Neural Sequence Modeling for Domain-Specific Language Processing: A Systematic Approach

Ming Zhu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ismini Lourentzou, Co-chair
Danfeng (Daphne) Yao, Co-chair
Edward A. Fox
Chris Brown
Wasi Uddin Ahmad

June 22, 2023
Blacksburg, Virginia

Keywords: Machine Learning for Natural Language Processing, Machine Learning for Code, Machine Learning for Healthcare, Information Retrieval, Question Answering, Entity Linking, Program Translation, Code Refinement, Sequence-to-Sequence Models

Neural Sequence Modeling for Domain-Specific Language Processing: A Systematic Approach

Ming Zhu

(ABSTRACT)

In recent years, deep learning based sequence modeling (neural sequence modeling) techniques have made substantial progress in many tasks, including information retrieval, question answering, information extraction, machine translation, etc. Benefiting from the highly scalable attention-based Transformer architecture and enormous open access online data, large-scale pre-trained language models have shown great modeling and generalization capacity for sequential data. However, not all domains benefit equally from the rapid development of neural sequence modeling. Domains like healthcare and software engineering have vast amounts of sequential data containing rich knowledge, yet remain under-explored due to a number of challenges: 1) the distribution of the sequences in specific domains is different from the general domain; 2) the effective comprehension of domain-specific data usually relies on domain knowledge; and 3) the labelled data is usually scarce and expensive to get in domain-specific settings. In this thesis, we focus on the research problem of applying neural sequence modeling methods to address both common and domain-specific challenges from the healthcare and software engineering domains. We systematically investigate neural-based machine learning approaches to address the above challenges in three research directions: 1) learning with long sequences, 2) learning from domain knowledge and 3) learning under limited supervision. Our work can also potentially benefit more domains with large amounts of sequential data.

Neural Sequence Modeling for Domain-Specific Language Processing: A Systematic Approach

Ming Zhu

(GENERAL AUDIENCE ABSTRACT)

In the last few years, computer programs that learn and understand human languages (an area called machine learning for natural language processing) have significantly improved. These advances are visible in various areas such as retrieving information, answering questions, extracting key details from texts, and translating between languages. A key to these successes has been the use of a type of neural network structure known as a "Transformer", which can process and learn from lots of information found online. However, these successes are not uniform across all areas. Two fields, healthcare and software engineering, still present unique challenges despite having a wealth of information. Some of these challenges include the different types of information in these fields, the need for specific expertise to understand this information, and the shortage of labeled data, which is crucial for training machine learning models. In this thesis, we focus on the use of machine learning for natural language processing methods to solve these challenges in the healthcare and software engineering fields. Our research investigates learning with long documents, learning from domain-specific expertise, and learning when there's a shortage of labeled data. The insights and techniques from our work could potentially be applied to other fields that also have a lot of sequential data.

Acknowledgments

I am profoundly grateful to a number of individuals whose support and guidance have been instrumental in the completion of this thesis.

First and foremost, I would like to express my deep gratitude to my advisors, Dr. Ismini Lourentzou and Dr. Danfeng (Daphne) Yao. During some of the most challenging times of my life, they were there to provide crucial assistance. Their mentorship has not only helped me excel in my academic endeavors and become a better researcher, but also taught me valuable life lessons on compassion and kindness. I am fortunate to have had them as my guiding lights in this journey.

I am indebted to Dr. Clifford A. Shaffer and Dr. Edward A. Fox for their unwavering support during difficult periods. Their wisdom, guidance, and support were pivotal in overcoming hurdles, and their influences will continue to inspire me.

I extend my sincere thanks to my committee members, Dr. Chris Brown and Dr. Wasi Ahmed, for their service. Their constructive feedback and diligent service have been instrumental in refining this work.

A special acknowledgment goes to Dr. Naren Ramakrishnan and Dr. Chang-Tien Lu from the NVC campus. Both are extraordinary researchers and leaders who have provided me with much-needed support during challenging times.

Lastly, I owe an immense debt of gratitude to my family and friends, whose relentless support, encouragement, and belief in me were the pillars of strength that kept me going. Without them, this journey would have been far more lonely and difficult.

This thesis is not just a result of my individual effort but the culmination of the contributions of many incredible people. To them, I offer my deepest thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Machine learning for natural language processing (ML4NLP) has made substantial progress over the last few years. The bag-of-words (BOW) [41] and TF-IDF [116] approaches were introduced in the last century and have been used for comprehending natural language (NL) sequences in a wide range of applications. To capture the directional dependency in sequences, Recurrent Neural Networks (RNNs) [43] were introduced to leverage context in sentences. Word2Vec [80] represents word semantics as distributional vectors learned from the context, which was widely used as static pre-trained embeddings. In 2017, the Transformer [127] model was introduced as a fully attention-based encoder-decoder model. It has been the dominating architecture of language models since then. In 2018, BERT [23] opened a new era of pre-training and fine-tuning for NLP. This learning paradigm has been the most effective method since then and revolutionized many language tasks through large-scale open domain self-supervised learning.

However, not all domains benefit equally from the rapid development of neural methods for sequential data. Some domains remain under-explored despite having large amounts of sequential data which contains rich knowledge. In the healthcare domain, there are many online knowledge bases such as Patient.info, WebMD and Healthline that house hundreds of thousands of topics and articles about biomedical information. In the software engineering

domain, there are websites like GitHub and GeeksforGeeks that host millions of open source codes in a number of programming languages. They can benefit a wide range of downstream tasks and a large population of users, if effective neural methods can be applied to the sequential data from these domains.

There are a number of challenges hindering the effective application of neural sequence modeling methods to domain-specific languages.

- **(1) Comprehension of domain-specific sequences.** The distribution of the data in specific domains is usually different from the general domain. For example, the average sequence length, the vocabulary, and the distribution of information can vary a lot across domains, which pose unique challenges for domain-specific language understanding.
- **(2) Absence of Domain Knowledge.** The effective comprehension of domain-specific data usually relies on domain knowledge, which is not explicitly presented in the sequences.
- **(3) Lack of Labelled Data.** Many successful methods heavily rely on labelled data, while in domain-specific settings, labelled data are usually scarce and expensive to get. Besides the common challenges across different domains, each domain also poses its own unique challenges.

There is a collection of literature that seeks to address the above challenges. One line of work focuses on large-scale pre-training with more data, more model parameters and longer training time [14, 23, 71, 99]. It has achieved state-of-the-art results on many domain-specific tasks and the performance improvement with respect to growing model sizes seems far from saturation. However, large-scale pre-training results in model size growing exponentially from ElMo's [95] 94M parameters to Megatron-Turing's [122] 530B, making the training and inference expensive and inefficient. Another line of work focuses on leveraging pre-

trained models to transfer knowledge from the general domain to specific domains [45, 114]. However, they face the new challenges of catastrophic forgetting [29, 59, 77] in training, and distribution shift in evaluation.

1.2 Outline

In this thesis, we explore a systematic approach to apply neural sequence modeling methods to domain-specific data for efficient and effective language understanding and generation. Our work focuses on two specific domains, healthcare and software engineering, which both contain rich sequential data and have a wide range of applications. We aim at addressing both common and domain-specific challenges in these two domains. More specifically, we investigate three research directions that play key roles in addressing the domain-specific language learning challenges: i) learning with long sequences, ii) learning from domain knowledge and iii) learning under limited supervision.

1.2.1 Part I: Learning with Long Sequences

This part delves into the critical challenges related to information retrieval and question-answering with long sequences of data, particularly within the healthcare domain. The proposed solutions focus on developing advanced neural architectures to efficiently process and retrieve relevant information from these sequences.

Chapter 3. A Hierarchical Attention Retrieval Model for Healthcare Question Answering [148]. The growth of the Web in recent years has resulted in the development of various online platforms that provide healthcare information services. These platforms contain large amounts of valuable information that could be beneficial for patients and other

types of information-seeking users. However, navigating through such knowledge bases to answer specific queries of healthcare consumers is a challenging task. A majority of such queries might be non-factoid in nature, and hence, traditional keyword-based retrieval models do not work well for such cases. Furthermore, in many scenarios, it might be desirable to get a short answer that sufficiently answers the query, instead of a long document with only a small amount of useful information. In this chapter, we propose a neural network model for ranking documents for question answering in the healthcare domain. The proposed model uses a deep attention mechanism at word, sentence, and document levels, for efficient retrieval for both factoid and non-factoid queries, on documents of varied lengths. Specifically, the word-level cross-attention allows the model to identify words that might be most relevant for a query, and the hierarchical attention at sentence and document levels allows it to do effective retrieval on both long and short documents. We also construct a new large-scale healthcare question-answering dataset, which we use to evaluate our model. Experimental evaluation results against several state-of-the-art baselines show that our model outperforms the existing retrieval techniques.

Hypothesis:

- For document ranking with respect to queries, an algorithm that assigns different weight to each sentence based on the query and context leads to better query-document matching, as compared to treating all sentences with equal importance.

Research Questions:

1. What kind of neural architecture can capture the hierarchical relevance between the query and each sentence of the document and the whole document?
2. Do the sentence-level attention weights reflect how likely a sentence is the answer to the given query?

3. How can the model capture the inter-sentence relationship among the answer spans of the query in a document?

Chapter 4. Question Answering with Long Multiple-Span Answers [149]. Answering questions in many real-world applications often requires complex and precise information excerpted from texts spanning across a long document. However, currently no such annotated dataset is publicly available, which hinders the development of neural question-answering (QA) systems. To this end, we present *MASH-QA*, a **M**ultiple **A**nswer **S**pans **H**ealthcare **Q**uestion **A**nswering dataset from the consumer health domain, where answers may need to be excerpted from multiple, nonconsecutive parts of text spanned across a long document. We also propose *MultiCo*, a neural architecture that is able to capture the relevance among multiple answer spans, by using a query-based contextualized sentence selection approach, for forming the answer to the given question. We also demonstrate that conventional QA models are not suitable for this type of task and perform poorly in this setting. Extensive experiments are conducted, and the experimental results confirm the proposed model significantly outperforms the state-of-the-art QA models in this multi-span QA setting.

Hypothesis:

- For questions where the answers span over non-consecutive sentences in a document, an algorithm that simultaneously selects all relevant spans yields better answers to queries than an algorithm that selects each sentence independently.

Research Questions:

1. How can the model capture the inter-sentence relationship among the answer spans of the query in a document?
2. How important is the context in selecting the answer span given a query?

3. How does multi-span answer selection method compare to single-span extraction methods in answering questions with single-span answers?

1.2.2 Part II: Learning from Domain Knowledge

In this part, we shift our focus to harnessing domain knowledge to enhance biomedical entity linking and cross-lingual code intelligence. The studies in this part highlight the importance of understanding and leveraging the intrinsic structures and latent properties within domain-specific data.

Chapter 5. Latent Type Modeling for Biomedical Entity Linking [150]. Entity linking is the task of linking mentions of named entities in natural language text, to entities in a curated knowledge-base. This is of significant importance in the biomedical domain, where it could be used to semantically annotate a large volume of clinical records and biomedical literature, to standardized concepts described in an ontology such as Unified Medical Language System (UMLS). We observe that with precise type information, entity disambiguation becomes a straightforward task. However, fine-grained type information is usually not available in the biomedical domain. Thus, we propose **LATTE**, a **LAT**ent **T**ype **E**ntity Linking model, that improves entity linking by modeling the latent fine-grained type information about mentions and entities. Unlike previous methods that perform entity linking directly between the mentions and the entities, LATTE jointly does entity disambiguation and latent fine-grained type learning, without direct supervision. We evaluate our model on two biomedical datasets: MedMentions, a large scale public dataset annotated with UMLS concepts, and a de-identified corpus of dictated doctor's notes that has been annotated with International Classification of Diseases (ICD) concepts. Extensive experimental evaluation shows our model achieves significant performance improvements over several state-of-the-art

techniques.

Hypothesis:

- Modeling latent properties with the guidance of domain knowledge yields improved results in biomedical entity linking, as compared to models that do not incorporate domain knowledge.

Research Questions:

1. What kind of neural architecture can model the latent types of entity mentions and candidates with the guidance of entity types?
2. How can the model learn from the entity type information in entity linking?

Chapter 6. A Machine Learning Benchmark for Cross-lingual Code Intelligence [151]. Recent advances in machine learning have significantly improved the understanding of source code data and achieved good performance on a number of downstream tasks. Open source repositories like GitHub enable this process with rich unlabeled code data. However, the lack of high quality labeled data has largely hindered the progress of several code related tasks, such as program translation, summarization, synthesis, and code search. This chapter introduces *XLCoST*, **Cross-L**ingual **Co**de **S**nippe**T** dataset, a new benchmark dataset for cross-lingual code intelligence. Our dataset contains fine-grained parallel data from 8 languages (7 commonly used programming languages and English), and supports 10 cross-lingual code tasks. To the best of our knowledge, it is the largest parallel dataset for source code both in terms of size and the number of languages. We also provide the performance of several state-of-the-art baseline models for each task. We believe this new dataset can be a valuable asset for the research community and facilitate the development and validation of new methods for cross-lingual code intelligence.

Hypothesis:

- Utilizing similar code comments across different programming languages to obtain fine-grained parallel data will enhance the development and validation of machine learning methods in this domain.

Research Questions:

1. How can we efficiently construct a benchmark dataset that covers a wide range of programming languages and tasks?
2. How can the quality of the data be ensured in this dataset?

1.2.3 Part III: Learning under Limited Supervision

The final part of the thesis deals with the challenge of learning under limited supervision. The core idea in this section is to demonstrate how one can leverage semi-supervised learning techniques and multilingual pre-training to enhance program translation.

Chapter 7. Multilingual Code Snippets Training for Program Translation [152]. Program translation aims to translate source code from one programming language to another. It is particularly useful in applications such as multiple-platform adaptation and legacy code migration. Traditional rule-based program translation methods usually rely on meticulous manual rule-crafting, which is costly both in terms of time and effort. Recently, neural network based methods have been developed to address this problem. However, the absence of high-quality parallel code data is one of the main bottlenecks which impedes the development of program translation models. In this chapter, we introduce *CoST*, a new multilingual **Co**de **S**nippet **T**ranslation dataset that contains parallel data from 7 commonly used programming languages. The dataset is parallel at the level of code snippets, which provides much more fine-grained alignments between different languages than the existing translation datasets. We also propose a new program translation model that leverages mul-

tilingual snippet denoising auto-encoding and **Mu**ltilingual **S**nippet **T**ranslation (MuST) pre-training. Extensive experiments show that the multilingual snippet training is effective in improving program translation performance, especially for low-resource languages. Moreover, our training method shows good generalizability and consistently improves the translation performance of a number of baseline models. The proposed model outperforms the baselines on both snippet-level and program-level translation, and achieves state-of-the-art performance on the CodeXGLUE translation task.

Hypothesis:

- Performing multilingual pre-training with snippet-level parallel code data enhances program-level translation performance, particularly for languages with low resource availability.

Research Questions:

1. How can the model effectively learn from the snippet-level parallel data?
2. How should the pre-training tasks be designed to align with the translation task?
3. How does multilingual snippet training perform on low resource languages?

Chapter 8 Alignment-Enhancing Parallel Code Generation for Semi-Supervised Code Translation. Code translation is the task of converting source code from one programming language to another. Sufficient parallel code data is essential for neural code translation models to learn the correct alignment across different languages. However, existing parallel code data is limited in quantity and supported languages. In this chapter, we propose a semi-supervised code translation method, **SPACoder**, that leverages snippet training, static analysis, and compilation to generate synthetic parallel code with enhanced alignment in a scalable way, and improves code translation by curriculum learning based on the alignment level of training instances. SPACoder can be generalized to multiple languages

and various models with little overhead. Extensive experiments show that SPACoder significantly improves code translation performance on C++, Java, Python, and C, outperforming state-of-the-art baselines by wide margins in execution-based evaluation (CA@1). Notably, we improve C translation by up to 43% with less than 150 annotated training instances.

Hypothesis:

- Curriculum learning on large amounts of noisy synthetic parallel code improves neural code translation.

Research Questions:

1. How can large amounts of synthetic parallel code be generated in a cost-efficient manner?
2. How can the alignment quality be improved in the synthetic parallel code?
3. How can the model effectively learn from the synthetic parallel code to improve neural code translation?

1.3 Contributions

Through this thesis, we sought to address challenges in neural sequence modeling for domain-specific language processing, focusing on the healthcare and software engineering domains. The chapters encapsulated in this thesis, grouped into three significant research directions, have led to new insights and advancements in the field.

Our initial studies in Part I revolved around efficiently retrieving and processing information from long sequences in the healthcare domain. We presented an attention-based hierarchical model, which demonstrated improved ranking for healthcare-related queries. We further extended this work to handle questions with long multiple-span answers. Our findings in

this section provide a roadmap for future work aimed at making healthcare information more accessible to the public and advancing the field of information retrieval.

In Part II, we delved into the utilization of domain knowledge for improving biomedical entity linking and cross-lingual code intelligence. By integrating latent type modeling, we managed to enhance the biomedical entity linking task. Further, our cross-lingual code intelligence benchmark has broadened the scope for development and evaluation of models in the software engineering domain. These studies reflect the value of domain knowledge in processing domain-specific data and inspire future work in this area.

Finally, in Part III, we tackled the challenge of learning under limited supervision for programming languages. Through the multilingual snippet training and semi-supervised parallel code generation, we showcased the significant potential of leveraging huge amount of unlabeled code data for improving code generation tasks, particularly in code translation.

In conclusion, this thesis, through its various hypotheses and research questions, lays a foundation for the systematic application of neural sequence modeling in domain-specific language processing. We hope our findings and contributions will be instrumental in furthering research in these areas and prove beneficial in an even broader range of domains containing large volumes of sequential data.

Chapter 2

Review of Literature

2.1 Neural Information Retrieval

With the success of deep neural networks in learning feature representation of text data, several neural ranking architectures have been proposed for text document search. Deep Structured Semantic Model (DSSM) [48] uses a simple feed-forward network to learn the semantic representation of queries and documents. It then computes the similarity between their semantic representations using cosine similarity. Convolutional Deep Structured Semantic Model (CDSSM) [121] uses convolutional layers on word trigram features, while the model proposed in [88] uses the last state outputs of LSTM encoders as the query and document features. Both these models then use cosine similarity between query and document representations, to compute their relevance. In [47], the authors propose convolutional neural network models for semantic matching of documents. The Architecture-I (ARC-I) model proposed in this work also uses a convolutional architecture to create document-level representations of query and document, and then uses a feed-forward network to compute their relevance. The InferSent Ranker [46] also uses a feed forward network to compute the relevance between query and documents, by summing up their sentence embeddings. All these methods use the document-level semantic representation of queries and documents, which is basically a pooled representation of the words in the document. However, in the majority of the cases in document retrieval, it is observed that the relevant text for a query is a very

short piece of text from the document. Hence, matching the pooled representation of the entire document with that of the query does not give very good results, as the representation also contains features from other less relevant parts of the document.

To overcome the problems of document-level semantic-matching based IR models, several interaction-based IR models have been proposed recently. In [38], the authors propose Deep Relevance Matching Model (DRMM), that uses word count based interaction features between query and document words, while the Architecture-II (ARC-II) proposed in [47] uses convolution operations to compute the interaction features. These features are then fed to a deep feed-forward network for computing the relevance score. The models proposed in [20, 135] use kernel pooling on interaction features to compute similarity scores, while MatchPyramid [90] uses the dot product between query and document word vectors as their interaction features, followed by convolutional layers to compute the relevance score. Other methods that use word-level interaction features include Attention-based Neural Matching Model (aNMM) [137], which uses attention over word embeddings, and [129], that uses cosine or bilinear operation over Bi-LSTM features, to compute the interaction features. The Duet model proposed in [82] combines both word-level interaction features, as well as document-level semantic features, in a deep CNN architecture, to estimate the relevance. One common limitation of all these models is that they do not utilize the inherent paragraph and sentence level hierarchy in documents, and hence, they do not perform well in case of longer documents.

2.1.1 Document Ranking

Document retrieval and ranking is a classical problem in the information retrieval community, which has attracted significant interest from researchers for many years. Early

methods in informational retrieval were largely based on keyword-based query-document matching [108, 115, 116]. With the advancement of machine learning algorithms, better retrieval mechanisms have been proposed. Logistic Inference [34] used logistic regression probabilities to determine the relevance between queries and documents. In [53], the authors used a Support Vector Machine (SVM) based approach for retrieval, which allows the retrieval system to be trained using the search engine click logs. Other traditional techniques in information retrieval include boosting-based methods [30, 136]. TF-IDF based similarity [104] and Okapi BM25 [109] are the most popularly used term-based techniques for document search and ranking. However, such techniques usually do not perform well, when the documents are longer [75], or have minimal exact word overlap with the query.

2.1.2 Passage Retrieval

Passage retrieval focuses on retrieving specific sections within documents relevant to a query. In the past few years, transformer-based [127] models, particularly BERT [23] and its variants [71, 125], have revolutionized the field with their ability to incorporate contextual information in the representation of text. This has resulted in a range of innovative methodologies for retrieval tasks, such as Dense Passage Retrieval (DPR) [57], which completely sidesteps term-matching and utilizes BERT embeddings for both queries and documents to calculate relevance. More recent developments have seen the incorporation of increasingly complex retrieval models. The two-stage approach in REALM [40] is one such example, blending the benefits of term-based and dense retrieval. Another area of focus is end-to-end trainable models that incorporate retrieval as part of the training process [32, 68]. Open-domain question answering [15] and the integration of retrieval and generation [50] have also been explored. Passage retrieval remains a challenge due to the complexity and variability of language, making it a vibrant area of ongoing research.

2.1.3 Information Retrieval for Healthcare

Early works in the domain of information retrieval for medicine and healthcare used tradition search methods such as TF-IDF [76] and BM25 [74]. MedQA [144] uses hierarchical clustering along with TF-IDF for answering definitional questions asked by physicians. The question-answering system proposed in [22] aimed at helping clinicians to search for treatments for any disease. However, their system is tailored to answer one specific type of question, and cannot be used to answer open-ended questions. As discussed in [67], physicians also often need to use such systems, and they have limited time to browse through every returned document. Hence, medical retrieval needs to be accurate, and should precisely serve the requirements of the users. Retrieval in this domain is complex, attributed to the non-factoid nature of queries, and longer documents. Hence, traditional IR techniques or semantic matching algorithms do not work well on such datasets.

2.2 Neural Machine Comprehension

Earlier works in QA used similarity based models for classifying answers based on their semantic similarity with the document [79, 145]. The public release of the SQuAD dataset motivated the development of attention-based neural models [134]. DrQA Reader [15] uses an RNN-based architecture, along with context-to-query attention, to compute the answer. BiDAF [120] uses bidirectional attention (query-to-context and context-to-query) for answer span prediction. With the advancements in language modeling (LM) techniques such as BERT [23] and XLNet [139], LM-based techniques have gained more popularity in recent times.

2.2.1 Question Answering Datasets

The WikiQA dataset [138] contains query-sentence pairs, and their relevance labels, based on articles from Wikipedia. The SQuAD datasets [102, 103] consist of question-answer pairs based on Wikipedia articles. The questions, however, are generally factoid, the answers are short, and the context is a small paragraph. The Natural Questions dataset [62] provides a more realistic setting, where the context is a full Wikipedia page, and the answer is a short snippet from the article. Some of the questions also include a long answer. MS-MARCO [87], SearchQA [24], and TriviaQA [54] contain questions and a short answer, and the questions are supported by more than one context document, some of which might be irrelevant to the question. CoQA [106] and NarrativeQA [60] are free-form QA datasets, where the answer is a short, free-form text, not necessarily matching a snippet from the context. ELI5 [26] is a long, free-form QA dataset, based on questions and answers from Reddit forums. However, since the evidence documents are collected using web-search, only 65% of supporting documents contain the answer.

2.2.2 Question Answering Datasets in Healthcare

Recently, many QA datasets from the medical domain have also been proposed. MedQUAD [1] and HealthQA [148] are consumer health QA datasets, that contain query-answer tuples, and their relevance labels. emrQA [89] contains rule-based questions constructed from medical records, while questions in CLiCR [123] are based on clinical report summaries.

2.3 Neural Entity Linking

Neural entity linking has attracted significant interest from researchers for many years. Most of the existing techniques in this domain can broadly be classified into three categories. *Context modeling* approaches model the context of mentions and the candidate entities at different levels of granularity to get the similarity between them. An example of such approaches is [28], which uses a set of vectors that include mention, mention context and the document that mention appears in, and vectors from entity article title and document, to compute the mention and entity representations, respectively. [39] also extensively makes use of context information in the entity linking process. *Type modeling* approaches make use of the entity types in the linking process. This is based on the observation that if the entity types are known, entity linking performance can be improved significantly [101]. The *Relation Modeling* approach models the latent relations between different mentions without direct supervision [66]. These relations can be used as a measure of coherency of the linking decisions, and can ultimately guide the entity linking process.

2.4 Neural Code Translation

2.4.1 Cross-Lingual Code Tasks

Cross-Lingual tasks in the code domain include Code Translation, Code Summarization, Code Synthesis, and Code Search. CodeBERT [27] pre-trained a BERT [23] based encoder on the source code, and then added a decoder to perform end-to-end training on code translation. CodeBERT is also used for Code Search tasks. PLBART [5] utilized an existing natural language translation model, BART [68], and also pre-trained it with source code. CodeTransformer [153] uses language agnostic features computed from the source code and

its abstract syntax tree for code summarization. OpenAI's Codex [16] framework makes use of GPT [98] language models fine-tuned on publicly available code from GitHub for code related downstream tasks. However, most of the models only explored a limited number of languages, due to the scarcity of multilingual parallel data.

2.4.2 Parallel Code Data

CodeXGLUE [73] is a popular benchmark that includes 14 datasets for 10 code related tasks. The tasks include clone detection, code translation, natural language code search, etc. However, this benchmark does not contain datasets with parallel codes from more than 2 languages. CoST Zhu et al. [152] is a code translation dataset for 7 programming languages. However, it is relatively small and only supports the translation task. AVATAR [6] presents another parallel dataset for Java-Python translation. The authors collect multiple solutions for problems scraped from competitive programming websites and then form n^2 possible combinations of parallel data. This is also constrained to only 2 languages. Project CodeNet [96] has an abundance of parallel programs in a wide range of languages. However, the programs are significantly different in logic and structure, thus the alignment is of low quality. Transcoder [110] introduces a dataset that supports computational accuracy evaluation, which is an execution-based evaluation metric. However, it does not provide pairwise data for training.

2.4.3 Neural Program Translation

One line of work has directly applied recent advances in natural language processing (NLP) to the programming language domain. Transcoder [110] combined cross-lingual masked language modeling [63], denoising auto-encoding, and back-translation, and applied them to

a source code setting. Another line of work incorporates the intrinsic features of programming languages to improve translation performance. [17] modeled this problem as translating a source tree into a target tree. GraphCodeBERT [37] improved upon CodeBERT [27] by adding data-flow graphs extracted from source code, improving the model's understanding of the code structure. Some other works [13, 97, 142] also make use of abstract syntax trees (ASTs) derived from the code. DOBF [112] added a de-obfuscation objective to the masked language model pre-training to leverage the structural aspect of programming languages.

Part I

Learning with Long Sequences

Chapter 3

A Hierarchical Attention Retrieval Model for Healthcare Question Answering

The growth of the Web in recent years has resulted in the development of various online platforms that provide healthcare information services. These platforms contain large amounts of valuable information that could be beneficial for patients and other types of information-seeking users. However, navigating through such knowledgebases to answer specific queries of healthcare consumers is a challenging task. Few retrieval models are focused on situations where the queries are non-factoid. Furthermore, in many scenarios, it might be desirable to get a short answer that sufficiently answers the query, instead of a long document with only a small amount of useful information. In this chapter, we propose a neural network model for ranking documents for question answering in the healthcare domain. The proposed model uses a deep attention mechanism at word, sentence, and document levels, for efficient retrieval for both factoid and non-factoid queries, on documents of varied lengths. Specifically, the word-level cross-attention allows the model to identify words that might be most relevant for a query, and the hierarchical attention at sentence and document levels allows it to do effective retrieval on both long and short documents. We also construct a new large-scale healthcare question-answering dataset, which we use to evaluate our model. Experimental

evaluation results against several state-of-the-art baselines show that our model outperforms the existing retrieval techniques. This chapter is adapted from a paper [148] published in The Web Conference in 2019, of which I am the first author and primary contributor.

3.1 Introduction

With the growth of the Web in recent years, a vast amount of health-related information is now publicly available on the Internet. Many people use online health information platforms such as WebMD[1] and Patient[2] to search for information regarding the symptoms, diseases, or any other health-related information they are interested in. In addition to consumers, often doctors and healthcare professionals need to look into knowledgebases that contain detailed healthcare information about diseases, diagnoses, and procedures [35, 117]. Despite the abundance of available information, it might be difficult for healthcare consumers to navigate through these documents to get the required healthcare information. Hence, effective retrieval techniques are required to allow consumers to efficiently use such platforms. Since healthcare documents usually include several details about the disease such as its symptoms, preventive measures, and common treatments, they are usually more elaborate, compared to other factual documents, which describe well-known facts (e.g., population of a town, capital of a country, or any other entity), and are very specific in nature. Hence, in such cases, it might be desirable to provide the consumers with a short piece of text that succinctly answers their queries. Furthermore, many questions that users have about health-related topics are very abstract and open-ended in nature, and hence traditional search methods do not work well in such cases.

Prompted by the success of deep neural networks in language modeling, researchers have

[1] https://www.webmd.com/
[2] https://patient.info/

What would happen if I didn't take antithyroid medicines?
It is usually advisable to treat an overactive thyroid gland (hyperthyroidism). Untreated hyperthyroidism can cause significant problems with your heart and other organs. It may also increase your risk of complications should you become pregnant. However, in many cases there are other treatment options. That is, radioactive iodine or surgery may be suitable options. See the separate leaflet called Overactive Thyroid Gland (Hyperthyroidism) for details of these other treatment options.

Figure 3.1: An example of a healthcare question, and its corresponding answer. The question and answer do not have any overlapping words. The highlighted text corresponds to the most relevant answer snippet from the document.

proposed several techniques that apply neural networks for effective information retrieval [38, 82] and question answering [120, 131]. This has been facilitated primarily due to the development of large training datasets such as TREC [128] and SQuAD [102]. However, both these datasets are primarily composed of factoid queries / questions, and the answers are generally short in length. Hence, systems trained on such datasets cannot perform well in a setting where a large proportion of the queries are non-factoid and open-ended, and the documents are relatively longer in length. Figure 3.1 shows an example of a typical question that a consumer would have regarding antithyroid medicines, and its corresponding answer paragraph, selected from the website *Patient.info* [3]. This domain-specific problem provides some unique challenges which require us to build a more comprehensive retrieval system.

- **Minimal overlap between question and answer words:** There is minimal or no word overlap between the question and answer text. As there are no matching terms, traditional keyword-based search mechanisms will not work for answering such

[3] https://patient.info/

questions.

- **Length of question and answer:** The question is longer than a typical search engine query. The answer is also typically longer than a sentence. Although, for illustration purposes, we show a short paragraph, in many cases, the answer, as well as the document containing it, might be even longer. Hence, neural semantic matching algorithms will not be effective in such cases, as they are ideally designed for short sentences. Therefore, an effective retrieval system would require a mechanism to deal with documents of varied lengths.
- **Non-factoid nature:** The question is very open-ended in nature, and does not ask for any specific factual details. However, a majority of the machine comprehension models are trained on datasets like SQuAD, which are comprised of factoid QA pairs. Such systems do not work well in a setting where the desired answer is more elaborate.

To overcome these problems, we propose **HAR**, a **H**ierarchical **A**ttention **R**etrieval model for retrieving documents for healthcare related queries. The proposed model uses a cross-attention mechanism between the query and document words to discover the most important words that are required to sufficiently answer the query. It then uses a hierarchical inner attention, first over different words in a sentence, and then over different sentences in a document, to successively select the document features that might be most relevant for answering the query. Finally, it computes a similarity score of a document with the query, that could be used to rank different documents in the corpus, given a query. The use of hierarchical attention also enables it to find the most important sentences and words, that could be important to answer a query, without the need of using an explicit machine comprehension module. To evaluate the performance of our model, we construct a large scale healthcare question answering dataset, using knowledge articles collected from the popular health services website *Patient.info*. Although we use this model in the healthcare domain,

where the questions are usually non-factoid in nature, and the documents are longer due to the presence of detailed description about different medical procedures, our model can be used in any other domain where the questions are open-ended, and the documents are longer.

The rest of this chapter is organized as follows: In Section 3.2, we describe our proposed neural retrieval model termed HAR, and provide the details about its architecture and the training procedure, including the optimization for the HAR model. The details about the data collection and annotation have been described in Section 3.3. In Section 3.4, we give details about our experimental evaluation, and the metrics and baseline techniques used in the evaluation process. Finally, Section 3.5 concludes the chapter, with possible directions for future research.

