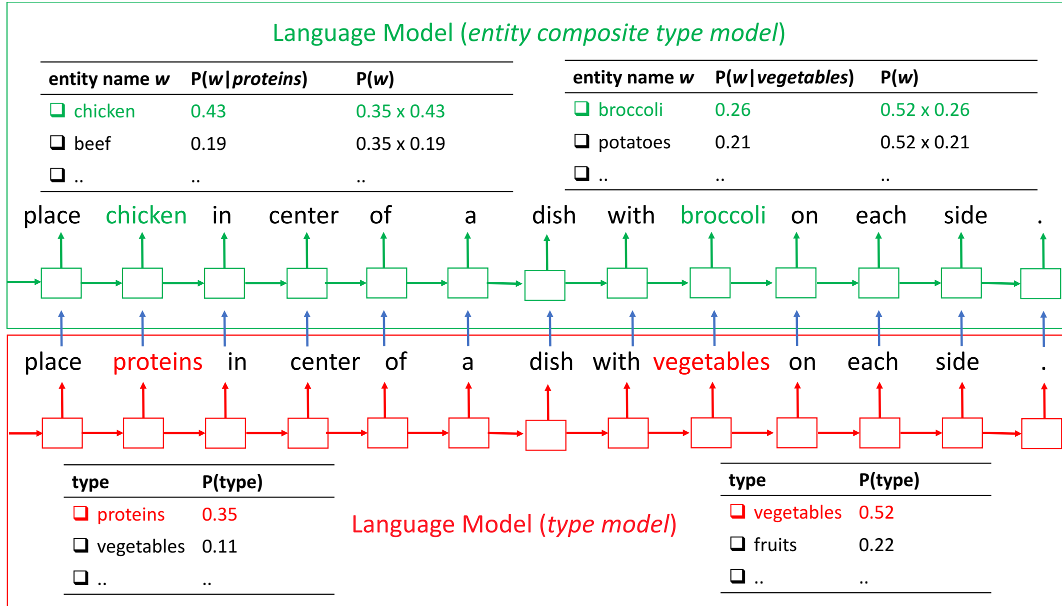


Figure 4.1: An example illustration of the proposed model. For a given context (i.e., types of context words as input), the *type model* (in bottom red block) generates the type of the next word (i.e., the probability of the type of the next word as output). Further, for a given context *and* type of each candidate (i.e., context words, corresponding types of the context words, and type of the next word generated by the *type model* as input), the *entity composite model* (in upper green block) predicts the next word (actual entity name) by estimating the conditional probability of the next word as output. We conduct joint inference over both models to leverage type information for generating text.

entities. One is a cooking recipe corpus[2] where each recipe contains a number of ingredients which are categorized into 8 super-ingredients (i.e., type); e.g., “proteins”, “vegetables”, “fruits”, “seasonings”, “grains”, etc. Our second dataset comprises a source code corpus of 500 open-source Android projects collected from GitHub. We use an Abstract Syntax Tree (AST) (Parsons, 1992) based approach to collect the type information of the code identifiers.

Our experiments show that although state-of-the-art language models are, in general, good to learn the frequent words with enough training instances, they perform poorly on the entity names. A simple addition of type information as an extra feature to a neural network does not guarantee to improve the performance because more features may overfit or need more model parameters on the same data. In contrast, our proposed

[2]Data is crawled from `http://www.ffts.com/recipes.htm`.

method significantly outperforms state-of-the-art neural network based language models and also the models with type information added as an extra feature.

Overall, followings are our contributions:

- We analyze two benchmark language corpora where each consists of a reasonable number of entity names. While we leverage an existing corpus for recipe, we curated the code corpus. For both datasets, we created auxiliary corpora with entity type information. All the code and datasets are released.[3]
- We design a language model for text consisting of many entity names. The model learns to mention entities names by leveraging the entity type information.
- We evaluate our model on our benchmark datasets and establish a new baseline performance which significantly outperforms state-of-the-art language models.

4.2 Related Work and Background

Class Based Language Models. Building language models by leveraging the deterministic or probabilistic class properties of the words (a.k.a, class-based language models) is an old idea (Brown et al., 1992; Goodman, 2001). However, the objective of our model is different from the existing class-based language models. The key differences are two-folds: 1) Most existing class-based language models (Brown et al., 1992; Pereira et al., 1993; Niesler et al., 1998; Baker and McCallum, 1998; Goodman, 2001; Maltese et al., 2001) are generative n-gram models whereas ours is a discriminative language model based on neural networks. The modeling principle and assumptions are very different. For example, we cannot calculate the conditional probability by statistical occurrence counting as these papers did. 2) Our approaches consider building two models and perform joint inference which makes our framework general and easy to extend. In Section 4.4, we demonstrate that our model can be easily incorporated with the state-of-art language model. The closest work in this line is hierarchical neural language models (Morin and Bengio, 2005),

[3] https://github.com/uclanlp/NamedEntityLanguageModel

which model language with word clusters. However, their approaches do not focus on dealing with named entities as our model does. A recent work (Ji et al., 2017) studied the problem of building up a dynamic representation of named entity by updating the representation for every contextualized mention of that entity. Nonetheless, their approach does not deal with the sparsity issue and their goal is different from ours.

Language Models for Named Entities. In some generation tasks, recently developed language models address the problem of predicting entity names by copying/matching the entity names from the reference corpus. For example, Vinyals et al. (2015) calculates the conditional probability of discrete output token sequence corresponding to positions in an input sequence. Gu et al. (2016a) develops a seq2seq alignment mechanism which directly copies entity names or long phrases from the input sequence. Wiseman et al. (2017) generates document from structured table like basketball statistics using copy and reconstruction method as well. Another related code generation model (Yin and Neubig, 2017b) parses natural language descriptions into source code considering the grammar and syntax in the target programming language (e.g., Python). Kiddon et al. (2016) generates recipe for a given goal, and agenda by making use of items on the agenda. While generating the recipe it continuously monitors the agenda coverage and focus on increasing it. All of them are sequence-to-sequence learning or end-to-end systems which differ from our general purpose (free form) language generation task (e.g., text auto-completion, spelling correction).

Code Generation. The way developers write codes is not only just writing a bunch of instructions to run a machine, but also a form of communication to convey their thought. As observed by Donald E. Knuth (Knuth, 1992), *"The practitioner of literate programming can be regarded as an essayist, whose main concern is exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what such variable means."* Such comprehensible software corpora show surprising regularity (Ray et al., 2015; Gabel and Su, 2010) that is quite similar to the statistical properties of natural language corpora and thus, amenable to

large-scale statistical analysis (Hindle et al., 2012). (Allamanis et al., 2017) presented a detailed survey.

Although similar, source code has some unique properties that differentiate it from natural language. For example, source code often shows more regularities in local context due to common development practices like copy-pasting (Gharehyazie et al., 2017; Kim et al., 2005). This property is successfully captured by cache based language models (Hellendoorn and Devanbu, 2017; Tu et al., 2014). Code is also less ambiguous than natural language so that it can be interpreted by a compiler. The constraints for generating correct code is implemented by combining language model and program analysis technique (Raychev et al., 2014). Moreover, code contains open vocabulary—developers can coin new variable names without changing the semantics of the programs. Our model aims to addresses this property by leveraging variable types and scope.

LSTM Language Model. In this paper, we use LSTM language model as a running example to describe our approach. Our language model uses the LSTM cells to generate latent states for a given context which captures the necessary features from the text. At the output layer of our model, we use Softmax probability distribution to predict the next word based on the latent state. Merity et al. (2017) is a LSTM-based language model which achieves the state-of-the-art performance on Penn Treebank (PTB) and WikiText-2 (WT2) datasets. To build our recipe language model we use this as a blackbox and for our code generation task we use the simple LSTM model both in forward and backward direction. A forward directional LSTM starts from the beginning of a sentence and goes from left to right sequentially until the sentence ends, and vice versa. However, our approach is general and can be applied with other types of language models.

4.3 A Probabilistic Model for Text with Named Entities

In this section, we present our approach to build a language model for text with name entities. Given previous context $\bar{w} = \{w_1, w_2, .., w_{t-1}\}$, the goal of a language model is to

predict the probability of next word $P(w_t|\bar{w})$ at time step t, where $w_t \in V^{text}$ and V^{text} is a fixed vocabulary set. Because the size of vocabulary for named entities is large and named entities often occur less frequently in the training corpus, the language model cannot generate these named entities accurately. For example, in our recipe test corpus the word "apple" occurs only 720 times whereas any kind of "fruits" occur 27,726 times. Existing approaches often either only generate common named entities or omit entities when generating text (Jozefowicz et al., 2016).

To overcome this challenge, we propose to leverage the entity type information when modeling text with many entities. We assume each entity is associated with an entity type in a finite set of categories $S = \{s_1, s_2, .., s_i, .., s_k\}$. Given a word w, $s(w)$ reflects its entity type. If the word is a named entity, then we denote $s(w) \in S$; otherwise the type function returns the words itself (i.e, $s(w) = w$). To simplify the notations, we use $s(w) \notin S$ to represent the case where the word is not an entity. The entity type information given by $s(w)$ is an auxiliary information that we can use to improve the language model. We use $s(\bar{w})$ to represent the entity type information of all the words in context $\bar{w}$ and use w to represent the current word w_t. Below, we show that a language model for text with typed information can be decomposed into the following two models: 1) a *type model* θ_t that predicts the entity type of the next word and 2) an *entity composite model* θ_v that predicts the next word based on a given entity type.

Our goal is to model the probability of next word w given previous context $\bar{w}$:

$$P\left(w|\bar{w};\theta_t,\theta_v\right), \tag{4.1}$$

where θ_t and θ_v are the parameters of the two aforementioned models. As we assume the typed information is given on the data, Eq. (4.1) is equivalent to

$$P\left(w, s(w)|\bar{w}, s(\bar{w});\theta_t,\theta_v\right). \tag{4.2}$$

A word can be either a named entity or not; therefore, we consider the following two

cases.

Case 1: next word is a named entity. In this case, Eq. (4.2) can be rewritten as

$$\begin{aligned} &P\left(s(w)=s|\bar{w}, s(\bar{w}); \theta_t, \theta_v\right) \times \\ &P\left(w|\bar{w}, s(\bar{w}), s(w)=s; \theta_v, \theta_t\right) \end{aligned} \tag{4.3}$$

based on the rules of conditional probability.

We assume the type of the next token $s(w)$ can be predicted by a model θ_t using information of $s(\bar{w})$, and we can approximate the first term in Eq. (4.3)

$$P(s(w)|\bar{w}, s(\bar{w}); \theta_t, \theta_v) \approx P(s(w)|s(\bar{w}), \theta_t) \tag{4.4}$$

Similarly, we can make a modeling assumption to simplify the second term as

$$\begin{aligned} &P(w|\bar{w}, s(\bar{w}), s(w), \theta_v, \theta_t) \\ &\approx P(w|\bar{w}, s(\bar{w}), s(w), \theta_v). \end{aligned} \tag{4.5}$$

Case 2: next word is not a named entity. In this case, we can rewrite Eq. (4.2) to be

$$\begin{aligned} &P\left(s(w) \notin S|\bar{w}, s(\bar{w}), \theta_t\right) \times \\ &P\left(w|\bar{w}, s(\bar{w}), s(w) \notin S, \theta_v\right). \end{aligned} \tag{4.6}$$

The first term in Eq. (4.6) can be modeled by

$$1 - \sum_{s \in S} P(s(w)=s|s(\bar{w}), \theta_t),$$

which can be computed by the *type* model[4]. The second term can be again approximated by (4.5) and further estimated by an *entity composition model*.

[4]Empirically for the non-entity words, $\sum_{s \in S} P(s(w)=s|s(\bar{w}) \approx 0$

Typed Language Model. Combine the aforementioned equations, the proposed language model estimates $P(w|\bar{w};\theta_t,\theta_v)$ by

$$P(w|\bar{w},s(\bar{w}),s(w),\theta_v)\times \begin{cases} P(s(w)|s(\bar{w}),\theta_t) & \text{if } s(w)\in S \\ (1-\sum_{s\in S}P(s(w)=s|s(\bar{w}),\theta_t)) & \text{if } s(w)\notin S \end{cases} \tag{4.7}$$

The first term can be estimated by an *entity composite model* and the second term can be estimated by a *type model* as discussed below.

4.3.1 Type model

The *type model* θ_t estimates the probability of $P(s(w)|s(\bar{w}),\theta_t)$. It can be viewed as a language model builds on a corpus with all entities replaced by their type. That is, assume the training corpus consists of $x=\{w_1,w_2,..,w_n\}$. Using the type information provided in the auxiliary source, we can replace each word w with their corresponding type $s(w)$ and generate a corpus of $\mathcal{T}=\{s(w_i),s(w_2),..,s(w_n)\}$. Note that if w_i is not an named entity (i.e., $s(w)\notin S$), $s(w)=w$ and the vocabulary on $\mathcal{T}$ is $V^{text}\cup S$.[5] Any language modeling technique can be used in modeling the *type model* on the modified corpus $\mathcal{T}$. In this paper, we use the state-of-the-art model for each individual task. The details will be discussed in the experiment section.

[5] In a preliminary experiment, we consider putting all words with $s(w)\notin S$ in a category "N/A". However, because most words on the training corpus are not named entities, the type "N/A" dominates others and hinder the *type model* to make accurate predictions.

4.3.2 Entity Composite Model

The *entity composite model* predicts the next word based on modeling the conditional probability $P(w|\bar{w}, s(\bar{w}), s(w), \theta_v)$, which can be derived by

$$\frac{P(w|\bar{w}, s(\bar{w}); \theta_v)}{\sum_{w_s \in \Omega(s(w))} P(w_s|\bar{w}, s(\bar{w}); \theta_v)}, \tag{4.8}$$

where $\Omega(s(w))$ is the set of words of the same type with w.

To model the types of context word $s(\bar{w})$ in $P(w|\bar{w}, s(\bar{w}); \theta_v)$, we consider learning a type embedding along with the word embedding by augmenting each word vector with a type vector when learning the underlying word representation. Specifically, we represent each word w as a vector of $[v_w(w)^T; v_t(s(w))^T]^T$, where $v_w(\cdot)$ and $v_t(\cdot)$ are the word vectors and type vectors learned by the model from the training corpus, respectively. Finally, to estimate Eq. (4.8) using θ_v, when computing the Softmax layer, we normalize over only words in $\Omega(s(w))$. In this way, the conditional probability $P(w|\bar{w}, s(\bar{w}), s(w), \theta_v)$ can be derived.

4.3.3 Training and Inference Strategies

We learn model parameters θ_t and θ_v independently by training two language models *type model* and *entity composite model* respectively. Given the context of type, *type model* predicts the type of the next word. Given the context and the type information of the all candidate words, *entity composite model* predicts the conditional actual word (e.g., entity name) as depicted in Fig 6.1. At inference time the generated probabilities from these two models are combined according to conditional probability (i.e., Eq. (4.7)) which gives the final probability distribution over all candidate words[6].

Our proposed model is flexible to any language model, training strategy, and optimiza-

[6]While calculating the final probability distribution over all candidate words, with our joint inference schema, a strong state-of-art language model, without the type information, itself can work sufficiently well and replace the *entity composite model*. Our experiments using (Merity et al., 2017) in Section 4.4.1 validate this claim.

Algorithm 2 Language Generation

Input:

Language corpus $\mathcal{X} = \{w_1, w_2, .., w_n\}$,
type $s(w)$ of the words,
integer number m.

Output: θ_t, θ_v, $\{W_1, W_2, .., W_m\}$

Training Phase:
Generate $\mathcal{T} = \{\ s(w_1), s(w_2), .., s(w_n)\}$
Train **type model** θ_t on $\mathcal{T}$
Train **entity composite model** θ_v on $\mathcal{X}$ using $[w_i; s(w_i)]$ as input

Test Phase (Generation Phase):

for $i = 1$ to m **do**

  for $w \in V^{text}$ **do**

    Compute $P(s(w)|s(\bar{w}), \theta_t)$

    Compute $P(w|\bar{w}, s(\bar{w}), s(w), \theta_v)$

    Compute $P(w|\bar{w}; \theta_t, \theta_v)$ using Eq.(4.7)

  $W_i \leftarrow argmax_w P(w|\bar{w}; \theta_t, \theta_v)$

tion. As per our experiments, we use ADAM stochastic mini-batch optimization (Kingma and Ba, 2014). In Algorithm 2, we summarize the language generation procedure.

4.4 Experiments

We evaluate our proposed model on two different language generation tasks where there exist a lot of entity names in the text. In this paper, we release all the codes and datasets. The first task is recipe generation. For this task, we analyze a cooking recipe corpus. Each instance in this corpus is an individual recipe and consists of many ingredients'. Our second task is code generation. We construct a Java code corpus where each instance is a Java method (i.e., function). These tasks are challenging because they have the abundance of entity names and state-of-the-art language models fail to predict them properly as a result of insufficient training observations. Although in this paper, we manually annotate the types of the recipe ingredients, in other applications it can be

acquired automatically. For example: in our second task of code generation, the types are found using Eclipse JDT framework. In general, using DBpedia ontology (e.g., "Berlin" has an ontology "Location"), Wordnet hierarchy (e.g., "Dog" is an "Animal"), role in sports (e.g., "Messi" plays in "Forward"; also available in DBpedia[7]), Thesaurus (e.g., "renal cortex", "renal pelvis", "renal vein", all are related to "kidney"), Medscape (e.g., "Advil" and "Motrin" are actually "Ibuprofen"), we can get the necessary type information. As for the applications where the entity types cannot be extracted automatically by these frameworks (e.g., recipe ingredients), although there is no exact strategy, any reasonable design can work. Heuristically, while annotating manually in our first task, we choose the total number of types in such a way that each type has somewhat balanced (similar) size.

We use the same dimensional word embedding (400 for recipe corpus, 300 for code corpus) to represent both of the entity name (e.g., "apple") and their entity type (e.g., "fruits") in all the models. Note that in our approach, the *type model* only replaces named entities with entity type when it generates next word. If next word is not a named entity, it will behave like a regular language model. Therefore, we set both models with the same dimensionality. Accordingly, for the *entity composite model* which takes the concatenation of the entity name and the entity type, the concatenated input dimension is 800 and 600 respectively for recipe and code corpora.

4.4.1 Retrieving Ingredient Types and the Results of Recipe Generation

4.4.1.1 Recipe Corpus Pre-processing:

Our recipe corpus collection is inspired by (Kiddon et al., 2016). We crawl the recipes from "Now You're Cooking! Recipe Software" [8]. Among more than 150,000 recipes in this dataset, we select similarly structured/formatted (e.g, title, blank line then ingredient lists followed by a recipe) 95,786 recipes. We remove all the irrelevant information (e.g., author's name, data source) and keep only two information: ingredients and recipes. We

[7]http://dbpedia.org/page/Lionel_Messi

[8]http://www.ffts.com/recipes.htm

set aside the randomly selected 20% of the recipes for testing and from the rest, we keep randomly selected 80% for the training and 20% for the development. Similar to (Kiddon et al., 2016), we pre-process the dataset and filter out the numerical values, special tokens, punctuation, and symbols.[9] Quantitatively, the data we filter out is negligible; in terms of words, we keep 9,994,365 words out of 10,231,106 and the number of filter out words is around ∼2%. We release both of the raw and cleaned data for future challenges. As the ingredients are the entity names in our dataset, we process it separately to get the type information.

4.4.1.2 Retrieving Ingredient Type

As per our type model, for each word w, we require its type $s(w)$. We only consider ingredient type for our experiment. First, we tokenize the ingredients and consider each word as an ingredient. We manually classify the ingredients into 8 super-ingredients: "fruits", "proteins", "sides", "seasonings", "vegetables", "dairy", "drinks", and "grains". Sometimes, ingredients are expressed using multiple words; for such ingredient phrase, we classify each word in the same group (e.g., for "boneless beef" both "boneless" and "beef" are classified as "proteins"). We classify the most frequent 1,224 unique ingredients[10] which cover 944,753 out of 1,241,195 mentions (top 76%) in terms of frequency of the ingredients. In our experiments, we omit the remaining 14,881 unique ingredients which are less frequent and include some misspelled words. The number of unique ingredients in the 8 super ingredients is 110, 316, 140, 180, 156, 80, 84, and 158 respectively. We prepare the modified *type corpus* by replacing each actual ingredient's name w in the original recipe corpus by the type (i.e., super ingredients $s(w)$) to train the *type model*.

[9] For example, in our crawled raw dataset, we find that some recipes have lines like "===MM-MMM===" which are totally irrelevant to our task. For the words with numerical values like "100 ml", we only remove the "100" and keep the "ml" since our focus is not to predict the exact number.

[10] We consider both singular and plural forms. The number of singular formed annotated ingredients are 797.

4.4.1.3 Recipe Statistics

In our corpus, the total number of distinct words in vocabulary is 52,468; number of unique ingredients (considering splitting phrasal ingredients also) is 16,105; number of tokens is 8,716,664. In number of instances train/dev/test splits are 61,302/15,326/19,158. The average instance size of a meaningful recipe is 91 on the corpus.

4.4.1.4 Configuration

We consider the state-of-the art LSTM-based language model proposed in (Merity et al., 2017) as the basic component for building the *type model*, and *entity composite model*. We use 400 dimensional word embedding as described in Section 4.4. We train the embedding for our dataset. We use a minibatch of 20 instances while training and back-propagation through time value is set to 70. Inside of this (Merity et al., 2017) language model, it uses 3 layered LSTM architecture where the hidden layers are 1150 dimensional and has its own optimization and regularization mechanism. All the experiments are done using PyTorch and Python 3.5.

4.4.1.5 Baselines

Our first baseline is ASGD Weight-Dropped LSTM (AWD_LSTM) (Merity et al., 2017), which we also use to train our models (see 'Configuration' in 4.4.1.4). This model achieves the state-of-the-art performance on benchmark Penn Treebank (PTB), and WikiText-2 (WT2) language corpus. Our second baseline is the same language model (AWD_LSTM) with the type information added as an additional feature (i.e., same as *entity composite model*).

4.4.1.6 Results of Recipe Generation

We compare our model with the baselines using *perplexity* metric—lower perplexity means the better prediction. Table 4.1 summarizes the result. The 3rd row shows that adding

Model	Dataset (Recipe Corpus)	Vocabulary Size	Perplexity
AWD_LSTM	original	52,472	20.23
AWD_LSTM *type model*	modified type	51,675	17.62
AWD_LSTM with type feature	original	52,472	18.23
our model	original	52,472	**9.67**

Table 4.1: Comparing the performance of recipe generation task. All the results are on the test set of the corresponding corpus. AWD_LSTM (*type model*) is our *type model* implemented with the baseline language model AWD_LSTM (Merity et al., 2017). Our second baseline is the same language model (AWD_LSTM) with the type information added as an additional feature for each word.

type as a simple feature does not guarantee a significant performance improvement while our proposed method significantly outperforms both baselines and achieves 52.2% improvement with respect to baseline in terms of perplexity. To illustrate more, we provide an example snippet of our test corpus: "place onion and ginger inside chicken . allow chicken to marinate for hour .". Here, for the last mention of the word "chicken", the standard language model assigns probability 0.23 to this word, while ours assigns probability 0.81.

4.4.2 Retrieving Token Types and the Results of Code Generation

4.4.2.1 Code Corpus Pre-processing

We crawl 500 Android open source projects from GitHub[11]. GitHub is the largest open source software forge where anyone can contribute (Ray et al., 2014). Thus, GitHub also contains trivial projects like student projects, etc. In our case, we want to study the coding practices of practitioners so that our model can learn to generate quality code. To ensure this, we choose only those Android projects from GitHub that are also present in

[11]https://github.com

Google Play Store[12]. We download the source code of these projects from GitHub using an off the shelf tool GitcProc (Casalnuovo et al., 2017).

Since real software continuously evolves to cater new requirements or bug fixes, to make our modeling task more realistic, we further study different project versions. We partition the codebase of a project into multiple versions based on the code commit history retrieved from GitHub; each version is taken at an interval of 6 months. For example, anything committed within the first six months of a project will be in the first version, and so on. We then build our code suggestion task mimicking how a developer develops code in an evolving software—based on the past project history, developers add new code. To implement that we train our language model on past project versions and test it on the most recent version, at method granularity. However, it is quite difficult for any language model to generate a method from the scratch if the method is so new that even the method signature (i.e., method declaration statement consisting of method name and parameters) is not known. Thus, during testing, we only focus on the methods that the model has seen before but some new tokens are added to it. This is similar to the task when a developer edits a method to implement a new feature or bug-fix.

Since we focus on generating the code for every method, we train/test the code prediction task at method level—each method is similar to a sentence and each token in the method is equivalent to a word. Thus, we ignore the code outside the method scope like global variables, class declarations, etc. We further clean our dataset by removing user-defined "String" tokens as they increase the diversity of the vocabularies significantly, although having the same type. For example, the word sequences "Hello World!" and "Good wishes for ACL2018!!" have the same type java.lang.String.VAR.

4.4.2.2 Retrieving Token Type

For every token w in a method, we extract its type information $s(w)$. A token type can be Java built-in data types (e.g., *int, double, float, boolean* etc.,) or user or framework

[12] https://play.google.com/store?hl=en

defined classes (e.g., *java.lang.String, io.segment.android.flush.FlushThread* etc.). We extract such type information for each token by parsing the Abstract Syntax Tree (AST) of the source code[13]. We extract the AST type information of each token using Eclipse JDT framework[14]. Note that, language keywords like for, if, etc. are not associated with any type. Next, we prepare the type corpus by replacing the variable names with corresponding type information. For instance, if variable var is of type *java.lang.Integer*, in the type corpus we replace *var* by *java.lang.Integer*. Since multiple packages might contain classes of the same name, we retain the fully qualified name for each type[15].

4.4.2.3 Code Corpus Statistics

In our corpus, the total number of distinct words in vocabulary is 38,297; the number of unique AST type (including all user-defined classes) is 14,177; the number of tokens is 1,440,993. The number of instances used for train and testing is 26,600 and 3,546. Among these 38,297 vocabulary words, 37,411 are seen at training time while the rests are new.

4.4.2.4 Configuration

To train both *type model* and *entity composite model*, we use forward and backward LSTM (See Section 4.2) and combine them at the inference/generation time. We train 300-dimensional word embedding for each token as described in Section 4.4 initialized by GLOVE (Pennington et al., 2014). Our LSTM is single layered and the hidden size is 300. We implement our model on using PyTorch and Python 3.5. Our training corpus size 26,600 and we do not split it further into smaller train and development set; rather we use them all to train for one single epoch and record the result on the test set.

[13]AST represents source code as a tree by capturing its abstract syntactic structure, where each node represents a construct in the source code.

[14]https://www.eclipse.org/jdt/

[15]Also the AST type of a very same variable may differ in two different methods. Hence, the context is limited to each method.

Model	Dataset (Code Corpus)	Vocabulary Size	Perplexity
SLP-Core	original	38,297	3.40
fLSTM	original	38,297	21.97
fLSTM [*type model*]	modified type	14,177	7.94
fLSTM with type feature	original	38,297	20.05
our model (fLSTM)	original	38,297	**12.52**
bLSTM	original	38,297	7.19
bLSTM [*type model*]	modified type	14,177	2.58
bLSTM with type feature	original	38,297	6.11
our model (bLSTM)	original	38,297	**2.65**

Table 4.2: Comparing the performance of code generation task. All the results are on the test set of the corresponding corpus. fLSTM, bLSTM denotes forward and backward LSTM respectively. SLP-Core refers to Hellendoorn and Devanbu (2017).

4.4.2.5 Baselines

Our first baseline is standard LSTM language model which we also use to train our modules (see 'Configuration' in 4.4.2.4). Similar to our second baseline for recipe generation we also consider LSTM with the type information added as more features[16] as our another baseline. We further compare our model with state-of-the-art token-based language model for source code SLP-Core (Hellendoorn and Devanbu, 2017).

4.4.2.6 Results of Code Generation:

Table 4.2 shows that adding type as simple features does not guarantee a significant performance improvement while our proposed method significantly outperforms both forward and backward LSTM baselines. Our approach with backward LSTM has 40.3% better perplexity than original backward LSTM and forward has 63.14% lower (i.e., better) perplexity than original forward LSTM. With respect to SLP-Core performance, our model is 22.06% better in perplexity. We compare our model with SLP-Core details in case study-2.

[16] LSTM with type is same as *entity composite model*.

4.5 Quantitative Error Analysis

To understand the generation performance of our model and interpret the meaning of the numbers in Table 4.1 and 4.2, we further perform the following case studies.

4.5.1 Case Study-1: Recipe Generation

As the reduction of the perplexity does not necessarily mean the improvement of the accuracy, we design a "fill in the blank task" task to evaluate our model. A blank place in this task will contain an ingredient and we check whether our model can predict it correctly. In particular, we choose six ingredients from different frequency range (low, mid, high) based on how many times they have appeared in the training corpus. Following Table shows two examples with four blanks (underlined with the true answer).

Example fill in the blank task
1. Sprinkle <u>chicken</u> pieces lightly with <u>salt</u>.
2. Mix egg and <u>milk</u> and pour over <u>bread</u>.