3.2 The Proposed Model

In this section, we introduce our proposed Hierarchical Attention Retrieval (HAR) model, which uses a deep attention mechanism for effective retrieval. The detailed architecture of our model is shown in Fig. 3.2. HAR is a novel neural network model that uses two powerful attention mechanisms to overcome the shortcomings of existing document retrieval models. Given a query q, the model computes a relevance score r_i with each candidate document d_i in the document knowledgebase $\mathcal{D}$. The different components of our model are described in detail below.

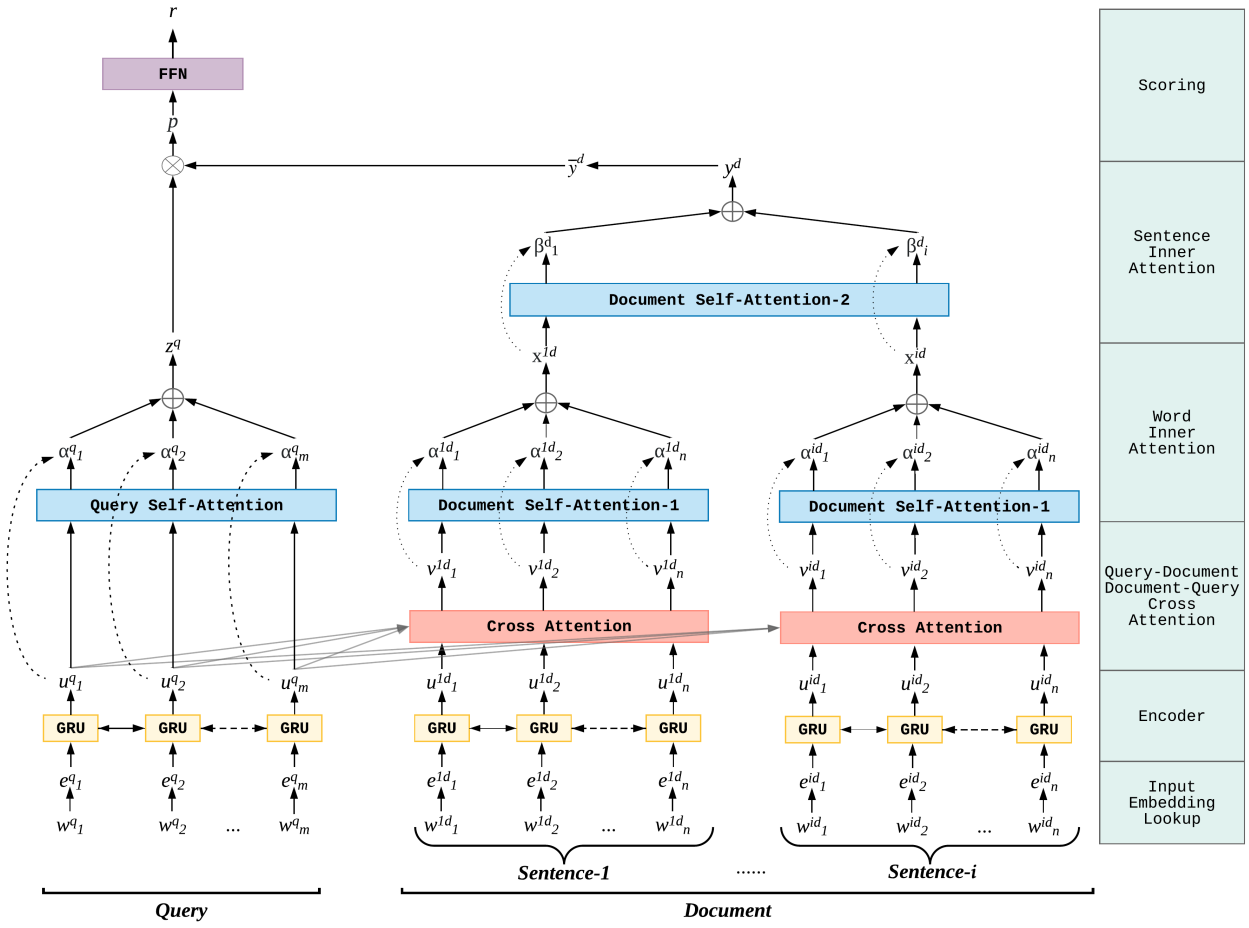

Figure 3.2: Architecture of the proposed HAR model.

3.2.1 Word Embeddings

The input layer in our model is an embedding lookup function which converts the query q and document words into fixed K-dimensional word vectors using a lookup matrix $\boldsymbol{E} \in \mathbb{R}^{V \times K}$ of V pre-trained word embeddings such as GloVe [93] or Word2Vec [81]. Let $\{w_t^q\}_{t=1}^m$ be the words in q. Let l be the number of sentences in document d, and $\{w_t^{id}\}_{t=1}^n$ be the words in sentence i in d. This layer converts each of the words in q and d into the word vectors $\{e_t^q\}_{t=1}^m$ and $\{e_t^{id}\}_{t=1}^n$, respectively. Here, m and n are the number of words in the query and each of the document sentences, respectively.

3.2.2 Encoder

We use two bidirectional RNN (Bi-RNN) [118] encoders to encode the inter-document sequential dependencies within query and documents words, respectively. This layer consists of two RNN layers in different directions, whose output is concatenated to get the H-dimensional contextual representation of each word. We split the documents into short sentences and encode all the sentences in parallel. Long sentences are segmented into shorter ones. We choose GRU [18] over vanilla-RNN or LSTM [43] because of its high performance and computational efficiency. Since we encode the document by sentence, GRU performs equally well as LSTM, because the encoder does not need to deal with very long sequences.

Query Encoder

The query encoder contains a simple Bi-GRU layer, which takes the query word embeddings $\{e_t^q\}_{t=1}^m$ as the input, and outputs the contextual representation $U^q = \{u_t^q\}_{t=1}^m \in \mathbb{R}^{m \times H}$.

$$u_t^q = BiGRU_Q(u_{t-1}^q, e_t^q) \tag{3.1}$$

Document Encoder

Since documents are usually longer than queries, we encode each sentence in the document separately, using a sentence-level Bi-GRU encoder. Given a sentence i, this layer takes the sentence word embeddings $\{e_t^{id}\}_{t=1}^n$ as the input, and returns the contextual word embeddings $U^{id} = \{u_t^{id}\}_{t=1}^n \in \mathbb{R}^{n \times H}$. After encoding each of the l sentences in the document through this encoder, the new document representation is $\{U^{1d}, .., U^{ld}\} \in \mathbb{R}^{l \times n \times H}$.

$$u_t^{id} = BiGRU_D(u_{t-1}^{id}, e_t^{id}) \tag{3.2}$$

3.2.3 Cross Attention between Query and Document

This layer is used to fuse the information from query words into the document words. It computes the relevance of each query word with respect to each word in the document. As we use a hierarchical modeling for documents, this layer can compute the attention-based embeddings of each word in sentence i in d with each word in the query q. We use the cross attention mechanism proposed in [133, 143], as this method has been proven to show superior performance in state-of-the-art reading comprehension systems. The attention layer computes the relevance between each pair of query and document words, using their contextual embeddings generated by the respective encoders. To calculate the relevance of query words with respect to document words, and vice-versa, we use a bi-directional attention mechanism [120], which is composed of document-to-query attention $D2Q$ and query-to-document attention $Q2D$. This is done by first computing a similarity matrix $S \in \mathbb{R}^{n \times m}$, which is then normalized over each row and column, using a normalization operation such as softmax. This generates normalized similarity matrices $\overline{S}_{D2Q} \in \mathbb{R}^{n \times m}$ and $\overline{S}_{Q2D} \in \mathbb{R}^{n \times m}$, respectively. Finally, the attention matrices $A_{D2Q} \in \mathbb{R}^{n \times H}$ and $A_{Q2D} \in \mathbb{R}^{n \times H}$ can be computed as described below.

Let $s_{xy} \in \mathbb{R}$ be an element of the similarity matrix S from row x and column y. Given $U^{id} \in \{U^{1d}, .., U^{ld}\}$ and U^q as inputs, the final output $V^{id} = \{v_t^{id}\}_{t=1}^n \in \{V^{1d}, .., V^{ld}\}$ of the cross attention layer can be computed as follows:

$$s_{xy} = w_c^T \cdot [u_x^{id}; u_y^q; u_x^{id} \odot u_y^q] \tag{3.3}$$

$$\overline{S}_{D2Q} = \text{softmax}_{\text{row}}(S) \tag{3.4}$$

$$\overline{S}_{Q2D} = \text{softmax}_{\text{col}}(S) \tag{3.5}$$

$$A_{D2Q} = \overline{S}_{D2Q} \cdot U^q \tag{3.6}$$

$$A_{Q2D} = \overline{S}_{D2Q} \cdot \overline{S}_{Q2D}{}^T \cdot U^{id} \tag{3.7}$$

$$V^{id} = [U^{id}; A_{D2Q}; U^{id} \odot A_{D2Q}; U^{id} \odot A_{Q2D}] \in \mathbb{R}^{n \times 4H} \tag{3.8}$$

In the above equations, ; is the concatenation operation, $\odot$ is element-wise multiplication, $\cdot$ is matrix multiplication, and $w_c \in \mathbb{R}^{3H}$ is a trainable weight vector.

3.2.4 Query Inner Attention

To encode variable length queries into a fixed size embedding, we use the self attention mechanism proposed in [70]. The importance of different words varies from document to document, and is dependent on the context in which they are used. This layer allows the model to give higher priority to more important words while creating a pooled representation of the query. This ensures that the query representation contains features from more significant words. Let A be the dimension of the pooled representation. Given query features $U^q = \{u_t^q\}_{t=1}^m$ as the input, this layer generates a pooled representation of $z^q \in \mathbb{R}^H$ as follows:

$$c_t^q = w_q^T(\tanh(W_q u_t^q)) \tag{3.9}$$

$$\alpha_t^q = \frac{\exp(c_t^q)}{\sum_{j=1}^{m} \exp(c_j^q)} \tag{3.10}$$

$$z^q = \sum_{t=1}^{m} \alpha_t^q u_t^q \tag{3.11}$$

3.2.5 Document Hierarchical Inner Attention

Since documents are longer in length, it is not necessary that the entire document is relevant to a query. In fact, in most cases, it is observed that part of the document that is relevant to a query is just a few sentences. Even inside each sentence, different words might have varying relevance to the query. Furthermore, because of the varied lengths of documents, a mechanism is required to get a fixed-dimensional representation of the document. Hence, we use a two-level hierarchical inner attention (as proposed in [140]), to get the document embedding.

Level-1: Attention over words in a sentence

The first level in our hierarchical attention encodes each sentence independently from other sentences at word-level, resulting in a fixed-dimensional representation of each sentence. This layer computes the importance of each word within the sentence, and then creates a pooled representation of each sentence weighted by the attention weights. For each sentence i in the document d, this layer takes the output vectors $V^{id} = \{v_t^{id}\}_{t=1}^{n} \in \mathbb{R}^{n \times 4H}$ from the cross attention layer as the input, and returns a sentence vector $x^{id} \in \mathbb{R}^{4H}$.

$$c_t^{id} = w_{d1}^T(\tanh(W_{d1}v_t^{id})) \tag{3.12}$$

$$\alpha_t^{id} = \frac{\exp(c_t^{id})}{\sum_{j=1}^{n}\exp(c_j^{id})} \tag{3.13}$$

$$x^{id} = \sum_{t=1}^{n}\alpha_t^{id}v_t^{id} \tag{3.14}$$

Level-2: Attention over sentences in a document

To ensure that sentences more relevant to the query are given higher importance while computing the similarity score, we use a second inner attention to compute the document representation. This layer takes the sentence embeddings $\{x_i^d\}_{i=1}^l$ as the input, and returns a document vector $y^d \in \mathbb{R}^{4H}$ as the output.

$$b_i^d = w_{d2}^T(\tanh(W_{d2}x^{id})) \tag{3.15}$$

$$\beta_i^d = \frac{\exp(b_i^d)}{\sum_{j=1}^{l}\exp(b_j^d)} \tag{3.16}$$

$$y^d = \sum_{j=1}^{l}\beta_j^d x^{jd} \tag{3.17}$$

3.2.6 Score Computation

The final layer in our model computes the score between the query representation z^q and document representation y^d. Since the dimension of y^d is 4 times the dimension of z^q, we first pass y^d through a feed-forward layer to compute $\overline{y}^d \in \mathbb{R}^H$. After this, we compute the similarity vector $p \in \mathbb{R}^H$ by performing element-wise multiplication of z^q and $\overline{y}^d$. Finally, we pass p through a feed-forward network to compute the final relevance score $r \in \mathbb{R}$.

$$\overline{y}^d = w_{d3}^T y^d + b_{d3} \tag{3.18}$$

$$p = z^q \odot \overline{y}^d \tag{3.19}$$

$$r = w_f^T p + b_f \tag{3.20}$$

3.2.7 Optimization

Negative sampling

Many retrieval datasets, such as the ones created using user click-logs, only have the query-document pairs, which serve as the positive data for the model. However, for the model to have sufficient discriminative power to give a score to every document proportional to their relevance with the query, the model also needs negative query-document pairs during the training process. Hence, we do negative sampling to generate negative data samples of query-document pairs for our model. For each query, the negative samples are composed of the following:

- *Irrelevant negative samples:* For the model to have a sufficient discriminative power that is needed to distinguish documents at a high level, we sample negative documents that have very low relevance to the query.
- *Partially relevant negative samples:* We define partially relevant negative documents as those that might have some relevance to the query, either due to some overlapping words, or because they are from the same topic, but do not contain the correct answer for the query. As suggested in [130], having such samples in the training dataset gives a higher discriminative power to the model, as compared to a model trained with randomly sampled negative pairs.

Loss function

We use pairwise maximum margin loss [53] as the objective function to be minimized. Given a query q, positive document d^{pos}, and k negatively sampled documents $\{d_1^{neg}, .., d_k^{neg}\}$, the loss is given by:

$$\mathcal{L} = \sum_{i=1}^{k} max(0, M - score(q, d^{pos}) + score(q, d_i^{neg})) \tag{3.21}$$

Here, M is the margin, by which we want the score of positive query-document pair to exceed that of a negative query-document pair.

3.3 HealthQA Dataset

We will now introduce the dataset created in this work to train and evaluate the proposed HAR model. We call this dataset *HealthQA*. It consists of question and document pairs from the healthcare domain. The details of this dataset are described below:

Table 3.1: Statistics of the HealthQA dataset.

Number of articles	1,235
Number of documents (article sections)	7,355
Number of questions	7,517
Average length of questions (in words)	8.04
Average length of documents (in words)	233.4
Average number of sentences in documents	13.54
Average length of sentences (in words)	17.24

3.3.1 Knowledge Articles

To create the HealthQA dataset, we collected healthcare articles from the popular health-services website Patient. We scraped all the articles from the *Health Topics* section of Patient. The website contains articles from a diverse set of healthcare domains such as child health, mental health, sexual health, details about treatments and medications, and several other healthcare domains. The articles on this website are much more detailed, as compared to other healthcare knowledgebases like MedlinePlus[4]. In total, we collected 1,235 health articles, with each article having an average of 6 sections. As the sections themselves are very long in these articles, we use each section as one document. Table 3.1 shows the statistics of the HealthQA dataset.

Figure 3.3 shows the distribution of percentage of documents with respect to the length of the documents (number of words). We can see that the proportion of documents with less than 50 words is fractional (5%). The dataset contains a large number of documents with 100-200 words, and a high proportion of documents containing more than 200 words.

[4]https://medlineplus.gov/

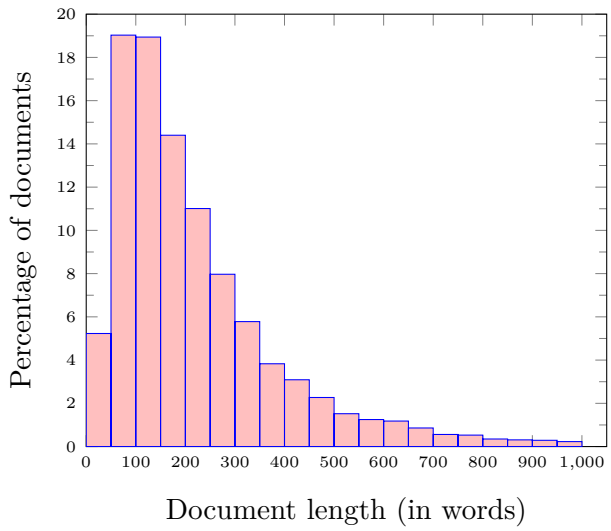

Figure 3.3: Percentage of documents vs. number of words.

3.3.2 Question-Answer Pair Generation

To create healthcare-related questions, we employed human workers from diverse age groups, and from different countries. For the dataset to have a diverse set of questions and answers that different people might have about healthcare, we hired six annotators, consisting of a combination of freelancers, graduate and undergraduate students. To ensure high quality of the dataset, and low error rates, we ensured that all the annotators had good English skills. For each document, workers were instructed to create 1 to 3 questions that can be asked using the information given in the documents. They were encouraged to use simple language in the queries, so that the questions follow the style of those asked by a common person, without any domain expertise in healthcare. All the generated questions also underwent an additional round of cross validation by the author, and any query-document pairs with errors or insufficient context were either corrected or discarded.

It was found that many articles on Patient have several subtitles, roughly one subtitle per paragraph, and in most cases, these subtitles could be rephrased into valid questions. Some

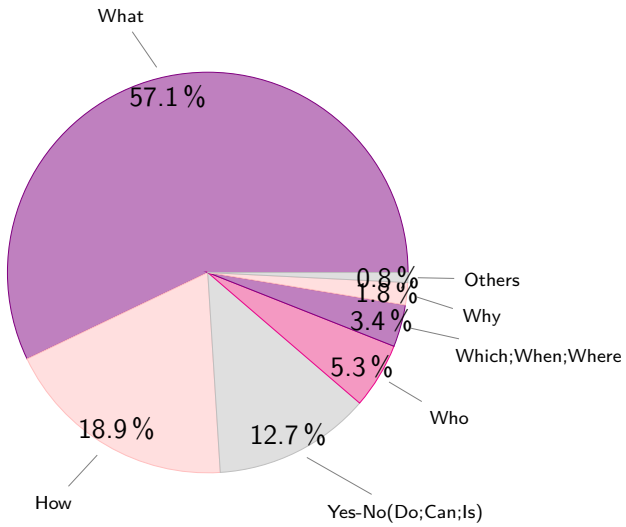

Figure 3.4: Percentage of questions by type.

of the titles have incomplete context, which can be made into a valid question by rephrasing them. Hence, workers were also allowed to use the subtitles as questions, by rephrasing them into valid questions.

Figure 3.4 shows the percentage of different types of questions in our dataset. The questions with the type "How" and "Why" are mostly non-factoid in nature. Such questions are open-ended, and require detailed answers. Although the questions with the type "What", that are generally factoid, have a large proportion in our dataset, after manual analysis, we found that a large proportion of such questions are also non-factoid. Examples of such questions include *"What can I expect in the future concerning gaming disorder?"*.

3.4 Experimental Results

3.4.1 Evaluation Metrics

We compare the performance of HAR with various baseline techniques using the following evaluation metrics:

- **Recall@K:** Recall@K for a query is defined as the ratio of the number of relevant documents in the top-K retrieved documents, with respect to the total number of relevant documents for that query. This is averaged over all the queries in the dataset. Since, in our case, each query has only one groundtruth document, Recall@K denotes the percentage of queries whose correct document was present in the top-K retrieved documents.

$$Recall@K = \frac{1}{Q_{test}} \sum_{i=1}^{Q_{test}} \frac{\#\text{ relevant documents in top-K for query } i}{\#\text{ total relevant documents for query } i} \tag{3.22}$$

- **Mean Reciprocal Rank (MRR):** The reciprocal rank is defined as the inverse of the rank of the first correct document d^{pos} for a given query. MRR is the mean of the reciprocal rank for all the queries in the test set. Since we have only one correct document for every query, MRR is well suited for our evaluation. As compared to Recall@K, MRR also takes into account the ranking order in the evaluation.

$$MRR = \frac{1}{Q_{test}} \sum_{i=1}^{Q_{test}} \frac{1}{rank(d^{pos}) \text{ of query } i} \tag{3.23}$$

3.4.2 Baseline Methods

For performance comparison, we use the following state-of-the-art retrieval models as the baselines:

- **TF-IDF:** This is the standard baseline for retrieval tasks, that uses TF-IDF representation for both the query and document, and cosine similarity as the similarity function.
- **CDSSM [121]:** It is an extension of DSSM, that uses a CNN-based model on letter trigrams from query and document as input features, and then computes the cosine similarity to calculate the relevance between query and document representations.
- **ARC-II [47]:** This model first computes the interaction feature vector between query and document using CNN layers. It then computes the score for the query-document interaction vector using a feed-forward network.
- **MV-LSTM [129]:** It is a neural semantic matching model that was proposed to find semantic similarity between a pair of sentences. The model uses the word embeddings obtained by passing the sentences through a Bi-LSTM, and then computes an interaction vector using cosine similarity, or a bilinear operation. It finally passes the interaction vector through a feed-forward network to compute the similarity score. In our implementation, we use cosine similarity to compute the interaction vector.
- **DRMM [38]:** It is a state-of-the-art neural ranking model that uses cosine similarity between query and document word vectors to compute their similarity, and then computes a histogram-like interaction vector by binning the cosine similarity scores into pre-defined intervals. It then passes these features through a feed-forward network to compute the scores.
- **KNRM [135]:** It is a neural ranking model that first computes cosine similarity between each query word with each of the document words. It then performs kernel

pooling, followed by a feed-forward network to compute the relevance score.

- **aNMM [137]:** This model first computes an interaction matrix by computing the cosine similarity between each of the query and document words. Similar to DRMM, this model also performs binning to compute a fixed-dimensional interaction vector. However, instead of using the counts of word-pairs that fall into a bin as its features, this model uses the total sum of the similarity between those word pairs as the bin features. It also uses an attention mechanism over the query word vectors, which is then combined with the interaction vector to compute the final relevance scores.
- **Duet [82]:** It is a hybrid neural matching model that uses the word-level interaction, and document-level similarity, in a deep CNN architecture, to compute similarity between two documents.
- **MatchPyramid [90]:** This model computes pairwise dot product between query and document word vectors to compute an interaction matrix. It then passes this matrix through CNN layers with dynamic pooling to compute the similarity score.

3.4.3 Implementation Details

The dataset was split into three parts: train, validation, and test. The number of queries in each split were 5274, 1109 and 1134 respectively. We implemented our model in Keras [36], with TensorFlow [2] as the backend. The model was trained using Adadelta optimizer [146], with an initial learning rate of 2.0. We used one Bi-GRU layer (one forward and one backward), and each GRU layer had an output dimension of 150 units. The maximum number of words in each query, and each document sentence was set to 15, and sentences greater than 15 words were split into multiple sentences. The maximum number of sentences in each document was set to 20. All the attention layers had a dimension of 300. We used a feed-forward network with 3 layers to compute the final score from the similarity vector.

Additionally, we used dropout of 0.2 after each layer. For each query, we had one positive document, 3 partially relevant negative documents, and 6 non-relevant negative documents in the experiments. For all the neural baselines, we used an open-source implementation MatchZoo[5]. The TF-IDF baseline was implemented using the `scikit-learn`[6] package.

We used GloVe [93] pre-trained word vectors with 300 dimensions. Since healthcare documents contain some medical words which cannot be found in GloVe, we used randomly initialized embeddings for such out-of-vocabulary words. We experimented with training GloVe word vectors on our document corpus, but it was found that the original pre-trained GloVe outperformed our medical word embeddings. We hypothesize that it was due to the fact that our corpus contained far less documents than those used in the original GloVe. Also, it is observed that people usually do not use very complex medical words in their queries, and hence, most of the queries were composed of words that were present in the pre-trained GloVe.

3.4.4 Results

Quantitative comparison against state-of-the-art models.

Table 3.2 shows the performance of HAR against the baseline retrieval models on the HealthQA dataset. For computing Recall@K, we set K as 1, 3, and 5. As we can see, the results for HAR are consistently better than all other baselines, across all the metrics, which can be attributed to its strong performance.

- **Effect of long documents on model performance:** We can observe from Table 3.2 that TF-IDF has very low performance on our dataset. With the exception of ARC-II,

[5]https://github.com/NTMC-Community/MatchZoo
[6]http://scikit-learn.org/stable/index.html

Table 3.2: Comparison of HAR model with other baseline models on HealthQA dataset.

Model name	MRR	Recall@1	Recall@3	Recall@5
TF-IDF	60.60	36.58	81.74	96.91
CDSSM	64.46	44.27	81.34	93.42
ARC-II	50.37	29.85	61.86	78.54
MV-LSTM	75.15	58.88	90.26	97.39
DRMM	74.08	57.08	90.08	99.01
KNRM	70.97	54.91	84.04	94.14
aNMM	74.72	58.25	90.35	98.29
Duet	69.66	53.29	84.31	93.87
MatchPyramid	81.82	69.43	93.69	98.92
HAR	**87.88**	**78.90**	**96.84**	**99.64**

the performance of TF-IDF is consistently lower than all the other models. This can be attributed to the non-factoid nature of our dataset where there is minimal overlap between the question and answer words, as well as the long length of documents in the corpus. Hence, a keyword-based method cannot perform well on such dataset, since its performance largely depends on the word overlap between the query and the document. In such cases, embedding-based methods yield better performance in general, as they can correlate queries and documents based on their semantic representation.

- **Document-representation based semantic similarity methods:** One key observation from our results is that neural models that match the query with documents solely based on their document-level representations do not perform well on the HealthQA dataset. CDSSM computes the representation of query and document separately, and then computes the similarity between their vector representations. Although using semantic representations can help in dealing with the problems that arise where the queries do not have matching words with the documents, this concept only works in problems such as sentence or paraphrase matching, where the lengths of both the query and the documents being matched are similar. In case of retrieval, queries are typically much shorter compared to the documents. Moreover, the actual part in

the document that is relevant to the query is only a few words or sentences. Hence, the vector representation of the document contains features from other parts of the documents that are irrelevant to the query. This leads to the poor performance of such models for retrieval tasks.

- **Effect of word interactions:** With the exception of ARC-II, other baselines such as MV-LSTM, DRMM, KNRM, aNMM, Duet, and MatchPyramid use some form of embedding-based pair-wise keyword interaction in the feature representation of the query-document pair. This results in their better performance compared to other baselines. Using word-level interaction features based on their vector representation allows the models to deal with the problems faced by traditional methods such as TF-IDF. However, by computing interaction in the early stages of the model, these models do not have any mechanism to incorporate the underlying structure of the query and document in the interaction feature generation or scoring process. This leads to their poor performance in a setting where the documents are longer in length.

By using a cross attention mechanism between the query and document, HAR is able to model the interaction features between the query and document words, while retaining the overall semantic meaning of the document sentences. The self attention mechanism then facilitates focussing on sentences and words in the document that are most relevant to the query. This mechanism helps HAR to achieve the highest performance compared to all other baseline methods. Most importantly, HAR achieves a considerably higher MRR and Recall@1 compared to all other methods. This implies that, in most of the cases, our model is able to rank the correct document with the highest score, demonstrating a higher reliability of HAR.

In Figure 3.5, we show the performance of HAR and other baseline methods on each of the question types given in Figure 3.4. Although questions of the type *"what"* are relatively

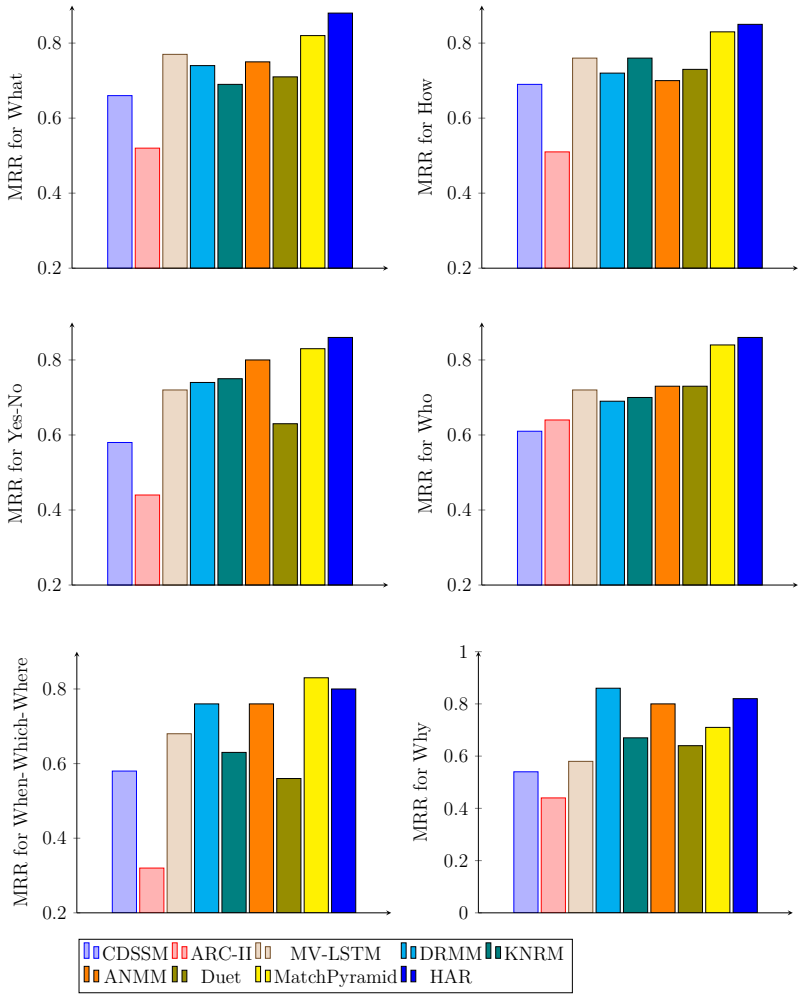

Figure 3.5: Performance of HAR and baseline methods on different types of questions.

easier to answer (as they contain many factoid questions), our model outperforms other baselines on these questions. We can also see that HAR gives high performance on question types *"how"* and *"why"*, which are non-factoid in nature, and difficult for a retrieval system. The performance of HAR is consistently higher than the baselines across all other question categories as well.

Qualitative results

For qualitative evaluation of the performance of HAR, we show an example of a question, and the retrieved document, obtained by MatchPyramid and HAR, in Figure 3.6. We compare with MatchPyramid, since it is the strongest baseline among all the other methods. The question shown here is about *"the effect of wisdom tooth removal on brushing"*. Although the document returned by MatchPyramid is relevant to the topic, which is *wisdom teeth removal*, it is not the one that can correctly answer the query. MatchPyramid computes interaction features at an early stage in the model. It does not retain the original query and document, and computes scores solely based on the interaction features. Due to this, the main intent of the question can sometimes be lost, as shown in the example here. By using a powerful attention mechanism, HAR has the ability to discriminate between two similar documents, based on the intent of the query. Hence, HAR is able to retrieve the correct document for the question.

Will removal of wisdom teeth affect my brushing ?	
MatchPyramid answer (incorrect): Most people need two or three days off work to recover from the effects of the anaesthetic after they have had wisdom teeth removed. Those with a more physical job may need an extra day or two. If your job involves driving you should be fine to return after 24 - 48 hours after having had anaesthetic, but you need to use your own judgement and seek advice from your dentist if you are unsure.	**HAR answer (correct):** You may find it uncomfortable to brush shortly after having wisdom teeth removed, especially around any wounds. For this reason, your dentist might recommend that you use a mouthwash to help keep the area clean.

Figure 3.6: An example of a question and its answer retrieved by MatchPyramid (left) and HAR (right).

Using HAR for answer extraction

As mentioned earlier, an added advantage of using the hierarchical attention mechanism is that it allows the model to discover the most probable answer snippet from the long document. This can be done by comparing the attention weights of different sentences in level-1 of hierarchical inner attention over documents. Since sentences with high attention weights have more contribution towards generating the document representation, they are likely to be more relevant to the query, as compared to sentences with low attention weights.

In Figure 3.7, we illustrate how the attention weights can be used to extract the most probable answer from the document. We show one question, and its corresponding highest-ranked document. The self attention weights over the query words are highlighted in blue, while the self attention weights at sentence level (level-1) of the document hierarchical attention are shown in red.

The question shown in Figure 3.7 is about *Alopecia Areata*, which is a condition that leads to hair loss in many people, mainly because the immune system of a person starts attacking the hair follicles. The question asks about the diagnosis procedure for this condition, demonstrated by the use of the word *test*. Since diagnosis is the main intent of this question, the feature representation uses highest attention weight for the word *test*, followed by other words that are useful in the question. The document here has the highest attention weight for the first sentence, that contains the answer to this question. Other highlighted sentences in the document are those which can provide additional information supporting the answer for this question.

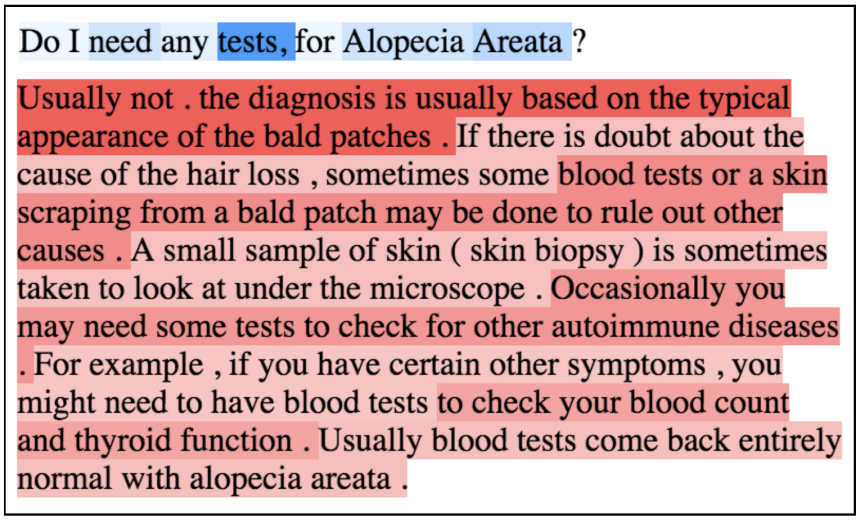

Figure 3.7: An example of a question and its answer document with highlightings based on the attention weights. The attention weights for the question are obtained from the query self attention, and those for the document are obtained from level-2 self attention.

3.4.5 Performance Analysis

Effect of attention mechanism

To quantitatively evaluate the effect of various components used in our HAR on the model performance, we compare the performance of HAR against its two variants. These are described below:

- **HAR without cross attention:** To evaluate the effect of using the cross attention mechanism on model performance, we evaluate the performance of a variant of HAR that does not use cross attention between query and document words. This model uses an inner attention over query words, and a hierarchical inner attention over document words and sentences. By removing the cross attention, the model is not able to use the interaction features between query and document word vectors in the scoring process. We refer to this model as *HAR-WCA*.

- **HAR without cross and hierarchical attention:** This is an even simpler version of HAR that neither uses cross attention between query and document words, nor the hierarchical inner attention in the document. This model uses a similar encoder as HAR for query and document, and then uses only one level of inner attention to get the query and document representations. It then computes the scores between these vectors similar to the scoring process used in HAR. It does not incorporate the underlying document structure in the scoring process due to the removal of hierarchical attention. We refer to this model as *HAR-simple*.

Model name	MRR	Recall@1	Recall@3	Recall@5
HAR-simple	82.139	69.883	94.770	99.008
HAR-WCA	83.667	72.047	95.942	99.369
HAR	**87.877**	**78.900**	**96.844**	**99.639**

Table 3.3: Performance comparison of HAR and its variants.

Table 3.3 shows the performance comparison of HAR with two of its variants. The performance of both *HAR-WCA* and *HAR-simple* is worse than the full model. We believe that since *HAR-simple* uses a single long encoder for documents, it is not able to embed the contextual dependencies in the encoded embeddings. Also, it does not use cross attention, thereby ignoring the keyword-interaction features. The performance of *HAR-WCA* is slightly better than that of *HAR-simple*. Since each sentence sequence is much smaller than the full document, the encoded representation is able to embed the context of the sentence in each word.

Hyperparameter sensitivity and model convergence

As mentioned earlier, by splitting the documents into short sentences and using the hierarchical attention mechanism, HAR does not need to deal with long sequences. This allows the model to be achieve high performance using computationally efficient GRU, which can

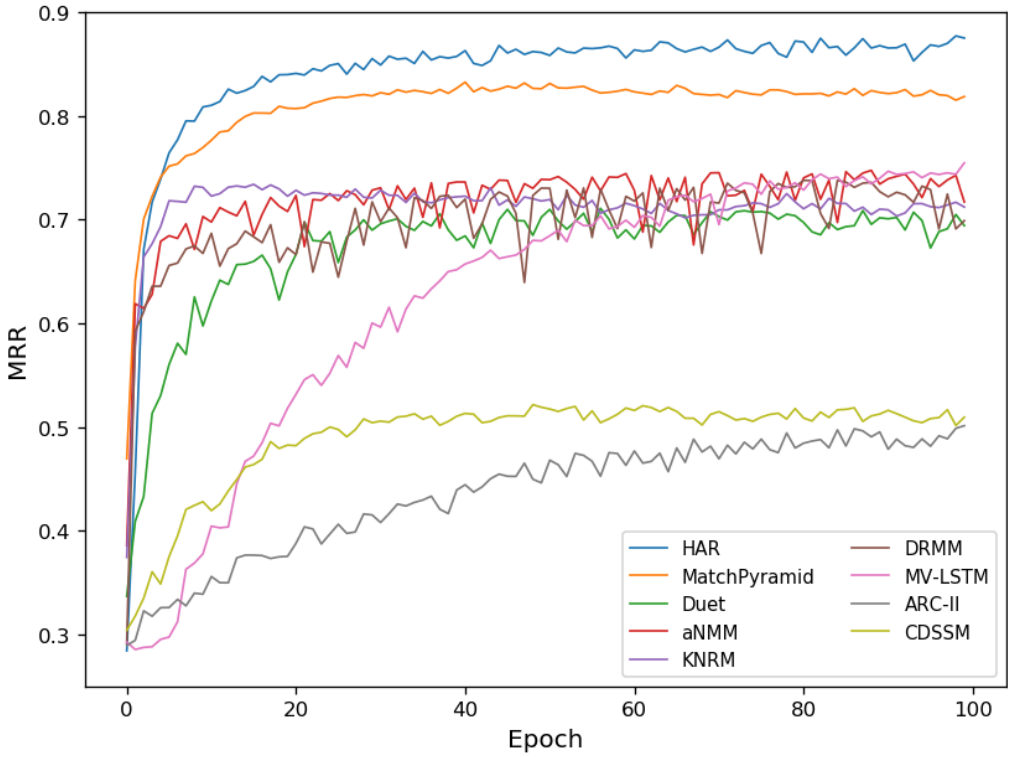

Figure 3.8: MRR convergence with training epochs.

effectively model short sequences. We also find that by using GRU, the model is able to converge quickly, as compared to a model that uses LSTM. We also evaluated the performance of HAR by varying different parameters of our model, such as the number of GRU hidden units and the number of feed-forward layers. We find that these parameters only have a marginal impact (∼1% MAP reduction) on the performance of our model. In Figure 3.8, we show the convergence of HAR and other baselines over training epochs. We show the MRR of different models on test queries, as the number of training epochs increase. We can see that HAR converges faster as compared to other baselines, demonstrating a better learning ability of our model.

3.5 Conclusion

In this chapter, we proposed a novel deep neural network architecture to rank documents for healthcare related queries. The model uses a combination of powerful attention mechanisms to develop a robust retrieval system. The attention mechanisms also enables the model to discover highly probable answer snippets from the documents, without the need for using a computationally expensive machine comprehension module. The model has been carefully designed by considering the special characteristics of question-answering in healthcare domain, such as the open-ended nature of queries, and longer document length.

To evaluate the proposed HAR model, we constructed a novel consumer-oriented healthcare question answering dataset, HealthQA. This dataset is comprised of questions regarding consumer healthcare topics. We evaluated our proposed model on this dataset, against several state-of-the-art baseline techniques. Our experimental results show that our model outperforms these techniques by a wide margin. We also show how our model can potentially be used to extract the most probable answer snippets from the highly-ranked documents, which can be explored further in future work. We hope that our proposed model will be useful for both healthcare and information retrieval communities, to make healthcare information more accessible to the people.

Chapter 4

Question Answering with Long Multiple-Span Answers

Answering questions in many real-world applications often requires complex and precise information excerpted from texts spanned across a long document. However, currently no such annotated dataset is publicly available, which hinders the development of neural question-answering (QA) systems. To this end, we present *MASH-QA*[1], a **M**ultiple **A**nswer **S**pans **H**ealthcare **Q**uestion **A**nswering dataset from the consumer health domain, where answers may need to be excerpted from multiple, nonconsecutive parts of text spanned across a long document. We also propose *MultiCo*, a neural architecture that is able to capture the relevance among multiple answer spans, by using a query-based contextualized sentence selection approach, for forming the answer to the given question. We also demonstrate that conventional QA models are not suitable for this type of task and perform poorly in this setting. Extensive experiments are conducted, and the experimental results confirm the proposed model significantly outperforms the state-of-the-art QA models in this multi-span QA setting. This chapter is adapted from a paper [149] published in the Findings of the Association for Computational Linguistics: EMNLP 2020, of which I am the first author and primary contributor.

[1]Code: https://github.com/mingzhu0527/MASHQA

What are tips for managing my bipolar disorder?
Along with seeing your doctor and therapist and taking your medicines, simple daily habits can make a difference. Start with these strategies. *(22 words truncated)* Pay attention to your sleep. This is especially important for people with bipolar disorder... *(178 words truncated)* Eat well. There's no specific diet... *(29 words truncated)* Focus on the basics: Favor fruits, vegetables, lean protein, and whole grains. And cut down on fat, salt, and sugar. Tame stress. *(81 words truncated)* You can also listen to music or spend time with positive people who are good company. *(73 words truncated)* Limit caffeine. It can keep you up at night and possibly affect your mood. *(47 words truncated)* Avoid alcohol and drugs. They can affect how your medications work. *(118 words truncated)*

Figure 4.1: An example of a question and its corresponding answer (highlighted) from MASH-QA. The answer consists of multiple sentences from the context. All the highlighted sentences will form the comprehensive answer. The context here is 632 words long, so we truncate a few portions of it.

4.1 Introduction

Developing neural networks for question answering (QA) has become an important and fast-growing area of research in the NLP community. Interest in this area is largely driven by the importance and effectiveness of such systems in virtual assistants and search engines. Driven by the development of large-scale datasets such as SQuAD [102, 103], most of the work in this domain focuses on the task of machine reading comprehension, where the objective is to find a single short answer span—typically ranging from a few words to one sentence in length—given a question and a paragraph context [120, 134]. Natural Questions [62] makes machine reading comprehension more challenging by providing questions with long contexts. This makes it more suitable for training a typical QA system, which extracts answers from long documents returned by a search engine.

Existing QA datasets mainly consist of questions with short answers—typically ranging from a few words to a sentence—from the context document. Even though the Natural Questions dataset [62] provided paragraph-length answers for certain questions, these long answers are generally the paragraphs that contain the short answers, making most of the information supplemental (not critical) in nature. Moreover, because of the open-ended nature of many questions, the final comprehensive, succinct and correct answers may need to be extracted from multiple spans or sentences from the document. This problem is exacerbated when several spans that contain the answer are not in the vicinity of each other. Especially, this is often the case in domains such as healthcare, where people seek information regarding their specific health conditions, and the precise answer for their queries usually comes from multiple sections or spans of a document.

In this work, we introduce *MASH-QA*, a large-scale dataset for question-answering, with many answers coming from multiple spans within a long document. MASH-QA is based on questions and knowledge articles from the consumer health domain, where the questions are generally non-factoid in nature and cannot be answered using just a few words. Fig. 4.1 shows an example question, and its corresponding context and answer from our dataset, which poses several unique challenges. First, the contexts are comprehensive healthcare articles, which can typically contain tens of paragraphs and hundreds of lines. Context of such length is challenging for existing neural QA models. Second, the answers are typically several sentences long, while current span extraction models usually predict very short spans. Another challenge in this setting is raised from the fact that answers can consist of multiple sentences from nonconsecutive parts of a document, which can often be many sentences or even paragraphs apart. This results in sparsely-scattered patterns of semantic relevance in the context with respect to the query. This means that even if the answer comes from different parts of the document, which might be surrounded by texts that have limited

relevance to the question, different answer snippets have some form of semantic relevance with each other, and are centered around the same topic as the question. Although our dataset is from the healthcare domain, we believe that this problem setting can be generalized to other domains, where the questions typically require long and detailed answers.

Previous research in question answering (QA) leveraged similarity models to select answers according to their semantic resemblance to the text [79, 145]. With the release of the SQuAD dataset, attention-driven neural models [15, 120, 134] were developed for extractive question answering. Recent advances in language modeling strategies, notably BERT [23] and XLNet [139], have seen language modeling-based methods rise to prominence in the current landscape. Considering the previous work and the challenges presented, we formulate our question-answering task as a sentence selection task, which should also model the semantic relevance existing between different answer sentences, even when they are not adjacent to each other in the context. Hence, we also propose *MultiCo*, a novel neural architecture that can address the challenges discussed above. Our model utilizes XLNet [139], which incorporates Transformer-XL units [21] to give semantic representations that capture the long-range dependencies existing in the long document context. We also use a sparsified attention mechanism, to ensure that the representations of sparsely scattered answer units are compactly aligned with each other. The main contributions of this chapter can be summarized as follows:

- We present a practical and challenging QA task, where the answers can consist of sentences from multiple spans of the long context. We introduce a new dataset called *MASH-QA* from the consumer health domain, that encompasses the challenges encountered in this task.
- We propose *MultiCo*, a novel neural model that deals with the long context problem, and is able to identify the sentences spanned across the document for forming the

answer. MultiCo adapts a query-based contextualized sentence selection approach, combined with a sparse self-attention mechanism.

- Extensive experiments are conducted to evaluate the proposed model on multiple datasets including *MASH-QA* and WikiQA. Our experimental results confirm that our approach outperforms state-of-the-art machine reading comprehension and semantic matching models.

To the best of our knowledge, this is the first work that introduces the QA setting with multiple discontinuous answer spans from a long document.

4.2 MASH-QA Dataset

4.2.1 Dataset Description

Since we focus on the task of multi-span question-answering from long documents, our dataset consists of *(question, context, [answer sentences])* tuples. Each tuple consists of a natural language question, which can be answered using one or more sentences from the context. Context here is a long document, a typical web article with multiple paragraphs. Each answer consists of several sentences, which can either belong to one single span, or multiple spans from the context document. Since questions in our dataset can have multiple sentences that form the answer, we provide the index of all correct answer sentences with each tuple. We refer to the single-span answer subset of our dataset as MASH-QA-S, and the multi-span answer subset as MASH-QA-M. Some of the basic statistics of our dataset are shown in Table 4.1.

	MASH-QA	MASH-QA-S	MASH-QA-M
# Contexts	5,577	4,674	4,788
# QA pairs	34,927	15,293	19,634
# Train QA	28,649	12,437	16,212
# Dev QA	3,081	1,405	1,676
# Test QA	3,197	1,451	1,746

Table 4.1: Basic statistics of MASH-QA dataset.

	Dataset	#QA	Context Source	QA Type	Answer Span	Context Length	Answer Length
Generic	WikiQA	3K	Wikipedia	Extractive	Single	238.4	11
	SQuAD-1.1	108K	Wikipedia	Extractive	Single	117.2	3.1
	Natural Questions	307K	Wikipedia	Extractive	Single	7320.3	85.2 (long)
	ELI5	270K	Web Search	Abstractive	Multiple	857.6	130.6
Healthcare	CLiCR	105K	Clinical Reports	Abstractive	Single	1385.4	2.7
	emrQA	400K	Medical Records	Extractive	Single	955.4	10.2
	MedQUAD	47K	Health articles	Ranking	Single	N/A	123.9
	HealthQA	8K	Health articles	Ranking	Single	N/A	233.4
	MASH-QA	35K	Health articles	Extractive	Multiple	696.1	59.6

Table 4.2: Comparison of MASH-QA dataset with other Question Answering datasets.

4.2.2 Data Collection and Processing

Our dataset consists of consumer healthcare queries sourced from the popular health website WebMD[2]. The website contains articles from a diverse set of domains related to consumer healthcare. Each healthcare section on the website also consists of questions related to common healthcare problems faced by people. The answers to these queries consist of sentences or paragraphs from the article associated with the relevant healthcare condition. These answers have been curated by healthcare experts, and can accurately answer the corresponding query. Because of the nature of the domain, correctness of the answer is especially important, as in domains such as healthcare, an incorrect answer to a consumer can have dire consequences.

[2] https://www.webmd.com/

Starts With	Percentage	Example
What	46.09	What are the symptoms of gastritis? What are tips for treating acne?
How	31.03	How can I prevent blisters? How does exercise help stress?
Can, Is, Are Do, Does	11.01	Can I prevent sinusitis? Is scalp psoriasis common?
When	3.65	When do I need eye protection? When is flu season in the U.S.?
Why	2.05	Why do we have tears? Why do I need dental exams?

Table 4.3: Common question types and their examples from the MASH-QA dataset.

For each question, we first split the answer into sentences. We also split each of the context documents into the constituent sentences. Next, for every answer, we map each of its sentences to the corresponding sentence from the context. We notice that some of the answer sentences have been manually edited by the healthcare experts who answered the question. In such cases, we select a set of candidate sentences from the context that are similar to the answer sentence using TF-IDF match, and then manually select the sentence that corresponds to the answer.

4.2.3 Dataset Characteristics

A comparison of our dataset with other QA datasets from general and healthcare domains is shown in Table 4.2. Table 4.3 shows some of the common question types from our dataset. We discuss some of the key observations below.

Answers with Multiple Spans A key characteristic of our dataset is that, for many questions, the answers are obtained using information from multiple, discontinuous spans from the document, making the task more challenging in nature. The existing multi-document or multi-span QA datasets are abstractive in nature, and the support documents were curated

using automatic techniques, such as web search. Because of this nature, the answer is not guaranteed to be found in the context, and the documents are often noisy, with limited relevance to the question. In contrast, our dataset contains multi-span answers that are curated by experts, which ensures that the different answer spans have information that is required to answer the question. Moreover, for a domain such as healthcare, we believe the extractive setting is ideal, since abstractive answers can introduce unpredicted errors resulting from answer generation.

Comprehensive and Compact Answers The answers in our dataset are generally comprehensive, and all the sentences in an answer contribute information that is important to answer the question. In existing datasets with long answers, a majority of the information in the long answer is supplemental in nature. Natural Questions, for example, provides a short answer for the question, and a long answer that was created by selecting the entire paragraph containing the short answer. The answers in our dataset, on the other hand, have multiple sentences, each of which contains a unique piece of information about the subject in the query. We believe that comprehensiveness and compactness of answers are vital in the healthcare domain, since answers with missing information can potentially mislead people, while answers with extra information can be overwhelming.

Question Types A majority of the questions in our dataset are non-factoid and open-ended in nature, and seek for detailed information about the health condition. A significant proportion of the questions are "How" type, and such questions generally tend to be open-ended. Although questions starting with "What" generally ask for specific facts, we find that many of these questions, such as the ones shown in Table 4.3, are in fact open-ended, and require long answers. Our dataset also contains many "Yes/No" type questions, which often require explanations.

4.3 The Proposed MultiCo Model

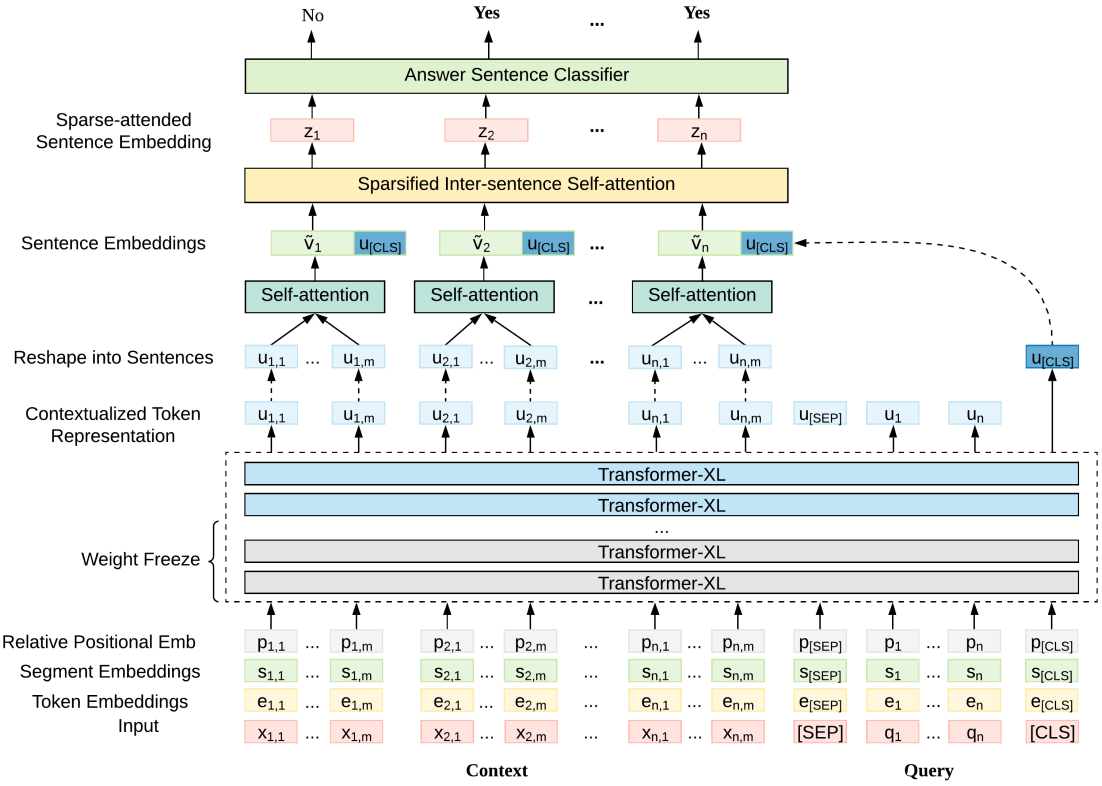

Figure 4.2: Architecture of the proposed MultiCo model.

Given a query and a document, the goal of our MultiCo model is to select the sentences that can accurately answer the query. An intuitive way to solve this problem would be to use a text matching model that takes the query and a sentence as the input, and predicts their relevance. However, as shown later, this approach does not capture the overall context of the sentences. Therefore, in our problem setting, where multiple sentences from a document can belong to the answer, it gives poor results. Hence, our proposed approach uses the concept of *query-based contextualized sentence selection* from the document.

4.3.1 Problem Formulation

Given a query Q and a context document $D = \{s_1, .., s_n\}$, where s_i refers to the i^{th} sentence in the document, the objective of our model is to classify each sentence s_i as relevant or not for the given query, conditioned on other sentences present in the document. Let $c_i \in \{0, 1\}$ be the relevance label that depicts whether sentence s_i belongs to the answer or not. Mathematically, we want to model the probability $P(s_i = c_i|Q, D)$ for $i \in \{1, .., n\}$.

4.3.2 Model Architecture

Figure 4.2 shows the architecture of our proposed model. The main components of our model are described in detail below:

Query and Context Encoder To encode the query and the long document context, we use XLNet [139] as the encoder. One of the main advantages of XLNet is that it is based on the Transformer-XL framework [21], which is specifically designed to deal with long documents. This makes it an ideal choice in our setting, as it can effectively encode the long context. Moreover, using a large pre-trained language model also allows us to obtain high quality token representations.

In our model, we first tokenize the query and each context sentence, and then pad each sentence up to a pre-defined maximum sentence length m. Let $\{X_1, .., X_n\}$ represent the sentences, where $X_i = \{x_{ij}\}_{j=1}^{m}$ represents the tokens in sentence s_i, and let $Q = \{q_j\}_{j=1}^{m}$ represent the query. Following [139], we concatenate a [CLS] token to the query, and a [SEP] token at the end of the last sentence. The encoded representations can be obtained by the equation below:

$$\boldsymbol{U}_1; ..; \boldsymbol{U}_n, \boldsymbol{u}_{[SEP]}, \boldsymbol{U}_q, \boldsymbol{u}_{[CLS]} = \text{XLNet}(X_1; ..; X_n, [SEP], Q, [CLS]) \tag{4.1}$$

Sentence Embeddings To obtain a fixed dimensional vector for each sentence s_i, we use self-attention [69] over the encoded representations $\boldsymbol{U}_i$ obtained in the previous step, to get the intermediate sentence embedding $\tilde{\boldsymbol{v}}_i$.

$$\begin{aligned} \boldsymbol{h}_{ij} &= \mathbf{w}_a \tanh(\mathbf{W}_a \boldsymbol{u}_{ij}) \\ \alpha_{ij} = \underset{j}{\text{softmax}}(\boldsymbol{h}_{ij}) \qquad \tilde{\boldsymbol{v}}_i &= \sum_{j=1}^{m} \alpha_{ij} \boldsymbol{u}_{ij} \end{aligned} \tag{4.2}$$

Here, α represents attention weights. Next, to add the overall context and query representations to the sentence representation, we concatenate the embedding of the [CLS] token returned by XLNet, to get the final sentence vector $\boldsymbol{v}_i = [\tilde{\boldsymbol{v}}_i; \boldsymbol{u}_{[CLS]}]$.

Sparsified Inter-Sentence Attention The multi-span nature of answers in our dataset requires us to have a mechanism to link the different answer sentences with each other. Moreover, the number of relevant sentences in the context is much less than the total number of sentences in the context. Hence, we use a sparsified inter-sentence attention layer based on α-entmax ($\alpha = 1.5$) [19, 94] to introduce sparsity.

$$\begin{aligned} \boldsymbol{g}_{ij} &= \mathbf{w}_b \tanh(\mathbf{W}_b[\boldsymbol{v}_i; \boldsymbol{v}_j]), \\ \beta_{ij} = \underset{j}{\alpha\text{-entmax}}(\boldsymbol{g}_{ij}) \qquad \boldsymbol{z}_i &= \sum_{j=1}^{n} \beta_{i,j} \boldsymbol{v}_j \end{aligned} \tag{4.3}$$

β_{ij} here represents attention weights of sentence i with respect to sentence j. For any given sentence, α-entmax above gives sparse attention weights over other sentences in the context. This makes the final representation only conditional on a small number of other sentences with similar semantic nature, and zeroes out the effect of other sentences, unlike the standard softmax. For any given vector $\boldsymbol{g}$, it can be calculated as follows.

$$\alpha\text{-entmax}(\boldsymbol{g}) = \text{ReLU}[(\alpha - 1)\boldsymbol{g} - \tau\mathbf{1}]^{1/\alpha - 1} \tag{4.4}$$

Here, τ is the threshold, which can be computed as per Peters et al. [94]. As we can see, the function will give a zero probability for all values of $g \leq 1/(\alpha-1)$, hence resulting in a sparse probability distribution.

Answer Classifier After computing the representation of each sentence with respect to the query and the overall context, we pass the sentence vector $\boldsymbol{z_i}$ through a multi-layer dense network, followed by softmax, to get the final answer probability distribution $\hat{\boldsymbol{y}}_i$.

$$\hat{\boldsymbol{y}}_i = \text{softmax}(\mathbf{W}_{out}\boldsymbol{z}_i + \mathbf{b}_{out}) \tag{4.5}$$

4.3.3 Optimization

Since we model the question-answering task as a sentence classification task, we use binary cross entropy as the loss function to train our model. Let $\boldsymbol{y}_i$ be the true binary labels for sentence s_i. The loss for each sentence can be computed as follows:

$$\mathcal{L} = -\sum_{j \in \{0,1\}} y_{ij} \log(\hat{y}_{ij}) \tag{4.6}$$

4.4 Experiments

4.4.1 Implementation Details

We implemented our model in TensorFlow [2]. The model was trained using the Adam optimizer [58], with a learning rate of 2×10^{-5}. The maximum length for query and context sentences was set to 32 tokens, and the maximum number of sentences in one segment was set to 13. For longer contexts, we split them into multiple segments of 13 sentences each, and append the query to each segment. We used a pre-trained version of XLNet (24 layers, 340M parameters), and allow only the top 12 layers to be trainable, as previous research [51] suggests that the semantic features are learned mainly by the top layers. All the experiments were run on servers with single Tesla K80 GPU on each.

4.4.2 Performance against Answer Sentence Classification Based Methods

In our first set of experiments, we would like to observe the performance of our model (which computes the probability of a sentence being part of the answer conditioning on both the query and the full context) comparing to pairwise models (which only use the query and the sentence under consideration) that classify the query-sentence as relevant or not using semantic matching. As suggested earlier, this is an intuitive way to solve the sentence classification task. Hence, for this task, we compare the performance of our model against other semantic matching baselines, that predict the relevance label for each sentence individually, given the *(query, sentence)* pair as the input.

Baselines and Evaluation Metrics We compare our model against various semantic matching models for this task. The semantic matching models which are used for our experiments were based on **BERT** [23], **RoBERTa** [71], and **XLNet** [139]. For all these models, we use the standard 24-layer pre-trained versions of their LARGE models, and fine-tune them to do semantic matching on *(query, sentence)* pairs. We also use **TANDA** [33], which utilizes a BERT-based architecture to answer questions using a pairwise (query, sentence) classification approach, as a baseline model.

We evaluate all the models on two levels: Sentence-level evaluation computes the **P**recision, **R**ecall, and **F1**-score based on the predicted label (relevant or not) of each sentence. This set of metrics will reward a model, even if the answer is partially correct. We also evaluate Answer-level Exact Match (EM), which computes the percentage of answers, whose predicted label matches the true label, for all the sentences in the answer. This will help us evaluate if the model can get the entire answer correct.

	Sentence			Answer
Model name	**Precision**	**Recall**	**F1**	**Exact Match**
TANDA	56.48	16.42	25.44	8.95
BERT	56.18	16.25	25.21	8.89
RoBERTa	57.70	19.06	28.65	9.40
XLNet	56.05	19.73	29.19	9.09
MultiCo	**58.16**	**55.90**	**57.00**	**22.05**

Table 4.4: Comparison of MultiCo with other baseline Classification models on MASH-QA dataset.

Results on MASH-QA As we can see from the results in Table 4.4, MultiCo significantly outperforms the classification baselines on the MASH-QA dataset, on both the sentence-level and answer-level metrics. Since we model the sentence conditional on both the query and other sentences in the context, our model can take into account the semantic dependencies that exist between multiple sentences in a document, and their relationship with the query.

Other techniques only use the query and the sentence under consideration, and do not take into account the association between different answer sentences, which leads to lower performance.

Model name	Precision	Recall	F1
TANDA	**68.47**	45.00	54.31
BERT	48.10	56.32	51.89
RoBERTa	56.23	53.92	55.05
XLNet	48.54	51.19	49.83
MultiCo	56.79	**56.92**	**56.86**

Table 4.5: Comparison of MultiCo with other baseline classification models on WikiQA dataset.

Results on other QA datasets We also evaluate the performance of our proposed model on other QA datasets, to observe its generalizability to other settings. Since there are no existing datasets that contain multi-span answers, the only dataset that can resemble our problem setting is WikiQA. Here, we only calculate the sentence-level metrics, as most of the answers in WikiQA contain only one sentence. The results presented in Table 4.5 show that our model outperforms all other baselines. A paired t-test indicates that our model outperforms RoBERTa with more than 95% confidence level (experimented with 5 different random seeds). The baselines have a better performance on WikiQA as compared to MASH-QA, which can be attributed to two factors: shorter context length, and fewer sentences per answer. Because of this, the techniques used in our model to handle these factors have minimal effect. Nonetheless, our model still outperforms the baselines, which shows that our technique can be generalized to other QA settings as well.

4.4.3 Performance against Span Extraction Based Methods

In this setup, we show the comparison of our proposed model with other span extraction based methods. This setup allows us to evaluate how the sentence selection/classification approach performs in contrast to approaches that predict the start and end indices of the answer span. Since such methods are designed only to predict a single start and end index, the applicability of such approaches is only limited to cases where the answer can only have one span from the context. Hence, for this setup, we only use the subset MASH-QA-S of our dataset that contains questions with single span answers.

Baselines and Evaluation Metrics We use the following baseline techniques in this experiment task: **DrQA Reader** [15] uses an RNN-based architecture, along with context-to-query attention, to compute the answer. **BiDAF** [120] uses bidirectional attention (query-to-context and context-to-query) for answer span prediction. We also use the QA versions of **BERT**, **SpanBERT** [55], and **XLNet**, as the baselines. For the former three models, we use the standard pre-trained versions of LARGE models, and fine-tune them on our dataset.

Since our objective here is to predict the answer span for the single answer, we use F1 and Exact Match (EM) as the evaluation metrics. F1 measures the overlap between the predicted and the true answers, and EM measures the percentage of overall predicted answers that exactly match the true answer.

Results The results for the span prediction task on single-span MASH-QA are shown in Table 4.6. As we can see, MultiCo outperforms all the other baselines by a wide margin. This can be attributed to the fact that most of the QA models proposed so far in the literature are mainly focused on the extractive QA datasets with short answers, that typically range up to a few words. The answers in MASH-QA on the other hand, are longer, making the task

Model name	F1	Exact Match
DrQA Reader	18.92	1.82
BiDAF	23.19	2.42
BERT	27.93	3.95
SpanBERT	30.61	5.62
XLNet	56.46	22.78
MultiCo	**64.94**	**29.49**

Table 4.6: Comparison of MultiCo with other baseline Question Answering models on MASH-QA-S dataset.

more challenging. For long answers, where the minimum answer unit is a sentence, models trained with sentence-level objective are likely to perform better than those with word-level objectives.

4.4.4 Qualitative Results

For qualitative analysis, we analyze the effect of using sparse attention on the model performance. In Fig. 4.3, we plot the heatmap of the attention weights obtained from the sparse attention layer, for two query-context pairs from our dataset. The first example here contains an answer with four consecutive sentences. As we can see, the attention weights for these sentences are high with respect to each other, and zeroed out with respect to non-answer sentences. Similarly, non-answer sentences only attend to other non-answer sentences. A similar trend is observed in the other example, that contains four answer sentences from two non-consecutive spans.