We further evaluate our model on a multiple choice questioning (MCQ) strategy where the fill in the blank problem remains same but the options for the correct answers are restricted to the six ingredients. Our intuition behind this case-study is to check when there is an ingredient whether our model can learn it. If yes, we then quantify the learning using standard *accuracy* metric and compare with the state-of-the-art model to evaluate how much it improves the performance. We also measure how much the accuracy improvement depends on the training frequency.

Table 4.3 shows the result. Our model outperforms the fill in the blank task for both cases, i.e., without any options (free-form) and MCQ. Note that, the percentage of improvement is inversely proportional to the training frequencies of the ingredients—less-frequent ingredients achieve a higher accuracy improvement (e.g., "Apple" and "Tomato"). This validates our intuition of learning to predict the type first more accurately with

Ingredient	Train Freq.	#Blanks	Accuracy			
			Free-Form		MCQ	
			AWD_LSTM	Our	AWD_LSTM	Our
Milk	14, 136	4,001	26.94	**59.34**	80.83	**94.90**
Salt	33,906	9,888	37.12	**62.47**	89.29	**95.75**
Apple	7,205	720	1.94	**30.28**	37.65	**89.86**
Bread	11,673	3,074	32.43	**52.64**	78.85	**94.53**
Tomato	12,866	1,815	2.20	**35.76**	43.53	**88.76**
Chicken	19,875	6,072	22.50	**45.24**	77.70	**94.63**

Table 4.3: Performance of fill in the blank task.

lower vocabulary set and then use conditional probability to predict the actual entity considering the type as a prior.

4.5.2 Case Study-2: Code Generation

Programming language source code shows regularities both in local and global context (e.g., variables or methods used in one source file can also be created or referenced from another library file). SLP-Core (Hellendoorn and Devanbu, 2017) is a state-of-the-art code generation model that captures this global and local information using a nested cache based n-gram language model. They further show that considering such code structure into account, a simple n-gram based SLP-Core outperforms vanilla deep learning based models like RNN, LSTM, etc.

In our case, as our example instance is a Java method, we only have the local context. Therefore, to evaluate the efficiency of our proposed model, we further analyze that exploiting only the type information are we even learning any global code pattern? If yes, then how much in comparison to the baseline (SLP-Core)? To investigate these questions, we provide all the full project information to SLP-Core (Hellendoorn and Devanbu, 2017) corresponding to our train set. However, at test-time, to establish a fair comparison, we consider the perplexity metric for the same methods. SLP-Core achieves a perplexity 3.40 where our backward LSTM achieves 2.65. This result shows that appropriate type information can actually capture many inherent attributes which can be exploited to

build a good language model for programming language.

4.6 Conclusion

Language model often lacks in performance to predict entity names correctly. Applications with lots of named entities, thus, obviously suffer. In this work, we propose to leverage the type information of such named entities to build an effective language model. Since similar entities have the same type, the vocabulary size of a type based language model reduces significantly. The prediction accuracy of the type model increases significantly with such reduced vocabulary size. Then, using the entity type information as prior we build another language model which predicts the true entity name according to the conditional probability distribution. Our evaluation and case studies confirm that the type information of the named entities captures inherent text features too which leads to learn intrinsic text pattern and improve the performance of overall language model.

4.7 Summary

In this chapter, we showed that modeling entity-type information improves the language modeling. We developed a joint inference model to compute the probability of a next token/word to generate. We use simple rule-based tools to find the entity-type information. Extensive experiments on two language generation tasks demonstrates the effectiveness of our approach. Our future works include the exploration of other sources of both domain/language universal or domain/language specific information to improve the generation more.

CHAPTER 5

Retrieve and Augment Relevant Policy Documents for Question Answering on Privacy Policies

In Chapter 2, we have discussed *Type-2* ***corpora level*** auxiliary supervision and in Chapter 3, and 4, *Type-2* ***feature level*** auxiliary supervision. In the following two Chapters, we will be presenting *Type-3* ***instance level*** auxiliary supervision. Particularly, in this Chapter, we first improve the instance retriever developed in Chapter 3 and then use them to retrieve and augment new instances in an example data imbalanced (i.e., a low-resource) application "question answering on privacy policies". New instances brought by our approach significantly enhance the task and scored a new state-of-the-art performance. This Chapter is based on our work Parvez et al. (2022).

5.1 Introduction

Understanding privacy policies that describe how user data is collected, managed, and used by the respective service providers is crucial for determining if the conditions outlined are acceptable. Policy documents, however, are lengthy, verbose, equivocal, and hard to understand (McDonald and Cranor, 2008; Reidenberg et al., 2016). Consequently, they are often ignored and skipped by users (Commission et al., 2012; Gluck et al., 2016).

To help the users better understand their rights, privacy policy QA is framed to answer sentence selection task, essentially a binary classification task to identify if a policy text segment is relevant or not (Harkous et al., 2018). However, annotating policy documents requires expertise and domain knowledge, and hence, it is costly and hard to obtain. Moreover, as most texts in policy documents are not relevant, the data is heavily

Segmented policy document $\boldsymbol{S}$
($\boldsymbol{s}_1$) We do not sell or rent your personal information to third parties for their direct marketing purposes without your explicit consent.
($\boldsymbol{s}_n$) ...We will not let any other person, including sellers and buyers, contact you, other than through your ...
Queries $\boldsymbol{I}$ annotating the red segment as irrelevant
($\boldsymbol{i}_1$) How does Fiverr protect freelancers' personal information?
($\boldsymbol{i}_2$) What type of identifiable information is passed between users on the platform?
Queries $\boldsymbol{R}$ annotating the red segment as relevant
($\boldsymbol{r}_1$) What are the app's permissions?
($\boldsymbol{r}_2$) What type of permissions does the app require?
Queries $\boldsymbol{D}$ that annotators disagree about relevance
($\boldsymbol{d}_1$) Do you sell my information to third parties?
($\boldsymbol{d}_2$) is my information sold to any third parties?

Table 5.1: QA (sentence selection) from a policy document $\boldsymbol{S}$. **Sensitive**: For queries $\boldsymbol{R}$ and $\boldsymbol{I}$, annotators at large tagged sentence $\boldsymbol{s}_1$ as relevant, and irrelevant respectively. On the other hand, sentence $\boldsymbol{s}_n$, though analogous to $\boldsymbol{s}_1$ in meaning, was never tagged as relevant. **Ambiguous**: For queries $\boldsymbol{D}$, experts interpret $\boldsymbol{s}_1$ differently and disagree on their annotations.

imbalanced. For example, the only existing dataset, PrivacyQA (Ravichander et al., 2019) has 1,350 questions in the training dataset, and the average number of answer sentences is 5, while the average length of policy documents is 138 sentences.

We mitigate data imbalance by augmenting positive QA examples in training set in this work. Specifically, we develop automatic retrieval models to supplement relevant policy sentences for each user query. The queries we keep unchanged as they usually have minor variants and are limited to a few forms (Wilson et al., 2016).

Augmenting privacy policies is challenging. First, privacy statements often describe similar information (Hosseini et al., 2016). Thus, their annotations are sensitive to small changes in the text (see Table 5.1), which may not be tackled using the existing augmentation methods based on data synthesis. For example, Kumar et al. (2020b) identifies that even linguistically coherent instances augmented via generative models such as GPT-2 (Anaby-Tavor et al., 2020) do not preserve the class labels well. Hence, we consider a retrieval-based approach to augment the existing policy statements to address this. Given a pre-trained LM and a small QA dataset, we first build a dense sentence

retriever (Karpukhin et al., 2020b). Next, leveraging an unlabeled policy corpus with 0.6M sentences crawled from web applications, we perform a coarse one-shot sentence retrieval for each query in the QA training set. To filter the noisy candidates retrieved, we then train a QA model (as an oracle) using the same pre-trained LM and data and couple it with the retriever.

Second, privacy policies are ambiguous; even skilled annotators dispute their interpretations, e.g., for at least 26% questions in PrivacyQA, experts disagree on their annotations (see Table 5.1). Therefore, a single retriever model may not capture all relevant policy segments. To combat this, we propose a novel retriever ensemble technique. Different pre-trained models learn distinct language representations due to their pre-training objectives, and hence, retriever models built on them can retrieve a disjoint set of candidates (verified in Section 6.4). Therefore, we build our retrievers and oracles based on multiple different pre-trained LMs (See Figure 6.1). Finally, we train a user-defined QA model on the aggregated corpus using them.

We evaluate our framework on the PrivacyQA benchmark. We elevate the state-of-the-art performance significantly (10% F1) and achieve a new one (50% F1). Furthermore, our ablation studies provide an insightful understanding of our model. We will release all data and code upon acceptance.

5.2 Methodology

The privacy policy QA is a binary classification task that takes a user query q, a sentence p from policy documents and outputs a binary label $z \in \{0, 1\}$ that indicates if q and p are relevant or not. As most sentences p are labeled as negative, our goal is to retrieve relevant sentences to augment the training data and mitigate the data imbalance issue. Given a QA training dataset $D = \{(q_i, p_i, z_i)\}_{i=1}^{m}$, for each question in D, we (1) retrieve positive sentences from a large unlabeled corpus. (2) filter the noisy examples using oracle models and aggregate final candidates. The final candidates are combined with the base data D to train the QA models. We use an ensemble of retrievers and oracles built

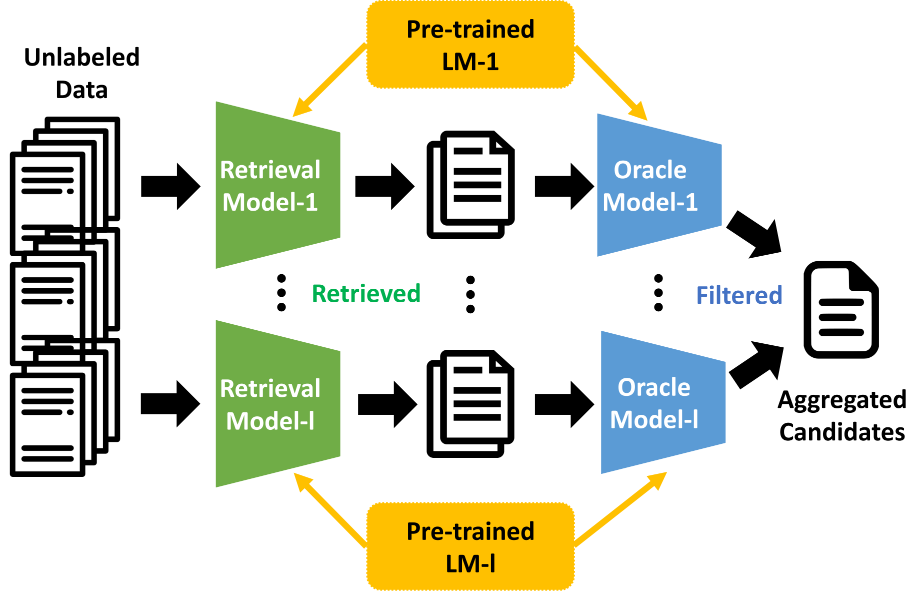


Figure 5.1: Our framework. Given a pre-trained LM, we train a (i) retriever, (ii) QA model (oracle) both on the small-size labeled data. From an unlabeled corpus, we first, retrieve the coarse relevant sentences (positive examples) for the queries in the training set and use the oracle to filter out noisy ones. We repeat this for multiple different pre-trained LMs. Finally we aggregate them to expand the positive examples in the training set and learn any user-defined final QA model.

upon various pre-trained LMs throughout the whole process. Details are discussed in the following.

5.2.1 Policy Document Retriever

Our retriever module is built upon the Dense Passage Retriever (DPR) model Karpukhin et al. (2020b). It consists of two encoders $Q(\cdot)$ and $P(\cdot)$ that encode the queries and the policy sentences, respectively. The relevance of a query q and a policy sentence p is calculated by the dot product of $Q(q)$ and $P(p)$, i.e., $\text{sim}(q, p) = Q(q)^T \cdot P(p)$. For each positive pair in D, it optimizes only the cross-entropy loss with in-batch negatives Henderson et al. (2017); Parvez et al. (2021). We train a retriever $\mathcal{R}_L$ on D, where the encoders in $\mathcal{R}_L$ are initialized with a pre-trained LM L. At inference, $\mathcal{R}_L$ retrieves the top-k most relevant policy sentences from an unlabeled corpus of policy sentences $\mathcal{P} = \{p_1, \dots, p_M\}$ for each query q_i in D, i.e., $\mathcal{R}_L(\{q_i\}_{i=1}^m, \mathcal{P}, k) = \{(q_i, p_j, 1) :$

$i \in [m], p_j \in \mathcal{P}_{\text{top}}(q_i, k)\}$, where $\mathcal{P}_{\text{top}}(q_i, k) = \arg\max_{\mathcal{P}' \subset \mathcal{P}, |\mathcal{P}'|=k} \sum_{p \in \mathcal{P}'} \text{sim}(q_i, p)$.

5.2.2 Filtering Oracle

To filter out the noisy retrievals from $\mathcal{R}_L(\{q_i\}_{i=1}^m, \mathcal{P}, k)$, we train a QA (i.e., a text-classification) model ($\mathcal{Q}_L$) using the training data D as an oracle to predict whether a query q and a (retrieved) policy sentence p are relevant or not (i.e., $\mathcal{Q}_L(q, p) \in \{0, 1\}$). Note that both $\mathcal{Q}_L$ and retriever $\mathcal{R}_L$ are built upon the same pre-trained LM L (e.g., BERT). However, the retriever model is a bi-encoder model that can pre-encode, index, and rank a large number of candidates. In contrast, our filtering oracle model is a cross-encoder text-classifier (e.g., BERT) that can achieve comparatively higher performances Humeau et al. (2019) (i.e., hence better as a filter) but can not pre-encode and hence can not be used for large scale retrieval (more differences are in Section 5.4). We verify the effectiveness of oracle filtering in Section 5.4. We denote retrieval outputs after filtering as $\mathcal{D}_L = \{(q, p, 1) : \mathcal{Q}_L(q, p) = 1, \forall (q, p, 1) \in \mathcal{R}_L(\{q_i\}_{i=1}^m, \mathcal{P}, k)\}$.

5.2.3 Retriever Ensemble and Data Augmentation

Unlike other NLP domains, a privacy policy sentence can frequently have multiple interpretations (see Table 5.1). Hence, a single retrieved corpus $\mathcal{D}_L$ may not capture all relevant candidates covering such diverse interpretations. To this end, we use a set of pre-trained LMs $\mathcal{L} = \{L_1, \ldots, L_l\}$ and aggregate all the corresponding retrieved corpora, $\mathcal{D}_{\text{aug}} = \bigcup_{L \in \mathcal{L}} \mathcal{D}_L$. In Section 6.4, we show that retrieved corpora using multiple pre-trained LMs with different learning objectives can bring a different set of relevant candidates. Lastly, we aggregate $\mathcal{D}_{\text{aug}}$ with D (i.e., final train corpus $\mathcal{T} = \mathcal{D}_{\text{aug}} \cup D$) and train our final QA model with user specifications (e.g., architecture, pre-trained LM).