The answers obtained from the baseline BERT model using the two QA approaches are also shown. Using the span extraction approach, BERT gives an incorrect short answer, while with the pairwise query-sentence classification approach, it only predicts one answer sentence correctly. We observe that these answers have been selected based on superficial cues. By linking semantically similar sentences, the sparse attention ultimately helps to link the query

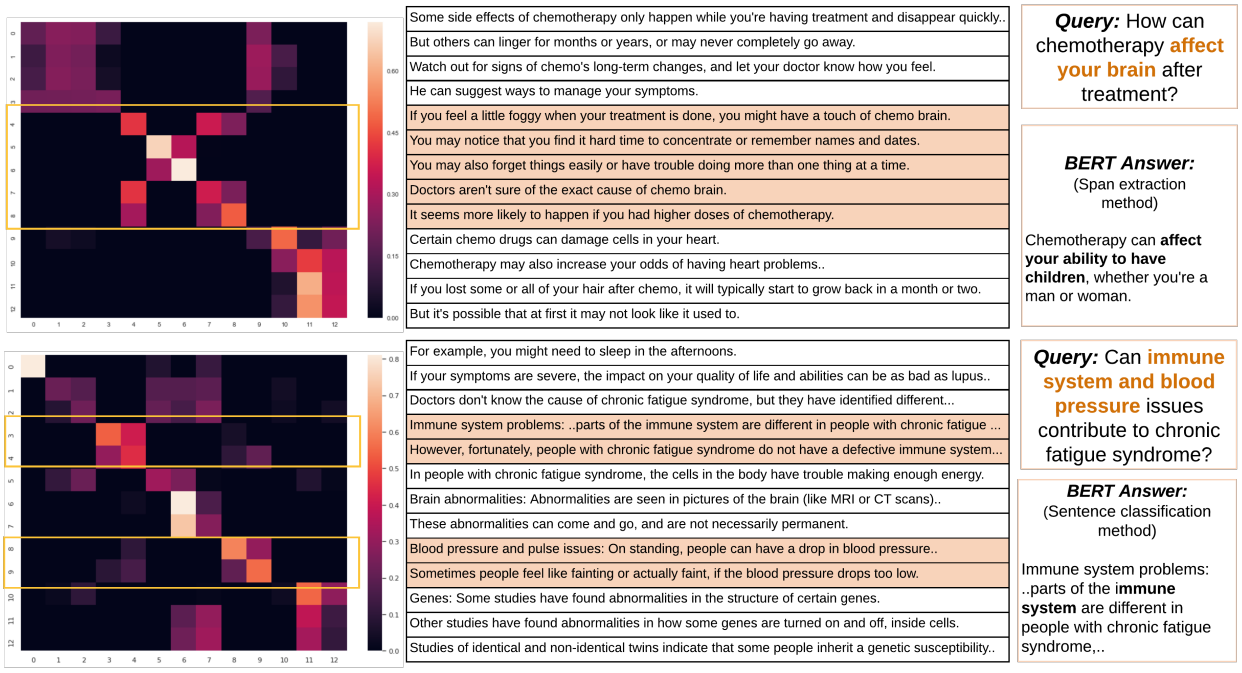

Figure 4.3: Heatmap of attention weights from the inter-sentence attention layer for two QA pairs. The matrices show the attention weights of each sentence with respect to every other sentence from the context. The high values of diagonal elements represent the weight of a sentence with respect to itself. Answers from BERT are shown on the right.

with answer sentences that have limited similarity with the query, but are similar to other answer sentences.

4.5 Conclusion

We proposed a novel form of question-answering, where answers to a question are obtained using multiple spans from a long document. To support this task, we introduce MASH-QA, a novel and challenging QA dataset from the consumer health domain. MASH-QA consists of questions that can be answered using information from multiple spans from the document. To motivate further research in multi-span QA, we also propose a novel QA architecture called MultiCo, that uses query-based contextualized sentence selection approach for finding

multi-span answers from long documents. By using a sentence-selection based objective, our model outperforms the existing state-of-the-art QA models by a wide margin.

Part II

Learning from Domain Knowledge

Chapter 5

Latent Type Modeling for Biomedical Entity Linking

Entity linking is the task of linking mentions of named entities in natural language text, to entities in a curated knowledge-base. This is of significant importance in the biomedical domain, where it could be used to semantically annotate a large volume of clinical records and biomedical literature, to standardized concepts described in an ontology such as the Unified Medical Language System (UMLS). We observe that with precise type information, entity disambiguation becomes a straightforward task. However, fine-grained type information is usually not available in the biomedical domain. Thus, we propose **LATTE**, a **LAT**ent **T**ype **E**ntity Linking model, that improves entity linking by modeling the latent fine-grained type information about mentions and entities. Unlike previous methods that perform entity linking directly between the mentions and the entities, LATTE jointly does entity disambiguation, and latent fine-grained type learning, without direct supervision. We evaluate our model on two biomedical datasets: MedMentions, a large scale public dataset annotated with UMLS concepts, and a de-identified corpus of dictated doctor's notes that has been annotated with ICD concepts. Extensive experimental evaluation shows our model achieves significant performance improvements over several state-of-the-art techniques. This chapter is adapted from a paper [150] published in the Proceedings of the AAAI Conference on Artificial Intelligence in 2020, of which I am the first author and primary contributor.

5.1 Introduction

With the advancements in the healthcare domain, we have witnessed a considerable increase in the amount of biomedical text, including electronic health records, biomedical literature and clinical trial reports [105]. To successfully utilize the wealth of knowledge contained in these records, it is critical to have automated semantic indexing techniques. *Entity linking* refers to the process of automatically linking mentions of entities in raw text, to a standardized list of entities in a knowledge-base. This process typically requires two steps. First, all the mentions of entities in the raw text are annotated using a standard Named Entity Recognition (NER) technique [64]. Next, the extracted mentions are linked to the corresponding entities in the entity disambiguation stage. Although a significant amount of work has been done in the domain of entity linking for text found on the web, where the objective is to link the mentions to standard knowledge-bases such as Freebase [12], most of the techniques cannot be directly transferred to the biomedical domain, which poses a number of new challenges to the entity linking problem.

Biomedical entity linking is the task of linking mentions in biomedical text, such as clinical notes, or biomedical literature, to medical entities in a standard ontology such as Unified Medical Language System (UMLS) [11]. In the healthcare domain, accurate entity disambiguation is crucial to the understanding of biomedical context. Many distinct biomedical concepts can have very similar mentions, and failure in disambiguation will lead to incorrect interpretation of the entire context. This will introduce huge risks in medical-related decision making. Moreover, biomedical entity linking can be useful in many other applications, which require automatic indexing of the text. For instance, it can be used by healthcare providers to automatically link the medical records of patients to different medical entities, which can then be used for downstream tasks such as diagnosis/medication decision making, population and health analytics [10], predictive modeling [52], medical information retrieval,

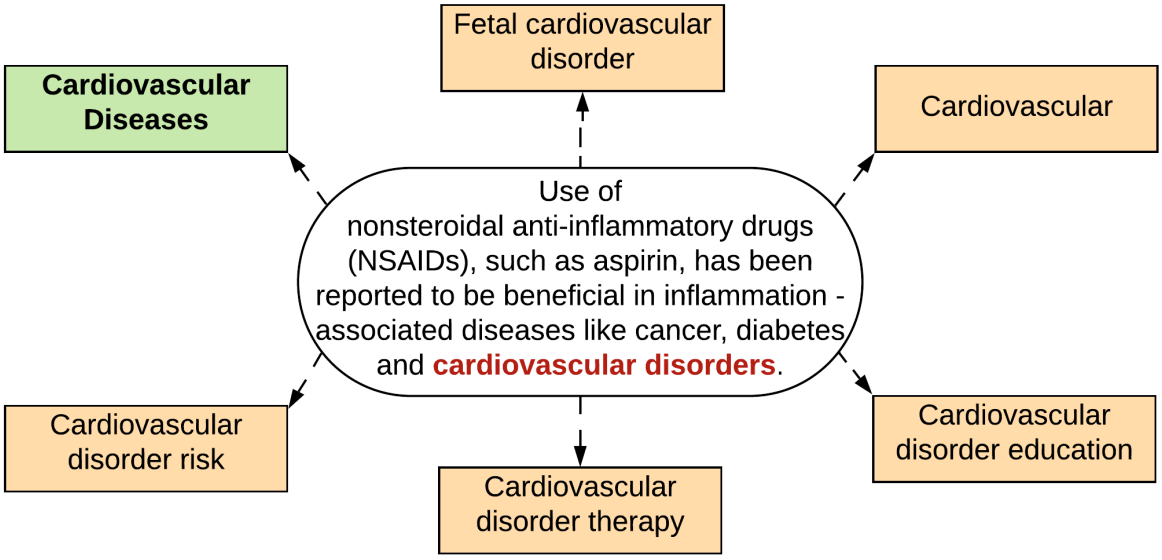

Figure 5.1: An example of biomedical entity linking. Phrase shown in red is the extracted mention, the orange boxes refer to the top candidate entities retrieved from the biomedical knowledge-base, and the green box is the ground truth entity for this mention. This example is selected from the MedMentions dataset.

information extraction [44], and question answering [141].

Entity linking on biomedical text differs from that on other general domains of text, such as web documents, in many ways. Consider the example in Figure 5.1, where *cardiovascular disorders* is a mention of the entity *Cardiovascular Diseases*, and others are the top candidate entities retrieved from UMLS.

- First, the mentions can be ambiguous. In this example, almost all the other candidates have words that exactly match those in the mention. If we only use surface level features, it will be hard to link the mention to the correct entity. This requires the model to have a good semantic understanding of the mention and its context.
- Second, the candidates can be confusingly similar, not only in surface, but also in semantic meaning. In many cases, additional information is required, such as fine-grained types, to distinguish the correct entity.

- Another challenge in medical entity linking is that the mentions and the context are usually longer in length compared to in the general domain. This makes the traditional entity linking techniques less effective on medical text.
- Finally, medical text contains many domain specific terms as well as abbreviations and typos. Thus, many terms cannot be found in standard pre-trained embeddings such as GloVe [93], and it makes neural models less effective due to a large number of out-of-vocabulary words.

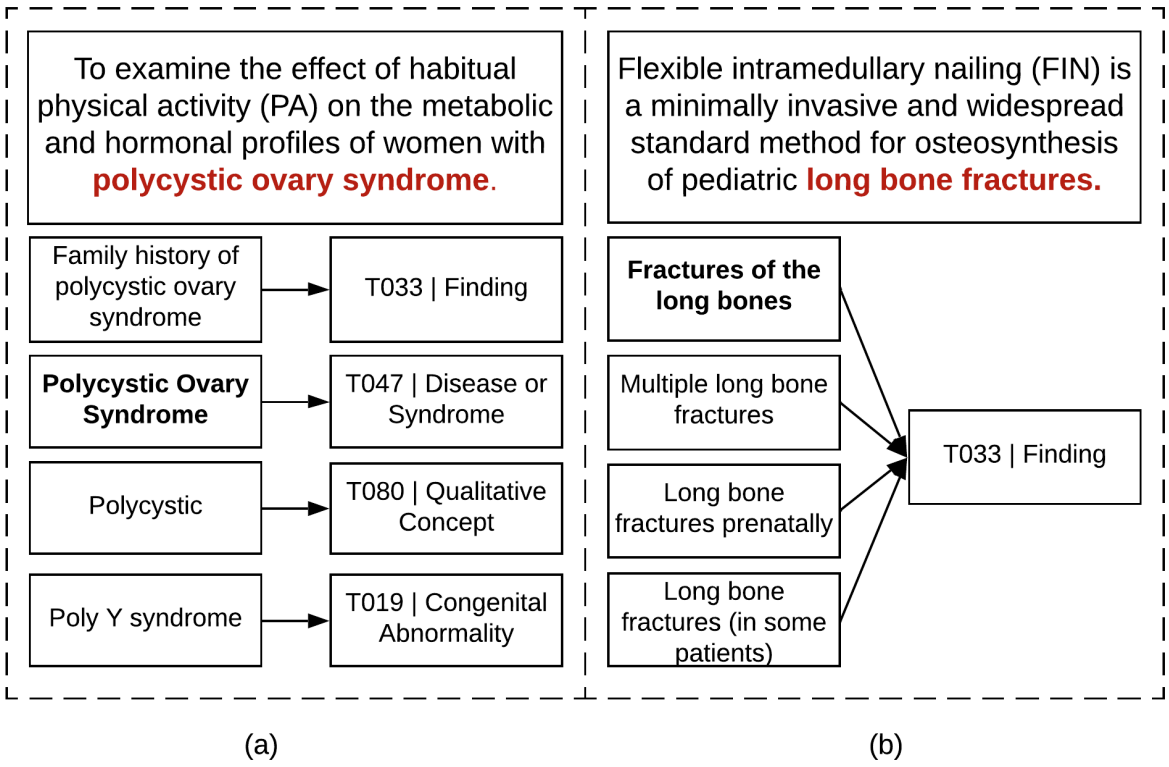

Figure 5.2: Examples of biomedical entity linking with type information.

A key observation in the process of entity linking is that if we have the fine-grained types of mentions in the raw text, and types of entities in the knowledge-base, entity disambiguation becomes much easier. For example, in Figure 5.2*(a)*, each candidate entity has a different semantic type from the UMLS Semantic Network [78]. If we can infer the correct mention

type, which in this case most likely is *Disease or Syndrome*, we can make the correct linking decision with no further effort. However, the type information in the biomedical domain is not always available, and the available ones are usually far from fine-grained.

Taking into account all these challenges, in this work, we propose **LATTE** (**Lat**ent **T**ype **E**ntity Linking model), a novel neural network based model for entity linking in the biomedical text. LATTE introduces the concept of latent-type modeling for entities and their mentions. The *latent* types refer to the implicit attributes of each entity. To guide the training process, we also use the coarse-grained *known* entity types as auxiliary supervision. To further enable our model to link the mentions with the entities from the knowledge-base, we use an attention-based mechanism, that equips the model to rank different candidate entities for a given mention, based on their semantic representations. We evaluate the performance of our model using a large scale entity linking dataset from the biomedical domain and a de-identified corpus of doctor's notes, against several state-of-the-art baselines.

The rest of this chapter is organized as follows: In Section 5.2, we describe our proposed model along with the details about the optimization and training process. In Section 5.3, we give the details about our experimental results including the evaluation metrics and baseline models. Finally, Section 5.4 concludes the chapter with possible directions for future work.

5.2 The Proposed Model

5.2.1 Motivation

We already showed in Section 5.1 that with precise entity type information (Figure 5.2*(a)*), entity disambiguation becomes a straightforward task. However, such detailed information is not usually available. For example, in the UMLS Semantic Network [78], there are only 127

types in total, while UMLS has about 900,000 unique entities [11]. In general, most known types are far from fine-grained. Furthermore, manually labeling all the entities for precise types requires a significant amount of resources and can be a daunting task. Therefore, we are motivated to model the latent fine-grained types for all the entities in the knowledge-base without direct supervision.

Latent Fine-grained Types: We argue that fine-grained types do exist. For example, the entities *Type 2 Diabetes Mellitus* and *Parkinson Disease* both have the semantic type *Disease or Syndrome*, but the former is a metabolic disorder, while the latter is a nervous system disorder. In this case, the finer-grained type can be the body system where the disease occurs. Similarly, in Figure 5.2*(b)*, for mention *long bone fractures*, all the candidates share the same semantic type *Finding*, yet they still have different intrinsic attributes which can be used to distinguish them from others. Here we see the intrinsic attributes as finer-grained types for each entity. Moreover, since there is no fixed set of fine-grained types, we do not have ground truth labels for them. This motivates us to model the fine-grained types as latent variables, and we model them using different constraints.

Binary Pairwise Relation Constraint: One constraint is the binary pairwise relation between mention and each candidate entity. Specifically, if one candidate is the ground truth entity for a given mention, the relation between them is labeled as 1; otherwise 0. We can learn the latent types from this pairwise information, as a mention and its ground truth candidate should share the same latent type distribution. Alternatively, we can see the pairwise relation label as a similarity measure between the mention and the candidates, and we can infer how similar the latent types of a mention and a candidate are from this similarity measure.

Type Hierarchy Constraint: Additionally, we can make use of the coarse-grained known types. Note that (1) known types can be of any kind and not necessarily semantic types from the UMLS Semantic Networks; (2) regardless of the number of known types, we consider them as coarse-grained, as we can always model finer-grained types. The known types are usually generic in nature, and they can be further divided into sub-types. We can view these sub-types as the previously mentioned latent fine-grained types. Thus we introduce a hierarchy in the types: the known types are the top-level nodes in the hierarchy, and the latent fine-grained types are the low-level nodes. Therefore, we can supervise on the known types to model the latent types.

Multi-tasking of Entity Linking and Type Classification: To model the latent types with both the Binary Pairwise Relation Constraint and the Type Hierarchy Constraint, we simultaneously optimize for both entity linking and type classification in our model. The entity linking module uses attention mechanism to obtain a similarity score between a mention-candidate pair, and is supervised on the pairwise relation labels. The type classification module consists of two type classifiers: one for mention, and the other for candidates. Both classifiers are supervised on the known type labels, and the weights are shared between them. The similarity of the two output latent type distributions is used as another mention-candidate similarity score. This score is combined with the previous score to obtain the final similarity score. By jointly optimizing the two tasks, we expect the entity linking performance to improve.

5.2.2 Problem Statement

Given a mention phrase (mention with context) p from a text in the biomedical domain, and a set of candidate entities $C = \{c_1, .., c_l\}$ from a knowledge-base, the model computes a relevance score $r_{p,c}$ for each entity in C, based on its relevance with the mention.

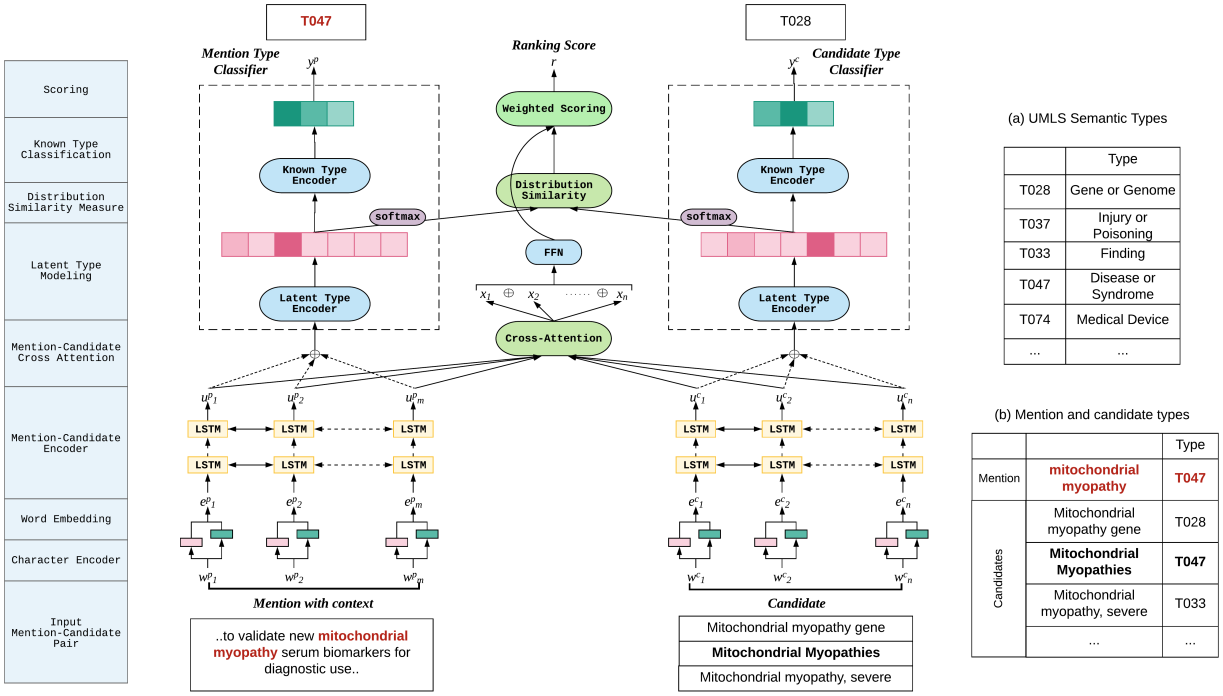

Figure 5.3: The overall architecture of the proposed LATTE model for biomedical entity linking. Table (a) shows part of the UMLS Semantic Types, which we use as the known types. Table (b) shows the type information of the mention and candidates in the given example.

5.2.3 Model Architecture

Various components of our model are described in detail below. The overall architecture of the model is illustrated in Figure 5.3. For all notations, we use the superscripts p and c for mention and candidate sequences respectively, where m and n denote their corresponding sequence lengths.

Embedding Layer: The first layer in our model is the embedding layer. This layer takes as input the word tokens $\{w_i^p\}_{i=1}^m$ and $\{w_i^c\}_{i=1}^n$ for the mention and the candidate sequences respectively, and returns the embedding vectors $\{e_i^p\}_{i=1}^m$ and $\{e_i^c\}_{i=1}^n$ for each word token. To overcome the problem of out-of-vocabulary words, we use a combination of word and character embeddings. First, the character embeddings for each character in a word are concatenated and passed through a convolutional neural network. The resultant vector is then concatenated with the word embedding, obtained from pre-trained embeddings like GloVe [93] to get the word representation.

Encoder: To get a contextual representation of the words, we use a multi-layer Bidirectional LSTM [43] encoder for both the mention and the candidate sequences. This layer takes the word representations from the embedding layer as the input, and returns the contextual representations $\{u_i^p\}_{i=1}^m$ and $\{u_i^c\}_{i=1}^n$ of words in the two sequences. The resultant vectors have the contextual information from both the backward and the forward context encoded in them.

$$\begin{aligned} \overrightarrow{u_i^p} &= \overrightarrow{LSTM}(\overrightarrow{u^p}_{i-1}, e_i^p) & \overleftarrow{u_i^p} &= \overleftarrow{LSTM}(\overleftarrow{u^p}_{i+1}, e_i^p) \\ \overrightarrow{u_i^c} &= \overrightarrow{LSTM}(\overrightarrow{u^c}_{i-1}, e_i^c) & \overleftarrow{u_i^c} &= \overleftarrow{LSTM}(\overleftarrow{u^c}_{i+1}, e_i^c) \\ u_i^p &= [\overrightarrow{u_i^p}; \overleftarrow{u_i^p}] & u_i^c &= [\overrightarrow{u_i^c}; \overleftarrow{u_i^c}] \end{aligned} \tag{5.1}$$

Cross-Attention Layer: This layer computes the interaction between the mention and the candidate vectors. It takes their encoded representations as the input, and computes the relevance between each pair of the mention and candidate word vectors, generated by the encoder layer. We use a bidirectional attention mechanism, as proposed in [120], for this layer. Specifically, we first compute the similarity matrix $S \in \mathbb{R}^{m \times n}$ between $\{u_i^p\}_{i=1}^m$ and

$\{u_i^c\}_{i=1}^n$. Each element s_{ij} of this matrix is calculated as follows:

$$s_{ij} = w_a^T \cdot [u_i^c; u_j^p; u_i^c \odot u_j^p] \tag{5.2}$$

After this, we compute the mention-to-candidate attention S^α, and candidate-to-mention attention S^β as

$$\begin{aligned} S^\alpha &= \underset{row}{\text{softmax}}(S), \\ \bar{S}^\beta = \underset{col}{\text{softmax}}(S), \text{ and } \quad S^\beta &= S^\alpha \cdot \bar{S}^{\beta^T}. \end{aligned} \tag{5.3}$$

Finally, attended vectors $\{x_j\}_{j=1}^n$ can be computed as

$$\begin{aligned} a_j^\alpha = \sum_i s_{ij}^\alpha u_i^c, \qquad a_j^\beta = \sum_i s_{ij}^\beta u_i^p, \\ x_j = [u_j^p; a_j^\alpha; u_j^p \odot a_j^\alpha; u_j^c \odot a_j^\beta]. \end{aligned} \tag{5.4}$$

All the attended vectors from the cross-attention layer are then concatenated to form $X = [x_1, .., x_n]$, and fed into a multi-layer feed-forward network, to obtain the attention-based relevance score f between the two sequences,

$$f = ReLU(w_f \cdot X + b_f). \tag{5.5}$$

Latent Type Similarity: This layer takes the output states of the encoder layer, and then concatenates them to form a fixed-dimensional vector $u^p = [u_1^p; ..; u_m^p]$ for the mention, and $u^c = [u_1^c; ..; u_n^c]$ for the candidate. The two vectors are then passed through feed-forward layers, followed by a softmax layer to obtain two probability distributions over k latent types. It then computes the similarity g between the two distributions of the mention and

the candidate using a standard distance metric like cosine similarity:

$$\begin{aligned} v^p &= w_l \cdot u^p + b_l, \quad \hat{v^p} = \text{softmax}(v^p), \\ v^c &= w_l \cdot u^c + b_l, \quad \hat{v^c} = \text{softmax}(v^c), \\ g &= \frac{\hat{v^p} \cdot \hat{v^c}}{||\hat{v^p}||\ ||\hat{v^c}||}. \end{aligned} \tag{5.6}$$

Known Type Classifier: To incorporate the known type information and to indirectly supervise the latent type modeling, we introduce the known type classifier, which is trained to predict the entity types of both the mention and candidate vectors. It takes the encoded representations v^p and v^c of the latent types, and then uses a feed-forward network with Rectifier Linear Unit (ReLU) activation, to predict their known types y^p and y^c, respectively.

$$\begin{aligned} y^p &= ReLU(w_k \cdot v^p + b_k) \\ y^c &= ReLU(w_k \cdot v^c + b_k) \end{aligned} \tag{5.7}$$

Ranking Layer: After computing the interaction score f, and the latent type similarity g, we use the ranking layer to obtain the relevance score between the mention and the candidate sequences. This module performs a weighted-combination of the two relevance scores, to compute the final relevance score r.

$$r = w_r^f \cdot f + w_r^g \cdot g \tag{5.8}$$

Here, w_r^f and w_r^g are learnable weights.

Dataset	Statistics	Train	Valid	Test
Med Mentions	#Documents	2,635	878	879
	#Mentions	210,891	71,013	70,364
	#Entities	25,640	12,586	12,402
3DNotes	#Documents	2,133	525	745
	#Mentions	22,266	5,373	8,065
	#Entities	2,026	1,030	1,209

Table 5.1: Statistics of the datasets used. Note that the "#Entities" refers to the number of unique entities.

5.2.4 Optimization

Our model incorporates two objectives, one for the type prediction, and another for candidate scoring. We jointly optimize these two objectives during our training process.

Type Classification loss: To incorporate the knowledge about the *known* categorical types into the semantic representation of mentions and the entities, we minimize the categorical cross-entropy loss. Given the known type $y \in \{y^p, y^c\}$ of a mention or a candidate, and its predicted type distribution $\hat{y}$, the loss is calculated as follows:

$$\mathcal{L}^{type} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j) \tag{5.9}$$

Mention-Candidate Ranking loss: For a given mention, we want to ensure that the correct candidate c_{pos} gets a higher score compared to the incorrect candidates c_{neg}. Hence, we use max-margin loss as the objective function for this task. Given the final scores $r_{p,c_{pos}}$ and $r_{p,c_{neg}}$ of p with respect to c_{pos} and c_{neg} respectively, the ranking loss is calculated as follows:

$$\mathcal{L}^{rank} = \max\{0, M - r_{p,c_{pos}} + r_{p,c_{neg}}\} \tag{5.10}$$

5.3 Experimental Results

5.3.1 Datasets

We use two datasets to evaluate the performance of the proposed model. MedMentions [83] contains 4392 abstracts from PubMed, with biomedical entities annotated with UMLS concepts. It also contains up to 127 semantic types for each entity from the UMLS Semantic Network [78], which we use for the known type classification. We use a de-identified corpus of dictated doctor's notes, which we refer to as 3DNotes. It is annotated with problem entities related to signs, symptoms and diseases. These entities are mapped to the 10th version of International Statistical Classification of Diseases and related health problems (ICD-10), which is part of UMLS. The annotation guidelines are similar to the i2b2 challenge guidelines for the problem entity [126]. We use the top categories in the ICD-10 hierarchy as the known types. For both datasets, we take 5 words before and after a mention as the mention context.

5.3.2 Candidate Generation

For MedMentions, we follow the approach of candidate generation described in [84]. We take only the top 9 most similar entities (excluding the ground truth entity) as the negative candidates. In addition, the ground truth entity will be considered as the positive candidate, thus forming a set of 10 candidates for each mention. For 3DNotes, we use a similar approach to generate candidates from ICD-10.

5.3.3 Evaluation Metrics

To evaluate the proposed model, we measure its performance against the baseline techniques using Precision@1 (the precision when only one entity is retrieved) and Mean Average Precision (MAP). These metrics were chosen considering the fact that our problem setup is a ranking problem. Note that, in our case, since each mention has only one correct candidate entity, Precision@1 is also equivalent to Recall@1.

5.3.4 Implementation Details

We implemented our model and all other baselines in PyTorch [92]. The model was trained using the Adam optimizer [58], with a learning rate of 10^{-4}. We used GloVe embeddings with 300 dimensions as the input word vectors, and the output dimension of the character CNN was 512, making each word a 812-dimensional vector. The encoders used two Bi-LSTM layers, where the output dimension of each individual LSTM layer was 512. The number of latent types, k, is set to 2048. The hyperparameter values were obtained based on the experimental results on the validation set.

5.3.5 Baselines

For the quantitative evaluation of the proposed LATTE model, we use the following state-of-the-art baseline methods for comparison.

- **TF-IDF:** This is a standard baseline for NLP tasks. Here, we use character level n-grams as the terms, with $n \in \{1, 2, 3, 4, 5\}$ and cosine-similarity for obtaining the candidate scores.
- **ARC-I [47]:** This is a semantic matching model that uses CNN layers to compute the

representation of the source and the candidate sequence, and then uses a feed-forward network to compute their similarity.

- **ARC-II [47]:** This is an extension of ARC-I, which instead computes the interaction feature vector between the two sequences using CNN layers.
- **MV-LSTM [129]:** It is a neural semantic matching model that uses Bi-LSTM as encoder for both mention and candidate, and then computes an interaction vector using cosine similarity or a bilinear operation.
- **MatchPyramid [90]:** This model computes pair-wise dot product between mention and candidate to get an interaction matrix. The matrix is then passed through CNN layers with dynamic pooling to compute the similarity score.
- **KNRM [135]:** This is a neural ranking model which first computes the cosine similarity between each query word and document words. It then performs kernel pooling to compute the relevance score.
- **Duet [82]:** It is a hybrid neural matching model that uses the word-level interaction and document-level similarity in a deep CNN architecture, to compute the similarity score.
- **Conv-KNRM [20]:** This model is an extension of KNRM, which instead uses convolutional layers to get n-gram representations of mention and candidates.

5.3.6 Results

Quantitative Results

Table 5.2 shows the performance of LATTE against the state-of-the-art baselines. On MedMentions, LATTE outperforms the baselines by a wide margin. On 3DNotes, paired t-tests indicate that LATTE outperforms the strongest baseline with confidence level of 90% (experimented with 5 different random seeds).

	MedMentions		3DNotes	
Model name	**P@1**	**MAP**	**P@1**	**MAP**
TF-IDF	61.39	67.74	56.89	69.45
ARC-I	71.50	81.78	84.73	90.35
ARC-II	72.56	82.36	86.12	91.38
KNRM	74.92	83.47	84.32	90.04
Duet	76.19	84.92	86.11	91.19
MatchPyramid	78.15	86.31	85.97	91.32
MV-LSTM	80.26	87.58	87.90	92.44
Conv-KNRM	83.08	89.34	86.92	92.08
LATTE-NKT	86.09	91.27	86.40	91.09
LATTE	**88.46**	**92.81**	**87.98**	**92.49**

Table 5.2: Comparison of LATTE with other baseline models on MedMentions and 3DNotes dataset. LATTE-NKT is trained without the supervision of known types classification. P@1 is short for Precision@1.

Effect of using interaction-based method: We can observe that TF-IDF and ARC-I, which compute the semantic representations of the mention and the candidate sequences independently, have lower performance as compared to all the other baselines. LATTE, as well as other models, uses some form of interaction-based semantic representation of the two sequences. The interaction mechanism can model the pairwise relevance between the words from the two sequences, and hence, can uncover the relationship between them more effectively.

Type modeling for entity linking: LATTE-NKT is a version of our model without known type modeling. In terms of architecture, LATTE-NKT do not have the known type encoders and type classifiers. We can see that all the other baselines, including LATTE-NKT, have lower performance than the full LATTE model. This shows that multi-tasking with type classification has strong positive effect on entity linking. Moreover, supervision on the known types guides the latent type modeling, which also contributes to the superior performance of LATTE.

Performance on datasets with different distributions: MedMentions is from PubMed

articles, which are more scholarly; 3DNotes is from dictated doctor's notes, which makes it colloquial in nature. The different distributions of the two datasets are also reflected in the out-of-vocabulary (OOV) words rate. Using GloVe embeddings, 3DNotes has 10.46% OOV words, while MedMentions has 58.34%. When the OOV rate is high, 1) character embeddings can help mitigate this problem as they capture the lexical similarity among the words. 2) type information provides an extra link between a mention and the corresponding entity, which is beyond lexical and semantic matching. This explains why LATTE performs better on MedMentions than 3DNotes. Since typical biomedical datasets tend to have a high OOV rate, we expect that the performance of LATTE on MedMentions can be generalized to that on other biomedical datasets.

	(a)		*(b)*		*(c)*	
Mention with Context	..IBO - positive patients became **SIBO** - negative after lubiprostone treatment..		..of sonoelastography with sonourethrography and **retrograde urethrography** in the evaluation of male..		..has been reported to ameliorate **obesity-associated metabolic disorders**. Antiadipogenic activities of γ-oryzanol	
Candidates with type	*Disease or Syndrome*	Small intestinal bacterial overgrowth **(LATTE)**	*Disease or Syndrome*	Retrograde urethrography **(LATTE)**	*Disease or Syndrome*	Obesity associated disorder **(LATTE)**
	Gene or Genome	Sibogella **(Conv-KNRM)**	*Medical Device*	Urethrographs	*Disease or Syndrome*	Metabolic disorders NEC
	Pharmacologic Substance	Sibol	*Disease or Syndrome*	Fluoroscopic retrograde cystourethrography **(Conv-KNRM)**	*Disease or Syndrome*	Hepatic metabolic disorders **(Conv-KNRM)**

Figure 5.4: Examples of entity linking result comparison between LATTE and a state-of-the-art baseline model (Conv-KNRM). Note that the red words are the mentions, and the green boxes are ground truth known types and candidates. (a) When candidates have different types, information of the correct mention type makes entity linking straightforward. LATTE learns how to classify mention types while doing entity linking. (b) When the mention or candidate words are out-of-vocabulary, measuring mention-candidate similarity becomes much harder. Character encoding and type classification mitigate this problem. (c) When candidates have the same type, LATTE is still capable of distinguishing the correct candidate from others.

Ablation Analysis

To study the effect of different components used in our model architecture, on the overall model performance, we also compare the performance of LATTE against its different variants

	MedMentions		3DNotes	
Model name	**P@1**	**MAP**	**P@1**	**MAP**
LATTE_base	80.02	86.94	84.08	90.15
LATTE_base+LT	86.09	91.27	86.40	91.09
LATTE_base+KT	87.73	92.33	87.80	**92.66**
LATTE	**88.46**	**92.81**	**87.98**	92.49

Table 5.3: Performance comparison of LATTE and its variants on MedMentions and 3DNotes Datasets.

(see Table 5.3).

- **LATTE_base:** This is the simplest variant of our model, which only contains the word embedding layer, encoder, the element-wise dot product as the similarity measure, and a feed-forward network to get a similarity score.
- **LT:** This module includes the latent type encoder, softmax and the distribution similarity measure. From this step, character embedding is included and the mention-candidate interaction is switched to cross-attention.
- **KT:** This module consists of the two known type classifiers, for mention and candidate respectively, depicted as the Known Type Classification layer in Figure 5.3. Note that during test, we do not have the known type labels.

As shown in Table 5.3, introducing latent type modeling with cross-attention boosts Precision@1 on MedMentions and 3DNotes by 6.07% and 2.32% respectively, which shows that matching mention and candidates have similar latent type distribution, and modeling this similarity improves the entity linking task. It also shows that the cross-attention mechanism is strong in capturing the semantic similarity between mention and candidates. Instead, if we add the known type supervision, there are 7.71% and 3.72% gains in Precision@1 with respect to the two datasets. This shows that multi-tasking with known type classification has strong positive effect on the entity linking task. Finally, adding latent type modeling along with know type classification further improves the Precision@1. This proves that the

hierarchical type modeling improves the entity linking task.

Qualitative Analysis

The example in Figure 5.4(a) is a common case in biomedical domain, where the mention is an abbreviated form of the entity name. Such cases are challenging for traditional text matching methods since the abbreviation has very few common features with the complete name. Moreover, biomedical terms usually appear at a much lower frequency, and hence it is hard for models to learn the mapping through training. LATTE overcomes this problem by exploiting the type information. Although the mention may have a lower frequency, each type has a large number of samples to train the type classifiers. Therefore our model can classify the mention type with higher confidence. If the candidates have different types, entity linking decisions can be made with the knowledge of the type classification result. Note that, instead of direct usage, the type classification result is incorporated in the similarity computation. Figure 5.4(b) shows the case when the mention has OOV words. OOV words problem is a major challenge in the biomedical domain. Many biomedical terms do not have pre-trained word embeddings, without which the text matching becomes clueless. This is also why the retrieved result of the baseline model is incorrect. The character encoding and type matching in LATTE address this problem effectively. Figure 5.4(c) shows that when the candidates have the same type, LATTE can successfully distinguish the correct entity from other candidates. This is because: 1) the cross-attention mechanism is powerful in matching the mention and candidates text and 2) as discussed in previous sections, the latent types can be different even when the candidates share the same known type. Therefore the latent type modeling of LATTE works effectively in this case.

5.4 Conclusion

We proposed a novel methodology, which models the latent type of mentions and entities, to improve the biomedical entity linking task. We incorporate this methodology in LATTE, a novel neural architecture that jointly performs fine-grained type learning and entity disambiguation. To the best of our knowledge, this is the first work to propose the idea of latent type modeling and apply it to biomedical entity linking. Our extensive set of experimental results shows that latent type modeling improves entity linking performance, and outperforms state-of-the-art baseline models. The idea of latent type modeling can be useful to a wider range of tasks, such as in other text matching tasks, and other non-biomedical domains. These can be possible directions for future work.

Chapter 6

A Machine Learning Benchmark for Cross-lingual Code Intelligence

Recent advances in machine learning have significantly improved the understanding of source code data and achieved good performance on a number of downstream tasks. Open source repositories like GitHub enable this process with rich unlabeled code data. However, the lack of high quality labeled data has largely hindered the progress of several code related tasks, such as program translation, summarization, synthesis, and code search. This chapter introduces *XLCoST*, **Cross-L**ingual **Co**de **S**nippe**T** dataset, a new benchmark dataset for cross-lingual code intelligence. Our dataset contains fine-grained parallel data from 8 languages (7 commonly used programming languages and English), and supports 10 cross-lingual code tasks. To the best of our knowledge, it is the largest parallel dataset for source code both in terms of size and the number of languages. We also provide the performance of several state-of-the-art baseline models for each task. We believe this new dataset can be a valuable asset for the research community and facilitate the development and validation of new methods for cross-lingual code intelligence[1]. This chapter is adapted from a paper [151] accepted in the Deep Learning for Code (DL4C) workshop in ICLR 2023, of which I am the first author and primary contributor.

[1] https://github.com/reddy-lab-code-research/XLCoST

Table 6.1: Comparison against other parallel code datasets (Py - Python, JS - JavaScript). Column "Size" refers to the number of parallel data pairs. *This number is for single programs, not pairs.

Dataset	Alignment	Task	Labelling	Size	Languages
CodeNet	Program	Multiple	Solutions to the same problem	13.9M*	55 programming languages
AVATAR	Program	Translation	Solutions to the same problem	57,414	Java, Py
CodeXGLUE	Method	Multiple	Matching function names	11,800	Java, C#
CoST	Snippet	Translation	Matching code comments	132,046	C++, Java, Py, C#, JS, PHP, C
XLCoST	Snippet	Multiple	Matching code comments	1,002,296	C++, Java, Py, C#, JS, PHP, C, English

6.1 Introduction

Recent advances in machine learning have benefited a number of code related tasks, such as code translation, code summarization, and code synthesis. Open-source code repository websites like Github provide an enormous amount of source code data, which enables the training of large-scale programming language models such as CodeBERT [27], PLBART [5], TransCoder [111] and CodeT5 [132]. These extensively pre-trained models have shown superior performance on benchmark datasets like CodeXGLUE [73].

Although open-source code data is abundant in quantity, it has several disadvantages when being used as training data for code-related models. First, most of the available code data is unlabeled. For tasks like Code Translation, Code Summarization, and Code Synthesis, high quality parallel data is critical for model training. However, it is difficult to mine parallel data from open-source projects. Second, labeled data is usually small in size. For example, the code translation data introduced in [152] only has around 70 programs for testing and 50 programs for validation. Due to the small size of evaluation data, the models trained on this dataset may not be thoroughly evaluated. Moreover, the available labeled datasets usually only cover a limited number of languages. For example, the Code Translation dataset in CodeXGLUE only covers 2 languages, Java and C#. Because of the scarcity of labeled data in some programming languages, code tasks in some low-resource languages remain unexplored.

In this chapter, we introduce *XLCoST*, a machine learning benchmark dataset that contains fine-grained parallel data in 7 commonly used programming languages (C++, Java, Python, C#, Javascript, PHP, C), and natural language (English). The data is parallel across 7 languages, at both code snippet level and program level. This means that, given a program in one language, the dataset contains the same program in up to 6 other programming languages. Each program is divided into several code snippets, and programs in all the languages are aligned at the snippet level. Moreover, each of the snippets is accompanied with a comment, and the comment for a particular snippet is the same across all the languages. Table 6.1 presents a comparative analysis of *XLCoST* in terms of the number of available parallel data samples against other widely used parallel code datasets. The dataset contains around 1 million parallel snippets and 123K parallel programs in total, which is significantly larger than many available parallel code datasets. We believe that this dataset is a valuable asset for the research community and can potentially benefit a number of code-related research problems.

To further facilitate the development and evaluation of models with a focus on source code, we also introduce 10 different cross-lingual tasks. These tasks can be divided into two categories: Generation and Retrieval. The generation tasks include Code Translation (Code-to-Code), Code Summarization (Code-to-Text), and Code Synthesis (Text-to-Code); the retrieval tasks include NL (Natural Language) Code Search and XL (Cross-Lingual) Code Search. Each task is at both snippet and program level.

To evaluate how challenging the tasks are with the proposed dataset, we run experiments on all the 10 tasks with a number of state-of-the-art baseline models. We also conduct an empirical study to understand how the model design relates with the performance on different tasks with *XLCoST* dataset. The primary contributions of this chapter are as follows:

- We introduce a new dataset which is parallel across 8 languages (7 programming languages and English) at both snippet level and program level. To the best of our knowledge, it is the largest **parallel** dataset for source code in both size and number of languages.
- We formulate 10 different cross-lingual tasks to facilitate the development and evaluation of models in this domain.
- We run experiments for all the 10 tasks on the proposed dataset with a number of state-of-the-art baseline models and provide insights about model design for the new challenges.

6.2 The $XLCoST$ dataset

The data for $XLCoST$ was collected from GeeksForGeeks[2], which is a website that houses thousands of data structures and algorithm problems along with solutions in up to 7 different programming languages: C++, Java, Python, C#, Javascript, PHP, and C. According to GeeksForGeeks, the solution programs for the same problem follow the same structure, down to the variable names. This results in the programs being semantically consistent across the different languages. In most cases, the programs for the same problem share the same set of comments in the same order, which indicates that they are parallel to the snippet level. This is where the fine-grained alignment in $XLCoST$ comes from.

6.2.1 Definitions

Problems: The problems are mostly about data structures and algorithms, as they are mainly designed for tutoring and coding interview preparation. Each problem has programs

[2]https://www.geeksforgeeks.org/

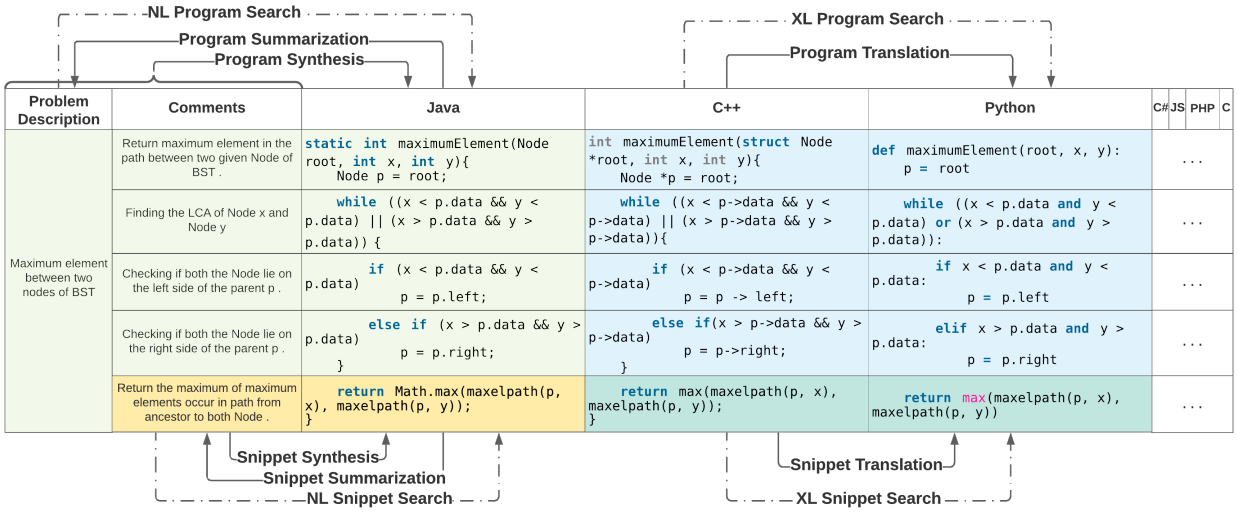

Figure 6.1: An illustration of the data and the tasks. The first column is the Problem Description; each cell in the second column is a Comment; each cell from the third column is a code Snippet. The combination of all the code snippets in a column is a Program (truncated due to space limitation). The arrows show the input and output data for each task. Solid lines are for generation tasks and dashed lines are for retrieval tasks. Note that the Program Synthesis task uses both Problem Description and Comments as input.

as solutions in up to 7 programming languages.

Programs: A program is a solution to a problem in a specific programming language. Each problem in this dataset may contain up to 7 programs (one for each language). The programs for the same problem share similar logic and structure.

Snippets: The code between two consecutive comments in a program is termed as a snippet (code before the first comment and after the last comment are also included). On an average, each program contains 8.81 snippets.

Description: Each problem also has a short description, for example, "Maximum Consecutive Increasing Path Length in Binary Tree."

Comments: The comments in each program in this dataset. The programs are well commented and each program has an average of around 9 comments.

Table 6.2: The train-valid-test split and basic statistics of *XLCoST* data. SN - Snippets; PR - Program.

	Snippet-level								Program-level							
Split	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**Total**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**Total**
train	93847	91089	81207	87583	70649	18027	3763	446165	9797	9623	9263	9345	8590	3087	463	50168
valid	4432	4460	3946	4436	3829	930	350	22383	492	494	472	491	475	158	60	2642
test	8118	8154	7293	8013	7033	1682	250	40543	909	911	887	899	886	308	51	4851
total	106K	104K	92K	100K	82K	21K	4363	509K	11198	11028	10622	10735	9951	3553	574	57661
Stats	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**Avg.**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**Avg.**
#lines	3.41	3.71	2.41	3.82	3.23	4	4.05	3.37	32.45	34.93	20.54	35.64	26.47	23.23	31.5	29.71
#code tokens	21.52	24.1	21.63	23.06	22.52	28.14	25.37	22.83	205	227.1	188.5	215.3	184.6	163.5	198	202
#text tokens	8.25	8.14	7.97	8.23	7.96	8.45	9.67	8.15	10.68	10.67	10.75	10.7	10.87	9.91	8.19	10.66
#SN/PR	–	–	–	–	–	–	–	–	9.52	9.42	8.51	9.33	8.2	5.81	7.77	8.81

6.2.2 Data Characteristics

The final dataset consists of 11,265 programming problems. As shown in Table 6.2, there are 57,661 unique programs. Each program consists of 8.81 snippets on average, which results in 509,091 snippets.

Multilingual: The dataset contains parallel data in 8 languages (7 commonly used programming languages and English).

Parallel: The dataset contains 4 types of parallel data: snippet-to-snippet, program-to-program, snippet-to-comment, program-to-problem (and comments), which further enables 10 different tasks.

Finely-aligned: The data is parallel at both snippet level and program level. To the best of our knowledge, this dataset is the finest-aligned among parallel code datasets.

Large: It is the largest parallel dataset for source code in terms of both size and number of languages.

Simple: Each program in this dataset is standalone without dependency on other programs. It ensures that the complexity of the tasks is controllable.

6.2.3 Data Collection and Processing

The data was scraped from different sub-pages of the GeeksForGeeks website. A majority of the problems on this site fall under two categories - Data Structures and Algorithms. The IP policies and regulations for GeeksForGeeks were carefully followed and we confirm that no data privacy policy was violated when collecting the data.

After collecting the data, we first removed duplicate problems, as some problems might be presented in multiple subcategories. Then we extracted problem description and solution programs in each available language from the page. Each program was sliced into code snippets by splitting at the comments, after which the comments and docstrings were removed from the programs. Any personal information such as the name of the code's contributor, was also removed from both the comments and the codes at this time. Eventually, we get 4 types of information from one page: 1) Problem Description; 2) Parallel programs in different languages; 3) Code Snippets; 4) Code Comments.

Data Alignment

The snippet-level alignment was done by matching comments in the solution programs (for the same problem) across different languages. As mentioned earlier, GeeksForGeeks programs follow a standard template, because of which the comments in different language programs (for the same problem) align parallelly in most cases. This yields parallel snippets that have the same functionality across different languages.

Misalignment detection: In some cases, the comments in different solution programs are not aligned. The misalignment can come from different numbers of comments, and the differences in the comment content. This is usually due to some solution program not strictly following the guidelines and templates. For solution programs with the same

number of comments, we evaluate the alignment by calculating the average similarity score of each pair of comments in the two programs (using Python *difflib.SequenceMatcher*[3]). If the average score is below a certain threshold (80% in our case), it would be categorized as misalignment and manual checking would be needed. Solution programs with different number of comments were automatically categorized as misaligned and sent for manual checking.

Manual checking and aligning: Manual checking was performed by two of the authors with good knowledge of programming languages and their functionalities. Based on the differences in number of comments, the misaligned programs were split into the following categories:

Category 0: The programs have the same number of comments. The type of misalignment usually only is due to different wording in the comments and can be easily fixed.

Category k: The difference in number of comments is k. When $k < 3$, extra comments needed to be discarded in some cases and code from these comments was moved to appropriate snippets to preserve the alignment with other languages. In some cases, there were also missing comments which had to be added along with the moving of the appropriate code block as in the previous case. When $k >= 3$, the programs will be discarded.

Data Splitting

Since the parallel programs are within each problem, splitting the data at problem level can naturally avoid data leakage. However, during the data processing, we noticed that some problems are very similar. For example, "Check if a large number is divisible by 3 or not" and "Check whether a large number is divisible by 53 or not". If one problem goes to the training set and the other goes to the test set, it can lead to potential data leakage and bias.

[3]https://docs.python.org/3/library/difflib.html

Table 6.3: An overview of the tasks. All the tasks have pairwise data at both snippet-level and program-level in 7 programming languages: C++, Java, Python, C#, Javascript, PHP, and C. The tasks can be divided into two categories: generation and retrieval. The generation tasks include Code Translation, Code Summarization and Code Synthesis; the retrieval tasks include NL (natural language) Code Search and XL (Cross-Lingual) Code Search.

Category		**Task**	**Data**	**Description**
Generation	Code Translation (Code-to-Code)	Snippet Translation	872K/47K/83K	Translate code snippet across programming languages
		Program Translation	106K/6K/11K	Translate program across programming languages
	Code Summarization (Code-to-Text)	Snippet Summarization	446K/22K/41K	Generate comment for given code snippet
		Program Summarization	50K/3K/5K	Generate problem description for given program
	Code Synthesis (Text-to-Code)	Snippet Synthesis	446K/22K/41K	Generate code snippet giving comment
		Program Synthesis	50K/3K/5K	Generate program from problem and comments
Retrieval	NL Code Search	Comment-to-Snippet Search	446K/22K/41K	Retrieve code snippet for given comment
		Problem-to-Program Search	50K/3K/5K	Retrieve program for given problem description
	XL Code Search	Snippet-to-Snippet Search	872K/47K/83K	Retrieve snippets in other languages for given snippet
		Program-to-Program Search	106K/6K/11K	Retrieve programs in other languages for given snippet

To address this concern, we first clustered all the similar problems into groups, and made the split at the group-level. In this way, we can ensure that similar problems go to the same split. To do so, we first calculate the similarity score (using Python *difflib.SequenceMatcher*) between every two pairs of problem descriptions, and group all the problems using various similarity score thresholds (60%-80%) based on length of the descriptions. The final split ratio in the data is around 85-5-10 for train-validation-test sets.

6.3 Code Tasks

The tasks can be divided into two categories: generation and retrieval. The generation tasks include Code Translation, Code Summarization, and Code Synthesis. The retrieval tasks include NL (natural language) Code Search and XL (Cross-Lingual) Code Search. All the tasks are at both snippet-level and program-level. Figure 6.1 shows the input and output

data for each of the tasks. Table 6.3 summarizes all the tasks introduced and some aggregate data statistics corresponding to each task.

Code Translation (Code-to-Code): Code Translation is the problem of converting source code from one programming language to another. Efficient and accurate code translation is valuable in scenarios like legacy code migration, software platform adaptation, etc. The *XLCoST* dataset provides parallel data in 7 common programming languages, supporting translation for 42 language pairs at both snippet and program level.

Code Summarization (Code-to-Text): The objective of the Code Summarization task is to generate natural language descriptions of the code that is given as input. We perform this task under two settings: generating snippet level summary by leveraging the comment-snippet pairings, and generating problem level summary using the problem description and program code pairings. Applications of this task include increasing the comprehensibility of uncommented or unfamiliar code to first time viewers and making it easier to collaborate as well as educate.

Code Synthesis (Text-to-Code): The Code Synthesis task focuses on generating source code from text inputs. It includes Snippet Synthesis and Program Synthesis. We use the comment of each code snippet as input to generate the code snippet for the Snippet Synthesis task, since they are of similar length (as shown in Table 6.2). However, programs are usually much longer (avg. 202 tokens) than problem descriptions (avg. 11 tokens). To generate programs, it is necessary that the input text is detailed and informative. Therefore, we use a combination of problem description and step-by-step comments as input to generate the entire program. Since the programs in *XLCoST* are well commented (9 comments/snippets per program on average) this ensures that the models have enough information to synthesize the whole program.

Code Search: The NL (Natural Language) Code Search in this chapter refers to using text input to retrieve relevant code. The snippet and program level task use Comment and Problem Description as query, respectively. XL (Cross-lingual) Code Search is the task of retrieving code that performs similar functions in multiple other languages given a piece of code in one particular language. Unlike NL code search, using code as queries to search for similarly functioning code in a multilingual setting is a relatively unexplored task. This task also includes both snippet and program level. To account for multiple correct answers, we use a modified *MRR* (Mean Reciprocal Rank) for evaluation.

6.4 Experiments

All the baselines were initialized with the pretrained weights and default configuration (including hyper-parameters) released by the corresponding original authors of the works. We changed the source and target sequence lengths to align with the dataset based on the task. The models were trained using 4 RTX 8000 GPUs with 48GB memory on each GPU. The code for training and evaluation is released in the GitHub repository of the dataset.

6.4.1 Evaluation Metrics and Baselines

We use the following metrics to evaluate different tasks proposed in this work: (i) BLEU [91] score to evaluate code-to-text generation tasks;, (ii) BLEU and CodeBLEU[4] [107] to evaluate code-to-code and text-to code generation tasks, and (iii) Mean Reciprocal Rank (MRR) to evaluate retrieval tasks.

We use the following models/methods for our comparison:

[4]We extended the CodeBLEU metric to support C and C++. Related code is released in the GitHub repo.

Naive Copy [73] directly copies the input source code as the output, which shows how similar two programming languages are. It is only used for translation tasks.

RoBERTa [71] is a robustly optimized version of BERT pretrained on huge natural language corpora. We use it only for retrieval tasks.

CodeBERT [27] uses the BERT [23] architecture pretrained on CodeSearchNet [49] data. We use the encoder-only version for retrieval tasks and encoder-decoder version (the decoder is randomly initialized) for generation tasks.

PLBART [5] is initialized with mBART [72] and further pretrained on a large-collection of Java and Python functions and natural language descriptions from Github and StackOverflow with denoising auto-encoding objective.

CodeT5 [132] employs the T5 [100] architecture and is pretrained on corpora of 8 programming languages (Java, Python, C#, JS, PHP, C, Ruby, Go) with identifier-aware objective.

6.4.2 Result Analysis

Table 6.4 shows the performance of baseline models for Code Translation, Code Synthesis, Code Summarization, and Code Search tasks.

Effect of Sequence-to-Sequence Pretraining: In Table 6.4, on average, CodeBERT performs significantly worse than PLBART and CodeT5 on almost all the generation tasks (refer to the first three sections of the table). Different from PLBART and CodeT5, which are both encoder-decoder models pretrained with sequence-to-sequence objectives, only the encoder in CodeBERT is pretrained, and the decoder weights are randomly initialized for sequence-to-sequence tasks. Experimental results show that the encoder-decoder architecture and sequence-to-sequence pretraining are better aligned with generation tasks and thus can potentially achieve superior performance.

Table 6.4: From top to bottom, the table contains results for Code Translation, Code Synthesis, Code Summarization, and Code Search at the snippet-level and program-level.

		Snippet-level							Program-level						
CodeBLEU	**Model**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
C++	Naive Copy	–	64.56	34.79	63.19	53.16	42.56	84.2	–	57.36	17.68	58.02	53.16	18.97	75.91
	CodeBERT	–	84.94	74.55	84.99	82.79	68.56	45.46	–	74.73	24.96	76.35	72.95	50.4	21.84
	PLBART	–	83.85	74.89	84.57	83.19	68.62	83.95	–	75.26	70.13	78.01	61.85	67.01	72.59
	CodeT5	–	**86.35**	**76.28**	**85.85**	**84.31**	**69.87**	**90.45**	–	**80.03**	**71.56**	**81.73**	**79.48**	**70.44**	**85.67**
Java	Naive Copy	70.85	–	35	78.43	57.81	42.49	69.74	64.25	–	39.87	72.68	57.81	42.51	62.48
	CodeBERT	87.27	–	58.39	92.26	84.63	67.26	39.94	79.36	–	8.51	84.43	76.02	51.42	21.22
	PLBART	87.31	–	58.3	90.78	85.42	67.44	72.47	81.41	–	66.29	83.34	80.14	67.12	63.37
	CodeT5	**88.26**	–	**74.59**	**92.56**	**86.22**	**69.02**	**82.78**	**84.26**	–	**69.57**	**87.79**	**80.67**	**69.44**	**78.78**
Python	Naive Copy	39.22	31.89	–	31.79	38.34	36.02	37.79	37.47	29.78	–	27.59	38.42	35.48	35.66
	CodeBERT	80.46	58.5	–	54.72	57.38	65.14	10.7	68.87	28.22	–	17.8	23.65	49.3	18.32
	PLBART	80.15	74.15	–	73.5	73.2	66.12	62.15	74.38	67.8	–	66.03	69.3	64.85	29.05
	CodeT5	**81.56**	**78.61**	–	**78.89**	**77.76**	**67.54**	**68.67**	**78.85**	**73.15**	–	**73.35**	**71.8**	**67.5**	**56.35**
C#	Naive Copy	69.78	78.71	34.77	–	57.85	42.53	66.73	64	73.63	40.09	–	57.79	42.96	60.87
	CodeBERT	86.96	90.15	56.92	–	84.38	67.18	40.43	78.52	82.25	10.82	–	75.46	51.76	21.63
	PLBART	84.98	6.27	69.82	–	85.02	67.3	75.74	80.17	81.37	67.02	–	79.81	67.12	57.6
	CodeT5	**88.06**	**91.69**	**73.85**	–	**85.95**	**68.97**	**81.09**	**83.59**	**85.7**	**69.52**	–	**80.5**	**69.63**	**77.35**
JS	Naive Copy	60.82	59.25	38.84	64.27	–	41.56	55.84	53.81	51.77	42.31	54.86	–	42.11	49.04
	CodeBERT	84.38	84.42	52.57	84.74	–	66.66	33.29	75.43	72.33	9.19	75.47	–	52.08	19.79
	PLBART	84.45	84.9	69.29	85.05	–	67.09	72.65	80.19	76.96	64.18	78.51	–	67.24	67.7
	CodeT5	**85.06**	**85.48**	**73.15**	**85.96**	–	**68.42**	**80.49**	**82.14**	**79.91**	**68.42**	**81.77**	–	**68.76**	**74.57**
PHP	Naive Copy	36.33	35.61	24.62	36.67	35.55	–	35.95	34.62	31.33	25.68	32.81	32.26	–	33.45
	CodeBERT	82.58	81.57	69.29	80.96	79.94	–	28.45	50.13	46.81	16.92	49.75	48.12	–	22.19
	PLBART	83.87	81.66	71.17	78	82.94	–	57.39	79.4	72.77	61.26	74.16	44.26	–	56.23
	CodeT5	**86.33**	**85.12**	**73.22**	**84.56**	**83.56**	–	**79.3**	**85.55**	**82.09**	**72.26**	**83.79**	**81.72**	–	**65.86**
C	Naive Copy	83.93	65.46	38.49	63.05	55.55	41.85	–	78.4	59.41	20.2	59.83	53.54	19.75	–
	CodeBERT	45.84	39.69	13.55	39.71	29.85	38.88	–	21.7	21.27	21.1	19.5	15.64	31.71	–
	PLBART	82.53	72.35	49.16	75.78	75.05	60.86	–	78.42	13.45	5.53	45.15	31.47	25.17	–
	CodeT5	**90.26**	**81.81**	**63.81**	**83.05**	**79.73**	**66.32**	–	**88.17**	**76.12**	**56.32**	**80.2**	**76.5**	**64.28**	–
CodeBLEU	**Model**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
Code Synthesis	CodeBERT	22.7	25.53	12.26	23.44	23.87	36.47	10.63	26.51	31.14	24.5	33.37	29.09	39.84	18.08
	PLBART	34.89	32.23	4.62	29.36	29.63	**37.56**	**22.88**	44.09	41.55	33.77	40.7	38.33	43.01	6.72
	CodeT5	**35.48**	**33.51**	**21.1**	**30.64**	**29.99**	36.37	21.93	**45.18**	**42.73**	**35.02**	**43.6**	**38.66**	**45.02**	**34.88**
BLEU	**Model**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
Code Summarization	CodeBERT	14.4	13.13	3.96	14.07	11.81	11.25	5.84	7.68	5.47	2.04	7.58	7.67	7.5	6.64
	PLBART	14.77	13.76	8	14.37	10.93	9.07	**7.5**	7.65	6.35	4.86	**9.23**	6.78	6.03	4.14
	CodeT5	**17.36**	**16.69**	**10.76**	**17.44**	**14.34**	**13.42**	6.63	**9.62**	**8.82**	**6.32**	7.75	**8.23**	**10.5**	**12.84**
MRR	**Model**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
NL Code Search	RoBERTa	25.77	25.85	27.08	25.64	26.78	33.47	36.14	51.47	50.4	48.98	52.24	50.05	62.01	**56.34**
	CodeBERT	**29.77**	**29.41**	**30.94**	**29.08**	**31.2**	**38.75**	**41.56**	**59.13**	**56.07**	**57.97**	**56.65**	**54.37**	**65.13**	47.13
XL Code Search	RoBERTa	41.73	41.25	36.16	41.18	43.17	**41.17**	37.1	48.28	47.66	46.11	46.4	47.6	43.76	40.15
	CodeBERT	**42.11**	**41.71**	**36.98**	**41.52**	**43.41**	41.09	**37.87**	**48.71**	**48.33**	**47.24**	**47.96**	**47.66**	**44.02**	**40.43**

Table 6.5: Transfer learning from Snippet-Level training for Program Translation task on low resource language C. ST - Snippet Transfer.

Model	C-C++	C-Java	C-Py	C-C#	C-JS	C-PHP	C++-C	Java-C	Py-C	C#-C	JS-C	PHP-C
CodeBERT	21.67	21.27	21.1	19.48	15.68	31.71	21.87	21.27	18.32	21.57	19.79	22.19
CodeBERT+ST	**38.85**	**37.55**	**19.79**	**33.52**	**27.1**	**37.61**	**31.99**	**30.52**	**24.07**	**34.16**	**29.67**	**28.35**
PLBART	78.42	13.43	5.53	45.14	31.42	25.17	72.61	63.4	29.01	57.6	67.71	56.15
PLBART+ST	**81.1**	**70.78**	**44.26**	**72.68**	**73.27**	**60.71**	**79.72**	**77.3**	**47.48**	**74.09**	**72.6**	**64.64**
CodeT5	88.17	76.15	56.3	80.2	76.42	64.28	85.67	78.76	56.44	77.38	74.56	65.8
CodeT5+ST	**89.06**	**79.04**	**62.61**	**80.53**	**78.59**	**68.31**	**88.96**	**82.08**	**60.97**	**80.93**	**79.58**	**77.58**

Table 6.6: Top compilation errors in each target language (Javascript not included).