5.3 Experiments

In this section, we evaluate our approach and present the findings from our analysis.

	PrivacyQA
Source	Mobile application
Question annotator	Mechanical Turkers
Form of QA	Sentence selection
Answer type	A list of sentences
# Unique policy docs	train: 27, test: 8
# Unique questions	train: 1350, test: 400
# QA instances	train: 185k, test: 10k
Avg Q. Length	train: 8.42 test: 8.56
Avg Doc. Length	train: 3.1k, test: 3.6k
Avg Ans. Length	train: 124, test: 153

Table 5.2: Brief summary of PrivacyQA benchmark.

Settings We evaluate our approach on PrivacyQA benchmark (a brief statistics of this dataset is in Table 5.2) and recall that this is in fact a text classification task. Following Ravichander et al. (2019), we use *precision*, *recall*, and *F1 score* as the evaluation metrics. As for the retrieval database $\mathcal{P}$, we crawl privacy policies from the most popular mobile apps spanning different app categories in the Google Play Store and end up with 6.5k documents (0.6M statements). By default, all retrievals use top-10 candidates w/o filtering. All data/models/codes are implemented using (i) Huggingface Transformers (Wolf et al., 2019b), (ii) DPR (Karpukhin et al., 2020b) libraries.

Baselines We fine-tune three pre-trained LMs on PrivacyQA as baselines: (i) *BERT*: Our first baseline is BERT-base-uncased (Devlin et al., 2019) which is pre-trained on generic NLP textual data. A previous implementation achieves the exiting state-of-the-art performance (BERT+Unams. in Ravichander et al. (2019)). (ii) *PBERT*: We adapt *BERT* to the privacy domain by fine-tuning it using masked language modeling on a corpus of 130k privacy policies (137M words) collected from apps in the Google Play Store (Harkous et al., 2018). Note that the retrieval database $\mathcal{P}$ is a subset of this data that is less noisy and crawled as a recent snapshot (more in Section 5.4) (iii) *SimCSE*: We take the *PBERT* model and apply the unsupervised contrasting learning SimCSE (Gao et al., 2021) model on the same 130k privacy policy corpus. We also consider three other retrieval augmented QA models based on individual pre-trained LM without ensemble: (iv) *BERT*-

Method	Oracle	Precision	Recall	F1
Human	-	68.8	69.0	68.9
W/o data augmentation				
BERT+Unans.	-	44.3	36.9	39.8
BERT (reprod.)		$48.0_{\pm 2.04}$	$37.7_{\pm 1.19}$	$42.2_{\pm 1.54}$
PBERT		$51.2_{\pm 0.41}$	$42.7_{\pm 0.64}$	$46.6_{\pm 0.42}$
SimCSE		$48.4_{\pm 0.80}$	$41.4_{\pm 0.67}$	$44.7_{\pm 0.71}$
Retriever augmented				
BERT-R	✗	$39.0_{\pm 0.78}$	$52.4_{\pm 1.65}$	$44.7_{\pm 0.40}$
	✓	$48.1_{\pm 1.38}$	$44.7_{\pm 0.85}$	$46.3_{\pm 0.46}$
PBERT-R	✗	$48.7_{\pm 1.91}$	$44.1_{\pm 1.79}$	$46.3_{\pm 1.61}$
	✓	$49.2_{\pm 1.56}$	$44.9_{\pm 1.97}$	$47.0_{\pm 1.21}$
SimCSE-R	✗	$47.0_{\pm 2.14}$	$44.5_{\pm 2.41}$	$45.7_{\pm 1.86}$
	✓	$48.6_{\pm 2.18}$	$43.9_{\pm 1.21}$	$46.1_{\pm 1.56}$
Ensemble retriever augmented				
Baseline-E	✗	$22.2_{\pm 0.80}$	$54.4_{\pm 0.80}$	$31.4_{\pm 0.78}$
REDCODER	✓	$47.4_{\pm 0.61}$	$50.5_{\pm 2.24}$	$\mathbf{48.9}_{\pm 0.78}$
REDCODER-EXT	✓	$51.0_{\pm 0.38}$	$48.7_{\pm 0.91}$	$\mathbf{49.8}_{\pm 0.70}$

Table 5.3: Test performances on PrivacyQA (mean$_{\pm\text{std}}$). BERT+Unans. refers to the previous SOTA performance (Ravichander et al., 2019). Retrieved candidates improves all the baseline QA models, especially when being filtered. Our ensemble retriever approach combines them and achieves the highest gains.

R: $\mathcal{L} = \{BERT\}$, (v) *PBERT-R*: $\mathcal{L} = \{PBERT\}$, (vi) *SimCSE-R*: $\mathcal{L} = \{SimCSE\}$. We first construct $\mathcal{T}$ (both settings: w/ and w/o oracle) and fine-tune on it the corresponding pre-trained LM as the final QA model. Finally, we consider one more ensemble retrieval augmented baseline (vii) *Baseline-E*, which is precisely the same as ours (settings below), except there are no intermediate filtering oracles.

Ours We construct out augmented corpus $\mathcal{T}$, discussed in Section 5.2.3, using the (i) all three aforementioned pre-trained LMs: $\mathcal{L} = \{BERT, PBERT, SimCSE\}$ (ii) domain adapted models only: $\mathcal{L} = \{PBERT, SimCSE\}$. For brevity, we call them: Ensemble Retriever Augmentation (ERA) and Ensemble Retriever Augmentation–Domain Adapted ERA-D. By default, we fine-tune *SimCSE* as the final QA model.

Query Type	%	B	PB	S	ERA
Data Collection	42	45	**46**	**46**	**48**
Data Sharing	25	**43**	37	41	**43**
Data Security	11	**65**	61	60	60
Data Retention	4	**52**	35	35	**56**
User Access	2	**72**	48	31	61
User Choice	7	41	**60**	42	31
Others	9	36	45	**52**	**55**
Overall	100	45	47	48	**49**

Table 5.4: F1-score breakdown (values are in Appendix). B, PB, S refers to retrievers *BERT-R*, *PBERT-R*, and *SimCSE-R*. Different models performs better for different types (black-bold). ERA combines them and enhances performances for all categories (except: red).

5.3.1 Results and Analysis

The results are listed in Table 5.3. Overall, domain adapted models *PBERT* and *SimCSE* excel better than the generic *BERT* model. The retrieval augmented models enhance the performances more, specially the recall score, as they are added as additional positive examples. However, they might contain several noisy examples (see Table 5.6 and Table 5.7), and filtering those out improves the precision scores for all three retrievers. Finally, ERA and ERA-D aggregate these high-quality filtered policies–leading toward the highest gain (10% F1 from the previous baseline) and a new state-of-the-art result with an F1 score of ∼50. Note that *Baseline-E* unifies all the candidates w/o any filtering performs considerably worse than all other models, including each retrieval model: *Baseline-E* augments more candidates as positives, which explains the highest recall score; in the meantime, as it does not filter any, the corresponding precision score is oppositely the lowest.

Table 5.4 and Table 5.5 show the performance breakdown for different query types. For questions related to data collection, data sharing, and data security, the performance difference among the models is relatively small ($\leq 5\%$ F1); for data retention and user access, *BERT-R*, that is pre-trained on generic NLP texts, performs significantly well ($> 15\%$ F1), possibly because the answers to these query types focus on providing

Query Type	total	B	PB	S	REDCODER
Data Collection	6280	1901	1157	1186	1806
Data Sharing	4734	1332	777	1092	1268
Data Security	994	416	399	423	393
Data Retention	453	150	98	110	173
User Access	221	89	47	43	87
User Choice	493	91	49	24	55
Others	28	2	1	2	4
Overall	10332	3135	2084	2334	2935

Table 5.5: Number of correct predictions. Note that F1-score is not proportional to the accuracy. B, PB, S refers to retrievers *BERT-R*, *PBERT-R*, and *SimCSE-R*. Different models performs better for different types (black-bold). ERA combines them and enhances performances for all categories

numerical evidence for the questions (e.g., How many days the data are retained?) that is less irrelevant to the domain of privacy policies; and for other types of questions the domain adapted models performs better (> 15% F1). In general, the individual retrieval augmented models learned w/ different corpora and objectives perform at different scales for each type, and combining their expertise, ERA enhances the performances for all types.

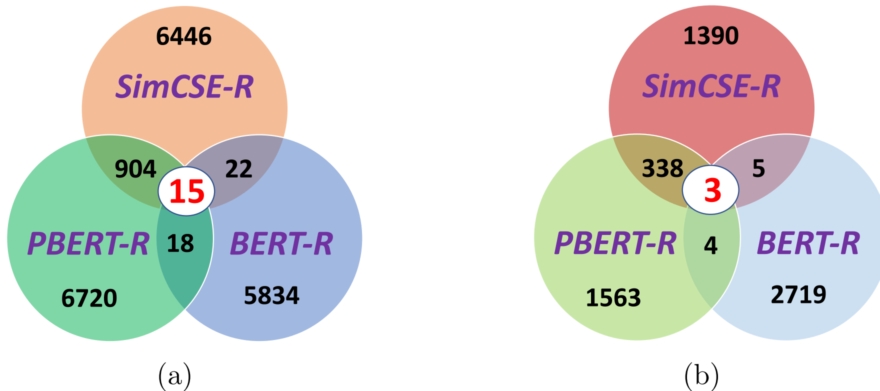


Figure 5.2: Venn diagram of low mutual agreement (<1%) among retrievers (a); even amplified after filtering (b).

Next, we show the Venn diagram of overlapping retrievals in Figure 5.2. Although policy statements describe similar information (i.e., have common phrases), they are often verbose and equivocal (i.e., multiple-different interpretations). Consequently, retrievers w/ different objectives and training corpora rank them differently. Therefore, although being

retrieved from the same corpus, candidates retrieved by different models rarely match fully but may have much overlapping information and improve the model performances equitably. For example, from Table 5.3, while the performances of *BERT-R*, *PBERT-R* and *SimCSE-R* w/ filtering are similar (∼46), their overlapping (exact match) is $< 1\%$ from Figure 5.2 (qualitative examples in Table 5.6 and Table 5.7). At the same time, the raw retrieval corpora have a high BLEU score of (≥ 0.78). This validates our hypothesis that retrievers built upon different pre-trained LMs learn diverse representations and consequently retrieve diverse candidates.

Q: who all has access to my medical information?
Gold: i) Apple HealthKit to health information and to share that information with your healthcare providers. ii) Your use of our Application with that healthcare institution may be subject to that healthcare institution's policies and terms.
Correct Retrievals: (i) We may share your information with other health care providers, laboratories, government agencies, insurance companies, organ procurement organizations, or medical examiners. -(SimCSE-R) (ii) Do not sell your personal or medical information to anyone. -(BERT-R) (iii) Lab, Inc will transmit personal health information to authorized medical providers. -(PBERT-R) (iv) To organ and tissue donation requests: By law, we can disclose health information about you to organ procurement organizations. -(BERT-R)
Incorrect Retrievals: (i) However, we take the protection of your private health information very seriously. -(SimCSE-R) (ii) All doctors, and many other healthcare professionals, are included in our database. -(PBERT-R) (iii) You may be able to access your pet's health records or other information via the Sites. -(BERT-R) (iv) will say "yes" unless a law requires us to disclose that health information.-(BERT-R) (v) do not claim that our products "cure" disease.-(BERT-R) (vi) Has no access to your database password or any data stored in your local database on your devices.-(BERT-R)

Table 5.6: A fraction of retrieval examples (i).

5.4 Ablation Study

Are oracles needed? From Table 5.4, in general, aggregating retrievals with oracle filtering enhances model performances than crude additions. Qualitative examples are in

Q: do you sell my photos to anyone?

Gold: i) We use third-party service providers to serve ads on our behalf across the Internet and sometimes on the Sites. (ii) These companies may use your personal information to enhance and personalize your shopping experience with us, to communicate with you about products and events that may be of interest to you and for other promotional purposes. iii) Your use of our Application with that healthcare institution may be subject to that healthcare institution's policies and terms. (iv) We may share personal information within our family of brands. (v) From time to time we share the personal information we collect with trusted companies who work with or on behalf of us. (vi) No personally identifiable information is collected in this process. (vii) We use third-party service providers to serve ads on our behalf across the Internet and sometimes on our Sites and Apps.

Correct Retrievals: (i) The Application does not collect or transmit any personally identifiable information about you, such as your name, address, phone number or email address. -(SimCSE-R) (ii) Some of this information is automatically gathered, and could be considered personally identifiable in certain circumstances, however it will generally always be anonymised prior to being viewed by Not Doppler, and never sold or shared. -(BERT-R) (iii) We also use the Google AdWords service to serve ads on our behalf across the Internet and sometimes on this Website. -(PBERT-R) (iv) To organ and tissue donation requests: By law, we can disclose health information about you to organ procurement organizations. -(BERT-R)

Incorrect Retrievals: (i) When you upload your photos to our platform or give us permission to access the photos stored on your device, your photo content may also include related image information such as the time and the place your photo was taken and similar "metadata" captured by your image capture device. -(SimCSE-R) (ii) These are not linked to any information that is personally identifiable.-(BERT-R)

Table 5.7: A fraction of retrieval examples (ii).

Table 5.6 and Table 5.7.

A common oracle. Performances of ERA (last row in Table 5.4) with a common oracle based on *SimCSE* for all the retrievers regardless of their corresponding pre-trained models are 49.2, 45.2, and 47.1, respectively–validating the requirement of filtering using the corresponding pre-trained LM.

Other pre-trained LM as the final QA model. Fine-tuning *PBERT* instead of *SimCSE* on $\mathcal{T}$ (last two rows in Table 5.3) becomes: 47.0, 47.1, 47.0 and 51.0, 45.9, 48.3, respectively. This shows that our approach is generic and enhances the performance regardless of the end model.

Which pre-trained LMs to use? Table 5.4 shows ERA-D that combines fewer pre-trained LMs may even outperform the one with more models, ERA. Though here we consider a simple approach (in-domain) for selecting the potential subset of models, this paves a new direction for future research (e.g., Parvez and Chang (2021)).

Qualitative examples. Table 5.6 and Table 5.7 show some example retrievals of different models. They are distinct from expert annotated ones and can bring auxiliary knowledge.

Privacy Policy Data Crawling & Retrieval Statistics We crawl our English retrieval corpus from Google App Store using the Play Store Scraper[1].

Query Type	No. of Retrieval
Data Collection	2893
Data Sharing	1848
Data Security	891
Data Retention	542
User Access	145
User Choice	335
Others	14

Table 5.8: Retrieval statistics per query type.