Language	Top-3 Compilation Errors in Each Target Language		
C++	expected '}' at end of input	stray '#' in program	'define' does not name a type
Java	';' expected	not a statement	unclosed character literal
Python	SyntaxError: invalid syntax	SyntaxError: unexpected EOF while parsing	IndentationError: expected an indented block
C#	Too many characters in character literal	Unexpected symbol 'end-of-file'	Newline in constant
PHP	Syntax error, unexpected '}',expecting EOF..	Syntax error, unexpected ')'..	Syntax error, unexpected EOF on line 1
C	expected declaration or statement at end of input	expected '=',',',';','asm' ... before ')' token	expected statement before ')' token

Effect of Pretraining on Specific Languages: CodeBERT is pretrained on CodeSearchNet, which contains data from 6 programming languages, Java, Python, Javascript, PHP, Ruby, and Go. PLBART is pretrained on Java and Python from GitHub data. CodeT5 is trained on the 6 languages from CodeSearchNet and additionally on C and C#. In Table 6.4, CodeT5 consistently outperforms the other two models for almost all generation tasks. When the source or target language is C, CodeT5 outperforms the other two by a wide margin. Pre-training on specific languages can potentially benefit the generation tasks with these languages as either input or output.

Performance on Low-Resource Languages: In Table 6.4, most models performs significantly worse on C compared to other languages, both when C is source or target language, in almost all the tasks (except for Code Search). As shown in Table 6.2, C has the least number of samples for all the tasks. It shows that tasks in low-resource languages are potentially more challenging.

Effect of Transfer Learning from Snippet-level Training: From Table 6.4, first section, we noticed that models perform significantly better at snippet-level than program-level on most language pairs in the translation task. This is because: 1) Snippets are much shorter than programs. As shown in Table 6.2, the average length of snippets is 1/7 of the programs. 2) Snippet data is much more than program data. As shown in Table 6.3, the amount of pairwise snippet data is 8 times larger than of program data. Motivated by this, we employ transfer learning from snippet-level training to improve the Program Translation performance on the low-resource language C. Table 6.5 shows the performance of each model with and without the transfer learning. For example, CodeBERT is trained only on program data; "CodeBERT + ST" (ST is short for Snippet Transfer) model is first trained on the snippet data, and then on program data. All the models' performance improves by a wide margin on all the language pairs after snippet-level transfer learning, both when C is the source or target language.

Top Compilation Errors in Generated Programs: Table 6.6 shows the top compilation error types from compiling the generated programs from the Program Translation task. We aggregated the results of generated programs from all the baselines by the target language, because: 1) the top error types of each baseline are very similar and 2) the space is limited. From this table, we can see that the top error types are mostly syntactic errors, such as bracket mismatch (C++, PHP, C), indentation mismatch (Python), missing ';' (Java). This indicates that the models need improvement in capturing the structure of the programs.

6.4.3 Limitations and Future Work

From our analysis of the results, we can conclude that Sequence-to-Sequence pretraining tasks, multilingual pretraining data, and Snippet-level Transfer Learning can potentially

improve the performance on multiple tasks and low resource languages. This is an important insight for the design and development of future models in this domain. A good code generation model should also be able to learn and preserve the structure of the code since the current models mostly make syntactic errors in generation. However, since the data we used is collected from the GeeksforGeeks website, it may not perfectly represent the complexity and variability of real-world software translation tasks. The difference in context, project size, and coding standards in real-world software projects may pose additional challenges not fully captured in the current dataset. For the evaluation of code generation tasks, we use CodeBLEU as metric, which evaluates the code syntax and semantics along with n-gram matching (as in BLEU). However, the evaluation can be further improved by using test cases. Automated test case generation can be explored in future work. The tasks we introduce aim to rigorously evaluate code models with the parallel data from the dataset. Therefore, not all the tasks have practical applications in real-world, especially the snippet-level tasks. One future direction is to make use of the comments and snippets to iteratively generate programs.

6.5 Conclusion

In this chapter, we introduce a new dataset which is parallel across 8 languages (7 programming languages and 1 natural language) at both snippet level and program level. To the best of our knoweldge, it is the largest parallel dataset for source code in terms of both size and number of languages. We also introduce 10 different cross-lingual tasks to facilitate the development and evaluation of models in this domain. Moreover, we run experiments for all the 10 tasks on the proposed dataset with a number of state-of-the-art baseline models and provided insights about model design for the new challenges. We believe that this dataset

will be of significant value to the research community and can potentially benefit a number of code-related research problems.

Part III

Learning under Limited Supervision

Chapter 7

Multilingual Code Snippets Training for Program Translation

Program translation aims to translate source code from one programming language to another. It is particularly useful in applications such as multiple-platform adaptation and legacy code migration. Traditional rule-based program translation methods usually rely on meticulous manual rule-crafting, which is costly both in terms of time and effort. Recently, neural network based methods have been developed to address this problem. However, the absence of high-quality parallel code data is one of the main bottlenecks which impedes the development of program translation models. In this chapter, we introduce *CoST*, a new multilingual **Co**de **S**nippet **T**ranslation dataset that contains parallel data from 7 commonly used programming languages. The dataset is parallel at the level of code snippets, which provides much more fine-grained alignments between different languages than the existing translation datasets. We also propose a new program translation model that leverages multilingual snippet denoising auto-encoding and **Mu**ltilingual **S**nippet **T**ranslation (MuST) pre-training. Extensive experiments show that the multilingual snippet training is effective in improving program translation performance, especially for low-resource languages. Moreover, our training method shows good generalizability and consistently improves the translation performance of a number of baseline models. The proposed model outperforms the baselines on both snippet-level and program-level translation, and achieves state-of-the-

Java

```
import java.io.*;
class GFG {
    // Function to check whether a number is
divisible by 7
    static boolean isDivisibleBy7(int num) {

        // If number is negative,
        // make it positive
        if( num < 0 )
            return isDivisibleBy7( -num );

        // Base cases
        if( num == 0 || num == 7 )
            return true;
        if( num < 10 )
            return false;

 // Recur for ( num / 10 - 2 * num % 10 )
        return isDivisibleBy7(...

        ...
```

Python

```
# Function to check whether a number
is divisible by 7
def isDivisibleBy7(num) :

    # If number is negative
    # make it positive
    if num < 0 :
        return isDivisibleBy7( -num )

    # Base cases
    if( num == 0 or num == 7 ) :
        return True
    if( num < 10 ) :
        return False

    # Recur for ( num / 10 - 2 * num %
10 )
    return isDivisibleBy7(...

    ...
```

PHP

```
<?php
// Function to check whether a
number is divisible by 7
function isDivisibleBy7( $num ){

    // If number is negative,
    // make it positive
    if( $num < 0 )
        return isDivisibleBy7( -$num );

    // Base cases
    if( $num == 0 || $num == 7 )
        return 1;
    if( $num < 10 )
        return 0;

// Recur for ( num / 10 - 2 * num % 10 )
    return isDivisibleBy7(...

    ...
```

C

```
#include <stdio.h>
// Function to check whether a number
is divisible by 7
int isDivisibleBy7( int num ) {

    // If number is negative,
    // make it positive
    if( num < 0 )
        return isDivisibleBy7( -num );

    // Base cases
    if( num == 0 || num == 7 )
        return 1;
    if( num < 10 )
        return 0;

 // Recur for ( num / 10 - 2 * num %
10 )
    return isDivisibleBy7(...

    ...
```

Figure 7.1: An example of a program and code snippets in different languages from our *CoST* dataset. Each column is one program (truncated) in a specific language. Each cell is one snippet. The snippets are aligned by matching the code comments in different languages. We show only four languages due to space constraints. All the remaining languages are shown in the Appendix.

art performance on CodeXGLUE translation task. The code, data, and appendix for this work can be found at https://github.com/reddy-lab-code-research/MuST-CoST. This chapter is adapted from a paper [152] published in the Proceedings of the AAAI Conference on Artificial Intelligence in 2022, of which I am the first author and primary contributor.

7.1 Introduction

Program Translation is the problem of converting source code from one programming language to another. Different from computer compilers which translate high-level programming languages to lower-level machine code, it mainly focuses on translation between high-level programming languages. Efficient and accurate program translation is of enormous value in a variety of scenarios, such as: 1) *Migrating legacy code to another language.* For instance, many industries spend several hundreds of millions of dollars to convert code written in older programming languages (such as FORTRAN and COBOL) to newer ones (such as Java and C++) [110]. 2) *Adapting software to different operating systems and platforms.* For instance, for an Android application to run on iOS and Web browsers, it needs to be re-developed

in Objective-C and Javascript. Traditional rule-based program translation usually relies on meticulous manual rule-crafting, which requires expertise in both programming languages, and requires an enormous amount of time and resources.

In recent years, deep learning based methods have been employed to address this problem. The success of transformer-based models [127] in natural language processing (NLP) has motivated researchers to utilize them for programming languages. A few recent works based on neural machine translation (NMT) have been applied to this task and achieved some impressive results [5, 110]. One of the important requirements for NMT models is the availability of high-quality parallel data for model training. Such data is even more critical for the program translation problem since it requires the generated code to be logically precise as well. However, existing code translation datasets have significant limitations. Most of the commonly used datasets [17, 56, 73, 85, 86] only contain two languages (Java and C#), and the alignment comes from mining similar function names from open source projects. Github has a huge number of open-source repositories in several languages. However, the data is not parallel and cannot be used for supervised translation. Project CodeNet [96] and Google Code Jam[1] datasets contain solutions submitted to coding problems in multiple programming languages. However, given that the alignment comes from solutions to the same problems, they are aligned at the task level. Since programs that solve the same problem can have a high diversity in terms of variable names, method design and logical flow, these datasets are not ideal to train program translation models. This especially becomes a bottleneck in case of low resource languages, since models for those languages cannot be trained using limited data with high variance in distribution.

The scarcity of high quality parallel data has become a bottleneck in program translation research. In this chapter, we introduce *CoST* (**Co**de **S**nippet **T**ranslation), a new dataset

[1]https://codingcompetitions.withgoogle.com/codejam/archive

that consists of parallel source code snippets from 7 common programming languages: C++, Java, Python, C#, Javascript, PHP, and C. It contains parallel data at multiple levels, first at the snippet level, and then at the program level, for every pair of languages. To the best of our knowledge, *CoST* is the only dataset that provides snippet-level alignment for the seven commonly used programming languages. This dataset is not only a great resource to the program translation research community, but also serves as a new benchmark to evaluate the program translation models for up to 42 (7 by 6) programming language pairs at both snippet-level and program-level. In addition to supporting pairwise training, many samples in our dataset contain equivalent code snippets across multiple languages, thus supporting the development of multilingual program translation methods. An example of a program and its snippets in multiple languages is shown in Figure 7.1.

To demonstrate the effectiveness of using finely-grained alignment from code snippets for program translation, we propose a multilingual program translation model that leverages the similarity between different programming languages and the snippet level alignment of the dataset. Our experimental results show that the proposed model outperforms a number of baseline models on most of the 42 language pairs, on both snippet-level and program-level translation. The improvements are especially significant in case of low resource languages, that greatly benefit from the multilingual training. We also achieved state-of-the-art performance on the CodeXGLUE [73] translation task. Moreover, our multilingual snippet translation (MuST) pre-training also shows good generalizability across different models. Extensive experiments show that it consistently improves the performance of multiple models on the translation of all the language pairs. In summary, the contributions of this chapter are listed below:

- We introduce *CoST*, a new dataset that consists of both snippet-level and program level parallel data from 7 programming languages. Our dataset can be used to train

Dataset	Alignment	Labeling	Size (pairwise)	Languages
Google Code Jam	Program	Solutions to the same problem	2,430,000*	20 programming languages
Project CodeNet	Program	Solutions to the same problem	13,916,828*	55 programming languages
Tree-to-tree Dataset1	Method	Compiler translation	20,000	CoffeeScript, JavaScript
Tree-to-tree Dataset2	Method	Matching function names	16,996	Java, C#
Phrase-Based Dataset	Method	Matching function names	21,821	Java, C#
CodeXGLUE	Method	Matching function names	13,300	Java, C#
CoST Dataset	Snippet	Matching code comments	132,046	C++, Java, Py, C#, JS, PHP, C

Table 7.1: Comparison between our dataset and other existing source code translation datasets. Tree-to-tree Dataset (1 and 2) are from [17]. Phrase-Based Dataset is from [56]. * The numbers given in these cases are those of single program samples, and not paired programs. Py is short for Python.

	C++	Java	Python	C#	Javascript	PHP	C
C++	–	13929	11930	13326	7596	3165	2188
Java	**1497**	–	11713	13905	7729	3194	2135
Python	**1419**	**1417**	–	11404	7165	3123	1779
C#	**1442**	**1495**	**1383**	–	7601	3192	2123
Javascript	**996**	**1009**	**962**	**994**	–	2917	1232
PHP	**548**	**552**	**545**	**552**	**512**	–	700
C	**267**	**281**	**263**	**273**	**196**	**135**	–

Table 7.2: The numbers of pairwise data in each language-pair. The upper triangle (in normal font) shows the number of parallel code snippets, and the lower triangle (in bold font) shows the number of parallel programs.

program translation models for up to 42 programming language pairs.

- We provide a new benchmark to evaluate program translation model on 42 programming language pairs. Extensive experiments demonstrate that models which achieve the best performance on some languages can do much worse on certain other languages.
- We propose a multilingual program translation model that leverages the similarity between different programming languages and the snippet level alignment of the dataset. The proposed model outperforms a number of baseline models and achieves state-of-the-art performance on the CodeXGLUE translation task.
- The MuST training method in our model has good generalizability and consistently improves the performance of several other models on program translation.

7.2 The Code Snippets Translation ($CoST$) Dataset

The **Co**de **S**nippets **T**ranslation ($CoST$) dataset consists of programs from 7 different languages: C, C++, C#, Python, Java, Javascript, and PHP, spanning across 1625 programming problems. The detailed statistics about the $CoST$ dataset are highlighted in Table 7.2. We define certain terms used in the context of this chapter as follows:

- **Programs:** These refer to the complete code solution in a specific language to a particular problem or task.
- **Snippet/Code snippet:** Each program may consist of one or more snippets which are in parallel to appropriate code snippets in other languages.

7.2.1 Data Collection and Processing

Our data was collected from the GeeksForGeeks website. The platform has a plethora of problem statements and solutions to those problems in up to 7 programming languages (C, C++, C#, Python, Java, Javascript, PHP). The platform also ensures that its contributors stick to a template in terms of the comments used in their programs and the code corresponding to those comments. By using the template, we could obtain a one-to-one correspondence between the code snippets in one language to those in other languages. In effect, this gives us a good number of parallel instances of code which can then be effectively used for code-to-code translation. However, there were a number of cases where this template did not work as anticipated. These cases include missing snippets, differences in functionality among languages resulting in vastly different program structures, and misaligned cells. To remedy this issue, we manually verified the code to identify different instances of non-compliance, and either modify the alignment or discard the example in extreme cases. Few of the URLs scraped from different pages sometimes pointed to the same program, thus resulting in dupli-

cate files. A duplication detection program was used to identify these duplicates and remove them.

7.2.2 Dataset Comparisons and Characteristics

As shown in Table 7.1, many of the existing source code translation datasets such as [17, 73] consisting of pairwise samples at the method level collect their samples from very similar publicly available repositories. However, they only have parallel data in two languages: Java and C#. Moreover, their mapping is at the method level, and there are relatively fewer method pairs available. Other datasets such as Google Code Jam (GCJ) and CodeNet [96] have an abundance of problem statements along with solutions, spanning over a wide range of languages. However, these datasets suffer from quality issues. For instance, in CodeNet, only about half of the problems are rated by the online judges to be an accepted solution to the problem. This leads to almost half the dataset having wrong solutions, which makes these erroneous samples unusable for the translation task. In contrast, our dataset contains programs which have been manually verified to ensure correctness at program and snippet levels, thereby resulting in higher quality and less noise.

A major drawback of the existing datasets is that the samples are aligned at program level, which implies less supervised alignment. Since program level alignment is based on programs doing similar tasks and achieving similar results on test cases, there is a significant amount of variation between the programs in multiple languages, due to differences in terms of method and variable names, as well as the logic flow. The granularity in our case is at the snippet level, which provides more supervision in contrast to the method level or program level mapping that exists in previous datasets. Moreover, the code snippets in our dataset are consistent in terms of variable and method names, and the programs in each language follow

similar logic flow.

7.3 The Proposed Method

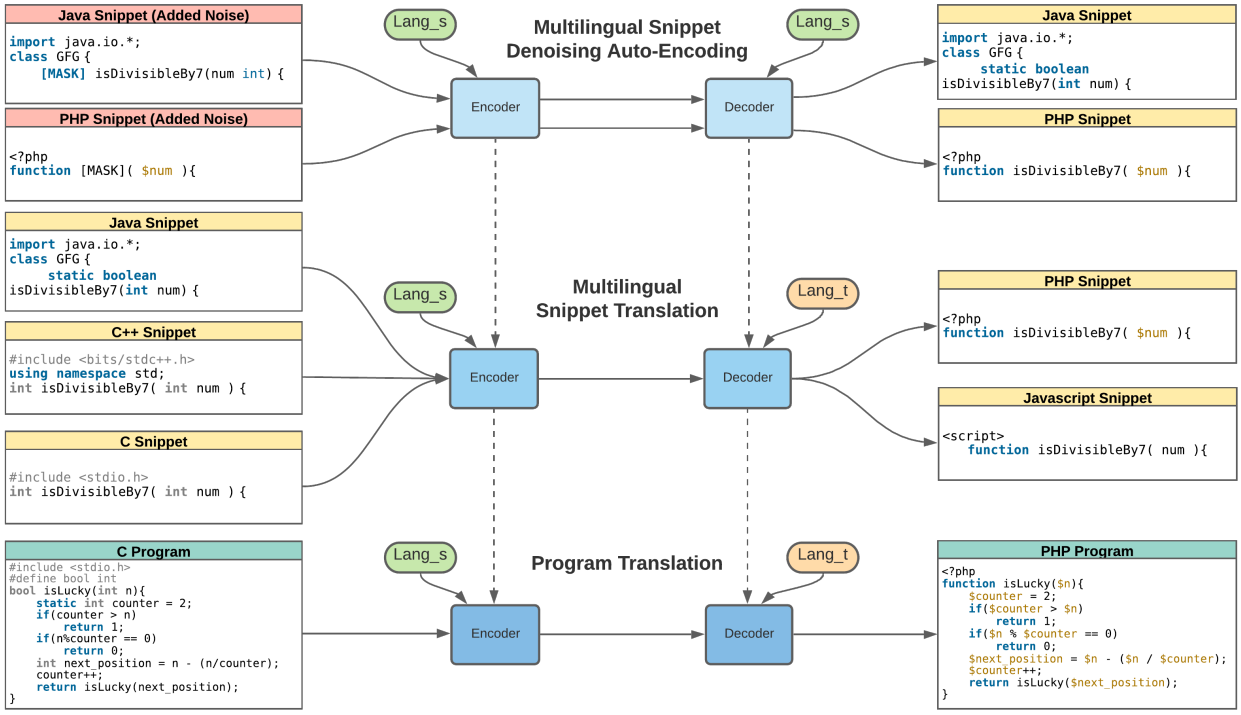

Figure 7.2: The training paradigm of the proposed MuST-PT model. We first train the model with multilingual snippet denoising auto-encoding, which helps the model to learn the similarity between different languages. Then we apply multilingual snippet translation (MuST) training to leverage the snippet-level alignment to increase the accuracy of program-level translation. Finally, we fine-tune the model on the program translation task to bridge the distribution gap between snippet and program data. *Lang_s* and *Lang_t* refers to source and target language, respectively. At each step of the training, the model takes both the code and the programming language as inputs.

7.3.1 Problem Formulation

Consider $L = \{l_1, .., l_k\}$ as the set of all languages, where l_i denotes a programming language.

Given a program $\mathbf{X}$ in language l_i, the objective of program translation is to generate a

program $\mathbf{Y}$ in the target language l_j. We represent a program consisting of m snippets as $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_m\}$, where $\mathbf{x}_i = (x_1, ..., x_n)$ denotes a snippet with n tokens. We further denote the monolingual snippet dataset in language l_i as $\mathcal{D}_{l_i}^{mono}$, and the bilingual snippet dataset for languages l_i and l_j as $\mathcal{D}_{l_i,l_j}^{bi}$.

7.3.2 Model Architecture

Given the sequence-to-sequence nature of the program translation problem, our model draws inspiration from the Transformer model [127], which has been shown to have state-of-the-art performance on many language generation tasks. The encoder-decoder based transformer model serves as the base model for our translation task. The model consists of an encoder E and a decoder G with parameters θ_E and θ_G, respectively, that are augmented to support code from multiple languages. This is done by using a unique identifier α_{l_i} for each language. Given the input token embeddings $\mathbf{x} = (x_1, ..., x_n)$, we add the language identifier to each token, such that $(x_1 + \alpha_{l_i}, ..., x_n + \alpha_{l_i})$ serves as the input to the encoder. The encoder representations $\mathbf{z} = E(\mathbf{x}, \alpha_{l_i})$ are then fed to the decoder along with the target language identifier α_{l_j} to generate output snippet tokens $\mathbf{y} = G(\mathbf{z}, \alpha_{l_j})$.

7.3.3 Model Initialization

We initialize the model parameters with the pre-trained weights of the DOBF model [112]. DOBF is a Transformer-based model trained with masked language modeling (MLM) and code deobfusctation objectives on Python and Java files from the GitHub public dataset available on Google BigQuery. The MLM objective helps the model to learn representations by leveraging the left and right contexts. The deobfusctation objective guides the model to recover the original class, function, and variable names from obfuscated code, which is a more

difficult task and requires a deeper understanding of the code, thereby providing a better learning signal to the model. By initializing our model with the weights of a sequence-to-sequence model pre-trained on source code, we can leverage its knowledge about the syntax and structure of the specific programming languages.

7.3.4 Multilingual Snippet Denoising Auto-Encoding

To train the model to perform translation on different language pairs, we first need to familiarize the model with all the 7 languages. Although the model is initialized with pre-trained weights from DOBF, the weights were learned from only two languages, Python and Java. Therefore, the model has no knowledge about other languages (C++, C#, Javascript, PHP, C). To address this issue, we first train the model with Denoising Auto-Encoding (DAE) objective [65] on snippets from all the languages. There are several advantages of doing this pre-training task. First, the sequence-to-sequence nature of DAE enables the model to decode all the languages, which is necessary for the translation task. Second, by sharing the same encoder and decoder across all the languages, all the languages are mapped into the same latent space. This helps the model to learn the similarities between different languages, which can be useful in the translation of low-resource languages. Third, the DAE only requires monolingual data, which is much more accessible than pairwise data. We use the same set of noise functions as TransCoder [110], which includes random word shuffle, random word dropout, and random span masking. Considering C as the noise model (non-learnable in this case), and $\mathbf{x}$ as the input sampled from $D_{l_i}^{mono}$, the DAE objective can be written as:

$$\mathcal{L}_{DAE}(\theta_E, \theta_G) = \sum_{l_i \in L} \mathbb{E}_{\mathbf{x} \sim D_{l_i}^{mono}, \tilde{\mathbf{x}} \sim C(\mathbf{x})} [-\log p_G(\mathbf{x} | E(\tilde{\mathbf{x}}, \alpha_{l_i}), \alpha_{l_i})] \tag{7.1}$$

7.3.5 Multilingual Snippet Translation (MuST)

In many language generation tasks, the performance goes down significantly as the length of input sequences increases. This is a common problem in sequence-to-sequence models due to the difficulty of capturing long-distance dependencies. Since source code programs usually contain at least tens of lines, achieving acceptable performance from translation models can be challenging. In order to alleviate this problem, we use code snippets translation as a pre-training method to improve the accuracy of program translation. Since the code snippets are much shorter than programs, they provide a fine-grained supervision to the translation model, and thus can help to address the problem of reduced performance for longer inputs.

Another problem encountered by many existing models is that program translation datasets are usually not balanced in size for all the languages. Some languages may have much less parallel data than others. For example, in the *CoST* dataset, there are 13K snippet pairs for Java and C++, but only 700 pairs for C and PHP. Less parallel training data can significantly affect the translation performance on low-resource languages. Therefore, in addition to snippet translation, we propose to leverage the multilingual training to improve the performance on low-resource languages. In *CoST* dataset, one code snippet may have corresponding snippets in multiple languages. Moreover, some languages are naturally similar in syntax, such as C++-C, Java-C, and Java-C#. This motivates us to use other languages to improve the translation of low resource languages, e.g. using C++-PHP and Java-PHP data to improve the translation of C-PHP. For a snippet pair $(x, y) \in \mathcal{D}^{bi}_{l_i,l_j}$, the objective function for this task can be written as:

$$\mathcal{L}_M(\theta_E, \theta_G) = \sum_{l_i,l_j \in L} \mathbb{E}_{(x,y)\sim\mathcal{D}^{bi}_{l_i,l_j}}[-\log p_G(\mathbf{y}|E(\mathbf{x}, \alpha_{l_i}), \alpha_{l_j})] \tag{7.2}$$

$$\mathcal{L} = \mathcal{L}_M + \lambda \mathcal{L}_{DAE} \tag{7.3}$$

The overall training objective of our model is given above. Here, λ is a hyper-parameter that represents the weight of DAE loss. After the multilingual snippet DAE and MuST pre-training, the model is capable of translating code snippets across all the 42 language pairs. However, because of the difference in length between code snippets and programs, the model cannot directly be used for program translation. Therefore, we further fine-tune the model on the program pairs from our dataset. We adopt a similar multilingual training strategy on the program-level pairwise data. The overall training process is illustrated in Fig. 7.2. We refer to the model as MuST-PT, which is short for the **Mu**ltilingual **S**nippet **T**raining for **P**rogram **T**ranslation model.

7.3.6 Implementation Details

In our model, the encoder and decoder consist of 12 and 6 transformer layers, respectively. The transformer units have a model dimension of 768, and 12 attention heads. The weight of the multilingual snippet DAE objective λ was set to 1.0 in the beginning, and decayed to 0.1 linearly in 30K steps, and then to 0 in 100K steps. The DOBF model we used for initializing our model is dobf_plus_denoising.pth, which can be found on the corresponding GitHub repository. Most of the settings during training were the same as DOBF [112]. Float 16 operations were used to speed up the training. The model was trained using the Adam optimizer [58] with a learning rate of 0.0001, and the same learning rate scheduler was used from the Transformer [127]. We used a batch size of 128 on all the 42 language pairs. The batches of different languages pairs were sent to the model alternatively during training. The model was trained with 4 RTX 8000 GPUs with 48GB memory on each GPU.

7.4 Experiments

7.4.1 Datasets

The datasets used for the experimental evaluation are below:

- *CoST*, **Snippets Dataset** We used the monolingual snippets to do the multilingual snippet DAE training, and the pairwise snippets to do the multilingual snippet translation (MuST) training. The train-validation-test data is split at the problem level, to ensure no overlapping snippets between the splits in any of the languages.
- *CoST*, **Programs Dataset** We used the pairwise program data to fine-tune the model for program translation.
- **CodeXGLUE Translation Dataset** CodeXGLUE stands for General Language Understanding Evaluation benchmark for code. It has 10 source code related tasks, and code to code translation is one of them. We used the translation dataset (Java-C#) from CodeXGLUE for evaluation.

7.4.2 Evaluation Metrics

- **BLEU** Given an input code sample, we use BLEU [91] score to evaluate the n-gram overlap between the generated and the ground-truth target code.
- **CodeBLEU** CodeBLEU [107] is for automatic evaluation of code synthesis. Besides n-gram match as in BLEU, it also evaluates the code syntax via abstract syntax trees (AST) and code semantics via data-flow.

7.4.3 Baseline Methods

- **Naive Copy** Naive Copy [73] directly copies the input source code as the translation output. This baseline shows how similar two programming languages are.
- **Transformer** The sequence-to-sequence transformer model [127] was originally designed for translation problem. We use it as a baseline to see how well a transformer model performs without any pre-training on source code corpus.
- **CodeBERT** CodeBERT [27] uses the BERT architecture pre-trained on source code corpus.
- **DOBF** DOBF [112] is the model from which the weights are used to initialize our model. It is pre-trained on Java and Python.
- **TransCoder** TransCoder [111] is an unsupervised program translation model pre-trained on Java, Python, and C++. We did not include TransCoder in Table 7.4 because it does not support input languages other than the ones it was pre-trained on (for languages other than the languages it was pre-trained on, we observe that the model's performance does not increase over training, indicating that the model does not support these languages.).

Due to space limitations, we did not include some baselines (PLBART, GraphCodeBERT, RoBERTa(code) [71], PBSMT [147]) from the CodeXGLUE translation task in other experiments.

7.4.4 Results Analysis

Translation Performance on Snippets Table 7.4 shows the translation performance of our model and the baseline models on all the 42 language pairs. Every model is evaluated on both the snippets dataset and the program dataset. The left part of the Table shows the

	Java-C#		C#-Java	
Method	**BLEU**	**CodeBLEU**	**BLEU**	**CodeBLEU**
Naive copy	18.54	-	18.69	-
PBSMT	43.53	42.71	40.06	43.48
Transformer	55.84	63.74	50.47	61.59
RoBERTa(code)	77.46	83.07	71.99	80.18
CodeBERT	79.92	85.1	72.14	79.41
GraphCodeBERT	80.58	-	72.64	-
PLBART	83.02	**87.92**	78.35	85.27
MuST-PT	**87.37**	86.82	**85.25**	**86.09**

Table 7.3: Results on the CodeXGLUE translation task. Our model achieves state-of-the-art performance on BLEU score of C#-Java and both BLEU and CodeBLEU on Java-C#.

BLEU score of each model on the snippets dataset. We can see that our model outperforms the baseline models, with significant performance gains on low resource languages like PHP and C. This shows that the multilingual training in both DAE and MuST is helpful in improving low-resource language translation.

Translation Performance on Programs The right part of Table 7.4 shows the BLEU score of each model on the program dataset. We can see that almost all the baseline models have much worse performance on programs than snippets. This can be attributed to the more challenging nature of program-level translation due to longer sequence length compared to snippets, and less training data than snippet level. However, our model's performance does not drop by much on program-level compared to snippet level. This shows that the MuST pre-training improves the program translation performance.

Translation Performance on CodeXGLUE We also evaluated our model on the CodeXGLUE translation task. Table 7.3 shows the BLEU and CodeBLEU of our model compared to the models on the CodeXGLUE translation task leaderboard. Our model achieved state-of-the-art performance on BLEU score of both Java-C# and C#-Java, and high CodeBLEU score on C#-Java conversion. This indicates that the DAE and MuST training in our model is

		Snippet-level							Program-level						
Lang	**Model**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**	**C++**	**Java**	**Py**	**C#**	**JS**	**PHP**	**C**
C++	Naive Copy	–	68.87	35.03	69.54	57.71	37.7	87.73	–	66.57	36.58	67.22	55.24	36.27	**84.86**
	Transformer	–	68.74	57.17	70.61	63.26	60.94	68.57	–	43.93	33.9	45.32	39.02	35.93	25.06
	CodeBERT	–	71.61	60.28	72.31	72.4	70.42	61.29	–	53.47	38.37	63.01	46.6	46.18	22.25
	DOBF	–	79.83	68.61	81.74	79.24	77.91	68.09	–	29.06	18.5	29.14	22.25	27.47	27.05
	MuST-PT	–	**80.27**	**71.2**	**82.98**	**81.01**	**83.29**	**87.55**	–	**79.15**	**64.1**	**81.15**	**68.85**	**71.18**	84.2
Java	Naive Copy	68.75	–	33.8	77.9	58.58	33.6	70.22	66.53	–	34.56	77.15	56.52	32.14	67.54
	Transformer	74.42	–	53.98	84.27	69.16	58.5	46.18	44.38	–	31.22	47.34	39.06	38.26	25.36
	CodeBERT	73.19	–	59.04	85.12	76.79	7.24	50.33	65.48	–	38.7	85.46	55.92	47.12	32.98
	DOBF	80.83	–	64.75	89.73	79.89	66.94	59.32	28.34	–	18.08	27.6	20.2	27.05	26.12
	MuST-PT	**85.23**	–	**70.06**	**90.13**	**81.87**	**80.39**	**81.16**	**84.28**	–	**61.12**	**89.93**	**69.53**	**69.83**	**78.71**
Py	Naive Copy	35.02	33.53	–	35.11	41.71	23.57	35.29	36.58	34.27	–	35.69	40.85	22.48	36.53
	Transformer	60.5	58.13	–	60.9	55.59	55.07	39.37	37.42	38.15	–	36.91	38.39	39.01	19.99
	CodeBERT	65.04	61.79	–	63.84	62.43	62.6	45.09	43.96	41.35	–	46.4	47.28	44.38	46.4
	DOBF	68.73	67.91	–	69.46	68.07	67.8	34.21	21.49	23.45	–	21.82	20.32	26.53	13.02
	MuST-PT	**75.37**	**70.89**	–	**72.35**	**70.46**	**75.49**	**70.64**	**66.16**	**64.57**	–	**63.23**	**66.47**	**70.9**	**58.7**
C#	Naive Copy	69.5	78.05	35.16	–	60.23	35.43	70.65	67.16	77.23	35.76	–	58.4	33.57	67.9
	Transformer	75.68	84.19	58.64	–	66.97	60.57	45.18	42.65	45.6	32.64	–	39.66	38.47	25.01
	CodeBERT	74.73	82.16	59.74	–	77.12	67.48	49.64	67.17	82.45	41.1	–	51.09	48.62	34.33
	DOBF	81.77	**86.73**	67.96	–	80.26	15.94	28.35	26.97	29.17	19.71	–	19.34	27.05	19.11
	MuST-PT	**85.34**	85.8	**71.11**	–	**82.74**	**81.64**	**81.12**	**84.72**	**87.76**	**62.03**	–	**70**	**70.66**	**78.78**
JS	Naive Copy	57.67	57.99	41.73	60.04	–	32.56	57.6	55.11	55.74	40.9	58.1	–	29.77	53.89
	Transformer	65.06	65.31	56.92	64.55	–	61.87	37.34	39.8	39.6	34.3	41.72	–	37.65	19.78
	CodeBERT	68.76	71.66	58.13	72.87	–	66.35	37.08	49.51	48.91	46.27	51.55	–	47.95	24.37
	DOBF	78.56	76.94	64.92	75.5	–	75.53	52.32	26.47	25.93	21.77	21.43	–	26.73	18.68
	MuST-PT	**78.95**	**78.03**	**66.47**	**78.91**	–	**78.69**	**78.54**	**73.01**	**73.39**	**63.88**	**73.32**	–	**76.44**	**70.2**
PHP	Naive Copy	37.66	33.65	23.6	35.41	32.66	–	37.46	36.24	32.17	22.54	33.56	29.97	–	35.73
	Transformer	58.47	56.06	51.45	56.27	56.43	–	29.29	33.78	35.67	31.52	37.54	37.07	–	20.11
	CodeBERT	65.08	60.84	54.59	63.77	63.92	–	29.75	40.43	37.64	33.01	41.33	41.31	–	18.63
	DOBF	68.18	65.84	63.45	70.14	63.21	–	23.78	26.69	26.28	19.91	23.52	20.63	–	18.31
	MuST-PT	**79.41**	**76.42**	**69.34**	**77.96**	**77.64**	–	**76.67**	**70.04**	**67.3**	**63.97**	**70.34**	**73.54**	–	**67.88**
C	Naive Copy	87.63	70.29	35.37	70.62	57.74	37.45	–	84.75	67.56	36.61	67.88	54.17	35.75	–
	Transformer	68.63	45.42	36.4	44.38	35.37	31.03	–	29.54	30.73	24.62	31.28	24.55	24.83	–
	CodeBERT	64.18	51.1	36.48	49.81	33.75	28.85	–	27.96	35.29	22.05	32.82	21.73	25.19	–
	DOBF	76.85	64.73	53.1	45.11	30.87	22.22	–	16.84	23.23	17.64	23.96	20.38	25.7	–
	MuST-PT	**88.58**	**79.24**	**66.49**	**80.68**	**80.35**	**82.94**	–	**84.92**	**76.84**	**55.71**	**78.39**	**66.13**	**70.62**	–

Table 7.4: BLEU scores of baseline and the proposed MuST-PT model on all the 42 language pairs on both *CoST* snippet and program datasets. Note that only multilingual DAE and MuST were applied for snippet-level translation. We did program-level fine-tuning for MuST-PT only for program-level translation.

Model	Java-Py	Py-Java	Java-C++	C++-Java	Java-C#	C#-Java	Py-C++	C++-Py	Py-C#	C#-Py	C++-C#	C#-C++
Naive Copy	34.56	34.27	66.53	66.57	77.15	77.23	36.58	36.58	35.69	35.76	67.22	67.16
Transformer	31.22	38.15	44.38	43.93	47.34	45.6	37.42	33.9	36.91	32.64	45.32	42.65
Transformer+MuST	40.9	43.97	58.35	54.61	73.7	71.68	42.86	39.06	43.42	42.34	57.84	57.49
CodeBERT	38.7	41.35	65.48	53.47	85.46	82.45	43.96	38.37	46.4	41.1	63.01	67.17
CodeBERT+MuST	55.5	57.66	81.09	78.69	90.47	86.76	58.91	55.98	59.13	55.45	79.05	81.54
TransCoder	24.98	21.98	30.09	30.42	44.85	29.4	23.03	23.52	40.4	18.81	41.91	25.3
TransCoder+MuST	60.73	65.53	87.09	81.64	91.74	27.7	68.7	62.92	66.52	16.88	82.4	29.44

Table 7.5: Multilingual Snippet Translation (MuST) training consistently improves the performance (measured by BLEU scores) of the baseline models on the *CoST* program translation dataset. This shows that the MuST pre-training method can be generalized to other models and benefit their translation performance.

effective on other program translation datasets.

Generalizability of MuST Training We combine some of the baselines with MuST training to see if the method is generalizable to more models. Table 7.5 shows the results of each baseline before and after MuST training. We can see that all the three baselines get significant improvement after MuST training, indicating that MuST is not only effective in our model setting, but also benefits other models. This demonstrates that MuST has good generalizability and can potentially benefit other program translation models.

7.5 Conclusion and Future Work

Scarcity of high quality parallel data has become the bottleneck of program translation research. In this chapter, we introduced a new multilingual code translation dataset *CoST*, with snippet-level parallel data across 7 programming languages. Our dataset provides fine-grained supervision for the translation of 42 language pairs. We also propose a new program translation model that leverages multilingual snippet denoising auto-encoding (DAE) and multilingual snippet translation (MuST) pre-training. Our extensive set of experiments show

that DAE and MuST are effective in improving program translation performance, especially for low-resource languages. We also achived state-of-the-art performance on CodeXGLUE translation task. The MuST training also shows good generalizability and improves the translation performance of a number of baseline models.

Looking forward, there are several avenues for potential future research stemming from this work. Firstly, the snippet-level parallel data in *CoST* can be utilized to facilitate more sophisticated pre-training approaches. As the dataset contains seven programming languages, research could be conducted into how to more effectively leverage the multilingual aspect of the data, possibly by exploring language-agnostic features and representations of code. Second, the MuST pre-training approach has shown promising results in program translation tasks. Future work could investigate its applicability to other code-related tasks, such as code summarization, comment generation, and text-to-code generation. Exploring the use of MuST with different model architectures or modifying the MuST pre-training tasks to better align with these new tasks could also be beneficial.

Acknowledgments

This work was supported in part by the US National Science Foundation grants IIS-1838730 and Amazon AWS credits.

Chapter 8

Alignment-Enhancing Parallel Code Generation for Semi-Supervised Code Translation

Code translation is the task of converting source code from one programming language to another. Sufficient parallel code data is essential for neural code translation models to learn the correct alignment across different languages. However, existing parallel code data is limited in quantity and supported languages. In this chapter, we propose a semi-supervised code translation method, **SPACoder**, that leverages snippet training, static analysis, and compilation to generate synthetic parallel code with enhanced alignment in a scalable way, and improves code translation by curriculum learning based on the alignment level of training instances. SPACoder can be generalized to multiple languages and various models with little overhead. Extensive experiments show that SPACoder significantly improves code translation performance on C++, Java, Python, and C, outperforming state-of-the-art baselines by wide margins in execution-based evaluation (CA@1). Notably, we improve C translation by up to 43% with less than 150 annotated training instances.

8.1 Introduction

Code translation is the task of converting source code written in one programming language (PL) to another. This process is valuable for migrating existing code to other languages, and can significantly reduce legacy code maintenance costs and new platform development. Traditional methods for code translation are usually rule-based, which require high expertise in multiple programming languages and considerable manual effort. Benefiting from recent machine learning advances, data-driven methods have shown promising results in automated code translation [17, 111, 113, 152]. Similar to neural machine translation, neural code translation usually relies on parallel code data for sequence-to-sequence (seq2seq) training. Parallel code data refers to pairs of code snippets from different programming languages that are functionally equivalent and bug-free. Sufficient parallel code data is essential for training models to learn the correct alignment of data structures, APIs, and grammatical rules across different languages. However, existing parallel code data is limited in quantity and supported languages [6, 17, 56, 73, 85, 86, 152].

To reduce the dependence on parallel code data, one line of work follows the "pre-training - fine-tuning" approach [5, 31, 112, 132]. Large-scale online code repositories such as GitHub, introduce a vast amount of open source code data. These methods pre-train large language models (LLMs) on open source code with self-supervised learning techniques to gain general knowledge about programming languages, and then fine-tune them on small specialized datasets to perform downstream tasks. Nevertheless, pre-training tasks such as masked language modeling (MLM) [23, 27, 37] are usually quite different from the downstream tasks such as code translation, and the performance on the latter is limited by the discrepancy. Another line of work takes an unsupervised learning approach for code translation. Established techniques from unsupervised neural machine translation (NMT) [8, 9, 65], such as back-translation and denoising auto-encoding, can be applied to code data

Input (Java)

```
Boolean areElementsContiguous(int arr[], int n) {
    HashSet<Integer> us = new HashSet<Integer>();
    for (int i = 0; i < n; i++)
        us.add(arr[i]);
    int count = 1;
    int curr_ele = arr[0] - 1;
    while (us.contains(curr_ele) == true) {
        count++;
        curr_ele--;
    }
    curr_ele = arr[0] + 1;
    while (us.contains(curr_ele) == true) {
        count++;
        curr_ele++;
    }
    return (count == (us.size()));
}
```

Transcoder-ST (C++)

```
Boolean areElementsContiguous (int arr [ ] , int n ) {
 HashSet < int > us ;
 for ( int i = 0 ; i < n ; i ++ ) {
  us . add ( arr [ i ] ) ;
 }
 int count = 1 ;
 int curr_ele = arr [ 0 ] - 1 ;
 while ( us . contains ( curr_ele ) == true ) {
  count ++ ;
  curr_ele -- ;
 }
 curr_ele = arr [ 0 ] + 1 ;
 while ( us . contains ( curr_ele ) == true ) {
  count ++ ;
  curr_ele ++ ;
 }
 return ( count == ( us . size ( ) ) ) ;
}
```

SPACoder (C++)

```
bool areElementsContiguous ( int arr [ ] , int n ) {
 unordered_set < int > us ;
 for ( int i = 0 ; i < n ; i ++ )
  us . insert ( arr [ i ] ) ;
 int count = 1 ;
 int curr_ele = arr [ 0 ] - 1 ;
 while (us . find ( curr_ele ) != us . end ( ) ) {
  count ++ ;
  curr_ele -- ;
 }
 curr_ele = arr [ 0 ] + 1 ;
 while (us . find ( curr_ele ) != us . end ( ) ) {
  count ++ ;
  curr_ele ++ ;
 }
 return ( count == ( us . size ( ) ) ) ;
}
```

Figure 8.1: An example of the "Shallow Translation" problem, with the Java function shown in the first column as input, the C++ translations from baseline method TransCoder-ST and our proposed method SPACoder (with CodeT5 as generator). The highlighted parts show that TransCoder-ST's translation directly copied types, data structures and statements from the input Java code, which are non-existent or grammatically incorrect in the target language C++, while SPACoder was able to correctly convert them in the corresponding C++ grammar.

effectively, achieving good performance on code translation without relying on any parallel data [4, 7, 25, 61, 111, 124]. However, unsupervised learning introduces significant noise in the training process, which is particularly harmful to code generation tasks that require precision. Moreover, without training on sufficient parallel code, the self-supervised and unsupervised models can potentially learn incorrect mappings of syntax and data structures from one language to another. For example, they might directly copy tokens and statements from the source language when generating in the target language, or translate the input code token by token and ignore the grammatical rules of the target language. We refer to this issue as "shallow translation". Figure 8.1 illustrates an example of shallow translation.

Considering the limitations of existing methods, we argue it is crucial to generate high-quality and well-aligned parallel code data in a scalable way. However, there exist several challenges in parallel code generation. Programming languages follow rigorous grammatical rules, while neural code generation relies on probabilistic decoding, which is prone to variance and noise. The alignment between programming languages also requires high pre-

cision and small changes can cause errors or unexpected behavior in run-time. Moreover, it is challenging to generate parallel code for different languages and types of programs in a cost-efficient way, which hinders parallel code generation at scale. In this chapter, we propose a novel code translation method leveraging Semi-supervised Parallel code generation with enhanced cross-lingual Alignment (**SPACoder**). We first generate synthetic parallel code with varying levels of alignment and quality by leveraging snippet training, static code analysis, and compilation. We then train the code translation model on the synthetic and annotated parallel code in a curriculum based on the alignment level, noise level, and quantity of each type of data. Compared to existing methods, SPACoder has several advantages. The grammatical correctness of the synthetic parallel code is improved by filtering by compilation, which effectively removes buggy hypotheses. The alignment in the synthetic parallel code is enhanced by using a generator trained on snippet-level aligned data and filtering the generated code by matching key information with the input code. Moreover, our method can be applied to different languages and types of programs with little overhead, which enables parallel code synthesis at scale. Extensive experiments show that the synthetic parallel code significantly improves the performance of code translation for multiple languages, and is particularly effective for low-resource language.

Our contributions can be summarized as follows: **(1)** We propose a novel semi-supervised code translation method, SPACoder, that leverages snippet training, static analysis, and compilation to generate synthetic parallel code with enhanced alignment in a scalable way, and improves code translation by curriculum learning based on the alignment level, noise level, and quantity of training instances. SPACoder can be generalized to multiple languages and various models with little overhead. **(2)** We introduce a curriculum-based training approach, where the code translation model is trained on both synthetic parallel code and annotated parallel code, considering the alignment level, noise level, and quantity of

each type of data. We demonstrate that curriculum learning improves the code translation model's performance and enhances alignment across different languages, resulting in more precise translations. **(3)** We evaluate SPACoder with two different underlying models and over 1 million synthetic parallel code pairs across 4 languages, split into different datasets by the level of quality and alignment. Extensive experiments show that SPACoder successfully improves code translation performance by up to 30% on C++, Java, and Python, outperforming state-of-the-art baselines on translation between Python and C++ by 5.7%, C++ and Python by 6%, and Python and Java by 8% in execution-based evaluation (CA@1). Notably, our method improves C translations by up to 43% with less than 150 annotated training instances.

8.2 Related Work

Parallel Code Data Parallel code data refers to code pairs from different programming languages that are functionally equivalent and bug-free. One type of existing datasets is characterized by relatively high alignment but is limited in size and supported languages. For example, CodeXGLUE [73] constructed a Java – C# translation dataset by matching function names from open-source repositories. MuST-PT [152] introduced a program translation dataset CoST, with snippet-level alignment that supports 7 programming languages. CoST was collected from coding tutorial website GeeksforGeeks[1], where each coding problem is provided with solutions in up to 7 languages, and the solutions have similar structure and code comments. AVATAR [6] only supports the translation between Java and Python. Another type of dataset is usually significantly larger in size and supports a wider range of languages, but the alignment quality is low. They are usually collected from competitive

[1]https://www.geeksforgeeks.org/

online code judgments. Given a coding problem, users can submit their own solutions in various supported languages and get judged based on online tests. The user-contributed solutions to the same problems are collected as parallel code in different languages. For example, Google Code Jam and Project CodeNet [96] were both collected in this manner. However, due to the diverse background and the large number of users, the solutions for the same problem have wide discrepancies in distribution across different languages, which lowers the quality of the alignment.

Neural Code Translation Recent advances in machine learning, especially in self-supervised learning techniques, have benefited a wide range of tasks [63, 71, 72, 119, 127]. Some techniques from NLP were transferred to programming languages and have achieved great success. Similar to BERT [23], CodeBERT [27] is a code language model pre-trained on CodeSearchNet [49] with Masked Language Modeling (MLM). PLBART [5] is pre-trained the same way as BART [68], with the Denoising Auto-Encoding (DAE) [65] on GitHub data. Although CodeBERT and PLBART are pre-trained on code, they model code the same way as natural language sequences without taking code-specific features into consideration. Inspired by T5 [100], CodeT5 [132] is pre-trained on CodeSearchNet but with an identifier-aware objective to align more with programming language distributions. All three models use general pre-training to gain programming language intelligence, without optimizing for any specific tasks. They require fine-tuning on task-specific data to perform downstream tasks. TransCoder [111] is an unsupervised code translation model that relies on back-translation to generate pseudo-parallel code data during training. However, back-translation introduces noisy code into the training process, compromising the model's ability to generate high-quality translations. TransCoder-ST [113] improves TransCoder by adding automated unit tests to filter out invalid translations and reduce noise from the back-translation process. However, obtaining unit tests for different languages is expensive, and running unit tests

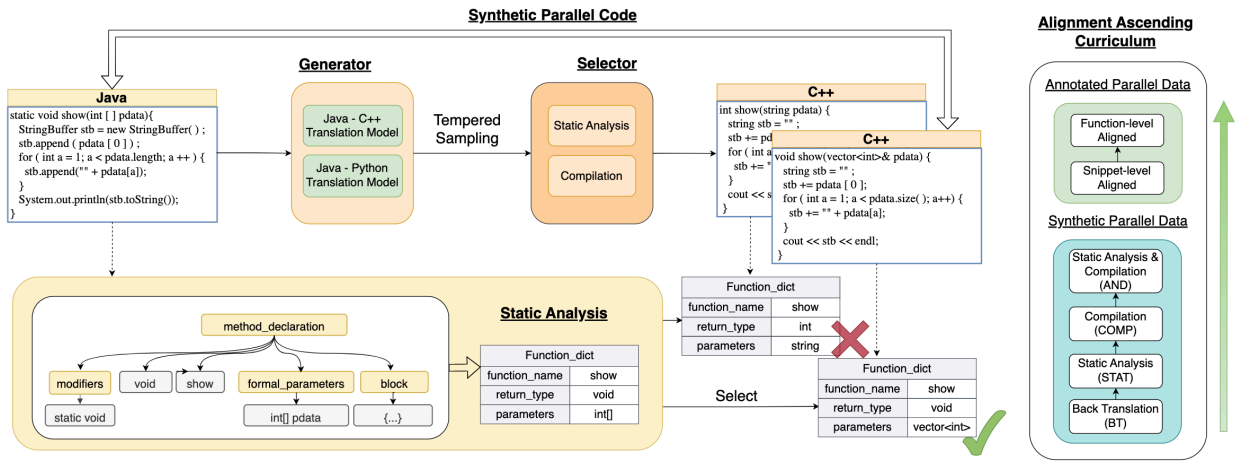

Figure 8.2: Overview of SPACoder for Code Translation. SPACoder utilizes a two-step process to generate high-quality translation hypotheses from monolingual code inputs. First, the generator produces multiple translation hypotheses using tempered sampling. Next, the selector applies static analysis and compilation techniques to select the most promising hypotheses. By employing various selection criteria, SPACoder generates synthetic parallel code datasets with varying alignment levels and quality. These synthetic datasets, along with annotated parallel code datasets, are organized into a curriculum, where the alignment and quality gradually improve. The proposed curriculum-based approach enhances code translation performance.

is unscalable for large amounts of code data. MuST-PT [152] leverages snippet-level DAE and translations for pre-training before fine-tuning on program-level data, which improves code translation performance. However, MuST-PT relies solely on a limited amount of finely aligned parallel code for training without utilizing widely available non-parallel code, which makes this method less scalable.

8.3 Method

The lack of parallel code data poses a challenge for training code translation models, which rely on large amounts of parallel data to achieve good performance. Semi-supervised methods can leverage monolingual data to generate synthetic parallel data but often struggle to

maintain alignment quality between the source and target languages. In this chapter, we focus on function-level code translation, as functions are the building blocks of programs. Figure 8.2 shows an overview of the proposed method.

8.3.1 Parallel Code Data Generation

To address the data scarcity challenge, we propose a parallel code generation method using semi-supervised learning. The method consists of two modules, a hypothesis generator f_G, and a selector f_D. The hypothesis generator f_G is a trained sequence-to-sequence model that takes as input a code snippet x_s from the source language s and generates a set of hypothetical translations $\mathcal{Y}_t^s = \{y_{t1}^s, y_{t2}^s, .., y_{tn}^s\}$ in the target language t, where n denotes the set size. Here, $\mathcal{Y}_t^s$ consists of several hypothetical translations for the same input code snippet x_s. The selector f_D comprises a set of k filtering criteria $\mathcal{F} = \{F_k\}_{k=1}^K$ where $\widetilde{\mathcal{Y}}_{t,k}^s = F_k(\mathcal{Y}_t^s)$ takes $\mathcal{Y}_t^s$ as input and outputs the subset of hypotheses $\widetilde{\mathcal{Y}}_{t,k}^s \subset \mathcal{Y}_t^s$ that passes the criteria.

Hypothesis Generation

The hypothesis generator f_G is initialized by training on a small parallel code dataset of L training instances, *i.e.*, $\mathcal{D}_L = \{(x_s, y_t)^{(l)}\}_{l=1}^{|\mathcal{D}_L|}$, in which x_s and y_t are parallel code from source language s and target language t. This step enables f_G to generate hypotheses with sufficient initial quality.

Snippet Training. We first train the generator f_G with a small annotated parallel code dataset in which the code pairs are aligned at snippet level. The snippet training helps the generator to learn fine-grained alignment between different languages and subsequently enables f_G to generate hypotheses with better alignment to the input code.

Tempered Sampling. Let $\mathcal{D}_U = \{x_i\}_{i=1}^{|\mathcal{D}_U|}$ be a monolingual dataset in source language

s, where each x_i is a function-level code block. With $\mathcal{D}_U$ as input, we can sample $\mathcal{H} = \{\mathcal{Y}_h^1, \mathcal{Y}_h^2, \ldots, \mathcal{Y}_h^i, \ldots, \mathcal{Y}_h^{|\mathcal{H}|}\}$ as output from f_G, where $\mathcal{Y}_h^i = \{y_{t1}^i, y_{t2}^i, .., y_{tz}^i\}$ is a set of translation hypotheses of x_i in target language t. To increase the diversity of the generated hypotheses and improve coverage for different possible translations of the input code, we make use of tempered sampling to acquire M different hypotheses from the generator for each input code. Tempered sampling makes use of a tuned scaled softmax to control the degree of randomness in the sampling process [3, 42].

Hypotheses Selection

The selector f_D takes $\mathcal{H}$ as input and outputs $\widetilde{\mathcal{H}} = \{\widetilde{\mathcal{Y}}_{h,k}^i\}_{i=1}^{|\widetilde{\mathcal{H}}|}$, in which $\widetilde{\mathcal{Y}}_{h,k}^i \subset \mathcal{Y}_h^i$ is the subset that passes the selection criteria $\widetilde{\mathcal{Y}}_{h,k}^i = F_k(\mathcal{Y}_h^i)$. If $\widetilde{\mathcal{Y}}_{h,k}^i$ contains more than one hypothesis, only one is kept, namely $y_{t,k}^i \sim \text{Sample}(\widetilde{\mathcal{Y}}_{h,k}^i)$, where Sample denotes the process of random sampling. Our preliminary experiments confirm that sampling more than one hypothesis does not yield improved performance. We pair all the $y_{t,k}^i$ with the input corresponding input code x_i to acquire pseudo parallel dataset $\mathcal{D}_S = \{(x_i, y_t)^{(l)}\}_{l=1}^{|\mathcal{D}_S|}$. In practice, we rely on cross-lingual static code analysis and compilation as selection criteria for the hypotheses.

Cross-Lingual Static Analysis. To ensure that the selected hypotheses have high alignment quality with the input code, we use cross-lingual static analysis to compare the key information of both the input code and all the hypotheses. Static code analysis is a technique used to analyze source code without executing the program. One way to perform static code analysis is through the use of an abstract syntax tree (AST). An AST is a tree-like data structure that represents the structure of a program's source code. It captures the high-level structure of the code and the relationships between its elements, enabling a deeper understanding of the code beyond the sequence-level. Figure 8.2 shows an example AST generated from a Java function.

Specifically, we compare the number of functions, and after matching each pair of functions from output with the input, we check whether the return types are equivalent, and if the parameter lists match in terms of the number of parameters and the type of each parameter. For non-typed languages such as Python, we skip the type part and only compare the number of functions and the parameter list of each function. Passing the cross-lingual static analysis is a strong indicator of the alignment quality of the hypotheses to the input, which helps in selecting the best hypotheses.

Compilation Filtering. We also leverage compilation to filter out hypotheses that contain errors. Specifically, we compile the generated code using the target compiler and check for any compilation errors. If a hypothesis fails to compile, it is discarded. This step further improves the quality of the selected hypotheses by ensuring that they are syntactically correct and can be compiled successfully.

8.3.2 Alignment-based Curriculum Learning

By pairing the hypotheses with their corresponding inputs, we obtain multiple synthetic parallel code datasets at different stages of the generation process. Without the selector, the generation is reduced to plain back-translation. We denote the unfiltered synthetic parallel data from the unfiltered hypotheses, as BT data. Similarly, we denote the synthetic parallel data from cross-lingual static analysis as STAT and COMP, respectively. In addition, we denote the subset of hypotheses that pass both criteria, static analysis and compilation, as AND data. We adopt a curriculum learning approach to train our code translation model, strategically leveraging the quality of the data at different stages. Our curriculum consists of multiple training phases, progressively incorporating different types of data. We first train with the unfiltered synthetic parallel data, allowing the model to grasp the basic translation

Input (Java)	Synthetic Output (C++)	Input (Python)	Synthetic Output (C++)

static void show(int [ ] pdata){
StringBuffer stb = new StringBuffer( ) ;
stb.append (pdata [0]) ;
for (int a = 1; a < pdata.length; a ++) {
stb.append("" + pdata[a]);
}
System.out.println(stb.toString());
}
	void show(vector<int>& pdata) {
	string stb = "" ;
	stb += pdata [0];
	for (int a = 1; a < pdata.size( ); a++) {
	stb += "" + pdata[a];
	}
	cout << stb << endl;
	}
		def f (a , b) :
		p , q = a , b
		while q != 0 :
		p , q = q , p % q
		return a / p * b
			float f (float a , float b) {
			int p = a , q = b ;
			while (q != 0) {
			p = q ;
			q = p % q ;
			}
			return a / p * b ;
			}

public static long gcd (long a , long b){
long c = a % b ;
if (c == 0) {
return b ;
}
else {
return gcd (b , c) ;
}
}
	long gcd (long a , long b) {
	long c = a % b ;
	if (c == 07)
	return b ;
	else
	return gcd (b , c) ;
	}
		def check (n , array) :
		if 1.1 <= n :
		array [0] += 1
		elif 0.6 <= n < 1.1 :
		array [1] += 1
		elif 0.2 <= n < 0.6 :
		array [2] += 1
		else :
		array [3] += 1
			void check (int n , int array [ ]) {
			if (1.0 <= n)
			array [0] ++ ;
			else if (1.0 <= n)
			array [1] ++ ;
			else if (0.2 <= n)
			array [2] ++ ;
			else array [3] ++ ;
			}

Figure 8.3: Synthetic parallel code examples from CodeNet-SPACoder, with PLBART [5] as generator. The synthetic parallel data demonstrates great alignment quality, with minor noise in some cases.

patterns. Next, we introduce the cross-lingual static analysis filtered data, which helps refine the model's understanding of language-specific code idioms and improve translation accuracy. Subsequently, we integrate the compilation filtered data, which further enhances the model's ability to generate syntactically correct translations. The curriculum then advances to utilize the intersection of both filtered datasets, combining the benefits of both data sources. We then introduce snippet-level annotated data to enhance translation performance in specific code segments. Finally, we conclude by training with function-level annotated data, enabling the model to capture higher-level structural patterns and produce more coherent translations. By following this carefully designed curriculum, SPACoder not only benefits from exposure to a diverse range of training data but also progressively refines its translation quality and alignment, leading to improved performance and robustness.

8.4 Experiments

Datasets. We make use of **CoST** [152], a code translation dataset that contains parallel code aligned at both function and snippet levels. We derive a pre-processed version of **CoST** for

execution-based training and evaluation, referred to as **ECoST** (Execution-based **CoST**). To create **ECoST**, we first execute all programs in **CoST** and remove the ones that throw compilation or run-time errors. To support execution-based evaluation, we also remove programs that execute but have empty output. **ECoST** has approximately 1,000 function-level training instances for C++, Java, and Python, and 150 for C. We employ a train/validation/test split ratio of approximately 70:5:25. We focus on the function-level code translation of four common programming languages, C++, Java, Python, and C. However, **CoST** consists of programs, not functions. To make **ECoST** support function-level experiments, we extract functions from each program through AST[2] and keep the rest of the program (referred to as `program_shell`) for evaluation later. Note that when there are multiple functions in one program, we keep all of them (except for the `main()` function).

Synthetic Parallel Code Generation. We use **CodeNet** [96] as a source of monolingual code inputs for parallel code generation. **CodeNet** is a large-scale dataset containing 13M programs spanning 55 languages. The programs in **CodeNet** originate from code submissions to online judge of programming problems. We select the "Accepted" submissions (*i.e.*, submissions that pass the online judge) in 4 languages, C++, Java, Python and C, from around 1,600 problems, which gives us approximately 1M programs. To ensure the quality of the input data, we set two filtering criteria: 1) the program should be modularized, which means it should contain at least one function (other than `main()` or `Main()` function), and 2) the program should be bug-free, which means it can be compiled without errors. After applying the two steps of filtering, only around 8% of the programs remain, approximately 87,000 examples. We experiment with two different models as the generator model, PLBART [5] and CodeT5 [132]. The monolingual **CodeNet** data are used as inputs to the generators to obtain the hypotheses through tempered sampling, and then get the synthetic parallel code

[2]https://tree-sitter.github.io/tree-sitter/

through selection by static analysis and compilation.

Baselines and Evaluation Metrics. We compare against five advanced code translation models. CodeBERT [27], PLBART [5], and CodeT5 [132] are programming language models pre-trained with self-supervised learning techniques on large-scale open-source code datasets. These models can perform code translation as a downstream task after fine-tuning on parallel code data. TransCoder [111] is an unsupervised code translation model that relied on back-translation for data augmentation. TransCoder-ST [113] improves TransCoder by leveraging unit testing to generate parallel code data. After generating the synthetic parallel code, we train code translation models using the generated data and evaluate the performance. CodeBERT, PLBART and CodeT5 need fine-tuning to perform code translation. Therefore, they are fine-tuned on **ECoST** with both snippet-level and function-level data. On the other hand, TransCoder and TransCoder-ST do not need fine-tuning as they are unsupervised methods. All models are evaluated on the **ECoST** test set. CodeBLEU[107] is a weighted sum of n-gram matching, AST matching, and data flow matching between source and target programs. Computation Accuracy (CA) [111] is a new metric introduced in TransCoder that measures if the hypothesis has the same execution output as the reference. We use CA@1 for all the evaluations. Model training details are included in the Appendix.

8.5 Results and Analysis

We evaluate two variations of our method, SPACoder-PLBART and SPACoder-CodeT5, by performing parallel code generation with PLBART and CodeT5 as generators and curriculum learning with their generated data respectively. The generated parallel code data is referred as CodeNet-SPACoder. We focus on two aspects, generated data quality and improvements in code translation performance.

PLBART	Number of Pairs						Selection Rate					
Selector	C++ – Java	C++ – Py	C++ – C	Java – Py	Java – C	Py – C	C++ – Java	C++ – Py	C++ – C	Java – Py	Java – C	Py – C
Back Translation	47540	63637	49550	37233	22919	39231	1	1	1	1	1	1
Static Analysis	25211	58157	14945	31228	13059	33882	0.53	0.91	0.30	0.84	0.57	0.86
Compilation	15258	36224	1893	13525	1562	11088	0.32	0.57	0.04	0.36	0.07	0.28
SA & Compilation	9278	34733	1200	12104	1313	10730	0.20	0.55	0.02	0.33	0.06	0.27

Table 8.1: Statistics of CodeNet-SPACoder, with PLBART [5] as generator. SA & Compilation refers to the intersection of the Static Analysis and Compilation selections.

PLBART	CodeBLEU						Computation Accuracy					
Dataset	Java – C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py	Java – C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py
CodeNet	21.61	21.70	22.14	19.16	20.31	18.92	0	3.13	0.64	1.57	3.37	0
ECoST-function	38.87	53.60	41.42	46.24	53.94	50.50	0.81	4.52	1.88	3.63	16.87	16.62
ECoST-snippet	**71.39**	**66.62**	**71.27**	**64.76**	**62.05**	60.62	25.54	24.40	27.15	23.87	32.23	32.33
CodeNet-SPACoder	69.02	65.92	70.96	63.54	61.77	**61.52**	**38.44**	**27.71**	**28.49**	**24.17**	**35.54**	**37.76**

Table 8.2: Performance comparison of the same model trained on existing parallel code data versus on CodeNet-SPACoder. The model used here is PLBART. The results from training on CodeNet-SPACoder demonstrate superior Computation Accuracy over existing parallel code data, indicating its high quality and effectiveness in improving code translation. Py is short for Python.