Table 5.8 shows the statistics of our (ERA) augmented corpus per each question category in the PrivacyQA training set.

Difference Between the Filtering Oracle and the Retriever The retriever model is a bi-encoder model whose model parameters are fine-tuned with in-batch negative loss (discussed in Section 5.2.1 in the main paper), hyper-parameters are tuned based on average rankings (DPR OFFICIAL PAGE) and that can pre-encode, index and rank a large number of candidates while our filtering oracle model is a cross-encoder text-classifier (e.g., BERT) that is fine-tuned w/o any additional in-batch negatives and in-general achieves comparatively higher performance (Humeau et al., 2019) (i.e, hence better as a filter) but can not pre-encode and hence can not be used for large scale retrieval.

Difference Between Pre-training and Retrieval Corpus 130k documents were

[1]https://github.com/danieliu/play-scraper

collected before 2018 and by that time, the GDPR[2] and CCPA[3] were not enforced by then. Thus, the 130k documents are out-of-date and some content might not be comprehensive as the retrieval corpus. Besides, the 130k documents provided by (Harkous et al., 2018) contains some noises since we observe that the documents are not all written in English. However, as the data size is larger, we still use it for pre-training. In contrast, our corpus was collected after 2020 and we filtered out some possible noises (e.g., filtering out non-English document) while crawling.

5.5 Related Works

A line of works focuses on using NLP techniques for privacy policies Wilson et al. (2016); Harkous et al. (2018); Zimmeck et al. (2019); Bui et al. (2021); Ahmad et al. (2021b). Besides the QA tasks as sentence selection, Ahmad et al. (2020b) propose another SQuAD-like Rajpurkar et al. (2016) privacy policy reading comprehension dataset for a limited number of queries. Oppositely, we focus on the more challenging one, which allows unanswerable questions and "non-contiguous" answer Ravichander et al. (2021). In relevant literature works, retrieval augmented methods are applied in various contexts including privacy policies (e.g., Van et al. (2021); Keymanesh et al. (2021); Yang et al. (2020)). Non-retrieval data aggregation has also been studied under different NLP contexts (e.g., bagging Breiman (1996), meta learning Parvez et al. (2019a)). However, we uniquely aggregate the retriever outputs using different pre-trained language models.

5.6 Limitations

In this paper, we show that leveraging multiple different pre-trained LMs can augment high-quality training examples and enhance the QA (sentence selection) task on privacy policies. Our approach is generic and such unification of different kinds of pre-trained

[2]https://gdpr-info.eu/

[3]https://oag.ca.gov/privacy/ccpa

language models for text data augmentation can improve many other low-resourced tasks or domains. However, it is possible that our approach:

- may not work well on other scenarios (e.g., domains/language or tasks etc.,).
- subject to the choice of particular set of models. For example, as mentioned in Section, 5.4, fine-tuning pre-trained models other than *SimCSE* Gao et al. (2021) as the final QA model achieve lower gain.
- may not work for certain top-k retrievals. For example, from Table 5.9, we get different results with different scales for variable top-k values (e.g., top-10, top-100).

Method	Filter	top-k	Precision	Recall	F1
BERT-R	✗	10	39.9	50.8	44.7
	✓	10	46.5	45.5	46.0
PBERT-R	✗	10	48.4	45.6	46.9
	✓	10	46.9	43.3	45.1
	✗	50	47.8	45.5	46.7
	✓	50	49.5	46.3	47.8
SimCSE-R	✗	10	48.4	47.2	47.8
	✓	10	49.4	44.8	47.0
	✗	100	42.1	41.3	41.7
	✓	100	51.0	45.2	47.9

Table 5.9: Model performances with and without filtering with top-k. In general, without filtering, augmenting the retrieved candidates enhances recall but may reduce the precision (and hence may not improve the overall F1). Filtering, however improves the performance specially with larger top-k candidates.

5.7 Conclusion

We develop a noise-reduced retrieval-based data augmentation method that combines different pre-trained language models. Although we focus on the privacy policy domain, our approach can also be applied to other domains. We will leave the exploration as future work.

5.8 Summary

Prior studies in privacy policies frame the question answering (QA) tasks as identifying the most relevant text segment or a list of sentences from the policy document for a user query. However, annotating such a dataset is challenging as it requires specific domain expertise (e.g., law academics). Even if we manage a small-scale one, a bottleneck that remains is that the labeled data are heavily imbalanced (only a few segments are relevant) –limiting the gain in this domain. In this Chapter, we propose a a novel retrieval based method for augmenting the rare class examples. Using our augmented data on the PrivacyQA benchmark, we elevate the existing baseline by a large margin (10% F1) and achieve a new state-of-the-art F1 score of 50%. Our ablation studies provide further insights into the effectiveness of our approach.

CHAPTER 6

Selecting and Augmenting Relevant Text Spans for Fast and Robust Text Classification

In this Chapter, we design a generic framework for learning a fast and robust text classification model that achieves high accuracy under different budgets (i.e., time needed for inference). We dynamically filter a large fraction of unimportant words by a low-complexity selector such that any high-complexity classifier only needs to process a small fraction of text, relevant for the target task. Next, we vary the selection rate of our proposed selector model and generate different versions of filtered text (fractured but relevant). We aggregate them as new training examples (i.e., *Type-3* ***instance level*** aggregation) for the end classifier, allowing it to achieve competitive performance on any fractured sentences at inference time. This Chapter is based on our work Parvez et al. (2019b).

6.1 Introduction

Recent advances in deep neural networks (DNNs) have achieved high accuracy on many text classification tasks. These approaches process the entire text and encode words and phrases in order to perform target tasks. While these models realize high accuracy, the computational time scales linearly with the size of the documents, which can be slow for a long document. In this context, various approaches based on modifying the RNN or LSTM architecture have been proposed to speed up the process (Seo et al., 2017; Yu et al., 2017). However, the processing in these models is still fundamentally sequential and needs to operate on the whole document which limits the computational gain. In contrast

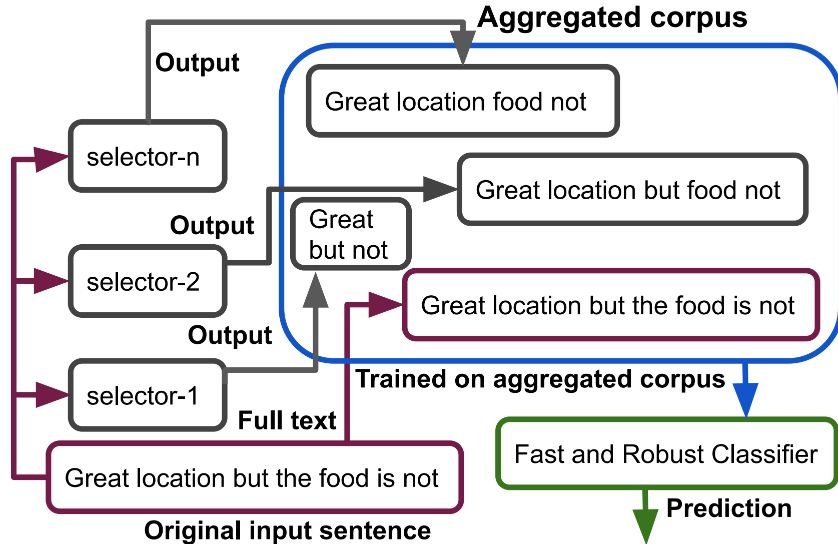


Figure 6.1: Our proposed framework. Given a selection rate, a *selector* is designed to select relevant words and pass them to the *classifier*. To make the classifier robust against fractured sentences, we aggregate outputs from different *selectors* and train the *classifier* on the aggregated corpus.

to previous approaches, we propose a novel framework for efficient text classification on long documents that mitigates sequential processing. The framework consists of a *selector* and a *classifier*. Given a selection budget as input, the *selector* performs a coarse one-shot selection deleting unimportant words and pass the remainder to the *classifier*. The *classifier* then takes the sentence fragments as an input and performs the target task. Figure 6.1 illustrates the procedure. This framework is general and agnostic to the architecture of the downstream *classifier* (e.g., RNN, CNN, Transformer).

However, three challenges arise. First, to build a computationally inexpensive system, the *selector* must have negligible overhead. We adopt two effective yet simple architectures to design *selector*s based on word embeddings and bag-of-words. Second, training multiple distinct models for different budgets is unfeasible in practice, especially when model size is large. Hence, our goal is to learn a single *classifier* that can adapt to the output of any *selector* operating at any budget. Consequently, this *classifier* must be robust so that it can achieve consistent performance with different budgets. Third, the input to the *classifier* in our framework is a sequence of fractured sentences which is incompatible with a standard *classifier* that trained on the full texts, causing its performance degrades

significantly. One potential but unfeasible solution is to train the *classifier* with a diverse collection of sentence fragments which is combinatorially numerous. Another approach is to randomly blank out text (a.k.a. blanking-noise), leads to marginalized feature distortion (Maaten et al., 2013) but this also leads to poor accuracy as DNNs leverage word combinations, sentence structure, which this approach does not account for. To mitigate this problem, we propose a data aggregation framework that augments the training corpus with outputs from *selector*s at different budget levels. By training the *classifier* on the aggregated *structured* blank-out text, the *classifier* learns to fuse fragmented sentences into a feature representation that mirrors the representation obtained on full sentences and thus realizes high-accuracy. We evaluate our approach through comprehensive experiments on real-world datasets[1].

6.2 Related Work

Several approaches have been proposed to speed up the DNN in test time (Wu et al., 2017; Choi et al., 2017). LSTM-jump (Yu et al., 2017) learns to completely skip words deemed to be irrelevant and skim-RNN (Seo et al., 2017) uses a low-complexity LSTM to skim words rather than skipping. Another version of LSTM-jump, LSTM-shuttle (Fu and Ma, 2018) first skips a number of words, then goes backward to recover lost information by reading some words skipped before. All these approaches require to modify the architecture of the underlying classifier and cannot easily extend to another architecture. In contrast, we adopt existing *classifier* architectures (e.g., LSTM, BCN (McCann et al., 2017)) and propose a meta-learning algorithm to train the model. Our framework is generic and a *classifier* can be viewed as a black-box. Similar to us, Lei et al. (2016) propose a *selector-classifier* framework to find text snippets as justification for text classification but their *selector* and *classifier* have similar complexity and require similar processing times; therefore, it is not suitable for computation gain. Various feature

[1] Our source code is available at:
https://github.com/uclanlp/Fast-and-Robust-Text-Classification

selection approaches (Chandrashekar and Sahin, 2014) have been discussed in literature. For example, removing predefined stop-words (see Appendix A), attention based models (Bahdanau et al., 2014; Luong et al., 2015a), feature subspace selection methods (e.g., PCA), and applying the L1 regularization (e.g., Lasso (Tibshirani, 1996) or Group Lasso (Faruqui et al., 2015), BLasso (?)). However, these approaches either cannot obtain sparse features or cannot straightforwardly be applied to speed up a DNN *classifier*. Different from ours, Viola and Jones (2001); Trapeznikov and Saligrama (2013); Karayev et al. (2013); Xu et al. (2013); Kusner et al. (2014); Bengio et al. (2015); Leroux et al. (2017); Zhu et al. (2019); Nan and Saligrama (2017); Bolukbasi et al. (2017) focus on gating various components of existing networks. Finally, aggregating data or models has been studied under different contexts (e.g., in context of reinforcement learning (Ross et al., 2010), Bagging models (Breiman, 1996), etc.) while we aggregate the data output from *selectors* instead of models.

6.3 Classification on a Test-Time Budget

Our goal is to build a robust *classifier* along with a suite of *selectors* to achieve good performance with consistent speedup under different selection budgets at test-time. Formally, a *classifier* $C(\hat{x})$ takes a word sequence $\hat{x}$ and predicts the corresponding output label y, and a *selector* $S_b(x)$ with selection budget b takes an input word sequence $x = \{w_1, w_2, \ldots, w_N\}$ and generates a binary sequence $S_b(x) = \{z_{w_1}, z_{w_2}, \ldots, z_{w_N}\}$ where $z_{w_k} \in \{0, 1\}$ represents if the corresponding word w_k is selected or not. We denote the sub-sequence of words generated after filtering by the *selector* as $I\big(x, S_b(x)\big) = \{w_k : z_{w_k} = 1, \forall w_k \in x\}$. We aim to train a *classifier* C and the *selector* S_b such that $I\big(x, S_b(x)\big)$ is sufficient to make accurate prediction on the output label (i.e., $C\Big(I\big(x, S_b(x)\big)\Big) \approx C(x)$). The selection budget (a.k.a selection rate) b is controlled by the hyper-parameters of the *selector*. Higher budget often leads to higher accuracy and longer test time.

6.3.1 Learning a *Selector*

We propose two simple but efficient *selector*s.

6.3.1.1 Word Embedding (WE) *selector*

We consider a parsimonious word-selector using word embeddings (e.g., GloVe (Pennington et al., 2014)) as features to predict important words. We assume the informative words can be identified independently and model the probability that a word w_k is selected by $P(z_{w_k} = 1|w_k) = \sigma(\theta_S^T \boldsymbol{w}_k)$, where θ_S is the model parameters of the *selector* S_b, $\boldsymbol{w}_k$ is the corresponding word vector, and σ is the sigmoid function. As we do not have explicit annotations about which words are important, we train the *selector* S_b along with a *classifier* C in an end-to-end manner following Lei et al. (2016), and an L1-regularizer is added to control the sparsity (i.e., selection budget) of $S_b(x)$.

6.3.1.2 Bag-of-Words *selector*

We also consider using an L1-regularized linear model (Zou and Hastie, 2005; Ng, 2004; Yuan et al., 2010) with bag-of-words features to identify important words. In the bag-of-words model, for each document x, we construct a feature vector $\boldsymbol{x} \in \{0,1\}^{|V|}$, where $|V|$ is the size of the vocabulary. Each element of the feature vector $\boldsymbol{x}_w$ represents if a specific word w appearing in the document x. Given a training set $\mathcal{X}$, the linear model optimizes the L1-regularized task loss. For example, in case of a binary classification task (output label $y \in \{1, -1\}$),

$$
\begin{aligned}
J(x_t, y_t) &= \log\big(1 + \exp(-y_t \theta^T \boldsymbol{x_t})\big) \\
\theta^* &= \arg\min_\theta \sum_{(x_t, y_t) \in \mathcal{X}} J(x_t, y_t) + \frac{1}{b}\|\theta\|_1,
\end{aligned}
$$

where $\theta \in R^{|V|}$ is a weight vector to be learned, θ_w corresponds to word $w \in V$, and b is a hyper-parameter controlling the sparsity of θ^* (i.e., selection budget). The lower the budget b is, the sparser the selection is. Based on the optimal θ^*, we construct a *selector*

Model	SST-2				IMDB				AGNews				Yelp			
	acc	r	t	speedup	acc	r	t	speedup	acc	r	t	speedup	acc	r	t	speedup
Baseline	**85.7**	100	9	1x	91.0	100	1546	1x	92.3	100	59	1x	**66.5**	100	3487	1x
Bag-of-Words	78.8	75	5.34	1.7x	91.5	91	1258	**1.2x**	92.9	97	48	1.2x	59.7	55	2325	**1.6x**
Our framework	82.6	65	4.6	**2x**	92.0	91	1297	**1.2x**	93.1	91	46	**1.3x**	64.8	55	2179	**1.6x**
	85.3	0	9	1x	**92.1**	0	1618	1x	**93.2**	0	57	1x	66.3	0	3448	1x

Table 6.1: Accuracy and speedup on the test datasets. r, t denotes the selection rate (%), test-time respectively. Test-times are measured in seconds. The speedup rate is calculated as the running time of a model divided by the running time of the corresponding baseline. For our framework, top row denotes the best speedup and the bottom row denotes the best test accuracy achieved. Overall best accuracies and best speedups are boldfaced. Our framework achieves accuracies better than baseline with a speedup of **1.2x** and **1.3x** on IMDB, and AGNews respectively. With same or higher speedup, our accuracies are much better than Bag-of-Words.

that picks word w if the corresponding θ_w^* is non-zero. Formally, the bag-of-words *selector* outputs $S_b(x) = \{\delta(\theta_w \neq 0) : w \in x\}$, where δ is an indicator function.

6.3.2 The Data Aggregation Framework

In order to learn to fuse fragmented sentences into a robust feature representation, we propose to train the *classifier* on the aggregated corpus of structured blank-out texts.

Given a set of training data $\mathcal{X} = \{(x_1, y_1), .., (x_t, y_t), .., (x_m, y_m)\}$, we assume we have a set of selectors $\mathcal{S} = \{S_b\}$ with different budget levels trained by the framework discussed in Section 6.3.1. To generate an aggregated corpus, we first apply each *selector* $S_b \in \mathcal{S}$ on the training set, and generate corresponding blank-out corpus $\mathcal{I}(\mathcal{X}, S_b) = \big\{I\big(x_t, S_b(x_t)\big), \forall x_t \in \mathcal{X}\big\}$. Then, we create a new corpus by aggregating the blank-out corpora: $\mathcal{T} = \bigcup_{S_b \in \mathcal{S}} \mathcal{I}(\mathcal{X}, S_b)$.[2] Finally, we train the *classifier* $C_\mathcal{T}$ on the aggregated corpus $\mathcal{T}$. As $C_\mathcal{T}$ is trained on documents with distortions, it learns to make predictions with different budget levels. The training procedure is summarized in Algorithm 1. In the following, we discuss two extensions of our data aggregation framework.

First, the blank-out data can be generated from different classes of *selectors* with

[2]Note that, the union operation is used just to aggregate the train instances which does not hinder the model training (e.g., discrete variables).

Algorithm 3 **Data Aggregated Training Schema**

Input:
Training corpus $\mathcal{X}$,
a set of *selectors* with different budget levels $\mathcal{S} = \{S_b\}$,
classifier class C

Output: A robust *classifier* $C_{\mathcal{T}}$

Initialize the aggregated corpus: $\mathcal{T} \leftarrow \mathcal{X}$
for $S_b \in \mathcal{S}$ **do**
  $S_b \leftarrow$ Train a selector $S_b \in S$ with budget level b on $\mathcal{X}$
  Generate a blank-out dataset $\mathcal{I}(\mathcal{X}, S_b)$
  Aggregate data: $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{I}(\mathcal{X}, S_b)$
$C_{\mathcal{T}} \leftarrow$ Train a *classifier* C on $\mathcal{T}$
return $C_{\mathcal{T}}$

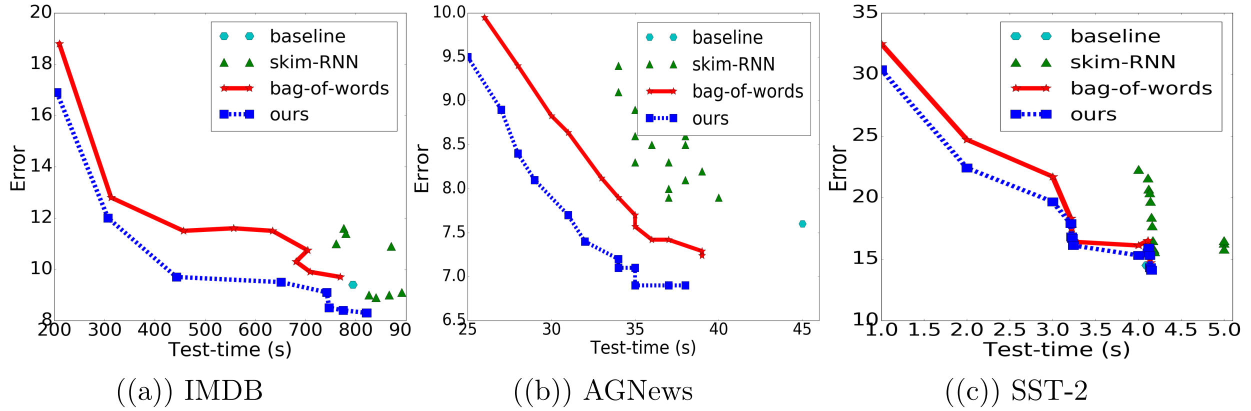
different features or architectures. Second, the blank-out and selection can be done in phrase or sentence level. Specifically, if phrase boundaries are provided, a phrase-level aggregation can avoid a *selector* from breaking compound nouns or meaningful phrases (e.g., "Los Angeles", "not bad"). Similarly, for multi-sentenced documents, we can enforce the *selector* to pick a whole sentence if any word in the sentence is selected.

World News	.. **plant** searched. Kansai Electric Power's **nuclear** power **plant** in Fukui .. **was searched by police** Saturday ..
Business	Telecom Austria taps the Bulgarian **market**. **Telecom** Austria, Austrias **largest** telecoms operator, obtained ..
Sci/Tech	.. Reuters **-** **Software security** companies **and** handset makers, including Finland's **Nokia** (NOK1V.HE), **are** ..

Table 6.3: Examples of the WE *selector* output on AGNews. Bold words are selected.

6.4 Experiments

To evaluate the proposed approach, we consider four benchmark datasets: *SST-2* (Socher et al., 2013), *IMDB* (Maas et al., 2011), *AGNews* (Zhang et al., 2015), and *Yelp* (Conneau et al., 2016) and two widely used architectures for classification: *LSTM*, and



((a)) IMDB ((b)) AGNews ((c)) SST-2

Figure 6.2: Performance under different test-times on IMDB, AGNews, and SST-2. All the approaches use the same LSTM model as the back-end. Bag-of-Words model and our framework have the same bag-of-words *selector* cascaded with this LSTM *classifier* trained on the original training corpus and aggregated corpus, respectively. Our model (blue dashed line) significantly outperform others for any test-time budget. Also its performance is robust, while results of skim-RNN is inconsistent with different budget levels.

BCN (McCann et al., 2017). We evaluate the computation gain of models in terms of overall test-time, and the performance in terms of accuracy. We follow Seo et al. (2017) to estimate the test-time of models on CPU and exclude the time for data loading.

In our approach, we train a *classifier* with both WE and bag-of-words *selector*s with 6 selection budgets[3] {50%, 60%, . . . , 100%} by the word-level data aggregation framework. We evaluate the computation gain of the proposed method through a comparative study of its performance under different test-times by varying the selection budgets in comparison to the following approaches: (1) *Baseline*: the original *classifier* (i.e., no *selector*, no data aggregation) (2) *skim-RNN*: we train a skim-RNN model and vary the amount of text to skim (i.e., test-time) by tuning θ parameter as in Seo et al. (2017). (3) Bag-of-Words: filtering words by the bag-of-words *selector* and feeding the fragments of sentences to the original *classifier* (i.e., no data aggregation). This approach serves as a good baseline and has been considered in the context of linear models (e.g., Chang and Lin (2008)). For a fair comparison, we implement all approaches upon the same framework using

[3]For the very large Yelp dataset, 3 selection budgets {50%, 60%, 70%} are used.

AllenNLP library[4], including a re-implementation of the existing state-of-art speedup framework skim-RNN (Seo et al., 2017)[5]. As skim-RNN is designed specifically for accelerating the LSTM model, we only compare with skim-RNN using LSTM *classifier*. Each corresponding model is selected by tuning parameters on validation data. The model is then frozen and evaluated on test-data for different selection budgets.

Figure 6.2 demonstrates the trade-off between the performance, and the test-time for each setting. Overall, we expect the error to decrease with a larger test-time budget. From Figure 6.2, on all of the IMDB, AGNews, and SST-2 datasets, LSTM *classifier* trained with our proposed data aggregation not only achieves the lowest error curve but also the results are robust and consistent. That is our approach achieves higher performance across different test-time budgets and its performance is a predictable monotonic function of the test-time budget. However, the performance of skim-RNN exhibits inconsistency for different budgets. As a matter of fact, for multiple budgets, none of the skim-RNN, and LSTM-jump address the problem of different word distribution between training and testing. Therefore, similar to skim-RNN, we anticipate that the behavior of LSTM-jump will be inconsistent as well[6]. Additionally, since LSTM-jump has already been shown to be outperformed by skim-RNN, we do not further compare with it. Next, we show that our framework is generic and can incorporate with other different *classifier*s, such as BCN (see Table 6.1)[7]. When phrase boundary information is available, our model can further achieve **86.7** in accuracy with **1.7x** speedup for BCN on SST-2 dataset by using phrase-level data aggregation. Finally, one more advantage of the proposed framework is that the output of the *selector* is interpretable. In Table 6.3, we present that our framework correctly selects words such as "Nokia", "telecom", and phrases such

[4]https://allennlp.org/

[5]The official skim-RNN implementation is not released.

[6]As an example, from Table 6 in Yu et al. (2017), the performance of LSTM-jump drops from 0.881 to 0.854 although it takes longer test-time (102s) than the baseline (81.7s).

[7]Because of the inherent accuracy/inference-time tradeoff, it is difficult to depict model comparisons. For this reason, in Figure 2, we plot the trade-off curve to demonstrate the best speedup achieved by our model for achieving near state-of-art performance. On the other hand, test results are tabulated in Table 1 to focus attention primarily on accuracy.

as “searched by police”, “software security” and filters out words like “Aug.”, “users” and “products”.

Note that nevertheless we focus on efficient inference, empirically our method is no more complex than the baseline during training. Despite the number of training instances increases, and so does the training time for each epoch, the number of epochs we require for obtaining a good model is usually smaller. For example, on the Yelp corpus, we only need 3 epochs to train a BCN *classifier* on the aggregated corpus generated by using 3 different *selector*s, while training on the original corpus requires 10 epochs.

6.5 Conclusion

We present a framework to learn a robust *classifier* under test-time constraints. We demonstrate that the proposed *selector*s effectively select important words for *classifier* to process and the data aggregation strategy improves the model performance. As future work we will apply the framework for other text reading tasks. Another promising direction is to explore the benefits of text classification model in an edge-device setting. This problem naturally arises with local devices (e.g., smart watches or mobile phones), which do not have sufficient memory or computational power to execute a complex *classifier*, and instances must be sent to the cloud. This setting is particularly suited to ours since we could choose to send only the important words to the cloud. In contrast, skim-RNN and LSTM-jump, which process the text sequentially, have to either send the entire text to the server or require multiple rounds of communication between the server and local devices resulting in high network latency.

6.6 Summary

This chapter builds a computationally inexpensive *selector* model to identify words/phrases/sentences in the input text relevant for the target task. This Chapter also presents a data augmentation technique to leverage the auxiliary supervision from the *selector*

model in a structured way and enhance the text classification task in performance, speed and robustness. On four benchmark text classification tasks, we demonstrate that the framework gains consistent speedup with little degradation in accuracy on various test-time budgets. In addition, being robust, the more test-time budget is afforded, the higher performance gain is achieved.

CHAPTER 7

Conclusion and Future Work

Natural language processing has brought about revolutionary improvement in different sectors of our life. However, it requires an immense amount of labeled data to train the NLP models which are very hard to acquire as annotating them needs a long time, specific human skillsets as well as user data in practice could be very heterogeneously different from ideal cases. In this dissertation, towards the goal of enriching supervision and combating training data scarcity, we cover two important directions: how to select/retrieve relevant data automatically and how to incorporate auxiliary supervision from the retrieved data.

While we have an enormous amount of open-sourced data which are user generated and free not all of them are useful and feasible to process. In Chapter 2, we develop a source data valuation framework what can quantifies the usefulness of a training corpora. Then, to retrieve relevant instances from an corpora, we build a dense retriever model in Chapter 3. In Chapter 5, we further improve the retriever. In Chapter 6, we develop a more fine-grained selector to identify relevant words or phrases in a text instance. As for the auxiliary data sources, in this dissertation, we have found that when selected/retrieved properly external resources, data form other existing tools/methods prevailed in the literature, standard linguistic information or domain knowledge as well as the underlying structure present in the base data itself can provide additional supervision to the NLP models.

With the retrieved relevant data, the next step is to come up with methods to from auxiliary supervising from them. In this dissertation, we show that auxiliary supervision can be obtained via a simple input-output formatted training data augmentation which is architecture agnostic and can be adopted w/ any off-the-self model w/ much changing.

However modification of the models may bring further improvements as well. For example (i) in Chapter 3, we develop a retriever augmented framework for multi-modal generative model that leverage both unimodal and bi-modal (code and text) retrieved candidates; (ii) in Chapter 4, we build a joint inference probabilistic model that leverages rule-based task-oriented auxiliary information and enhance the generation even further. Regarding the extents of enhancements of downstream tasks, we find that incorporating the non-parametric auxiliary information in a principled way can easily update model memory without retraining them and enhance a wide range of NLP applications in various aspects including performance, speed, robustness and interpretability.

However, several questions and related problems still remained as open problems such as (i) we consider the text-code multi-modality problems in this dissertation. Can auxiliary information improve other multi-modal low-resource NLP problems? One practical use-case would be medical image captioning. (ii) how to perform the multi-modal retrieval of auxiliary information based on the data structure present in the candidates (e.g., context flow tree of the source code)? (iii) Retrieved information on-the-fly guides the generation task as a template. Can these be useful for making the generation diverse such as automated diverse dialogue generation tasks and so on. We leave them as a future exploration of this dissertation.