8.5.1 Quality of the Synthetic Parallel Code

Statistics of CodeNet-SPACoder. With 86,972 monolingual code as input, we manage to generate 516,142 and 529,108 synthetic parallel code pairs in 6 language pairs from PLBART and CodeT5, respectively. Table 8.1 shows the statistics of the synthetic parallel code data generated by PLBART. Note that the datasets resulting from static analysis and compilation are not subsets of back-translation, because for back-translation we randomly pick a hypothesis from the 10 sampled hypotheses, and for static analysis and compilation we select the hypothesis from the ones that pass the selection criteria. From the selection rate, we can observe that static analysis is the most lenient to Python, as it is a weakly-typed language. Compilation has the least selection rate on C. This is due to data scarcity as the generator has poor performance on C due to being trained with less than 150 examples.

Quantitative Analysis. We quantitatively evaluate the quality of CodeNet-SPACoder. Specifically, we train PLBART using only the generated data from SPACoder-PLBART

	CodeBLEU						Computation Accuracy					
Model	Java – C++	Py – C++	C++-Java	Py – Java	C++ – Py	Java – Py	Java – C++	Py – C++	C++ – Java	Py – Java	C++ –Py	Java – Py
CodeBERT	61.75	50.18	29.71	42.21	46.99	46.69	13.44	4.82	10.22	3.93	6.33	5.74
PLBART	71.39	66.62	71.27	64.76	62.05	60.62	25.54	24.40	27.15	23.87	32.23	32.33
CodeT5	72.76	64.99	72.13	64.26	59.16	61.25	37.63	19.28	41.13	23.87	20.78	24.77
Trancoder	72.54	66.47	70.36	63.61	56.29	55.29	49.73	25.60	40.86	22.36	41.87	46.22
Trancoder-ST	71.47	61.28	70.96	64.81	58.85	57.70	51.08	36.14	44.09	35.35	43.98	**51.96**
SPACoder-PLBART	74.55	68.43	72.90	67.14	63.09	63.47	41.94	35.24	40.05	33.84	38.55	41.09
SPACoder-CodeT5	**74.94**	**69.25**	**74.85**	**69.64**	**65.10**	**65.95**	**51.08**	**41.87**	**49.19**	**43.20**	**50.00**	49.55

Table 8.3: Performance comparison of two implementations of SPACoder with PLBART and CodeT5 against baseline approaches. The metrics used for comparison are CodeBLEU and Computation Accuracy (CA@1). Across both measures, SPACoder outperforms the baseline approaches, demonstrating its effectiveness in code translation.

and compare it with PLBART trained on other existing parallel data. Table 8.2 shows the performance from training on each dataset. We observe that on Computation Accuracy, CodeNet-SPACoder from PLBART outperforms the annotated datasets on all the language pairs, which is a strong indicator of its high quality and good alignment. CodeNet has the worst performance, with close to zero on Computation Accuracy, which is potentially due to its poor alignment quality.

Qualitative Analysis. We further perform qualitative analysis and manually inspect samples of the generated data. Table 8.3 illustrates four examples from the synthetic parallel code, with two in Java – C++, and two in Python – C++. The Java and Python codes are the monolingual input from CodeNet, and the C++ codes are the synthetic codes. The generated code snippets are in good alignment with their corresponding inputs, with correct mapping of types, data structures, and syntax. Note that the synthetic codes still contain some noise, for example, there are some mistranslations of the numbers from the input code. However, Table 8.3 and 8.4 results indicate that it does not impede the effectiveness of the synthetic code in improving code translation performance.

	CodeBLEU						Computation Accuracy					
Model	C++ – C	Java–C	Python – C	C – C++	C–Java	C – Python	C++ – C	Java – C	Python – C	C – C++	C – Java	C – Python
PLBART	40.66	56.85	43.66	42.77	32.49	52.98	2.60	0	1.56	5.19	0	14.06
SPACoder-PLBART	**79.08**	**72.37**	**61.73**	**80.34**	**68.79**	**61.92**	**33.77**	**28.77**	**17.19**	**48.05**	**23.29**	**28.12**
CodeT5	82.06	74.16	62.25	80.04	71.25	61.06	66.23	47.95	25.00	64.94	39.73	28.12
SPACoder-CodeT5	**82.26**	**74.59**	**63.87**	**81.24**	**74.21**	**66.65**	**68.83**	**56.16**	**31.25**	**64.94**	**45.21**	**51.56**

Table 8.4: Performance comparison before and after applying SPACoder on low-resource language C. The results show substantial performance improvements across all measures after the application of our method, indicating the effectiveness of SPACoder on low-resource languages.

	CodeBLEU						Computation Accuracy					
Model	Java – C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py	Java-C++	Py – C++	C++ – Java	Py – Java	C++ – Py	Java – Py
Base Model	38.87	53.6	41.42	46.24	53.94	50.50	0.81	4.52	1.88	3.63	16.87	16.62
BT	67.91	66.32	70.54	65.50	61.78	62.40	19.62	23.8	32.26	27.19	35.24	37.76
BT + STAT	74.35	68.23	71.56	66.19	62.45	62.83	38.71	28.92	34.95	29.31	36.45	38.97
BT + STAT + COMP	74.22	68.12	72.26	66.20	62.13	62.11	40.59	34.34	36.02	28.10	35.54	38.97
SPACoder	**74.55**	**68.43**	**72.90**	**67.14**	**63.09**	**63.47**	**41.94**	**35.24**	**40.05**	**33.84**	**38.55**	**41.09**

Table 8.5: SPACoder ablation study, showing the curriculum performance improvements in code translation as each synthetic parallel code dataset is added to the alignment-ascending curriculum. The results demonstrate the cumulative contribution of each synthetic dataset to enhancing the effectiveness of the training curriculum. The base model is PLBART trained on the ECoST-function dataset.

8.5.2 Improvement in Code Translation Performance

Comparison with Baseline Models. Table 8.3 shows the CodeBLEU and Computation Accuracy performance on C++, Java, and Python of the baseline models and SPACoder-PLBART and SPACoder-CodeT5. In terms of CodeBLEU, both SPACoder models outperform all baselines, with SPACoder-CodeT5 surpassing the best baseline performance by 8% on Python – C++ and Java – Python translation. In terms of Computation Accuracy, SPACoder-CodeT5 outperforms the best baseline performance by 5% on Python – C++ and C++ – Java, 6% on C++-Python, and 8% on Python-Java. Moreover, both SPACoder models outperform their respective generator models on all the language pairs and both metrics by a wide margin. Compared to CodeT5, SPACoder-CodeT5's Computation Accuracy on Python – C++ and Python – Java improves by 20%, and on Java – Python and C++ – Python the improvements are 25% and 30%, respectively.

Input (C++)	SPACoder (Python)	Transcoder-ST (Python)	CodeT5 (Python)	Groundtruth (Python)

int getMin(int arr[], int n) {
return *min_element(arr, arr + n); }
int getMax(int arr[], int n) {
return *max_element(arr, arr + n); }
	def getMin (arr , n) :
	return min (arr)
	def getMax (arr , n) :
	return max (arr)
		def getMin (arr , n) :
		return * min_element (arr , n)
			def getMin (arr , n) :
			return min_element (arr, arr + n)
			def getMax (arr , n) :
			return max_element (arr, arr + n
			)
				def getMin(arr,n):
				return min(arr)
				def getMax(arr,n):
				return max(arr)

Test Code (Python)	**Execution Output**			
arr = [12,1234,45,67,1] n = len(arr) getMin(arr, n); getMax(arr, n)	1 1234	SyntaxError: can't use starred expression here	NameError: name 'min_element' is not defined	1 1234
Input (Python)	**SPACoder (Java)**	**Transcoder-ST (Java)**	**CodeT5 (Java)**	**Groundtruth (Java)**

def nextPowerOf2(n):
count = 0;
if (n and not (n & (n - 1))):
return n
while(n != 0):
n >>= 1
count += 1
return 1 << count;
	static int nextPowerOf2 (int n) {
	int count = 0 ;
	if (n != 0 && (n & (n - 1)) == 0)
	return n ;
	while (n != 0) {
	n >>= 1 ;
	count ++ ;
	}
	return (1 << count) ;
	}
		public static int nextPowerOf2 (int n
		) {
		int count = 0 ;
		if ((n != 0) && (! (n & (n - 1))
		)) {
		return n ; }
		while ((n != 0)) {
		n >>= 1 ;
		count ++ ;
		}
		return 1 << count ;
		}
			static int nextPowerOf2 (int n) {
			int count = 0 ;
			if (n != 0 && ! (n & (n - 1)))
			return n ;
			while (n != 0) {
			n >>= 1 ;
			count ++ ;
			}
			return (1 << count) ;
			}
				static int nextPowerOf2(int n) {
				int count = 0;
				if (n > 0 && (n & (n - 1)) == 0)
				return n;
				while(n != 0) {
				n >>= 1;
				count += 1;
				}
				return 1 << count;
				}

Test Code (Java)	**Execution Output**			

public static void main(String args[]) {
int n = 0;
System.out.println(nextPowerOf2(n));
}
	1	error: bad operand type int for unary operator '!'	error: bad operand type int for unary operator '!'	1

Input (Java)	**SPACoder (Python)**	**Transcoder-ST (Python)**	**CodeT5 (Python)**	**Groundtruth (Python)**

static int findTrailingZeros(int n) {
int count = 0;
for (int i = 5; n / i >= 1; i *= 5)
count += n / i;
return count;
}
	def findTrailingZeros (n) :
	count = 0
	i = 5
	while n // i >= 1 :
	count += n // i
	i *= 5
	return count
		def findTrailingZeros (n) :
		count = 0
		for i in range (5 , n // i + 1 , 5) :
		count += n // i
		return count
			def findTrailingZeros (n) :
			count = 0
			for i in range (5) :
			count += n // i
			return count
				def findTrailingZeros(n):
				count = 0
				while(n >= 5):
				n //= 5
				count += n
				return count

Test Code (Python)	**Execution Output**			
n = 100 print("Count of trailing 0s " + "in 100! is", findTrailingZeros(n))	Count of trailing 0s in 100! is 24	UnboundLocalError: local variable 'i' referenced before assignment	ZeroDivisionError: integer division or modulo by zero	Count of trailing 0s in 100! is 24

Figure 8.4: Qualitative translation results from SPACoder and the baseline methods given the same input. In all three examples, the baselines' results exhibit the "Shallow Translation" problem, where code snippets are directly copied or translated token by token from the source language, causing compilation and run-time errors in the target language. SPACoder's translation shows its strong ability in correctly aligning the syntax and APIs across different languages.

Performance on Low-resource Languages. In ECoST, C only has less than 150 parallel code pairs with each language, making it suitable for evaluating in more challenging low-resource language settings. As shown in Table 8.1, the compilation rate is the lowest when C is involved, as the generator is not able to generate high-quality data when the training data of C is significantly less. Table 8.4 shows the performance of the two implementations of SPACoder and their respective generators. For PLBART, SPACoder improves the CodeBLEU by up to 40% and improves the Computation Accuracy by up to 43%. This shows that the augmentation of parallel code generation works well in low-resource language settings, where the generator's performance is weak. For CodeT5, the improvement in Computation Accuracy is up to 23%.

Impact of Curriculum on Translation Performance. Table 8.5 shows the results of an ablation study designed to incrementally add each synthetic and annotated parallel code dataset to the alignment-ascending curriculum used for training the code translation model. Starting with a base model trained solely on the annotated dataset ECoST (function-level), we progressively add each dataset one by one into the curriculum. The results demonstrate that each added synthetic dataset enhances the model's performance on both metrics. Notably, the best performance is achieved when all synthetic and annotated datasets are included in the curriculum (SPACoder), underlining the cumulative contribution of each dataset in the curriculum.

Qualitative Analysis. Figure 8.4 shows examples of various model translations and their execution outputs given the same input code. The first column corresponds to the code used as input in the source language, and the last column corresponds to the ground truth translation in the target language. All examples are from the ECoST test set. We compare SPACoder-CodeT5 with two other baselines, TransCoder-ST and CodeT5. In the first two examples, we observe that both baselines demonstrate the "shallow translation" problem. In

the C++ – Python example, both TransCoder-ST and CodeT5 directly copy from the input code. While `min_element` is a valid built-in function defined in header `<algorithm>` in C++, it does not exist in Python, resulting in compilation errors for both baselines. TransCoder-ST also exhibits inability in translating multiple functions at once. In the Python – Java example, both TransCoder-ST and CodeT5 translate the keyword `"not"` in Python to `"!"` in Java. However, operator `"!"` cannot be used when the operand is an integer. By translating at token level, these baselines fail in taking the context into consideration, causing run-time errors. In both cases, SPACoder-CodeT5 is able to translate the function calls and statements from the source language to the target language correctly. In the Java – Python example, both baselines fail at translating a complex `for` loop, while SPACoder correctly translates this in a different way from the ground truth, showing a strong capability of understanding the input code and mapping it into a different language.

8.6 Conclusion

In this chapter, we introduce SPACoder, addressing the limitations of existing methods for code translation. By leveraging semi-supervised parallel code generation with enhanced cross-lingual alignment, SPACoder overcomes the challenges of generating high-quality and well-aligned parallel code data in a scalable manner. We demonstrate the effectiveness of SPACoder through extensive experiments conducted on multiple languages and models. The synthetic parallel code generated by SPACoder significantly improves the performance of code translation, outperforming state-of-the-art baselines by a significant margin. Notably, our method achieves remarkable gains in C translations even with a limited number of annotated training instances. Our work showcases the importance of generating parallel code data with good quality and alignment in order to enhance code translation capabilities.

Future work can extend to more tasks that benefit from large amount of parallel data.

Acknowledgments

This work was supported in part by a grant from the Virginia Commonwealth Cyber Initiative (CCI).

Chapter 9

Conclusions

9.1 Revisit of Hypotheses

9.1.1 Part I: Learning with Long Sequences

- **H1:** For document ranking with respect to queries, an algorithm that assigns different weight to each sentence based on the query and context leads to better query-document matching, as compared to treating all sentences with equal importance.

  Result: True. We designed a neural architecture, HAR [148], that successfully captures the hierarchical relevance between the query and each sentence of the document and the whole document. The effectiveness of this approach is proven by the model's high performance in ranking documents in response to queries compared to a number of baselines.

- **H2:** For questions where the answers span over non-consecutive sentences in a document, an algorithm that simultaneously selects all relevant spans yields better answers to queries than an algorithm that selects each sentence independently.

  Result: True. Through the development of the MultiCo [149] architecture, we demonstrated that it is possible to capture the inter-sentence relationship among answer spans of a query in a document and form better answers compared to independent sentence selection.

9.1.2 Part II: Learning from Domain Knowledge

- **H3:** Modeling latent properties with the guidance of domain knowledge yields improved results in biomedical entity linking, as compared to models that do not incorporate domain knowledge.

  Result: True. We verified this hypothesis with the development and evaluation of LATTE [150]. LATTE jointly performed fine-grained type learning and entity disambiguation. The performance improvement compared to state-of-the-art baselines indicates the benefit of modeling latent types guided by domain knowledge.

- **H4:** Utilizing similar code comments across different programming languages to obtain fine-grained parallel data enhances the development and validation of machine learning methods in this domain.

  Result: True. Our construction and utilization of the XLCoST [151] dataset, which leveraged similar code comments across different languages, indeed resulted in the enhancement of development and validation of models, thereby validating our hypothesis.

9.1.3 Part III: Learning under Limited Supervision

- **H5:** Performing multilingual pre-training on snippet-level parallel code data enhances program-level translation performance, particularly for languages with low resource availability.

  Result: True. The effectiveness of multilingual pre-training on snippet-level parallel data was proven through the development and evaluation of our program translation model MuST-PT [152], which showed enhanced performance, especially for low-resource languages.

- **H6:** Implementing curriculum learning with large amounts of noisy synthetic parallel code enhances the performance of neural code translation.

Result: True. By leveraging semi-supervised parallel code generation with enhanced cross-lingual alignment in SPACoder, the implementation of curriculum learning significantly improved the performance of code translation, as evidenced by the experimental results.

9.2 Conclusion

In this thesis, we have tackled the unique challenges posed by domain-specific language understanding, specifically within the realms of healthcare and programming languages. We have carefully addressed the unique issues concerning the comprehension of domain-specific sequences, the necessity for domain knowledge, and the scarcity of labelled data within these domains. Through our exploration, we have uncovered pivotal insights that are categorized within three parts: Learning with Long Sequences, Learning from Domain Knowledge, and Learning under Limited Supervision.

In Part I, we have presented an innovative deep neural network architecture that ranks documents for healthcare related queries, which has demonstrated substantial improvement over existing techniques. We have also proposed a novel question-answering system that utilizes multiple spans from a long document to provide comprehensive answers. This innovative approach to question-answering has not only elevated the performance of our model but also paved the way for future research in multi-span QA.

In Part II, we focused on integrating domain knowledge into our models. We introduced LATTE, a novel neural architecture for biomedical entity linking that leverages latent type modeling. Our results demonstrated that integrating such domain knowledge can lead to improvements in performance. Moreover, we presented a new dataset parallel across multiple programming languages, opening new avenues for research in cross-lingual tasks and code-

related research problems.

Finally, in Part III, we addressed the challenge of learning with limited supervision. We proposed a new program translation model utilizing multilingual snippet denoising auto-encoding and translation pre-training, leading to significant improvements, especially for low-resource languages. Additionally, we introduced SPACoder, a novel approach for code translation that leverages semi-supervised parallel code generation and enhanced cross-lingual alignment, demonstrating remarkable performance even with limited annotated training instances.

The work presented in this thesis has broad implications for the field of neural sequence modeling for domain-specific languages. By addressing both general and domain-specific challenges, we have shown that machine learning and natural language processing methods can be successfully applied in a range of different contexts. We have also shown that there is still a great deal of potential for further research in this area, especially when it comes to further exploring the methods and techniques we have developed.

However, our work is just the beginning. In each part of our study, we have opened new avenues for exploration and improvement. The impact of our work reaches beyond just the healthcare and programming language domains and can potentially extend to a multitude of other domain-specific languages. We hope our efforts will inspire and guide future research in this field, ultimately leading to greater understanding and utility of domain-specific languages in the realm of machine learning.

Bibliography

[1] Asma Ben Abacha and Dina Demner-Fushman. A question-entailment approach to question answering. *BMC bioinformatics*, 20(1):511, 2019.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[3] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[4] Mayank Agarwal, Kartik Talamadupula, Fernando Martinez, Stephanie Houde, Michael Muller, John Richards, Steven I Ross, and Justin D Weisz. Using document similarity methods to create parallel datasets for code translation. *arXiv preprint arXiv:2110.05423*, 2021.

[5] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 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, 2021.

[6] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. Avatar: A parallel corpus for Java-Python program translation. *arXiv preprint arXiv:2108.11590*, 2021.

[7] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Summarize and generate to back-translate: Unsupervised translation of programming languages. *arXiv preprint arXiv:2205.11116*, 2022.

[8] Mikel Artetxe, Gorka Labaka, Eneko Agirre, and Kyunghyun Cho. Unsupervised neural machine translation. *arXiv preprint arXiv:1710.11041*, 2017.

[9] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. An effective approach to unsupervised machine translation. *arXiv preprint arXiv:1902.01313*, 2019.

[10] Parminder Bhatia, Busra Celikkaya, Mohammed Khalilia, and Selvan Senthivel. Comprehend medical: a named entity recognition and relationship extraction web service. *arXiv preprint arXiv:1910.07419*, 2019.

[11] Olivier Bodenreider. The unified medical language system (UMLS): integrating biomedical terminology. *Nucleic acids research*, 32(suppl_1):D267–D270, 2004.

[12] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250. AcM, 2008.

[13] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2018.

[14] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[15] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading Wikipedia to answer open-domain questions. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1870–1879, 2017.

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[17] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL https://proceedings.neurips.cc/paper/2018/file/d759175de8ea5b1d9a2660e45554894f-Paper.pdf.

[18] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[19] Gonçalo M Correia, Vlad Niculae, and André FT Martins. Adaptively sparse transformers. 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 2174–2184, 2019.

[20] Zhuyun Dai, Chenyan Xiong, Jamie Callan, and Zhiyuan Liu. Convolutional neural networks for soft-matching n-grams in ad-hoc search. In *Proceedings of the Eleventh*

ACM International Conference on Web Search and Data Mining, pages 126–134. ACM, 2018.

[21] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

[22] Dina Demner-Fushman and Jimmy Lin. Answer extraction, semantic clustering, and extractive summarization for clinical question answering. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 841–848. Association for Computational Linguistics, 2006.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

[24] Matthew Dunn, Levent Sagun, Mike Higgins, V Ugur Guney, Volkan Cirik, and Kyunghyun Cho. SearchQA: A new Q&A dataset augmented with context from a search engine. *arXiv preprint arXiv:1704.05179*, 2017.

[25] Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. Understanding back-translation at scale. *arXiv preprint arXiv:1808.09381*, 2018.

[26] Angela Fan, Yacine Jernite, Ethan Perez, David Grangier, Jason Weston, and Michael Auli. ELI5: Long form question answering. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3558–3567, 2019.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

[28] Matthew Francis-Landau, Greg Durrett, and Dan Klein. Capturing semantic similarity for entity linking with convolutional neural networks. *arXiv preprint arXiv:1604.00734*, 2016.

[29] Robert M French. Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135, 1999.

[30] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, 4(Nov): 933–969, 2003.

[31] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[32] Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners. *arXiv preprint arXiv:2012.15723*, 2020.

[33] Siddhant Garg, Thuy Vu, and Alessandro Moschitti. Tanda: Transfer and adapt pre-trained transformer models for answer sentence selection. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7780–7788, 2020.

[34] Fredric C Gey. Inferring probability of relevance using the method of logistic regression.

In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 222–231. Springer-Verlag New York, Inc., 1994.

[35] Paul N Gorman, Joan Ash, and Leslie Wykoff. Can primary care physicians' questions be answered using the medical journal literature? *Bulletin of the Medical Library Association*, 82(2):140, 1994.

[36] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[37] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[38] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W Bruce Croft. A deep relevance matching model for ad-hoc retrieval. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 55–64. ACM, 2016.

[39] Nitish Gupta, Sameer Singh, and Dan Roth. Entity linking via joint encoding of types, descriptions, and context. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2681–2690, 2017.

[40] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.

[41] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[42] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[43] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[44] Raphael Hoffmann, Congle Zhang, Xiao Ling, Luke Zettlemoyer, and Daniel S Weld. Knowledge-based weak supervision for information extraction of overlapping relations. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 541–550. Association for Computational Linguistics, 2011.

[45] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[46] Phu Mon Htut, Samuel Bowman, and Kyunghyun Cho. Training a ranking function for open-domain question answering. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 120–127, 2018.

[47] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in neural information processing systems*, pages 2042–2050, 2014.

[48] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 2333–2338. ACM, 2013.

[49] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[50] Gautier Izacard and Edouard Grave. Distilling knowledge from reader to retriever for question answering. *arXiv preprint arXiv:2012.04584*, 2020.

[51] Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. What does BERT learn about the structure of language? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3651–3657, 2019.

[52] Mengqi Jin, Mohammad Taha Bahadori, Aaron Colak, Parminder Bhatia, Busra Celikkaya, Ram Bhakta, Selvan Senthivel, Mohammed Khalilia, Daniel Navarro, Borui Zhang, et al. Improving hospital mortality prediction with medical named entities and multimodal learning. *arXiv preprint arXiv:1811.12276*, 2018.

[53] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142. ACM, 2002.

[54] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

[55] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. SpanBERT: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

[56] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 173–184, 2014.

[57] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 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, 2020.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[59] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

[60] Tomáš Kočiskỳ, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. The NarrativeQA reading comprehension challenge. *Transactions of the Association for Computational Linguistics*, 6:317–328, 2018.

[61] Kusum Kusum, Abrar Ahmed, Bhuvana C, and V. Vivek. Unsupervised translation of programming language - a survey paper. In *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 384–388, 2022. doi: 10.1109/ICAC3N56670.2022.10074182.

[62] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

[63] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv e-prints*, pages arXiv–1901, 2019.

[64] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 260–270, 2016.

[65] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations*, 2018.

[66] Phong Le and Ivan Titov. Improving entity linking by modeling latent relations between mentions. *arXiv preprint arXiv:1804.10637*, 2018.

[67] Minsuk Lee, James Cimino, Hai Ran Zhu, Carl Sable, Vijay Shanker, John Ely, and Hong Yu. Beyond information retrieval: medical question answering. In *AMIA annual symposium proceedings*, volume 2006, page 469. American Medical Informatics Association, 2006.

[68] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, 2020.

[69] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. In *International Conference on Learning Representations (ICLR)*, 2017.

[70] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *International Conference on Learning Representations (ICLR)*, 2017.

[71] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. 2019.

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

[73] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[74] Gang Luo, Chunqiang Tang, Hao Yang, and Xing Wei. MedSearch: a specialized search engine for medical information retrieval. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 143–152. ACM, 2008.

[75] Yuanhua Lv and ChengXiang Zhai. When documents are very long, BM25 fails! In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 1103–1104. ACM, 2011.

[76] Wenlei Mao and Wesley W Chu. Free-text medical document retrieval via phrase-based vector space model. In *Proceedings of the AMIA Symposium*, page 489. American Medical Informatics Association, 2002.

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

[78] Alexa T McCray. The UML semantic network. In *Proceedings. Symposium on Computer Applications in Medical Care*, pages 503–507. American Medical Informatics Association, 1989.

[79] Yishu Miao, Lei Yu, and Phil Blunsom. Neural variational inference for text processing. In *International conference on machine learning*, pages 1727–1736, 2016.

[80] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[81] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[82] Bhaskar Mitra, Fernando Diaz, and Nick Craswell. Learning to match using local and distributed representations of text for web search. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1291–1299. International World Wide Web Conferences Steering Committee, 2017.

[83] Sunil Mohan and Donghui Li. MedMentions: A large biomedical corpus annotated with UMLS concepts. *arXiv preprint arXiv:1902.09476*, 2019.

[84] Shikhar Murty, Patrick Verga, Luke Vilnis, Irena Radovanovic, and Andrew McCallum. Hierarchical losses and new resources for fine-grained entity typing and linking. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 97–109, 2018.

[85] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 651–654, 2013.

[86] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Divide-and-conquer approach for multi-phase statistical migration for source code. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596. IEEE, 2015.

[87] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. MS MARCO: a human-generated machine reading comprehension dataset. 2016.

[88] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 24(4):694–707, 2016.

[89] Anusri Pampari, Preethi Raghavan, Jennifer Liang, and Jian Peng. emrQA: A large corpus for question answering on electronic medical records. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2357–2368, 2018.

[90] Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Shengxian Wan, and Xueqi Cheng. Text matching as image recognition. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2793–2799. AAAI Press, 2016.

[91] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 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, 2002.

[92] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary

DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[93] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[94] Ben Peters, Vlad Niculae, and André FT Martins. Sparse sequence-to-sequence models. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1504–1519, 2019.

[95] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of NAACL-HLT*, pages 2227–2237, 2018.

[96] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Project CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[97] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 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, 2017.

[98] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training.

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

[100] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21 (140):1–67, 2020.

[101] Jonathan Raphael Raiman and Olivier Michel Raiman. DeepType: multilingual entity linking by neural type system evolution. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[102] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 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, 2016.

[103] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822*, 2018.

[104] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142, 2003.

[105] Chandan K Reddy and Charu C Aggarwal. *Healthcare data analytics*. Chapman and Hall/CRC, 2015.

[106] Siva Reddy, Danqi Chen, and Christopher D Manning. CoQA: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

[107] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan,

Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[108] Stephen E Robertson and K Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information science*, 27(3):129–146, 1976.

[109] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at TREC-3. *NIST Special Publication Sp*, 109:109, 1995.

[110] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/ed23fbf18c2cd35f8c7f8de44f85c08d-Paper.pdf.

[111] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *NeurIPS*, 2020.

[112] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. *arXiv preprint arXiv:2102.07492*, 2021.

[113] Baptiste Roziere, Jie Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*, 2021.

[114] Sebastian Ruder and Barbara Plank. Strong baselines for neural semi-supervised learning under domain shift. *arXiv preprint arXiv:1804.09530*, 2018.

[115] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975. ISSN 0001-0782. doi: 10.1145/361219.361220. URL http://doi.acm.org/10.1145/361219.361220.

[116] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[117] Rudolf Schneider, Sebastian Arnold, Tom Oberhauser, Tobias Klatt, Thomas Steffek, and Alexander Löser. Smart-MD: Neural paragraph retrieval of medical topics. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 203–206. International World Wide Web Conferences Steering Committee, 2018.

[118] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[119] Vikash Sehwag, Saeed Mahloujifar, Tinashe Handina, Sihui Dai, Chong Xiang, Mung Chiang, and Prateek Mittal. Robust learning meets generative models: Can proxy distributions improve adversarial robustness? In *International Conference on Learning Representations*.

[120] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *International Conference on Learning Representations (ICLR)*, 2017.

[121] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. Learning semantic representations using convolutional neural networks for web search. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 373–374. ACM, 2014.

[122] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[123] Simon Suster and Walter Daelemans. CliCR: a dataset of clinical case reports for machine reading comprehension. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1551–1563, 2018.

[124] Marc Szafraniec, Baptiste Roziere, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *arXiv preprint arXiv:2207.03578*, 2022.

[125] Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. BEIR: A heterogeneous benchmark for zero-shot evaluation of information retrieval models. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

[126] Özlem Uzuner, Brett R South, Shuying Shen, and Scott L DuVall. 2010 i2b2/VA challenge on concepts, assertions, and relations in clinical text. *Journal of the American Medical Informatics Association*, 18(5):552–556, 2011.

[127] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[128] Ellen M. Voorhees and Dawn M. Tice. Building a question answering test collection. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research*

and Development in Information Retrieval, SIGIR '00, pages 200–207, New York, NY, USA, 2000. ACM. ISBN 1-58113-226-3. doi: 10.1145/345508.345577. URL http://doi.acm.org/10.1145/345508.345577.

[129] Shengxian Wan, Yanyan Lan, Jiafeng Guo, Jun Xu, Liang Pang, and Xueqi Cheng. A deep architecture for semantic matching with multiple positional sentence representations. In *AAAI*, volume 16, pages 2835–2841, 2016.

[130] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. Learning fine-grained image similarity with deep ranking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1386–1393, 2014.

[131] Wenhui Wang, Nan Yang, Furu Wei, Baobao Chang, and Ming Zhou. Gated self-matching networks for reading comprehension and question answering. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 189–198, 2017.

[132] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[133] Dirk Weissenborn, Georg Wiese, and Laura Seiffe. Making neural QA as simple as possible but not simpler. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017)*, pages 271–280, 2017.

[134] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. In *International Conference on Learning Representations (ICLR)*, 2017.

[135] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. End-to-end neural ad-hoc ranking with kernel pooling. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 55–64. ACM, 2017.

[136] Jun Xu and Hang Li. AdaRank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 391–398. ACM, 2007.

[137] Liu Yang, Qingyao Ai, Jiafeng Guo, and W Bruce Croft. aNMM: Ranking short answer texts with attention-based neural matching model. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 287–296. ACM, 2016.

[138] Yi Yang, Wen-tau Yih, and Christopher Meek. WikiQA: A challenge dataset for open-domain question answering. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2013–2018, 2015.

[139] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. In *Advances in neural information processing systems*, pages 5754–5764, 2019.

[140] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1480–1489, 2016.

[141] Wen-tau Yih, Ming-Wei Chang, Xiaodong He, and Jianfeng Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. In *Pro-*

ceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1321–1331, 2015.

[142] Pengcheng Yin and Graham Neubig. 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, 2017.

[143] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. QANet: Combining local convolution with global self-attention for reading comprehension. *International Conference on Learning Representations (ICLR)*, 2018.

[144] Hong Yu, Minsuk Lee, David Kaufman, John Ely, Jerome A Osheroff, George Hripcsak, and James Cimino. Development, implementation, and a cognitive evaluation of a definitional question answering system for physicians. *Journal of biomedical informatics*, 40(3):236–251, 2007.

[145] Lei Yu, Karl Moritz Hermann, Phil Blunsom, and Stephen Pulman. Deep learning for answer sentence selection. *arXiv preprint arXiv:1412.1632*, 2014.

[146] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[147] Richard Zens, Franz Josef Och, and Hermann Ney. Phrase-based statistical machine translation. In *Annual Conference on Artificial Intelligence*, pages 18–32. Springer, 2002.

[148] Ming Zhu, Aman Ahuja, Wei Wei, and Chandan K Reddy. A hierarchical attention retrieval model for healthcare question answering. In *The World Wide Web Conference*, pages 2472–2482, 2019.

[149] Ming Zhu, Aman Ahuja, Da-Cheng Juan, Wei Wei, and Chandan K Reddy. Question answering with long multiple-span answers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3840–3849, 2020.

[150] Ming Zhu, Busra Celikkaya, Parminder Bhatia, and Chandan K Reddy. LATTE: Latent type modeling for biomedical entity linking. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9757–9764, 2020.

[151] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. XLCoST: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.

[152] Ming Zhu, Karthik Suresh, and Chandan K Reddy. Multilingual code snippets training for program translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11783–11790, 2022.

[153] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*, 2021.