Agić, Ž., Johannsen, A., Plank, B., Martínez Alonso, H., Schluter, N., and Søgaard, A. (2016). Multilingual projection for parsing truly low-resource languages. *Transactions of the Association for Computational Linguistics*, 4:301–312. 30

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2020a). A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics. 32, 53

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021a). Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668. ix, 33, 34, 37, 42, 43, 44, 45, 54

Ahmad, W., Chi, J., Le, T., Norton, T., Tian, Y., and Chang, K.-W. (2021b). Intent classification and slot filling for privacy policies. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4402–4417. 89

Ahmad, W., Chi, J., Tian, Y., and Chang, K.-W. (2020b). PolicyQA: A reading comprehension dataset for privacy policies. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 743–749. 89

Ahmad, W., Zhang, Z., Ma, X., Hovy, E., Chang, K.-W., and Peng, N. (2019). On difficulties of cross-lingual transfer with order differences: A case study on dependency parsing. 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 2440–2452, Minneapolis, Minnesota. Association for Computational Linguistics. 10, 17, 20, 25

Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual string embeddings for sequence labeling. In *COLING 2018, 27th International Conference on Computational Linguistics*, pages 1638–1649. 19

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2017). A survey of machine learning for big code and naturalness. *arXiv preprint arXiv:1709.06182*. 61

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37. 32

Allamanis, M., Peng, H., and Sutton, C. A. (2016). A convolutional attention network for extreme summarization of source code. In Balcan, M. and Weinberger, K. Q., editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML*

2016, New York City, NY, USA, June 19-24, 2016, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2091–2100. JMLR.org. 53

Anaby-Tavor, A., Carmeli, B., Goldbraich, E., Kantor, A., Kour, G., Shlomov, S., Tepper, N., and Zwerdling, N. (2020). Do not have enough data? deep learning to the rescue! In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7383–7390. 78

Arnold, K. C., Chang, K.-W., and Kalai, A. (2017). Counterfactual language model adaptation for suggesting phrases. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017*, pages 49–54. 57

Artetxe, M. and Schwenk, H. (2019). Massively multilingual sentence embeddings for zero-shot cross-lingual transfer and beyond. *Transactions of the Association for Computational Linguistics*, 7:597–610. 30

Arwan, A., Rochimah, S., and Akbar, R. J. (2015). Source code retrieval on stackoverflow using lda. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*, pages 295–299. IEEE. 54

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*. 95

Baker, L. D. and McCallum, A. K. (1998). Distributional clustering of words for text classification. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 96–103. ACM. 59

Bengio, E., Bacon, P.-L., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*. 95

Blitzer, J., Dredze, M., and Pereira, F. (2007). Biographies, Bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 440–447, Prague, Czech Republic. Association for Computational Linguistics. 8, 17

Bolukbasi, T., Wang, J., Dekel, O., and Saligrama, V. (2017). Adaptive neural networks for efficient inference. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 527–536, International Convention Centre, Sydney, Australia. PMLR. 95

Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. (2010). Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. 33

Breiman, L. (1996). Bagging predictors. *Machine learning*, 24(2):123–140. 89, 95

Brill, E. and Moore, R. C. (2000). An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293. Association for Computational Linguistics. 57

Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992). Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479. 59

Bruce, K. B. (1993). Safe type checking in a statically-typed object-oriented programming language. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298. ACM. 57

Bui, D., Shin, K. G., Choi, J.-M., and Shin, J. (2021). Automated extraction and presentation of data practices in privacy policies. *Proceedings on Privacy Enhancing Technologies*, 2021(2):88–110. 89

Cai, Y. and Wan, X. (2019). Multi-domain sentiment classification based on domain-aware embedding and attention. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 4904–4910. AAAI Press. viii, 23

Casalnuovo, C., Suchak, Y., Ray, B., and Rubio-González, C. (2017). Gitcproc: a tool for processing and classifying github commits. pages 396–399. ACM. 71

Chandrashekar, G. and Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28. 95

Chang, Y.-W. and Lin, C.-J. (2008). Feature ranking using linear svm. In *Causation and Prediction Challenge*, pages 53–64. 99

Choi, E., Hewlett, D., Uszkoreit, J., Polosukhin, I., Lacoste, A., and Berant, J. (2017). Coarse-to-fine question answering for long documents. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 209–220. 94

Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. (2020). ELECTRA: Pre-training text encoders as discriminators rather than generators. In *International Conference on Learning Representations*. 45

Commission, F. T. et al. (2012). Protecting consumer privacy in an era of rapid change. *FTC report*. 77

Conneau, A., Rinott, R., Lample, G., Williams, A., Bowman, S., Schwenk, H., and Stoyanov, V. (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. 17, 28

Conneau, A., Schwenk, H., Barrault, L., and Lecun, Y. (2016). Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*. 98

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. 45

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (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. 10, 19, 82

Dubey, P. (1975). On the uniqueness of the Shapley value. *International Journal of Game Theory*, 4(3):131–139. 10

Faruqui, M., Tsvetkov, Y., Yogatama, D., Dyer, C., and Smith, N. A. (2015). Sparse overcomplete word vector representations. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1491–1500. 95

Feng, S., Wallace, E., Grissom II, A., Iyyer, M., Rodriguez, P., and Boyd-Graber, J. (2018). Pathologies of neural models make interpretations difficult. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3719–3728. 10

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020a). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. 33, 43, 45

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020b). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics. 38

Fu, T.-J. and Ma, W.-Y. (2018). Speed reading: Learning to read forbackward via shuttle. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4439–4448. 94

Gabel, M. and Su, Z. (2010). A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM. 60

Galley, M., Brockett, C., Sordoni, A., Ji, Y., Auli, M., Quirk, C., Mitchell, M., Gao, J., and Dolan, B. (2015). deltableu: A discriminative metric for generation tasks with intrinsically diverse targets. *arXiv preprint arXiv:1506.06863*. 57

Gao, T., Yao, X., and Chen, D. (2021). SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference on Empirical Methods in Natural*

Language Processing, pages 6894–6910, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics. 82, 90

Gharehyazie, M., Ray, B., and Filkov, V. (2017). Some from here, some from there: cross-project code reuse in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 291–301. IEEE Press. 61

Ghorbani, A. and Zou, J. (2019). Data Shapley: Equitable valuation of data for machine learning. *International Conference on Machine Learning*, pages 2242–2251. 9, 10, 12, 15, 28

Ghorbani, A. and Zou, J. (2020). Neuron Shapley: Discovering the responsible neurons. *arXiv preprint arXiv:2002.09815*. 30

Gillick, D., Kulkarni, S., Lansing, L., Presta, A., Baldridge, J., Ie, E., and Garcia-Olano, D. (2019). Learning dense representations for entity retrieval. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 528–537, Hong Kong, China. Association for Computational Linguistics. 37

Gluck, J., Schaub, F., Friedman, A., Habib, H., Sadeh, N., Cranor, L. F., and Agarwal, Y. (2016). How short is too short? implications of length and framing on the effectiveness of privacy notices. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*. 77

Goodman, J. (2001). Classes for fast maximum entropy training. *CoRR*, cs.CL/0108006. 59

Gu, J., Lu, Z., Li, H., and Li, V. O. K. (2016a). Incorporating copying mechanism in sequence-to-sequence learning. *CoRR*, abs/1603.06393. 60

Gu, X., Zhang, H., Zhang, D., and Kim, S. (2016b). Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. 32

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., et al. (2021). Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations*. 33, 34, 38, 45

Guo, D., Tang, D., Duan, N., Zhou, M., and Yin, J. (2019). Coupling retrieval and meta-learning for context-dependent semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 855–866, Florence, Italy. Association for Computational Linguistics. 43

Guo, J., Che, W., Yarowsky, D., Wang, H., and Liu, T. (2015). Cross-lingual dependency parsing based on distributed representations. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1234–1244. 30

Guo, R., Kumar, S., Choromanski, K., and Simcha, D. (2016). Quantization based fast inner product search. In *Artificial Intelligence and Statistics*, pages 482–490. PMLR. 36

Harer, J., Reale, C., and Chin, P. (2019). Tree-transformer: A transformer-based method for correction of tree-structured data. *arXiv preprint arXiv:1908.00449*. 53

Harkous, H., Fawaz, K., Lebret, R., Schaub, F., Shin, K. G., and Aberer, K. (2018). Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 531–548. 77, 82, 89

Hashimoto, T. B., Guu, K., Oren, Y., and Liang, P. S. (2018). A retrieve-and-edit framework for predicting structured outputs. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc. 54

Hayati, S. A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., and Neubig, G. (2018). Retrieval-based neural code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 925–930, Brussels, Belgium. Association for Computational Linguistics. 54

Hellendoorn, V. J. and Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA. ACM. x, 57, 61, 73, 75

Henderson, M., Al-Rfou, R., Strope, B., Sung, Y.-H., Lukács, L., Guo, R., Kumar, S., Miklos, B., and Kurzweil, R. (2017). Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*. 80

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. (2016). On the naturalness of software. *Commun. ACM*, 59(5):122–131. 57

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE. 61

Hosseini, M. B., Wadkar, S., Breaux, T. D., and Niu, J. (2016). Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series*. 78

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018a). Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, page 200–210, New York, NY, USA. Association for Computing Machinery. 53

Hu, X., Li, G., Xia, X., Lo, D., Lu, S., and Jin, Z. (2018b). Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International*

Joint Conference on Artificial Intelligence, IJCAI-18, pages 2269–2275. International Joint Conferences on Artificial Intelligence Organization. 53

Hucka, M. and Graham, M. J. (2018). Software search is not a science, even among scientists: A survey of how scientists and engineers find software. *Journal of Systems and Software*, 141:171–191. 54

Humeau, S., Shuster, K., Lachaux, M.-A., and Weston, J. (2019). Poly-encoders: Architectures and pre-training strategies for fast and accurate multi-sentence scoring. In *International Conference on Learning Representations*. 81, 88

Humeau, S., Shuster, K., Lachaux, M.-A., and Weston, J. (2020). Poly-encoders: Architectures and pre-training strategies for fast and accurate multi-sentence scoring. In *International Conference on Learning Representations*. 34, 48

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. (2019). Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*. x, 41, 42, 45, 53

Iyer, S., Cheung, A., and Zettlemoyer, L. (2019). Learning programmatic idioms for scalable semantic parsing. 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 5426–5435, Hong Kong, China. Association for Computational Linguistics. 43

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083. 53

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2018). Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics. 40, 42

Izacard, G. and Grave, E. (2021). Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880, Online. Association for Computational Linguistics. 55

Ji, Y., Tan, C., Martschat, S., Choi, Y., and Smith, N. A. (2017). Dynamic entity representations in neural language models. *arXiv preprint arXiv:1708.00781*. 60

Jia, R., Dao, D., Wang, B., Hubis, F. A., Gurel, N. M., Li, B., Zhang, C., Spanos, C., and Song, D. (2019a). Efficient task-specific data valuation for nearest neighbor algorithms. *Proceedings of the VLDB Endowment*, 12(11):1610–1623. 10, 12, 30

Jia, R., Sun, X., Xu, J., Zhang, C., Li, B., and Song, D. (2019b). An empirical and comparative analysis of data valuation with scalable algorithms. *arXiv preprint arXiv:1911.07128*. 10, 12

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*. 62

Karayev, S., Fritz, M., and Darrell, T. (2013). Dynamic feature selection for classification on a budget. In *International Conference on Machine Learning (ICML): Workshop on Prediction with Sequential Models*. 95

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. (2020a). Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics. ix, xv, 34, 36, 37, 38, 39, 42, 45

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. (2020b). Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781. 79, 80, 82

Keymanesh, M., Elsner, M., and Parthasarathy, S. (2021). Privacy policy question answering assistant: A query-guided extractive summarization approach. *arXiv preprint arXiv:2109.14638*. 89

Kiddon, C., Zettlemoyer, L., and Choi, Y. (2016). Globally coherent text generation with neural checklist models. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 329–339. 60, 67, 68

Kim, J.-K., Kim, Y.-B., Sarikaya, R., and Fosler-Lussier, E. (2017). Cross-lingual transfer learning for pos tagging without cross-lingual resources. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2832–2838. 30

Kim, M., Sazawal, V., Notkin, D., and Murphy, G. (2005). An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM. 61

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980. 66

Knuth, D. E. (1992). Literate programming. *CSLI Lecture Notes, Stanford, CA: Center for the Study of Language and Information (CSLI), 1992*. 60

Koh, P. W. and Liang, P. (2017). Understanding black-box predictions via influence functions. *arXiv preprint arXiv:1703.04730*. 30

Kumar, I. E., Venkatasubramanian, S., Scheidegger, C., and Friedler, S. (2020a). Problems with Shapley-value-based explanations as feature importance measures. *ICML Workshop on Workshop on Human Interpretability in Machine Learning (WHI)*. 30

Kumar, V., Choudhary, A., and Cho, E. (2020b). Data augmentation using pre-trained transformer models. In *Proceedings of the 2nd Workshop on Life-long Learning for Spoken Language Systems*, pages 18–26. 78

Kusner, M. J., Chen, W., Zhou, Q., Xu, Z. E., Weinberger, K. Q., and Chen, Y. (2014). Feature-Cost Sensitive Learning with Submodular Trees of Classifiers. In *AAAI*, pages 1939–1945. 95

Lample, G. and Conneau, A. (2019). Cross-lingual language model pretraining. *CoRR*, abs/1901.07291. viii, 22

Lample, G., Conneau, A., Ranzato, M., Denoyer, L., and Jégou, H. (2018). Word translation without parallel data. In *International Conference on Learning Representations*. 30

LeClair, A., Jiang, S., and McMillan, C. (2019). A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*, page 795–806. IEEE Press. 53

Lei, T., Barzilay, R., and Jaakkola, T. S. (2016). Rationalizing neural predictions. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 94, 96

Leroux, S., Bohez, S., De Coninck, E., Verbelen, T., Vankeirsbilck, B., Simoens, P., and Dhoedt, B. (2017). The cascading neural network: building the internet of smart things. *Knowledge and Information Systems*, pages 1–24. 95

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M. F., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc. 55

Li, H., Xing, Z., Peng, X., and Zhao, W. (2013). What help do developers seek, when and how? In *2013 20th working conference on reverse engineering (WCRE)*, pages 142–151. IEEE. 33

Li, J., Monroe, W., and Jurafsky, D. (2016). Understanding neural networks through representation erasure. *CoRR*, abs/1612.08220. 10

Liang, Y. and Zhu, K. Q. (2018). Automatic generation of text descriptive comments for code blocks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 5229–5236. 53

Lin, C.-Y. and Och, F. J. (2004). ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, pages 501–507, Geneva, Switzerland. COLING. 43

Lin, Y.-H., Chen, C.-Y., Lee, J., Li, Z., Zhang, Y., Xia, M., Rijhwani, S., He, J., Zhang, Z., Ma, X., Anastasopoulos, A., Littell, P., and Neubig, G. (2019). Choosing transfer languages for cross-lingual learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3125–3135, Florence, Italy. Association for Computational Linguistics. xiii, 9, 10, 11, 16, 24, 25

Liu, M., Song, Y., Zou, H., and Zhang, T. (2019a). Reinforced training data selection for domain adaptation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1957–1968. viii, 22, 23

Liu, N. F., Gardner, M., Belinkov, Y., Peters, M. E., and Smith, N. A. (2019b). Linguistic knowledge and transferability of contextual representations. 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 1073–1094, Minneapolis, Minnesota. Association for Computational Linguistics. 9

Liu, P., Qiu, X., and Huang, X. (2017). Adversarial multi-task learning for text classification. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1–10, Vancouver, Canada. Association for Computational Linguistics. viii, 17, 19

Liu, S., Chen, Y., Xie, X., Siow, J. K., and Liu, Y. (2021). Retrieval-augmented generation for code summarization via hybrid {gnn}. In *International Conference on Learning Representations*. x, 42, 53, 55

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019c). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*. 45

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*. ix, 40, 41, 45, 54

Lundberg, S. M., Erion, G. G., and Lee, S.-I. (2018). Consistent individualized feature attribution for tree ensembles. *arXiv preprint arXiv:1802.03888*. 30

Luong, M.-T., Pham, H., and Manning, C. D. (2015a). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*. 95

Luong, T., Pham, H., and Manning, C. D. (2015b). Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, Lisbon, Portugal. Association for Computational Linguistics. 45

L'heureux, A., Grolinger, K., Elyamany, H. F., and Capretz, M. A. (2017). Machine learning with big data: Challenges and approaches. *Ieee Access*, 5:7776–7797. 13

Ma, X., Xu, P., Wang, Z., Nallapati, R., and Xiang, B. (2019). Domain adaptation with bert-based domain classification and data selection. In *Proceedings of the 2nd Workshop on Deep Learning Approaches for Low-Resource NLP (DeepLo 2019)*, pages 76–83. viii, 18, 23

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics. 98

Maaten, L., Chen, M., Tyree, S., and Weinberger, K. (2013). Learning with marginalized corrupted features. In *International Conference on Machine Learning*, pages 410–418. 94

Maleki, S. (2015). Addressing the computational issues of the Shapley value with applications in the smart grid. 30

Maltese, G., Bravetti, P., Crépy, H., Grainger, B., Herzog, M., and Palou, F. (2001). Combining word-and class-based language models: A comparative study in several languages using automatic and manual word-clustering techniques. In *Seventh European Conference on Speech Communication and Technology*. 59

Manning, C., Raghavan, P., and Schütze, H. (2008). Xml retrieval. In *Introduction to Information Retrieval.* Cambridze University Press. 36

McCann, B., Bradbury, J., Xiong, C., and Socher, R. (2017). Learned in translation: Contextualized word vectors. In *Advances in Neural Information Processing Systems*, pages 6297–6308. 94, 99

McDonald, A. M. and Cranor, L. F. (2008). The cost of reading privacy policies. *Isjlp*, 4:543. 77

McDonald, R., Petrov, S., and Hall, K. (2011). Multi-source transfer of delexicalized dependency parsers. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 62–72. 10

Meng, T., Peng, N., and Chang, K.-W. (2019). Target language-aware constrained inference for cross-lingual dependency parsing. 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 1117–1128. 30

Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and Optimizing LSTM Language Models. *arXiv preprint arXiv:1708.02182*. x, 61, 65, 69, 70

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*. 57

Morin, F. and Bengio, Y. (2005). Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer. 59

Nan, F. and Saligrama, V. (2017). Adaptive classification for prediction under a budget. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4727–4737. Curran Associates, Inc. 95

Nasehi, S. M., Sillito, J., Maurer, F., and Burns, C. (2012). What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE. 54

Neyman, J. (1992). On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection. In *Breakthroughs in Statistics*, pages 123–150. Springer. 13

Ng, A. Y. (2004). Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM. 96

Niesler, T. R., Whittaker, E. W., and Woodland, P. C. (1998). Comparison of part-of-speech and automatically derived category-based language models for speech recognition. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 1, pages 177–180. IEEE. 59

Nivre, J., Abrams, M., Agić, Ž., and et al. (2018). Universal dependencies 2.2. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. 17

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics. 43

Parsons, T. W. (1992). *Introduction to compiler construction.* Computer Science Press New York. 58

Parvez, M. R., Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics. 6, 80

Parvez, M. R., Bolukbasi, T., Chang, K.-W., and Saligrama, V. (2019a). Robust text classifier on test-time budgets. 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 1167–1172. 89

Parvez, M. R., Bolukbasi, T., Chang, K.-W., and Saligrama, V. (2019b). Robust text classifier on test-time budgets. 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 1167–1172, Hong Kong, China. Association for Computational Linguistics. 92

Parvez, M. R., Chakraborty, S., Ray, B., and Chang, K.-W. (2018). Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics. 6, 32, 56

Parvez, M. R. and Chang, K.-W. (2021). Evaluating the values of sources in transfer learning. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5084–5116. 6, 88

Parvez, M. R., Chi, J., Ahmad, W. U., Tian, Y., and Chang, K.-W. (2022). Retrieval enhanced data augmentation for question answering on privacy policies. *arXiv preprint arXiv:2204.08952*. 7, 77

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543. 72, 96

Pereira, F., Tishby, N., and Lee, L. (1993). Distributional clustering of english words. In *Proceedings of the 31st annual meeting on Association for Computational Linguistics*, pages 183–190. Association for Computational Linguistics. 59

Petrov, S. and McDonald, R. (2012). Overview of the 2012 shared task on parsing the web. 17

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R., and Lanza, M. (2014). Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. 54

Rabinovich, M., Stern, M., and Klein, D. (2017a). Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada. Association for Computational Linguistics. 54

Rabinovich, M., Stern, M., and Klein, D. (2017b). Abstract syntax networks for code generation and semantic parsing. *CoRR*, abs/1704.07535. 57

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9. 45

Rahimi, A., Li, Y., and Cohn, T. (2019). Massively multilingual transfer for NER. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 151–164. 10

Rahman, M. M., Barson, J., Paul, S., Kayani, J., Lois, F. A., Quezada, S. F., Parnin, C., Stolee, K. T., and Ray, B. (2018). Evaluating how developers use general-purpose web-search for code retrieval. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 465–475. 54

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. (2016). SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392. 89

Ravichander, A., Black, A. W., Norton, T., Wilson, S., and Sadeh, N. (2021). Breaking down walls of text: How can NLP benefit consumer privacy? In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4125–4140, Online. Association for Computational Linguistics. 89

Ravichander, A., Black, A. W., Wilson, S., Norton, T., and Sadeh, N. (2019). Question answering for privacy policies: Combining computational and legal perspectives. 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 4947–4958. xi, 78, 82, 83

Ray, B., Nagappan, M., Bird, C., Nagappan, N., and Zimmermann, T. (2015). The uniqueness of changes: Characteristics and applications. MSR '15. ACM. 60

Ray, B., Posnett, D., Filkov, V., and Devanbu, P. (2014). A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM. 70

Raychev, V., Vechev, M., and Yahav, E. (2014). Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM. 61

Reidenberg, J. R., Bhatia, J., Breaux, T. D., and Norton, T. B. (2016). Ambiguity in privacy policies and the impact of regulation. *The Journal of Legal Studies*, 45(S2):S163–S190. 77

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Zhou, M., Blanco, A., and Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*. 43

Robertson, S. and Zaragoza, H. (2009). *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc. 34, 36

Ross, S., Gordon, G. J., and Bagnell, J. A. (2010). No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686. 95

Roth, A. E. (1988). *The Shapley value: essays in honor of Lloyd S. Shapley*. Cambridge University Press. 10

Ruder, S. and Plank, B. (2017). Learning to select data for transfer learning with Bayesian optimization. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 372–382, Copenhagen, Denmark. Association for Computational Linguistics. 8, 30

Sadowski, C., Stolee, K. T., and Elbaum, S. (2015). How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201. 33, 54

Seo, M. J., Min, S., Farhadi, A., and Hajishirzi, H. (2017). Neural speed reading via skim-rnn. *CoRR*, abs/1711.02085. 92, 94, 99, 100

Shapley, L. S. (1952). A value for n-person games. Technical report, Rand Corp Santa Monica CA. 10

Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., and Matsumura, T. (2019). Automatic source code summarization with extended tree-lstm. In *International Joint Conference on Neural Networks, IJCNN 2019 Budapest, Hungary, July 14-19, 2019*, pages 1–8. IEEE. 53

Sim, S. E., Umarji, M., Ratanotayanon, S., and Lopes, C. V. (2011). How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(1):1–25. 54

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642. 98

Stolee, K. T., Elbaum, S., and Dobos, D. (2014). Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–45. 54

Sundararajan, M. and Najmi, A. (2019). The many Shapley values for model explanation. *CoRR*, abs/1908.08474. 30

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112. Curran Associates, Inc. 36

Täckström, O., Das, D., Petrov, S., McDonald, R., and Nivre, J. (2013). Token and type constraints for cross-lingual part-of-speech tagging. *Transactions of the Association for Computational Linguistics*, 1:1–12. 8, 30

Tang, S., Ghorbani, A., Yamashita, R., Rehman, S., Dunnmon, J. A., Zou, J., and Rubin, D. L. (2020). Data valuation for medical imaging using Shapley value: Application on a large-scale chest x-ray dataset. *arXiv preprint arXiv:2010.08006*. 30

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288. 95

Tommasi, T. and Caputo, B. (2009). The more you know, the less you learn: from knowledge transfer to one-shot learning of object categories. In *British Machine Vision Conference*. 10

Trapeznikov, K. and Saligrama, V. (2013). Supervised sequential classification under budget constraints. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 581–589. 95

Tripathi, S., Hemachandra, N., and Trivedi, P. (2020). On feature interactions identified by Shapley values of binary classification games. *arXiv preprint arXiv:2001.03956*. 30

Tu, Z., Su, Z., and Devanbu, P. (2014). On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280. ACM. 61

Van, H., Yadav, V., and Surdeanu, M. (2021). Cheap and good? simple and effective data augmentation for low resource machine reading. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2116–2120. 89

Van den Broeck, G., Lykov, A., Schleich, M., and Suciu, D. (2021). On the tractability of shap explanations. *In Proceedings of the 35th Conference on Artificial Intelligence (AAAI)*. 10

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008. 37, 38, 45

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc. 60

Viola, P. and Jones, M. (2001). Robust real-time object detection. *International Journal of Computer Vision*, 4. 95

Vu, T., Wang, T., Munkhdalai, T., Sordoni, A., Trischler, A., Mattarella-Micke, A., Maji, S., and Iyyer, M. (2020). Exploring and predicting transferability across NLP tasks. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7882–7926, Online. Association for Computational Linguistics. 9, 10

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*. 18

Wei, B., Li, G., Xia, X., Fu, Z., and Jin, Z. (2019). Code generation as a dual task of code summarization. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 6563–6573. Curran Associates, Inc. 32

Wilson, S., Schaub, F., Dara, A. A., Liu, F., Cherivirala, S., Giovanni Leon, P., Schaarup Andersen, M., Zimmeck, S., Sathyendra, K. M., Russell, N. C., Norton, T. B., Hovy, E., Reidenberg, J., and Sadeh, N. (2016). The creation and analysis of a website privacy policy corpus. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1330–1340. 78, 89

Wiseman, S., Shieber, S. M., and Rush, A. M. (2017). Challenges in data-to-document generation. *CoRR*, abs/1707.08052. 56, 60

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. (2019a). Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771. 19

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. (2019b). Huggingface's transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771. 82

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics. 42

Wu, F., Lao, N., Blitzer, J., Yang, G., and Weinberger, K. Q. (2017). Fast reading comprehension with ConvNets. *CoRR*, abs/1711.04352. 94

Wu, S. and Dredze, M. (2019). Beto, bentz, becas: The surprising cross-lingual effectiveness of BERT. 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 833–844, Hong Kong, China. Association for Computational Linguistics. viii, 8, 21, 22

Xia, X., Bao, L., Lo, D., Kochhar, P. S., Hassan, A. E., and Xing, Z. (2017). What do developers search for on the web? *Empirical Software Engineering*, 22(6):3149–3185. 54

Xu, F. F., Jiang, Z., Yin, P., Vasilescu, B., and Neubig, G. (2020). Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online. Association for Computational Linguistics. 33

Xu, F. F., Vasilescu, B., and Neubig, G. (2021). In-ide code generation from natural language: Promise and challenges. *arXiv preprint arXiv:2101.11149*. 32

Xu, Z., Kusner, M., Chen, M., and Weinberger, K. Q. (2013). Cost-Sensitive Tree of Classifiers. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 133–141. 95

Yang, Y., Jin, N., Lin, K., Guo, M., and Cer, D. (2020). Neural retrieval for question answering with cross-attention supervised data augmentation. *arXiv preprint arXiv:2009.13815*. 89

Yang, Y., Li, X., Wang, P., Xia, Y., and Ye, Q. (2020). Multi-source transfer learning via ensemble approach for initial diagnosis of alzheimer's disease. *IEEE Journal of Translational Engineering in Health and Medicine*, 8:1–10. 9

Yao, Y. and Doretto, G. (2010). Boosting for transfer learning with multiple sources. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1855–1862. 9

Yih, W.-t., Toutanova, K., Platt, J. C., and Meek, C. (2011). Learning discriminative projections for text similarity measures. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*, pages 247–256. 37

Yin, P. and Neubig, G. (2017a). A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics. 32, 54

Yin, P. and Neubig, G. (2017b). A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696. 57, 60

Yu, A. W., Lee, H., and Le, Q. (2017). Learning to skim text. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1880–1890. 92, 94, 100

Yuan, G.-X., Chang, K.-W., Hsieh, C.-J., and Lin, C.-J. (2010). A comparison of optimization methods and software for large-scale l1-regularized linear classification. *Journal of Machine Learning Research*, 11(Nov):3183–3234. 96

Zhang, J., Wang, X., Zhang, H., Sun, H., and Liu, X. (2020). Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1385–1397. IEEE. 54, 55

Zhang, X., Zhao, J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657. 98

Zhou, J. T., Pan, S. J., Tsang, I. W., and Ho, S.-S. (2016). Transfer learning for cross-language text categorization through active correspondences construction. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2400–2406. 30

Zhu, P., Acar, A., Nan, F., Jain, P., and Saligrama, V. (2019). Cost-aware inference for iot devices. In *AISTATS*. 95

Zimmeck, S., Story, P., Smullen, D., Ravichander, A., Wang, Z., Reidenberg, J., Russell, N. C., and Sadeh, N. (2019). Maps: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies*, 2019(3):66–86. 89

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320. 96